A STRUCTURED PAGING SYSTEM

C. A. R. Hoare

Summary. This paper attempts to extend the methods of structured programming to the design and description of parallel algorithms, in particular a paging system. It is argued that the proposed algorithm, though rather simple, may well be adequate in a variety of hardware and operating system environments.

1. Introduction.

The principles and practices of structured programming have been expounded and illustrated by relatively small examples (Dahl, Dijkstra, Hoare, 1972). Systematic methods for the construction of parallel algorithms have also been suggested (Dijkstra, 1968, a,b). This paper attempts to apply structured programming methods to a program intended to operate in a parallel environment, namely a paging system for the implementation of virtual store. The design decisions are motivated by considerations of cost and effectiveness, and it is argued that the resulting design, though rather simple, may well be adequate in a variety of hardware and operating system environments.

The notations used in the paper are based on those of PASCAL (Wirth 1971) and SIMULA 67 (Dahl, 1972). However, the reader is warned not to expect that any automatic compiler will be written to translate these notations into machine code - that must almost certainly be done by hand.

The purpose of a paging system is taken to be the sharing of main and backing store of a computer among a number of users making unpredictable demands upon them; and to do so in such a way that each user will not be concerned whether his information is stored at any given time on main or backing store. For the sake of definiteness, the backing store is taken here to be a sectored drum; but the system could readily be adapted for other devices. Our design does not rely on any particular paging hardware, and it should be implementable in any reasonable combination of hardware and software. Furthermore, it does not presuppose any particular structure of virtual store (linear, two-dimensional, "cactus", etc.) provided to the user program.

2. The Hardware.

A paging system is essentially concerned with the physical storage devices of a particular computer installation, and if it is to be run on many installations, some degree of abstraction must be used in referring to the hardware. For example, symbolic constants

must be used to refer to installation parameters such as

M - the number of pages that will fit in main store
D - the number of pages that will fit on the drum
L - the number of words on a page

We can now define certain ranges for variables, using the PASCAL range definitions, for example:

```
type mainpageframe  =  0..M-1;
type drumpageframe  =  0..D-1;
type line           =  0..L-1;
```

A page may now be regarded as an array of words, where a word has to be defined in a machine dependent fashion:

```
type  word =
type  page = array line of word;
```

Furthermore, the physical main and backing store may be regarded abstractly as arrays of pages:

```
mainstore:array mainpageframe of page;
drumstore:array drumpageframe of page;
```

Individual pages of these stores will be denoted by single sub script:

```
mainstore [m],  drumstore [d]
```

and individual words of mainstore by double subscript

```
mainstore[m,i].
```

Of course, the individual word drumstore[d,i] can never actually be referred to in our program, since a drum provides access only to complete pages.

3. Dynamic Storage Allocation

It is evident that a paging system is concerned with the dynamic allocation and deallocation of pageframes to processes. We shall therefore require a dynamic storage allocation system for pageframes. In this section we describe an allocator for mainpageframes; an allocator for drumpageframes is sufficiently similar that it need not be separately described. The most important data item in any resource allocator is the pool of currently free items of resource. This can be declared:

```
pool : powerset mainpageframe;
```

We then use operations

```
m:=anyone of(pool); pool:=pool-[m];
```

to acquire a free pageframe m, and an operation:

```
pool:=pool v[m];
```

to return m to the free pool.

However, in a parallel programming environment there are two additional problems, namely exclusion and synchronisation.

3.1. Mutual Exclusion.

Suppose that one program calls the function anyone of(pool) just after another one has completed the same function. It will then obtain the same value. This will frustrate the whole purpose of the allocator, which is to prevent duplicate use of the same pageframe by two different user programs. A group of operations which must not be executed reentrantly by several programs is known as a critical region.

The solution we adopt is to introduce a program structure known as a monitor. A monitor consists of one or more items of data, together with one or more procedures operating on that data. Thus a monitor is very like an object in SIMULA 67 (Dahl, 1972), with the additional understanding that the bodies of the procedures local to the object will not be executed in parallel or interleaved with each other.

If the only access to variables local to a monitor is through calls of procedures local to that same monitor, the user programmer can be protected against most forms of timedepen dent coding error.

