Inter-Office Memorandum

To	Pilot, Mesa, D0 architecture	Date	14 July 1977
From	Paul McJones	Location	Palo Alto
Subject	Page faults and multiple processes per MDS	Organization	SDD/SD

XEROX

Archive: Pilot

Filed on: <McJones>FaultEvents.memo

With the current processor architecture, running multiple processes capable of taking page faults within a single MDS leads to problems. If significant resources (i.e. one or more resident frames) are not to be dedicated on a per-process basis, changes to the processor are indicated. Two different proposed changes are sketched here. Additionally, an analogy is drawn between handling page faults, which requires dedicated, resident frames, and handling frame heap allocation traps, which requires dedicated (possibly nonresident) frames. Each of the proposed hardware changes can be extended to handle frame heap allocation traps.

Handling a page fault requires a definite amount of resident memory for local frames. In a system which chooses not to dedicate this memory on a per-process basis, there must be a way to queue a fault until sufficient such memory is available to handle it. The current processor architecture, in which page fault handler frame space is associated with the MDS, does not provide an efficient way to dedicate this frame space specifically to page fault handling, or to regulate accesses to it by multiple processes within the MDS. One remedy for this situation would maintain the association of frame space for handling page faults with an MDS, but would provide a way to suspend a faulting process until enough of the space is available. Another remedy would decouple the handling of page faults from any particular MDS by converting page faults into messages, thereby achieving the necessary serialization of access to dedicated frame space using the "dual" approach to the first proposal.

The situation now

Traps

Currently every processor-detected trap forces the running process to execute a KernelFunctionCall (KFC) instruction with an offset in the SD (system dispatch table) determined by the particular trap. The SD entry must contain either a procedure descriptor or a pointer to a fixed frame. In the former case, a call is done, which implies allocating a new local frame from the frame heap. In the latter case, execution continues in the fixed frame.

Handling page faults

The page fault handler must be very careful about generating page faults. In particular, the original KFC which invokes it must not generate a page fault, or an infinite loop will result. Several ways to prevent this suggest themselves:

1. Pin the whole frame heap in resident pages. While it may be that the frame heap will be "hot" enough that it will all stay resident anyway, it would be nice to avoid the logical necessity to pin it; the rest of this memo is concerned with how to do that.

2. Reserve an entry of the AV (allocation vector) for the page fault handler and stock it with resident frames. There seem to be several ways to run multiple processes per MDS in this scheme. The second and third schemes below are not intended as solutions in themselves, but as motivation for the hardware-change proposals to follow.

   a. Reserve enough resident frames to handle a page fault by each process.

   b. Reserve resident frames for fewer instances of the page fault handler than there are processes. This requires additional reserved resident per-process frames (2?) for a "first level page fault handler" procedure which invokes a monitor to wait until enough frames are available to handle the fault. Later we will look at how the "first level handler" could be pushed into the hardware, eliminating the per-process reserved frames.

   c. Use the idea of a "first level handler" in a different way, to send a message describing the page fault; one or more processes running in an environment with dedicated resident frames would receive such messages and handle the page faults. In a large system with many MDS's this would allow sharing among them the page fault handler's dedicated frames. Later we will look at how the one (?) resident frame per process required by this scheme on the current hardware could be saved by pushing the message-sending "first level handler" into the hardware, making it attractive even in a small system with one MDS and minimal real memory.

3. Use a fixed frame for the page fault handler (and probably for all the code it calls). This does not seem fruitful: there is no way to prevent another process from entering this frame before the first one has returned from it.

   Monitors as currently defined (<Redell>MesaProcesses, MesaProcessImpl .ears) can't be used to serialize access to the page fault handler frame(s) because a process must have "a place to stand" to enter a monitor. The monitor entry sequence is a loop several instructions long which must be executed one or more times in the context of a local frame, but the problem is we don't have a local frame.

   Disabling process scheduling (using the IncrementWakeupDisableCounter instruction) is equally fruitless. We dare not leave process scheduling disabled for the duration of the page fault handling episode, but it can't be reenabled until the original process has exited the handler's fixed frame.

A possible change: hardware frame reservation

Eliminating the per-process resident frames required by scheme 2.b. above would require something like a per-MDS "semaphore": a count of the number of frames remaining in the special AV slot, together with a queue of processes waiting for such frames.

A page fault trap would decrement the semaphore counter (by a standard amount) conditional on the result being nonnegative. If the processor was able to decrement the count it would continue as with the current architecture to XFER to the handler; if it was not able to decrement the semaphore it would queue the process on the semaphore (and run another process). Incrementing the semaphore counter when a process finished handling a page fault would have to be atomic with freeing the frames. (This could be done in a way analogous to the trick currently used for reenabling wakeups after handling a frame heap allocation trap.) Whenever the counter was incremented, the processor would also remove one or all processes from the queue to retry the originally faulting instruction.

