INTERRUPTS

We met interrupts before in the chapter *COMMUNICATION BETWEEN PROCESSOR AND SYSTEM*, and we have frequently had occasion to mention them throughout this section on *MANAGEMENT*, which perhaps demonstrates their value in implementing the system. (It could also demonstrate their nuisance value in being an exception to many rules, but we shall not pursue that line of thought.) Now we shall deal with them in their more general aspects. We saw that when a processor receives an interrupt, it is switched from the process it is currently executing to a process identified by the type of the interrupt. This transition is quite automatic and doesn't involve the operating system; but the system has to be concerned in some related operations, of which we shall inspect two in a little detail : setting up the interrupt table, and using the system clock interrupts.

First, though, we must describe the interrupt handling itself. Notice that, so far as we're concerned, the interrupt *is* the process switch. What happens after that isn't part of the definition; it's determined by the code of the interrupt handler, which is at the disposal of the programmer. There are two common cases :

- Very little processing is required – a character is read and stored, or a flag is set somewhere. There need be very little interference with the running process, with no extensive context switching. This is the "unexpected procedure entry", so called because the only noticeable change is in the addressing space used.

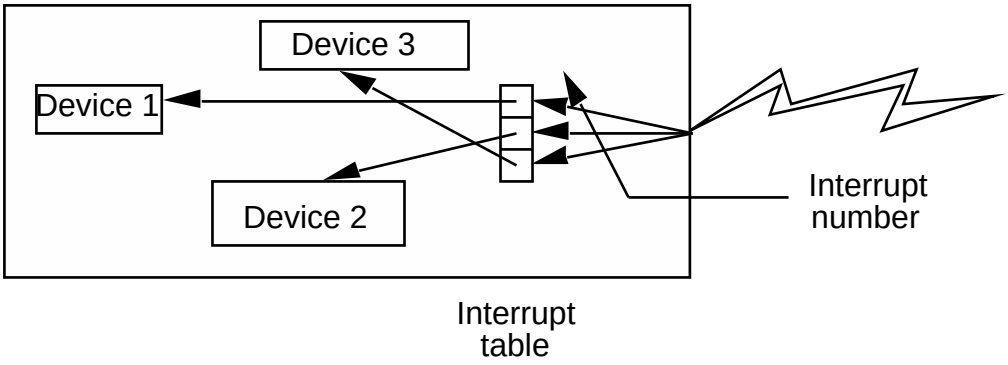
- The interrupt is used for process management, and considerable interference with the running process (typically its suspension) is expected.

(Interrupts are often discussed in literature about MS-DOS. These are usually not the general interrupts we describe here, but software interrupts implementing what we have called supervisor calls or system calls. The MS-DOS software – particularly IO.SYS – is intended to insulate you from the real hardware, in effect providing a virtual machine, so that MS-DOS is in principle transportable from one processor to another without requiring any modification to programmes; that being so, it can hardly take note of such processor-dependent features as specific hardware interrupts.)

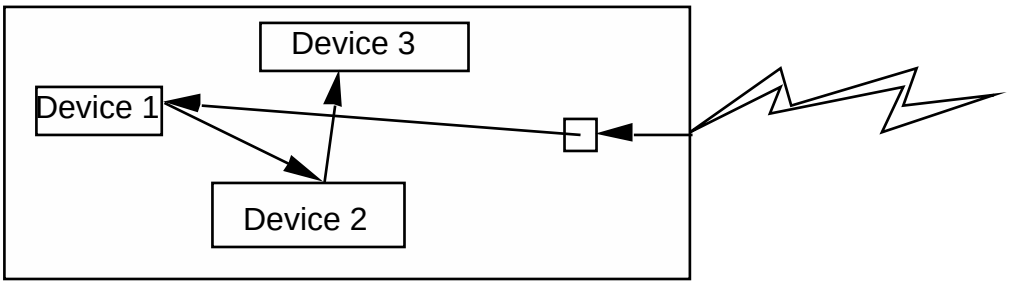
GETTING THE INTERRUPTS TO THE RIGHT PLACE.

