COMMUNICATION BETWEEN PROCESSES

We saw in the last chapter that the system call mechanism worked by cooperation between a process and a separate activity which wasn't quite a process. In this chapter we shall see how much the same sort of cooperation between threads can be used to advantage.

We speak of "threads", because in this context it is a more appropriate term than "processes". Almost as a byproduct of multiprogramming it becomes possible to use parallel processing methods to complete a task. Some calculations naturally break down into discrete sections which can be carried out in parallel, and – provided that these sections can be identified – it is simple to implement them as a set of threads executing in parallel. An obvious example is a simple addition of two arrays :

```
for element := 1 to N
do   a[ element ] := b[ element ] + c[ element ];
```

A slightly less obvious example is the simple assignment :

```
a := b + c + d * e
```

where the two expressions `b + c` and `d * e` can be evaluated at the same time.

In a single processor system this might not seem to offer any advantages since only one thread can run at a time. Even in this case, though, a major advantage is that when one thread is waiting for some resource, another thread working on the same job can still run. Further benefits might be expected in a multiprocessor system, where true parallel execution is possible with several threads running simultaneously. The threads can be organised either as lightweight threads of a single parent process (appropriate for a single calculation which can be split into parts) or they can be full heavyweight processes (appropriate when the processes are essentially engaged on separate tasks, as might be the case if one is a utility which can be called upon for service by others).

However appropriate it might be to think of threads, though, much work done in this area has not distinguished between processes and threads until quite recently, and it has usually been assumed that full processes are concerned – whence the title of the chapter, and our concentration from here on with processes. In fact, in many cases it doesn't matter much what sorts of object are concerned, and if they're described by different PCBs much the same discussions apply. The main difference is that threads usually have much easier access to shared memory.

In the examples above, the processes running in parallel were independent of each other – though even in these cases some sort of interaction is necessary to ensure that all the processes wait until all have finished before continuing with the rest of the combined process. More generally, in order to cooperate processes must be able to communicate with each other. Communication between processes is dignified with the title *interprocess communication* (IPC). The information exchanged might be conventional data or it might be a signal indicating that something has happened, which is used for coordinating the affairs of different processes.

METHODS.

There are two different ways for processes to communicate : they can share a resource (such as an area of memory) which each can alter and inspect, or they can communicate by exchanging messages. In either case, the operating system must be involved. As we shall shortly see, these two methods are not mutually exclusive. Within the two groups, they are not even easy to classify; we present them below in approximate order of increasing system involvement, but there's no sense of a sudden break anywhere.

From the processes' point of view, there are two important differences between message passing and resource sharing. One is concerned with the two sorts of information which the communication mechanisms are intended to handle, while the other is based on the permanence of the information.

- In a resource-sharing method, both data for exchange and information about events are transmitted by encoding them as changes to the shared resource, and are received when the receiving process decodes the changes. Typically, the sender changes the value of a variable in a shared memory area, and the receiver inspects the variable if and when it needs to know about the information, and interprets what it finds. The receiver is never worried with information it does not want, but it must take the initiative when it needs to know.

  In a message-passing system, information comes in two forms : the mere receipt of a message is itself information, and might be sufficient if all that is required is notification that an event has occurred; and other information can be conveyed as the contents of the message. In effect, a process which receives a message also receives the information that the sending process has reached the point in its execution where it sends the message, and this is just the sort of information which is needed to coordinate processes' execution. If that's the only information to be transferred, then there is no need for any body to the message. In either case, though, the receiver is forced to act, at least to the extent of receiving the message.

- Information conveyed by resource sharing is static. It is represented by a change in state of the resource, which remains perceptible to all communicating parties until it is explicitly changed by some process. In contrast, the information in a message passing system behaves as a stream; once a message is sent, it is (usually) no longer accessible to the sender, and when received it must be saved by the receiver if it is to be used later.

