A SCHEDULING PHILOSOPHY FOR MULTI-PROCESSING SYSTEMS

Butler W. Lampson*
University of California
Berkeley, California

Introduction

One of the essential parts of any computer system is a mechanism for allocating the processors of the system among the various competitors for their services. These allocations must be performed in even the simplest system, for example, by the action of an operator at the console of the machine. In larger systems more automatic techniques are usually adopted; batching of jobs, interrupts and interval timers are the most common ones. As the use of such techniques becomes more frequent, it becomes increasingly difficult to maintain the conventional view of a computer as a system which does only one job at a time; even though it may at any given instant be executing a particular sequence of instructions, its attention is switched from one such sequence to another with such rapidity that it appears desirable to describe the system in a manner which accommodates this multiplexing more naturally. It is worthwhile to observe that these remarks apply to any large modern computer system and not just to one which attempts to service a number of on-line users simultaneously.

This paper is concerned with a variety of ideas which bear on the analysis of a multiprocessing system. Most of these ideas center around two fundamental issues:

a. How decisions can be made about which instructions to execute next. Some of the basic ideas on this subject in the next two sections were originated by Saltzer.

b. How programs are protected from themselves and each other. Papers by Dennis and LeClerc have influenced the treatment of this subject.

Detailed consideration is given to methods for multiplexing processors, and it is shown that suitable hardware and software can make it possible to schedule all the processing done by the system in a uniform manner, so that interrupt routines do not have to be treated differently from ordinary user programs. Finally, a variety of mechanisms are considered for interlocking and synchronizing parallel computations.

Preliminaries

We begin by defining some terms which will permit a precise discussion of what happens in a complex computer system. The most important of these terms is process. The intuitive basis for the idea represented by this word is the observation that certain sequences of actions follow naturally, one upon the other, and are more or less independent of other sequences. For example, a disk-to-printer routine and a matrix inversion program running on the same processor are two quite distinct programs, which can normally execute entirely independently of each other. The independence is not complete, however, since information may be written into the disk file by the inversion program while earlier information is read by the printer driver. Furthermore, the fact that clearly different programs are being executed is not essential; two inversions operating on different data would have just as much right to be considered distinct. Indeed, it might well be that a single inversion program could be coded in such a way that several parts of it could be in execution simultaneously.

To summarize, the essential characteristic of a process is that it has, at least conceptually, a processor of its own to run on, and that the state of its processor is more or less independent of all the other processors on which all the other processes are running. Since there are usually not as many physical processors as there are processes, it becomes necessary to create enough logical processors by multiplexing the physical ones. Techniques for doing this are one of the subjects of this paper.

The State Vector

We now consider more carefully what is involved in switching a physical processor from one process to another. When a processor is executing instructions, there exists a collection of information (called the full state vector) which is sufficient to completely define its state at any given moment, in the sense that placing the processor in some

* The work reported in this paper was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract SD-185

arbitrary state and then resetting it from the full state vector will cause execution of instructions to proceed as though nothing had happened. For a central processor of conventional organization the full state vector includes:

a. The contents of the program counter.

b. The contents of the central registers of the processor. For convenience we will confine our attention to points in time between the execution of instructions at which registers directly accessible to the programmer are the only ones of interest. The use of address mapping schemes involving segmentation may make it necessary to consider also certain non-accessible registers involved in indirect addressing. This complication will be ignored.

c. The address space of the processor and the contents of every address in it. By this we mean a list of all the legal memory addresses which may be generated by the program and the contents of each one.

d. The state of all input-output devices attached to the processor.

The information just specified consists of a rather large number of bits. Furthermore, it contains considerably more detail than is usually desired. In a time-sharing system, for example, the physical state of the input-output devices is not normally of interest to the program, and even the contents of the next word to be read from a file may not be information which we wish to associate with a process, since such an association makes it difficult for two processes to share the file.

Similar considerations apply to the contents of the words in the address space; these might be changed by another process. The address space itself, however, has a better claim to be regarded as an integral part of the process. To clarify this point, we pause to consider (or define) exactly what the address space is. We may think of all the words which can possibly be addressed by any process as being numbered in some arbitrary fashion. Each word is then identified by its number, which we will call its absolute address. Note that we are not saying anything about the physical location of the words, and that the absolute address we have just defined has nothing to do with this location, but is simply a conceptual tool. Then the address space of a process (also called the map) is a function from the integers into the set consisting of the absolute addresses of all the words in the system and two symbols U and N. I.e., it associates with every address generated by the program an absolute address, or specifies that the address is undefined or not available. The address space may also carry information about the accessibility of a word for which it supplies an absolute address, specifying for example, that the word may not be written into.

Observe now that what the address space really does is to define the memory of the process. Any of the words of this memory may also be part of the memory of some other process, which may refer to them with the same or different addresses. The words, however, are considered to have an existence independent of their addresses in any particular address space.

We recognize, then, that the definition we want for the state vector of a process is not a direct analog of the one used in system theory. In fact, it leaves unspecified a number of things which can affect the future execution of a process, namely just those things which we wish to think of as being shared between processes. With this point in mind, we define the state vector (or stateword) of a process to consist of the program counter, central registers and address space of the processor on which it is running. From the above discussion we conclude that the process can still exist even if it is not running on the processor, since the state vector carries sufficient information to allow it to be restarted. A process should be sharply distinguished from a program, which is a sequence of instructions in memory. We can speak of either a process or a processor executing a program. The process is the logical, the processor the physical environment for this execution. It seems reasonable to say that the process is executing even if it is not running on any processor.

In order to give practical substance to the distinction between process and program, it is necessary to require that execution of the program should not cause it to be modified. If this restriction is observed, it is clear that more than one process can be executing the same program at the same time. Note that it is not necessary to have more than one processor for this to be possible, since we do not insist that a process be running in order to be executing. To return to our matrix inversion example: in a particular problem it may be necessary to invert six independent matrices, and to this end six processes may be established, one to work on each matrix but all executing the same program. On the other hand, one of these processes may, after inverting a matrix, go on to execute a different program which calculates its eigenvalues.

Operations on Processes

In this section we take up some points which are independent of processor multiplexing. A number of these points center around the observation that a process does not always want to execute instructions even if there is a physical processor available. A typical example of such a process is a compiler which has exhausted its available input and is waiting for more cards to be read. This is a special case of a very general situation, in which a process is waiting for some external condition to be satisfied and has nothing useful to do in the meantime. There are basically two ways in which this situation can be handled. The simplest is for the process to loop, testing a flag which records the state of the external condition. The objection to this scheme is that it is wasteful if there is any other process which could use the processor. Alternatively, the process can record somewhere the fact that it is waiting for, say, 14 cards to be read, and *block* itself. It will then execute no more instructions until the 14th card has been read and it has received a *wakeup* signal. The part of the system responsible for handling block and wakeup instructions will be called the *scheduler*. When a processor is executing instructions for a process, the process is said to be *running*.

From the point of view of the scheduler, then, the life history of a process is an alternation of running periods and blocked ones. Each running period is terminated by a request to the scheduler that the process should be blocked; this request may be made by the process itself or by some other process. Each blocked period is terminated by a wakeup signal whose characteristics we have just described. Since the process can do nothing to help itself while it is blocked, it must make arrangements to be woken before blocking itself. Let us consider what the nature of these arrangements might be.

The simplest situation is one in which the process has requested some service to be performed and wishes to wait until it is completed. Typical services have to do with input-output: a process might block itself until an input buffer is filled. In this case the process performing the service is expected to provide the wakeup signal. This idea is so simple as to require only one further comment. A reciprocal situation is that of the input-output process: it expects to be woken by a demand for service and blocks itself when it has done its job. The situation can be sketched as in figure 1. Readers familiar with the concept of co-routines will find this picture familiar.

Race Conditions

A slightly more involved situation is shown in figure 2. Here the input is assumed to be buffered, so that the computing process, after making a request for input, is able to continue running for a while. In the figure it continues to run on buffered input data until after the input-output process has completed its work and sent it a wakeup, which has no effect on an already running process. Finally, it blocks itself, but by this time the wakeup has passed and will never come again.

