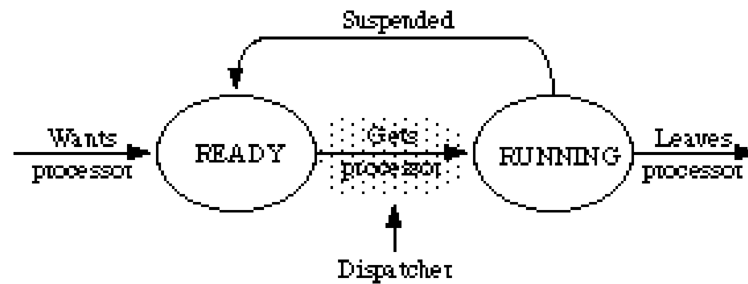
DISPATCHERS

We remark before embarking on this discussion of dispatchers that our view of the relationship between dispatchers and interrupts is a little different from that found in most accounts of this part of the scheduling system. We have regarded the dispatcher's job as that of allocating a processor to a process when the processor becomes available, but have excluded any involvement in causing processors to become available. This we have placed firmly in the area of the processes themselves (which can terminate, or request services which lead to their suspension) and the interrupt system (including the clock interrupts). This distinction seems to us to give a much clearer and more rational separation of tasks between different system components.

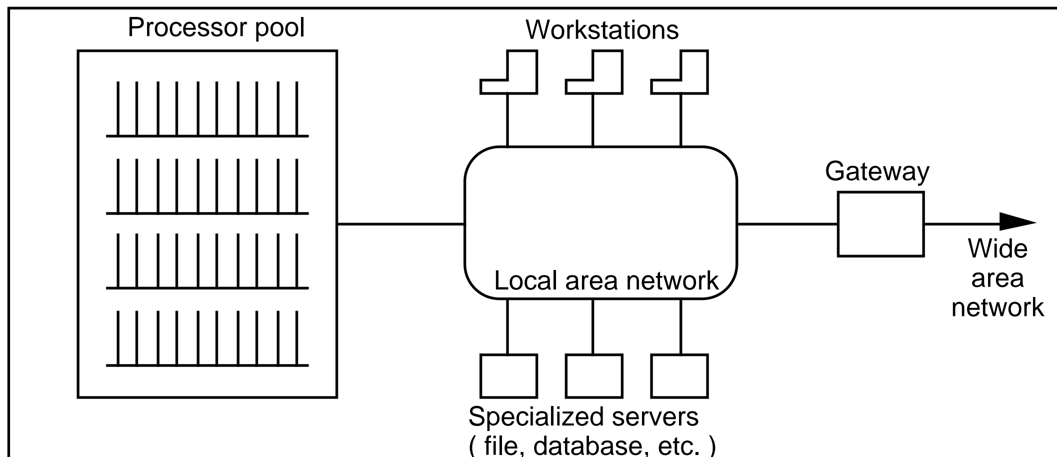
We therefore regard the dispatcher as the system component which administers the *runnable* state of the process state diagram which we met in the chapter *PROCESSES IN ACTION*. We divide the *runnable* state into two substates, *ready* and *running*, related as suggested by this diagram :



A process which is made *runnable* first enters the *ready* state; it waits there until the dispatcher gives it a processor (or gives it to a processor, depending on your point of view), when it starts *running*. The process might stop running because it can't continue at the moment, when it leaves the *runnable* state, or because it is temporarily suspended, usually in response to an interrupt of some sort, when it remains *runnable*.

WHAT A DISPATCHER DOES.

The nature of the dispatcher's job depends on the balance between processes requiring service and processors available for service. In a single-processor, or tightly coupled multiprocessor, system with more processes than processors, the dispatcher is used when a processor becomes vacant to allocate the processor to some waiting process. There is no need for a dispatcher as such in a system with a superfluity of processors; under such conditions, processors can be regarded as just another resource, like memory or disc space, and can be allocated by ordinary resource allocation methods. The Amoeba system[REQ15] is an example; it is designed as a distributed system in which any process can use several processors, so it is equipped with a large number of processors available for allocation. The main features of its architecture are shown in this diagram :



The special feature of traditional dispatchers is their own use of a processor which it is their job to allocate. When the dispatcher is executed, it must decide which process to run on the vacant processor, make appropriate modifications to the process table, which includes the ready queue, and restore the state of the new process and set it going. The processor can give a lot of help with this operation; generally, the more recent and the more elaborate the processor, the more help it will give for process switching and dispatching.

