On the implementation of virtual storage: a top-down approach (part 0)

0. Introduction

In this note we investigate how, in a virtual storage environment, information is to be addressed and what administration is needed for this purpose. The approach taken is usually called *step-wise refinement* : starting with an abstract program gradually more details are introduced as the development proceeds. In this respect this note not only is a study of some aspects of a possible component of operating systems; it can equally well be considered as a programming exercise.

Our starting point is the following specification:

```
type word;
     index;
     page  = index → word;
     pagename;
     store = pagename → page;
var  vs : store;

proc read(?p: pagename; ?d: index; !x: word) =
|[ {pre:  vs•p•d = X } {post:  vs•p•d = X ∧ x = X } ]|;
proc write(?p: pagename; ?d: index; ?x: word) =
|[ {pre:  x = X } {post:  vs•p•d = X } ]|.
```

(**Notational remark:** for (abstract) types U and V the abstract type U → V denotes the type of all functions from U to V ; in more concrete representations the corresponding type is the array although other representations, of course, are possible.)

The unit of addressability of the store is word . Values of type word are uninterpreted in this context. The only operation on variables of this type is assignment. The type pagename is the collection of values used to identify the pages constituing the virtual store of the process. For the time being we consider each process in the system in isolation: possible interactions between processes -- such as *page*

sharing -- will be taken into account in part 1. Accordingly, the nomenclature of the pages is a *local* nomenclature within the process. Further details of this nomenclature are considered irrelevant here. The fact that each page is a collection of words instead of a single word is reflected by the choice of the types index and page . For the purpose of this discussion the internal structure of pages is of minor importance; it is only included for the sake of completeness. The virtual store is a collection of pages identified by means of pagenames; formally, the virtual store is a function of type pagename → page . The variable vs represents the virtual store.

Our purpose is the derivation of more and more detailed implementations of the two procedures read and write . The following may be considered both as an operational version of the above specification and as the zeroest, most abstract approximation of our goal:

```
proc read(?p: pagename; ?d: index; !x: word) =
|[ x := vs•p•d ]|;
proc write(?p: pagename; ?d: index; ?x: word) =
|[ vs•p•d := x ]|.
```

1. Primary store and secondary store

In this version the "idea" of virtual storage is taken into account: the virtual store vs is partitioned into two variables ps -- primary store -- and ss -- secondary store -- with the operational understanding that the "capacity" of ps is relatively small but accesses to ps take much less time than accesses to ss . Because ps is not intended to be a fixed subset of vs its implementation requires administration of the way in which the pages of vs are distributed over ps and ss . For this purpose we introduce a variable pp -- present pages -- in which the *names* of the pages present in primary store are recorded. The actual operations of reading and writing a word from or into the store are now confined to pages in primary store; accesses to a page in secondary store will result in copying the page into primary store first. The detection of the fact that an accessed page is not in primary store is called a *page fault*. The crucial assumption behind the idea of virtual storage is that page faults are the exception rather than the rule. Because this part is common to both read and write we "factor it out" by the introduction of a procedure

getpage . This is the only occasion at which pages are moved into primary store; this is called *demand paging*. As a result of these design decisions we obtain our first approximation; the fact that the capacity of primary store is limited has not yet been taken into account:

```
var  ps, ss : store;
     pp : pagename → bool;
```

(representation) invariant:

```
P0: (A i: i∈pagename : (pp•i ∧ vs•i = ps•i) ∨ (¬pp•i ∧ vs•i = ss•i))

proc read(?p: pagename; ?d: index; !x: word) =
|[ { P0 }
   getpage(p)
   { P0 ∧ pp•p , hence: vs•p = ps•p }
;  x := ps•p•d
   { P0 ∧ x = vs•p•d }
]|;
proc write(?p: pagename; ?d: index; ?x: word) =
|[ { P0 }
   getpage(p)
   { P0 ∧ pp•p , hence: vs•p = ps•p }
;  ps•p•d := x
   { P0 ∧ vs•p•d = x }
]|;
proc getpage(?p: pagename) =
{ pre:  P0 ∧ vs = VS }
{post:  P0 ∧ vs = VS ∧ pp•p }
|[ do ¬pp•p → { P0 ∧ vs = VS ∧ ¬pp•p, hence: vs•p = ss•p }
                ps•p , pp•p := ss•p , true
                { P0 ∧ vs = VS ∧ pp•p }
   od { P0 ∧ vs = VS ∧ pp•p }
]|.
```

