Monitors: An Operating System Structuring Concept

C.A.R. Hoare
The Queen's University of Belfast

This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

Key Words and Phrases: monitors, operating systems, scheduling, mutual exclusion, synchronization, system implementation languages, structured multiprogramming
CR Categories: 4.31, 4.22

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This paper is based on an address delivered to IRIA, France, May 11, 1973. Author's address: Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland.

1. Introduction

A primary aim of an operating system is to share a computer installation among many programs making unpredictable demands upon its resources. A primary task of its designer is therefore to construct resource allocation (or scheduling) algorithms for resources of various kinds (main store, drum store, magnetic tape handlers, consoles, etc.). In order to simplify his task, he should try to construct separate schedulers for each class of resource. Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. Such a collection of associated data and procedures is known as a *monitor*; and a suitable notation can be based on the *class* notation of SIMULA67 [6].

```
monitorname: monitor
  begin ... declarations of data local to the monitor;
    procedure procname (... formal parameters ...);
      begin ... procedure body ... end;
    ... declarations of other procedures local to the monitor;
    ... initialization of local data of the monitor ...
  end;
```

Note that the procedure bodies may have local data, in the normal way.

In order to call a procedure of a monitor, it is necessary to give the name of the monitor as well as the name of the desired procedure, separating them by a dot:

```
monitorname.procname(... actual parameters ...);
```

In an operating system it is sometimes desirable to declare several monitors with identical structure and behavior, for example to schedule two similar resources. In such cases, the declaration shown above will be preceded by the word **class**, and the separate monitors will be declared to belong to this class:

```
monitor 1, monitor 2: classname;
```

Thus the structure of a class of monitors is identical to that described for a data representation in [13], except for addition of the basic word *monitor*. Brinch-Hansen uses the word *shared* for the same purpose [3].

The procedures of a monitor are common to all running programs, in the sense that any program may at any time attempt to call such a procedure. However, it is essential that only one program at a time actually succeed in entering a monitor procedure, and any subsequent call must be held up until the previous call has been completed. Otherwise, if two procedure bodies were in simultaneous execution, the effects on the local variables of the monitor could be chaotic. The proce-

dures local to a monitor should not access any nonlocal variables other than those local to the same monitor, and these variables of the monitor should be inaccessible from outside the monitor. If these restrictions are imposed, it is possible to guarantee against certain of the more obscure forms of time-dependent coding error; and this guarantee could be underwritten by a visual scan of the text of the program, which could readily be automated in a compiler.

Any dynamic resource allocator will sometimes need to delay a program wishing to acquire a resource which is not currently available, and to resume that program after some other program has released the resource required. We therefore need: a "wait" operation, issued from inside a procedure of the monitor, which causes the calling program to be delayed; and a "signal" operation, also issued from inside a procedure of the same monitor, which causes exactly one of the waiting programs to be resumed immediately. If there are no waiting programs, the signal has no effect. In order to enable other programs to release resources during a wait, a wait operation must relinquish the exclusion which would otherwise prevent entry to the releasing procedure. However, we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signalling program without any danger that a third program will interpose a monitor entry and seize the resource instead.

In many cases, there may be more than one reason for waiting, and these need to be distinguished by both the waiting and the signalling operation. We therefore introduce a new type of "variable" known as a "condition"; and the writer of a monitor should declare a variable of type condition for each reason why a program might have to wait. Then the wait and signal operations should be preceded by the name of the relevant condition variable, separated from it by a dot:

```
condvariable.wait;
condvariable.signal;
```

Note that a condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program. In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition; but this queue is invisible both to waiters and signallers. This design of the condition variable has been deliberately kept as primitive and rudimentary as possible, so that it may be implemented efficiently and used flexibly to achieve a wide variety of effects. There is a great temptation to introduce a more complex synchronization primitive, which may be easier to use for many purposes. We shall resist this temptation for a while.

