KEEPING TRACK OF PROCESSES

Now that we have some idea of how we can expect processes to behave, we can return to the question of how to manage them. We have already introduced the notion of a process table as a means of keeping track of the potentially many processes running in a computer systems, now we must decide just what sort of management we want, and what information we must keep in the process table in order to put it into effect.

We want management in order to make sure that the several processes in the system can proceed with their own business without interference from others. The system must be able to maintain the continuity and independent identity of each process in order to support the abstraction we need in order to write programmes easily. It must do so even through context switches, which must be quite invisible to the processes.

What do we mean by continuity ? We mean that there should be no changes which are not consequences of the normal execution of the process's code. Code must be executed in the normal sequence, and resources used by the process must not mysteriously appear, disappear, or change – generally, what the process "knows" about the system must behave in an orderly way. We can describe the process's "knowledge" rather more formally by identifying what we shall call, for want of a better word, the process's *disposition*. (In fact, "state" is a better word, but we've already used that to mean something else. Be warned that some writers don't distinguish, using "state" in both senses.) Here's a definition :

> The *disposition* of a process is the collection of information available to the process which describes what it is doing, and what resources it has, at the moment.

Some of this is held in processor registers, some is held in structures in memory; the process state, which we defined earlier, is a part of the disposition. This is what we have to manage, and the system must be so designed that the state information is preserved whenever the process is not being executed, in such a way that it can be restored for the next period of execution. This is the purpose of the process table - to provide a clearly identified point from which the operating system can find any process property which it requires to know. We can identify these properties by following the execution of a process as it performs its designed task.

PROCESS STRUCTURE.

First, then, we've already seen that in virtual memory systems the code and data of a process may be moved from primary memory to disc and back again, and will not necessarily return to the addresses they occupied before. We have to be sure that we can do all this while preserving the important characteristics of the processes, and that each process can be uniquely identified by the operating system and the operating system knows which process is currently using which resources.

When the code of a process is being executed, a processor somewhere has an address in the process's code in its programme counter (the instruction address register, which is a much better name but apparently out of favour). Typically, there will be other processor registers which contain information of various sorts – processor status registers, memory addresses, and such other indicators as are needed to determine the processing mode. The precise details are an accident of implementation, but in general there will be some collection of information which defines the processor state. If we can stop the processor, then at some later time reset it to exactly the same state (including processor registers and memory contents) then we can exactly restart the process.

As well as using memory, the process might be using disc files and files or streams associated with other input and output devices. For example, an open file residing on a disc device has an associated data structure with the information the operating system

needs in order to find the file, read the next record, and so on. This linking information must be stored somewhere accessible to the process. It is clearly part of the operating system's job to make sure that the associations between resources and processes are managed so that only the currently authorised process has access to each resource.

The process structure is therefore a collection of several components. It must include the resources being used by the process – code, data, devices, and anything else necessary for the execution to continue – and it must include the processor registers which define what is happening at the moment, and how it should continue to happen. This is the essential "knowledge" we mentioned earlier, and this is what the operating system must preserve. All this must be kept in the process control block.

DO WE NEED PROCESS TABLES ?

PCBs and process tables are not the only way for the operating system to keep track of what's happening, though we find our earlier arguments strongly persuasive. Another approach – in some ways, the obvious approach – is to store the information for all processes in the part of the system which makes most use of it. For example, the memory manager could hold information as to which process had access to which pages; device drivers and open file structures could be similarly dealt with by different, separate, parts of the operating system.

This design makes it possible for the operating system to answer a question like "which process is using this file?", and that's very useful in some circumstances. If we look at the system from the point of view of the processes, though, we might want to turn the questions round, giving questions like "what files does this process have open?". We certainly want to maintain this point of view when we think of the disposition of a process, because the relationships between a process and objects which it controls or uses outside the processor and memory is just as much a part of the process's disposition as its internal structure. We will therefore want to preserve the essential parts of this information with the rest of the process's disposition. Of course there is nothing to stop an operating system using both approaches except increasing complexity and the need to maintain consistency.

PROCESS CONTROL BLOCK CONTENTS.

There is no standard for the structure of a PCB; every operating system designer produces a new version with every new operating system. The only principle is that everything which the operating system might need to know about a process should be accessible, directly or indirectly, from the data structure which represents its PCB, but there are many ways of arranging the data concerned which will satisfy this criterion. This has led some computists to define a process as "that which is described by a PCB". Whatever the details, though, there are certain items which are commonly components of a PCB, because they are usually useful. Here are some typical PCB fields :

- **Memory location :** The operating system must know where a process's code and data are in memory, so that it can interpret addresses generated by the code and protect the areas from damage. When a process finishes, the operating system must reclaim any memory which it had been using. A typical PCB therefore contains information on the memory a process uses, either the location and extent of significant parts of the process in memory or the location of its addressing table.

