A MODEL DISPATCHER

C.A.R. Hoare

1. Introduction.

The purpose of a dispatcher is to share one or more central processors among a greater number of "virtual" processors, and to do so efficiently, reliably, and unobtrusively.

(1) Efficiency: the volume of code and data must be minimised; the time taken to change program must be negligible.

(2) Reliability: when a processor is faulty, its withdrawal should not affect the efficient working of the system as a whole.

(3) Unobtrusiveness: the dispatcher should have a narrow interface to the virtual machine, and should place minimal constraints on the way in which the virtual machine behaves.

In particular, the dispatcher should not need to know either the number of actual machines it has at its disposal, or the number of virtual machines it is servicing.

2. The functions of a dispatcher.

The functions of a dispatcher are:

(1) synchronisation.

(2) provision of timing signals.

(3) loadshedding and resumption on behalf of a paging system.

(4) determination of relative priorities of user virtual machines.

We do not regard it as the responsibility of the dispatcher to collect statistics or accounting information, or to enforce time limits. These functions can be carried out on the user's own virtual machine by a higher level of the operating system.

3. Entry points.

The basic entries to the dispatcher are:

(1) relinquish: entered by a virtual machine which (as a result of P operation) can no longer run, and therefore relinquishes its actual machine.

(2) resume(him:virtual machine): entered as a result of a V operation to cause "him" to be resumed. The calling program continues for the time being.

(3) wait(interval:time): the calling program waits for at least the interval specified.

(4) lendback (priority): this is entered fairly regularly by the user virtual machine, and permits a higher priority virtual machine (if any) to proceed in its place. It should not be entered from within a "critical region".

The following entries relate to the loadshedding system:

(5) loadshed: causes one program to be loadshed.

(6) resumeload: causes a loadshed program to be resumed.

(7) shedme: entered by a user who expects to wait a long time (e.g., for terminal response).

(8) unshedme: entered by user after the long wait is over.

The main purpose of these is to prevent the shedding of a multiple access program too soon after its long wait; but to permit shedding after it has done a heavy burst of computation.

The function carried out individually by a virtual machine will be known as

adjustpriority;

it is assumed that each virtual machine will invoke adjustpriority regularly in virtual time, say once every 2000 instructions executed by that machine.

4. The Hardware Background.

It is obviously essential that the code of the dispatcher be treated as a critical region, so that it cannot be entered simultaneously by two actual processors. We therefore postulate a

hardware mutex:semaphore

which is a Boolean semaphore provided in hardware. On a single-processor installation this is represented by the bit which inhibits interrupts. The individual operations of the dispatcher will be protected by an implicit P on this semaphore before entry, and a V on exit.

We also postulate in each actual machine a register

me:virtual machine

which contains the identity of the virtual machine which that actual machine is serving. This may, for example, be a pointer to the dumped registers of that virtual machine. The dispatcher can determine which virtual machine is to run next on the given actual machine by changing the value of this register. Note that the dispatcher itself uses the same actual machine as the virtual machine which calls it.

We also postulate a hardware function which delivers an indication now of the current real time, measured say in milliseconds.

Finally we introduce into the workload a number of "dummy" virtual machines equal to the number of actual machines. These machines all always have bottom priority, and so they are serviced only when there is nothing else to do. Their purpose is to conduct a test of the hardware of an actual processor. If the test fails, the virtual machine retains hold of the actual machine until it is mended. The dispatching system is designed so that all other actual and virtual machines continue to operate, unaffected by such detected failure, either in a machine test or in a user program. This gives a high degree of reliability to the system as a whole.

5. Data Design.

The most important data item maintained by the dispatcher is the queue of virtual machines waiting for process time:

queue: sequence of record v:virtual machine; p:priority end

This queue will be sorted in sequence of priority; thus insertion of a new item must cause it to be inserted in the right place, and removal of the first item will always yield the virtual machine with highest priority. High priority will be indicated by a low value of p, and vice-versa.

The queue of machines waiting for elapse of an interval has a similar structure

waitqueue: sequence of record v:virtual machine;rewaken:time end.

This is also held in sequence of reawakening.

When a waiting -program is resumed, it is good policy to permit it to run as soon as possible afterwards, at least until it is prepared to lend back the processor. We therefore introduce an

urgentqueue: sequence of virtual machine;

This queue observes an FCFS discipline; in any case it will hardly ever contain more than one member. If appropriate hardware is available, the emptiness of this queue may reduce the frequency with which virtual machines invoke "lendback".

The loadshedding system maintains two additional queues:

sheddable,shed: queue of virtual machine

Both of these observe an FCFS discipline. In addition there are variables

lastshed:time

giving the time of last invocation of shedme, and

shedrequest:virtual machine,

indicating the virtual machine (if any) which is requested to shed itself as a result of an invocation of loadshed.

6. The Algorithm.
Note. v:virtual machine, t:time
are a temporary variable used in the following:

relinquish: if urgent ≠ empty then me from urgentqueue
else if now>waitqueue.first.reawaken then (me,t) from waitqueue
else (me,t) from queue;

```
resume(him):   urgent:^ him;

wait (interval):  waitqueue:^ (me,now+interval);

                  relinquish;

lendback(priority): if   shedrequest=me, then
                         begin  shed:^ me;
                                shedrequest:=null
                         end
                    else  queue:^ (me,priority);

loadshed: if lastshed >now-one second then lastshed:=null
        else if sheddable ≠ empty then  shedrequest from sheddable
                                        else resumeload;

resumeload:  v from shed;
             sheddable:^ v;
             urgent:^ v;

shedme:      lastshed:=now;

             remove me from sheddable;


unshedme:    lastshed:=null;
             sheddable:^ me
```

7. Semaphores.

Semaphores can be described by a simple class declaration:

```
class semaphore;
  begin  b: Boolean;
         queue: sequence of virtual machine;

    procedure  P;
      if b then b:=false
           else  {queue:  me; relinquish}

    procedure V;
      if queue = empty then b:=true
           else  {t from queue; resume(t)}

end
```

Of course, the bodies of the procedure, P and V on the same semaphore must not be interleaved. This can on most machines be implemented by a hardware "test and set" instruction on b; otherwise the hardware mutual exclusion must be set to protect every P and V operation.

8. Adjust priority.

The objective of the adjustment of priorities is to obtain overall satisfactory service for all user virtual machines. We therefore postulate for each virtual machine a minimum rate at which it requires to be run. If in the recent past it has fallen below this rate, it is given high priority, and if it has recently progressed faster than this rate, it is given low priority. Thus in the long term, each program will proceed at a speed roughly proportionate to its required rate.

We assume that the routine "adjust priority" is entered every Q (say 2000) instructions of a virtual machine, and that each actual machine takes M milliseconds to execute Q instructions. Suppose that a virtual machine wishes to be run at a rate not more than R times slower than this, i.e., each quantum of Q instructions should be executed within an interval of RM milliseconds.

We can now calculate the target real time by which the next Q instructions should be executed

target:= now + RM.

But if the previous target was not met, this target should be made earlier to enable the program to catch up; and if the Q instructions were completed ahead of the previous target, the next target should be made later, to give other machines a better chance. Thus we use the formula

target:=(target+now)/2 + RM;

We now set the priorities proportional to the target, so that the earliest target has highest priority. Since"now" is monitonically increasing, care must be taken to deal with the occasionnal possibility of overflow.