As the simplest example of a monitor, we will design a scheduling algorithm for a single resource, which is dynamically acquired and released by an unknown number of customer processes by calls on procedures

```
procedure acquire;
procedure release;
```

A variable[1]

```
busy:Boolean
```

determines whether or not the resource is in use. If an attempt is made to acquire the resource when it is busy, the attempting program must be delayed by waiting on a variable

```
nonbusy:condition
```

which is signalled by the next subsequent release. The initial value of busy is false. These design decisions lead to the following code for the monitor:

```
single resource:monitor
begin busy:Boolean;
    nonbusy:condition;
  procedure acquire;
    begin if busy then nonbusy.wait;
              busy := true
    end;
  procedure release;
    begin busy := false;
          nonbusy.signal
    end;
  busy := false; comment initial value;
end single resource
```

Notes

1. In designing a monitor, it seems natural to design the procedure headings, the data, the conditions, and the procedure bodies, in that order. All subsequent examples will be designed in this way.
2. The acquire procedure does not have to retest that busy has gone false when it resumes after its wait, since the release procedure has guaranteed that this is so; and as mentioned before, no other program can intervene between the signal and the continuation of exactly one waiting program.
3. If more than one program is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting program. This gives a simple neutral queuing discipline which ensures that every waiting program will eventually get its turn.
4. The single resource monitor simulates a Boolean semaphore [7] with *acquire* and *release* used for *P* and *V* respectively. This is a simple proof that the monitor/condition concepts are not in principle less powerful than semaphores, and that they can be used for all the same purposes.

[1] As in PASCAL [15], a variable declaration is of the form: ⟨*variable identifier*⟩:⟨*type*⟩;

2. Interpretation

Having proved that semaphores can be implemented by a monitor, the next task is to prove that monitors can be implemented by semaphores.

Obviously, we shall require for each monitor a Boolean semaphore "*mutex*" to ensure that the bodies of the local procedures exclude each other. The semaphore is initialized to 1; a *P*(*mutex*) must be executed on entry to each local procedure, and a *V*(*mutex*) must usually be executed on exit from it.

When a process signals a condition on which another process is waiting, the signalling process must wait until the resumed process permits it to proceed. We therefore introduce for each monitor a second semaphore "*urgent*" (initialized to 0), on which signalling processes suspend themselves by the operation *P*(*urgent*). Before releasing exclusion, each process must test whether any other process is waiting on *urgent*, and if so, must release it instead by a *V*(*urgent*) instruction. We therefore need to count the number of processes waiting on *urgent*, in an integer "*urgentcount*" (initially zero). Thus each exit from a procedure of a monitor should be coded:

```
if urgentcount > 0 then V(urgent) else V(mutex)
```

Finally, for each condition local to the monitor, we introduce a semaphore "*condsem*" (initialized to 0), on which a process desiring to wait suspends itself by a *P*(*condsem*) operation. Since a process signalling this condition needs to know whether anybody is waiting, we also need a count of the number of waiting processes held in an integer variable "*condcount*" (initially 0). The operation "*cond.wait*" may now be implemented as follows (recall that a waiting program must release exclusion before suspending itself):

```
condcount := condcount + 1;
if urgentcount > 0 then V(urgent) else V(mutex);
P(condsem);
comment This will always wait;
condcount := condcount − 1
```

The signal operation may be coded:

```
urgentcount := urgentcount + 1;
if condcount > 0 then {V(condsem); P(urgent)};
urgentcount := urgentcount − 1
```

In this implementation, possession of the monitor is regarded as a privilege which is explicitly passed from one process to another. Only when no one further wants the privilege is *mutex* finally released.

This solution is not intended to correspond to recommended "style" in the use of semaphores. The concept of a condition-variable is intended as a substitute for semaphores, and has its own style of usage, in the same way that while-loops or coroutines are intended as a substitute for jumps.

In many cases, the generality of this solution is unnecessary, and a significant improvement in efficiency is possible.

1. When a procedure body in a monitor contains no **wait** or **signal**, **exit** from the body can be coded by a simple *V*(*mutex*), since *urgentcount* cannot have changed during the execution of the body.
2. If a *cond.signal* is the last operation of a procedure body, it can be combined with monitor exit as follows:

```
    if condcount > 0 then V(condsem)
else if urgentcount > 0 then V(urgent)
                        else V(mutex)
```

3. If there is no other wait or signal in the procedure body, the second line shown above can also be omitted.
4. If *every* signal occurs as the last operation of its procedure body, the variables *urgentcount* and *urgent* can be omitted, together with all operations upon them. This is such a simplification that O-J. Dahl suggests that signals should always be the last operation of a monitor procedure; in fact, this restriction is a very natural one, which has been unwittingly observed in all examples of this paper.