A dispatcher is not the only way to switch processes. The original spooling systems relied on interrupts to switch between the three (or so) processes involved; but this will only work if all processes but one are entered by distinct interrupts, and the process entry point is all that is needed to effect the process switch. The same sort of principle was used by early systems which operated foreground and background jobs, and has resurfaced in some microcomputer systems. Something like a dispatcher becomes necessary when the system runs many processes, and depends on a process table.

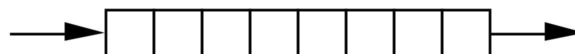
The "desk accessories" in the Macintosh systems are an interesting example. They are typically quite small utility programmes which run permanently in the system and can be used at will to perform little tasks, maybe while executing another programme. The early Macintosh operating systems were not multiprogramming systems, so it was necessary to use special means to execute them – typically an explicit choice from a menu. Version 7 of the system comes much closer to conventional multiprogramming, and the distinction between desk accessories and ordinary programmes has almost disappeared.

HOW THE DISPATCHER WORKS.

The dispatcher's job is to organise and manage the *ready* state. It must take a process from some pool of ready processes, and to set the process running on an available processor. The two components of this operation are, first, selecting the process, and, second, setting it going.

Selecting a process to run.

The simplest sort of dispatcher is the "round robin" single-queue dispatcher. All ready processes are attached to the end of a single ready queue, and the dispatcher always removes the process from the head of the queue and sets it running. This is simple and straightforward, and it works. Without further elaboration, though, it makes no distinction between different processes – which is fair in a way (equality is not often really fair*), but only satisfactory if the processes are all similar. If not, we need a way to select processes based on their priorities.

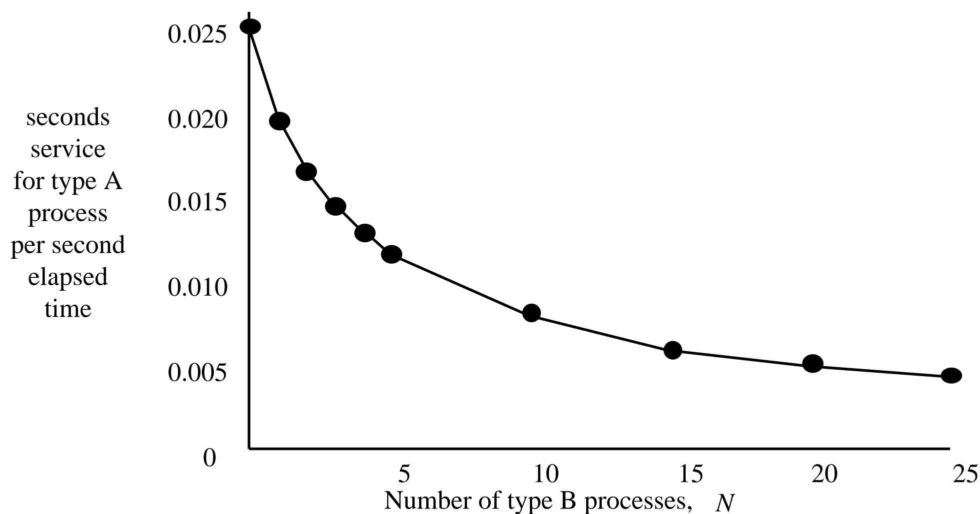


Unfortunately, if all we have is the dispatcher, we can only do so rather imperfectly. That's because of the design of the processor. The conventional computer is designed to execute code automatically, proceeding from one instruction to the next in a well-defined way, and it will keep on doing so until the instructions it is executing continue to be executable, or until someone switches off the power. (Recall that a computer can only either execute a programme, or break down.) Therefore, once a programme gets hold of the processor, it can in principle keep hold for ever. That isn't

* - Two "equal" entities are either identical in all respects, in which case they are receiving identical treatment, or only identical in some respects. In this, more common, case, equal treatment based on the equal attributes is quite likely to be "unfair" when viewed in terms of unequal attributes, and vice versa.

automatically a bad thing : if that programme is the one we want to run, then the longer it keeps doing useful work the better. On the other hand, the fact that we are discussing dispatchers suggests that there are other things which we want to do as well, and under these circumstances the potential for entrenchment is unwelcome.

