6 Virtual Storage

Nil posse creari de nilo
– Lucretius, De Rerum Natura, I, 155

6.1 Introduction

In this chapter, mechanisms to support virtual storage will be modelled. Virtual storage affords a considerable number of advantages to the operating system designer and user. Virtual storage is allocated in units of a page and pages can be collected into independent segments; virtual storage defines clear boundaries between address spaces so that each process can maintain its own address space. This clearly provides a measure of protection between processes but requires additional methods for exchanging data between processes.

In virtual storage systems, main store is shared between processes in the usual way but is defined as a sequence of *page frames*, each a block of store one page long. The storage belonging to each process is swapped into main store when required and copied to a paging disk (or some other mass-storage device) when not required. Strategies for selecting pages to copy to the paging disk and for determining which page to bring into main store must be defined.

6.2 Outline

The storage system to be designed is to have the following features:

- The virtual store should have four segments: one each for code, data, stack and heap.
- The system uses *demand paging* with reference counting for victim selection.
- Pages can be shared (and unshared) between processes.
- Segments can be shared (and unshared) between processes;

- Storage should be mapped in the sense that disk blocks can be directly mapped to main-store pages and vice versa.
- Message passing will be used for IPC.

The virtual storage system is composed of:

- A page-fault handler. This is invoked when a page fault occurs. It determines the identity of the *logical* page that caused the page fault. It invokes the page-fault driver process and passes to it the identifier of the faulting process and the page number. It unreadies the faulting process.
- A page-fault driver. This takes a message from the page-fault ISR and sends a message to the paging disk to retrieve the page whose reference caused the fault. If there are no free page frames in (main) physical store, it selects a victim page in physical store and sends it to the paging disk. When the faulting page is received from the paging disk, it is copied into a main store page whose physical address is identified with the logical page number in the faulting process. The faulting process is then readied and the driver waits for the next page fault.

The above scheme is sub-optimal. As part of the model, optimisations are suggested, particularly for the interactions between the driver and paging disk.

Even though it is sub-optimal, the above scheme is logically sufficient. It is, therefore, appropriate to concentrate on it as the model for this chapter. This exemplifies the method adopted in this book: capturing the logical functioning of the model is much more important than optimisation. The optimisation included here is introduced as an example of how it can be done without too much of a compromise to the model.

The structure of this kernel is shown in Figure 6.1. Comparison of this figure with the corresponding one in Chapter 4 (Figure 4.1) reveals their similarities. In the current kernel, virtual and not real storage forms the basis of the system. Apart from the need for structures and operations to support virtual storage (the subject of this chapter), the main difference lies in the kernel bootstrapping operations (which are not considered in this book).

6.3 Virtual Storage

In this section, the basic structures required to model a virtual store are introduced.

The following axiomatic definition defines the number of real pages (pages in real store or physical pages) and the size of the page frame. Neither constant is assigned a value, so the specification is of the loose variety.

$$\begin{array}{|l}
numrealpages : \mathbb{N} \\
framesize : \mathbb{N}
\end{array}$$

The basic virtual address is represented by an atomic type:

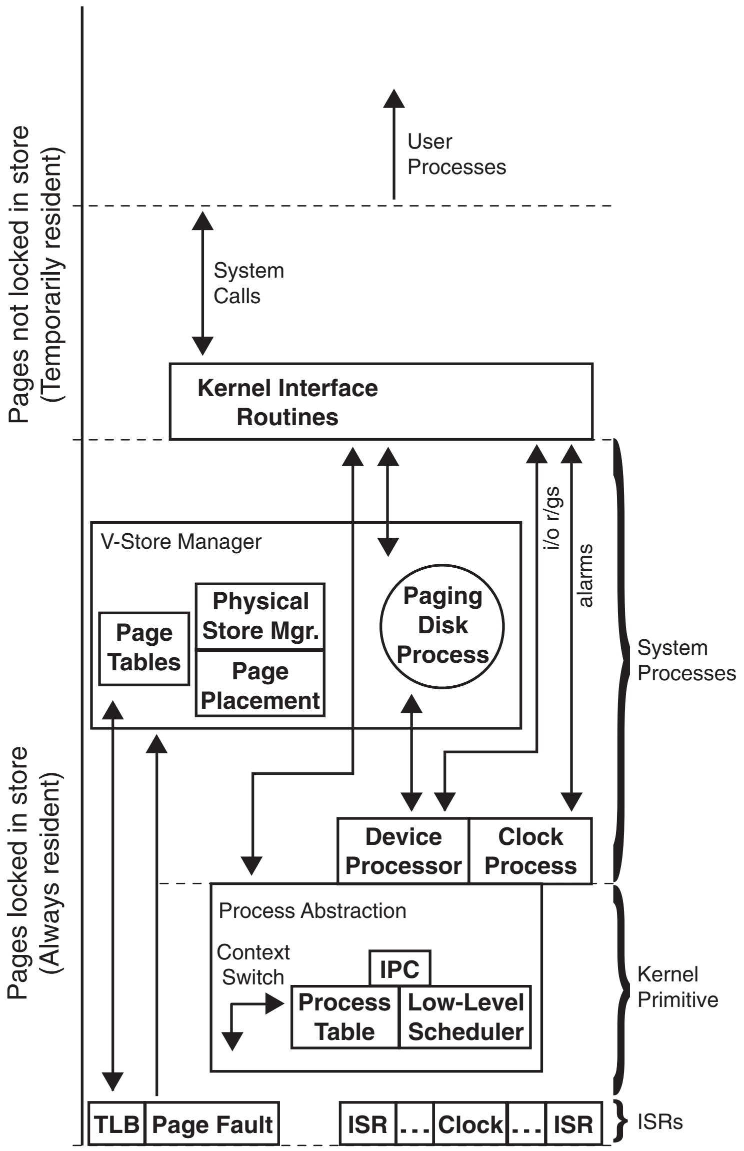

Fig. 6.1. *The layer-by-layer organisation of the kernel, including virtual storage-management modules.*

$[\mathit{VIRTUALADDRESS}]$

$$\mathit{maxvirtpagespersegment} : \mathbb{N}$$

This is the maximum number of virtual pages that a process can own.

$\mathit{PAGEOFFSET} == 1 \mathrel{..} \mathit{framesize}$
$\mathit{PHYSICALPAGENO} == 1 \mathrel{..} \mathit{numrealpages}$
$\mathit{LOGICALPAGENO} == \mathbb{N}$

where:

- *PAGEOFFSET* denotes the offsets into a page;
- *PHYSICALPAGENO* denotes the indices of pages in the main-store page frame;
- *LOGICALPAGENO* denotes the indices of logical pages (pages belonging to a process).

Note that *LOGICALPAGENO* is not bounded above. The reason for this is that the actual number of logical pages that a process can have is a hardware-dependent factor and is not relevant to the current exercise.

For every virtual address, the hardware performs an address translation that maps the virtual address to a logical frame number and an offset into that frame. The signature of this function, *addresstrans*, is:

$$\mathit{addresstrans} : \mathit{VIRTUALADDRESS} \rightarrow \mathit{DECODEDADDRESS}$$

The definition of this function is hardware-specific and is, in any case, not particularly relevant to the current exercise.

The type *DECODEDADDRESS* is defined as:

$$\mathit{DECODEDADDRESS} == \mathit{LOGICALPAGENO} \times \mathit{FRAMEOFFSET}$$

and has the following projections:

$$\begin{array}{l}
\mathit{dlogicalpage} : \mathit{DECODEDADDRESS} \rightarrow \mathit{LOGICALPAGENO} \\
\mathit{dpageoffset} : \mathit{DECODEDADDRESS} \rightarrow \mathit{PAGEOFFSET} \\
\hline
\forall \mathit{addr} : \mathit{DECODEDADDRESS} \bullet \\
\qquad \mathit{dlogicalpage}(\mathit{addr}) = \mathit{fst}\ \ \mathit{addr} \\
\qquad \mathit{dpageoffset}(\mathit{addr}) = \mathit{snd}\ \ \mathit{addr}
\end{array}$$

For a segmented, paged architecture, the address-decoding function can be defined as:

$$\mathit{saddresstrans} : \mathit{VIRTUALADDRESS} \rightarrow \mathit{SDECODEDADDRESS}$$

where:

$SDECODEDADDRESS == SEGMENT \times LOGICALPAGENO \times PAGEOFFSET$

and:

$$\begin{array}{|l}
saddrseg : SDECODEDADDRESS \rightarrow SEGMENT \\
spageno : SDECODEDADDRESS \rightarrow LOGICALPAGENO \\
spagoffset : SDECODEDADDRESS \rightarrow PAGEOFFSET \\
\hline
\forall saddr : SDECODEDADDRESS \bullet \\
\quad saddrseg(saddr) = fst\ saddr \\
\quad spageno(saddr) = fst(snd\ saddr) \\
\quad spagoffset(saddr) = snd^2\ saddr
\end{array}$$

The *PAGEMAP* translates logical to physical page numbers. *PAGEFRAME* translates physical page numbers to the actual pages in store. Finally, *PAGE* defines a storage page as a vector of *PSU*; the vector has *PAGEOFFSET* elements. (*PSU* is, it will be recalled, the *Primary Storage Unit*, which is assumed to be a byte.)

$PAGEMAP == LOGICALPAGENO \twoheadrightarrow PHYSICALPAGENO$
$PAGEFRAME == PHYSICALPAGENO \rightarrow PAGE$
$PAGE == PAGEOFFSET \rightarrow PSU$

The empty page is defined as follows:

$$\begin{array}{|l}
\hline
NullPage : PAGE \\
\hline
NullPage = (\lambda\, i : PAGEOFFSET \bullet 0) \\
\hline
\end{array}$$

It is a page (vector of bytes), *PAGEOFFSET* bytes long, with each byte set to zero.

Pages are associated with a number of flags, including, among others:

- in core (i.e., in main store);
- locked In (i.e., must always remain in main store);
- shared (i.e., shared between at least two processes).

For each process, these properties can be represented by sets (thus simplifying the modelling of pages).

For the remainder of this section, a segmented virtual store will be modelled. The hardware nearest to the model presented here is the Intel X86 series. The reader is warned that this is a *logical* model of segmented, paged storage, not an exact rendition of any existing hardware implementations. Furthermore, details such as the *Translation Lookaside Buffer*, or *TLB*, (the associative store that typically holds a few entries from the current process' page table) are not modelled in detail, the reason being that, as usual, hardware differs considerably in implementation and, in particular, the size of the TLB can vary considerably among MMUs[1].

[1] *Memory Management Unit.*

Although most segmented hardware supports more than four segments per process, for the present, only three segments will be considered. Linux on X86 requires three segments: one each for code, stack and data, although the hardware permits a maximum of 16 segments (the virtual address size is 32 bits). The segment names are as follows:

$$SEGMENT == \{\mathsf{code}, \mathsf{data}, \mathsf{stack}, \mathsf{heap}, \ldots\}$$

A great many programming languages now require heap (dynamic) storage. A problem for dynamic store, as with stacks, is that, quite frequently, it has to be expanded. Within a three-segment organisation, the heap is part of either the stack or data segment; this can limit the maximum size of the heap somewhat on 32-bit machines. To simplify manipulation, it is assumed here that the data segment contains static data (global variables, literal pools, fixed-length buffers and so on) and the heap is given its own segment. The heap can, therefore, grow to the maximum segment size at runtime, as can the stack segment.

$$\begin{array}{|l}
usedsegment : SEGMENT \\
\hline
\forall s : SEGMENT \bullet \\
\quad usedsegment(s) \Leftrightarrow s \in \{\mathsf{code}, \mathsf{data}, \mathsf{stack}, \mathsf{heap}\}
\end{array}$$

It is now possible to define per-process page tables.

Each segment is composed of a number of pages. The following function translates a physical address and segment into a logical page number:

$$\begin{array}{|l}
pages\_in\_segment : APREF \times SEGMENT \times \\
\qquad (APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \mathbb{F}\, LOGICALPAGENO) \rightarrow \\
\qquad\qquad \mathbb{F}\, LOGICALPAGENO \\
\hline
\forall p : APREF;\ sg : SEGMENT;\ f : APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \\
\qquad\qquad \mathbb{F}\, LOGICALPAGENO \bullet \\
\quad pages\_in\_segment(p, sg, f) = f(p)(sg)
\end{array}$$

The following (inverse) functions mark and unmark pages. They are higher-order functions that take the specification of a page and a page attribute map as arguments and return the modified page attribute map.

$$\begin{array}{|l}
mark\_page : APREF \times SEGMENT \times LOGICALPAGENO \times \\
\qquad (APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \mathbb{F}\, LOGICALPAGENO) \rightarrow \\
\qquad\qquad (APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \mathbb{F}\, LOGICALPAGENO) \\
unmark\_page : APREF \times SEGMENT \times LOGICALPAGENO \times \\
\qquad (APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \mathbb{F}\, LOGICALPAGENO) \rightarrow \\
\qquad\qquad (APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \mathbb{F}\, LOGICALPAGENO) \\
\hline
\forall p : APREF;\ sg : SEGMENT;\ lpno : LOGICALPAGENO; \\
\qquad f : APREF \twoheadrightarrow SEGMENT \twoheadrightarrow \mathbb{F}\, LOGICALPAGENO \bullet \\
\quad mark\_page(p, sg, lpno, f) = f(p) \oplus \{sg \mapsto (f(p)(sg) \cup \{lpno\})\} \\
\quad unmark\_page(p, sg, lpno, f) = f(p) \oplus \{sg \mapsto (f(p)(sg) \setminus \{lpno\})\}
\end{array}$$

It is now proved that these two functions are mutual inverses.

Proposition 124. *$mark\_page$ and $unmark\_page^{-1}$ are mutually inverse.*

PROOF. Write $f(p)(sg) = h$, then:

$$\begin{aligned} mark &= f(p) \oplus \{sg \mapsto (h \cup \{lpno\})\} \\ unmark &= f(p) \oplus \{sg \mapsto (h \setminus \{lpno\})\} \end{aligned}$$

Calculating the value of $unmark \circ mark$, we obtain:

$$(f(p) \oplus \{sg \mapsto (h \cup \{lpno\})\}) \oplus \{sg \mapsto (h \setminus \{lpno\})\}$$

Writing $f(p)(sg) = s$, then $mark = s \cup \{lpno\}$ and $unmark = s \setminus \{lpno\}$, so:

$$\begin{aligned} &unmark \circ mark \\ &\quad = (s \setminus \{lpno\}) \circ (s \cup \{lpno\}) \\ &\quad = (s \cup \{lpno\}) \setminus \{lpno\} \\ &\quad = s \end{aligned}$$

Conversely:

$$\begin{aligned} &mark \circ unmark \\ &\quad = (s \cup \{lpno\}) \circ (s \setminus \{lpno\}) \\ &\quad = (s \setminus \{lpno\}) \setminus \{lpno\} \\ &\quad = s \end{aligned}$$

Therefore, *mark* and *unmark* are mutual inverses and the proposition is proved. □

The page table abstraction can be modelled as follows. The variable *freepages* represents those pages in main store that are not allocated to any process. The page table proper is *pagetable*. The variables *executablepages*, *writablepages* and *readablepages* are intended to refer to pages the owner has marked executable (i.e., code pages), read-only (e.g., a constant data segment) and read-write (e.g., a stack). Pages can be shared between processes and some are locked into main store. When a page is locked, it cannot be removed from main store. The kernel's own storage is often marked as locked into main store. It is so locked because a page fault could prevent the kernel from responding in time to a circumstance. It is also necessary to keep track of those pages that are currently in main store: these are referred to as being "in core", hence the name of the variable, *incore*. The *pagecount* counts the number of pages in each segment of each process. There is an *a priori* limit to the number of pages in a segment and *pagecount* is intended to keep track of this and to provide a mechanism for raising an error condition if this limit is exceeded. The final variable, *smap*, is a relation between elements of *PAGESPEC*; it denotes those pages that are shared and it will be explained in more detail below.

There are different ways to organise page tables. The simplest is a linear sequence of page references. As virtual storage sizes increase, simple linear structures do not perform well, so tree-like structures are to be preferred. These trees can be arranged to perform mapping on two or three levels. The model defined here is intended to be suggestive of a tree structure, even though it can also be implemented as a table.

The class that follows defines an abstract data type. It represents the page table type. The type exports a large number of operations and has the most complex invariant in this book.

$PageTables$

$$\upharpoonright(INIT, HaveFreePages, NumberOfFreePages, AllocateFreePage,\\ MakePageFree, PhysicalPageNo, InitNewProcessPageTable,\\ RemoveProcessFromPageTable, AddPageToProcess, HasPageInStore,\\ IncProcessPageCount, DecProcessPageCount,\\ LatestPageCount, UpdateMainstorePage,\\ RemovePageFromProcessTable, RemovePageProperties,\\ RemovePageFromProcess, IsPageInMainStore, MarkPageAsIn,\\ MarkPageAsOut, IsSharedPage, MarkPageAsShared,\\ UnsharePage, IsLockedPage, LockPage, UnlockPage,\\ MakePageReadable, MakePageNotReadable, MakePageExecutable,\\ IsPageExecutable, MakePageNotExecutable, MakePageWritable,\\ IsPageWritable, MakePageNotWritable)$$

$$\begin{array}{l}
freepages : \mathbb{F}\, PHYSICALPAGENO\\
pagetable : APREF \mathrel{⇻} SEGMENT \mathrel{⇻} PAGEMAP\\
executablepages, writablepages, readablepages,\\
sharedpages, lockedpages,\\
incore : APREF \mathrel{⇻} SEGMENT \mathrel{⇻} \mathbb{F}\, LOGICALPAGENO\\
pagecount : APREF \mathrel{⇻} SEGMENT \mathrel{⇻} \mathbb{N}\\
smap : PAGESPEC \leftrightarrow PAGESPEC
\end{array}$$

$InvPageTables$

$$\begin{array}{l}
0 \leq \#freepages \leq numrealpages\\
\mathrm{dom}\, incore \subseteq \mathrm{dom}\, pagetable\\
\mathrm{dom}\, sharedpages \subseteq \mathrm{dom}\, pagetable\\
\mathrm{dom}\, lockedpages \subseteq \mathrm{dom}\, pagetable\\
\mathrm{dom}\, pagecount = \mathrm{dom}\, pagetable\\
\mathrm{dom}\, executablepages = \mathrm{dom}\, pagetable\\
\mathrm{dom}\, writablepages = \mathrm{dom}\, pagetable\\
\mathrm{dom}\, readablepages = \mathrm{dom}\, pagetable
\end{array}$$

INIT

$$\begin{array}{l}
freepages' = \varnothing \\
\mathrm{dom}\, pagetable' = \varnothing \\
\mathrm{dom}\, sharedpages' = \varnothing \\
\mathrm{dom}\, lockedpages' = \varnothing \\
\mathrm{dom}\, incore' = \varnothing \\
\mathrm{dom}\, pagecount' = \varnothing \\
\mathrm{dom}\, executablepages' = \varnothing \\
\mathrm{dom}\, writablepages' = \varnothing \\
\mathrm{dom}\, readablepages' = \varnothing \\
\mathrm{dom}\, smap = \varnothing
\end{array}$$

$HaveFreePages \mathrel{\widehat{=}} \ldots$

$NumberOfFreePages \mathrel{\widehat{=}} \ldots$

$AllocateFreePage \mathrel{\widehat{=}} \ldots$

$MakePageFree \mathrel{\widehat{=}} \ldots$

$PhysicalPageNo \mathrel{\widehat{=}} \ldots$

$InitNewProcessPageTable \mathrel{\widehat{=}} \ldots$

$RemoveProcessFromPageTable \mathrel{\widehat{=}} \ldots$

$AddPageToProcess \mathrel{\widehat{=}} \ldots$

$HasPageInStore \mathrel{\widehat{=}} \ldots IncProcessPageCount \mathrel{\widehat{=}} \ldots$

$DecProcessPageCount \mathrel{\widehat{=}} \ldots$

$LatestPageCount \mathrel{\widehat{=}} \ldots$

$UpdateMainstorePage \mathrel{\widehat{=}} \ldots$

$RemovePageFromPageTable \mathrel{\widehat{=}} \ldots$

$RemovePageProperties \mathrel{\widehat{=}} \ldots$

$RemovePageFromProcess \mathrel{\widehat{=}} \ldots$

$IsPageInMainStore \mathrel{\widehat{=}} \ldots$

$MarkPageAsIn \mathrel{\widehat{=}} \ldots$

$MarkPageAsOut \mathrel{\widehat{=}} \ldots$

$IsSharedPage \mathrel{\widehat{=}} \ldots$

$MarkPageAsShared \mathrel{\widehat{=}} \ldots$

$UnsharePage \mathrel{\widehat{=}} \ldots$

$IsLockedPage \mathrel{\widehat{=}} \ldots$

$LockPage \mathrel{\widehat{=}} \ldots$

$UnlockPage \mathrel{\widehat{=}} \ldots$

$MakePageReadable \mathrel{\widehat=} \ldots$

$MakePageNotReadable \mathrel{\widehat=} \ldots$

$MakePageExecutable \mathrel{\widehat=} \ldots$

$IsPageExecutable \mathrel{\widehat=} \ldots$

$MakePageNotExecutable \mathrel{\widehat=} \ldots$

$MakePageWritable \mathrel{\widehat=} \ldots$

$IsPageWritable \mathrel{\widehat=} \ldots$

$MakePageNotWritable \mathrel{\widehat=} \ldots$

It will be noted that the invariant is partially stated in the class definition. The remainder is specified by the *InvPageTables* schema defined below after the other operations have been defined. This will bring the invariant closer to some of the proofs in which it is required.

