X

April 10, 1969

THOUGHTS ABOUT RESEARCH TOWARDS MULTICS-2

Michael J. Spier and Elliot I. Organick

One of the attractive potentials unique to Multics is the opportunity to design subsystems consisting of cooperating processes, some of which are non-sequential in nature. A Multics process that would live in its Wait Coordinator and respond to series of external events by performing computations peculiar to each event, would be an example of a non-sequential process. Processes that perform answering service, agent or broker functions, are examples of non-sequential processes. It is believed that the design of subsystems with large numbers of intercommunicating processes would be greatly simplified if a programmer were provided a chance to create and fully utilize non-sequential processes of general (unrestricted) capability. Unfortunately, with the present system design such non-sequential processes that can be created must be artificially restricted to insure satisfactory response characteristics. Such restrictions severly weaken the potential value of these processes.

Consider a non-sequential Multics process that is designed to service event calls over a set of distinct and functionally independent event call channels. Unless special precautions are taken, service to one event call adversely affects service for another. This condition arises when service for two different event calls is initiated but must in each case be temporarily suspended awaiting receipt of a needed message before being allowed to complete the intended task. When the second service is delayed it is

impossible to resume work on the first even though said message has (by now) arrived. Effectively, the decision to wait for receipt of one message (the second event wait) masks the arrival of messages previously waited for. Since the "blocks" are stacked, wakeups can be achieved only in LIFO order. The result is that event call tasks, once begun, cannot enjoy the luxury of calling ipc$block without risking (indefinite) delays, i.e., delays that may extend well beyond the time at which the waited-for message has been received.

The undesirable characteristics we have just described are intrinsic to Multics processes as we now know them because there is essentially only one ~~stack~~ process per ~~process.~~ address space Each successive invokation (activation) of the Wait Coordinator is associated with a new stack frame. It is impossible to recognize arrival of events on the wait-lists of previous activations of the Wait Coordinator as long as the stack frame for that activation is buried below the top of the stack. Abnormal returns to recognize the desired events are out of the question because such a practice would destroy the history of service that has begun for subsequent (and functionally independent) event calls. There appear to be but three ways out of this bind:

(a) Learn to be very clever and program event call driven computations without incurring (explicitely nor implicitely) any calls to ipc$block. This approach is impractical in the general case (it requires the

reprogramming of any system-facility which may call ipc$block, e.g., the I/O system) and certainly requires first-rate programming skill even in benign cases. This approach was used in programming the present answering service.

(b) Let service to each of the "conflicting" event calls be disjointed by creating separate processes --one to serve each individual event call channel (in other words, replace a single multiple-task non-sequential process by as many dedicated sequential processes as there are tasks.) This approach is feasible, but suggests the proliferation of processes, each with a separate but essentially identical address space, each employing a separate and sizeable secondary-storage as well as wired-down primary-storage commitment. In effect, their address-space would essentially differ only in the stack. Nearly all other aspects of those dedicated sequential processes might be identical.

(c) Alter the Multics design of a process in some fundamental way so as to achieve a cake-and-eat-it-too effect. But, any fundamental design change --such as might be motivated here-- should be viewed as a research effort which, if successful, would contribute to constructive approaches for new long-range Multics developement.

A chief purpose of this memorandum is to sketch one such research approach. However, before developing this sketch, we shall digress to examine some implications concerning Multics research in general. It can be argued that research is always timely. In particular, with (what we shall call) Multics-1 nearing completion, it appears especially timely to consider types of research that might guide eventual efforts towards significant departures, i.e., Multics-2. ← The name implies revolution, not evolution.

Ways to Think About Multics-2

A first question to ask is: "Must the Multics 'series' be upward compatible?" Our thoughts are these:

Multics-1, being a research effort, need not be completely upward compatible with all future Multics', but should be useful as,

(a) a concept and experience resource, and as

(b) an instrument (tool) on which to build or test not only minor revisions, but significant new departures.

Moreover, many of its major modules, e.g., Basic File System, Linker, I/O Control etc., are likely to be incorporated, i.e., reusable in any revised context. For example, the Hardcore-ring of Multics-1 effectively creates a virtual machine on which one could, in principle, build new types of subsystem capabilities.

A second question to ask is what if any obvious methodology suggests itself for the pursuit of a Multics-2.

We propose here one such approach. It consists of three steps:

1. Define a set of added capabilities that are deemed desirable and which are compatible with current system objectives. (An example would be coexisting processes that somehow "live" in the same "universal" address space, i.e., share an entire descriptor segment rather than individual segments.)
2. Identify those original design principles (circa 1965, 1966) that have been de-facto altered during the implementation period (1967-1969). Infer a modified set of design principles as appropriate (e.g., modified concepts re: distributed supervisor, ring-protection mechanism.)
3. Attempt to draft a Multics-2 that achieves the capabilities defined in (1) and that is consistent with the modified set of design principles derived in (2).

AN EXAMPLE APPLICATION OF THE METHODOLOGY

We now propose to illustrate how this methodology might be followed to produce a new "process" concept that may offer the cake-and-eat-it-too advantages we spoke of in the introductory part of this memo.

1. Proposed Objective (added capability)

Any Multics-2 process is to be capable of becoming the root of a tree of processes. Each subsidiary process in the tree is created and eventually destroyed by its parent. A process' priority in the race for a processor is determined by its relative location within the tree structure, with priority inversely proportional to the location's depth. A higher priority process' need for a processor is always satisfied at the expense (pre-emption) of a lower-priority running process.