An interrupt arrives at a processor as an electrical signal from outside. Within the processor's memory, there are typically several interrupt-handling processes at arbitrary memory positions. Somehow, the interrupt must be interpreted so that the correct interrupt procedure receives the signal. How can this be done ?

Just how this is accomplished varies from processor to processor, but there are two main methods which differ primarily in the extent to which the processor itself can differentiate between different interrupt signals. In a processor which can discriminate, it is usual to organise interrupts into different priority levels. In the simple case, there is one device associated with each interrupt level, and therefore a different interrupt handling procedure for each level. The device must somehow communicate its level to the processor, which can then select the correct entry from a table of addresses for the interrupt-handling processes, and branch to that address. It is the operating system's job to ensure that this interrupt table is properly set up and maintained. This might be managed by special software used when any system component requiring interrupt services is installed, or it might (less satisfactorily) be managed by documenting the standards required and relying on people to conform. (They usually do : this is one part of the system which *must* work.)



A less accommodating processor might provide only two interrupt levels, or even only one. (These are usually equivalent, for if there are two it is common for one to be identified as a "non-maskable interrupt" which the software cannot switch off, and which is reserved for emergencies such as power failures.) In this case, the software will have to take some action to determine how to deal with an interrupt when it happens. A common method is to use a "daisy chain"; the interrupt is sent to each interrupt handler in turn until one accepts it. Clearly, the interrupt handlers must apply some sort of test, which might be to interrogate its own device to see whether it has emitted an interrupt.



Each interrupt handler must execute some code of this form :

```
Read my machine's status;
If it is "waiting for response to interrupt"
then deal with the interrupt
else wake up the next procedure in the chain.
```

This works simply and straightforwardly provided that all the procedures behave themselves, but it's very sensitive to misbehaviour. It's also comparatively slow, though unless microseconds really are important the slowness is usually tolerable.

The two methods are not mutually exclusive. In a system with several levels of interrupt and a large number of devices to manage it is likely to be impossible to provide a separate interrupt level for each device. In such cases, interrupts which really must be handled quickly are given unique levels as far as possible, and daisy chain, or similar, methods used to share the lower levels between several devices each. However it's arranged, though, the more interrupts one tries to handle with a single processor, the greater the risk of congestion and consequent delays and perhaps missed interrupts. This is one of the reasons behind the use of device controllers which we described in *DEVICE CONTROL HARDWARE*.

It is perhaps rather less obvious that we should somehow make provision for what must happen when the interrupt has been recognised and dealt with. We shall not attempt to present a thorough treatment, as details are hardware-dependent and vary significantly, but two operations must be carried out whatever method is used. These are clearing up any mess left by the interrupt handling, which includes permitting interrupts again if (as is usual) they had been inhibited, and getting back to the interrupted process.

INSTALLING AND REMOVING INTERRUPT HANDLERS.

We have already commented (in *CHANGING THE CONFIGURATION*) on the desirability of being able to add new devices to a computer system and remove them once

there without disturbing the system as a whole, and we particularly noted that the interrupt handlers were part of this operation. Now we know a bit more about them, we can discuss these questions in a little more detail.