Notice that the repetition in getpage is a *pseudo repetition*: its repeatable statement

always is executed at most once. For the sake of brevity, we shall use such pseudo repetitions more often. Finally, notice that getpage leaves the contents of the virtual store unaffected.

2. The limited size of primary store

In our next approximation we take into consideration the fact that the capacity of primary store is limited. Moreover, we implement a slight optimisation. Let the positive constant W -- the *window size* -- be the maximal number of pages that may be in primary store at the same time. As a shorthand, we record in the variable w the number of pages present in primary store. Then the relation $0 \leqslant w \leqslant W$ is to be kept invariant. To this end, getpage must be modified in such a way that whenever $\neg pp \cdot p \wedge w = W$ holds one of the pages is removed from primary store before page p is moved into primary store. The selection of the page to be removed is delegated to the procedure victim , the implementation of which is considered to fall outside the scope of this little study: here, we are not interested in *replacement algorithms*.

The optimisation consists in the suppression of the assignment $ss \cdot q := ps \cdot q$ in those cases where it is certain that $ss \cdot q = ps \cdot q$ already holds, namely when the page in primary store has not been modified (written into). To this end, we record in the variable mp which of the pages in primary store have been accessed via the procedure write .

```
var  ps, ss : store;
     pp, mp : pagename → bool;
     w      : int;
```

invariant:

Q: $P0 \wedge P1 \wedge P2$, where

P0: $(\mathbf{A}\, i:\ i \in \text{pagename} :\ (pp \cdot i \wedge vs \cdot i = ps \cdot i) \vee (\neg pp \cdot i \wedge vs \cdot i = ss \cdot i))$

P1: $w = (\mathbf{N}\, i:\ i \in \text{pagename} :\ pp \cdot i) \wedge 0 \leqslant w \leqslant W$

P2: $(\mathbf{A}\, i:\ i \in \text{pagename} :\ \neg pp \cdot i \vee mp \cdot i \vee ps \cdot i = ss \cdot i\ )$

The derivation of the following code now leaves us hardly any choice:

```
proc read(?p: pagename; ?d: index; !x: word) =
|[ { Q }
   getpage(p)
   { Q ∧ pp•p , hence: vs•p = ps•p }
;  x := ps•p•d
   { Q ∧ x = vs•p•d }
]|;
proc write(?p: pagename; ?d: index; ?x: word) =
|[ { Q }
   getpage(p)
   { Q ∧ pp•p , hence: vs•p = ps•p }
;  ps•p•d , mp•p := x , true
   { Q ∧ vs•p•d = x }
]|;
proc getpage(?p: pagename) =
{ pre:  Q ∧ vs = VS }
{post:  Q ∧ vs = VS ∧ pp•p }
|[ do ¬pp•p → { Q ∧ vs = VS ∧ ¬pp•p, hence: vs•p = ss•p }
              do w = W → |[var q : pagename;
                           { Q ∧ w = W , hence: 1 ⩽ w }
                           victim(q)
                           { pp•q }
                         ;  do mp•q → { Q ∧ pp•q ∧ mp•q }
                                        ss•q , mp•q := ps•q , false
                                 od { Q ∧ w = W ∧ pp•q ∧ ¬mp•q , hence: }
                                 { pp•q ∧ vs•q = ss•q }
                         ;  pp•q , w := false , w-1
                         ]| { Q ∧ 0 ⩽ w < W }
              od { Q ∧ vs = VS ∧ ¬pp•p ∧ 0 ⩽ w < W }
           ;  ps•p , pp•p , mp•p , w := ss•p , true , false , w+1
              { Q ∧ vs = VS ∧ pp•p }
   od { Q ∧ vs = VS ∧ pp•p }
]|;
proc victim(!q: pagename) =
|[ {pre:  (E i: i∈pagename : pp•i) } {post: pp•q } ]|.
```