The semaphore would take only a few words at a standard place in each MDS, although of course it would have to be resident. Processes waiting on a semaphore could be queued through an existing word of the PSB. With only a single size of reserved frames for all the procedures involved in handling a page fault, there would be some wasted space. In a system with many MDS's, the resident space (and accompanying breakage) penalty would have to be paid for each MDS. Finally, modifying the entry vectors of these procedures to use the reserved AV would be a minor nuisance.

Another proposal: hardware fault messages

Scheme 2.c. above converted a page fault to a message sent to a process running in dedicated resident frames. As with scheme 2.b., the per-process dedicated resident frames can be eliminated with new hardware features.

New processor features

We define a new process state, HaltedByPageFault; the PSB of a Halted... process has a field faultedPage: PageNumber (one of the other PSB fields can probably be reused for this purpose). An instruction Restart is provided to clear the Halted... status of all processes with a given faultedPage value. Finally, a mechanism to "receive messages from" faulted processes is provided.

> There could be a HaltedByPageFaultList, on which the hardware would queue processes using their PSB.link field, and an associated wakeup word or condition variable.

As alluded to earlier, the process receiving the fault messages may or may not be in the same MDS as the faulting process. The important thing is that this receiving process have enough resident frames: because it is running in an MDS with all-resident frames, or because it has fixed frames, or even because it is running off a reserved AV slot in the same MDS as the faulting process.

Programming the swapper

With these facilities, writing the page fault handler becomes fairly straightforward. A SwapIn process repeatedly waits for a process to fault, then consults the swapping tables and performs an appropriate ReadPageSet call (see <McJones>FilePageTransfer.cears). A Terminator process repeatedly executes a WaitPageTransferred call, then snaps the page to the correct location, updates the swapping tables, and finally Restarts processes faulted on the newly transferred page. Of course pages must also be swapped out; this is probably done by a SwapOut process in conjunction with the Terminator process.

> There will certainly be limitations on the number of simultaneous transfers in progress at any one time. However now the necessary serialization is possible. Faulting processes immediately enter the Halted... state, but the SwapIn process may not get to them for a while if, for example, it is waiting in a monitor for a storage block to become available.

A comparison of the proposals

Each proposal requires a new process state, some queue management by the page fault trap, and a way to restart waiting/halted processes. Both schemes will work in a system with one MDS, but the second proposal (fault messages) also allows sharing the resident frames dedicated for page fault handling across all MDS's. (One might argue that this interaction between MDS's would be undesirable, but there will be other sharing anyway (e.g. of disks); the message mechanism allows extra flexibility [via policy programmed in the SwapIn process], whether or not it is used.)

Allocating PSBextensions

In the above we have glossed over another problem area. The proposed process-monitor implementation factors a saved process state into a PSB and a separate PSBextension which is only present for a process which has been preempted. The PSBextension holds the evaluation stack, which is required to be empty when a process WAITS. There need be allocated only one PSBextension per preemptible priority level as long as a given process is allowed to run until it WAITS (thus freeing the PSBextension) before another process of the same or lower priority runs. Any implementation of the page fault handler must save the stack of the faulting process, and in either of these proposals the fault will store the stack in the PSBextension for the current priority level, "using up" that PSBextension. (The same limitation of static allocation of PSBextensions would come up with an implementation of time-slicing using a timer driven process at, say, the second-to-the-lowest priority, which reordered the lowest priority Queue of the readyList.)

If multiprogramming during swap-in is to be achieved, more PSBextensions are needed. One way to achieve this would be to run each faultable process at a different priority level, an artificial constraint. Instead we could have the basic process microcode maintain a list of PSBextensions for each level; when a list became empty the only PSB's at that level eligible for scheduling would be those which already had a PSBextension. (In a fancy system design allowing limited recursion of page fault handling it would still be necessary to run the faultable part of the page fault handler at a level with one more PSBextension than processes able to take page faults.)

Frame heap allocation traps and others

Handling a frame heap allocation trap requires a mechanism to provide frames for the mechanism which is itself providing frames. Either of the two proposed mechanisms for reserving resident frames for page fault handling can be used for this purpose too. In the semaphore scheme, there would be two semaphores and two reserved AV entries, one for the page fault handler (with resident frames) and one for the allocation fault handler (with nonresident frames). In the message scheme, there would be an additional process state, HaltedByAllocationFault, an additional message queue (still threaded through the PSB), and an additional Restart instruction. Assuming frame heap allocation faults are considerably rarer than page faults, the message scheme would be advantageous in a system with multiple MDS's.

In general, the message scheme allows a trap to be handled in a totally different environment from that in which it was generated. Thus if the breakpoint trap generated a fault message, the debugger could live in a different MDS than the debuggee, avoiding stealing local address space from the debuggee (and, for what it is worth, isolating the debugger from some forms of damage).