A process can be disruptive even without monopolising the processor for ever; we'll illustrate the problem with a grossly oversimplified example. We've seen that it's common for processes to run for a while, then to have to wait for some resource which isn't immediately available – more input, or virtual memory, or whatever. The running period is sometimes called a *processing burst* (as opposed to an *input-output burst*). Suppose we have processes of two sorts in a system. Processes of type A (for Admirable) have processing bursts of 1 millisecond, while processes of type B (for Burdensome) have bursts of 10 milliseconds. There are altogether 40 processes in the system, which uses a simple round-robin dispatcher. The graph shows how the proportion of the time available for one of the type A processes changes as the proportion of type B processes increases.



It's interesting to interpret the graph as an indication of how the service given to a type A process is affected as more and more type A processes turn into type B processes. If we start with 40 type A processes, then if a single process changes from type A to type B the service to every other type A process drops to 80% of its previous value. After one tenth of the processes have changed to type B ($N = 4$ in the graph), the service given to the remaining type A processes has decreased to only slightly more than half (52%) of its original value.

This was a well known phenomenon in overladen time-sharing systems; all worked well while everyone was editing source programmes, but as soon as someone started to run a compiler everyone knew, and by the time a few people had got that far the rest might as well go home. We hope it doesn't happen much any more – but very similar things are likely in any system where a resource is fully used : sudden surges of demand for computer networks can make them perform very badly for short periods.

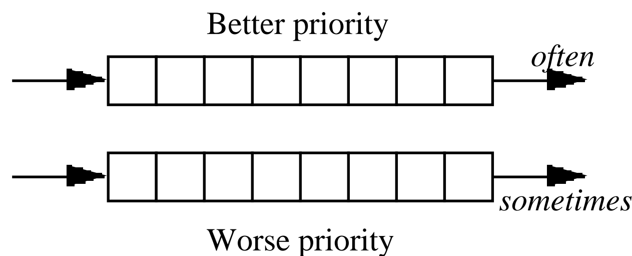
We have deliberately made the most of our example to emphasise the point. While the behaviour we've described can and does occur, the details depend on circumstances. Suppose that the A processes spend their time polling terminals to see whether anything has happened; then, provided that they poll sufficiently frequently (say, a few times per second), all will be well. Even when half the processes in our example are of type B, every process is served about 4.5 times each second. That might be quite adequate for the

purpose – if so, there is, in effect, no degradation of performance. Nevertheless, the potential is there, and we would like to do better if we can.

Notice an interesting fact. Here we have two interpretations of the same condition : one concludes that the performance has been reduced to 20% of the original, while the other concludes that there is no change. This is one of the reasons why it is extraordinarily difficult to measure useful system performance by looking at what's happening in the system.

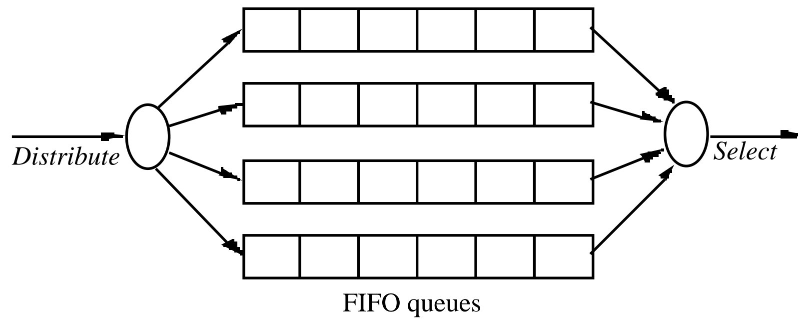
We can do a lot better when we get interrupts as well – see the chapter on *INTERRUPTS* – but, until then, the best we can manage is to control the probability with which processes will be selected, and the easy way to manage this is to use a dispatcher with several queues. The multiple queues are not wasted, for we shall use these again with the interrupts. (Notice that we *don't* want to start searching through a queue for better-priority processes when we run the dispatcher – that just wastes time. Instead, we check the priority when we make the process *ready*, and select an appropriate queue at that time.)