Significant improvements in efficiency may also be obtained by avoiding the use of semaphores, and by implementing conditions directly in hardware, or at the lowest and most uninterruptible level of software (e.g. supervisor mode). In this case, the following optimizations are possible.

1. *urgentcount* and *condcount* can be abolished, since the fact that someone is waiting can be established by examining the representation of the semaphore, which cannot change surreptitiously within noninterruptible mode.
2. Many monitors are very short and contain no calls to other monitors. Such monitors can be executed wholly in noninterruptible mode, using, as it were, the common exclusion mechanism provided by hardware. This will often involve *less* time in noninterruptible mode than the establishment of separate exclusion for each monitor.

I am grateful to J. Bezivin, J. Horning, and R.M. McKeag for assisting in the discovery of this algorithm.

3. Proof Rules

The analogy between a monitor and a data representation has been noted in the introduction. The mutual exclusion on the code of a monitor ensures that procedure calls follow each other in time, just as they do in sequential programming; and the same restrictions are placed on access to nonlocal data. These are the reasons why the same proof rules can be applied to monitors as to data representations.

As with a data representation, the programmer may associate an invariant $\mathcal{I}$ with the local data of a monitor, to describe some condition which will be true of this data before and after every procedure call. $\mathcal{I}$ must also be made true after initialization of the data, and before *every* wait instruction; otherwise the next following procedure call will not find the local data in a state which it expects.

With each condition variable b the programmer may associate an assertion B which describes the condition under which a program waiting on b wishes to be resumed. Since other programs may invoke a monitor procedure during a wait, a waiting program must ensure that the invariant $\mathcal{I}$ for the monitor is true beforehand. This gives the proof rule for waits:

$\mathcal{I}\ \{b.wait\}\ \mathcal{I}\&B$

Since a signal can cause immediate resumption of a waiting program, the conditions $\mathcal{I}\&B$ which are expected by that program must be made true before the signal; and since B may be made false again by the resumed program, only $\mathcal{I}$ may be assumed true afterwards. Thus the proof rule for a signal is:

$\mathcal{I}\&B\{b.signal\}\mathcal{I}$

This exhibits a pleasing symmetry with the rule for waiting.

The introduction of condition variables makes it possible to write monitors subject to the risk of deadly embrace [7]. It is the responsibility of the programmer to avoid this risk, together with other scheduling disasters (thrashing, indefinitely repeated overtaking, etc. [11]). Assertion-oriented proof methods cannot prove absence of such risks; perhaps it is better to use less formal methods for such proofs.

Finally, in many cases an operating system monitor constructs some "virtual" resource which is used in place of actual resources by its "customer" programs. This virtual resource is an abstraction from the set of local variables of the monitor. The program prover should therefore define this abstraction in terms of its concrete representation, and then express the intended effect of each of the procedure bodies in terms of the abstraction. This proof method is described in detail in [13].

4. Example: Bounded Buffer

A bounded buffer is a concrete representation of the abstract idea of a sequence of portions. The sequence is accessible to two programs running in parallel: the first of these (the producer) updates the sequence by appending a new portion x at the end; and the second (the consumer) updates it by removing the first portion. The initial value of the sequence is empty. We thus require two operations:

(1) $append(x{:}portion)$;

which should be equivalent to the abstract operation

$sequence := sequence \cap \langle x\rangle$;

where $\langle x\rangle$ is the sequence whose only item is x and $\cap$ denotes concatenation of two sequences.

(2) $remove(\textbf{result}\ x{:}portion)$;

which should be equivalent to the abstract operations

$x := first(sequence);\ sequence := rest(sequence)$;

where *first* selects the first item of a sequence and *rest* denotes the sequence with its first item removed. Obviously, if the sequence is empty, *first* is undefined; and in this case we want to ensure that the consumer waits until the producer has made the sequence nonempty.

We shall assume that the amount of time taken to produce a portion or consume it is large in comparison with the time taken to append or remove it from the sequence. We may therefore be justified in making a design in which producer and consumer can both update the sequence, but not simultaneously.

The sequence is represented by an array:

$buffer$:**array** $0..N-1$ **of** $portion$;

and two variables:

(1) $lastpointer{:}0..N-1$;

which points to the buffer position into which the next append operation will put a new item, and

(2) $count{:}0..N$;

which always holds the length of the sequence (initially 0).

