M0105

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

December 12, 1968

SENSITIVE ISSUES IN THE DESIGN OF MULTI-USE SYSTEMS*

by F. J. Corbató

Abstract

An attempt is made to isolate and discuss fifteen critical issues ranging from technical to managerial which affect the complexity and difficulty of constructing computer systems which serve multiple users. The present memo is a slightly edited version of a transcript so that it still had the rough colloquial flavor or the oral presentation. Nevertheless, it is being distributed as is because of the current interest in technical management experience gained developing large-scale software systems.

---

* The present paper is an edit from a transcript of a talk presented at a technical symposium on Advances in Software Technology held in February, 1968, at the opening of the Honeywell EDP Technology Center, Waltham, Massachusetts.

SENSITIVE ISSUES IN THE DESIGN OF MULTI-USE SYSTEMS

by Professor F. J. Corbató

Multi-use computer systems are potentially large, both in the amount of effort required to put them together and in the number of users trying to use them at the same time. Functionally, such systems serve many people with diverse interests simultaneously in real-time. As systems, they usually include a great deal of software; they should also include both data processing and time-sharing capabilities.

Generally speaking, there are two extreme forms of multi-use systems; each type can be illustrated in a graph of system cost versus the number of users. First, there is the kind of system that can be started with a small number of users at a fairly small system cost; it can be expanded for more users, but only at a rather steep increase in system cost. The JOSS system, which is a Rand-built dedicated system tuned to a particular computer, is an example of this type. It is cheap to run in a small way but expensive to extend. The other kind of system starts out with a very large threshold, but has a smaller derivative with respect to the number of users. Multics, which tries to be all things to all men, is an example of this latter type. A small Multics is a physical impossibility at the present time. Since Multics must have a large amount of equipment and probably a minimum of a quarter million words in primary memory, there is a large cost commitment that necessitates a lot of customers to pay for it.

The major problems in the design of multi-use systems are (1) deciding what to build and (2) making sure that one's expectations do not exceed one's accomplishments. These problems involve a number of "sensitive issues." The justification for discussing these issues is found in certain unfortunate conditions currently discernible in the computer environment.

Typically, a large system is about twenty-five percent hardware and seventy-five percent software. But the minute a system is discussed, the talk is about memory - cycle times or about fancy instructions for doing floating-point or about accomplishing something so small in a Byzantine way that it has no real significance in the overall system picture. The talk does not ordinarily get around to the reasons why software is complicated. Another condition is that most software systems are usually late in delivery and below their performance estimates. There are some horrendous examples when you look at the industry at large. But the reason is not that people did not try; it is that they did not fully understand what they were getting into. Another condition is that certain traditional management techniques are ineffective. For instance, it has been demonstrated that system programming using the human wave approach, is somewhat wanting. Because computer software is getting more and more serious as computing becomes important to business -- as it accounts for the expenditure of one to ten percent of a company's resources -- there should be no slip-ups in supplying services. On the other hand, as the need for more ambitious systems increases, they come in larger units, and the mix-ups are larger when they happen.

In the design of multi-use systems, there are some signals in administrative and technical areas that are symptomatic of danger. They indicate that the planned goal of the project will not be achieved. Fifteen of these danger signals are discussed in this essay.

1. The first danger signal is when the designers of the system won't document. They don't want to be bothered trying to write out in words what they intend to do. Instead they seem to want a vague charter, and they don't want to spell out the mechanisms they intend to use. The general attitude goes something like this, "I will write when I've coded it, when I've figured out how it works." This is an old school of programming, one that is obsolete. The hardware people seem to have set the pace in showing the proper discipline. No one would dream of giving a hardware designer a wiring gun to create a computer. We would just as soon give a two-year-old girl a paint brush and tell her to paint a mural. Yet, that is how a lot of large systems have been thrown together. Basically, the first thing that comes to mind is made to be the system. In general, a person is not very capable of designing unless he can list several ways of doing something. Then, he has a choice of how to optimize the design. This whole process of design review and argumentation among the design team as to the proper choice of paths is currently very poorly done in the programming world, and this is part of the cause of a lot of difficulty. Most designs are not scrutinized very well. The reason is that often very little in the way of lucid explanation is written down in advance. To some extent if something cannot be explained, then there is a good chance it isn't understood. A mechanism or a program is not designed until an ordinary mortal can understand it -- say even the hardware engineer who is not a programmer. This is a reasonable criterion because if all the decisions are forced to hinge on the judgement of the most specialized people -- people that know the shorthand of system programming a little better --