Suppose we decide to use two queues, which we designate worse-priority and better-priority. We can make the priority effective by building the dispatcher to select a process from the worse-priority queue only if the better-priority queue is empty. This certainly takes note of the priority; unfortunately, it might do so too well, as a steady supply of better-priority processes might block the worse-priority queue for ever, resulting in a phenomenon known as *starvation* (or sometimes *livelock*) : processes which are ready and willing to run never do so, because they are never given some resource which they require – in this case, a processor. Notice that this runs counter to our intention, as proclaimed in the chapter entitled *WHAT'S AN OPERATING SYSTEM ?*, to construct *functional* systems.



An alternative is to dispatch one process from the worse-priority queue for every, say, ten from the better-priority queue – or, of course, if the better-priority queue is empty. This is a little more expensive in dispatcher processing, but it ensures that all the processes will run eventually, so satisfying our requirement for a functional system as well as observing priority.

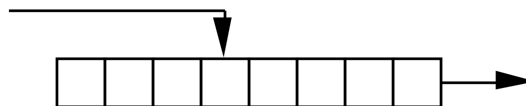
More generally, we can use a multiple-queue system. We identify certain classes of process which we think will be useful in managing the scheduling, and we set up a queue for each. As processes enter the ready state, they are assessed by a classifier algorithm, and allotted to the appropriate queue; when a processor is free, a queue is selected according to some suitable selection algorithm, and a process from that queue is dispatched.



The queues group together processes with similar characteristics, and queue attributes define the nature of the service which they will receive. For each queue, it is possible to define the frequency with which processes are selected, which might depend on the occupancy of other queues. The distributing algorithm must be able to classify the process in a significant way, typically relying on indications of the process's priority and recent performance.

As with any component of the dispatcher, it is important that the algorithms used for distribution and selection should be very fast – if they're not, we might lose more time in administering the dispatcher than we gain by the clever algorithms. The distribution algorithm, in particular, does not have time to go searching for information about the processes, so it is important that any information required by the dispatcher should be kept in the process table. (More precisely, it should be kept in memory in a readily identifiable place – but that's the whole point of the process table.) The multiple-queue method is a way of achieving this optimisation, for by assigning a process to a particular queue the system is in effect remembering that the process has certain characteristics, so doesn't need to work that out again.

We can achieve a similar result with a single queue by using a priority queue, in which processes are kept in order of priority, and new processes are inserted in the appropriate place. To keep the system functional, the priorities of all queued processes are uniformly improved with time. This takes a little more administration, but has the advantage of treating all processes in the same way.



The effect of these more elaborate queue organisations is worth a little thought. While two of them certainly guarantee a functional system, they also change the meaning of "priority". It is no longer true that a process of better priority will unconditionally take precedence over a process of worse priority; all we can say now is that a process of better priority is less likely to experience long delays. The queue disciplines themselves give no guarantee about the absolute length of the delay, which, in the worst case, can increase without limit. Starvation might have been eliminated, but something might well get very hungry indeed.

That is because any system which exhibits starvation as we have described it above is overloaded. If the worse-priority processes never run, and the processor is always busy (which it must be, or the worse-priority processes could be dispatched), then there is simply too much work to process, and the effect of a clever dispatcher is occasionally to increase the length of the better-priority queue to run a worse-priority process. The only sensible answer is to get an additional (or a faster) processor.

The best we can do with a clever dispatcher is to even out fluctuations. We can prevent worse-priority processes waiting all day until there's a lull in demand overnight, but only at the expense of delaying some better-priority processes, which doesn't make much sense if priority is to mean anything at all. Eventually the processor time must be found, and that can only be done by either waiting for free time on the processors you have or by getting more processors.