In implementation, the necessary exclusion can be assured by associating a Boolean mutual exclusion semaphore (Dijkstra, 1968) with each monitor and by surrounding each call on a procedure of the monitor by a P and V on this semaphore. Alternatively, in suitable cases, the required effect can by achieved by inhibiting interrupts during execution of the procedure bodies. Since details of implementation are very hardware-dependent, we shall introduce a high-level notation for declaration of monitors:

```
monitor   mfree;
   begin pool : powerset mainpageframe;
        function acquire:mainpageframe;
           begin .. body of acquire .. end;
        procedure release(m:mainpageframe);
           begin .. body of release .. end;
        pool:= all mainpageframes;
   end mfree;
```

Calls on the two procedures will be written using the PASCAL and SIMULA 67 notation for components of a structure:

```
m:=mfree.acquire;
mfree.release(m).
```

The monitor concept described here corresponds to the secretary mentioned at the end of (Dijkstra, 1973).

3.2. Synchronisation.

Any resource allocator has to face the problem of exhaustion of its resource. In a single-programming environment, any further request for that resource will be refused, and often lead to immediate termination of the program; but if there are many processes, it is more reasonable merely to postpone the request until some

other process kindly releases an item of the resource. Of course, if this never happens, we have deadlock; but this will be averted by a preemptive technique to be described later. What is required is some method whereby one process, on detecting that it cannot proceed, can wait until some other process brings about the condition that will enable it to proceed.

Methods of obtaining such synchronisation are rather machine-dependent, so again we shall introduce a high-level notation, which can probably be implemented on many machines with reasonable efficiency. For each condition on which a program may have to wait, we introduce a variable of type "condition", for example:

```
nonempty:condition.
```

There are only two operations defined on a condition variable, and they are waiting and signalling. When a process needs to wait until some condition becomes true, it "waits on" the corresponding condition variable, for example:

```
nonempty.wait.
```

When some other process has made the condition true, it should issue a signal instruction on thatvariable:

```
nonempty.signal.
```

If there are no processes waiting on the condition, the instruction has no effect; otherwise it terminates the wait of the longest waiting process, and only that process. (The convention that at most one process shall proceed after a signal differs from the familiar "event" type signalling (Brinch Hansen, 1972); but it appears, at least in the cases considered here, to be both more convenient and more efficient).

It is important to clarify the relationship between synchronisation and mutual exclusion. When a wait instruction is given from within a monitor, the mutual exclusion for that monitor is assumed to be released, since otherwise it would be impossible for any other process to enter a procedure of that monitor to make the condition true. Furthermore, when a wait ends as a result of a signal, the waiting program resumes immediately in place of the signalling program, and the signalling program resumes only when the

previously waiting program releases exclusion again. Thus exclusion is not released between the signal and the resumption of the waiting process; since if it were there would be a risk that some other process might enter the monitor during the interval and make the condition false again.

Using the monitor concept for mutual exclusion and the condition variable for synchronisation, the programming of a simple allocator for main page frames can be given:

```
monitor mfree;
   begin pool : powerset mainpageframe;
         nonempty:condition;
   function acquire : mainpageframe;
     begin  m:mainpageframe;
            if pool = empty then nonempty.wait;
            m:=anyone of ( pool );
            pool := pool - [m];
            acquire:=m
     end acquire;
   procedure release (m:mainpageframe);
     begin pool:=pool v[m];
           nonempty.signal
     end;
   pool:=all mainpageframes;  note initial value of pool;

end mfree.
```

Of course, the implementation of such a simple resource allocator presents little interest in itself. It has been treated here at such length merely in order to introduce and illustrate the use of monitors and condition variables, which are the basic program structuring methods which we shall require for the rest of the paging system design.

4. Drum transfers.

It is evident that a paging system will also be concerned with input and output of pages between main and backing store. This will usually be effected by special hardware (a channel) which operates in parallel with all user programs, except possibly for the one which is waiting for completion of transfer. We shall therefore require a scheduler to prevent interference between transfer instructions given by separate programs, and to secure the necessary synchronisation between the programs and the asynchronous channel. As before we shall use monitors and conditions; and to begin with we shall assume that the drum is capable of only one transfer per revolution.

The monitor for drum transfers will obviously have to store details of any outstanding transfer command; that is, its direction, and the relevant main and drum page frames. If there is no outstanding command, we use the convention that the direction is "null":

```
direction:(in,out,null);
m:mainpage frame;
d:drumpageframe.
```

The operations provided by the monitor to the user are

```
input(dest:mainpageframe; source:drumpageframe);
```

and output(source:mainpageframe; dest:drumpageframe).