the situation is fraught with danger. What test can be used to determine whether or not a person has really finished designing something? An obvious difficulty is trying to sift out glibness from the articulate expression of ideas. It's a real problem to figure when a fellow is basically glossing over problems. How do you smoke this out? One test might be to ask the question -- will two programmers who are given the same specification program the same thing? If they won't program the same thing or if there is room for a large amount of "artistic expression", there has been no designing; it is being left to somebody else. As an aside, this discipline has been tried in the Multics project. It has not been a total success partly because it's a new experience for programmers. There should be better results in the future, however, because the direction is right.

2. The second danger signal is when designers won't or can't implement. What is referred to here is the lofty designer who sketches out on the blackboard one day his great ideas and then turns the job over to coders to finish many months later. That is clearly an exaggeration but if the design isn't clean enough and understood enough so that it is sensible for the responsibility of implementing it to the rest with the designer right to the bitter end, trouble is going to appear. The reasoning here is that even if a designer has the best of intentions of looking for all the pitfalls in his design documents, there are unexpected little surprises or mechanisms that just don't work out very well -- things which are five times slower than he expected because he forgot about the bookkeeping that had to go on. This kind of thing happens all the time. It's not a disaster providing someone recognizes the problem early and solves it. Such

recognition and solution usually require the designer. But if the work is farmed out to somebody who didn't design it, there emerges a whole new set of communication and control problems. So while it's not impossible for the designer to let somebody else do the work, trouble is in the offing the minute that happens.

3. The next danger signal is when the design needs more than ten people. This doesn't mean that all the support people, the secretaries, the technicians, and the like must add up to no more than ten. But when the crucial kernel of the design team is more than ten people, a larger scale project is coming into being. This is the point where communication problems begin to develop. It is impossible to talk in an involved way to more than ten people a day. In addition, all the offices cannot be placed physically near each other. Specialization begins to develop so that people are no longer interchangeable. Although bright man A can take over bright man B's job in case he's sick, it will take him three months just to understand the issues that bright man B knew. Suddenly there is no longer any mobility in a technical sense. Another problem, of course, is that the minute there are more than ten people, not everybody is equally proficient, and supervision problems begin to appear. Management is always trying to do an ambitious system. And a manager is always pushing people in as deep as they can swim. Thus, a critical thing is supervising programmers. A supervisor has to watch like a hawk for a man beginning to drown. A symptom is that he doesn't produce any results for a week or a month, but is still at his desk every morning with his pencil in his hand. If he isn't found out for a month, a month is lost

because obviously the job is still to be done. Everyone is always being pushed to the limit, and programmers in a given situation can easily vary by a factor of ten in their performance.

There is another issue too if the ten-person effort is exceeded. Here is a proposed formula. In a large organization of M people, it is necessary to take the attitude that one person can closely supervise only about six more. Let m be the number of levels of management that the organization has, so that $M \sim 6^m$. Intuitively, the number of levels of management, m, times some sort of an index of system intricacy, S, should be a constant. That is $mS \cong$ constant. In other words, when there are many levels of technical management in a project, the project is going to be of the more cautious type, one which is much easier to understand. It has to have fewer problems and less intricacy, or communication and comprehension problems will develop between levels. In large systems nobody understands everything. Each person sees what is in front of his nose, but has difficulty grasping the totality of the system.

4. If a project cannot be finished or made use of in one year, there is potential trouble, because the chances of underestimation are strong. It is going to take two years, and a year of grace must be acquired in order to finish. Another aspect of a long-term project is that a personnel turnover of roughly twenty percent per year must be assumed. One man in five in a design group is going to be gone for one reason or another. It's just unrealistic to assume an invariant implementation and design group. Again, if the effort extends over a long period of time, the documentation must be in order, because of the continual transference of responsibility. Another

reason it is important to be careful with any project longer than a one year period is that the attention span of a corporation or organization is limited. If something doesn't work within a given time, they may decide that it is a bad idea. In other words, sponsors have limited patience, and everyone has a sponsor in some sense.