This situation can of course be alleviated by having the input process set a flag indicating that it has filled the next buffer. The computing process could then test this flag and not block itself if the operation is complete. This approach, however, will only serve to make the bad situation shown in figure 2a less common; it will not eliminate it entirely, since it is still possible for the input process to send its wakeup in the interval between an unsuccessful test and the subsequent block. What is necessary is to reduce this interval to zero, which can be done with a simple device called a *wakeup-waiting* switch. There is one of these for each process. Whenever a wakeup signal arrives at a running process, the switch is turned on. Wherever the process is blocked, it is turned off, and the process can also turn it off explicitly. If a process tries to block itself and the wakeup-waiting switch is on, the switch is turned off and the process is simply allowed to continue running without interruption. This sequence of events is illustrated in figure 2b.

Note that a single switch for each process is quite sufficient. Any function which might be accomplished with a stack of wakeup signals handled by the scheduler can equally well be accomplished by communication through shared memory. The details of such communication are too obvious to be worth discussing; in general a simple task queue will suffice.

The proper way to program a process which blocks itself periodically, like the ones in figure 2, is to check explicitly after a wakeup signal is received that there is in fact work to be done, and to block again if there is not. This procedure will circumvent the problems which would otherwise arise if the work for which a wakeup is received gets done before the process tries to block. Often the simplest way to make this check is with a loop back to the code which made the decision to block. Thus, for example, the following routine to read data from an input buffer being filled by another process:

```
a.  Is input buffer empty?
b.  If not, read data and exit.
c.  Otherwise, block
d.  When wakeup arrives, go to (a).
```

An additional complication is introduced by the fact that it is not unusual for several processes to be blocked waiting for the same condition to occur. If this situation can arise, the process generating the wakeup must be prepared to send it not just to one process, but to an entire wakeup list of processes. Alternatively, it may be preferable to leave this process unaware of the problem and to make one of the processes being woken up responsible for sending wakeups to the others (if such action is appropriate; in the case of several processes competing for a single device it may not be). The possible existence of wakeup lists is not important to the basic scheme and has been mentioned only to illustrate a complication which can be handled without difficulty.

Suspension

We now have two states for a process: blocked and running, and two basic operations on processes: block and wakeup. Let us consider what else is required. Two things are obvious: it is necessary to be able to create and destroy processes, and to establish restrictions on the kind of access allowed to one process by another. These problems, however, have to do with the general mechanism in the system for granting capabilities, with which this paper is not concerned. The precise operations required to create a process will be considered in a later section.

It is, however, very convenient for a process to have one other property in addition to being blocked or running. Suppose, for example, that a program involving several processes is being debugged on-line. A valuable tool in this situation is the ability to stop execution of the program, examine the state of things, and then continue as though nothing had happened. During the pause in execution, all the processes being debugged are suspended; i.e., they are prevented from executing. If a wakeup signal is directed at a suspended process, it is recorded and will be acted upon when the process is released. In other words, being suspended is a quality unrelated to being blocked; a suspended process can be either blocked or running, but it will execute no instructions until it is released.

Approximately the same effect can be obtained by reading the state words of all the processes, destroying them, and then recreating them later and putting back the statewords. This approach, however, requires a potentially large amount of information to be recorded and then restored. It is also likely to be quite time consuming, especially since it is usually not possible to destroy a process at any arbitrary point in its execution, e.g. when it is operating on system tables. Furthermore, if any wakeup signals for a process arrive after it has been destroyed, they will be lost, which is not very satisfactory.

Acquisition of Capabilities

We conclude this section with a brief discussion of the relationship between processes and protection. It has been taken for granted so far, as in almost all of the published literature, that a process is the basic entity to which capabilities should be attached. Since this doctrine cannot easily be made to cover all situations, it is usually modified in running systems by more or less inelegant devices for allowing a process to have different capabilities depending on what it is doing.

To make this point clearer, consider an ordinary unprivileged process which wishes to obtain permission to use a tape unit, for example. It will make some kind of "call on the system." That is, it will transfer to a system routine which will determine whether the process is authorized to use tape units and whether the specified unit is free. If everything is in order, the system will grant the desired permission. In order to avoid unnecessary digression, let us assume that permission is granted by turning on a bit in the stateword, and that the hardware allows execution of instructions for this tape unit only when the bit is on. The point is that the process running the system program must have much greater capabilities than the user process; it has at least the capability to set the hypothetical bit just introduced, and the user process does not have that capability or it would have set the bit itself. In fact, we can dispense with the bit if anyone can set it, since it would serve no useful purpose.

This discussion suggests that the "call on the system" mentioned above is really a wakeup directed to a process with great capabilities, which we will call a system process. There must also be some mechanism for communicating data between user and system processes, so that the latter can find out what is wanted and the former obtain information about the fate of its request. We may assume that the two processes share some memory, and neglect a host of questions about how such an arrangement can actually be implemented with reasonable efficiency. The situation is then quite clear: the user process puts information about its request into this shared memory, wakes up the system process, and blocks. The system process examines the request, modifies the user process stateword appropriately, records what it has done in the shared memory, wakes up the user process and blocks. The entire interaction depends on cooperation between the two processes, which is precisely why it is a suitable mechanism for its purpose: the user process cannot force the system process to do anything, but can only call attention to its desires.

It is worth noticing, however, that the use of two processes in the manner just discussed is completely different from the usual applications of multiprocessing, where two processes are running (more or less) in parallel. Here no parallel execution takes place at all; the system process runs when, and only when, the user process is blocked. The entire transaction in fact looks exactly like a subroutine call and return implemented in a very clumsy way. The only reason that the system process is a process rather than a program is that it has capabilities which we wish to deny to the user process.

If we approach the discussion of capabilities from a different viewpoint, it will become clear why very few systems use the multiple process scheme described above. The first point to observe is that instead of thinking of a process as starting out with no power and acquiring capabilities, one may think of it as starting out with absolute power and then being restricted by the system's protection mechanisms. This is of course the customary way of looking at things; the word "absolute" is especially appropriate on a mapped machine, since protection is usually enforced primarily by the map and the ability to use absolute addresses allows a process to do anything. Let us consider briefly the fundamental characteristics of protection systems.

The simplest way of describing what is required is to say that certain processes must be prevented from executing certain words as instructions, namely those which violate the constraints of the protection they operate under. For purposes of implementation, however, it has proved convenient to make a further distinction and say that a process must be prevented from

1. Accessing, changing or transferring control to certain words in the physical memory of the machine (memory protection).
2. Executing instructions with certain operation codes, which are often called privileged (control protection).

To accommodate the first requirement a variety of arrangements has been tried, which fall into two general classes, namely the protection of particular physical regions of core and the provision of an address mapping function which transforms every address generated by the program before sending it to the physical memory. The latter scheme is generally recognized as superior, and has been discussed at some length by various authors. We turn our attention here to the problems of control protection.

Protection Systems

What might be called the minimal solution to these problems, and one which has been implemented on many machines, is the provision of two modes called "monitor" and "user", or "master" and "slave" modes. In the first, or monitor, mode all the instructions of the machine can be executed, including those which change the address space of the process. In user mode, on the other hand, all the opcodes which might interfere with the operation of the system or of another user are prohibited. This prohibition is usually enforced by causing a switch into monitor mode and a transfer to a standard system routine whenever attempted execution of a privileged opcode is detected. These include any opcode which is undefined or which halts the machine, all input-output opcodes, and all opcodes which invoke the scheduling or address mapping hardware.

One of the implications of this kind of organization is that calls on the system, say in the form of a switch to monitor mode and a transfer to one of a small number of standard entry points, must be provided for any operation which a user program may be authorized to perform and which requires execution of privileged instructions. Because the ability to execute such instructions attaches to a process, and because there are only two possible states which a process can be in as far as its authority to execute such instructions is concerned, there is no way of avoiding a call to a system routine every time the need for a privileged instruction arises. Furthermore, this routine will in general have to check the validity of the call every time it is entered. Some improvement can be obtained by providing a number of different modes in which various classes of instructions are prohibited, but the number of input-output devices attached to a system is likely to be large enough to prevent this approach from effecting much improvement, since in general every device must be protected independently of every other one. Matters are still worse when it comes to mass storage devices and it is desired to allow access only to certain areas of the device.