We define the function

$$seq\ (b,l,c) =_{df} \textbf{if}\ c = 0\ \textbf{then}\ empty\ \textbf{else}\ seq(b,l\ominus 1,c-1)\ \cap\langle b[l\ominus 1]\rangle$$

where the circled operations are taken modulo N. Note that if $c \neq 0$,

$$first(seq(b,l,c)) = b[l\ominus c]$$

and

$$rest(seq(b,l,c)) = seq(b,l,c-1)$$

The definition of the abstract sequence in terms of its concrete representation may now be given:

$$sequence =_{df} seq(buffer, lastpointer, count)$$

Less formally, this may be written

$$sequence =_{df} \langle buffer[lastpointer\ominus count], buffer[lastpointer\ominus count\oplus 1], \ldots, buffer[lastpointer\ominus 1]\rangle$$

Another way of conveying this information would be by an example and a picture, which would be even less formal.

The invariant for the monitor is:

$$0 \leq count \leq N\ \&\ 0 \leq lastpointer \leq N-1$$

There are two reasons for waiting, which must be represented by condition variables:

$nonempty{:}condition$;

means that the count is greater than 0, and

$nonfull{:}condition$;

means that the count is less than N.

With this constructive approach to the design [8], it is relatively easy to code the monitor without error.

```
bounded buffer:monitor
  begin buffer:array 0..N − 1 of portion;
        lastpointer:0..N − 1;
        count:0..N;
        nonempty,nonfull:condition;
    procedure append(x:portion);
      begin if count = N then nonfull.wait;
            note 0 ≤ count < N;
            buffer[lastpointer] := x;
            lastpointer := lastpointer ⊕ 1;
            count := count+1;
            nonempty.signal
      end append;
    procedure remove(result x:portion);
      begin if count = 0 then nonempty.wait;
            note 0 < count ≤ N;
            x := buffer[lastpointer⊖count];
            nonfull.signal
      end remove;
    count := 0; lastpointer := 0;
  end bounded buffer;
```

A formal proof of the correctness of this monitor with respect to the stated abstraction and invariant can be given if desired by techniques described in [13]. However, these techniques seem not capable of dealing with subsequent examples of this paper.

Single-buffered input and output may be regarded as a special case of the bounded buffer with $N = 1$. In this case, the array can be replaced by a single variable, the *lastpointer* is redundant, and we get:

```
iostream:monitor
begin buffer:portion;
      count:0..1;
      nonempty,nonfull:condition;
  procedure append(x:portion);
    begin if count = 1 then nonfull.wait;
      buffer := x;
      count := 1;
      nonempty.signal
    end append;
  procedure remove(result x:portion);
    begin if count = 0 then nonempty.wait;
      x := buffer;
      count := 0;
      nonfull.signal
    end remove;
  count := 0;
end iostream;
```

If physical output is carried out by a separate special purpose channel, then the interrupt from the channel should simulate a call of *iostream.remove*(x); and similarly for physical input, simulating a call of *iostream.append*(x).

5. Scheduled Waits

Up to this point, we have assumed that when more than one program is waiting for the same condition, a signal will cause the longest waiting program to be resumed. This is a good simple scheduling strategy, which precludes indefinite overtaking of a waiting process.

However, in the design of an operating system, there are many cases when such simple scheduling on the basis of first-come-first-served is not adequate. In order to give a closer control over scheduling strategy, we introduce a further feature of a conditional wait, which makes it possible to specify as a parameter of the wait some indication of the priority of the waiting program, e.g.:

```
busy . wait (p);
```

When the condition is signalled, it is the program that specified the lowest value of p that is resumed. In using this facility, the designer of a monitor must take care to avoid the risk of indefinite overtaking; and often it is advisable to make priority a nondecreasing function of the time at which the wait commences.

This introduction of a "scheduled wait" concedes to the temptation to make the condition concept more elaborate. The main justifications are:

1. It has no effect whatsoever on the *logic* of a program, or on the formal proof rules. Any program which works without a scheduled wait will work with it, but possibly with better timing characteristics.
2. The automatic ordering of the queue of waiting processes is a simple fast scheduling technique, except when the queue is exceptionally long—and when it is, central processor time is not the major bottleneck.
3. The maximum amount of storage required is one word per process. Without such a built-in scheduling method, each monitor may have to allocate storage proportional to the number of its customers; the alternative of dynamic storage allocation in small chunks is unattractive at the low level of an operating system where monitors are found.