5. The next danger signal for a project arises when more than one big advance is attempted. One advance is about all that can be coped with at once. The Multics project has been on the ropes for a while, but is now coming off. One of the reasons is that there are three big advances in it. One of them is implementing the software using a subset of PL/I. A second advance was new hardware which wasn't seasoned yet. An a third advance was a new system which had about every innovation that could be thought of. The price of this ambition was very heavy design requirements. Of course, this danger signal must be taken with a grain of salt, because it would be a disaster if people got so timid and cautious that nothing interesting was ever tried. To some extent it is necessary to gamble some to get anywhere.

6. The next danger signal is the assumption that the system will be finished in a fixed period of time. If the system is large, the assumption is totally unrealistic. It was historically correct with smaller machines, but with bigger systems it is incorrect. Large systems are never finished; this means that there is a requirement of constant evolution as the using population is generating new demands of what it wants. Yesterday's breakthrough is tomorrow's expectation as ordinary service. This trend has been seen in computer languages.

The pressure to keep growing puts new strains on systems. New features have to be evolved. New input-output equipment appears. People want to use it, and the system has to accommodate it. What this really reflects is that attention has to be paid to modularity on functional lines. This attention extends to anticipating the maintenance needs of software. An analogy can be found in hardware where there is a tremendous difference between an engineering prototype or breadboard and a production model which is built to be maintained on a continuous basis.

7. The next warning occurs if a compiler to implement the system cannot be afforded. If the reason it can't be afforded is because the object code is going to be too mushy, too clumsy, or too inefficient, then performance estimates are already too close. That's only talking about a factor of two or so. If performance is already being counted on so closely, then some risks are already being taken because the estimates are probably not that accurate. There should be slack enough to take on the luxury of a compiler. Another sobering number that has been said before, but is worth repeating is that one hundred lines of debugged code per man-month is a very realistic estimate when a project is really looked at. We have seen this work out in two systems. At first we didn't believe it for ourselves. We believed it for government contracting and the like, but we thought we were better. But then we reviewed our CTSS experience. The overall development of CTSS lasted roughly forty-eight months; an average of ten men and forty-eight thousand words of code is about what we did do. It has been applied to Multics, and by the time we get it completed we will have averaged thirty persons over forty-eight months, and the figures are again going to be reasonable. Furthermore, these estimates seem to hold regardless of whether the line of code is written for a compiler or an assembler.

A compiler language minimizes details at the source code level by a factor of maybe five to ten. Furthermore, a compiler language contributes to the maintainability, minimizes the problem of personnel turnover, and assists documentation. However, a compiler language is not a panacea, because it introduces a scary element in systems design. If system development is traced against time, then at first very little gets done because work must be started on a compiler. Then, too, when the compiler works, it will probably take a little longer to get familiar with the compiler and get organized. Also, the object code will probably work poorly at first. But gradually mobility goes up and things can be changed faster than in assembly language. Local inefficiencies can be found, algorithms can be retuned, and whole sections can be recoded and even reprogrammed; suddenly performance goes up rapidly in a system-wide sense. Now if everyone can wait until performance is satisfactory, the project is all right. One of the problems, of course, is that people sometimes evaluate a system at an early stage and if the time scale is wrong, there is trouble.

8. Another danger signal is the omission of key hardware. The notion of what hardware is key, of course, depends on what is being attempted. For example, in a time-sharing system if there is inadequate secondary storage, there is going to be trouble. Communication channels must be in order for handling teletypes and consoles. Here some pretty ugly problems may be encountered, such as keyboards locking up because the computer hasn't got around to unlocking them yet, if the wrong line discipline is established for some of the devices. Further, there has to be some sort of minimal protection scheme to allow the supervisor to keep control over malfunctioning programs. There must be some

sort of alarm clock which normally only the supervisor program can set and which will prevent a runaway process from going on indefinitely. Finally one less obvious need is for a calendar clock to keep bookkeeping straight. One of the biggest difficulties in a multi-processsing multi-access system is the terrific amount of bookkeeping to keep track of the chronology of events and frequently to generate unique names; a good unique name generator is the time of occurrence, since the chronology is thereby preserved. Thus, a calendar clock acts like a good unique name generator provided it includes the year. This latter requirement occurs because many file-purging algorithms are based on the date that something happens, and this date may extend back over one year. Because false purges are extremely serious mishaps, it is dangerous to rely on an operator to put the date into the system every day. An even worse arrangement is to have the year assembled into the system. Too often it is the same system programmer that quit six months ago that is the only one who knows where to change the date at the end of each year.

