MAKING COMPUTERS KEEP SECRETS

by

Leo Joseph Rotenberg

S.B., Massachusetts Institute of Technology
(1965)

S.M., Massachusetts Institute of Technology
(1966)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1973

PART I

Signature of Author _____
Department of Electrical Engineering, June 25, 1973

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

MAKING COMPUTERS KEEP SECRETS

by

Leo Joseph Rotenberg

Submitted to the Department of Electrical Engineering on June 25, 1973 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

This dissertation presents a unified design of protection mechanisms for a computer utility that (1) prevent accidental unauthorized releases of information, (2) prevent tyranny by dividing and limiting the power of the administrators of the utility, (3) preserve the independence of independent users of the utility, (4) accommodate to organizations having disparate traditional superior-subordinate relations, and (5) support proprietary services that allow users to build on the work of others in a context that protects the interests of lessors and lessees of services. The design includes specifications of both hardware and software protection mechanisms, including walls defined by domains and capabilities, and a hardware device, the Privacy Restriction Processor, that records the copying and combining of information in the computer by propagating privacy restrictions among restriction sets associated with segments and processes. The propagated restrictions prevent accidental unauthorized releases of information. But when a secret can be encoded into the timing or occurrence of system actions to prevent output of secrets, the encoded secret can escape. However, such escaping secrets can be detected and the offending computation can be arrested by the operating system.

The dissertation includes an analysis of the social concepts, systems, and conventions to which a computer utility is necessarily connected. The emergence of a 1984-like negative-utopia is shown to be a possible consequence of the ongoing development of techniques for penetrating and taking over computer systems.

THESIS SUPERVISOR: Robert M. Fano
TITLE: Ford Professor of Electrical Engineering

Acknowledgements

I wish to thank Professor Robert M. Fano for encouraging and guiding this research. He had the patience and insight to argue with me as forcefully as was appropriate in every circumstance, but he never overreacted to my overheated rhetoric, and he never made me feel stupid for disagreeing with him. Thanks also to Professor Jerome H. Saltzer for his helpful criticisms and suggestions, and to Professor James D. Bruce for consenting, advising, and editorial assistance.

I owe special thanks to Professor Robert M. Fano, Robert Frankston, Professor Jerome H. Saltzer, and James Rumbaugh, whose potent criticisms proved to me that my proposed secret-keeping mechanisms were still imperfect. They prompted the most fevered moments of this research.

I wish to thank Marjorie Polster for assistance in editing the chapter "Society and Information." Thanks also to David R. King and Jeffrey A. Meldman for reviewing and commenting on the section "Privacy and U.S. Law."

Finally, thanks to Betty Rotenberg, Sam Rotenberg, and Sandy Rotenberg for assistance in proofreading the typescript of this report.

Table of Contents

Figures

Chapter 1

Overview

> "I think [the computer] is probably the most powerful single tool by quite a bit that man has ever invented. It gives us enormous capabilities to augment our human capacities, but therein, of course, lies its danger, too. We have to watch it with great care."
>
> -- Jerome B. Wiesner [US71]

The quality of life might someday be improved by computers. Early applications of computers to work that was already being done before the computer arrived on the scene have not much influenced the quality of life, except for persons working in the computer industry. But the computer makes possible new services and new interconnections of organizations and individuals which might have a profound impact on the quality of life in the future. For example, centralized emergency medical records would improve the quality of medical care received by accident victims. Computer-aided medical diagnosis would help medical doctors cope with the explosion of medical knowledge. As the computer is taught to provide complex and useful knowledge-based services, we might expect the quality of life to improve.

But the computer will also support systems that severely degrade the quality of life. For example, many modern weapons systems include computers. More to the point of this research, the computer could easily provide the basis of a centralized

dossier system, imprisoning citizens in institutional evaluations based on records held by unforgivingly long-memoried computers. The development of computer and communications technology makes the dehumanized negative-utopia of 1984 [Or49] ("Big Brother is Watching You") a clear option for the future.

The broad goal of this thesis is to contribute to the development of computer technology that will lead to an improvement in the quality of life. We recognize that computers are increasingly important components of social systems, and therefore we expect that the design of computers will have subtle, and possibly profound, effects on social systems. We hope that computerized central dossier systems will never emerge, but we do not expect the institutional trend towards placing personal information in computers to be completely reversed. (*) Therefore we feel it is necessary to build computers that can keep secrets, so that storage of personal information in computers will not reduce individual privacy. (When we say that a computer keeps secrets, we mean that the computer prevents unauthorized releases of information.) A strong technological optimism underlies this felt necessity: we hope that negative social effects can be prevented with sophisticated technological fixes. A simpler and sounder approach might be to outlaw dangerous applications of tech-

(*) This trend must be watched and might require regulation by government.

nology, but that approach has not been the focus of this thesis.

Institutions decide whether, when, and how to release information to individuals or other institutions in society, and thus individual privacy is most affected by the institutions that hold personal information. We expect that at least some institutions will act to protect individual privacy, and these institutions would be poorly served by computers that couldn't keep secrets. Thus this work is directed towards opening technological options for humane institutions.

We have succeeded in finding a new mechanism for preventing unauthorized releases of information from computers. This mechanism acts to associate authorizations with information itself, rather than information containers, and thereby prevents accidental unauthorized releases of information. It does not appear to be possible to prevent all unauthorized releases of information from a computer, because it is extremely difficult to prevent a cleverly written program from signalling information to a human who interacts with the computer. But our mechanism detects such signalling as it occurs, whereupon the operating system can arrest the offending computation. These two achievements of our mechanism provide a new capability for information protection.

Our original goal was to design a mechanism which would absolutely prevent all unauthorized releases of information.

This is evidently not possible, roughly because computer systems are not closed systems. People embed computers in social systems, and computer systems radiate. Cathode-ray tube terminals radiate photons, central processors radiate at radio frequencies, and printers inundate social systems with information on paper. When considering that one can snap a picture of a CRT terminal with a camera, or the ubiquitous use of office copying machines, it is clear that making computers keep secrets will never be more than a part of any information-security envelope.

The problem of providing protection for information and information systems is toughest in the context of a computer utility. A data bank of sensitive information maintained in a computer utility might become the target of organized attacks aimed to steal, modify, or destroy information. Since the services of computer utilities will be available to everyone who agrees to pay for them, the attackers can use the facilities of the utility to mount their attack. Our information protection mechanisms must be able to defeat attacks raised up inside the computer utility itself by malicious users.

Protection of information and information systems has not been a priority requirement in the development of commercial computer systems. Studies carried out by James Anderson and Daniel Edwards [Bra73, An72] have uncovered several design and implementation weaknesses in security provided by

commercial computer systems. They found that protection mechanisms which are "added on" to existing operating systems can be penetrated by seven different classes of attacks. For a computer system to be secure, it must be designed with security as a primary objective. This thesis contains such a design: we present a "paper computer" which can be secured against penetration attacks, and which can keep secrets.

Computers that keep secrets must store authorizations that specify how and to whom information is to be released. In addition to authorizations concerning release of information, computers will store authorizations that relate to all the available rights of control over computing objects. These rights of control are the handles used by people to control computers. As computers become more and more energetic actors in social arenas, they approach the status of supporting the entire nervous system of society. As this occurs, people with power over computers will have more power over society, and therefore the design principles by which power is licensed and limited in society must be applied to computers. The paramount design principle is the prevention of tyranny. This principle arises naturally in democratic societies, and its implication for a computer utility is the necessity of dividing and limiting the power of people and organizations over the utility. Organizations that use the computer utility will require independent rights of control over computing objects, and the administrators of the computer utility must not have

the power to abridge the independence of users. Furthermore, the power of the computer utility's administrators must be divided and limited to provide a system of checks and balances in the administration of the computer utility.

A computer's authorization system is an interface between the computer and established organizational authority. In every organization that uses a computer, the question of who controls the stored authorizations must be asked and the answer must be expressed in terms of the computer's authorization system. An inflexible computer authorization system would probably be burdensome to an organization whose structure and style were not congruent to the organizational model used by the computer's designers. A computer utility must be sufficiently flexible that user organizations can distribute rights of control over computing objects in ways that are natural to the organizations. In other words, a computer utility's authorization mechanism must adapt to varying styles and modes of organizational decision-making processes.

We have designed an authorization mechanism for our "paper computer" which satisfies the criteria enunciated in the previous paragraphs. Our mechanism preserves the independence of authority of the independent organizations that use the computer utility. To allow organizations to distribute organizational power in ways that do not excessively disturb their traditional superior-subordinate social relations, our mechanism includes a system of protocols, defined by the

organizations, whose purpose is to embody in the computer the social rules which were formally or informally followed in the pre-computer era. By using appropriately defined protocols, organizations can prevent adverse effects on the quality of organizational life due to the introduction of advanced computerized information systems.

In addition to secrecy systems and authorization systems, we have studied security problems associated with computer-based services. Such services will probably improve the quality of life, but several factors retard their growth. First, current computer technology is unable to protect the investment in programs and data that provide services. Under the capitalist system, the reward expected by developers of services is a monetary return for rental of programs or data or for services rendered. This reward can be assured by means of contracts between the lessor and lessees of services, but it can be more securely protected by computer technology that keeps the programs, data, and methods of providing services secret. Second, current computer technology is unable to protect from theft the data which the user of a service feeds to that service to be processed, unless the user controls the computer that provides the service. A service implemented in a computer utility might easily steal data, or sabotage its users. Users of services will require protection from these harms. The third factor is the awkwardness encountered in using current computer technology to build on the work of

others in the form of programs and data. Having to "reinvent the wheel" increases the cost of developing computer-based services.

In our "paper computer", users can develop and lease proprietary services in an environment that provides protection from most of the harms suggested above. Building on the work of others is encouraged and facilitated by the mechanisms presented.

In summary, we have designed a computer utility with hardware and software mechanisms for protecting the privacy of information, mechanisms for storing authorizations and interfacing to a bureaucracy or other organizational form, and mechanisms to support proprietary services in a computer utility in a context that protects the interests of lessors and lessees of services. The central technological contribution of the thesis is the privacy restriction mechanism described in chapters 6 and 7. The sociological contribution of the thesis is most concentrated in the social view in chapter 2 and the investigation of authority hierarchies in chapter 8.

Finally, we present a plan of the thesis. Chapter 2, "Society and Information," describes and analyses the social environment impacted by information technology. From our analysis of the complex social scene, we develop a set of requirements which must be satisfied by a computer utility. Chapter 2 is not crucial to the technological development that

follows.

Chapter 3, "Elementary Protection Mechanism," describes early work in the realm of computer protection mechanisms and generalizes from the examples presented to arrive at the concept of the domain. This chapter serves as a technological and philosophical introduction, but it is not essential; readers who know roughly what a domain is will find the development beginning in chapter 4 to be reasonably self-contained, except for occasional references to "goring the ox," which is explained in section 3.5.

Chapter 4, "Additional Protection Mechanisms," describes how processes call and return between domains, passing arguments and results between domains in a sectioned stack and in shared segments. This chapter also describes the operating system which supports the environment of domains, processes, and a naming hierarchy for computing objects.

Chapter 5, "Proprietary Services," describes nine protection problems associated with the use of services encapsulated in domains, such as services which steal information or sabotage their lessees, and lessees who conspire to steal secrets from lessors. Technological solutions are presented for some of these problems.

Chapter 6, "Privacy Restrictions," describes a mechanism which can protect information owners from would-be copiers of their information. The mechanism acts by propagating restrictions among restriction sets associated with segments and

processes and by striking down output to users and input to domains. But because the operation of the mechanism itself can be used as a signal, the mechanism is not leakproof: clever programs will successfully signal. A system of alarms is developed to deter such cleverness.

Chapter 7, "Privacy Restriction Processor," describes the hardware and software of a multiprocessing computer system which implements privacy restrictions.

Chapter 8, "Authority Hierarchies," describes the authorization mechanism of our design. Authority hierarchies are the computing objects which represent independent users of the computer utility. An authority hierarchy is a tree of "offices," each of which represents some collection of rights of control over computing objects. Domains, especially "home domains" of officials of organizations, are the "agents" of offices. Protocols associated with each authority hierarchy mediate some attempts by agents of offices to exercise rights of control over computing objects. Mechanisms for sharing authority, delegating authority, and sharing delegated authority are presented. Locksmithing, and the authorization system's most powerful lock and key, are introduced.

Chapter 9, "Conclusions," summarizes the nature of protection systems, surveys the sources of complexity of computer protection systems, and speculates on robotic watchers.

Chapter 2

Society and Information

2.1. Introduction

Institutions hold information. Governments, universities, manufacturers, hospitals, insurance companies, credit bureaus and the corner drug store all require, for their daily operation and continued existence, a large amount of information. Some of it is specialized, as when it is directly related to the function of the institution, and only similar institutions hold such information. Patients' histories held by hospitals and transcripts held by universities are examples of this. Also, institutions have some general information needs, i.e., information needs which are common to all (or almost all) institutions. For example, most institutions hold information concerning inventory, accounts receivable, accounts payable, payroll, and personnel.

Statistical information, generated from specialized or general information, is another of the general information needs of most institutions: managers of institutions use statistical analyses to help them understand what their institution is doing. For example, insurance companies do statistical and actuarial studies to set their rates, and the U.S. Government publishes the economic indicators that help fill the general statistical information needs of the business community.

Governments maintain numerous information services, including libraries and clearinghouses that publish technical information, market information, maps, etc. The U.S. Government conducts a census every ten years, making statistical information about the population generally available. Also, governments maintain many large data banks and dossier systems holding information about individuals. We will discuss data banks and dossiers at length in section 2.4.

Individuals hold information, although neither to the extent nor in the manner (with some exceptions) of a large institution. Almost all persons hold financial records and personal correspondence, and many people build libraries, both for professional purposes and for leisure.

Society is a vast, intricate, information-dependent system; and the purpose of this chapter is to explore some aspects of society relating to information and to that superfast scary information machine, the computer. We are interested in questions of the form, "What is the social relevance of a transfer of such-and-such a type of information?" The first important variable is the value of the information transferred; this is discussed in the next section. Then we turn to defining the concepts privacy, disclosure, and surveillance; and we review U.S. law relating to privacy.

In section 2.4, we discuss data banks and dossier systems, and their computerization; and we review some safeguards for the proposed National Data Center. Section 2.5 identifies the responsible actors in the process of institutional information transfer. Section 2.6 describes how surveillance spurs responsible action. When information transfer is computerized, surveillance can be programmed into the computer.

Finally, section 2.7 explores criminal activity, police surveillance, and computer penetration techniques; and section 2.8 summarizes requirements on computer systems which can be inferred from the considerations of this chapter.

2.2 The Value of Information

Some information has great value to individuals and to society. The value of information to decision-makers is a clear example: wise decisions are not likely without knowledge of options and knowledge of expected outcomes. Knowledge can be regarded as the possession of information. The remaining paragraphs in this section detail the value of certain specific types of information.

Information about the exercise of power and the formation of public policy is invaluable to the life of a democracy. It is well known that the wisdom of a policy

is improved by public debate of its merits. For instance, no public debate was held in 1964 and 1965 when the U.S. Government escalated the covert war against North Vietnam through the commitment of U.S. ground forces authorized to take offensive action [DoD71]; and our escalation is considered a mistake by most Americans today (in 1971).

Secret voting by representatives hampers the democratic process. But when constituents know how their representatives are voting, legislators will be more responsive to the public will. Thus we expect the elimination of secret voting in the U.S. House of Representatives to improve the quality of democracy here. [Hu70a,Hu70b]

Freedom of the press is essential to the operation of representative democracy, because the press is the transmission channel for information about public policy and the exercise of power. The response of organized groups in society to such information provides feedback to government.

Information about the physical universe, including the branches of knowledge we call science, engineering, medicine and nutrition; is of great value in improving the quality of life (when applied wisely), and has some power to form world views. Of course, science and engineering, together with ignorance and greed, created today's pollution crisis. But the solution must include still more science

and engineering; ecology and pollution preventment technology.

The operation of markets requires a flow of information to traders. The New York Stock Exchange ticker is the mechanism of one such flow. It is an exceptional source of market information in that it is inexpensive and available to all. Markets are generally dominated by clubs ("The Establishment") whose power comes from mutual support and access to inside information. For example, some information might have prompted nine major stockholders of the Penn Central Company to unload their holdings just before the subsidiary Penn Central Transportation Company filed for bankruptcy on June 21, 1970 [Bed71]. The wealth of dominant clubs comes from the exploitation of non-members, such as the buyers of the soon-to-be-devalued Penn Central stock in the example just given. These clubs require privacy to operate, and they have privacy.

Information about law and legal rights becomes important to individuals in times of conflict. Such information has always been available to the wealthy, while recent social movements have begun to make legal rights such as civil rights and welfare rights available to poor people.

Credit bureaus collect, hold, and disseminate credit and "other" information about individulas. This is an essential service for the people who want and get credit, and for the institutions that provide it.

Ordinary operating information, such as accounts re-

ceivable files, can be essential to the survival of an institution. It is not widely enough recognized that the loss or destruction of certain essential files can force a company into bankruptcy. [Bri71]

Sources of information considered reputable by the public can publish false reports, forged by intelligence agents, purposely designed to create political unrest or scandal. This is called "disinformation." For example, there was "the fabrication -- by a White House aide -- of a letter to the editor alleging that Sen. Edmund S. Muskie (D-Maine) condoned a racial slur on Americans of French-American descent as 'Canucks.'" [Ber72] The letter was published in the Manchester Union Leader less than two weeks before the 1972 New Hampshire primary.

These examples of the uses of information, gathered from the worlds of politics, academia, the economy, and the cloak-and-dagger community; give an idea of the scope of life touched upon by the exchange of information in our society. In all of our examples, the value of information is related to the values of expected outcomes of decisions made by persons or institutions which have access to the information. Our description of this relationship has been qualitative, but recent studies by Hirschhorn [Hi71] have applied economic tools to quantifying the value of information applied to the production decision.

2.3. Privacy, Disclosure, and Surveillance

> "It is necessary at the outset to sharpen the intuitive concept of privacy. As a first approximation, privacy seems to be related to secrecy, to limiting the knowledge of others about oneself. This notion must be refined. It is not true, for instance, that the less that is known about us the more privacy we have. Privacy is not simply an absence of information about us in the minds of others; rather it is the control we have over information about ourselves."
>
> -- Charles Fried [Fr68]

This idea of privacy, i.e., control over information about ourselves, is a large part of the concept. In addition, privacy means the right to be let alone [Wa90], the right to act anonymously, and the right to act without undue confusion, paranoia, or fear of the chilling effect of government. Confusion, paranoia, and fear can arise when government conducts surveillance of political activities, as in these United States.

> "I know that many, many students are afraid to participate in political activities of various kinds which might attract them, because of their concern about the consequences of having a record of such activities appear in a central file. ... I don't know to what extent these student fears have any justification, but I can tell you that they are real fears and that they frequently have caused students to back away from activities which attracted them. I might add here that I am not referring to confrontations or planned violence, but participation in seminars, political study groups, etc., that were seriously questioning governmental and social arrangements or policies.
>
> -- Jerome B. Wiesner [US71]

Privacy is part of the bundle of rights retained by autonomous humans after they have given up to society those other rights which society requires to provide for the survival of the community. If this bundle of rights is too slim, as when a society's survival seems to require extensive surveillance, the dignity of the citizen and his "inviolate personality" [Wa90] are abridged.

In giving information voluntarily, an individual is exercising his right to privacy by deciding what to give. In many circumstances, however, there is an element of coercion in the disclosure of information. For example, an individual accepts some degradation of his privacy in exchange for credit. That is, he must provide some information about his financial condition, and expect to have the facts he gives verified and his personal affairs investigated if his application for credit is to be approved.

Surveillance is the coercive negation of privacy. In other words, surveillance is the willful invasion of an individual's privacy for the purpose of gathering information about actions, associations, conversations, thoughts, motives, etc.

> "Surveillance is obviously a fundamental means of social control. Parents watch their children, teachers watch students, supervisors watch employees, religious leaders watch the acts of their congregants, policemen watch the streets and other public places, and government agencies watch the citizen's performance of various legal obligations and prohibitions. Records are kept by authorities to organize the task of indirect surveillance and to identify trends

> that may call for direct surveillance. Without such surveillance, society could not enforce its norms or protect its citizens, and an era of ever increasing speed of communication, mobility of persons, and coordination of conspiracies requires that the means of protecting society keep pace with the technology of crime."
>
> -- Alan Westin [Wes67]

An invasion of privacy sometimes produces embarrassing information for the invader, as in the case of marital infidelity or a criminal record. Such information can be used by one individual to gain power over another, or if made public, can lead to a crippling loss of face on the part of the exposed individual. It is for this reason that public disclosure of a very personal nature is sometimes considered tortious.

The emerging computerized personal data banks represent a threat to individual privacy. The availability of data banks of personal information to public and private officeholders in America reduces the level of privacy that can be enjoyed by the people about whom information is stored. The individual is in a very poor position to protect his privacy by himself, because so much information about him is held by institutions. The quality of privacy in our society, therefore, is controlled by the institutions that hold personal information.

Every society finds some balance between privacy and surveillance, which is a balance between the individual's

right to be secure in his person and the community's right to know some things about the individual. In analysing public policy that affects this balance, it is essential to realize the value of privacy.

> "...privacy is not just one possible means among others to insure some other value, but...it is necessarily related to ends and relations of the most fundamental sort: respect, love, friendship and trust. Privacy is not merely a good technique for furthering these fundamental relations; rather without privacy they are simply inconceivable. They require a context of privacy or the possibility of privacy for their existence...To respect, love, trust, feel affection for others and to regard ourselves as the objects of love, trust and affection is at the heart of our notion of ourselves as persons among persons, and privacy is the necessary atmosphere for these attitudes and actions, as oxygen is for combustion."
>
> -- Charles Fried [Fr68]

> "...one of the central elements of the history of liberty in Western societies since the days of the Greek city-state has been the struggle to install limits on the power of economic, political, and religious authorities to place individuals and private groups under surveillance against their will."
>
> -- Alan Westin [Wes67]

2.3.1. Privacy and U. S. Law

The rights of individuals to protection from invasions of privacy has been acknowledged many times in the history of U. S. law. However, these protections are scattered throughout the law, and each protection is limited and narrow. As a result, the law does not define any unified and comprehensive concept of privacy. In this section, we will describe some Constitutional protections of privacy, some statutory protections of

privacy, and the impact of developing technology on privacy and the response of the legal system to this impact.

The Bill of Rights provided the first protections of privacy, in the First, Fourth, and Fifth Amendments. The First Amendment guarantees the right of assembly (among other things). This right of peaceful assembly includes the right to form associations freely, for any purposes, no matter how unpopular (although this right does not protect purposeful complicity in advocating violent overthrow of the government). The protection of freedom of association also protects privacy of association, and guarantees freedom to participate in political life without unjustified governmental interference.

The Fourth Amendment guarantees protection of "persons, houses, papers, and effects, against unreasonable searches and seizures." The basic purpose of the amendment is to safeguard privacy and security of individuals against arbitrary invasions by government officials. The word "unreasonable" is crucial, and questions of reasonableness are examined by courts in deciding whether to grant search and seizure warrants. Warrants are issued on the basis of probable cause, supported by oath or affirmation; and they must describe specifically the place to be searched and the persons or things to be seized.

The Fifth Amendment guarantees that no person may be forced to bear witness against himself. The guarantee reflects the concern of society for the right of each individual to be let alone. The purpose of the guarantee is to enable the

citizen to create a zone of privacy which the government may not force him to surrender to his detriment. Also, the amendment forbids torture of witnesses and defendants, which was just going out of style in Europe at the time the Constitution was written.

The Ninth Amendment states that the enumeration of rights in the Constitution shall not be construed to deny or reduce other rights retained by the people. In Griswold v. Connecticut, the Supreme Court held that "specific guarantees in the Bill of Rights have penumbras, formed by emanations from those guarantees that help give them life and substance." Particularly mentioning the First, Third, Fourth, Fifth, and Ninth Amendments, the Court held unconstitutional a Connecticut law forbidding the use and dissemination of birth control information and devices. The Court spoke of the "intimate relation of husband and wife" as "a right of privacy older than the Bill of Rights." This landmark decision in effect created a new constitutional right to marital privacy, establishing in the law the idea that there are personal zones of individual experience which must be inviolate.

Various statutes enacted by Congress provide for some protection of individual privacy. We will describe a few of these protections, provided by laws governing the Census and fair credit reporting.

The Census Bureau collects and disseminates information on population, housing, commerce, agriculture, governments,

and many other things. The Census statutes prohibit the Secretary of Commerce and all officers and employees of the Department of Commerce (which includes the Census Bureau) from using census information for any purpose other than the statistical purposes for which that information was supplied. The officers and employees are prohibited from making any publication of census information whereby the data furnished by any particular establishment or individual can be identified, and they are prohibited from permitting anyone, other than sworn officers and employees of the Commerce Department, access to the individual reports. The Secretary of Commerce may furnish certified copies of certain returns to Governors, courts, or individuals, for proper purposes, but information so furnished may never be used to the detriment of the persons to whom the information relates. Also, any sworn officer or employee who publishes or communicates census information without proper authorization can be fined up to $1000.00 or imprisoned up to two years or both. These statutory prohibitions and requirements have proven to be effective protection of confidentiality and privacy for institutions and individuals that provide information to the Census Bureau.

Congress passed the Fair Credit Reporting Act to protect consumers against inaccurate, outdated, and out-of-context information in consumer reports. Such reports tell how the subject pays bills, and whether the subject has been sued,

arrested, or filed for bankruptcy, etc. Some consumer reports give neighbors' and friends' views of character, general reputation, and manner of living.

> "The Fair Credit Reporting Act requires credit bureaus to:
>
> (1) Follow reasonable procedures to assure maximum possible accuracy of information;
> (2) Disclose to the consumer, upon request, the "nature, substance" and sources of information in the file, and recipients of the report within the preceding six months;
> (3) Provide an opportunity for a subject to challenge the completeness or accuracy of any item in his file, to record the dispute if it is not resolved, and to correct any error;
> (4) Limit access to credit reports to those with a court order, the consumer's consent, or "a legitimate business need for the information;"
> (5) Delete adverse information which is 7-14 years old;
> (6) Notify the subject when detrimental public information is included in a report to be used for employment purposes, or to make sure that the information is current."
>
> -- Note, Yale Law Journal [YLJ71]

The note [YLJ71] goes on to cite weaknesses and problems in the Act in four major areas:

> "...the subject's right to (1) be notified of the existence of the report and inspect it; (2) correct or explain detrimental entries; (3) control access to the report; and (4) be protected by the Act while still pursuing common law remedies."

For example, with respect to the subject's right to inspect and correct private detrimental information, the Act is not very effective.

> "While under the Act the credit bureau must disclose the nature, substance and sources of information contained in the files on demand, it apparently need not let the subject read the report. Further, the sources of information about the subject's "character, general reputation, personal characteristics and mode of living" need not be revealed. The identity of the source is essential to any attempt to rebut the statements, which may be based on bad motive, lack of opportunity to observe, or similar grounds.
>
> . . .
>
> "There seems to be no reason why only sensitive public information in employment reports need be current or reported to the subject. But the most critical limitation is that the subject need never be notified when a report containing detrimental private information is being sent to a user, and there is no requirement to keep it current either. ... These problems drastically reduce the effectiveness of the statute."
>
> -- Note, Yale Law Journal [YLJ71]

The note, after concluding that the Fair Credit Reporting Act is "a first, if short, step toward solution, but may cause more problems than it solves," suggests further legislation required for the protection of consumers.

While the Constitution and statutes of Congress served reasonably well to protect individual privacy in the context of eighteenth century technology, new technological developments have seriously threatened the environment of privacy created by the framers of the Constitution. The telegraph (1850's) and telephone (1880's), and wiretapping, were developed. Microphones (1870's) and audio recording inventions (1890's) appeared, and microphone eavesdropping came into active use

before World War I. "Instantaneous photography" (1880's) allowed candid snapshots of persons and events without the subjects' prior consent. Radio technology and miniaturization extended the flexibility of microphone eavesdropping. The computer (1940's) can support dossier systems that carry out data surveillance of massive populations.

In the 1890's and early 1900's, mass-circulation newspapers published exposés of the private lives of public figures, based on the newly available candid snapshot technology. In a famous article, Samuel Warren and Louis Brandeis argued that such commercialized gossip was an unreasonable intrusion against the "right to be let alone."

> "...The intense intellectual and emotional life, and the heightening of sensations which came with the advance of civilization, made it clear to men that only a part of the pain, pleasure, and profit of life lay in physical things. Thoughts, emotions, and sensations demanded legal recognition, and the beautiful capacity for growth which characterizes the common law enabled the judges to afford the requisite protection, without the interposition of the legislature."
>
> . . .
>
> "...The intensity and complexity of life, attendant upon advancing civilization, have rendered necessary some retreat from the world, and man, under the refining influence of culture, has become more sensitive to publicity, so that solitude and privacy have become more essential to the individual; but modern enterprise and invention have, through invasions upon his privacy, subjected him to mental pain and distress, far greater than could be inflicted by mere bodily injury."
>
> . . .

> "The general object in view is to protect the privacy of private life, and to whatever degree and in whatever connection a man's life has ceased to be private, before the publication under consideration has been made, to that extent the protection is to be withdrawn. Since, then, the propriety of publishing the very same facts may depend wholly upon the person concerning whom they are published, no fixed formular can be used to prohibit obnoxious publications."
>
> -- Samuel Warren and Louis Brandeis [Wa90]

A majority of the states have adopted the principles proposed by Warren and Brandeis, but this common-law right to privacy has not been applied to surveillance by agencies of government.

The use of wiretaps by police agencies without the prior approval of a competent court was tolerated for more than a century by U.S. law. In the Olmstead case of 1928, the Supreme Court held (in a 5-to-4 decision) that a federal wiretap of a bootlegger's telephone was not a search and seizure covered by the Fourth Amendment. They chose not to demand that telephone taps satisfy the Fourth Amendment's rule of reasonableness, because no physical intrusion had occurred and because the conversation overheard "was not tangible" and was therefore exempt from Fourth Amendment protection. It is noteworthy that other eavesdropping technology can be used without physical intrusion, e.g., highly directional microphones and microphones that can listen through walls.

In the 1960's, the Supreme Court moved away from the property concepts enunciated in Olmstead, and Congress has

furthered this movement with the Crime Control Act of 1968 (Pub.L. 90-351). Title III of this Act makes it a crime to intercept wire or oral communication, or to use any device for intercepting oral communication in many cases, or to disclose an intercepted communication or to use information from an intercepted communication while knowing that it was intercepted. The statute prohibits the manufacture, distribution, possession, and advertising of wire or oral communication intercepting devices; provides for confiscation of the prohibited devices; prohibits the use of intercepted wire or oral communications as evidence; and establishes a civil cause of action against persons who illegally intercept oral or wire communication. Perhaps most important, the statute authorizes the interception of wire or oral communication by federal and state agents when approved by a judge of competent jurisdiction, and such legally intercepted communications can be used as evidence in court. The statute specifies procedures for authorized interceptions, and it provides for reports to Congress on the level of bugging. Also, the statute establishes a National Commission for the Review of Federal and State Laws Relating to Wiretapping and Electronic Surveillance which will be appointed in 1974 and make its final report in 1975.

The Crime Control Act's requirement for a warrant to authorize interception of wire or oral communication does not apply to interceptions undertaken in the interests of national security, because the Executive branch has traditionally used

wiretapping as part of espionage and counter-espionage activities directed against foreign powers, and because Congress did not want to expand, contract, or define Presidential authority to intercept wire or oral communications in matters affecting national security. However, the claimed "Mitchell doctrine" of an unlimited right to use wiretapping for domestic security surveillance without court approval was not upheld by the Supreme Court (in U.S. v. U.S. District Court for the Eastern District of Michigan).

While Congress has acted to redefine the protection provided by law against wiretapping and electronic surveillance, Congress has not acted to protect individual privacy against data surveillance, except in a limited way in the context of consumer credit reporting as described above.

> "Although the United States is the most advanced nation in the world in the field of computer science, we must look elsewhere to find comprehensive legislative proposals for solving the computer-privacy problem--in particular in Canada, Great Britain, and Germany. Under bills before the Ontario legislature and the British Parliament, (1) all data banks would be registered, (2) every person on whom a data bank maintains a file would receive a printout containing the file's original contents and have the right to demand printouts at later points in time, and (3) each printout would be accompanied by a statement of the file's use since the previous printout was supplied and a list of those who had received data from it. In addition, the individual could object to any item in the dossier and secure an expungement order from the Registrar of Data Banks, if he could show that the entry was incorrect or unfair. Civil liability and penalties also would be available if the bank supplied erroneous information or violated the act's provisions.

> "By and large these are remarkable proposals. I say this even though neither bill expressly deals with file security or snooping, prescribes the proper scope of data acquisition and input, contains limitations on dissemination, applies to all data banks that might contain information of a potentially damaging nature, imposes a duty of care on data bank operators except in the ex post facto sense of relying on individuals to seek correction, or requires the use of hardware or software controls to meet privacy-protecting standards. Many of these objectives would be achieved indirectly, however, because the possibility of liability under the proposed statute will encourage data banks to upgrade their practices."
>
> -- Arthur R. Miller [Mi71]

2.4. Data Banks

The U.S. Government maintains a score or so large data banks about individuals in the Internal Revenue Service, the Social Security Administration, the Veterans Administration, the Federal Bureau of Investigation, etc. These data banks exist to serve the operating needs of the agencies that maintain them. Of course, data banks are found at all levels of government. At the state level, motor vehicle registrations and operator licensing are accomplished with the help of data banks. At the county level (e.g., in California), welfare data banks are found. Cities and counties maintain data banks to administer property taxes, sewer taxes, and so forth.

Governments are not alone in maintaining data banks. The credit bureaus maintain credit and "other" information in their private dossiers on more than 100 million Americans [US68b], and life insurance companies maintain a shared data bank of medical information on policyholders.

In addition to these data banks of operating information for government and private agencies, governments at all levels maintain data banks of dossiers about political extremists and activists, and criminals. For example, the Federal Bureau of Investigation maintains the Security Index of 10,000 persons to be detained "in the event of a national emergency," and their National Crime Information Center holds information about wanted persons, stolen cars and other property. Also, the U.S. Government maintains data banks of dossiers about persons investigated for security clearances. Such investigations are conducted by the Civil Service Commission, the FBI, and the Armed Services Intelligence agencies.

Computers can hold data banks efficiently. The ongoing development of low-cost random access mass storage devices and data communication services allows us to envision a not-too-distant future when all these data banks will be on-line to computers linked in networks in such a way that anyone anywhere with a terminal to the network could access any data bank and find out anything recorded about anyone.(*) This is a nightmare.

(*)The FBI's National Crime Information Center (NCIC) is an example of what this technology can do. If a policeman sees a suspicious car, he can report the license plate number to his dispatcher, who will key it in at a console attached to a computer at their state police headquarters, which will relay the number to NCIC in Washington, D.C. If the car is stolen, NCIC gets a "hit" in its files and reports back to the state computer, which informs the dispatcher, who informs the officers who requested the information.

Because the technology is available, many people are afraid that the computerization of data banks will lead to computer networks that give every police officer, government official, and bank vice-president access to all the computerized information about everyone; which would be the end of personal privacy as we have known it. This fear has a potent force, which was felt when a National Data Center [US68a] was proposed as a step toward a unified Federal statistical data base. The danger that such a data base can be used as a dossier system prompted the hearings conducted by Senator Edward V. Long and Congressman Cornelius E. Gallagher which have served to delay the proposal while safeguards to protect personal privacy are incorporated.

People would be more comfortable about data banks if their rights to know of their files' existence, and to see their own files and to challenge errors, were clearly established. Such rights now exist with respect to credit bureaus and investigative reports [Han171a, YLJ71], but not with respect to files held by agencies of government [Han171b]. The right to see one's file will be difficult to obtain in some cases because of conflicts of privacy. A conflict of privacy exists when disclosure of the file contents to the file subject would violate the privacy of some person who contributed to the file.

Congress, and perhaps all the state legislatures, must act to establish the civil rights of seeing and challenging one's file in most data banks. In addition, a person should be notified as to when his file has been accessed, by whom, to what

extent, for what reason, and toward what end the information has been obtained. Furthermore, procedures must be devised to resolve problems with files in the context of conflicts of privacy.

2.4.1. Safeguards for a National Data Center

A statistical data bank is a much milder establishment of surveillance than the computer network suggested above. And certainly our expectation of rational government is that it should use the best possible statistical information in formulating its policies. Therefore we expect a National Data Center [US68a], with some safeguards, might be implemented.

Any system intended to protect personal privacy will naturally replace identifying data, in each record of the National Data Center's data bank, with a code number. Such usual identifying data as names, addresses, and Social Security numbers will be related in a one-to-one fashion to code numbers by a separate file. Agencies which contribute data to the bank do not need access to the (code number, personal identity) file, since their contributions have identifying information which can be translated into code numbers after the data are received by the National Data Center. The users of the data bank, who might be statisticians, have no need for the (code number, personal identity) file, except for such applications as detailed studies of "interesting" subsets of the population. In such studies the researcher will have devised a questionnaire, and selected a group of code numbers representing

individuals whose responses to the questionnaire are to be solicited. Such studies should have to be approved by a board of review whose charter includes the goal of protecting personal privacy to a reasonable extent, balancing that against the value of the information being sought to the well-being of the community. If the study is approved, the questionnaires can then be sent to the population selected, and the results made available to the researchers, with the personal identities remaining unknown (to the researchers).

The agency that grants access to the (code number, personal identity) file holds the power to allow use of the data base as a surveillance mechanism; that is, to invade privacy. The control of such power is a problem that will have to be solved by Congress.

> "...common-law notions of privacy are aimed too acutely at protection of undue publicity and emotional distress to meet the problems of privacy in a centralized information system. Nor are suits at law a particularly effective means of affording relief to those whose privacy is invaded by the system...
>
> Legislation specifically relevant to the organization of the proposed information center would appear to be a more appropriate legal solution. With so many forms of interrelated access possible, careful legislation would be needed to explicate precisely who is entitled to have access to what. Criminal sanctions for misbehavior would provide a much stronger deterrence than would the vague fear of a possible lawsuit."
>
> -- Jeffrey A. Meldman [Me69]

Users having access to a data bank through a statistical information retrieval program can use a well-known

technique [Ho70] to violate the privacy of individuals. The technique involves requesting of the information retrieval program the number of individuals satisfying given criteria. The criteria are chosen from known information which describes the individual under investigation. The investigator can keep adding criteria to the list presented to the information retrieval program until the program reports that there is only one person satisfying the criteria; the investigator then knows that he has "pinned down" his man. Through subsequent use of the retrieval program, the investigator can determine everything in the data bank concerning that individual by adding the additional property he is interested in to the list of criteria and querying the program. If the program reports that one individual satisfies the criteria, the investigator knows that his man satisfies the additional property; if the program reports that no individuals satisfy the criteria, the investigator knows that his man does not satisfy the additional property.

The statistical snooping technique just described can be prevented by modifying retrieval programs so that they will not disclose the number of individuals satisfying given criteria when that number is below a certain threshold. In addition, users who make suspicious requests can be reported to some higher authority by the retrieval program. These prevention techniques reduce, but do not eliminate, the possibility of using statistical data systems to violate

individual privacy. It is not yet clear whether it will be possible to implement a National Data Center with reasonable safeguards to protect the privacy of individuals and groups.

2.5. Transfers of Information

Information flows in our society are staggering; our environment is information-rich, and we are attention-poor. The media provide us with all manner of news, opinion, speculation, some educational material, and heavy doses of entertainment, escape, and advertising; in short, a glut of inputs. Similarly, the processes of learning and teaching engender a cornucopia of information.

The flows of information that are of primary concern in this thesis are those flows that begin and end in the files of institutions. For example, applications for credit and reports of credit ratings, applications for admission to schools and transcript reports, applications for employment and letters of reference and recommendation, and applications for insurance. In addition, every financial transaction carries with it a packet of explanatory information, which sometimes involves persons.

Because the flow of information can have good as well as harmful effects, it is natural to ask, for any particular packet in the information flow, "Who is responsible for that?" Actually this question isn't precise because the process of information transfer has many components to be responsible for.

Figure 2-1 shows a model of institutional information release that allows us to isolate some components of information transfer. Shown is one office of an institution, called office O, which releases data into a data channel which could be the mails, a courier, or a network of computers. The person in charge of office O is Q. The triangle surrounding him connotes his authority over O. Also shown is the chain of command over Q, up to the president of the institution, P. The legal environment of the institution is shown as a force acting from above.

When the office O releases a packet of information, the office becomes responsible for releasing the information in the first place, for designating the recipient of the information, and for choosing a data channel that offers a level of security commensurate with the value of the information. Since Q is in charge of O, he is responsible for all this. To a lesser extent (usually), so are the officials in the chain of command above him; and the institution is legally responsible for releasing the information if there is an appropriate law.

Since Q probably doesn't do all the work of office O himself, we must examine the operation of O more closely. One way to view O is as a collection of workers. Another view is as a collection of procedures; that is, O's work is defined by the procedures used by the workers who work there. Since the workers ought to know the procedures, Q must be

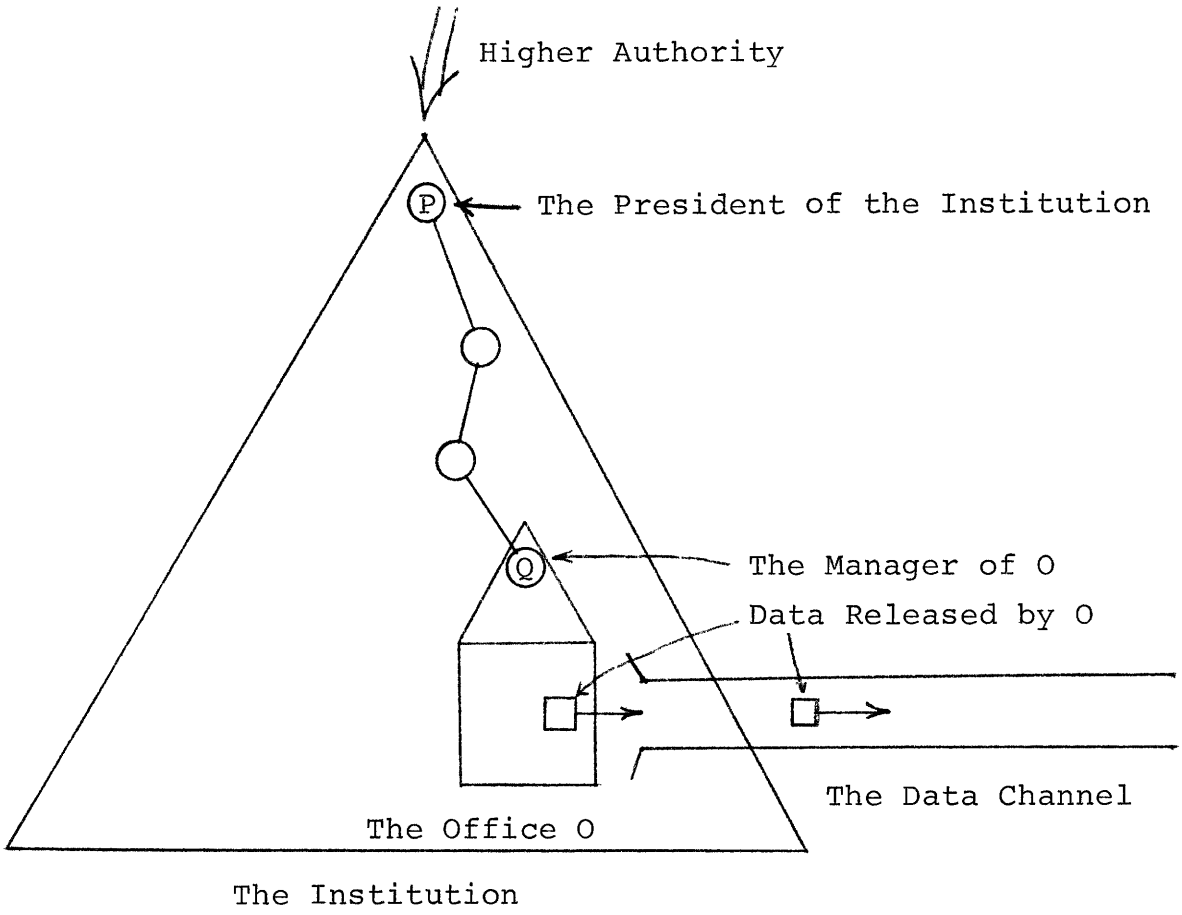


Figure 2-1. Model of institutional information release.

responsible for seeing that they do. Also, Q should be responsible for seeing that the procedures are reasonable. For example, if O were a university registrar's office, and the data were transcripts, a reasonable prerequisite for the release of a transcript would be a signed request from the subject of the transcript.

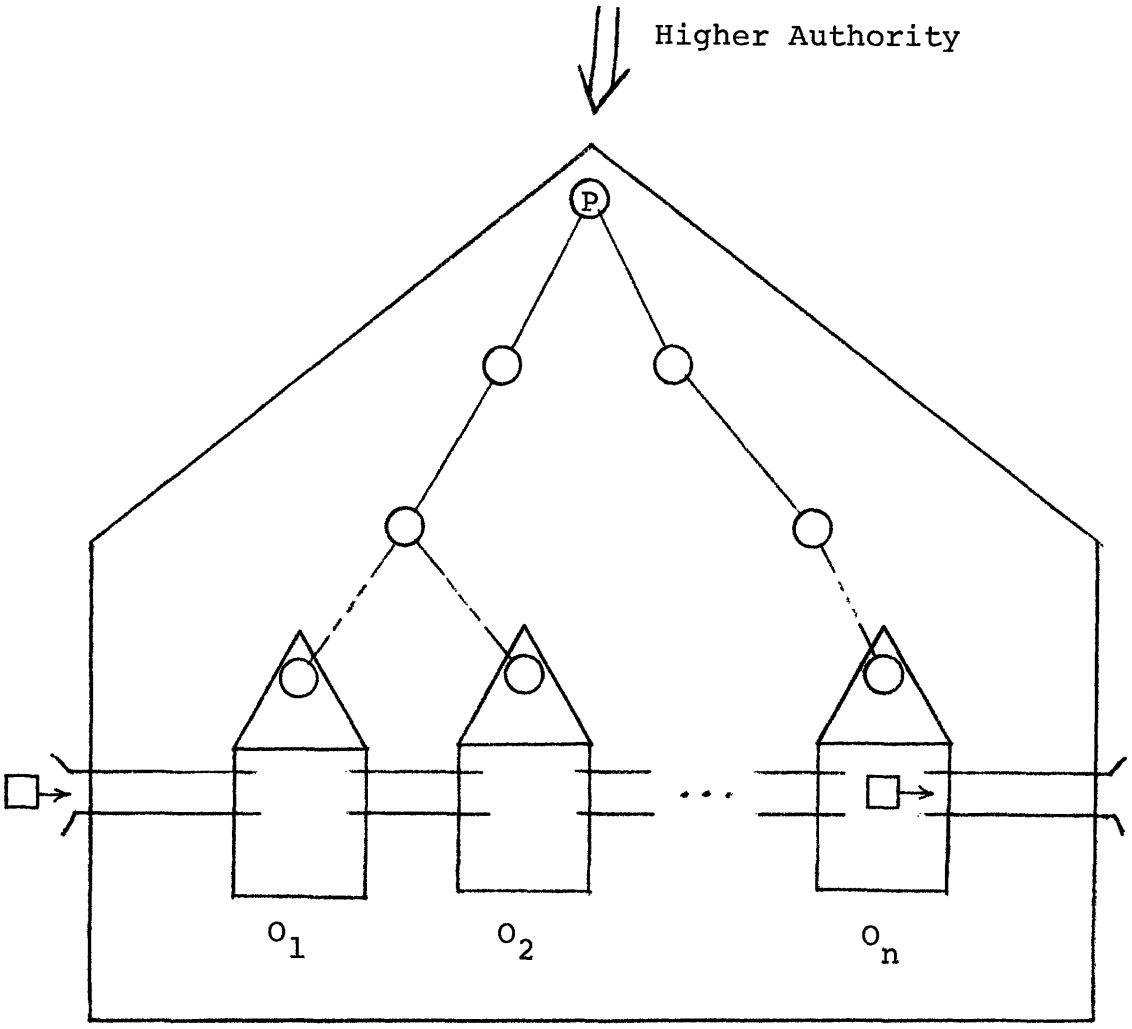
It sometimes happens that information is released through the improper action of a worker. For example, policemen in New York City sold printouts from the FBI's NCIC to private investigators [Com71]. If the worker isn't caught, he can't be held responsible for what he did. Of course, if the improper release goes undetected, no one will worry about holding anybody responsible for it. But if it is detected, the office becomes responsible for having allowed it. That responsibility might then be pinned on some worker whose negligence or lack of vigilance made the improper release possible.

Much more important than the procedures of office O are the policies from which those procedures are derived. For example, is it the policy of the institution to release information about individuals without their consent? The information release policies of institutions are crucially

important features of the social environment, and such policies must be reasonable(*) and responsible.

Figure 2-2 shows a model of the data channel from figure 2-1, expanded to take into account the institution that implements it. The path that a packet of data follows through the data channel is shown as a sequence of "offices" O_1, $O_2,\ldots,O_n$. In this context the word "office" is not always meant literally. Rather, the O_i can represent stations (which includes mailboxes and the storage components of store-and-forward message switching systems) and vehicles (such as mail

(*) "Reasonable" is the well-known lawyer's word that is intended to encapsule the judgement of a group of "reasonable men." As times change, so does the definition of "reasonable." For example, Ralph Nader would not have been considered a reasonable man in 1960, whereas he is today (in 1971). The men who wrote and passed Connecticut's law against the sale and distribution of birth control information and devices were reasonable men in their time. But in our time the Supreme Court struck down their law as unconstitutional. To this author the Connecticut legislators appear self-righteous and definitely not reasonable.



The Data Channel

Figure 2-2. Model of the data channel.

trains and electronic digital data transmission systems(*)) as well as offices where data packets are sorted and scheduled for delivery to other offices or to the addressee.

The overall responsibility of the data channel is to deliver the data packet to its addressee within a reasonable time, and to hold the data packet securely during the time it is in the data channel, which means to protect the packet from malicious or accidental damage or loss, and also to prevent the data in the packet from being released to anyone except the addressee. The responsibility of the data channel to hold information securely falls on all the offices O_1, $O_2,\ldots,O_n$; while the responsibility to deliver the information

(*) When the data transmission vehicles of the data channel are electronic transmission lines, the data channel is susceptible to wiretapping. This problem has received considerable attention in the context of military communications systems. In [Ba64] a solution is proposed which involves point-to-point and end-to-end encryption of the information. That is, the message is encrypted at the sending station and decrypted at the addressee's station, which is end-to-end encryption; and in addition the (encrypted) message is encrypted and decrypted every time it passes through a transmission line between stations. This latter encryption of the already once-encrypted message is point-to-point encryption. The end-to-end encrypted message contains the address of the message in its original (clear, unencrypted) form so that the intermediate stations can forward it; but when point-to-point encryption is used, that address is encrypted when the message is sent between stations. Thus point-to-point encryption prevents wiretappers from determining the addressees to whom traffic is directed, and also can raise the work factor of the entire system. (The work factor is the measure of the resources the wiretapper must expend to cryptanalyse and decrypt the message. A sometimes useful rule for choosing encryptions is to choose one whose work factor is so high that the cost of breaking the crypt is greater than the value of the information obtained thereby.)

with reasonable dispatch falls on those offices that do sorting, and the vehicles that move the packet. Of course, these responsibilities fall on a given office O_i, with respect to a given packet of information, only after the given packet has entered that office. In other words, the responsibilities of the data channel with respect to the packet are bound to the trail the packet takes through the channel.

Figure 2-3 shows a model of an institution absorbing information from a data channel. The office O, where the information is delivered, becomes responsible for accepting the information, for estimating the probability that the identification of the source of the message is a forgery, and for the further distribution of the information inside the institution. For example, figure 2-3 shows a packet of information from the files of office O being transferred to office R. The office O is responsible for releasing the packet to R.

Any office that uses information received by office O is responsible for its own usage. Thus, when R uses the information it got from O, R is responsible for its usage. Similarly, if R releases information that it holds, either inside or outside the institution, then it is responsible for that.

An important general responsibility of all the offices in figures 2-1, 2-2, and 2-3, is to hold information with precautions taken to insure its security. As before,

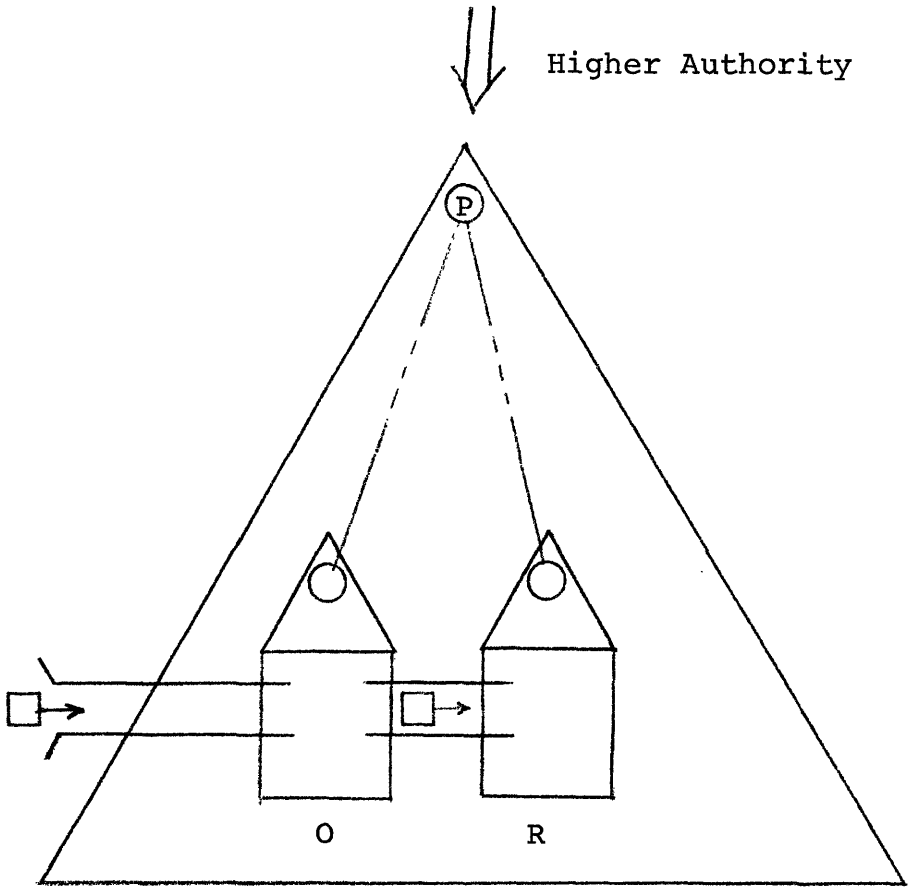


Figure 2-3. Model of institutional information absorbtion.

"security" means the prevention of accidental or malicious damage or loss, as well as the prevention of unauthorized release.

2.6. Surveillance and Responsibility

Officials of institutions and managers of offices are faced with the problem of encouraging their workers to be responsible. To make workers responsible, institutions often hire inspectors to carry on inspection and surveillance of work and workers. Our government has organized inspection and auditing of certain institutions to protect the public interest by keeping these institutions responsible. For example, the Securities and Exchange Commission (S.E.C.) requires disclosures of information by corporations before they may sell stock to the public, the Federal Aviation Agency inspects aircraft and issues airworthiness certifications to declare which aircraft are permitted to carry passengers, and the Post Office has a corps of Postal Inspectors who try to prevent mail theft.

Inspection, auditing, and surveillance must be carried out by independent, impartial agencies. In the case of aircraft certification, this means that the inspector who examines and certifies the aircraft must have no interest in (i.e., not stand to profit by) seeing the aircraft carry paying passengers. In the case of the disclosures required by the S.E.C., this means that the auditors who certify the disclosure must not stand to profit by sales of the audited

corporation's stock. In both these cases, the aircraft inspector and the auditor are supposed to protect the public interest. By their actions they restrict, discipline, and make responsible the actions of others.

Some work must be inspected, especially when the work (or lack of it) can hurt people or cause other undesirable effects. Sometimes inspection of work is not adequate protection, and surveillance must be used. Whenever surveillance is a condition of employment, such surveillance should stop at boundaries which are known to the employee and respected by his employer.

2.6.1. Surveillance of Information Transfers

Surveillance can be used to improve the security with which an institution holds information. For example, institutions which work with classified information use guards and closed-circuit TV to construct a security envelope. Of course, no such precautions are 100 percent effective -- witness the work of Dr. Daniel Ellsberg. [DoD71]

Surveillance can be used to insure that releases of information by offices are authorized. If, when an office releases information, a record is made of which worker released what information, then that worker can be held responsible. Since the worker will know that such work is being watched, he will endeavor to "cover himself" by determining that the release is properly authorized.

Similarly, surveillance can be used to hold offices and the workers in them responsible for information they receive.

A typical method is to require the execution of a receipt to be returned to the office or person who released the information.

2.6.2. Programmed Surveillance

When the work of offices to be watched is carried out by a computer, the surveillance can be programmed into the computer. Figure 2-4 shows a clerk using a computer to manipulate a data base. Also shown is a surveillance file, maintained by the computer, which records all the clerk's actions and all other actions that affect the data base.

If the surveillance log is to include all actions that affect the data base, it must be impossible to access the data base except through authorized programs which collect surveillance information. This is a crucial requirement: it must be possible to bind a data base to some caretaker programs and insure that no other programs can access the data base. Furthermore, the caretaker program which writes the surveillance log must be correct, and protected from modification. We are confident that such protection can be provided by an authorization system that permits only certain programmers and administrators to modify the caretaker programs, and surveillance over those authorized programmers and administrators. This surveillance over programmers and administrators can also be provided by a program. Figure 2-5 shows a programmer modifying the program of figure 2-4. He

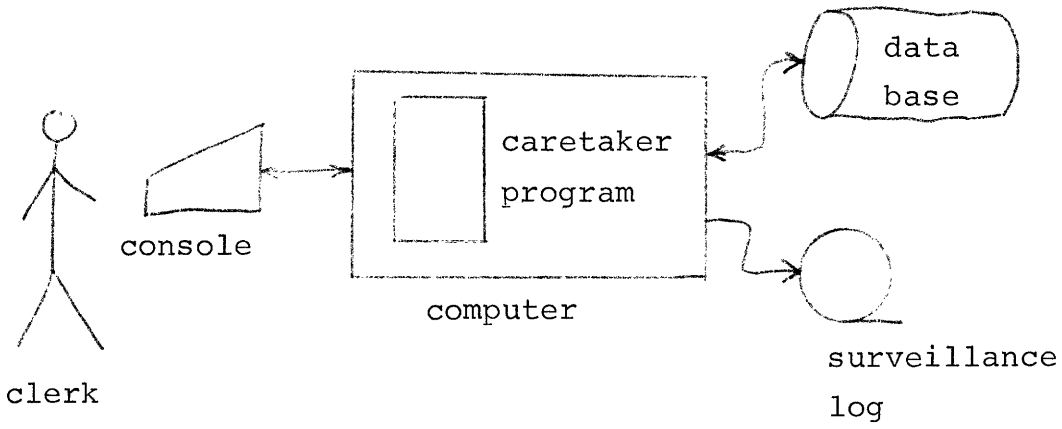


Figure 2-4. Programmed surveillance.

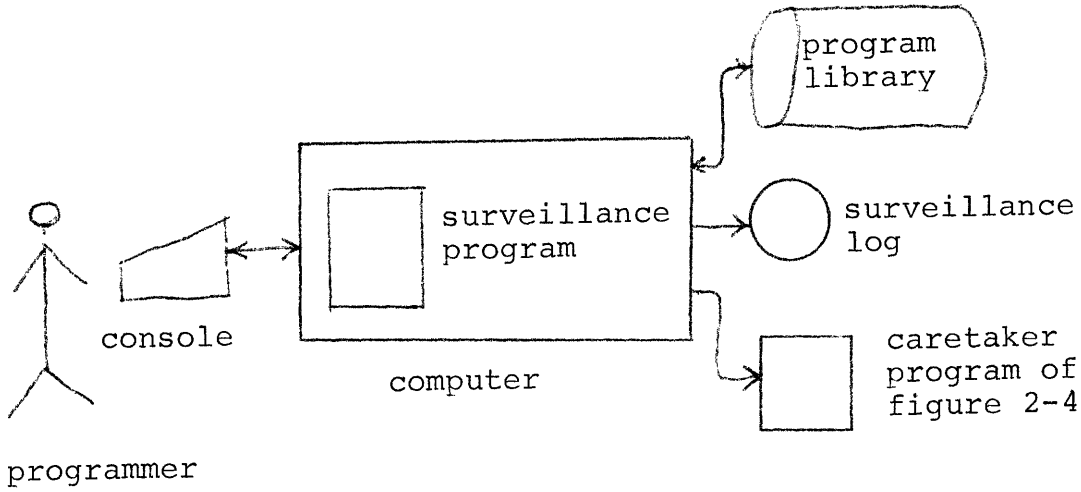


Figure 2-5. Surveillance over a programmer.

would modify the source program, compile it, test it, and finally install it. His installation triggers the writing of a record in the surveillance log of figure 2-5, so he is held responsible for a change. This surveillance log can be designed to record exactly what the change was, by including a copy of the program before the change was made, and another copy with the change incorporated, in the log.

The reader will no doubt want to know how changes to the surveillance program of figure 2-5 are watched. Of course, this cascading of surveillance could go on forever; but we postulate that the surveillance program of figure 2-5 is relatively fixed and unchanging, and that its correctness is guaranteed by human certifiers. Whenever a change must be made to it, the program is re-certified.

2.7. The Computer as a Social Arena

The computer is a powerful tool for processing information, and information has a powerful force in society. One way for society to protect itself from this potentially dangerous tool is to hold those persons who use it, especially programmers, responsible for their actions. But just as important is the problem of understanding what goes on inside the computer, and determining who should be responsible for the events that occur there. For example, such an event would be the passage of information, inside the computer, into the possession of a blackmailer.

In general, it is important to be able to say who is responsible for events, in the computer, that have potentially harmful effects. There are three major categories of events which are likely to have some harmful effects. These are: (1) unauthorized release of information, which might violate the privacy of some individual or group, (2) unauthorized modification of information, which might leave inaccurate information in the computer, and (3) reduced availability of the computer system. The reader should note that the concept of harm expressed here is as perceived by the authority which controls the information in the computer, and this will coincide with the reader's concept of harm only to the extent that the reader shares the value system of that authority. But apart from such differences in values, the three classes of harmful events above serve to outline a precise definition of computer security, which consists of secrecy (no unauthorized release of information), integrity (no unauthorized modification of information), and availability (no system failures which reduce the level of service).

In the following chapters, we develop a design for a secure computer system, and within the context of that design it is possible to say what programs are responsible for. Appendix 5 contains our enumeration of the responsibilities of programs.

2.7.1. Criminal Activity

It is likely that some criminal activity will go on within multi-access computers. A criminal could use a computer for his everyday data processing, or to spy in some way on another user. As a counterforce to criminal activity, multi-access computers might include spying mechanisms to be monopolized by government; e.g., to be used by police agencies with the approval of a competent court. When police have obtained a search warrant to seize information held in a computer, the warrant will say exactly what is being searched for, since "general warrants" are unconstitutional. The spying mechanisms used by police can be programmed to refer to the warrant and thereby prevent the police from conducting "fishing expeditions" -- i.e., sifting through files apart from the ones which they told the court they were looking for when they asked for the warrant. Also, the spying mechanisms can conduct surveillance over the spying, to make the officers who direct the spying responsible to the higher authority to whom the surveillance records will be released.

2.7.2. Computer Penetration Technology

Some considerable money and effort is being spent on the problem of building secure computer systems, and a part of that money and effort is going into the study of ways to break into, penetrate, and take over computer systems. For example, the RISOS project (Research In Secured Operating Systems) at Lawrence Livermore Laboratory is studying ways

and means of penetrating A.E.C. and military computer systems. After they develop the penetration technology, they will write guidelines for designers of future, more secure systems.

It is likely that two standards for secure computers will come into existence: a commercial standard and a military standard. A computer meeting the military standard would be more secure than computers meeting the commercial standard, because whenever a penetration technique is found which can be used to take over a computer, a protection mechanism to defeat the penetration technique will be added to the military standard. The military officers responsible for secure computing would have to take this step because penetration of a computer-based military command and control system would allow sabotage. When a penetration technique is very expensive, e.g., costing more than the value of the information or services provided by the average commercial computer, the protection mechanism to defeat the penetration technique is not likely to be included in the commercial standard. Therefore computers that are secure by commercial standards will probably be penetrable by techniques developed by the military.

The crucial question is: who will control this dangerous penetration technology?

It is clear that the agency which can penetrate commercial computers will have the power to invade individual privacy, carry out blackmail, etc. As things are developing now (in 1972), this power will be held by the military, the

C.I.A., and generally, the Executive branch of government. The old problem of controlling illegal use of Executive power will become more serious as new computer penetration techniques are developed. Twelve years to 1984, and the awesome, ugly power to create a 1984-like negative-utopia [Or49] is coalescing.

2.8. Requirements on Computers

From the considerations of this chapter, and related technological considerations, we can infer a number of properties of computers which will be socially beneficial.

First, computers should keep secrets. That is, computers should release information into society only when the release is authorized. Computers must contain authorization mechanisms which inform the computer whether and how to release information; and modifications to stored information must be similarly controlled.

Second, no individual should have absolute power, or even a large amount of power, over any computer which serves substantially important social functions. Power over computer utilities will probably be divided in patterns that reflect the interests in the social arena affected by computers, but the principle of preventing tyranny should prevail.

Third, computers should be auditable. If a computer is supposed to keep secrets, only an audit of its hardware and software can build trust among the computer's users that it

is keeping secrets. Computers with important and sensitive social functions, such as electronic libraries which should not keep lists of who reads what, ought to be open to audits by any citizen. That is, every citizen should have the right to obtain the data necessary to carry out an audit and evaluate the propriety of such a computer's actions. If such an audit might violate someone's privacy, the conflict between the right to audit and the right to privacy can be resolved by using a professional auditor who would be bound by a professional code of auditing ethics and procedures. The professional auditor would report on the propriety of the audited computer's actions, while avoiding unreasonable infringements on privacy.

Fourth, it should be easy to bind a data base in the computer to a caretaker program in such a way that references to the data base can be made only through calls to the caretaker program. A data base and caretaker program are said to be encapsulated when bound together in this way. Encapsulation protects the data base from the actions of all programs other than the caretaker program. The caretaker program can control what use is made of the information in the data base (e.g., release only statistical summaries), and it can collect surveillance concerning releases of information and modifications to stored information. Other important surveillance information can be collected only by the operating system, or with its secret assistance [Ro71].

In the remainder of this thesis, we present a design for a secure computer system which meets the above four requirements.

Chapter 3

Elementary Protection Mechanisms

3.1. Introduction

The purpose of this chapter is to review the means by which the goals of protecting the secrecy and integrity of information were achieved in multi-access computer systems that were conceived as prototype computer utilities. As before, "secrecy" is achieved when the computer prevents unauthorized releases of information, and "integrity" is achieved when the computer prevents unauthorized modification of stored information.

