Computer Science 340

Operating Systems

1994 EXAMINATION : ANSWERS AND NOTES.

Material in this font was added after marking the scripts.

GENERALLY APPLICABLE COMMENT.

One answer began : "How detailed should I be ? I'll stop at the hardware level.". (He gave quite a good answer, though with clear leanings towards a specific system with which he was familiar.) This is always a difficulty : most operating systems questions do have answers on several levels, and it can be difficult to set limits in a question without using words which will present you with most of the answer. On the other hand, it's interesting that perhaps 90% or more of the people answering the questions clearly interpreted them as I'd intended.

I interpret that as a suggestion that people who have been to the lectures or read the notes or the textbook are working in the same context as I am, and can interpret the questions in the way I intend. People who don't take an active part in the course, but rely on their general knowledge to pass the examination, may occasionally misinterpret the questions.

I have no desire at all to force knowledgeable people to toe my party line if they don't want to, but this observation suggests that, if only as strategy for the examination, they should perhaps read through the notes. It will do them no harm, and may give them an acquaintance with a different point of view.

QUESTION 1.

(a) **User authentication :**

Danger : That someone (A) might be able to attach to the system falsely claiming to be someone else (B), and thereby fraudulently gain access to B's property on the system.

Safeguard 1 : Passwords (or other secret knowledge) : Each attempt to attach to the system must be accompanied by presentation of a password (or successfully meeting some other criterion depending on knowledge), known only to the system and B.

Safeguard 2 : Keys (or other physical objects) : Each attempt to attach to the system must be accompanied by presentation of a key (or some other thing which is uniquely identifiable and known to be possessed by the authorised person). The system checks the object, comparing it with data which it has recorded in its information on B.

Safeguard 3 : Signature (or other personal attribute) : Each attempt to attach to the system must be accompanied by the act of signing a name (or performing some other action, or presenting some body part, which is uniquely identifiable as performed by or part of the authorised person). The system checks the object or action, comparing it with data which it has recorded in its information on B.

Protection :

Danger : That an accident may occur spontaneously or because of someone's mistake, and impair the proper operation of the system.

Safeguard 1 : Preserve copies : Backup, archive, multiple file versions, undo : The system provides facilities which may be automatic or discretionary for storing information, locally or remotely, which can be used to restore the system to its state before the accident occurred.

Safeguard 2 : Protect access : Passwords, file protection, system call : The system imposes some simple check on attempts to gain access to property in the system. The check is typically non-specific, based on comparison of easily determined attributes of subject and object. (Passwords are slightly less overt, but depend for their security on the discretion of the subjects, which is not under control of the system.)

Security.

Danger : That someone (A) may by devious means gain unauthorised access to property belonging to someone else (B).

Safeguard 1 : Subject-based security : Capabilities : B gives a capability to each authorised subject, which must be presented at every attempt to gain access to the object. The capability is guaranteed unforgeable, and identifies the object and the allowed mode of access.

Safeguard 2 : Object-based security : Access-control lists : Access to an object is guarded by a list of authorised subjects, specified by B and maintained secure by the system. This list identifies specific subjects or groups of subjects, and the privileges with which they may be permitted access to the object.

Dangers were only rarely described clearly. Quite commonly, a description of an unfortunate consequence was given rather than an identification of the danger. "Users might read the wrong files" could happen because of bad authentication or bad security.

Several people gave two varieties of one sort of safeguard - for example, both system backup and file generations instead of the single method of preserving copies. I usually accepted that, though it wasn't what I'd intended. If the two answers were very similar (use partitioned memory, and use memory protection), I counted it as one method.

A sentence of the form "For each of these areas, explain A, and briefly describe B." expects a description of B for each area. For area := 1 to 3 begin A; B end. It is not the same as "For each of these areas, explain A. Briefly describe B.". For area := 1 to 3 begin A end; B.

(b) An object-based system is appropriate : people need not be registered or be forced to jump through administrative hoops.

People must not have direct access to Q1; only those using (say) Q1GUARD may do so; but Q1GUARD writes the log records which are needed. This requires that Q1GUARD survive throughout the execution of Q1 : so we need an operating system in which several processes may run simultaneously, with at least the ability to wait until their children have died. (Without such a system, the Q1 user would have to run, say, Q1TIDYUP afterwards - but one might perhaps rely on some economic pressure, to do with a charge for the length of the run, to encourage people to do so.) A minor assumption is that the name of the user is available from the operating system in some way; without this information, Q1GUARD can't record the charges in the correct account. Communication between the person and Q1 must also be possible, so either the Q1GUARD must act as an intermediary throughout (again requiring that it run concurrently with Q1), or it must be able to ensure that Q1's input and output streams are linked to the person using it.

The access matrix is something like the diagram below. ("User" means that the subject - a programme - has the same rights as the subject executing it. This is the normal assumption, and we only need special provision in cases where some different protection rules are required.)