Notice that, as in the previous version, the assignment ps•p := ss•p represents the copying of a page from secondary store into primary store; on the other hand, the assignment ss•q := ps•q represents the copying of a page from primary store back into secondary store. The procedure victim simply assigns to its result parameter the name of a page in primary store; it has no side effects on the variables used here, but it may affect other variables such as those used for the administration needed for the replacement algorithm.

For the above programs it can be easily shown that $w = W$ is invariant too; apparently, states satisfying $w < W$ only occur during the initial phase of the process. This fact can be exploited to simplify the code of getpage . By means of a suitably chosen initialisation of pp and mp -- introducing *dummy pages* that are never referenced -- w can be initialised to W ; hence, the case analysis on w and the variable w itself can be eliminated. I regard such a transformation as a *coding trick* that may justifiably be applied only as a *final optimisation* of the program code. We do not pursue this idea any further here.

3. Implementation

In this section we introduce a slightly more concrete representation of the variables ps, ss, pp, and mp . We start with a more specific version of the types index, page, and pagename :

```
const  M;     { the number of pages constituing the virtual store }
       N;     { the number of words within a page }

type   index  = [0..N-1];
       page   = array index of word;
   pagename   = [0..M-1];
```

The variables ps, ss, pp, and mp are functions with domain pagename ; these variables, therefore, could be represented by means of 4 array variables with this domain. They can also be represented by a single array variable the elements of which are *records* with 4 *fields*, one for each of the abstract variables. We choose the latter possibility. When we look at the invariants P0 through P2 we discover

that the values of ps·i and mp·i are only relevant in those states where pp·i is true ; hence, the record fields corresponding to these two components need only be defined when pp·i is true . This circumstance can be reflected by the choice of a *variant record*. The array of records thus obtained is called the *page table*.

Until now we have neglected the problems associated with *primary store management*: we have obtained a design with the property that at any time at most W pages are present in primary store but we have not addressed the question *where* in primary store these pages are located. We simply have assumed the existence of the variable ps which maps the names of the present pages to their actual contents. Similar remarks apply to the pages in secondary store. The usual solution to this is not to record in the page table the contents of the pages but to record the addresses -- in primary and secondary store respectively -- of the pages instead. In a PASCAL-like programming notation this decision can be reflected by the use of *pointer types*.

Combination of all of the above yields the following definitions:

```
type pagedescriptor = record ssp : ↑page
                             ; case pr : bool
                               of true : (psp : ↑page
                                          ;  m : bool
                                          )
                               end
                      end;
     pagetable      = array pagename of pagedescriptor;
var  pt : pagetable;
```

These definitions do not reflect the fact that usually primary and secondary store are technically of a completely different nature: the above definition unjustly suggest that primary and secondary store are of the same kind. For our purpose this is harmless.

The relation between the new variable pt and the old variables ps, ss, pp, and mp is fixed by the additional representation invariant:

$$P3: (\forall i: 0 \leqslant i < M : ss{\cdot}i = pt[i]{\cdot}ssp\uparrow \wedge pp{\cdot}i = pt[i]{\cdot}pr \wedge ps{\cdot}i = pt[i]{\cdot}psp\uparrow \wedge mp{\cdot}i = pt[i]{\cdot}m)$$

For the purpose of primary storage allocation we assume the availability of two procedures acquire(!fp: ↑page) and release(?fp: ↑page) by means of which a segment of primary store, sufficiently large to hold one page, can be allocated and deallocated respectively. The segments thus allocated are called *page frames*. When, initially, W many page frames are allocated to the process then allocation of page frames to pages can be considered as a purely local activity of the process. On the other hand, it is possible to conceive one global pool of page frames from which all processes obtain their page frames. In the latter case acquire and release are global procedures. W can now be interpreted as to represent the maximal number of page frames the process is allowed to have allocated simultaneously.