I shall yield to one further temptation, to introduce a Boolean function of conditions:

```
condname.queue
```

which yields the value true if anyone is waiting on *condname* and false otherwise. This can obviously be easily implemented by a couple of instructions, and affords valuable information which could otherwise be obtained only at the expense of extra storage, time, and trouble.

A trivially simple example is an *alarmclock* monitor, which enables a calling program to delay itself for a stated number n of time-units, or "*ticks*". There are two entries:

```
procedure wakeme (n:integer);
procedure tick;
```

The second of these is invoked by hardware (e.g. an interrupt) at regular intervals, say ten times per second. Local variables are

```
now:integer;
```

which records the current time (initially zero) and

```
wakeup:condition;
```

on which sleeping programs wait. But the *alarmsetting* at which these programs will be aroused is known at the time when they start the wait; and this can be used to determine the correct sequence of waking up.

```
alarmclock:monitor
begin now:integer;
    wakeup:condition;
  procedure wakeme(n:integer);
    begin alarmsetting:integer;
      alarmsetting := now + n;
      while now < alarmsetting do wakeup.wait(alarmsetting);
      wakeup.signal;
      comment In case the next process is due to wake up at the
      same time;
    end;
  procedure tick;
    begin now := now + 1;
      wakeup.signal
    end;
  now := 0
end alarmclock
```

In the program given above, the next candidate for wakening is actually woken at every tick of the clock. This will not matter if the frequency of ticking is low enough, and the overhead of an accepted signal is not too high.

I am grateful to A. Ballard and J. Horning for posing this problem.

6. Further Examples

In proposing a new feature for a high level language it is very difficult to make a convincing case that the feature will be both easy to use efficiently and easy to implement efficiently. Quality of implementation can be proved by a single good example, but ease and efficiency of use require a great number of realistic examples; otherwise it can appear that the new feature has been specially designed to suit the examples, or vice versa. This section contains a number of additional examples of solutions of familiar problems. Further examples may be found in [14].

6.1 Buffer Allocation

The bounded buffer described in Section 4 was designed to be suitable only for sequences with small portions, for example, message queues. If the buffers contain high volume information (for example, files for pseudo offline input and output), the bounded buffer may still be used to store the *addresses* of the buffers which are being used to hold the information. In this way, the producer can be filling one buffer while the consumer is emptying another buffer of the same sequence. But this requires an allocator for dynamic acquisition and relinquishment of *buffer addresses*. These may be declared as a type

```
type bufferaddress = 1..B;
```

where B is the number of buffers available for allocation.

The buffer allocator has two entries:

```
procedure acquire (result b:bufferaddress);
```

which delivers a free *buffer address b*; and

```
procedure release(b:bufferaddress);
```

which returns a *bufferaddress* when it is no longer required. In order to keep a record of free buffer addresses the monitor will need:

```
freepool:powerset bufferaddress;
```

which uses the *PASCAL* powerset facility to define a variable whose values range over all sets of *buffer addresses*, from the empty set to the set containing all *buffer addresses*. It should be implemented as a *bitmap* of B consecutive bits, where the ith bit is 1 if and only if i is in the set. There is only one condition variable needed:

```
nonempty:condition
```

which means that *freepool* $\neq$ *empty*. The code for the allocator is:

```
buffer allocator:monitor
begin freepool:powerset bufferaddress;
    nonempty:condition;
  procedure acquire (result b:bufferaddress);
    begin if freepool = empty then nonempty.wait;
      b := first(freepool);
      comment Any one would do;
      freepool := freepool − {b};
      comment Set subtraction;
    end acquire;
  procedure release(b:bufferaddress);
    begin freepool := freepool − {b};
      nonempty.signal
    end release;
  freepool := all buffer addresses
end buffer allocator
```

The action of a producer and consumer may be summarized:

```
producer: begin b:bufferaddress; . . .
            while not finished do
              begin bufferallocator.acquire(b);
                . . . fill buffer b . . .;
                bounded buffer.append(b)
              end; . . .
          end producer;

consumer: begin b:bufferaddress; . . .
            while not finished do
              begin bounded buffer.remove(b);
                . . . empty buffer b . . .;
                buffer allocator.release(b)
              end; . . .
          end consumer;
```

