COMMUNICATION BETWEEN PROCESS AND SYSTEM

TYPES OF INTERACTION BETWEEN PROCESS AND SYSTEM.

In the preceding chapters, we have described various operations which the system must perform on the processes, most of which are invisible to the processes, and indeed are carried out for reasons which have nothing directly to do with a process's own requirements. In other cases, though, the operating system is called upon by a process to perform various necessary functions through system calls, sometimes known as supervisor calls. Now we must discuss them in more detail, for these components of the API are among our basic tools for exchanging control of the processor between running process and operating system, and as such fundamental to process management.

These are not the only interactions between system and process. While *system calls* are intended for planned communication between process and system, we must also provide for at least two sorts of unplanned requirement for system services which might arise while a process is running. These are undesired events detected by hardware or software as the code is executed (usually processing errors of some sort, such as arithmetic overflow, or attempts to address memory outside the process's address space or to execute an illegal instruction), sometimes called *exceptions*, and events which require immediate service originating outside the process (typically in peripheral devices), called *interrupts*. While they are similar in that they are unprogrammed calls on the system, they differ in their relationship with the current environment. The exceptions are connected with events within the execution of the current process, and access to the process's memory areas might be required if any action, including preparing error reports, is needed. The interrupts are not necessarily dependent on the current process for treatment by the system.

Having clarified the confused situation, we now issue a warning : in practice, the whole issue is confused anyway, and we again face the problem of arbitrary nomenclature. Our classification into three sorts of interaction is an attempt to cover the field, but certainly doesn't exclude other possibilities. System calls need not be the same as software interrupts, exceptions and interrupts need not be distinguishable, different sorts of exception (such as those for illegal instructions and reserved instructions) can be directed to different system code, and so on. It is not helpful that the terms themselves are not used consistently. The fact underlying all these methods is that it is possible to provide processor hardware with which the usual sequence of code execution can be suspended in an orderly way to permit other activities to be carried out; given such hardware, it can be used in many different ways. We think that our classification covers the main features.

Further to confuse the issue, the "interrupts" commonly discussed in MS-DOS manuals are system calls. Real interrupts exist, but are normally caught by device drivers in IO.SYS and not allowed out into the rest of the system. That's why manuals contain phrases such as "When IO.SYS receives input from the keyboard, serial A, or serial B device, it generates a software interrupt.". There's a good reason for this design : MS-DOS itself was originally intended to be independent of any specific hardware, and that obviously wouldn't be so if it had to deal directly with such hardware-specific things as real interrupts.

The common feature of these three rather similar sorts of event is that they must be handled by the operating system, either because some active component requires service from an identified operating system function, or because some active component has lost control so that system intervention is necessary. Details of the implementation obviously

depend strongly on the hardware available. Here we shall assume that the types of event can be distinguished (by parameters, flags, direction to different system addresses, etc.), and comment on their subsequent treatment by the system.

One component of the common treatment is essential : execution of the code which makes the call is suspended for a duration while something else happens in a different address space, and it must then be possible to return to the interrupted process after the event has been dealt with. Because of this requirement, the current state of the processor must be preserved, in much the same way as for a procedure call. Indeed, though we shall draw some significant distinctions shortly, interrupts can often be thought of as unexpected procedure calls. Consider this comparison :

Procedure call	System call	Interrupt	Exception
Initiated by the process	Initiated by the process	Initiated by something external	Initiated by the system after process misbehaviour
The same thread continues	The same thread continues ?	The thread is interrupted	The thread is interrupted
The new address space belongs to the same process	The new address space belongs to the system	The new address space belongs to a process specified for the interrupt	The new address space belongs to the system, but the process's address space might be relevant

One can debate the suggestion that "the same thread continues" after the system call. The argument is that the processor starts work on code which, though in a different place, is still concerned with something required by the calling process, so maintains the essential continuity – in effect, the system call is just a way of having one of the process's elementary tasks performed somewhere else. On the other hand, one can equally argue that, in some cases at least, the system call is a branch to another process which has been waiting until it is required, and has perhaps been in existence since before the running programme started. This notion might seem more reasonable if you consider that a system call is the way into client-server services and remote procedure calls, both of which are ways of carrying on the same process somewhere else.