The result of this transformation is (annotations have been omitted):

```
proc read(?p: pagename; ?d: index; !x: word) =
|[ getpage(p)
;  x := pt[p]•psp↑[d]
]|;
proc write(?p: pagename; ?d: index; ?x: word) =
|[ getpage(p)
;  pt[p]•m := true  ;  pt[p]•psp↑[d] := x
]|;
proc getpage(?p: pagename) =
|[ do ¬pt[p]•pr → do w = W → |[var q : pagename;
                                 victim(q)
                               ;  do pt[q]•m → pt[q]•ssp↑ := pt[q]•psp↑
                                               ;  pt[q]•m     := false
                                  od
                               ;  release(pt[q]•psp)
                               ;  pt[q]•pr := false  ;  w := w - 1
                               ]|
                  od
               ;  pt[p]•pr := true  ;  w := w + 1
               ;  acquire(pt[p]•psp)
               ;  pt[p]•psp↑ := pt[p]•ssp↑  ;  pt[p]•m := false
   od
]|.
```

The temptation to apply the coding trick mentioned at the end of the previous section is now a little bit more pressing: it seems somewhat superfluous to have deallocation of a page frame being immediately followed by allocation of one, whereas the same page frame could equally well be "recycled" without temporary deallocation.

4. Implications for the design of the central processor

We conclude this part with a few remarks on processor design. Accesses to individual words in the store -- virtual or not -- may be considered as the elementary operations by means of which the processor manipulates the contents of the store. Hence, it is essential that these operations be implemented as efficiently as possible. If virtual storage is to be an essential component of a computing system then the above operations must, at least partly, be implemented in hardware. When page faults indeed are relatively rare it suffices to have a hardware implementation of 0) the detection that the page accessed is present in primary store, followed by 1) the actual read or write operation. This can be achieved by equipping the processor with two instructions read and write , as follows:

```
proc read(?p: pagename; ?d: index; !x: word) =
|[ do ¬pt[p]•pr → "page fault interrupt" od
;  x := pt[p]•psp↑[d]
]|;
proc write(?p: pagename; ?d: index; ?x: word) =
|[ do ¬pt[p]•pr → "page fault interrupt" od
;  pt[p]•m := true  ;  pt[p]•psp↑[d] := x
]|.
```

These instructions differ from the equally named procedures in the previous section in that the test on presence of the page in primary store has been included here (hence, this test can be removed from getpage). These instructions can be executed without any delay by the hardware of the processor in those cases where the page accessed indeed is in primary store. The expression "page fault interrupt" indicates that when the page is absent the processor is supposed to temporarily stop execution of the current program and execute a procedure call to the procedure getpage in very much the same way as if an interrupt had occurred. This event therefore is called an *internal* or *software interrupt*. Such a mechanism makes it possible to delegate the

more complicated but less frequently executed parts of an instruction to an operating system procedure. This approach not only results in a cheaper processor without significant loss of efficiency, but also gives rise to increased flexibility; e.g. the choice of the replacement algorithm can now be left to the user of the machine. As a matter of fact, each process can now have "its own" replacement algorithm! Although internal interrupts and -- for reasons of contrast: *external* -- interrupts are superficially of the same kind, we must stress that there is (at least) one marked difference that represents a serious pitfall. The processor's reaction to an external interrupt is usually -- exceptions to this rule do exist -- postponed until completion of the instruction currently executed. As a consequence, despite the possibility of interrupts single instructions may be considered as atomic actions (here, I deliberately ignore interference between instructions executed by different processors in a multi-processor installation). The processor's reaction to an internal interrupt, however, cannot be postponed until completion of the instruction causing the interrupt: the interrupt is a signal that the instruction cannot be completed before certain additional conditions are satisfied. As a consequence, such instructions may generally not be considered as atomic actions (not even in single-processor installations). Moreover, the state information to be saved by the processor such that later resumption of the instruction is possible can, both qualitatively and quantitatively, be quite different from the state information to be saved after occurrence of an external interrupt. We conclude that the seemingly simple idea of the internal interrupts should only be applied with great care. Notice that in our examples the situation is remarkably simple: when the page fault interrupt occurs the instructions have not yet caused any state changes; to all intents and purposes the page fault interrupt may be thought of as having occurred immediately before execution of the read or write instruction started.