These operations merely record the outstanding command; its execution is accomplished by an operation

```
execute,
```

which will be written here as part of the monitor, but will in practice be implemented by the hardware of the drum channel, and perhaps by its intimate interrupt routines.

There are two possible reasons for waiting by a user program; firstly, when there is already an outstanding command and therefore a new one cannot be recorded, and secondly when its command has

been recorded but has not yet been finished. Thus we introduce two condition variables

```
free, finished:condition
```

The drum itself never waits; if there is nothing for it to do, it idles for one revolution.

The programming of the monitor is now simple.

```
monitor  drummer;
  begin  direction:(in,out,null);
         m:mainpageframe;
         d:drumpageframe;
         free,finished:condition;
  procedure input(dest:mainpageframe; source:drumpageframe);
    begin  if  direction ≠ null then free.wait;
           direction:=in;
           m:=dest;
           d:=source;
           finished.wait;   free.signal
    end input;
  procedure  output  ... very similar ...;
  procedure  execute;
    begin
    case direction of
            in:mainstore[m]:=drumstore[d],
           out:drumstore[d]:=mainstore[m],
           null:do nothing for one revolution;
    direction:=null; finished.signal  free.signal
   end  execute;
    direction:=null
end  drummer
```

The activity of the hardware of the drum channel may be described as a process:

```
process drum hardware;
    while true do drummer.execute;
```

Suppose now that several drums are available, and capable of simultaneous transfers. It is very important to have a separate monitor for each drum channel, so that simulteneity of transfers is not inhibited by the exclusion of the monitor. Since the monitors are identical it would be unfortunate to have to write out the whole text several times; instead we borrow the idea of a SIMULA 67 class (Dahl, 1972) and declare each separate monitor as a different object of the class, each with its own workspace

```
class drummer;
    monitor begin  ... as before ....
            end;
```

We then declare several "instances" of this class:

```
drum 1, drum 2:drum.
```

Thus a class declaration is very similar to a PASCAL record type declaration with the addition of procedures as well as data components.

On most modern drums the number S of pages (sectors) that can in principle be transferred in a single revolution is considerably greater than one. Such a sectored drum may be treated in the same way as S separate drums, each capable of one transfer per revolution. We therefore introduce a whole array of monitors, one for each sector:

```
sector drum : array 0..S-1 of drummer.
```

The action of the drum channel hardware may be described as an autonomous process:

```
process sector drum hardware;
        while true do
            for s = 0 to S-1 do sectordrum[s].execute.
```

Let "sectorof" be a function giving the sector of a drum page frame. Since the user is not interested in the sector structure of the drum, we provide the simple procedures:

```
procedure input (dest:mainpageframe; source:drumpageframe);
        sectordrum[sectorof(source)].input(dest,source);
procedure output  .... very similar ....
```

For convenience, we use the same identifiers "input" and "output" for procedures local to the monitors, and for global procedures, which are not inside a monitor, and which can therefore be shared in reentrant fashion by all user programs.

Using these procedures, the programmer may maintain the illusion that all drum transfer instructions relating to different page frames are carried out in parallel with each other, since any necessary exclusion and synchronisation is carried out behind the scenes.

The scheduler described in this section, like that of the previous section, displays little sophistication. Nevertheless, it illustrates again the use of monitors and conditions, and introduces the important new concept of a class of monitors, of which separate instances can be declared separately; and it shows how this concept can be used to set up an array of monitors, where each monitor can operate independently and in parallel with the others. This structure will be used again the design of the paging system, where operations on individual pages must appear to take place in parallel with each other.

The efficient use of a sectored drum requires that the drumpage-frames in use be reasonably evenly distributed over the sectors. One method of achieving this is that the monitor which allocates drumpageframes should have a separate pool for each sector:

```
pool : array 0..S-1 of powerset 0.T-1,
```

where T is the number of tracks on the drum,(i.e. T = D÷S). The allocator should also maintain a cyclic pointer

```
s : 0..S-1,
```

which points to the sector on which the most recent drumpageframe

was allocated. The next allocation should be on the nearest possible following sector. If output instructions closely follow acquisition (which they will), this will ensure that any rapid sequence of outputs will be timeshared to the maximum possible extent.

The design of a monitor for allocation of drum page frames should be undertaken as an exercise by the interested reader.