  With this method, information can be missed by the receiver. If it doesn't happen to inspect the shared resource sufficiently frequently, the sender might change it two or more times in the interval.

That's only half the story, though. The other half is from the system's point of view, which shows much less distinction between the methods. (There are exceptions to some of these comments, which we shall discuss later. The exceptions are usually found in techniques which in some sense blur the distinction between the two extremes of resource sharing and message passing which we are discussing here.) In both cases, there must be some memory reserved for the communication – messages must be saved somewhere if they are sent before the receiver is ready for them. Similarly, the existence of the message must be recorded as some sort of bit or bits in the system. The differences are in how the memory is used (directly, for a shared resource; through system procedures, for message passing) and in how the receiving process is notified of the presence of a message (not, with a shared resource; by the system, for message passing).

SHARED RESOURCES.

What sort of resource can be shared ? It must be something which can coordinate events or pass information. A process wishing to convey information to another must take some action which changes the state of the shared resource, and a process wishing to refer to the information must inspect the shared resource. If we want to pass information, the resource is clearly some sort of memory. It would be possible to provide a special sort of memory for the job, but in practice ordinary memory of one sort or another is used. If we just want to coordinate events, though, special memory becomes more practicable, as we might only need one bit, or perhaps one number.

Here are three variants of the resource sharing idea. All are used in practice.

Shared files : Files can certainly be used to communicate between processes; we saw how they could be shared between processes in the chapter *SHARING FILES.* For use as a communication channel, one process writes a file which another one reads. In practice, though, this method is not completely satisfactory. When a process tries to read from a file which doesn't exist because the writing process hasn't written it yet, the operating system should return an error signal to the reading process. What should the reading process do now ? Does the message mean that the partner process has not yet set up the file, or does it mean that something has gone wrong ? Perhaps the question can be resolved by waiting for a while – but in that case when should it attempt to open the file again ?

Supposing that this difficulty can be overcome (which it can if the file can be set up before either process begins), we must also worry about the speed of communication. We saw in the earlier discussion of shared files that unless special precautions are taken there are likely to be delays before material written by one process can be perceived by another. That might not matter; if we just want to pass on information for use some time later, there might be no urgency involved. On the other hand, if fast communication is important, then the processes must share the file buffer – in which case, it's hard to see any significant difference between the shared-file method and the shared-memory method which we'll discuss next.

The big advantage of shared files is that it is often comparatively easy to implement in a system where other communication mechanisms are not provided, and provided that speed of communication is not important it can be effective.

Shared memory : Sharing primary memory is the fastest way for processes to communicate by sharing a resource. That's easy to imagine, but, as with the files, not so easy to implement reliably.

There are pitfalls. The most significant difficulties are associated with synchronisation : unless the processes sharing the memory use methods to stop two or more of them from writing to the memory simultaneously, the information stored there might be garbled. That's only a problem if different processes try to change the same items of data, and one way to resolve it is to reserve different areas of the data structure for each process to write into. Sometimes that's possible, but unfortunately it isn't satisfactory as a general solution. We will look at solutions to this and other similar problems shortly.

The job of the operating system in this type of communication is limited to allocating the shared memory, providing joint access to it, and providing the mechanisms the processes can use to maintain the integrity of the data stored there. We mentioned the first two of these questions in the earlier discussions on memory; later in this section we concentrate on the last.

Flags : If there are no data, in the ordinary sense, to be transferred, but just an indication that some event has happened, then all we need is one bit of memory somewhere which can be changed and inspected by the processes which wish to communicate. If they are used for communication between two conventional processes, these can just be regarded as an extreme case of shared memory. In some early systems, though, hardware registers were provided for this purpose, often associated with lights on the system console; they were not usually thought of as means of communication between processes, because the idea of a process wasn't well understood.

Nowadays they appear more in communications contexts, as indicators of the state of a device – for example, one can inspect a bit in a device's status information to determine whether it has successfully dealt with its previous instruction. This is an example of communication for purposes of synchronisation between a process driving a device and the process which *is* the device.