The implementation of the instructions read and write requires that the processor has access to the process's page table. This is easy: one of the processor's index registers must be reserved to hold the address of the process's page table; this register then belongs to -- what Edsger W Dijkstra called -- the *primary allocation commitment* of the process.

Finally, notice that by now processor design cannot be completely separated from operating system design. This is not as strange as it at first sight may seem; it merely is yet another confirmation of the fact that there is no point in designing

whatever mechanism without any knowledge on how the mechanism is supposed to be used. In our case, the decision to implement a virtual storage system has far-reaching consequences for the architecture of the central processor. Conversely, the overhead due to the implementation of virtual storage on machines in which the processor has not been designed for the purpose is mostly so large that virtual storage is not a realistic proposal for such machines.

(end of part 0)

Eindhoven, 1987.5.19
Rob Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology

On the implementation of virtual storage (part 1)

(**note**: this is a continuation of rh94.0)

5. Page sharing

Until now we have studied the implementation of virtual storage for a single process in isolation. We now consider the situation where, in a multiprocessing environment, different processes may access a common collection of pages -- containing, for instance, shared variables or library procedures -- . Generally, some of the pages used by a process will be *shared* pages and all of its other pages will be *private* pages. Not surprisingly, shared pages will require a somewhat more complicated administration than private pages; hence, within the administration the distinction between private and shared pages must be recorded. In what follows we shall try to treat shared pages and private pages on equal footing as much as possible. In particular, we are heading for a design such that for pages in the process's *window* -- see later -- the distinction between private and shared pages is void; i.e. when a page is in the window the code executed during an access to that page is independent of the kind of page. As a consequence, the same instructions read and write , as introduced in section 4, can be used for both kinds of pages. Moreover, the efficiency of accesses to such pages is independent of the kind of page accessed.

Until now pages were identified by means of a nomenclature local to the process to which the pages belonged. In view of the coexistence of shared and private pages we wish to retain such a local nomenclature. On the other hand, a global nomenclature for the identification of the shared pages within the system as a whole is needed. For this purpose we introduce *global page names*, to be represented by values of the new type gpagename . The global page names can be used as selectors into a *global page table* in which information on the shared pages pertinent to all processes is recorded whereas, as before, each process keeps a *local page table* for its own purposes. The local page tables then provide the mapping of (local) page names to global page names.

One and the same page can now be present in primary store for a number of processes. In order to come to grips with this we define, for each process, the

window of that process as the collection of pages -- either shared or private -- present in primary store for that process. The first sentence of this paragraph then can be formulated as: one and the same page can be in a number of windows simultaneously. A page now is present in primary store if and only if it is in the window of at least one process. Only one copy of the page, however, may be present in primary store at the same time: because we did not exclude the possibility that processes modify shared pages, such pages must be considered as volatile; hence, duplication of pages is better avoided. Each process keeps track of the pages in its window by means of the field pr in its page table, as before; in the global page table we record for each page the number of windows the page is in. When this number is zero the page is in no window; hence it can be removed from primary store. As long as this number is positive the page is in some window and must, therefore remain present. The primary and secondary store addresses and the boolean indicating that the page has been modified pertain to the page, not to the individual processes; hence, this information should be recorded in the global page table. These observations lead to the following definitions for the global page table:

```
type gpagename;
     gpagedescr = record ssp : ↑page
                         ; case pc : [0..]
                           of [1..] : (psp : ↑page
                                       ; m : bool
                                       )
                           end
                  end;
     gpagetable  = array gpagename of gpagedescr;
var  gt : gpagetable;
```

[x..] denotes $\{i \mid x \leq i\}$.