The following schema represents the test that there are pages in main store (physical pages) that are free.

HaveFreePages

$freepages \neq \varnothing$

NumberOfFreePages

$np! : \mathbb{N}$

$np! = \#freepages$

The following operation models the allocation of a free page to a process. It removes the page denoted by *ppno*! from the set of free pages, *freepages*.

AllocateFreePage

$\Delta(freepages)$
$ppno! : PHYSICALPAGENO$

$ppno! \in freepages$
$freepages' = freepages \setminus \{ppno!\}$

Proposition 125. *AllocateFreePage implies that* $\#freepages' = freepages + 1$.

Proof.

$$\begin{aligned}
&\#freepages' \\
&\quad = \#(freepages \setminus \{ppno?\}) \\
&\quad = \#freepages - \#\{ppno?\} \\
&\quad = \#freepages - 1
\end{aligned}$$

$\square$

Proposition 126. *If $freepages = n$, $AllocateFreePage^n$ implies $freepages' = 0$.*

Proof. By induction, using the last proposition. □

The next operation returns a page to the set of free pages.

$$\begin{array}{|l}
\hline \textit{MakePageFree} \\
\Delta(freepages) \\
ppno? : PHYSICALPAGENO \\
\hline
freepages' = freepages \cup \{ppno?\} \\
\hline
\end{array}$$

Proposition 127. *$MakePageFree$ implies that $\#freepages' = \#freepages - 1$.*

Proof.

$$\begin{aligned}
&\#freepages' \\
&\quad = \#(freepages \cup \{ppno?\}) \\
&\quad = \#freepages + \#\{ppno?\} \\
&\quad = \#freepages + 1
\end{aligned}$$

□

Proposition 128. *$AllocateFreePage[p/ppno!] \fatsemi MakePageFree[p/ppnp?]$ implies that $freepages' = freepages$.*

Proof. The sequential composition can be written as:

$$\begin{aligned}
&\exists freepages'' : \mathbb{F}\, PHYSICALPAGENO \mid freepages'' = freepages \setminus \{ppno!\} \bullet \\
&\quad freepages' = freepages \cup \{ppno?\}
\end{aligned}$$

Renaming and simplifying:

$$freepages' = (freepages \setminus \{p\}) \cup \{p\}$$

Then:

$$\begin{aligned}
&(freepages \setminus \{p\}) \cup \{p\} \\
&\quad = (freepages \setminus \{p\}) \cup \{p\} \\
&\quad = freepages \cup (\{p\} \setminus \{p\}) \\
&\quad = freepages \cup \varnothing \\
&\quad = freepages
\end{aligned}$$

□

The *PhysicalPageNo* operation contains a use of the *pagetable* variable. This variable is a higher-order function. Its use might appear a little odd.

Essentially, to obtain the physical page number corresponding to a logical page number, the process has to locate the segment in which the page occurs and then translate the logical page number.

PhysicalPageNo

$$
\begin{array}{l}
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
ppgno! : PHYSICALPAGENO \\
\hline
ppgno! = pagetable(p?)(sg?)(lpno?)
\end{array}
$$

When a process is allocated, it is given an entry in the page table. The following schema models this operation. It just adds the process' identifier as a key in the subtables and sets everything to zero (empty or $\varnothing$).

InitNewProcessPageTable

$$
\begin{array}{l}
\Delta(incore, sharedpages, lockedpages) \\
p? : APREF \\
\hline
pagetable' = pagetable \cup \{p? \mapsto \mathsf{code} \mapsto \varnothing\} \cup \{p? \mapsto \mathsf{data} \mapsto \varnothing\} \\
\qquad\qquad \cup\{p? \mapsto \mathsf{stack} \mapsto \varnothing\} \cup \{p? \mapsto \mathsf{heap} \mapsto \varnothing\} \\
incore' = (incore \cup \{p? \mapsto \mathsf{code} \mapsto \varnothing\}) \cup (\{p? \mapsto \mathsf{data} \mapsto \varnothing\} \\
\qquad\qquad (\cup\{p? \mapsto \mathsf{stack} \mapsto \varnothing\}(\cup\{p? \mapsto \mathsf{heap} \mapsto \varnothing\}))) \\
sharedpages' = (sharedpages \cup \{p? \mapsto \mathsf{code} \mapsto \varnothing\}) \cup (\{p? \mapsto \mathsf{data} \mapsto \varnothing\} \\
\qquad\qquad (\cup\{p? \mapsto \mathsf{stack} \mapsto \varnothing\}(\cup\{p? \mapsto \mathsf{heap} \mapsto \varnothing\}))) \\
lockedpages' = (lockedpages \cup \{p? \mapsto \mathsf{code} \mapsto \varnothing\}) \cup (\{p? \mapsto \mathsf{data} \mapsto \varnothing\} \\
\qquad\qquad (\cup\{p? \mapsto \mathsf{stack} \mapsto \varnothing\}(\cup\{p? \mapsto \mathsf{heap} \mapsto \varnothing\})))
\end{array}
$$

Proposition 129. *InitNewProcessPageTable implies that the new process has no pages.*

PROOF. For a process to have pages, it must have at least one page in *at least one* segment. However, for a process, p, and all segments, sg, $pagetable'(p)(sg) = \varnothing$. □

Corollary 11. *InitNewProcessPageTable implies that the new process has no in-core pages.*

PROOF. Similar to the above. □

Similar results can be proved for all other page attributes, e.g., locked pages.

Conversely, when a process terminates or is killed, its storage is returned to the free pool and all of the information associated with it in the page tables is removed. The following schema models this operation:

RemoveProcessFromPageTable

$\Delta(pagetable, incore, sharedpages, lockedpages)$
$p? : APREF$

$pagetable' = \{p?\} \ntriangleleft pagetable$
$incore' = \{p?\} \ntriangleleft incore$
$sharedpages' = \{p?\} \ntriangleleft sharedpages$
$lockedpages' = \{p?\} \ntriangleleft lockedpages$

Proposition 130. *The predicate of RemoveProcessFromPageTable implies that $p \notin \operatorname{dom} pagetable$.*

Proof. The first line of the predicate is $pagetable' = \{p?\} \ntriangleleft pagetable$. Taking domains:

$$\begin{aligned}\operatorname{dom} pagetable' &= \operatorname{dom}(\{p?\} \ntriangleleft pagetable)\\ &= (\operatorname{dom} pagetable) \setminus \{p?\}\end{aligned}$$

□

Proposition 131. *For any $p : APREF$, $RemoveProcessFromPageTable[p/p?]$ implies that the process:*

1. *is no longer incore, swappable, etc.; and*
2. *is no longer in the page table for its owning process.*

Proof. Each conjunct of the predicate employs the domain subtraction operation ($\ntriangleleft$) to remove p from the domain of each function. This implies that p is removed from each table. □

Propositions about page attributes can be proved. They follow the pattern of the last proposition.

When the storage image of a process is augmented by the addition of fresh pages, the following operation is the basic one used to extend the process' page table entry. Each page is specified as a process reference, a segment and a logical page number; in addition, the physical page number of the page to be added is also included. Since the process and segment are already present in the table, the logical to physical page number mapping is added to the table at the specified point.

AddPageToProcess

$\Delta(pagetable)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$
$ppno? : PHYSICALPAGENO$

$pagetable' = pagetable(p?)(sg?) \cup \{lpno? \mapsto ppno?\}$

There now follows a predicate that returns true when the process specified by $p?$ has at least one page in main store:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{HasPageInStore}\ }}\\
p? : \mathit{APREF}\\
\hline
p? \in \mathrm{dom}\, \mathit{incore}\\
\hline
\end{array}$$

The per-segment page count is incremented by the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{IncProcessPageCount}\ }}\\
\Delta(\mathit{pagecount})\\
p? : \mathit{APREF}\\
sg? : \mathit{SEGMENT}\\
\hline
\mathit{pagecount}' = \mathit{pagecount}(p?) \oplus \{sg? \mapsto \mathit{pagecount}(p?)(sg?) + 1\}\\
\hline
\end{array}$$

The counter is decremented by the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{DecProcessPageCount}\ }}\\
\Delta(\mathit{pagecount})\\
p? : \mathit{APREF}\\
\hline
\mathit{pagecount}' = \mathit{pagecount}(p?) \oplus \{p? \mapsto \mathit{pagecount}(p?)(sg?) - 1\}\\
\hline
\end{array}$$

When a page is added to a segment, the page count is incremented. When a page is removed from a segment, the page count is decremented. The current value of the page count is obtained by the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{LatestPageCount}\ }}\\
p? : \mathit{APREF}\\
sg? : \mathit{SEGMENT}\\
\mathit{lpno}! : \mathit{LOGICALPAGENO}\\
\hline
\mathit{lpno}! = \mathit{pagecount}(p?)(sg?)\\
\hline
\end{array}$$

If the logical to physical page mapping is changed, the following schema performs the update in the page table.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{UpdateMainStorePage}\ }}\\
\Delta(\mathit{pagetable})\\
p? : \mathit{APREF}\\
sg? : \mathit{SEGMENT}\\
\mathit{lpno}? : \mathit{LOGICALPAGENO}\\
\mathit{ppno}? : \mathit{PHYSICALPAGENO}\\
\hline
\mathit{pagetable}' = \mathit{pagetable}(p?)(sg?) \oplus \{\mathit{lpno}? \mapsto \mathit{ppno}?\}\\
\hline
\end{array}$$

When a page is removed from the page table, the entry representing it must be removed. The removal operation is defined as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemovePageFromPageTable}} \\
\Delta(pagetable) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
(\exists\, pmap : PAGEMAP \mid pmap = pagetable(p?)(sg?) \bullet \\
\qquad pagetable' = pagetable(p?) \oplus \{sg? \mapsto (\{lpno?\} \lhd pmap)\})
\end{array}$$

The removal of a page also requires the removal of the attributes of that page. The attributes are removed using the *unmark_page* function (when a page is allocated, the attributes it possesses are marked using the *mark_page* function).

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemovePageProperties}} \\
\Delta(executablepages, readablepages, writablepages, sharedpages, \\
\qquad lockedpages, incore) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
executablepages' = unmark\_page(p?, sg?, lpno?, executablepages) \\
readablepages' = unmark\_page(p?, sg?, lpno?, readablepages) \\
writablepages' = unmark\_page(p?, sg?, lpno?, writablepages) \\
sharedpages' = unmark\_page(p?, sg?, lpno?, sharedpages) \\
lockedpages' = unmark\_page(p?, sg?, lpno?, lockedpages) \\
incore' = unmark\_page(p?, sg?, lpno?, incore)
\end{array}$$

Finally, the high-level operation to remove a page from a process is defined as follows:

$$\begin{array}{l}
RemovePageFromProcess \mathrel{\widehat=} \\
\qquad RemovePageFromProcessTable \mathbin{;} \\
\qquad MakePageFree \mathbin{;} \\
\qquad RemovePageProperties
\end{array}$$

It is possible to determine whether a page is in main store by determining whether it is in the *incore* attribute. The following schema defines this predicate. Note that it uses the *pages_in_segment* function.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{IsPageInMainStore}} \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
lpno? \in pages\_in\_segment(p?, sg?, incore)
\end{array}$$

Proposition 132. *$IsPageInMainStore$ iff $lpno?$ is an in-core page.*

Proof. The predicate states that $lpno? \in pages\_in\_segment(p?, sg?, incore)$. By the definition of $pages\_in\_segment$,

$$lpno? \in incore(p?)(sg?)$$

$\square$

Proposition 133. *$IsSharedPage$ iff $lpno?$ is a shared page; that is, iff $lpno?$ is an element of $sharedpages(p)(sg)$, for some p and sg.*

Proof. Similar to the previous proof. $\square$

Proposition 134. *$IsLockedPage$ iff $lpno?$ is a locked page; that is, iff $lpno?$ is an element of $lockedpages$.*

Proof. Similar to the previous proof. $\square$

When a page is swapped into main store, it is marked as being "in". The following schema performs this marking. The schema that immediately follows marks pages as "out" (i.e., as not being main-store resident).

```
── MarkPageAsIn ──────────────────────────
│ Δ(incore)
│ p? : APREF
│ sg? : SEGMENT
│ lpno? : LOGICALPAGENO
├──────────
│ incore′ = mark_page(p?, sg?, lpno?, incore)
└──────────────────────────────────────────
```

Proposition 135. *$MarkPageAsIn$ implies that $lpno?$ is an element of $incore'$.*

Proof. As can be seen from the schema, the predicate is $incore' = mark\_page(p?, sg?, lpno?, incore)$. Substituting the definition of $mark\_page$:

$$\begin{aligned} incore' &= mark\_page(p?, sg?, lpno?, incore) \\ &= incore(p?) \oplus \{sg \mapsto (incore(p?)(sg) \cup \{lpno?\})\} \end{aligned}$$

$\square$

Proposition 136. *For fixed arguments, $p : APREF$, $s : SEGMENT$ and $l : LOGICALPAGENO$, $MarkPageAsIn[p/p?, s/sg?, l/lpno?]^n$ has the same effect as $MarkPageAsIn[p/p?, s/sg?, l/lpno?]$.*

Proof. This proposition is to be taken as:

$$MarkPageAsIn^n \Leftrightarrow MarkPageAsIn$$

The proposition is proved by substitution from the following general property of sets:

$$(S \cup \{x\}) \cup \{x\} = S \cup \{x\}$$

(i.e., the absorbtive law of set union). □

$$\begin{array}{|l}
\hline
\textit{MarkPageAsOut} \\
\Delta(incore) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
incore' = unmark\_page(p?, sg?, lpno?, incore) \\
\hline
\end{array}$$

Proposition 137. *MarkPageAsOut is satisfied iff lpno? is not an element of incore.*

Proof. Substituting the definition of *unmark_page* into the predicate of the schema *MarkPageAsOut*:

$$\begin{aligned}
incore' &= unmark\_page(p?, sg?, lpno?, incore) \\
&= incore(p?) \oplus \{sg? \mapsto (incore(p?)(sg?) \setminus \{lpno?\})\}
\end{aligned}$$

□

Proposition 138. *For fixed arguments, $p : APREF$, $s : SEGMENT$ and $l : LOGICALPAGENO$, $MarkPageAsOut[p/p?, s/sg?, l/lpno?]^n$ has the same effect as $MarkPageAsOut[p/p?, s/sg?, l/lpno?]$.*

Proof. The statement of the proposition is to be taken as:

$$MarkPageAsOut^n \Leftrightarrow MarkPageAsOut$$

The proposition is proved by substitution from the following general property of sets:

$$(S \setminus \{x\}) \setminus \{x\} = S \setminus \{x\}$$

□

Proposition 139. *$MarkPageAsIn[l/lpno?] \fatsemi MarkPageAsOut[l/lpno?]$ implies that $incore' = incore$.*

Proof. Writing out the predicates and performing the obvious substitutions:

$$unmark\_page(p?, sg?, lpno?, mark\_page(p?, sg?, lpno?, incore))$$

The result follows from the fact that *unmark_page* and *mark_page* are mutually inverse. □

Proposition 140. $MarkPageAsOut[l/lpno?] \mathbin{;} MarkPageAsIn[l/lpno?]$ *implies that* $incore = incore'$.

Proof. Writing out the predicates and performing the obvious substitutions:

$$mark\_page(p?, sg?, lpno?, unmark\_page(p?, sg?, lpno?, incore))$$

The result follows from the fact that *unmark_page* and *mark_page* are mutually inverse. □

The next few schemata set and unset attributes in pages. The attributes are represented by the various tables in the *PageTables* class, such as *sharedpages*, *readable* and *locked*. The schemata naturally fall into three sets: one to perform a test and one to set the attribute and one to unset it. The schemata in each of these sets have the same structure. That structure is the obvious one and is quite simple. For these reasons, the schemata will not be described in English: the formal notation can stand on its own.

IsSharedPage

$$\begin{array}{|l}
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
lpno? \in pages\_in\_segment(p?, sg?, sharedpages)
\end{array}$$

MarkPageAsShared

$$\begin{array}{|l}
\Delta(sharedpages) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
sharedpages' = mark\_page(p?, sg?, lpno?, sharedpages)
\end{array}$$

UnsharePage

$$\begin{array}{|l}
\Delta(sharedpages) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
sharedpages' = unmark\_page(p?, sg?, lpno?, sharedpages)
\end{array}$$

$$\begin{array}{|l}
\textit{IsLockedPage} \\
\hline
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
lpno? \in pages\_in\_segment(p?, sg?, lockedpages) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{LockPage} \\
\hline
\Delta(lockedpages) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
lockedpages' = mark\_page(p?, sg?, lpno?, lockedpages) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{UnlockPage} \\
\hline
\Delta(lockedpages) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
lockedpages' = unmark\_page(p?, sg?, lpno?, lockedpages) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{MakePageReadable} \\
\hline
\Delta(readablepages) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
readablepages' = mark\_page(p?, sg?, lpno?, readablepages) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{IsPageReadable} \\
\hline
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
lpno? \in readablepages(p?)(sg?) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{MakePageNotReadable} \\
\hline
\Delta(readablepages) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
readablepages' = unmark\_page(p?, sg?, lpno?, readablepages) \\
\hline
\end{array}$$

MakePageExecutable

$\Delta(executablepages)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$executablepages' = mark\_page(p?, sg?, lpno, executablepages)$

IsPageExecutable

$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$lpno? \in executablepages(p?)(sg?)$

MakePageNotExecutable

$\Delta(executablepages)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$executablepages' = unmark\_page(p?, sg?, lpno, executablepages)$

MakePageWritable

$\Delta(writablepages)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$writablepages' = mark\_page(p?, sg?, lpno, writablepages)$

IsPageWritable

$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$lpno? \in writablepages(p?)(sg?)$

MakePageNotWritable

$\Delta(writablepages)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$writablepages' = unmark\_page(p?, sg?, lpno, writablepages)$

Finally, we come to *InvPageTables*, the page table invariant. As can be seen, this is quite an invariant. Because of its length, it was not possible to make it fit within the Object-Z class box without obfuscating the specification even more. The invariant is mostly concerned with ensuring that each page is represented correctly. For example, no free page has a corresponding page-table entry; all locked pages are always in main store, and so on. Readers can inspect the various clauses of the invariant for themselves.