MESSAGE PASSING.

As with resource sharing, there are several varieties of message passing. Here we have distinguished three sorts, one of which has several internal variants. They differ in what (if anything) is passed between the processes, and in the machinery for transferring the messages.

Simple message passing : If (as is quite likely) the requirement is to be able to move information reliably from one process to another, shared memory and its tricky implementation can be avoided by sending messages.

Identifying the sender and receiver is important; obviously enough, messages have to be secure. We do not want messages with private information being received by any process which happens to be in a receptive state. Given this requirement, though, there are different ways of satisfying it. Many systems allow communication only to particular processes - the sender must know exactly who the receiver is, and vice versa. Other systems provide communication channels which behave like open mailboxes whereby anyone who has access to the channel can send and receive, but in this case security is imposed by guarding the granting of access to the channel.

Message systems can be implemented in a variety of ways. One important characteristic is the nature of the coordination of the message transmission and reception with the other work of the processes involved. We say that the communication can be *synchronous* or *asynchronous*.

- though we'd rather not. Later (MAKING DEVICES WORK), when we have more ammunition, we'll explain why we'd rather not. Meanwhile, think critically about what the terms mean and what they should mean.

Asynchronous communication is the way we usually use messages in ordinary life; a process sends a message to some other entity, then carries on with its own affairs. We shall see that even in this situation it is possible to use message passing to synchronize processes.

It is the operating system's job to provide the machinery which makes the method work. Clearly enough, it must provide some form of buffer storage to preserve the messages until they are retrieved, and some way for the receiving processes to be informed that messages are waiting, or for them to inspect the queue of messages which have been sent to them. This leaves us with more questions to answer : how big should this message buffer be ? – what happens if the buffer is full ? – should the sending process be suspended until there is more room (– in which case the communication would no longer be strictly asynchronous) ? - does it matter if the receiving process simply ignores the messages ?

With *synchronous communication*, the sender cannot continue until the receiver has accepted the message; the processes are therefore forced to keep in step with each other.

To implement synchronous communication, the system might take the message and deliver it immediately into the receiver's address space, or it might interrupt the receiver process to force it to accept the message. In either case, the system must know the identity of the recipient at the time when the message is sent.

Rendezvous methods : In the rendezvous synchronising method, no message is ever exchanged except when both communicating processes are ready for the transfer, and the information can always be copied directly from sender to receiver. This method is the basis of communication and synchronisation in the programming language Ada. It differs from the simple synchronous message passing method in that it can be bidirectional; as both communicating processes are cooperating in the transfer, information can be copied in either direction.

A second advantage of the rendezvous technique is that it gives more information about what is happening in the system, and is therefore better suited to programme proof methods. If process A receives a message from process B, all that process A can infer is that process B has reached a particular point in its execution, and nothing can be said about what else it might have done. On the other hand, if two processes communicate by rendezvous, then it is certain that *both* processes must have reached corresponding stages in their executions and gone no further, and this additional information can be very valuable.

Semaphores : The methods we have already described are easy to understand in terms of the messages we use in daily life, and they are interesting here because they are being used in a new way. Semaphores are new. They are used purely for coordination; they are an operating system's traffic lights. The only information they carry is that something has, or hasn't, happened, which you can think of as a message zero bits in length. Here we use the first form of information carried by a message – the simple fact of its existence.

SOME EXAMPLES OF MESSAGE PASSING.

Those three types of message illustrate the basic techniques which can be used in systems which rely on message passing for process communication and synchronisation. These and related methods are used in practice in various combinations, and here are two examples of methods used in the Unix system.