We will proceed by exploring means by which computations in multi-access computer systems are protected from accidental and/or malicious interferences. We will describe two different methods of protecting the memory of processes in multi-access computer systems. From these examples (many others are available and could be used equally effectively [De66, Bu61, Fa68]) we will extract the concept of the domain. Then, to explain why the domain is the most useful concept upon which to build computer protection mechanisms, we will abstractly examine the concept of protection, and offer a metaphorical model of protection in general, whereupon the domain will be seen as a special case.

In the following examples, we will consider two multi-access computer systems, and regard them as collections of

processes. Each user of the system has his own process, which will execute programs as commanded by the user. We agree with Thomas [Th71] that a process is best defined to be the activity of an active part, called a processor, which interprets and changes a passive part, called a process state. The process state is a collection of information that includes working registers, a program counter, a program, data, etc; while the action of a processor is defined by a state transition rule which specifies how the processor is to modify the process state. Since multi-access computer systems are time-shared, the processors are multiplexed among the processes. When a process does not have a processor, it is considered to be not running, and the operating system maintains its process state until such time as a processor is again available to run the process.

Our symbol for the processor is ♂, and our symbol for a process state is ♀. Our symbol for a running process is ⚥. We use these symbols because they emphasize the combining of Yin and Yang in a process.

3.2. CTSS

The Compatible Time-Sharing System (CTSS) [Cr65] was implemented using a specially modified IBM 7094 computer. The process state of a CTSS process was the tuple ♀ = (pc, registers, base, bound, M). The first component of the tuple, pc, is an integer which serves as the program counter of the

process. The second component, registers, represents the central processor registers of the processor that is the active part of this process. In CTSS, implemented on an IBM 7094, the registers are the AC, the MQ, the index registers, sense indicators, instruction register, etc. The third and fourth components, base and bound, are integers which control the memory protection tests in the state transition rule. These components represent registers which were added on to the 7094 to make memory protection possible. The fifth component, M, represents the user memory bank of CTSS. We will proceed by assuming that M = {(word#, bitstring)}, a set of ordered pairs. Each of these ordered pairs represents one word of the memory; word# is an integer which is the address of the represented word, and bitstring is a string of bits which represents the contents of the word. To insure that word addresses are well-defined, we require that the set {(word#, bitstring)} not contain two or more ordered pairs with the same first component word#. (Another way of expressing this restriction is to say that the set {(word#, bitstring)} is a function in the set-theoretic sense.)

The state transition rule of the CTSS processor is shown in figure 3-1. Notice that all references to the memory M by the process must be to words whose addresses are $\leq$ bound, and all addresses which pass this test are added to base and the resultant sum is used as the address in M. So the process'

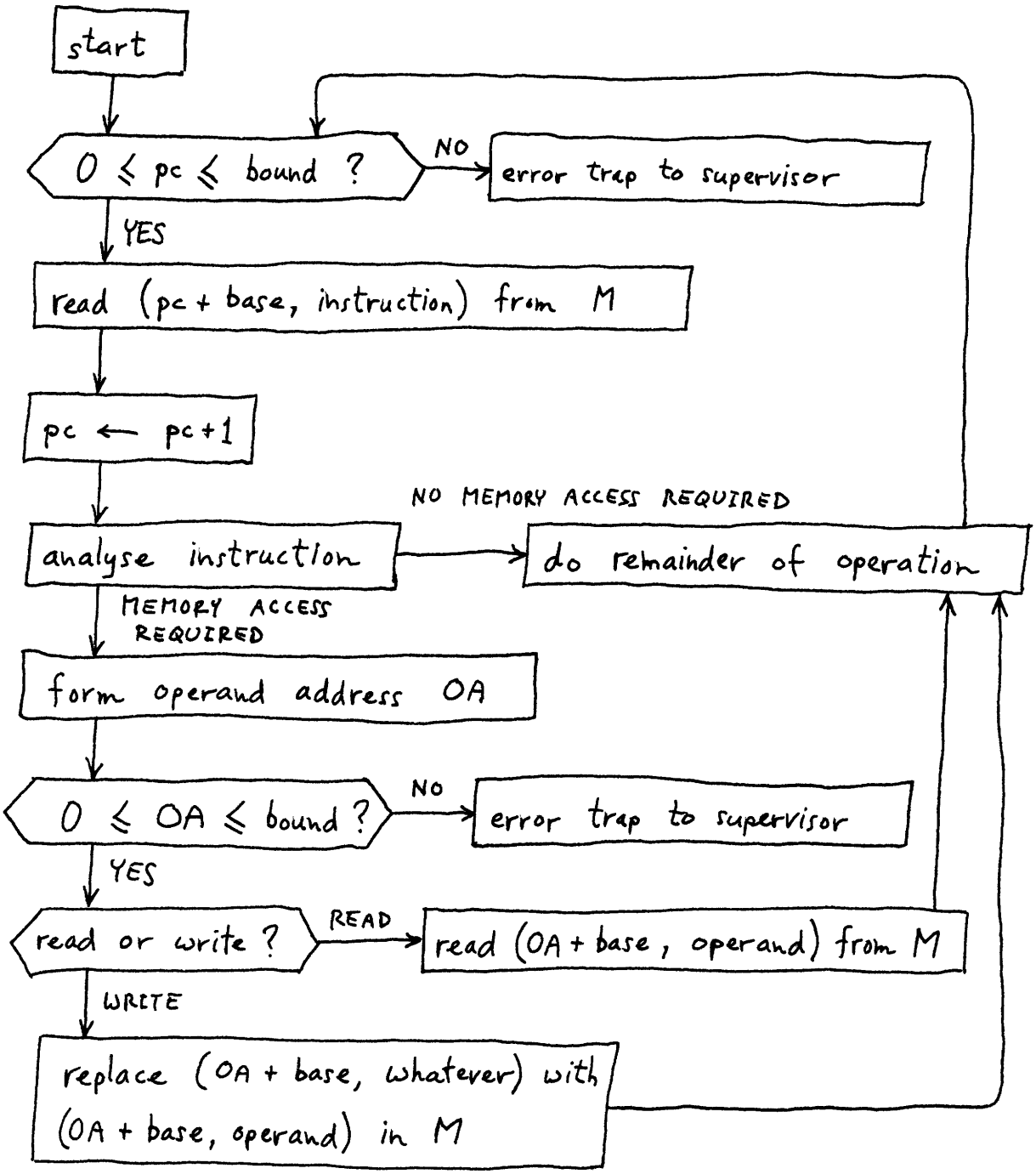


Figure 3-1. State transition rule of CTSS processor.

memory references are confined to a contiguous portion of M, as shown in figure 3-2. The supervisor of CTSS arranges to have the process' "core image" in that contiguous portion of M whenever the processor is assigned to the process, and the supervisor places the correct values of base and bound in the process state to prevent the process from reading or writing information anywhere except its "core image". When processes are not being run, their "core images" are swapped out to the drum and their pc and registers are tucked away in the supervisor's memory bank.

It is useful to consider a different view of the CTSS process. In the model just presented, all the processes share the same memory bank M, restricted by the base and bound registers. This is how things actually were, from the system designer's point of view. But from each process' point of view (or, from the user's point of view), each process had its own isolated contiguous memory space. (People who were there or heard the stories know that this wasn't completely true in the beginning!)

We now present the CTSS process seen as having its own isolated contiguous memory space. The process state is the tuple ♀=(pc, registers, {(word#, bitstring)}). The first two components, pc and registers, are just as they were previously. The third component is the memory space, which is a function in the set-theoretic sense, and that function has

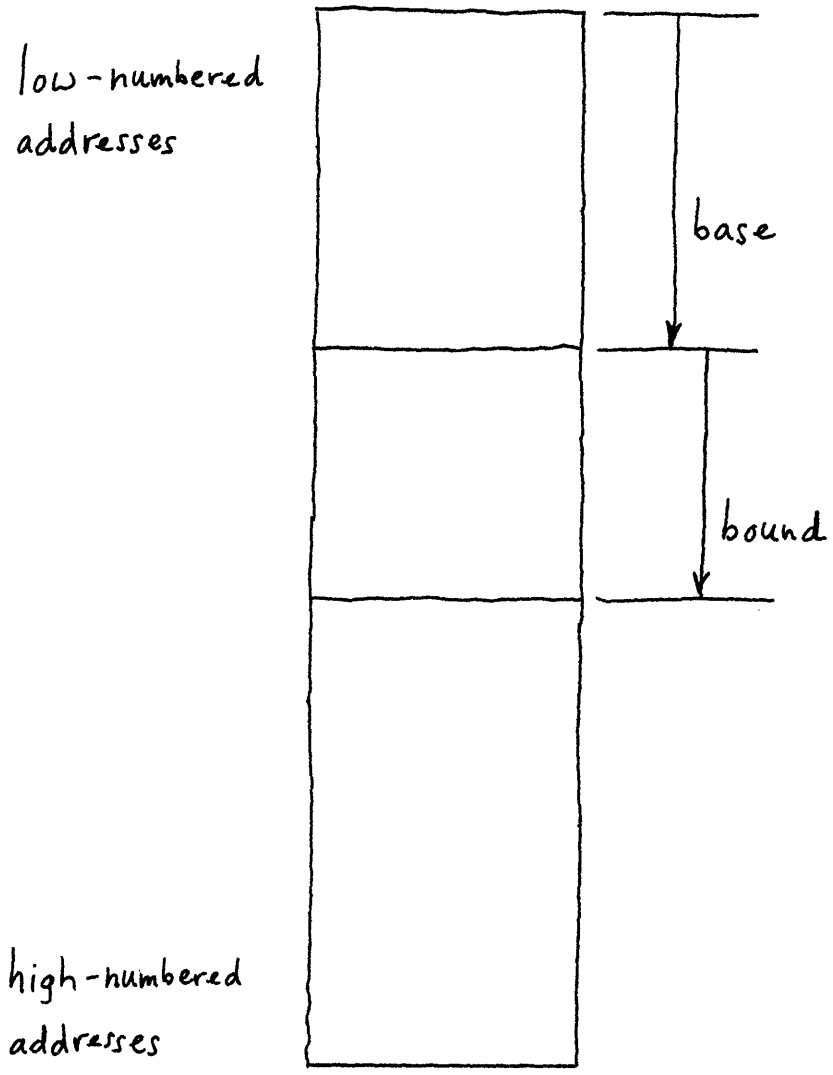


Figure 3-2. The user memory bank, M, of CTSS.

as its domain an interval [0, bound] of the integers. The state transition rule is given in figure 3-3. In the figure, M = {(word#, bitstring)} denotes the third component of the process state. Since M, in the state transition rule, always means the memory of the process being executed; each process is confined to referencing only its own memory.

This model of CTSS is more elegant that the first one because it is easier to see that processes are isolated; no arguments about swapping and base/bound registers are required. The critical reader will argue that this elegance is purchased at the price of hiding the actual protection mechanism; and while this is correct, nevertheless this elegant point of view is necessary to support the generalization which follows later in this chapter.

CTSS provided a good measure of protection of secrecy and integrity by keeping the memory spaces of processes isolated. Certainly, no process in CTSS was permitted to read or write in any other process' memory space. CTSS provided long-term storage for information in files, and a permission mechanism was available which allowed information owners to say which processes were allowed to read and write their files. But when a process read a file, it obtained in its memory space a copy of the information in the file, rather than having direct access to a single copy shared directly with other processes. This situation was improved in the Multics system with the

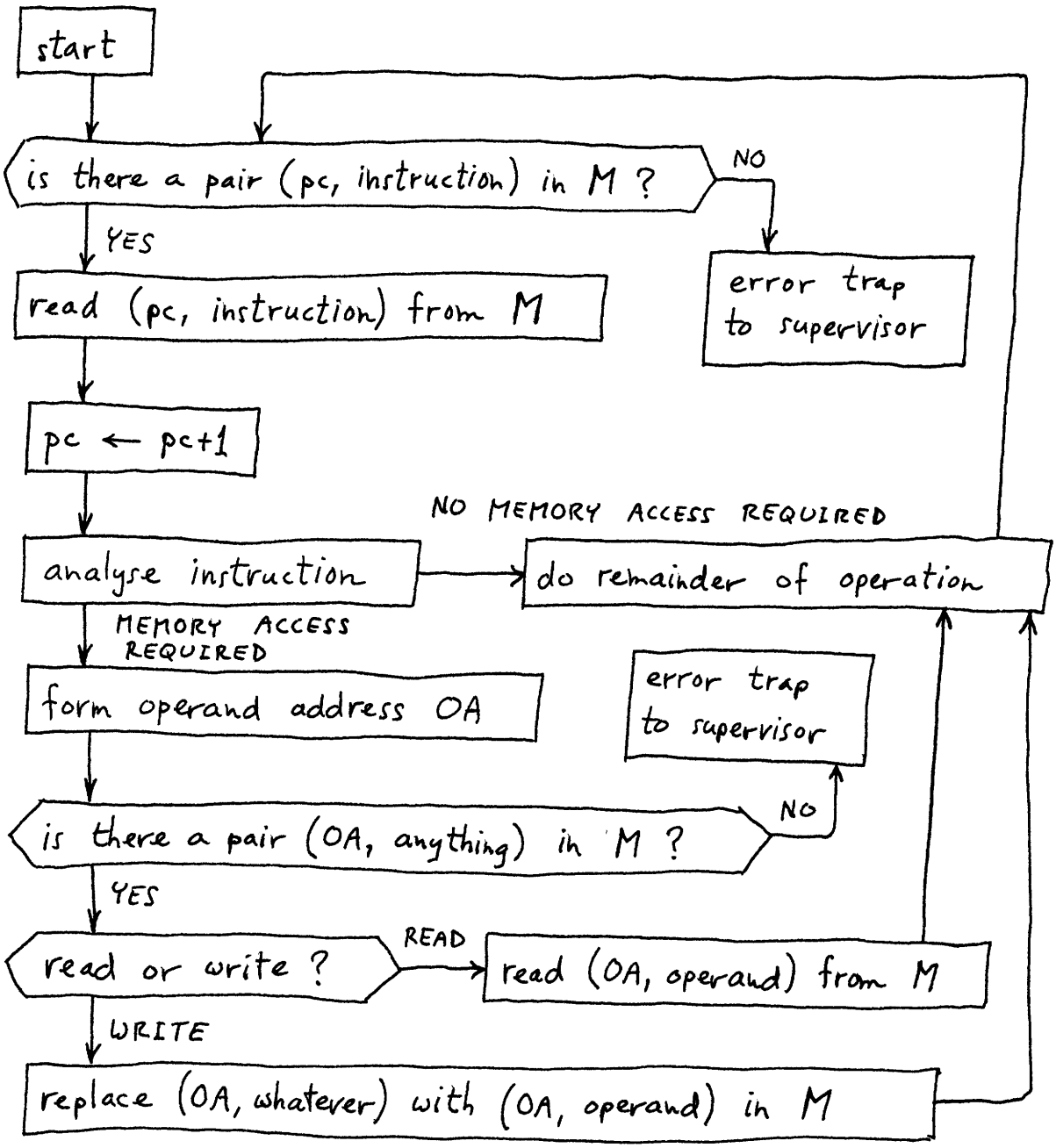


Figure 3-3. State transition rule of CTSS processor.

introduction of the segmented address space.

3.3. Segmented Address Space

The Multiplexed Information and Computing Service (Multics) [Cor72] was implemented using a Honeywell 645 computer, which was specially designed by General Electric in 1965. The Multics process possesses an address space composed of many independent segments. A segment is a contiguous linear array of words of memory. Segments hold programs and data. Processes refer to the segments they reference using two-dimensional addresses of the form (seg#, word#). The first component of this address selects a segment from the array of segments which is the address space of the process, and the second component selects a word from that segment.

We can model the process state of a Multics process with the tuple ♀ = (pc, registers, {(seg#, (mode, length, page_table_addr))}). The first component of the state, pc, is an address having the form (seg#, word#); as before pc is the program counter of the process. The registers are arithmetic, base, and index registers. (Note that the base registers also hold addresses of the form (seg#,word#).) The third component represents the descriptor segment of the Multics address space. This third component must be a function in the set-theoretic sense, so it defines a mapping from the set of segment numbers to the set of triplets {(mode, length, page_table_addr)}.

When the processor sees that the process is making a reference to a particular segment, it applies this mapping to obtain a triplet (mode, length, page_table_addr) for the segment. (Actually, the mapping is simply a reference to the array in the descriptor segment, using the seg# as the index.) From the components mode and length, the processor determines if the reference is permissible; and if it is, the processor uses the page_table_addr, an absolute address, to find the segment's page table. From this point to the referencing of the referenced word, standard virtual memory techniques are applied: a page table word is selected from the page table, yielding the absolute address of the page, and the desired word is selected from the page. The critical reader will have noticed we haven't mentioned the segment fault and page fault events which permit multiplexing the system memory; he should remember that memory multiplexing is not the subject we are addressing.

Figure 3-4 shows the state transition rule of the Multics processor. The rule applies three different tests to the mode component of the triplet (mode, length, addr). These tests are notated e(mode), r(mode), and w(mode), and they test for execute permission (e(mode) = 1), read permission (r(mode) = 1), and write permission (w(mode) = 1) respectively. Thus the mode component can be represented by a 3-bit string. Furthermore, the state transition rule refers to segments with

$\varphi = (pc, registers, AS)$ where
$AS = \{(seg\#, (mode, length, page\_table\_addr))\}$

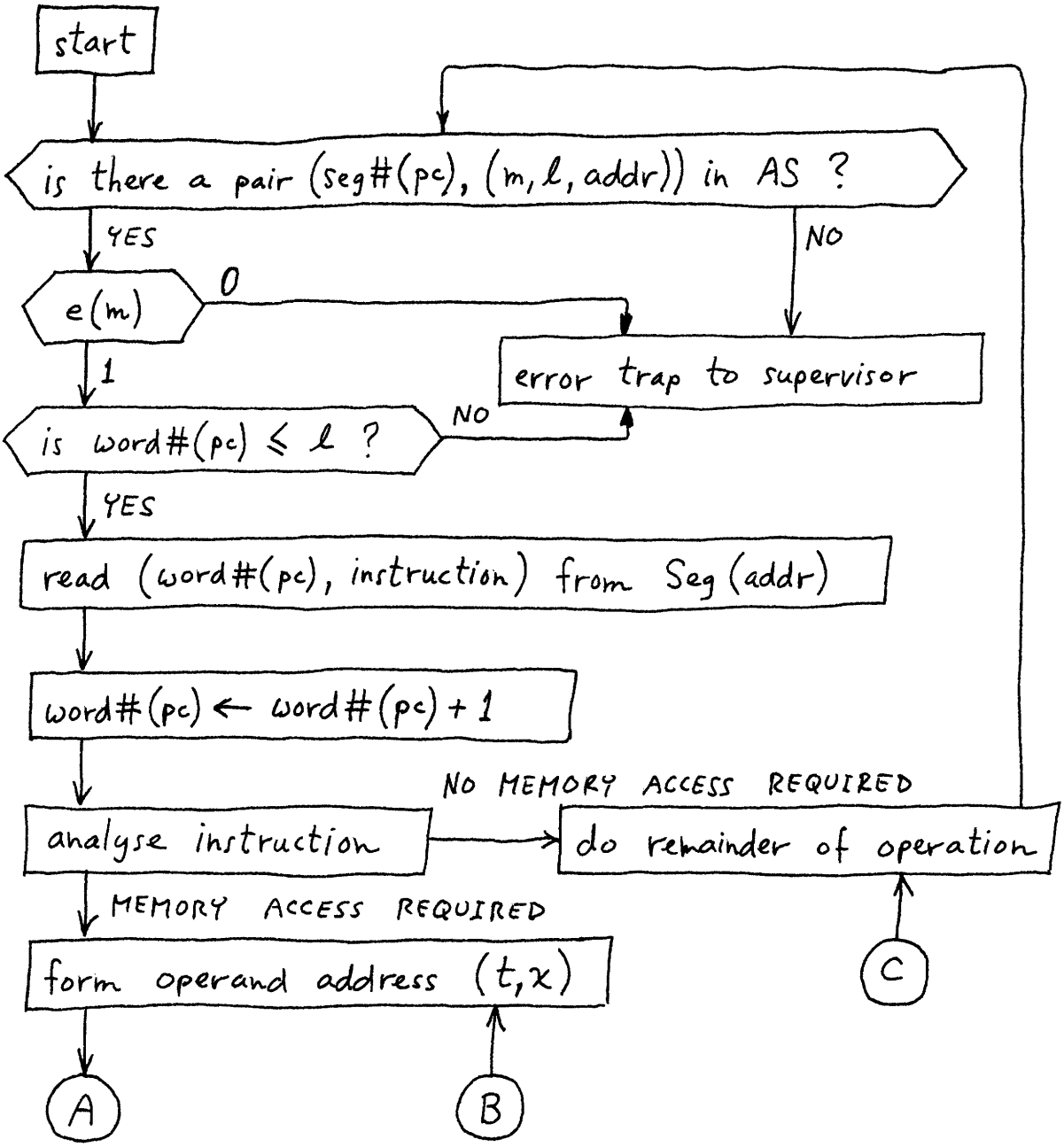


Figure 3-4, part 1. State transition rule of Multics processor.

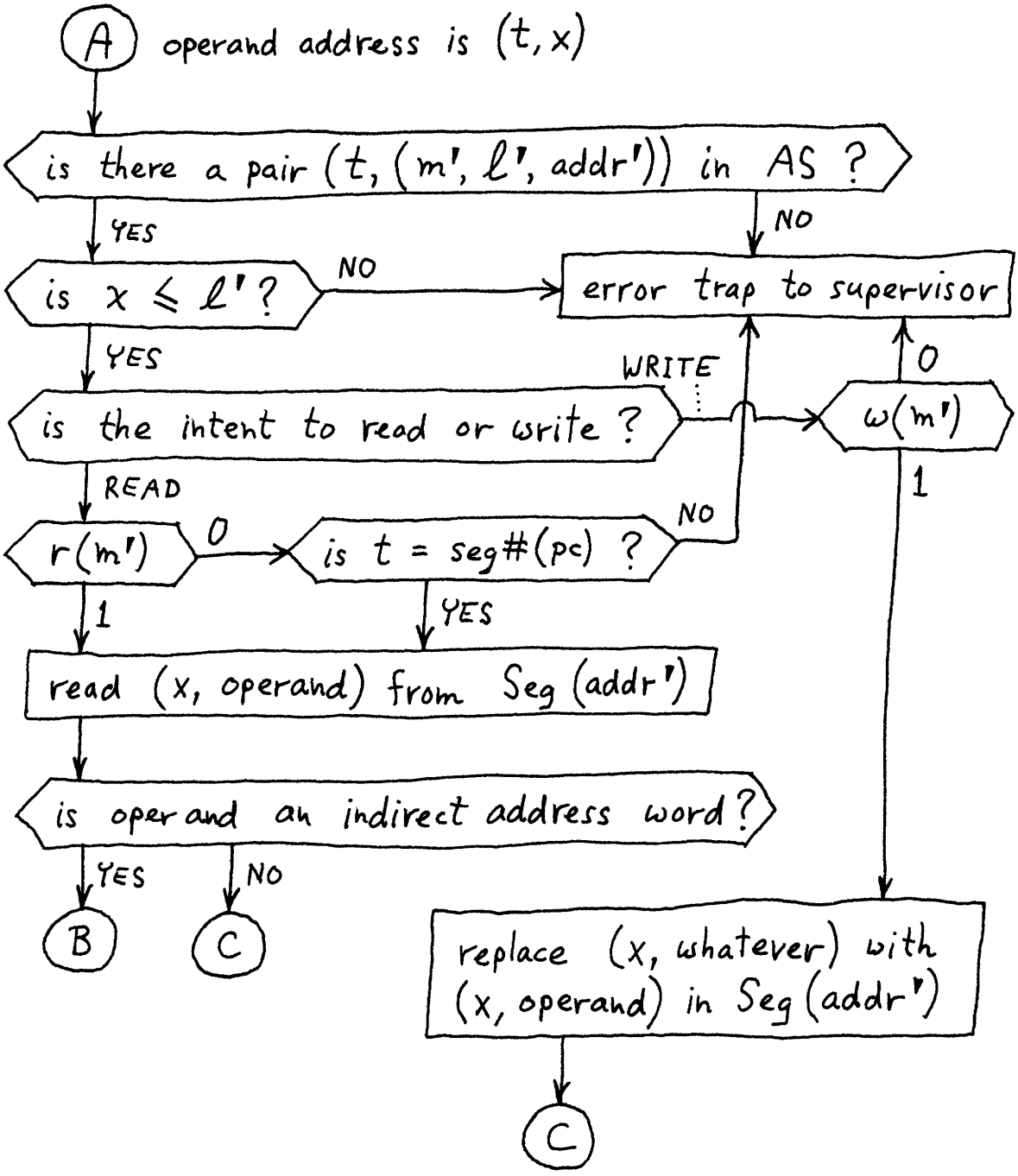


Figure 3-4, part 2. State transition rule of Multics processor.

the notation "Seg(page_table_addr)". This simply means the segment whose page table address is given. For the purposes of this model, segments are assumed to be sets of ordered pairs {(word#, bitstring)} which are functions in the set-theoretic sense, and these functions are assumed to have domains (in the set-theoretic sense again) which are intervals [0, length(Seg)] of integers. This definition of segment is more precise than the verbal definition given above ("A segment is a contiguous linear array of words of memory"), and it makes notating the state transition rule of figure 3-4 easy.

One of the more curious parts of figure 3-4 is the test in cases of read intent, "is t = seg#(pc)?". The purpose of this test is to allow executable segments to direct processes to read constants out of themselves without having the r(mode) bit on, and it makes a good example of a point which will be brought up later in this chapter.

Multics protects the secrecy and integrity of information by including the r(mode) and w(mode) bits in the triplets that make up AS, the address space. It is the information owner who has the authority to say what these mode bits shall be, for any triplet (mode, length, addr) for which Seg(addr) belongs to him (the information owner). The information owner authorizes processes to access his information by making entries on an access control list associated with the segment. An access control list entry names a process or processes

(using a name space which is not important to us in this context) and specifies the permissible access with a 3-bit string. Once the information owner does this, a process which is named by the access control list can obtain a triplet in its address space and reference the information owner's segment.

3.4. Domains

At this point it is appropriate to draw from the preceding examples that important feature which they have in common: the domain. First consider the CTSS process whose process state was the tuple ♀ = (pc, registers, {(word#, bitstring)}). The things which the process can get at (in terms of reading or writing) are the bitstrings in the set {(word#, bitstring)}, and in addition its own program counter and registers. And these very things cannot be gotten at by any other process, and so they are protected from any harmful process that might be lurking about. Of course, what the bitstrings are protected from is unauthorized reading and writing by the harmful process.

Now consider the process of Multics, whose process state is the tuple ♀ = (pc, registers, AS), where AS = {(seg#, (mode, length, page_table_addr))}. Aside from its own program counter and registers, the process can get at the bitstrings in the segments Seg(pt_addr) where pt_addr is the third component of a triplet (m,1,pt_addr) which appears in the address space AS of the process. These bitstrings cannot

be gotten at by any other process, provided no other process has a triplet in its address space whose third component is pt_addr. Whether or not such an additional triplet exists, the bitstrings in Seg(pt_addr) are protected from unauthorized reading and writing because all the triplets (m,l,pt_addr) are authorized by the owner of Seg(pt_addr).

The following definition springs from these examples: a domain is a dynamic set of abilities to use computing objects. By "dynamic," we mean varying as a function of time. Two domains can be equivalent for periods of time, just when the sets of abilities are equal, but this is not likely to occur often in practice. The abilities which constitute the domain are used by processes in the context of rules which bind processes to domains. These rules are called the Postulates of Domains.

Postulate 1: Every process is bound to one domain at a time.

Postulate 2: Only processes bound to a domain can use the domain's abilities.

Postulate 3: All information entering a process state is authorized to enter by an ability in the domain the process is bound to.

Postulate 4: All information leaving a process state is authorized to leave by an ability in the domain the process is bound to.

In the examples just given, the domains have been part of the process state. In the CTSS process, the domain is the set {(word#, bitstring)}, all of whose computing objects are bit strings which the process uses for program and data storage. In the Multics process, the set AS is a domain, in which the computing objects are segments, and the degree of freedom of use of segments is specified by modes. So the binding of processes to domains, as required by Postulate 1, was easily accomplished: the binding is implicit in the definition of the process state.

The state transition rules force each process to use only the abilities of the domain the process is bound to, thus enforcing Postulate 2. This depends crucially on the correct interpretation of the symbols "M" and "AS" in the state transition rules of CTSS and Multics, respectively. These symbols must be interpreted to mean the domains to which the processes of CTSS and Multics are bound.

The state transition rules provide no way for information to enter or leave the process state except through references to the memory M in CTSS, and references to segments in the address space of Multics, thereby enforcing Postulates 3 and 4.

The Postulates imply immediately that all the activity of computation carried on by processes is authorized, particularly including using instructions, reading data, and modifying data. Controls placed on the bindings of processes

to domains by mechanisms yet to be introduced allow information owners to be protected from the actions of both blundering and hostile processes.

We will need from time to time to speak of the abilities to use computing objects that a domain is made up of; we shall call these capabilities. In the Multics process, the triplet (mode, length, page_table_addr) is a capability which gives the ability to read, execute, or write Seg(page_table_addr), depending on the bits of mode. Rather than regarding a domain strictly as a set of capabilities, we will for reasons of easy implementation take a domain to be an array of capabilities indexed by a capability number; and this is called capability list, or C-list [De66].

Many workers have recognized that the domain is a fundamental concept in computer protection systems [De66,La71,Sc72a, Gr68]. To see why this is so, it is necessary to delve into the nature of protection.

3.5. Abstract Protection

The purpose of this section is to say what protection is, in general. The Random House Dictionary of the English language says that protection is "act of protecting; state of being protected; preservation from injury or harm." We find the center of the idea to be "preservation from harm."

Abstractly, a protection problem consists of an object, or a state of an object (the thing being protected) which we

call the ox; a harmful agent which seeks to gore the ox; and a concerned community whose interests will be harmed if the ox is gored. The reader is urged not to take these mundane words, "gore" and "ox", too literally. They are simply metaphors which allow us to deal with the question of protection on an abstract level. The essential quality of the ox is that it serves the purposes of the concerned community, and the essential quality of goring the ox is that it disserves those purposes. The reader is further urged not to attach excess significance to the word "community"; all that is meant is a set of people who are concerned about the ox.

When a protection problem exists, the concerned community will go out and hire a protection engineer; and he will design and construct a protection mechanism, which is simply a wall placed between the ox and the harmful agent, preventing the action of the latter. The choice of materials for building this wall, for example between bricks and transistors, obviously depends on the nature of the harmful agent.

The protection engineer deals with threats, wall-building technologies, costs, and work factors. A threat is a method of goring the ox. Information about threats is available from the protection engineer's own imagination, from history (e.g. the Trojan Horse), and from the remains of previously gored oxes. Wall-building technologies can be studied in the architecture of banks and police stations, or in an accredited

Institute of Technology. Usually the protection engineer can apply more than one technology to the problem, and thus generate several different wall designs for the concerned community to choose from. Their choice will be based on the costs of the proposed walls, and their work factors. The costs will include the sums of the design costs (which might include research and development), the implementation costs, and the maintenance costs for the finished walls. In addition, there may be a cost in time, as for example the time to complete research and development. And there may be a cost due to reduced functionality of the ox, if it should happen that the protection mechanism interferes somehow with the purpose of the ox. The work factors of the walls are the measures of how much time, energy, or other resources must be expended to gore the ox. A higher work factor provides more protection, but usually at a higher cost.

The concerned community will choose to build a wall based on the expected value (the work factor) and the available resources. Leonardo DaVinci designed many fortifications, but none of his designs were implemented. Perhaps he was too interested in running up the work factors, mindless of the available resources. In fact, it is in that direction that the problem is most interesting.

Now, how does this abstract model of protection help to explain the domain as defined in section 3.4? First we must

say what are the oxes, and what harmful agents seek to gore them. To do this, we need only return to the goals of this chapter: protecting the secrecy and integrity of information stored in the computer. The secrecy of information is an ox. An agent can gore this ox by making an unauthorized copy of the information. Similarly, the integrity of information is an ox which can be gored by modifying or erasing the information without authority. The concerned community consists of the information owners, who wish to prevent such unauthorized reading and writing.

The domain is a wall which cannot be penetrated by the activity of processes not bound to the domain. This property is established by Postulate 2. The concern behind this postulate is that processes not bound to the domain might be harmful agents. Since some of these processes are under the control of unknown users who might be malicious, this is a reasonable concern. The purpose of Postulate 2 is to keep the activities of every process confined to the limits established by some domain. Together, the postulates erect walls (defined in detail by C-lists) which limit the scope of the activities of processes. Therefore, the domain is a protection mechanism.

But the domain is not a complete protection mechanism. First, it says nothing about harmful activities that don't come from processes. Second, it provides little protection

against a Trojan Horse program that gains control of an innocent process and turns that process to some harmful task. Generally, a Trojan Horse program is one which, in addition to doing whatever it is advertised to do, does something that the program's users don't know about and wouldn't want done.

This fact, that a process can be innocent sometimes and not so innocent at other times, is the reason for the curious test, "is t = seg#(pc)" in figure 3-4. When the process is executing the segment in question, the state transition rule allows the process to read words from the segment. The process is presumed to be harmless because it is obeying the will of the segment in question. As soon as the process is executing some other segment, it might be trying to copy and steal the first segment. The owner of the segment can make it available as a program which cannot be stolen by authorizing e(mode) = 1, r(mode) = 0, and of course w(mode) = 0 to prevent any modification.

When a process is executing a program whose innocence cannot be assumed, it might suffice to place that program off in its own domain. This idea is expanded in the next chapter, and the resulting system is adequate to deal with suspected Trojan Horses. The only other approach to a Trojan Horse is to look inside it; i.e. audit the suspicious program. Certainly auditing of some kernel of supervisor programs of multi-access computer systems will be necessary to insure

security, but in our research we have tried to eliminate the necessity for auditing as much as possible because the cost of auditing will rise with the cost of living, while the cost of computer hardware continues to fall. When auditing is necessary and important, the program can be audited by several auditors independently, operating under a code of ethics which says they mustn't discuss with one another what they're auditing.

Now we must return to the problem of harmful activities that don't come from processes. For example, a malicious maintenance engineer using the computer's maintenance panel. And we suggest simply locking up the maintenance panel. In this research, we have assumed that all the activity comes from processes. From this assumption the reader can see the scope of this research: we have focused on processes. Certainly processes are the most interesting, active elements of multi-access computer systems; they amply deserve all this attention.

Chapter 4

Additional Protection Mechanisms

4.1. Introduction

The first purpose of this chapter is to present new protection mechanisms which will support proprietary services in a computer utility, and will reduce the cost of operating system development. These two goals are elaborated in section 4.2. The first notable protection mechanism presented here is the sectioned stack, which will be described in section 4.4. The second notable protection mechanism is a hardware processor design which will support an operating system partitioned into compartments separated by secure barriers. The detailed specification of this processor is given in Appendix 1.

The second purpose of this chapter is to present a naming and authorizing system which will provide a suitable context for the support of proprietary services. This naming and authorizing system, similar to the file hierarchy of Multics, is described in section 4.7. The third notable protection mechanism presented here is the system of certifications and warrants, described in section 4.7, which allows users to place trust in and depend on audits of programs performed by others.

The remainder of this introduction will be devoted to reviewing and defining the three types of computing objects;

i.e., processes, domains, and segments; upon which we will build our new protection mechanisms. In addition, we present a useful notation for these concepts.

As before, a domain is defined to be a set of abilities to use computing objects; and these abilities are called capabilities. These capabilities are organized into a linear array, called a capability list, or C-list. We call the index of a capability in a C-list its capability number. The Postulates of Domains stated in chapter 3 are our first design principles. For completeness, they are restated below.

Postulate 1: Every process is bound to one domain at a time.

Postulate 2: Only processes bound to a domain can use the domain's capabilities.

Postulate 3: All information entering a process state is authorized to enter by a capability in the domain the process is bound to.

Postulate 4: All information leaving a process state is authorized to leave by a capability in the domain the process is bound to.

A process is the activity of a processor, defined by a state transition rule, accessing and modifying a process state, which is a collection of information.

A segment is a contiguous linear array of words of memory, numbered from zero to some adjustable upper bound. A segment capability allows a process bound to the domain containing the capability to reference the segment designated by the capability

with a two-dimensional address of the form (seg#,word#). The component seg# is the capability number of the segment capability, and the component word# selects a word from the segment designated by the capability. Segment capabilities include 3-bit modes to control read access, write access, and execute access.

Figure 4-1 illustrates our notation for processes, domains, segments, and capabilities. The domain is shown as a large circle with the capabilities, represented by small triangles, drawn inside. The large circle serves to evoke the walling-off function of the domain. A single running process is shown bound to the domain. The domain contains capabilities for the two segments P and D. The capability for P has mode "e", so the process can obey an instruction stream that comes from P. In other words, P is executable as a program in this domain. The capability for D has mode "rw", so the process can read data from and write data into the words of D.

When we speak of the actions which the program P directs the process to do, we often use a shorthand expression of the form, "The domain does such-and-such." In this expression the program and process being spoken of must be determined from context, but this will not be difficult. Another shorthand expression is "The program in the domain does such-and-such," in which the process being spoken of must be identified from context.

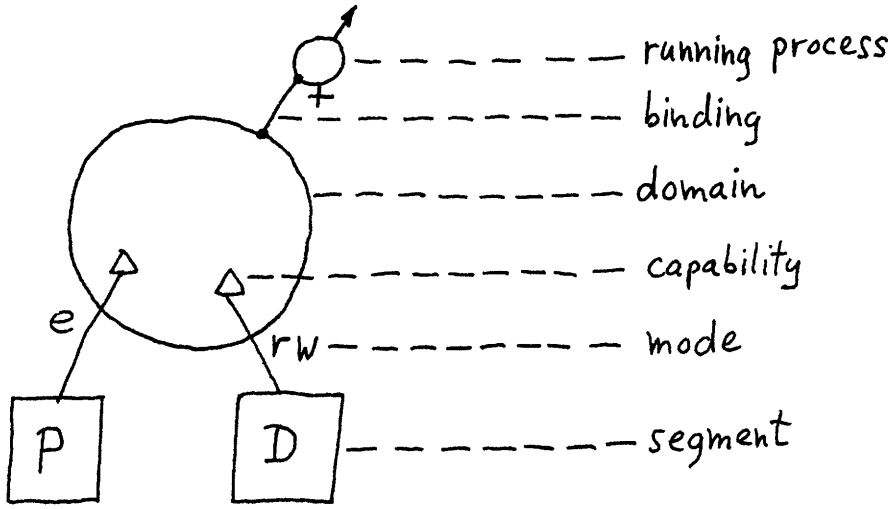


Figure 4-1. Notation for processes, domains, segments, capabilities and modes.

Figure 4-2 shows how this notation is used to represent more than one process sharing a domain, and also more than one domain sharing a segment. Part (a) of the figure shows two processes bound to one domain. Postulate 2 implies that these processes have equal access rights to the segments designated by capabilities in the domain. Part (b) of the figure shows a segment for which capabilities exist in two domains. Such a segment is called shared because a capability for it exists in more than one domain. Note that the different domains in figure 4-2(b) have different access rights to the shared segment, because the two capabilities declare different modes.

4.2. Goals

One of the long-standing goals of architects of multi-access computer systems is to allow users of computers to build on the work of others, especially in the form of programs and data. At university computer centers, users can make their subroutines available to one another through a common library. These programs are available for free, whereas in the software marketplace, programs are available for sale or lease. The first goal of this chapter is to allow users to build on the work of others in the form of programs and data, while simultaneously protecting the secrecy, integrity, and availability of the information being processed. The domain mechanism is useful in providing this

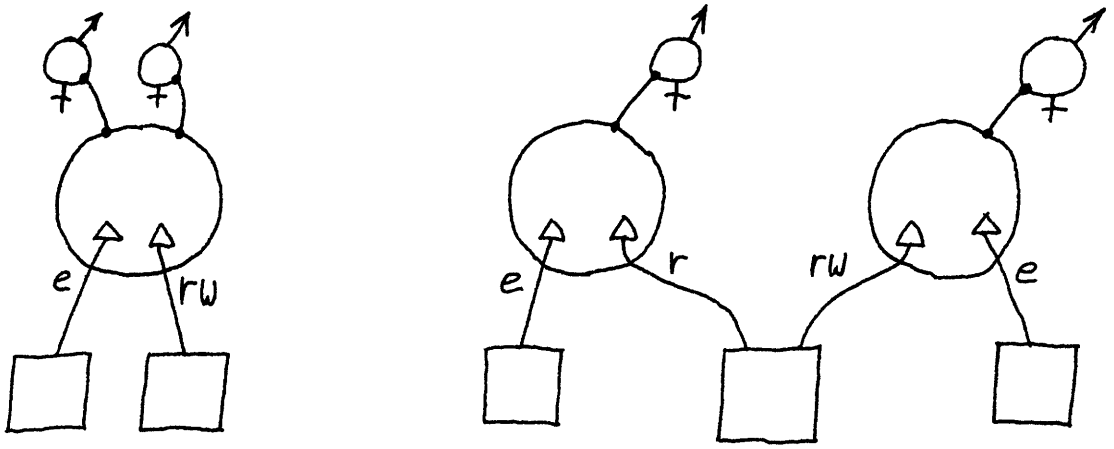


Figure 4-2. Sharing of domains and segments.

protection, provided that the domains established to protect the users' computations are small enough.

To see why small domains are necessary, consider figure 4-3. The figure shows four segments for which capabilities exist in some domain D: the segments A and B are programs, while the segments X and Y are data. A process bound to D can read and write X and Y, while executing either A or B. Now suppose that A normally uses only data segment X, and B normally uses data segment Y. There is a problem with the arrangement in figure 4-3 because A might direct a process to access Y and B might direct a process to access X. If this happened, A might spy on, modify, or destroy B's data, and vice versa. This would not be a problem if the authors of A and B trusted each other. But the necessary trust may not exist in many cases, as when B is a proprietary program obtained on a lease. Roughly speaking, the problem can be solved by using two domains.

The second goal of this chapter is to design secure barriers between the modular components of the operating system. This is expected to make the operating system faster and less expensive to debug because the impact of a single operating system failure can directly affect the data bases of only one small part of the operating system. The barriers that will be used are those of domains; the new element is the hardware processor design to support domains in which

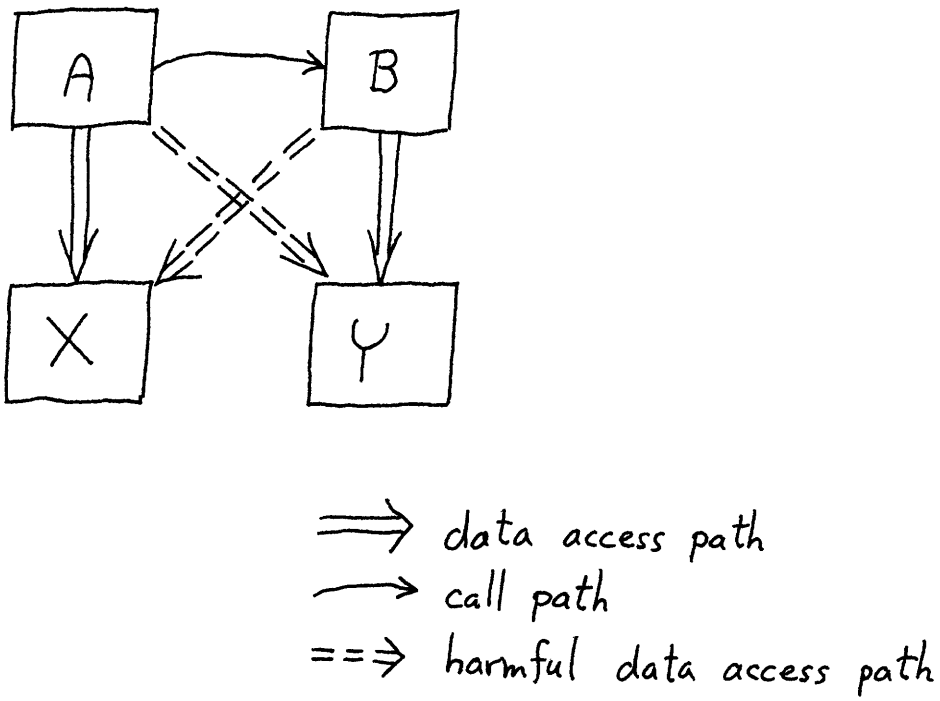


Figure 4-3. Two program segments which can cause processes to access two data segments.

the operating system can be placed.

4.3. Building on the Work of Others

In the previous section, we introduced the problems which can arise when two programs share a domain. These problems; namely, reduction of secrecy, integrity, and availability of the information being processed; can be avoided if the two programs are in different domains. But a new problem immediately presents itself: the communication of arguments and results between the two programs. The purpose of introducing two domains was to isolate the two programs, but this isolation must not be complete if the usefulness of the called program to the caller is to be retained.

Lampson's message system [La71] is one way of allowing the two programs to communicate. Each domain has an associated process, as illustrated in figure 4-4. The program A requests B's service by sending B a message, using an operating system primitive. The operating system copies A's message into B's input buffer. In Lampson's message system, the sharing of segments by domains is not allowed: all communication between the isolated subsystems is via messages. The operating system prefixes the identity of A's domain to A's message, so that B will be able to detect messages from non-customers. B returns its results in a second message. The greatest advantage of the message system is its elegance and simplicity. The PRIME system at Berkeley is an example of a message system.

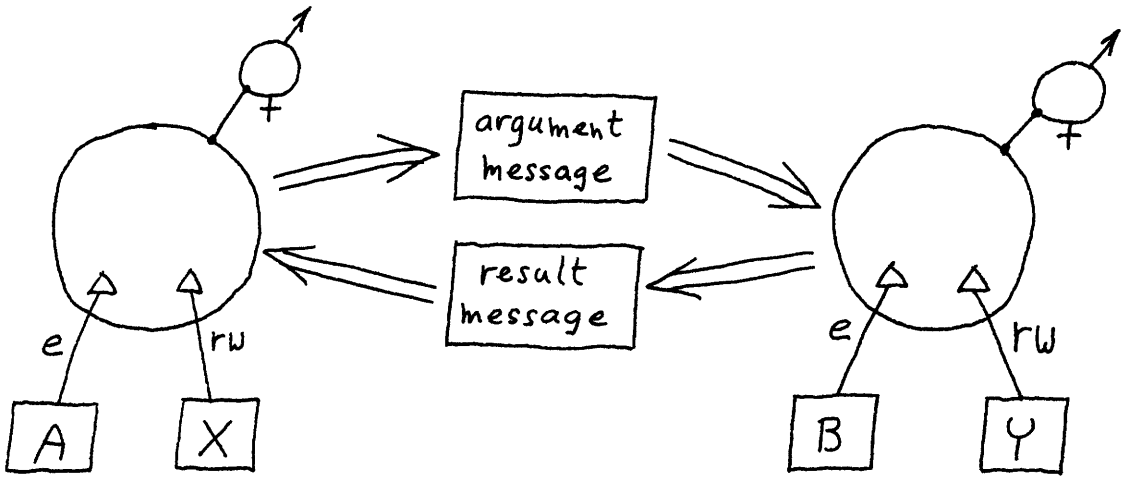


Figure 4-4. The message system.

We find that the message system has some diseconomies built into it, however. First of all, it requires two processes to do what one process could easily do. Furthermore, it does not behave well during periods of peak load. Messages pile up in B's input queue and service deteriorates (in terms of turnaround time). If B detects this condition and requests more processes to run in its domain to handle the load, then the system is creating processes when it should be processing messages. Furthermore, we feel it is inelegant to have one group of processes responding to messages while another group of processes waits. The system's users must pay the overhead costs for all of these processes.

To avoid the inelegant proliferation of processes and the diseconomy of excess process overhead, we have chosen to define our process in such a way that when it requires a service, it binds itself to the domain that provides the service. This happens when a process executes a call-domain instruction which specifies a domain entry capability. Figure 4-5 shows a domain entry capability, represented by a forked arrow emanating from the triangle symbol. The forked arrow points to the domain which can be called using the capability, and the program which the calling process is to execute. The domain entry capability also contains the word# address (within the program segment) where the calling process is to begin execution, although our notation does not represent this

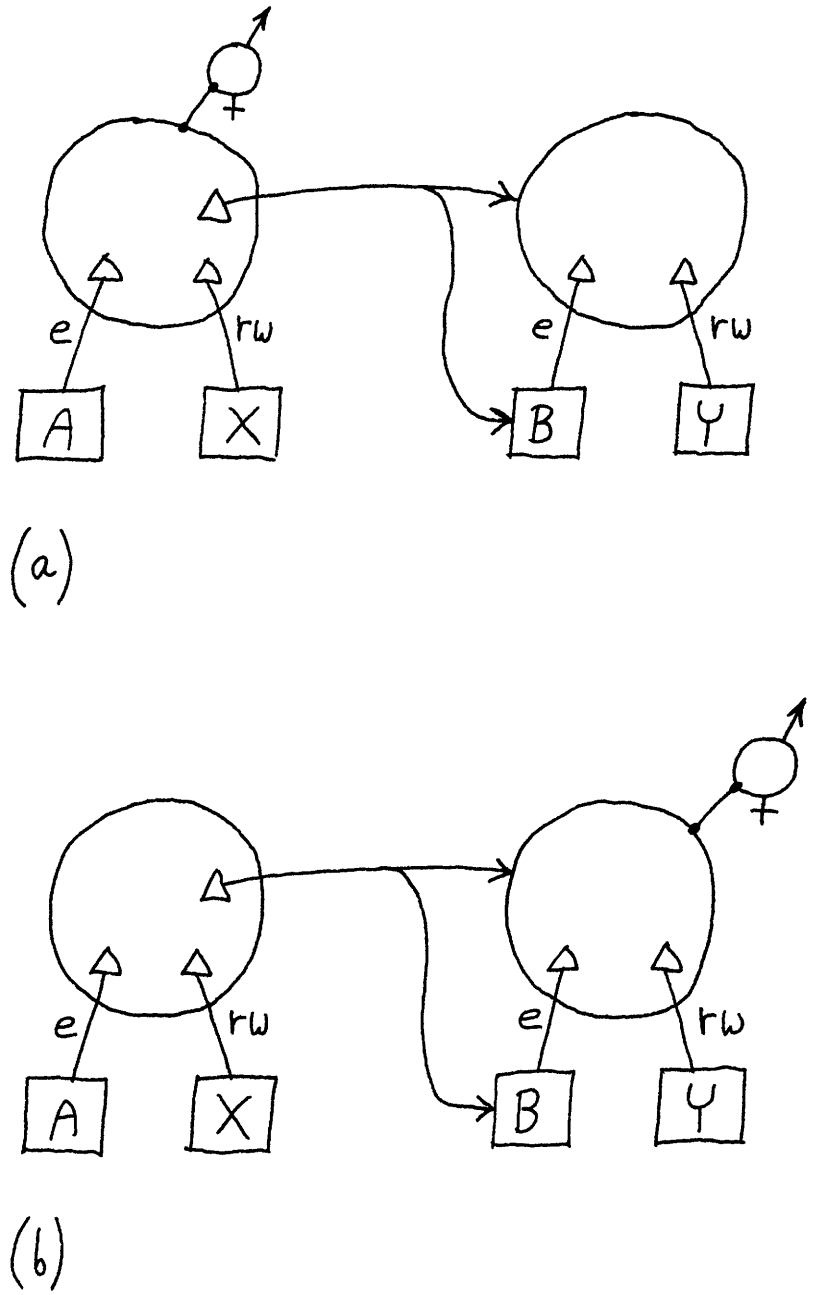


Figure 4-5. A process calling between domains.

important detail. Figure 4-5(a) shows a process before it executes the call-domain instruction, and figure 4-5(b) shows the process bound to the called domain, just after execution of the call-domain instruction. The return address in the calling domain, to which the process will jump when the called domain issues a return-domain instruction, is saved in a push-down stack which is a part of the process state. The processor state transition rule tells how to manipulate this push-down stack in response to call-domain and return-domain instructions.

When the process is bound to A's domain, it has no access to B or Y, but A can direct the process to call B because the process can use the domain entry capability. After the process is bound to B's domain, it can access B and Y and nothing more; and it is under the control of B. But there remains the problem of passing arguments and results between calling and called domains.

4.4. Argument Passing and Reclaiming

One simple way to allow the two domains of figure 4-5 to communicate arguments and results is to have these domains share a segment. (We are passing over the even simpler case when the arguments and results will fit into the registers of the process state.) Such a segment, called an argument segment, is shown in figure 4-6. The program A creates the segment and obtains a capability for it with mode "rw". Then A calls the pass-segment operating system entry point to give

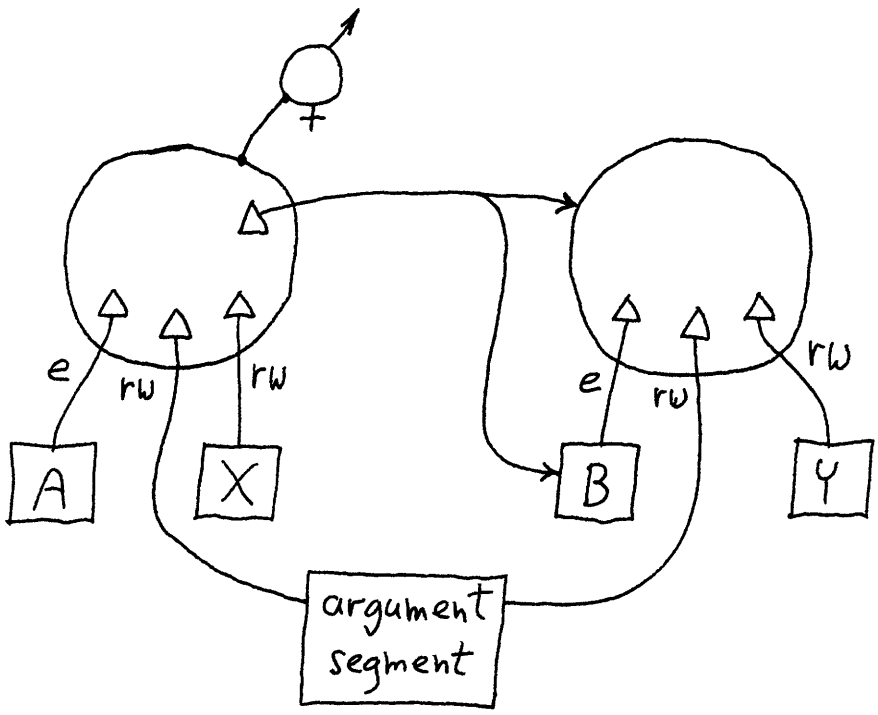


Figure 4-6. An argument segment.

B's domain a capability for the segment. Figure 4-6 depicts this point in time. The segment will probably have different segment numbers in the two domains, so the pass-segment primitive will return to A the segment number which B should use to reference the segment. A then places its arguments in the argument segment, loads the segment number for B to use into a register agreed upon by convention, and directs the process to call B by issuing a call-domain instruction.

When the process arrives in B's domain, B validates the segment number of the argument segment by calling the operating system primitive is-arg-seg. This primitive's purpose is to assure B that the given segment number is in fact the segment number of an argument segment that was passed, by the pass-segment primitive, from the domain which directed the executing process to call B. Upon being reassured, B reads its arguments from the argument segment, does its work, places its results in the argument segment, and directs the process to return.

When A regains control of the process, it can call the supervisor primitive reclaim-segment to remove the capability for the argument segment from B's domain. The purpose of this precaution is to prevent any further accessing of the argument segment by any processes bound to B's domain.

It is worth noting that the primitives allow A and B to work together even though neither of them trusts the other. B need not trust that A sends across the correct segment

number, and A need not trust that B will leave the argument segment alone after the process returns. This lack of trust is good to the extent that it helps A and B to catch errors that might occur. But extreme lack of trust would deter A's author from using B at all; as might happen if A's author knew that B would copy and steal the arguments passed by A. Such problems are treated at length in the next chapter.

The argument segment mechanism has the disadvantage of being a moderately expensive method of passing arguments and results, since it requires the creation of a segment and invoking up to three operating system primitives. When the arguments and results are only a dozen or so words of memory, as is often the case when the side effect of the call is the important thing, the large information capacity of an entire segment is not required and so the expense becomes a burden.

To meet the need for a less expensive argument passing mechanism, we introduce the sectioned stack. This new mechanism supplements, but does not replace, the argument segment mechanism. (Argument segments are economical for large arguments.)

The sectioned stack is a segment which is part of the process state, together with two registers called Min and Max which define an accessible portion of the stack. Figure 4-7 shows a sectioned stack and its accessible portion. In figure 4-7, the notation ♀⊇X means that the process state ♀ contains x as a component. The components of ♀ shown in figure 4-7 are the registers Min and Max, and a page table address (notated [→])

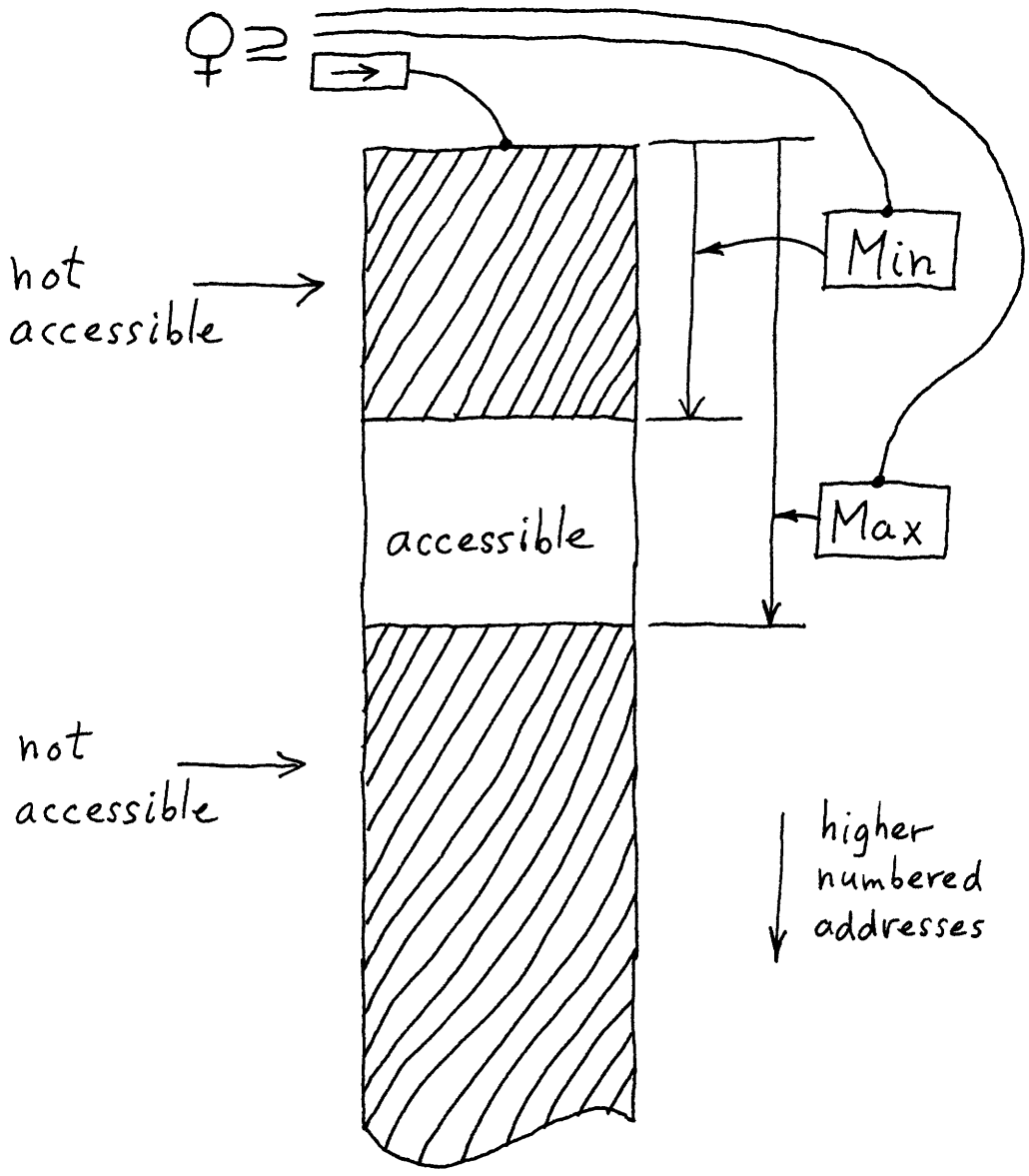


Figure 4-7. A process state and its sectioned stack.

for the sectioned stack segment. The state transition rule of the processor prevents the process from accessing words of the sectioned stack with addresses less than or equal to Min or greater than Max.

The sectioned stack is designed to be used both for passing arguments and results between domains and for storing procedure activation records. We feel it is elegant to associate a stack segment with each process, because the use of such a stack segment protects the secrecy and integrity of arguments, results, and procedure activation records in a way that absolutely prevents access by other processes. Roughly speaking, the stack-per-process concept keeps the processes out of each other's hair. This protection arises from the fact that the sectioned stack is part of the process state, rather than being accessible via a capability in a domain, which can be used by any process bound to the domain.

Figure 4-8 shows five snapshots of the sectioned stack, illustrating both argument passing and the storage of procedure activation records (which include storage of temporaries, e.g., automatic variables declared in programs written in PL/I). In the first snapshot, the accessible portion of the stack contains just the procedure activation record for A. In the second snapshot, A has prepared for a call to B by increasing Max sufficiently to make space in the accessible portion for the arguments and results; and A has loaded its arguments for B into the stack. A now calls B, and the third

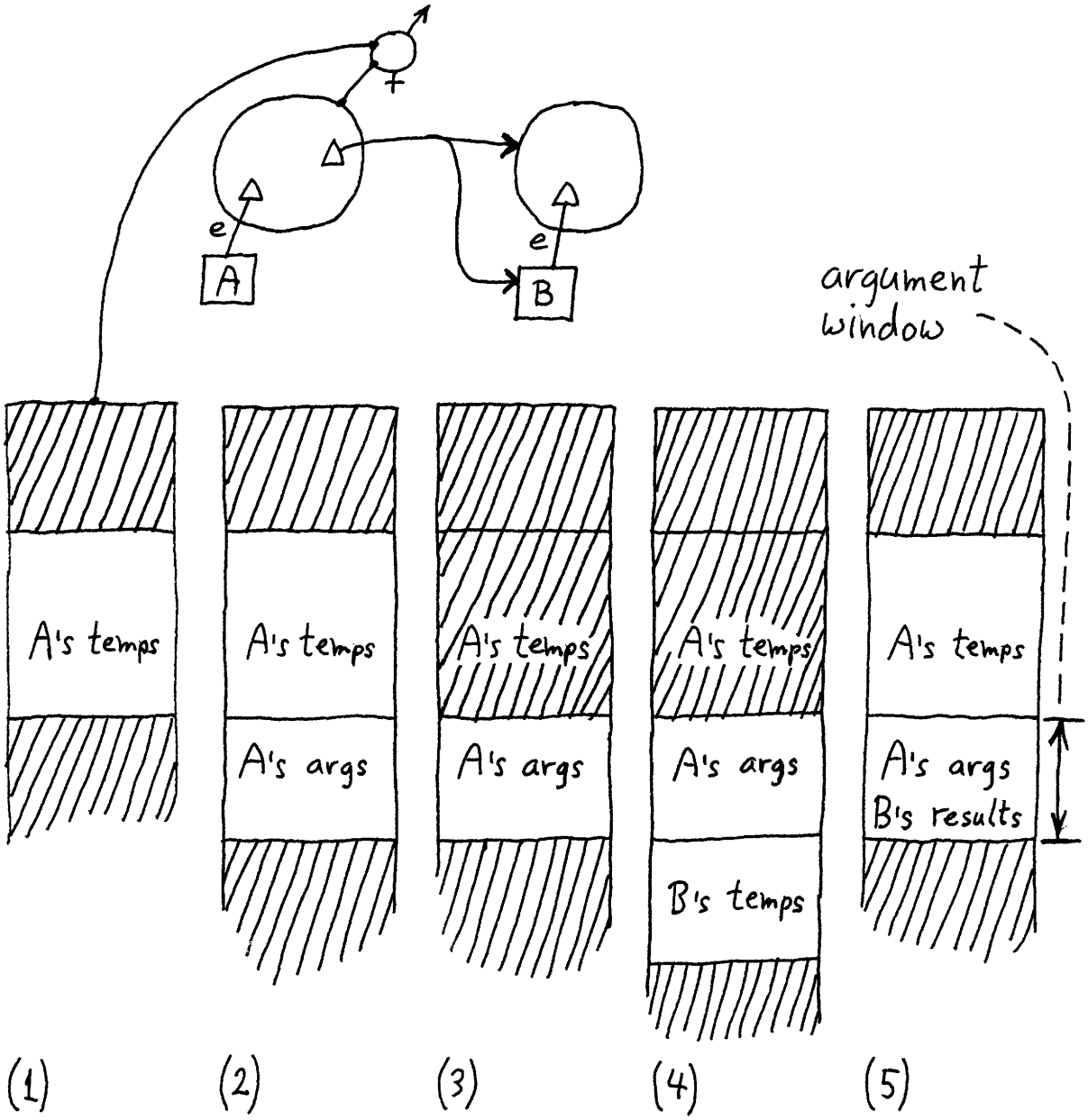


Figure 4-8. Snapshots of sectioned stack.

snapshot shows the stack just after the call. Note that Max has remained unchanged, while Min has been increased to protect A's temporaries from the action of B. The portion of the stack which holds the arguments and results is called the argument window of the sectioned stack, because it is the region which both A and B can access. Behind the scenes, in a history stack in the process state, the old values of Min and Max (just before the call) are saved. Now B increases Max to make room for its temporaries, as shown in snapshot four. B runs, does its work, and loads its results into the argument window. Then B returns to A, and snapshot five shows the stack just after the return. The saved values of Min and Max have been restored, so A can once again access its temporaries which were protected from B. B's temporaries, on the other hand, are erased automatically, so A can't access them later. Erasing is controlled by the reduction in the value of Max.

The sectioned stack allows caller and callee to share one segment for storage of temporaries, arguments, and results; even though caller and callee do not trust one another. The secrecy and integrity of A's variables are protected by the "wall" erected around Min: Min may be reduced only by the return-domain instruction. This is accomplished by the processor state transition rule. The secrecy of B's variables is protected from the action of A by erasing B's variables

just before control is returned to A. The integrity of B's variables is protected simply because control is not given to A while B's variables are yet to be used again.

Erasing the stack as Max is reduced is specified in the state transition rule. In fact this activity can be overlapped with other activity of the process, as for example by zeroing words in the machine's cache. The cache is primarily the topmost element of the memory device hierarchy, but it has the bandwidth and could be wired to zero stack words for the processor.

If B should call a third domain containing a program C, the sectioned stack will serve to protect B's variables from the action of C and vice versa; and it also protects A's variables from the action of C and vice versa.

4.5. The Binding of Processes to Domains

We first introduced the idea that processes are bound to domains in chapter 3, where the binding served to help define the set of computing objects that the process is permitted to extract information from, to manipulate, or to otherwise use; according to the Postulates of Domains. In section 4.3, we introduced the idea that this binding may be time-varying. This permits a process to obtain a service from a program encapsulated in a domain D by calling domain D using a domain entry capability and the processor's call-domain instruction; and while the service is being performed, the process is bound to domain D. When the service encapsulated in D has

finished its work, it issues a return-domain instruction and then the process becomes bound again to the domain that called D.