Each entry of each local page table must at least contain the global page name of the page to which the entry corresponds and the boolean pr indicating whether the page is in the window of the corresponding process; the latter information is needed for local purposes such as the replacement algorithm. The design decision, taken above, that the way of access to a page in the window should be the same for both private and shared pages implies that the global page table may not be involved with such accesses. Hence, the local page table must contain all information needed to perform an access to pages in the process's window. So, the local page table must

contain the primary store address of the page and must contain a boolean m in which the fact that the page has been modified can be recorded. Thus we obtain the following definition for local page descriptors of shared pages:

```
type pagedescriptor = record gp : gpagename
                           ; case pr : bool
                             of true : (psp : ↑page
                                      ;  m : bool
                                      )
                             end
                    end;
```

(**Note:** actually, the definitions given -- here and in section 3 -- for pagedescriptor should be combined into one single definition; the two definitions having very much in common this is easy. We leave this as an exercise to the interested reader.)

The relation between the information in the local page tables and the global page table can now be formulated in the form of a few additional invariants. For this purpose we must be able to identify the local variables of different processes. In what follows the dummy X denotes a process and for any name v the local entity of process X with that name is denoted by X•v .

P4: (A X,i,j: i,j ∈ X•pagename : i = j ∨ X•pt[i]•gp ≠ X•pt[j]•gp)
P5: (A X,i: i ∈ X•pagename : X•pt[i]•psp = gt[X•pt[i]•gp]•psp)
P6: (A j: j ∈ gpagename :
gt[j]•pc = (N X : : (E i: i ∈ X•pagename : X•pt[i]•gp = j ∧ X•pt[i]•pr))
)

P4 expresses that, within each process, no shared page occurs under more than one local name. P5 expresses that the field psp in the local page table is a copy of the corresponding field in the global page table. In order that this duplication of information does not introduce gross inefficiencies the primary store addresses of pages better remain constant during their presence. Hence, reallocation of primary store with respect to present pages must be avoided. P6 formally expresses that, for all j , gt[j]•pc is the number of windows in which page j is present.

The relation between the local fields m and the global field m has to be chosen very carefully in order to meet our requirements. The obvious suggestion, namely to let, for each page, the global m be the disjunction of all the relevant local m's , is no good because truthifying a local m then requires an access to the global page table in order to also truthify the corresponding global m . Apparently, a weaker relation is needed. The following approach seems viable. In its local m's each process records which of the pages in its window have been modified by the process. As soon as a page is removed from the process's window the value of the page's local m is incorporated into the page's global m . For this purpose the following invariant, which replaces the former P2 , will do.

P2a: $(\forall j: j \in gpagename : gt[j]\cdot pc = 0 \vee gt[j]\cdot m \vee (\exists X,i : X\cdot pt[i]\cdot gp = j : X\cdot pt[i]\cdot pr \wedge X\cdot pt[i]\cdot m) \vee gt[j]\cdot psp\uparrow = gt[j]\cdot ssp\uparrow)$

We now have all that is needed to construct the following programs. The design is such that, as required, the code for read and write needs no changes. The code for getpage needs only to be changed in two places. For the sake of clarity, these two parts of getpage are confined to two new procedures, get and put , such that the new design differs from the old one only in the construction of get and put . In other words, all aspects of the design that pertain to page sharing are confined to get and put . The programs given here pertain to shared pages only; again, implementation of the case analysis to distinguish private and shared pages is left to the reader.

```
proc getpage(?p: pagename) =
|[ do ¬pt[p]·pr → do w = W → |[var q : pagename;
                                victim(q)
                              ; put(q)
                              ]|
                 od
               ; get(p)
  od
]|;
```

```
proc get(?p:pagename) =
|[var g : gpagename;
  pt[p]•pr := true  ;  w := w + 1
; g := pt[p]•gp
; with gt[g]
  do ≪ if pc = 0 → pc := 1  ;  acquire(psp)  ;  psp↑ := ssp↑  ;  m := false
       [] pc > 0 → pc := pc + 1
       fi
     ; pt[p]•psp := psp  ;  pt[p]•m := false
     ≫ { pc > 0 }
  od
]|;
proc put(?q:pagename) =
|[var g : gpagename;
  g := pt[q]•gp
; with gt[g]
  do ≪ { pt[q]•pr, hence: pc > 0 }
       m := m ∨ pt[q]•m  ;  pt[q]•m := false
     ; if pc = 1 → do m → ssp↑ := psp↑  ;  m := false od
                 ; release(psp)  ;  pc := 0
       [] pc > 1 → pc := pc - 1
       fi
     ≫
  od
; pt[q]•pr := false  ;  w := w - 1
]|.
```