An alternative approach to the whole problem of control protection, and one which is capable of eliminating the problems we have been considering, is to attach the authority to execute privileged instructions not to the process executing them but to the memory locations from which they are executed. Almost any memory protection scheme for a time-sharing system will allow an area of memory to be made read-only to a process. A system routine which controls the memory protection hardware can obtain read/write access to this area and therefore can put into it anything it likes. What we envision now is an extension of the memory protection mechanism which allows this area to be made not merely read-only, but also privileged. If a memory location is part of a privileged area, then the control protection hardware will allow a privileged instruction fetched from this location to execute. Otherwise, execution of such an instruction will be suppressed.

The system then simply arranges that the memory available to any process which is authorized to execute privileged instructions contains those instructions which the system wants to allow the process to execute, and no others; the process executes them by addressing them with an execute instruction. The result of such an arrangement is that from the point of view of the system control protection is absorbed into memory protection: the ultimate authority which a process can exercise is determined by the memory it can write, since not only the ability to execute privileged instructions but also the state of the memory protection system itself is determined by the contents of certain areas of memory. A process can do anything if it can write into privileged memory and can set up the memory protection for itself or for another process.

It therefore follows that to change the capabilities of a process it suffices to change the map. The operation of waking up a system process can be replaced by a subroutine call which makes some memory accessible which was formerly forbidden. One popular implementation falls back temporarily on the monitor mode concept--a transfer of control which changes the protection (called a leap) must go through a standard routine which, running in monitor mode, is not subject to the usual restrictions of the system. This scheme is completely general and quite elegant; its only drawback is that leaps are likely to be slow.

A mapping scheme proposed by Evans and Le Clerc eliminates this objection and extends the idea. Its foundation is the observation that a single procedure does not require access to the entire address space of the process of which it is a part. This consideration leads to an addressing scheme in which each segment is able to address a small number (say 16) of other segments. Each of these may in turn address other segments, so that the entire or global address space may be very large, even though the local address space, or that directly accessible from a single segment, is quite small. This arrangement has substantial advantages in terms of address bits saved. Its more important contribution, however, is the organization which it forces on the programmer and enforces on his behalf: each part of the program is able to address exactly that data which it needs, and nothing else.

Such a drastic reduction in the number of entities which can be independently protected (segments) also permits a new approach to the specification of protection. It is now possible to put the bits specifying the protection of a segment A not on the segment itself but on the link between A and the segment B which refers to it, since B can refer only to 16 segments. The protection information for an entire process is then specified not by a vector of protection bits for each segment, but by a matrix M which defines the relation of every segment to every other. An entry in this matrix, M_{ij}, says either that segment j is not accessible from i or that it is identified in i by the number s ($0 \leq s \leq 15$) and that i is allowed certain kinds of access to it (read, write, execute, privileged, for example).

Scheduling

In this section we consider some algorithms which may be used to implement a scheduler, and complete the discussion of basic principles with an analysis of processor multiplexing. The scheduler consists of two sets of procedures which are logically quite independent of each other, and a data base which connects them.

User Interface

The first set of procedures may be called the *user interface*. Its function is to implement the basic operations which may be called for by a user process:

block
wakeup
suspend
release
test and reset wakeup waiting switch
change priority (see below)

If processor multiplexing is not required this is a very straightforward matter. We define four arrays indexed by process or processor number (these are equivalent if there is no multiplexing):

run[i] is 1 if processor i is executing instructions, 0 if it is not. This value controls the processor.

suspend[i] is 1 if process i is suspended, 0 if it is not.

runstate[i] is 0 if the process was blocked more recently than it was woken up, 1 otherwise.

wws[i] is 1 if the wakeup waiting switch for process i is on, 0 if it is off.

These arrays, together with some other data to be introduced shortly, constitute the *scheduler data base*. Pseudo-Algol procedures for the basic operations are:

```
    procedure block(i); begin if wws[i]=0 then
run[i]:=runstate[i]:=0 else wws[i]:=0 end;
    procedure wakeup(i); begin if runstate[i]=1
then wws[i]:=1; runstate[i]:=1; if suspend[i]=0
then run[i]:=1 end;
    procedure suspend(i); begin run[i]:=0;
suspend[i]:=1 end;
    procedure release(i); begin suspend[i]:=0;
run[i]:=runstate[i] end;
    integer procedure trwws(i); begin trwws:=
wws[i]; wws[i]:=0 end
```

In the absence of multiplexing this is the entire implementation of the scheduler. It is assumed that the action specified by one of these procedures is taken instantaneously as far as the rest of the system is concerned. We defer a consideration of how this can be done and of the general problems of interlockin parallel processes to a later section. Note that we have not said anything about how these procedures are executed; it will be sufficient for the moment to imagine a special processor whose sole function is to do this.

Multiplexing

As soon as we begin to consider processor multiplexing life becomes much more complicated and more interesting. In order to introduce the complications one at a time, let us assume that the user interface procedures continue to function as before, but that the array run no longer directly controls the operation of a processor. In fact, since there are no longer enough processors to allow one to be assigned to each program, the data base arrays are not directly related to processors at all. To specify the indirect relationship, we define two more arrays:

processor[i] specifies the processor assigned to process i. It is 0 if no processor is assigned.

process[j] specifies the process running on processor j. It is 0 if no process is running.

A process i is called *ready* if run[i]=1 but processor[i]=0. The procedures which establish connections between processes and processors constitute the second half of the scheduler, which is called the *enforcer*. The algorithm used by the enforcer to assign processors is called the *scheduling algorithm*. A very simple and very unsatisfactory scheduling algorithm might be the following:

```
         i:=0
ploop:   for j:=1 step 1 until np do
         if process[j]=0 then begin
qloop:     i:=if i=nq then 1 else i+1;
           if run[i]=0 ∨ processor[i]≠0
           then goto qloop; processor[i]:=j;
           process[j]:=i end
         else if run[process[j]]=0 then begin
           process[j]:=processor[process[j]]:=
           goto qloop end;
         goto ploop;
```

Here *np* is the number of processors, *nq* the number of processes. This procedure simply cycles around the processes in fixed sequence, assigning free processors to processes which wish to run as it finds them. Again we assume that where there is any possibility of confusion the enforcer's actions are performed instantaneously.

Priorities

In order to improve on this algorithm it is necessary for the scheduler to have some idea about the relative importance of different processes. For purposes of discussion we will consider that two measures of importance exist:

> an integer assigned to each process called its *priority*. The enforcer regards process i as more important than process j if priority[i]>priority[j]. The array priority is added to the scheduler data base.
>
> The amount of time which has elapsed since a process was ready and assigned a particular priority. For processes of equal priority the enforcer operates on a first come, first served basis.

Two modifications to the user interface are required to accommodate this new idea. One is a change in wakeup which allows a priority to be supplied along with the wakeup signal:

```
procedure wakeup(i,p); begin
  if runstate[i]=1 then wws[i]:=1;
  runstate[i]:=1; priority[i]:=p;
  if suspend[i]:=0 then run[i]:=1 end;
```

The other is a new operation:

```
  procedure chpri(i,p); begin priority[i]:=p
end;
```

These two operations make it possible to establish priorities for the various processes. The way in which this is done will determine which processes get to run, and will therefore have an important influence on the behavior of the system. It should be clear, however, that priority assignment is not an integral part of the scheduler, and it will therefore receive only passing consideration.

We expect the enforcer to select from the processes with run[i]=1 those np which have the highest priority at each instant. The precise length of an instant will be considered later; obviously we want it to be as short as possible provided the overhead stays low. A typical situation which the scheduling algorithm might encounter in a two-processor system might involve four processes computing with various priorities, another four processes blocked for ordinary input/output operations, and two processes awaiting real-time interrupts. At one moment the system should be running two of the computing processes, at the next one of these and then a process activated because of the receipt of teletype input, and at the next two top-priority real-time processes. See figure 3.

We now proceed to consider a possible implementation of the multiplexing philosophy just described. The idea behind this scheme is that the scheduler should not be aware of a process at all until it is woken up, i.e., it should look only at processes on a list called the *ready list*. This list should be organized so as to make the selection of highest priority processes as natural as possible and to facilitate the introduction of new processes. The former requirement can be very simply satisfied by keeping the process numbers of the ready processes in a table of consecutive registers in order of their priority. This arrangement is, however, extremely inconvenient when it comes to adding processes, since on the average half of the list will need to be moved for each addition.