The first purpose of this section is to note the dependence of overall system security on the non-forgeability of the binding of processes to domains. If a process could forge its domain binding, it could call any service encapsulated in any domain in the computer system, whether it was authorized to use that service (by means of a domain entry capability) or not. The non-forgeability of the binding of processes to domains is a property of the computer system which is a logical consequence of the processor state transition rule, which tells how and when this binding is to be changed. The state transition rule specifies that the binding of processes to domains is to be changed only by the call-domain instruction, the return-domain instruction, and system faults. The complete state transition rule is given in Appendix 1.

Because system security depends on the correctness of the binding of processes to domains, the processor hardware which implements this binding, including but not limited to the process state components dom_id, vb, and dom_pt_addr; should be constructed from failure-detecting, failure-correcting circuitry. (These process state components are defined in Appendix 1.) The engineer who designs a hardware realization of the state transition rule of Appendix 1 should compute bounds

on the probability of the binding being incorrect and this failure going undetected; and an explication of this computation should be demanded by would-be users of the hardware.

The second purpose of this section is to explain a design choice relating to address spaces. In our model, address spaces are associated with domains, and therefore the binding of a process to a domain serves to select an address space for the process. In other words, the meaning of a segment number is defined in the context of a domain, and every process bound to a given domain referring to a given segment uses the same segment number. This feature of our model should already be clear to the reader from our discussion in section 4.4 of validating the segment numbers of argument segments. Address spaces are associated with domains also in a new computer with advanced protection mechanisms being constructed at the University of Cambridge. [Ne72]

It is possible to associate address spaces with processes instead of with domains, in which case the domains must be part of the process state. Figure 4-9 shows two such processes, each of which contains a domain that encapsulates the procedure and data segments P and D. The capabilities in the first process (notated φ_1) for P and D are assigned segment numbers 5 and 6, respectively; whereas the capabilities in the second process (notated φ_2) for P and D are assigned segment numbers 6 and 7, respectively. We assume that there is no co-ordination

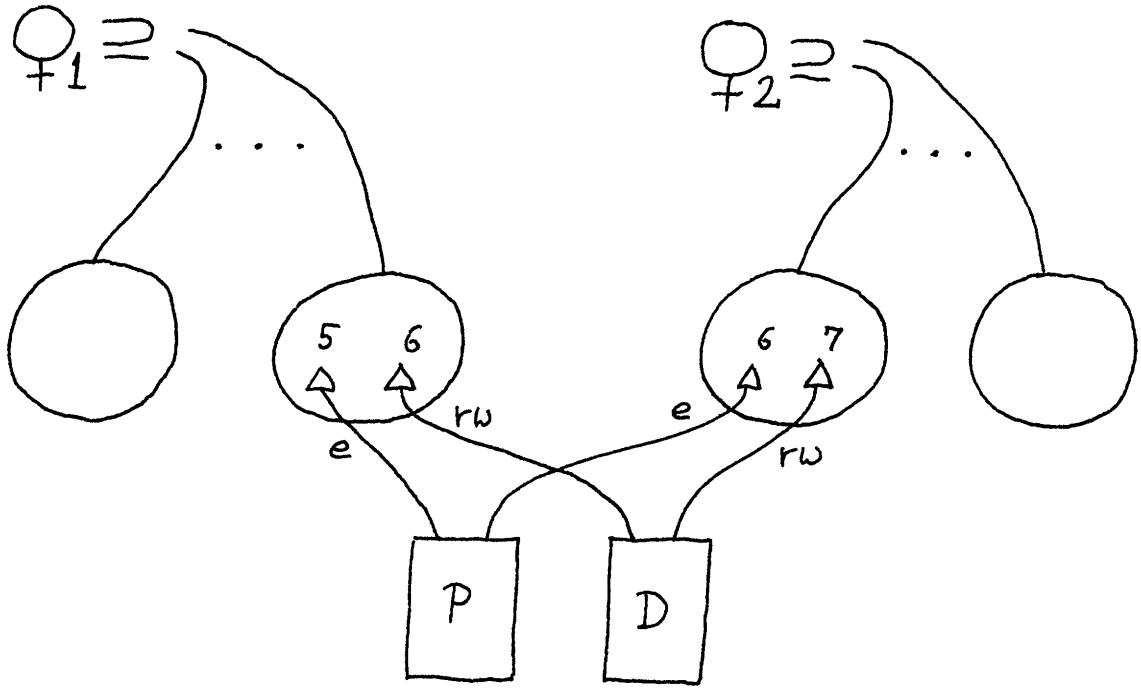


Figure 4-9. Two processes sharing "one domain".

in the assignment of segment numbers between processes; so in general, P and D will have different segment numbers in the different processes. This method is used by Schroeder [Sc72b] in his model of mutually suspicious subsystems. The problem with this method is that when the program P must maintain a data base in a linked list format, which is quite common when P is a generalized data base manager (e.g., the File Manager in Multics [MIT72,p.4-55]), the pointers must be stored in process-independent form. In the example of figure 4-9, this problem is manifest whenever it is necessary to store a pointer that points into D. The process-dependent form of pointers, (seg#,word#), cannot be used because D has different seg#'s in the different processes. Process-independent pointers must be used instead, and each process must translate these pointers into the (seg#,word#) form required by the hardware for effective addressing. This translation is not necessary when address spaces are associated with domains. Thus this translation step appears to be a waste of time, and is likely to deter the growth of parallel computation methods when address spaces are associated with processes, because of the added burden placed on parallel computations.

The third purpose of this section is to explain two interrelated design choices relating to the name space of domains from which a selection is made by the binding of a process to a domain. Stated simply, the questions are whether this name

space should be large or small, and whether it should be implemented once per-process or once per-system.

It might appear economical to design a small name space for the domain binding if the designer expected each process to use only a few domains. That is, if the process is expected to bind itself to only a small number of domains in its lifetime (e.g., 8), then the domain binding component of the process state might be implemented as a small register in the processor hardware (e.g., 3 bits). But this seemingly economical choice has a diseconomy built into it, because the community that is using this computer system with a small name space for the domain binding will be using it to build services that use services built by other. Eventually, which could be quite soon if the per-process name space were very small, somebody would build a service that used up all the available domain names. But then noone could build a service that used this service, and planned obsolescence would strike again. The conclusion of this argument is that the name space for the domain binding should be a large name space.

The second question relating to the name space of domains is whether it should be implemented once per-process or once per-system. We feel that a per-system name space is more elegant, and this choice insures that the name space will be large. Also, some economies of scale might be realized from maintaining one central, per-system name space as opposed to maintaining many per-process name spaces.

In our design, the domain binding name space is a collection of unique identifiers, called domain identifiers. These domain identifiers are found, for example, in domain entry capabilities, where they serve to identify the domain to be called. The operating system maintains a mapping from domain identifiers to page table addresses of the C-lists(*) of domains, so that the processors can effectively examine the capabilities of a domain, given the domain's unique identifier. This mapping, which is an associative memory for the domain binding name space,is called the Active Domain Table (ADT). The ADT is introduced in Appendix 1 and described in Appendix 2; but we suggest that the following section be read before Appendix 2.

4.6. The Operating System

The purpose of this section is to introduce our design of an operating system which will multiplex the resources of a hardware computer whose processors are defined by the state transition rule in Appendix 1. Our design provides for multiplexing the memory and processor resources of the hardware computer, for supporting the protection mechanisms described in this chapter, and for protecting the operating system itself.

(*) Recall that C-list means capability list. The C-list of a domain is a linear array of capabilities that defines the domain.

In our design, we have assumed that the memory resources of the computer are made available in the form of segments. In fact, this assumption of a segmented virtual memory is supported by the technique of paging, which requires the operating system to maintain page tables for all segments. We will describe how the limited memory space which is available for page tables is multiplexed among all the segments which require page tables.

The processor resources of the hardware computer are multiplexed among all the processes being evolved by the computer, using the technique of time-sharing. We will describe the traffic controller, a part of the operating system, which accomplishes this sharing.

The operating system supports the protection mechanisms defined in this chapter in several ways. The operating system maintains the C-lists which define the domains of the system, and it protects the domains by requiring that all changes to C-lists be properly authorized. The operating system supports processes calling and returning between domains by maintaining the Active Domain Table, which holds information needed by such processes. It also maintains a collection of sectioned stacks, one for each process; and it insures that each process can access its own, and only its own, sectioned stack. Finally, the operating system implements the three primitives which allow entire segments to be used to hold arguments and results of an inter-domain call.

In addition to the protection mechanisms for users just described, the operating system is programmed to protect itself. This protection, like the protection for users, comes from the walls defined by domains; and also from complete validation of arguments passed to the operating system by calling processes; and also from authorization mechanisms, described in section 4.7, which limit what the operating system will do for processes calling it.

Figure 4-10 shows six domains of the operating system. After we describe the functions of these six domains, we will explain our reasons for partitioning the operating system into domains with these particular functions. The arrows between domains in figure 4-10 represent domain entry capabilities which allow the operating system domains to call one another. The arrows with floating tails notated "<type> fault" also represent the movement of processes between domains, but these are processes coming from any domain in the system, in response to a fault.

We now turn to describing the individual domains of the operating system. The ADT domain maintains the Active Domain Table, which provides a mapping from domain identifiers to page table addresses of the C-lists of domains. When a process calls or returns between domains, the domain that the process is becoming bound to will be called the target domain. Whenever a process calls or returns between domains, the processor must change a component of the process state to point to the

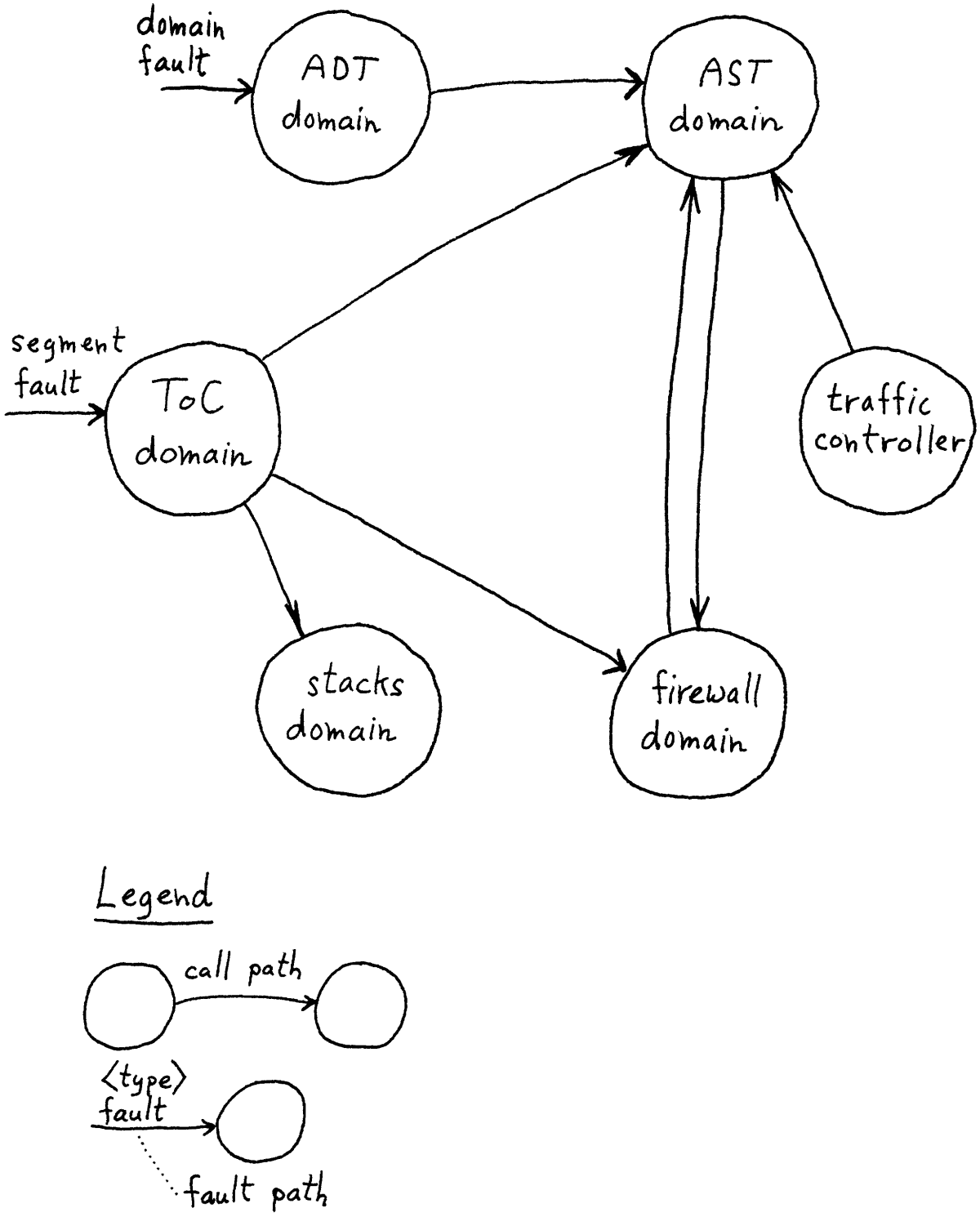


Figure 4-10. Domains of the operating system.

page table of the C-list of the target domain. If the needed information is in the ADT, the movement of processes between domains is very quick. When the needed information is not in the ADT, a domain fault occurs and the ADT domain makes the target domain active. Appendix 2 details the structure of the ADT, the algorithm that responds to domain faults, and the process synchronization strategies which are required because every process in the system is using the ADT.

Page table addresses for C-lists of domains of the operating system will always be in the ADT. Thus calls between operating system domains are quick, and a cascade of domain faults is not possible.

The AST domain maintains the Active Segment Table, which is the place where page tables of segments are stored. The memory space for page tables is multiplexed because the processes refer to time-varying collections of segments and the page tables for these segments occupy expensive high-speed memory. But the ADT contains page table addresses, and so the maintenance of the AST and the ADT must be co-ordinated.

The traffic controller domain multiplexes the processor resources of the hardware computer. Recall that each process has a sectioned stack. Because each process being evolved by a processor is bound to its sectioned stack by means of a page table address, the traffic controller co-ordinates its work with the AST domain in order to guarantee that the sectioned stack segments of running processes are active.

One domain of the operating system, the firewall domain, is responsible for manipulating and protecting C-lists. Every C-list segment in the computer system is accessible in the firewall domain, including the C-list of the firewall domain itself. The firewall domain protects itself and its C-lists by not allowing any C-lists, including its own C-list, to be accessible from any other domain.

We must note that the protection which the firewall domain enjoys depends upon the correctness of the page tables which define access paths to C-list segments. So the protecting power of the firewall domain could be violated by accident or maliciousness in the AST domain.

While the firewall domain has the power to manipulate C-lists, it does not have the responsibility for deciding what manipulations are to be performed. This is done by the AST domain and the ToC domain, and these two domains are the only domains which have domain entry capabilities which allow calling the firewall domain.

The Table of Contents (ToC) domain maintains the segments which serve to explain and associate information with C-list entries, similar to the KST in Multics [Ben72]. For every C-list segment in the firewall domain, there is a corresponding table of contents segment in the ToC domain. The ToC domain responds to segment fault events, which in effect are requests that inactive segments be made active. It does this

by determining the identity of the inactive segment from the appropriate table of contents segment and by calling the AST domain and the firewall domain to make that segment active and accessible (to the extent specified by mode information stored in the table of contents segment).

Appendix 3 details the strategies of the ToC domain, the AST domain, and the traffic controller in multiplexing the page table memory space.

Finally, the stacks domain allows the operating system to have unrestricted access to the sectioned stacks of processes. Such access is required to implement the argument segment primitives introduced in section 4.4. Appendix 4 details these argument segment primitives.

The boundaries between the domains of the operating system, as just outlined, were chosen with two design principles in mind. The first principle is functional modularity, according to which the domains are defined by the maintenance requirements of particular data bases, e.g., the AST, or the ADT; and the domains are limited to the functions of maintaining those data bases because we want to minimize the number of program modules which can access the crucial operating system data bases. This desire to minimize the number of programs in each operating system domain springs from the fact that programs are the sources of bugs, and bugs are the source of high costs in developing operating systems. It will be easier to debug an operating system which is partitioned into domains,

because the direct damage to operating system data bases caused by each bug will be localized and minimized.

The second design principle which guides our partition of the operating system is simplicity in allocating segment numbers in the domains of the operating system. For example, the ToC domain and the firewall domain have a congruent allocation structure in their address spaces. Each C-list segment is allocated a segment number in the address space of the firewall domain, and similarly for the table of contents segments in the ToC domain. Every C-list has an associated table of contents segment, which is why we call the allocation structures of the two address spaces congruent. In fact, each C-list and its associated table of contents segment could have the same segment number, since segment numbers have meaning in the contexts defined by domains. The ToC domain and the firewall domain could be combined into a single domain which would contain capabilities for both C-list segments and table of contents segments. But this would complicate slightly the allocation structure of the address space of the combined domain. Also, because any implementation of this design will place some limit on the number of capabilities in a domain, combining the ToC domain with the firewall domain would reduce by a factor of two the total number of domains supported by the operating system.

4.7. Naming and Authorization

The purpose of this section is to describe the structure and control of a name space, implemented by the operating system, which users of the computer will use to catalogue segments and domains. This name space takes the form of a hierarchy of directories and their entries, and is called the naming hierarchy. The naming hierarchy is similar to the file hierarchy of Multics [MIT72].

Names of computing objects which are catalogued by the naming hierarchy are defined in terms of directories. A directory is a node in the tree of the naming hierarchy, consisting of entries which point to other directories, or to computing objects like segments and domains. Each entry has an associated name, which uniquely selects the entry from the directory. A unique directory, the Root, is not pointed to by any entry in another directory. The tree name of a computing object catalogued in the naming hierarchy is the sequence of entry names which defines a path from the Root directory to the computing object. Figure 4-11 shows a naming hierarchy containing four directories, a segment, and a domain. The tree name of the segment is (a,x,x); the tree name of the domain is (a,x,y). The entries in directories are represented by lines drawn from the directories to the object pointed to by the entry, and entry names are written next to this line.

Control over the naming hierarchy is established by giving control over particular directories to particular domains.

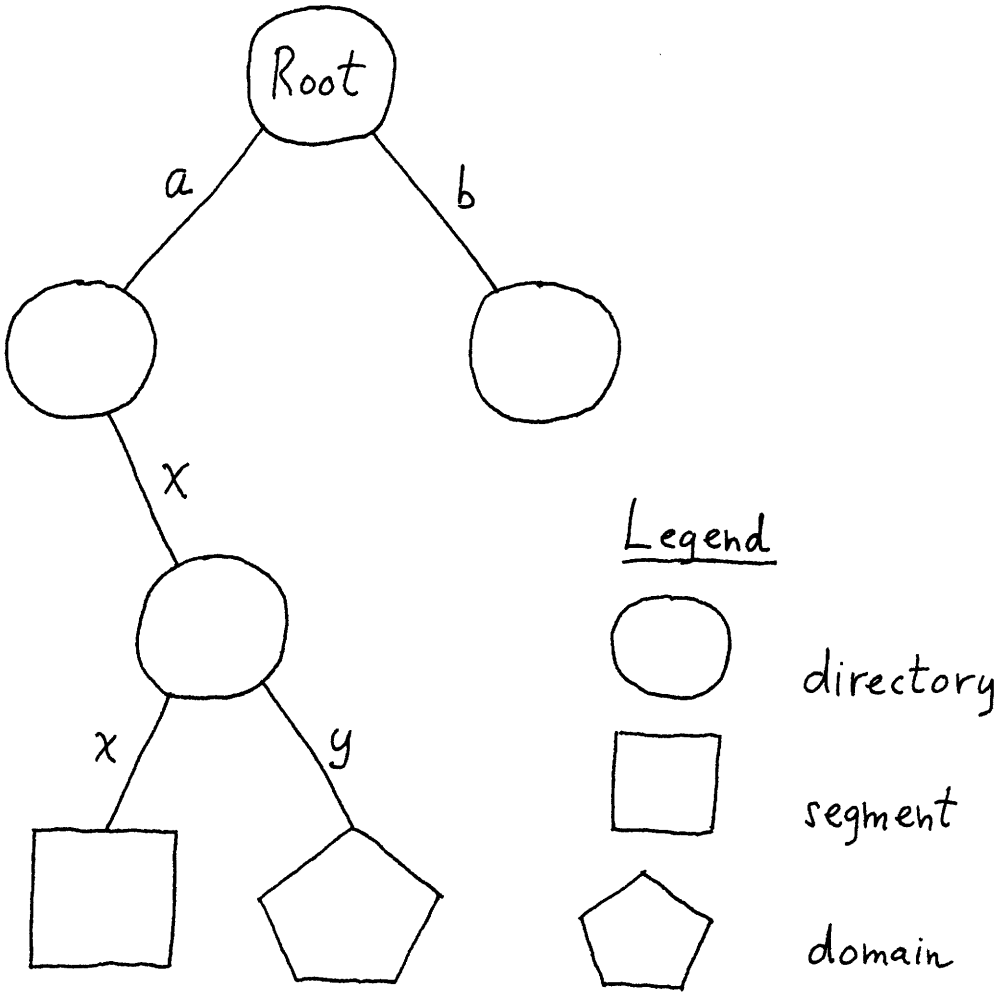


Figure 4-11. A naming hierarchy.

This control is implemented with an access control packet (acp) associated with each entry of each directory. The contents of acps are different for each type of object that can be represented by an entry in a directory. We will describe what an acp must contain to control access to directories, segments, and domains.

The acp of a directory consists of a list of terms, and each term consists of the tree name of a domain and a mode of access to the associated directory. The meaning of each term is that the named domain may access the directory using operating system primitives which are permitted by the given mode of access. Directory-accessing primitives are available which obtain a list of the entries in a directory, and which display the status of single entry, including its acp. These operations are permitted if the call to the primitive comes from a domain with "read" access to the directory. This is the first mode. A second mode, "modify", allows the authorized domains to successfully invoke primitives which create and destroy entries in directories, rename entries in directories, and modify access control packets associated with entries in directories.

Figure 4-12 shows how one directory of the naming hierarchy is placed under the control of a particular domain. The directory (users,Proj,Pers), a typical user's directory, is under the control of the domain (users,Proj,Pers,home), which is the domain named by the single term in the acp of (users, Proj,Pers). The characters after the colon in our denotation

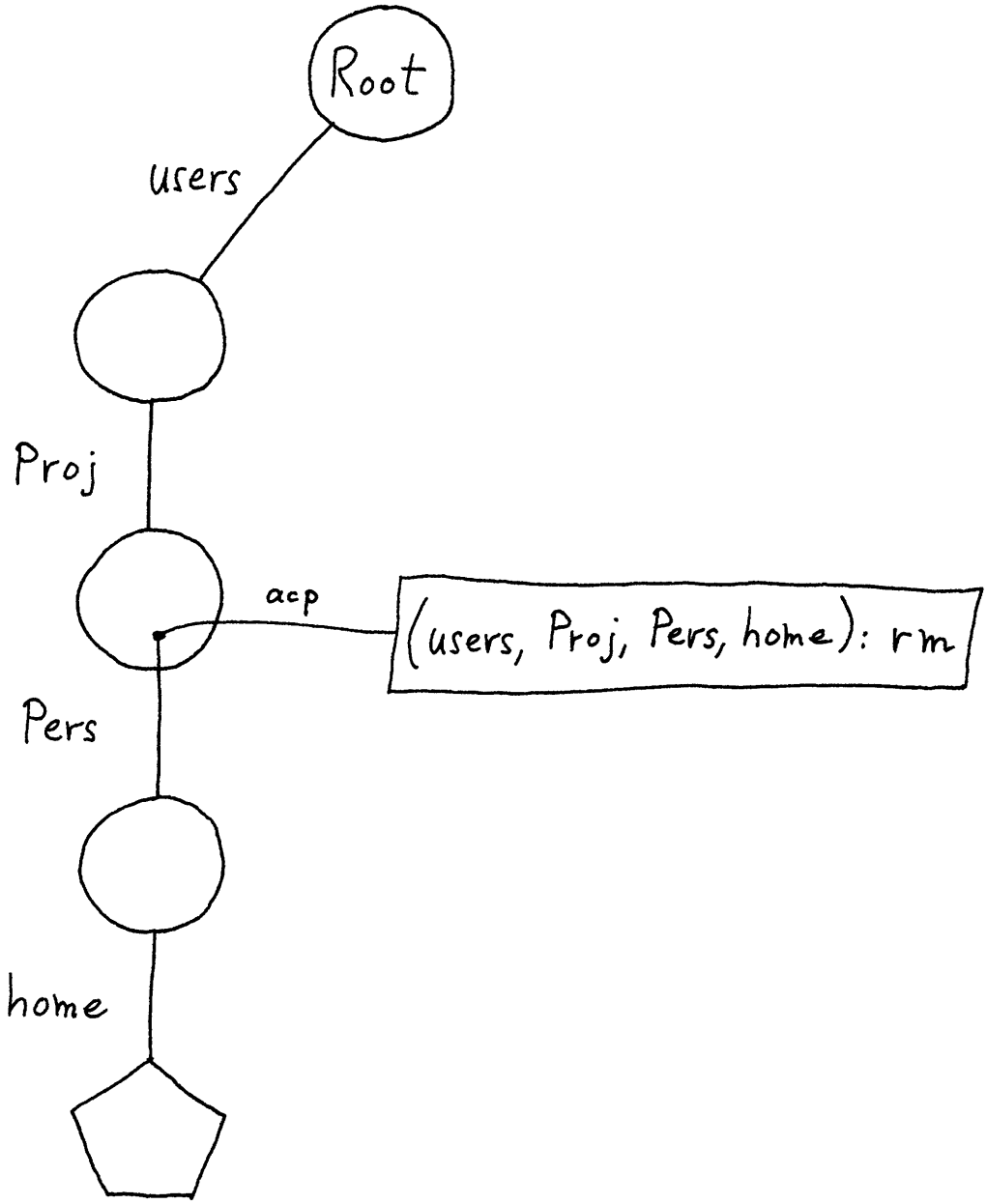


Figure 4-12. The acp of the directory (users, Proj, Pers).

of the acp term, "rm", declare that the named domain has read ("r") and modify ("m") access to the directory with which the acp is associated. Our example assumes that users (persons) are organized into projects, so (users,Proj) will have directory entries for other persons belonging to the project Proj; and (users) will have directory entries for other projects. This pattern of user and project directories is conventional in Multics.

The acp of a segment consists of a list of terms, and each term consists of the tree name of a domain and a mode of access to the associated segment. The meaning of each term is that the named domain may obtain a segment capability, giving those modes of access specified by the term, for the segment. The modes of access to segments defined by the processor state transition rule are the modes which can be specified by the terms of a segment's acp: read ("r"), execute ("e"), and write ("w").

In addition to the domain tree name and mode, each term of a segment acp contains a one-bit copy flag, the purpose of which is to authorize (prohibit) the named domain to pass (from passing) segment capabilities for the segment to other domains, using the pass-segment primitive. But note that this copy flag does not prevent any domain that has read access to a segment from making a copy of the information in a new segment and passing a capability for that new segment.

Suppose a user is commanding a process bound to a domain that has modify access to a directory which contains an entry for a segment. Such a user has the power to modify the acp of the segment. If the user commands his process to call the operating system to remove a term from that acp, the effect is to revoke the access of the domain named by the term to the segment whose acp was modified. It is not difficult to design an operating system which implements access revoking by immediately removing segment capabilities from the affected domains. For example, under some conditions Multics provides immediate access revoking.

Figure 4-13 shows how access to one segment is authorized for one domain. The segment (users,Proj,Pers,memo) is readable and writable in the domain (users,Proj,Pers,home).

The acp of a domain is a much more complicated object than the acps for segments and directories. This complexity arises from the fact that domains have several different types of uses. In the following paragraphs we will describe five different modes of usage for domains, and extract from each description the need for a separate component of the access control packet of a domain.

The previous two figures in this section both showed a user's home domain, located (in terms of its name) in a user's directory (users,Proj,Pers). Figure 4-12 showed that the home domain typically has control of the user's directory. The purpose of the home domain is to provide a protected

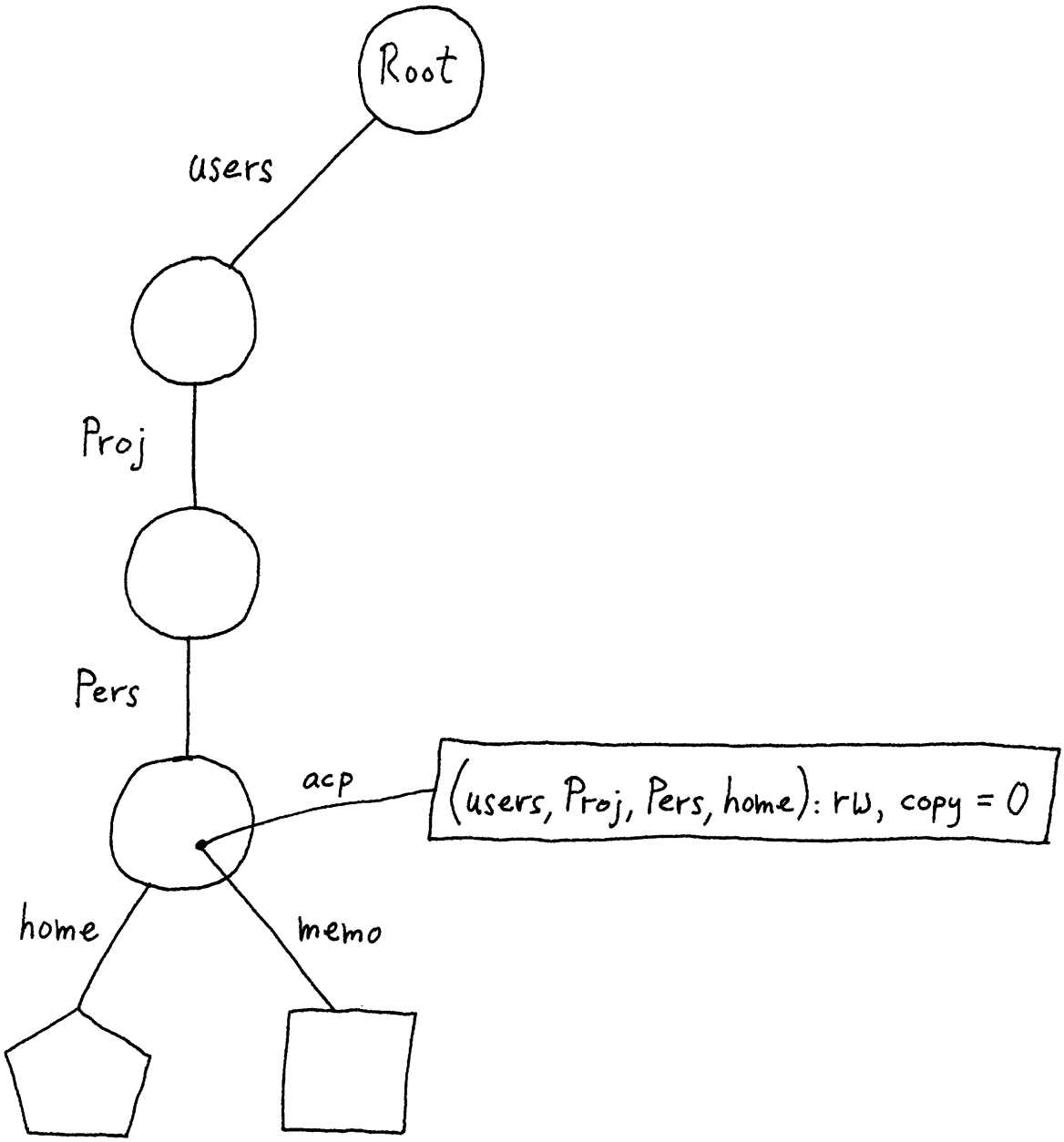


Figure 4-13. The acp of the segment (users, Proj, Pers, memo).

environment for the user's process to be bound to while the process is obeying the user's commands to examine and manipulate his user's directory, or its entries (or their entries, if some of the entries of the user's directory are directories, and so on). For example, the user could command his process, running bound to his home domain, to edit the segment (users, Proj,Pers,memo) shown in figure 4-13; also, the user could command his process to modify the acp of (users,Proj,Pers, memo) so as to share the segment with another user. This process would relay this request to the operating system by invoking the appropriate primitive, and the request would be honored because the home domain, from which the primitive is invoked, has modify access to the directory (users,Proj,Pers).

Now suppose a foreign program were introduced into this user's home domain, and suppose that program gained control of a process bound to the domain. The foreign program could perfectly easily invoke the operating system primitive to modify the acp of (users,Proj,Pers,memo), giving access to the segment to a spy. For this reason, it is necessary for the owner of a domain to have control over the collection of segments for which his domain contains a segment capability. Protection from foreign programs is the first use for such a control mechanism. In addition, this control can be used by a project administrator to define limited service subsystems in home domains of project members, when the home domains are under the control of the project administrator.

To provide control over the collection of segments for which a domain contains capabilities, the access control packet for a domain contains a component called seg-limit. This component consists of two lists: a list of terms which represent individual segments, and a list of acceptable certifications. Each term which represents an individual segment consists of the tree name of the segment, and a mode which is some combination of read ("r"), execute ("e"), and write ("w"). The interpretation of each term is that the domain with which the acp is associated may obtain a segment capability for the segment named by the term, with a mode of access not greater than the mode specified by the term. The owner of the domain will place terms representing individual segments in the seg-limit component of his domain's acp when he has certified for himself that his domain should have a capability for the segment.

The second list in the seg-limit component of the acp of a domain allows the domain owner to depend on certifications performed by others. For example, the command interpreter and the directory-listing and manipulating command programs which run in home domains will be certified by some central authority, and the list of acceptable certifications in the seg-limit component of the acp of a domain allows the domain owner to express his trust and acceptance of such certifications. Certifications are designated by ordered pairs consisting of the tree name of a domain and a character string

called a warrant. Any domain can originate a certification by invoking an operating system primitive; warrants serve to distinguish between different certifications that originate in the same domain. The operating system primitive that originates certifications will attach a certification to the segment specified by the domain invoking the primitive, and that certification will consist of the tree name of the invoking domain together with a warrant specified by the invoking domain. The certification originating primitive will also associate a mode of access, specified by the invoking domain, with the certification attached to the segment being certified. The meaning, then, of a certification specifier in the second list of the seg-limit component of the acp of a domain, is that the domain is allowed to obtain segment capabilities for segments to which the specified certification is attached, and the mode of the domain's segment capability must not be greater than the mode associated with the certification attached to the segment.

Now that we have described the mechanism which establishes the domain owner's authority over the collection of segments for which his domain contains a capability, it is appropriate to say that the domain owner is responsible for the composition of that collection of segments. In particular, the domain owner is responsible for bringing together the collection of capabilities for program segments in his domain, and therefore the domain owner is responsible, together with the authors

of the program segments, for what those program segments make processes do. This responsibility of the domain owner is shared, to some extent, when the domain owner depends on certifications performed by others. But in simple cases (i.e., when no certification specifiers appear in the seg-limit component of the acp of the domain, and the domain owner wrote or certified all the programs for which the domain contains a capability), a single social entity, the domain owner, is completely responsible for what the programs in a domain make processes do.

Another instructive mode of usage of a domain is for isolating a borrowed program. Working on the theory that the borrowed program might be a Trojan Horse(*), the borrower decides not to put a segment capability for the program in his home domain, and instead he creates a new domain to encapsulate the borrowed program. The borrower might be afraid that the borrowed program will spy on him by making calls and thereby passing information to some domain belonging to the program's author (or one of the author's friends). To give the borrower some control over this sort of behavior, the acp of a domain contains a component called call-out. The call-out component

(*) A Trojan Horse program is one which, in addition to doing whatever it is advertised to do, does something that its users don't know about and wouldn't want done.

is simply a list of tree names of domains that may be called by the domain with which the acp is associated. The call-out component of the acp of a domain is consulted by the operating system whenever it is about to add a domain entry capability to the domain, to insure that the domain which could be called through the domain entry capability being added is a domain to which calls are allowed.

The third interesting mode of usage of a domain is for sharing a data base and controlling access to it with a caretaker program. The programs that use the data base will all reside in domains that have domain entry capabilities for calling the domain that encapsulates the data base and its caretaker. The owner of the data base will be the owner of the domain that encapsulates it, and he will want to control access to his data base, not only through writing (or borrowing an appropriate) caretaker program, but also through being able to say which domains can call his domain. To meet this need, the acp of a domain contains a component called call-in. The call-in component is simply a list of tree names of domains that may call the domain with which the acp is associated. The call-in component of the acp of a domain D is consulted by the operating system whenever it is about to add to any domain a domain entry capability which allows calls to D, to insure that the domain obtaining the domain entry capability is allowed to call D.

The fourth important mode of usage of a domain is for debugging a newly written program. The author of the program might use a certified debugger in his home domain to create and control a domain encapsulating the newly written program. Debuggers require some unusual features to operate, such as the ability to interrupt processes executing the program being debugged, the ability to insert breakpoints in the program being debugged, etc. The design of a debugging protection environment is outside the scope of this thesis, but we can hypothesize that the acp of a domain will require a component called debug to authorize the operation of the debugger.

The fifth important mode of usage of a domain is for encapsulating a proprietary service. There are many important protection problems associated with providing proprietary services to users of a computer utility, and these are covered in the next chapter. We will need to refer to the user domain of a proprietary service; this domain is the domain for which the service is working. The user domain of a proprietary service has one or more domain entry capabilities which it uses to call the proprietary service. The owner of the user domain of a proprietary service needs to have some powers of control over the service his domain is using, and the control information is placed in a component of the access control packet of the user domain called the proprietary call-out component. This component is replicated in the access control

packet once for each proprietary service which the user domain is a user of, and its structure and meaning is described in detail in the next chapter.

4.8. Summary

We began this chapter with the three simple concepts--segment, process, and domain--which emerged from our review of elementary protection mechanisms in chapter 3. We defined our process to have a binding to a domain, rather than to contain domains, because we observed that processes and domains are logically distinct and roughly of equal importance. We noted that small domains (in terms of the number of program segments' authors) are required for adequate protection. To allow users of a computer utility to build on the work of others, we introduced the domain entry capability and two methods of passing arguments and results between domains: shared argument segments and the argument window of the sectioned stack. We presented the design of an operating system for our computer utility that protects itself with a collection of domains, supports the protection mechanisms defined previously, and multiplexes the memory and processor resources of the hardware computer. The details of this hardware and software design may be found in Appendices 1 through 4. Finally, we defined a naming hierarchy with an authorization mechanism, the system of access control packets, for controlling computing objects.

Chapter 5

Proprietary Services

5.1. Summary of Problems

Users of computers want to build on the work of others in the form of programs and data. Thus there is a demand for useful programs and data. A software industry has emerged and considerable effort is being expended to develop proprietary programs which can be rented out. The purpose of this chapter is to explore in detail the privacy and protection problems which must be solved in order to offer the services of proprietary programs and data to users of a computer utility. This problem was investigated in an abstract setting by Vanderbilt [Va69]. This chapter extends his results in the practical setting provided by the domain-supporting mechanisms of chapter 4.

We have investigated nine important problems which must be solved in any practical computer utility in order to offer proprietary services in a context that protects the interests of all the users, especially the lessor and lessee of the proprietary service. We have already shown the basic strategy for solving the problem: proprietary services are encapsulated in domains.

The mechanisms of chapter four are an adequate solution to the first three of the important problems we will present in this chapter. The first problem is to protect the integrity of a proprietary service encapsulated in a domain

by restricting access to the entry points of the domain. The entry points are the addresses in the program segments where processes calling into the domain are expected to begin executing instructions. The operation of the programs in the domain will not be reliable if calling processes can begin execution of programs at arbitrary points. The domain entry capability solves this problem by specifying the entry point address at which the calling process begins execution in the called domain. The operating system primitives which create domain entry capabilities will not allow any domain entry capabilities to specify an entry point address not authorized by the authority(ies) that control the called domain. Furthermore, access to the entry points which are valid entry points is restricted to domains having an appropriate domain entry capability.

When a process returns from a called domain to the calling domain, the same protection problem presents itself. The return point, i.e. the address in a program segment where the returning process should resume execution in the calling domain, must be protected from any modification after being established by the calling domain, at the time of the call. Our sectioned stack solves this problem by making available an inaccessible region of the stack segment in which to store the return address. The return-domain instruction can access this otherwise inaccessible region to retrieve the stored return address

and effect the return.

The second important protection problem is to pass arguments and results between the calling and called domains. Our solution to this problem is the sectioned stack for small arguments, and argument segments for large arguments. These mechanisms allow calling and called domains to pass arguments and results without compromising the secrecy or the integrity of any other information. Other mechanisms for solving these protection problems have been developed by Schroeder [Sc72b] for a process in which the domains are part of the process state and every segment in the address space of the process has the same segment number in every domain of the process. Schroeder's solution is to introduce hardware processor features to dynamically create capabilites for argument and result subsegments at every cross-domain call, and to dynamically destroy the capabilites created by such a call when the corresponding cross-domain return occurs.

The third important protection problem is to protect proprietary programs and methods from being stolen. The thief could either steal the program, or steal copies of its intermediate results and deduce therefrom its method of operation. The domain mechanism itself, and the information-erasing activity of the sectioned stack, provide a solution to this problem. The program segment is protected from being stolen by the lessees of the service the program imple-

ments because the program segment cannot be accessed directly by any lessee: a lessee's domain does not have a segment capability for the program; instead it has a domain entry capability to call another domain which does have such a segment capability. This level of protection for a program segment can also be achieved by the "e" mode of access defined in Appendix 1. If a domain has a capability for a program segment and the mode of the capability is just "e", other programs in that domain cannot steal the given program, because they cannot read its words as data. This protection is available even though the program is permitted to read constants out of itself.

But to protect a program's methods from being stolen, the program's intermediate results must be unavailable to the would-be thief. By placing its intermediate results only in the sectioned stack and in data segments which can be accessed only in the domain that encapsulates the proprietary service, the program protects itself from this threat. Intermediate results in the sectioned stack will be erased when the process returns to the calling domain, while the intermediate data segments can be accessed only

by the proprietary program and other programs in its domain (*).

The fourth important protection problem is to protect the secrecy of argument information passed to a proprietary program. That is, the user of a proprietary program might be concerned that the data he supplies to the program could be stolen. For example, suppose the proprietary program was written by company R and did circuit analyses; and suppose that the would-be user is an engineer working for company G. He wants to use the program to analyse a circuit but he won't use it if he thinks that his competitor, R, will learn his circuit design as a result. In other words, the problem is to allow the proprietary program to "know" the circuit, which is necessary for the program to do its job, but to prevent the program's owner from knowing the circuit.

(*) The program, or its data, might yet be stolen by means of the compiler-caller two-pronged attack. This attack method requires a conspiracy between, or identity of; a caller (user) of the proprietary service, and the author of the compiler which was used to compile the proprietary program. The compiler inserts extra instructions in the proprietary program, and the caller passes particular argument values which trigger the execution of the extra instructions. The extra instructions can provide the caller with copies of the proprietary program and any of its intermediate results. This method of attack can be prevented by auditing the compiler, or by examining its output with another program, a compiler output verifier, which will detect the insertion of unwanted instructions.

It would be possible to audit a proprietary program to insure that it does not contain spying code that reports to someone what it worked on. Since auditing is a human function, this is a very expensive solution. A more attractive possibility is to design an environment for the proprietary service that prevents it from doing such spying. Such an environment can be made available in a computer under the control of a third party (i.e., neither the lessor nor the lessee of the proprietary service). Then the integrity of the system's protected, constrained environments would be the responsibility of that third party; and the use of proprietary services would involve agreements with him. The logical choice to serve as this third party is the administration of the computer utility where the services are offered. The environment which prevents argument spying is composed of benign domains, and is described in section 5.2.

The fifth important protection problem is the possibility that a proprietary service will spy on its caller by hiding a few bits in its results. These few bits would be derived from the arguments passed by the caller, so the reader should regard this problem to be a special case of the argument spying problem just presented. Unfortunately, the benign domain mechanism does not solve this problem. Since we are forced to treat it separately, we give it a name: the hidden data flow problem.

Hidden data can replace the low-order bits of floating-point numerical results, or fill the space unused by a varying string result, without requiring any variation in the storage formats expected by the caller of the proprietary service. If those formats include areas in which blocks can be allocated and freed, the hidden data can occupy a hidden block. The proprietary service and its brother spy might need to use an error-detecting encoding to deal effectively with the hit-or-miss mechanics of getting the hidden data to the brother spy.

More obscure methods of hiding data can surely be found (*). Our treatment of the problem in this chapter is at the level of a game, with teams of spies and counter-spies, similar to the communities of users and breakers of crypts. The problem cannot be adequately solved without the privacy restriction mechanism introduced in chapter 6.

(*) Sometimes hidden data can be encoded in a major distortion of the results returned by the proprietary program, provided the brother spy has access to a mechanism that detects the distortion. For example, a list of stockholders of a certain corporation could be obtained by spies who program a proprietary service for preparing tax returns. The program would produce incorrect returns for all users who were stockholders of the given corporation. The IRS publishes lists of taxpayers who file incorrect returns; this is the detection mechanism followed by a broadcast. The spies obtain a partial list of stockholders by intersecting the lists published by IRS with a list of the users of their proprietary service.

The sixth important protection problem is the possibility that a proprietary service will spy on its caller by sending information through the information channel whose purpose is to allow the preparation of detailed invoices for services rendered. The problem is to establish the communication necessary for billing without allowing communication for spying. Like the hidden data problem, this problem can be regarded as a special case of the argument spying problem. One solution that eliminates spying is to eliminate billing and charge a fixed monthly rental. Other methods, which throttle the rate at which spied information is communicated through the billing mechanism, are described in section 5.4.

The seventh important protection problem is a conflict that develops between the owner (maintainer) of a proprietary service, and its users. The owner wants to fix bugs that are found in the service, and perhaps also upgrade the level of service, which sometimes involves substantial modification. The users, on the other hand, don't like to see the service change at all, unless it is in response to a problem they're having with it. So for any given change, many users are against it.

A typical owner response to this problem is to make new releases of his service available to his users, and to refuse to maintain any but the most recently released version. Thus, the owner conserves the resources expended for maintenance,

while the users are forced to update to the latest release when they encounter problems with older versions.

Now suppose the service is offered through a computer system under the control of a third party. The users of an old version of the service might want that version to remain unchanged. Other users will switch to the latest version as soon as it is available, because they are willing to adapt to the change (they have the resources to expend to do it). But if the owner of the service were to remove the old version, he might arouse the ire of his old-version users. Thus the users need an agreement with the owner, enforced by the computer system, that no proprietary service in use may be modified. (This would not prevent a new version of the service from being offered.) The enforcement would create the needed trust between user and owner, because the user would know that the proprietary service could not be modified. An operating system mechanism to record and enforce such agreements is described in section 5.5.

The eighth important protection problem is the possibility that a proprietary service will stop working, or begin to give out bad answers, at a time chosen by some clever enemy of the user of the service. For example, a failure could be timed to coincide with a demonstration to some would-be client of the user. As with the argument spying problem, two methods

of solution seem feasible; audit the proprietary program, or place it in an environment that prevents it from exhibiting the threatening behavior described above. Such an environment would have to deny the proprietary service knowledge of who it is working for, and the time of day; in fact, the proprietary service would have to be denied all sources of input that might be influenced by the enemy, because such inputs could contain encoded signals to the service. We will call a service in such an environment a blind service. Blind services are defined and discussed in section 5.9.

Finally, the ninth important protection problem is the desire of the competitive owner of a proprietary service to withhold the use of his service from his competitors. Competition is a way of life in America, and because the development of software is best accomplished by small groups of workers, the software marketplace might be served reasonably well by a large number of small, independent, competitive suppliers.

The problem of denying one's service to one's competition is compounded when one service is part of another. To be precise, suppose owner A builds a service S, and suppose another owner, B, builds a service T that uses (calls) S. If A does not want his competitor A' to use S, he must make an agreement with B in which B promises A not to sell the services of T to any user that A doesn't approve of. Now A

must expend time and energy approving of applicants who want to use T, in addition to screening applicants who want to use S. Competition has its inefficiencies. The privacy restriction mechanism of chapter 6 can eliminate some of these inefficiencies. We return to this problem in section 6.11.

5.2. Argument Spying and Benign Domains

The purpose of this section is to present the definition of a constrained environment in which a proprietary service will find it extremely difficult to do argument spying. This constrained environment begins with a domain containing a proprietary service which can be called from its user's domain. Whenever a user's process calls into the domain of the proprietary service, the program which implements the service has an opportunity to copy information, to compute, to call other domains (including the operating system); in short, to struggle by these means to communicate the arguments, or information derived from them, to a spy hiding someplace in the computer utility.

The proprietary service could write the arguments into a segment shared with a domain containing a spy process, for example. Therefore the constrained environment may not contain any shared writable segments, nor any writable segments which are ever readable by any other domains. The constrained environment must have some writable segments to allow it to remember things for its caller, but these segments must not

be useful for spying. Thus, all the segments which are writable from domains of the constrained environment must be under the control of an authority dedicated to the prevention of spying. This authority will allow each writable segment to be accessible in one, and only one, of the domains of the constrained environment. Having made the above strong statement, it is necessary immediately to moderate it in two important cases: first, if the segment serves as an argument segment for two or more domains of the constrained environment, and second, if all the parties having bona fide rights of control over information in the segment agree to its release. The authority which enforces these rules will be called the Proprietary Services Administration (PSA), and the writable segments of the constrained environment will be called closeted segments.

Another method by which the proprietary service might try to communicate the arguments is by calling into a domain that works for the spy. To prevent the would-be spies from gaining any advantage from this ploy, we require domains which are called from domains of the constrained environment to be themselves members of the constrained environment.

Domains of the constrained environment may also call operating system domains, but not at every entry point. For example, the primitive which writes entry names into the naming hierarchy will not be accessible by domains in the

constrained environment. The operating system will carry the responsibility of keeping a domain in its constrained environment once it has been placed there, and the responsibility of preventing any domain of the constrained environment from writing information anywhere other than in closeted segments. Domains of the constrained environment will not be allowed to do input or output.

Another method by which the proprietary service could communicate information to a spy is through a process that calls into the domain of the proprietary service from some domain that works for the spy. The author of the proprietary service could make a special entry point for this purpose, unknown to the user of the service. To prevent this method of spying, we require that every domain of the constrained environment be callable only from other domains of the constrained environment, and from the domain of the user of the proprietary service.

This completes the definition of the constrained environment. To see this, reflect on the fact that there are only a few ways for information to get in and out of domains (*):

(*) Information is in a domain if it can be read as data by a process bound to the domain. Information gets out of a domain by getting into some other domain, or by leaving the computer system through an output device.

through segments, through processes calling and returning (calls to the operating system being a special case), and through input/output.

To summarize, a constrained environment that works for a user domain D is a set C of domains such that (1) all the writable segments of domains in C are closeted segments, (2) domains in C can be called only from other domains in C, and from domain D, (3) domains in C can invoke only limited services of the operating system, as described above, and (4) domains in C can call only other domains in C, and the operating system. If we assume for the moment that there is no flow of billing information out of the domains of C, and if we postulate also that the operating system refuses to allow any information to be written in user-accessible places by domains of C, it follows that all the information that leaves domains of C must enter domains of C, or domain D. The information that enters domain D is the result of the computation of the proprietary service in the constrained environment, plus any hidden data placed in the results by the proprietary service.

We will call the domains of the constrained environment benign domains, because they do not have the power to do argument spying. We can prove rigorously that they do not have such power by considering the path through the benign domains taken by a process calling from the user domain. An example

will clarify the construction on which the proof depends. Figure 5-1 shows a constrained environment and the path of a process through the benign domains, represented as a tree of domain invocations. Each domain invocation is the process entering a domain. The arcs between the domain invocations represent calls and sebsequent returns by the process. When a domain invocation calls more than one domain before returning to its caller, more than one arc will emanate from that domain invocation in the tree. Each domain invocation in the tree of figure 5-1 is labelled with the name of the domain invoked, and a number i which means that the invocation is the $i^{\underline{th}}$ invocation of the named domain by the process.

Our proof proceeds by induction on the tree of domain invocations. For the basis step, let x be a domain invocation (X,i) at the bottom of the tree. The only ways for information to leave the domain X at invocation x are through a segment, through a call to the operating system, and through a return to the caller of X. (Since (X,i) is at the bottom of the tree of domain invocations, the process will not call any domains other than the operating system.) We have already postulated that calls to the operating system will not result in releases of information when the calls originate in benign domains. Furthermore, information cannot leave domain X through a segment because all the writable segments of X are closeted. So we need be concerned only about infor-

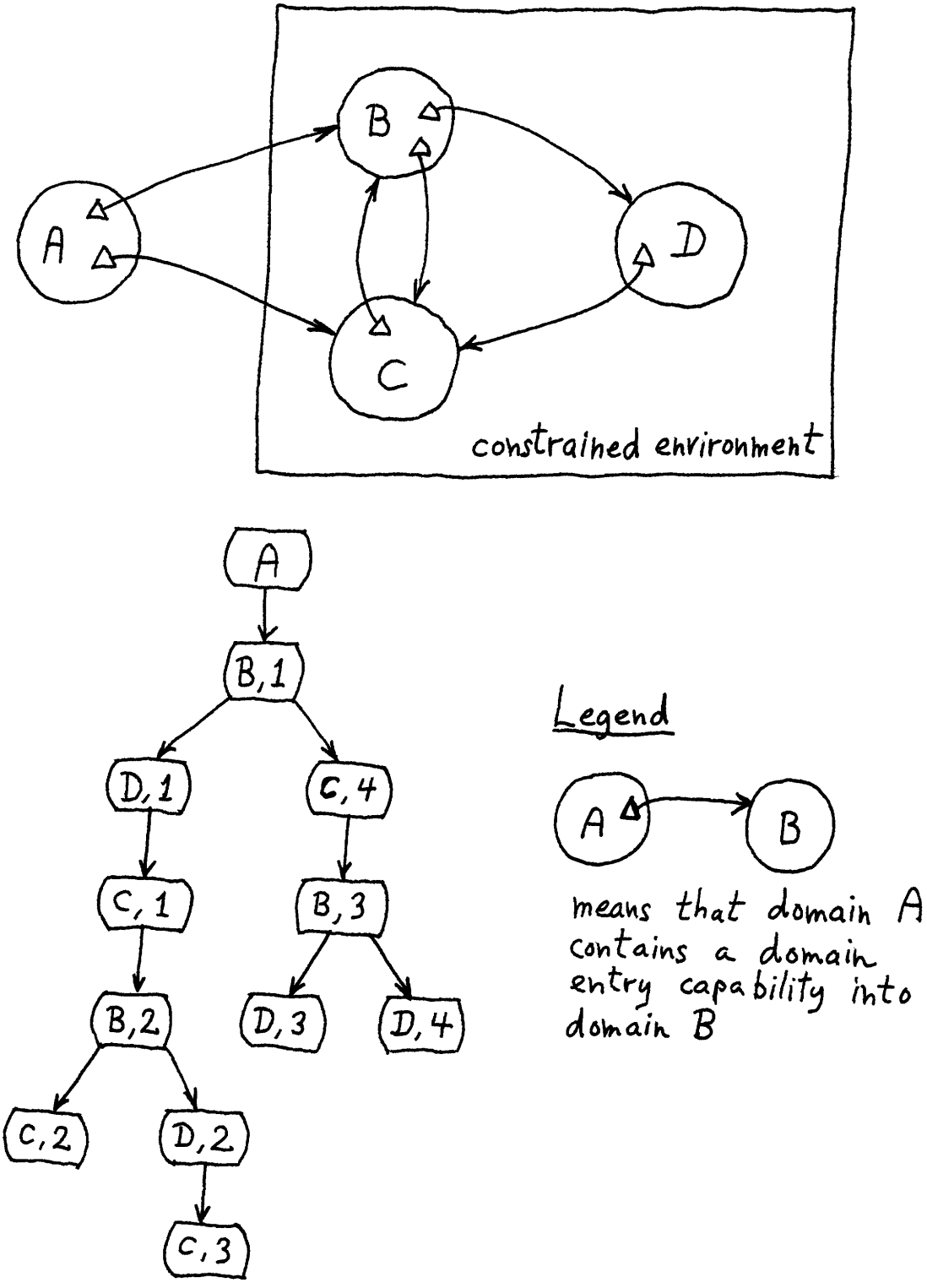


Figure 5-1. A constrained environment, and one possible path of a process calling from domain A.

mation leaving domain X through the process returning. The process will return to the user domain of the constrained environment, or to one of the domains of the constrained environment. If the return is to the user domain of the constrained environment, we need not be concerned about argument spying because the user domain had access to the arguments in the first place. If the return is to some domain of the constrained environment, the argument information which we are concerned about remains in the constrained environment.

For the induction step, let x be a domain invocation (X,i) not at the bottom of the tree. The only ways for information to leave the domain X at invocation x are through a segment, through a call to the operating system, through a call to another domain of the constrained environment, and through a return to the caller of X. By induction hypothesis, calls to other domains of the constrained environment will not allow information to leave the constrained environment. Since X is a benign domain, calls to the operating system will not result in releases of information, and all the writable segments of X are closeted. So once again, we need be concerned only about information leaving domain X through the process returning. If the return is to the user domain of the constrained environment, argument spying is not possible because the user domain has access to the arguments already. If the return is to a domain of the constrained environment, no information leaves the constrained

environment.

So, by the principle of mathematical induction, the only information that can leave a constrained environment must flow into the user domain of the constrained environment.

It is important to note that this result depends crucially on the postulated properties of the operating system. Those parts of the operating system which act to constrain the operation of benign domains and sequester closeted segments must be audited.

5.3. The Proprietary Services Administration

The Proprietary Services Administration (PSA) is a part of the operating system of the computer which creates and controls domains encapsulating proprietary services, in response to requests coming from domains, or owners of domains, that want to use the services. PSA is responsible for establishing constrained environments, benign domains, and closeted segments. When a service is implemented by using other, previously established services, PSA will create domains for each of the component services.

The domains created by PSA are given names chosen by PSA, in a part of the naming hierarchy which is under the control of PSA. The names chosen by PSA are not predictable because of races for slots of PSA's name space between concurrent users of PSA, and this creates a problem for the owner of a domain who wants to authorize his domain to call a domain

which will be created by PSA. He cannot know the name of the called domain in advance, but he does know the name of the service which PSA will install in that domain once it is created. For this reason, the owner of the domain which will use (call) a proprietary service cannot employ the call-out component of the acp of his domain to specify the domain to be called. Instead, he will use the proprietary call-out component, in which he specifies the name of the service to be placed in the domain created by PSA.

When a user names a service whose implementation depends on other services, PSA must take this name and effectively obtain from it the names of the component services. These details concerning the structure of a service should be provided by the implementor of the service, and for this purpose we introduce the service declaration segment. In this segment, the lessor (implementor) of the service names the services which are components of the service being declared, by giving the tree names of their service declarations. Similarly, the proprietary call-out component of the acp of the user domain of a proprietary service implicitly names the domain which encapsulates the service by specifying the tree name of the service declaration.

When a domain owner authorizes his domain to call a proprietary service, he might want to demand that the service occupy a constrained environment. This demand can be regarded

as a conditional authorization, of the form "if the service is benign, then let my domain call the domain which encapsulates the service." The demand can also be regarded simply as a declaration. The proprietary call-out component of the access control packet of the user domain of a proprietary service is the place where such demands are specified, and PSA will honor the demand, if it is there, and construct a constrained environment for the service with which the demand is associated. Similarly, a service declaration can contain a demand that a component service occupy a constrained environment, because the owner (implementor) of the service might want the algorithm of his service protected from any possibility of theft through observation of the information it passes to the component service through processes calling the component service.

The access control packet of a service declaration segment is used to store authorizations which allow domains to use the declared service. The acp of a service declaration segment has three components: a normal component, a user component, and an outer-service component. The normal component of an acp of a service declaration segment S has the same form as the acp of an ordinary segment, and it is used for the same purpose: to specify the modes of access

(either "r" or "rw") which domains named by the terms of the normal component may have to segment S. The other components of an acp of a service declaration segment S relate to the use of the service declared by S.

The user component of an acp of a service declaration segment S consists of a list of terms, and each term must be the tree name of a domain which is allowed to use the service declared by S. PSA will not create a domain encapsulating the service declared by S for any would-be user domain unless the user domain is named by a term of the user component of the acp of the service declaration S. The outer-service component of an acp of a service declaration segment S consists of a list of terms, and each term must be the tree name of a service declaration segment S'. The meaning of each term is that the service declared by S is allowed to be used as a component service of the service declared by S'. PSA will not create a domain encapsulating the service declared by S as a component of another service unless that other service's declaration is named by a term of the outer-service component of the acp of the service declaration S.

In the next two sections, we discuss solutions to the protection problems associated with billing information and mutually agreed maintenance; and the impact of these solutions on the design of PSA. Then, in section 5.6, we present a detailed example of the operation of PSA.

5.4. Billing Information

If a communication channel whose purpose is the preparation of invoices is available to a proprietary service, the channel can be used to spy on arguments. The spying would not go unnoticed by the lessee of the service, since he would vigilantly observe the additional information (probably encoded) on the face of his bill. This notification depends on the lessee seeing all the information which the proprietary service sends into the invoice channel -- if the lessor has the chance to edit out the encoded message of his spy program before the lessee sees the bill, the lessee is kept in the dark. The Proprietary Services Administration therefore provides the service of accepting invoice information from proprietary services, preparing bills from the information, and sending one copy of each bill to both lessor and lessee. We will call this method of billing the open account channel. It is characterized by the reproduction in the bill of long character strings emitted by the proprietary service, giving the service a high rate of information transfer to its brother spy, if it should ever start to use it.

A throttled account channel can be made available to the proprietary service in the form of an operating system primitive which accepts an integer account item code, which is an integer between 1 and N, where N is agreed to by lessor and lessee. Thus the expressive power of the account channel

is limited, but certainly not eliminated; information can be encoded in the sequence of item codes emitted. The item codes are used by the operating system in preparing bills, with fixed charges being associated with each item code. Charges are fixed for each item code in order to reduce the expressive power of the proprietary service generating the bill. As before, one copy of each bill is sent to both lessor and lessee.

It is clear that to eliminate completely the passage of spied data through the accounting channel, it is necessary to eliminate the accounting channel. In its place, two accounting methods are viable options. The first, and the simplest, is to have no accounting information at all collected by the computer utility. Instead, the lessor could charge the lessee a flat monthly fee. If this option is chosen, the lessee can be sure that the lessor is not getting any information through the billing mechanism. However, this mechanism will not notify the lessor if the lessee should begin to rent out the service to other users. If the service is designed to compute a result based on its inputs without remembering or using any information from previous calls, then the lessee could rent out the service to other users without sharing any of his own information. So the lessor of such a memoryless computational service could not accept a flat monthly fee.

The second alternative to having an accounting channel is to let the lessor charge a fee based on the number of processes per month which enter (by calling) the domain that encapsulates the proprietary service. This information can be collected by installing a counter and a control bit in the blocks of the ADT defined in figure A2-5, and adding to the logic of the state transition rule instructions to increment the counter, when the control bit is on, for every process that calls into the domain. Once a month the operating system would reset this counter, and report to the lessor and lessee the value it had reached when it was reset. Having this information, the lessor would be reassured that the lessee was not selling the service behind his back. But the lessee might be concerned that the lessor has this detailed information concerning this aspect of the lessee's activities, i.e. the number of times the service was called. While this concern may seem far-fetched, it is possible that the lessor can draw some intelligent inferences from the number of processes that called into the domain of the service. To make this drawing of inferences more difficult, the reported number of entrances could be made approximate -- e.g. the system could report a number of dozens, or scores, or hundreds. So if the system reported "3 dozen" the actual number could be anything between 36 and 47.

The lessor and lessee of a proprietary service must agree to use one of the billing mechanisms described here before the service is put into operation. The lessor of a service records in the service declaration of his service the type or types of bills which his service is prepared to generate, and the lessee of a service records in the proprietary call-out component of the acp of the user domain of the service the type or types of bills which he is willing to let the service generate. PSA will give the domain encapsulating the proprietary service, domain entry capabilities for PSA primitives which implement the most expressive billing method which the lessor and lessee have agreed to. Also, if the service declaration specifies more than one acceptable billing method, PSA will give the domain encapsulating the service a domain entry capability for a PSA primitive which will return to the service the type of billing information it should generate.

In order of their expressiveness, the billing methods available through PSA are

(1) the open accounting channel

(2) the throttled accounting channel

(3) fees based on the number of calls per month

(4) fees based on the approximate number of calls per month

(5) flat monthly fees.

Only the first two of these require the domain encapsulating the proprietary service to have domain entry capabilities for sending accounting information to PSA.

If the lessor and lessee specify billing through a throttled accounting channel, the number of account item codes which will be used must be specified in the service declaration and in the acp of the user domain, and these numbers must agree; and the dollar amount associated with each item code must be specified by the service declaration.

If the lessor and lessee specify billing through a fee based on the approximate number of calls per month, the divisor which accomplishes the approximating effect must be specified in the service declaration and in the acp of the user domain, and these numbers must agree.

When a service is implemented through calls to other services, all the domains encapsulating the services will be generating bills which must be paid by the owner of the user domain of the overall service. This multiplicity of sources of billing information does not present any great problem: the lessee of the overall service and the lessors of all the component services must agree on billing methods by means of the strategies presented above. The service declaration of each component service is compared with the proprietary call-out component of the acp of the user domain of the overall

service, and a billing method is selected for each component service.

But when a component service is forced to occupy a constrained environment by a demand in the service declaration of some other service working for the user domain of the overall service, the service declaration that demanded the constrained environment may also limit the types of billing methods available to the constrained service. For example, figure 5-2 shows four domains linked together by domain entry capabilities. (Program segments are not shown.) Domains C and D occupy a constrained environment because the service declaration of service 1 demanded it. The lessor of service 1 is afraid that argument information passed from domain B (which encapsulates service 1) to domain C will be passed on to a spy. But the constrained environment will not prevent billing information from flowing out of domain C to the owner of domain A, who might be the spy. Therefore the lessor of service 1 becomes a third party to the billing information agreement between the lessor of service 2 and the owner of the user domain, and similarly for the agreement between the lessor of service 3 and the owner of the user domain. In the service declaration of service 1, the lessor of service 1 records the type or types of bills which he is willing to let the domains in the constrained environment

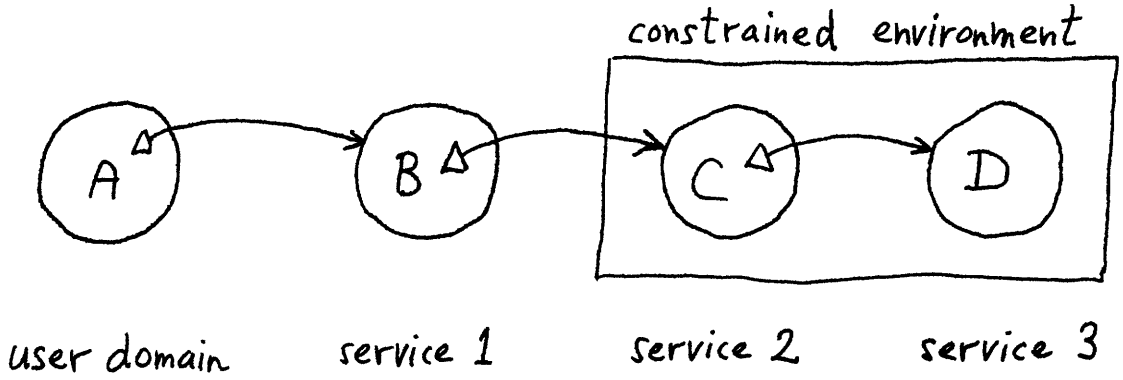


Figure 5-2. A constrained environment demanded by the service declaration of service 1.

generate, and PSA will enforce these restrictions. In addition, the lessor of service 1 may demand that he receive copies of the bills generated by domains C and D, so that he can inspect them and detect suspicious flows of information.

