Inter-Office Memorandum

To Pilot Group

Date July 21, 1977

From Dave Redell

Location Palo Alto

Subject Handling page faults in Pilot

Organization SDD/SD

XEROX

Filed on: <Redell>PageFaults.ears

Introduction

A previous memo [1] discussed issues immediately surrounding the occurrence of a page fault and suggested characterizing page faults as I/O-like events, to be serviced by a handler process. This memo investigates the subsequent processing of page faults; the discussion emphasizes structural issues, but is implicitly driven by performance considerations.

The main quantitative goals of the proposal are minimization of memory residency, code complexity, and computational overhead, in roughly that order. The resulting mechanism is mildly recursive, but termination of the recursion after a single cycle is guaranteed. It would appear that a similar approach could be used in managing the tables of the file system.

The swapper

The Pilot virtual memory mechanism consists of two layers: the Virtual Memory Manager and the Swapper [2]. The Swapper consists of three main processes, plus supporting data structures.

SwapIn process: handles page faults using the Active Segment Map and the PageTransfer level of the File system[3].

SwapOut process: writes pages (esp: deactivated swap units) to the disk in anticipation of reuse of their pageframes by the SwapIn process.

SwapFinish process: handles I/O completions as notified by the PageTransfer level, updating the swapper tables and awakening any faulted processes.

Active Segment Map: The main table of the swapper, mapping (for purposes of this discussion) virtual page numbers to swap unit descriptions:

```
SwapUnit: TYPE = RECORD[
  file: FileHandle,
  fileBase: FilePageNumber
  base: PageNumber,
  count: PageCount];
```

In handling page faults, the main participants are the SwapIn and SwapFinish processes, which can be sketched as:

```
SwapIn: PROCESS =
    BEGIN
    faultPage, bufferBase: PageNumber;
    faultFilePage: FilePageNumber;
    su: SwapUnit;

    DO
        faultPage ← WaitForPageFault[];
        su ← FindActiveSwapUnit [ faultPage ];
        faultFilePage ← su.fileBase + (faultPage - su.Base);
        bufferBase ← GetBufferPages [ su.count ];
        ReadPageSet
          [ su.file, su.fileBase, bufferBase, su.count, su.base, faultFilePage];
        ENDLOOP
    END;

SwapFinish: PROCESS =
    BEGIN
    page, bufferPage, base: PageNumber;
    filePage: FilePageNumber;
    su: SwapUnit;
    op: PageOperation;

    DO
        [op,,filePage,bufferPage,base] ← WaitPageTransferred[];
        SELECT op FROM
          read =>
            BEGIN
            su ← FindActiveSwapUnit [ base ];
            page ← su.base + (filePage - su.fileBase)
            MovePagesInRelocatedMemory [ bufferPage, page, 1 ];
            <fix up VM tables...>;
            WakeUpFaultedProcesses [ page ]
            END;
          write =>
            .
            .
            .
          ENDCASE
        ENDLOOP
    END;
```

The swapper as sketched here is quite simple and fast, but this is largely because it assumes that the descriptions of *all* existing swap units are present in the resident Active Segment Map. While this might be feasible in a configuration which made only modest use of the virtual memory, it would, in general, cramp our style, by making the simple *presence* of data in virtual memory intrinsically expensive, whether it was being used or not. We are thus faced with optimizing the tradeoff between two important goals:

- Insuring that all page faults can be handled quickly. This suggests simple algorithms and tables resident in real memory.
- Minimizing the cost of retaining large amounts of inactive information in virtual memory. This implies that the descriptions of such "cold" swap units should themselves migrate to the disk.

This all suggests some form of caching scheme. The Active Segment Map can serve as a cache on a more general Segment Map, most of which can reside on disk. (It would also be nice if, whenever the Active Segment Map is capable of describing the entire working set of the computation, the cost of the backing Segment Map machinery could atrophy to nearly zero.) There seem to be four general ways to build such a caching scheme, as determined by two questions:

Is the cache managed by the Swapper, or by the higher level VM Manager?

Is the Segment Map on the disk accessed via the virtual memory (i.e. page faults) or via direct calls on the File System (i.e reads/writes)?

Note that none of the four approaches rely on the virtual memory to automatically keep the active parts of the Segment Map in real memory for use by the swapper; instead active swap unit descriptions are always *copied* into the Active Segment Map, for two reasons:

- Swap unit descriptions are examples of the kind of small objects which everyone agrees should be cached explicitly to avoid page breakage.
- The Active Segment Map should be a simple, resident table, while the Segment Map is large and should be optimized for external searching (e.g. a B-tree?); decoupling their representations seems like a good idea.

The four approaches are thus:

1. Swapper caches entries accessed via reads: This would greatly increase the complexity of the Swapper. All of the added code would be resident.
2. Swapper caches entries accessed via page faults: This appears to simplify access to the disk, but actually introduces subtle complexity by making the Swapper fully recursive.
3. VM Manager caches entries accessed via reads: This pushes the complexity up to the higher level, which is not intrinsically resident.
4. VM Manager caches entries accessed via page faults: This further simplifies things by allowing the VM Manager, which already runs in virtual memory, to access the Segment Map in this way.