Signals : An ordinary message, once sent, has no effect until the recipient notices it. A signal is a message (usually with no contents) sent with a software interrupt to ensure that the recipient *does* notice it. The term comes from Unix, where signals were originally introduced (as we saw earlier in *CLEVER TRICKS WITH PROCESSES*) primarily as a simple and elegant means of dealing with processes which had in one way or another gone wrong. In its original form, the **kill( )** supervisor call could be used by a process to send a message to the operating system, but that's all. It's a very big all, though, because once we have a mechanism to get the attention of the operating system in this way, we can extend it to do other things – such as to pass on messages. The **kill( )** system call was therefore augmented to do other things; it can now convey a signal from one process to another, or from the system to a process. (It is, quite inappropriately, still called **kill( )**.) Unfortunately, this was not particularly well done; everything vaguely like a message was pushed into the same package, and the result is untidy.

A process receiving a signal can deal with it in a variety of different ways : it can ignore the signal, it can deal with the signal (usually executing a special signal-handling routine and then continuing what it was doing), or it can do nothing. There is a difference between ignoring the signal and doing nothing. If the process has not told the operating system that it is going to ignore a particular signal (using another system call, **signal( )**), then when that signal arrives the process will be stopped, which was the original idea of the signals.

Can I, then, stop all your processes by sending them signals ? Clearly enough, that cannot be permitted, and some restriction must be enforced. (Notice how the introduction of a single apparently simple feature causes ripples to spread far and wide through the system.) The Unix answer is to restrict such signalling to processes owned by the same user - which means that I can't send a signal to your process, even if we both agree that it would be a useful thing to do.

Another curiosity is that the interrupt is not strong enough to wake up the receiving process if it's asleep; to continue the metaphor, it merely leaves a very obvious note on the bedside table. The process will therefore attend to the signal when it wakes up in the normal course of events; if it's waiting for something that doesn't happen very often, that could be a long time. For the historical signal, that was perfectly satisfactory, as if the object is to destroy the process it probably doesn't matter a lot when it happens so long as it doesn't take any other action before it dies, but if you want to send it a message to cause some action this is hardly satisfactory.

We can reasonably conclude that this implementation of signals is not a triumph of good design. Of course, we're not the first to notice that, and other methods of communication have been devised. This story does emphasise the principle that if you want some facility in an operating system it's far better to design it with the rest of the system; if you try to add it later, it's hard to make it work well, because it has to fit around all the constraints which are already there.

Pipes : The traditional channel for conveying messages from one process to another in a Unix system is a pipe. A pipe is a shared stream which flows between processes : processes may write to the stream or read from it. In fact, though Unix goes to some lengths to dress it up as a file, it is a stream. (Unix tries to dress up anything outside the programme as a file.) Is this a shared resource or a message-passing mechanism ? Once a process has written a message to a pipe, it can only be retrieved by reading from the "other end" of the pipe, so our earlier definition of a shared resource doesn't fit.

Given that the pipe must work in a single-processor system, it must therefore operate as a buffer. The operating system coordinates the processes in a convenient way and places a limit on how much the pipe can hold. Imagine processes (one or several) writing into one end of the pipe and reading (also one process or several) from the other end. The operating system does not directly synchronise this activity, but it does control the flow of data into and out of the pipe. Any single pipe write is guaranteed not to be interleaved with another as long as the amount of data is less than the size of the pipe.

If a process wants to read from a pipe which is empty, what happens depends on how the pipe is being used. If there is a possibility that the a message might appear some time – that is, if at least one process still has the pipe open for writing – then the process will be put to sleep until the read can be completed. If there is no such process, then the read operation returns with an end-of-file result.

In a similar way if a process wants to write to a pipe which is currently full the process will be put to sleep until there is some space in the pipe. If there is no process which has the pipe open for reading the writing process is sent a signal by the kernel. It doesn't make sense to allow processes to go on waiting for an event which will never occur. (It is debatable whether it makes much more sense to destroy the programme summarily if it hasn't made provision for the signal.)