- **Open streams and files :** We saw that the state of streams, and the corresponding files, if any, was a component of the process's disposition, so the PCB must provide a route to all such entities with which the process is working.

- **Devices, and other resources :** Reusable resources to which the process has access, especially exclusive access, should also be identifiable from the PCB. When the process finishes for any reason, all resources must be returned to the system, so that they can be made available to other processes.

- **Processor registers :** As we saw earlier, in order to stop a process and restart it again, the values in the processor's registers, including the programme counter, must be saved. It is mildly interesting that the processor registers are one part of the PCB which the process itself never uses; the values are only significant when the process is not running. In practice, some processors provide instructions which store the processor state in places which we would not necessarily guess to be part of the PCB (typically, on the top of the process's running stack), but we shall avoid complicating special cases by simply regarding the location, wherever it is, as part of the PCB. And, just to complete the link, we observe that the stack, if there is one, is part of the process's memory, so we can certainly find it from the PCB. An interesting example, which emphasises that these decisions really are purely matters of design, is the comparison between Ultrix (a DEC version of Unix designed for

the Vax architecture) and Minix[EXE3] (a Unix-like system for Intel processors); in Ultrix, the register contents are kept on the stack, while in Minix they are stored in specific locations in a conventional PCB.

- **Process's identity :** There are many reasons why we want processes to have names. Not the least is so that we can find a process's PCB in the process table ! For example, when the operating system has to send a message to a particular process or perform some operation on a process that is not currently running it must be able to identify the correct process. The PCB therefore includes a process identification field, usually an integer. Some operating systems use the location of the corresponding PCB, either as its process table index or a pointer, as the means of identifying the process. This might be adequate for single processor systems provided that the PCBs never move about, but, as we shall see later, has problems of uniqueness in multiprocessor systems.

- **Process state** : Because the operating system must be able to determine a process's state, the PCB includes a state field. We shall see examples of this below.

- **Process Priority :** Any operating system which handles more than one process must use some algorithm to decide which process will be the next one to run, and many systems use some sort of process priority to make the decision. Information on priority or time spent running will also be stored in the PCB as this information must be readily available whenever the process is not running.

Additional items are commonly found in shared systems :

- **Owner's identity** : In any shared system, each PCB commonly has a field identifying its owner. This can be used for granting privileges to the process and some forms of communication. It is also obviously necessary in any system in which people are to be charged for the resources used by their work.

- **Which terminal :** An important special case of open streams is the identity of the associated terminal. When the operating system has to send a message to the owner of a particular process, for example, in case of a processing error, it is necessary to know to what terminal the message should be sent.

Other fields are included as the system designers think appropriate. For example, if the system provides facilities with which processes can start up new processes, this "parent-child" relationship is often important. In this case links of some sort connecting such related processes will be maintained in the PCB. Unix is an example : each PCB includes a field which identifies the current parent of its process, which in Unix means the process which started the PCB's process.

Apart from links to related processes there are sometimes links to other processes as well. For example a common method of keeping track of processes waiting for resources is to put them into a linked data structure. Though it is common to consider them as part of the PCB, these links have nothing to do with the process, and are better seen as belonging to the scheduler or other system component which uses them.

This list is not exhaustive, but it serves to make the point that the PCB is an important structure. As with other structures in the operating system, it is used to impose order on what could otherwise be a forbiddingly complicated structure.

TWO PARTS TO A PROCESS CONTROL BLOCK.

Not all the information stored in a PCB is of the same importance, and different parts are important at different times. Some information, such as that required to determine which process should run next, must be immediately available to the operating system all the time. Other information is only needed occasionally, and a slight delay in retrieving it is not critical, as for example in identifying the terminal associated with the process.

The relative importance of parts of the process's disposition can vary, particularly between times when the process is currently running and when it is not. Some of the

system data kept about the process, such as the maximum permissible memory size, is only needed while it is running; other information, such as how long a process has been waiting to run, is only needed when the process is not running. This makes no difference in principle, but was sometimes exploited to reduce processes' demand for memory. For example, the PCB in the Unix operating system is split into two sections, one kept permanently in memory, and the other (called the *u area*) perhaps swapped out when the process is not in memory. This was a useful contrivance when computer memories were small and expensive, but it is much less significant now.

REFERENCES.

EXE3 : A.S. Tanenbaum : *Operating systems design and implementation* (Prentice-Hall, 1987).

QUESTIONS

Why must PCBs be stored in memory which cannot be changed by an ordinary process ?

Why is it useful for the operating system to know the parent process of each process ?

Is it just as useful for the operating system to know the child processes ?

Is it necessary for PCB information to be kept private ? Should a process be able to read other processes' PCBs ? Should a process be able to read its own PCB ?