A slight improvement can be effected by replacing the table with a linked list, so that insertion requires only a splicing of pointers. It still requires a search through the list, however, to find the appropriate point at which to make the insertion. In order to eliminate this search it will be necessary to introduce more structure into the ready list. One way to do this is the following: restrict priorities to be integers in the range from 1 to n, and equip the scheduler with an n-word table called the priority list. Each entry of this table contains either 0 or a pair of process numbers, *plfp* and *plrp*. Associated with each process there are also two process numbers, *fqp*[i] and *bqp*[i].

The significance of these arrangements is as follows: every non-zero priority list entry contains pointers to the head and tail of a queue of ready processes which have the priority given by the index of the entry in the priority list. When a new process arrives, it is added to the tail of the queue; processes are run beginning at the head. The queue is kept as a symmetric list to facilitate deletions.

To clarify the ready list pointer structure, the situation in figure 3 is displayed in full in figures 4 and 5.

Management of the Ready List

In order to take advantage of this data structure it is desirable to integrate the enforcer with the user interface procedures, since the enforcer needs to act only in response to a block, wakeup, or change priority operation (suspend and release, if they involve the enforcer at all, are equivalent to block and wakeup). Constant scanning of the scheduler data base is therefore not necessary. The pointer manipulations now become sufficiently complex, however, that a description in words seems more satisfactory than explicit procedures.

There are two basic cases in which the enforcer is activated. The first is when a process is woken up. It is necessary to

W1) Enter this process in the ready list at the appropriate priority on the tail of the queue (unless it is already on the ready list at a higher priority).
W2) Check to see whether this priority is higher than that of some running process. If not, there is nothing to do.
W3) If so, switch the processor running the lowest priority process so that it will run the newly introduced process instead. The abandoned process remains on the ready list and will run in due course. It has been pre-empted.

The second case is when a running process blocks and its wakeup-waiting switch is not set.

B1) Remove its queue cell from the ready list.
B2) Find the lowest priority running process. Examine its forward pointer.
B3) If it is a pointer to another queue cell, run the process in that queue cell, which is now the lowest priority process.
B4) Otherwise, scan down the priority list from the entry at the priority level of the lowest priority running process. When a non-zero entry is found, run the process in the queue cell pointed to by its head pointer.

A change priority operation, the only other one which affects the enforcer, can be implemented with the sequence

```
block
priority[i]:=new priority
wakeup
```

In practice, of course, this can be improved upon, since there are many cases in which no changes in processor assignment are called for.

If it is assumed that the priority list and the queue cells are kept in memory, then the cost of block and wakeup operations in memory references (assuming that queue pointers are packed two per word) is

W1) Three references to enter a new cell in a queue (one if level was empty). Two of these (to splice pointers) can be made in parallel. One more reference is needed to record the new state of the process; it can be made in parallel with all of the first three.
W2,3) One reference to switch processes on a processor (not counting loading and storing of the state vector), followed by another reference to record the new state of the pre-empted process. This assumes that the priority and queue cell address of every running process are kept in registers associated with the processor running it.

B1) Three references to delete a process (two if it is the only one on its priority level). Two of these (to splice pointers) can be made in parallel.
B3) Two references to get the process number of the new lowest priority process if it is on the same level as the current one. Only one is required if the process blocked is the lowest one, since the pointer to the next one is obtained in the splicing operation.
B4) Two references also if it has lower priority, plus references wasted in the scan.

The cost of the scan can be reduced to one reference by providing a bit word with one bit for each priority level and turning this bit on if the level is occupied. This scheme requires an extra reference every time a level becomes empty or ceases to be empty. Its value therefore depends on the density of empty levels.

If the actual lowest priority process (the one farthest down the queue at the lowest priority level) is not known, step B3 is complicated by the possibility that we may have to pass over queue cells for already running processes. If there are many priority levels containing only one process, the cost of entering or deleting a process can be reduced by one cycle if we treat this as a special case and put the single process number directly in the ready list.

Observe that the algorithms described above can be implemented in a mechanism, independent of any processor which is able to read and write memory, accept block and wakeup requests, and send to a processor the information that it should dump the process it is now running and start executing another one. If the process number of the new process is held in a fixed memory location unique to the processor, then only one control line is required from the scheduler to each processor. The line simply says: switch processes.

If we neglect the possibility of wakeup signals which come from the outside world, however, it is also clearly possible for everything to be done by one of the processors being scheduled. Since every scheduling operation is initiated by an ordinary user process, there is no problem in finding a processor to do the work. The only other difficulty is in getting a processor to switch processes. This may be done with a signal which any processor can send to any other, including itself, provided this signal is not sent until all the other manipulations connected with the scheduling operation have been completed. See Saltzer for a more detailed discussion of this point.

Timers

So far we have seen two mechanisms which can cause a process to lose its processor:

it may be blocked, by itself or by some other process

it may be pre-empted by a higher priority process.

If every process could be counted on to run for only a short period before blocking, these mechanisms would be entirely sufficient. Unfortunately, in the real world processes run for sufficiently long periods of time that it is necessary to have some method for stopping their execution, or at least reducing their priority, after a certain amount of time has elapsed.

The period for which a process is allowed to run before such disagreeable things begin to happen to it is usually called a quantum. It may vary with the priority, the process, the time of day or anything else, and how its value is established is a policy decision with which we are not concerned. Some attention must be paid, however, to the methods by which the policy decision, once arrived at, is to be enforced. It is desirable that these methods should allow as much flexibility as possible in the choice of quantum and should not add significantly to the cost of scheduling.

What we wish to do is to attach to each process as part of its stateword an integer which we call a timer. When the process is running, the timer is held in a hardware register and decremented at fixed intervals. When it reaches zero, the process is forced to transfer to a standard location where the system leaves a transfer to a routine which decides what to do. Two alternatives seem plausible:

1. Assign a lower priority to the process, an action which may cause it to be deprived of its processor.
2. Leave the process at the same priority, but give preference to all the other processes at the same priority level.

Neither one, however, need be built into the scheduling mechanism.

The choice of an initial setting for the timer is another policy decision, which can be made when the process is blocked or when it is woken up. The latter alternative is somewhat more appealing, since it allows for the possibility that different wakeup signals may arrive for the process, with different priorities and requiring different amounts of time. The drawback is that a wakeup signal must carry an additional piece of information. In either case we may observe that the choice of quantum and the action to be taken when it is exhausted can, like the assignment of priority, be left within limits to the discretion of the user.

One other point which ought to be brought out in regard to the kind of priority multiplexing scheme we have been discussing: it does not guarantee that a ready process will ever be run; if enough processes of higher priority exist, it will in fact not be run. This may be exactly what is intended, but if it is not, the design of the priority and quantum assignment algorithms must take the unpleasant possibility into account. This might be done by restricting the frequency with which a process may enter the ready list with high priority. Another alternative is to restrict the length of time it may run at high priority, although this one may be subverted by the fixed overhead imposed by the need to swap in the memory for the process. Still a third approach is to increase the priority of processes which have been waiting for a long time at lower levels.

Fixed Time Scheduling

Everything that we have said so far about wakeup signals has implied that they originate in some definite action on the part of some unblocked process, whether this process be within the system or entirely external to it. A signal originated by an external process might be the result of a circuit breaker opening. There is one particular class of wakeup signals, however, which demand special consideration, and these are the signals arising from the elapse of specified intervals of time. It is extremely common for a process to wish to block itself until, say, noon arrives, or for 5 seconds to give a user opportunity to respond to some stimulus, or for that fraction of 100 ms remaining since it was last awakened if it wishes to give attention to some input signal at that interval.

Another important case arises from the following observation: there is a large class of applications for which a user sitting at a teletype prefers a uniform 2-second response time to one which has, say, a uniform distribution between .5 and 2 seconds, or even one which is .5 seconds with probability .75 and 2 seconds the rest of the time. This is a fortunate circumstance from the point of view of the system, since it is only required to provide service sometime within a 2-second period chosen at its convenience. It does, however, require reasonably flexible timing facilities, both to give warning of the close approach of the deadline and to delay the generation of output until 2 seconds have elapsed.

Such requirements can of course be satisfied by the provision of an interval timer for each one; when a process blocks itself for n μs, one of these timers is set for that interval and the signal which it will emit on its expiration is routed to the process. This solution is, however, obviously wasteful in the extreme.

