June 1973 ON PL/I TASKING IN MULTICS MSB XX
by A. Bensoussan

Introduction.

The purpose of this document is to investigate how Multics could provide processes that exhibit the addressing capabilities required by PL/I tasking. First, those requirements of the PL/I language that cannot be met by the current capabilities of the Multics system are identified. Then, the capabilities that should be added to the Multics system in order to meet these requirements are discussed. Basically, they amount to being able to run several processes in the same address space, a possibility that has always been excluded since the early design phase of Multics. Finally, the major modifications that have to be made to the current supervisor in order to implement these additional capabilities are described.

Addressing capabilities required by PL/1 tasking.

One requirement of PL/I tasking is that any information available to the main program be also available to the created task. For the moment, let us designate by process group the whole family of processes generated by the execution of a PL/I program that uses the tasking facility. It is clear that, if each process of the group was a standard Multics process, it would not be easy for one process to use a pointer manufactured by another, since the value of a pointer is process dependent. In order to make the value of a pointer invariant within the process group, it is necessary that the mapping between segment numbers and segments be identical in all processes of the group.

Another requirement of PL/I tasking is that an entry constant must have the same value in the main program and in the task. Which means, in Multics terms, that the search rules must be identical in all processes of the process group. Since the segment number assignment and the search rules interpretation are the two and only two functions provided by the KST, it follows that the two requirements mentionned so far can be satisfied simply by sharing the KST between all processes of the group.

The next PL/I tasking requirement relevant to our study is that internal and external static variables be common to

the main program and all the descendant tasks. Internal static variables, in Multics, are allocated in the linkage section of the procedure in which they are declared. External static variables are allocated in the segment "stat_" created in the process directory, and they are accessed through ITS pointers located in the linkage section of the procedure in which they are referenced. One can make static variables common to all processes of the group simply by sharing the linkage sections and sharing the segment "stat_" among them. Note that by sharing the linkage sections one also shares the links to all external references of the type "a$b". This is still consistent with what we have said up to now; in fact, an external reference of this type is either an entry constant or an external static variable, and, in both cases, the PL/I language requires that they be common to all processes of the group.

Although all matters of access control are irrelevant to the PL/I language, it is reasonable to think that, because all processes of the group cooperate in solving the same problem, they could have common access rights to all segments of their address space. It follows that all processes of the group could also share the descriptor segment.

User view of the process group concept.

Trying to satisfy the addressing requirements for the tasking in PL/I, we have arrived at a special class of processes, yet unexploited by the Multics system, that we refered to in this document as a process group. A process group is basically a collection of processes having the same address space. We think that this concept is worth being implemented as a Multics facility, available not only to support PL/I tasking but available also outside the PL/I context, for any application that would need a collection of inexpensive and not too sophisticated processes. From the user point of view, a process group is a set of processes with the following properties:

- The mapping between segment numbers and segments is identical in all processes.
- The search rules are identical in all processes.
- The evaluation of an external reference is identical in all processes.
- PL/I internal and external static variables are common to all processes.

- Access rights are identical for all processes.
- The process directory is common to all processes.
- Each process has, of course, its own stack, and more precisely its own set of stacks, one for each ring, and its own process identifier.
- The overhead associated with creating and supporting a given number of processes is much smaller if they are all part of the same process group than if they were independent. The reasons being that processes of the same group share the descriptor segment, the KST, the linkage sections, the process directory, the "stat_" segment; they generally experience less linkage faults and less segment faults; also they can communicate at very little cost due to the fact that segment numbers mean the same thing to all processes.

Implementation of the process group concept.

A significant number of modifications to the Multics supervisor will be needed to implement the concept of process group as defined above. The most important changes that can be identified at this point are in the following areas:

1. Process creation and termination.

A process should be able to ask the supervisor to create a new process in its address space. The new process would then be created using the same process directory, KST, descriptor segment, linkage segment. The name of the new process would be added to the process directory, which becomes the process group directory. Any segment created in the process directory and meant to be per process would have a name suffixed by the process identifier. For example, the ring 4 stack for the process whose process identifier is xyz would be "stack_4.xyz". On the other hand segments that are meant to be common to the group would be named without such a suffix.

Eight consecutive segment numbers, starting at the first available number in the KST, must be reserved for the stack segments for the new process. The first of these numbers is the segment number of the ring zero stack for this process and is displayed in the DBR. As we will see later, the way segment numbers are assigned to the PDS and PRDS has to be changed.

When a process of the group terminates, its name is removed from the list of names given to the process directory, and only segments whose names are suffixed by its process identifier are deleted. Segments common to the group are deleted only when the last process of the group is terminated.

2. KST management.

The KST can now be accessed by several processes and therefore the actions of these processes must be synchronized. A single lock for the entire KST seems to be a simple and adequate strategy.