In the case of a system call, then, does the same thread continue or doesn't it ? The answer is, unfortunately, yes, but there is a clear difference between the two possibilities : the thread of continuing computation which might be thought of when the programme was designed does continue, but the thread of execution which is of concern to the operating system is likely to switch to a different process. The uncertainty is a symptom of the two views of programme execution which we mentioned earlier. As our concern is now at the level of executing the programmes, we shall take the second view henceforth.

SYSTEM CALLS.

We already know something about system calls. We first met them (in the chapter *ONWARDS AND UPWARDS – THE OPERATING SYSTEM*) as a means for processes constrained by memory protection methods to gain access to operating system functions. They were developed when it became clear that ordinary programmes could not be permitted to have uncontrolled access to the operating system, and depend on special hardware for their implementation. They constitute an essential part of any protection or security system (see the chapter *SAFETY)*. After a system call, the processor has been switched from its *normal mode* of operation to a privileged state which we called *supervisor mode*, in which certain processor instructions which are inaccessible in normal mode can be executed. Other names for these two processor modes are *user* and *kernel* modes, or *problem* and *supervisor* states.

It is common to implement system calls in the API so that they can be used just like calls to external procedures from the high-level-language programmer's point of view, in accordance with the suggestion that the same thread continues. In practice, this might be yet another bit of the system illusion; so far as the mechanism is concerned, it is frequently more convenient to operate system and process as coroutines. This is certainly a more realistic view from the operating system's side, as it makes sense of system calls which never return (because something was found to be amiss during the execution), or don't return for a very long time (because the process has to wait for a file to be opened, or a swapped-out page of memory to be fetched), or return twice (**fork( )**).

We saw that a system call must do two things : it must cause a branch to an area of operating system code, to carry out the request of the programme, and it must simultaneously change the mode of the processor so that the privileged instructions can be executed. Most of the more ambitious processors have an instruction designed to perform this task. This instruction is variously called an operating system trap, software interrupt, supervisor call (SVC), or change-mode-to-kernel (CHMK), and doubtless other names will evolve.

If the system call is to be fully secure, it is essential that the operating system must take control immediately after the branch, so the hardware is normally constructed to direct all these branches to the same place. System code at that location receives all the branches, and passes on responsibility for handling the requests executed after the branch to appropriate parts of the system as determined by parameters passed in the ordinary way, or set in processor registers. We shall call this code which protects entry into the operating system the *system call interface*.

EXCEPTIONS.

We have already seen how in principle such faults should be handled to convey useful information back to the user (*SOURCES OF INFORMATION*); now we should explain how to make it work.

We are not going to do so, because we don't know a general answer. Ideally, as we said before, the exception should be reported in terms appropriate to the programme in which it occurred, and therefore likely to be comprehensible to anyone using the programme. Just how to do so evidently depends on the nature and structure of the programme, so is outside the scope of the system. All we can say is that it should be possible for the programme to provide code which will be executed if an exception occurs. It is probably (though debatably) preferable that the system does not transfer control back to the process during exception handling, to avoid the possibility of the process's promptly repeating the exception and initiating an error loop which is very hard to break.

In practice, the usual system response to an exception is the simplest : stop the process. As an exception usually means that the process has gone wrong, that isn't as unreasonable as it sounds. It becomes even more reasonable when one reflects that very few programming languages provide you with means to deal with exceptions in a sensible way. It would obviously be appropriate for the system to give any information which might help to determine what had gone wrong, such as informative error messages (essential, but see above) and memory dumps (the other extreme, but useful if you have the understanding or software to interpret them).

More sophisticated approaches are possible. For some programming languages – PL/I and Ada, for example – they are necessary, for the designers of these languages have provided means of encoding responses to exceptions. The compilers for these languages must be able to generate links from the operating system's exception handling components to code prescribed in the programme. Exceptions are also defined in Java, though as Java can be executed with a virtual machine the exception management can be handled without explicit system help – but even so it becomes much simpler if the system already provides effective exception-handling procedures.

INTERRUPTS.

An interrupt causes a switch to supervisor mode and a "procedure entry" to an interrupt handling routine. Interrupts are useful to ensure that the processor can react to external events. They are commonly used for switching a processor to code which handles peripherals, and also, in the special case of the clock interrupt, to allow the system to determine whether a processor should be reallocated to a different process.