		Subject		
		Person	Q1GUARD	Q1
Object	Q1GUARD	Yes	-	User
	Q1	No	Yes	-
	Accounting files	No	Yes	User
	Person's files	Yes	User	User

A capability system is appropriate. People without special privileges have capabilities for their own files and for publicly available utilities such as Q1GUARD, but not for Q1 itself. Q1GUARD is endowed with a capabilty for Q1 and the accounting files by its owner (perhaps, though not necessarily the operating system) when it is set up. A subject calling Q1GUARD must pass capabilities for his own files with the call, which is handled by the usual calling mechanism.

Free memory access cannot affect the file access control itself, because that is based on access control lists which are held on the disc : even if copies of the lists were obtained, they would be of no direct help. (Of course, diligent analysis of the contents of memory might give you a copy of Q1, in which case other security measures would become ineffective.)

I expected some detail of how the access control was managed - just leaving it to the operating system wasn't enough. You were supposed to be telling me how the operating system should do it !

Not many people worried about getting access to the local files for Q1.

Several people suggested that access to Q1 should be by a system call, implying (sometimes explicitly specifying) a special one. It's obviously possible, but not a good solution; you wouldn't sell many Q1s if the customers also had to buy a new operating system.

Many people identified "the subject" as the person and "the object" as Q1, without considering the different rôles of various entities in the transactions which comprise the overall job.

A very ingenious answer included the suggestion that Q1 could be driven as a device. You would have to open and close it (whereupon the "device driver" would look after the timing and the log), and the communications would be via read and write.

The last part of this half was a mistake. We'd changed it from an older version which mentioned memory dumps on the grounds that people might not be familiar with the idea. It should have specified read access. I marked it as it appeared on the examination paper, but required that a reasonably detailed explanation of the dangers was given.

QUESTION 2

(a) Both data and instructions must be transferred between the programme and the file system, so the required data structures are buffers for the data and a file information block for the instructions.

Buffers are areas of memory with no structure except that implied by the file data they hold. When the file is used, they accommodate blocks of data in transit between the programme and the file itself, matching the possibly different requirements of file system and programme for quantity of data and speed of transmission. When the file is opened, the buffers must be allocated - though that isn't their *function*, and is therefore not covered by the question.

File information blocks are data structures containing divers information about the file. They are used when files are opened (and closed, and for other administrative tasks), and they transmit information about the file - such as its name and how it is to be used - to the system from the programme. They have little active significance when the file is used, but act as sources of data such as device type and buffer addresses.

This was too easy - I should have asked how the structures were used too. (Even so, people contrived to lose marks !)

Some suggested that one of the data structures was a stream. I accepted that. (I should have thought of it, but my mental picture of the interface was too thin.)

People who offered the input-output request block got some credit under the "administrative transactions" heading.

Answers in which one of the components (such as the file information block) was split into two (perhaps curiously named) were marked as for the single component.

The file descriptor (with terminology used in the course) is not part of the answer. It records the properties of the static file on the storage medium, not attributes of communication between file and process.

(b) • Find the file attributes (which may be "there is no such file") from the file system, check that the requested access is permitted.
• Fill in details of the file information block.
• Allocate buffer space.
• Report the operation complete.

(If it's a new file, the system may or may not construct a file directory entry. The file information block is sufficient to work with, and it is quite common for files to remain anonymous, and not be entered into the directory until they are closed.)

When the attributes specified, explicitly or by default, don't match the actual attributes of the file :
• Security violation;
• Open a non-existent file for reading;
• Attempt random access on a stream file;
etc.

Also bookwork, widely, though not always accurately, learnt by heart.

Several people gave me the answer from a previous incarnation of the question. That's interesting, because it's wrong. (I accepted it anyway.) It isn't very wrong, but the second point said "Construct the file information block" - whereas the FIB should be constructed, typically by a compiler, before the programme is executed. Observe that if you stick with the old answer, the order is important : if you make the file information block first, you have somewhere to put the buffer pointer.

Several answers included references to input-output request blocks. That's inappropriate at the process level (though fair enough for the system), because you can't set up the request queue system until the file information block is complete.

Several people contrived to answer this part with no reference at all to the buffer and file information block they'd described adequately in part (a). Strange.

Many people believe that part of the **open** operation is to copy the file into memory.

(c) INITIAL STATE :

The file state may be open or closed;
it may be opened by any process for reading;
it may not be opened for writing;
the number of processes currently using the file is in a usage counter.

STEPS IN THE OPERATION :

1 : Make a new empty file in the same directory as the data file, with access restricted to the operating system and writing permitted;
2 : Open the data file for reading;
3 : Copy the data file to the new file;

4 : Close the data file;
5 : Change the new file according to the instructions;
6 : Close the new file;
7 : Change the new file's attributes to prohibit writing and to permit simultaneous reading by any number of processes;
8 : Redirect the directory pointers to associate the new file's information with the original file name (and amend attributes as required);
9 : Monitor the usage counter of the old file, and delete the file when the counter falls to zero.

(Step 8 assumes that the changing process has direct access to the directory. Even with "system privileges" (assumption 5) that's by no means guaranteed, but the direct action shows that very quick changeover is possible in principle. Even if such direct action isn't feasible, the principle of fully preparing the new file before changeover ensures that the switch can be made as quickly as possible.)

Assumption 1 guarantees that the changes can be made without suspending processes which are reading the file when the change operation is initiated, and that requests which arrive after the change has begun but before it is completed can be satisfied by the old file.
Assumption 2 is necessary to ensure that the shift of file pointer only affects processes making new requests to open the file, which allows new processes to begin using the amended file before old processes have finished using the old version.
Assumption 3 implies that we cannot use the fact that the changing process is executing as a guarantee that it has uninterrupted access to the system or to the original file.
Assumption 4 dictates that we cannot wait until processes have finished using the old version of the file before making the new version available.
Assumption 5 is a necessary basis for our implementation's messing about with the file directories.

Not so easy.

Few people gave the detail requested. Very few listed "significant initial properties"; rather more pointed out the significance of the assumptions. Several tried to guess what the assumptions were for before solving the problem, and gave odd answers. The best way was to solve the problem, then look back to see where the assumptions had been used.

Several people worried about atomically renaming files. Very few noticed that all that was required was the replacement of a pointer in the directory structure.

Many answers required (not always in the right place) that the changing process should wait until the original file was inactive, but didn't mention a usage counter. It isn't a big point, but isn't always provided by a system, so is a reasonable item to list as an initial assumption.

QUESTION 3.

(a) (i) The system configuration file supplies those parameters which are required to establish the initial state of the system when it is started, and which must be determined for each system. (It may contain other parameters too, but the variable ones must be there.) It is used to tailor the behaviour of the system to the local conditions of configuration and workload, because manual entry of the parameters, though possible, is tedious and error-prone.

The system configuration file is typically set up by a system manager, who must adjust the parameters to suit the workload and system concerned. It is executed by the system cold-start, and possibly warm-start, routines.

A personal configuration file supplies parameters which establish the initial state of a computer session. Such files are unnecessary if people using the system are all satisfied with a default

session configuration, but that is often not the case. Different people may require different screen management, different access to devices and software, or even different operating systems.

A personal configuration file is set up by its owner, and executed automatically as part of the login procedure.

Answers amounting to "The system configuration file describes the system configuration" gained no credit.

There was confusion between system configuration file and the system startup process. The configuration file is executed as part of the startup process; it prescribes how variable parts of the system are to be set up.

There was also much confusion between personal configuration files and the system's user data files. The personal configuration is the stuff you can change at will - .login, .cshrc, etc., files; the user data is normally inaccessible, at least in part.

(ii) Disc layout :

The system disc layout determines how the system discs are used. It may specify what areas and what locations are to be used for various purposes, such as system area, file system, swapping, spool space.

It is set in the system configuration file.

It must be included in the configuration file because it cannot in general be predetermined, for two reasons. First, different systems have different disc configurations, with different collections of discs of different capacities and speeds; second, each system's workload will make different demands for the various services provided, and will therefore require different layouts.

Search path. :

The search path is an ordered list of file directories. When a file is sought by name, the directories of the search path are used as base directories for the search until a file of the required name is found.

It is set in a personal configuration file.

It is useful to retain the search path in the personal configuration file because different people, having different sets of directories, require different search paths. The search path is therefore an attribute of the job or session, and cannot be set for the system as a whole.

You can lose marks by defining things insufficiently. "Disc layout determines the physical set up of the disc devices"; "The search path defines the default path used by the system" : both of those statements (slightly paraphrased) occurred in a single answer. They might be right, but they might equally be wrong - I couldn't tell because "physical set up" and "default path" weren't defined.

Most people didn't understand what I meant by "disc layout". The most common answer (I think - I didn't count) was about finding the file directories. I gave some marks, but as I've used the term fairly consistently (course notes, lectures, past examination papers, textbook (not consistently, but see Figure 12-2, page 386)) I thought you should have at least mentioned the meaning I intended.

(b)

	Batch	Interactive
Administration	Determine who can use the system under what circumstances.	
High-level scheduler	Determine whether a job can be accepted by the system.	Determine whether an interactive session may be started.
Low-level scheduler	Determine when a new process may be accepted into a running state.	?
Dispatcher	Determine whether a ready process may have access to the processor	
Interrupt	Give instant service to an interrupt request.	

Similarities : At short times and at long times, the sort of work done makes little difference to what must happen; something has to decide what the processor has to do next, and somebody has to do much the same sort of administration.

Differences : The differences in the middle range reflect the different emphases on efficient use of the system (batch) and efficient use of people (interactive) by supplying computing services immediately on request.

I made a mistake in just asking for a list of scheduling operations. People who had learnt the table got the marks for that straight away, though not necessarily any more. People who had seen the point of it (or maybe learnt the rest of the answer) got the rest. Of the others, some really knew what they were talking about, and did very well; the rest floundered.

(c) Process priorities are intended to indicate to the operating system some measure of the relative importance of the various processes in execution. The intention is that the operating system should take these priorities into consideration when making scheduling decisions.

Priorities can in principle be taken into account in a multiprogramming system whenever any decision must be made. In practice a simple-minded implementation of priority-guided scheduling can give unexpected or undesirable results. Within the operating system, priorities can usefully be implemented at any of the three lowest scheduling levels listed in the table above.

Low-level scheduler : processes of low priority may be deferred until processes of higher priority have been executed. This is a useful tool for load balancing in batch systems, where delays of minutes or hours are not very important.

Dispatcher : Priorities may be implemented either by making the ready queue a priority queue, or by adjusting time slice length or frequency of attention according to the priority.

Interrupt : In a system in which interrupt priorities can be implemented, it may be possible to exploit these to reflect the processes' priorities, but it is more usual to make the interrupt priorities reflect the urgency of dealing with the interrupts themselves.

If the choice of which process to run in the low-level scheduler or in the dispatcher is based on process priority alone, it is possible that processes of low priority will be held up indefinitely by a continuous stream of processes of higher priority. This phenomenon is *livelock* or *starvation,* and is not usually welcome; in practice, low priorities are regarded as retarding forces, not impregnable walls. One way to guarantee that all processes will eventually be executed is to increase the priorities of waiting processes from time to time; then their priorities must eventually rise to the level of those of the higher priority processes, so they will eventually run.

Answers amounting to "The process priority describes the priority of the process" gained no credit.

Answers implying that the priorities determined the scheduling order were less acceptable than the specimen.

There were many possible answers to the "How can they be implemented .... ?" part; I gave credit for serious attempts, but for full marks I expected some comment on the different levels.

Several people think that priorities can be implemented by including a priority number in each process's PCB; nothing else is required - scheduling happens by magic. For a sensible answer, it is necessary to say what the scheduler does with the priorities.

Despite claims made in many answers, a process priority is not an integer. It is true that in many systems integers are used to represent priorities, but that is quite a different thing.

QUESTION 4.

(a) Page fault : the sequence of events initiated in a paged virtual memory system when a process refers to an address in a page not currently resident in primary memory.

What happens when a process wishes to use a swapped-out page :

1. Suspend the process until the required page is read.
2. Find space for the page (perhaps including writing out another page).
3. Request the page, then wait until it arrives.
4. When the page arrives, amend page tables and page map appropriately.
5. Return the process to the ready queue.

"What is a page fault ?" does not mean "When does a page fault happen ?". Please answer the question set, not the one you wish I had set.

A high proportion of answers didn't include any reference to the process's waiting until the page had arrived from the disc. Many people also forgot to change the page table.

(b) In all cases, there is an initial sequence of four page faults which fills up the memory. I ignore these henceforth.

The frequencies of P and Q interrupts are equal :

The page requests are in this order :

PA1, QA2, PB3, QB4, PC5, QC6, PA7, QA8, PB9, QB10, PC11, QC12.

The allocations : page faults are in **bold** characters.

Cyclic

PA1	**QA2**	**PB3**	**QB4**
PC5	**QC6**	**PA7**	**QA8**
PB9	**QB10**	**PC11**	**QC12**

LRU

PA1	**QA2**	**PB3**	**QB4**
PC5	**QC6**	**PA7**	**QA8**
PB9	**QB10**	**PC11**	**QC12**

The frequency of page faults is the same as the frequency of interrupts for both processes. It is always necessary to use five other pages between successive uses of the same page, so neither algorithm can preserve a page long enough for reuse.

P interrupts are twice as frequent as Q interrupts :

The page requests are in this order :

PA1, PB2, QA3, PC4, PA5, QB6, PB7, PC8, QC9, PA10, PB11, QA12, PC13, PA14, QB15, PB16, PC17, QC18.

The allocations : page faults are in **bold** characters.

Cyclic

PA1	**PB2**	**QA3**	**PC4**
PA5	PB7		PC8
QB6	**QC9**	**PA10**	**PB11**
		PA14	PB16
QA12	**PC13**	**QB15**	**QC18**
	PC17		

LRU

PA1	**PB2**	**QA3**	**PC4**
PA5			PC8
	QB6	**PB7**	
		PB11	
QC9	**PA10**		**QA12**
	PA14		
PC13		**QB15**	**PB16**
PC17	**QC18**		

For the CYCLIC algorithm : The frequency of Q page faults is the same as the Q interrupt frequency. Five different pages are required between successive calls, so there is no chance that a page can be reused. The frequency of P page faults is only one half of its interrupt frequency. There are alternately three and four other pages required between successive calls, so half the time the required page is still there.

For the LRU algorithm : The performance is again exactly the same as that of the cyclic algorithm, though the sequence is different. The rather poor performance of this algorithm is a consequence of the iterative structure of the process, for which the least recently used segment is quite likely to be the next required - so several times a page is replaced immediately before it is required again.

P interrupts are three times as frequent as Q interrupts :

The page requests are in this order :

PA1, PB2, PC3, QA4, PA5, PB6, PC7, QB8, PA9, PB10, PC11, QC12,PA13, PB14, PC15, QA16, PA17 PB18, PC19, QB20.

The allocations : page faults are in **bold** characters.

Cyclic

PA1	**PB2**	**PC3**	**QA4**
PA5	PB6	PC7	
QB8	**PA9**	**PB10**	**PC11**
	PA13	PB14	PC15
QC12			
	QA16	**PA17**	**PB18**
PC19	**QB20**		

LRU

PA1	**PB2**	**PC3**	**QA4**
PA5	PB6	PC7	**QB8**
PA9	PB10	PC11	**QC12**
PA13	PB14	PC15	**QA16**
PA17	PB18	PC19	**QB20**

For the CYCLIC algorithm : The frequency of Q page faults is the same as the Q interrupt frequency. Five different pages are required between successive calls, so there is no chance that a page can be reused. The frequency of P page faults is again one half of its interrupt frequency. There are alternately three and four other pages required between successive calls, so half the time the required page is still there. (The other half of the time, it has *just* been overwritten !)

For the LRU algorithm : We win at last - or, at least, P does. After the initial period, P needs no page faults at all, because the comparatively infrequent Q interrupts always come after all the P pages have been used, leaving the single resident Q page as the least recently used. Q therefore still requires one page fault for each interrupt - but, then, it always did.

"Comment on the frequency ..." doesn't just mean "State what happens". It requires a little discussion; I think the obvious comment in the context is a suggestion of the reason for the observed behaviour, but I was open to others.

Some people did rather well by being careful; some did very badly by not being careful. I knew it would take some time to work out all the examples (I answer the questions too), so I deleted part (c) of the original question; nevertheless, far too many people didn't trouble to work it out. I would have thought that assignment 3 would have alerted you to the odd behaviour of memory management strategies. (I remark that a comment which turns up in course evaluation is that the assignments for 340 don't match the examination questions. I don't see any reason why they should, but in this case they did, and it made no discernible difference.)

Almost everyone who tried to work out the answer without working through the examples made a mess of it.

Quite a number of people got the wrong answer because they didn't carry on counting until a regular pattern was established.

Another significant group managed to work out (presumably by not reading the question) that a process with twice as many interupts would run at half the speed.

I was surprised that several answers were given in terms of overall interrupt frequency rather than the frequencies for the two processes. The overall frequency is a perfectly good answer to the question, but the separate frequencies (from which the overall frequency follows immediately, of course) seemed so obvious and gave all the interesting information that I didn't go any further. I think it's also much clearer when the cycle is complete, because the patterns of behaviour are more obvious.

In an even more confusing set of answers, the results were given as page faults per P interrupt. I think they were right, though.

I was also surprised, on reading an answer in which the frequencies for the different cases weren't listed, to discover that I hadn't asked for them. The answer demonstrated that the omission wasn't very important, as it's clearly impossible to present a sensible comment without stating the evidence.

People who just gave me three pages of incomprehensible tables got no marks. You may have done a lot of work, but if you knew what it meant you should have said so.

QUESTION 5.

(a) An interface is said to be *consistent* if the consequence of each possible input action is similar no matter what the system is doing at the time. The consequences need not be identical in different circumstances, but they should be recognisably and plausibly related.

The *mode* of a system is the set of its responses to input actions. It is determined partly by the operating system and partly by any programme which is being executed.

Consistency is important because it simplifies the interface. Someone who uses a consistent interface has to remember how to perform a few classes of actions which can be used in most circumstances, each class covering a set of loosely related operations which are effected in correspondingly related ways; with an inconsistent interface, each individual operation could require an

action which bore only accidental relationships with any other, so the burden of facts to be remembered could be much heavier. In a "modeless" system (which could better be called a "single-mode" system), there is only one set of responses no matter what programme is being executed. The system is therefore consistent.

Double-clicking has at least two meanings. To the Finder, double-clicking an icon, or anywhere on a line of the file list except on the name itself under `View by Name` , means "select and open", with "open" interpreted sensibly enough according to the nature of the icon; double-clicking the file name means "select for editing"; double-clicking the file name again (perhaps even without moving the pointer !) means "select this word", as it does in word processing software. The first and last meanings are not obviously related; the principle of consistency is at least bent. The "select for editing" meaning is just about consistent with the first interpretation, but, as it draws a distinction between file name and icon which is not usually evident, could itself be seen as an additional element of complexity. (That's tidied up a bit in Version 7 of the system : double-clicking, even on the name, in `View by Name` consistently selcts and opens; a single click on the name selects the name for editing.)

The menu bar is a quite elegant solution to the innate paradox of the modeless system : if you can't have modes, how can you run different programmes ? The answer is to retain the basic and consistent idea of selection, but to modify that by providing, in a *consistent* way, a set of things - actions, values, etc. - appropriate to the programme which can be selected. In this way, the variability inherent in running different programmes is exercised by the programmes themselves in offering different sets of things; the actions of someone using the programme remain consistent.

"Trash" does not behave consistently. Dragging a file icon to "Trash" removes the icon from the display, but the name still appears in many menus, and it is not uncommon to find that an attempt to make a new file of the same name will be met with the "Replace existing file ?" question - and that if you answer "yes", the new file appears in "Trash". It is also disconcerting to use "Trash" as a means of ejecting a disc and disposing of its icon; the connotations of gone-for-ever are certainly not consistently applied in this case !

(Part of the trouble is in the implementation. There is in fact no separate directory called Trash; instead, a file in "Trash" remains in its original directory, but is flagged as being in "Trash". The intention is presumably to simplify the implementation; if a separate directory were used it would be necessary to retain all of the file's path name somehow, whereas using the method they chose keeps that information in the ordinary directory structure. This behaviour underlines the difficulty of maintaining a system illusion when the visible behaviour of the system does not correspond to the underlying structure.)

(That was a Version 6 answer; some of the awkward bits have been tidied up in Version 7, so here's an alternative answer.)

It is hard to find a meaning for "Trash" which is consistently maintained in the system. If you think it means "destroy", you will hesitate before dragging the icon of a valuable disc to it; if you think it means "eject", then why do you drag file icons to it ? (and "eject" from the file menu does something different); if you think it means "erase", why can't you drag selected blocks of text to it from a text document ?

(In fact, there is a fairly consistent meaning : it is "Remove from the memory of the operating system" - but the point of the desktop is that people aren't supposed to have to know about the operating system. Would "Forget" be a better name ?)

"Consistency means that the user interface is consistent". If I didn't know before, I do now. Please realise that a statement like that is not an answer to the question.

Some claimed that consistency required that every action always caused exactly the same operation. That isn't even an ideal to aim at - it would be far too limiting.

Being able to do the same thing in different ways doesn't make the interface inconsistent - provided that you can always do it in the same different ways.

Some of the answers to the three points marked with • were too brief. They might have been right, or they might have been all their authors could remember. I gave less than full marks. A sentence of explanation, or an example of what you mean, is usually worth while. In particular, I did expect some explicit mention of consistency in each case.

Most people followed my party line throughout, which you obviously all know. It is worth recording here that some who didn't, and who supported their beliefs well, received high marks. I may be opinionated, but I'm reasonable. Consistently (appropriately), some who trotted out my opinions without support got low marks.

Some people thought that the behaviour of double-clicking on icons for files and folders illustrated the point. If they thought so, then - of course - it *did* illustrate the point, but I have no trouble in thinking of both these operations as "opening". This is an excellent illustration of the importance of the system genie : if people have different preconceptions of the machinery of the system then they will perceive the interface operations in different ways.

Notice that if double-clicking *almost always* opens an icon, that may be worse - for the exceptional cases - than two or more clearly distinguished modes.

One answer was that double-clicking "will always take you to the next level". (It was a good answer.) It's interesting in its total dependence on a sytem genie in which "level" has a very clear meaning.

Another was that double-clicking always selects (one click) then does the obvious next thing.

For the menu bar, I wanted some reference to the fact that the menus for different environments can be wildly different, and how this can be reconciled with the principle of consistency.

Many people thought it important that the same items always appear on the menu bar. That's nice, but it's almost like asking that all icons should look the same. The more important point is that the operations are consistent even if the words aren't.

That "Trash" items stay in the Trash isn't in itself inconsistent, though some people clearly find it odd. That you can still change the names of files in Trash is an example of consistency. Do not confuse consistency with nonconformity to your system genie. (It's therefore not particularly relevant to the question, but still an interesting point, that someone pointed out that dropping something in a rubbish bin wasn't a very good metaphor for keeping a safe off-line copy of your valuable files.)

(b) At the very least, the UIMS must generate a signal which tells the process associated with the window that something has happened; without such a signal, the interface click will not cause the process to do anything.

To ensure that urgent signals are handled immediately, they must be treated as interrupts : a process receiving a signal must be scheduled and executed forthwith. The process may then decide not to do anything, but that must be the process's decision.

On receiving the signal, the process must be able to acquire whatever information it needs to determine how it should react. For a mouse click, the significant information is the position of the pointer, and the nature of the click itself - which button, or (particularly for a one-button mouse) whether it as a double-click, and down-or-up. This information can be sent with the interrupt itself, which then becomes a message, or it may be available through subsequent procedure calls on the UIMS.

A process with several windows may receive correspondingly many sorts of signal. If the operating system can manage multithreaded processes, the signals can simply be directed to the

appropriate thread, and otherwise no change is needed. (It is then the process's responsibility to ensure that concurrent responses to signals from different windows do not interfere.) If processes cannot be run as separate threads, then the signals must all be directed through the same channel, and the process must identify the source of the signal when it is received. (This is the basis of the Macintosh's event queue.)

This part was not well done. Part of that is perhaps because of the phrasing of the question, though the intended topic of switching arbitrarily between different activities is identified. At least as significant, though, was that few people distinguished clearly between the user interface management system and the operating system. (That's a system genie problem; note that your system genie shouldn't be the same as that assumed by the Macintosh designers, because you're an expert.) Where you draw the lines in a specific system may not be at all clear, but in principle the user interface system, the operating system kernel, and an ordinary process are different entities. The question is about communication between the user interface and an ordinary process.

The question was phrased rather openly because I didn't want to rule out any answers. I didn't care whether you thought all communication between user interface and process was conducted by passing messages, or by some sort of system call, or interrupts, or what, but I did want a description of the necessary transactions and a plausible mechanism.

There's also confusion between a process and a window : there was mention of bringing a process to the foreground. (It's true that some aspects of the Macintosh implementation don't help to clarify the underlying principles !)

This question was about what signals are *produced by* the UIMS, not about what it does internally. It was also about what signals *must be* produced by the UIMS, so reasons were expected. Some answers said nothing about the processes or their management, and concentrated on managing the window; as that has essentially nothing to do with switching between activities, they received no marks.

Several answers included descriptions of the sorts of information which must be available with the signals. That wasn't really what I meant, but I could see why they gave the information, so gave some credit.

QUESTION 6.

(a) Heavyweight processes consist of a resource environment (memory, open files, devices, etc) and corresponding security domain as well as a single locus of control (the value of registers, particularly the PC and a program stack).

Threads are loci of control. They must exist inside some resource environment (called tasks in Mach) but they do not encapsulate this information. Thus several threads can be running within the same environment. This gives them the advantage of being easy to switch between due to the smaller amount of information which needs to be altered when switching context. Threads within the same environment also share resources by their very nature rather than having to request the right to share resources as would have be done with heavyweight processes. Threads also provide advantages with multiprocessors. A program which uses threads can have several threads running in parallel if moved to a multiprocessor.

(b) If the server calls fork everytime a request arrives the new process can handle the request. The blocking problem is certainly solved with this approach. However one of the advantages of a thread based server is that the resources are shared; with a fork based server only open files are shared. It is of course possible to request areas of shared memory between the fork generated processes but this adds to the complexity of the task. These are separate processes and switching between them still has greater overhead.

(c) For a heavyweight process to migrate means that all the resources the process had access to on the original processor are either accessible from the new processor in exactly the same way (e.g. files) or that copies of the resources are created on the new processor (e.g. memory). The resource environment must look the same to the process.

For a thread to migrate by itself to another processor seems a contradiction. There is some meaning in this if we consider that the descriptions we made above have detached the locus of control from the resource environment. The resource environment the thread is executing in changes and the locus of control just keeps on moving to a new processor as well as to a new instruction. This is similar to making a remote procedure call. A small amount of environment will migrate with the thread (register values, possibly some stack information) everything else will be new. For any of this to work there must already be a resource environment (or task) in existence on the other processor into which the thread can continue executing.

QUESTION 7.

(a) (i) If the client fails the server doesn't really need to worry because the send is asynchronous. The operating system must either ignore sends to non-existent processes or the server must be set up in to handle errors resulting from such sends.

(ii) If the server fails the client is stuck. Its service cannot be provided. In a situation like this there should be a timeout of the receive and the operating system should see that the receive is from a specific no longer existing process and signal the client or terminate it.

(iii) No. The server must receive the request and then send the result regardless and the client must send the request and receive the result.

A synchronous send would block the server until the client accepted the result.

(b)

```
procedure Lock;
begin
        send(Mutex, giveMeMessage);
        receive(Mutex, gotMessage)
end;

procedure Unlock;
begin
        send(Mutex, releaseMessage)
end;

process Mutex;
begin
        while true do begin
                receive(whichProcess, giveMeMessage);
                currentProcess := whichProcess;
                send(currentProcess, gotMessage);
                receive(currentProcess, releaseMessage)
        end
end;
```

QUESTION 8.

(a) **Device table :** an operating system structure which contains a device descriptor for each device in the system; the system's source of information on its devices.

Device descriptor : a structure which contains information about a single device, providing or pointing to all the information needed to use the device. This includes such things as the state of the device, what it can do, routines for standard operations, location of buffer areas, and so on.

