STARTING AND STOPPING

We wonder whether you have noticed that essentially all that we have said since the beginning of the course has assumed that we have a computer system running. How does it arrive at this desirable state ? This topic is most appropriate for our final section, but we have not left it until now for purely æsthetic reasons; this is its proper place. As we couldn't discuss scheduling until we knew what had to be scheduled, so we can't discuss starting and stopping until we know what has to be started and stopped – which we shall see is, to a great extent, the scheduler.

WHAT ARE WE TRYING TO START OR STOP ?

That's not as silly a question as it sounds. A large operating system nowadays covers an extraordinarily wide range of activities, all proceeding (more or less) at once, and it isn't immediately obvious just what must be done to get it going or to close it down.

The state of the system is always defined by the contents of the various tables with which it keeps track of what's going on. Here's a selection :

- The **process table** has some sort of entry for every piece of code executed, except for the system core (nucleus, kernel, etc.) which could just as well be hardware, but is implemented as code for convenience.

- The **file table** belonging to each process identifies the files which that process is using.

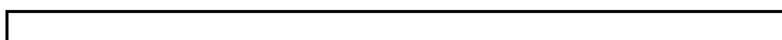
- The **device table** lists the devices known to the system, and gives access to their handlers, interrupt routines, and so on.

- The **user table** lists useful information about everyone currently using the system.

- The system **memory map** and the processes' **memory tables** identify what memory is available and how the processes use it, as well as virtual memory locations as required.

– and so on. These tables are the core of the system's basic knowledge of what's happening, and they are the system's path to anything else it needs to know. Each of them contains information about something that's going on in the system. Typically, each entry in every table contains information which must be consistent with some other information somewhere else, such as accounting information in the user table which must be consistent with information in the user data files on the disc, or addresses of interrupt routines in the device tables which must point to the memory locations where the routines are. In a distributed system, the need for consistency extends throughout the whole networked system; starting and stopping in this context will normally include exchanges of communications to ensure that all parts of the network which need to know will maintain a consistent view of what's going on. To make the system work, this whole intricate pattern must be set up so that all these requirements for consistency are satisfied. To start the system, we have to set up the tables and make sure that they are internally consistent; to stop the system, we must make sure that any important information held in the tables is saved somewhere before the tables are destroyed.

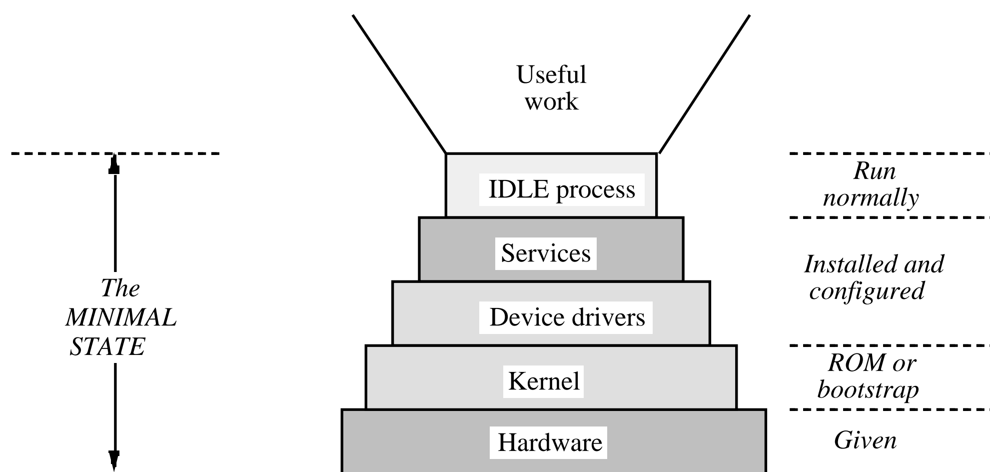
Help is at hand, though. For every table, there is a procedure of some sort which administers it. (More precisely, in principle there can be such a procedure, and we shall suppose for the moment that there is.) This isn't a fortunate coincidence : it's a natural development, which is probably inevitable if you want to build an orderly and comprehensible system. It is these procedures which look after the necessary changes to the tables as the system runs; and they are important to us now because of a commonsense, though not always regarded, principle :



If there exists a facility for performing some task,
ALWAYS USE IT.

