NCR ADVA RY

NCR\CDC PRIVATE

MEMO

DATE: November 20, 1974

TO: R. E. Wagner LOCATION: PGAASL

FROM: E. C. Crary ecc LOCATION: ESCASL EXT: 57

SUBJECT: OS STRUCTURE

After our discussions during my last trip, I came away with the impression that some confusion exists concerning Address Spaces and their relationship to Processor States, Asynchronous Processing and Serialization of Access. Perhaps the confusion is only mine, but I would like to express my understanding of these concepts if for no other reason than to demonstrate its inadequacy and thus the need for further documentation.

Processor State is used to protect the OS environment from corruption by errant code of a less trusted nature; in particular, to protect the OS from its user. In this context, the OS environment is a collection of services upon which the user code may make requests. The user requests are normally of an explicit nature, but may be implicit as a result of some signal. A page fault is the most obvious case of an implicit service request.

With the advent of mass storage and larger memories, the OS grew to include functions that are not services to be performed for the user code, but are jobs in their own right. These System Jobs include spooling and job scheduling. A distinction must be made between system jobs and services as they are structured in a different manner within the framework of the OS.

Address Spaces are used to define jobs, or in some systems, job-steps. The address space is the basic unit of accountability and the unit to which resources are assigned. With the inception of multi-programming, the address space also became a mechanism for relocation.

The confusion on the user of address space arises because it does not always have a complete definition or cause total relocation. We are apt to lose sight of the relationship of the OS services to the user code. But if address space is considered from the services point of view instead of the users, then we see that the user resides in the same address space.

The point I wish to make is that the address space is not a mechanism used to isolate the OS services environment from the user code. In terms of isolation, it is only used to protect one job from another. In particular, the OS services of paging (I/O by implicit request) and explicit I/O are not jobs, but belong in the same address space as their requestor.

This whole discussion comes around, of course, because I am told that paging and device drivers will not be in the same address space as their requestors. And that structure, I do not understand. I am given as many reasons for this as people whom I asked. One reason was for protection, another reason was because of the privileged nature of the code--again, a protection problem. I also heard that it gave modularity, it provided for asynchronous processing, it provided for serialization of access to the IORP queues, and even that it had something to do with the amount of "nailed down" memory. The rest of this note will try to explore these various ideas.

Protection is becoming a sacred cow, and I would like to see some confidence in all the mechanisms we have specified in the virtual memory apparatus. First, we have rings--these, in connection with Read/Write/Execute permit bits allows any portion of the OS services to be made totally inaccessible from the user. To protect the tables of one portion of the system from access by some arbitrary other system routine, we have a local lock/key mechanism. This assures us that code in one, and only one segment can modify a data structure in another segment; so the OS environment is protected from errant code in itself. Symbolic, as opposed to linear segmentation, is in itself a protection mechanism; a bad index cannot arbitrarily point to a different segment. And finally, privileged instructions are not enabled by the state of the processor, but by special enable bits in the segment descriptor. I am not convinced that separating the services from their requestor by using a different address space could isolate them from errant code any further. A single bit failure in the Segment Table Address register is potentially more dangerous than the failure of any bit in these other mechanisms.

Modularity is like structured programming; it is a good thing but we all have our own explanation of what it is and how to achieve it. I submit that while there are mechanical methods for achieving modularity, these do not guarantee the survival of clean interfaces. The development of special cases, of short cuts and of paths for optimizing performance soon destroy the clean interface and the modularity it provides. The externalizing of the implementation of a function soon proliferates the use of a particular implementation of the function throughout the system and creates a monolith rather than modularity. Separation of OS services into different address spaces will not cause modularity if performance suffers or the interfaces are not properly defined in the first place.

For the moment, I skip to the issue of "nailed memory". I don't see any reason less memory is nailed if the service is in a separate address space rather than the user's address space and shared. There still must be a copy of the code available when it is to be executed, and it can be paged out when not needed. The logic that says less memory is nailed appears to be a snow job.

Asynchronous--or parallel--processing may be either at the job level or at the service level; it does not imply automatically a separate address space. At the job level, or with a separate address space, we have known the function as multiprogramming. At the service level, I believe the literature uses the term multi-tasking to describe parallel processing. We in Control Data are more familiar with this latter as RA+1 calls for services in CYBER. In any case, the asynchronous processing of service requests takes place in the same address space as the requesting code.

Serialization of Access can be created in several manners. The CYBER uses a dedicated PPU in a master/slave relationship to the rest of the processors to allocate access to shared resources, while IPL plans to use special memory port functions to tie-break random requests by multi-processors. Likewise, if a processor is "locked out" so it cannot safely have access, there are two policies that can be taken. The first of these is the "wait" policy where the CPU continues to test the interlock structure until it is available. This is an uninteresting policy since it appears as simply a part of the task.

The second policy is the "give up and reschedule" policy. In this case, the CPU as a resource is given up by the executing task and another task is dispatched. The task that gives up in this case is scheduled to be re-dispatched whenever the resource for which it was vying becomes available.

The point is that dispatching, and the enqueuing of tasks on the dispatch lists, create serialization of access; and it is not created by mere separation of address space. It is not the conversion of a system service to a system job in a separate address space that provides serialization of access, asynchronous processing, modularity, protection or memory compaction.

Again, I must ask: Why should device drivers not be in the address space of the requesting user? Do I misunderstand some issue?

ECC:bl

cc: M. Carter
F. Clapp
L. Monheit
G. Nelson
V. Raman