To improve on it we create a list called the <u>fixed-time</u> <u>list</u>, each entry of which contains:

1. A process number and priority.
2. A time.
3. A pointer to the next entry on the list.

The contents of the first entry on this list is kept in a fast register and the time constantly compared with a real-time clock. As soon as the latter becomes greater, the specified process is woken up, the entry put on a free storage list, and the contents of the new first entry loaded into the registers. Any process desiring service at a fixed time simply puts the proper entry on this list and blocks itself. Note that it is possible for the contents of the registers to be changed by the appearance of a request with time earlier than that now in them.

This arrangement ensures that the scheduler will at all times be aware of the requirements of processes which wish to run while it remains ignorant of those whose wakeup time has not yet arrived. In case there are not enough processors to run all the processes demanding service, the priority mechanisms of the scheduler can be relied upon to allocate processors to the most important ones. This means that, given information about the distribution of processes at various priorities, it is possible to compute beforehand what kind of service a process can obtain at a given priority level or, alternatively, what priority it must have to obtain a given level of service. For those processes on the fixed time list precise information is available. For those being woken up in other ways only probabilities can be known beforehand. This uncertainty can be eliminated by assigning higher priorities to the fixed-time processes than to any others, or, if the cost of this scheme is unacceptable, it is still possible to work out expected average and maximum delays. Since the system can enforce its decisions about priority and running time, it can guarantee the correctness of its estimate for these delays (barring hardware failure or bugs in the system programs). The important point is that no computations need to be made when it is time to wake up a process; everything can be worked out when the process makes a request for service, and it can be informed exactly what kind of response can be obtained at what price.

Guaranteed Service

Some elementary observations about this response may be in order here. These are intended to suggest the scope of the problem; very little attention has been given to it, and much work needs to be done. Let us ignore the existence of non-fixed-time processes for simplicity, and let us also assume that no time elapses between the arrival of a wakeup signal and the execution of the first useful instruction of a process. Then if there are n processors we can guarantee to n processes any kind of service they may require, simply by giving them all the highest priority. After that, life becomes a little more complex, and some information must be required of the processes about the frequency, quality and duration of service which they require. This information might include any of the following items:

1. With what frequency will the process wakeup? Must it run at fixed times, or only at fixed intervals with an arbitrary origin? For most sampling processes the latter will suffice. What times and intervals does it wish to run at?

2. What errors in the satisfaction of the above requirements can be tolerated, and with what probability. Is it acceptable,for instance, to miss 1 percent of the samples entirely? This might frequently be the case. What is the desired distribution of error versus frequency of occurrence. Perhaps a 2 ms error is completely acceptable, one of 5 ms tolerable 10 percent of the time, and one of 10 ms intolerable. Or perhaps 40 μs accuracy is required with 0 tolerance for greater error. Presumably a real system will have a minimum error, determined by the response time of the scheduler, which is independent of priority.

3. How long will the process run each time? Perhaps no more than 200 μs. Perhaps 1 ms usually, 2 ms 10 percent of the time. To what extent can it tolerate interruptions? Must it perform 5 ms of computation in 5 ms of real time, or is it acceptable for 7 ms of real time to elapse?

Needless to say, not all users will be able or willing to supply accurate information about all of these items. For those who do not, the system can make worst-case assumptions and charge accordingly; an incentive is thus provided for the user to state his requirements as precisely as possible.

When all this information has been collected for all the fixed-time processes and suitable statistics and worst case information supplied for the others, a routine can be run to figure out what service can be provided with what probability. This routine will certainly be complex and slow if it is to do a good job, but this is not particularly objectionable, since it runs only when a user requests a different grade of service, not whenever he is served. Too frequently running of the routine can be discouraged by charging for it. As each new request for service comes in, it is thus possible to determine whether it can be satisfied (together with all the other requests already accepted) and at what cost; and if not, what service can be provided. The user can thus obtain precise information about what to expect. If the system is overloaded, of course, some users will not be able to get what they want. The proper response to this situation is to expand the system (if the user requests are justified; this is of course a policy decision which cannot be made by the system); if the service allocation routine is good, there can be reasonable assurance that the system is doing as well as it can.

Hardware Implementation

The concepts and techniques described in the last three sections have been developed to provide the sole scheduling mechanism for a reasonably large time-shared system. Nearly all such systems in existence or under development are based on a dichotomy between two different schedulers: one implemented in hardware and usually called the interrupt system, the other implemented in software and used to schedule user programs. Such an organization has a number of drawbacks. First of all, it leads to a sharp distinction between interrupt routines, which are regarded as part of the basic system, usually run in some unprotected mode, and cannot call on the system services available to ordinary programs; and user processes, which cannot respond directly to external signals and cannot run for short periods of time without incurring overhead considerably greater than their running time.

Secondly, the software scheduling system is usually quite slow and cumbersome. It requires a considerable amount of time to convince both the processor hardware and the system software that a different process is being run, and the computation required to properly handle the highly non-uniform load of user processes is substantial. As a result the time required to schedule a process is on the order of milliseconds, except possibly for a very small class of processes which can be given special treatment. On the other hand, a process scheduled in this way can call on all the services which the system provides without taking any special precautions (can, for example, open a disk file, which is a highly non-trivial operation), and it runs within the elaborate framework of protection normal for a user process which prevents it from damaging other users or degrading the service too much and insulates it from the many of consequences of its own folly.

Thirdly, it is nearly impossible to establish any kind of communication between the priority system established by the interrupt hardware and the one defined by the software.

The usual rule is simply that all interrupt routines take precedence over any "normal" processes. It is, of course, possible to persuade the software scheduler to in effect turn an interrupt routine into an ordinary process, but this is a messy and time-consuming procedure. Related to this problem is the fact that the interrupt system's priority scheme is not likely to be satisfactory in demanding situations; many routines need much more flexibility to establish their priorities than the hardware can conveniently allow, especially where the set of interrupts which are expected during any given five-minute period is a small and varying subset of all those which might be handled in a week.

A fourth point which is somewhat unrelated to the first three is that most interrupt systems do not switch enough of the state of a processor automatically. The most obvious omission in most of them is the interval timer and elapsed time clock.

Policy and Administration

These considerations suggest that it might be worthwhile to develop a system which would eliminate interrupts and drastically speed up the software scheduler by centralizing all responsibility for assigning processors to processes into one mechanism. A basic objection to any such proposal might well be the following: once something has been built into hardware it is very difficult to change. Scheduling algorithms are not well understood, and it is not likely that we can lay down rules today for deciding what processes to run which will satisfy us next year. Therefore, we should not freeze our present inadequate ideas into the system forever.

To see why this argument is weak, observe that a broad distinction can be made between a policy-making and an administrative module in a system. The latter performs some function in a manner controlled by parameters supplied to it by the former. Of course, it is true that the organization of an administrative module affects the kinds of parameters that can be fed to it, and consequently determines the system's policy within certain limits. These limits are very wide, however, in many cases of practical interest. Consider, for example, an input-output buffering system. By adjusting the number of buffers, the blocking factor, the organization of buffers into pools and the priorities of files competing for buffers, the behavior of the system can be varied over a wide range.

Similarly with a scheduler: its behavior is determined by the priority assignment algorithms, the choice of quanta for running processes and the action taken on quantum overflow, as well as by decisions of the swapper and the input-output system. The scheduler itself simply provides a framework for enforcing the decisions taken by policy modules.

Furthermore, the scheduler is such a basic part of the system that it is difficult to see how it could be drastically altered without a complete revision of the rest of the system. The effort required for such a change is probably greater than that required to rebuild any reasonable piece of hardware, so that the flexibility offered by software is likely to be illusory.

A Hardware Scheduler

With these preliminaries out of the way, let us consider how an interrupt system might be replaced by a more powerful mechanism. The functions of an interrupt system are three:

to continually monitor a fairly large number of external signals and take appropriate action when one of them changes state.

to start a processor executing instructions for a new process within a fairly small number of microseconds after an interrupt arrives, regardless of what it is doing at the time.

to recognize a sequence of priorities among interrupt signals and keep the processors executing the highest priority ones.

To replace it we clearly need a piece of equipment which functions independently of the processors being scheduled, and which is capable of examining say fifty external lines, recognizing that one of them is high, putting a process on the ready list and giving it a processor, all within a period of perhaps 10 μs. Present-day hardware technology allows such a device, which we will henceforth call the scheduler, to be built with a read-only microprogram memory and some small number of internal registers, say 5 or 10, and to operate with a cycle time of 100 to 200 ns. The major delays it encounters are due to the main memory of the system. This, however, is likely to be organized in numerous modules, so that the scheduler can make several references to main memory in parallel if this is convenient.