We mentioned this principle before when discussing how we should start a new process (*LIFE AND DEATH AMONG THE PROCESSES*); it isn't coincidental that the context there was also one of starting up something from scratch. The importance of the principle is evident if you think about the alternative. Suppose at some point in the system you use a different facility to perform such a task. Then how do you know that the two methods do the same thing ? And even if they do the same thing now, can you guarantee that they always will as each of them is patched and altered to conform to developing system requirements ? Consider the complexity of the network of tables, values, links, and so on which must all be exactly right if our system is to work properly. Clearly, if we are to have any reasonable hope of consistency, we must use the same procedures to build the system as we use to keep it running.

That gives us the key to starting up the system : so far as possible, begin with a set of empty tables, and use the standard methods to bring the system up to full working strength. So what's full working strength ? Roughly, when the system is not doing any useful work, but is ready to deal with any normal event. The user table will be empty, but all the devices will be in the device table, the process table will contain process control blocks describing each of the device processes, the memory map will be properly set up, and so on. A lot of service processes are also usually necessary – the procedures which administer the tables, accounting procedures, system logging procedures, and anything else that happens to be handy. The device processes and the service processes will be waiting for something to happen, with just one or two – perhaps a clock process, and maybe a deadlock checker, for example – occasionally responding to clock interrupts. We'll call this the MINIMAL STATE. Even this is hardly a simple state, but at least it's a clear target for our construction attempts. It looks something like this :



There is one more thing to say about this minimal state : what's going on ? Not nothing, even if we don't count the clock-dependent activities; the processor has to keep running so that it can cope with a job when one turns up, so we invent the IDLE (or NULL) PROCESS. This is a real process, but has no specific job to do except to occupy the processor; it can do anything at all provided that it has zero priority, is absolutely guaranteed never to fail, and will let other processes take over the processor(s) if required. If the system has a preemptive scheduler, this NULL PROCESS can be as simple as :

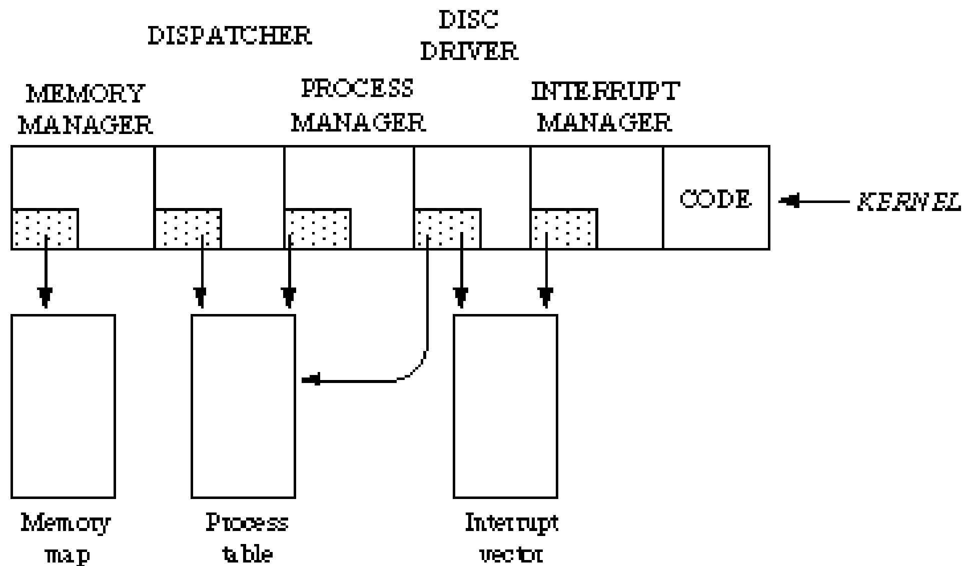
```
while true do ;
```

An alternative that will also work with a non-preemptive scheduler is :

```
queue me for execution at minimum priority; stop.
```

GETTING TO THE MINIMAL STATE.