We shall discuss interrupts and their influence on scheduling in more detail in the chapter *INTERRUPTS*, but it is convenient to discuss here a question of principle concerning the interaction between interrupts and processes. If you refer to the table above, you will see that the unique feature of an interrupt is that it is not in any way originated by the running process. The question is, therefore : how can the system manage the potential conflict of interests ? Relevant issues include how to minimise the disruption caused by the interrupt, how to ensure that security is preserved, and – on a more mercenary level – who will be charged for the time taken by the interrupt. The common feature of these concerns is the possibility of the interrupt code using resources – time or code – belonging to the interrupted process.

When interrupts were first invented, such questions didn't arise; without memory protection, anything running in the system could address any part of memory, and multiprogramming was rare. The interrupt saved the current execution address somewhere, and replaced the processor's address register with the entry point of the interrupt procedure. It was then up to the interrupt routine's programmer to make sure that the interrupt code did no harm, which usually meant that the contents of any processor registers used had to be saved and restored before returning, but apart from that the processor could just execute the interrupt-handling code and then return. As only one process was running, the interrupt time could reasonably be charged to that process. Further, no process stack was recognised by the system, so changing the code address and other registers really did destroy the significant connections to the process's address space.

With a shared system using memory protection and possibly a process maintaining a process stack and using memory redirection through page or segment tables, things are not so simple. The interrupt which occurs while process A is running might not be doing process A's work, so perhaps it should be charged to someone else; it is not self-evident that the interrupt code will be within A's address space; even if it is, do we really want the code that deals with B's interrupts to have unhindered access to A's data ?

There are various possible answers to these questions, and in practice different systems use different methods. If we are really very concerned about security, then we can make sure that every interrupt is dealt with by a heavyweight context switch to a special interrupt process, devoted to that interrupt. That's quite a straightforward thing to do; it's æsthetically satisfying, too, as the separate functions are thoroughly decoupled, it's very good for security, and it's hard for anything to go wrong. Unfortunately, it's also quite expensive; it means that for every interrupt there must be two full context switches. For that reason, it's rarely done unless security is a prime issue.

If we don't want to switch contexts, we are stuck with process A's context, and we have to rely on the interrupt code to behave itself. We also have to get to the interrupt code. We can do so directly if the interrupt code is accessible from A's addressing table, as in systems using memory redirection in which operating system code is automatically shared between all processes, or if the processor has an absolute addressing mode, which is more likely in systems without memory redirection; in either case, we must make sure that access is only permitted while interrupts are being dealt with. In other cases, the interrupt mechanism must provide for the changeover. However it's managed, much of A's process structure remains undisturbed, and to that extent we can say that the interrupt is handled in the context of the current process.

And, that being so, A's owner will be charged for the time consumed by the interrupt. Does it matter ? There are two reasons why we shouldn't worry too much about it. First, in most systems it makes sense to think of interrupts as a sort of steady background noise. Almost all ordinary processes use them for input and output, for

memory management, and so on, and the cost in time to your process is more or less balanced out by the cost of your time to other people's processes. Second, in any well constructed operating system, the time taken by any interrupt service routine is kept as short as possible, because it is unwise to have a processor running in an abnormal state for an extended period. The load is therefore minimised.

DEALING WITH PROCESS INTERRUPTIONS.

While all these forms of event must be handled immediately, they have slightly different requirements which affect the detail of the techniques used to deal with them. As well as that, different operating systems use different methods, and vagaries of processor hardware might also determine what can and can't be done.

Nevertheless, there is a widely adopted sequence of operations which is close to inevitable after an interrupting event, and we shall describe this sequence, with notes on variations or examples where it seems appropriate. We'll stress the system calls, because that is our main topic, but bear in mind that similar considerations apply throughout.

0 (or –1, if you prefer, because this is not strictly *after* the interrupting event) : The event occurs. The process takes no deliberate action for an interrupt or exception, but it must ask for a system call. In Unix, it may use a function from the C library; in MS-DOS, an assembly language INT directive. Generally, there might be parameters to transfer. These might be left on the process's stack, or in machine registers, according to the conventions established by the operating system. Operating systems commonly provide procedures which look after the details to assist programmers in making the system calls correctly.

1 : The software uses some of the information passed to find the code which must deal with the call. For interrupts and exceptions, the appropriate device or exception handler is identified; for a system call, the nature of the call is determined and parameters transferred.