All the external interrupt lines are directed into the scheduler, which in its normal state loops constantly, examining them and the request lines from the processors for activity. Associated with each line is a fixed core location (possibly relative to a programable base register). When the scheduler finds an active line it goes to this location to find out what to do. In most cases the line will carry a wakeup signal, and the core location will contain either 0, which causes the signal to be ignored, or the arguments for a wakeup operation, which are a process number and a priority. The process number is actually a pointer to a block of words which contain:

- the state word for the process (or its core or drum address).
- bits which specify whether it is blocked, ready, or running, what processor it is running on, whether it is suspended or not, and the wakeup waiting switch.
- forward and backward pointers for the ready list queue it is on, if any.

If the process is ready or running at a higher priority than its current one, there is nothing to do. Otherwise, it must be added to the ready list or moved to the appropriate level; the steps required to accomplish this have already been presented. If its priority is high enough it must be given a processor, which is done by storing its stateword address into a standard place and sending the selected processor a switch signal. The scheduler then sets the status bits for the newly running process and for the one which has been pre-empted and goes on its way.

Processor Switching

The processor which receives the switch signal must store the stateword of the process it is currently running in the proper place, which it is responsible for remembering, and pick up the address of its new stateword from the cell where the scheduler left it. Note that the stateword of a process is always associated with the process itself and never with the one which pre-empts it. This means that it is not necessary to exit from "interrupt routines" in the order in which they are entered. In fact, the whole idea of an interrupt routine does not have much meaning. The time required to store a stateword and pick up a new one will depend on memory speed and the number of central registers, but with a memory of reasonable bandwidth and 1 μs cycle time it should not be more than 4 or 5 μs.

The stateword, of course, defines the process. The information it contains, together with a minimal amount of other status information and temporary storage for essential system routines, is all that is needed to allow a process to run (although to do anything useful it will have to have some user program and data storage as well). All this data can be held in a block of memory locations which we may call the context block. The address of this block is then sufficient information to give to a processor when it starts to run the process, and the operation of creating a process consists precisely in creating a new context block. The layout of a context block is shown in figure 6.

In a paged swapping system it will probably be convenient to assign a page to the context block, which may then be identified by its drum address if it is not in core. Of course, it is not possible for a processor to run the process if its context block is out of core, but the scheduler can detect this situation and wakeup a system process instead. This process, which might be called the context block swapper, can then take responsibility for bringing the context block into core and waking up its process again.

The algorithms for block and change priority have already been considered and present no new problems. The treatment of the timer has also been considered. Recall that it does not involve the scheduler at all; decisions about what to do when a timer trap occurs are matters of policy and must be left as flexible as possible.

It should be pointed out that there is a distinction between interrupts, which are wakeup signals, and traps, which are forced transfers of control within a single process. This distinction is made very sharp by a hardware scheduler, which has complete jurisdiction over wakeups but knows nothing about traps. In addition to the timer trap, there are also likely to be traps for various conditions having to do with memory addressing, for protection violations, for floating-point overflow, and possibly for a variety of other conditions. A call on the system by a user process is also a kind of trap, and indeed in some systems it takes the same form as an illegal instruction execution. This observation should illuminate the relationship of a trap to a wakeup signal, especially in the light of our earlier discussion of system calls.

Locks

Need for Interlocking Processors

It is very often the case in a large system, whether it be an entire time-sharing complex or simply an applications program, that independent processes work on the same data base. When the data base is being modified, it is generally not in a fit state to be looked at. It is therefore necessary for a process which intends to modify the data base to lock out any other process which might want to modify or look at it. The sequence of events required is

1. Test the lock to see if it is set. If so, loop in this step. If not, go on.
2. Set the lock. There must not be any opportunity between steps one and two for another process to set the lock. If this event should occur, both processes would proceed to access the data base simultaneously, exactly the condition we are trying to avoid.
3. Examine or modify the data base.
4. Clear the lock.

In some cases these precautions are required only when data are being changed. At other times, especially when pointers are involved, it is dangerous even to look at the data if another process might be modifying them. The details will vary with the specific application, but the nature of the problem remains the same.

Possible Implementations

At least two techniques exist for implementing locks. The first is to provide a machine instruction of the following general form: test the contents of the memory word addressed. If it is negative, skip. Otherwise make it negative and take the next instruction. If we call this instruction TSL for test and set lock, then the sequence

```
TSL   LOCK
BRU   OK     branch unconditionally
BRU   *-2
```

will not allow control to reach OK unless the lock has been found not to be set, and when control does reach OK the lock will be set again. Probably two memory references to LOCK will be required by TSL. If this is the case, access to that cell by any other process must be inhibited between the two references of the TSL.

This mechanism allows an arbitrary number of locks to exist. A lock is cleared by storing some positive number in the lock cell. A minimum of two instructions must be executed, and a minimum of four memory cycles is required. The biggest drawback is that a process hung up waiting for a lock to be cleared expends memory cycles without doing any useful work. These memory references degrade the performance of the rest of the system.

An alternative method is to supply each process with a lock register consisting of n bits. The equivalent of TSL hangs the process until a specified bit is off, then turns it on and proceeds. Two instructions are still required, but only two memory references. The cost of waiting for a lock to clear is simply the cost associated with the processor which is delayed; there is no drain on the rest of the system. There are two drawbacks: the number of different locks which may be set is limited by the length of the lock register, and a physical connection between processors other than the memory is required, even though it is a simple one. Furthermore, it is not clear what to do with the lock register if a process is blocked and the processor given to another process.

If processors are being shared, either method has the following further drawback. Suppose there is only one processor, and that process A is running, sets a lock, and is deprived temporarily of its processor in favor of process B, which attempts to set the same lock. Process B will hang, but the lock will never be cleared, since A will never be able to continue (unless B is pre-empted by a timer runout). This is rather serious. The difficulty can be avoided by increasing the priority of process A so much that it cannot be pre-empted, but this scheme has obvious disadvantages.

The problem is handled in most existing systems by precisely this means, however. "Increasing the priority of process A" is accomplished by disabling the interrupt system, so that any interrupt signals which come in are stacked until an enable instruction is executed. The process which executes the disable instruction will run without interference, since on most systems all the mechanisms for taking the processor away from it are dependent on interrupts.

A General Interlocking Mechanism

If the first method described above (TSL) is used, the following solution to this problem is possible. When a process tests a lock and finds it set, it blocks itself. The wakeup signal is supposed to be the clearing of the lock. It is definitely not desirable, however, to require that the process clearing the lock be aware of other processes which desire to be woken when it is cleared, since the tribulations of these other processes are not the proper business of the former, but rather of the scheduler. Secondly, there may be more than one process waiting for the same wakeup.

We proceed to describe a technique which overcomes both of these problems and which has many other applications as well. For the sake of clarity a particular version of this technique is described in great detail. This version has some significant drawbacks, but since a full discussion of them and consideration of how they may be avoided would require many pages and would not be particularly relevant to the problem of locks which we are considering, it has been omitted. The reader who is offended by the details about to be described is invited to skip over them, remembering only the basic method for handling locks which has been outlined in the last paragraph.

The device about to be described is probably a minimum perturbation of conventional hardware. What we do is to complicate the indirect addressing mechanism of the hardware, so that if an indirect address chain leads through a word with a certain bit set, the next word in memory will be executed as an instruction. Very likely it will be a subroutine call. A desirable refinement is to allow this to take place on a read access, a write access or both. Two bits in the indirect address word are required for this. Once this convention has been adopted by the hardware it is no longer possible in general to use an instruction word, say, as an indirect address, since these two bits will be interpreted in a manner inconsistent with their use in the instruction. This suggests that it might be desirable to restrict the application of the convention to special load and store instructions. In fact, however, most systems will probably have indirect address words with many more address bits than an instruction word, in order to allow for the addressing of a large amount of memory. If this is true, the addition of two mode bits is not serious. Indeed, many more mode bits would be desirable to control the way in which the indirection proceeds from level to level.