How do we get to the minimal state from our initial set of empty tables, using the system procedures as far as possible ? We must evidently have some orderly procedure which we can use to start up all the processes waiting, and the null process running, in the minimal state. To use the system procedures to the full, we want to do this in the most usual way we can manage – so we execute a STARTUP PROCESS which contains instructions to start up all the other processes, after which it either dies or turns into the null process. Here's a picture of the startup process :



That's a very schematic picture, so don't take it too literally, but it illustrates what has to be in a startup process and what has to happen. The sequence of events is indicated in the diagram, but in more detail it goes something like this :

```
( The real bootstrap ) :
     copy this block into memory;
     branch to CODE.

CODE :
     set up memory manager tables ( mainly empty, except for the
kernel and the memory tables themselves );
     point the memory manager at the tables;
     get memory from the memory manager for the process table;
     set up the process table;
     point the process manager at it;
     point the dispatcher at it;
     use the process manager to construct a PCB for ( what's left
of ) CODE;
     use the memory manager to reserve space for the interrupt
vector;
     set up the interrupt vector;
     point the interrupt manager at the interrupt vector;
     use the process manager to queue the disc driver;
     start the dispatcher ( which runs the disc driver, which uses
the interrupt manager to set up its interrupt vector, etc., as
described below );
     use the disc driver to get the disc contents;
     find the configuration file on the disc;
     interpret the configuration file;
     wait ( as the idle process ).
```

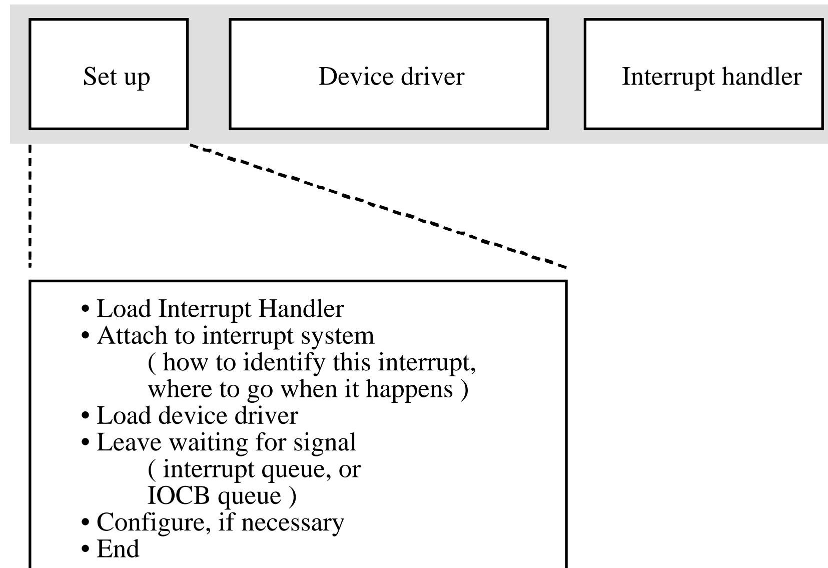
NOTES :

The interrupt manager's job is essentially to maintain the interrupt vector and deal with unrecognised interrupts; all details of interrupt handling are dealt with by the various interrupt handlers for the devices used.

Starting the other processes may itself conveniently be carried out in two stages. For each device or service process which must always be waiting, the startup process begins by executing, in the ordinary way, a specific programme for that process. This programme has two components, one for installing the process, and the process itself; we saw a simple example in the chapter *DEVICE CONTROL SOFTWARE*. (Notice that this approach lends itself well to the use of a small kernel with all other processes self-contained. The same operations were necessary in the older monolithic operating systems, but weren't always collected together like this.) First, the installing process which sets the appropriate initial values is executed. This process will look after details like getting the interrupt entry address for a device handler into the right place and making sure that the device is there, opening the required user information files for the login programme, allocating disc buffer areas and constructing appropriate tables for the spooling system, and so on. This first process can then start the real device handler, login programme, spooler, or other similar process, and die. The real device handler, etc., will itself, like any other process, begin to run, and continue – usually not for very long – until it reaches a point where it waits for an interrupt or signal or whatever, whereupon the ordinary scheduler will queue it in some appropriate way. The startup process then resumes (being the only resumable process) and goes on to the next task. Here's a possible form for the startup process :

```
for each device attached to the system
do   run the programme to start the device's process;
for each service
do   run the programme to start the service's process;
while true
do   ;
```

We can imagine the software package for the device like this :



But whence cometh the startup programme and the empty tables and the lists of devices and services to be started ? We would like this to be as normal a programme as possible, too, but we're running out of information sources – specifically, we have to know where on the disc to find the programme if we're to use the ordinary programme loader. And whence, for that matter, comes the ordinary programme loader ? Clearly, there comes a point at which we are forced to do something special. Well, we've always known that, but at least by this progressive construction technique we can push back the special material a long way.