5. Virtual Store.

A virtual store, like the actual stores, can be regarded as an array which maps virtual page frames onto virtual pages.

virtualstore:array virtual pageframe of virtual page;

where virtual page is a class to be defined in the next section. The concept of a virtual page frame may be simply defined as a range.

type virtual page frame = 0..V-1;

where V is a constant several times larger than D. This will give a single-dimensional structure of virtual store. In some operating systems, a two-dimensional structure is preferred, in which case the virtual page frame can be described as a record:

type virtual page frame = record s:segmentno;p:page no end.

In this case the virtual store array may well be implemented as a tree structure, so that no space is wasted on storing unused pages at the end of segments. But the paging system designed in this paper will be equally valid for both these cases, and perhaps many others.

As far as the user program is concerned, the most important operations are

1. a function fetch (i:virtual address):word, which delivers the content of the i^{th}"location" of virtual store.
2. a procedure assign (i:virtual address; w:word) which stores the value w in the i^{th} "location" of virtual store.

A virtual address is defined as a pair:

```
type virtual address = record p:virtual page frame;
                              ℓ:line
                       end.
```

These two procedures can be implemented using the same structure as input and output on a sectored drum, by calling on procedures of the same name local to the relevant virtual page.

```
function fetch (i:virtual address):word;
        fetch:= virtualstore[i.p].fetch(i.ℓ);

procedure assign(i:virtual address; w:word);
        virtualstore[i.p].assign(i.ℓ,w).
```

Note that these procedures are not protected b y mutual exclusion, so that when one program is held up waiting for a page transfer, the other programs can continue to operate on other pages.

6. Virtual pages.

In this section we implement the virtual page class which was introduced in the previous section. This class must obviously provide the user program with the procedures fetch and assign; in addition it contains:

(1) procedure bring in;

This has no effect on the apparent content of the page, but merely ensures that it is located in main store, where its individual words can be accessed.

(2) procedure throw out;

This also has no effect on the apparent content of the page; but it ensures that the page no longer occupies a main page frame.

The data local to each virtual page must include an indication

of whether the content of the page is currently held in main or drum store; and we also admit a third possibility, that the content of the page is equal to an "all clear" value, in which case no actual storage is allocated to it. This is the value to which each page is initialised. The required indication is given in a variable local to each virtual page

where:(in,out,clear).

The physical location of the content of each page will be recorded in one of the two local variables:

```
m:mainpageframe;
d:drumpageframe;
```

In programming a paging system, it is essential to ensure that no two virtual pages ever point to the same actual page frame, and in particular, that they never point to a free pageframe; in other words, every operation of the virtual page must preserve the truth of:

(1) where = in $\supset$ m $\tilde{\in}$ mfree.pool

(2) where = out $\supset$ d $\tilde{\in}$ dfree.pool,

and further, the only operations on m and d are acquiring and releasing them to their respedtive pools.

```
      class virtual page;
monitor begin where:(in,out,clear);
                m:mainpageframe;
                d:drumpageframe;

      note where = in ⊃ m ∈ mfree.pool & where = out ⊃ d ∉ dfree.pool;
      procedure bring in; note ensures where = in;
         if where ≠ in then
           begin m:=mfree.acquire;
                 if where = clear then mainstore[m]:=allclear
                                  else {input(m,d); dfree.release(d)};
                 where:=in;
           end;

      procedure  throwout;   note  ensures  where ≠ in;
         if where = in then
           begin d:=dfree.acquire;
                 output(m,d);
                 mfree.release(m);
                 where:=out
           end throwout;

function fetch(ℓ:line):word; note  =  content virtual page[ℓ]
           begin bring in;
                 fetch:=mainstore[m,ℓ]

procedure  assign (ℓ:line; w:word);   note content virtual page[ℓ]:=x;
           begin bring in;
                 mainstore[m,ℓ]:=w
           end;

   where:=clear ; note  initial value of each virtual page is all clear;

end virtual page.
```

7. Automatic Discard.

The most characteristic feature of a paging system is that pages can throw themselves out automatically to drum, and release the main page frame they occupy, independently of the programs which may or may not be using them. Of course the rate at which pages throw themselves out must be regulated in such a way that the output and subsequent input do not overload the drum channel. Since the rate of input cannot persistently exceed the rate of output, it is sufficient to control the latter. In order to ensure that the average output rate does not exceed one page per drum revolution, it is sufficient to ensure that each page remains in mainstore for an average of M drum revolutions after it has been input, where M is the number of pages in mainstore.