NOTE ON ROUND ROBIN.

A round robin is a petition in which all the signatures are arranged in a circle so that no name stands out as head of the list and therefore, perhaps, ringleader. (No pun intended.) We have absolutely no idea why this name has become attached, quite inappropriately, to dispatchers.

Making the process run.

Once the process is selected, the rest is easy. The dispatcher must carry out any necessary housekeeping on the ready queue (such as detaching therefrom the process to be executed), mark the new active process as running, set the new process's register values in the machine registers, and finally branch to the appropriate location within the new process's code.

All that is perhaps obvious enough; it is nevertheless important, because it tells us that all those required data must be readily accessible. The register contents must therefore be saved either in the process table, or in somewhere readily accessible from the process table – such as the process's memory area. It is not uncommon for processors nowadays to have an instruction which dumps the contents of their registers onto the current process's stack, and another to reverse the transfer. This is another example of an operating system function which has migrated from software to hardware.

Making the process stop running.

This is not really the dispatcher's job, but after thinking about how to set the process going it's reasonable to ask how to stop it so that another process can be dispatched. Many processors give up the processor voluntarily quite frequently as they make system calls for various sorts of service, and once the system has been called it can suspend the process if that's desirable. But how can you deal with a running process which doesn't make any system calls for a long period ? The simple answer, for a simple single-processor computer without any refinements, is that – short of turning off the power – you can't. Once a process has acquired the processor, it is in sole charge of what happens, and it will keep running until it decides to stop. This is not necessarily bad : if the process is doing useful work, it is using the processor at 100% efficiency. Unfortunately, it is not necessarily good either : if the process is faulty and is stuck in a loop, it is using the processor at 0% efficiency. Even if it isn't faulty, it isn't very satisfactory if a low priority process once running continues to run for the next three hours, while more important processes wait.

We are not the first people to notice this, so it will come as no surprise to learn that (we think) every practical processor is provided with a way out of this position. That brief discussion was not wasted, though, for it leads us to an important conclusion : in order to guarantee that we can escape from the running process we must have some *independent* process to get us out. There must be some stimulus from outside the processor which will direct it to start executing some other code – and that is what we call an *interrupt*. In a multiprocessor system, the interrupt can be caused by another processor, but we don't need anything so complicated : someone who presses a button can be just as effective, and is indeed an independent process, just as we wanted. Other sorts of machine which satisfy our requirements are peripheral devices which need attention and – very important – the system clock. We discuss interrupts in more detail in the chapter *INTERRUPTS*, but for the moment notice that every one of these sources of interrupts is an independent process of some sort.

Which is all very well, but if that's the case why do so many computer systems of all sorts still contrive to get themselves into positions where a process can get into a loop and force you to restart the whole system ? We confidently assert that the only reason for this sort of behaviour with modern machinery is bad system design. It is always possible to provide some interrupt button which can interrupt the current process and return control to the system; if you reply that this might be dangerous if the rogue process has corrupted

the system, then we ask why there was no proper memory protection. And so on. It really is possible to design very reliable systems; it is sad that suppliers, by and large, choose not to. It is sadder still that their customers put up with it.

Another Macintosh example illustrates the point. In the early and not so early versions of the Macintosh software, it was fairly common for the process you were running to seize up and stop responding to any normal input. There was nothing you could do except restart the whole system. In recent versions, programmes can still get into a catatonic state, but now you have a good chance of escaping by simultaneously pressing the , option, and escape keys. That stops the running programme, but doesn't interfere with the rest of the system. You are not supposed to ask how it fits in with the Macintosh principles of intuitively obvious actions, though as the intuitively obvious action is to put a hammer through the screen that's perhaps just as well.

COMPARE :

Lane and Mooney[INT3] : Chapter 6.

REFERENCES.

REQ15 : S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, H. van Staveren : *IEEE Computer* **23#5**, 44 (May 1990).)

QUESTIONS.

Does the argument about the usefulness of clever dispatchers change if some of the processes have absolute deadlines rather than priorities ?