Finally, we must consider accounting and billing techniques for processor time expended and storage space utilized by proprietary services. Since a hardware mechanism is used to move processes between domains on inter-domain call and return, it is not easy to know precisely when a process enters and leaves each domain it is bound to. Therefore bills for processor time expended should go the user the process is working for. On the other hand, it is possible to measure the storage utilization of a proprietary service (e.g. in page-days of secondary storage). The owner of the user domain of the service should pay these costs.

5.5. Mutually Agreed Maintenance

The mutually agreed maintenance problem has two parts. First, the user of a proprietary service wants to be sure that the service is not going to change when he doesn't want it to change. Simple operating system mechanisms accomplish this form of protection in a straightforward way; they will be described presently. The second part is that the user of

a proprietary service wants the service to be fixed quickly whenever a lurking bug appears, but this desire conflicts with the user's desire to keep his information protected; and in addition the process of finding and fixing a bug threatens the privacy of information belonging to others. This second part of the problem cannot be resolved by operating system mechanisms.

To implement the idea of a service not changing, the operating system requires a feature for keeping a segment from changing. We will call an unchangeable segment frozen. A frozen segment may not be written once it has been made frozen; and it can never by unfrozen, once frozen. The access control packet of a segment must contain a bit which indicates that the segment is frozen, and if the bit is on the packet may not contain terms giving "w" access to any domain. To freeze a service, all that is required is to freeze all the program segments of the service and all the data segments of the service which are provided by the lessor of the service. Data provided by the user of the service may be stored in writable segments in the domain that encapsulates the service and allowed to affect the operation of the service, but this will not violate the frozenness of the service. In other words, a service is frozen when the lessor cannot affect what the service does.

The owner of the user domain of a service can demand that the service be frozen with a variable in the proprietary call-out component of the acp of the user domain. The lessor of the service can assert that his service is frozen in his service declaration, and if the assertion is made the lessor must provide a list of all the segments which implement the service. When PSA constructs a domain to encapsulate the service, it will honor the demand of the user domain owner by initializing the seg-limit component of the acp of the domain encapsulating the service with the list of segments provided in the service declaration, and by checking to see that each of these segments is frozen, and by requiring that the seg-limit component may be expanded only by the addition of closeted segments. These constraints on the seg-limit component of the domain encapsulating the service insure that the lessor can do nothing to affect the service after he has frozen all the segments which he listed in the service declaration. Furthermore, the lessor is required to freeze the service declaration, and PSA will check to see that it is frozen when it constructs domains encapsulating the service.

When a service is implemented through calls to other services, and the owner of the user domain of the overall service demands that the service be frozen, then all the component services of the overall service must be frozen.

PSA will propagate this requirement from the proprietary call-out component of the acp of the user domain, onto all the component services of the overall service.

When a bug is discovered or suspected in a proprietary service, a knowledgeable programmer must test or debug the service. Since bugs are sometimes data-dependent, it is necessary for the debugging programmer to have access to data which exercises the bug, i.e., causes the bug to appear. Since having access to such data might tend to violate somebody's privacy, appropriate permissions must be secured before debugging can take place. For example, figure 5-3 shows a proprietary service composed of two component services and two constrained environments. The outer constrained environment was requested by the owner of the user domain, while the inner constrained environment was requested by the lessor of service 1, the service encapsulated in domain B. If the owner of the user domain suspects that there is a bug in the service, he will produce documented evidence of the bug and take it to the lessor of service 1 and complain; and give the lessor of service 1 permission to investigate the workings of domain B. This permission must be given by communicating it to PSA, since PSA has control of domain B. Now suppose that the debugging programmer employed by the lessor of service 1 decides that the problem is in domain C. He will have

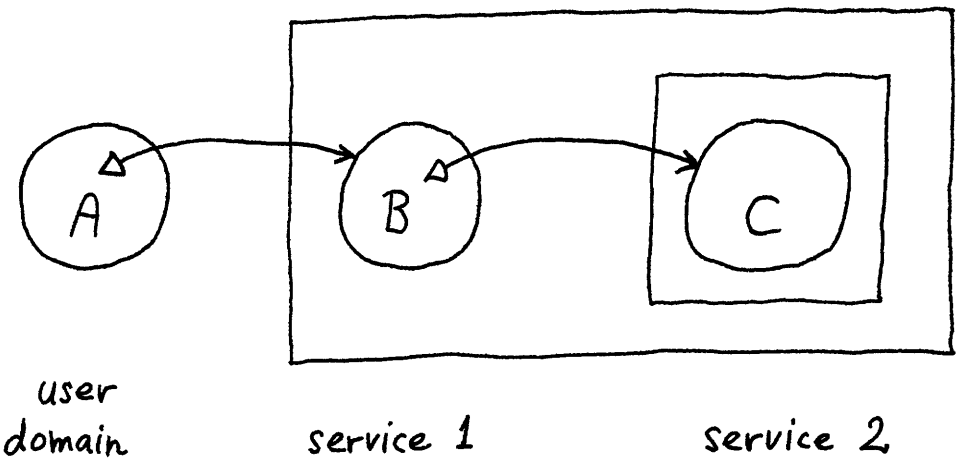


Figure 5-3. A service with a bug.

to produce documented evidence for the lessor of service 2, and both the owner of the user domain and the lessor of service 1 will have to give the lessor of service 2 permission to investigate domain C. Again, these permissions must be communicated to PSA, since PSA has control of domain C. PSA requires two permissions before allowing a debugging programmer to investigate domain C because domain C is an occupant of two constrained environments.

This example illustrates a general rule: if a domain occupies n constrained environments, then permissions from the n social entities whose oxes are being protected by the constrained environments, must be communicated to PSA in order to allow the operation of the domain to be investigated.

Once a bug has been found, the domain whose investigation led to an understanding of the bug can be fixed immediately. The fix is incorporated in a corrected program segment, and a new service declaration must be prepared which names the corrected segment in place of the one with the bug. Both the new program and the new service declaration must be frozen, if this is required by users of the service. Then PSA is requested to update the domain that was investigated, by replacing the program with the bug with the new program segment specified by the new service declaration.

Other domains encapsulating the same service can be updated to incorporate the change, when this is requested

by the owner of the user domain of the service. When this updating requires a modification of the format of data segments, the lessor of the service will prepare a program to accomplish re-formating of the data segments. PSA will run this program, in the domain encapsulating the service, when the lessee of the service requests that his incarnation of the service be updated.

The lessor of a service makes an updated version of his service available to users by creating a new service declaration. Since different versions of the service are implemented with (slightly) different programs, the service declaration must name the program segments which implement the service. This part of the service declaration was introduced in the context of the requirement that program segments be frozen, but in fact all service declarations must name all the program segments that implement the service, so that PSA will know which segments to create capabilities for in domains created by PSA which encapsulate the service. If a new version of a service differs from the old version because of re-formated data bases, the service declaration of the new version must name the program which will accomplish re-formating.

5.6. An Example of the Operation of PSA

In this section, we present an example to show the data

structure created by PSA when PSA constructs constrained environments. Figure 5-4 shows a constrained environment constructed around the domain B, which encapsulates a service constructed from component services encapsulated in domains C_1, D, and C_2; all working for the user domain, A. (The domains C_1 and C_2 each encapsulate the same service, but they operate on different data.) Figures 5-5, 5-6, and 5-7 show the service declarations for the servies encapsulated in domain B, domains C_1 and C_2, and domain D, respectively. In any real computer utility, the service declarations would be expressed in some computer language; but we are expressing them in English because our goal is to communicate to the reader. (The details of a computer language to express service declarations would obscure the point of this example.) Figure 5-8 shows that part of the naming hierarchy which is used to catalogue the service declarations and other segments which define the three services in our example. Our figure 5-5 represents the contents of segment (user, factory1, product1, sd.v1); figure 5-6 represents the contents of (users, factory1, product2, sd.v4); and figure 5-7 represents the contents of the segment (users, factory2, product1, sd.v3). Figure 5-9 shows the lessee's sector of the naming hierarchy, including the user domain A of the service in our example.

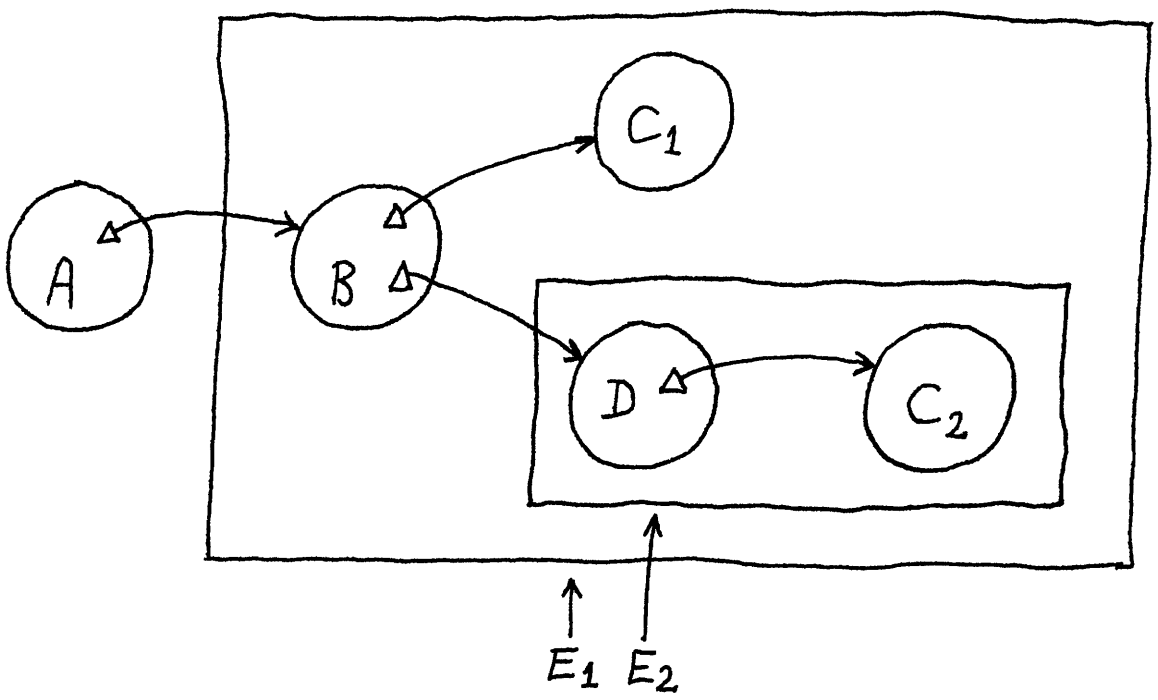


Figure 5-4. A proprietary service working for domain A.

"The lessor of this service is Benign Enterprises, Inc. The lessor's mailbox is at (users, factory1, mailbox).

The short name of this service is "service1".

The segments which implement this service, with required access modes, are: (users, factory1, product1, prog.v1), mode "e"; (users, factory1, product1, data.v1), mode "r".

This service is frozen.

The lessor will accept billing on an open accounting channel, a throttled accounting channel (with 5 symbols having the following meanings: $2.00, $10.00, $1.00, $3.00, $7.00), or a fee based on the number of calls per month, or the approximate number of calls per month (measured in 20's).

The component service referenced by the name "service2" is declared by segment (users, factory1, product2, sd.v4).

The component service referenced by the name "sneaky-pete" is declared by the segment (users, factory2, product1, sd.v3). This component service must occupy a constrained environment. Bills generated in this constrained environment must be based on a throttled accounting channel (with 4 symbols), a fee based on the number of calls per month, or the approximate number of calls per month (measured in 20's); and the lessor of this service must receive copies of bills generated in this component's constrained environment."

Figure 5-5. Service declaration of service encapsulated in domain B.

"The lessor of this service is Benign Enterprises, Inc. The lessor's mailbox is at (users, factory1, mailbox).

The short name of this service is "service2".

The segment which implements this service, with required access mode, is: (users, factory1, product2, prog.v4), mode "e".

This service is frozen.

The lessor will accept billing based on an open accounting channel, a throttled accounting channel (with 3 symbols having the following meanings: $1.00, $2.00, $4.00), or a fee based on the number of calls per month, or the approximate number of calls per month (measured in 20's)."

Figure 5-6. Service declaration of service encapsulated in domains C_1 and C_2.

"The lessor of this service is Sharptooth Enterprises, Inc. The lessor's mailbox is at (users, factory2, mailbox).

The short name of this service is "confidence".

The segment which implements this service, with required access mode, is: (users, factory2, productl, prog.v3), mode "e".

This service is frozen.

The lessor will accept billing based on an open accounting channel, a throttled accounting channel (with 4 symbols having the following meanings: $7.00, $8.00, $6.00, $2.00), or a fee based on the number of calls per month, or the approximate number of calls per month (measured in 20's).

The component service referenced by the name "sawhorse" is declared by the segment (users, factoryl, product2, sd.v4)."

Figure 5-7. Service declaration of service encapsulated in domain D.

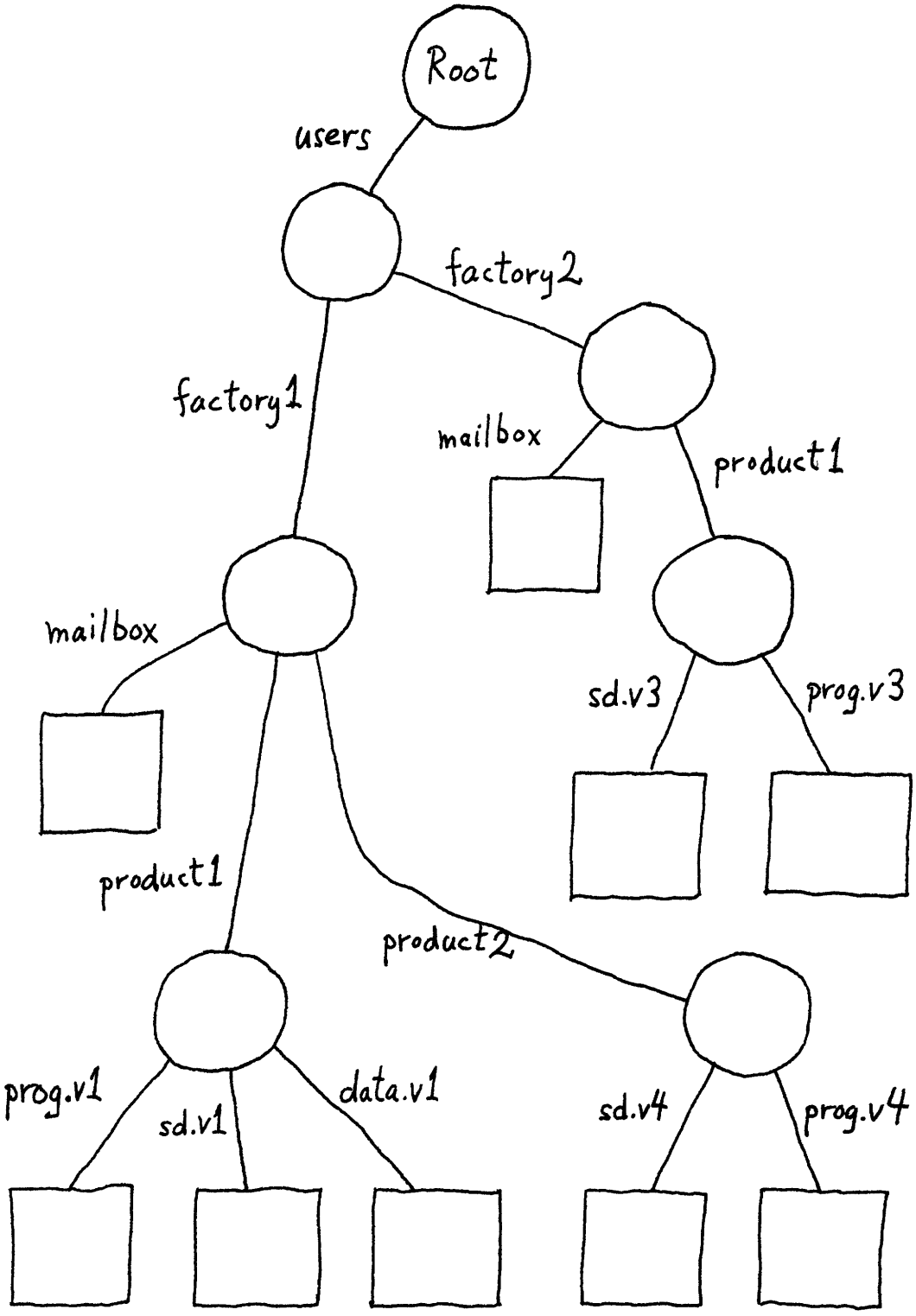


Figure 5-8. Lessors' sector of the naming hierarchy.

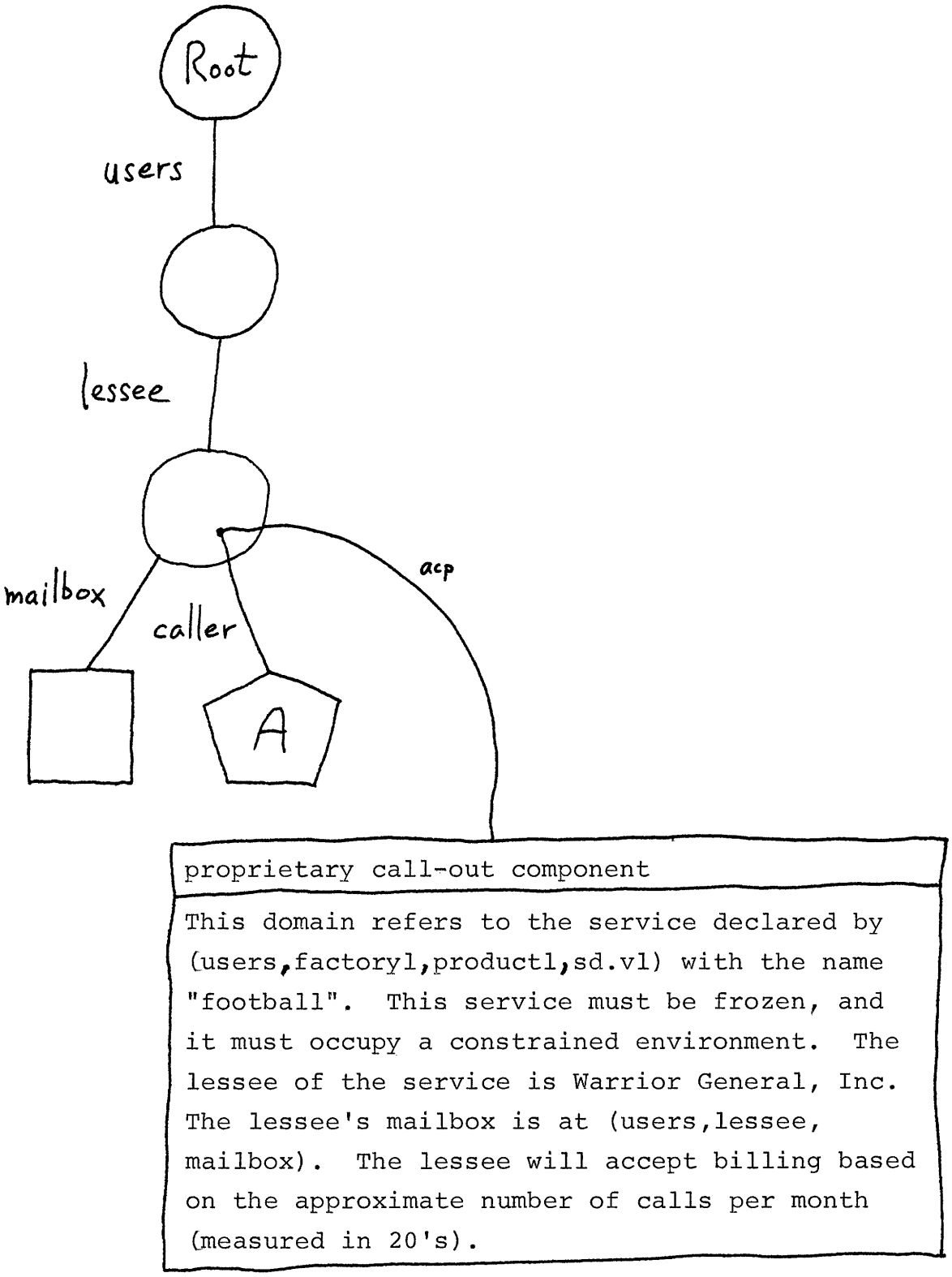


Figure 5-9. The lessee's sector of the naming hierarchy, and the acp of the user domain A.

From the acp of the user domain, we can see that the billing method which will actually be used by the service in our example will be a fee based on the approximate number of calls per month, because this is the only method to which the lessee will agree. The mailboxes which are shown in figures 5-8 and 5-9 serve to receive bills generated by PSA. The mailboxes are named in the service declarations and in the acp of the user domain so that PSA will know where they are.

Figure 5-10 shows the structure in the naming hierarchy which PSA creates to organize the benign domains and closeted segments of the constrained environment shown in figure 5-4. This structure provides a unique directory for each of the domains B, C_1, D, and C_2; and these directories provide a place to catalogue the data segments, especially closeted segments, used by the domains B, C_1, D, and C_2. A unique directory is provided for each domain of the protected environment in order to avoid name clashes between segment names. For example, if both the domains C_1 and C_2 create a closeted segment named "own-data", no name clash results because one of these segments will be entered in the directory (PSA-data, 473, service2) and the other in (PSA-data, 473, servicel, sneaky-pete, sawhorse). In both directories, the name of the segment entry will be "own-data".

All the directories in the naming hierarchy below the

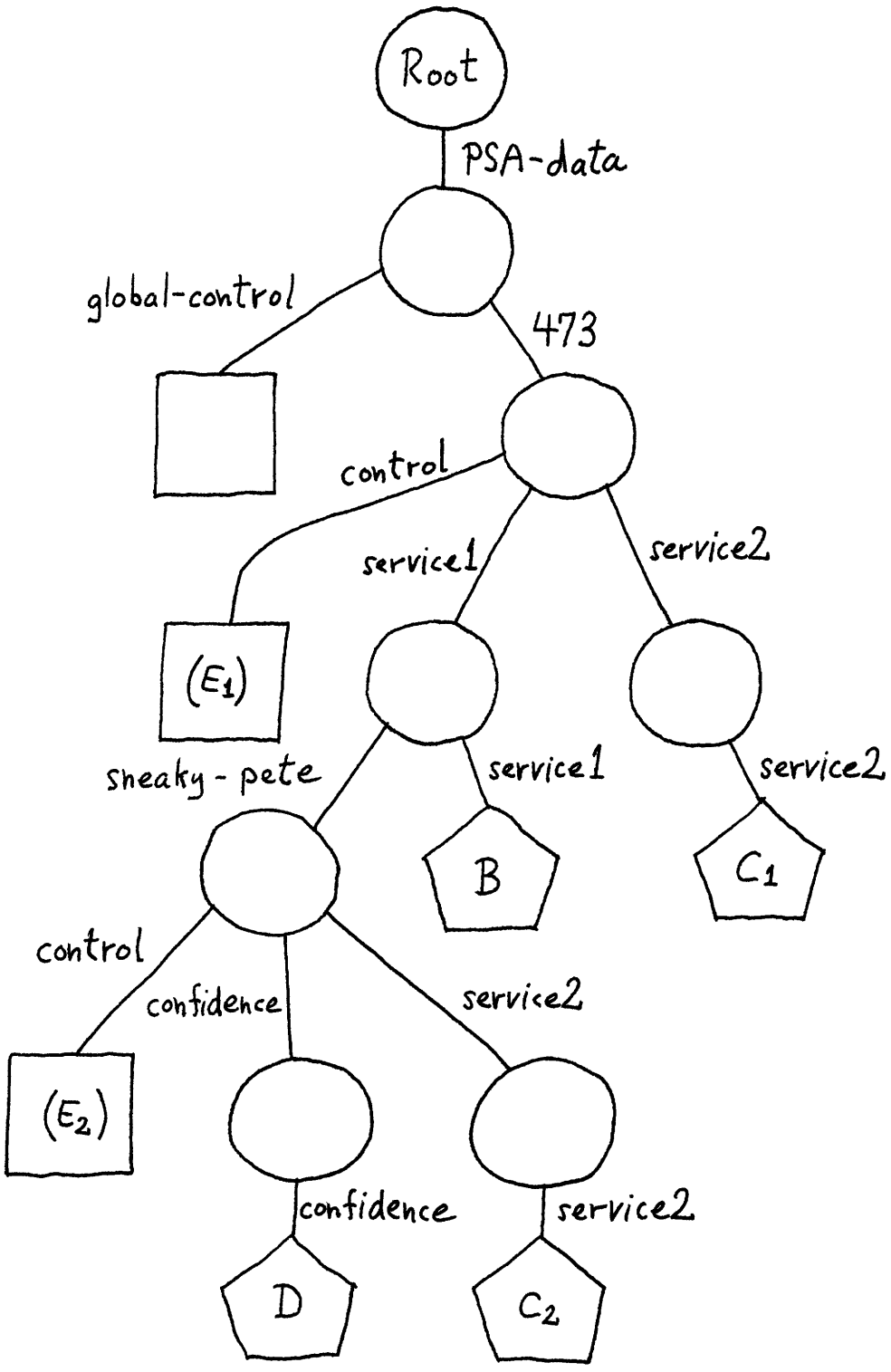


Figure 5-10. The naming hierarchy structure created by PSA to implement the service of figure 5-4.

directory (PSA-data) are under the control of PSA. This is because all these directories were created by PSA originally, and PSA keeps control over everything it creates. Since control over closeted segments and benign domains depends on having control over the directories which these segments and domains are entered in, control over these directories is crucial. Most crucial of all is the question of control over the directory (PSA-data). Assuming that PSA is implemented with a program encapsulated in the domain (system, PSA), figure 5-11 shows how that domain is given exclusive access to (PSA-data).

The name space of entry names in the directory (PSA-data) is the name space for whose slots concurrent users of PSA will race. These races are resolved on the basis of a lock implemented in the segment (PSA-data, global-control). It is in this segment that PSA remembers the use of each entry in the directory (PSA-data). For example, PSA must remember that the entry named "473" was created to structure a service working for the domain (users, lessee, caller), and that the user domain refers to the service with the name "football".

The directory (PSA-data, 473) is used to organize those domains in the outer constrained environment E_1 which are not part of any contained constrained environment. The segment (PSA-data, 473, control) is PSA's scratchpad

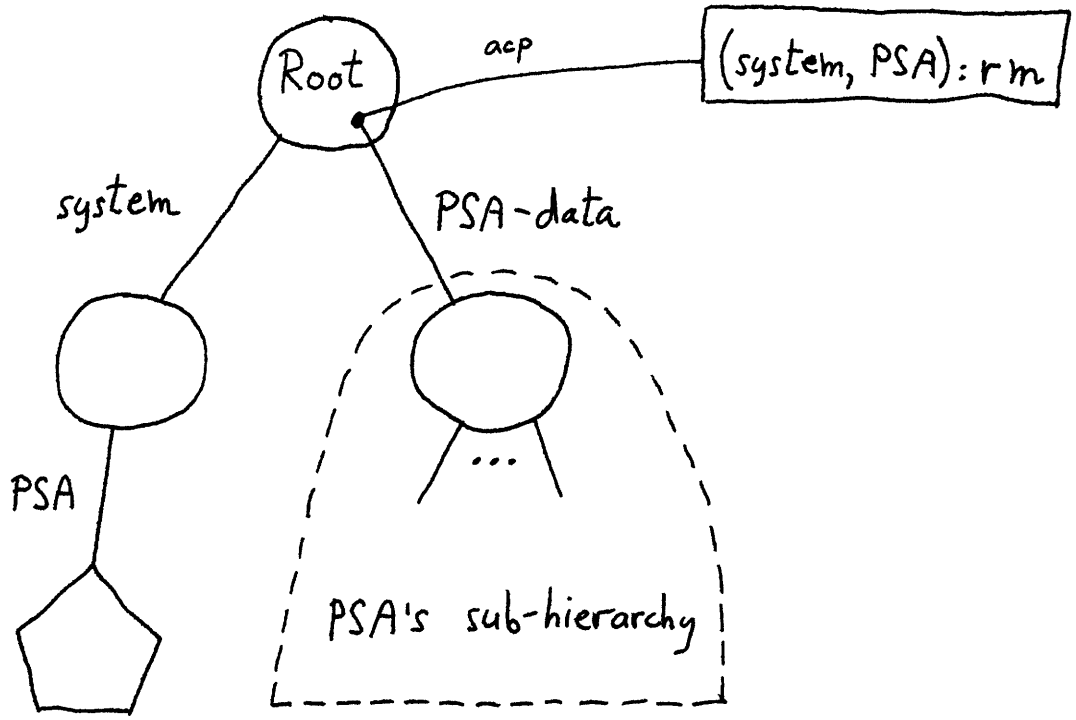


Figure 5-11. The acp on the directory (PSA-data).

for information about E_1, such as the tree names of the service declarations of services encapsulated in B and C_1. The short names of services, declared in service declarations, are used as entry names in (PSA-data, 473).

The directory (PSA-data, 473, servicel, sneaky-pete) is used to organize the domains in the inner constrained environment E_2. As with the outer constrained environment, this directory has a segment entry named "control", and PSA uses this segment for information about E_2. The other entry names used in (PSA-data, 473, servicel, sneaky-pete) are the short names of services, as declared in service declarations.

Before it builds the structure shown in figure 5-10, PSA will insure that all the demands and provisos specified in the acp of the user domain, A, and in the service declarations of the three services involved, can be met.

After the structure shown in figure 5-10 is constructed, PSA will fill in the call-in and call-out components of the acps of the domains B, C_1, D, and C_2 so that the domain entry capabilities shown in figure 5-4 can be created. We will not specify when the domain entry capabilities will be created since this question is bound up with the problem of specifying the operating system's control structure, which is beyond the scope of this thesis.

5.7. The Hidden Data Game

Suppose that the user of a proprietary service were afraid that the service was hiding data in its results. The user could employ a counter-spy program to sift through the results, looking for hidden data. Of course, the user will require some assurance that the counter-spy program is not the brother spy; so the counter-spy program will have to be audited.

The counter-spy program would go through the results, zeroing all unused bits. This is effective in dealing with storage formats that do not use all their allocated bits, like varying strings. Floating point results present a special problem, since data can be hidden in their low-order bits. One way to control the problem is to have programmers specify more accurately the necessary precision of floating-point data. When this is done, the counter-spy program, from the declarations of the results, will check that the low-order bits not needed for the specified precision are zero. This will force the proprietary service hiding data in floating-point results to alter those results more severely than otherwise, thus increasing the risk of using this method. An alternative approach to floating point results is to have the counter-spy program perturb the low-order bits -- enough to scramble the hidden data, but not so much as to reduce the precision of results

required by the user.

If the results include an area allocated in blocks (e.g. by PL/I allocate and free statements), the counter-spy program must check to see that there are no hidden blocks. The blocks which are expected to occupy the area will be connected together by pointers in a known way. The counter-spy program can copy all the expected blocks into another area, and then examine the blocks remaining in the first area. Unexpected blocks probably contain hidden data. Also, data can be hidden in an area's pool of unused space. Furthermore, each block which was expected should be sifted through by the methods of the paragraph above.

These ad hoc methods are suggested to deal with the threat of hidden data because there does not appear to be a uniform method of countering the threat that works in all cases. In fact, it is not possible to enumerate the places in a proprietary service's results where hidden data might appear without knowing the meanings of all the result data returned. These meanings are implicit in the declarations provided to the counter-spy program described above. Because the operating system cannot deal with meanings very well, each user of a proprietary service must protect himself -- for example, with the counter-spy program driven by appropriate declarations. The point is that the operating

system cannot provide the declarations.

It is possible to enumerate the domains into which hidden data might flow, which are the domains where the brother spy might reside. But this probably isn't useful because the enumeration might be quite long, and because an investigation of any of the domains on the list would require an invasion of privacy, which would require in turn a court order. Showing probable cause to get such a court order would be difficult. The privacy restriction mechanism of chapter 6 makes hidden data much easier to deal with.

5.8. Sneaky Signalling

There is a way by which a proprietary service can broadcast information, namely by manipulating the working set size of processes which are using the service. There is no way to stop a proprietary service from doing this except through auditing. Because the possibility of this sort of behavior exists, the operating system primitives which give out information about the working set sizes of processes must not be widely available. In other words, information about the working set size of a process using a proprietary service might be very sensitive information.

A proprietary service can encode information in its time of running (that is, the amount of processor time

required for an invocation of the service by some user process). Therefore this quantity is a piece of sensitive information.

A proprietary service can encode information in its time of delivering an answer if it has access to information about the passage of real time.

A proprietary service can broadcast information through a lock that the service is allowed to set and reset, provided that the service does not occupy a constrained environment. This is because setting a lock involves writing into the lock datum, and the only writable data accessible to services in a constrained environment are the contents of closeted segments. (Data in a process state is also writable, but such data is not used for locks for inter-process communication because data in a process state can be accessed only by one process.) Since a closeted segment S is accessible only in one domain D, setting a lock in S cannot broadcast information to processes outside (i.e., bound to some domain other than) the domain D.

5.9. The Threat of Sabotage

The purpose of this section is to define the blind service, and discuss its properties. A blind service is a service placed in an environment which protects the user of the service from the possibility that the service will stop working, or begin a sabotage campaign, at a time

chosen by some enemy of the user. This environment, called a blind environment, operates by denying to domains in the environment all unfrozen sources of information. This prevents the enemy from sending signals to his saboteur service. Also, domains in the blind environment can't find out whom they're working for, or what time it is. This prevents narrowly directed sabotage campaigns, unless the saboteur service can determine from its input the identity of the user of its services; and the timing of a sabotage campaign is made difficult since the saboteur service must deduce the passage of real time from the limited set of events to which it is privy. We do not offer any proof that a blind environment accomplishes any prevention of sabotage campaigns because such a conclusion depends on the quantity and quality of useful information which a saboteur can deduce from its inputs.

To be precise, a blind environment is a set B of domains such that:

(1) B is a constrained environment,

(2) all domains $D \varepsilon B$ encapsulate frozen services,

(3) all domains $D \varepsilon B$ can't access any unfrozen objects except closeted segments,

(4) all domains $D \varepsilon B$ can't find out whom D is working for, and

(5) all domains $D \varepsilon B$ can't find out what time it is.

A blind service is a service implemented in blind environment.

The owner of the user domain of a proprietary service may demand that the service occupy a blind environment with a variable in the proprietary call-out component of the acp of the user domain of the service. Similarly, the owner of a service S that uses a component service T may demand that T occupy a blind environment with a variable in the service declaration of S. Furthermore, owners of services will specify in service declarations whether their services will operate in blind environments, and PSA will not make a service available to a user who demands that the service occupy a blind environment unless the service declaration specifies that the service will operate in a blind environment.

Those parts of the operating system which are responsible for limiting the flow of information into blind environments must be audited.

5.10. Summary

There are many ways for the lessor of a proprietary service to harm a lessee of his service, and there are a few ways for a lessee to harm the lessor. Some of these harms can be prevented by the technological means presented in this chapter, all of which depend on a correctly implemented operating system which includes a Proprietary Services Administration (PSA). PSA constructs constrained environments using benign domains and closeted segments

in order to make it very difficult for a proprietary service to do argument spying. PSA implements account channels which proprietary services must use to send invoices for services rendered. PSA allows users of services to demand that the services be frozen, to guarantee mutually agreed maintenance of services. Finally, PSA implements blind environments which make it difficult for a service to carry out a sabotage campaign which is narrowly directed (i.e., against a particular set of users) or well timed.

PSA is not useful in solving a number of problems presented in this chapter, particularly the hidden data flow problem, the compiler-caller two-pronged attack problem, and the sneaky signalling problem. It is likely that legal protections will evolve to fill the gaps in the available technological protections.

Chapter Six

Privacy Restrictions

6.1. Invasion of Privacy

The mechanisms of the previous two chapters were motivated largely by the privacy needs of would-be users of proprietary services. The fact that we must regard the hidden data flow problem as a game, to be won by the more clever team, is indicative of the limitations of the mechanisms already described. In fact, if a user, e.g. a systems programmer, has the power to make copies of information and authorize others to access the copies; then he can invade the privacy of the original information owners (or others named by the copied information). This method of invasion of privacy is illustrated in figure 6-1. A systems programmer and a spy are shown communicating with their own domains through consoles. We are assuming that the programs in use in these domains (D_1 and D_2) include command interpreters which give the users at their consoles control over the processes bound to the domains. The systems programmer has a domain entry capability into D_3, a domain holding a data base; and we are assuming that the systems programmer can obtain records from the data base by calling D_3. The systems programmer steals information for the spy by directing his process to call into D_3 to obtain the information, and then to copy the information provided by D_3 into the shared segment A. The spy can direct his process to cause the output of information from

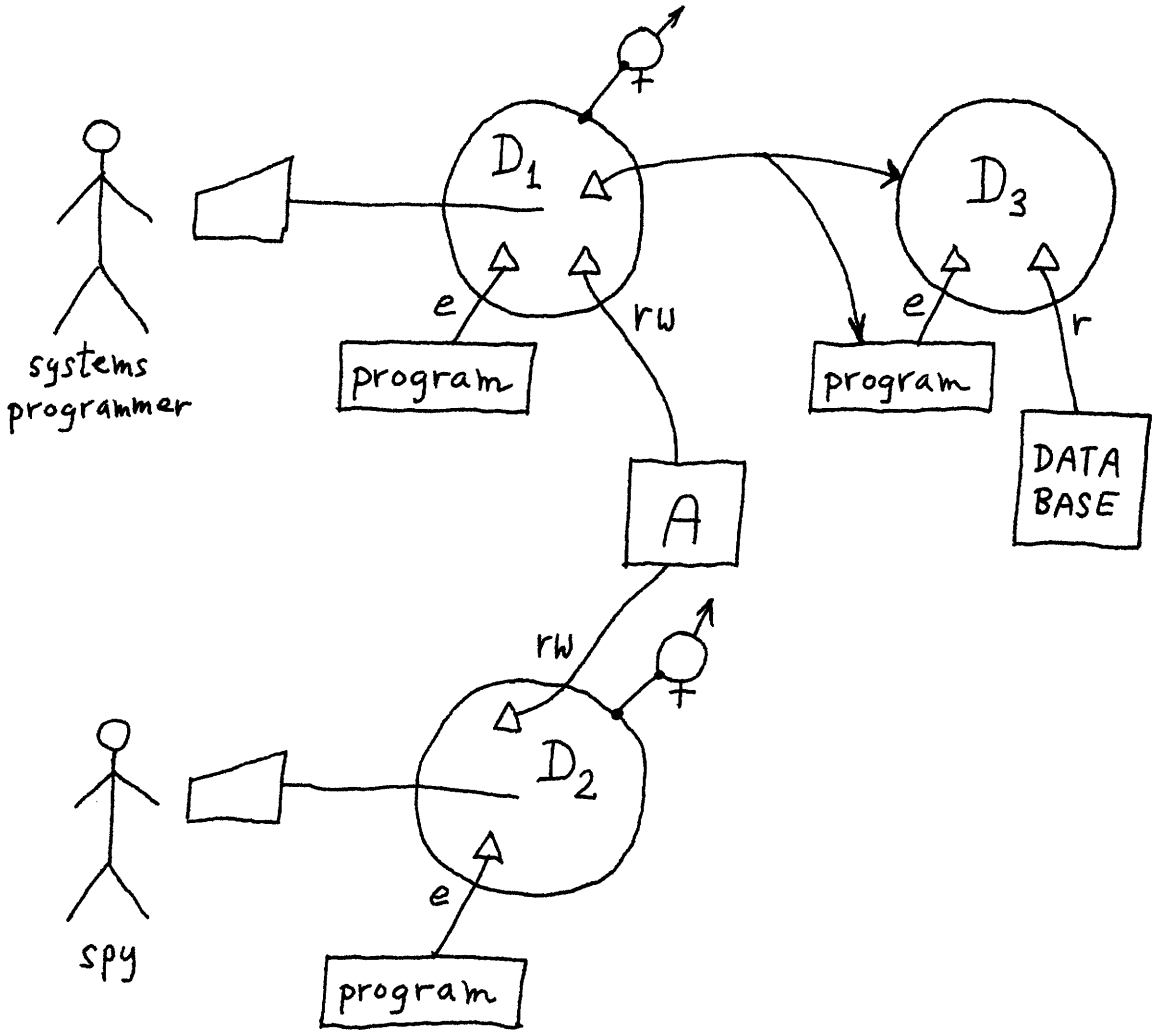


Figure 6-1. Invasion of privacy by a systems programmer.

segment A to his console, thereby obtaining the information stolen for him by the systems programmer.

Furthermore, it would be easy for the systems programmer to write a program to automate this entire operation by using segment A as a mailbox for the spy's requests. The spy could put messages in A which identified the records he wanted, and the system programmer's process would read these messages, make calls to D_3 and place the answers in additional messages in A.

The purpose of this chapter is to deter such invasions of privacy, and to raise the work factor for the humans perpetrating the invasion. Prevention of such an invasion of privacy requires that the flow of information to the spy be cut off, and two times at which this flow might be cut off are immediately evident. First, the information might be prevented from entering the spy's domain (*). Alternatively, the information might be prevented from appearing at the spy's terminal. This second, more permissive alternative is described in section 6.2. We call this scheme permissive because the spy's process is allowed to read the information whose output is to be prevented; but this permissive scheme is shown to have a flaw: the spy process can output the secret information using the very mechanism which is supposed to protect the information! This problem appears to be intrinsic to this type of protection mechanism. While it is

(*) Recall that information is in a domain if it can be read as data by a process bound to the domain.

not possible to prevent the unauthorized output of information, it is possible to detect it and trigger corrective action.

The privacy restriction mechanism is similar to, but more highly evolved than, the system of security compartments implemented in ADEPT-50 [Wei69].

6.2. Privacy Restrictions

Our method of preventing the invasion of privacy involves associating a set of privacy restrictions with every segment and every process in the computer. The crucial idea is that the restrictions are associated with the information contained in the segments and processes, and whenever information moves between segments and processes, the restrictions follow along. Whenever information is about to leave the computer (e.g. to appear at a user's console) the restrictions associated with the information have the power to prevent output.

Figure 6-2 shows our notation for sets of privacy restrictions. R_P is the restriction set of the process in figure 6-2, and R_A and R_B are the restriction sets of the segments. For the moment, the restrictions should be thought of as primitive, indivisible objects. We will describe the ownership and interpretation of restrictions presently.

The restrictions are propagated from one restriction set to another as processes execute load and store instructions which reference segments. Suppose the process of figure 6-2 reads from segment A with a load instruction. Then before the

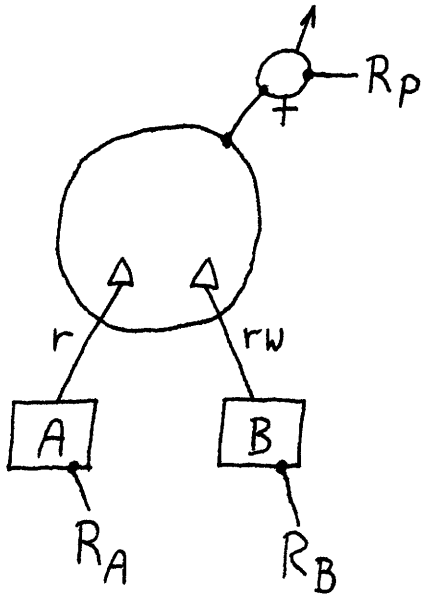


Figure 6-2. Sets of privacy restrictions associated with segments and processes.

process executes another instruction, the process' restriction set is updated as follows: $R_P = R_P \cup R_A$. Thus restrictions on the information in A are propagated to the process when the process reads A. Similarly, suppose the process of figure 6-2 writes information into segment B with a store instruction. Then before the process executes another instruction, the segment's restriction set is updated as follows: $R_B = R_B \cup R_P$. Thus restrictions on information in the process state are propagated to the segment when the process writes B.

The reader should remember that segments are not the only information-holding entities in a computer system. Names on entries in directories contain information, and these names can be read and written by user computations. Therefore, they must have restriction sets also, and these restriction sets must be updated and used as just described for segments.

In order to show how restrictions prevent the output of information, we must introduce a formal mechanism for identifying users of the computer system to whom output is directed. Our strategy is to adopt the convention that users are required to associate themselves with formal objects called principals when they use the computer. The purpose of the principal is to identify a person or a set of persons playing a specific role in an institution. Although in general a set of persons will be able to associate themselves with a given principal, only one person at a time will be allowed to so

associate himself.

Now we introduce a function which serves to define the effect of restrictions in preventing output. Let $\mathcal{R}$ be the set of all restrictions, and let $\mathcal{P}$ be the set of all principals. Then let $f: \mathcal{R} \rightarrow 2^{\mathcal{P}}$. That is, for every restriction r, f(r) is a set of principals, and these principals are the ones to whom output of information is to be restricted when the restriction r is associated with the information.

Although we write the function f as a single function defined for every restriction, this is mainly a notational convenience for the following discussion of the operating system primitive send. In fact, for each restriction r the set of principals f(r) is defined by the owner of restriction r. In other words, different pieces of the function f are defined by the different authorities who control the release of information stored in the computer.

The effect of restrictions is most easily explained if we assume that information can leave the computer only if it is displayed at a terminal where a user is logged in and associated with a principal. Furthermore, we assume that there is exactly one operating system primitive, called send, which can initiate the output of information to a terminal. Suppose a process P calls the send primitive to output information from segment A to a terminal where the user is associated with principal Q. The send primitive will allow the

output only if

$$Q \in \bigcap_{r \in R_A \cup R_P} f(r).$$

In words, Q must be in the intersection of the f(r)'s as r varies over $R_A \cup R_P$. ("r" is a dummy variable in the formula.) In other words, Q must be in the set f(r) for every restriction r in both of the restriction sets R_A and R_P.

It is fairly easy to see why we want $Q \in \bigcap_{r \in R_A} f(r)$ for output to be allowed, and somewhat difficult to see why we must also insist that $Q \in \bigcap_{r \in R_P} f(r)$. Restrictions arrived in R_A as A was written. When the information is to be output from A, there is no way to tell which of the restrictions in R_A were originally associated with the information being output, so all of the restrictions are applied. That is, we require $Q \in f(r)$ for all r in R_A for output to be allowed to Q's terminal.

To see why we also insist that $Q \in \bigcap_{r \in R_P} f(r)$, consider the program in figure 6-3. It will output the bit string **secr** no matter what restrictions are associated with it unless we insist that $Q \in \bigcap_{r \in R_P} f(r)$. It works by **outputting** constants which have no restrictions whatever associated with them, and the restrictions on the information that governs the choice between the two constants, the restrictions on **secr**, get no further than R_P.

When a process calls send and send finds that $Q \notin \bigcap_{r \in R_A \cup R_P} f(r)$, the output to Q's terminal will not be allowed and when this happens, we say the restriction strikes.

```
declare secr(100) bit based(p);  /* a secret */

declare one segment integer init(1);  /* a constant in its
                                         own segment */

declare zero segment integer init(0);  /* ditto */


do i = 1 to 100;

          if p -> secr(i) then call send(one);

                          else call send(zero);

          end;
```

Figure 6-3. A program to output any secret bit string.

Now we illustrate the use of privacy restrictions to fix the invasion of privacy problem shown in figure 6-1. Figure 6-4 shows figure 6-1 reproduced with restriction sets and principals added. The systems programmer is associated with principal Q1 and the spy is associated with principal Q2. All that is required to solve the problem is to associate a restriction r with the data base segment accessed by D_3, and define f(r) so that $Q2 \notin f(r)$. Of course $Q1 \in f(r)$, so the systems programmer (associated with Q1) can see information that comes from the data base. But when the system programmer's process writes information from the data base into the shared segment, the restriction r is propagated to R_A, whereupon the spy, associated with principal Q2, will not be permitted to see any information from A displayed at his console. Furthermore, as soon as the spy's process reads from segment A, it will be unable to send any information to the spy's console, because the process will have r in its restriction set.

6.3. Information Leakage Despite Restrictions

Restrictions striking are signals. That is, when a restriction strikes this fact is observable to the user whose output was not permitted. This might not be the case when the user doesn't know what to expect of the program producing the output, but many users will know what to expect from their programs. In particular, the program of figure 6-5 is so simple that any reader can quickly see what to expect from

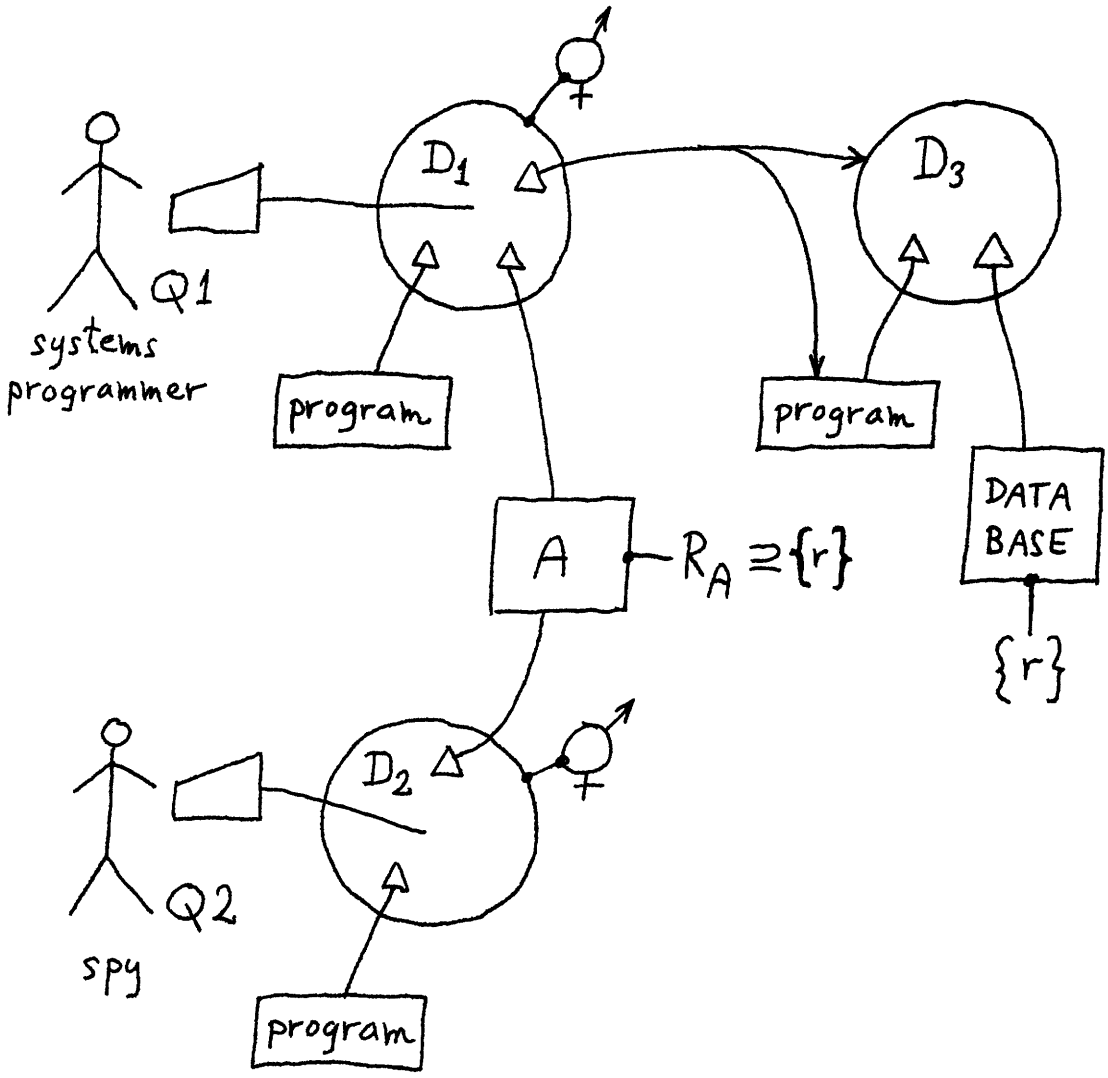


Figure 6-4. Preventing invasion of privacy with privacy restrictions.

it. The purpose of the program is to output an array of 100 bits, called "secret", as a pattern of striking restrictions. The first do loop of the program creates 100 segments in the directory whose name is held by the variable dir, using the operating system primitive make-seg. The entries in the directory are given the names "1", "2", ..., "100" by the make-seg primitive, and the domain where the program is being executed is given capabilities for the newly created segments. The segment numbers of the newly created segments are stored in the array seg(100). As each segment is created, its zeroeth word is initialized to the index of the segment in the "array" of segments in the directory named by dir. (The function "ptr" constructs a pointer of the form (seg#,word#) from its arguments.) The second do loop stores the secret as a pattern of restrictions in the restriction sets associated with the segments just created. We are assuming that the secret is stored in a segment S whose restriction set R_S contains a restriction which will prevent output of the secret to the user who is running the program. When the test "if secret(i)" of the second do loop is executed, the restrictions of R_S are propagated to R_P, the restriction set of the process. If the i^{th} bit of secret is 1, the program stores a zero into the first word of the i^{th} segment, thereby propagating the restrictions of R_S to the restriction set of the i^{th} segment. When this encoding is completed, the program directs its executing process to self-destruct by calling the

```
declare secret(100) bit(1) based(p);
declare seg(100) integer;
declare i integer;
   :
do i = 1 to 100;
          call make-seg(dir,char(i),seg(i));
          ptr(seg(i),0) -> word = i;
          end;
do i = 1 to 100;
          if secret(i) then ptr(seg(i),1) -> word = 0;
          end;
call logout;

login;
do i = 1 to 100;
          call send(ptr(seg(i),0) -> word);
          end;
```

Figure 6-5. A program which encodes and outputs information as a pattern of restrictions.

operating system primitive logout. The user immediately logs back in again, and commands his process to execute the third do loop. The purpose of the logout-login sequence is to get a process which is not contaminated by restrictions of R_S in its restriction set, which is accomplished because the login command creates a process with an empty restriction set. (*) The third do loop calls the operating system primitive send 100 times, attempting to output the zeroeth words of each of the 100 segments created earlier. If the output is successful, the index of the segment in the array of 100 segments is printed at the terminal. If a restriction strikes and output is not allowed, the index does not appear at the terminal. Assuming that the program runs to completion, the user has a list, printed by his terminal, of the zero bits of the secret; and a notification from the system that a restriction struck for every one bit of the secret.

Some deterrence against this use of restrictions to leak information might be provided by a mechanism which notifies the owner of a restriction whenever his restriction prevents output of information to any user. That is, the restriction mechanism would provide the information owner with a message to notify him that his information is being sneaked out of the computer. In the example above, one message would be

(*) If the login command gave the user a process with the same restriction set which that user's process had when it last logged out, this method of information leakage would require a crony's process to execute the third do loop.

sent for every one bit of the secret bit string. But in fact this mechanism would compound the problem, by allowing encoding of information as a pattern of restrictions in a single restriction set. Figure 6-6 shows a program which demonstrates this method. The first do loop creates 100 restrictions, in the directory named by the variable dir, with entry names "1",...,"100". The create-r primitive establishes each new restriction's owner and the restriction definition f(r). For our example, we will assume that the owner of these restrictions is the author of the program of figure 6-6, and we assume they are defined with f(r) = the empty set. The second do loop of the program encodes the secret as a pattern of restrictions in the restriction set of the process by using the operating system primitive place-p, which adds the restriction specified by its arguments to the restriction set of the process. Finally, the program directs the process to call the operating system primitive send, and all the restrictions in R_P strike. Some of the striking restrictions are from R_S, having arrived in R_P when the process executed "if secret(i)". The remainder of the striking restrictions are the ones which encode the secret. Now suppose that messages are sent to all the restriction owners. The author of the program of figure 6-6 would receive a message for every one bit of the secret. Of course each message would identify the striking restriction, and so by examining all the messages the secret can be reconstructed.

```
declare secret(100) bit(1) based(p);
declare i integer;
   .
   .
   .
do i = 1 to 100;
          call create-r(dir,char(i), ... );
          end;
do i = 1 to 100;
          if secret(i) then call place-p(dir,char(i));
          end;
call send("!");
```

Figure 6-6. A program which encodes information as a pattern of surveillance-generating restrictions in R_P.

From these examples it is clear that the privacy restriction mechanism does not erect walls through which information cannot flow, once the information can be read by a hostile program which can request output to a terminal, because the restriction mechanism provides a method of signalling for the hostile program. It should be noted that the privacy restriction mechanism does provide some protection against accidental, non-malicious unauthorized releases of information; but it is less effective in preventing the non-accidental, well-planned theft of information. The mechanism can, however, be used to raise an alarm against suspected theft of information.

An alarm which signals possible theft of information should be raised whenever the number of times a given restriction strikes down output to a given principal exceeds a fixed limit, specified by the restriction owner. The alarm should be two-pronged: the restriction owner should be notified that the principal exceeded his limit, and the principal which exceeded the limit should be denied any further use of the computer until the situation has been investigated by an appropriate authority. When a restriction strikes down output to a principal the time the limit is exceeded, the computation which is sending output to the principal should be stopped and saved for examination by the "appropriate authority" introduced above. Since the restriction owner and the principal whose computation exceeded the limit might be

responsible to different authorities, the choice of the "appropriate authority" can be non-trivial.

We will denote the limit introduced above as $L_1(r,Q)$, where r is a restriction and Q is a principal. The owner of r defines the value of this function for all principals Q. The limit defined by $L_1(r,Q)$ is the number of times the restriction r will quietly strike down output to principal Q, where "quietly" means that the alarm reaction defined above is not triggered.

The reader should note that even when the limit is set to zero, a would-be thief can easily use a single striking restriction to signal 10 or 12 or so bits of information. The method is similar to the program of figure 6-5, modified as shown in figure 6-7. The program creates 2^{10} segments, and then places the restrictions on the secret, which arrive in the restriction set of the process when it evaluates the expression "fixed(secret)", onto just one of the 2^{10} segments created previously. The segment whose restriction set is chosen to be contaminated in this way is the secretth segment in the array of 2^{10} segments. Later, an uncontaminated process executes the second do loop of figure 6-7, and the sequence of numbers "1","2","3",... begins to appear at the terminal. When the executing process requests output from the contaminated segment, the restriction strikes; and the user, seeing that it struck, knows the 10 bits of the secret by mentally adding 1 to the last number to appear at his

```
declare secret bit(10) based(p);
declare seg(1024) integer, i integer;
   :
do i = 1 to 1024;
          call make-seg(dir,char(i),seg(i));
          ptr(seg(i),0) -> word = i;
          end;
ptr(seg(fixed(secret)),1) -> word = 0;
call logout;

login;
do i = 1 to 1024;
          call send(ptr(seg(i),0) -> word);
          end;
```

Figure 6-7. A program which encodes 10 bits with one striking restriction.

terminal, and converting that number to binary notation.

All these methods of outputing secret information assume that the information is readable by a program which is in a domain that can request output to a terminal where the spy is willing to let appear the garbage generated by the encoding schemes discussed above. The next line of defense against information theft is, therefore, to try to prevent the secret information from being readable in the spy's domain. In the next section, a mechanism to accomplish this is described and evaluated.

6.4. Walls Around Sets of Domains

In this section, we present an extension to the privacy restriction mechanism which allows restrictions to prevent information from entering domains. Recall that information is <u>in</u> a domain if it is readable as data by a process bound to the domain. Information enters domains through segments, through processes calling and returning between domains, and through input from some device attached to the computer.

To allow restrictions to apply to domains, we introduce a new function $d: \mathcal{R} \rightarrow 2^{\mathcal{D}}$, where $\mathcal{D}$ is the set of all domains. That is, for each restriction r, d(r) is a set of domains. Now suppose a process P is calling or returning to a domain D. The call or return is permitted to occur provided D is a member of d(r) for every $r \in R_p$. Using set-theoretic notation, this requirement is written

$$D \in \bigcap_{r \in R_p} d(r).$$

The reason for this rule is as follows: for each restriction r in R_p, P's process state does contain or might contain information associated with the restriction r. The restriction owner has defined a set of domains d(r) where the information associated with r is permitted to be read. So if $D \notin d(r)$, the process will not be allowed to bind itself to domain D. The intersection operator ("$\bigcap$") applies this rule for every restriction in the restriction set of the process, R_p.

Now we want to introduce a similar rule for segments, but this is difficult because of the sharing of segments by domains. If a segment is readable in one domain and writable in another, then information written into the segment by a process bound to the latter domain immediately enters the former domain. The situation can be simplified by reducing the flexibility of allowed segment sharing, so that if a segment has a writer domain, that writer domain is the only domain from which the segment can be read. Thus, if a segment can be read from more than one domain, it cannot be written by processes bound to any domain. This scheme does not allow a segment to be readable in one domain and writable in another, at the same time. Segment sharing is not outlawed, but it must be mediated by operating system primitives, defined as follows:

1) initiate-read(seg#,code);

The segment S, specified by its segment number, is made readable in the domain D where the primitive was invoked, pro-

vided the segment is not writable in any domain, and provided further that

$$D \in \bigcap_{r \in R_S} d(r),$$

where R_S is the restriction set of segment S. The output argument code indicates to the caller the success of his request for access. If the segment is writable in some domain when initiate-read is invoked, initiate-read waits until the segment is no longer writable in any domain. If initiate-read finds that $D \notin d(r)$ for some $r \in R_S$, the restriction r strikes.

2) initiate-write(seg#,code);

The segment S, specified by its segment number, is made readable and writable in the domain D where the primitive was invoked, provided the segment is not readable or writable in any domain, and provided further that

$$D \in \bigcap_{r \in R_S} d(r),$$

where R_S is the restriction set of segment S. As above, the output argument code indicates success.

3) terminate-read(seg#);

The domain where the primitive was invoked ceases to be a reader of the specified segment.

4) terminate-write(seg#);

The domain where the primitive was invoked ceases to be a writer of the specified segment.

These primitives insure that when information enters domains through segments, the restriction sets of the segments

are examined and the formula $D \in \bigcap_{r \in R_S} d(r)$ is evaluated; and if $D \notin d(r)$ for any $r \in R_S$, the information in segment S will not be permitted to enter domain D.

Finally, information can enter domains through some device attached to the computer. For simplicity, we will treat this as a special case of information entering a domain through a segment. The segment serves as a buffer: the information flows from the device to the segment, and then the segment is made readable in the domain which requested the input. The device owner can specify that input from the device to any buffer segment B results in propagating a set of restrictions, R_{device}, into R_B. Then, when the buffer is to be made readable in the requesting domain D, the operating system requires that

$$D \in \bigcap_{r \in R_B} d(r).$$

This completes the specification of the extension of the privacy restriction mechanism to erect walls around sets of domains. As before, these walls provide protection against accidents, but they are not very effective against well-planned efforts of a spy to steal information, when the spy can place a program inside the wall. The methods of information theft developed in section 6.3 are easily applicable to the problem of getting information through this wall. For example, the method of theft shown in figure 6-5 can be applied by placing the program of figure 6-8 inside a domain which has access to the secret information. This program creates

```
declare secret(100) bit(1) based(p);
declare seg(100) integer;
declare (i, code) integer;
   :
do i = 1 to 100;
          call make-seg(dir,char(i),seg(i));
          call initiate-write(seg(i),code);
          ptr(seg(i),0) -> word = i;
          end;
do i = 1 to 100;
          if secret(i) then ptr(seg(i),1) -> word = 0;
          end;
do i = 1 to 100;
          call terminate-write(seg(i));
          end;
```

Figure 6-8. A program to encode 100 bits.

100 segments and encodes the secret information as a pattern of contaminated restriction sets -- contaminated, as before, with the restrictions from the segment holding the secret. After the secret has been encoded, the spy can command his process, running bound to a domain where the secret information is not permitted to enter, to execute the program of figure 6-9. This program attempts to initiate-read each of the segments which encode the secret. When a striking restriction prevents a successful initiate-read, the return argument code is set to a non-zero value. Thus, the secret information is obtained by examining the values of the return argument code.

The spying method just illustrated can be defeated if the system augments the restriction set of the process with the striking restrictions whenever initiate-read or -write returns a non-zero code because $D \notin \bigcap_{r \in R_S} d(r)$, provided the striking restrictions also do not allow output to the spy's terminal, through an appropriately defined f(r). It can also be defeated if there is a limit on the number of times a restriction will quietly strike down input to a domain; such that when the limit is exceeded, the offending process and its entire computation are saved for later examination by an appropriate authority, the restriction owner is notified, and the user whose process triggered this action is denied further access to the system until a time set by the "appropriate authority" just introduced (again). But this alarm mechanism, like the alarm mechanism of the previous section, cannot pre-

```
declare segment integer, i integer, code integer;
   :
do i = 1 to 100;
          call initiate(dir,char(i),segment);
          call initiate-read(segment,code);
          if code = 0 then call send(0);
                      else call send(1);
          end;
```

Figure 6-9. A program to print out 100 encoded bits.

vent a single striking restriction from signalling 10 or 12 or so bits of information. This is illustrated by the program of figure 6-10, which encodes 12 bits by creating 2^{12} segments and contaminating the restriction set of one of them -- the secretth one -- with the restrictions on the secret information. As before, the segments are created in the directory named by the variable dir. Once the information has been encoded, the spy can run the program shown in figure 6-11 which will output to the spy's console the zeroeth word of each of the 2^{12} segments. We are assuming that the spy's process is running bound to a domain which the secret information is not permitted to enter, so when the program of figure 6-11 tries to initiate-read the contaminated segment, the alarm described above goes off, and output to the spy's terminal is shut off by the system. So then the spy knows the 12 bits, by adding 1 to the last number to appear at his terminal and converting to binary.

We will denote the limit just introduced as $L_2(r,D)$, where r is a restriction and D is a domain. The owner of r defines the value of this function for all domains D. The limit $L_2(r,D)$ is the number of times the restriction r will quietly strike down input to domain D, where "quietly", as before, means that the alarm reaction is not triggered.

```
declare secret bit(12) based(p);
declare seg(4096) integer, i integer;
   :
do i = 1 to 4096;
          call make-seg(dir,char(i),seg(i));
          call initiate-write(seg(i),code);
          ptr(seg(i),0) -> word = i;
          call terminate-write(seg(i));
          end;
call initiate-write(seg(fixed(secret)),code);
ptr(seg(fixed(secret)),1) -> word = 0;
call terminate-write(seg(fixed(secret)));
```

Figure 6-10. A program to encode 12 bits with one striking restriction.

```
declare (segment, code, i) integer;
   :
do i = 1 to 4096;
          call initiate(dir,char(i),segment);
          call initiate-read(segment,code);
          call send(ptr(segment,0) -> word);
          end;
```

Figure 6-11. A program to output 12 encoded bits.