Since each page is capable of independent activity, it is necessary to associate a process with each page, which is responsible for throwing the page out after it has been in store for long enough. This process will be called

process automatic discard;

It is invoked when the page is first brought into mainstore, and is then supposed to proceed "in parallel" with the user program. However, the first thing the process does is to wait for M drum revolutions; and it is assumed that exclusion is released during this wait, so that other user programs may invoke the other operations on the page during the interval. Of course, at the end of the wait, the exclusion is seized again while the page is actually being thrown out.

The process is easily coded:

```
process automatic discard;
    begin wait about M drum revolutions;
          throw out
    end
```

It remains to design an efficient implementation of the wait; and we should try to ensure that it is impossible for pages to get into synchronisation, and attempt to throw themselves out simultaneously. Our solution assumes the existence of a procedure which will wait at least one drum revolution.

With each main page frame we associate a condition on which the automatic discarder waits whenever it needs to delay for M revolutions

delay: array mainpageframe of condition.

We also introduce a process known as the cyclic discarder which is responsible for signalling each condition at an interval of approximately M drum revolutions. This can be accomplished by a simple loop:

```
process cyclic discarder;
      while true do
            for m = 0 to M-1 do
                begin delay[m].signal;
                    wait at least one drum revolution.
                end;
```

An important point to note is that the cyclic discarder has been carefully designed so that it is never held up. If, for example, it attempted to make a direct entry to throw out, it might be held up by normal exclusion during a bring in operation on the same page. If that bring in were itself held up waiting for a free mainpageframe, a deadly embrace would result.

8. Load control.

The major defect of the system described above is that it reacts badly to an overload of the mainstore, which occurs when the rate at which programs wish to acquire page frames is consistently greater than the rate at which they are being released. In these circumstances, some or all of the programs running may spend nearly

all their time waiting to acquire a main page frame; and by the time they get it, they may well find that one of their own pages has been automatically discarded, and a new main page frame must be acquired again immediately.

The only effective solution in these circumstances is to reduce the requirement for free page frames or to increase the supply. Both may be accomplished by suspending one of the programs currently under execution, since this program will no longer be able to acquire pages, and the pages which it currently possesses will be thrown out in due course by the automatic discarder. If there is only one program under execution, and it is spending most of its time waiting for main page frames, it too should be discontinued, since here the only solution is to redesign the program or to buy more mainstore.

After a program has been suspended, it would obviously be foolish to suspend another program until all its pages have been discarded. This suggests that the suspension should be carried out (when necessary) at the end of each scan of the store by the cyclic discarder.

When a program has been suspended, it is necessary to decide when it is to be resumed. The simplest procedure is to resume a program at regular intervals, irrespective of whether there is room for it in mainstore, and to suspend some other program if there turns out not to be. The rate of resumption should be adjusted to ensure that the overhead involved in the suspension/resumption cycle is acceptable.

The one remaining question is which program to select for suspension or resumption? The simplest answer seems the best: Suspend the program which has been longest unsuspended, and resume the program which has been longest suspended. The one exception is that a program which is waiting for a response from a multiple access terminal should not be suspended during its wait, nor for a reasonable time afterwards. But the details of this will not be further treated here.

The cyclic discarder takes the form:

```
process  cyclic discarder;
     while true do
     begin for i = 0..19 do
           begin for m = 0 to M-1 do
                 begin delay[m].signal;
                     wait at least one drum revolution
                 end;
                 if size (mfree) <3 then suspend a program
           end;
           resume longest suspended program
   end cyclic discarder.
```

An alternative ~~solution~~ to loadshedding is to allow the rate of automatic discard to increase when there are no free main page frames. This unfortunately leads to a phenomenon known as thrashing, in which pages are discarded as fast as the programs using them can bring them back. This all too common elementary mistake in control engineering is avoided by ensuring that the maximum rate of discard remains less than the minimum rate of recall. After all the ink that has been spilled on this subject, it is very difficult to realise that the mistake and its solution are so simple.

9. Refinements.

This section discusses a number of detailed refinements, which either improve efficiency, or adapt the algorithm for more direct implementation of hardware.

9.1. On some occasions, the programmer is no longer interested in the current content of a page, for example, if it contains an array local to a block which he is about to exit. In this case, it is possible immediately to release both the main and drum page frames, and return the apparent content of the page to its initial clear state. This is accomplished by a call on a procedure local to each virtual page:

```
  procedure clear;
begin case where of
        in:begin where:=clear; delay[m]:=null;mfree.release(m)end,
        out:begin where:=clear; dfree.release(d)end,
          clear:do nothing;
end.
```

Note the instruction delay[m]:=null is intended to cancel the wait of the automatic discarder.