2 : Unless the hardware is otherwise designed, the processor registers are still pointing to the interrupted process. This might or might not matter. For an interrupt we might wish to enforce a context switch to preserve security, while for an exception it might be desirable to retain the environment to assist in diagnosis. With supervisor calls a switch in context to some suitable system code is usually desired. The context change is the most complicated case, so we'll assume that's what we're talking about from here on. We therefore suppose that the code of the system call interface requests the process manager (the dispatcher, of which we shall see much more later) to suspend the current process and to cause the processor to continue with the system code executing the required system call. The system call interface can also do whatever is needed to convey any parameters to the system code.

3 : The current context of the process is saved in the PCB, thus suspending the process. It can now safely be left alone for as long, or as short, as necessary. A branch to the system call code is executed. Executing the system code might change the saved context, for example, by storing a returned value into some saved registers. If the system so decided – for example, if the event was an exception marking a fatal error, or if anything goes wrong while executing a system call – the process might never be restarted. That's one way to traverse the link from *waiting* to *finishing* in the process state transition diagram; the process is tidied up and dismembered as we have already described. We shall continue on the assumption that this does not happen.

4 : Eventually – perhaps immediately, perhaps when requested resources have been made available, perhaps after other processes have had a turn at running – the suspended process is resumed. A system call is commonly made by a system library procedure, so the process is likely to restart in the library procedure code.

5 : The library code will then look after the returned values, including any error indicators, in some sensible way, after which it will return as an ordinary procedure to the code which called it. In systems in which you can make system calls directly as software interrupts or some equivalent, the programmer must make provision for dealing with the returned state. For a low-level system, there might be conventions which determine how results are returned through processor registers; at a higher level, a procedure might be included in the API which deals with the nasty bits and presents a respectable high-level procedural face to the world.

This is the simple case where no interrupts have occurred during the process interruption. We have also glossed over many different places in which errors can occur. For example, the system call interface must be constructed to take great care in checking the parameters which are passed, for they might be used in a privileged environment when the normal checks are not applied. Under these circumstances, an invalid address used as the parameter of a **read( )** system call could cause data to be read into some portion of memory outside the calling process's address space.

MEANWHILE, BACK AT THE OPERATING SYSTEM ....

We've seen that a process making a system call is likely to be suspended while the system carries out its part of the task, and that all manner of things might happen before it is restarted. What is the system doing between processes ? Should we regard the system code as a process ? – or is it better to think of it as something special, essentially an extension of the hardware necessary for keeping the system going ?

Which brings us back once again to our search for a definition of a process, but now we can do rather better. That's because we have invented the process table, in which – by definition – there is a process control block for every process. Our definition has been established for us ! – for now all we need say is that a process is that which is defined by a PCB, and our definitions will necessarily be consistent. (In small or old systems which don't have recognisable PCBs, we'll have to fall back to our vaguer ideas, but in practice we don't much want to talk about processes in these systems, so we can live with it.)

That's almost cheating, but not quite. What we've done is to use a fairly vague notion of a process to help us define something quite like the vague notion, and we've constructed PCBs with which we can keep track of an activity which is pretty clearly a fairly ordinary specimen of the rather vague idea which started us off. Experience has shown that this is really a very useful approach – so now we're tidying up our nomenclature so that the idea of a process is much more clearly defined.

This view helps us to answer our question. No, the system's administrative activity which follows a system call isn't to be regarded as a process, because it isn't subject to the disciplines we expect of a process. In particular, it can operate on processes (changing their PCBs, for example) in ways which are quite abnormal. We emphasise that here we are discussing only the strictly managerial operations with which the system looks after the switch between processes, and any other operations required to keep the system running; we do not include any work which might be done to continue the "thread of computation" which we introduced earlier, and which we agreed to regard as a different process.

Of course, there is no reason why our agreement should be binding on people who write operating systems, who might choose to carry out some comparatively simple request without setting up a process to do it. This blurring of the arbitrary boundaries used to be quite common, but is increasingly seen as a nuisance. This is one of the reasons for the movement towards microkernel systems which we mentioned earlier. One of the features of a microkernel system is that as much as possible of the system code is executed as more or less conventional processes; they might be highly privileged, or otherwise have special attributes, but they are executed like any other process.

QUESTIONS.

How would you go about providing means for a process to handle exceptions ? Bear in mind that the code to handle the exceptions must be executable when the exception happens, and that the system must not lose control of subsequent events.

What process is running when interrupt code is being executed ?