6.5. The Conflict between Disclosure and Privacy

In the previous two sections, we described alarm mechanisms triggered by a count exceeding a limit, where the quantity being counted was the number of times a restriction struck down output to a particular user(principal), or input to a particular domain. The action which follows the alarm includes checkpointing the computation which set off the alarm, calling in an "appropriate authority" to investigate the saved computation, and notifying the restriction owner whose limit was exceeded. If two or more striking restrictions set off an alarm at the same time, it might be that the notifications just specified are being used to encode secret information, and so it might be appropriate to notify one of the restriction owners before the other. Therefore the computer system should not notify either(any) of the restriction owners at the time of the alarm. The "appropriate authority" introduced above should notify the restriction owners, in an order (and with a timing) based on his investigation of the saved computation. (The reader should note that the above strategy assumes that there is no ranking of restrictions available for use in deciding whom to notify first. Such a ranking might be defined if all the restriction owners were subject to a single authority.)

The important thing about the decision of the "appropriate authority" is that he is deciding whether the notifications he can authorize will result in disclosure of sensi-

tive information. To do this he must understand the meaning of the presence of the striking restrictions in the restriction set that struck down output to a terminal or input to a domain. If one of the striking restrictions encodes secret information, notification results in disclosure; whereas if the striking restriction represents the presence, in the information that was to be output to a terminal or input to a domain, of information that the restriction owner is responsible for protecting, then notification results in an increased awareness on the part of the restriction owner of where his information is flowing inside the computer. In the latter case, notification furthers protection of privacy, because the restriction owner will know that an unauthorized release of information was attempted, and he can take remedial action.

When a single restriction strikes and sets off no alarm, the restriction owner should be notified because the striking restriction represents an attempted unauthorized release of information. But if two or more restrictions strike and set off no alarm, the possibility remains that one or more of the striking restrictions are being used to encode sensitive information. Once again, the problem is to decide whether the purpose of each striking restriction is disclosure (through encoding) or the protection of privacy. But there is no easy way to decide this. The only obvious clue that would indicate that restrictions are being used to encode information is a

large number of striking restrictions. Therefore, our heuristic solution to the problem of what to do when two or more restrictions strike is to sound the alarm if there are many striking restrictions, and otherwise, to notify the restriction owners. If the decision is to sound the alarm, the computation which set off the alarm is checkpointed and an "appropriate authority" is called in to investigate the saved computation and notify the restriction owners in some reasonable order. If the decision is to notify the restriction owners, the computation whose output to a terminal or input to a domain was struck down is allowed to proceed: the offending process signals an error condition and looks for an enabled condition handler in the domain to which it is bound.

Now we must define how many restrictions striking are enough to set off the alarm. In order to let the restriction owners choose the amount of protection they are receiving, we associate a limit $L_3(r)$ with every restriction r. When a set of restrictions $\{r_1, r_2, \ldots, r_n\}, n \geq 2$ strikes without sounding the alarms defined in the previous two sections, the alarm will be sounded nevertheless if for some i, $L_3(r_i) \leq n$, where $1 \leq i \leq n$. It is clear that this algorithm will sometimes err, but the fact that none of the previously defined alarms were set off by the striking restrictions indicates that the information being protected by the striking restrictions is not the most sensitive or valuable information stored in the computer.

6.6. After the Restriction Owner is Notified

After a restriction owner is notified that his restriction has struck down output to a terminal or input to a domain in the computation of a user associated with a principal Q, the restriction owner will establish communication with the user associated with Q, and ask him what question his program was answering, and what its sources of data were, especially the data which carried with it the striking restriction. Upon receiving user Q's response, the owner of the restriction will have to decide: (1) Is it reasonable for user Q to get an answer to his question, in light of the reduction of privacy which such a release would imply for the restriction owner or other parties whose privacy the restriction owner is responsible for? (2) Does he (the restriction owner) believe what user Q said his program was doing? The restriction owner must make a judgement between disclosure and privacy.

If the judgement, once made, is to release the information, the restriction owner will command the computer system to lift the restriction from the answers generated by Q's program, perhaps replacing it with a new, slightly looser restriction. Assuming that the restriction struck down output to a terminal, then if r is the restriction which struck, the restriction owner could replace r, in the restriction set associated with Q's answer, with a new restriction r' such that $f(r') = f(r) \cup \{Q\}$. With the restriction thus loosened, Q's process will be able to use the send primitive to get the

answer to Q's terminal.

Thus we see the computer system must have primitives to place, lift, and replace restrictions; create and destroy restrictions; and define and redefine f(r)'s and d(r)'s. These primitives are defined in section 6.9.

It is important to note that the judgement which the restriction owner must make might be a delicate and difficult judgement. This is because the judgement depends on the sensitivity of the information which carried the striking restriction in the first place, the nature of the computation which Q's program performed on this information, and the nature and sensitivity of Q's program's other inputs. For example, if Q's program aggregates the information which carried the striking restriction, as for example by computing the average of a set of numbers, then the answer is likely to be considered less sensitive than the input data. But if Q's program combines input information from two different sources, the answer might well be more sensitive than either of its inputs. For example, combining input about a person's income where the information sources are the Internal Revenue Service and the Census Bureau is illegal. This illustrates the fact that the sensitivity of information is not an absolute, but rather is a variable which depends upon the context in which the information is used.

If the restriction owner does not trust the user associated with Q, he (the restriction owner) will want to audit

Q's program before making his judgement on the question of lifting the striking restriction. He will want to see whether Q's program is aggregating the sensitive information, or combining it with other information. If the judgement is to release the information (subject to some specified restriction), and if Q intends to use the program periodically, the restriction owner might be faced with the task of periodically auditing Q's program. This is unreasonable on the face of it, because auditing is a costly task, performed by people. Therefore the computer system must contain mechanisms for freezing Q's program, once it has been audited, so that it cannot be changed; and mechanisms for associating with the frozen program the capability to lift or replace the restriction owner's restriction. Mechanisms for freezing this type of program are implemented by the computer system's Restriction Removal Administration (RRA). The strategy of the RRA is to get a copy of Q's program from Q (the source program), and release it to the information owner, who will audit it. When the information owner agrees to release the information, the RRA compiles the program from its copy of the source program, and installs the program in a domain which is under the control of the RRA. The information owner gives this domain the capabilities for lifting or replacing restrictions, and the RRA gives Q the right to call the domain. The primitives which this domain can use to lift and replace restrictions are given in section 6.9.

The RRA strategy given extends easily to the case in which Q's program is combining information that carries restrictions belonging to more than one restriction owner. Q's program must be audited by all the restriction owners, and all must agree to the release of the program's outputs and give the domain containing the RRA's frozen copy of the program the necessary capabilities for lifting or replacing restrictions.

6.7. Benefits and Costs of Privacy Restrictions

The privacy restriction mechanism is a tool which society can use to (1) hold information more securely, and(2) request and enforce judgements between disclosure and privacy. The mechanism is vigilant in its action to keep restrictions associated with information by propagating restrictions to restriction sets associated with information destinations (segments and processes' states) whenever information is copied or combined by the computer. The mechanism erects walls around domains, striking down any attempt to make information enter a domain which it is not permitted to be in by the function d(r), where r is a restriction associated with the information. Because each domain is under the control of a specific authority in society, and because the seg-limit component of the access control packet of a domain allows such authorities to place strict controls on the collection of programs which can direct the actions of processes bound to domains, established authority has effective means of preventing information pro-

tected by privacy restrictions from being read by hostile programs.

When a restriction strikes, the restriction owner will be asked to make a judgement between disclosure and privacy. The privacy restriction mechanism will not operate to make such judgements better in any moral sense, but it will make them occur more frequently. Members of society will be more aware of the balance between disclosure and privacy, provided they are aware of the increased volume of decisions.

Policy decisions between disclosure and privacy will be expressed in programs, as described in the previous section, and as a result the complexity of these decisions can be allowed to increase, to the limit of the programmer's art. To the extent that the ability to handle complexity allows policies which are more fair, the privacy restriction mechanism is a force for good.

The privacy restriction mechanism has six major costs. First, there is the cost of hardware to perform and control union operations on restriction sets associated with processes and segments. This hardware is described in the next chapter. Second is the cost of executing system software which makes decisions regarding information transfers, which decisions depend on restriction sets and the functions d(r) and f(r). Third is the cost of using larger numbers of smaller segments, which will result from the necessary efforts of programmers to keep their restriction sets straight. Information which a

programmer expects to carry a uniquely different set of restrictions must be stored in its own segment to avoid inadvertent association of the restrictions with other information. Fourth is the cost of developing software to harness the privacy restriction mechanism to solve real problems. Fifth is the cost of developing and auditing programs that aggregate or combine information in ways that demand a propagation of restrictions more complex than the union operation performed by hardware. And finally, sixth is the cost of responding to alarms generated by striking restrictions.

The first three costs are likely to fall dramatically with the cost of computing hardware. Two of the costs (numbers 4 and 5) are one-time software costs whose levels are tied to the productivity of programmers and auditors. The sixth cost arises from a requirement for the services of a highly trained investigator, and the amount of this cost will depend on the rate at which alarms are set off, and the productivity of the investigator.

Good estimates of these costs will not be possible until the operation of a prototype system, serving a real community of users, can be studied.

6.8. Process Synchronization

When two processes have access to the same segment, they can interact in ways that defeat the purpose of the privacy restriction mechanism, or that make its implementation inefficient. For one example, suppose that a process requests

that information be sent from a segment to a terminal. The send primitive evaluates $\bigcap_{r \in R_P \cup R_{seg}} f(r)$, and suppose the output is permitted and started; i.e., the send primitive causes the computer hardware to begin sending information to the terminal. Now suppose that another process writes information into the given segment, in such a place that the newly written information will be sent to the terminal. If this were allowed to happen, the newly written information would escape from the computer despite the restrictions associated with it. This is called the sender-writer problem.

Another example of difficulties introduced by multiprocessing is the writer-reader problem. This occurs when two processes are sharing a segment -- one writing and the other reading. Suppose that the processes are implemented in a multiprocessing computer system, and that there are two processors available, one assigned to each process we are considering; and suppose that the writer process writes a word in the segment, and immediately afterwards the reader process reads that word. When these events occur, the restriction set of the segment must be updated (as follows: $R_{seg} = R_{seg} \cup R_{P_w}$, where P_w is the writer process) when the writer process writes, and then the restriction set of the reader process must be updated ($R_{P_r} = R_{P_r} \cup R_{seg}$, where P_r is the reader process) when the reader process reads. Thus we see that restrictions must pass from the restriction set of the writer process to the restriction set of the reader process.

One logical way to accomplish this is to maintain a single central copy of R_{seg} in the multiprocessing computer system. All the processors would refer to this central copy. But this reduces the efficiency and availability of the computer, because of contention for access to the central mechanism which holds R_{seg}, and because if the central mechanism breaks down, the entire computer system becomes unavailable.

Our solution to these process synchronization problems is to require that writable segments have no readers. This requirement is not the same as the similarly worded requirement introduced in section 6.4, because in that case the writers and readers of segments which were discussed were domains, whereas in this case the writers and readers are processes. To be precise, we call a process which can read a segment S a reader process of S, and we call a process which can write a segment S a writer process of S. Our strategy is to allow segments to have one reader-writer process, or many reader processes, but never one reader process and a different writer process at the same time. So when a segment has a reader process which isn't a writer process itself, it has no writer process.

When a segment has no writer process, its restriction set will not change. A multiprocessing computer system would be free to make copies of such an unchanging restriction set. A correctness proof for the send primitive would be free to assume that the segment would not change.

When a segment has a writer process, the writer process is a reader process of the segment but the segment has no other reader processes. When a writer process calls send, send can assume the segment won't change provided send doesn't return until the output is complete. A multiprocessing computer system would not be required to broadcast changes in the restriction set of a segment with a writer process, because only one processor of the system at a time would be permitted to deal with a segment with a writer process.

Segment sharing between would-be reader and writer processes must be mediated by operating system primitives. Our design incorporates this mediation into the primitives which were introduced in section 6.4 to mediate sharing of segments by domains. The primitives operate by manipulating a somewhat modified segment capability, in the context of a slightly modified processor state transition rule. The segment capability introduced in Appendix 1 was the 4-tuple (type,mode, length,addr). To implement the idea of a segment having a single writer process, the segment capability must be expanded by the inclusion of a component whose purpose is to identify the writer process, if the segment has one. The new segment capability is therefore the tuple (type,mode,proc_id, length,addr), where proc_id is a unique identifier of a process and mode is a 2-bit string (the bit w(mode) having been superceded by the proc_id component). The processor state transition rule (figure A1-2, part 3) is modified by the re-

placement of the test of the w(mode) bit by the test, "is proc_id(♀) = proc_id(cap)?", and the test which validates segment reading is modified by the inclusion of tests to exclude reading by non-writer processes when the segment has a writer, which is indicated by a non-zero value in the proc_id component of the segment capability. No process will have a zero proc_id. A fragment of the modified state transition rule is shown in figure 6-12.

Reading words as instructions from segments will be considered by the privacy restriction mechanism to be equivalent to reading data words. Therefore the instruction fetch logic of the state transition rule (figure A1-2, part 1) must be modified, as shown in figure 6-13. The purpose of the additional tests is to exclude execution of a segment by non-writer processes when the segment has a writer.

Having modified our definitions of process and capability to support our segment-sharing process-synchronizing strategy, we can proceed to define the primitives which implement the strategy. These definitions supercede the definitions of the same-named primitives in section 6.4.

1) initiate-read(seg#,code);

The segment S, selected by its segment number, is made readable in the domain D from which the primitive was invoked, and the invoking process is remembered as a reader process of S, provided that the segment S is not writable by any process in any domain, and further provided that $D \in \bigcap_{r \in R_S} d(r)$. The primi-

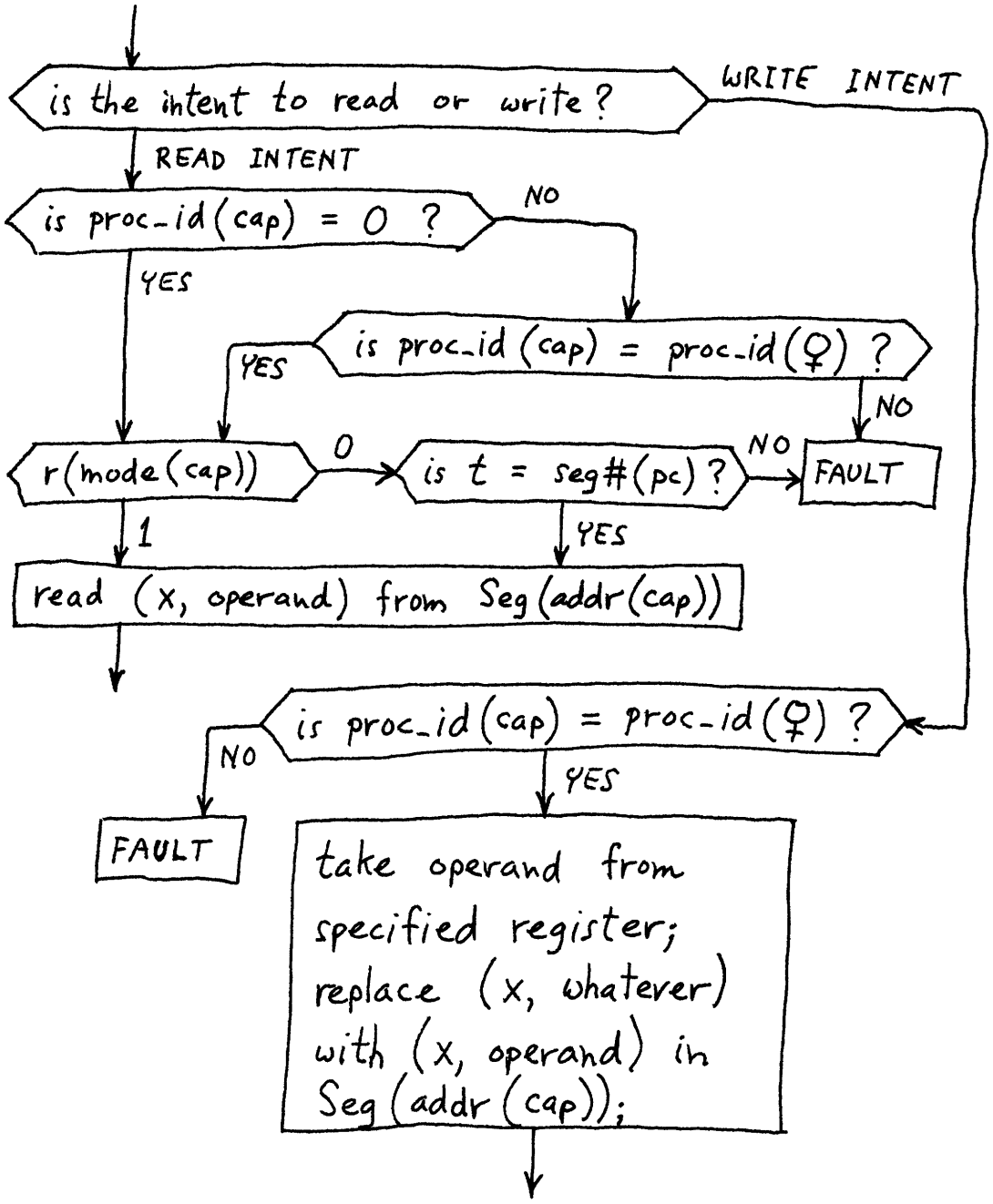


Figure 6-12. State transition rule - modified operand fetch logic.

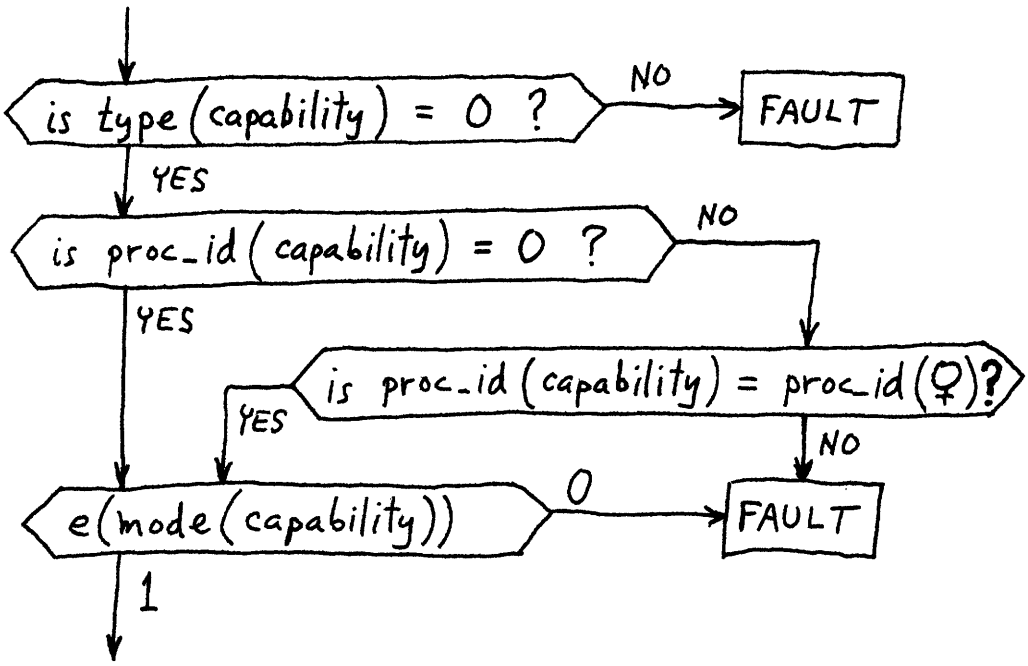


Figure 6-13. State transition rule – modified instruction fetch logic.

tive makes the segment S readable in domain D by turning on the r(mode) bit in the segment capability for S in D's C-list. If S has a writer (process or domain), the process which invoked the primitive is made to wait until S has no writer, at which time the executing process is made a reader. If information in S is not permitted by restrictions to enter domain D, and if none of the striking restrictions set off the alarm defined by $L_2(r,D)$, the striking restrictions are associated with the output argument code whose purpose is to indicate the outcome of invoking the primitive.

2) terminate-read(seg#,code);

The process invoking the primitive ceases to be remembered as a reader of the segment S, selected by its segment number in the domain D from which the primitive was invoked. If the process invoking the primitive is the last remembered reader of S in domain D, S ceases to be readable in domain D. This is accomplished by turning off the r(mode) bit in the segment capability for S in D's C-list. The output argument code indicates if invoking the primitive was successful.

3) initiate-execute(seg#,code);

4) terminate-execute(seg#,code);

These primitives are identical to the above two primitives, except that the e(mode) bit of the segment capability is manipulated. We consider execute access to a segment to be a special case of read access in this context, so the list of remembered reader processes introduced above will include processes

which invoked initiate-execute as well as those which invoked initiate-read.

5) initiate-write(seg#,code);

The segment S, specified by its segment number, is made readable and writable by the process invoking the primitive, in the domain D where the primitive was invoked, provided either the segment has no reader processes and no writer processes or the invoking process is already the writer process, and provided that $D \in \bigcap_{r \in R_S} d(r)$. The primitive makes the segment readable and writable by the invoking process in domain D by placing the proc_id of the invoking process into the proc_id component of the segment capability for S, and by turning on the r(mode) bit in the segment capability for S, in D's C-list. If S has either readers or writers, the process invoking the primitive is suspended and placed in a queue of would-be writers. At such future time that there are no readers, no writers, and no higher-priority would-be readers or writers, the suspended process is made a writer of S. If information in S is not allowed by restrictions to enter domain D, and if none of the striking restrictions set off the alarm defined by $L_2(r,D)$, the striking restrictions are associated with the output argument code whose purpose is to indicate the outcome of invoking the primitive.

6) terminate-write(seg#,code);

The process invoking the primitive, and the domain D where the primitive was invoked, cease to be readers and writers of the segment S, selected by its segment number in domain D. This

is accomplished by zeroing the proc_id component of the segment capability for S, and by turning off the r(mode) bit in the segment capability for S, in D's C-list; provided the invoking process actually is a writer of S. The output argument code indicates if invoking the primitive was successful.

The critical reader will have noticed that the above definitions of primitives do not make any reference to the access control packets of segments which are being made accessible by the primitives. We are assuming that the access control packet of a segment is consulted at the time a segment is assigned a segment number in a domain, and if access is not to be permitted, no segment number is assigned. If access is to be permitted, the mode specified by the acp of the segment is intersected with the mode specified by the seg-limit component of the acp of the domain, and this mode (possibly more restrictive than the two modes it was formed from) is stored in the table of contents segment for the domain; and it is this intersected mode which must be consulted by the primitives just defined. These primitives will not give any domain more access to any segment than the amount of access specified by the intersected mode.

When the owner of the segment or domain modifies his authorizations stored in the access control packet, the intersected mode must be recomputed. If, at the time the intersected mode is recomputed, the domain which is affected has a greater amount of access than permitted by the new intersected mode,

that access should be revoked. The reader should note that we are giving priority to the revoker of access, placing his interests above those of the user of the access (the domain and process(es) affected). This is a social design choice because the choice takes the form, "Whose purposes shall be served first?" There are no great technological problems involved in revoking access; for example, Multics implements immediate access revoking.

6.9. Restriction Administration Primitives

Restrictions are named and controlled through the use of the naming hierarchy. For this purpose, we define a new kind of directory entry: the restriction entry. This type of entry contains the following information about the restriction: (1) the definition of the set of principals f(r), i.e. the value of f(r) for this r; (2) the definition of the set of domains d(r), again for this r; (3) the definitions of the three limit functions $L_1(r,Q)$, $L_2(r,D)$, and $L_3(r)$; and (4) the name of the principal who is to be the recipient of notifications generated by the striking of this restriction.

The access control packet of a restriction entry consists of a list of terms, and each term consists of a tree name of a domain and a four-bit mode. The modes of access to a restriction are called read ("r"), modify ("m"), place ("p"), and lift ("l"). The meaning of each term of the acp of a restriction is that the named domain is allowed to successfully invoke primitives: to examine the restriction entry if the "r" mode

bit is on, to modify the restriction entry if the "m" bit is on, to place the restriction in the restriction set of a segment or a process if the "p" bit is on, and to lift (remove) the restriction from the restriction set of a segment or process if the "l" bit is on.

The arguments of the primitives are defined with the following declarations.

directory	char(*)	This is the tree name of a directory.
entry	char(*)	This is the name of an entry of a directory.
f	char(*)	This character string specifies a set of principals, the set f(r).
d	char(*)	This character string specifies a set of domains, the set d(r).
L1	char(*)	This character string specifies the function $L_1(r,Q)$, for one r.
L2	char(*)	This character string specifies the function $L_2(r,D)$, for one r.
L3	integer	This is the limit $L_3(r)$.
principal	char(*)	This is the identity of a principal.
seg_name	char(*)	This is the tree name of a segment.
seg#	integer	This is the segment number of a segment capability.

The restriction administration primitives are defined as follows:

1) create-r(directory,entry,f,d,L1,L2,L3,principal);

A new restriction is created in the specified directory with

the specified entry name, provided that the domain where the primitive was invoked is permitted to modify the directory, and provided the restriction set of the process invoking the primitive is empty. This last proviso prevents the encoding of secret information as a pattern of created restrictions. The remaining arguments of the primitive are input arguments used to initialize the contents of the new restriction entry. The acp of the new restriction entry is initialized to contain a single term giving "rmpl" access to the domain where the primitive was invoked.

2) delete-r(directory,entry);

The restriction specified by its entry name in the specified directory is deleted, provided the domain where the primitive was invoked is permitted to modify the directory and the restriction entry, and provided the restriction set of the process which invoked the primitive is empty. This last proviso prevents secret information from being encoded as a pattern of deleted restrictions. After it is deleted, a restriction can no longer strike or sound any alarm.

3) read-r(directory,entry,f,d,L1,L2,L3,principal);

The contents of the restriction entry, specified by its entry name in the specified directory, are copied into the variables named by the remaining arguments, provided the domain where the primitive was invoked has "r" access to the restriction.

4) modify-r(directory,entry,f,d,L1,L2,L3,principal);

The contents of the restriction entry, specified by its entry name in the specified directory, are discarded and replaced by the contents of the variables named by the remaining arguments, provided the domain where the primitive was invoked has "m" access to the restriction, and provided the restriction set of the process which invoked the primitive is empty. This last proviso prevents secret information from being encoded as a pattern of modified restrictions.

5) place-r(directory,entry,seg_name);

The restriction specified (by its entry name in the specified directory) is added to the restriction set of the specified segment, provided the domain where the primitive was invoked has "p" access to the restriction, and provided the domain where the primitive was invoked has authority to place restrictions on the segment. This latter authority is granted through the use of a new mode in the access control packet of a segment. This new mode is called "p", so the rule for the place-r primitive is that the domain where the primitive was invoked must have "p" access to both the restriction and the segment.

6) lift-r(directory,entry,seg_name);

The specified restriction is removed from the restriction set of the specified segment, provided the domain where the primitive was invoked has "l" access to the restriction.

7) replace-r(dir1,entry1,dir2,entry2,seg_name);

The first four arguments must specify two restrictions r_1 and r_2. Provided the domain where the primitive was invoked has "l" access to r_1 and "p" access to r_2 and "p" access to the segment specified by seg_name, and provided r_1 is a member of the restriction set of the specified segment, r_1 is removed from and r_2 is added to the restriction set of the segment in an indivisible operation.

8) place-s(directory,entry,seg#);

The specified restriction is added to the restriction set of the specified segment, provided the domain where the primitive was invoked has "p" access to the restriction, and provided that the specified segment is writable in the invoking domain, and provided the invoking process is a writer process of the segment.

9) lift-s(directory,entry,seg#);

The specified restriction is removed from the restriction set of the specified segment, provided the domain where the primitive was invoked has "l" access to the restriction, and provided that the specified segment is writable in the invoking domain, and provided the invoking process is a writer process of the segment.

10) place-p(directory,entry);

The specified restriction is added to the restriction set of the invoking process, provided the domain where the primitive

was invoked has "p" access to the restriction.

11) lift-p(directory,entry);

The specified restriction is removed from the restriction set of the invoking process, provided the domain where the primitive was invoked has "l" access to the restriction.

6.10. Individual Privacy and the Computer of the Future

The privacy restriction mechanism can be used to give individuals greater rights of privacy than ever before. Suppose there were a public computer utility which implemented the privacy restriction mechanism, and suppose that individuals used the utility to prepare and submit income tax returns. The Internal Revenue Service (IRS) would use the utility to accept and process tax returns completely within the utility, in the context of policies and procedures for destroying all hard copies generated by the utility. Because of these policies and procedures, individuals could be sure that their income tax information was available only through the computer utility. Individuals could then use the privacy restriction mechanism to retain control over release of income tax information held by the Internal Revenue Service.

An individual (call him K) who uses the utility to prepare and submit income tax returns would keep several different types of information in the utility. First, K would keep the relevant records from which his tax return would be prepared; and second, K would generate trial tax returns in an effort to pay the lowest possible tax. These types of information would

not be released to anyone but K. The third type of information in this example is the tax return that K actually submits to IRS.

The first and second types of information would be kept by K in segments under his control and having an access control packet containing one term giving K's home domain "rwp" access, with the copy flag on so that K's home domain can pass the segment as an argument to a domain encapsulating an income tax preparation service. In addition, the first and second types of information would be protected by a restriction r_1 owned by K, such that $f(r_1) = \{K\}$, $d(r_1) = \{$home-domain(K), tax-service-domain-working-for-K$\}$, $L_1(r_1,Q) = 0$, $L_2(r_1,D) = 0$, and $L_3(r_1) = 1$. Note that when we say, "$f(r_1) = \{K\}$", we are using the symbol "K" to stand for the principal with which the individual K associates himself. This should not cause any ambiguities, since the meaning is clear from the context. The principal to be notified when r_1 strikes is K.

To protect himself from his tax preparation service, K must require that the service be encapsulated in a benign domain. This will prevent any hostile program in the service from using any of the spying techniques described in section 6.4, because all the writable segments created by a benign domain must be closeted segments, under the control of PSA. K would place the restriction r_1 in the restriction sets of segments containing input to the tax preparation service, and so r_1 would be propagated to restriction sets of the service's

closeted segments, and to restriction sets of the service's output argument segments. Thus the restriction r_1 would prevent K's tax records and his trial tax returns from being accessible in any domains other than K's home domain, and the benign domain encapsulating his tax preparation service; and the information restricted by r_1 will be permitted to appear only at K's terminal; and any attempt to steal K's information in a sneaky way will set off an alarm because of the low values of $L_1(r_1,Q)$, $L_2(r_1,D)$, and $L_3(r_1)$.

When K chooses a tax return to submit to IRS, he will replace the restriction r_1 in the restriction set of the segment containing the tax return information with a new restriction r_2, such that r_2 is also owned by K, $f(r_2) = \{K\} \cup P(IRS)$, where P(IRS) is a set of principals associated with the Internal Revenue Service whose users might need to examine K's tax return, $d(r_2) = \{\text{home-domain}(K)\} \cup D(IRS)$, where D(IRS) is a set of domains working for the Internal Revenue Service, $L_1(r_2,Q) = 0$, $L_2(r_2,D) = 0$, $L_3(r_2) = 1$, and the principal to notify if r_2 strikes is K. (The Internal Revenue Service will publish specifications of the sets P(IRS) and D(IRS) for taxpayers using this method of submitting tax returns.) In addition, K will give "1" access to r_2 to some IRS data-aggregating domains. To insure that this requirement does not unreasonably degrade K's privacy, these data-aggregating domains must be maintained by the Restriction Removal Administration, and the frozen programs in the data-aggregating domains must be avail-

able for auditing by any taxpayer or interested association of citizens.

The restriction r_2 will prevent K's tax return from being made available to anyone but the authorized agents of the Internal Revenue Service. K will submit his tax return by commanding his process to call an IRS tax-return-accepting domain, passing the tax return as an argument segment. That domain will make a copy of the tax return, and the privacy restriction mechanism will propagate the restriction r_2 to the copy.

In this example we have not specified higher values of L_1, L_2, and L_3 for r_2 than we specified for r_1, because we are assuming that K is both very concerned about possible release of his tax return information and also affluent enough to afford the services of investigators to respond to alarms which his restriction sets off. We are assuming that the processing performed by IRS is more complex than the processing required to prepare a tax return, and therefore more likely to accidentally cause the restriction r_2 to strike. If K were not so affluent, he would probably specify higher values for $L_1(r_2,Q)$, $L_2(r_2,D)$, and $L_3(r_2)$, and obtain a reduced amount of protection for his tax return.

In addition to the enhancement of personal privacy which the privacy restriction mechanism makes possible, the mechanism can be used to loosen restraints on the use of statistical information which arise from the possibility of indirect dis-

closure.

> "Indirect as well as direct disclosures must be considered, and these can be a major source of difficulty. Thus, suppose a small county has six hardware stores, and that a city within the county has four of them. If retail sales are published for the county, and also for the city (we assume each would individually meet disclosure requirements) an indirect disclosure occurs. Each of the two stores in the balance of the county could directly determine his competitor's sales by taking the difference between the county statistics and the city statistics. Thus, if disclosure is to be avoided the data for the city can be made available, and not the county, or for the county and not the city. Indirect disclosures should be avoided, at least in any sensitive type of information.
>
> . . .
>
> "The consequences of indirect disclosures are that priorities are necessary in determining which statistics will be made available and which will not, in order to avoid making available some relatively unimportant information and thereby subsequently denying statistics that have highly important uses.
>
> . . .
>
> " ... the priority problem means that the first comer, who may have a limited use or need in terms of public interest, may forclose the possibility of later retrieval of other more important information."
>
> -- Morris H. Hansen [Hans71]

Two methods of applying the privacy restriction mechanism to the loosening of indirect disclosure restraints can be suggested. The first method is not very practical, because it depends on not publishing any of the statistics, but making the statistics available through the computer utility instead, to disjoint sets of users. The lack of practicality is evident when considering the problem of keeping these sets of users disjoint. Suppose we call the two statistics A and B. We

associate restrictions r_A and r_B with A and B, respectively; and we require that $f(r_A) \cap f(r_B) = \emptyset$ and $d(r_A) \cap d(r_B) = \emptyset$. So the restrictions will insure that no principal has access to both A and B, and no domain has access to both A and B. But this elegant "solution" will be punctured by any principal in $f(r_A)$ (or $f(r_B)$) who communicates A (B) to some principal in $f(r_B)$ ($f(r_A)$), perhaps communicating by means external to the computer utility.

A more realistic solution to the problem can be suggested, involving publishing one of the statistics, say A, and holding B in the computer utility. B will not be released to any principal, but it can be released to data-aggregating programs that have been audited and certified to produce results from which B cannot be reconstructed. If the restriction r_B is associated with B, as before, we will have $f(r_B) = \emptyset$, and $d(r_B)$ defined to be a set of domains maintained by the Restriction Removal Administration which encapsulate programs which have been audited by the authority that controls the use of B.

Let SA (for Statistical Authority) denote the authority responsible for the use of B. A would-be user of B will submit a program to RRA to be frozen and subsequently audited by SA. If the program passes the audit, whose purpose is to insure that B cannot be reconstructed from the result produced by the program; then SA will permit the segment containing B to be read by the domain D in which RRA installs the audited program, and SA will include D in the definition of $d(r_B)$, and

SA will give domain D "l" access to r_B and "p" access to a new restriction r_D, so that the program in D can lift r_B from its result, replacing it with r_D. The user will be allowed to call D, and the answer produced by D will be protected by r_D, which will remain under the control of SA. The set $f(r_D)$ will include the principals for whom the program in D is working, so that D's result can get to its would-be users.

In this way the statistic B can be put to some use, albeit at some expense, despite the fact that statistic A was published and the possibility of indirect disclosure requires that B not be publicly known.

6.11. Proprietary Services Revisited

The user of a proprietary service can use privacy restrictions to help him win the hidden data game, and the lessor of a proprietary service can use privacy restrictions to deny his competitors the use of his service.

If a user of a proprietary service is concerned about the threat of hidden data, he can associate a restriction r with the argument data which will be input to the proprietary service. Since the results produced by the service depend on the input arguments provided to the service by its caller, the restriction r will be associated with the results, and also with any hidden data generated by the service. Now, by appropriately defining f(r) and d(r), the user of the proprietary service can limit the flow of hidden data to a set of domains d(r). Furthermore, the set of principals f(r) who are permitted to

see the results of the proprietary service will be the only principals allowed to see any hidden data, if it is there. By limiting f(r) and d(r) to principals and domains which he trusts, the user of the proprietary service can reduce his worries over the threat of hidden data. On the other hand, if the user of the proprietary service doesn't trust anyone, he can use f(r) and d(r) to reduce the scope of his distrust, and conserve thereby his suspicious energies.

The lessor of a proprietary service can deny his competitors the use of his service by associating a restriction r with the program segments which implement his service, and defining f(r) and d(r) to include his customers (and their customers, ...) and their domains, but to exclude his competitors and their domains. Whenever a process is using the proprietary service, the restriction r will be propagated to the restriction set of the process, and from there it will be propagated to the restriction sets associated with the results produced by the service. Since his competitors and their domains are excluded from the sets f(r) and d(r), the lessor of the service can be sure that his competitors will not get to use any results produced by his service.

Chapter Seven

Privacy Restriction Processor

7.1. Introduction

The purpose of this chapter is to describe the hardware components and system strategies of a multiprocessing computer system which implements the privacy restriction mechanism which was defined in the previous chapter. The major new component of this hardware design is the Privacy Restriction Processor (PRP). The other hardware components are Processing Units (PUs) (which have been called "processors" in previous chapters), memory, and a central communicator. Figure 7-1 shows how these modular components are interconnected to form a multiprocessing computer system. The arrangement of PUs and memory boxes is straightforward: each PU can access each memory. Also, each PU can communicate directly with all other PUs in the system, through the central communicator. This communication is required to efficiently manage the system's segmented virtual memory. The function of the central communicator could be performed by the memory boxes, but we show it as a distinct hardware box to underscore its independence.

Each PU has an attached PRP. For the moment, we will restrict our attention to a single PU-PRP pair. Recall that the PU is the active component of a process P. The purpose of the attached PRP is to hold P's restriction set; to hold the restriction sets of segments being read and written by P; and to

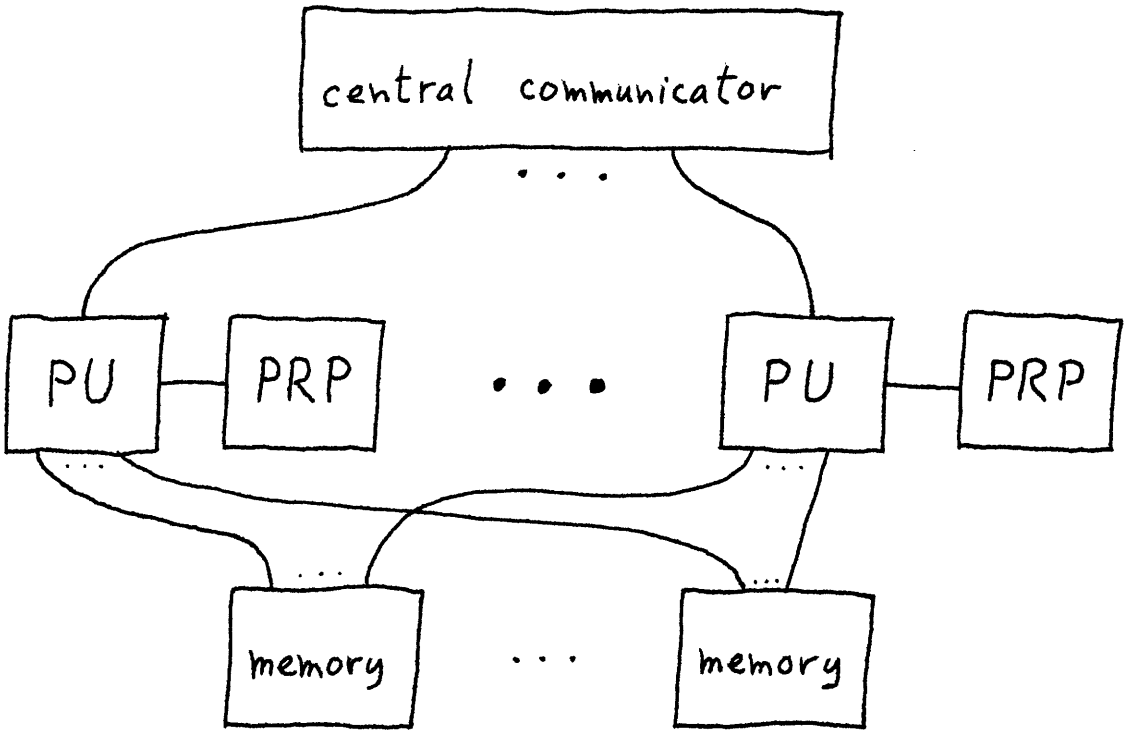


Figure 7-1. Multiprocessing computer system.

perform (set-theoretic) union operations on the restriction sets it holds, in response to the actions (reading and writing of segments) of the attached PU. The PRP has a small high-speed memory for holding restriction sets, backed up by restriction set storage in main memory. Therefore an additional function of the PRP is to manage its restriction set memory, reading and writing restriction sets from and into the main memory. We will assume that the PRP uses its attached PU's ports to memory for this.

Our description of the PRP will proceed from a consideration of the important events which occur in a multiprocessing computer system. The complete list of important events is given in section 7.3. Of these, the most important events are the reading and writing of segments by a PU. Therefore we begin by describing how a PU goes about reading and writing segments. Our description is an abstraction from the state transition rule in Appendix 1. (*)

The PU contains programmable registers including general registers, index registers, base registers, and a program counter. The base registers and program counter contain segmented addresses of the form (seg#,word#). This form, (seg#, word#), is also the form of the effective addresses generated by running processes. Segment numbers are meaningful in the context of particular domains. Each process state contains

---

(*) The principal difference between the state transition rule of Appendix 1 and the abstraction used here is the merging, for the purposes of this description, of the instruction fetch logic with the operand fetch logic.

a binding to a domain's C-list, thus providing a context for the interpretation of effective addresses. The PU interprets an effective address (seg#,word#) by using seg# as an index in the C-list to access the capability for the addressed segment. From the capability the PU can determine if the intended access is permitted in terms of mode (i.e., if reading, writing, or executing the segment is permitted), and if word# $\leq$ length(segment). Provided this is true, the PU will complete the reference to the segment.

In order to speed up memory accesses, each PU contains an associative memory to hold segment capabilities that were used recently. This is similar to the associative memories in the processors of Multics [Sc71]. Each word of the associative memory has the form (seg#,mode3,length,page_table_addr). The PU, in interpreting an address (seg#,word#), first searches the associative memory for a word whose seg# field is equal to the seg# of the effective address. If such a word is found, the PU uses the other information in that word instead of looking up the capability in the C-list. Thus the system saves a memory cycle. But if there is no match found in searching the associative memory, the PU fetches the capability from the C-list, loading the newly fetched capability into the associative memory. Thus subsequent accesses to the same segment don't have to look up the capability. Since the associative memory is finite, the least recently used word in it might be deleted when a new capability is loaded.

Recall from section 6.8 that segment capabilities have the form (type,mode,proc_id,length,page_table_addr), where mode is a 2-bit string which tells whether the segment identified by the capability is readable, and whether the segment is executable, by processes bound to the domain defined by the C-list from which the capability came; and proc_id identifies the writer process of the segment, if it has one. When a segment capability is loaded into a PU's associative memory, the information in these two components (mode,proc_id) is condensed into a 3-bit mode string called mode3. The following three functions define what each bit of mode3 will be whenever a new capability is loaded into a PU's associative memory.

$$w(\text{mode3}) = (\text{proc\_id}(\text{capability}) = \text{proc\_id}(♀));$$

$$r(\text{mode3}) = r(\text{mode}) \wedge (\text{proc\_id}(\text{capability}) = 0 \vee (\text{proc\_id}(\text{capability}) = \text{proc\_id}(♀)));$$

$$e(\text{mode3}) = e(\text{mode}) \wedge (\text{proc\_id}(\text{capability}) = 0 \vee (\text{proc\_id}(\text{capability}) = \text{proc\_id}(♀)));$$

Recall that proc_id(♀) is the identifier of the process being evolved by the PU. The reader should compare these three functions with the state transition rule fragments shown in figures 6-12 and 6-13 and note that figure 7-2 and figures 6-12 and 6-13 perform equivalent access validation tests.

Figure 7-2 is the fragment of the PU's state transition rule which performs access validation tests and uses the associative memory. In figure 7-2, "AM" means associative memory. "P's intent" refers to whether the process P being evolved by

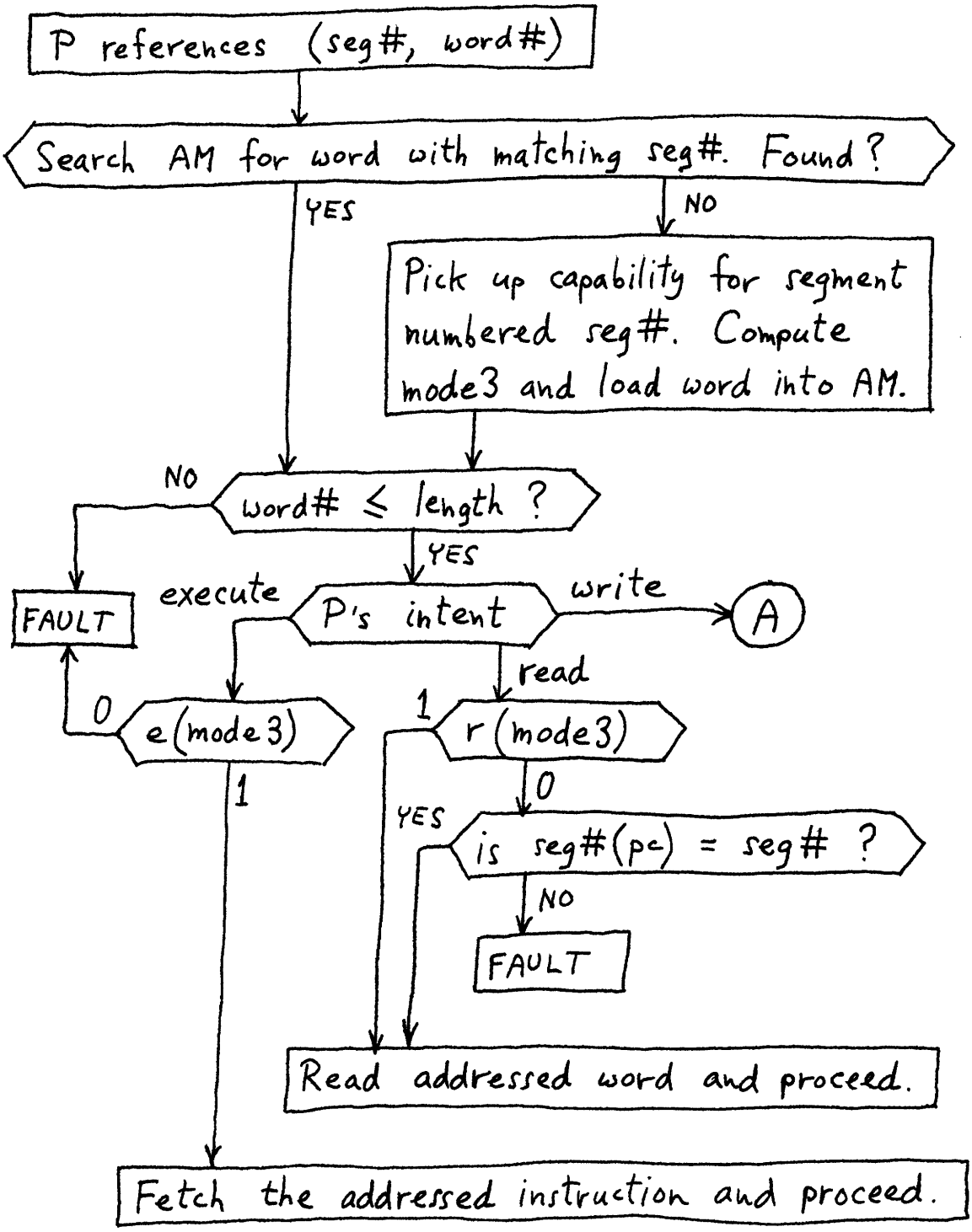


Figure 7-2, part 1. Fragment of PU state transition rule.

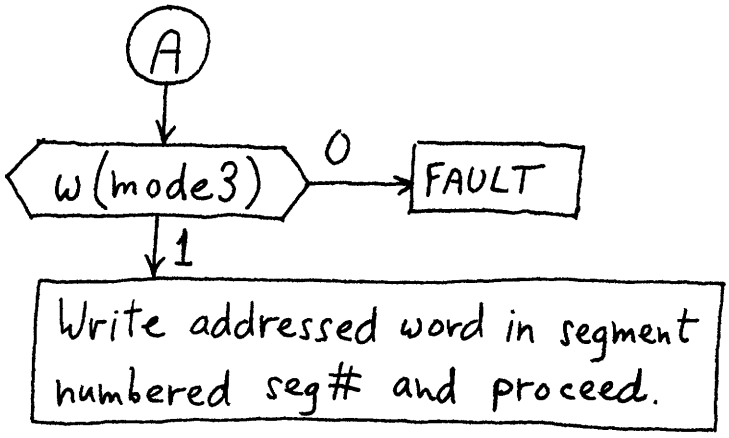


Figure 7-2, part 2. Fragment of PU state transition rule.

the PU is reading a word as data, reading it as an instruction, or writing data. "FAULT" means a trap to the operating system due (in the cases in this flowchart) to an attempted access violation. Figure 7-2 is an abstraction from the state transition rule in Appendix 1, modified to accomplish process synchronization required by the privacy restriction mechanism.

7.2. Associative Memory Control Bits for the PRP

For the moment, assume that we have built a PRP to hold restriction sets of processes and segments and do the appropriate union operations on these sets as the PU proceeds. The PU must tell the PRP what to do, but it is necessary for the PU to avoid telling the PRP to do too much. For example, suppose a process P stores a word into segment A. The PRP will perform the operation $R_A = R_A \cup R_P$. Now suppose P stores another 99,999 words into A. It would be terribly wasteful to do the union operation 100,000 times if R_P is unchanging and thus the operation accomplishes nothing the last 99,999 times.

Of course it is possible to avoid this waste. The technique is to add two bits to each word of the PU associative memory. If A is a segment whose capability is held in the associative memory, we will call these bits U_read(A) and U_write(A). When U_read(A) is 1, the meaning is "do a union when reading from A"; and when U_write(A) is 1, the meaning is "do a union when writing into A". The U_read and U_write bits are set to 0 after a union is actually performed, so subsequent unions will not be performed until the bits are reset

to 1.

Whenever a union operation expands R_p (i.e., whenever an operation of the form $R_p = R_p \cup R_A$ makes R_p a larger set), all the U_write bits in the associative memory are reset to 1. Whenever the execution of the restriction administration primitive lift-p reduces R_p, all the U_read bits are reset to 1. This is accomplished by means of a special instruction. But when R_p is expanded by a union operation, this is noticed by the hardware and the U_write bits are reset to 1 according to the state transition rule fragment in figure 7-3. Figure 7-3 is an expansion of figure 7-2, and thus it shows how to modify a PU to accomodate a PRP; and in particular, figure 7-3 shows how the U_read and U_write bits are used.

7.3. Events

Many important events which occur in multi-processing computer systems are affected by the introduction of the Privacy Restriction Processor. The PRP responds to events occurring in the associated PU, and it also responds to events generated by the action of other PUs and communicated by one of the system's central communicators. Among the events occurring in the associated PU, in addition to the reading and writing of segments, the PRP responds to the execution of special instructions by the PU. This instruction set is given in section 7.6. The union operations performed by the PRP, together with its responses to special instructions and the system's central communicator, collectively are the PRP's tactics for

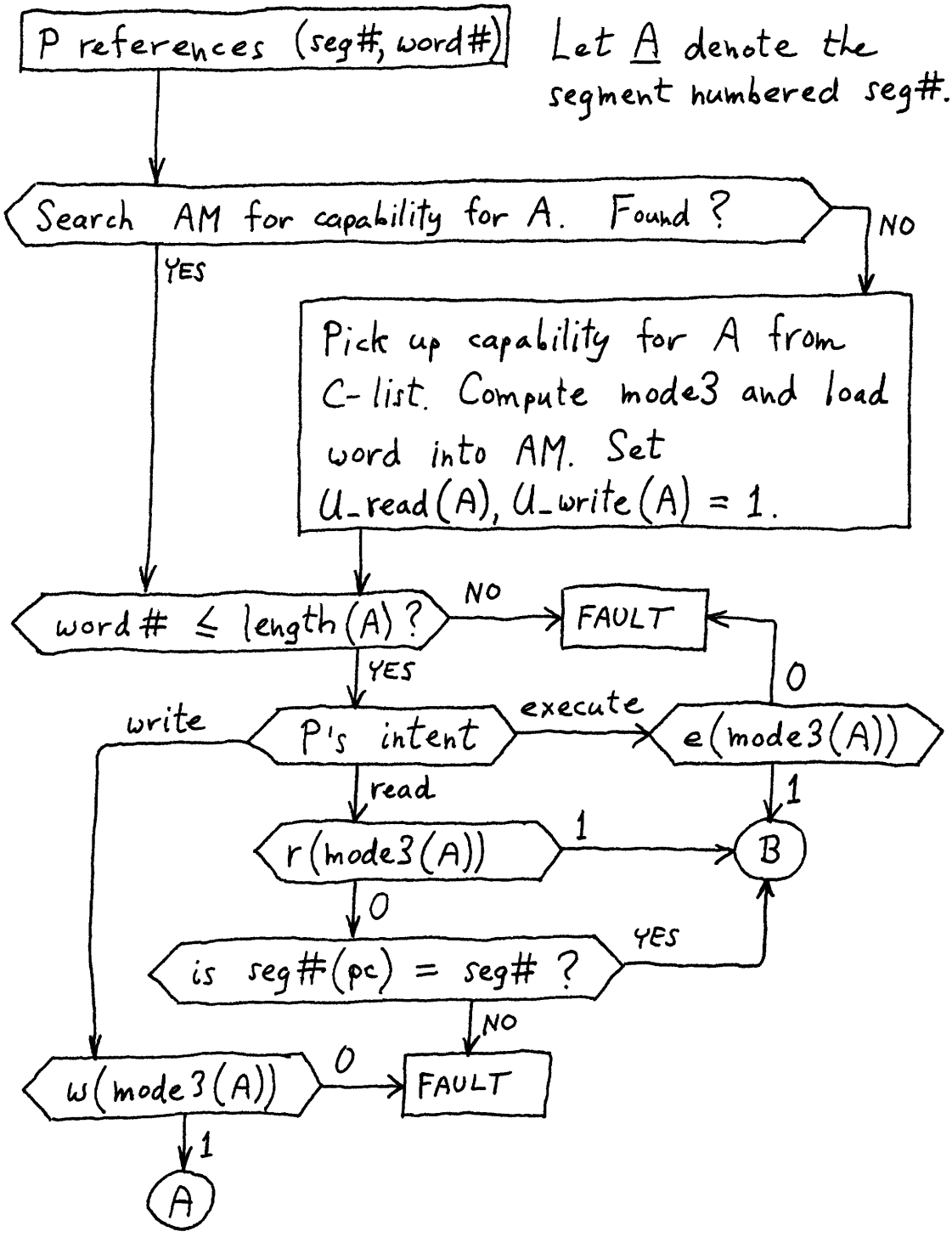


Figure 7-3, part 1.
Same fragment of PU state transition rule as in figure 7-2, expanded to show control over PRP.

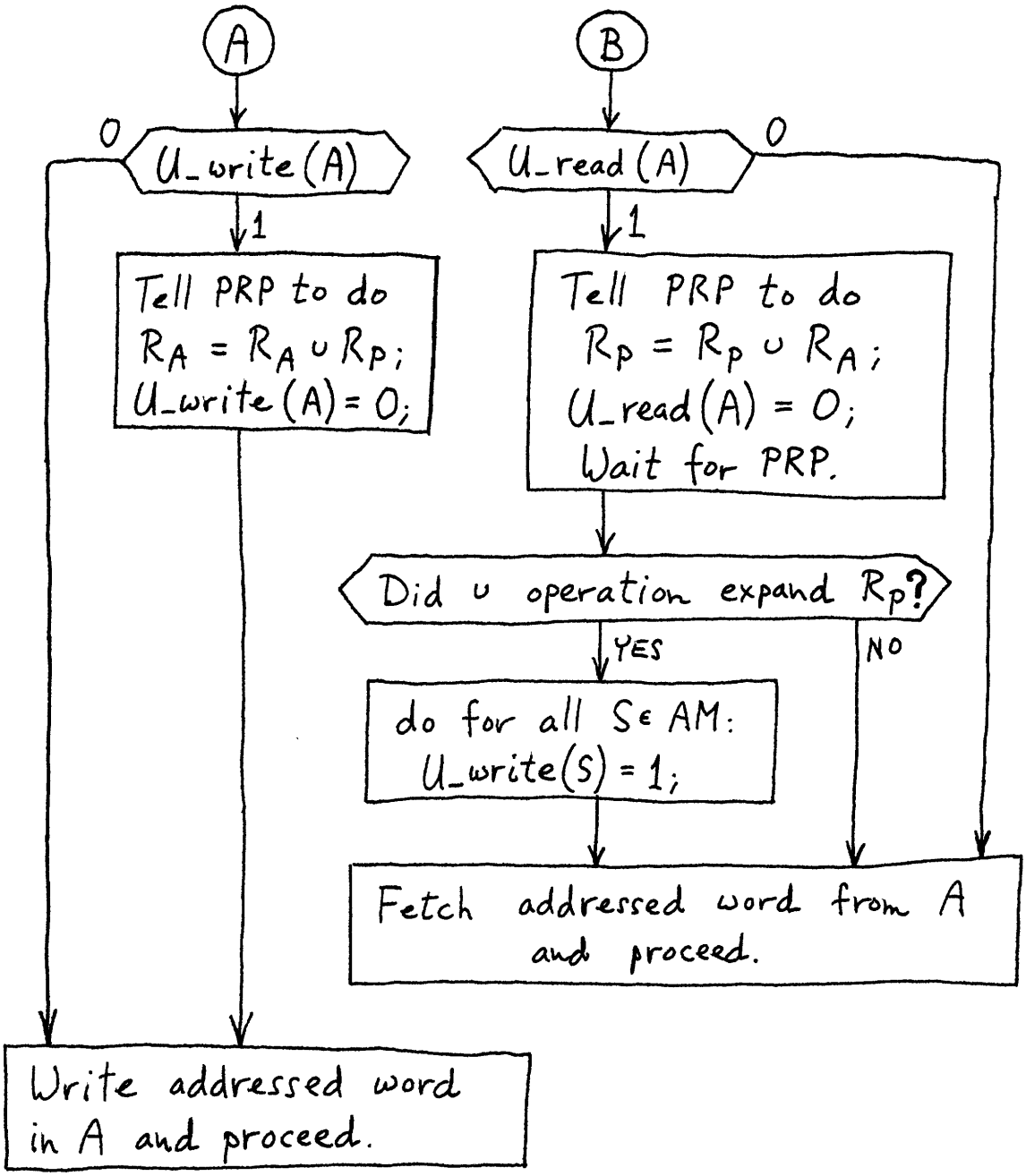


Figure 7-3, part 2.
Same fragment of PU state transition rule as in figure 7-2, expanded to show control over PRP.

implementing restriction sets. These tactics are derived from a consideration of strategies for handling the important events occurring in a multiprocessing computer system.

The use of the special instructions just introduced is permitted only in authorized operating system domains. The dom_ids of authorized domains will be wired into or set in switches in the PU, and the PU will fault whenever an unauthorized domain issues one of the special instructions. Instructions which signal the central communicator, and input/output instructions, are similarly protected from unauthorized use.

The following list of important events which occur in multiprocessing computer systems was derived from contemplation, from conversations with graduate students and faculty of the Computer Systems Research Group of Project MAC, and from our study of Multics. The list is as complete as this student could make it.

In the list of events, P means any process being evolved by a PU.

1) a) P reads from a segment.
   b) P writes into a segment.
   c) The PU evolving P picks up a segment capability.
   d) The PU evolving P deletes a segment capability from its associative memory.
2) P does a send to output information.
3) The PRP does a union operation that produces an oversize restriction set. (Restriction overflow.)
4) a) P does an initiate-read.

4) b) P does a terminate-read.

c) P does an initiate-execute.

d) P does a terminate-execute.

e) P does an initiate-write.

f) P does a terminate-write.

5) a) P does a call-domain.

b) P does a return-domain.

6) a) P does a create-r.

b) P does a delete-r.

c) P does a read-r.

d) P does a modify-r.

e) P does a place-r.

f) P does a lift-r.

g) P does a replace-r.

h) P does a place-s.

i) P does a lift-s.

j) P does a place-p.

k) P does a lift-p.

7) The operating system reassigns a PU to a new process.

8) A PU takes a fault. Especially interesting is the segment fault which needs to make a segment inactive, since this involves the central communicator.

9) A PU takes an interrupt.

10) a) The operating system starts up.

b) The operating system reconfigures a PU into the system, or out of the system.

c) The operating system shuts down.

11) a) Segments are backed up by a "daemon" process under the control of the operating system.

b) Segments are retrieved from backup media by a "daemon" process under the control of the operating system.

7.4. Formats

From a consideration of the strategies for handling the events just introduced, a set of PU-dependent and PRP-dependent storage formats emerges. In fact, the formats and the strategies coalesce together in the mind of the designer. Some of the formats were presented in previous sections, and are repeated here for completeness.

7.4.1. Segment Capability

mode	Two bits called "r" and "e" which control whether processes may read or execute the segment.
proc_id	The unique identifier of the process which may write the segment, if it is writable. (All zeroes means no writing allowed.)
page_table_addr	The absolute address of the segment's page table, provided fault = 0.
fault	A validity bit for page_table_addr. The PU takes a segment fault when it picks up a segment capability with fault = 1.
length	The length of the page table.
has_R_set	A bit which says whether this segment has a restriction set. (*)

(*) All of the segments which contain operating system data bases will not have restriction sets, and therefore the has_R_set bit in segment capabilities in operating system domains will be 0. If operating system data base segments did

7.4.2. PU Associative Memory Word

This is the associative memory introduced in section 7.1, which holds segment capabilities for speedy reference by the PU.

mode3	Three bits called "r", "e", and "w" to control reading, executing, and writing. These bits are computed as specified in section 7.1.
U_read	A bit which says the PRP must do a union when the PU reads from this segment.
U_write	A bit which says the PRP must do a union when the PU writes into this segment.
has_R_set	A bit which says whether the segment has a restriction set.
page_table_addr	The absolute address of the segment's page table.
length	The length of the page table.
seg#	The segment number of the segment in the domain to which the process being evolved by the PU is bound.

7.4.3. Domain Entry Capability

dom_id	The unique identifier of the domain which can be called through the use of this capability.
(seg#,word#)	The address of a procedure entry point in the called domain, where calling processes which use this capability will begin

---

have restriction sets, those restriction sets would gradually accumulate almost every restriction in the system, because all the processes use the services of the operating system. Then this large collection of restrictions would be propagated to all the processes, and the system would stop working.

execution.

R_check — A bit which says whether to check that the called domain is in $\bigcap_{r \in Rp} d(r)$ when a call is made using this capability, and similarly for the paired return. (*)

7.4.4. PRP Associative Memory Word

page_table_addr — The absolute address of the page table of a segment for which a restriction set is held in the PRP.

location — The location of the restriction set in the PRP's restriction set memory.

size — The size of the restriction set.

modified — A bit which says whether the restriction set of the segment has been modified.

writable — A bit which says whether the associated PU is evolving a writer process of the segment.

7.4.5. Restriction Set Control Word

available — A validity bit. If a PRP begins to pick up a restriction set for which available = 0, it will cause its attached PU to fault.

size — The size of the restriction set.

7.4.6. Restriction

oversize — A bit which says that this restriction stands for a set of restrictions stored elsewhere.

(*) The check whether the target domain (the destination of the call or return) is in $\bigcap_{r \in Rp} d(r)$ is necessary to erect walls around sets of domains, as described in section 6.4. Recall that d(r) is the set of domains that information restricted by r is permitted to be in. The purpose of the R_check bit, which makes the test optional, is to speed up calls and returns between domains of the operating system.

unique_id A unique identifier.

7.4.7. Restriction Set Storage

We are assuming that restriction sets are stored adjacent to the bases of page tables. Thus, the address of the restriction set control word of a segment's restriction set is page_table_addr - 1. This assumes a paged segmented virtual memory, but this is not essential. In an unpaged implementation, the restriction set would be located just before the segment itself (from the point of view of the physical memory allocation mechanism). In such an implementation, page_table_addr would be replaced by a segment_base_address, and length would be the length of the segment.

Figure 7-4 shows the storage format of a page table and a restriction set.

7.5. Strategies

In this section we present strategies for handling the events presented in section 7.3. Recall that P means any process being evolved by a PU.

1a) P reads from a segment. (Let A be that segment.)

Provided the reading is permitted by the mode test and the length test, and provided the segment has a restriction set, and provided the segment's U_read bit is on, the PU will tell the PRP to do the union operation $R_P = R_P \cup R_A$, and the PU waits to see if R_P was expanded. The PU identifies A to the PRP by telling the PRP A's page_table_addr. The PU also tells the PRP the bit w(mode3) for the segment A, just in case the

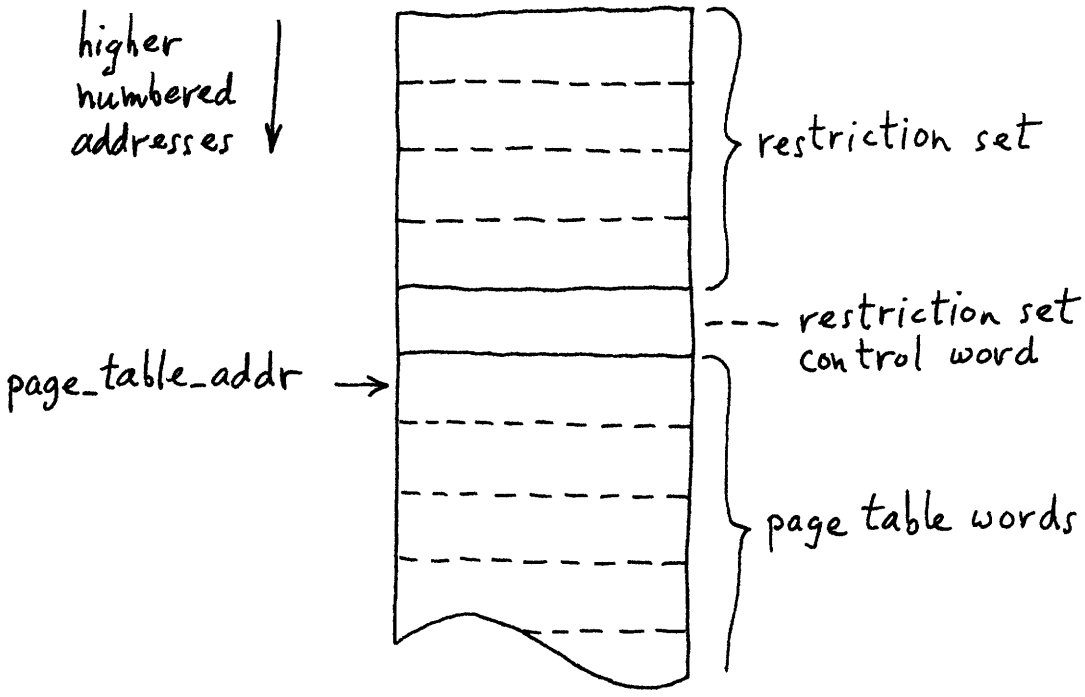


Figure 7-4. Restriction Set storage format.