9.2. Prefetching.

On certain occasions a user program knows that it will shortly require several pages to be in mainstore, for example, a complete array or a complete compiler. If these pages are brought in only when first referenced, each page will involve a delay of up to a full drum revolution. However, if the program gives advance warning of its requirement, perhaps the whole set of pages could be brought in within one or two revolutions. To achieve this, the programmer should be able to issue a series of "prefetch" instructions, and have them executed "in parallel" with each other, and perhaps even with continuation of his own program. Prefetching may also be used to speed initial program loading, or the resumption of a program suspended by loadshed.

This facility can be implemented simply by declaring a process with a body consisting of a single instruction

```
process prefetch;
      bring in
```

Note that the operation bringin does not contain a wait, and therefore exclusion of the virtual page is set until the operation is complete - which will be quite soon if the page is already in. If the user program attempts to access the page before the process has been completed, it will be held up in the normal way by the exclusion mechanism of the monitor. This means that there is no risk whatsoever that the invocation of a process local to a monitor can lead to time-dependent errors in a user program.

The coding of the prefetch process may not be as simple as it looks, since it is not acceptable that it should need any workspace local to the virtual page. It may therefore be necessary to associate local administrative workspace of this process with the mainpageframe, rather than the virtual page; in which case the mainpageframe must be acquired before process can proceed in parallel.

9.3. Unchanged.

When a page is used to store code, it is very likely that the content of the page remains unchanged for long periods. The same will be true of certain data pages as well. This means that when the time comes to throw out the page, the copy which already exists on the drum is still valid, and the output instruction can be avoided, provided that the relevant drumpageframe has not been ~~returned to dfree.~~ released

We therefore introduce for each virtual page a variable

unchanged:Boolean

which is set true after input of each page and set false by the first assignment to that page. Each operation of the paging system must preserve the truth of

$$\text{where} = \text{in} \;\&\; \text{unchanged} \supset \text{mainstore}[m] = \text{drumstore}[d] \;\&\; d \in \text{dfree}$$

9.4. Locking.

When a page is engaged in an autonomous peripheral transfer, it is usually necessary to ensure that the page is not thrown out until the transfer is complete. Similarly, considerations of efficiency sometimes dictate that a copy of the absolute mainstore address be made, for example in an associative slave store; and the page must not be thrown out while this address is in use. We therefore provide a function

function lock(v:virtual page frame):main page frame;

which "locks" the virtual page into mainstore, and delivers its address as result. The effect may be reversed by calling

procedure unlock(v:virtual page frame).

If a virtual page is being shared among several programs, and more than one of them lock it, it is essential that the page remains locked until they have all unlocked it. It is therefore necessary to introduce for each mainpageframe a count of the number of times it has been locked:

```
lockcount:array mainpageframe of integer;
```

which is incremented by locking and decremented by unlocking.

9.5. Usebit.

When a page automatically discards itself, it will frequently happen that this page will be accessed again almost immediately; and in this case an unnecessary delay of up to two revolutions has been intruded upon at least one user program. An obvious symptom that a page is going to be used again in the near future is that it has been used in the recent past. We therefore associate with each virtual page a

```
usebit:Boolean
```

which is set to true by bringin in every fetch or assign instruction, and is set false by the automatic discard process at regular intervals. Thus the page will be thrown out only if it remains unaccessed throughout the interval.

```
process automatic discard###;
    begin delay[m].wait;
        while usebit v lockcount>0 do
         begin usebit:=false;
               delay[m].wait
         end;
        throw out.
    end;
```

Since at least two delays are involved before a page automatically discards itself, the rate at which the cyclic discarder scans the store should be increased by a factor of two.

9.6. Explicit discard.

On occasions a programmer knows that he will not require to access a page again for a long period, for example it could be a completed page of an input or output file, or a piece of program that has been finished with. In this case it is kind to inform the paging system of this fact, so that the page may be sooner thrown out. This may be done by calling:

```
procedure discardnow(v:virtual page).
```

But if there is any risk at all that the page is being shared by some other program, or that it may after all be required back on mainstore sooner than expected, it would be better to use a milder suggestion:

```
procedure discardsoon(v:virtual page).
```

The implementation of these is trivial

```
discardnow:
   if where = in then
    begin usebit:=false
        delay[m].signal
    end


discardsoon:
   if where = in then usebit:=false
```

The explicit discard can also be used by the loadshedding mechanism, and before each interaction of a multiple access program, provided that the pages in private use by each program are known.