Each process in the tree is defined as being a potential execution point (pseudo-processor) associated with a set of process state characteristics (pseudo-SCU data) that can be "restored" to get the process to run once more. The state of a process is defined from an (expanded) APT-like table entry. These entries are tree-linked to mirror the hierarchical relation among co-existing processes. Execution priority is determined by the tree level of the process.

The APT entry includes:

(a) a list of received event messages (currently known as the ITT queue.)

(b) process state variables (analogous to processor machine conditions) including certain items now kept in the process' data blocks, all the items that are currently kept in the APT entry, a pointer to the process' "local" ~~memory~~ address space (stack) and a pointer to the process' "universal" ~~memory~~ address space (descriptor segment.)

To clarify the above, let us remark that we consider the stack to be part of the process' state. The stack should be regarded as being a collection of software-implemented registers. It has been observed that one never refers symbolically to one's stack in terms of <stack>|[offset], in fact, when programming a higher-level language one is completely unaware of the existence of a segment named "stack", and even had one wanted to refer to it symbolically it would have been impossible because the stack has no linkage section and no associated linkage-pointer and consequently the expression <stack>|[offset] cannot be evaluated. Just as it is impossible to implement a paging mechanism in a paged environment, so is it impossible to refer symbolically to one's stack because the very same stack is necessary for the implementation of the symbolically-addressable ~~memory~~ address space. Evidently, the stack must be referenced indirectly through a dedicated hardware register. In the current Multics, due to the fact that

GE-645 registers talk in terms of segment numbers, it was necessary to incorporate the stack segment into the general file system framework. One may, however, conceive of a system in which the stack pointer is an absolute-address register (similiar to the DBR). In such a system, the stack would be outside of the scope of the descriptor segment, loosing none of its paging capabilities yet allowing more than one processor to execute in parallel on separate ("local address space") stacks using a shared ("universal address space") descriptor segment.

If one wants to program a non-sequential, multi-purpose process in this environment, one pictures the process, say A, spawning new daughter processes, one per purpose, each sharing the same address space with A. Each daughter is responsive to a different class (type) of events. The parent spawns a daughter process by executing a primitive, which somehow has the effect of establishing the desired new APT entry in association with the parent's current DBR value, i.e., effectively defining a new execution point within the same universal address space.

Terminology is all screwed up

2. Modified Design Principles in Multics-1

Actual implementation has resulted in a series of de-facto compromises of the distributed supervisor concept. These compromises have arisen from a need to avoid interference. Major parts of each of the following modules now operate locked to one process(or) at a time, and for this reason it may be that they can be regarded as if they

Why locked → process ?

were separate, dedicated processes (though not "Multics processes"):

(a) page control

(b) segment control

(c) directory control

(d) traffic control

Conjecture: Ideally, we would like to see those functions wholly or partially performed by the hardware (in fact, we may currently have enough understanding of Paging to try ~~and to~~ envision a hardware-implemented paging mechanism.) Therefore, it may seem appropriate to regard the present implementation of those functions as software-simulated "hardware", and to talk about the Basic File System and about the Traffic Controller as if they were a pair of "black boxes" providing us with a virtual 2-dimensional address-space and virtual processors, respectively.

3. A Proposed Multics-II

Consistent with the above conjecture, suppose we picture that File System, Linker and Traffic Controller/IPC can be regarded as a collection of independent, dedicated and privileged modules which can be executed as independent Multics-1 processes (in other words, let us regard Multics-1 as being our basic machine, and attempt to start our research effort from there.) We picture that these dedicated processes would be invoked by faults or other breaks in the execution of a Multics-2 process. A Multic~~s-II~~s-2v (henceforth called user-) process may create a daughter process by first

requesting the establishment of an address space (DBR) by the File System "black box", then requesting from the Traffic Controller "black box" the creation of a new execution-point associated with that address space. As process creation happens in two stages, it is possible for the parent process to specify to the Traffic Controller, as parameter, its own DBR-value, in which case the new daughter process becomes in fact an additional execution point in the parent's universal memory space. A daughter may thus be created in the "image of its parent" (i.e., made to share the same universal address space with its parent) or be given a new address space (that is subsidiary in some delegated sense to that of its parent.) In any case, address spaces for user-processes no longer include (the equivalent of current) hardcore procedures. Furthermore, as we are no longer aware of the actual hardware, our processes are no longer directly associated with the notion of wired-down storage; also, they no longer call upon any supervisory procedures, as it becomes evident that all control functions (which are currently handled by the supervisor's non-hardcore rings) will be managed by the parent process, and therefore be completely out of the user-process' scope.

In fact, in view of the above, we may now re-define the meaning of the ring protection mechanism and say that a ring corresponds to a level in our APT tree structure. This allows us to better understand the parent/daughter

relationship as we may now clearly see that a daughter's address space is indeed a subset of the parent's address space, and that the parent has unrestricted access to the daughter's address space whereas the daughter may only have as much access in the parent's address space as the parent may deem necessary.

All event signalling/reception is handled by the Traffic Controller "black box" which acts as a broker. A process is said to be in the "ready" state if the event queue associated with its APT entry contains at least one event message (upon waking up, however, processes manage their events in the normal (current) manner.) Processes are chosen for running according to their tree-level priority. Within any tree-level, priority is awarded according to the classical (multi-level) scheduling ~~algorithm~~ discipline.

We can see now that whereas in Multics-1 a non-sequential process can at best employ a single processor at a time, a Multics-2 process permits in addition to non-sequentialism real multi-processor parallelism.