This buffer allocator would appear to be usable to share the buffers among several streams, each with its own producer and its own consumer, and its own instance of a bounded buffer monitor. Unfortunately,

when the streams operate at widely varying speeds, and when the freepool is empty, the scheduling algorithm can exhibit persistent undesirable behavior. If two producers are competing for each buffer as it becomes free, a first-come-first-served discipline of allocation will ensure (apparently fairly) that each gets alternate buffers; and they will consequently begin to produce at equal speeds. But if one consumer is a 1000 lines/min printer and the other is a 10 lines/min teletype, the faster consumer will be eventually reduced to the speed of the slower, since it cannot forever go faster than its producer. At this stage nearly all buffers will belong to the slower stream, so the situation could take a long time to clear.

A solution to this is to use a scheduled wait, to ensure that in heavy load conditions the available buffers will be shared reasonably fairly between the streams that are competing for them. Of course, inactive streams need not be considered, and streams for which the consumer is currently faster than the producer will never ask for more than two buffers anyway. In order to achieve fairness in allocation, it is sufficient to allocate a newly freed buffer to that one among the competing producers whose stream currently owns fewest buffers. Thus the system will seek a point as far away from the undesirable extreme as possible.

For this reason, the entries to the allocator should indicate for what stream the buffer is to be (or has been) used, and the allocator must keep a count of the current allocation to each stream in an array:

```
count:array stream of integer;
```

The new version of the allocator is:

```
bufferallocator:monitor
  begin freepool:powerset bufferaddress;
      nonempty:condition
      count:array stream of integer;
    procedure acquire(result b:bufferaddress; s:stream);
      begin if freepool = empty then nonempty.wait(count[s]);
        count[s] := count[s] + 1;
        b := first(freepool);
        freepool := freepool − {b}
      end acquire;
    procedure release(b:bufferaddress; s:stream)
      begin count[s] := count[s] − 1;
        freepool := freepool − {b};
        nonempty·signal
      end;
    freepool := all buffer addresses;
    for s:stream do count[s] := 0
  end bufferallocator
```

Of course, if a consumer stops altogether, perhaps owing to mechanical failure, the producer must also be halted before it has acquired too many buffers, even if no one else currently wants them. This can perhaps be most easily accomplished by appropriate fixing of the size of the bounded buffer for that stream and/or by ensuring that at least two buffers are reserved for each stream, even when inactive. It is an interesting comment on dynamic resource allocation that, as soon as resources are heavily loaded, the system must be designed to fall back toward a more static regime.

I am grateful to E.W. Dijkstra for pointing out this problem and its solution [10].

6.2 Disk Head Scheduler

On a moving head disk, the time taken to move the heads increases monotonically with the distance traveled. If several programs wish to move the heads, the average waiting time can be reduced by selecting, first, the program which wishes to move them the shortest distance. But unfortunately this policy is subject to an instability, since a program wishing to access a cylinder at one edge of the disk can be indefinitely overtaken by programs operating at the other edge or the middle.

A solution to this is to minimize the frequency of change of direction of movement of the heads. At any time, the heads are kept moving in a given direction, and they service the program requesting the nearest cylinder in that direction. If there is no such request, the direction changes, and the heads make another sweep across the surface of the disk. This may be called the "elevator" algorithm, since it simulates the behavior of a lift in a multi-storey building.

There are two entries to a disk head scheduler:

(1) *request(dest:cylinder)*;

where

```
type cylinder = 0..cylmax;
```

which is entered by a program just *before* issuing the instruction to move the heads to cylinder *dest*.

(2) *release*;

which is entered by a program when it has made all the transfers it needs on the current cylinder.