9. Another danger signal is when a system is not a line-of-sight system. This means that all of the terminals, consoles or what-have-you are not in the same room, within shouting distance of the operator. If everybody is in the same room -- such as is the case in the PDP-10 time-sharing system -- a lot of things can be done off-line, in effect, by casually and informally talking to one another. A case in point is resource allocation; "Hey, Joe, I want to use this paper tape reader." "No, I'm already using it. You can't have it." That kind of thing can be done and the system doesn't have to be concerned. But when a system has remote users, there is a difference. If a man

wants a tape unit mounted and he's five miles away from the computer, he must have already prearranged to leave his tape in the central computer room, where an operator must somehow mount it at the right time and in the right place. This takes a lot of cumbersome administrative bookkeeping. Synchronizing the tape mounting with a phone call is one way of trying to make it work out. A more desirable solution is to try to synchronize it with a dialogue going through the computer, but then there must be communication with the operators through some output device. Also, suppose the operator signals "I can't find your tape; what do you want me to do next?" Now new decision branches are showing up. It's not too hard to think that one through. But these are just the normal cases. There are still situations like a tape snapping in the middle of reading. The operator can go over and stop the tape unit, but what has he done? Has he stopped the whole computer? If he has to clear the whole computer to reset the situation, a disaster is in the making because the other n-1 users aren't interested in the tape breaking. Recovery procedures and taking care of the unexpected are a major part of the problem. Part of the difficulty comes from the fact that the operator must intervene in a single user's process, and in a certain fixed logical sense unbind the tape from that process and reassign it to a pool of unbound tape units.

Another place where a system which is not line-of-sight requires extra complexity is in providing for an orderly way of taking the machine down. There are times when for some reason the dropping of the service is to be scheduled. How is this to be done? It is possiible to assume that everyone had his watch synchronized and was going to get off in time. A phone call to everybody could be tried, but that's awfully cumbersome. In fact, the only reasonable solution is to arrange some scheme of automatic log-out for the user so that the material he is working with at the moment is preserved. Then when service is resumed, he'll be able to restart what he was doing without loss of information. This requirement is a rather big order and whether it can or cannot be done turns out to be an acid test for most systems. To meet the requirement of automatic logout, what must be done in effect is unbind all of the specific bindings to the hardware -- the status of particular tape units that are being used, particular disc files that are open, and the like. They must be unbound because when the system is brought back up, it may be necessary to reconfigure equipment, and there may thus be different permutations of the tape units or different arrangements of the disc file. So it's an intricate and non-trivial problem.

10. Another danger signal is when the system is required to remember information. In a time-sharing system, consoles have a small band width communication channel to the computer, so that one of the following extremes must be taken. Either all the information is kept close at hand and the large computer used almost as a desk calculator, or all information is put into the computer with very little at the console. For obvious reasons, the latter is what happens. Now all the key programs are in the computer. If after the first year all the work that has been built up painfully and tediously suddenly evaporates, it will be a major jolt. As time goes on the situation will only get

worse. What it means is that the management of the computation center is forced to supply some insurance against catastrophe. In fact, the more reliable the equipment gets, the more this insurance is necessary. This is analogous to a fire in a city. No ordinary citizen can maintain his own private fire department. He has to rely on the city when a fire occurs. In the case of information storage, protection against outright disaster is relatively easy. The difficult problem arises in the case of the semi-disaster, where the information of only a few people gets lost and the aim is to restore their information, not that of everyone. Since it is not desired to reset all information files to an earlier status just because one bit was dropped out of somebody's program, the resetting must be done selectively. This adds a whole new set of problems that are non-trivial.

11. Another danger signal is when the system must grow without bounds. One of the first problems encountered, is that the memory capacity has to grow without bound. But at this time, there is conflict with the point just discussed: namely, the system must be backed up. The straightforward way to organize a backup is just to dump the contents periodically. But the larger the amount of information involved, the more impossible it is to dump the contents without major service disruptions or mishap. Solving this problem requires fairly sophisticated techniques of incrementally dumping material as generated. It's frankly, an order of magnitude more complicated and the particular solution will vary with circumstances.