The special material is called the *cold start programme* (or bootstrap, or initial programme load, or ...). This might either contain the startup programme, or – better, if we stick faithfully to our principles – we can prescribe that the startup programme must always be loaded at some specific disc address, and then build that address into the cold start programme. The cold start programme itself can either be stored at a specific disc address (commonly something full of zeros), and be loaded by hardware on starting the system, or nowadays be built (semi)permanently into the computer as ROM.

Notice that there's nothing cut-and-dried about this sequence, and there's a lot of room for flexibility if you want it. The main design principle is to do as little special coding as possible, taking great care to ensure that, wherever it's feasible, system activities are managed right from the start by the components which will be managing them when the system is in full flight. Just what the startup process can do depends on how much of the system is there when the process starts; for example, if the file system is already there you can use an ordinary filename to identify the startup process's code file. That's a design decision. If the system is essentially all there when the process starts, there might be comparatively little to do – which is why newer computers usually have quite a lot of the system permanently present in memory as ROM.

STARTING FROM SCRATCH : THE "COLD START".

Working backwards, then, and taking a typical case, we can work out how to start the system from scratch. This operation is variously called a *cold start* or *IPL* (Initial Programme Load). It goes something like this :

1. Make the kernel (the basic operating system code) available for execution : the dispatcher, memory management, disc management, etc. This is the part of the operating system which is more conveniently regarded as an extension of the hardware rather than as just another process.

2. Dummy up a request to the kernel to run the STARTUP programme; start the system.

3. The startup programme requests the kernel to run a setting-up programme for each peripheral, each system service, and, finally, the idle process.

4. As each setting-up programme runs it does necessary housekeeping to make sure its peripheral is there and running (if it can), then leaves an appropriate device-handling programme waiting for interrupts.

– and now it's ready to go.

The first step in that list is the tricky one, as it must be accomplished somehow by hardware. We have already mentioned ROM; this gives us an easy way to make the kernel available without any complications at all. Generally, the more system you can afford to keep permanently resident in ROM, the easier the startup procedure – both because there's less to do, and because the resident parts of the system are already available to do it with. For example, the Macintosh ROM contains all the basic file system, which is essentially ready to go after only minor preliminaries.

There are some constraints on how much you can put into ROM, generally based on the rather obvious notion that anything you might want to change while the system is running must be changeable, so can't be hardware or ROM. An obvious example is the system tables, which have to be changed as the system state changes. Even with a large ROM, therefore, there remains something for the startup procedure to do : all the initial states of the tables must be properly established.

Before ROM was practicable, there was usually a short hard-wired programme which could read a sector (usually the first) from the disc and execute it, so the cold start programme proper would be kept on the disc. When even that amount of additional hardware was an embarrassment, there were the real bootstrap procedures. We began our notes with one such (see the chapter *IN THE BEGINNING ...*); we end with another (see the final chapter, *A GENUINE BOOTSTRAP*).

Older systems did have some advantages : some didn't need any restart procedure at all when switched on, because their magnetic core memories retained their magnetised states when turned off. Modern semiconductor memories only retain their states if regularly refreshed, and die when the power source is removed.

CONFIGURATION.

How does the startup programme know what to start ? How do the services know what sort of services to provide ? Most systems provide for some sort of *configuration file*. We discussed the configuration file earlier (in the chapter *CONFIGURING THE COMPUTER SYSTEM*). A part of the startup process is to set up the required configuration, and, as more of the standard startup operation disappears into ROM, so the startup process approaches an interpreter for the configuration file.