PRP is not holding R_A when this event occurs. The reason why the PRP needs to know the value of w(mode3) will be given in the description of event 1c.

Whenever the PU starts to tell the PRP to do $R_P = R_P \cup R_A$ or $R_A = R_A \cup R_P$, the PU will wait for the PRP to finish its previous operation, and if the PRP is signalling for a fault (e.g. restriction overflow), the PU faults.

The PRP responds to the request to do the union operation $R_P = R_P \cup R_A$ by first looking in its associative memory for a word representing R_A. If none is found, the PRP picks up the restriction set R_A from main memory, as described in event 1c. The PRP uses its Union Processor to compute $R_P \cup R_A$, and replaces R_P with this result. The PRP tells the PU whether this operation expanded R_P.

1b) P writes into a segment. (Let A be that segment.)

Provided the writing is permitted by the mode test, the length test, the has_R_set bit test, and the U_write bit test, the PU will tell the PRP to do the union operation $R_A = R_A \cup R_P$. The PU identifies A to the PRP by telling the PRP A's page_table_addr. The PU also tells the PRP the bit w(mode3), which will be 1, just in case the PRP is not holding R_A when this event occurs.

The second paragraph under event 1a applies to this event also.

The PRP responds to the request to do the union operation $R_A = R_A \cup R_P$ by first looking in its associative memory for a

word representing R_A. If none is found, the PRP picks up the restriction set R_A from main memory, as described in event 1c. The PRP uses its Union Processor to compute $R_P \cup R_A$, and replaces R_A with this result (this modifies only the PRP's current copy of R_A), and sets the modified bit in the associative memory word representing R_A to 1 if this operation modified R_A.

1c) The PU evolving P picks up a segment capability.

The PU loads the segment capability into its associative memory. If the has_R_set bit is 1, the PU tells the PRP the page_table_addr of the segment, and the value of w(mode3) for the segment. The PRP examines its associative memory to see if it is holding the restriction set for the segment. If the PRP is holding the restriction set for the segment, it does not need to do anything. (*) If the PRP is not holding the restriction set for the segment, the PRP proceeds to pick up the restriction set from main memory. This action, i.e. picking up the restriction set from main memory, might also be taken in response to events 1a and 1b, as just described. If the available bit in the restriction set control word is 0, the PRP signals the PU to fault. Otherwise, the PRP picks up

(*) If the PRP is holding the restriction set when this event occurs, the PRP could compare the value of w(mode3) obtained from the PU with the writable bit for the restriction set, stored in the PRP associative memory word. If they differ, an operating system error is indicated and the PRP should tell the PU to fault. If w(mode3) = 0 and writable = 1, the writable segment has somehow obtained a non-writer process. If w(mode3) = 1 and writable = 0, the segment has somehow obtained a writer process without its restriction set, which must have been used previously by a reader process, first being flushed out of the PRP.

the restriction set from next to the page table and the PRP loads a word into its associative memory to represent the fetched restriction set. In that word, the modified bit is initialized to 0 and the writable bit is set to the value of w(mode3).

The PRP maintains LRU information for its associative memory, so when its restriction set storage is full and it has to pick up another restriction set, it replaces the least recently used restriction set in main memory, provided it is marked modified.

1d) The PU evolving P deletes a segment capability from its associative memory.

The PU tells the PRP the page_table_addr of the deleted capability, provided the has_R_set bit is 1. The PRP will replace the restriction set for that segment in main memory, provided it is marked modified. When the PU picks up a segment capability and has to delete a segment capability from its associative memory to make room for the new one, the PU first tells the PRP about the deletion, and then the PU tells the PRP about the new segment capability.

2) P does a send to output information. (Let A be that segment from which output is requested.)

There are three cases to consider, depending on whether the process invoking the send is a reader or a writer of A.

Case I. P is a reader of A. Since A has a reader process, it has no writer processes, and so R_A is unchanging. Using the copy of R_A stored next to the page table, and the

value of R_P stored in the PRP, the send primitive calculates $\bigcap_{r \in R_A \cup R_P} f(r)$ and decides whether to permit or to strike down the output. If there is any restriction $r \in R_A \cup R_P$ such that the principal to whom output is directed is not in f(r), then the restriction r strikes. (The calculation of $\bigcap_{r \in R_A \cup R_P} f(r)$ is detailed in the paragraph after the discussion of Case III.) If no restriction strikes, send starts the output. Then, either send returns to the calling procedure immediately or it waits for the output to be finished. If the return is immediate, the operating system will not allow A to have any writer processes until the output is finished. We are assuming that output occurs directly from A, without an intervening buffer.

Case II. P is a writer of A. So R_A might be in the PRP associated with the PU evolving P. The send primitive tells the PRP to flush out R_A, using a special instruction. The PRP responds by replacing R_A in main memory, if it is marked modified. Then send calculates $\bigcap_{r \in R_A \cup R_P} f(r)$ and decides whether to do the send or initiate the striking of some restriction. (See the paragraph on calculating $\bigcap_{r \in R_A \cup R_P} f(r)$.) If no restriction strikes, send starts the output and makes P wait. When the output process completes, it wakes up the process P, and send directs P to return.

Case III. P is neither a reader nor a writer of A. Output is not permitted in this case.

Now we describe the calculation of $\bigcap_{r \in R_A \cup R_P} f(r)$. This computation is complicated by the possibility that restrictions

in R_A or R_P stand for larger sets of restrictions, indicated by the condition oversize = 1. These larger sets of restrictions will have been stored in the Oversize Restriction Table (ORT), an operating system per-system data base, upon the occurrence of a restriction overflow. We will call a restriction for which oversize = 0, an ordinary restriction. Each oversize restriction is the root of a tree, defined by the ORT, whose terminal branches are ordinary restrictions. So the first step in calculating $\bigcap_{r \in R_A \cup R_P} f(r)$ is to represent $R_A \cup R_P$ as a set of ordinary restrictions, by following out the trees in ORT for the oversize restrictions. Then, for each of these ordinary restrictions, its address in the naming hierarchy (a pair (directory,entry)) is determined from the Master Restriction Table (MRT), another per-system data base. Using these addresses, the value f(r) is obtained for each r in the (possibly expanded) $R_A \cup R_P$. From these f(r)'s the intersection $\bigcap f(r)$ is calculated.

3) The PRP does a union operation that produces an oversize restriction set. (Restriction overflow.)

The normal size for a restriction set is between 0 and 4 restrictions. When the PRP does a union operation, the result could be as big as 8 restrictions. Whenever the result of a union operation is larger than 4 restrictions, the PRP signals the PU to fault. The PRP has a result register which will hold 8 restrictions, so nothing is lost but time. The fault takes the PU into the PRP domain of the operating system,

which tells the PRP to store the oversize restriction set it generated in the ORT, using a special instruction. The PRP domain associates a newly created restriction, for which oversize = 1, with the oversize restriction set stored in the ORT. Then the PRP domain replaces the oversize restriction set in the PRP's result register with the newly created restriction just introduced, using a special instruction. Finally, the PRP domain returns from the fault.

4) The process synchronization primitives.

The strategies are designed to allow restriction sets to remain in the PRP as long as possible, so that time spent by the system loading restriction sets from main memory into the PRP is minimized. Thus, restriction sets of segments with reader processes are allowed to remain in the PRP upon the occurrence of the process exchange event (event 7) and the call-domain and return-domain events (event class 5), and are only flushed out when the last reader process does a terminate-read. Restriction sets of segments with a writer process are marked writable in the PRP associative memory word. Such restriction sets are allowed to remain in the PRP upon call-domain and return-domain events in order to speed access of processes having write access to a segment in several domains. However, they (the "writable" restriction sets) are flushed out of the PRP by the process exchange event in order to guarantee that at most one PRP holds a copy of the potentially modifiable restriction set.

4a) P does an initiate-read.

No special action is required beyond that specified in section 6.8.

4b) P does a terminate-read. (Let A be the segment being terminated.)

If P is the last remembered reader of A, the terminate-read primitive uses the central communicator to tell every PRP in the system to flush out R_A. Since A has no writer process, none of these copies of R_A will be marked modified and so the PRPs will simply delete them. This action is taken in anticipation of A's getting a writer process: if such a process writes A, expanding R_A, and then gets rescheduled onto a PU whose PRP holds the old R_A, the restrictions newly added to R_A would not be properly propagated.

4c) P does an initiate-execute.

No special action is required beyond that specified in section 6.8.

4d) P does a terminate-execute.

The same action as specified for event 4b is performed.

4e) P does an initiate-write.

No special action is required beyond that specified in section 6.8.

4f) P does a terminate-write. (Let A be the segment being terminated.)

The terminate-write primitive tells the PRP attached to the PU evolving P to flush out its copy of R_A, using a special instruction. The PRP will replace R_A in main memory, provided

it is marked modified; and the PRP will delete its associative memory word representing R_A. As in the case of event 4b, this action is taken in anticipation of A getting a new writer process. Only the PRP attached to the PU evolving P will be holding a copy of R_A because R_A is marked writable, because writable segments have exactly one writer process, and because the process exchange event (event 7) causes the PRP to flush all restriction sets marked writable.

5a) P does a call-domain.

If the R_check bit of the specified domain entry capability is 1, the PU takes a fault into the PRP domain. The PRP domain computes $\bigcap_{r \in R_P} d(r)$, following out oversize restriction trees as necessary, and decides whether P will be allowed to proceed into the target domain, or whether to strike down the call. The PRP domain obtains the current value of R_P from the PRP, using a special instruction. If and when the PRP domain returns from the fault, the PU allows P to become bound to the target domain. The PU stores the value of the R_check bit with the return address of the call in the sectioned stack.

The PU clears its associative memory whenever the process it is evolving becomes bound to a new domain, because segment numbers have different meanings in different domains.

The reason for allowing the computation of $\bigcap d(r)$ to be skipped is to provide speedy calls and returns between domains of the operating system.

5b) P does a return-domain.

If the R_check bit stored with the return address in the sectioned stack is 1, the PU takes a fault into the PRP domain. As in event 5a, the PRP domain decides whether P will be permitted to proceed into the target domain, or whether to strike down the return; based on whether the target domain is in $\bigcap_{r \in R_P} d(r)$.

NB: It is ironic that the fast inter-domain call and return sought in chapter 4 must be slowed down by this necessary check. But note that the cost of performing this check can be paid by the owners of the restrictions in R_p. If this is done, the restriction owners are paying for the protection of their information, rather than the owner of P paying for that protection; but this would allow malicious users to pile up costs for restriction owners. To alleviate this effect, the owner of P and the restriction owners might share the costs.

6a) P does a create-r.

The create-r primitive makes an entry in the Master Restriction Table for the newly created restriction. Note that whenever a restriction entry is renamed or moved from one directory to another, the MRT must be updated.

6b) P does a delete-r.

The restriction is marked deleted in the MRT. The unique_id which represents the deleted restriction cannot be reused unless it is garbage-collected out of every restriction set in the computer system.

6c) P does a read-r.

No special action is required beyond that specified in section 6.9.

6d) P does a modify-r.

No special action is required beyond that specified in section 6.9.

6e) P does a place-r. (Let A be the segment named by seg_name.)

If the named segment is not active (i.e., does not have a page table), the computer contains only one copy of its restriction set and the place-r primitive is easy to do. It's more interesting when other processes are reading or writing the segment. Typically P is neither a reader nor a writer of the segment, but we give P priority over readers and writers because doing a place-r is quick and so the disruption to readers and writers is minimal. The place-r primitive turns off the available bit in the restriction set control word for R_A, and then place-r uses the central communicator to flush out R_A from all PRPs. The PRPs respond by replacing R_A in main memory if they are holding it and it is marked modified, and in any case by deleting their associative memory words for R_A. After receiving acknowledge from all the PRPs signalled by the central communicator, place-r has access to the only copy of R_A in the computer. The place-r primitive then adds the restriction named by its arguments (directory,entry) to R_A, and does oversize processing if R_A overflows. Then place-r turns the

R_A available bit back on, and wakes up all the processes that needed R_A and faulted because of the available bit being off.

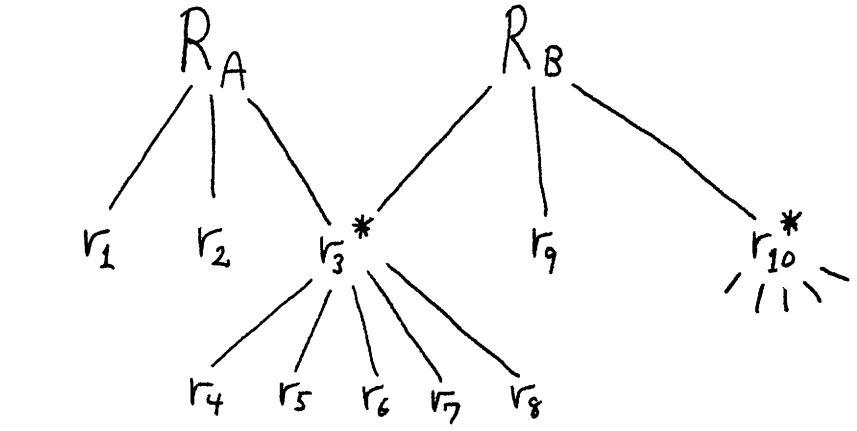
6f) P does a lift-r. (Let A be the segment named by seg_name.)

This is similar to place-r, except that a restriction is being removed from R_A. The priority of P over readers and writers of A is the same, as is the use of the available bit and the central communicator. A slight complication arises when R_A includes oversize restriction sets and the restriction to be removed is not in the topmost level of R_A. Figure 7-5(a) shows an R_A which contains two ordinary restrictions r_1 and r_2 and the oversize restriction r_3^*. (The "*" simply means that r_3 is oversize.) Suppose that lift-r is told to lift r_4 from R_A. It would not be correct to remove r_4 from the definition of r_3^*, because this would have the effect of removing r_4 from other restriction sets containing r_3^*, such as R_B in figure 7-5(a). The lift-r primitive must make an isolated copy of R_A, and remove r_4 from that. Figure 7-5(b) shows the isolated copy of R_A implemented as a new oversize restriction r_{11}^*.

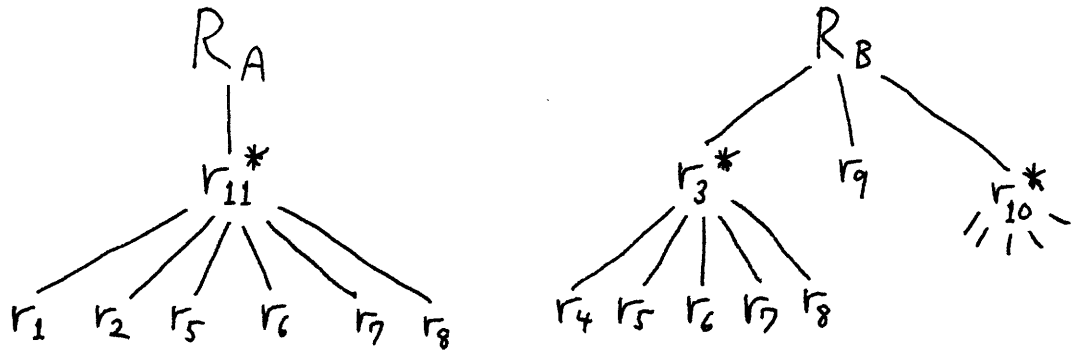
6g) P does a replace-r.

This is simply a lift-r followed by a place-r, performed without turning the restriction set available bit on between the end of the lift-r and the beginning of the place-r.

The following two events, invocations of the primitives place-s and lift-s, differ from invocations of the primitives place-r and lift-r in the following ways: the invoking process must be a writer process of the referenced segment, and



(a)



(b)

Figure 7-5. Effect of lift-r primitive on oversize restriction sets.

that segment is specified with a segment number rather than a tree name. For details, see section 6.9.

6h) P does a place-s. (Let A be the segment numbered seg#.)

Since P must be a writer process of A, place-s uses a special instruction to flush R_A out of the PRP attached to the PU evolving P, whereupon place-s has access to the only copy of R_A in the computer. The primitive place-s then adds the restriction named by its arguments (directory,entry) to R_A, and does oversize processing if R_A overflows.

6i) P does a lift-s. (Let A be the segment numbered seg#.)

Since P must be a writer process of A, lift-s uses a special instruction to flush R_A out of the PRP, as above in event 6h. The primitive lift-s then removes the restriction named by its arguments from R_A, taking account of oversize restriction sets as described above for event 6f.

6j) P does a place-p.

The primitive uses a special instruction to copy out R_P from the PRP attached to the PU evolving P, adds the restriction named by its arguments to R_P, does oversize processing if R_P overflows, uses another special instruction to load R_P back into the PRP, and uses a third special instruction to set all the U_write bits to 1 in the associative memory of the PU evolving P.

6k) P does a lift-p.

This primitive is similar to place-p, except that a re-

striction is removed from R_p, which requires taking account of oversize restriction sets as described above for event 6f; and the lift-p primitive finishes by using a special instruction to set all the U_read bits to 1 in the associative memory of the PU evolving P.

7) The operating system reassigns a PU to a new process.

This event begins with an interrupt which takes the PU to be reassigned into the traffic controller domain of the operating system. None of the segments in the traffic controller domain have restriction sets, so the restriction sets stored in the PRP attached to the PU are unchanging. The traffic controller makes a copy in its data base of the process state of the process being interrupted. The traffic controller uses a special instruction to copy out R_p from the PRP attached to the PU, and the traffic controller stores this R_p associated with the process state of P. Then the traffic controller uses a special instruction to tell the PRP to flush out all restriction sets of writable segments. The PRP responds by replacing in main memory all such restriction sets which are marked modified; and by deleting its copies of such restriction sets whether marked modified or not. The traffic controller picks another process to run, loads its R_p into the PRP with a special instruction, loads its process state into the PU, and lets it continue its work. The loading of the new process state into the PU terminates this operation, and the new process returns out of the traffic controller domain.

8) A PU takes a fault.

Control of the PU passes to an operating system domain where the fault is handled (except for user-domain faults, which cause control to be passed to the program which signals the condition associated with the fault (*)). We restrict our attention here to the segment fault, because the response to a segment fault is to make the segment involved active, which may require that some other segment be made inactive. When a segment, say segment A, must be made inactive, the following things are done. First, every segment capability for A is modified by setting its fault bit to 1. Then the central communicator is used to tell every PU to clear out of its associative memory any word with the page_table_addr of A. The central communicator also tells every PRP to flush out R_A, if it has a copy of it. The PRPs respond by replacing R_A in main memory if it is marked modified, and by deleting copies of R_A whether marked modified or not. After receiving acknowledge from all the PRPs and PUs signalled by the central communicator, A's restriction set and page table can be removed from the Active Segment Table to make room for the page table and restriction set of the new segment.

9) A PU takes an interrupt.

The PU stores the process state of the process it was

(*) When there is no handler enabled for the signalled condition in the domain where the condition occurred, it is not clear what should be done. This is one of the problems which must be solved by the operating system's control structure, which is outside the scope of this thesis.

evolving someplace, and picks up the process state of an interrupt-handler process, which has the effect of taking the PU into the operating system domain that handles the interrupt, and the PU clears its associative memory because it is moving between domains. We are assuming that none of the segments in the operating system domain that handles the interrupt will have a restriction set. So while the PU is evolving the interrupt-handler process, the PRP sits and does nothing. When the interrupt handling is done, the PU picks up the process state of the interrupted process, clears its associative memory, and continues evolving the interrupted process.

10a) The operating system starts up.

While the operating system is starting up, one PU is doing the work of startup while the other PUs wait. These other PUs will be brought into the system by means of reconfiguration events (event 10b). So we need only consider the PU doing the startup. It begins its task using an absolute mode of addressing memory, and shifts over to a segmented addressing mode only after initializing the mechanisms which support the system's segmented virtual memory. While the PU is using the absolute mode of addressing memory, the attached PRP sits and does nothing. When the PU shifts over to the segmented addressing mode, the attached PRP clears its associative memory and sets its register for holding R_p to $\emptyset$. Subsequent use by the PU of segments which have restriction sets, as indicated by the has_R_set bit of segment capabilities, will result

in union operations being performed by the PRP.

10b) The operating system reconfigures a PU into, or out of, the system.

The first process to run on a PU being configured in, and the last process to run on a PU being configured out, is that PU's idle process. Idle processes are usually bound to the idle process domain of the operating system. (This is where PUs being configured in have their security hardware tested before being put to work.) The idle process domain uses a special instruction to clear out the PRPs attached to PUs being configured in and out. A PRP responds to this instruction by clearing its associative memory and its register holding R_p to $\emptyset$.

10c) The operating system shuts down.

If we assume that shutdown is terminated by making the last working PU evolve an idle process, we can be sure that that idle process will clear that PU's PRP.

11a) Segments are backed up by a "daemon" process under the control of the operating system.

The backup daemon is given a segment capability for the segment to be backed up, and the has_R_set bit in the capability is 0. In this way the daemon avoids acquiring all the restrictions associated with the segments on which it works. When the backup daemon backs up a segment, it also copies the restriction set of the segment onto the backup media. Therefore the backup daemon has the privilege of obtaining copies of segments' restriction sets.

11b) Segments are retrieved from backup media.

The retrieval daemon deletes the on-line copy of the damaged segment, and its restriction set; and replaces these with the segment contents and restriction set retrieved from the backup media.

7.6. Tactics

Here are the elementary operations which the PRP must perform to effectively support the strategies of the previous section.

1) The PRP must accept information from the attached PU about where restriction sets are located. The PU initiates this when it picks up a segment capability.

2) The PRP must pick up restriction sets from main memory. The PRP must request its attached PU to fault if a restriction set's available bit is 0.

3) The PRP must replace restriction sets in main memory. This must be done in response to:

a) deletion of a segment capability from the PU associative memory.

b) need for space in the PRP restriction set memory. The least recently used restriction set is selected.

c) a special instruction, which specifies the segment's page_table_addr.

d) a request from the central communicator. Two kinds of requests will get this response:

d1) a request to flush a restriction set, specified by page_table_addr.

d2) a request to flush a segment capability from all PUs' associative memories, the segment being specified as before by page_table_addr.

e) a special instruction which causes all writable restriction sets to be flushed from the PRP.

4) The PRP must clear itself to null contents in response to a special instruction.

5) The PRP must do union operations $R_P = R_P \cup R_A$ and $R_A = R_A \cup R_P$ in response to requests from the PU. The PRP must request the PU to fault when the result of a union operation is too big.

6) The PRP must put down R_P in, and pick up R_P from, main memory in response to special instructions.

7) The PRP must put down its result register in, and pick up a new result register contents from, main memory in response to special instructions. (This is for handling restriction overflow.)

[8) The PU must set all the U_read bits in its associative memory to 1, or set all the U_write bits to 1, in response to special instructions. This tactic is mentioned here for completeness.]

The elementary operations listed above are the only operations which the PRP is required to perform. It is obvious that a digital electronic device which performs the given elementary operations can be constructed.

7.7. Summary

We have described the hardware Privacy Restriction Processor, and the hardware and software strategies and tactics which permit a multiprocessing computer system to implement the system of privacy restrictions developed in chapter 6. The PRP consists of an associative memory which records what restriction sets the PRP is holding, a location-addressed restriction set memory, a Union Processor for forming unions of restriction sets, memory for the restriction set of the process being evolved by the PU the PRP is attached to, and control logic which co-ordinates the parts just enumerated to accomplish the actions (tactics) described in section 7.6.

Chapter Eight

Authority Hierarchies

8.1. Introduction

The purpose of this chapter is to describe means by which the administrators of a computer utility can be prevented from having absolute power over all the information stored in and processed by that utility. In chapter 2, we noted the requirement that noone should have such great power over society's information systems. But in chapter 4 we introduced a hierarchically controlled naming hierarchy, similar to the file hierarchy of Multics, whose hierarchical control mechanism gives the administrators who control the Root the power to extend their control to include every computing object named by the naming hierarchy. A naming hierarchy with a built-in monocratic authority structure is not adequate in social contexts where several independent authorities share a single computer utility, because the several authorities would be risking loss of independence.

The monocratic authority structure of the naming hierarchy introduced in chapter 4 arises from the nature of control over directories. Control over directories is exercised by means of operating system primitives which examine and modify directories; and control over directories is authorized by access control packets (acps), which name the domains which may successfully invoke the primitives which examine and modify the directories. Because the naming hierarchy is a tree of direc-

tories, every directory except the Root is an entry of a superior directory, which provides a place for storage of directories' access control packets. In other words, a given directory's acp is stored as a part of its entry in the superior directory in the naming hierarchy, and the acp of the Root, since the Root is the most-superior directory, is stored in some unique place (but not in any directory).

The authority structure of the naming hierarchy is monocratic because the right to modify a directory carries with it the right to modify access control packets in that directory. For example, figure 8-1 shows how users are organized into projects and given directories to serve as "personal Roots" from which to build naming sub-hierarchies. The directory (users,MultLab,Rotenberg) is Rotenberg's personal Root, to be used as the starting point in catalogueing the computing objects created by Rotenberg. The user Rotenberg's control over this directory is authorized by the acp which gives the domain (users,MultLab,Rotenberg,home) read ("r") and modify ("m") access to the directory. But because of the monocratic authority structure, Rotenberg cannot be sure that he will retain exclusive control over his directory. The administrator of the MultLab project, Clark, can obtain control of Rotenberg's personal Root by simply requesting the operating system to modify the acp of (users,MultLab,Rotenberg). This request will be honored provided it comes from Clark's home domain, (users, MultLab,Clark,home), because that domain has the right to mod-

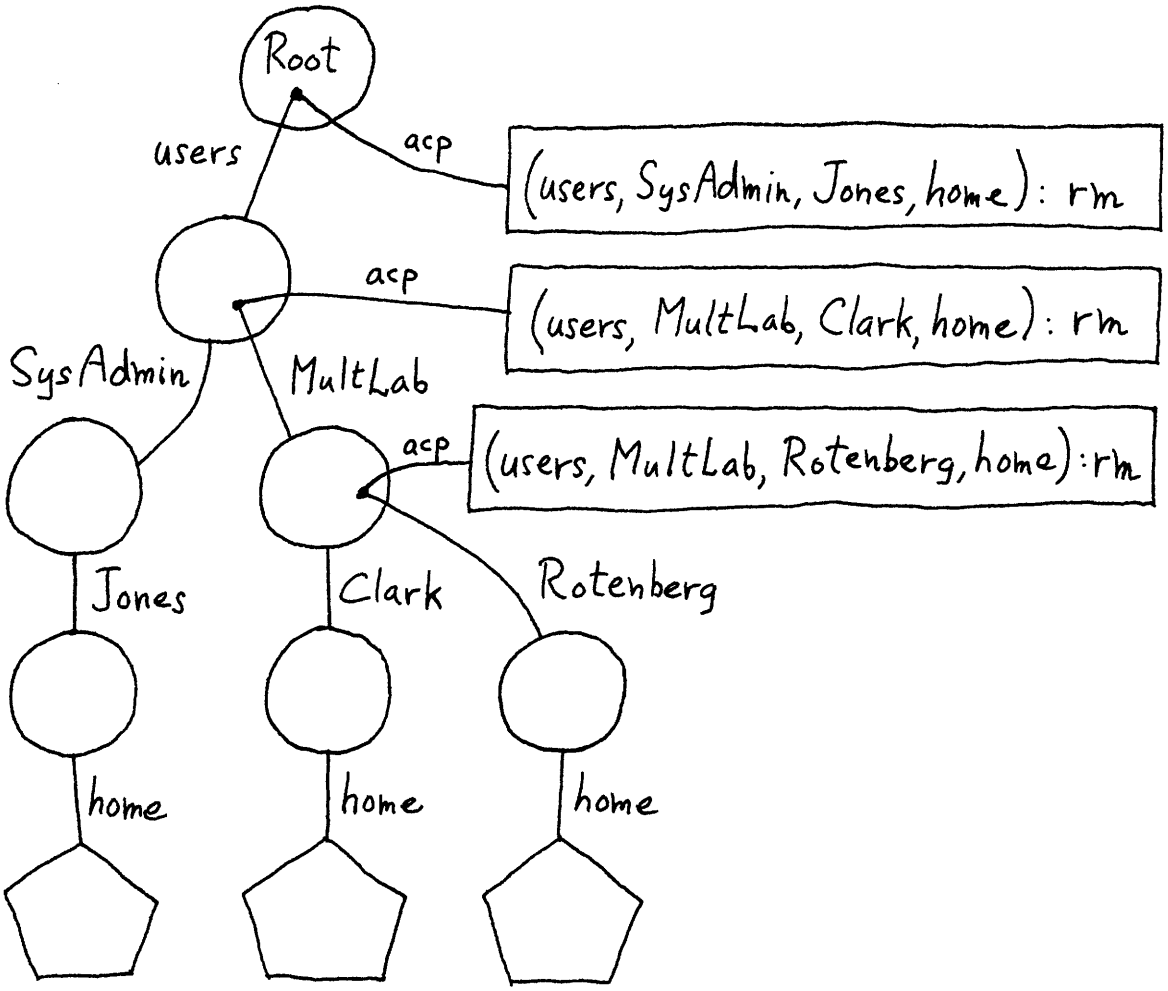


Figure 8-1. Naming hierarchy with monocratic authority.

ify the directory (users,MultLab), which includes the right to modify all the access control packets in (users,MultLab). Similarly, the system administrator Jones can obtain control of the directory (users,MultLab) because her home domain is authorized to modify the directory (users), and having done that, Jones can then obtain control of (users,MultLab,Rotenberg). Figure 8-2 shows how the access control packets of figure 8-1 would be modified by the authoritarian actions just described.

This chapter will formulate a solution to the problem of monocratic authority in a computer utility, but it is important to recognize that the solution should not interfere with the reasonable exercise of rational authority. We will not propose that Jones' control over the directory (users) be eliminated, because such control is required to add new projects to the utility, and to delete old ones. Similarly, Clark needs control over the directory (users,MultLab) to add users to the MultLab project, and to delete old ones. Our solution introduces two new computing objects, offices and authority hierarchies, for the purpose of controlling changes to access control packets.

8.2. Authority Hierarchies

In place of the monocratic authority structure of the naming hierarchy just described, we propose that a computer utility should contain independent authority-expressing mechanisms for independent authorities in society. Corporations

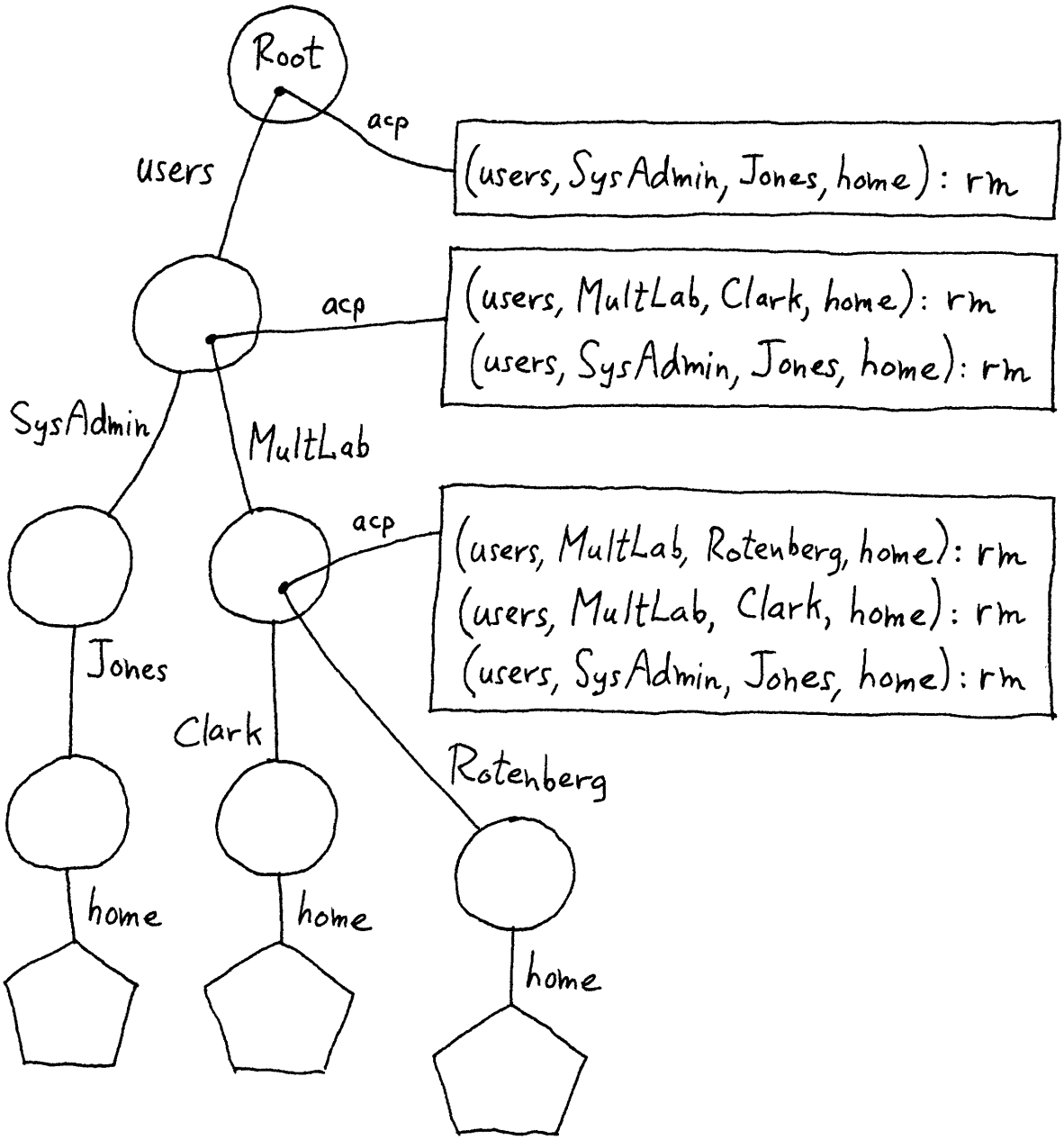


Figure 8-2. Naming hierarchy with monocratic authority exercised.

and branches of governments are examples of independent authorities in society. The structure in the computer utility which represents and serves an independent authority in society is the authority hierarchy. Each authority hierarchy consists of a tree of offices. Each office contains a list of entries (possibly an empty list), and each entry points to another office of the authority hierarchy. Each entry of an office has an associated name (a character string) which serves to uniquely identify the entry in the context of the containing office. One office of each authority hierarchy, called the most-superior office, is not pointed to by any entry of any office. Each authority hierarchy has a unique hierarchy name (a character string) which serves to distinguish each authority hierarchy from all other authority hierarchies in the computer utility. (The name space of hierarchy names is administered by the computer utility administrators, subject to the constraint that they cannot rename an authority hierarchy. The reason for this constraint will become evident presently.) Every office in an authority hierarchy can be named by a hierarchy tree name, which is the sequence of office entry names that defines a path through the tree of offices from the most-superior office to the office being named. An arbitrary office can be specified with an office name, which is an ordered pair consisting of a hierarchy name and a hierarchy tree name.

The purpose of each office is to authorize changes to

some collection of entries of directories. The changes authorized by offices include modifications to access control packets, renaming of directory entries, and deletion of directory entries. These three types of changes were previously authorized by the "modify" mode of access to directories. With the introduction of offices and authority hierarchies, we are eliminating the mode "modify" and replacing it with the mode "append", which gives the right to add new entries to directories. This "append" right is all that remains of the previously defined "modify" right.

In order to record which offices control which directory entries, each entry of a directory contains the office name of some office. Requests to modify directory entries are made by processes invoking operating system primitives. Each office contains an agent list which the operating system consults when it receives requests to modify entries of directories. The agent list is a list of domain names. A request to modify an entry of a directory will be honored if the request is made by a process calling from a domain D such that the name of D is an element of the agent list of the office named by the office name contained in the directory entry whose modification was requested. Thus, when a domain D is an agent of an office O, requests that originate in D can exercise the powers of office O. In most of the examples in this chapter, the domains named by agent lists are the home domains of users, i.e. persons. Since users command processes bound to their

home domains, users can exercise the powers of offices.

The effect of introducing authority hierarchies to our model of a computer utility is to make control of directory entries, and especially of access control packets, fundamentally different from control of all other computing objects (which is the function of access control packets). Adding a new mechanism makes the model more complex, but it allows the model to express an important social relationship: the interaction between the exercise of power and the status quo. The acps represent the status quo. They tell what access is permitted by what domains to each computing object. To change an acp is to exercise a right of control over the computing object whose acp is changed, and this exercise of a right is the fundamental unit of power over the computer utility. Because computers are becoming the nervous system of our society, it is important to know who has power over computers. The authority hierarchies provide would-be observers of the social scene with a structured expression which tells who has power in the computer utility.

Figure 8-3 illustrates how the naming hierarchy of figure 8-1 can be placed under the control of two authority hierarchies, named SysAdmin and MAC, shown in part 2 of the figure. (The reader should note that the names "SysAdmin" and "MAC" are being used for two purposes in figure 8-3: to name authority hierarchies, and to name entries in the directory (users). It is not difficult to tell from the context of each in-

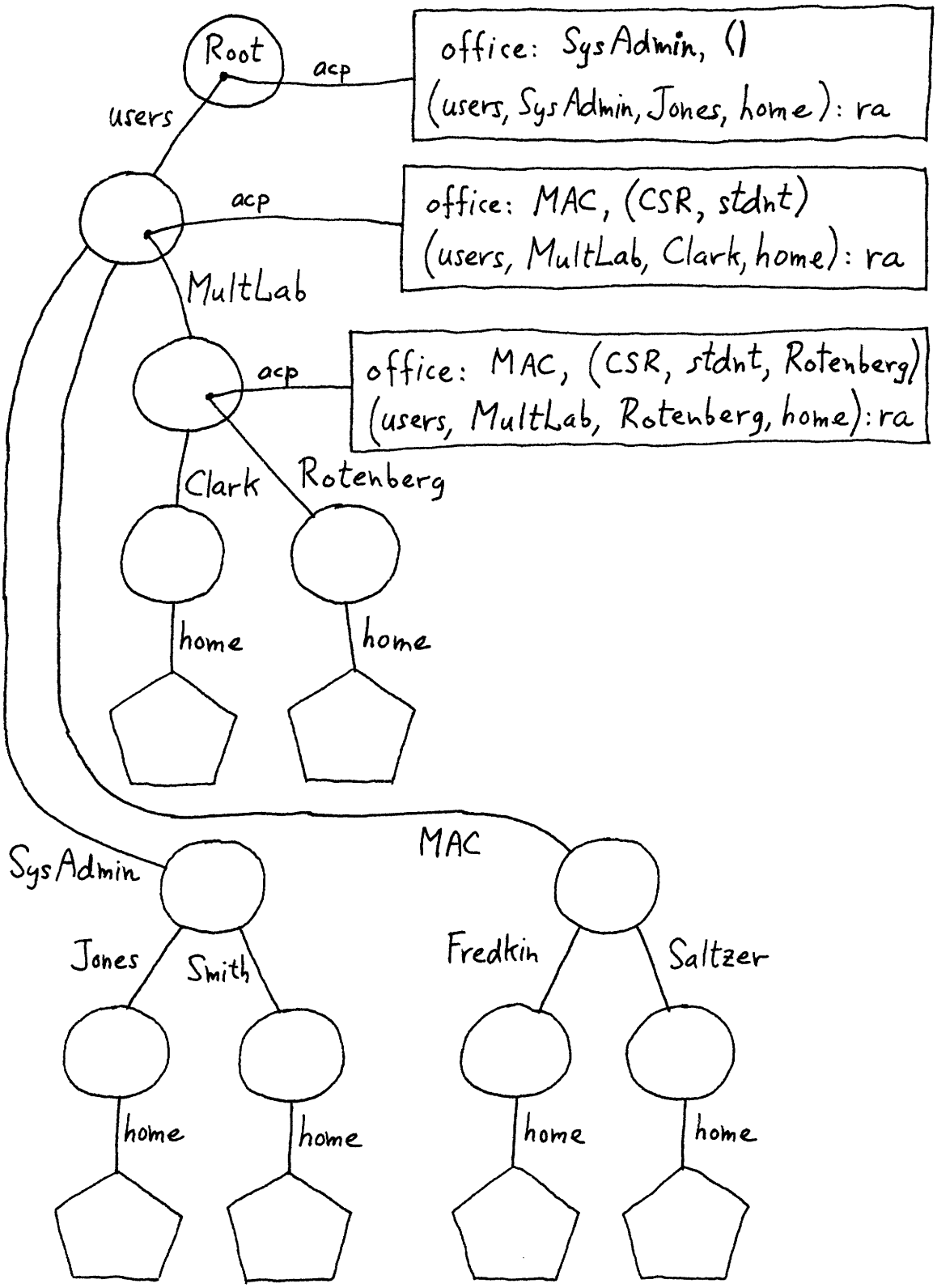


Figure 8-3, part 1. Parts of the naming hierarchy controlled by two authority hierarchies.

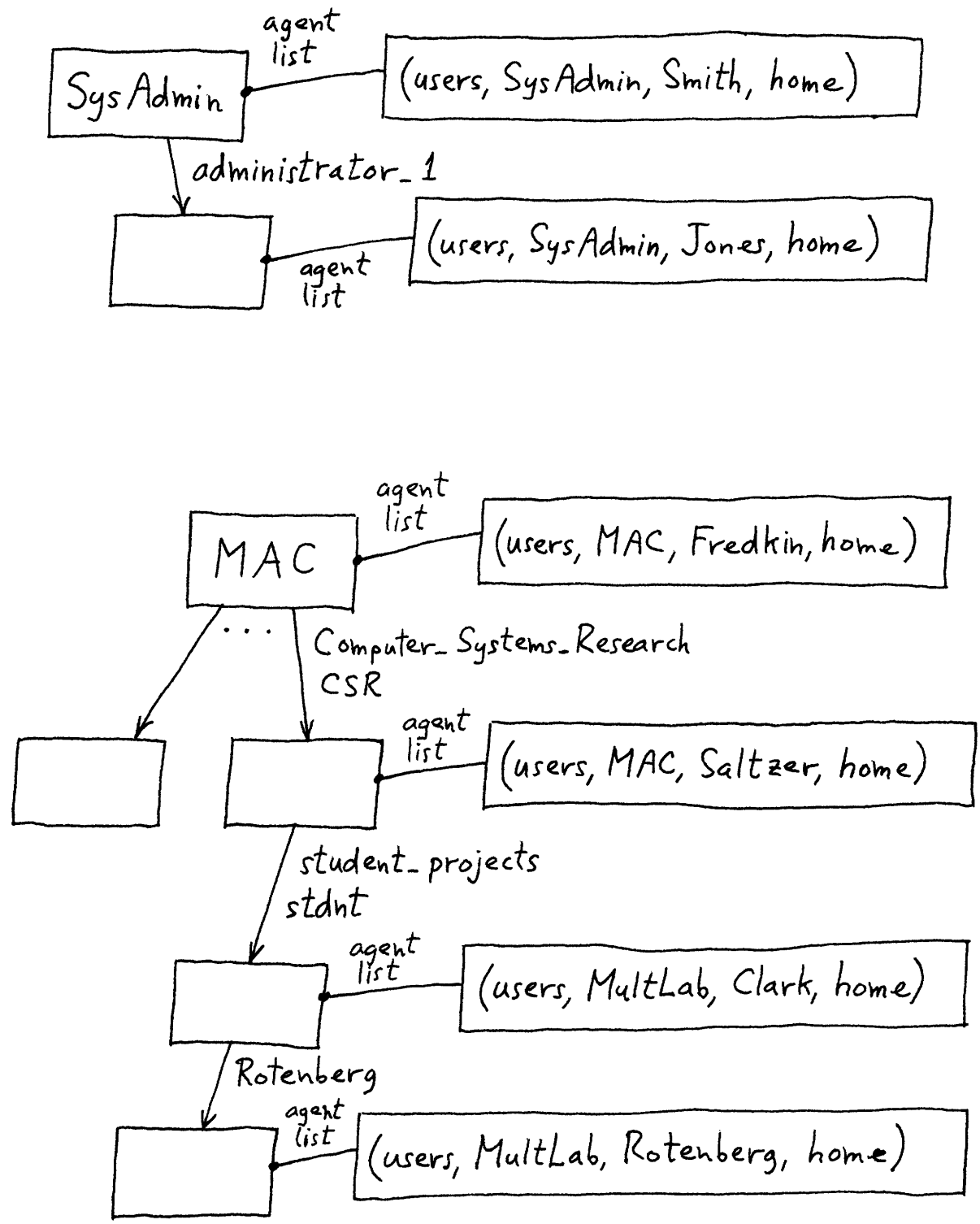


Figure 8-3, part 2. Two independent authority hierarchies.

stance of these names which meaning is intended.) In figure 8-3, part 1, and in other figures in this chapter, the office name which tells which office authorizes changes to an entry of a directory is shown as part of the access control packet associated with the entry. For example, the acp of the directory (users) in figure 8-3 names the office "SysAdmin,()". The empty list in the office name denotes the most-superior office of the SysAdmin authority hierarchy. Thus the office (SysAdmin,()) controls the directory (users). When we say, "controls the directory", we mean that the given office controls changes to the directory entry which points to the given directory. This includes control over changes to the acp of the given directory, plus control over renaming and deleting the entry which points to the given directory.

Figure 8-3, part 2, shows the two authority hierarchies SysAdmin and MAC. The MAC authority hierarchy consists of many offices, of which four are shown in detail. The office (MAC,(CSR,stdnt,Rotenberg)) controls the directory (users, MultLab,Rotenberg); while the office (MAC,(CSR,stdnt)) controls the directory (users,MultLab). In the MAC authority hierarchy, we are using abbreviations for the names of entries in offices. "CSR" is an abbreviation for "Computer_Systems_ Research" and "stdnt" is an abbreviation for "student_projects".

Figure 8-3 is an example of a non-monocratic authority structure. To see this, consider what would happen if the user Smith, a member of the SysAdmin project, attempted to

gain control of the directory (users,MultLab). To do this, she would have to modify the acp of that directory, but she cannot modify it because she is not named by the agent list of the office (MAC,(CSR,stdnt)). To be more precise, she does not have control of a domain named by the agent list. Furthermore, she cannot modify any agent list of the MAC authority hierarchy unless she happens also to be one of the authority system's locksmiths. We will describe the locksmithing function in section 8.4.

In the situation just described, in which persons associated with one authority hierarchy are trying to access an object which is under the control of another authority hierarchy, it is easy to say what the desired outcome is: access should not be permitted unless authorized by the appropriate office of the controlling authority hierarchy. This answer is easy to see because it springs immediately from the underlying philosophy that authority hierarchies represent <u>independent</u> authorities. It is more difficult to say what should happen when a person's domain is named by the agent list of an office of a given authority hierarchy, and that person wants to access a computing object which is under the control of another office of the same authority hierarchy. For example, suppose Clark wanted to obtain control of the directory (users,MultLab, Rotenberg). We will describe protocols for the exercising of "higher" authority in section 8.3.

To complete the description of our example, we must say

how the acps shown in figure 8-3 were initialized. We will assume that the Root and the directory (users) already exist as this description begins. The first action of interest occurs as the system administrator Jones creates the MultLab directory. Because the acp of the directory (users) contains one term giving "read" and "append" access to Jones' home domain (whose name is (users, SysAdmin,Jones,home)), the request to create the directory (users,MultLab) must have come from (users,SysAdmin,Jones,home). The system's rule is that the domain which creates a computing object is given access to that object. Therefore the acp of the directory (users,MultLab) is initialized to contain a single term giving "read" and "append" access to (users,SysAdmin,Jones,home). But who should have the authority to modify this newly created acp? To provide an answer to this question, we introduce a new concept: the authority of a domain. The authority of a domain is specified by an office name, and it is this office name which is placed in newly created acps to control their subsequent modification. The authority of a domain is recorded in the acp of the domain, and for our example we will assume that the authority of (users,SysAdmin,Jones,home) is (SysAdmin,(administrator_1)). (When a domain is created, its authority is initialized to be identical to the authority of the creating domain.)

Figure 8-4 shows the naming hierarchy just after the directory (users,MultLab) is created. The system administrator Jones would continue by creating the directory (users,MultLab,

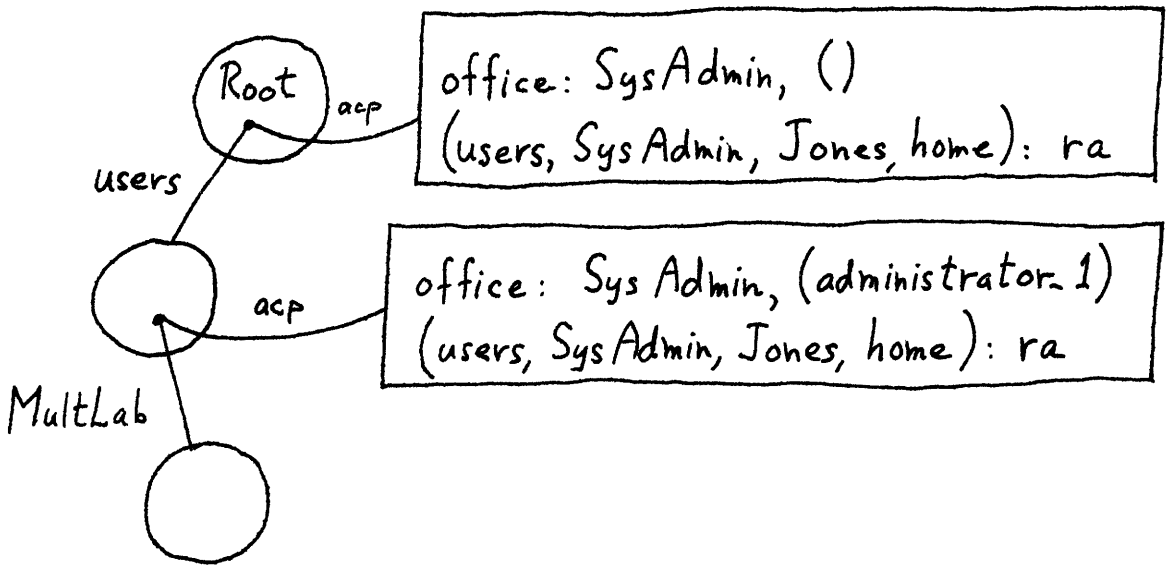


Figure 8-4. Naming hierarchy after creation of (users, MultLab).

Clark) and the domain (users,MultLab,Clark,home). We will assume that the user Clark has been registered with the utility's accounting subsystem, so that he can log in and command a process bound to the domain (users,MultLab,Clark,home), and we will assume that Saltzer has created the office (MAC,(CSR, stdnt)) and initialized its agent list to name the domain (users,MultLab,Clark,home).

The next action of interest occurs as the system administrator Jones passes control of the directory (users,MultLab) over to the office (MAC,(CSR,stdnt)). Jones accomplishes this by commanding her process to invoke an operating system primitive which will change the office name recorded in the acp. Like any other request to modify an acp, the request will be honored if the request is made from a domain named by the agent list of the office named by the office name contained in the acp whose modification is requested. In this case, the request is made by Jones' process calling from Jones' home domain, and so it is honored. Figure 8-5 shows the naming hierarchy just after the acp of (users,MultLab) is modified.

The next action of interest occurs as Clark, whose home domain is the agent of (MAC,(CSR,stdnt)), causes the deletion of the term "(users,SysAdmin,Jones,home): ra", from the acp of (users,MultLab). Once this is done, the directory (users, MultLab) cannot be accessed in any way by the system administrator Jones, or any other system administrator. Then Clark would command his process to add, to the acp of (users,MultLab

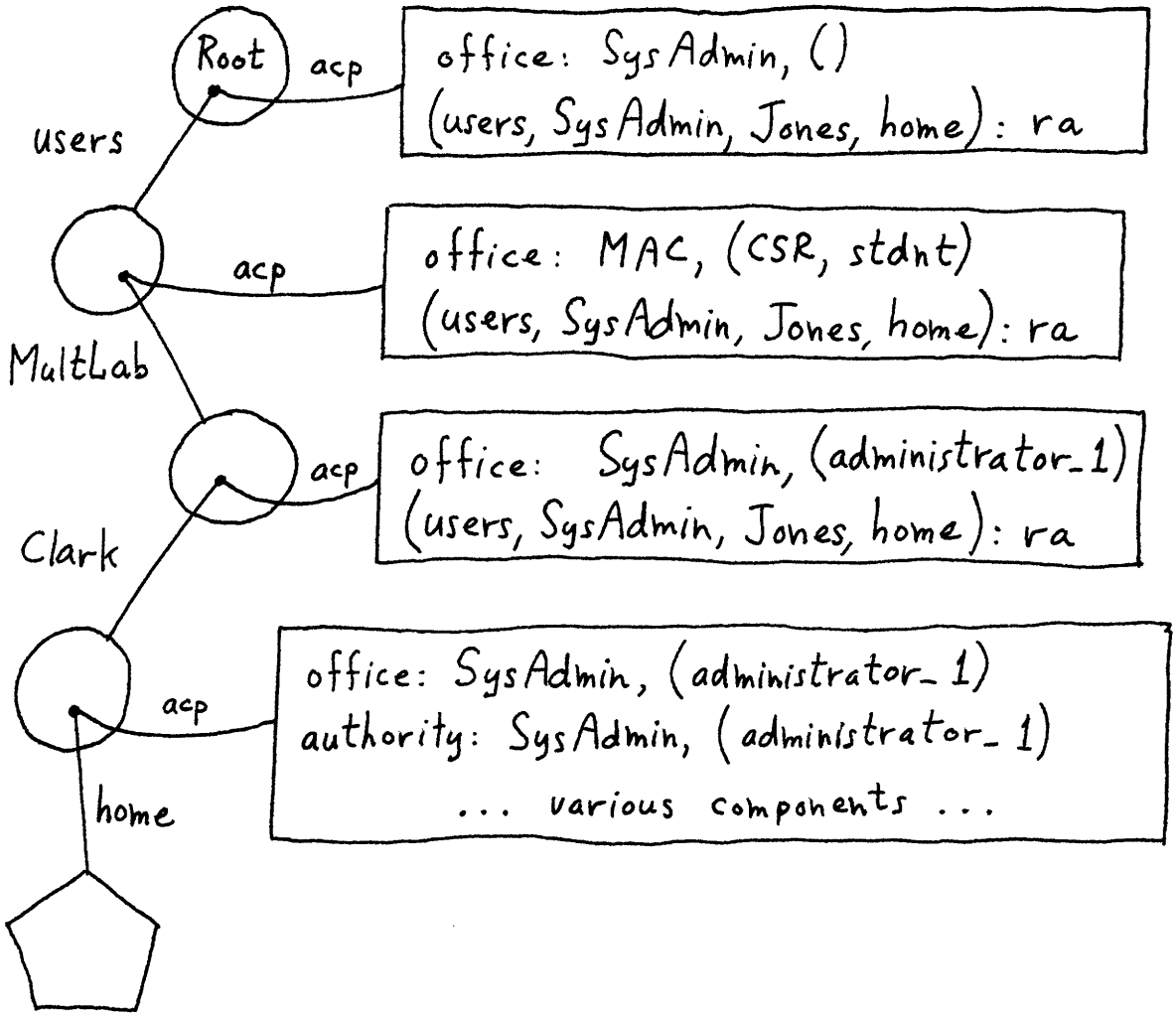


Figure 8-5. Naming hierarchy after modification of acp of (users, MultLab).

), the term "(users,MultLab,Clark,home): ra". Thus Clark establishes his control over the "MultLab directory".

Now the process just described for the directory (users, MultLab) is repeated for the directory (users,MultLab,Clark) and Clark's home domain, (users,MultLab,Clark,home). After Clark has control of the acp of (users,MultLab,Clark,home), he would change the authority of (users,MultLab,Clark,home) to (MAC,(CSR,stdnt)).

Finally, Clark would request an independent audit of the acps of (users,MultLab), (users,MultLab,Clark), and (users, MultLab,Clark,home); and an audit of the collection of segments, especially program segments, for which the domain (users,MultLab,Clark,home) contained capabilities. If Clark were to request this information from his own process bound to his home domain (users,MultLab,Clark,home), he could be fooled by a naming hierarchy simulator program placed in his home domain by the system administrator Jones (who created the home domain). Such a simulator program could pretend, for example, that (users,MultLab) was under the control of (MAC,(CSR,stdnt)) when it was really still under the control of (SysAdmin, (administrator_1)). Therefore an independent audit is necessary.

8.3. Higher Authority and Protocols

It is a common occurrence in organizations for officials to issue directions to, or make requests of, their subordinates. It might happen that a subordinate is requested to ex-

ercise some right of control over a computing object, e.g. to change an access control packet. For example, figure 8-6 shows an object whose acp can be modified by commands issued by person B, because B commands a process bound to a domain named by the agent list of the office (Auth,(sub)), which is the office named by the acp. Similarly, person A commands a process bound to a domain named by the agent list of the office (Auth,()). We say that A is an agent of (Auth,()). B is an agent of (Auth,(sub)). Now suppose that A is B's superior in the organization represented by Auth. A could ask B to change the acp of the object, and B could object and refuse to comply until the side effects of the change, which B ought to be expert about because he is an agent of (Auth,(sub)), are understood. If A had the power to change the acp himself, he might go ahead and do it with insufficient thought to the side effects. Yet in some organizations, superiors will require this power. The problem for a computer utility is to implement an authority hierarchy mechanism which will serve organizations having disparate traditional behavior patterns in their superior-subordinate relationships.

A protocol is a rule which the computer follows in responding to the request of a higher authority. We are assuming that the computer will infer the authority relationships between people by noting who is an agent of what office. For example, the computer would infer that A is B's superior because A is an agent of an office which is superior to the off-

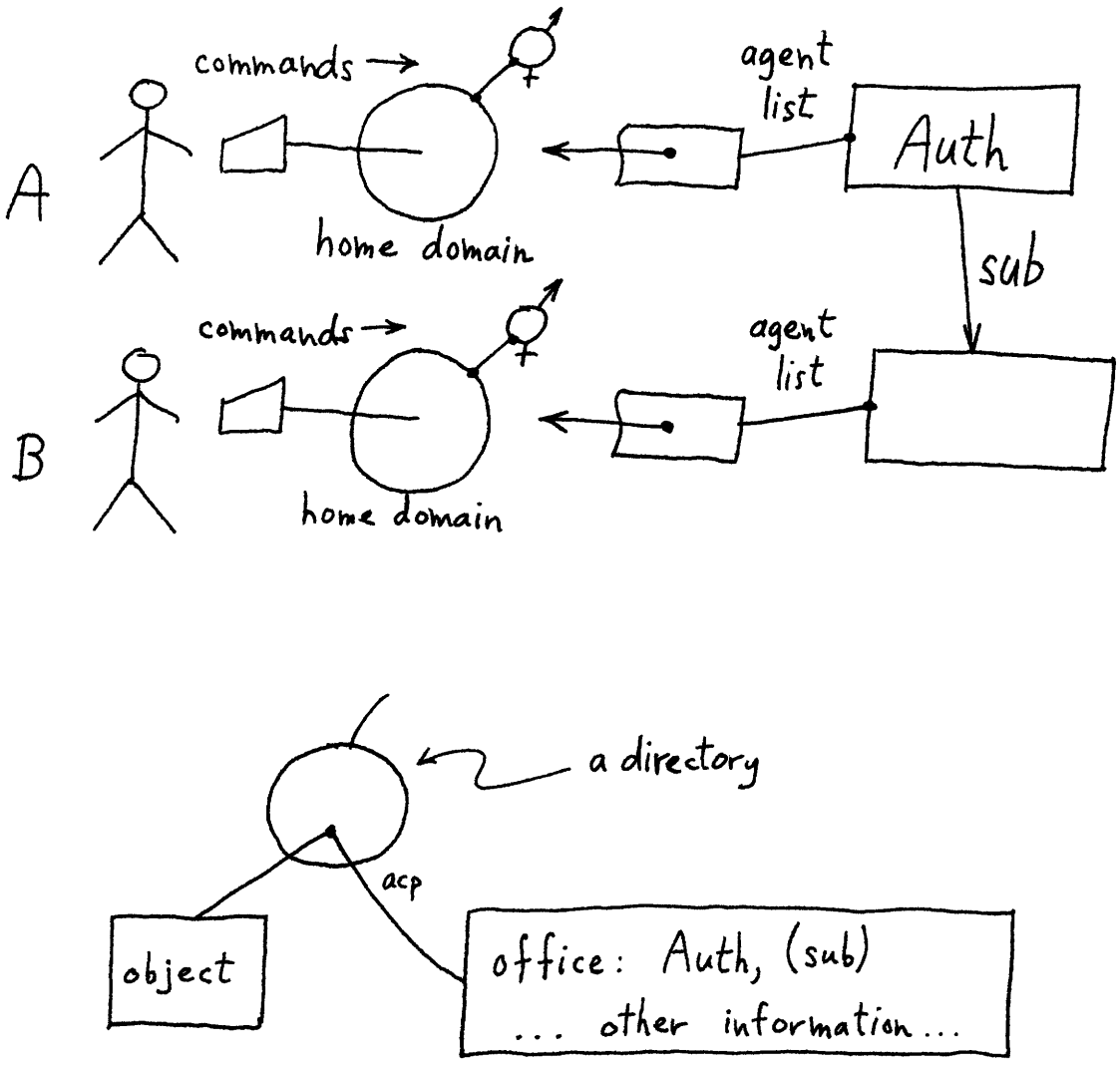


Figure 8-6. Superior, subordinate, and computing object.

ice that B is an agent of. Each organization will choose its own protocols. Each protocol will apply to requests of some particular kind which originate from agents of some particular set of offices. In other words, each protocol will have a definite scope, which is the set of requests to which the protocol applies. The people who are agents of offices of authority hierarchies ought to know and understand the protocols associated with their authority hierarchies, so that they will know what to expect in dealing with other members of their organizations. In a real sense, the protocols associated with an authority hierarchy are a statement of "the rules of the game" of the organization represented by the authority hierarchy, at least as regards exercising rights of control over computing objects. When protocols are changed, affected people will want to be notified or consulted in advance, because protocols have the power to protect agents of offices in important ways.

Protocols are composed of the following five types of activities:

1. Notification. Agents of offices, or particular persons specified by the protocol, are notified when the protocol is invoked. In the example of figure 8-6, a protocol is invoked when A commands his process to modify the acp of the object, because the operating system discovers that A's domain is not named by the agent list of the office specified in the acp of the object, but it is named by the agent list of an

office superior to that office. The protocol might notify all agents of the office (Auth,(sub)).

2. Delay. The protocol can specify a time delay of a specific length, or until messages are received from persons notified, before any action (activity of type 5) will take place. In our example, B's opportunity to explain to A the side effects of A's requested modification can be built into the protocol, as a delay. Also, a delay might be used to resolve conflicts of privacy.

3. Polling. The protocol can poll a group of people and use their votes to decide whether to perform the action whose request caused the invocation of the protocol. In our example, all the agents of (Auth,()) and (Auth,(sub)) might vote on A's requested change to the acp of the object.

4. Auditing. A journal of invocations of the protocol is kept or published. This might be used to inform members of the organization how other members voted on polls conducted by the protocol.

5. Action. The action whose request caused the invocation of the protocol is performed. This might be immediately preceeded by additional notifications.

Protocols will be specified (in a formal language) by the organizations for which they will work. Protocols will be implemented by programs (compiled from the protocol specifications) encapsulated in domains which are under the control of an independent group of administrators, the Protocol Adminis-

tration. The Protocol Administration will have its own authority hierarchy, and the offices of that authority hierarchy will have control of that part of the naming hierarchy where domains encapsulating protocols are catalogued. When the Protocol Administration finds that its work requires the use of a protocol, such protocols will be under the control of the authority system's locksmiths.

An organization changing a protocol will communicate to the Protocol Administration the new protocol together with its intended scope and a list of persons to be notified when the protocol is installed. In addition to the notifications requested by the organization, the Protocol Administration will send notifications of the installation of the new protocol to all the members of the organization who were receiving notifications from any protocol(s) replaced by the new protocol. In this way, members of organizations are kept informed when their organization's "rules of the game" are changing.

The Protocol Administration will keep records of protocols installed and notifications sent, and these records will be subject to subpoena. This is intended to deter members of organizations authorized to transmit new protocols to the Protocol Administration from using this power in any improper way.

The following types of requests will be honored by the computer utility according to a protocol:

1. Requests by a higher authority (agent of a superior office) to change an access control packet which is controlled

by an inferior office.

2. Requests by a higher authority to change the agent list of an inferior office.

3. Requests by a higher authority to activate surveillance over actions of agents of inferior offices.

4. Requests by any agent of any office to change the agent list of that office.

8.4. Locksmithing

The locksmiths in our design of a computer utility authorization system have the following two functions:

1. To control the agent lists of all the most-superior offices.
2. To install and maintain protocols for the Protocol Administration's authority hierarchy.

The reason for placing control of agent lists of most-superior offices in the locksmithing function is that some catastrophe might kill all the agents of the most-superior office of an organization's authority hierarchy, and the computer utility must provide a way for that organization to install new most-superior agents. The reason for placing control of the Protocol Administration's protocols in the locksmithing function is to avoid the recursion problem which would be encountered if the Protocol Administration controlled the protocols of its own authority hierarchy. The locksmiths would send notifications concerning changes to such protocols, just as described in section 8.3 for other protocols. The locksmiths would keep records of all their actions, and such records would be subject

to subpoena.

The locksmithing functions are implemented by primitives of the operating system, and the use of these primitives is authorized by a special office, called the locksmiths' office, which has an agent list that names the locksmiths' home domains. While this mechanism succeeds in telling the computer utility who the locksmiths are, it introduces the problem of authorizing changes to the agent list of the locksmiths' office. An infinite tower of superior offices could be introduced to solve the problem of authorizing locksmiths, but this is akin to mathematical fantasy and is not practical. A finite tower of superior offices leaves untouched and still troublesome the problem of responding to a catastrophe that kills all the agents of the most-superior office. Roughly speaking, the problem of authorizing the locksmiths is a recursion with no fixpoint, like the conundrum "Who watches the watchers?".

The simplest way to authorize the locksmiths while avoiding the recursion problem introduced above is to record the agent list of the locksmiths' office on a loop of paper tape, or a mini-disk, mounted on a peripheral device of the computer utility (*), conceivably associated with a security officer's console. In this way the secure authorizing of locksmiths depends on physical locks and keys on the device where the agent

---

(*) Every request by a process to use a locksmithing primitive will cause a scan of the agent list. The agent list could be read into the computer utility every time a locksmith uses a locksmithing primitive, or it could be read in only occasionally.

list is mounted, and on the secure operation of input from that device. Although the secure operation of input/output functions of a computer utility is outside the scope of this thesis, we do not doubt that hardware and software mechanisms can be devised to protect input and output operations. Thus the problem of authorizing locksmiths is reduced to the problem of controlling the key that opens the physical device where the agent list is mounted. This key might be locked in a safe deposit box in a bank, so that an independent authority (the bank) will have some control over, and keep records of, the use of this key.