We now make the convention that a lock L will always be referenced indirectly through another cell, say L1. Initially, when no one is waiting for the lock to be cleared, things look like this

```
L     DATA   -1      lock set
L1    ZRO    L       indirect word pointing
                       to L
```

If someone blocks waiting for the lock, this becomes

```
L     DATA     -1
L1    ZRO      L2,1
      CALL     WAKEUP1
L2    ZRO      L
```

where the tag on L1 will cause a store instruction which indirects through it to execute the CALL of a routine to wake up the blocked process. If a second process wishes to block, it will add another link to the indirect chain, making it

```
L     DATA     -1
L1    ZRO      L3,1
      CALL     WAKEUP1
L3    ZRO      L2,1
      CALL     WAKEUP2
L2    ZRO      L
```

A process wishing to test the lock will do the following

1. TSL*   L1

This is counted as a read reference to memory, not a store.

2. If the lock is set (i.e., if the TSL skips) construct a two word pair, say at LN, with the contents of L1 in the first word and a call to a wakeup routine in the second.

3. Store
```
        ZRO    LN,1
```
in L1

4. Block

5. Go to 1

This sequence will work even if several processes are waiting on the same lock. They will all be woken up (if there are enough processors) but only one will get to set the lock. The others will have to block again. Various permutations of the order in which entries are put on the indirect chain can be envisioned.

The wakeup routine must do the following things

1. Put the two words addressed by L1 into L1 and L1+1.

2. Execute the instruction which addressed L1. This may cause more wakeup routines to be called, and will eventually clear the lock.

3. Generate the wakeup signal.

4. Return to the program which cleared the lock.

If it is known that the instruction which caused the routine to be called was a lock clear, and if it is not desired to wake up more than one process when this event occurs, then step 2 can be omitted.

Short-Term Interlocks

It will not have escaped the observant reader that some stages of the block and wakeup sequences outlined here will require their own lock mechanisms. If other programs tamper with the indirect chain or with the lock between steps 1 and 3 of either sequence, disaster will occur. Because of the very special character of the two critical sequences it seems quite reasonable to introduce a special-purpose protection mechanism to safeguard them. It is extremely desirable that this mechanism should have very low overhead, since it must be invoked whenever a lock is set but will be of use only infrequently. It is also desirable that it should be applicable in other situations where execution of a small number of instructions must be protected, e.g. the operations of obtaining cells from a free storage list or returning them to it; the time required for these operations is so small that even the rather fast method we have described would more than double the execution time.

We therefore introduce one more instruction called PROtect, whose function is to ensure that during some short period after the execution of PRO:

a. The process which executes the PRO cannot lose its processor.

b. No other PRO can be executed. If another processor attempts to do a PRO, it is forced to wait until the current one is complete. Simple hardware synchronization techniques can ensure this.

The "short period" mentioned above is probably best measured in memory references by the processor and about 15 is probably the right number. Time is not satisfactory since the amount of time required to execute an instruction is unlikely to be predictable in advance, and instruction executions is even worse, since an indirect addressing loop can cause the processor to hang without executing any instructions. Since a PRO cannot hang up a processor for more than a small number of memory cycles it does not need to be a privileged instruction.

Details

We can now write out the lock testing sequence in full:

```
START   PRO
        TSL *   L1
        BRU     OK
        LDA     CALLWK   pick up instruction
                         to call wakeup routine
        STA     LN+1
        LDA     LNI
        XMA     L1       exchange memory and the
        STA     LN       A register
        :
LNI     ZRO     LN,1
```

The PRO extends at least to the STA instruction. To clear the lock, we write

```
CLEAR   LDA   =0
        PRO
        STA * L1
```

The store indirect may cause a transfer to the wakeup routine. The PRO is necessary to prevent the lock from being cleared while someone is in the process of setting it. The wakeup is done by:

```
WAKEUP  ZRO   0   the return link is
                  stored here
        LDX   L1
        LDA X 0   load indexed
        STA   L1
        LDA X  1
        STA   L1+1
        LDX   WAKEUP
        EXU X  0  execute the instruction
                  which addressed <1
        WAKE      wakeup the process res-
                  ponsible for this
                  routine
        BRR   WAKEUP  return to the
                  instruction after the
                  STA* of the CLEAR
                  sequence
```

The PRO needs to cover only the first three instructions of the wakeup sequence.

Conclusions

With this much machinery (PRO is the only new instruction which is really essential) we have a very satisfactory system for interlocking independent processes. Short sequences of instructions can be protected by PRO; if every other sequence of instructions which is executed by another process and references the sensitive data is also covered by a PRO, it is not possible for two such sequences to be executed simultaneously. Larger operations on shared data bases can be interlocked with locks in memory. The cost of setting and clearing such a lock is only a few instructions.

The methods we have been discussing depend on cooperation among the processes referencing a shared data base and on correct programming of each reference to the data. As van Horn and others have pointed out, the bugs introduced by incorrect handling of this problem occur in a random and generally irreproducible manner and are very difficult to remove. Van Horn has proposed a scheme which enforces proper handling of shared data; it does however require more substantial hardware modifications than the methods suggested here.

References

1. Dennis, J et al, Machine Structures Group Memos 19, 40, 41, M.I.T., Cambridge, Mass. (1966).

2. Dennis, J. B., "Segmentation and the Design of Multiprogrammed Computer Systems," JACM 12, 4 (Oct. 1965), pp. 589-602.

3. Dennis, J., and Van Horn, E., "Programming Semantics for Multiprogrammed Computations," CACM 9, 3 (March 1966) pp. 143-155.

4. LeClerc, Jean-Yves, "Memory Structures for Interactive Computers," Project Genie Document 40.10.110, University of California, Berkeley (May, 1966).

5. Saltzer, J. H., "Traffic Control in a Multiplexed Computer System," MAC-TR-30 (Thesis), M.I.T., Cambridge, Mass. (July, 1966).

6. Van Horn, E. C., "Computer Design for Asynchronously Reproducible Multiprocessing: MAC-TR-34 (Thesis), M.I.T., Cambridge, Mass. (November 1966).

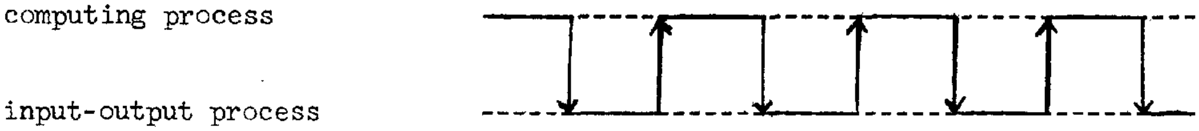
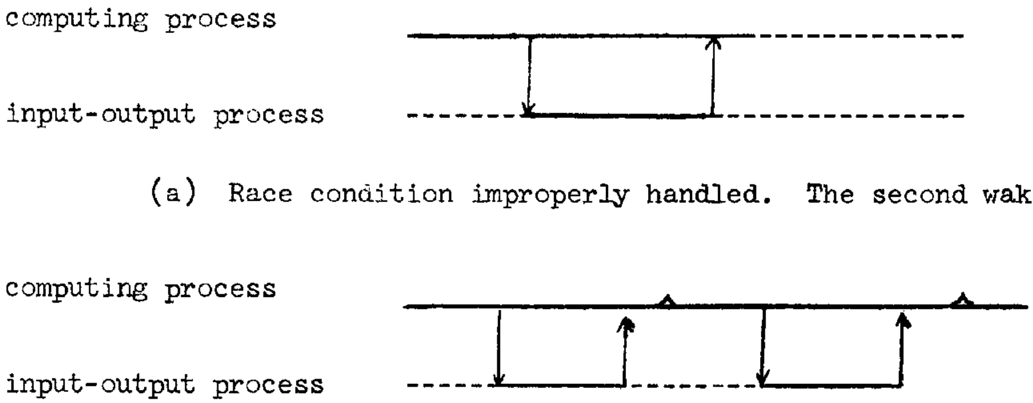


Figure 1: Receiprocally blocked processes. Solid horizontal lines indicate running periods, dashed ones blocked periods. The arrows are wakeup signals.



(a) Race condition improperly handled. The second wakeup is lost.

(b) Race conditions handled correctly. The cusps are attempts to block thwarted by the wakeup-waiting switch.