3. Linkage segment management.

Linkage segments, like the KST, can now be accessed by several processes whose actions must also be coordinated. Here again, a single lock per linkage segment seems to be adequate.

4. References to the PDS and PRDS in ring zero.

The current Multics implementation takes advantage of the fact that a given descriptor segment is associated with only one process, and uses the following conventions: All processes must use the same segment number to reference their own PDS; also, they must use the same segment number to reference the PRDS associated with the processor they are executing on. Note that, before giving the processor to a process, the traffic controller adjusts the segment descriptor word reserved for the PRDS to make it point to the PRDS associated with the current processor.

Because the PDS and the PRDS have invariant segment numbers, one can reference them through ITS pairs, shared by all processes. These ITS pairs are found in the shared linkage section of the referencing procedure and are produced during system initialization by the pre-linker resolving external references of the type:

pds$x	or	prds$x	in PL/I
<pds>|[x]	or	<prds>|[x]	in ALM.

In any event, it is possible to live with these conventions so long as only one PDS and one PRDS need be described in a given descriptor segment. If several processes are to use the same descriptor segment, one must be able to describe simultaneously several PDS's, one for each process, and several PRDS's, one for each processor; and therefore one cannot keep these

conventions any longer.

It follows that the method by which ring zero procedures reference the PDS and the PRDS must be changed, at the source level. Any procedure that needs to reference the PDS or PRDS would have, first, to get a pointer to the base of this segment, a pointer that it would keep in automatic storage and use to access the given segment, instead of referencing it by its symbolic name.

5. Fault and interrupt handling.

The fault and interrupt mechanism, also, makes extensive use of the fact that the PDS and the PRDS have invariant segment numbers. The store control unit instruction stores the control unit in the appropriate PDS or PRDS using judicious indirect addressing based on the invariance of their segment numbers. All store control unit instructions are located in the fault-interrupt vector and have one of the following types :

(1) SCU A,* ===> A : ITS (pds_segno , offset, *)
(2) SCU B,* ===> B : ITS (prds_segno, offset, *)

where A and B are absolute addresses. Type (1) is used for storing the control unit in the PDS; type (2) is used for storing the control unit in the PRDS. The fault-interrupt vector, as well as the ITS pairs located at absolute addresses A and B are shared by all processes and are assigned values at system initialization.

It is clear that, if the segment number of the PDS is process dependent, one cannot expect, at least with the present hardware, the SCU instruction to select the appropriate PDS. The first approach that one can think of is to store the control unit always in the PRDS and let the software move it to the PDS, once it has determined its segment number. Unfortunately the segment number of the PRDS is not invariant either, and cannot appear in the interrupt vector shared by all processors.

The problem would be simple if each processor could have its own interrupt vector; although this was allowed in the 645, it is not possible in the 6180. It would be equally simple if the processor number could be used as an index register; but it is not the case in the present hardware. One could also think of using the approach taken by GECOS, which is to direct all interrupts to the same processor; but, although Multics directs all I/O device interrupts to the same processor, this approach is

not feasible either because the traffic controller generates interrupts that may be directed to any processor in the system. A solution has been developed, however, which does not require any hardware modification nor traffic controller restriction, and is presented below.

In order to cause different processors to store their control units in different areas, all SCU instructions of the fault-interrupt vector use indirect addressing with tally modification, as follows :

SCU T,ad =====> T : TALLY (A,N,L)

where T and A are absolute addresses, N the number of processors in the system and L the number of words occupied by the control unit information. The hardware behaviour, under the effect of the "ad" modifier, can be described as follows :

- Lock the memory word T.
- Save the address A in an internal register R.
- Add L to the value of A in word T.
- Subtract 1 from the value of N in word T.
- Unlock the memory word T.
- Store the control unit at the address saved in R.

That is, if all processors in the system were to receive an interrupt signal at the same instant, one processor would store its control unit at locotion A, another processor at location A+L, and so on, and the last processor would store its control unit at location A+(N-1)L. The order in which processors are selected is immaterial; what is relevant is the fact that, starting at location A, there is an array of N elements, each of which may hold one and only one control unit.

All SCU instructions are followed by a TRA instuction, that transfers control always to the same procedure. This procedure can be viewed as a front end to the current fault-interrupt interceptor, and its purpose is, in essence, to accomplish what is currently accomplished by the SCU instruction plus the conventions about the PDS and PRDS segment numbers. This front end fault-interrupt interceptor starts by incrementing by 1 (in one cycle) a counter that counts the number of control units effectively present in the array. Then it proceeds with a locking sequence, using the STAC instuction and carefully handling the case of a non zero A-register. Then it identifies, in the array of control units, which entry belongs to the executing processor; it does so by comparing the processor number stored in each control

unit with the executing processor number. Then, from the fault or interrupt number, found in the control unit, it determines which of the PDS or the PRDS is to be used in this particular case. It now copies the control unit information into the selected segment, in which it also stores the bases and the registers as they were at the time the fault or interrupt occured. Then it compacts the useful portion of the array of control units by carefully copying the last stored entry into the current one and carefully updating the tally word, using the virtues of the STACQ instruction. Then it decrements by 1 (again in one cycle) the count of stored control units. Finally, it clears the lock it had set at the beginning and transfers to the appropriate handler for this specific fault or interrupt, after having set the stack and the linkage pointers to the appropriate values.