8.5. Sharing Computing Objects

Whenever two officials, neither of whom is a subordinate of the other, have a substantial interest or expertise in controlling a computing object, they might agree to share control of that object. We are restricting our attention to the case where neither official is the other's subordinate because when the pair is a superior-subordinate pair, the superior can tell the subordinate what to do, or the superior's rights of control are established by a protocol as described in section 8.3, or both. The computer utility must have a mechanism to allow control of computing objects to be shared by officials or offices which are not directly or transitively related by the superior-subordinate relationship.

One way of modifying the authority hierarchy mechanism to accomodate the new relationship between officials just de-

scribed is to allow acps to contain two, or perhaps more, office names. This achieves an encoding of the information that the named offices share the acp in question, but no information is provided about which rights of control are to be exercised by which office. The mechanism must include means for offices to share rights of control over acps, and for that purpose we introduce a new kind of node, the protocol block, which may be used in constructing authority hierarchies. Recall that each office contains a list of entries (possibly an empty list), such that each entry points to an immediately inferior office. To include protocol blocks in authority hierarchies, we expand this definition and allow an entry of an office to point to either another office, or to a protocol block. As before, each entry has an associated name, and each office of every authority hierarchy except the most-superior office is pointed to by exactly one entry. Protocol blocks, however, may be pointed to by more than one entry. Figure 8-7 shows a protocol block which is pointed to by an entry named "share1" in the office (Auth,(sub,office1)) and an entry named "share2" in the office (Auth,(office2)). We say that the offices (sub,office1) and (office2) share the protocol block shown in figure 8-7. The purpose of the protocol block is to contain bindings to protocols which will determine how the computer utility will respond to requests that relate to shared computing objects, or to the protocol block itself. Figure 8-7 shows two protocols bound to the protocol block (sub,

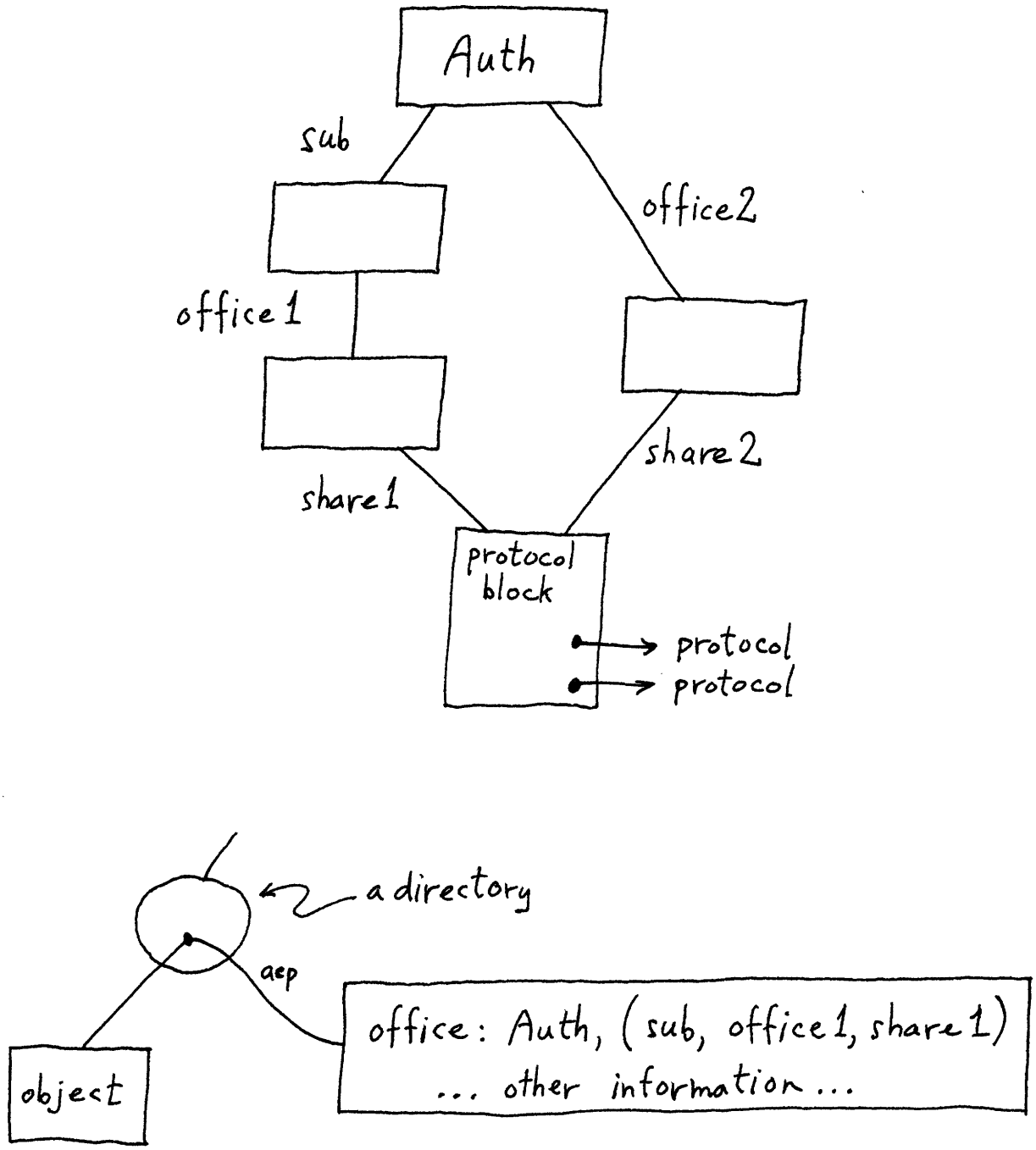


Figure 8-7. A shared object and a protocol block.

officel,sharel).

Protocol blocks establish the sharing of control over acps that name the protocol block instead of some office. Figure 8-7 shows a computing object whose acp contains one of the names of the protocol block in the figure; thus the protocols associated with the protocol block effectively determine how the two offices (Auth,(sub,officel)) and (Auth,(office2)) share control of the computing object.

Each protocol block must have at least two associated protocols. The first of these must contain the algorithm for deciding whether to honor requests to change acps which point to the protocol block, when such requests come from agents of the offices that share the protocol block. For example, the protocol which decides whether to honor a request to change the acp shown in figure 8-7 might poll all the agents of the offices that share the protocol block, and use the agents' votes as the basis for its decision. But when a higher authority requests a change to an acp controlled by a protocol block, a protocol associated with the authority hierarchy containing the office of the higher authority will be used to decide whether to honor the request.

The second protocol which must be associated with each protocol block must contain the algorithm for deciding whether to honor requests to revoke the participation of one of the offices that share the protocol block. If one office can revoke the participation of the other offices that share the

protocol block, that one office has more power than the other offices. If no office can revoke the participation of any other sharing office, a state of trust must exist between the sharing offices. What each trusts is the wisdom of the agents of the other sharing offices, and the propriety of the first protocol (defined in the previous paragraph).

8.6. Programmed Decision Making

It is very likely that the basic access control mechanisms described in this thesis will be unable to meet specialized requirements of users. This is because no provision has been made to allow access control decisions to depend on complex, user-specified considerations.

This limitation of the users' freedom to specify access can be eliminated by introducing a variant form of the access control packet, called a call packet. The call packet contains a domain entry capability which effectively points to an entry point of a domain which takes the place of detailed access information recorded in the access control packet. The call packet also contains an argument list which describes the arguments expected by the entry point specified by the domain entry capability. Whenever the operating system needs to make an access control decision based on the contents of an access control packet, and the acp is a call packet, the operating system will construct the specified argument list and call the specified domain with the specified arguments; and the called domain will return its decision to the operating system; and

the operating system will enforce that decision.

When the user can write a program to make access decisions, whether it is a program that is called because it is named in a call packet or a caretaker program for a data base encapsulated in a domain, the user can modularize the program, if it is large and complex, along the lines defined by the user's organizational delegation of authority. For example, figure 8-8 shows a Venn diagram of a set of requests A which P is responsible for granting or refusing, and a subset B of A which Q is responsible for granting or refusing. We are assuming that P has delegated to Q the authority and responsibility for deciding requests in B. If P and Q carry out this decision-making function by writing programs to make the decisions, those programs would be joined together as shown in figure 8-9. P's program would examine the incoming requests, and for requests in B, P's program calls Q's program. B must be a recursive set, and P's authority for defining B is exercised by writing the program fragment that tests whether requests are in B.

It is very likely that P will require Q's program to be encapsulated in a domain other than the one in which P's program is encapsulated, to prevent Q's program from usurping P's authority. Figure 8-10 shows how an authority hierarchy which expresses the fact that Q is P's subordinate is used to protect the authority of P's program. The authority hierarchy, at the upper left of figure 8-10, is named AP; and we assume that

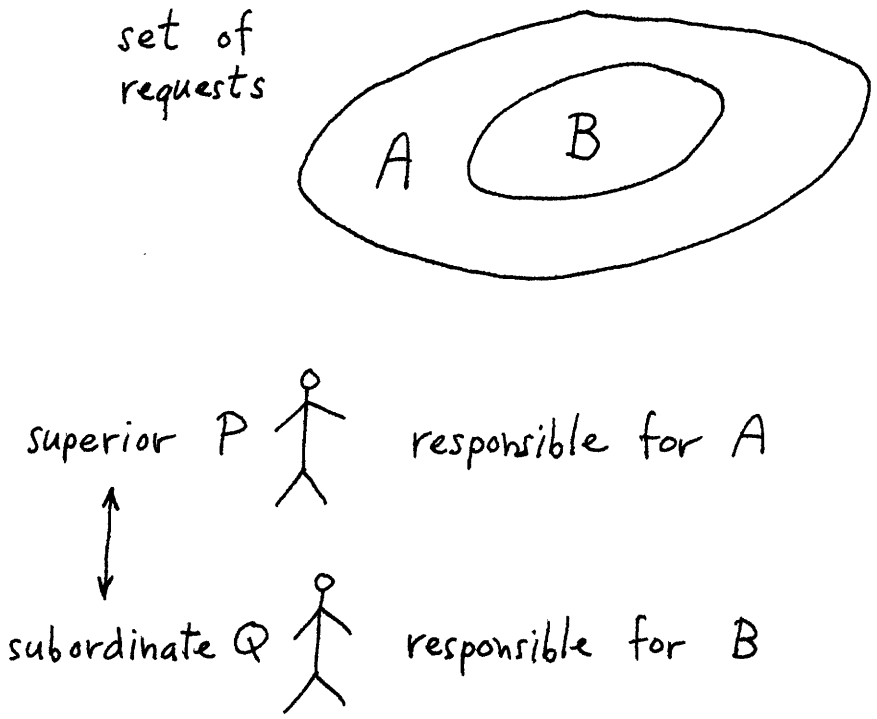


Figure 8-8. Sets of requests.

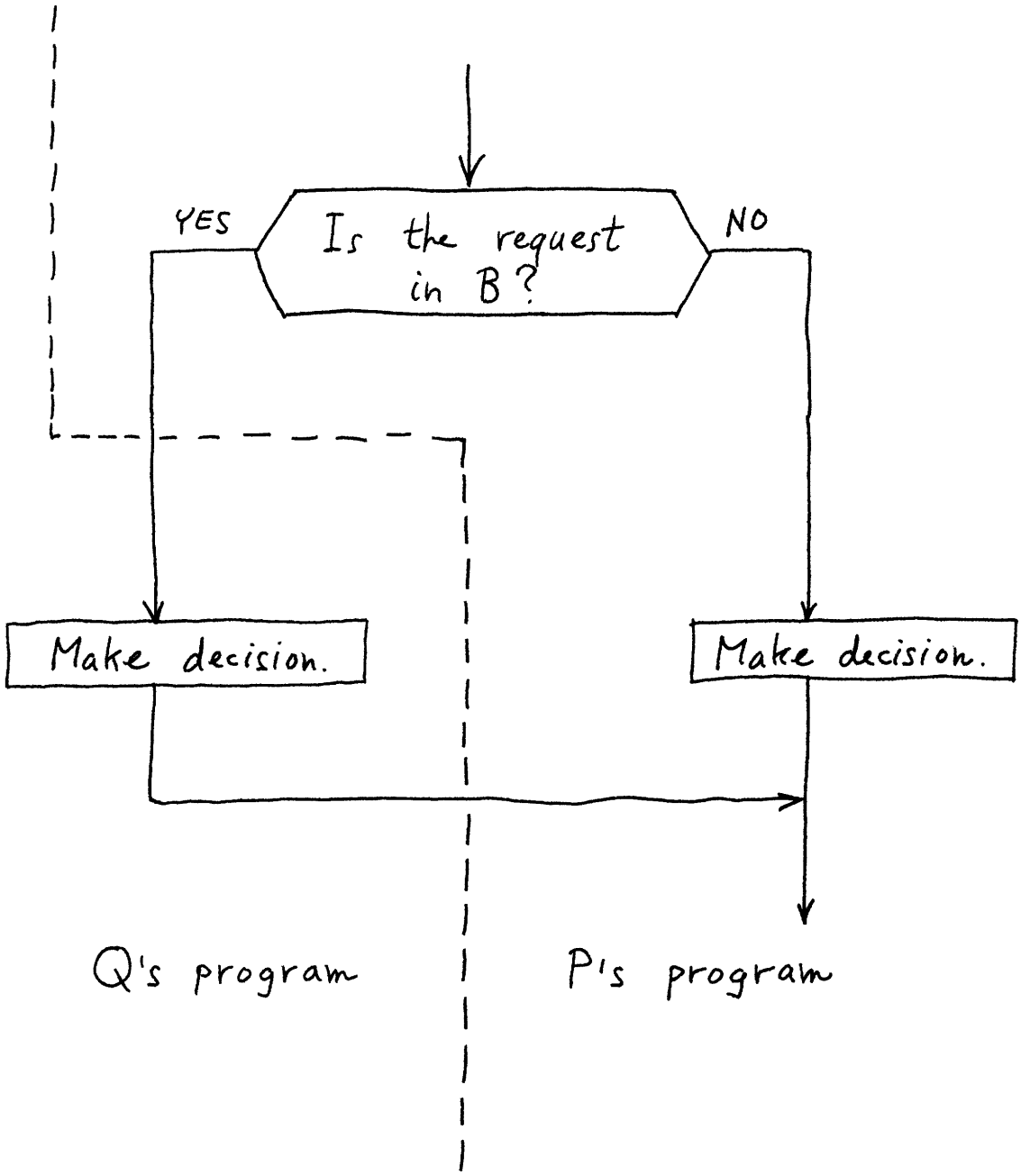


Figure 8-9. Program to decide requests in A.

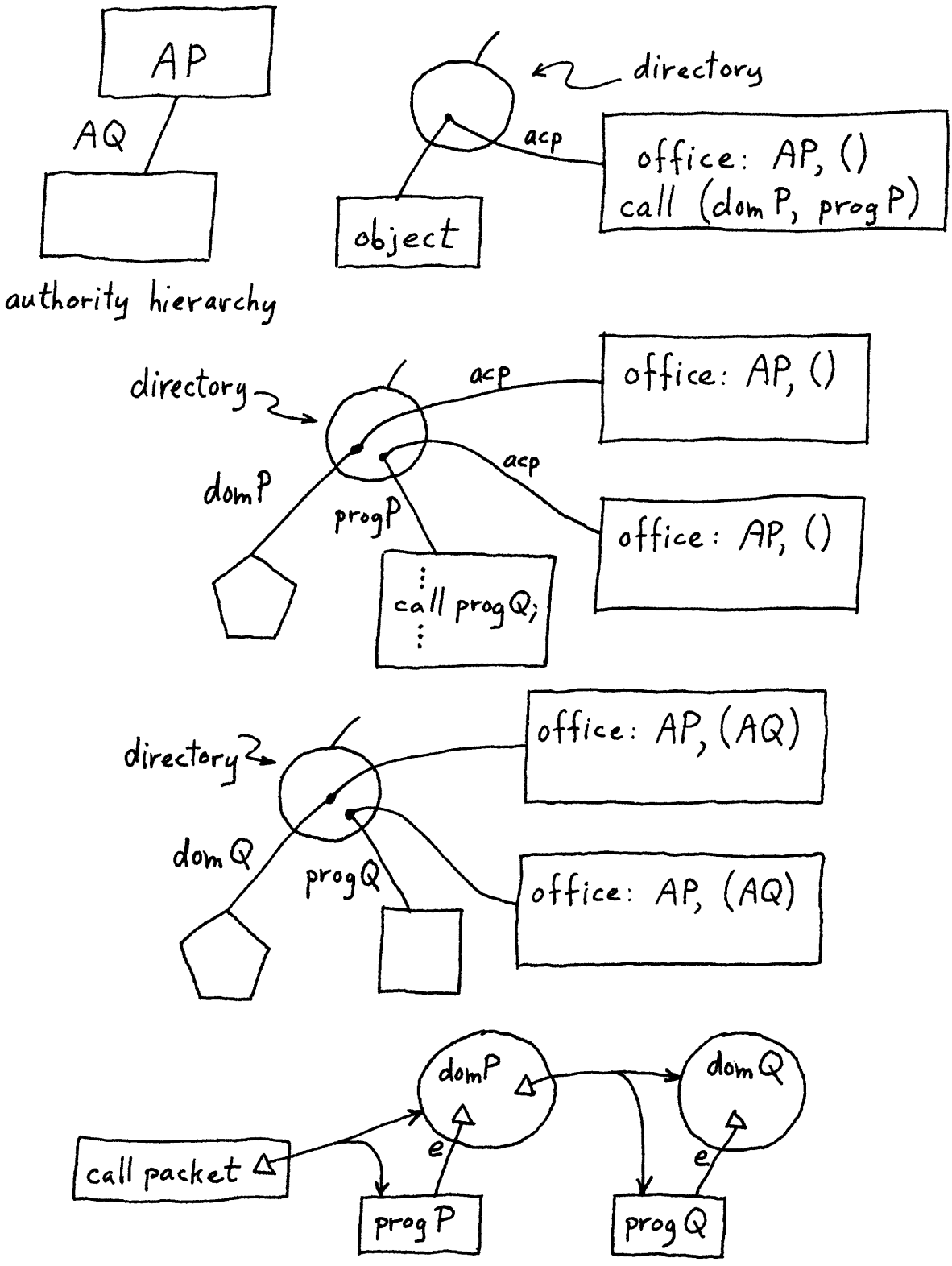


Figure 8-10. The isolation and subordination of the program progQ.

the person P is an agent of the office (AP,()) and the person Q is an agent of the office (AP,(AQ)). The set of requests A is generated by requests for access to the computing object at the upper right of figure 8-10, which has a call packet that invokes P's program progP in the domain domP, both of which are under control of the office (AP,()). When progP gets a request in B, it calls progQ in the domain domQ, both of which are under the control of the office (AP,(AQ)). P could have even more control over progQ if domQ were under the control of (AP,()).

8.6.1. Sharing Delegated Authority

If a superior official P with two subordinates Q and R has the authority and responsibility for granting or refusing a set of requests A, P might delegate authority over overlapping subsets of A to the subordinates Q and R. Figure 8-11, part 1, top right, shows a Venn diagram of two subsets B and C of A such that $B \cap C \neq \emptyset$. We are assuming that A has delegated authority over B to Q, and authority over C to R. When a request in $B \cap C$, the shaded part of the Venn diagram, must be acted on, Q and R might consult together and reach a joint decision. The remainder of figure 8-11, and figure 8-12, together show one way for P, Q, and R to automate a process by which they might reach a joint decision.

We are assuming that each of P, Q, and R will write decision-making programs which will be joined together as shown in figure 8-11, part 1, center. P's program progP

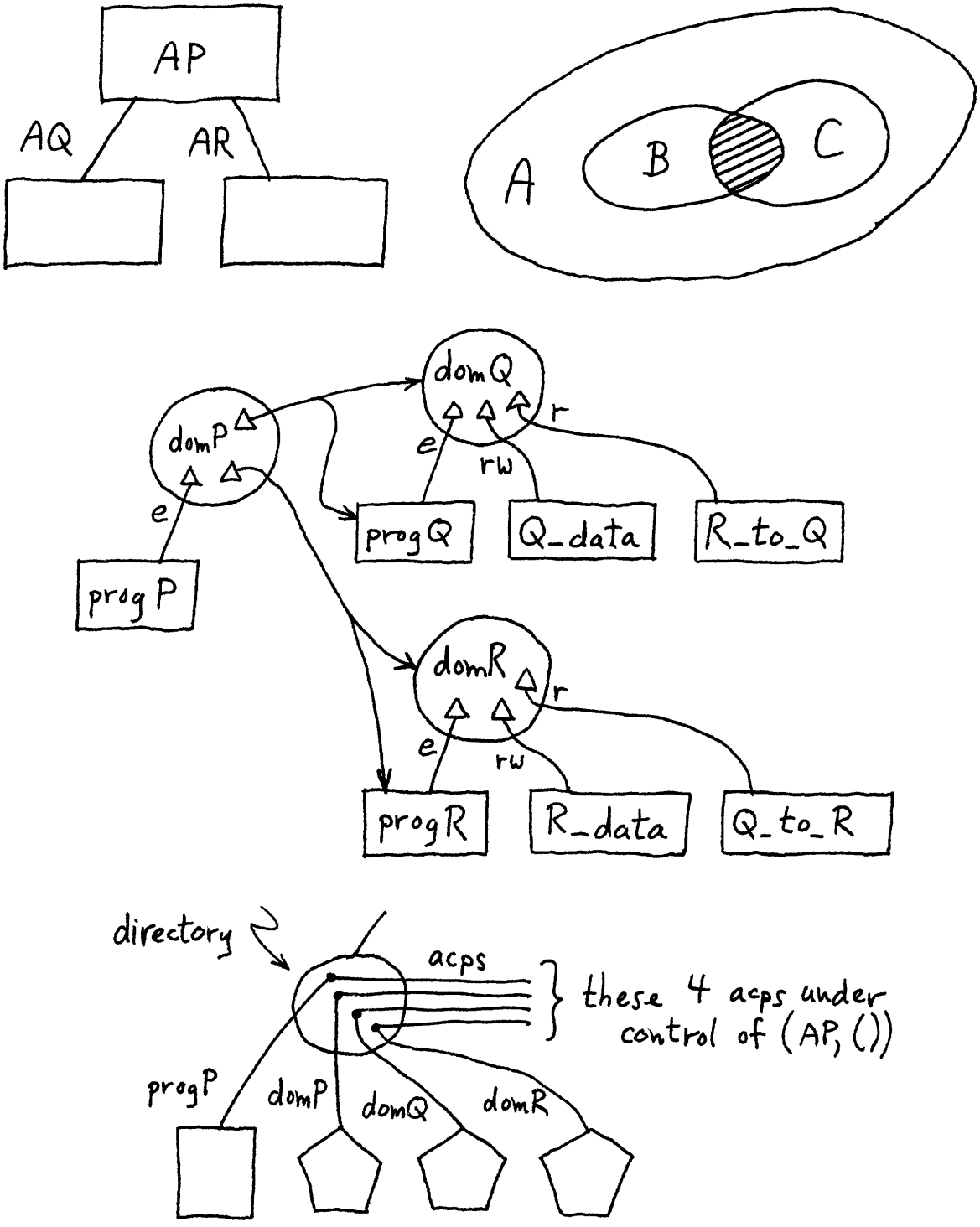


Figure 8-11, part 1. Structures for implementing shared delegated authority.

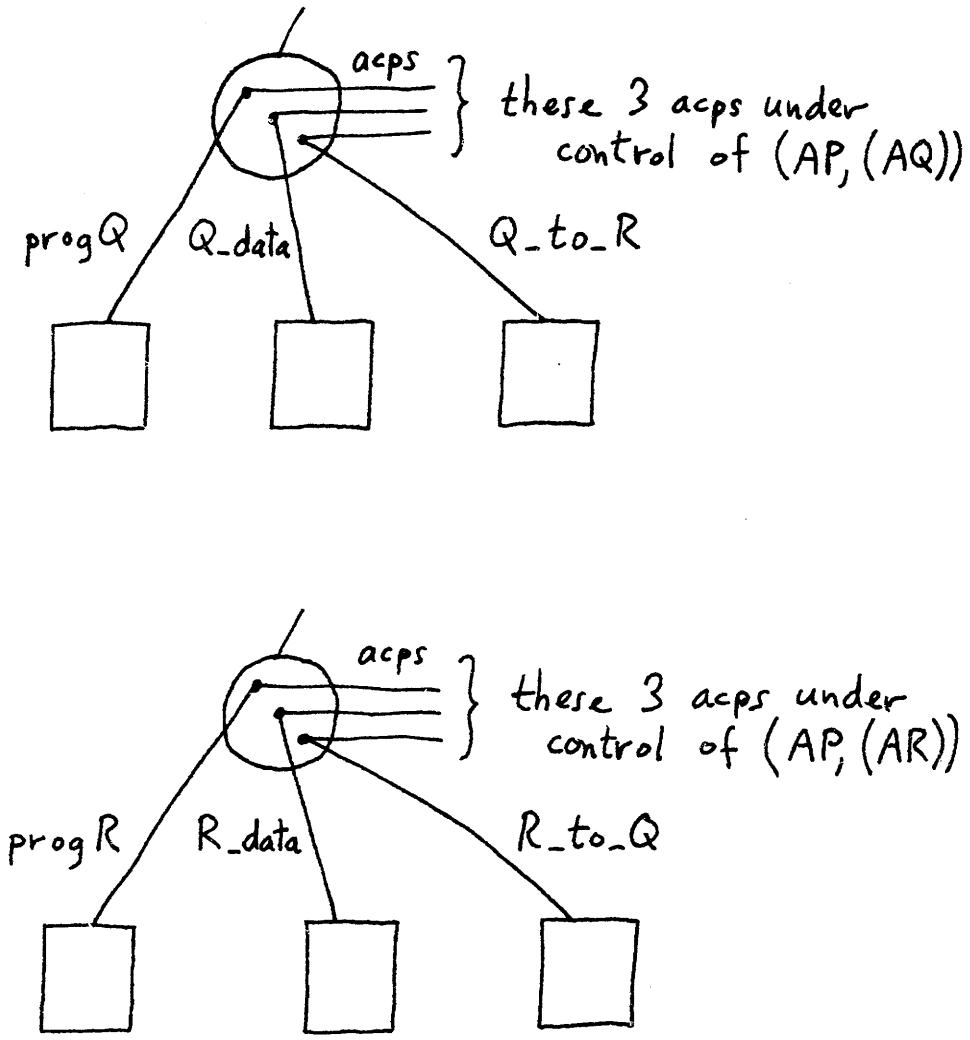


Figure 8-11, part 2. Structures for implementing shared delegated authority.

in domain domP is called by processes which need responses to requests in A. The program progP is shown in figure 8-12. progP will call Q's program progQ in domain domQ if the request it is working on is in B, and progP will call R's program progR in domR if the request is in C. For simplicity, we assume that the needed decision is a one-bit, yes-or-no answer. The line "if is_b & is_c then return(ok_b & ok_c);" in P's program gives both Q and R effective authority to veto the request. If the request is in A - (B ∪ C), progP uses its internal procedure decide_here to make the decision.

P's authority over Q and R is represented in the computer utility by the authority hierarchy shown in figure 8-11, part 1, top left. We are assuming that P is an agent of (AP,()), Q is an agent of (AP,(AQ)), and R is an agent of (AP,(AR)). P's authority over the set of requests A is exercised by P's program progP, and P's delegation of authority to Q and R is expressed by the calls in progP to progQ and progR. These latter two programs are prevented from usurping the authority of progP because they are encapsulated in isolated domains domQ and domR, respectively, and because domains domQ and domR are under the control of P. Figure 8-11, part 1, bottom, shows that these two domains encapsulating the subordinate programs, in addition to the domain domP and the program progP, are under the control of the office (AP,()) of which P is an agent.

When a request to be decided is in B ∩ C, a joint decision must be reached. progP will call both progQ and progR, and

```
progP:    procedure(request) returns(bit(1));
          declare (is_b,is_c,ok_b,ok_c) bit(1) initial("0"b);
          declare (progQ,progR) external entry returns(bit(1));

          is_b = belongs_to_B();
          is_c = belongs_to_C();
          if is_b then ok_b = progQ(request,is_c);
          if is_c then ok_c = progR(request,is_b);
          if is_b & is_c then return(ok_b & ok_c);
          if is_b then return(ok_b);
          if is_c then return(ok_c);
          return(decide_here());

belongs_to_B:       procedure returns(bit(1));
                    .
                    .
                    .
                    end belongs_to_B;

belongs_to_C:       procedure returns(bit(1));
                    .
                    .
                    .
                    end belongs_to_C;

decide_here:        procedure returns(bit(1));
                    .
                    .
                    .
                    end decide_here;

          end progP;
```

Figure 8-12. P's program.

the latter two programs will have the information that the request is in B∧C because of the arguments is_c and is_b passed to progQ and progR, respectively, by progP. We are assuming, in this simple example, that progQ is aware of special considerations which apply to requests in R's bailiwick by means of information in the segment R_to_Q. Thus, when progQ is called with parameter is_c equal to 1, progQ will use information in R_to_Q in making its decision. Similarly, when progR is called with parameter is_b equal to 1, progR will use information in the segment Q_to_R in making its decision. Figure 8-11, part 2 shows how the data segments R_to_Q and Q_to_R are kept under the control of R and Q, respectively. The acp of R_to_Q names the office (AP,(AR)), of which R is an agent; and the acp of Q_to_R names the office (AP,(AQ)), of which Q is an agent.

A more complex model of consultation between the two subordinates can be programmed if progP will create a second process, and make one process call progQ while the other process calls progR. These two processes, controlled by progQ and progR, could send each other messages about the request to be decided; and thus the decision-making computation could be interactive and flexible, up to the limit of the programmer's art.

8.6.2. Graft

The purpose of this section is to point out that graft can be programmed in the context of the interactive message-based decision-making computation suggested above. (*) The programs progQ and progR might sell favorable decisions for promises of payments to be made to Q and R, respectively, which promises might be received through messages.

Graft can be prevented by insuring that progQ and progR communicate only with each other, or by letting concerned persons audit progQ and progR.

(*) This distressing thought due to Stephen N. Zilles.

Chapter 9

Conclusions

9.1. On the Nature of Protection Systems

Protection systems are composed of walls and watchers. For example, banks have vaults with extremely tough walls, and guards at every door, and also automated watchers in the form of electronic or photographic cameras which can be activated in the event of a robbery. In the computer protection system presented in this thesis, the walls are provided by domains, as described in section 3.5; and the privacy restriction mechanism is a watcher. The privacy restriction mechanism effectively watches the copying and combining of information as directed by programs in the computer, by propagating restrictions among restriction sets. The propagated restrictions are then used to define new walls which have the effect of striking down output to users or input to domains. Having a watcher in our protection system in addition to walls is the crucial ingredient which allows the system to prevent accidental unauthorized releases of information.

Protection systems depend on non-forgeable objects. In the computer described in this thesis, some of the non-forgeable objects are the process state components which bind processes to domains, the names of domains, the agent lists of offices, and privacy restrictions. The operations in the

computer which modify any of these non-forgeable objects all require some form of authorization. For example, the domain entry capability effectively authorizes modifications to the binding of a process to a domain.

An interesting taxonomy of protection systems, due to Wilkes [Wi68], distinguishes list oriented systems from ticket oriented systems, but both types require non-forgeable objects. In a list system, control of access to the protected object is specified by a list of authorized accessors. Whenever an access is attempted, the list is searched for the name of the would-be accessor; thus protection depends on non-forgeability of such names. In our design, an example of a list system may be observed in the access control packets of segments, which contain lists of domains which are allowed to access the segments.

In a ticket system, access to the protected object is granted to any accessor who can present the proper ticket; thus protection depends on non-forgeability of the tickets. In our design, segment capabilities are tickets presented by domains to the processor hardware to validate requests, made by processes bound to the domains, to read, write, or execute (as instructions) segments.

9.2. Survey of the Sources of Complexity

A computer that keeps secrets is necessarily part of a larger information system, embedded in society, that keeps secrets. The design of such a computer is an interdisciplinary

task; that is, such a computer is a system which must operate correctly in terms of criteria established in several applicable bodies of knowledge, particularly social and technical knowledge. It is appropriate to think of a body of knowledge as a mountain range, with students climbing all over it, and hopefully up it, in the sense of acquiring knowledge of deeper results. Research makes the mountains grow, as results pile up, and individual specialized disciplines seem to grow their own mountains. In the context of this metaphor, interdisciplinary work produces a bridge between mountains. Foundations for interdisciplinary work are laid in the mountains of the relevant disciplines, and we have found that complexities introduced in such foundations are likely to become manifest in unexpected ways.

We have observed three sources of complexity in our design: (1) the tension between future good and future evil that might transpire as a result of use of a computer utility, (2) the social conventions by which American capitalist society is organized, and (3) the logical limitations introduced by the theory of recursive functions.

The computer is a tool that people will use to further their individual and group purposes, both good and evil. The goods and evils created by the technology described in this thesis are consequences of the tradeoffs that must be built into a computer utility that serves the conflicting interests of individuals and groups that want privacy, a society that

requires some disclosures, and computer users who want to share information. Probably no technologist will ever invent any technology that has only good effects. Technologists with moral sensibilities will direct their energies into projects whose social outcomes they will value, but decisions about how to apply available technology will be influenced by a wide range of considerations and interests that reflect values apart from those that motivate the seminal technologists. Those values which are most widely accepted will be codified as laws to require and outlaw good and evil technology, respectively.

The social conventions of our society's capitalist economic life have contributed considerably to the complexity of our design, most notably to the design of the Proprietary Services Administration, described in chapter 5, and the design of access control packets for domains, described in chapters 4 and 5. Other social conventions have contributed still other complexities; e.g., man's willingness to trust others finds expression in the system of warrants described in section 4.7, which allows users of segments to be assured by others that particular program segments are not Trojan Horses.

Finally, the cold, hard realities of recursive function theory impose limitations on our design. In chapter 6, we noted that when two restrictions strike at the same time and

both set off alarms, the computation whose activity is thus interrupted must be checkpointed and examined later by an "appropriate authority" to determine whether the purpose of each striking restriction is to protect privacy, or to leak information. We conjecture that this problem is unsolvable, in the recursive function theoretic sense. That is, we conjecture it is impossible to write a program to make this decision. If the problem were solvable, then the notification to the restriction owner whose striking restriction is trying to leak information could be suppressed, and the computer system would be more secure.

Our conjecture is based on the idea that programs can have hidden purposes. In a formal logical system for proving theorems about programs, sentences in a formal language are associated with paths through which control flows between statements of the program. The validity of the sentences associated with the program can be established by proving that if the sentences associated with the entry points are true when control reaches the entry points, then whenever control reaches any other path (the entry point is a special path), the sentence associated with that path will be true. In this context, the purpose of a program is to make true the sentence associated with the path through which control leaves the program. Now suppose this computer system is programmed with a programming system which requires programmers

to supply these sentences which, in effect, tell what their program does; and suppose that the programming system checks for itself that the sentences are valid, as defined above. A programmer might write a program whose purpose is to leak information and supply a valid set of sentences which hide this purpose. For example, if the programmer wrote the sentence "true" for every path in his program, then the sentences say nothing about what the program does, and yet whenever control is in any path, the sentence associated with the path is true. In this example, all the purposes of the program are hidden.

There is only one straightforward way to find the hidden purposes of a program, and that is to enumerate all the sentences which are provably true when control leaves the program. This will be a recursively enumerable set of sentences, but probably not a recursive set (this is the heart of our conjecture), analogous to the fact that the set of sentences provable from most interesting sets of axioms will be a recursively enumerable, not recursive,set. So, if we were trying to find an effective procedure to deal with the situation of two restrictions striking at the same time and sounding two alarms, we would arrive at the problem of determining the hidden purposes of programs; and when the set of hidden purposes is not recursive, this straightforward way peters out.

We are not sure whether to despair or be gleeful in the face of this seemingly unsolvable problem. But since it appears that the computer cannot muddle through the situation of two restrictions striking simultaneously sounding two alarms, the mechanism in chapter 6 checkpoints the computation and calls in an "appropriate authority," a human, who must proceed to solve the particular case at hand of the problem that is probably unsolvable in general. We expect the human to solve the particular case because the human will bring additional specialized information to bear on the problem, and because human problem-solving is aided by flashes of insight.

Our design of a computer that keeps secrets has included a relatively large number of different, intricate mechanisms. The complexities of our design grew out of the complexities of the design's foundations. If there were an abstraction that encompassed all the sources of complexity, perhaps in the form of a theory that was simple and internally coherent and that explained both society and computers, then the mechanisms of our design would be simple when explained in terms of that theory. But we have found no such theory. Computers are defined and limited by natural and logical laws, while protection mechanisms are developed and maintained to protect the interests of concerned communities or individuals. A simple theory that explained computer protection mechanisms

would have to encompass concepts ranging from the sensitivities and selfishness of humans to the non-recursiveness of certain sets. Probably no such theory exists, or can exist.

9.3. On Robotic Watchers

"It all depends on whose ox is gored."

-- (folk wisdom)

The ultimate important questions which must be asked about protection systems are, "Who benefits?", and "Is it fair?" Such questions are relatively easy to ask and answer with respect to the walls of protection systems, because the walls are visible to everyone and any thinking person can observe the walls, ask the questions, and decide as an individual whether the walls are fair. This is slightly more true with respect to physical walls such as those of banks and international borders, which are visible to the eye, than with respect to walls erected inside computer systems, because some technical expertise is required to understand the walling-out function provided by domains.

But the utlimate questions, "Who benefits?" and "Is it fair?", are much harder to answer with respect to watchers. The watchers we have in mind are the police forces, especially the free-wheeling type of investigators like James Bond(*) or those of the C.I.A., who carry on such "important" work as

(*) James Bond is a fictional superhero created by Ian Fleming.

spying, overthrowing governments, and other "dirty tricks."

The basic problem with the free-wheeling investigators is that they operate in secret, and therefore citizens have no way of deciding for themselves if the actions of these watchers are fair unless the secrecy is removed. But the investigators have argued, quite successfully, that secrecy is essential to their functioning. One way we see to solve this problem is to replace the free-wheeling investigators with a fleet of robots controlled by computers, and to open to public scrutiny the programs that control those robots, which we call robotic watchers. Note that we are not proposing to make public the investigative files compiled by robotic watchers; such files about specific cases being worked on would remain secret. But the procedures by which the robotic watchers operated would be available for public examination, and individuals could answer the questions "Who benefits?" and "Is it fair?" for themselves by reading the procedures. The processes of public scrutiny, criticism and debate would shape the robotic watchers, and make them more fair. Eventually, robotic watchers might become more highly trusted than politicians.

An example of a simple robotic watcher should help to clarify this conception: imagine an electronic device with radar "eyes" mounted on a police car, observing all the cars on the road (including those in the opposite lanes) for the

purpose of estimating the drunkenness of the drivers. Such a machine, if it worked, would help save many innocent lives. It is known that drunk drivers make many mistakes, so it seems reasonable to expect that such behavior could be detected by a robot.

However, two factors make the emergence of sophisticated robotic watchers unlikely. First, there is the risk of instant totalitarianism at some time in the future, which might occur if the power of the robotic watchers were to be seized in a coup. Second is the high probability of encountering unsolvable problems (in the recursive function theoretic sense) which tend to make a proposed robotic watcher impossible to implement adequately. These factors will probably limit the applications to which robotic watchers will be applied to simple watching operations that assist human investigators.

Bibliography

[An72] Anderson, James P., Computer Security Technology Planning Study, Electronic Systems Division, Air Force Systems Command, ESD-TR-73-51, Vol. I, ESD-TR-72-XXXX, Vol. II

[Ba64] Baran, Paul, On Distributed Communications: Security, Secrecy, and Tamper-Free Considerations (IX), Rand Corporation, RM-3765-PR

[Bed71] Bedingfield, Robert E., Inquiry Urged on Trading in Stock of Penn Central, The New York Times, March 29, 1971, p. 1

[Ben72] Bensoussan, A., Clingen, C. T., and Daley, R. C., The Multics Virtual Memory: Concepts and Design, Comm. ACM Vol. 15, No. 5, May 1972, pp. 308-318

[Ber72] Bernstein, Carl, and Woodward, Bob, FBI Finds Nixon Aides Sabotaged Democrats, Washington Post, October 10, 1972, p. 1

[Bra73] Branstad, Dennis K., Privacy and Protection in Operating Systems, Operating Systems Review, Vol. 7, No. 1, January 1973

[Bri71] Bride, Edward J., Businesses not Security-Conscious, Computerworld, May 12, 1971, p. 1

[Bu61] Burroughs Corporation, The Descriptor - a definition of the B5000 Information Processing System, Detroit, Mich., February 1961

[Com71] Computerworld (no author cited), Detective Sentenced for Selling Data, May 12, 1971, p. 4

[Cor72] Corbató, F. J., Clingen, C. T., and Saltzer, J. H., Multics: The First Seven Years, AFIPS Conf. Proc. 40, 1972 SJCC, pp. 571-583

[Cr65] Crisman, P. A., Ed., The Compatible Time-Sharing System, M.I.T. Press, 1965

[De66] Dennis, Jack B., and Van Horn, Earl C., Programming Semantics for Multiprogrammed Computations, Comm. ACM, Vol. 9, No. 3, March 1966, pp. 143-155

[DoD71] Department of Defense, "The Pentagon Papers," The New York Times, Bantam Books, July 1971

[Fa68] Fabry, Robert S., Preliminary Description of a Supervisor for a Computer Organized Around Capabilities, Quarterly Progress Report No. 18, Institute of Computer Research, U. of Chicago, August 1968

[Fr68] Fried, Charles, Privacy, Yale Law Journal 77 (1968)

[Gr68] Graham, Robert M., Protection in an Information Processing Utility, Comm. ACM Vol. 11, No. 5, May 1968, pp. 365-369

[Han171a] Hanlon, Joseph, Consumers Get Some Protection With Credit Law, Computerworld, April 28, 1971, p. 1

[Han171b] Hanlon, Joseph, High Court Refuses to Hear Case on Dossier Access, Computerworld, June 2, 1971, p. 3

[Hans71] Hansen, Morris H., Insuring confidentiality of individual records in data storage and retrieval for statistical purposes, AFIPS Conf. Proc. 39, 1971 FJCC, pp. 579-585

[Hi71] Hirschhorn, Lawrence A., Toward a Political Economy of Information Capital, Ph.D. Thesis, M.I.T., September 1971

[Ho70] Hoffman, Lance J. and Miller, W. F., Getting a Personal Dossier from a Statistical Data Bank, Datamation, May 1970

[Hu70a] Hunter, Marjorie, House Backs End of Teller Votes on Amendments, The New York Times, July 28, 1970, p. 1

[Hu70b] Hunter, Marjorie, Congress Reform Passed by House in 326-to-19 Vote, The New York Times, September 18, 1970, p. 1

[IBM64] IBM Corp., IBM System/360 Principles of Operation Form No. A22-6821

[Ki62] Kilburn, T. et al., One-Level Storage System, IRE Trans. on Electronic Computers, April 1962, pp. 223-235

[La69] Lampson, Butler W., Dynamic Protection Structures, AFIPS Conf. Proc. 35, 1969 FJCC, pp. 27-38

[La71] Lampson, Butler W., Protection, Proc. Fifth Princeton Conference on Information Sciences and Systems, Princeton University, March 1971, pp. 437-443

[Me69] Meldman, Jeffrey A., Centralized Information Systems and the Legal Right to Privacy, Marquette Law Review 52 (1969)

[Mi71] Miller, Arthur R., The Assault on Privacy: Computers, Data Banks and Dossiers; The University of Michigan Press, Ann Arbor, 1971

[MIT72] M.I.T. Project MAC, Multics Programmers Manual 1972

[Ne72] Needham, R. M., Protection systems and protection implementations, AFIPS Conf. Proc. 41, 1972 FJCC, pp. 571-578

[Or49] Orwell, George, 1984, New York: Harcourt, Brace and Company, 1949

[Ro71] Rotenberg, Leo, Surveillance Mechanisms in a Secure Computer Utility, Computers and Society, March 1971

[Sc71] Schroeder, Michael D., Performance of the GE-645 Associative Memory while Multics is in Operation, ACM Workshop on System Performance Evaluation, ACM, April 1971, pp. 227-245

[Sc72a] Schroeder, Michael D., and Saltzer, Jerome H., A Hardware Architecture for Implementing Protection Rings, Comm. ACM, Vol. 15, No. 3, March 1972, pp. 157-170

[Sc72b] Schroeder, Michael D., Cooperation of Mutually Suspicious Subsystems in a Computer Utility, M.I.T. Project MAC, MAC-TR-104, September 1972

[Th71] Thomas, Robert H., A Model for Process Representation and Synthesis, M.I.T. Project MAC, MAC-TR-87, June 1971

[US68a] U. S. Congress, House Committee on Government Operations, Privacy and the National Data Bank Concept. 90th Cong., 2nd Sess. (1968)

[US68b] U. S. Congress, House Committee on Government Operations, Hearings on Commercial Credit Bureaus. 90th Cong., 2nd Sess. (1968)

[US71] U.S. Congress, Senate Committee on the Judiciary, Federal Data Banks, Computers and the Bill of Rights. 92nd Cong., 1st Sess. (1971)

[Va69] Vanderbilt, Dean H., Controlled Information Sharing in a Computer Utility, M.I.T. Project MAC, MAC-TR-67, October 1969

[Wa90] Warren, Samuel D. and Brandeis, Louis D., The Right to Privacy, Harvard Law Review 193 (1890)

[Wei69] Weissman, C., Security controls in the ADEPT-50 time-sharing system, AFIPS Conf. Proc. 35, 1969 FJCC, pp. 119-133

[Wes67] Westin, Alan F., Privacy and Freedom, Atheneum, New York, 1967

[Wi68] Wilkes, Maurice V., Time-Sharing Computer Systems, American Elsevier, 1968

[YLJ71] Yale Law Journal, Vol. 80, No. 5, April 1971, p. 1035, Protecting the Subjects of Credit Reports, (Richard A. Block, student contributor)

Appendix 1

Process State and State Transition Rule

The purpose of this appendix is to define the process state and state transition rule of our process which jumps between domains and has a sectioned stack. This definition of a process is different from those of Lampson [La69] and Schroeder [Sc72b] because domains are not a part of our process; rather the domains exist in the environment of all the processes.

The process state, denoted ♀, is a tuple whose components are listed in figure A1-1. The first three components of ♀ effectively bind the process to a domain; these components are called dom_id, vb, and dom_pt_addr. The component vb is a validity bit which signifies, when it is 1, that the value of dom_pt_addr is the absolute address of the page table of the C-list of the domain whose unique identifier is dom_id. The component dom_id is logically adequate by itself to bind the process to the identified domain. But for reasons of efficiency, the component dom_pt_addr is included also. Our design aims to minimize the cost of letting processes jump between domains, to the extent that it can be minimized when the domains are not a part of the process state. The fourth component, pc, is the program counter; and it has the form (seg#,word#), as in the Multics process. The registers of the process include accumulators, base registers, index registers, and perhaps also floating point

component	purpose
dom_id	to identify the domain to which the process is bound
vb	validity bit for dom_pt_addr
dom_pt_addr	to locate the C-list of the domain identified by dom_id
pc	program counter of process
registers	general computation
stack_pt_addr	to locate the sectioned stack of the process
Min, Max	to define the accessible portion of the sectioned stack
proc_id	to identify the process
fault#	to identify the causes of faults

Figure A1-1. The components of the process state ♀.

registers. The component stack_pt_addr is the absolute address of the page table of the process' sectioned stack. The components Min and Max define the accessible portion of the sectioned stack. The component proc_id is a unique identifier of the process. It is not modified by the state transition rule at all. Finally, the component fault# is used to record the reasons why when the process takes a fault. Our state transition rule does not ever show this component being modified explicitly, but in fact it is set by the activity of every box labelled "FAULT". These boxes all transfer control of the processor to the box labelled "all FAULTs" at the top of part 9 of figure A1-2, whereupon the fault# is examined.

Figure A1-2 is the state transition rule of our process. This state transition rule completely defines all protection-related activity of the processor ♂ which evolves our process (except for the privacy restriction mechanism and processor defined in chapters 6 and 7).

Figure A1-2, part 1 shows the processor logic for validating the process' binding to a domain, followed by instruction fetch logic. If the validity bit vb is 0, the processor searches a system data base called the Active Domain Table (ADT) trying to find dom_pt_addr(dom_id). This is the absolute address of the page table of the C-list of the domain identified by dom_id. If the identified domain is active, the search is successful and the processor can proceed. Otherwise the processor generates a fault, jumping thereby into the

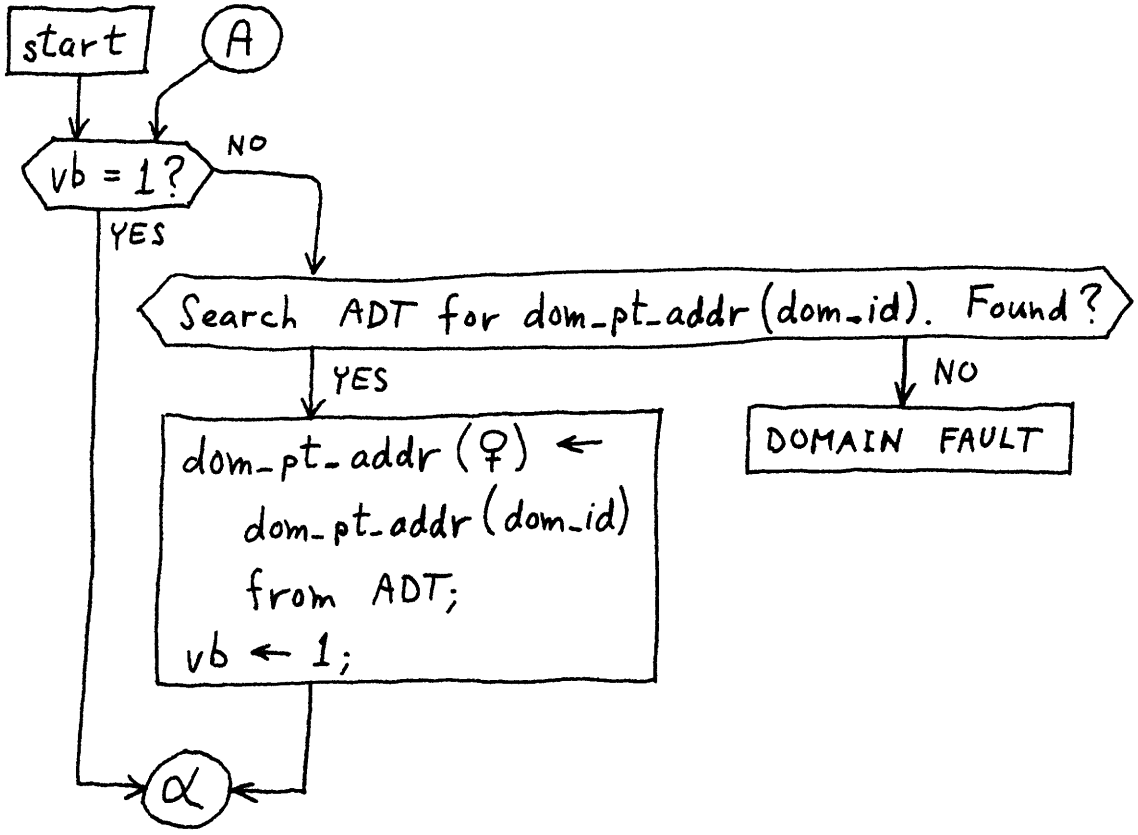


Figure A1-2, part 1(a). State transition rule - domain binding.

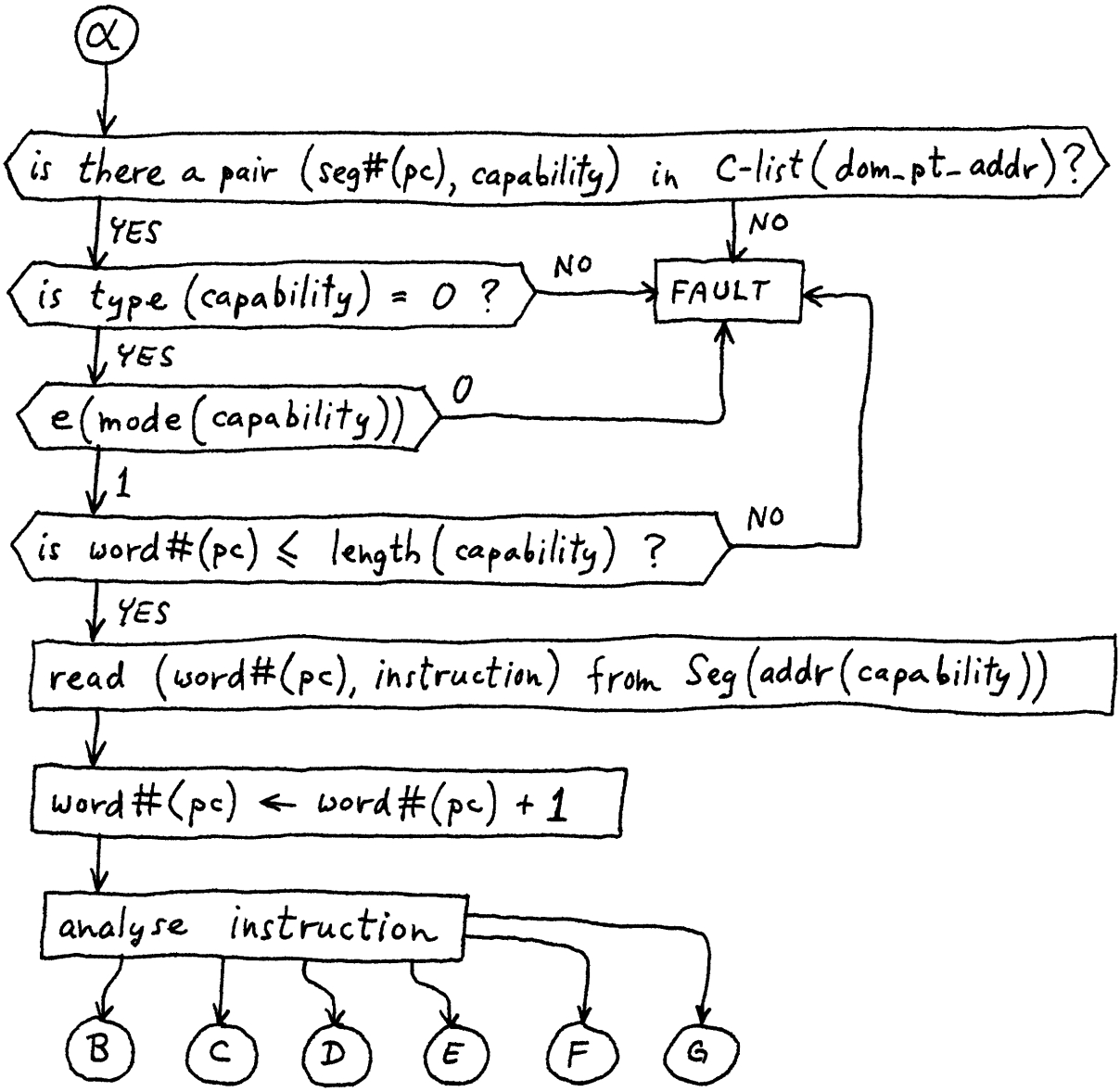


Figure A1-2, part 1(b). State transition rule - instruction fetch.

operating system program which will activate the domain. We describe the search of ADT in detail in Appendix 2.

The instruction fetch logic begins by examining a capability from the array C-list(dom_pt_addr). By "C-list (dom_pt_addr)", we mean the C-list stored in the segment whose page table address is dom_pt_addr. For the purposes of notating the state transition rule, we define a C-list to be a set of ordered pairs {(cap#,capability)} which is a function in the set-theoretic sense. A capability is a 4-tuple or a triple, depending on its type. Segment capabilities are 4-tuples, having the form (type, mode, length, addr), subject to the constraint that the first component, i.e., type(capability), must have the value 0. The other components of a segment capability are referred to as mode(capability), length(capability), and addr(capability). The component mode is a 3-bit string whose bits are referred to as e(mode), r(mode), and w(mode). The component length tells the length of the segment. The component addr is the absolute address of the base of the page table of the segment. We refer to the segment as Seg(addr(capability)), and for the purposes of notating the state transition rule we define Seg(addr (capability)) to be a set of pairs {(word#,bitstring)} which is a function in the set-theoretic sense.

The instruction fetch logic checks that the capability selected by seg#(pc) is a segment capability, that its e(mode) bit is on, and that the segment is long enough to contain a

word#(pc)-th word. The fetch logic then reads that word from the segment, indexes the program counter, and analyses the fetched instruction. The state transition rule implements six different classes of instructions. These are register-to-register functional operation instructions (e.g., "ADD"), memory reference instructions (e.g., "LOAD" and "STORE), a conditional transfer instruction (which can be made to transfer unconditionally), stack growing and shrinking instructions, the call-domain instruction, and the return-domain instruction. These last two are the only members of singular instruction classes.

Figure A1-2, part 2 shows the state transition rule for register-to-register operations. Our model does not have much detail in this area because the realm in which these operations operate, to wit the registers of the process state, is a uniformly protected collection of information. There can be little argument against the assertion that a process has the right to read and write its own registers, and that is all these operations require.

Figure A1-2, parts 3 and 4 show the state transition rule for references to memory. All such references begin with the formation of an operand address, a two-part address (t,x). If t is equal to the constant STACK#, the address refers to the sectioned stack. (If t is to be represented by a 15-bit field, we would choose 2^{15}-1 as the value of STACK#.)

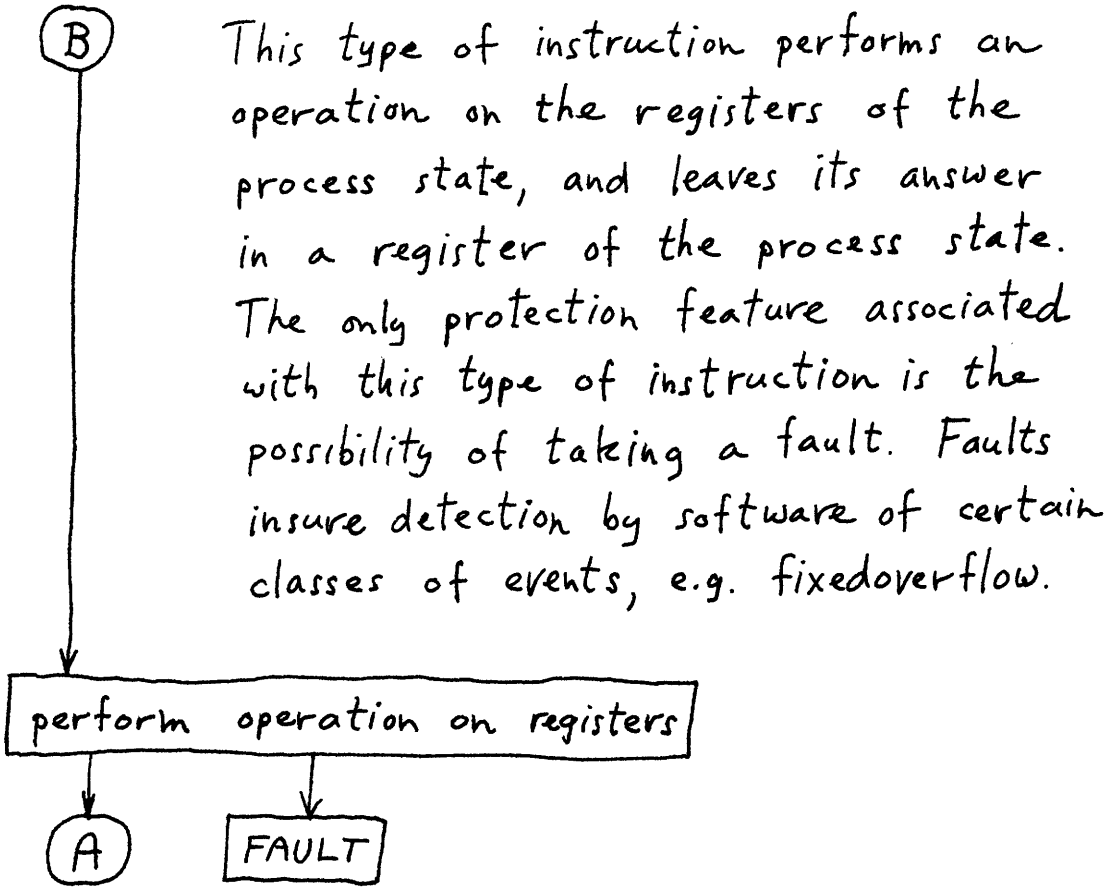


Figure A1-2, part 2. State transition rule - register-to-register operations.

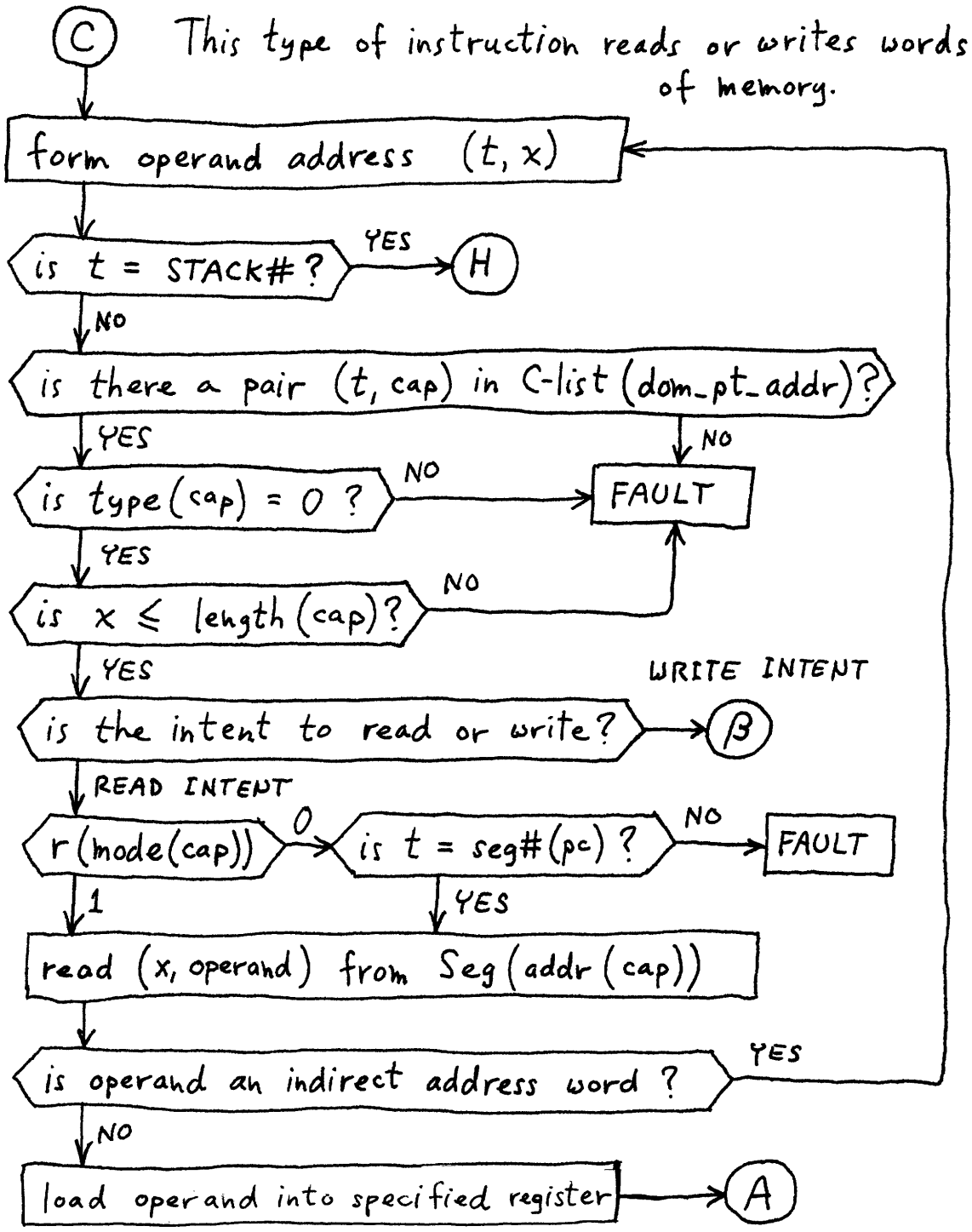


Figure A1-2, part 3(a). State transition rule - references to segments.

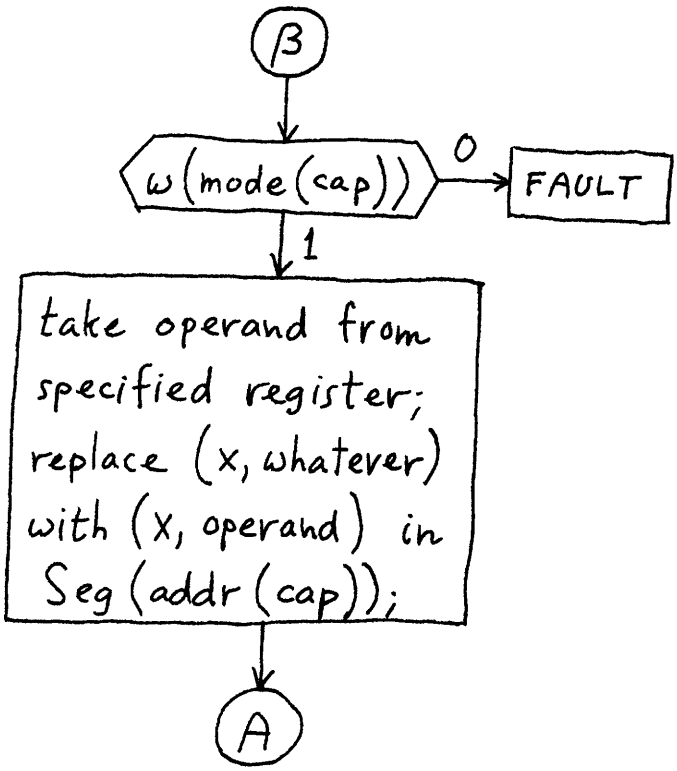


Figure A1-2, part 3(b). State transition rule - references to segments.