10. The complete program.

This section describes the complete paging system, with all its refinements. The reader may study it to check compatibility of the refinements, or to confirm his understanding. Alternatively, he may consult it for reference or even omit it altogether.

```
class virtual page;
monitor   begin where:(in,out,clear);
                    m:mainpageframe;
                    d:drumpageframe;
            unchanged,usebit:Boolean;

note where = in ⊃ m∈mfree & content = mainstore[m],
     where = out ∨ where=in & unchanged  d∈dfree & content=drumstore[d],
     otherwise content = all clear.


procedure bringin; note ensures where = in, content unchanged;
begin usebit:=true;
     if where ≠ in then
     begin m:=mfree.acquire;
           if where=clear then begin mainstore[m]:=all clear;
                                     unchanged:=false
                               end
                          else begin input(m,d);
                                     unchanged:=true
                               end;
           where:=in;
           automatic discard
      end
end bringin;
```

```
  procedure throw out; note ensures where ≠ in, content unchanged;
      if where = in then
         begin if ¬ unchanged then begin d:=dfree.acquire;
                                         output(m,d)
                                    end;
               mfree.release(m);
               where:=out
         end throwout;

  process  automatic discard; note no change;
begin  delay[m].wait;
       repeat  usebit:=false;
               delay[m].wait
       until  lowkcount[m] >0 & ¬ usebit;
       throwout
end automatic discard;


  procedure clear;  note content:=all clear;
begin case where of
      in: begin delay[m]:=null;
                if unchanged then dfree.release(d);
                mfree.release(m)
          end,

      out:  dfree.release(d);
end clear;
```

where:= clear!

```
    function fetch (ℓ:line):word; note = content[ℓ];
      begin bringin; fetch:=mainstore[m,ℓ]end;

    procedure assign(ℓ:line; w:word); note content[ℓ]:=w;
      begin bringin; mainstore[m,ℓ]:=w;
            if unchanged then begin dfree.release(d);
                                    unchanged:=false
                              end
      end;

    process prefetch;
         bringin ;

    function lock:mainpageframe;
         begin bringin; lockcount[m]:=lockcount[m] +1; lock:=m end

    procedure unlock;
         lockcount[m]:=lockcount[m] -1;

    procedure  discard now;
       if where = in then
        begin  usebit:=false;
               delay[m].signal
        end;

    procedure discard soon; usebit:=false;

  where:=clear

end  virtual page;
```

11. Evaluation.

The quality of an operating system algorithm depends on its efficiency, reliability, and convenience. In this section we attempt to evaluate the system designed here, even without knowing details of hardware, operating system or workload. If such evaluation is not possible, this will reveal in what respects the system designed is machine-dependent, or is not truly general-purpose.

11.1. Efficiency.

The most important aspect of the efficiency of a paging system is the minimisation of the amount of space occupied in mainstore by the data and code of the system itself. Of course, a great deal depends here on the skill of the coders, particularly in implementing the monitor/condition/process features, but the general simplicity of the algorithm and the small amount of data involved suggest that the code will be short. It is also important that the data particular to a virtual page should fit into a single word, since the virtual store array, which must be locked in mainstore, may contain many entries. This seems to be achievable for most likely value of M and D. If it is not, d can be stored elsewhere during the periods in which the page is in mainstore. The main difficulty is likely to be the exclusion semaphore which protects each virtual page, since only a few bits can be spared for this.

A second aspect of efficiency is that the total time spent in the paging system is minimised. Here a great deal depends on the hardware design of an associative store which will enable the software of the paging system to be bypassed in all cases except where a drum transfer is involved. Since the rate of drum transfers is quite low, the time taken to execute code of the paging system is not critical, and since such operation involves a programmed loop, efficiency will be proportional to compactness of code. A further hopeful point is that the careful structuring of the system as a whole means that it is directly implementable on a genuinely multi-processing computer, even if each processor has its own associative store.

A third aspect of efficiency is the maximisation of the density of utilisation of main store. Of course, this depends mainly on the cooperation of the user programmers; but the system should be able to work adequately even without it. At least it must be free from persistently inadequate behaviour, such as "thrashing" or indefinite waiting for free pages. The cyclic discarder and load-shedding mechanism ensures that the system cannot spend more than a very small (and controllable) proportion of its time doing this. Although the algorithm will never make the "right" decision, it will never persistently make the wrong one.