6. Existing procedures using static variables.

Because PL/I tasking has not been available in the Multics system, Multics users have always considered static variables to be private to a process, and never subject to sharing. As a result, a procedure that performs correctly today may not perform correctly any longer when executed as part of a multi-task PL/I program. Not only must one learn how to write procedures that behave equally well with or without tasking, but also one must audit all existing PL/I procedures supported by the system and correct them whenever a static variable has improperly been used. Hopefully, ring zero procedures can be exempted from this auditing; Multics system programmers did know that all ring zero procedures were supposed to be pre-linked at system initialization, causing all linkage sections to be shared by all processes, and therefore they could not possibly assume that ring zero static variables would be private to each process.

It is important to realize that, even though ring zero procedures are exempted, this auditing is a long project, and that it should not be underestimated. It may very well be done incrementally in several successive phases, but it definitely has to be done. One cannot guarantee that programs such as, for example, the basic command loop, the various editors, the debugger, the quit handler, the interprocess communication facility, the I/O switch facility, the PL/I I/O run-time routines, or the file manager will work properly in a tasking environment without a careful analysis of what they are supposed to accomplish when they use static variables.

Other implementation alternatives.

This paragraph describes other alternatives that might be available for solving the PDS and PRDS problems.

First, if a variation of the ITS modifier were added to the hardware, no change would be required to the way ring zero procedures reference the PDS and the PRDS, and no change would be needed to the fault-interrupt handling. This special feature, that one may refer to as an ITSR modifier (R for Relative), would be recognized by the processor, and interpreted like an ITS in which the segment number would have been incremented by the segment number of the ring zero stack, displayed in the DBR. One could, then, make the following conventions: in any process, the segment number, relative to the ring zero stack segment number, of the PDS is invariant; also the segment number, relative to the ring zero stack segment number, of the PRDS is invariant. Because they are invariant, the relative segment numbers of the PDS and PRDS may appear in shared ITSR pairs. Before giving the processor to a process, the traffic controller, aware of this convention, would accordingly adjust the segment descriptor word describing the PRDS for the current processor in this process. The only significant software modification would consist of changing, during system initialization, ITS pairs into ITSR pairs whenever they point to a PDS or PRDS.

Next, the hardware should definitely provide a more decent mechanism to separate control units stored by different processors when they are interrupted. The proposed change is to add, in the hardware, the possibility of indexing by the processor number. The availability of this feature would eliminate the complexity of the front end fault-interrupt interceptor, introduced by using the tally modification capability to safely store the control unit.

Finally, another possibility of avoiding changes related to the PDS and PRDS references, as well as changes in the fault-interrupt handling, would be to give up the idea of sharing the descriptor segment. One could then continue to honor the current conventions about the invariance of the PDS and PRDS segment numbers. Of course, the KST would still be shared and all processes of the group would have a different set of stacks, with a different set of segment numbers; since the ring zero stack and the PDS are the same segment, this segment would be described by two different segment numbers in each process, which is perfectly acceptable. The major objection to this solution is that it would increase the overhead associated with a process group because of the multiplicity of descriptor

segments which, in turn, implies more overhead in segment fault handling. However, this solution could be retained for an initial implementation of the process group concept.

Conclusion.

The technique by which Multics can satisfy the addressing capabilities required by PL/I tasking seems to be well understood at this point. The concept of process group, the properties of which derive primarily from the PL/I requirements, has been introduced. In Multics terms, a process group is a set of processes using the same descriptor segment, the same KST, the same linkage segments, the same process directory and the same PL/I static variables. Process groups will be the basis for implementing PL/I tasking in Multics. In addition, process groups will be available outside the PL/I context, for applications that would need a large number of inexpensive cooperating processes.

From the number of lines devoted in this document to each area that needs modifications, the reader may be left with the impression that the most critical area is the interrupt handling. It is true that additional complexity has been introduced in this area, but this complexity would automatically disappear should the hardware provide adequate support for handling interrupts in a multi-processor configuration. On the other hand, the auditing of static variables used in PL/I procedures could appear to be of secondary importance; but one should remember that it will be the most sensitive, the longest, and the most expensive part of the process group implementation.

In addition to the addressing capabilities provided by a process group, PL/I tasking requires that processes be related by hierarchical relationships. The present Multics system does not make any provision for establishing a hierarchy between processes and a separate study will have to be made on this subject if PL/I tasking is to be implemented.