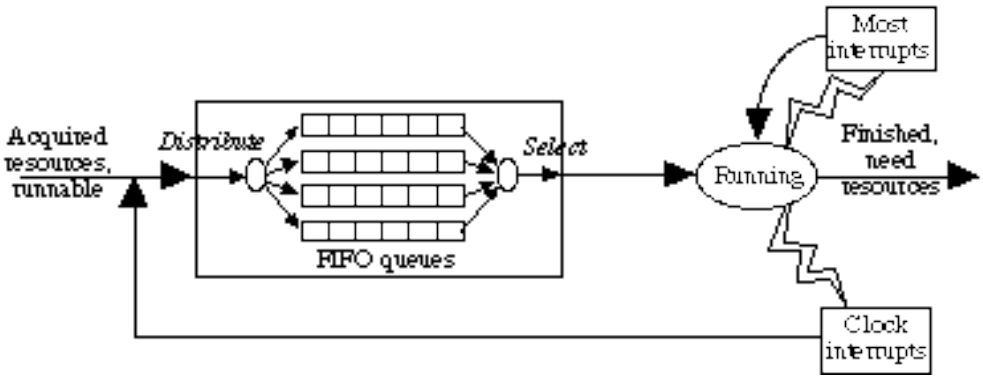
First, then, how do we add a new interrupt handler to a system ? The principle is simple : whatever the design of the interrupt manager, at some position there is an address to which a branch is executed when the interrupt is detected, and the entry address of the new procedure must be inserted at that point. If there is no doubt that the procedure will be able to handle any interrupt which is directed to that position, little else is necessary, and the procedure can end with an appropriate return-from-interrupt instruction. A daisy chain is a little more demanding; in this case the installer must copy the address initially at the branch point into a location, typically within its own address space, at which it can be used to "wake up the next procedure in the chain" as required by the algorithm above; by this means the new procedure becomes the first interrupt procedure in the chain, and all previous procedures are pushed along one position.

Removing an interrupt handler is not so easy, except in the case when only one handler can correspond to an interrupt. In this case, it is only necessary to return the branch address to its original value when no procedure was defined; as this is a standard value, that's easy. A daisy chain poses much harder problems, though. To decouple a procedure from the middle of such a chain is easy in principle; it is only necessary to copy the "next procedure" address from the replaced procedure into the procedure which precedes it in the chain - but how can you find that address ? Unless you impose some sort of discipline, it could be anywhere in the procedure's space.

It is not particularly difficult to invent ways to get round the problem, but the problem's very existence shows that careful system design is essential if smooth addition and removal of device handlers is to be achieved.

PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING.

In the chapter on *DISPATCHERS* we considered the question of how to stop a process once it is running on a processor. We found that you can either wait for it to finish whatever it's doing, or you can interrupt it in some way (which we have now identified with the system clock) and push it off in the interrupt handler. The act of forcibly displacing a process in this way is (mis)called *preemption*, and both preemptive and non-preemptive schedulers can be constructed. The difference lies in how the dispatchers are used. In a non-preemptive scheduler, the dispatcher is only called when a process voluntarily gives up the processor; but in a preemptive scheduler the clock interrupt handler typically, though not necessarily, returns the interrupted process to the *ready* state, then (perhaps after performing other duties not connected with process scheduling) calls the dispatcher to decide what to do next. Notice that it is not strictly the dispatcher itself which determines which regime the system implements, though the dispatcher and the clock interrupt procedure are clearly closely related, and the distinction is often blurred. This diagram illustrates the idea :



NOTE ON PREEMPTION.

The notion of preemption comes from law. A "right of preemption" over property is the right to acquire it ***when the present owner gives it up*** *– colloquially, "first refusal". It has nothing whatever to do with the* ***eviction*** *of the present owner, so is a pretty stupid name for the technique used in scheduling. It would help us all if the semiliterati of computing who determine the vocabulary we are constrained to use would occasionally deign to use a dictionary.*

READY QUEUE STRUCTURE.

The diagram shows a structured *ready* state, with perhaps more than one component queue. We introduced the idea in the *DISPATCHERS* chapter, but we emphasise it here because it becomes much more useful with a preemptive scheduler. (Notice, though, that the dispatcher and interrupt handler are independent entities, so you can use a single-queue dispatcher if you so desire.) The introduction of the clock interrupt means that we are no longer at the sole mercy of the running process. Now, if we think that the service given to processes in the better-priority queue should be, say, ten times as good as that for the worse-priority processes, we can make the dispatcher choose one process from the worse-priority queue for every ten choices from the better-priority queue.