The funny brackets ≪ and ≫ indicate that the enclosed pieces of text must be considered as atomic actions. A pagedescriptor in the global page table may be accessed simultaneously by different processes; in order to avoid interference between such accesses it suffices to subject them to *mutual exclusion*. Notice, however, that pagedescriptors corresponding to different pages are disjoint; hence, accesses to different entries of the global page table never interfere. Therefore, mutual exclusion is only needed for accesses to the *same* page table entry. This can be implemented by means of one binary semaphore per pagedescriptor. Because the copying of a page from or to secondary store now occurs within one of the

atomic actions, completion of these actions can take quite some time. With respect to the suggested solution -- one semaphore per page table entry -- this causes no problems; the alternative solution in which all accesses to entries -- the same or not -- of the global page table are performed under mutual exclusion must however be rejected.
(**Note:** it probably is possible, using the technique of the split binary semaphore, to obtain from the above solution a solution in which one binary semaphore per *page frame* plus one global binary semaphore are sufficient.)

6. Afterthoughts

Although we have followed a rather informal, operational pattern of reasoning we have been able to capture most of the essential requirements in strictly formal form: the invariants P0 through P6 are formal specifications of the allowable system states. These invariants restrict the number of programs that meaningfully can be written rather drastically; thus, they provide rather strong heuristic guidance for the construction of the actual code. Within the latter process operational considerations have hardly played a role; in this respect the invariants also provide -- at least to a large extent -- the interface between the operational interpretations and the formal code of the programs. Although I tend to be rather satisfied with the result I think that a less operational treatment, relying more on strict formula manipulation must be possible; in this respect the above may be considered as one, but only one, step forward.

This and other recent experiments show that the technique of starting as abstractly as possible and then taking gradually more detail into account lends itself very well for use with formal techniques; it might very well turn out to be the only way in which we will be able to keep the formulae involved manageable. The above design pleases me very much in two respects. Firstly, this is the first time that I have been able to code the access procedures for a virtual store in such a way that I am rather confident of their correctness. Secondly, the incorporation of page sharing went surprisingly smoothly, much more smoothly than I expected. Notice that we have introduced page sharing starting with the rather detailed version obtained in section 3 . It probably would have been better to take the more abstract version from section 2 for our starting point, thus postponing the choice of actual

representations until after the treatment of page sharing.

We conclude this section -- and this study -- with a few minor remarks. Firstly, the use of *variant parts* in record definitions has been more annoying than useful. As a consequence, statements in which references to, for instance, pt[p]•psp occur must have pt[p]•pr as a precondition; this requirement restricts, rather drastically, the order in which such statements may be written down. I have tried to construct the code of the programs in such a way that all these restrictions are respected; the order in which the statements now occur in the code is not always the order I would have chosen otherwise. Secondly, I am not satisfied with the treatment of the modification booleans and the associated invariant P2a ; although the present version seems to be sound it took me quite some time to find it. Moreover, I think that something is wrong with the whole approach. The question whether a page has been modified only pertains to pages present in primary store. It seems wiser to associate the variable recording this with the *page frame* instead of the *page* itself. This means that we have one boolean variable per page frame; then it is unnecessary to keep a complicated distributed administration of page modifications. Some hardware support here is both easy to implement and easy to use.

Eindhoven, 1987.5.26
Rob Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology