SCHEDULING : WHO CAN DO WHAT, AND WHEN.

Once we have decided on our system configuration and have contrived to get it up and running, we must take further decisions which will, we hope, ensure that it runs efficiently. Just what these decisions are depends on the sort of system you have; the decisions for a timesharing mainframe are quite different from the decisions for a collection of isolated microcomputers. In addition to those decisions, the system itself has to take decisions, especially when dealing with time scales too short for people to manage.

A SUMMARY OF AIMS.

If we're proposing to achieve aims, we had better know what they are. "Providing computer services to people" is a great slogan, but we have to go into more detail if we want a set of criteria with which we can evaluate a computer system. Here is a set of aims, which reflect our stated goals by falling into two groups : a traditional group, concerned with the machinery, and a rather less traditional group, concerned with people.

Maximise the work done :

- Keep going as much as possible.
- Keep going as fast as possible.
- Avoid unnecessary waiting.

Keep the people happy :

- Be fair.
- Keep everything moving, even if slowly.
- Be reliable, even if slow.
- Avoid sudden changes in behaviour.

The topics are by no means independent – indeed, they affect each other in many ways, and much of detailed system management is a matter of balancing them against each other.

The result for the system is a spectrum of decisions covering a range of time scales from years or more to microseconds or less – or, if you prefer it in frequencies, 30 nHz to 1 MHz. All these decisions can be seen as part of the activity of scheduling. Perhaps some indication of the significance of scheduling can be gained from this introduction to a published paper[MAN2] :

> *"In the last thirty years, since the introduction of large-scale computer services, users have spent a disproportionate amount of time waiting either for the return of batch output or for terminal response. For most users, computing is immutably linked with delays. This paper suggests that most of these delays are unnecessary. Present scheduling algorithms that queue most work are inefficient. Proper scheduling and charging mechanisms would lead to almost all work processed during the day being processed immediately. Work that is not processed immediately can be reliably processed to meet a prespecified deadline. Such systems also provide the management statistics necessary for the efficient operation of an existing installation and for correct decisions about investment in new equipment."*

TOOLS FOR THE JOB.

While the global view of scheduling can be summarised as determining who can do what, and when (see the title of the chapter), at the executive level it appears as the task of responding to a sequence of specific questions from entities of various sorts, at all levels, which all amount to "Can I do that, now ?". The possible answers are drawn from quite a

small set : *yes, no, later,* and *wait here*; they may be augmented by additional tests, such as *pay here, show me your credentials,* and (in the case of interrupts) *suit yourself,* but the first four are the scheduling decisions proper.

Different answers are appropriate in different contexts (we shall say more about these later), but in all cases the answer should be that which is best for the present and future performance of the system. Determining this answer is the essence of scheduling, and it can be difficult to decide just what it should be. In all cases, the intention is to strike a balance between accepting work as it is offered and controlling the load at a level which continues to give acceptable service.

What are the "entities" which make the requests ? It will perhaps not come as a surprise that they are all quite directly related to our aim of getting work done for people, with the different levels defined by the way we organise the work. At the highest level we have the *people* themselves (or, for that matter, ourselves – and if our assertion that people take the highest place is regarded as discriminatory, we agree entirely : people *are* more important than machines). People organise their work as *jobs* which have to be done; they used to be embodied as collections of punched cards, but now are more likely to be terminal sessions. The jobs, in turn, are commonly composed of *tasks,* each of which might require different programmes or files. Each task is executed as a *process,* or perhaps several processes. The processes, or services used by the processes, are likely to generate *interrupts,* which are not quite as closely tied to the hierarchy of work units as the others, but are still requests for service.

Because any or all of these entities can, and do, require scheduling services from time to time, and in many cases the essentially same system responses are appropriate, it is convenient to define a term which can mean any of them, according to context. We shall use the word "activity" for this purpose while discussing scheduling; notice, though, that (like most of the other words for activities) it is used less rigorously in other contexts.

We remark that this hierarchy is a direct consequence of the way we have chosen to organise our work; a different sort of organisation would lead to a different structure. The sorts of organisation which we have used so far seem to fit into this structure reasonably well, perhaps with bits missed out (examples later), but that doesn't mean that there is no significantly different organisation which could be handled in a different way. We don't know of any.

The responses we listed are our only means of controlling the flow of work; they are the "tools for the job" of the heading above. In constructing the operating system, the clever trick is knowing which of the possible answers is appropriate as activities request service. The choice of response always depends on the present or expected availability of some sort of resource; recall that resource management has always been considered a primary function of operating systems. Each of the responses is in principle possible at each of the time scales, given appropriate interpretations of the actions in the case of administration, though not all are equally sensible. In the table below, we've called all units of work "requests" for convenience.

Answer		*Description*
Colloquial	*Formal*	
No	Reject	Refuse to accept the request. This is the only possible response if the required resources aren't available, and won't be. The dearth of resource can happen in two ways : either the system does not have the resources required by the request, or the requester can't provide the resources (typically money) required by the system.

Wait here	Suspend	Accept the request, but defer its execution. This response is appropriate if the required resources aren't available now, but will be "soon". Devices might be out of service, operators might be unavailable, unforeseen demands for memory might have temporarily overloaded the system.
Later	Queue	Accept the request, but require it to wait its turn. This is a possible response if the required resources are available, but are currently in use.
Yes	Run	Accept the request, and execute it immediately. Everything it needs is ready and available.

Of these possible actions, the **suspend** and **queue** operations are the tricky ones, as they are "maybe" decisions; both offer service some time, but not just now, and both therefore lead to collections of activities waiting for service. Obviously enough, if we want to provide a functional system, the algorithms we use to provide service must guarantee that each activity delayed must be served in a finite, and preferably short, time, but we must still find a way to select the activity which in some sense most deserves a resource when it becomes available. We shall say more about some of the technical reasons for giving or withholding service as we discuss the various scheduling levels further, but there are always criteria of deservingness which cannot be determined mechanically by the system – urgency of work, special contract arrangements, company politics, and others even less pleasant to contemplate. These are traditionally taken into account by associating with each job a numerical value of *priority*. Priorities might be associated with individuals or groups through information from the user data file, or they might be associated with individual jobs entering the system.

PRIORITIES.

In practice, priorities might be initially set in the ways suggested in the previous paragraph, but once a job is under way they are likely to be changed to reflect the current state of the system and what the system can determine about the job's characteristics. Generally, priorities might be determined by resources required, time of day, state of bank balance, performance while running, or any other factor thought to be important in deciding the urgency of the activity. The rules are established by the installation manager, or they are built into the operating system software, or by both methods. The operating system's contribution to the priority can be set to encourage desirable behaviour, where the definition of desirable behaviour depends on the environment. Some examples :

- In a university interactive system, there are many interactive users who should all be treated equally : encourage short interactive operations, discourage extensive processing.

- In a batch system, priorities are implemented by allocating jobs to appropriate queues. There are several different scheduler queues for jobs with different characteristics : discourage lies about characteristics by penalising jobs in inappropriate queues.

- In a process control system, all processes involved are known beforehand. Priorities can be set rather precisely to reflect the various processes' urgencies.

- A print server in a network might determine the priorities of print jobs received from the size of the file to be printed.

Perhaps it is appropriate to sum up that material by adding another "tool for the job". While this is, at least in part, not directly related to the demands for and availability of resources, it can make a significant contribution to the effectiveness of the scheduling system. This response to circumstances is to determine the level of service which the activity should have by *setting or changing its priority*. The priority can be set initially

from prior knowledge, and can be changed in the light of the job's behaviour when running. This is clearly a useful tool with which we can organise the disciplined and orderly use of the system's resources. That being so, it is unfortunate that, though it's easy to define priorities, it is by no means as easy to implement them so that the priorities really are reflected in the processes' shares of the system. We shall say more about this question in the chapter on *DISPATCHERS*.

SCHEDULING AND PEOPLE.

While our immediate concern is to decide what operating system features are necessary to supervise the orderly running of the system, it is appropriate to point out how the people who use the system are involved. For one thing, of course, they are involved because the computers are there to provide the people with service – that has been our theme throughout the course.

In addition to such general considerations, though, people are particularly significant in scheduling, for at least two reasons. First, the general question of "who can do what, and when" is necessarily qualified by "who wants to do what, and when"; either the supply must conform to the demand, or we must also take on the job of finding ways of shifting the demand to fit the system. Peak usage times are important; and special times – often hours and half-hours – commonly mark surges of specialised demand because people change their activities according to the clock.

The second reason for the importance of people in scheduling has become prominent as the orientation of computer systems has evolved through the stages we investigated in the *HISTORY* section. In the early batch systems, people's only function (apart from operators) was to provide a supply of work to do. With timesharing systems, in contrast, people began to play a direct part in scheduling by determining which programmes they wanted to run, and requiring instant service from the system. Now with window interfaces people can switch from process to process, or thread to thread within a process, simply by moving a mouse and clicking.

We have mentioned some of these interactions in our descriptions of scheduling levels below, but it is always useful to bear in mind that scheduling is now very much a cooperative venture between the operating system and whoever is (or are) using it, in which it is the function of the system to support the decisions made by the people.

THE SCHEDULING LEVELS.

Here are some notes on five levels of scheduling found in operating systems. They are not all found in all operating systems, but they are all found in some. This description is biased towards a typical large shared system, because that shows most of the features in the description, but the pattern applies, with some modification, over a much wider range of systems. We comment on some examples after presenting the tables.

Level :	**Administration**
Question :	Can I use your computer ?
Answer :	Yes; No; Pay here; Credentials ?
Manages :	People.
Timescale :	Days to years.
Decisions :	Who may use the machinery, and how much, what work to support. Allocate resources accordingly.
The system software should :	Register new projects; distribute resources; maintain usage and accounting records; let people query their current balances.
Needs :	Information about people : what sort of work; creditworthiness; authorisation.
	Information about the system : capacity.
	System structures : the userdata file, and secure access thereto.
Description :	Essentially manual, with results which must be conveyed to the operating system.
Nature :	Politics.

If you are running your own personal computer, this level doesn't exist, but it has to be handled somehow in any commercial or organisational system – even if that system is composed of separate microcomputers. Without communication between the components of the system, the automatic functions of the userdata system are not possible, but they can be managed in a distributed system connected by a network.

Level :	**The high-level scheduler**
Question :	Can I use your computer now ?
Answer :	Yes; No; Later; Wait here.
Manages :	Jobs
Timescale :	Minutes
Decisions :	Which jobs to accept into the system, set priorities.
The system software should :	Select jobs from those waiting which are likely to make best use of the resources currently available.
Needs :	Information about policies : decisions from the administrative level.
	Information about the user : privileges, account state.
	Information about the system : current state.
	Information about the job : resource requirements, priority.
	System structures : userdata file, job queues.
Description :	Human decisions, implemented by machinery.
Nature :	Administration.

Jobs are quite curious entities. So far as the organisation of the work goes, they are real enough – the pile of punched cards, or the disc file defining a batch job or command file, or the terminal session, is an identifiable and significant unit of work – but in computing terms jobs are processes which just happen to be devoted to controlling other processes instead of doing more obviously directly useful work. For a terminal session, the job process is executed inside the head of whoever is using the terminal, which (as we shall see) has its unfortunate side but at least imposes no load on the computer system. For the punched cards, batch file, or command file (all essentially equivalent) the position is different; the job process must be executed inside the computer, and it must persist throughout the time during which the component tasks are active. That caused problems for early systems without facilities for multiprogramming.

The problems were addressed by, in effect, ignoring the process-like properties of the job. The (usually) monitor system would simply read one card at a time, or one line from a command file, and deal with it independently of anything else that might be happening. The job control languages which developed in this environment were primitive. The need for some sort of coherence between job steps – for example, to check that a programme had completed normally or a source programme had compiled without syntax errors – was first handled by the growth of curious special-purpose flags in the operating system, set by the event concerned, but liable to be destroyed by the next system operation. (An example is the Unix shell status variable.)

Most oddly, the same sort of machinery was usually still used when multiprogramming became common. It certainly wasn't necessary – we saw (in *JOB-CONTROL LANGUAGES*) that Burroughs WFL was structured as a high-level language, and compiled, and it was indeed run as a separate process just as our description suggests.

Level :	**The low-level scheduler**
Question :	Can I run this specific task now ?
Answer :	Yes; No; Later; Wait here.
Manages :	Tasks.
Timescale :	Seconds.
Decisions :	Select which queued task to run next.
The system software should :	Consider what system resources aren't being used to capacity, run a process that uses those resources (not very practicable in an interactive system).
Needs :	Information about the system : current state.
	Information about the job : associated privileges, priority.
	Information about the task : resource requirements.
	System structures : the process table.
Description :	Fairly automatic, little freedom for decisions : software.
Nature :	Middle management ?

We remarked earlier that the levels were defined by the way we organise our work. One could argue that interactive work is so organised that this level doesn't exist. We prefer the view that the level does exist, but is executed on a different processor – that is, the brain of whoever is controlling the job. After all, looking at events from the point of view of the processor, there's no essential difference between interactive work and a command file running on a different processor and sending its instruction through a network. The only significant factor for the system is whether it is required to execute tasks as soon as possible, or whether it can defer them until better resources are available.

However it's done, the low-level scheduler's function is to start a new process. In terms of the process state diagram (*PROCESSES IN ACTION*), the low-level scheduler manages the transition from *Under construction* to *Runnable*. That doesn't mean that the low-level scheduler *is* the process construction operation, for processes can be constructed in other ways. In a system which implements jobs as separate processes, the high-level scheduler also starts processes, and we saw earlier that processes themselves can (in some systems) start other processes. The distinguishing feature of the low-level scheduler is that it institutes a new job step, which is the entity we have called a task.

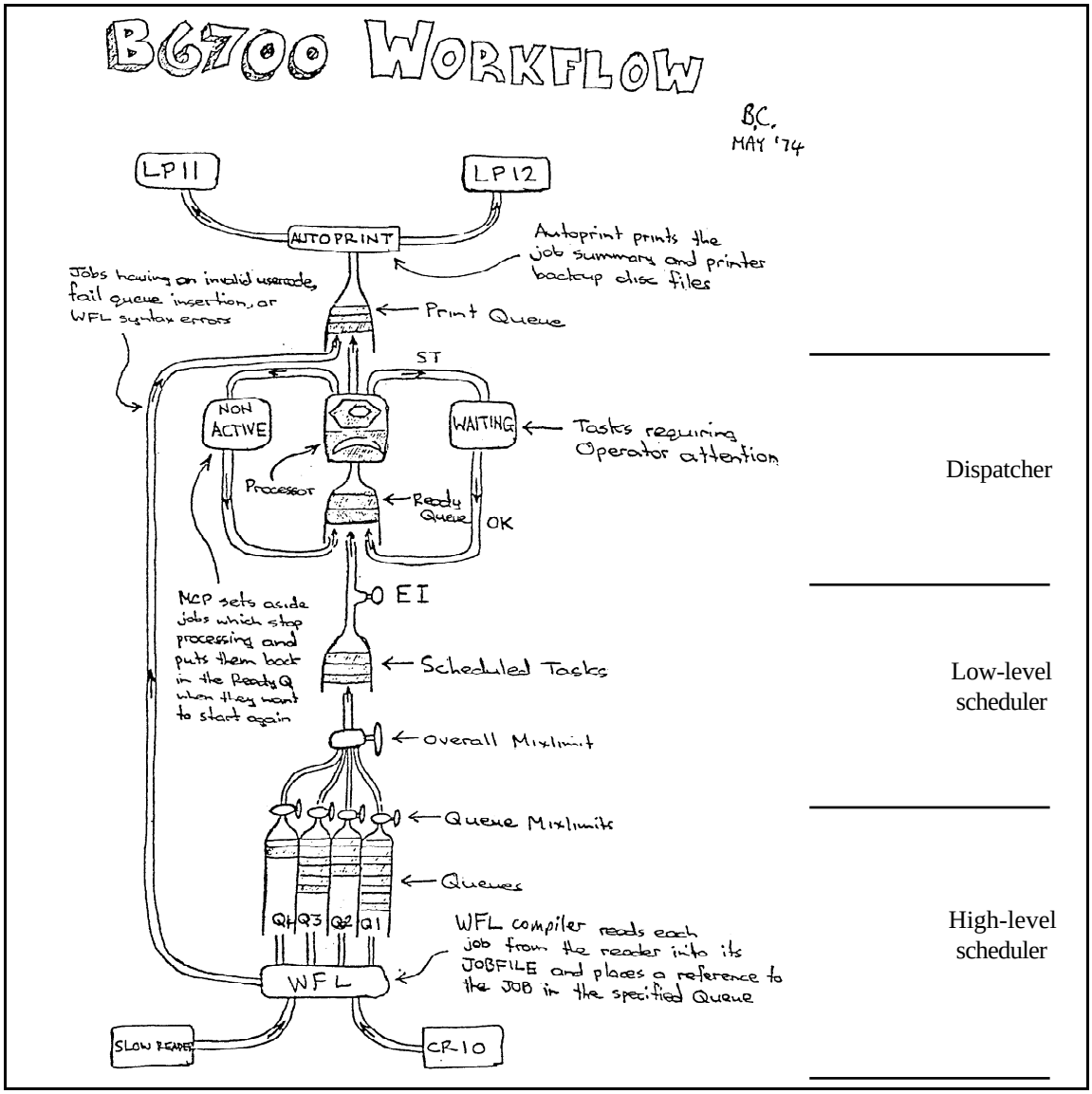
Level :	**Dispatcher**
Question :	Can I use your processor now ?
Answer :	Yes; No; Later; Wait here.
Manages :	Processes.
Timescale :	Milliseconds.
Decisions :	Which *ready* process should be given an available processor, and for how long.
The system software should :	Maintain records of *ready* processes; whenever a processor is available, select a process to run taking into account order of appearance, priority, recent behaviour.
Needs :	Information about the system : what processes are ready.
	Information about the processes : priorities, performance.
	System structures : the *ready* queue.
Description :	Completely mechanical, soft-firm-hardware.
Nature :	Engineering.

The dispatcher manages the *runnable* state of the process state diagram. We shall say a lot more about it in the next chapter; for the moment, notice that, as it might have to run every few milliseconds, it is quite important that it should run very quickly, which means that it can't be very clever.

Level :	**Interrupt manager**
Question :	Can I use your processor now, or else ?
Answer :	Suit yourself; No.
Manages :	Interrupt requests.
Timescale :	Microseconds.
Decisions :	None, really. At this level, operation must be essentially unconditional : the interrupt has to be accepted or rejected, depending on whether or not interrupts are inhibited at the moment. A limited amount of queueing is possible, but long queues of urgent requests don't make much sense. The action must be taken now, whatever happens to be going on in the computer.
The system software should :	Execute an interrupt procedure in response to an interrupt request.
Needs :	Information about the system : interruptible or not, running process. System structures : interrupt vector, *running* state.
Description :	Completely mechanical, hardware.
Nature :	Engineering.

Interrupts also merit a chapter to themselves (called *INTERRUPTS*). Under ordinary circumstances, they must be accepted, for they belong to some process which we have already permitted to run, and we must provide the services required if we are to maintain a functional system. We saw in *PROBLEMS OF CONCURRENT PROCESSING* that there were occasions when we wanted to disable interrupts, but we must be careful to do so only for very short periods.

To show how it all (or at least some of it) fits together, here's the picture we saw in the chapter on *BATCH SYSTEMS* with the middle three levels marked. The administration level happens outside the computer, so isn't represented; and the interrupt level isn't relevant, as the Burroughs MCP is a non-preemptive system. (In fact, all the real interrupts happened on an auxiliary processor which handled all communication with the outside world, leaving only some special interrupts for inter-processor communication.)



PROCESSING MODES AND SCHEDULING.

How much use today is that diagram of an ancient batch processing system ? If you're interested in principles (which we are), it's as good as it ever was; it illustrates some significant features of process management in a straightforward way. If you're interested in practical system management today (which we are), it doesn't cover all the possibilities, but it's a good base case with which we can compare other systems to see where they differ in principle, and therefore where we might need changes in the scheduling machinery.

Generally, what sort of scheduling system you use depends on many factors, and we shall inspect some of these in our discussion. Here we comment briefly on a few systems of different types to illustrate something of the range of possibilities. We shall not attempt to cover current practice exhaustively, because the range is enormous and continually changing. Instead, we shall try to understand something of how the different techniques can be used, so that when yet more new ways of organising operating systems come along we are in a position to choose a suitable set of scheduling techniques. We shall say more on the significance of the different scheduling levels in the various cases later (*SCHEDULING IN ACTION*).

A batch system is designed to maximise the efficient use of the machinery. It will therefore use any of the available tools which help it to improve the performance. To do so effectively, it needs a lot of information about what is going to happen at every level, so it will require that information about jobs be presented in advance.

Typically : On acceptance into the system by the high-level scheduler, jobs are put into queues according to their resource requirements as given in the job attributes defined by their owners. At any moment, a certain number (defined by the system manager) of jobs from each queue may be active. Each active job offers its next task for consideration by the low-level scheduler. The low-level scheduler accepts tasks for execution according to their task attributes (which may be taken as identical with the job attributes, or separately specified) and the current state of the system. On acceptance, a process is associated with the task, the process is placed in the system's ready queue, and it comes under the control of the dispatcher.

An interactive system is designed to give good immediate service to people who use it. This limits the repertoire of tools which it can use. It can reasonably reject an attempt to log in to start a new interactive session (the interactive equivalent of a new job) if whoever is trying to log in isn't known to the system or has run out of money, but apart from that must always try to run every session as requested. A task should therefore only be suspended or queued for very short times – which is to say, by the dispatcher – or after some sort of dialogue with the task's owner to ensure that the interruption in execution is acceptable. It is possible to change a task's priority if its performance is such that other tasks are seriously hindered. A particular consequence of this requirement is that the low-level scheduler disappears : we have no choice but to proceed with every request to start a new task. Alternatively, we can say that low-level scheduling is now a distributed manual function, performed by several independent human processors according to whim and fancy.

Typically : Subject to proper identification and availability of resources, an attempt to log in is accepted, and a new session is started. There is no queue at anything but the lowest levels. Every active session generates requests that tasks be executed, and it is expected that – unless it's impossible – all such requests will be implemented forthwith. The only remaining control is through the priorities of the active processes, and the way in which these are handled at the dispatcher level.

A GUI microcomputer system is a sort of interactive system. It poses rather special scheduling problems because process switching, typically initiated by clicking on different graphical objects displayed on the screen, is very easy; it requires only that the GUI manager knows which window is active on the screen, and directs interrupts from the terminal to that window's process. Such interrupts will normally identify the destination process, the type of event, and the identity of the window. (Notice that the main system does not need to know which process is represented as active on the screen, and that simply making a new window active has no repercussions on the scheduler.) Distinguishing between streams in processes is almost as easy if the operating system can associate streams with threads rather than processes; if it can't, then it is up to the process to sort out which window corresponds to which thread, and to direct its operations accordingly. For a long time, the common maximum number of active processes was one, apart from those initiated by interrupts, but this restriction is becoming less severe with the more ambitious microcomputer systems. More recently, it has become possible to run several processes simultaneously by multiprogramming methods.

Typically : Low-level scheduling is strictly governed by the interface events, so it happens at the interrupt level. In practice, as it is almost invariably the case that one event to a particular process is followed by many more in order to complete the operation required on that process, it is almost equivalent to regard this as a manual implementation of the low-level scheduler. The person using the system can switch at will between different processes or between different activities of the same process, which is precisely the function of a low-level scheduler.

A collection of isolated machines has very little in the way of scheduling (apart from that provided by the machines' own operating systems) unless some form of access control (keys to the door, for example) or booking system is provided; these are administrative and high-level scheduling techniques. Services (printing, for example) are likely to be provided by setting aside a machine for the purpose,

and requiring people who wish to use it to carry their material there on discs. The physical queue of people can then, with a bit of imagination, be regarded as a low-level scheduling device to control the global printing task.

A networked microcomputer system resembles a multiprocessor interactive system in some respects, but providing each terminal with its own more or less powerful processor, memory, and disc store reduces or eliminates the need for communications between processors - which is just as well, because it isn't always simple. The networked system is in other respects closer to the same system without a network, with the main difference being the provision of services through the network rather than by carrying discs to dedicated machines. The high-level scheduler is absent (unless there's a booking system); low-level scheduling, as with the interactive systems, is manual, and likely to be severely constrained by restrictions on the number of concurrent processes permitted. In such cases, though, dispatching has usually been a manual operation, initiated by some form of explicit request to switch from one process to another. "True" multiprogramming, with interrupt-based timesharing, is still comparatively uncommon.

Typically : At the highest level, scheduling is likely to be rudimentary; unless some sort of booking system is in force (which requires significant communications activity between the microcomputers and some central controller to run properly), you can use a microcomputer if you can get one. Any lower levels are handled by the microcomputer software itself.

COMPARE :

Lane and Mooney[INT3] : Chapter 6; Silberschatz and Galvin[INT4] : Part 2.

REFERENCES.

MAN2 : H. Wills : "Fundamentals of pricing and scheduling computer services and investment in computer equipment", *Computer Journal* **3 3**, 266 (1990)

QUESTIONS.

How do the scheduling levels apply to different sorts of system ? Consider isolated microcomputers; distributed microprocessor systems with processors, servers, communications, and so on; anything else you can think of.

How does the Macintosh event queue structure fit in with the description of the GUI system ?