This invariant clearly demonstrates the need for higher-order logics!

$$
\begin{array}{l}
\mathit{InvPageTables} \mathrel{\widehat=} \\
(\forall\, ppno : \mathit{PHYSICALPAGENO} \bullet \\
\quad ppno \in \mathit{freepages} \Leftrightarrow \\
\quad \neg\,(\exists\, p : \mathit{APREF};\ sg : \mathit{SEGMENT};\ lpno : \mathit{LOGICALPAGENO} \bullet \\
\qquad\qquad ppno = \mathit{pagetables}(p)(sg)(lpno))) \\
(\forall\, p : \mathit{APREF};\ sg : \mathit{SEGMENT} \bullet \\
\quad p \in \operatorname{dom} \mathit{lockedpages} \land sg \in \operatorname{dom} \mathit{lockedpages}(p) \Rightarrow \\
\qquad \mathit{lockedpages}(p)(sg) \subseteq \mathit{incore}(p)(sg)) \\
(\forall\, p : \mathit{APREF};\ sg : \mathit{SEGMENT} \bullet \\
\quad p \in \operatorname{dom} \mathit{incore} \land sg \in \operatorname{dom} \mathit{incore}(p) \Rightarrow \\
\qquad \mathit{incore}(p)(sg) \subseteq \operatorname{dom} \mathit{pagetable}(p)(sg)) \\
(\forall\, p : \mathit{APREF};\ sg : \mathit{SEGMENT} \bullet \\
\quad p \in \operatorname{dom} \mathit{sharedpages} \land sg \in \operatorname{dom} \mathit{sharedpages}(p) \Rightarrow \\
\qquad \mathit{sharedpages}(p)(sg) \subseteq \operatorname{dom} \mathit{pagetable}(p)(sg)) \\
(\forall\, p : \mathit{APREF};\ sg : \mathit{SEGMENT} \bullet \\
\quad p \in \operatorname{dom} \mathit{lockedpages} \land sg \in \operatorname{dom} \mathit{lockedpages}(p) \Rightarrow \\
\qquad \mathit{lockedpages}(p)(sg) \subseteq \operatorname{dom} \mathit{pagetable}(p)(sg))
\end{array}
$$

These clauses are intended to represent the disjointness of segments.

$$
\begin{array}{l}
(\forall\, p : \mathit{APREF} \mid p \in \operatorname{dom} \mathit{pagetable} \bullet \\
\quad (\forall\, sg_1, sg_2 : \mathit{SEGMENT} \bullet \\
\qquad (sg_1 \neq sg_2 \land \\
\qquad\quad sg_1 \in \operatorname{dom} \mathit{pagetable}(p) \land sg_2 \in \operatorname{dom} \mathit{pagetable}(p)) \Rightarrow \\
\qquad\qquad \operatorname{dom} \mathit{pagetable}(p)(sg_1) \cap \operatorname{dom} \mathit{pagetable}(p)(sg_2) = \varnothing)) \\
(\forall\, p : \mathit{APREF} \mid p \in \operatorname{dom} \mathit{pagetable} \bullet \\
\quad (\forall\, sg_1, sg_2 : \mathit{SEGMENT} \bullet \\
\qquad (sg_1 \neq sg_2 \land \\
\qquad\quad sg_1 \in \operatorname{dom} \mathit{pagetable}(p) \land sg_2 \in \operatorname{dom} \mathit{pagetable}(p)) \Rightarrow \\
\qquad\qquad \operatorname{ran} \mathit{pagetable}(p)(sg_1) \cap \operatorname{ran} \mathit{pagetable}(p)(sg_2) = \varnothing)) \\
(\forall\, p : \mathit{APREF};\ sg : \mathit{SEGMENT} \bullet \\
\quad p \in \operatorname{dom} \mathit{executablepages} \land sg \in \mathit{executablepages}(p) \Rightarrow \\
\qquad \mathit{executablepages}(p)(sg) \subseteq \operatorname{dom} \mathit{pagetable}(p)(sg)) \\
(\forall\, p : \mathit{APREF};\ sg : \mathit{SEGMENT} \bullet \\
\quad p \in \operatorname{dom} \mathit{readablepages} \land sg \in \mathit{readtablepages}(p) \Rightarrow \\
\qquad \mathit{readablepages}(p)(sg) \subseteq \operatorname{dom} \mathit{pagetable}(p)(sg))
\end{array}
$$

$$\begin{array}{l}
(\forall p : APREF;\ sg : SEGMENT \bullet \\
\quad p \in \operatorname{dom} writablepages \land sg \in writablepages(p) \Rightarrow \\
\quad\quad writablepages(p)(sg) \subseteq \operatorname{dom} pagetable(p)(sg))
\end{array}$$

$$\begin{array}{l}
(\forall pg : PAGESPEC \mid pg \in \operatorname{dom} pmap \bullet \\
\quad (\exists p : APREF;\ sg : SEGMENT;\ lpno : LOGICALPAGENO \bullet \\
\quad\quad (p = pgspecpref(pg) \land \\
\quad\quad\quad sg = pgspecseg(pg) \land \\
\quad\quad\quad lpno = pgspeclpno(pg)) \Rightarrow \\
\quad\quad (p \in \operatorname{dom} pagetable \land sg \in \operatorname{dom} pagetable(p) \land \\
\quad\quad\quad lpno \in \operatorname{dom} pagetable(p)(sg))))
\end{array}$$

$$\begin{array}{l}
(\forall pg : PAGESPEC \mid pg \in \operatorname{ran} pmap \bullet \\
\quad (\exists p : APREF;\ sg : SEGMENT;\ lpno : LOGICALPAGENO \bullet \\
\quad\quad (p = pgspecpref(pg) \land \\
\quad\quad\quad sg = pgspecseg(pg) \land \\
\quad\quad\quad lpno = pgspeclpno(pg)) \Rightarrow \\
\quad\quad (p \in \operatorname{dom} pagetable \land sg \in \operatorname{dom} pagetable(p) \land \\
\quad\quad\quad lpno \in \operatorname{dom} pagetable(p)(sg))))
\end{array}$$

Proposition 141.

$$\begin{array}{l}
InitNewProcessPageTable \Rightarrow \\
\quad (\forall s : SEGMENT \bullet \\
\quad\quad incore = \varnothing \\
\quad\quad \land\ sharedpages = \varnothing \\
\quad\quad \land\ lockedpages = \varnothing)
\end{array}$$

PROOF. By the predicate of the *InitNewProcessPageTable* predicate. □

Proposition 142. $\forall p : PAGE \bullet locked(p) \Leftrightarrow \neg\ swappable(p)$

PROOF. The class invariant states that $\operatorname{dom} lockedpages \subseteq \operatorname{dom} pagetables$. This permits us to infer that:

$$\begin{array}{l}
\forall p : LOGICALPAGENO \bullet \\
\quad p \in lockedpages(p)(sg) \\
\quad\quad \Rightarrow p \in (\operatorname{dom} pagetable(p)(sg))
\end{array}$$

for all $p \in IPREF$ and $sg \in SEGMENT$.

Again, by the same invariant:

$$lockedpages(p)(sg) \subseteq incore(p)(sg)$$

(again, for all p and sg as above).

These two formulæ ensure that every locked page exists in store.

$\Rightarrow$: By the predicate of *PageFaultDriver.findVictimLogicalPage* (q.v.):

$pagetable(p)(sg)(l) = v! \land l \notin lockedpage$

First, applying the substitutions $[p/p!, sg/sg!, l/l!]$, then simplifying and rearranging the predicate of *findVictimLogicalPage*, we obtain:

$$\begin{array}{l} p \in \operatorname{dom} pagetable \\ \quad \land sg \in \operatorname{dom} pagetable(p) \\ \quad \land l \in \operatorname{dom} pagetable(p)(sg) \\ \quad \land l \notin lockedpages(p)(sg) \\ \quad \land pagetable(p)(sg)(l) = v \end{array}$$

where v is the page to be swapped.

From this, it can be concluded that v cannot be a locked page.

$\Leftarrow$. If a page is not swappable, it is locked. Consider the set of swappable pages, S, by the definition of *findVictimLogicalPage* in the substitution instance above, any logical page, l, cannot be in S because $l \notin lockedpages(p)(sg)$. □

Proposition 143. *For all processes, lockedpages* $\cap$ *swappablepages* $= \varnothing$*.*

Proof. This follows immediately from the previous proposition. □

Proposition 144. *If a page, p, is in main store, IsPageInMainStore*$[p/...]$ *is satisfied.*

Proof. By the predicate of *IsPageInMainStore*, it is clear that

$$lpno? \in pages\_in\_segment(p, s, incore)$$

and that: $pages\_in\_segment(x, y, f) = f(x)(y)$; its range is a set.

For a page actually to be in store, the following must be satisfied:

$$\begin{array}{l} IsPageInMainstore \text{ implies that:} \\ \quad (\exists\, p : IPREF;\ sg : SEGMENT;\ pp : PHYSICALPAGENO \bullet \\ \qquad pagetable(p)(sg)(lpno) = pp \\ \qquad \land pp \notin freepages) \end{array}$$

By the invariant:

$$\begin{array}{l} \forall\, pp : PHYSICALPAGENO \bullet \\ \quad pp \in freepages \Rightarrow \\ \qquad \neg\ (\exists\, p : IPREF;\ sg : SEGMENT;\ l : LOGICALPAGENO \bullet \\ \qquad\quad pp = pagetables(p)(sg)(l) \end{array}$$

By the application of contraposition $(p \Rightarrow q \Leftrightarrow q \Rightarrow p)$, the result is obtained.

It is now necessary to show that

$$\begin{array}{l} l \in incore(p)(sg) \Rightarrow \\ \quad (\exists\, pp_1 : PHYSICALPAGENO \bullet pagetables(p)(sg)(l) = pp) \end{array}$$

This can be done using the invariant. □

Proposition 145. *If a page, p, is locked, IsPageInMainStore[p/...] is satisfied.*

PROOF. Similar to that of the last proof. □

Proposition 146. *A locked page can never be swapped out.*

PROOF. By Propositions 142 and 144. □

Proposition 147. *A free page is in the process table of no process.*

PROOF. Assume that process, p, has segment, s, in which the logical page number, l is mapped to physical page, n. In this case, $n = pagetable(p)(s)(l)$.
By the class invariant,

$$\begin{array}{l} \forall\, ppno : PHYSICALPAGENO \bullet \\ \quad ppno \in freepages \Leftrightarrow \\ \qquad\qquad \neg\, (\exists\, pid : APREF;\ sg : SEGMENT;\ lpn : LOGICALPAGENO \bullet \\ \qquad\qquad\quad ppno = pagetable(pid)(sg)(lpn)) \end{array}$$

By Universal Instantiation:

$$\begin{array}{l} n \in freepages \Leftrightarrow \\ \quad \neg\, (\exists\, pid : APREF;\ sg : SEGMENT;\ lpn : LOGICALPAGENO \bullet \\ \qquad\qquad n = pagetable(pid)(sg)(lpn)) \end{array}$$

From the assumption that $n \in freepages$, a contradiction ensues. Therefore,

$$n \notin freepages$$

as required. □

Proposition 148. *A free page is not incore.*

PROOF. The invariant states that:

$$\begin{array}{l} \forall\, ppno : PHYSICALPAGENO \bullet \\ \quad ppno \in freepages \Leftrightarrow \\ \qquad\qquad \neg\, (\exists\, pid : APREF;\ sg : SEGMENT;\ lpn : LOGICALPAGENO \bullet \\ \qquad\qquad\quad ppno = pagetable(pid)(sg)(lpn)) \end{array}$$

This implies that:

$$freepages \cap pagetable(p)(sg) = \varnothing \qquad (6.1)$$

The class invariant also states that:

$$\begin{array}{l} p \in \mathrm{dom}\, incore \wedge sg \in \mathrm{dom}\, incore(p) \Rightarrow \\ \qquad incore(p)(sg) \subseteq \mathrm{dom}\, pagetable(p)(sg) \end{array}$$

This and equation (6.1) above imply that $pg \in freepages \Leftrightarrow \neg\ incore(p)(sg)(l)$ for all values of p, sg and l. □

Proposition 149. *A free page is not:*

1. *executable;*
2. *readable; or*
3. *writable.*

Proof. Since a free page is not in the page table, it follows from the invariant that it cannot have any of these attributes. □

6.3.1 The Paging Disk Process

The paging disk holds pages while they are not in main store. The virtual store software copies pages to and from the paging disk to implement the swapping process that underlies the huge address space illusion. Pages are swapped out of store when they are not required and swapped back in when their owning process (or one of them, if the page is shared) refers to them.

The paging disk is part of the subsystem whose design will be discussed in the next few subsections. The subsystem's organisation and interactions are shown in Figure 6.2. It consists of an ISR, a handler or driver process and the paging disk proper.

The paging disk is assumed to be infinite in size. It is represented by a mapping from logical to physical pages (*pagemap*). In addition, the disk has an interface to the message-passing system (*msgmgr*) so that it can communicate with the other processes in the storage-management subsystem. For most of the time, *pagemap* will be the focus of attention.

The class defining the paging disk process is now defined.

PagingDiskProcess

$\upharpoonright(INIT, PageIsOnDisk, StorePageOnDisk, RetrievePageFromDisk,$
$\quad RemoveProcessFromPagingDisk, OnPageRequest)$

$pagemap : APREF \twoheadrightarrow\!\!\!\!\!\!+ SEGMENT \twoheadrightarrow\!\!\!\!\!\!+ LOGICALPAGENO \twoheadrightarrow\!\!\!\!\!\!+ PAGE$
$msgmgr : MsgMgr$

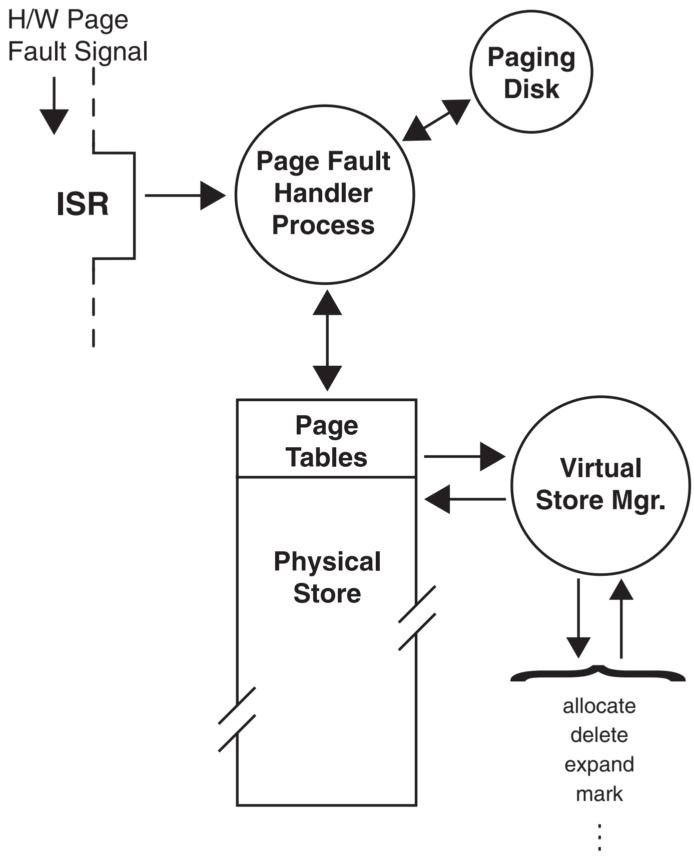

Fig. 6.2. *Interactions between virtual storage components.*

INIT

$msgs? : MsgMgr$

$msgmgr' = msgs?$

$\mathrm{dom}\, pagemap' = \varnothing$

$PageIsOnDisk \mathrel{\widehat=} \ldots$

$StorePageOnDisk \mathrel{\widehat=} \ldots$

$RetrievePageFromDisk \mathrel{\widehat=} \ldots$

$RemoveProcessFromPagingDisk \mathrel{\widehat=} \ldots$

$OnPageRequest \mathrel{\widehat=} \ldots$

The following schema is a predicate that is true when the specified page of the specified process is present on the paging disk.

```
── PageIsOnDisk ──────────────────────────
│ p? : APREF
│ sg? : SEGMENT
│ lpno? : LOGICALPAGENO
├──────────
│ p? ∈ dom pagemap
│ sg? ∈ dom pagemap(p?)
│ lpno? ∈ dom pagemap(p?)(sg?)
└──────────────────────────────────────────
```

Pages are stored on the paging disk when they are swapped out of main store. In order for a page to be stored on the paging disk, it must be placed there and indexed properly. The *StorePageOnDisk* operation does this:

```
── StorePageOnDisk ───────────────────────
│ Δ(pagemap)
│ p? : APREF
│ sg? : SEGMENT
│ lpno? : LOGICALPAGENO
│ pg? : PAGE
├──────────
│ pagemap′ = pagemap ⊕ {p? ↦ {sg? ↦ {lpno? ↦ pg?}}}
└──────────────────────────────────────────
```

Proposition 150. *If a logical page image is already on disk and that page is swapped out again, that page image is overwritten.*

Proof. Let $p = pagemap(p?)(sg?)(lpno?)$ and let $pg?$ be $pagemap'(p?)(sg?)(lpno?)$, so $\{p? \mapsto \{sg? \mapsto \{lpno? \mapsto p\}\}\} \in pagemap$ and $\{p? \mapsto \{sg? \mapsto \{lpno? \mapsto pg?\}\}\} \in pagemap'$. Therefore, $pagemap' = pagemap \oplus \{p? \mapsto \{sg? \mapsto \{lpno? \mapsto pg?\}\}\}$. □

When its owning process references a page that is not in main store, the paging disk is instructed to retrieve it from the paging disk. The following schema models the basics of this operation:

```
── RetrievePageFromDisk ──────────────────
│ p? : APREF
│ sg? : SEGMENT
│ lpno? : LOGICALPAGENO
│ pg! : PAGE
├──────────
│ pg! = pagemap(p?)(sg?)(lpno?)
└──────────────────────────────────────────
```

When a process terminates (or is terminated), all of its pages must be removed from the paging disk. In our model, this can be done very easily by removing the process' identifier from the domain of the *pagemap*. This removes all references to the process from the map—hence, removes the process from the disk, thus:

```
── RemoveProcessFromPagingDisk ──────────────
    $\Delta(pagemap)$
    $p? : APREF$
  ──────────
    $pagemap' = \{p?\} \ndres pagemap$
──────────────────────────────────────────────
```

Proposition 151. *The predicate of RemoveProcessFromPagingDisk implies that $p \notin \operatorname{dom} pagemap'$.*

Proof.

$$\begin{aligned}&\operatorname{dom} pagemap'\\ &\quad = \operatorname{dom}(\{p?\} \ndres pagemap)\\ &\quad = (\operatorname{dom} pagemap) \setminus \{p?\}\end{aligned}$$

Therefore, $p \notin \operatorname{dom} pagemap'$. □

The paging disk has an operation that handles requests for operations. The operations it can perform are:

- store a page (*STOPG*);
- retrieve a page (*GTPG*), and
- delete a process from the paging disk (*DELPROCPG*).

The following schema defines this operation. The schema uses overloaded calls to the *RcvMessage* primitive provided by the message-handling subsystem. The operation is really just a large "or" based on the type of message that has just been received. The infinite loop is modelled by the exterior universal quantifier. While the operation has nothing to do, it waits for the next message to arrive.

```
── OnPageRequest ──────────────────────────────
    $\forall i : 1 \mathrel{..} \infty \bullet$
        $(\exists pdsk, fhandler : APREF;\ msg : MSG \mid$
                    $pdsk = \mathsf{pagedisk} \land fhandler = \mathsf{faultdrvr} \bullet$
            $msgmgr.RcvMessage[pdsk/caller?, fhandler/src?, msg/m!]$
            $\land (\exists p : APREF;\ sg : SEGMENT;$
                        $lpno : LOGICALPAGENO;\ pg : PAGE \bullet$
                $(msg = STOPG\langle\!\langle p, sg, lpno, pg\rangle\!\rangle$
                    $\land storePageOnDisk[p/p?, sg/sg?, lpno/lpno?, pg/pg?])$
```

$$
\begin{array}{|l}
\quad \lor (msg = GTPG\langle\!\langle p, sg, lpno\rangle\!\rangle \\
\qquad \land retrievePageFromDisk[p/p?, sg/sg?, lpno/lpno?, pg/pg!] \\
\qquad \land (\exists rpmsg : MSG \mid rpmsg = ISPG\langle\!\langle p, sg, lpno, pg\rangle\!\rangle \bullet \\
\qquad\quad msgmgr.SendMessage \\
\qquad\qquad [fhandler/dest?, pdsk/src?, rpmsg/m?])) \\
\quad \lor (msg = DELPROCPG\langle\!\langle p\rangle\!\rangle \\
\qquad \land removeProcessFromPagingDisk[p/p?]))) \\
\hline
\end{array}
$$

Proposition 152. *OnPageRequest does what it should.*

Proof. The proof is by cases on message type. It is assumed that all domains are correct so the error cases ignored in the schema need not be introduced here.
Case 1. $m = STOPG$, a request to store a page on disk. The predicate of $storePageOnDisk[p/p?, sg/sg?, lpno/lpno?, pg/pg?]$ states:

$$pagemap' = pagemap \oplus \{p \mapsto \{sg \mapsto \{lpno \mapsto pg\}\}\}$$

so $pg? = pagemap'(p)(sg)(lpno)$.
Case 2. $m = GTPG$, a request to retrieve a page. This follows from the fact that *retrievePageOnDisk* is a function: $pg? = pagemap(p?)(sg?)(lpno?)$.
Case 3. $m = DELPROCPG$. By the last proposition. □

6.3.2 Placement: Demand Paging and LRU

There are significant issues to be resolved in the design of the virtual storage mechanism. This section deals with the general area of *placement.* Placement is concerned with where pages are to be removed and included in main store. When a process refers to a page that is not currently in main store, it generates a *page fault*, an asynchronous signal that interrupts the process and leads to the satisfaction of the reference. To do this, the support software has to identify the page that is being referenced and then identify a page in physical store that can be swapped out to the paging disk, thus making space for the referenced page to be copied into main store. The issue is slightly complicated by the fact that the system might have some free physical pages in main store (indeed, it might be a policy to keep a block of such pages in reserve). For present purposes, it is assumed that *all* physical pages are allocated and that there is no pool of free pages kept in reserve.