The local data of the monitor must include a record of the current headposition, *headpos*, the current direction of *sweep*, and whether the disk is *busy*:

```
headpos:cylinder;
direction:(up, down);
busy:Boolean
```

We need two conditions, one for requests waiting for an *upsweep* and the other for requests waiting for a *downsweep*:

```
upsweep, downsweep:condition
```

```
dischead:monitor
begin headpos:cylinder;
    direction:(up, down);
    busy:Boolean;
    upsweep,downsweep:condition;
  procedure request(dest:cylinder);
    begin if busy then
      {if headpos < dest ∨ headpos = dest & direction = up
        then upsweep·wait(dest)
        else downsweep.wait(cylmax-dest)};
      busy := true; headpos := dest
    end request;
```

```
procedure release;
  begin busy := false;
    if direction = up then
      {if upsweep.queue then upsweep.signal
                        else {direction := down;
                              downsweep.signal}}
    else if downsweep.queue then downsweep.signal
                            else {direction := up;
                                  upsweep.signal}
  end release;
  headpos := 0; direction := up; busy := false
end dischead;
```

6.3 Readers and Writers

As a more significant example, we take a problem which arises in on-line real-time applications such as airspace control. Suppose that each aircraft is represented by a record, and that this record is kept up to date by a number of "writer" processes and accessed by a number of "reader" processes. Any number of "reader" processes may simultaneously access the same record, but obviously any process which is updating (writing) the individual components of the record must have exclusive access to it, or chaos will ensue. Thus we need a class of monitors; an instance of this class local to *each* individual aircraft record will enforce the required discipline for that record. If there are many aircraft, there is a strong motivation for minimizing local data of the monitor; and if each read or write operation is brief, we should also minimize the time taken by each monitor entry.

When many readers are interested in a single aircraft record, there is a danger that a writer will be indefinitely prevented from keeping that record up to date. We therefore decide that a new reader should not be permitted to start if there is a writer waiting. Similarly, to avoid the danger of indefinite exclusion of readers, all readers waiting at the end of a write should have priority over the next writer. Note that this is a very different scheduling rule from that propounded in [4], and does not seem to require such subtlety in implementation. Nevertheless, it may be more suited to this kind of application, where it is better to read stale information than to wait indefinitely!

The monitor obviously requires four local procedures:

startread	entered by reader who wishes to read.
endread	entered by reader who has finished reading.
startwrite	entered by writer who wishes to write.
endwrite	entered by writer who has finished writing.

We need to keep a count of the number of users who are reading, so that the last reader to finish will know this fact:

```
readercount:integer
```

We also need a *Boolean* to indicate that someone is actually writing:

```
busy:Boolean;
```

We introduce separate conditions for readers and writers to wait on:

```
OKtoread, OKtowrite:condition;
```

The following annotation is relevant:

```
OKtoread ≡ ¬ busy
OKtowrite ≡ ¬ busy & readercount = 0
invariant: busy ⟹ readercount = 0
```

```
class readers and writers:monitor
  begin readercount:integer;
      busy:Boolean;
      OKtoread, OKtowrite:condition;
    procedure startread;
      begin if busy ∨ OKtowrite.queue then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        comment Once one reader can start, they all can;
      end startread;
    procedure endread;
      begin readercount := readercount − 1;
        if readercount = 0 then OKtowrite.signal
      end endread;
    procedure startwrite;
      begin
        if readercount ≠ 0 ∨ busy then OKtowrite.wait
        busy := true
      end startwrite;
    procedure endwrite;
      begin busy := false;
        if OKtoread.queue then OKtoread.signal
                          else OKtowrite.signal
      end endwrite;
    readercount := 0;
    busy := false;
  end readers and writers;
```

I am grateful to Dave Gorman for assisting in the discovery of this solution.

7. Conclusion

This paper suggests that an appropriate structure for a module of an operating system, which schedules resources for parallel user processes, is very similar to that of a data representation used by a sequential program. However, in the case of monitors, the bodies of the procedures must be protected against re-entrance by being implemented as critical regions. The textual grouping of critical regions together with the data which they update seems much superior to critical regions scattered through the user program, as described in [7, 12]. It also corresponds to the traditional practice of the writers of operating system supervisors. It can be recommended without reservation.

However, it is much more difficult to be confident about the condition concept as a synchronizing primitive. The synchronizing facility which is easiest to use is probably the conditional *wait* [2, 12]:

```
wait(B);
```

where B is a general Boolean expression (it causes the given process to wait until B becomes true); but this may be too inefficient for general use in operating systems,

because its implementation requires re-evaluation of the expression B after every exit from a procedure of the monitor. The condition variable gives the programmer better control over efficiency and over scheduling; it was designed to be very primitive, and to have a simple proof rule. But perhaps some other compromise between convenience and efficiency might be better. The question whether the signal should always be the last operation of a monitor procedure is still open. These problems will be studied in the design and implementation of a pilot project operating system, currently enjoying the support of the Science Research Council of Great Britain.