12. Another danger signal is where the system is open ended. That is, programms that introduce new properties into the system can be written

on it. Such a system is in contrast to one that is closed -- e.g., an airline reservation terminal -- where an ordinary clerk cannot extend the services at his fingertips and what he can do is very seriously constrained. Another example of a closed system is the JOSS System, which works through an interpretive system. A user can write any legal JOSS program, but if he attempts anything else, he gets his hands slapped. The issue here is that as soon as a system is open ended, it gives rise to a whole new host of protection problems involving both accidents and maliciousness.

13. A somewhat related danger signal is when there are over ten system maintainers. Here, I'm talking about an on-line system that is actually being maintained on-line. In such a situation, there may be a supervisor program which every system programmer is able to work on. If every system programmer has the same rights and privileges, ten is about the most that can be tolerated in the event that some mishap occurs. If more people are in the same group, there is no way to account for who made a change. A smaller group of people is also a way of trying to avoid administrative bottlenecks. Normally, the only way of coping with a situation where a lot of people have access to one master file is to get very heavy handed and be very careful. But it will be realized that things cannot be done in a hurry. For example, the password of a user is a very sensitive piece of information to let the system identify who it is that is dialed in or is working with the system. The issuing of passwords can be trusted to only one man if responsibility is to be isolated. But if he's out of town or sick, there is a problem. In other words, an administrative

bottleneck has started. The same problem comes up when computer time runs out at midnight on a Friday night, and the desire is to work over the weekend. The problem is how to acquire authorization quickly to get more computer time in the account so that work can be continued. There is no desire to call up the director of the project at his home at midnight on a Friday night. There might be some willingness to call up the supervisor and beg for a little lenience. Clearly large groups of system maintainers introduce problems.

14. The next danger signal is when continuous service is wanted. Of course, the decision must be made as to what is meant by _continuous_. If _continuous_ means "never stop," not even a microsecond, it is doubted that anyone knows how to do that today. But if the system can be down for a minute or two, then there are solutions known. However, the real problem of continuous service comes up in the normal system housekeeping. A normal desire is to be able to use the machine offline in some sense, doing system updating, changing the supervisor, improving the library programs, etc. When are these things to be done if the system is running all the time? Another example is in the running of an airline reservation system; if it is really being run continuously, then when is the disk file or the drum to be backed up? When can the information be gotten out of the system?

15. The last danger signal is when the system requires the ability to permit combinations of sharing, privacy, and control. That systems which have these requirements are needed is clear. The accounting files in a department store are an example. Certainly, arrangements should be made so that every clerk can enter transactions and that an accountant can correct mistakes, but not the clerk. Thus, a whole question

of responsibility and accountability is merged with the mechanics of the system. A further concern is about auditors being able to check to see that everybody in the chain of responsibility is doing as expected. Hence, it gets to be a very subtle problem of access control. In addition, there is a problem of deciding who people are. Just because a man says he is John Doe at the end of the terminal isn't proof at all that he is. It's non-trivial, too, in the sense that just putting passwords and locks in wildly doesn't solve the problem. First of all there's the obvious problem of having padlocked the door which is embedded in cardboard. But there are other problems. Suppose a lock is put on a door that many people use. One man loses his key. The key is compromised. And now the obvious decision is to change the lock quickly and, of course, this means that new keys must be passed out to the n people who already have the keys. Clearly, this is clumsy if n is large. In fact, there is tremendous awkwardness if the access control problem is not handled correctly. There are solutions, but it is not a closed subject. This is partly becasue it is not just a question of authorizing individuals. Often the desire is to discuss a group of people as a class because when a group of people is specified, there is no need to know every member of the group. Thus access control represents a potentially intricate area in a system.

That's a very long list of danger signals. Perhaps some people may conclude that it isn't wise to want to work on a new system. Unfortunately, this is the wrong conclusion to draw. I haven't spent time encouraging people to try new ideas because I assume there are many who don't need it. In fact, the only way to make progress in developing systems is to try. There is no magic set of answers to all of the problems I have raised. Today the people who are most knowledgeable in this area of multi-use systems are those who have tried to do something themselves. Thus, the only real issue in choosing a project is to bite off something that can be chewed, and it is to this point that the list of danger signals has been addressed.