Figure 2: Race Conditions in the Scheduler

$\boxed{\text{C,6}}$ $\boxed{\text{C,6}}$ $\underline{\text{C,7}}$ $\underline{\text{C,8}}$ Computing processes

I/O,4 I/O,3 I/O,6 I/O,8 Processes blocked for I/O

I,1 I,2

(a) A possible state of the system. The running processes are boxed, the ready ones underlined. Numbers indicate priorities; the highest priority is 1.

$\boxed{\text{C,6}}$ $\boxed{\text{C,6}}$ $\underline{\text{C,7}}$ $\underline{\text{C,9}}$

I/O,4 I/O,3 $\underline{\text{I/O,6}}$ $\underline{\text{I/O,8}}$

I,1 I,2

(b) Another possible state

$\boxed{\text{C,6}}$ $\underline{\text{C,6}}$ $\underline{\text{C,7}}$ $\underline{\text{C,8}}$

$\boxed{\text{I/O,4}}$ I/O,3 $\underline{\text{I/O,6}}$ $\underline{\text{I/O,8}}$

I,1 I,2

(c) A higher-priority process becomes ready

$\underline{\text{C,6}}$(1) $\underline{\text{C,6}}$(2) $\underline{\text{C,7}}$(3) $\underline{\text{C,8}}$(4)

$\underline{\text{I/O,4}}$(5) I/O,3(6) $\underline{\text{I/O,6}}$(7) $\underline{\text{I/O,8}}$(8)

$\boxed{\text{I,1}}$(9) $\boxed{\text{I,2}}$(10)

(d) Two urgent real-time processes are running. Process numbers in parentheses.

Figure 3: Running, ready and blocked processes in a two-processor system.

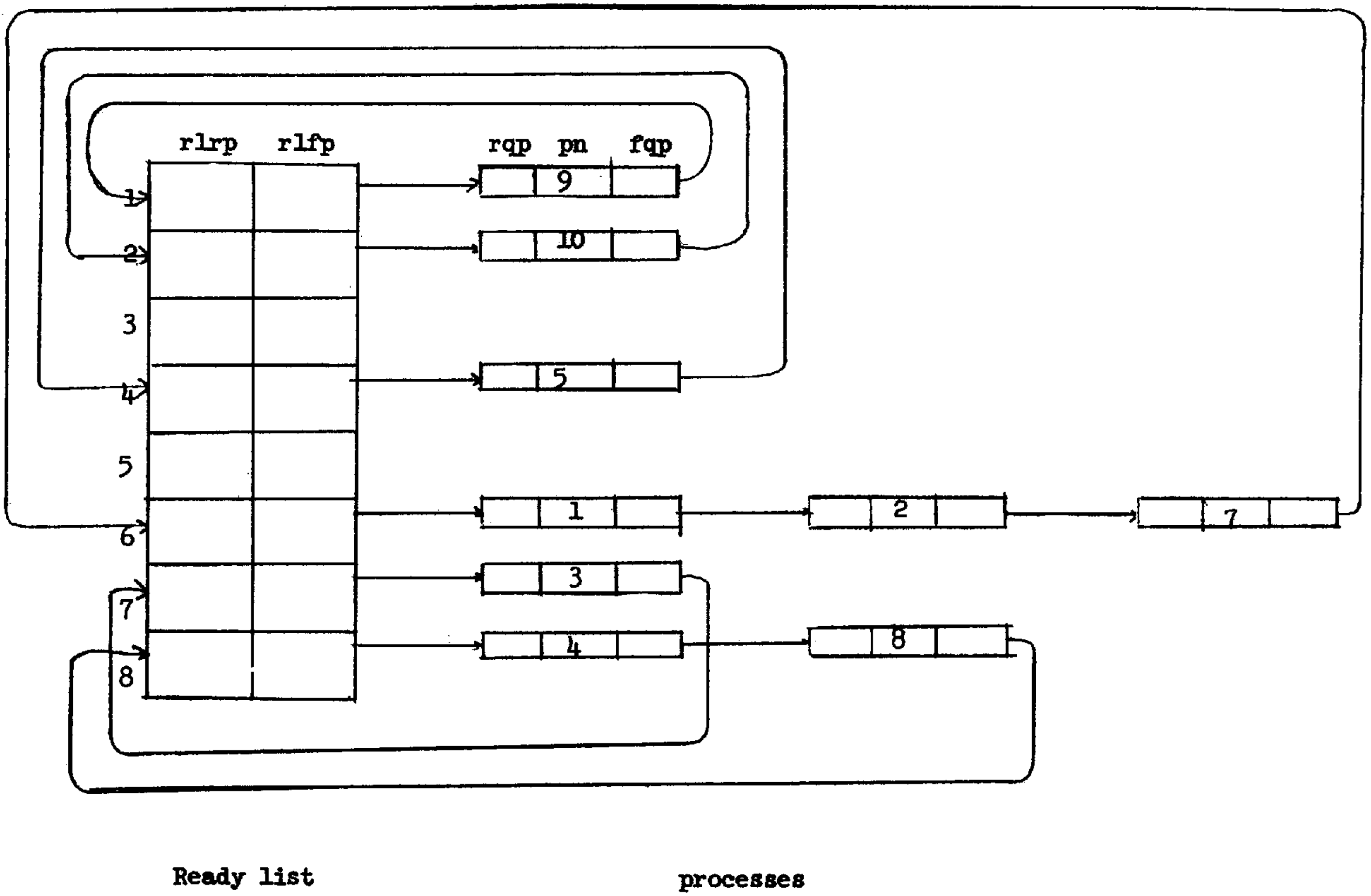


Figure 4; Ready list and queues for figure 3d

Priority	rlfp	rlrp
1	9	9
2	10	10
3	0	0
4	5	5
5	0	0
6	1	7
7	3	3
8	4	8

Figure 5a: Ready list for Figure 3d

Process	Priority	fqp	rqp	processor	run
1	6	2	*6	0	1
2	6	7	1	0	1
3	7	*7	*7	0	1
4	8	8	*8	0	1
5	4	*4	*4	0	1
6	-	-	-	0	0
7	6	*6	2	0	1
8	8	*8	4	0	1
9	1	*1	*1	2	1
10	2	*2	*2	1	1

Figure 5b: Ready list queues for Figure 3d

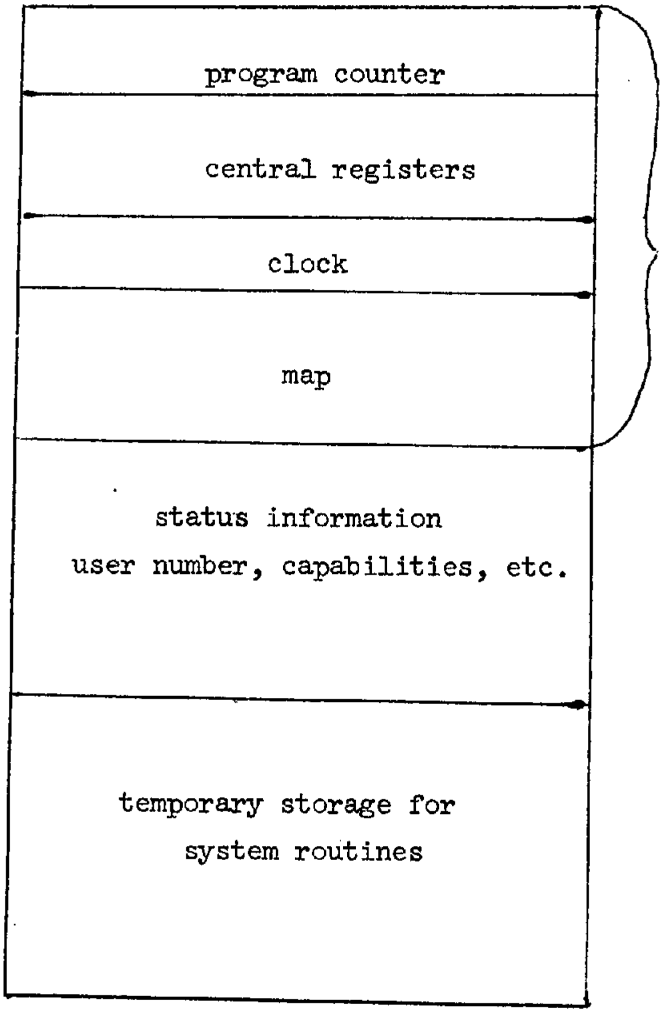


Figure 6: Contents of a context block