The placement algorithm just outlined is *demand paging.* This is the most common approach. It is documented, like many other possible approaches, in most textbooks on operating systems (e.g., [26, 11, 29, 5]). Demand paging is the most commonly used approach and makes reasonable assumptions about the hardware and the software. It just assumes that the hardware can detect a page fault and that the software can find the referenced page and locate the page in main store where it can be stored; if there is no free page, demand paging assumes that there is a way to find a victim page that can be swapped out to make space.

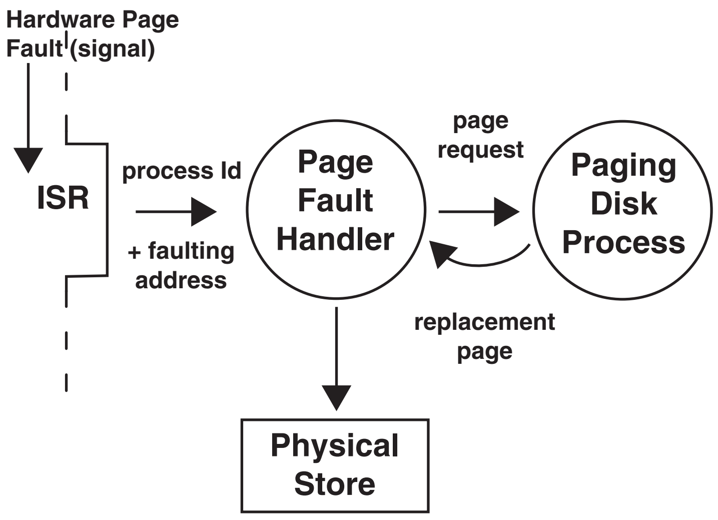

Fig. 6.3. *Process organisation for handling page faults.*

The placement algorithm used in this model has two identifiable aspects:

1. finding pages to remove;
2. pages to include.

The second aspect is solved for us: the pages to include are always those that cause a page fault when referenced by a process.

The general organisation of the processes that handle page faults is shown in Figure 6.3.

6.3.3 On Page Fault

A page fault occurs when a virtual address refers to a page that is not in physical store.

When such a fault occurs, it is assumed that the address causing the fault is available to the operating system. From this address, it is assumed that the following parameters can be computed: segment number, logical page number and the offset into the logical page. In addition, it is assumed that the identity of the process causing the page fault can be determined (it should be the currently running process on a uni-processor; in the terms of this book, the value of *currentp*).

The case in which a page is shared is identical. The important thing to remember is that the owning process causes the page fault. If a page is shared, the important thing is for the page not to be swapped in (or out) more than once. That is why the *smap* is included in *PageTables*. When a shared page is to be swapped, this map is consulted. If the page is shared, it should already be in store. Conversely, if a page is to be swapped in, the *smap* should be consulted for the same reason.

In this section, a page that is to be removed from main store in order to free a page frame will be called a *victim* or *candidate*. It is always a restriction on victim page selection that victim pages are never locked into main store.

To define a relatively simple candidate-finding algorithm, it is necessary to associate page frames in main store with a bit that is set by the hardware whenever a page is referenced and a counter. The counter is a single byte whose value is computed by rotating the reference bit into the top bit of the counter byte and or-ing the counter with the contents of the counter shifted down one bit (ignoring the bottom bit). The types of the reference bit and the counter are:

$$BIT == \{0, 1\}$$
$$N_{256} == 0 \mathrel{..} 255$$

(N_{256} is just the naturals $0 \mathrel{..} 2^{16} - 1$—i.e., a 16-bit unsigned.)

The computation of the counter value forms part of the predicate of schema *ComputeHitCounts*. To define a swap-out procedure, it is necessary to extend *PageFrames* a little so that information on page usage is represented.

The class and its operations are relatively straightforward.

PageFrames

$\upharpoonright(INIT, GetPage, OverwritePhysicalPage, ClearRefBitsAndCounter,$
$\quad ComputeHitCounts, IsVictim, VictimPhysicalPageNo)$

$frames : PAGEFRAME$
$refbit : PHYSICALPAGENO \pinj BIT$
$count : PHYSICALPAGENO \pinj N_{256}$

$\operatorname{dom} count = \operatorname{dom} refbit$
$\operatorname{dom} count \subseteq \operatorname{dom} frames$
$\# \operatorname{dom} refbit = numrealpages$

INIT

$$(\forall i : 1 \mathrel{..} numrealpages \bullet$$
$$\quad frames'(i) = NullPage)$$
$$\operatorname{dom} refbit' = \varnothing$$
$$\operatorname{dom} count' = \varnothing$$

$GetPage \mathrel{\widehat{=}} \ldots$

$OverwritePhysicalPage \mathrel{\widehat{=}} \ldots$

$ClearRefBitsAndCounter \mathrel{\widehat{=}} \ldots$

$ComputeHitCounts \mathrel{\widehat{=}} \ldots$

$IsVictim \mathrel{\widehat{=}} \ldots$

$VictimPhysicalPageNo \mathrel{\widehat{=}} \ldots$

GetPage

$$pageno? : PHYSICALPAGENO$$
$$fr! : PAGE$$

$$1 \leq pageno? \leq numrealpages$$
$$fr! = frames(pageno?)$$

This operation retrieves a page.

The infix function *after* is required by the next schema. Its definition is repeated for convenience:

$$\_after\_ : \operatorname{seq} X \times \mathbb{N} \rightarrow \operatorname{seq} X$$

$$\forall m : \operatorname{seq} X;\ offset : \mathbb{N} \bullet$$
$$\quad \operatorname{dom}(m\ after\ offset) = (1 \mathrel{..} \#m - offset) \land$$
$$\quad (\forall n : \mathbb{N} \bullet$$
$$\qquad (n + offset) \in \operatorname{dom} m \Rightarrow (m\ after\ offset)(n) = m(n + offset))$$

OverwritePhysicalPage

$$\Delta(frames)$$
$$pageno? : PHYSICALPAGENO$$
$$pg? : PAGE$$

$$frames' = frames \oplus \{pageno? \mapsto pg?\}$$

This operation overwrites a page in main store. The input *pageno?* is the index of the page frame in main store and *pg?* is a page full of data.

Proposition 153. *The predicate of the substitution instance of the predicate* $OverwritePhysicalPage[p/pageno?, pg/pg?]$ *replaces the page indexed by p in frames by the page, pg and only that page.*

Proof. The $\oplus$ operation can be defined as:

$$(f \oplus g)(x) = \begin{cases} f(x), x \in \operatorname{dom} f \\ g(x), \text{ otherwise.} \end{cases}$$

Then, for the predicate of the schema:

$$(\mathit{frames} \oplus \{\mathit{pageno}? \mapsto \mathit{pg}?\})(x) = \begin{cases} \mathit{frames}(x), x \in \operatorname{dom} \mathit{frames} \\ \{\mathit{pageno}? \mapsto \mathit{pg}?\}, x = \mathit{pageno}? \end{cases}$$

□

The clearing of the reference bits and reference counter in a physical page is defined as follows:

```
ClearRefBitsAndCounter
  $\Delta(refbit, count)$
  $ppno? : PHYSICALPAGENO$
  ----------
  $refbit' = refbit \oplus \{ppno? \mapsto 0\}$
  $count' = count \oplus \{ppno? \mapsto 0\}$
```

The following operation computes the hit count for each page. That is, it computes the number of times the page has been referenced since it was copied into main store. It must be performed on a cyclic basis but this model does not specify how the cycle is implemented—hardware is the optimal way to compute such counts because the counter must be updated on each reference. The computation operation is defined by the following schema:

```
ComputeHitCounts
  $\Delta(pcount, count)$
  ----------
  $(\forall i : PHYSICALPAGENO \mid i \in \operatorname{dom} frames \bullet$
      $(\exists pcount : N_{265} \bullet$
          $pcount = (count(i)/2)_{\mathrm{mod}\ 256} + refbit(i) * 2^7 \wedge$
          $count' = count \oplus \{i \mapsto pcount\}))$
```

The lowest *count* value is chosen as the victim:

```
IsVictim
  $(\exists j : PHYSICALPAGENO \mid j \in \operatorname{dom} count \bullet$
      $count(j) = min(\operatorname{ran} count))$
```

The physical page of the victim must be obtained:

```
VictimPhysicalPageNo
  $victim! : PHYSICALPAGENO$
  ----------
  $(\exists : PHYSICALPAGENO \mid i \in \operatorname{dom} count \bullet$
      $\wedge\ count(i) = min(\operatorname{ran} count)$
      $\wedge\ i = victim!)$
```

This algorithm is not foolproof but is a reasonable, hardware-independent choice. There are many alternative algorithms in the literature (see, for example, [26] or [29]) but the best will always be determined by the hardware on which the operating system runs. The assumption made here is a minimal one (because many processors implement reference bits in page frames); some machines might provide reference counters directly, while others might record the time of the last reference to each page. It is to be hoped that, in the future, victim determination will be considerably simplified by more co-operative hardware.

The *FindVictim* operation is “safe” in the sense that it will always find an in-core page. The reason for this is that the pageout operation defined above requires that $\neg$ *HaveFreePages* be true before *FindVictim* is called. This ensures that none of the candidate pages is in *freepages*.

$$
\begin{aligned}
PGMSG ::=\ & DELPROCPG\langle\!\langle AREF\rangle\!\rangle \\
\mid\ & GETPG\langle\!\langle AREF \times SEGMENT \times LOGICALPAGENO\rangle\!\rangle \\
\mid\ & STOPG\langle\!\langle AREF \times SEGMENT \times LOGICALPAGENO \times PAGE\rangle\!\rangle \\
\mid\ & ISPG\langle\!\langle AREF \times SEGMENT \times LOGICALPAGENO \times PAGE\rangle\!\rangle
\end{aligned}
$$

$$FMSG ::= BADADDR\langle\!\langle APREF \times SEGMENT \times LOGICALPAGENO\rangle\!\rangle$$

The ISR handling page faults can be defined as the following class. It is a subclass of the message-based ISR defined in the last chapter.

PageFaultISR

$\upharpoonright(INIT,$
$\quad OnPageInterrupt)$

GenericMsgISR

$sched : LowLevelScheduler$
$ptab : ProcessTable$

INIT

$schd? : LowLevelScheduler$
$pt? : ProcessTable$

$sched' = schd? \wedge ptab' = pt?$

OnPageInterrupt

$intaddr? : VIRTUALADDRESS$

$$
\begin{array}{l}
\exists\, fmsg : FMSG;\ cp : APREF;\ sg : SEGMENT; \\
\qquad\qquad lpno : LOGICALPAGENO;\ offset : \mathbb{N} \bullet \\
\quad saddrseq(saddresstrans(intaddr?)) = sg \\
\quad \land\ spageno(saddresstrans(intaddr?)) = lpno \\
\quad \land\ sched.CurrentProcess[cp/cp!] \land sched.MakeUnready[cp/pid?] \\
\quad \land\ ptab.DescrOfProcess[cp/pid?, pd/pd!] \\
\quad \land\ ctxt.SwapOut \land fmsg = BADADDR\langle\!\langle cp, sg, lpno\rangle\!\rangle \\
\quad \land\ (\exists\, isrid, fdvrid : APREF \mid fdrvrd = \mathsf{faultdrvr} \bullet \\
\qquad SendInterruptMsg[fdrvrid/driver?, fmsg/m?]) \\
\quad {}_{9}^{\circ}\, ctxt.SwapIn
\end{array}
$$

When a page fault occurs, the ISR sends a message to the *PageFault-Driver*. The page-fault handler or driver is defined by the following class. As is usual with classes that represent processes, it exports only one operation, here *DoOnPageFault*. The driver also contains routines that access page tables but they are not exported. The idea is that once the driver starts, it has exclusive access to these data structures. The data structures, however, still need to be protected by locks.

PageFaultDriver

$\upharpoonright(INIT, DoOnPageFault)$

$$
\begin{array}{l}
sched : LowLevelScheduler;\ ptab : ProcessTable;\ pts : PageTables; \\
vsm : VStoreManager;\ pfs : PageFrames;\ msgman : MsgMgr \\
lck : Lock
\end{array}
$$

INIT

$$
\begin{array}{l}
mmgr? : MsgMgr;\ sch? : LowLevelScheduler \\
ptb? : ProcessTable;\ pgtabs? : PageTables \\
vstoreman? : VStoreManager \\
pgfrms? : PageFrames;\ lk? : Lock
\end{array}
$$

$$
\begin{array}{l}
msgman' = mmgr? \land sched' = sch? \land ptab' = ptb? \land pts' = pgtabs? \\
vsm' = VStoreManager \land pfs' = PageFrames \land lck' = lk?
\end{array}
$$

$$
\begin{array}{|l}
\mathit{findVictimLogicalPage} \mathrel{\widehat{=}} \ldots \\
\mathit{haveVictim} \mathrel{\widehat{=}} \ldots \\
\mathit{findVictimPage} \mathrel{\widehat{=}} \ldots \\
\mathit{swapPageToDisk} \mathrel{\widehat{=}} \ldots \\
\mathit{retrievePageFromDisk} \mathrel{\widehat{=}} \ldots \\
\mathit{storePageOnDisk} \mathrel{\widehat{=}} \ldots \\
\mathit{genOnPageFault} \mathrel{\widehat{=}} \ldots \\
\mathit{onPageFault} \mathrel{\widehat{=}} \ldots \\
\mathit{DoOnPageFault} \mathrel{\widehat{=}} \ldots \\
\hline
\end{array}
$$

The driver has to find a victim page to swap out when there is no free store. The following schema defines this operation. The page to be swapped out is a *logical* one at this point. The schema maps the logical page to a physical page by another operation.

$$
\begin{array}{|l}
\hline
\mathit{findVictimLogicalPage} \\
p! : \mathit{APREF} \\
sg! : \mathit{SEGMENT} \\
lpno! : \mathit{LOGICALPAGENO} \\
victim! : \mathit{PHYSICALPAGENO} \\
\hline
(\exists\, p : \mathit{APREF};\ s : \mathit{SEGMENT};\ l : \mathit{LOGICALPAGENO} \mid \\
\qquad\qquad p \in \operatorname{dom} \mathit{pagetable} \land sg \in \operatorname{dom} \mathit{pagetable}(p) \\
\qquad\qquad\qquad \land\ l \in \operatorname{dom} \mathit{pagetable}(p)(sg) \bullet \\
\qquad \mathit{pagetable}(p)(sg)(l) = victim! \\
\qquad \land\ l \notin \mathit{lockedpages}(p)(sg) \\
\qquad \land\ p! = p \land sg! = s \land lpno! = l) \\
\hline
\end{array}
$$

The schema simplifies to:

$$
\begin{array}{|l}
\hline
p! : \mathit{APREF} \\
sg! : \mathit{SEGMENT} \\
lpno! : \mathit{LOGICALPAGENO} \\
victim! : \mathit{PHYSICALPAGENO} \\
\hline
p! \in \operatorname{dom} \mathit{pagetable} \land sg! \in \operatorname{dom} \mathit{pagetable}(p!) \\
\land\ lpno! \in \operatorname{dom} \mathit{pagetable}(p!)(sg!) \\
\land\ \mathit{pagetable}(p!)(sg!)(lpno!) = victim! \\
\land\ lpno! =\notin \mathit{lockedpages}(p!)(sg!) \\
\hline
\end{array}
$$

The following definition is a synonym. It tests the reference count of the victim physical page.

$haveVictim \mathrel{\widehat{=}} pfs.IsVictim$

The final operation to locate a victim page is the following:

$$\begin{aligned}&findVictimPage \mathrel{\widehat{=}}\\ &\quad pfs.VictimPhysicalPageNo[victim/victim!]\\ &\qquad \wedge findVictimLogicalPage[victim/victim!]\end{aligned}$$

It is possible to put a few properties of pages on a more formal basis.

Proposition 154. *Locked pages can never be victims.*

Proof. The predicate of *findVictimLogicalPage* contains the conjunct $l \notin lockedpages(p)(sg)$, where l is the logical page number, p the process identifier and sg the segment. □

Proposition 155. *Free pages can never be victims.*

Proof. Since free pages do not belong to any process (by Proposition 147), they do not appear in *pagetable*, so they cannot be victims because the quantifier in *findVictimLogicalPage* ranges over *pagetable*. □

Proposition 156. *Faulting processes are not ready.*

Proof. The predicate of *PageFaultISR.OnPageInterrupt* contains a reference to the schema $MakeUnready[cp/pid?]$, where cp is the current process (i.e., $cp = currentp$), so it is the process that caused the page fault. The action of *MakeUnready* is to remove the process from the ready queue. □

Proposition 157. *A faulting process cannot be executed.*

Proof. By the previous proposition, *MakeUnready* implies that *cp* cannot be scheduled until it is returned to the ready queue. Furthermore, *cp* cannot continue because *OnPageInterrupt* calls *sched.ScheduleNext* to select the next process to run; this process will not be *cp*. □

Corollary 12. *A faulting process is blocked.*

Proof. This is a consequence of the previous proposition. □

The operation that actually swaps a page from main store to the paging disk is the following:

$$
\begin{array}{l}
\mathit{swapPageToDisk} \mathrel{\widehat=} \\
\quad (\mathit{findVictimPage}[p/p!, sg/sg!, lpno/lpno!] \\
\qquad \land \mathit{vsm.MarkSharedLogicalPageAsOut}[p/p?, sg/sg?, lpno/lpno?] \\
\qquad \land \mathit{pfs.GetPage}[pg/fr!, victim!/pageno?] \\
\qquad \land \mathit{storePageOnDisk}[p/p?, sg/sg?, lpno/lpno?, pg/pg?] \\
\qquad \land \mathit{pfs.ClearRefBitsAndCounter}[victim!/ppno?] \\
\qquad \land \mathit{storePageOnDisk}[p/p?, sg/sg?, lpno/lpno?, pg/pg?]) \\
\qquad\qquad\qquad \setminus \{pg, p, sg, lpno\} \\
\qquad \lor \mathit{Skip}
\end{array}
$$

First, a victim is located and then marked as being not in store (*MarkShared-LogicalPageAsOut*). The page causing the page fault is then demanded from the paging disk and the victim is sent to the disk for temporary storage. The reference bits of the victim are then cleared so that the referenced page (the one causing the page fault) can be written to it by *storePageOnDisk*.

The next operation to be defined is *retrievePageFromDisk*. This operation, as its name suggests, communicates with the paging disk process to locate the page that is to be brought into main store as a consequence of the last page fault. The reference to this page is, in fact, the one that caused the page fault that resulted in the current execution of the *PageFaultDriver*. It is defined by the following schema:

retrievePageFromDisk

$$
\begin{array}{l}
p? : \mathit{APREF} \\
sg? : \mathit{SEGMENT} \\
lpno? : \mathit{LOGICALPAGENO} \\
pg! : \mathit{PAGE} \\
\hline
(\exists\, msg_o, msg_r : \mathit{PGMSG};\ src, dest : \mathit{APREF};\ pg : \mathit{PAGE} \mid \\
\qquad\qquad msg_o = \mathit{GETPG}\langle\langle p?, sg?, lpno? \rangle\rangle \\
\qquad\qquad\quad \land src = \mathsf{faultdrvr} \land dest = \mathsf{pagedsk} \bullet \\
\quad \mathit{msgman.SendMsg}[src/src?, dest/dest?, msg_o/m?] \mathbin{⨾} \\
\quad (\mathit{msgman.RcvMsg}[src/caller?, dest/src?, msg_r/m!] \\
\qquad \land msg_r = \mathit{ISPG}\langle\langle p?, sg?, lpno?, pg! \rangle\rangle))
\end{array}
$$

As can be seen, this operation mostly handles messages. First, the operation sends a message requesting that the paging disk retrieve the page specified by the parameters $p?$, $sg?$ and $lpno?$; the page is represented by $pg!$. The page, $pg!$, is returned by the paging disk in the *ISPG* message.

The operation to store a page on disk is similar to the last one.

storePageOnDisk

$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$
$pg? : PAGE$

$$(\exists\, msg : PGMSG;\ src, dest : APREF \mid$$
$$msg = STOPG\langle\!\langle p?, sg?, lpno?, pg? \rangle\!\rangle$$
$$\wedge\ src = \mathsf{faultdrvr} \wedge dest = \mathsf{pagedsk} \bullet$$
$$mgman.SendMsg[src/src?, dest/dest?, msg/m?])$$

Again, this operation mostly deals with messages. In this case, it contains just one send operation. Messages of type *STOPG* request the paging disk to store the page specified by $p?$, $sg?$, $lpno?$; the page is denoted by $pg?$.

The next operation schema defines the operation performed whenever a page fault occurs. This is a complex operation and its definition reflects this complexity. The operation receives a message containing a specification of the page that was referenced and determined (by the hardware) not to be in main store (it is on the paging disk).