Pipes are created with the **pipe** system call :

```
int mypipe[ 2 ];
pipe( mypipe );
```

mypipe is an array of two integers. We have met them before; we want to call them *file handles*, but Unix calls them *file descriptors*. They are numbers returned to a process by any successful opening of a device or file. They are used to identify the device or file in subsequent operations. This is an unfortunate name because file descriptors are usually structures containing all necessary information about files. In this text we are determined to keep ourselves pure, so we shall write of file handles, but you should be aware of the Unix usage – and you might even hear it in the lectures.

After the **pipe** call *mypipe[1]* contains the file handle which should be used to write to the pipe, and it behaves in just the same way as any other Unix file handle – for example,

```
write( mypipe[ 1 ], buffer, length );
```

Correspondingly, *mypipe[0]* contains the file handle to be used for reading from the pipe.

There is one serious limitation with ordinary pipes. To use a pipe to send messages from process A to process B, process A must know the writing file handle and process B must know the reading file handle. Suppose that the pipe is opened in process A; now process A knows both file handles, but process B knows neither. How are we to get the reading file handle to process B ? Obviously we need some form of inter-process communication – which is why we are trying to set up the pipe.

It happens, not entirely by chance, that there is one other mechanism in Unix with which we can coordinate the information available to two processes. This is the rather curious way in which new processes are made with the **fork( )** and **exec( )** operations. Recall that open files are shared between parent and child processes; so are open pipes. The pipe can therefore be shared if A opens it, and then starts B by forking. But how does B find out which file handle to use for the pipe ? You have to pass it somehow, because file handles, being ordinary variables, are not shared (which emphasises their artificiality : they are not really parts of the file system, as we pointed out in the chapter *STREAMS IN PROGRAMMES*); but by exercising some care and relying on the rather precise rules followed by Unix in setting up file handles you can work out in B what the value is. Alternatively, you can pass the file handle through a shared file, or you can pass it as a parameter when you make the new process. These methods work, but unfortunately the requirement that pipe descriptors (these are the *real* descriptors – the system structures which are identified by the file handles) can only be communicated by inheritance means that only *related* processes can communicate with each other using pipes.

But, but, but ... surely there's another paradox in the suggestion that we can pass the handle through a shared file ? If we have to go to such lengths to share pipes, don't we have to do the same to share files ? Well, no, we don't – because files are persistent, they are known to the operating system independently of the programmes, and they have real external names. Provided that we know the name of a file, we can open it in quite independent processes and thereby establish communication by sharing a resource.

If you have been following that argument closely, you might find yourself wondering why we don't take the obvious next step : why not give pipes names too ? Then they would behave in the same way as files (which seems to be in line with the Unix way of doing things) and we'd have no difficulty in setting up communication between any pair of processes we liked. The only reason for our hesitation is that the Unix gurus themselves hesitated for a long time. Eventually, though, named pipes made their appearance. A named pipe operates exactly like a pipe, it is just created differently. Using named pipes any two processes on the same machine can communicate with each other. There are still problems across networks. A named pipe (created with **mknod**) is sometimes known as a FIFO file.

Some good news : In modern Unix systems there are more general ways of communicating between processes. System V uses "messages" which allow one server process to communicate easily with several client processes, with each client only receiving messages intended for it. A fairly recent development found in many systems is the provision of "sockets" which allow interprocess communication both between processes on the same machine and between processes on different machines using a network.

These methods are much more complicated than pipes and arise because of the added complexity of communicating in multiprocessor environments.

We will return to these problems in the wider context of distributed operating systems.

COMPARE :

Silberschatz and Galvin[INT4] : Section 4.6.

QUESTIONS.

We were rude about the implementation of signals between processes in Unix. How could you do it better ? How would you design a similar system from scratch ?

Why can't two independent processes share a pipe by passing its file handles through a shared named file ?

How can the Unix system establish communication between any two arbitrary processes using a pipe ? Work out the sequence of events in some detail.