11.2. Reliability.

An operating system module should be able to present the appearance of a machine more reliable than the bare hardware of a computer. The only way in which this paging system contributes to the simulation of reliability is that it can run on machines on which there are known bad patches in mainstore or on drum; this is accomplished by appropriate initialisation of the pools of mfree and dfree. Additional aspects of reliability which should be treated are:-

(1) If a location of drum or mainstore becomes unreadable, the relevant user program should be automatically restarted, and the relevant page withdrawn from use.

(2) Periodic dumps of pages to backing store (perhaps disc) should take place on suitable occasions, so that if the central processor fails, all user programs can be restarted when it is mended.

(3) Periodic dumps of pages from backing store onto a removable medium should take place on suitable occasions, so that if all or part of the backing store fails, users can recover their permanently stored material.

These functions should be carried out by other parts of the operating system (e.g., a filing system); it may be that their successful implementation will depend on making a few adaptations to the paging system designed here, for example, to control explicitly the backing store pageframe allocated to a page.

A further aspect of reliability is, of course, that the software be free from error, particularly from time-dependent error. It is hoped that the clear structuring of this paging system will contribute towards this objective.

11.3. Convenience.

A paging system is in itself a very convenient tool for the user programmer; it has only two entry points, with only one or two parameters; and will administer overlays without any explicit intervention from the user program. Nevertheless, it cannot pretend to plan overlays, and since drum access is some five orders of magnitude slower than main store access, it is obvious that some degree of cooperation from the user (or his language implementor) would be desirable. A good paging system should make this cooperation easy and natural.

(1) The most effective form of cooperation is that the programmer should place in physically contiguous locations of virtual store all the material that he is going to refer to during each phase of his program, where a phase is an interval measured preferably in seconds. This will minimise the number of pages that he needs in mainstore at a time. Programming languages of the future will have to be designed to assist in this planning.

(2) The second most important aspect of cooperation is that the user should clear those pages which he has finished using. This can be done automatically in a "stack" language like ALGOL 60.

(3) Thirdly, if the drum is being used for input and output files, pages which have been wholly read or written should be explicitly discarded. This can be done automatically by a filing system.

(4) The prefetch instruction is a rather dangerous one, since its inconsiderate use can clog the mainstore and drum channel; its use should be confined to the operating system itself and to "real time" applications programs; and an ordinary user should not be able to prefetch more than a few pages at a time.

(5) Of course, for pages being stored on disc or other slower backing store, the explicit cooperation of the programmer will be more necessary to avoid too frequent head movement.

(6) Finally, the paging system is here described with sufficient simplicity that it can actually be understood by its more serious users; and when a craftsman understands his tools he is likely to use them more successfully.

Conclusion.

A final evaluation of the quality of the system cannot be made until several implementations have proved successful in practice. But I hope that the way in which it has been designed, described and evaluated, will motivate others to undertake such implementation; at least their program documentation problems will have been largely solved.

Since nothing is new about this algorithm (except possibly its structure), acknowledgement is due to M. Melliar Smith for the elegance of the cyclic discarder, E.W. Dijkstra for central concepts of structure, exclusion, and synchronisation, and to these, together with R.M. McKeag, J. Bezivin, and P. Brinch Hansen, for ideas, discussion, inspiration, and criticism on points too numerous for me even to remember.

References

BRINCH HANSEN, P. (1972). Structured Multiprogramming, Comm. ACM., 15, 7, 574-578.

DAHL, O-J., DIJKSTRA, E.W., HOARE, C.A.R. (1972). Structured Programming, Academic Press, 1972.

DAHL, O-J. (1972). Hierarchical Program Structures. (in the above).

DIJKSTRA, E.W. (1968a). Cooperating Sequential Processes, in Programming Languages, ed. F. Genuys, Academic Press 1968.

DIJKSTRA, E.W. (1968b). A constructive approach to the problem of program correctness. BIT 8, 174-186 (1968).

DIJKSTRA, E.W. (1973). Hierarchical Ordering of Sequential Processes, in Operating Systems Techniques, ed. C.A.R. Hoare and R.H.Perrott, Academic Press.

WIRTH, N. ( ). The programming language PASCAL, Acta Informatica. 1, 1, 35-63