The operation determines whether there are any page frames free in main store. If there are, the page is located on the disk and copied into a free page. To do this, the operation sends a message to the paging-disk process requesting the page.

If there are no page frames free in main store, a page that is currently resident in main store must be placed on the paging disk in order to make space for the one that caused the page fault. A suitable resident page must have a low frequency of reference in the near future and the reference-count mechanism specified above is used to determine which it is. This page is referred to as the *victim* in the following schema and the schemata defining the selection operations. It should be noted that the victim can belong to any process whatsoever but cannot be a page that is locked into main store. The victim is swapped to the paging disk and the one causing the page fault is retrieved and copied into the newly vacated page frame in main store. The process that caused the page fault is then returned to the ready queue and the ISR waits for the next page fault.

It should be noted that the victim can never be a locked page (i.e., a page locked into main store). This condition is imposed so that, in particular, pages locked into main store by the kernel (typically pages containing kernel code and data structures) cannot be swapped out. It is undesirable to swap kernel pages out of store because they might be involved in the operation currently being performed or they might be ISRs and the scheduler.

In some kernel designs, it is possible for kernel processes to be stored in swappable pages. In such a case, the pages would contain data or processes that are considered to be of lower importance, implementing operations that the kernel can afford to have blocked while some of their store is paged out.

The kernel assumed here is the one modelled in Chapter 4 and extended in Chapter 5. It is assumed that *all* of its components (data structures and processes) must be stored in pages that are locked into main store (and hence are stored in pages with the *locked* attribute). (The schemata defined above for victim selection, it should be noted, depend on the commutativity of conjunction.)

$$
\begin{array}{l}
\mathit{genOnPageFault} \mathrel{\widehat{=}} \\
\quad \exists\, \mathit{fmsg} : \mathit{FMSG};\ \mathit{me}, \mathit{src} : \mathit{APREF} \mid \mathit{me} = \mathsf{faultdrvr} \land \mathit{src} = \mathsf{hardware} \bullet \\
\qquad \mathit{msgman.RcvMessage}[\mathit{me}/\mathit{dest}?, \mathit{src}/\mathit{src}?, \mathit{fmsg}/\mathit{m}!] \\
\qquad \land (\exists\, \mathit{fpid} : \mathit{APREF};\ \mathit{fs} : \mathit{SEGMENT}; \\
\qquad\qquad\qquad\quad \mathit{flpno} : \mathit{LOGICALPAGENO}; \\
\qquad\qquad\qquad\quad \mathit{destpg} : \mathit{PHYSICALPAGENO} \bullet \\
\qquad\quad \mathit{fmsg} = \mathit{BADADDR}\langle\!\langle \mathit{fpid}, \mathit{fs}, \mathit{flpno} \rangle\!\rangle \\
\qquad\quad ((\mathit{lck.Lock} \fatsemi \\
\qquad\qquad ((\mathit{pts.HaveFreePages} \land \mathit{pts.AllocateFreePage}[\mathit{destpg}/\mathit{ppno}!]) \\
\qquad\qquad\quad \lor (\mathit{haveVictim} \\
\qquad\qquad\qquad \land \mathit{swapPageToDisk}[\mathit{destpg}/\mathit{victim}!])) \land \mathit{lck.Unlock}) \fatsemi \\
\qquad\quad (\mathit{lck.Lock} \fatsemi \\
\qquad\qquad (\mathit{retrievePageFromDisk}[\mathit{fpid}/\mathit{p}?, \mathit{fs}/\mathit{sg}?, \mathit{flpno}/\mathit{lpno}?, \mathit{page}/\mathit{pg}!] \\
\qquad\qquad\quad \land \mathit{pfs.ClearRefBitsAndCounter}[\mathit{destpg}/\mathit{ppno}?] \\
\qquad\qquad\quad \land \mathit{pfs.OverwritePhysicalPage}[\mathit{page}/\mathit{pg}?, \mathit{destpg}/\mathit{pageno}?] \\
\qquad\qquad\quad \land \mathit{vsm.MarkSharedLogicalPageAsIn} \\
\qquad\qquad\qquad\qquad [\mathit{fpid}/\mathit{p}?, \mathit{fs}/\mathit{sg}?, \mathit{flpno}/\mathit{lpno}?] \\
\qquad\qquad\quad \land \mathit{lck.Unlock}))) \\
\qquad\quad \fatsemi (\mathit{lck.Lock} \land \mathit{sched.MakeReady}[\mathit{fpid}/\mathit{pid}?]) \\
\qquad\quad \fatsemi (\mathit{sched.ScheduleNext} \land \mathit{lck.Unlock}))
\end{array}
$$

$$
\mathit{onPageFault} \mathrel{\widehat{=}} (\mathit{sched.CurrentProcess}[\mathit{p}/\mathit{cp}!] \land \mathit{genOnPageFault}[\mathit{p}/\mathit{p}?]) \setminus \{p\}
$$

$$
\begin{array}{l}
\mathit{DoOnPageFault} \mathrel{\widehat{=}} \\
\quad (\forall\, i : 1 \mathinner{\ldotp\ldotp} \infty \bullet \mathit{onPageFault})
\end{array}
$$

Proposition 158. *If there are free pages, no page is swapped out.*

Proof. The first component of the sequential composition in the predicate of *genOnPageFault* contains the disjunction:

$$
\begin{array}{l}
(\mathit{pts.HaveFreePages} \land \mathit{pts.AllocateFreePage}[\mathit{destpg}/\mathit{ppno}!]) \\
\quad \lor (\mathit{haveVictim} \land \mathit{swapPageToDisk}[\mathit{destpg}/\mathit{victim}!])
\end{array}
$$

If there are free pages, *pts.HaveFreePages* is satisfied. □

Proposition 159. *If there are no free pages in main store, a page is swapped out.*

Proof. There is a disjunction in the composition forming *genOnPageFault*'s predicate:

$$(pts.HaveFreePages \land pts.AllocateFreePage[destpg/ppno!]) \\ \quad \lor (haveVictim \land swapPageToDisk[destpg/victim!])$$

In this case, if *HaveFreePages* is not satisfied, $\neg$ *HaveFreePages* must be satisfied and the swap is performed by *swapPageToDisk*. □

Proposition 160. *If a process fault occurs, the referenced page overwrites a page frame freed by swapping out.*

Proof. In the previous proposition, the page *destpg* was swapped out to disk. In the second component of *genOnPageFault*'s predicate, the operation *overWritePhysicalPage* occurs as a conjunct. By $p \land q \vdash p$, the result follows. □

Proposition 161. *If a process faults, the referenced page is brought into store.*

Proof. The operation *retrievePageFromDisk*[*page*/*pg*] is a conjunct in the second component of the sequential composition in *genOnPageFault*. □

Proposition 162. *If there are free pages, only a free page is written when a page is swapped in.*

Proof. The predicate of schema *genOnPageFault* contains the following references:

$$pts.AllocateFreePage[destpg/ppno!] \ldots \fatsemi \ldots \\ \quad retrievePageFromDisk[\ldots, page/pg!] \\ \quad pts.OverwritePhysicalPage[page/pg?, destpg/pageno?]$$

The physical page, *destpg*, is the one overwritten by *page* in the operation defined by the schema *OverwritePhysicalPage*. The physical page *destpg* is allocated by *AllocateFreePage*. The page retrieved from disk is *page*; it is retrieved by *retrievePageFromDisk* (the omitted arguments refer to the faulting process). □

Proposition 163. *Newly swapped pages have a zero reference count.*

Proof. Reference bits are cleared in the newly swapped page by *genOnPageFault*. The second component of this schema's predicate contains, as one of its conjuncts, a reference to *pfs.ClearRefBitsAndCounter*[*destpg*/*ppno*?]—this is a reference to a page frame in physical store, not to the contents (which is denoted by *page* in this predicate). □

Proposition 164. *Newly swapped pages cannot be victims unless all pages are victims.*

Proof. By the victim-finding operation, the victim has the minimum reference count:

IsVictim

$$(\exists j : PHYSICALPAGENO \mid j \in \mathrm{dom}\, count \bullet \\ count(j) = min(\mathrm{ran}\, count))$$

A newly swapped-in page—one that has not yet been referenced—has a reference count of 0. To become a victim, there must be no page with reference count > 0. □

A process can be waiting on a device when one of its pages is chosen to swap out. If the driver copies data and puts it into buffers associated with the waiting process' PCBs, there is only the issue of swapping out the page. However, there is no way *a priori* of knowing whether the page just swapped out will cause a page fault when its owning process next executes. Since swapping out does not affect the operations of the device upon which the victim is waiting, it would appear valid just to pick any process that is not locked into main store.

The reader should note that, logically, this is perfectly adequate. As a proposed implementation, this is unlikely to work well. It is to be expected that page faults will be relatively frequent. The paging disk has a latency time that must be taken into account. When a user process causes a page fault, it must be blocked. Clearly, processes ought to be blocked for the shortest possible time. The paging disk, however, serialises requests. All of this suggests that the swapin/swapout operations should be as fast as possible.

Moreover, the specification, as it stands, allows the ISR to respond to as many interrupts as it can but it must also wait for the driver. The driver's message input is restricted to one immediate and one outstanding message. This suggests that the ISR should enqueue messages on the driver and immediately halt.

Furthermore, the context of the faulting process *must* be swapped immediately. This is because it cannot progress and *must* be taken off the processor: alternatives are hard to discern. Removal of the current process' registers from the processor by the ISR is, therefore, justified.

Similarly, the page disk can lose requests if it just hangs between instructing a disk search and reading the result.

This subsystem also shows limitations with the message-passing régime. Because a synchronous method has been adopted, the sender must wait if the receiver is not in a state to receive. This has the implication that, should the page disk process not be ready to accept another request (either because it is waiting for the disk or because it is processing another request), the page-fault handler will have to wait. This has the implication that the page-fault handler might miss an interrupt.

The presence of operations to add and remove process data from the paging disk complicates matters also. Luckily, these requests will be less common than simple page faults.

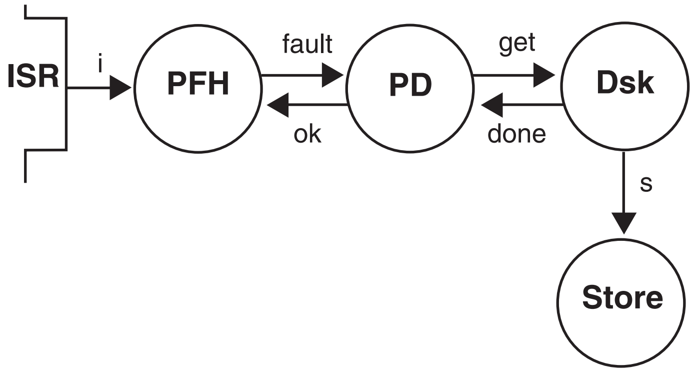

Fig. 6.4. *The actual specification.*

The question for us is the following. Even though the classes and operations presented above provide an adequate logical model, they do not take into consideration all of the pragmatic issues. Should the specification be altered to reflect these pragmatic issues?

The specification presented above can be represented diagrammatically as in Figure 6.4. As can be seen, the virtual-storage management subsystem consists of the ISR (denoted by *ISR* in the figure), the page-fault driver process (denoted by *PFH* in the figure) and the disk driver process (denoted by *PD*). The figure also includes the disk itself (denoted by *DSK*) and main store (denoted by *STORE*).

The arrows in Figure 6.4 denote the messages or interactions between processes. The arrow labelled *i* denotes the message sent by the *ISR* to the page-fault handling process (*PFH*). This message contains the specification of the page that was referenced (in terms of the segment and logical page numbers) and the process (its *APREF*). The page-fault handler sends a *fault* message containing the same information to the paging-disk process, which in turn sends a request (as a *get* message) to the disk proper (actually, to the disk driver). The disk driver retrieves the page and sends a *done* message to the paging-disk handler. The *done* message denotes the fact that the retrieval has been successfully completed.

Once the page has been written to main store, process *PD* sends an *ok* message to the page-fault handler process, *PFH*. On reception of the *ok*, the page-fault handler can wait for another page fault.

It is clearly a highly desirable property for any optimisations of the basic (logical) specification to behave in the same way as the specification. It is also highly desirable, given the present context, to be able to demonstrate this in a formal way. In order to achieve this, the proposed optimisations are formalised as CCS [21] processes so that they can be manipulated in formally sound ways. CCS is chosen as the representation because we are interested in the interactions between the component processes of the subsystem, not in the specification of the components. The processes to be modelled do not have properties suggestive of the use of the π-calculus (e.g., mobility), so CCS appears sufficient.

The subsystem can be represented in CCS as the following set of equations:

$$\begin{aligned}
&\mathit{ISR} = \bar{i}.\mathit{ISR}\\
&\mathit{PFH} = i.\overline{\mathit{fault}}.\mathit{ok}.\mathit{PFH}\\
&\mathit{PD} = \mathit{fault}.\overline{\mathit{get}}.\mathit{done}.\bar{s}.\overline{\mathit{ok}}.\mathit{PD}\\
&\mathit{DSK} = \mathit{get}.\overline{\mathit{done}}.\mathit{DSK}\\
&\mathit{STORE} = s.\mathit{STORE}
\end{aligned}$$

The overall arrangement is represented in CCS as:

$$\mathit{VM}_1 = (\mathit{ISR} \mid \mathit{PFH} \mid \mathit{PD} \mid \mathit{DSK} \mid \mathit{STORE}) \setminus \{i, \mathit{fault}, \mathit{ok}, \mathit{get}, \mathit{done}, s\}$$

(Note that actions are hidden using the \ operation.)

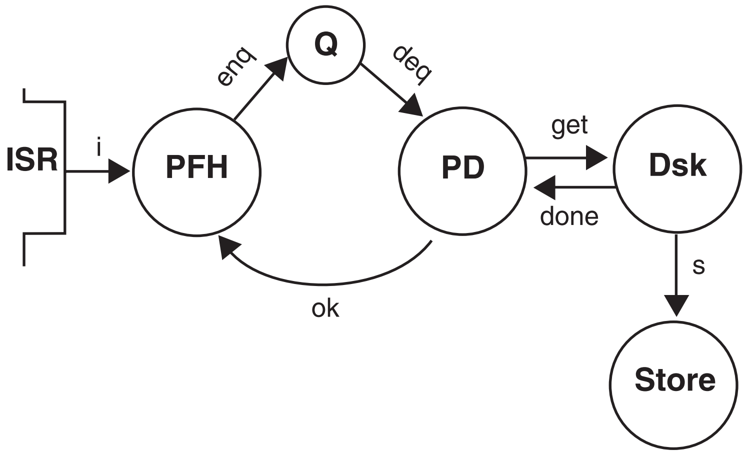

Fig. 6.5. *The specification using a queue.*

As noted above, the design depicted in Figure 6.4 and represented by the above set of CCS process definitions can be optimised. An obvious optimisation is to introduce a queue of requests between the page-fault handler (PFH) and paging-disk process (PD). This is the arrangement shown in Figure 6.5. In the arrangement shown in Figure 6.5, the PFH process places new requests into the queue. The queue is represented by the process named Q in the figure, and the operation of sending a request (really, just an enqueuing of the request) is represented by the *enq* arrow. Requests are removed from the queue process by the paging-disk process, PD, in exactly the same way they are in the first case. When the page has been copied to main store, the paging-disk process, PD sends the page-fault handling process an *ok* to inform it that: (i) the copy has been performed and that (ii) the process that caused the fault just rectified can now be unblocked.

The argument is that this second version can process more page faults per unit time. In this case, PFH does not now wait for the page fault to be rectified before it can wait for a new fault. Instead, it passes the request to the rest of the subsystem and then immediately blocks on the page-fault ISR.

This second arrangement can be represented by CCS processes as follows:

$$\begin{aligned} VM_2 = init.(ISR \mid PFH_2 \mid Q \mid PD_2 \mid DSK \mid STORE) \\ \backslash\{i, enq, deq, ok, done, get, init, s\} \end{aligned}$$

For the definition of this subsystem, processes ISR, DSK and $STORE$ remain as in the first case. The remaining processes must be redefined as follows (the subscripts will be explained below).

$$\begin{aligned} &PFH_2 = i.\overline{enq}.ok.PFH_2 \\ &Q = init.enq.Q_1 \\ &Q_1 = enq.Q1 + deq.Q1 \\ &PD_2 = deq.\overline{get}.done.\bar{s}.\overline{ok}.PD_2 \end{aligned}$$

It is clearly necessary to distinguish between the two versions of PFH that have been defined at this point (a third version will be added shortly). For this reason, subscripts were introduced into the specifications.

It would be useful for these two specifications (models) to be equivalent in some sense. One important sense is that they should be *observationally* equivalent; another is that they should be *bisimilar*.

The property of *observational* equivalence of two processes is very much the intuitive one: two processes are observationally equivalent when they cannot be distinguished by an external observer. In other words, the externally visible events that can be perceived by an external observer are determined by the observer to be the same in content and in order, no matter which process is observed. In the cases of VM_1 and VM_2, the externally observable events are restricted by the hiding operator (\, as in Z).

Bisimilarity is an equivalence that also takes hidden actions into account. (Those readers unfamiliar with the concept should consult [21], Chapter 4, for an extended treatment.)

It is now possible to engage in formal reasoning about processes VM_1 and VM_2 and to prove two important propositions about them. Rather than engaging in a hand proof, it was considered interesting to employ automation. To this end, the *Concurrency Workbench of the New Century* [8] was employed as a tool. The Concurrency Workbench can be used to determine a number of properties of CCS and CSP processes, including observational equivalence and bisimilarity. The propositions concerning equivalence of the various versions of the subsystem were all proved using the Concurrency Workbench.

Proposition 165. *Processes VM_1 and VM_2 are observationally equivalent.*

PROOF. Both VM_1 and VM_2 were encoded in the input format required by the CWB and tested using the `eq -S obseq` command. The CWB system determined that VM_1 and VM_2 are observationally equivalent. □

Matters are different when it comes to bisimilarity. Processes VM_1 and VM_2 are quite dissimilar in internal structure, and process VM_2 offers an initial event, *init*, which initialises the process Q, so it does not appear, at first sight, that VM_1 and VM_2 will be bisimilar. Indeed, this is the case.

Proposition 166. *Processes VM_1 and VM_2 are not bisimilar.*

PROOF. Both VM_1 and VM_2 were encoded in the input format required by the CWB and tested using the `eq -S bisim` command. The CWB system determined that VM_1 and VM_2 are *not* bisimilar. □

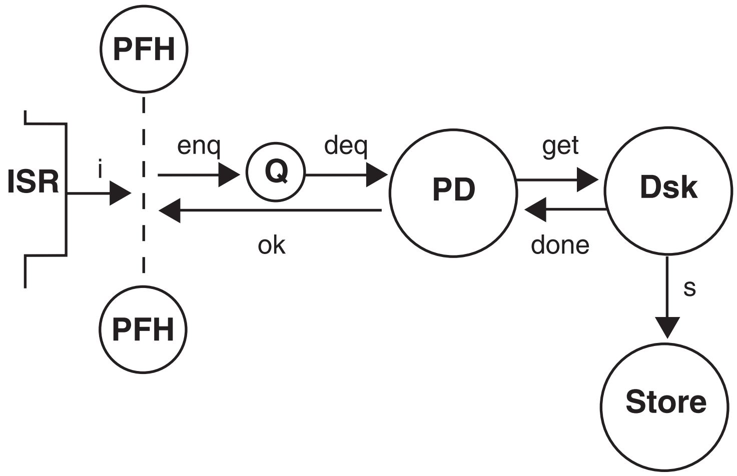

Fig. 6.6. *The ideal specification.*

Finally, a second specification can be considered as an optimisation. This version retains the queue of requests. It differs from the second version by creating a new instance of the *PFH* process whenever an interrupt occurs. This instance performs the same operations as in the second version but immediately terminates (denoted by the **0** process). The CCS specification is (again, subscripts are used to differentiate between versions):

$$VM_3 = init.(ISR \mid PFH_3 \mid Q \mid PD_2 \mid DSK \mid STORE) \\ \backslash\{i, enq, deq, ok, done, get, init, s\}$$

where:

$$PFH_3 = i.(\overline{enq}.ok.\mathbf{0} \mid PFH_3)$$

Proposition 167. *VM_3 is observationally equivalent to VM_2.*

Proof. The models for VM_2 and VM_3 were encoded in the input format required by the CWB and tested using the `eq -S obseq` command. The CWB system determined that the two are observationally equivalent. □

Proposition 168. *VM_3 is bisimilar to VM_2.*

Proof. The models for VM_2 and VM_3 were encoded in the input format required by the CWB and tested using the `eq -S obseq` command. The CWB system determined that the two are bisimilar. □

From the first of these two results, we have the next proposition:

Proposition 169. *Process VM_1 is observationally equivalent to VM_3.*

Proof. By transitivity of the equivalence relation.

(In addition, the encoded models were tested using the `eq -S obseq` command and the result was supported by the CWB system.) □

We also have the following negative result:

Proposition 170. *Processes VM_1 and VM_3 are not bisimilar.*

Proof. Again, by transitivity of the equivalence relation, this result would be expected.

(In addition, the CWB supported the claim.) □

The last result is not as bad as it sounds. All that is required, here, is that the processes *appear* the same as far as an external observer is concerned. This property is sufficient for the optimisations to be considered equivalent to the original. However, as far as this chapter is concerned, the first version (the one referred to as the "logical" one) is the one that will be adopted.

With this discussion of optimisation out of the way, it is possible to return to the main theme. There follow some propositions dealing with the properties of the logical (Z) model of page faults.

Proposition 171. *Page faults are serviced in the order in which they occur.*

Proof. There are two senses:

1. Each page fault is an interrupt. In this sense, page faults are all processed in order.
2. In this sense, page faults are serviced by the handler process which services messages sent to it by the page-fault ISR. These messages are sent in temporal (sequence) order. That the handler process operates upon these messages in the correct order is a consequence of the correctness of the message-passing operations.

Each sense implies the statement of the proposition. □

Proposition 172. *Every page fault is serviced eventually (i.e., the page-fault mechanism is fair).*

Proof. This proof follows from the correctness and fairness of the message-passing operations. □

Proposition 173. *The process causing a page fault has status* pstrunning *when the fault occurs.*

Proof. Only a running process can cause a page fault.

ISRs cannot cause faults and are not processes.

Drivers are processes. They are locked into main store and, by hypothesis, have all the necessary pages locked in as well.

Only user processes have pages that can be swapped into and out of store.

The fact that a fault occurs implies that the process causing it is executing. There can only be one process that is executing at any one time (i.e., is the current value of *currentp* and has a status of pstrunnning).

This is a property of the scheduler. □

Proposition 174. *A faulting process is not unblocked until the replacement page has been swapped into physical store.*

Proof. The service operation *onPageFault* first blocks the faulting process by setting its state to pstwaiting and by removing it from the processor. Therefore *blockFaultingProcess* implies that $currentp \neq currentp' \wedge regs' \neq regs$, where *regs* are hardware registers.

The *onPageFault* operation takes the form of a sequential composition. There are three operations, thus forming a composition of the form $S_1 \fatsemi S_2 \fatsemi S_3$. The second operation is itself a sequential composition; it is this part that performs the swapping operation. The operations are, therefore, totally ordered and the order can be directly related to temporal succession.

Let p denote the faulting process and let the composition be written as above. Then post $S_1 \Rightarrow status(p) = \mathsf{pstwaiting}$ and $currentp' \neq p$.

Throughout S_2, $status(p) = \mathsf{pstwaiting}$.

Finally, pre $S_3 \Rightarrow status(p) = \mathsf{pstwaiting}$, while post $S_3 \Rightarrow status(p) = \mathsf{pstready}$ but $p \neq currentp$—the last being a property of *MakeReady*. It cannot be true because the service process is currently executing. □

Proposition 175. *If a process, p, causes a page fault, that process is not marked ready until the faulting page has been replaced.*

PROOF. This follows immediately from the previous proposition. □

Proposition 176. *After execution of onPageFault, there is exactly one physical page in physical store such that its logical page number in the faulting process maps to the physical page number.*

PROOF. The proposition implies that exactly one page is mapped into store by the page-fault mechanism.

Inspection of the predicate of *genOnPageFault* shows that only one page is introduced into store. □

Proposition 177. *If a process, p, causes a page fault, after that page fault has been serviced, the process is in the ready state.*

PROOF. That the process is blocked follows immediately from Proposition 174. The predicate of *genOnPageFault* contains an instance of *MakeReady* with the faulting process' identifier substituted for the formal parameter. □

Proposition 178. *If a page fault occurs and a page has to be swapped out, that page can be a page belonging to the faulting process or to another process.*

PROOF. There are two cases to consider.
Case 1: If there are free pages, they are consumed. Only the faulting process is affected.
Case 2: *HaveVictim* is called. The victim page is found in physical store by *pfs.IsVictim*:

$$
\begin{aligned}
&findVictimPage \mathrel{\widehat=} \\
&\qquad pfs.VictimPhysicalPageNo \\
&\qquad \wedge\ findVictimLogicalPage
\end{aligned}
$$

The value returned by *pfs.VictimPhysicalPageNo* is just a physical page number with the minimum reference count. The schema does not mention processes. Similarly, *findVictimLogicalPage* in this class merely looks up the owning process' identifier and the segment in which the page occurs, as well as the logical page number of the victim page. Therefore, the page can belong to *any* process in the system except those that lock it into main store (i.e., driver and system processes). □

Proposition 179. *If a page fault occurs and there are free pages in physical store, physical store is updated; only one process is affected.*

Proof. By the definition of *onPageFault*:

$pts.HaveFreePages \wedge pts.AllocateFreePages$

implies that the physical page to which the swapped-in page is to be written is a free page in physical store.

In this case, only the faulting process is affected. This is because the allocation of a free page relates only to a single process (the process to which the allocation is made). □

Corollary 13. *If a page fault occurs within process, p, and if a page has to be swapped out, only a maximum of two processes are affected by the page-swapping operation.*

Proof. Immediate from the two preceding propositions. □

Proposition 180. *No pages are swapped unless a page fault occurs.*

Proof. Inspection reveals that only the schema *genOnPageFault* references the relevant operations. □

6.3.4 Extending Process Storage

Processes very often make requests to increase the amount of store they use. In a paged system, this allocates more pages to the process. This subsection contains a model of the operations. It is rarer for processes to release store but operations to perform this task are defined as well. Pages can also be shared between processes. Sharing can be used, for example, to implement message passing in virtual storage.

First, there is a problem with allocating pages. The problem is that there might not be a physical page free when the request is made. The system could block the requesting process until there is a free page in the page frame. This solution does not fit well with the aims of virtual storage. Furthermore, when a new process is created, it will request a set of pages to hold code, data, stack, etc. It would not be acceptable to block the creation of the process until main store becomes free.

A solution more in keeping with the purpose of virtual store is the following. When a request is made to allocate a new page and there is no free physical store, an empty page is allocated on disk. The new page is allocated to the requesting process and can be written to by the requesting process.

This mechanism can be used when allocating extra pages to a new page.

Therefore, it is necessary to begin with the definition of the empty page:

$$\begin{array}{|l}
pageofzeroes : PAGE \\
\hline
\forall pg : PAGE \bullet \\
\quad pageofzeroes = \lambda\, i : 1 \mathbin{..} framesize \bullet 0
\end{array}$$

This structure need not be held in a page; it can be produced in a loop and then set into a buffer of smaller size. Ideally, it should be packed into a single disk block and then handed to the paging disk. This needs to be iterated n times, where n is the number of disk blocks per page.

Pages must be specified when allocating, deallocating and sharing. A method for specifying them is required. Here, we define a structure for this purpose:

$$\begin{array}{|l}
PAGESPEC == APREF \times SEGMENT \times LOGICALPAGENO
\end{array}$$

The following axiomatic definitions are required to manipulate objects of type *PAGESPEC*. The relation that comes first will be discussed below. The rest of the definitions are: the constructor function (*mkpgspec*) and the accessor functions (*pgspecpref* to obtain the process reference, *pgspecseg* to obtain the segment name and *pgspeclpno* to obtain the logical page number).

$$\begin{array}{|l}
smap : PAGESPEC \leftrightarrow PAGESPEC \\
mkpgspec : APREF \times SEGMENT \times LOGICALPAGENO \rightarrow PAGESPEC \\
pgspecpref : PAGESPEC \rightarrow APREF \\
pgspecseg : PAGESPEC \rightarrow SEGMENT \\
pgspeclpno : PAGESPEC \rightarrow LOGICALPAGENO \\
\hline
\forall p : APREF;\ sg : SEGMENT;\ lpno : LOGICALPAGENO \bullet \\
\quad mkpgspec(p, sg, lpno) = (p, sg, lpno) \\
\\
\forall ps : PAGESPEC \bullet \\
\quad pgspecpref(ps) = fst\ ps \\
\quad pgspecseg(ps) = fst(snd\ ps) \\
\quad pgspeclpno(ps) = snd^2\ ps
\end{array}$$

The type *PAGESPEC* could have been defined earlier in this chapter. If it had been used there, it would have been necessary to represent and manipulate all virtual addresses and page references in terms of *PAGESPEC*. Although this seems attractive, we believe, it would have complicated the specifications somewhat.

The axiomatic definitions begin with *smap*, a relationship between elements of *PAGESPEC*. This is the *sharing map*. When two elements of *PAGESPEC* are in the relation, the processes mentioned as the first component of *PAGESPEC* share the page referred to by the two *PAGESPEC*s.

The operations to allocate, deallocate and share pages are collected into a (somewhat *ad hoc*) class called *VStoreManager*. The class is intended to act as a component in an interface library. It exports most of its operations so that a wide variety of combined operations can be defined elsewhere.

VStoreManager

$\upharpoonright(INIT, AddNewMainStorePageToProcess, AddNewVirtualPageToProcess,$
$CanAddPageToProcess, MarkLogicalPageAsShared, UnshareLogicalPage,$
$WithdrawLogicalPage, SharedLogicalPageSharers, IsSharedLogicalPage,$
$RemoveSharedLogicalPageOwner, RemoveLogicalPageSharer,$
$RawShareLogicalPageBetweenProcesses, ShareLogicalPageBetweenProcesses,$
$ReturnSharedLogicalPageToOwner, MarkSharedLogicalPageAsIn,$
$MarkSharedLogicalPageAsOut, ShareLogicalSegment,$
$ReleaseSharedSegment, ReleaseSegmentPagesExcept,$
$CanReleaseSegment, SharedPagesInSegment, CanReleaseProcessVStore)$

$pts : PageTables$
$pfs : PageFrames$

INIT

$pgtabs? : PageTables$
$pfrms? : PageFrames$

$pts' = pgtabs?$
$pfs' = pfrms?$

$makeEmptyPage \mathrel{\widehat{=}} \ldots$

$AddNewMainStorePageToProcess \mathrel{\widehat{=}} \ldots$

$AddNewVirtualPageToProcess \mathrel{\widehat{=}} \ldots$

$CanAddPageToProcess \mathrel{\widehat{=}} \ldots$

$MarkLogicalPageAsShared \mathrel{\widehat{=}} \ldots$

$UnshareLogicalPage \mathrel{\widehat{=}} \ldots$

$WithdrawLogicalPage \mathrel{\widehat{=}} \ldots$

$SharedLogicalPageSharers \mathrel{\widehat{=}} \ldots$

$IsSharedLogicalPage \mathrel{\widehat{=}} \ldots$

$RemoveSharedLogicalPageOwner \mathrel{\widehat{=}} \ldots$

$RemoveLogicalPageSharer \mathrel{\widehat{=}} \ldots$

$RawShareLogicalPageBetweenProcesses \mathrel{\widehat{=}} \ldots$

$ShareLogicalPageBetweenProcesses \mathrel{\widehat=} \ldots$

$ReturnSharedLogicalPageToOwner \mathrel{\widehat=} \ldots$

$MarkSharedLogicalPageAsIn \mathrel{\widehat=} \ldots$

$MarkSharedLogicalPageAsOut \mathrel{\widehat=} \ldots$

$ShareLogicalSegment \mathrel{\widehat=} \ldots$

$ReleaseSharedSegment \mathrel{\widehat=} \ldots$

$ReleaseSegmentPagesExcept \mathrel{\widehat=} \ldots$

$CanReleaseSegment \mathrel{\widehat=} \ldots$

$SharedPagesInSegment \mathrel{\widehat=} \ldots$

$CanReleaseProcessVStore \mathrel{\widehat=} \ldots$

First, there is the hidden operation that creates an empty page:

```
── makeEmptyPage ──────────────────
│ $pg! : PAGE$
├──────────
│ $pg! = pageofzeroes$
└──────────────────────────────────
```

This operation is hidden because it is somewhat undesirable for everyone to manipulate the buffers in which new pages are created.

The operation to add a new page to a process using a free-store page is the following. It allocates the page frame, adds the page to the process and clears the page (this is not really necessary but is a nice feature[2]).

```
$AddNewMainStorePageToProcess \mathrel{\widehat=}$
    $(pts.HaveFreePages$
        $\land ((pts.IncProcessPageCount \fatsemi (pts.LatestPageCount)$
            $\land pts.AllocateFreePage[ppno/ppno!]$
        $\land pts.AddPageToProcess[ppno/ppno?, lpno!/lpno?]$
        $\land makeEmptyPage[pg/pg?]$
        $\land pts.OverwritePhysicalPage[ppno/pageno?, pg/pg?]) \setminus \{ppno\}))$
```

Note that an alternative is to swap out a process' page. The approach here is simpler but much more profligate.

The next operation uses the prceding one and then stores the page that it has created on the paging disk. Note that the operation requires there to be *at least* one page-sized buffer in the kernel.

```
$AddNewVirtualPageToProcess \mathrel{\widehat=}$
    $(\neg\, pts.HaveFreePages$
        $\land (pts.IncProcessPageCount \fatsemi pts.LatestPageCount)$
```

[2] Some (civilised?) operating systems perform this operation on allocating new store.

$$
\begin{array}{l}
\quad \land (\exists\, ppno : PHYSICALPAGENO \mid ppno = 1 \bullet \\
\qquad (makeEmptyPage[pg/pg!] \\
\qquad\quad \land\ pts.AddPageToProcess[lpno!/lpno?, ppno/ppno?] \\
\qquad\quad \land\ StorePageOnDisk[lpno!/lpno?, pg/pg?])) \setminus \{pg\})
\end{array}
$$

The value assigned to *ppno* is purely arbitrary. The physical page number of any page on disk has no relevance because it will be mapped to another physical page when swapped into main store.

The following is a predicate. It is true if the process denoted by $p?$ can add at least one page to its $sg?$ segment.

```
┌─ CanAddPageToSegment ──────────────────────
│ p? : APREF
│ sg? : SEGMENT
├──────────────
│ pagecount(p?)(sg?) < maxvirtpagespersegment
└────────────────────────────────────────────
```

The actual operation to add a new page to a process is the following. Depending upon the state of main store, it adds the page directly or stores it on the paging disk:

$$
\begin{array}{l}
AddNewPageToProcess \mathrel{\widehat{=}} \\
\quad AddNewMainStorePageToProcess \\
\qquad \lor\ AddNewVirtualPageToProcess
\end{array}
$$

Proposition 181. *If there are no free pages in main store, a zero page is created for it and written to disk.*

Proof. The schema *AddNewPageToProcess* is a disjunction:

$$
\begin{array}{l}
AddNewMainStorePageToProcess \\
\quad \lor\ AddNewVirtualPageToProcess
\end{array}
$$

Each disjunct is a conjunction, with mutually exclusive first conjuncts.

In particular, *AddNewVirtualPageToProcess* has a predicate containing the following:

$$
\begin{array}{l}
makeEmptyPage[pg/pg!] \\
\quad \land \\
\quad \ldots \\
\quad \land\ StorePageOnDisk[\ldots, pg/pg?]
\end{array}
$$

The existential quantifier can be removed by the one-point rule, and the result follows from the fact that $p \land q \Rightarrow p$. □

Proposition 182. *If there are free pages, one is allocated for the newly created page.*

Proof. Similar to the above, concentrating on *AddNewMainStorePage*. Again, $p \land q \vdash p$ allows the conclusion to be drawn. □

The only thing that needs to be done when a page is shared between processes is to mark it. This is actually done by adding *PAGESPEC*s to the sharing map, *smap*:

$$
\begin{array}{|l}
\textit{MarkLogicalPageAsShared} \\
\hline
\Delta(smap) \\
ownproc?, shareproc? : APREF \\
ownseg?, shareseg? : SEGMENT \\
ownlp?, sharelp? : LOGICALPAGENO \\
\hline
(\exists\, atrip1, atrip2 : PAGESPEC \bullet \\
\qquad atrip1 = mkpgspec(ownproc?, ownseg?, ownlp?) \land \\
\qquad atrip2 = mkpgspec(shareproc?, shareseg?, sharelp?) \land \\
\qquad smap' = smap \cup \{(atrip1, atrip2)\}) \\
\hline
\end{array}
$$

Unsharing, conversely, removes a pair of *PAGESPEC*s from the sharing map:

$$
\begin{array}{|l}
\textit{UnshareLogicalPage} \\
\hline
\Delta(smap) \\
p? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
smap' = smap \unrhd \{mkpgspec(p?, sg?, lpno?)\} \\
\hline
\end{array}
$$

If a page is to be reallocated or removed from store completely, it must be totally removed from the sharing map:

$$
\begin{array}{|l}
\textit{WithdrawLogicalPage} \\
\hline
\Delta(smap) \\
owner? : APREF \\
sg? : SEGMENT \\
lpno? : LOGICALPAGENO \\
\hline
smap' = \{mkpgspec(p?, sg?, lpno?)\} \unlhd smap \\
\hline
\end{array}
$$

In later operations, it is necessary to determine which processes share a given page. The page is specified by the *sg*? and *lpno*? inputs and is shared by (at least) *p*?. The relational image is used to determine all the sharing *PAGESPEC*s and, hence, the sharing processes:

SharedLogicalPageSharers

$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$
$srs! : \mathbb{F}\, PAGESPEC$

$smap(\!|\ \{mkpgspec(p?, sg?, lpno?)\}\ |\!) = srs!$

The following predicate is true when the page is shared:

IsSharedLogicalPage

$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$mkpgspec(p?, sg?, lpno?) \in \mathrm{dom}\, smap$

When, say, a process terminates or when a shared page used to contain a message is withdrawn, the entry for the page must also be removed from the *smap*:

RemoveSharedLogicalPageOwner

$\Delta(smap)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$smap' = \{mkpgspec(p?, sg?, lpno?)\} \lhd smap$

All the sharers of a page can be removed from the *smap* by the following operation:

RemoveLogicalPageSharers

$\Delta(smap)$
$p? : APREF$
$sg? : SEGMENT$
$lpno? : LOGICALPAGENO$

$(\exists\, pg : PAGESPEC \mid pg = mkpgspec(p?, sg?, lpno?) \bullet$
$\quad smap' = smap \rhd \{pg\})$

The following pair of schemata define the operation of sharing a page between two processes. The first schema defines the actual operation, while the second hides the logical page number. The first operation returns the logical page number of the shared page. Sometimes, this is something of a nuisance, so the second operation hides it. The first operation returns the page number so that it can be referenced, while the second hides the page number—approximately, it states that a page is shared but does not say which it is.

$$
\begin{array}{l}
\mathit{RawShareLogicalPageBetweenProcesses} \mathrel{\widehat=} \\
\quad ((\mathit{pts.IsLockedPage}[\mathit{owner?}/\mathit{p?}, \mathit{ownseg?}/\mathit{sg?}, \mathit{ownlpno?}/\mathit{lpno?}] \\
\qquad \land \mathit{pts.LockPage}[\mathit{sharer?}/\mathit{p?}, \mathit{shareseg?}/\mathit{sg?}, \mathit{sharelpno!}/\mathit{lpno?}]) \\
\qquad \lor \mathit{Skip}) \\
\quad \land \mathit{NextNewLogicalPageNo}[\mathit{sharer?}/\mathit{p?}, \mathit{shareseg?}/\mathit{sg?}, \mathit{sharelpno!}/\mathit{lpno!}] \\
\quad \land (\mathit{pts.PhysicalPageNo}[\mathit{owner?}/\mathit{p?}, \mathit{ownseg?}/\mathit{sg?}, \mathit{ownlpno?}/\mathit{lpno?}, \mathit{ppno}/\mathit{ppno!}] \\
\quad \land \mathit{pts.AddPageToProcess}[\mathit{sharer?}/\mathit{p?}, \mathit{shareseg?}/\mathit{sg?}, \mathit{sharelpno!}/\mathit{lpno?}] \\
\quad \land \mathit{MarkLogicalPageAsShared}[\mathit{owner?}/\mathit{ownproc?}, \mathit{sharer?}/\mathit{shareproc?}, \\
\qquad\qquad\qquad\qquad \mathit{ownlpno?}/\mathit{ownlp?}, \mathit{sharelpno!}/\mathit{sharelp?}] \\
\quad \land \mathit{pts.MarkPageAsShared}[\mathit{owner?}/\mathit{p?}, \mathit{ownseg?}/\mathit{sg?}, \mathit{ownlpno?}/\mathit{lpno?}] \\
\quad \land \mathit{pts.MarkPageAsShared}[\mathit{sharer?}/\mathit{p?}, \mathit{shareseg?}/\mathit{sg?}, \mathit{sharelpno!}/\mathit{lpno?}]) \\
\qquad\qquad \setminus \{\mathit{ppno}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ShareLogicalPageBetweenProcesses} \mathrel{\widehat=} \\
\quad \mathit{RawShareLogicalPageBetweenProcesses} \setminus \{\mathit{sharelpno!}\}
\end{array}
$$