It would appear that approach 4 is preferrable, if no other lurking complexities arise. It is important to note, in this regard, that *this approach introduces a new upward dependency of the Swapper on the VM Manager*. When, in the course of handling a page fault, the Swapper needs a swap unit description which is missing from the Active Segment Map, it notifies a "Helper" in the VM Manager (in a manner to be described); the Helper then fetches the description from the Segment Map in virtual memory, which may cause a second round of page faults. All that is necessary to guarantee that no third round page fault can extend the series is to make all code and data of the Helper, including the Segment Map, *permanently active* (i.e force their swap unit descriptions to reside permanently in the Active Segment Map.) These appear to be the only remnants of the VM Manager which must remain in real memory during times when the entire working set fits into the Active Segment Map. Like the Helper, the rest of the VM Manager can be swapped out when it is not needed. Unlike

the Helper, the other parts of the VM Manager have no intrinsic need to be permanently active.

The Swapper's Helper

The essence of the "Helper" approach is that SwapIn, when in need of a missing swap unit description, "calls for help." It would be nice to do this by having SwapIn notify a separate Helper process of the page fault and then forget the entire affair. This is the simplest way of insuring that SwapIn is ready to process further page faults, including any generated by the Helper. It does mean that the Helper has inherited the responsibility to insure ultimate completion of the original page fault episode. As shown below, it can easily do this by simply referencing the faulted page itself, causing SwapIn to effectively reprocess the original fault.

```
Helper: PROCESS =
    BEGIN
    faultPage: PageNumber;
    su: SwapUnit;

    DO
        faultPage ← GetCallForHelp[];
        su ← FindSwapUnit [ faultPage ]; -- may fault on Segment Map
        AddActiveSwapUnit [ su ];
        [] ← LongPointerToPage [ faultPage ] ↑; -- will fault on original page
        ENDLOOP
    END;

SwapIn: PROCESS =
    BEGIN
    faultPage, bufferBase: PageNumber;
    faultFilePage: FilePageNumber;
    su: SwapUnit;
    found: BOOLEAN;

    DO
        faultPage ← WaitForPageFault[];
        [found, su] ← FindActiveSwapUnit [ faultPage ];
        IF ~ found THEN
            CallForHelp [ faultPage ]
        ELSE
            BEGIN
            faultFilePage ← su.fileBase + (faultPage - su.Base);
            bufferBase ← GetBufferPages [ su.count ];
            ReadPageSet [ su.file, su.fileBase, bufferBase, su.count, su.base,
            faultFilePage]
            END
        ENDLOOP
    END;
```

It should be noted that this approach serializes access to the Segment Map via the single Helper. Thus, although some number of page faults may proceed in parallel (see [1]), only one at a time may touch the Segment Map. This is probably not important, but if it were, multiple Helper processes would seem to be an effective solution.

A few details

The code sketched above calls several useful procedures which were never defined, but whose general intent should be clear. There are a few of these involving synchronization and buffering which should be mentioned, however. They are sketched below in terms of monitors.

Three monitors are defined:

$\mathbf{M}_{pf}$ synchronizes page fault handling with the faulted process(es))

$\mathbf{M}_{asm}$ synchronizes access to the Active Segment Map

$\mathbf{M}_{help}$ synchronizes communication between the Swapper and the Helper.

These provide the following facilities:

```
Mpf: MONITOR =
BEGIN
  WaitForPageFault: ENTRY PROCEDURE [] RETURNS [PageNumber]
    Waits for the I/O-style wakeup from the processor generated by each fault. It
    returns the PageNumber of the faulted page.
  WakeUpFaultedProcesses: ENTRY PROCEDURE [PageNumber]
    Restarts all processes which have faulted on the specified page. {Restarting all processes
    hung on any page fault would be simple and correct, if perhaps too expensive.}
END;

Masm: MONITOR =
BEGIN
  FindActiveSwapUnit: ENTRY PROCEDURE [PageNumber] RETURNS [BOOLEAN,
  SwapUnit]
    Returns TRUE plus a description of the swap unit containing the indicated page (or
    FALSE if no such swap unit is described in the Active Segment Map.)
  AddActiveSwapUnit: ENTRY PROCEDURE [SwapUnit]
    Adds the specified swap unit to the Active Segment Map
  DeleteActiveSwapUnit: ENTRY PROCEDURE [PageNumber]
    Removes the indicated swap unit from the Active Segment Map. This is used for
    replacement of cold entries and updating when a swap unit is redefined. {Note that
    various subtle interactions between DeleteSwapUnit and the algorithms described here are being
    glossed over to simplify the discussion.}
END;

Mhelp: MONITOR =
BEGIN
  CallForHelp: ENTRY PROCEDURE [PageNumber]

    Puts a message identifying the indicated page in a small buffer, notifies the Helper
    process, and returns. The number of slots in the buffer should be equal to the
    number of concurrent page faults allowed.
  GetCallForHelp: ENTRY PROCEDURE [] RETURNS [PageNumber]
    Gets a message from the buffer, waiting if necessary, and returns the PageNumber
    contained in the message.
END;
```

References

[1] Paul McJones, "Page faults and multiple processes per MDS" 14 July 77, <McJones>FaultEvents.ears

[2] _____, "Pilot Storage Facilities" 25 May 77, <McJones>PilotStorage.ears

[3] Paul McJones & Dave Redell, "File Page Transfer Interface" 23 June 77, <McJones>FilePageTransfer.ears