In fact, that's an oversimplification, as many processes don't use the whole of their time-slices; they have completed their immediate tasks before a clock interrupt arrives, and then revert to some waiting state until a resource they need becomes available. For an example of this behaviour, think of a process occupied in reading from a device. It might wait for the next input datum to arrive, then deal with the input, then wait for some more. If the operations on the input take even a few thousand machine instructions, they will be complete in a few milliseconds, and might well require less than one time slice. It is important to realise that *this is what we really want.* That's because process switching is expensive, and we want to do as little of it as possible. While the clock interrupts are useful in making sure that all the processes get a share of the processor (recall our aim of keeping all the processes going), they do take up time; and if all processes would politely hand over the processor to the next in line without the clock interrupts, we wouldn't need time-slicing at all.

SETTING THE TIME SLICE THICKNESS.

How long should the time-slice be ? That depends on several factors, as the length of the time-slice affects several aspects of the system's performance.

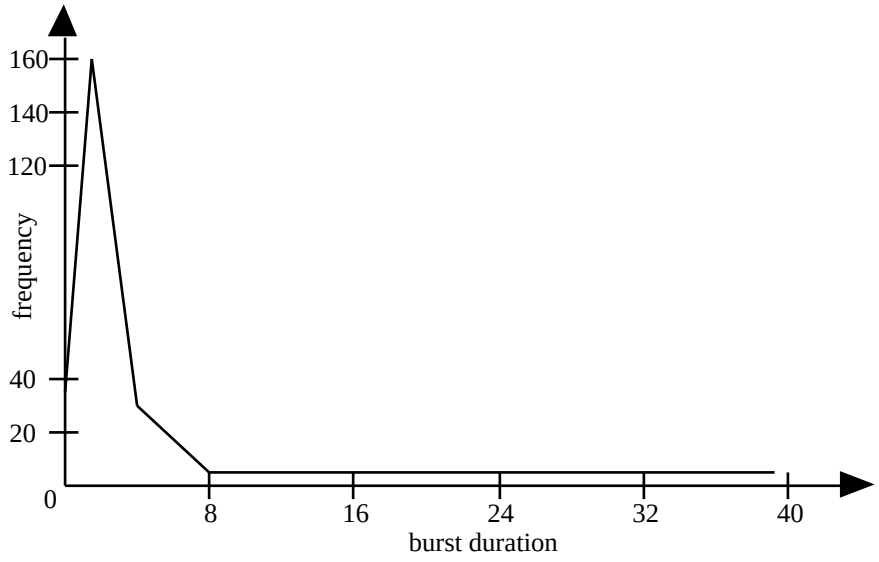
If the time slice is THICK :

- there will be fewer interrupts, so less time is wasted;
- immediate tasks are more likely to complete within one slice, so resources might be released quicker;
- processes receive infrequent service, with long gaps between time slices.

If the time slice is THIN :

- there is much more process switching overhead (because fewer processes voluntarily relinquish the processor);
- processes get more frequent service.

An important determinant of the performance is – as we remarked earlier – the probability with which a process completes what it wants to do within a time-slice. Processes which run for long periods are expensive on process-switching overhead, as they might have to be handled several times by the clock interrupt procedure. The correct choice of time-slice length is therefore dependent on the length of the process's processing bursts. These vary greatly between processes, with some perhaps only storing a character when it's received from a device and immediately going to sleep again, while others might be able to work for seconds or even minutes at a time on calculations. A common form of distribution is illustrated in the graph below[INT4].



For a system with this distribution of burst times, a time-slice of about 10 milliseconds would be appropriate.

With a simple system clock, the time slice is fixed, and we can only hope that the hardware designers worked through the argument we've just given, took some notice, and took the trouble to find out what sort of timeslice would be appropriate. In more favourable cases, though, we might be better off : some processors have an adjustable time-slice, so the dispatcher can then determine the length of the time-slice as it runs each new process. Even without such facilities, it is easy to give a time-slice extending over several interrupts simply by counting – which is why the clock interrupt procedure may not automatically move the interrupted process to the *ready* queue. The interrupt overhead itself is unavoidable, but that is negligible; it is the much more significant time taken for process switching and associated administration which hurts, and that need only happen at the end of the time-slice.