Proposition 183. *If two processes share a page, pg, that page will appear in the page tables of both processes.*

Proof. The page, $pg?$, is already assumed to be in the page table of its owning process. The predicate of *RawShareLogicalPageBetweenProcesses* contains, $\mathit{AddPageToProcess}[\mathit{sharer?}/\mathit{p?}, \mathit{shareseg?}/\mathit{sg?}, \mathit{sharelpno!}/\mathit{lpno?}]$ as a conjunct. The arguments to the substitution refer to the process that is receiving the page. □

Proposition 184. *Schema RawShareLogicalPageBetweenProcesses makes the specified page shared by both processes.*

Proof. The predicate of *RawShareLogicalPageBetweenProcesses* contains an instance of *MarkPageAsShared*, which simplifies to $\mathit{smap}' = \mathit{smap} \cup \{(s_1, s_2)\}$, where:

$$s_1 = \mathit{mkpgspec}(\mathit{ownproc?}, \mathit{ownseg?}, \mathit{ownlp?})$$

and

$$s_2 = \mathit{mkpgspec}(\mathit{sharer?}, \mathit{shareseg?}, \mathit{sharelp?})$$

and where $\mathit{ownproc?}$ is the owning process identifier and $\mathit{sharer?}$ is the identifier of the process being granted the right to share the page; $\mathit{ownseg?}$ and $\mathit{shareseg?}$ denote the segments in the owner and sharer's spaces; $\mathit{ownlp?}$ is the owning process' logical page number and $\mathit{sharelp?}$ is the logical page number in the sharer's segment. □

The only point to note about the following is that it unshares the page in question.

$$
\begin{array}{l}
\mathit{ReturnSharedLogicalPageToOwner} \mathrel{\widehat{=}} \\
\quad \mathit{pts.IsSharedPage}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}] \\
\quad \wedge\ \mathit{pts.UnsharePage}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}] \\
\quad \wedge\ \mathit{pts.RemovePageProperties}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}] \\
\quad \wedge\ \mathit{pts.RemovePageFromPageTable}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{sharelpno?}/\mathit{lpno?}] \\
\quad \wedge\ \mathit{UnshareLogicalPage}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}]
\end{array}
$$

Proposition 185. *When a page is unshared, it is removed from one of the page tables.*

Proof. The operation *ReturnSharedLogicalPageToOwner* removes the specified page from one of the processes that share it. This is done by removing the page from the sharer's page table by means of the conjunct:

$$\mathit{RemovePageFromPageTable}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}]$$

The other conjuncts cancel various attributes, in particular:

$$\mathit{RemovePageProperties}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}]$$

as well as removing the information that this page is shared by process *sharer?* by

$$\mathit{UnsharePage}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}]$$

and

$$\mathit{UnshareLogicalPage}[\mathit{sharer?}/p?, \mathit{shareseg?}/\mathit{sg}/, \mathit{sharelpno?}/\mathit{lpno?}]$$

. □

Shared pages need to be marked as in-store or out-of-store. The following pair of schemata define these operations. They will be used below.

$$
\begin{array}{l}
\mathit{MarkSharedLogicalPageAsIn} \mathrel{\widehat{=}} \\
\quad (\mathit{pts.IsSharedPage} \wedge \mathit{pts.MarkPageAsIn} \wedge \\
\qquad (\mathit{SharedLogicalPageSharers}[\mathit{srs}/\mathit{srs!}] \wedge \\
\qquad\quad (\forall\, \mathit{pg} : \mathit{PAGESPEC} \mid \mathit{pg} \in \mathit{srs} \bullet \\
\qquad\qquad (\exists\, p : \mathit{APREF};\ \mathit{sg} : \mathit{SEGMENT}; \\
\qquad\qquad\qquad\qquad \mathit{lpno} : \mathit{LOGICALPAGENO} \bullet \\
\qquad\qquad\quad \mathit{pg} = \mathit{mkpgspec}(p, \mathit{sg}, \mathit{lpno}) \\
\qquad\qquad\quad \wedge\ \mathit{pts.MarkPageAsIn}[p/p?, \mathit{sg}/\mathit{sg?}, \mathit{lpno}/\mathit{lpno?}])) \setminus \{\mathit{srs}\})) \\
\quad \vee\ \mathit{pts.MarkPageAsIn}
\end{array}
$$

Proposition 186. *When a shared page is swapped in, it is swapped in in all sharing processes.*

PROOF. The operation *MarkPageAsIn* occurs within the scope of the universal quantifier which ranges over elements of *PAGESPEC*. These elements are exactly those describing the shared page in processes other than its owner. The operation *MarkPageAsIn* is applied to each descriptor (rather, to its components) and records the fact that the page has entered main store in each of the processes that share it. □

$$
\begin{array}{l}
\mathit{MarkSharedLogicalPageAsOut} \mathrel{\widehat=} \\
\quad (\mathit{pts.IsSharedPage} \land \mathit{pts.MarkPageAsOut} \land \\
\quad\quad (\mathit{SharedLogicalPageSharers}[\mathit{srs}/\mathit{srs}!] \land \\
\quad\quad (\forall\, \mathit{pg} : \mathit{PAGESPEC} \mid \mathit{pg} \in \mathit{srs} \bullet \\
\quad\quad\quad (\exists\, p : \mathit{APREF};\ \mathit{sg} : \mathit{SEGMENT}; \\
\quad\quad\quad\quad\quad\quad \mathit{lpno} : \mathit{LOGICALPAGENO} \bullet \\
\quad\quad\quad\quad \mathit{mkpgspec}(p, \mathit{sg}, \mathit{lpno}) = \mathit{pg} \land \\
\quad\quad\quad\quad \mathit{pts.MarkPageAsOut}[p/p?, \mathit{sg}/\mathit{sg}?, \mathit{lpno}/\mathit{lpno}?]))) \setminus \{\mathit{srs}\}) \\
\quad \lor\ \mathit{pts.MarkPageAsOut}
\end{array}
$$

Proposition 187. *When a shared page is swapped out, it is swapped out in all sharing processes.*

PROOF. The operation defined by *MarkPageAsOut* occurs inside the universal quantifier in the predicate of *MarkSharedLogicalPageAsOut*. The universal quantifier ranges over all *PAGESPEC* elements that are related to the process in which the page is being swapped. Each element of type *PAGESPEC* records the process identifier, segment identifier and logical page number of the page in the sharing processes. Therefore, the quantifier ranges over all descriptions of the page in the sharing process. The *MarkPageAsOut* operation is applied to each of these descriptions, thus recording the fact that the page is swapped out. □

Proposition 188. *The operation MarkSharedLogicalPageAsOut can never mark a locked page as swapped out.*

PROOF. By Proposition 154. □

The next schema defines the operation to return the next logical page number available in the segment, $sg?$, of process $p?$ (both $sg?$ and $p?$ are inputs to the schema):

$$
\begin{array}{l}
\mathit{NextNewLogicalPageNo} \mathrel{\widehat=} \\
\quad \mathit{pts.IncProcessPageCount} \mathbin{⨾} \mathit{pts.LatestPageCount}
\end{array}
$$

Segments, too, can be shared and unshared. This is not obviously useful but it turns out to be. When a process spawns a copy of itself or creates a child process that shares its code, the code segment can be shared if it is read-only (executable is a page attribute defined at the start of this chapter).

ShareLogicalSegment

$$
\begin{array}{l}
ownerp?, sharerp? : APREF \\
ownerseg?, sharerseg? : SEGMENT \\
\hline
(\forall\, lpno : LOGICALPAGENO \mid \\
\qquad\qquad lpno \in \operatorname{dom} pagetable(ownerp?)(sharerp?) \bullet \\
\quad (\exists\, ps_1, ps_2 : PAGESPEC;\ nxtlpno : LOGICALPAGENO \bullet \\
\qquad ps_1 = mkpgspec(ownerp?, ownerseg?, lpno) \land \\
\qquad NextNewLogicalPageNo[nxtlpno/lpno!] \land \\
\qquad\quad ps_2 = mkpgspec(sharerp?, sharerseg?, nxtlpno) \land \\
\qquad\quad pts.MarkPageAsShared \\
\qquad\qquad\qquad [ownerp?/p?, ownerseg?/sg?, lpno/lpno?] \land \\
\qquad\quad MarkLogicalPageAsShared[lpno/ownlp?, nxtlpno/sharelp?]))
\end{array}
$$

ReleaseSharedSegment

$$
\begin{array}{l}
\Delta(pagetable) \\
p? : APREF \\
sg? : SEGMENT \\
\hline
pagetable' = \{sg?\} \lhd pagetable(p?)
\end{array}
$$

ReleaseSegmentPagesExcept

$$
\begin{array}{l}
\Delta(pagetable) \\
p? : APREF \\
sg? : SEGMENT \\
except? : \mathbb{F}\, LOGICALPAGENO \\
\hline
pagetable' = ((\operatorname{dom} pagetable(p?)(sg?)) \setminus except?) \lhd pagetable(p?)(sg?)
\end{array}
$$

Proposition 189. *If $except? = \varnothing$, ReleaseSegmentPagesExcept implies that the predicate of CanReleaseSegment is satisfied.*

PROOF. The predicate of *CanReleaseSegment* is:

$$\operatorname{dom} pagetable(p?)(sg?) = \varnothing$$

The predicate of *ReleaseSegmentPagesExcept* is:

$$pagetable' = ((\operatorname{dom} pagetable(p?)(sg?)) \setminus except?) \lhd pagetable(p?)(sg?)$$

Clearly (substituting $\varnothing$ for $except?$):

$$((\operatorname{dom} pagetable(p?)(sg?)) \setminus \varnothing) = \operatorname{dom} pagetable(p?)(sg?)$$

Now,

$$(\mathrm{dom}\, pagetable(p?)(sg?)) \lhd pagetable(p?)(sg?)$$

implies that $\mathrm{dom}\, pagetable(p?)(sg?) = \varnothing$. This is the predicate of the schema *CanReleaseSegment*, so we are done. □

Proposition 190. *A process whose entire virtual store has been released has no pages.*

Proof. Obvious given the last proposition and the appropriate schema definitions. □

The following predicate is true if and only if all the pages in the segment, $sg?$, have been deallocated:

```
── CanReleaseSegment ──────────────────────────
│ $p? : APREF$
│ $sg? : SEGMENT$
├──────────
│ $\mathrm{dom}\, pagetable(p?)(sg?) = \varnothing$
└───────────────────────────────────────────────
```

The pages in the segment $sg?$ of process $p?$ are output by this operation. They constitute the set $sps!$.

```
── SharedPagesInSegment ───────────────────────
│ $p? : APREF$
│ $sg? : SEGMENT$
│ $sps! : \mathbb{F}\, LOGICALPAGENO$
├──────────
│ $(\exists\, psg : \mathbb{F}\, PAGESPEC \bullet$
│       $psg = smap(\!|\, \{pg : PAGESPEC;\ lpn : LOGICALPAGENO \mid$
│                 $pg = mkpgspec(p?, sg?, lpn) \bullet pg\}\, |\!) \land$
│       $sps! = \{pg : PAGESPEC;\ lpno : LOGICALPAGENO \mid pg \in pgs \bullet$
│             $pgspeclpno(pg)\})$
└───────────────────────────────────────────────
```

If a process has no pages and no segments (i.e., they have never been allocated or all have been released), its page-table entry can be released. The following schema is a predicate defining this case.

```
── CanReleaseProcessVStore ────────────────────
│ $p? : APREF$
├──────────
│ $\mathrm{dom}\, pagetable(p?) = \varnothing$
└───────────────────────────────────────────────
```

6.4 Using Virtual Storage

6.4.1 Introduction

This section is a rather looser model than the above model. It deals with the user-level interfaces. The view of virtual store is that much more akin to real store: a linear sequence of locations that can be addressed sequentially. The concept of the virtual address is also different in this section. Instead of a complex structure, virtual addresses are considered atomic, in effect a subset of $\mathbb{N}$.

6.4.2 Virtual Addresses

From this point on, the specification will deal with virtual addresses, not with logical and physical pages. This amounts to the user process' perspective on virtual storage. Hitherto, all the perspective has been that of the virtual storage constructor, so virtual address spaces and virtual addresses have been seen in terms of their components.

For this reason, the following is required:

$$\begin{array}{l} AllocatePageReturningStartVAddress \mathrel{\widehat{=}} \\ \quad AddNewPageToProcess \land ComputePageVAddress[lpno!/lpno?] \end{array}$$

This operation, as its name suggests, allocates a page and returns the virtual address of its start. (Note that the logical page number is *not* hidden; there are reasons for this, as will be seen.)

The following operation computes a virtual address from its components.

$$\begin{array}{|l} \multicolumn{1}{l}{ComputePageVAddress} \\ \hline lpno? : LOGICALPAGENO \\ vaddr! : \mathbb{N} \\ \hline vaddr! = (lpno? - pgallocstart) * framesize \\ \hline \end{array}$$

Here, *pgallocstart* is a constant. It is defined as:

$$\begin{array}{|l} \hline\hline pgallocstart : \mathbb{N} \\ \hline pgallocstart = 0 \\ \hline \end{array}$$

The value of 0 is completely arbitrary, as is now explained.

Some systems map a virtual copy of the operating system onto the virtual address space of each user space (and some privileged processes, too). The virtual copy can be allocated at the start or at the end of virtual store. The allocation of user-process pages has to respect this. Furthermore, there are special addresses that are reserved by the hardware; for example, interrupt vectors, device buffers and status words. These, too, must be allocated somewhere that cannot be directly accessed by untrusted user processes.

The *pgallocstrt* represents the logical page number of the first page in a segment that can be allocated.

For clarity and simplicity, it will be assumed here that the operating system virtual copy is located entirely in its own space. Transfer to the operating system from user space is achieved by a "mode switch". The mode switch activates additional instructions, for example those manipulating interrupts and the translation lookaside buffer. How the mode switch is performed is outside the scope of this book for reasons already given. Mode switches are, though, very common on hardware that supports virtual store.

In some systems, there are parts of the kernel space that are shared between all processes in the system. These pages are pre-allocated and added to the process image when it is created. Because they are pre-allocated, the allocation of user pages in that process must be allocated at some page whose logical page number is greater than zero. The constant *pgallocstart* denotes this offset.

Usually, the offset is used in the data segment only. For simplicity, the offset is here set to 0. Moreover, it is uniformly applied to all segments (since it is 0, this does not hurt).

The one hard constraint on virtual store is that some *physical* pages must never be allocated to user space. These are the pages that hold the device registers and other special addresses just mentioned.

Virtual-store pages are frequently marked as:

- execute only (which implies read-only);
- read-only;
- read-write.

Sometimes, pages are marked write-only. This is unusual for user pages but could be common if device buffers are mapped to virtual store pages.

The operations required to mark pages alter the attributes defined at the start of this chapter. The operations are relatively simple to define and are also intuitively clear. They are operations belonging to the class defined below in Section 6.5.2; in the meanwhile, they are presented without comment.

$$\begin{aligned}
&\mathit{MarkPageAsReadOnly} \mathrel{\widehat{=}}\\
&\quad \mathit{MakePageReadable} \land\\
&\quad ((\mathit{IsPageWritable} \land \mathit{MakePageNotWritable}) \lor\\
&\quad ((\mathit{IsPageExecutable} \land \mathit{MakePageNotExecutable})))
\end{aligned}$$

$$\begin{aligned}
&\mathit{MarkPageAsReadWrite} \mathrel{\widehat{=}}\\
&\quad (\mathit{MakePageReadable} \mathbin{⨾} \mathit{MakePageWritable}) \land\\
&\quad (\mathit{IsPageExecutable} \land \mathit{MakePageNotExecutable})
\end{aligned}$$

$$\begin{aligned}
&\mathit{MarkPageAsCode} \mathrel{\widehat{=}}\\
&\quad (\mathit{MakePageExecutable} \mathbin{⨾} \mathit{MakePageReadable}) \land\\
&\quad (\mathit{IsPageExecutable} \land \mathit{MakePageNotExecutable})
\end{aligned}$$

An extremely useful, but generic, operation is the following. It allocates n pages at the same time:

$$
\begin{array}{l}
\mathit{AllocateNPages} \mathrel{\widehat=} \\
\quad (\forall p : 1 \mathrel{..} \mathit{numpages?} \bullet \\
\quad\quad (p = 1 \land \mathit{AllocatePageReturningStartVAddress}[\mathit{startaddr?}/\mathit{vaddr!}]) \\
\quad\quad \lor (p > 1 \land \\
\quad\quad\quad (\exists \mathit{vaddr} : \mathit{VADDR} \bullet \\
\quad\quad\quad\quad \mathit{AllocatePageReturningStartVAddress}[\mathit{vaddr}/\mathit{vaddr!}])))
\end{array}
$$

As it stands, this operation is not of much use in this model. The reason for this is that it does not set page attributes, in particular the read-only, execute-only and the read-write attributes. For this reason the following operations are defined. The collection starts with the operation for allocating n executable pages.

The following allocation operation is used when code is marked as executable and read-only. This is how Unix and a number of other systems treat code.

$$
\begin{array}{l}
\mathit{AllocateNExecutablePages} \mathrel{\widehat=} \\
\quad (\forall p : 1 \mathrel{..} \mathit{numpages?} \bullet \\
\quad\quad (p = 1 \land \mathit{AllocatePageReturningStartVAddress}[\mathit{startaddr?}/\mathit{vaddr!}] \\
\quad\quad\quad \land \mathit{MarkPageAsCode}) \\
\quad\quad \lor (p > 1 \land \\
\quad\quad\quad (\exists \mathit{vaddr} : \mathit{VADDR} \bullet \\
\quad\quad\quad\quad \mathit{AllocatePageReturningStartVAddress}[\mathit{vaddr}/\mathit{vaddr!}] \land \\
\quad\quad\quad\quad \mathit{MarkPageAsCode})))
\end{array}
$$

Similarly, the following operations allocate n pages for the requesting process. It should be remembered that the pages might be allocated on the paging disk and not in main store.

$$
\begin{array}{l}
\mathit{AllocateNReadWritePages} \mathrel{\widehat=} \\
\quad (\forall p : 1 \mathrel{..} \mathit{numpages?} \bullet \\
\quad\quad (p = 1 \land \mathit{AllocatePageReturningStartVAddress}[\mathit{startaddr?}/\mathit{vaddr!}] \\
\quad\quad\quad \land \mathit{MarkPageAsReadWrite}) \\
\quad\quad \lor (p > 1 \land \\
\quad\quad\quad (\exists \mathit{vaddr} : \mathit{VADDR} \bullet \\
\quad\quad\quad\quad \mathit{AllocatePageReturningStartVAddress}[\mathit{vaddr}/\mathit{vaddr!}] \land \\
\quad\quad\quad\quad \mathit{MarkPageAsReadWrite})))
\end{array}
$$

$$
\begin{array}{l}
\mathit{AllocateNReadOnlyPages} \mathrel{\widehat=} \\
\quad (\forall p : 1 \mathrel{..} \mathit{numpages?} \bullet \\
\quad\quad (p = 1 \land \mathit{AllocatePageReturningStartVAddress}[\mathit{startaddr?}/\mathit{vaddr!}] \\
\quad\quad\quad \land \mathit{MarkPageAsReadOnly}) \\
\quad\quad \lor (p > 1 \land \\
\quad\quad\quad (\exists \mathit{vaddr} : \mathit{VADDR} \bullet \\
\quad\quad\quad\quad \mathit{AllocatePageReturningStartVAddress}[\mathit{vaddr}/\mathit{vaddr!}] \land \\
\quad\quad\quad\quad \mathit{MarkPageAsReadOnly})))
\end{array}
$$

To support the illusion of virtual storage, virtual addresses can be thought of as just natural numbers (including 0):

$VADDR == \mathbb{N}$

and the virtual store for each process can be considered a potentially infinite sequence of equal-sized locations:

VirtualStore

$vlocs : \mathrm{seq}\, PSU$
$maxaddr : \mathbb{N}$

$\#locs = maxaddr$

It must be emphasised that each virtual address space has *its own copy* of the *VirtualStore* schema. Page and segment sharing, of course, make regions of store belonging to one virtual address space appear (by magic?) as part of another.

The usual operations (read and write) will be supported. However, when the relevant address is not present in real store, a *page fault* occurs and the *OnPageFault* driver is invoked with the address to bring the required page into store.

For much of the remainder, a block-copy operation is required. This is used to copy data into pages based on addresses. For the time being, it can be assumed that every address is valid (the hardware should trap this, in any case).