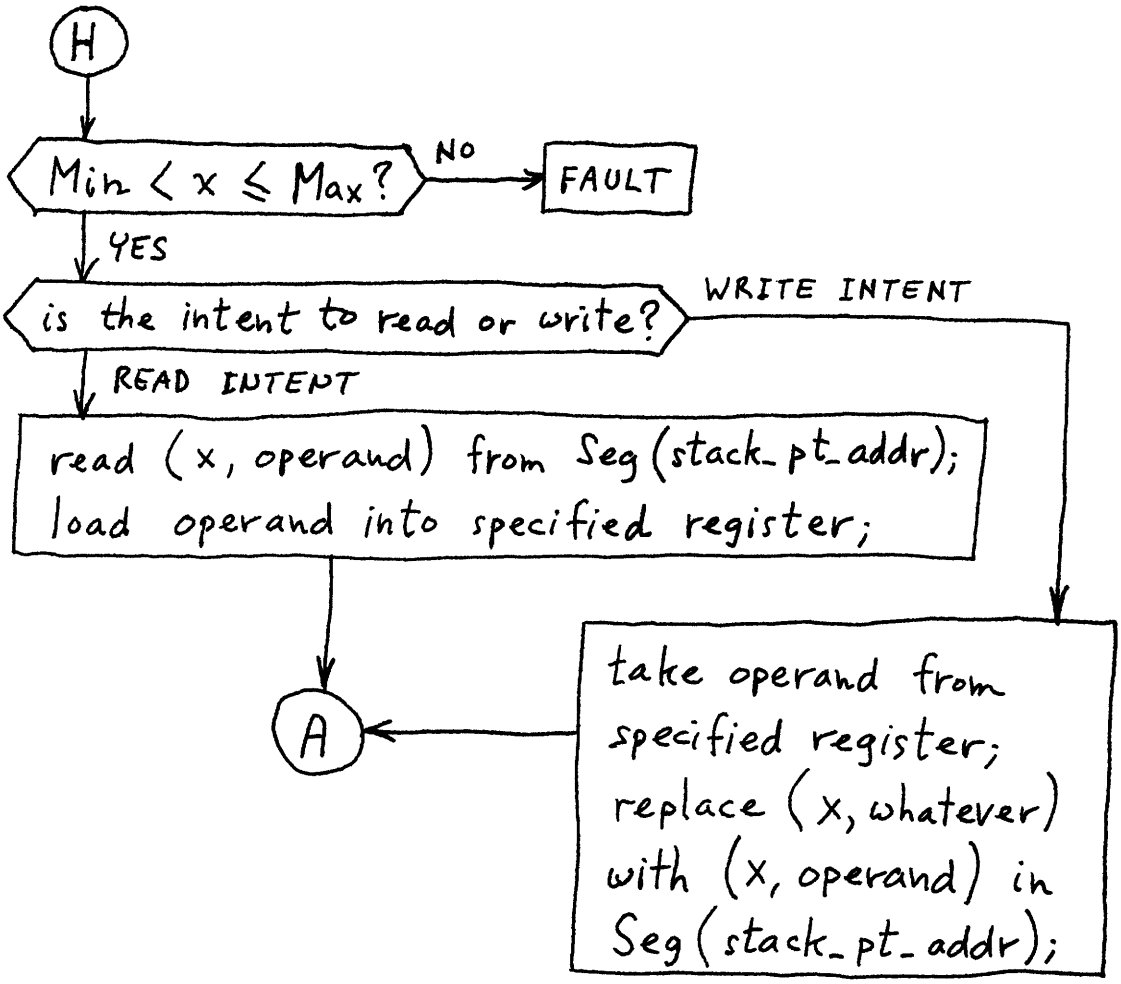


Figure A1-2, part 4. State transition rule - references to stack.

The tests in figure A1-2, part 3 which determine what reading and writing operations are allowed; are similar to the tests shown in figure 3-4, part 2, our model of Multics. Figure A1-2, part 4 shows that a reference to the sectioned stack is allowed only if Min < x $\leq$ Max. Thus, the sectioned stack has no accessible words when Min = Max. The state transition rule will keep Min less than or equal to Max, as will be seen.

Figure A1-2, part 5 shows the state transition rule for transfers of control. We are assuming that the conditional transfer instruction is used for unconditional transfers, as in the IBM System/360 [IBM64]. The transfer target address is validated before the transfer is effected, so that when the target address is invalid the programmer trying to fix the bug will know exactly which instruction attempted the transfer. This feature is adopted from Multics [Sc72a].

Figure A1-2, part 6 shows the state transition rule for growing and shrinking the accessible portion of the sectioned stack. A process may grow its accessible portion at any time and by any (positive) amount; and it may shrink its accessible portion all the way down to zero length. Note that words of the stack are zeroed as Max is reduced.

Figure A1-2, part 7 shows the state transition rule for the call-domain instruction. The instruction specifies an argument window size and a domain entry capability. Domain

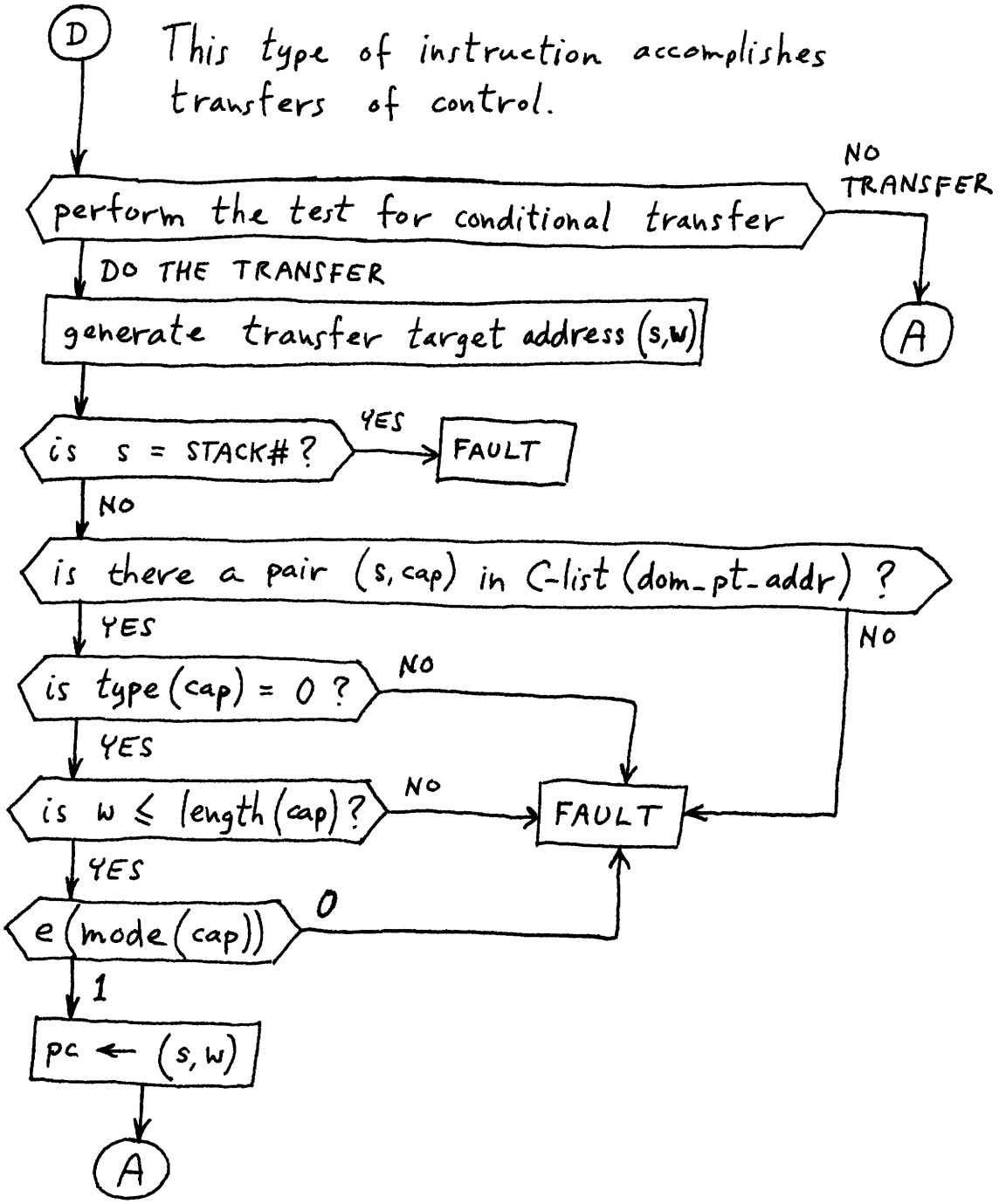


Figure A1-2, part 5. State transition rule - transfers of control.

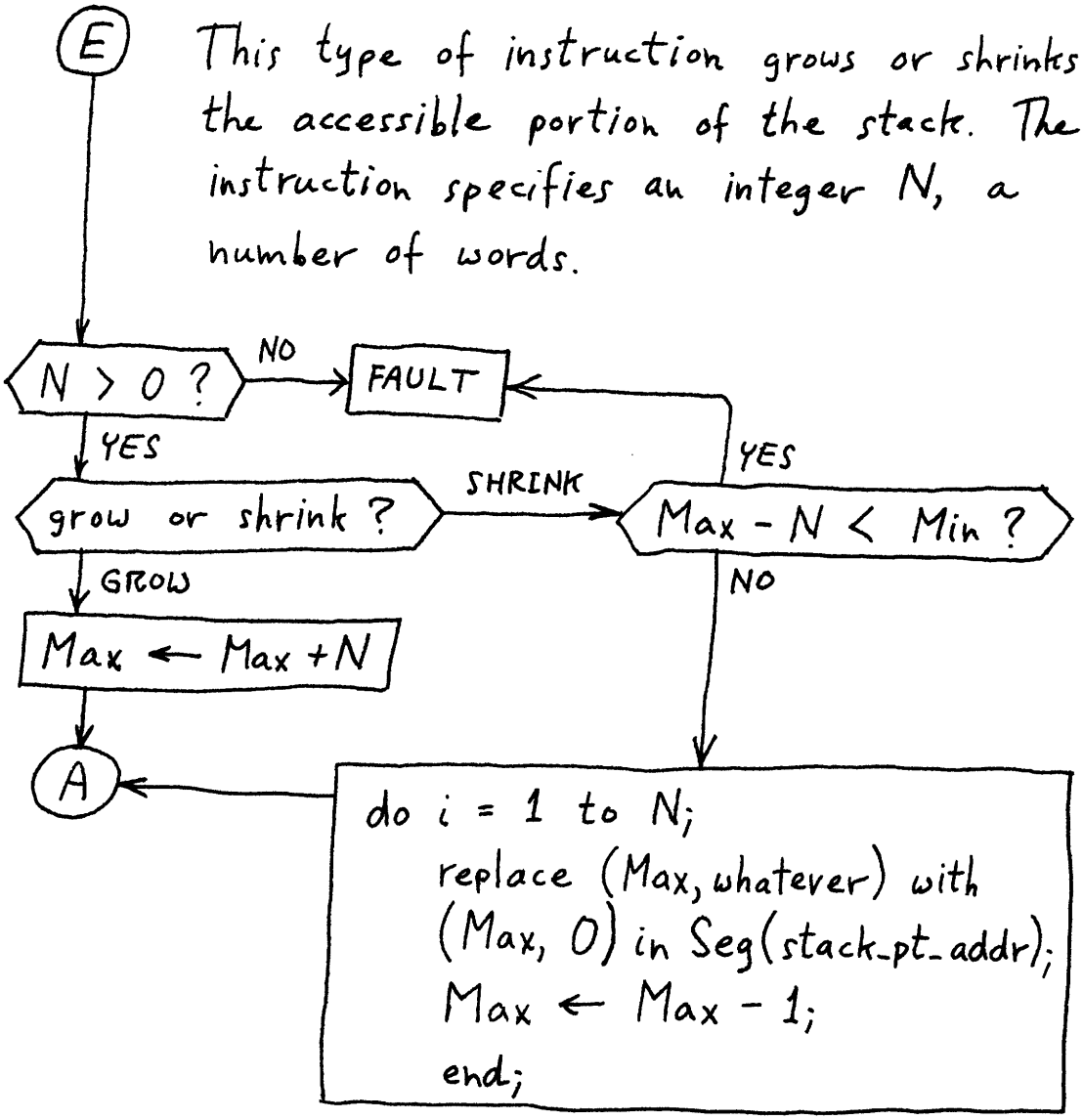


Figure A1-2, part 6. State transition rule—stack manipulation.

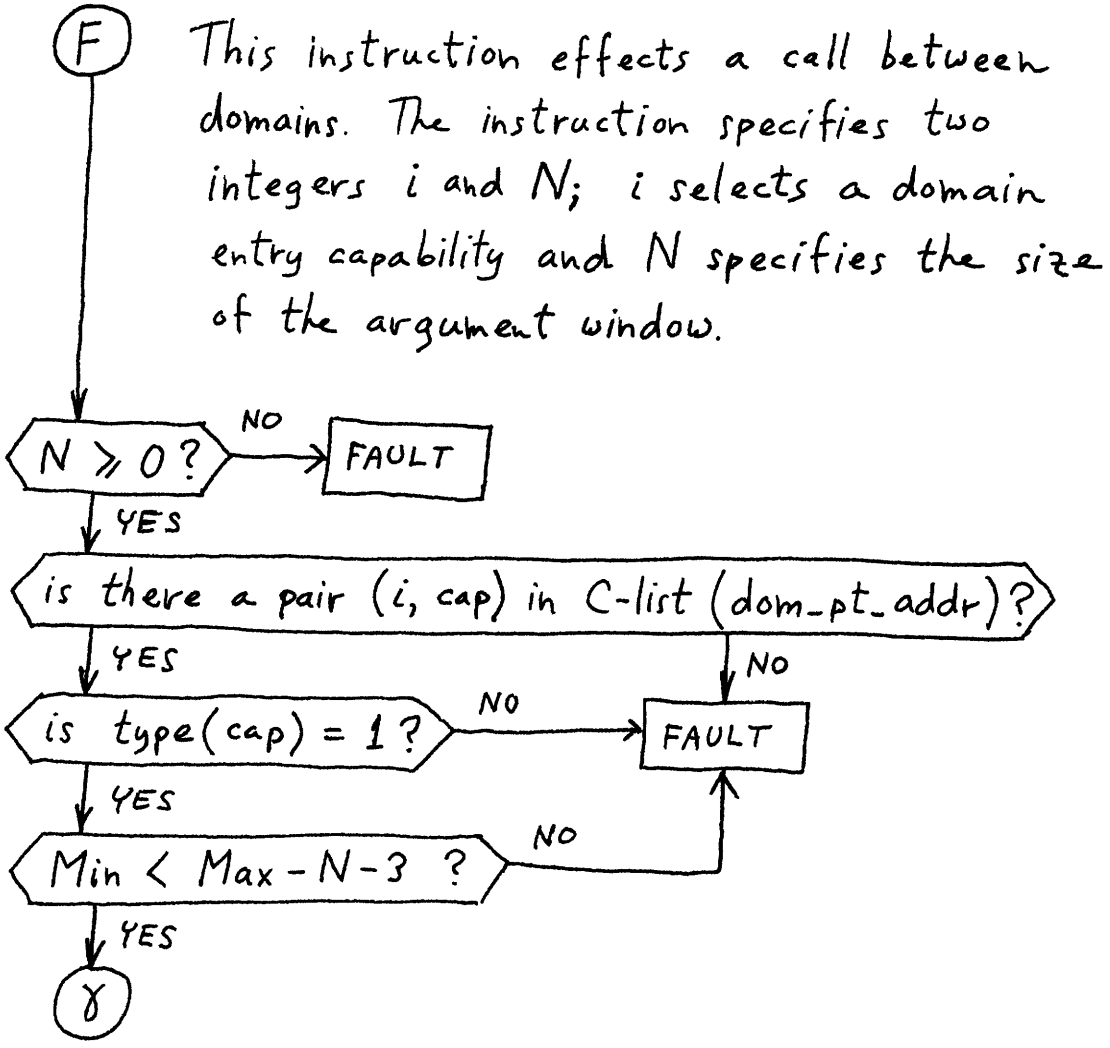


Figure A1-2, part 7(a). State transition rule - call-domain instruction.

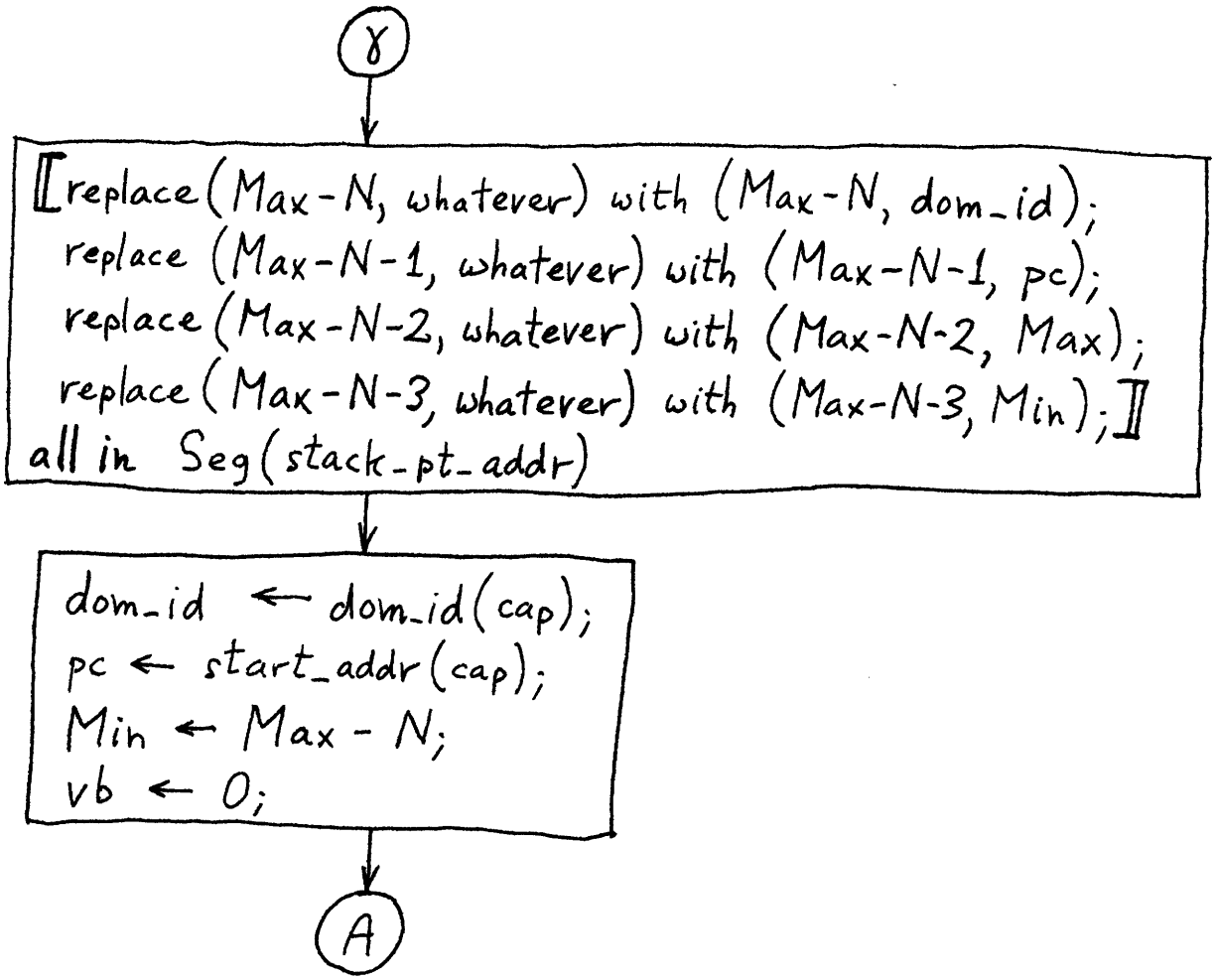


Figure A1-2, part 7(b). State transition rule - call-domain instruction.

entry capabilities are triples of the form (type,dom_id, start_addr), subject to the constraint that the first component, i.e., type(capability), must have the value 1. The other components of a domain entry capability are referred to as dom_id(capability) and start_addr(capability). The component dom_id is the unique identifier of the called domain. The component start_addr is a two-part address of the form (seg#,word#) which is interpreted to be an address in the address space of the called domain.

The call-domain instruction uses four words of the sectioned stack to save the values of dom_id, pc, Min, and Max from the process state. These are used to effect a subsequent return-domain. By convention, the calling program allocates four words just before the argument window for this purpose. (In a real implementation, Min and Max might fit into one word; but that isn't too important.)

Figure A1-2, part 8 shows the state transition rule for the return-domain instruction. The saved values of dom_id, pc, Min, and Max are recovered from the stack; stack words are zeroed if Max is to be reduced; and the process state components dom_id, pc, Min, and Max are replaced.

The reader should note that in figure A1-2, parts 7 and 8, whenever dom_id is modified the validity bit vb is set to 0. Thus, when the next domain binding test is performed (figure A1-2, part 1), the ADT will be searched to find the absolute address of the base of the page table of the C-list of the new domain.

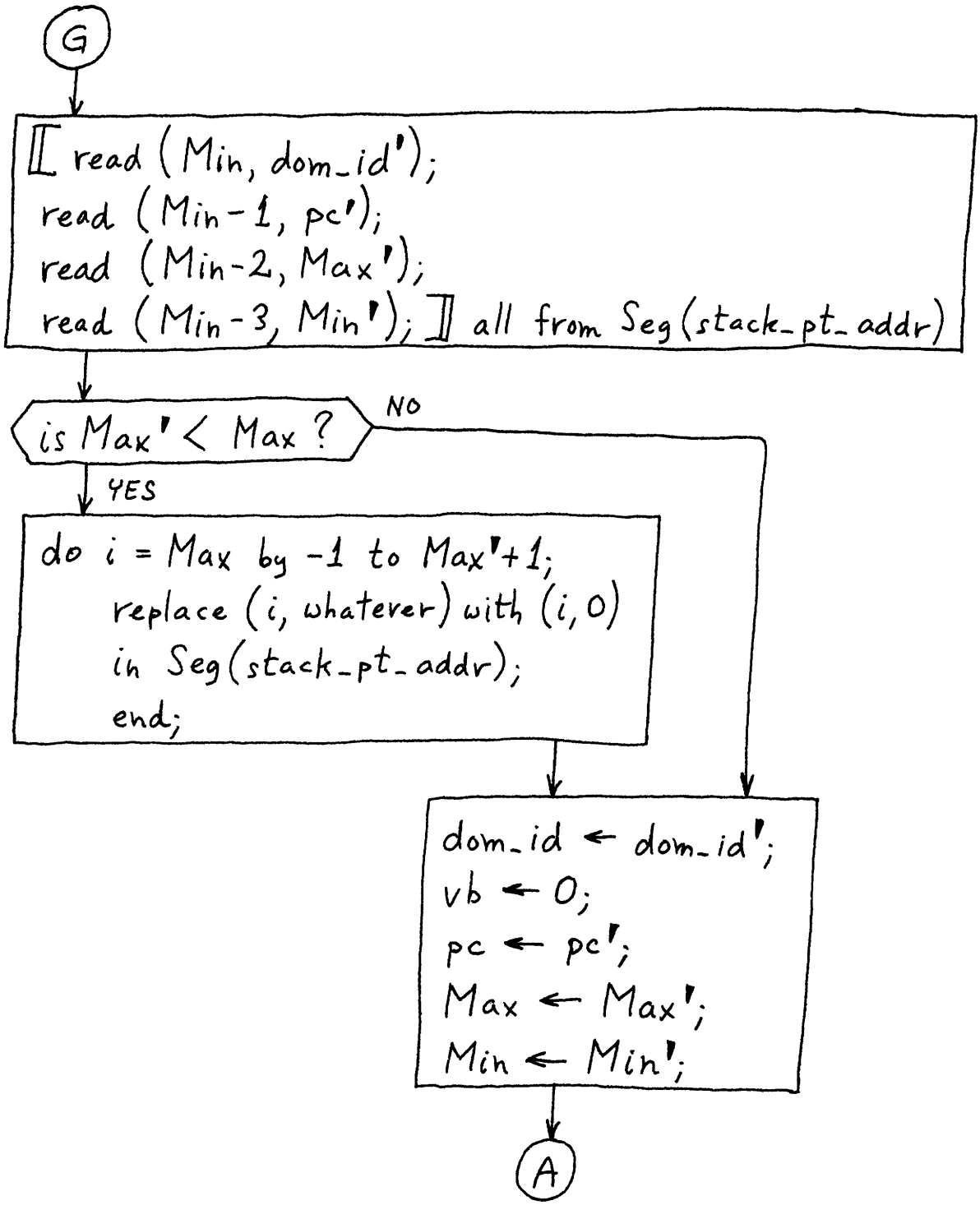


Figure A1-2, part 8. State transition rule - return-domain instruction.

Figure A1-2, part 9 shows the state transition rule which governs the processor's responses to FAULTs. The process state, including the reason for the fault, is stored in the sectioned stack, and control is transferred to an appropriate program. The state transition rule recognizes two classes of faults: system faults (e.g., DOMAIN FAULT) and ordinary faults (e.g., fixedoverflow). In cases of ordinary faults, control of the processor is transferred to the constant address ADDR2 where, in every domain, there is a program (called the fault interceptor module in Multics) that knows how to respond to the fault -- typically by searching for an enabled fault handler, i.e., a program in the domain which has previously declared it should be notified of certain faults, should they occur.

In cases of system faults, the state transition rule moves the process into a new domain. The new domain is selected by the functions SYS_DOM(fault#) and SYS_DOM_PT (fault#); and control is transferred to the location selected by the function ADDR1(fault#). These functions are wired into the processor, or their values are set in switches in the processor.

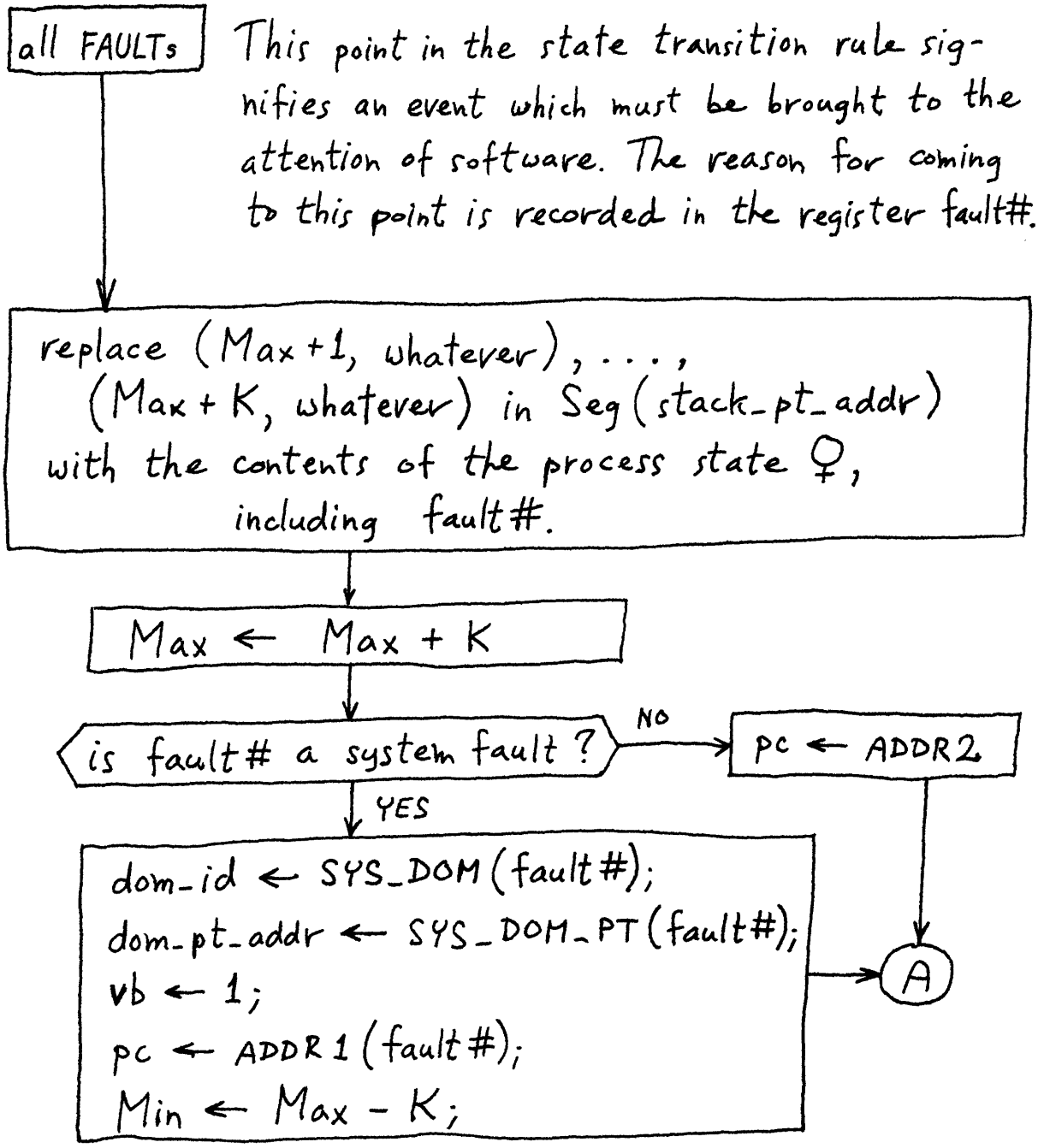


Figure A1-2, part 9. State transition rule - fault logic.

Appendix 2

ADT Reference and Management

The purpose of the Active Domain Table is to allow processors to find quickly the base address of the page table of the C-list of a given domain, identified by that domain's unique identifier. A processor making such a reference to the ADT will hash the unique identifer, obtaining thereby an index into a table of pointers at the base of the ADT. We are assuming that every processor knows where the ADT is located in memory (this information could be set in switches wired into the processor, or taken from a register set by the operating system), and we are assuming that the ADT is an unpaged segment. Each pointer from the table of pointers leads to a list of blocks in the ADT, each of which contains the unique identifier of an active domain, and its corresponding dom_pt_addr. The processor searches this list looking for a block containing the unique identifier it's interested in, and either finds it and proceeds, or doesn't and faults. Figure A2-1 shows one list of blocks in the ADT.

If the desired block is not found, the processor takes a fault into an operating system domain responsible for maintaining the ADT. Figure A2-2 shows this domain, and one other operating system domain. The fact that fault events are occurring is notated with the arrow on the left pointing at the ADT domain. Names of domains are written inside the

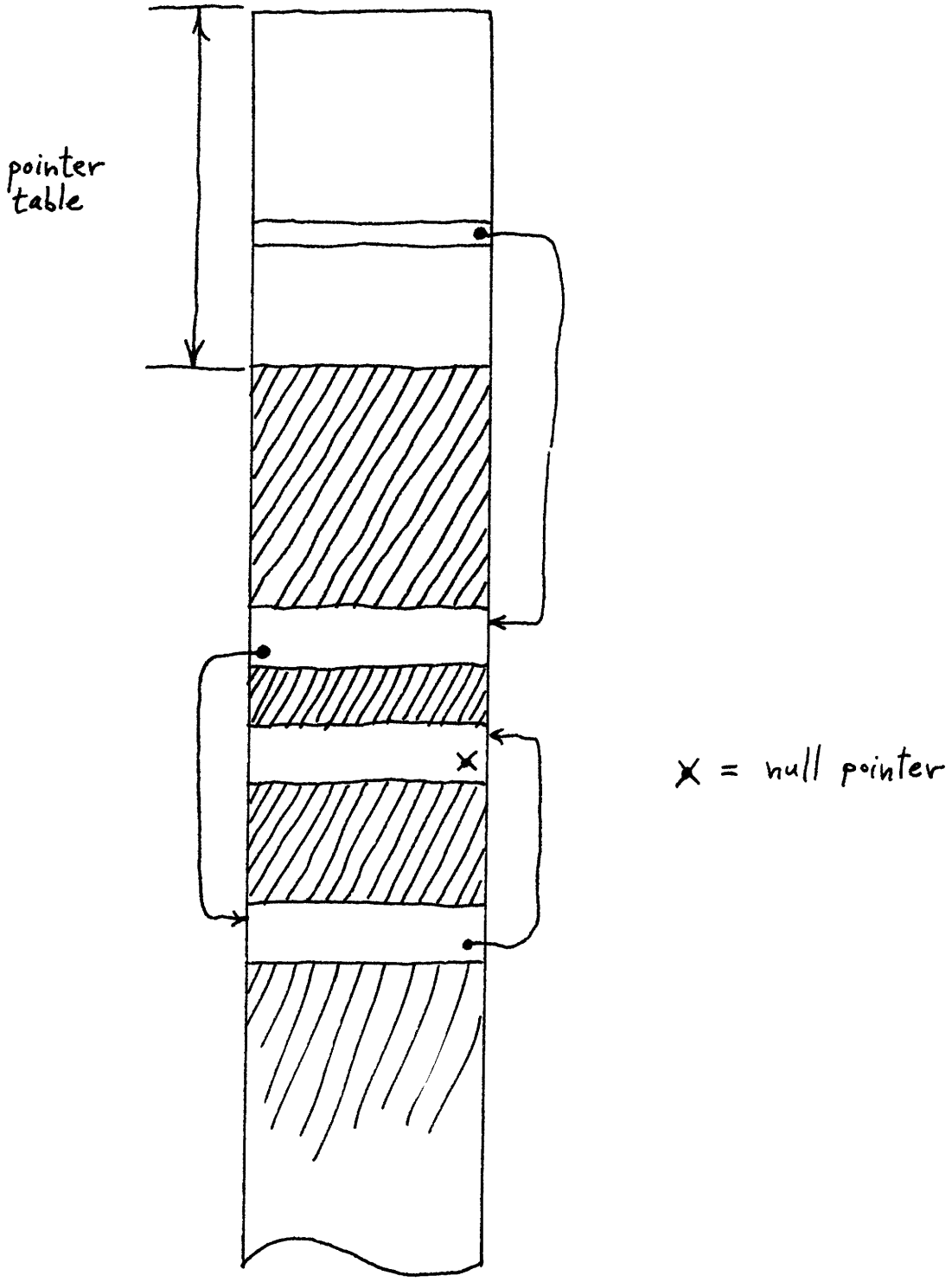


Figure A2-1. A list of blocks in the Active Domain Table.

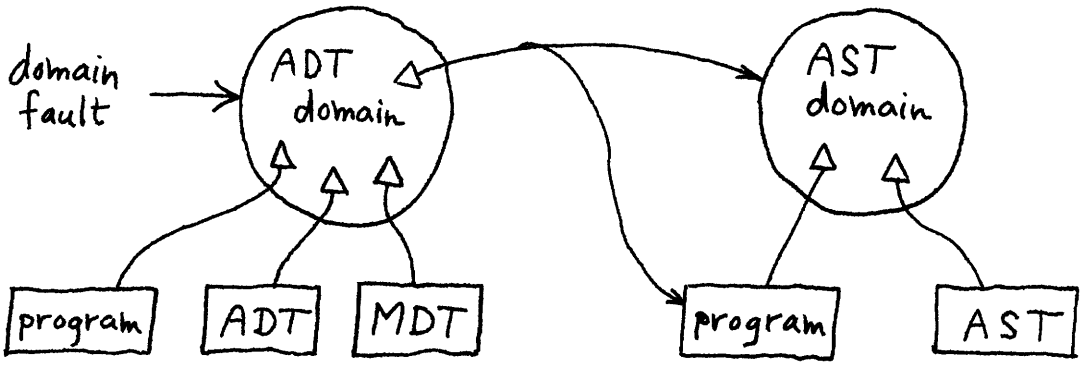


Figure A2-2. Operating system domains involved in handling a domain fault.

circular figures that represent the domains. The AST domain maintains the Active Segment Table (AST), which contains page tables and file maps of active segments. We can associate a valid page table address with a segment just when it is active, so the strategy for handling the domain fault is to activate the C-list segment of the selected domain, and load a block into the ADT holding that domain's dom_id and dom_pt_addr.

From information recorded by the processor in the stack at the time of the fault, the program in the ADT domain retrieves the dom_id of the domain that wasn't active. The ADT domain uses the Master Domain Table (MDT) to look up the unique identifier of the C-list segment of the desired domain. (The MDT is a large, paged segment.) The ADT domain calls the AST domain to make active the C-list segment, specified by its unique identifier.

The AST domain determines whether the specified segment is active, and if not it is made active, and in any event the AST domain returns the C-list segment's page table address. The strategies by which the AST domain operates are described in Appendix 3.

After the return from the AST domain, the ADT domain searches for a free block in the ADT, frees one if necessary, loads the new dom_id and dom_pt_addr into the block in the ADT, threads the block onto the appropriate list, and returns from the fault. After the return from the fault, the

process once again searches in the ADT, finds what it's looking for, and thus can continue.

The task of freeing blocks in the ADT to make room for new information is the price that must be paid to share the ADT's services among all the processes in the system. A global lock on the ADT is required to properly synchronize ADT reference events and ADT management events, and successful searches in the ADT must set a bit to inform the program in the ADT domain that an ADT block has been used.

The global lock on the ADT can be explained most easily in terms of a read-alter-rewrite memory cycle. This is a service performed by a memory controller for a processor, wherein a processor receiving the service is guaranteed that no other processor is accessing the addressed word at the same time. The service consists of reading the addressed word and delivering it to the processor, waiting for the processor to alter the word, and re-writing the altered word in the address from which it was first read. We will assume that the first word of the ADT segment is devoted to this global lock, and we will call it ADT_lock. When ADT_lock is greater than zero, a number of processors are searching in the ADT. When ADT_lock equals zero, the ADT is not in use. When ADT_lock = -1, the ADT is being modified. Only one process is allowed to modify the ADT at a time. Processors searching in the ADT are given priority over a processor which wants to modify the ADT.

Figure A2-3 shows the state transition rule of our processor which locks the ADT by adding 1 to ADT_lock, provided ADT_lock $\geq$ 0. This figure is an expansion of part of figure A1-2, part 1, where the search of the ADT was introduced. Figure A2-4 shows the state transition rule for two new instructions, Lock_ADT and Unlock_ADT, which a process will issue when it needs to modify the ADT and is finished modifying the ADT, respectively.

Figure A2-5 shows a PL/I-like declaration for a block of the ADT. The component block.next is the offset in the ADT of the next block to be searched if this block does not contain the desired dom_id. When block.next = 0, there is no next block. A backward pointer block.back allows quick removal of a block from a search list. When a searching processor finds the dom_id it needs in a block, the block.-used bit is set to 1. The pointer block.c_next defines a cycle through all the blocks that can be removed from the ADT -- a cycle which is followed by a process trying to free a block, according to the algorithm shown in figure A2-6.

The algorithm considers in turn each of the blocks which are threaded together by the component block.c_next. The algorithm selects for removal blocks which have their "almost" bit on and their "used" bit off. The algorithm turns a block's "almost" bit on when the C-list associated with the block no longer has any pages in the fast memory device. The ADT must be big enough so that processes won't hang up for long

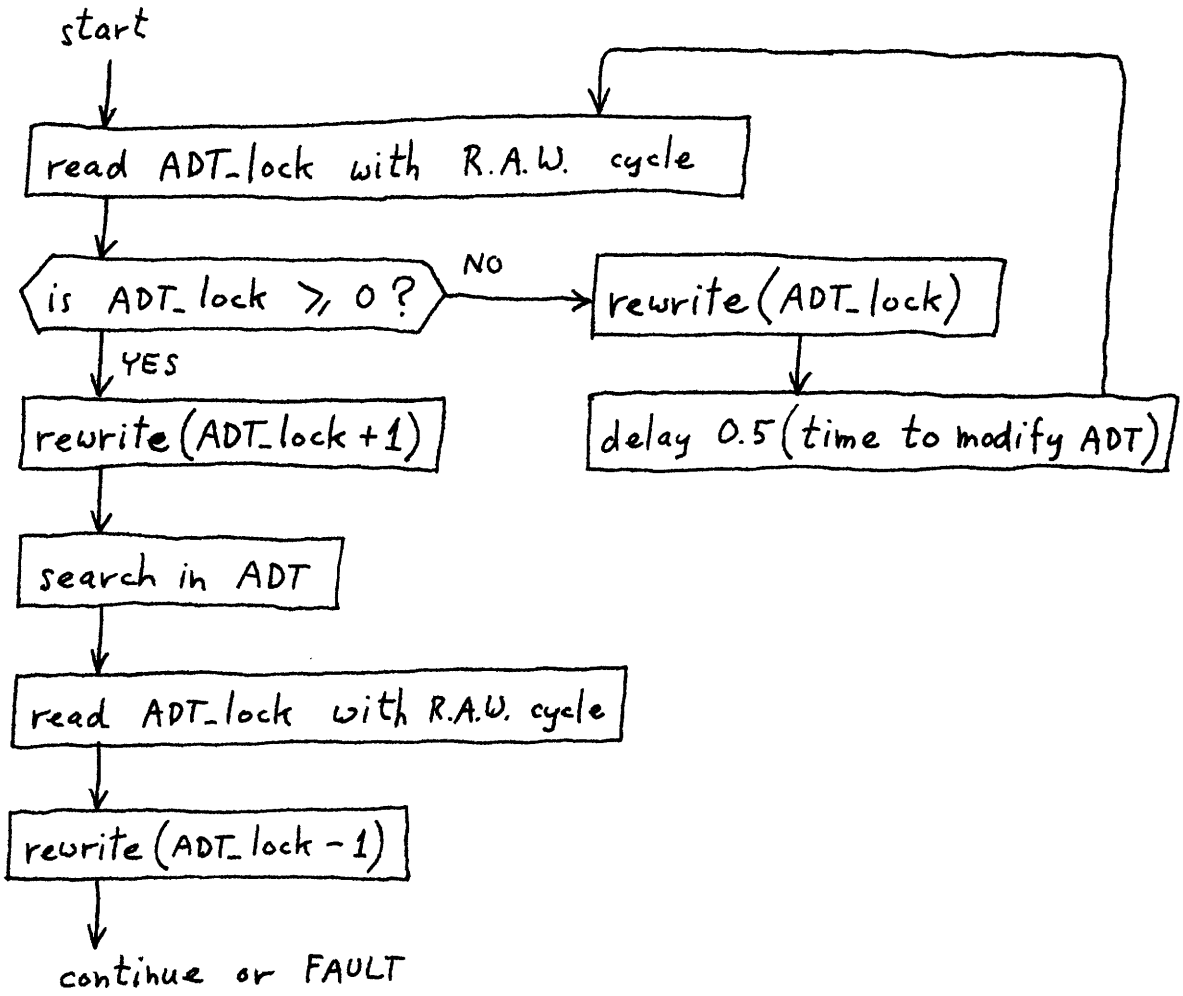


Figure A2-3. State transition rule -
locking the ADT to search it.

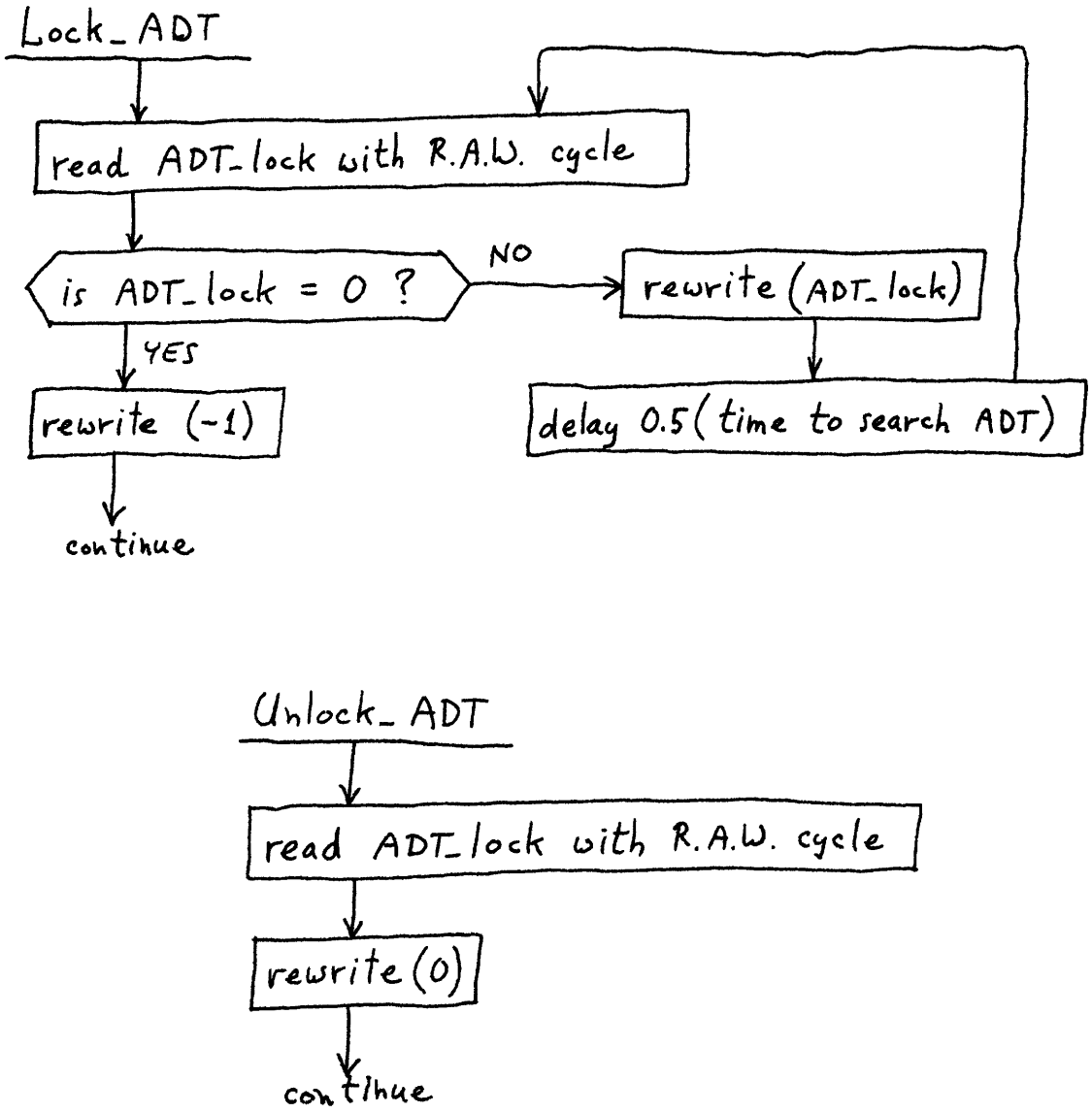


Figure A2-4. State transition rule - Lock_ADT and Unlock_ADT instructions.

```
declare 1 block based(p),
          2 dom_id integer,
          2 dom_pt_addr integer,
          2 next integer,  /* next in search list */
          2 used bit(1),  /* 1 = recently found by search */
          2 almost bit(1),  /* 1 = ready to go */
          2 back integer,  /* previous in search list */
          2 c_next integer;  /* next in review cycle */
```

Figure A2-5. Declaration of ADT block.

```
declare s semaphore init(1), (p,q) pointer;
   ⋮
P(s);
do while ("1"b);
     if p -> block.used then do;
          p -> block.used = "0"b;
          p -> block.almost = "0"b;
          end;
     else if p -> block.almost then do;
          Lock_ADT;
          if ¬p -> block.used then do;
               call free_block(p);
               Unlock_ADT;
               q = p;
               p = ptr(seg#(ADT),p -> block.c_next);
               go to found_one;
               end;
          else do;
               Unlock_ADT;
               p -> block.almost = "0"b;
               end;
          end;
     else if pages_in_core(p -> block.dom_pt_addr) = 0
          then p -> block.almost = "1"b;
     p = ptr(seg#(ADT),p -> block.c_next);
     end;
found_one:  V(s);
            call unfreeze_activation(q -> block.dom_pt_addr);
               ⋮
```

Figure A2-6, part 1. Algorithm to find an old domain in ADT.

Notes.

(1) free_block(p) disentangles the block at p from the ADT search list it's in.

(2) seg#(ADT) is a constant, the permanent segment number of the ADT in the ADT domain.

(3) pages_in_core is a call to the AST domain to be told the number of pages of the specified segment which occupy fast memory devices.

(4) unfreeze_activation is a call off to the AST domain to permit the specified page table to be removed from the AST.

(5) P(s) and V(s) are semaphore operators which provide mutual exclusion of processes modifying the ADT.

Figure A2-6, part 2. Algorithm to find an old domain in ADT.

searching for blocks associated with recently unused C-lists, but this should not be an irksome constraint.

The program that begins at the label found_one will place new values of dom_id and dom_pt_addr into the found block, and then thread that block onto the appropriate search list in the ADT. If the block is threaded onto the end of the search list, the ADT need not be locked during the threading.

Appendix 3

Memory Multiplexing

We are assuming that the technique of paging [Ki62] is used to multiplex the high speed memory device among all the segments in the system. Every segment which can have a page in the high-speed memory must have a page table in the AST, and such segments are called active. In this respect, the system here being described is very similar to Multics [Ben72]. The purpose of this appendix is to discuss the multiplexing of the AST among all segments to which references are being generated by running processes.

Every segment capability in every C-list has a fault bit in addition to the four components (type,mode,length,addr) introduced in Appendix 1. This fault bit is a validity bit for the addr component: when the bit is 0, addr is valid: i.e., addr is the address of the page table of the segment. But when the fault bit is 1, the segment might not be active--that is, it might not have a page table and so the addr component is regarded to be invalid and so the processor faults if it should need to use this capability. The fault is called a segment fault and the system's response to the segment fault is to make the referenced segment active, store the correct page table address in the addr component of the capability that caused the fault, and turn off that capability's fault bit. Thus part of the job of the fire-wall domain is helping the system respond to a segment fault.

Figure A3-1 shows the domains of the operating system which are involved in responding to a segment fault. The domain to which the process goes when the fault occurs is the Table of Contents (ToC) domain. From information recorded by the processor in the stack at the time of the fault, the program in the ToC domain retrieves the dom_id of the domain where the process was executing and the seg# of the segment being referenced. From the Master Domain Table, the program in the ToC domain looks up the segment number of the C-list in the firewall domain, which by convention is also the segment number of the table of contents segment for that C-list in the ToC domain. The Master Domain Table can be regarded as a function {(dom_id,(!_id,seg#))} which maps domain unique identifiers to pairs whose first component, !_id, is the unique identifier of the C-list segment, and whose second component, seg#, is the segment number of the C-list in the firewall domain. We are assuming that every C-list has a segment number in the firewall domain which does not change as long as the C-list exists. This assumption is not essential--we could arrange to multiplex the name-space of segment numbers in the firewall domain over a larger collection of C-lists--but the assumption makes our description simpler.

Armed with the segment number of the table of contents segment, the program in the ToC domain looks up, in the appropriate table of contents, the unique identifier of the

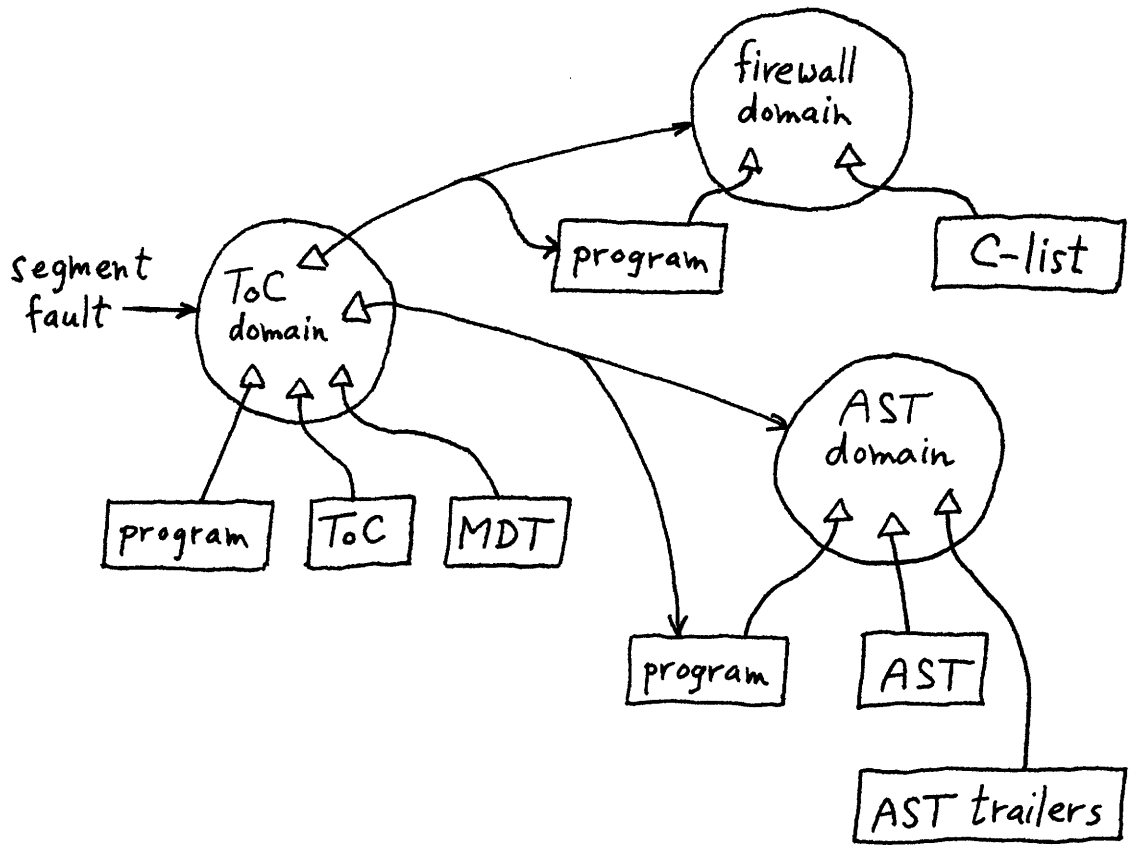


Figure A3-1. Operating system domains involved in handling a segment fault.

segment to which a reference caused the fault. Then the ToC domain calls the AST domain to activate the segment, specifying the unique identifier of the segment, and the C-list segment number and seg# of the segment, so the AST domain can remember in its AST trailers segment the locations of all active capabilities (i.e., with fault bits off) for active segments. The AST domain returns the page table address of the segment, now guaranteed to be active (though it could have been active before). The ToC domain calls the firewall domain to turn off the fault bit in the segment capability and replace the addr component of the capability with the new true page table address supplied by the AST domain. After the firewall domain returns, the ToC domain returns from the fault, and the process will reference the segment thereafter without generating a segment fault, as long as the segment remains active.

While the services of the AST are multiplexed among all the segments in the system, some segments are active all the time. For example, operating system domains' C-lists are always active, as are the MDT and the table of contents segments of operating system domains. After a segment fault, the ToC domain will cheerfully generate another segment fault referencing an inactive table of contents segment for the domain in which the first fault occurred; because the table of contents segment of the ToC domain, which is used to respond to the second segment fault, is always active.

The C-lists of active domains are also required to be active segments, so that the dom_pt_addr's stored in the ADT will remain valid.

When the AST domain needs space in the AST for a new page table, it will de-activate a segment (not, of course, any segment marked "always-active"). C-list segments are handled differently from other segments in this regard, because the addr components of segment capabilities are pointers into the AST. Because of this, C-list segments are de-activated when the domains they define are de-activated (i.e., no longer represented in the ADT). We will return to C-list segments shortly. When the AST domain needs space in the AST, it chooses for de-activation a non-C-list segment with the smallest number of pages in core. After initiating the removal of the segment's remaining pages from core (if any), the program in the AST domain reads the "trailers" associated with the page table to determine which C-lists hold segment capabilities pointing to the page table. The AST domain calls the firewall domain to get the fault bits in these segment capabilities turned on. If the system's central processors have associative memories for holding segment capabilities, the AST domain broadcasts a request that they be cleared (or, to be economic, that any capability in an associative memory pointing to the given page table be cleared). After following this procedure, the page table is available for re-use.

In the above procedure for de-activating a segment, it is necessary to turn on fault bits in segment capabilities which contain the page table address of the segment. The page of the C-list containing the segment capability must be paged in for the firewall domain to turn the fault bit on. But this use of the C-list page must not affect the C-list page usage reported by the pages_in_core entry point of the AST domain called from the algorithm of figure A2-6, because that usage figure is intended to reflect activity of running processes bound to the domain defined by the C-list. So the pages_in_core entry point will not tell the true number of pages in core, but rather the number of pages used as C-lists, as opposed to used as data. The implication for the processor design is that every page table word has two usage bits (rather than one, as in Multics). One usage bit is set for any usage (as in Multics); the other is set for usage to retrieve capabilities from C-lists. These latter usage bits provide the basis for the number returned by pages_in_core.

When a domain is de-activated, the AST domain is informed by a call from the algorithm of figure A2-6. The AST domain will de-activate the C-list segment, but only after setting all the fault bits in its segment capabilities back on and removing all references to the C-list segment from the AST trailers. This requires a call to the firewall domain.

This completes our description of memory multiplexing, except we have not yet described the special status of sectioned stack segments in the memory multiplexing scheme. This special status is required because of the involvement of sectioned stacks in processor multiplexing, to which we now turn.

The operating system includes a domain dedicated to multiplexing the system's processors among the processes which are eligible to run. This is the traffic controller domain, and it is entered by two major methods: either through a call to its entry point named wait, or through a timer runout fault. The process that is directed to call this entry point or takes this fault is giving away its processor. The program in the traffic controller domain selects a new process to give the processor to, and prepares for the process switch by insuring that the new process' sectioned stack segment is active. This is trivial if the process is one from a pool of so-called loaded processes: loaded processes have active stacks by definition. If the selected process is not loaded, the traffic controller calls the AST domain to activate the selected process' stack segment, and adds the selected process to the pool of loaded processes.

Then the traffic controller issues its process-exchange instruction and the processor stores in memory some of the components of the process state, and reads from different words of memory new values for these same components. The

locations stored into and read from are specified by registers of the process chosen by convention. The components of the process state which are stored and modified are registers, stack_pt_addr, Min, Max, and proc_id. Since the first four components of the process state, i.e., dom_id, vb, dom_pt_addr, and pc; are not modified, the processor continues to execute the program in the traffic controller domain. The fact that the processor is running a new process will not be manifest until the process leaves the traffic controller domain, which is done by a return. This is a return from the wait entry point of the traffic controller or a return from a timer runout fault--whichever is indicated by the contents of the sectioned stack of the new process.

Before the new process leaves the traffic controller, it examines a per-processor data base to see if the old process should be unloaded, and if so it calls the AST domain to accomplish this. (The process can find out what processor it's running on by examining a fixed field of the state component fault#.)

The AST domain will keep sectioned stacks active until a notification through a call from the traffic controller domain to de-activate them.

There is a small, but crucial protection issue here, which is the privileged hardware instruction process-exchange. The processor must refuse to execute this instruction unless

the executing process is bound to the traffic controller domain, which the processor can detect by examining the process state component dom_id. The value of dom_id for the traffic controller domain can be wired into the processor.

Appendix 4

Argument Segment Primitives

The purpose of this appendix is to define precisely the operating system primitives which allow domains to share argument segments.

The primitive which is used to create a capability for the argument segment in the domain to be called is pass-segment(seg#,dom#,mode,copy_flag,new_seg#,code). The first argument, seg#, is the segment number of the argument segment in the calling domain. The argument dom# is the capability number of a domain entry capability, which defines the domain to be called. The argument mode is a 3-bit string which defines the mode component of the capability to be created in the called domain, except that this mode of usage to be given to the called domain may not authorize greater access privileges than are available in the calling domain. In other words, the pass-segment primitive will not create new access privileges; it only extends privileges which existed previously. The fourth argument, copy-flag, is a bit which will be remembered in the table of contents segment of the called domain and used to authorize or deny any further passing of the argument segment to domains called by the called domain. In short, copy-flag is a bit which authorizes additional capabilities for the argument segment. The fifth argument, new-seg#, is an output argument which will be the segment number of the

argument segment in the called domain. The last argument, code, is an output which signals success.

The pass-segment primitive is implemented mainly in the Table of Contents domain. The program there accesses the table of contents of the calling domain to see that the copy flag of the selected segment is on for the calling domain, and that the mode of access to be given is contained in the mode of access posessed by the calling domain. Provided this is so, the program obtains the dom_id of the called domain from the table of contents entry for dom#, and uses the Master Domain Table to obtain the segment number of the C-list and table of contents of the called domain. In the table of contents of the called domain, the program obtains a new segment number and records the unique identifier of the segment being passed (obtained from the table of contents of the calling domain), the mode being allowed, and the copy flag specified; all associated with the new segment number. The ToC domain calls the firewall domain to create the capability in the called domain. Finally, the ToC domain returns the new segment number to the caller of pass-segment.

To facilitate the reclaiming of passed segments, the pass-segment primitive records, in the entry for the passed segment in the calling domain's table of contents, the identity of the called domain and the segment number of the segment there. In other words, the pass-segment primitive grants

access to the segment by creating a new capability, but "with a string attached." The grantor can "pull on the string" at any time and take the new capability back. The philosophy behind attaching a string to capabilities is simply the realization that a capability might be used unexpectedly to gore the grantor's ox; coupled with the judgement that the computer system should serve the purposes of the user whose ox is being gored through the use of the new capability, rather than the purposes of the capability-borrowing ox-gorer. Clearly, this is not a technical judgement, but a social one: whose purposes shall be served first? Different workers have made different judgements on this question, see for example Vanderbilt [Va69] and Fabry[Fa68], where are described systems in which it is not possible to take back capabilities once they have been granted.

The primitive which reclaims a passed segment capability is reclaim-segment(seg#,dom#,code). The argument seg# is the segment number of the argument segment in the calling domain. The argument dom# is the capability number of a domain entry capability, which defines the domain from which a capability for the passed segment is to be removed. The argument code is an output which signals success.

The reclaim-segment primitive is also implemented mostly in the Table of Contents domain. The program there accesses the table of contents of the calling domain to verify that

seg# is a segment number, that dom# is the capability number of a domain entry capability for calling a domain which we will refer to as "the called domain," and that the specified segment was in fact passed to the called domain. Provided all this is so, the program will proceed to reclaim the capability from the called domain, and from all domains to which the capability was passed by the called domain (if the original call to pass-segment specified a copy-flag of 1), and from all domains to which the capability was passed by those domains, and so on. The program in the Table of Contents domain can find all these domains because of the "strings" recorded in tables of contents by the pass-segment primitive. These "strings" define a tree whose nodes are capability slots in domains' C-lists where are found passed capabilities to be reclaimed. A simple recursive program can follow the tree to its ends. At each node of the tree, a call to the firewall domain is made to remove the capability associated with the node from its C-list.

There is an interesting process synchronization problem here, closely associated with the social question mentioned previously: should the purposes of the capability-reclaimer, or the purposes of a would-be passer of the capability, be served first? Our answer to this question is expressed as a priority given to reclaiming processes over passing processes. This priority is implemented with a list of segment unique identifiers of segments whose capabilities are being reclaimed,

maintained by the ToC domain. The pass-segment primitive will consult this list and make the executing process wait, if it has been asked to pass a capability for a segment on the list. For each segment on the list, there will be a process executing the reclaim-segment primitive, reclaiming a tree of capabilities for the segment. When that process has finished, it will remove the entry it placed in the list, and wake up any processes which the pass-segment primitive directed to wait because of the entry on the list. The awakened process might find that the capability specified by the seg# argument to pass-segment has been removed by the reclaiming process.

Finally, the primitive which validates the segment number of an argument segment is is-arg-seg(seg#,code). The argument seg# is allegedly the segment number of an argument segment passed from the calling domain. The argument code is an output whose value signals whether the seg# is valid, in the above sense. The is-arg-seg primitive is implemented mainly in the Table of Contents domain. The program there verifies that seg# is the segment number of an argument segment by examining the table of contents segment of the called domain. (Recall that the "called domain" is the one which calls is-arg-seg.) The table of contents segment also records the identifier of the domain from which the segment was passed. It is necessary to check that this is the calling domain, i.e., the domain that directed the process to call the domain that

called is-arg-seg. To do this, it is necessary to examine a protected word of the sectioned stack. To allow this, there is a privileged domain of the operating system in which every sectioned stack is accessible as an ordinary segment. This domain is called the stacks domain. The program there, when called by the is-arg-seg primitive, will return the unique identifier of the domain which caused the second inter-domain call before the call to the stacks domain. (The second call before the call to the stacks domain is the call into the called domain, since the called domain called the ToC domain, and the ToC domain called the stacks domain.)

In fact, the stacks domain is used by the pass-segment and reclaim-segment primitives also, to find out the identifier of the domain from which they were called. The call to the stacks domain is required because the stored identifier of the calling domain is made inaccessible by the sectioned stack.

Appendix 5

Taxonomy of Responsibilities of Programs

The purpose of this appendix is to define the responsibilities of programs. We assume that the institutions or persons who own the programs will be required by society to shoulder the responsibilities incurred by the programs. The events which generate responsibilities for programs are simply the events defined by the state transition rule (the reading and writing of information in segments, and the call-into to and returning from programs), and the events defined by calls to the primitives of the operating system.

In defining the responsibilities, we use some words and phrases which require prior explanation. We refer to control of processes, which is exercised by programs, because the state transition rule requires processes to take instructions from the program designated by the program counter pc until some instruction causes a transfer of control to another program. We refer to the release of information, which includes sending information out of the computer to a user, making a copy of information in the computer (e.g., in an argument window or an argument segment), and passing an argument segment to a called domain. We mean the term release to include also actions whose side effects will lead to future movement of information, such as calls on operating system primitives to add a term to a segment's

access control packet, or to add a term to the seg-limit component of a domain's access control packet, or to redefine f(r) or d(r) in a restriction's access control packet, or to modify the restriction set of a segment or a process. We refer also to the distribution of information, by which we mean a set of releases (possibly empty) of that information. By use of information we mean the undertaking of any computation with the given information as input.

By maintenance of information we mean the avoidance of any modifications to the information that leave it in an incorrect or inconsistent state. When a block of data has a lock bit associated with it, one possible rule for examining or manipulating the block is to set the lock bit first, and reset it when done; and to wait for it to be reset whenever an attempt is made to set it when it's already set. This convention is an example of proper maintenance. "Proper" is a somewhat fuzzy word (like "reasonable"); whenever we use it we open an area of discourse for lawyers.

Now we turn to enumerating the responsibilities of programs.

A program that directs a process to read words from segments (or the process' stack) and compute results from them, is responsible for so directing the process. Similarly, a program that directs a process to write words in segments (or in its stack) is responsible for so directing the process.

In other words, the program is responsible for what it makes a process do.

A program that calls another program is responsible for releasing control of the executing process, and it is responsible for the release of the arguments of the call to the called program. It is responsible for the content of the arguments only to the extent that it computed them. The called program becomes responsible for the proper distribution, use, and maintenance of the arguments it received, and for the proper control of the executing process. If the called program is in another domain, all the programs in the called domain become responsible for the proper distribution, use, and maintenance of the arguments. This includes the contents of the argument window, and all argument segments passed to the called domain.

A program that calls an operating system primitive is responsible for the effects and the side effects of the call. For example, a program that exercises the authority of an office is responsible for proper use of that authority.

A program that returns to its caller is responsible for releasing control of the executing process, and it is responsible for the release of results to the calling program. It is responsible for the content of the results only to the extent that it computed them. The program to which control is returned becomes responsible for the proper distribution,

use, and maintenance of the results it receives, and for the proper control of the executing process. If the program returned to is in another domain, all the programs in that domain become responsible for the proper distribution, use, and maintenance of the results.

This enumeration of responsibilities is included here because it is necessary to be able to say who is responsible for potentially harmful events (see section 2.7). But the reader should not jump to the conclusion that this enumeration makes possible the analysis of all harmful events. System crashes, for example, are difficult to analyze because of the complex patterns in which the simple events described above are combined.

Biographical Note

Leo J. Rotenberg was born in January 1944 in Philadelphia, Pennsylvania. He attended the Central High School of Philadelphia and was graduated in June 1961 with the Degree of Bachelor of Arts. He attended the Massachusetts Institute of Technology and earned the Degrees of Bachelor of Science in Mathematics in February 1965 and Master of Science without specification of field (for work in recursive function theory) in June 1966. He studied and helped design computer operating systems while a research assistant at MIT Project MAC. He studied Mathematics at the University of Wisconsin at Madison and was employed there as a teaching assistant. He returned to MIT in September 1968. In August 1969 he abandoned and destroyed work on a doctoral thesis, "Assured Parallel Computation on Structured Data", because of the clear usefulness of such technology for aiming Anti-Ballistic-Missile systems. He has published once [Ro71]. He is a member of Sigma Xi, the Association for Computing Machinery, and the American Civil Liberties Union.