USING THE DISPATCHER QUEUES.

We end up with an abundance of controls with which we can try to implement priority schemes in the dispatcher of a preemptive scheduler. As we saw, we don't want the

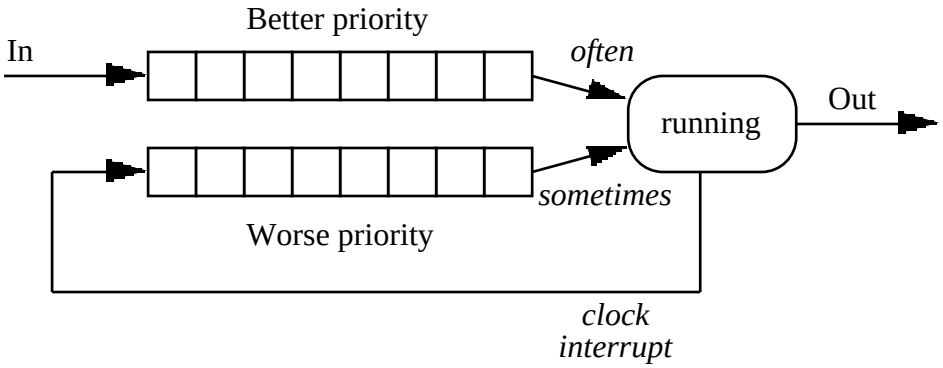
dispatcher to be occupied with much computation, so we attach all the information we need to the queues. There are several ways to manage the details.

- If we want a strict priority system, then we simply label the queues in order of priority. The dispatcher will take a ready process from the best-priority queue if there is one, otherwise it will take a process from the queue of next best priority, and so on.

- If we want a system of relative priorities, in which a worse-priority process will receive worse service than a better-priority process, but cannot be caused to wait for ever, then we can label the queues by their relative frequencies of execution – again making allowance, of course, for occasions when queues turn out to be empty.

- If we also want to control the time-slice lengths, then we can add these values to the queue specifications.

In illustration, consider this typical arrangement. A dispatcher uses two *ready* queues. A process becoming *ready* from a *waiting* state is placed in the queue of better priority, which is inspected more frequently by the dispatcher, but given a shortish time-slice. If a process is still running at the end of its time-slice, it is moved to a *ready* queue by the clock interrupt handler, and we arrange that it should always be put into a queue of worse priority, inspected by the dispatcher only once in a hundred times. This might sound rather harsh; but we soften the blow by giving processes from the worse-priority queue a time-slice ten times as long as that associated with the better-priority queue.



The result of this arrangement is that the "best" processes – those which relinquish the processor before their time-slices are completed – will always return to it from a waiting state, and will always receive good service. In contrast, processes requiring intensive use of the processor will receive one time slice at the better priority, but will then be demoted to the worse-priority queue, where they will receive less attention (perhaps we wish to regard our system as primarily for interactive work), but with fewer interrupts, so they will run more efficiently.

By adjusting the available parameters, it is possible to change the behaviour of the system over a wide range. This is likely to be most effective if the exercise is supported by a good statistical base of the sort of work run on the system, good measurements of the effects of changes, and a clear idea of what the system is supposed to be doing.

COMPARE :

Lane and Mooney[INT3] : Chapters 6 and 8; Silberschatz and Galvin[INT4] : Chapter 5.

QUESTIONS.

Notice our rude comment about the misuse of the word "preemption". What would a *real* preemptive scheduler be like ? Would it be any use ?

Explore possible ready queue structures. Devise a structure which could give fast service to interactive work, fiercely penalise processor-intensive work run from interactive sessions, but permit processor-intensive work to be run as a batch process at medium priority.

Does the introduction of interrupts nullify the objections we raised in the *DISPATCHERS* chapter to clever dispatchers ?