CopyVStoreBlock

$\Delta VirtualStore$
$data? : \mathrm{seq}\, PSU$
$numelts? : \mathbb{N}$
$destaddr? : VADDR$

$$vlocs' = (\lambda\, i : 1 \,..\, (destaddr? - 1) \bullet vlocs(i)) \\ \frown \langle data? \rangle \frown (vlocs\ \ after\ \ (destaddr? + numelts?))$$

If any of the addresses used in this schema are not in main store, the page-faulting mechanism will ensure that it is loaded.

Operations defining the user's view of virtual store are collected into the following class. It is defined just to collect the operations in one place. (In the next section, the operations are not so collected—they are just assumed to be part of a library and are, therefore, defined in Z.)

The class is defined as follows. The definition is somewhat sparse and contains only two operations, *CopyVStoreBlock* and *CopyVStoreFromVStore*. In a full implementation, this class could be extended considerably. The point, here, though, is merely to indicate that operations similar to those often implemented for real store can be implemented in virtual storage systems at a level above that at which virtual addresses are manipulated as complex entities.

$$
\begin{array}{|l}
\textit{UsersVStore} \\
\hline
\upharpoonright(\mathit{INIT}, \mathit{CopyVStoreBlock}, \mathit{CopyVStoreFromVStore}) \\
\\
\quad \begin{array}{|l}
\hline
\mathit{vlocs} : \operatorname{seq} \mathit{PSU} \\
\mathit{maxaddr} : \mathbb{N} \\
\hline
\#\mathit{locs} = \mathit{maxaddr} \\
\hline
\end{array} \\
\\
\quad \begin{array}{|l}
\mathit{INIT} \\
\hline
\mathit{maxaddress?} : \mathbb{N} \\
\hline
\mathit{maxaddr}' = \mathit{maxaddress?} \\
\hline
\end{array} \\
\\
\mathit{CopyVStoreBlock} \mathrel{\widehat{=}} \ldots \\
\mathit{CopyVStoreFromVStore} \mathrel{\widehat{=}} \ldots \\
\hline
\end{array}
$$

Data can be copied into a virtual-store page (by a user process) by the following operation:

$$
\begin{array}{|l}
\textit{CopyVStoreBlock} \\
\hline
\Delta \mathit{VirtualStore} \\
\mathit{data?} : \operatorname{seq} \mathit{PSU} \\
\mathit{numelts?} : \mathbb{N} \\
\mathit{destaddr?} : \mathit{VADDR} \\
\hline
\mathit{vlocs}' = (\lambda\, i : 1 \mathrel{..} (\mathit{destaddr?} - 1) \bullet \mathit{vlocs}(i)) \\
\qquad\qquad\qquad\qquad {}^\frown\langle \mathit{data?} \rangle {}^\frown (\mathit{vlocs} \ \mathit{after} \ (\mathit{destaddr?} + \mathit{numelts?})) \\
\hline
\end{array}
$$

A similar operation is the following. It copies one piece of virtual store to another. It is useful when using pages as inter-process messages: the data comprising the message's payload can be copied into the destination (which might be a shared page in the case of a message) from the page in which it was assembled by this operation:

$$
\begin{array}{|l}
\textit{CopyVStoreFromVStore} \\
\hline
\Delta \mathit{VirtualStore} \\
\mathit{fromaddr?} : \mathit{VADDR} \\
\mathit{toaddr?} : \mathit{VADDR} \\
\mathit{numunits?} : \mathbb{N}_1 \\
\hline
(\exists\, \mathit{endaddr} : \mathit{VADDR} \mid \mathit{endaddr} = \mathit{fromaddr?} + \mathit{numunits?} - 1 \bullet \\
\qquad \mathit{vlocs}' = (\lambda\, i : 1 \mathrel{..} (\mathit{toaddr?} - 1) \bullet \mathit{vlocs}(i)) \\
\qquad\qquad\qquad\qquad {}^\frown(\lambda\, j : \mathit{fromaddr?} \mathrel{..} \mathit{endaddr} \bullet \mathit{vlocs}(j)) \\
\qquad\qquad\qquad\qquad {}^\frown(\mathit{vlocs} \ \mathit{after} \ \mathit{endaddr} + 1)) \\
\hline
\end{array}
$$

6.4.3 Mapping Pages to Disk (and Vice Versa)

Linux contains an operation called `memmap` in its library. This maps virtual store to disk store and is rather useful (it could be used to implement persistent store as well as other things, heaps for instance).

A class is defined to collect the operations together. Again, this class is intended only as an indication of what is possible. In a real system, it could be extended considerably; for example, permitting the controlled mapping of pages between processes, archiving of pages, and so on.

$$
\begin{array}{|l}
\hline
\textit{PageMapping} \\
\upharpoonright(\textit{INIT}, \textit{MapPageToDisk}, \textit{MapPageFromDiskExtendingStore}) \\
\quad \begin{array}{|l}
\hline
\textit{usrvm} : \textit{UserVStore} \\
\textit{pfr} : \textit{PageFrames} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{INIT} \\
\textit{uvm}? : \textit{UserVStore} \\
\textit{pgfrm}? : \textit{PageFrames} \\
\hline
\textit{usrvm}' = \textit{uvm}? \\
\textit{pfr}' = \textit{pgfrm}? \\
\hline
\end{array} \\
\quad \textit{MapPageToDisk} \mathrel{\widehat=} \ldots \\
\quad \textit{MapPageFromDiskExtendingStore} \mathrel{\widehat=} \ldots \\
\quad \textit{writePageToDisk} \mathrel{\widehat=} \ldots \\
\quad \textit{readPageFromDisk} \mathrel{\widehat=} \ldots \\
\hline
\end{array}
$$

The operations in this class are all fairly obvious, as is their operation. Commentary is, therefore, omitted.

$$
\begin{array}{|l}
\hline
\textit{writePageToDisk} \\
\ldots \\
\textit{diskparams}? : \ldots \\
\textit{pg}? : \textit{PAGE} \\
\hline
\ldots \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{MapPageToDisk} \mathrel{\widehat=} \\
\quad (\textit{pgt}.\textit{PhysicalPageNo}[\textit{ppno}/\textit{ppgno}!] \land \\
\qquad \textit{pfr}.\textit{GetPage}[\textit{ppno}/\textit{pageno}?, \textit{pg}/\textit{fr}!] \land \\
\qquad \textit{writePageToDisk}[\textit{pg}/\textit{pg}?]) \setminus \{\textit{ppno}, \textit{pg}\}
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{readPageFromDisk}} \\ \hline
\ldots \\
\textit{diskparams}? : \ldots \\
\textit{pg}! : \textit{PAGE} \\ \hline
\ldots \\ \hline
\end{array}$$

This operation extends the store of the requesting process. It is used when reading pages from disk. The disk page is added to the process' virtual storage image.

$$\begin{array}{l}
\textit{MapPageFromDiskExtendingStore} \mathrel{\widehat{=}} \\
\quad \textit{usrvm.AllocatePageReturningStartVAddress}[\textit{pageaddr}!/\textit{vaddr}!] \mathbin{;} \\
\qquad (\textit{readPageFromDisk}[\textit{pg}/\textit{pg}!] \land \\
\qquad (\exists\, \textit{sz} : \mathbb{N} \mid \textit{sz} = \textit{framesize} \bullet \\
\qquad\quad \textit{usrvm.CopyVStoreBlock} \\
\qquad\qquad\qquad [\textit{sz}/\textit{numelts}?, \textit{pg}/\textit{data}?, \textit{pageaddr}!/\textit{destaddr}?])) \\
\qquad\qquad\qquad\qquad\qquad \setminus \{\textit{pg}\}
\end{array}$$

There ought to be an operation to delete the page from the image. However, only the most careful programmers will ever use it, so the operation is omitted. It is, in any case, fairly easy to define.

Note that there is no operation to map a disk page onto an existing virtual-store page. This is because it will probably be used extremely rarely.

The operations in this class could be extended so that the specified disk as well as the paging disk get updated when the frame's counter is incremented. This would automatically extend the disk image. A justification for this is that it implements a way of protecting executing processes from hardware and software failure. It can be used as a form of journalling.

This scheme can also be used on disk files. More generally, it can also work on arbitrary devices. This could be an interesting mechanism to explore when considering virtual machines of greater scope (it is an idea suggested by VME/B). Since this is just speculation, no more will be said on it.

6.4.4 New (User) Process Allocation and Deallocation

This section deals *only* with user-process allocation and deallocation. The general principles are the same for system processes but the details might differ slighty (in particular, the default marking of pages as read-only, etc.).

When a new process is created, the following schema is used. In addition, the virtual-store-management pages must be set up for the process. This will be added to the following schema in a compound definition.

UserStoreMgr

$\upharpoonright(\mathit{INIT}, \mathit{MarkPageAsReadOnly}, \mathit{MarkPageAsReadWrite}, \mathit{MarkPageAsCode},$
$\quad \mathit{AllocateNPages}, \mathit{AllocateNExecutablePages}, \mathit{AllocateNReadWritePages},$
$\quad \mathit{AllocateNReadOnlyPages}, \mathit{CopyVStoreBlock}, \mathit{CopyVStoreFromVStore},$
$\quad \mathit{AllocateNewProcessStorage}, \mathit{ReleaseSharedPages},$
$\quad \mathit{FinalizeProcessPages}, \mathit{AllocateCloneProcessStorage})$

$\mathit{usrvm} : \mathit{UserVStore}$
$\mathit{pgt} : \mathit{PageTables}$

INIT

$\mathit{uvm}? : \mathit{UserVStore}$
$\mathit{ptbl}? : \mathit{PageTables}$

$\mathit{usrvm}' = \mathit{uvm}?$
$\mathit{pgt}' = \mathit{ptbl}?$

$\mathit{MarkPageAsReadOnly} \mathrel{\widehat{=}} \ldots$
$\mathit{MarkPageAsReadWrite} \mathrel{\widehat{=}} \ldots$
$\mathit{MarkPageAsCode} \mathrel{\widehat{=}} \ldots$
$\mathit{AllocateNPages} \mathrel{\widehat{=}} \ldots$
$\mathit{AllocateNExecutablePages} \mathrel{\widehat{=}} \ldots$
$\mathit{AllocateNReadWritePages} \mathrel{\widehat{=}} \ldots$
$\mathit{AllocateNReadOnlyPages} \mathrel{\widehat{=}} \ldots$
$\mathit{CopyVStoreBlock} \mathrel{\widehat{=}} \ldots$
$\mathit{CopyVStoreFromVStore} \mathrel{\widehat{=}} \ldots$
$\mathit{AllocateNewProcessStorage} \mathrel{\widehat{=}} \ldots$
$\mathit{ReleaseSharedPages} \mathrel{\widehat{=}} \ldots$
$\mathit{FinalizeProcessPages} \mathrel{\widehat{=}} \ldots$
$\mathit{AllocateCloneProcessStorage} \mathrel{\widehat{=}} \ldots$

AllocateNewProcessStorage

$p? : \mathit{APREF}$
$\mathit{codepages}? : \mathrm{seq}\, \mathit{PSU}$
$\mathit{codesz}?, \mathit{stacksz}?, \mathit{datasz}?, \mathit{heapsz}? : \mathbb{N}$

$(\exists\, \mathit{sg} : \mathit{SEGMENT};\ \mathit{codeszunits} : \mathbb{N} \mid$
$\qquad \mathit{sg} = \mathsf{code} \land \mathit{codeszunits} = \#\mathit{codepages}? \bullet$

$$
\begin{array}{l}
\quad\quad AllocateNExecutablePages \\
\quad\quad\quad\quad\quad [codesz?/numpages?, addr/startaddr!] \fatsemi \\
\quad\quad\quad usrvm.CopyVStoreBlock \\
\quad\quad\quad\quad\quad [codepages?/data?, codeszunits/numelts?, \\
\quad\quad\quad\quad\quad\quad addr/destaddr?]) \\
(\exists sg : SEGMENT \mid sg = \mathsf{data} \bullet \\
\quad\quad AllocateNReadOnlyPages[datasz?/numpages?]) \\
(\exists sg : SEGMENT \mid sg = \mathsf{stack} \bullet \\
\quad\quad AllocateNReadWritePages[stacksz?/numpages?]) \\
(\exists sg : SEGMENT \mid sg = \mathsf{heap} \bullet \\
\quad\quad AllocateNReadWritePages[heapsz?/numpages?])
\end{array}
$$

$$
\begin{array}{l}
\textit{ReleaseSharedPages} \\
\hline
\Delta(smap) \\
p? : APREF \\
\hline
(\forall ps : PAGESPEC \mid ps \in \operatorname{dom} smap \wedge pgspecpref(ps) = p? \bullet \\
\quad\quad (\exists s : PAGESPEC \mid (ps, s) \in smap \bullet \\
\quad\quad\quad\quad smap' = smap \setminus \{(ps, s)\})) \\
\wedge\ (\forall ps : PAGESPEC \mid ps \in \operatorname{ran} smap \wedge psgpecpref(ps) = p? \bullet \\
\quad\quad (\exists s : PAGESPEC \mid (s, ps) \in smap \bullet \\
\quad\quad\quad\quad smap' = smap \setminus \{(s, ps)\}))
\end{array}
$$

Once this schema has been executed, the process can release all of its pages:

$$
\begin{array}{l}
FinalizeProcessPages \mathrel{\widehat=} \\
\quad\quad pgt.RemovePageProperties \wedge pgt.RemoveProcessFromPageTable
\end{array}
$$

$$
\begin{array}{l}
\textit{AllocateCloneProcessStorage} \\
\hline
p? : APREF \\
clonedfrom? : APREF \\
stacksz?, datasz?, heapsz? : \mathbb{N} \\
\hline
(\exists sg : SEGMENT \mid sg = \mathsf{code} \bullet \\
\quad\quad ShareLogicalSegment[clonedfrom?/ownerp?, p?/sharerp?, \\
\quad\quad\quad\quad\quad\quad sg/ownerseg?, sg/sharerseg?]) \\
(\exists sg : SEGMENT \mid sg = \mathsf{data} \bullet \\
\quad\quad AllocateNReadOnlyPages[datasz?/numpages?]) \\
(\exists sg : SEGMENT \mid sg = \mathsf{stack} \bullet \\
\quad\quad AllocateNReadWritePages[stacksz?/numpages?]) \\
(\exists sg : SEGMENT \mid sg = \mathsf{heap} \bullet \\
\quad\quad AllocateNReadWritePages[heapsz?/numpages?])
\end{array}
$$

This works because of the following argument. The first of the two schemata (*ReleaseSharedPages*) above first removes the process from all of the pages that it shares but does not own. Then it removes itself from all of those shared pages that it does own. This leaves it with only those pages that belong to it and are not shared with any other process.

If a child process performs the first operation, it will remove itself from all of the pages it shares with its parents; it will also delete all of the pages it owns. The parent is still in possession of the formerly shared pages, which might be shared with other processes. As long as the parent is blocked until all of its children have terminated, it cannot delete a page that at least one of its children uses. Thus, when all of a process' children have terminated, the parent can terminate, too. Termination involves execution of the operations defined by *ReleaseSharedPages* and *FinalizeProcessPages*.

The only problem comes with clones. If the clone terminates before the original, all is well. Should the original terminate, it will delete pages still in use by the clone. Therefore, the original must also wait for the clone to terminate.

An alternative—one that is possible—is for the owner to "give" its shared pages to the clone. Typically, the clone will only require the code segment and have an empty code segment of its own. If the code segment can be handed over to the clone in one operation (or an *atomic* operation), the original can terminate without waiting for the clone or clones. Either is possible.

The allocation of child processes is exactly the same as cloning. The difference is in the treatment of the process: is it marked as a child or as a completely independent process? Depending upon the details of the process model, a child process might share code with its parent (as it does in Unix systems), whereas an independent process will tend to have its own code (or maybe a copy of its creator's code). In all cases, the data segment of the new process, as well as its stack, will be allocated in a newly allocated set of pages. In this chapter's model, data and stack will be allocated in newly allocated segments. The mechanisms for sharing segments of all kinds have been modelled in this chapter, as have those for the allocation of new segments (and pages). The storage model presented in this chapter can, therefore, support many different process models.

6.5 Real and Virtual Devices

There is often confusion between real and virtual devices. It is sometimes thought that the use of virtual store implies the use of virtual devices. This is not so. In most operating systems with virtual store, the devices remain real, while in some real-store operating systems, devices are virtual.

Virtual devices are really interfaces to actual, real ones. Virtual devices can be allocated on the basis of one virtual device to each process. The virtual device sends messages to and receives them from the device process. Messages are used to implement requests and replies in the obvious fashion. Messages to the real device from the virtual devices are just enqueued by the device process and serviced in some order (say, FIFO).

The interface to the virtual device can also abstract further from the real device. This is because virtual devices are just pieces of software. For example, a virtual disk could just define read and write operations, together with return codes denoting the success of the operation. Underneath this simple interface, the virtual device can implement more complex interfaces, thus absolving the higher levels of software from the need to deal with them. This comes at the cost of inflexibility.

This model can be implemented quite easily using the operations already defined in this book. Using message passing, it can be quite nicely structured.

There is another sense in which devices can be virtualised. Each device interface consists of one or more addresses. Physical device interfaces also include interrupts. Operations performed on these addresses control the device and exchange data between device and software. The addresses at which the device interface is located are invariably fixed in the address map. However, in a virtual system, there is the opportunity to map the pages containing device interfaces are mapped into the address space of each process. (This can be done, of course, using the sharing mechanism defined in this chapter.) This allows processes directly to address devices. However, some form of synchronisation must be included so that the devices are fairly shared between processes (or virtual address spaces). Such synchronisation would have to be included within the software interface to each device and this software can be at as low a level as desired.

A higher-level approach is to map standard addresses (by sharing pages) into each address space but to include a more easily programmed interface. Again, the mechanisms defined in this book can be used as the basis for this scheme.

6.6 Message Passing in Virtual Store

At a number of points in this chapter, the idea of using shared pages (or sets of shared pages) to pass messages between processes has been raised. The basic mechanisms for implementing message passing have also been defined.

When one process needs to send a message to another, it will allocate a page and mark it as shared with the other process. Data will typically be placed in the page before sharing has been performed. The data copy operation can be performed by one of the block-copy operations, *CopyVStoreBlock* or *CopyVStoreFromVStore* (Section 6.5.1).

The receiving process must be notified of the existence of the new page in its address space. This can be achieved as either a synchronous or an asynchronous event—the storage model is completely neutral with respect to this. In a system with virtual storage, message passing will be implemented as system calls, so notification can be handled by kernel operations. For example, the synchronous message-passing primitives defined in Chapter 5 can easily be modified to do this. What is required is that the message call point to a page and not to a small block of storage. Equally, the asynchronous mechanism outlined in Chapter 3 can be modified in a similar fashion.

Message passing based on shared pages will be somewhat slower at runtime than a scheme based upon passing pointers to shared storage blocks (buffers), even when copying buffers between processes is required. The reason for this is clear from an inspection of the virtual storage mechanisms. For this reason, it would probably be best to implement two message-passing schemes: one for kernel and one for user messages. The kernel message scheme would be based on shared buffers within kernel space; user messages would use the shared-page mechanism outlined above.

In some cases, additional system processes are required in addition to those executing inside the kernel address space and they will be allocated their own virtual store. In order to optimise message passing between these processes and the kernel, a set of pages can be declared as shared but not *incore* (i.e., not locked into main store). The set of pages can be pre-allocated by the kernel at initialisation time, so no new pages need to be allocated. All that remains is for the pages to be given to the processes. This can be achieved using the primitives defined in this chapter.

6.7 Process Creation and Termination; Swapping

Process creation, activation and termination are unaffected by the virtual storage mechanisms. The virtual storage subsystem must be booted before any processes are created, so all processes, even those inside the kernel, are created in virtual address space. The primitives to allocate and deallocate storage have been defined above (Sections 6.5.1 and 6.5.3). The operations to create and delete processes can be implemented in a way analogous to those defined in Chapter 4 (and assumed in Chapter 5), with the virtual-store primitives replacing those handling real store. The most significant difference between the two schemes is that the virtual-store allocation operations are not as limited in the amount of store they can allocate. The virtual storage operations are only limited by the number of pages permitted in a segment and not be the size of main store.

Virtual store also has advantages where swapping is concerned. It is possible to include a swapping system in a virtual-store-based system. As with the scheme defined in detail in Chapter 4, the swapper will transfer entire process images between main store and the swap disk (or swap file). Under virtual storage, the swapper treats the page as the basic unit for transfer. The swapper reads the page table and swaps physical pages to disk. Not all segments need be swapped to disk; code segments might be retained in main store while there are active child processes. The process is, however, complicated by the fact that a process image is likely to be shared between the paging disk and main store.