Another question which will be studied will be that of the disjointness of monitors: Is it possible to design a separate isolated monitor for each kind of resource, so that it will make sensible scheduling decisions for that resource, using only the minimal information about the utilization of that resource, and using no information about the utilization of any resource administered by other monitors? In principle, it would seem that, when more knowledge of the status of the entire system is available, it should be easier to take decisions nearer to optimality. Furthermore, in principle, independent scheduling of different kinds of resource can lead to deadly embrace. These considerations would lead to the design of a traditional "monolithic" monitor, maintaining large system tables, all of which can be accessed and updated by any of the procedures of the monitor.

There is no a priori reason why the attempt to split the functions of an operating system into a number of isolated disjoint monitors should succeed. It can be made to succeed only by discovering and implementing good scheduling algorithms in each monitor. In order to avoid undesirable interactions between the separate scheduling algorithms, it appears necessary to observe the following principles:

1. Never seek to make an optimal decision; merely seek to avoid persistently pessimal decisions.
2. Do not seek to present the user with a virtual machine which is better than the actual hardware; merely seek to pass on the speed, size, and flat unopiniated structure of a simple hardware design.
3. Use preemptive techniques in preference to non-preemptive ones where possible.
4. Use "grain of time" [9] methods to secure independence of scheduling strategies.
5. Keep a low variance (as well as a low mean) on waiting times.
6. Avoid fixed priorities; instead, try to ensure that every program in the system makes reasonably steady progress. In particular, avoid indefinite overtaking.
7. Ensure that when demand for resources outstrips the supply (i.e. in overload conditions), the behavior of the scheduler is satisfactory (i.e. thrashing is avoided).
8. Make rules for the correct and sensible use of monitor calls, and assume that user programs will obey them. Any checking which is necessary should be done not by a central shared monitor, but rather by an algorithm (called "user envelope") which is local to each process executing a user program. This algorithm should be implemented at least partially in the hardware (e.g. base and range registers, address translation mechanisms, capabilities, etc.).

It is the possibility of constructing separate monitors for different purposes, and of separating the scheduling decisions embodied in monitors from the checking embodied in user envelopes, that may justify a hope that monitors are an appropriate concept for the structuring of an operating system.

Acknowledgments. The development of the monitor concept is due to frequent discussions and communications with E.W. Dijkstra and P. Brinch-Hansen. A monitor corresponds to the "secretary" described in [9], and is also described in [1, 3].

Acknowledgment is also due to the support of IFIP WG.2.3., which provides a meeting place at which these and many other ideas have been germinated, fostered, and tested.

Received February 1973; revised April 1974

References

1. Brinch-Hansen, P. Structured multiprogramming. *Comm. ACM 15*, 7 (July 1972), 574–577.
2. Brinch-Hansen, P. "A comparison of two synchronizing concepts," *Acta Information 1* (1972), 190–199.
3. Brinch-Hansen, P. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
4. Courtois, P. J., Heymans, F., Parnas, D.L. Concurrent control with readers and writers. *Comm. ACM 14*, 10 (Oct. 1971), 667–668.
5. Courtois, P.J., Heymans, F., Parnas, D.L. Comments on [2]. *Acta Informatica 1* (1972), 375–376.
6. Dahl, O.J. Hierarchical program structures. In *Structured Programming*, Academic Press, New York, 1972.
7. Dijkstra, E.W. Cooperating Sequential Processes. In *Programming Languages* (Ed. F. Genuys), Academic Press, New York, 1968.
8. Dijkstra, E.W. A constructive approach to the problem of program correctness. *BIT 8* (1968), 174–186.
9. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, Academic Press, New York, 1972.
10. Dijkstra, E.W. Information streams sharing a finite buffer. *Information Processing Letters 1*, 5 (Oct. 1972), 179–180.
11. Dijkstra, E.W. A class of allocation strategies inducing bounded delays only. Proc AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 933–936.
12. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, Academic Press, New York, 1972.
13. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica* 1 (1972), 271–281.
14. Hoare, C.A.R. A structured paging system. *Computer J. 16*, 3 (1973), 209–215.
15. Wirth, N. The programming language PASCAL. *Acta Informatica 1*, 1 (1971), 35–63.