TO STOP THE SYSTEM (presumably a COLD STOP ?) :

It is not considered good practice to pull the plug on a running computer; nobody is happy when a power failure does it for you. The reason for this unhappiness (apart from interruption of services) is that the interruption destroys all the information represented by the pattern of tables and links and processes and files and buffers and so on which we mentioned earlier. The most obvious consequence is often that files are not properly closed, and changes or additions which should have been made have vanished. In a microcomputer system, that might be the only noticeable effect, but that's partly because microcomputer systems are built assuming that they'll be switched off at silly times, or that people will change discs without closing files, and otherwise to cater for people's profound incompetence. Larger systems, particularly older ones, are more complex and might be less robust, and after sudden interruptions accounting information can be lost, and files or whole file systems compromised. Many processors have power-fail interrupts, and provision to save such important information as is possible in the fraction of a second of life which remains to them before their capacitors discharge and everything dies, but it still isn't a good idea. (A better response is to use the power-fail interrupt to switch to the backup power supply – but that's more expensive.)

A power failure can have interesting consequences for accounting programmes. One way to measure resource use is to maintain a log of events (which many systems do anyway), to analyse this log later to identify programme starts and ends, and to charge for time used. In case of sudden interruptions, running programmes will not end, so will have apparently infinite sessions. It has been known for billing software to be confused by such logs, and to issue silly bills. More commonly, the interruptions are easy to identify, provided that the restart is noted in the log, which it certainly should be – but then what do you do about charging ? The obvious answer is not to charge for the interrupted session, but that solution isn't necessarily satisfactory if your customers are running long-lasting interactive systems which are normally expected to run for ever. None of these problems is insoluble, but they need thought.

How *should* we shut down the system ? We found we could develop a fairly rational way to start up the system; we can use a similar approach to decide how to stop it. Once again, the minimal state plays a role, as this is the state in which we can switch off the system without hurting anyone. There are therefore two stages to stopping the system.

- **Reaching the minimal state :** The aim is to empty the user table; the technique is to let people log out (in a batch system, complete currently accepted work), but not to accept any attempts to log in (don't accept any new jobs). This dismantles those parts of the complicated network of interactions which concern our clients' tasks and files, and does so in the normal way using the standard software for the job. They should now be safe.

  In practice, with an interactive system things are rarely as smooth as that. People forget about a scheduled stop, and leave programmes running; in any case, people don't watch their terminals continuously, so if some sort of emergency should arise and you send urgent messages begging them to log out they won't notice. There is very little you can do about this except to carry on with your routine and stop the system. It is probably advisable to make sure that you did give adequate warning, and can prove it. The offenders won't be safe, but nothing can be done about it without special knowledge of the running programmes. (That isn't strictly true; in many systems it might be possible in principle to save enough of the relevant parts of memory to remember the current state of the processes and any files and peripheral devices involved, but we don't know of any system sufficiently benevolent to do so.)

- **After the minimal state :** Just as there were certain things which had to be done to set up the system services and devices, so there might be necessary preliminaries to turning them off. Some devices might benefit from being reset to a resting state before switching off; communications networks might need to be informed that the machine will not be available until further notice. Account balances may be written to their files, and the files closed. Finally, the stop itself might be recorded in a system log file. All these operations must be defined in the corresponding device or service software, just as the starting sequences are, and there must be an operating system instruction which executes these, in some specified order if that's important.

Then you can switch off the machine – or, in some cases, such as the bigger Macintosh systems, even that's built into the shutdown process.

WARM STARTS – AND, PRESUMABLY, WARM STOPS.

Now we have analysed the starting and stopping process, it is clear that we might not always need to perform cold starts – there are less drastic possibilities which restart only part of the system rather than the whole lot. These are often called *warm starts*, and are used for several purposes.

For example, a comparatively painless warm start for a batch system doesn't empty the user table, but only completes running processes. Then the states of all the scheduler queues can be saved, the system stopped and restarted, and the queues returned to their previous states.

A warm start can also be used to repair a system which has got itself into an unsatisfactory state; the aim is to undo the part of the system structure which has become damaged and then to allow it to rebuild in the ordinary way. We mentioned the stop-everything-and-restart-it approach to resolving deadlock in the *DEADLOCK* chapter – and also mentioned some of its less happy features.

COMPARE :

Lane and Mooney[INT3] : Sections 17.2 to 17.4; Silberschatz and Galvin[INT4] : Section 3.8.

QUESTIONS.

How must you construct your system in the interests of safety and reliability if people are likely to remove discs without closing files ?

Think of all the tables used by the operating system (start with the list at the beginning of the chapter if you can't think of any). What part of the system has the responsibility of setting up each table ? What manages the table ? How do you know that any important information in the table is saved when the system is shut down ?