Input-Output request block : an IORB is a structure representing a single request for service from a device. This may be a request for information, or a control instruction, or an actual data transfer.

Device driver : a device driver is a procedure which handles the administration of a device at the hardware level.

Interrupt handler : device-specific software which receives interrupts from its device. Different types of interrupt are distinguished, and dealt with appropriately.

(b) Smoothly to withdraw a device from service and install a replacement, it is necessary to ensure that transactions intended for the old version are satisfactorily completed, software changed to that needed to manage the new device, and communication with the system reestablished for normal running.

Closing down the old device : It must be possible to mark the device as temporarily unavailable in the device table, so that new attempts to open the device can be detected and handled by the system. (What the system does in such a case depends on circumstances, and isn't part of the question, but it may be possible to use an alternative equivalent device (if there is one), or to queue the request until the changeover is complete (sensible for a batch system), or engage in dialogue with someone wishing to use the device interactively.) Transactions already in progress can be completed; if the change isn't urgent, processes which have begun to use the device may be permitted to continue until they close it.

Removing the old device : The device may require special closing-down operations; these can only sensibly be incorporated into the device driver. Also, as well as physically unplugging the device and carting it away, it must be removed from the system tables. Its device table entry must be removed, and its interrupt handler (if any) must be unlinked from the interrupt handling system and deleted. Finally, the device driver (and interrupt handler) must be deleted. Both the device table and the interrupt handling mechanism must therefore be changeable without reconfiguring the system, and appropriate operator instructions must be provided to initiate these actions.

Installing the new device : This operation is the inverse of the previous one. The new interrupt handler and device driver must be loaded into the system, and linked into the input-output system as appropriate. Again, the device table and interrupt handling mechanism must be changeable, and the device installation procedure must be able to incorporate the required changes.

Making the new device available : If this device is to be used as a full replacement for the old one, its device table entry must in some respect contain identical information. This is typically some sort of device type field, set by operator instruction when the device is installed, and interrogated by the file system when a request to open a file using a specific device is received. With this field set, the device can be made available, and all should proceed properly.

The immediate implications are that there must be a modular system structure to make it possible, operator instructions to make it happen, and device-specific code to provide detailed instructions. (Not specifically requested, but a useful comment.)

Summarising the implications for the system architecture :

There must be system instructions which are executable by the operator to remove a device from service and to install a new device. (A single instruction to suspend and replace a device would be possible, but its only "advantage" would be that the new device descriptor could be made to occupy the same device

table slot as the old one, and this isn't particularly helpful unless you want to keep the device indices in the table permanently constant, which is generally not a very useful idea.) During the execution of this instruction, many system components will be affected, and each of these must be constructed to permit the necessary changes without affecting its performance.

The **device table** must include an availability flag for the device which is interrogated before any stream connected with the device is opened. It must also include some indication that the device is currently in use. It must be possible to mark a device table slot as empty

The **device descriptor** must be composed largely of pointers, for the procedures for the different types of device may be of different sizes, so cannot be preallocated specific memory areas.

The **device driver** must include any special procedures needed to close down the device, and the system must be able to execute these as part of the closing-down sequence. (Perhaps each device will have a closing procedure in the device table.)

The **interrupt handler** must be detachable from the interrupt management structures of the system, and the new interrupt handler must be insertable.

(Further comments on the secondary implications for system structure which follow from the constraints imposed by these primary requirements are also valid, provided that the necessity for the structure is clearly explained. For example, if device drivers and interrupt handlers are to be replaced, the system must not require that these be confined to a limited area of memory, or that a single memory range be used to identify areas subject to system-level protection and security.)

Many people spent a lot of paper on describing how you could send people warning messages, save pending requests, and so on (the easy stuff), and then magically disposed of the device descriptor and software (the tricky bit) in a few words, which in many cases didn't even mention the device descriptor and software.

Very few answers mentioned removing the old device driver and interrupt handler code; none (?) said anything about doing things to the interrupt vectors. It is not an accident that the question states that the replacement could be different from the original; it's to make sure that you allow for changes in device driver and interrupt handler.

Several people wanted to put requests to close down the device in an IORB and queue it in the device request queue. That's possible, but unnecessarily complicated; closing-down procedures should be available in the device driver, but they should be directly accessible from the device descriptor. (People who also wanted to use an IORB to install the new device might ask themselves what was going deal with the IORB.)

A few said that it would be done by using a supervisor call. They had presumably forgotten that this was an operating systems paper.

Several people seemed to expect that the request to change the device would come from an ordinary process. (Hence, perhaps, the IORBs ?) That doesn't sound very plausible to me. Certainly on a shared system you wouldn't want people arbitrarily changing system devices - and on an isolated system, there doesn't seem to be much point in making it depend on a process.

The second part of this part was comparatively rarely attempted.

(c) Wait for transaction to finish :

Advantages : Easy; don't have to make special provision for handling device errors; you always know the context in which any error happens, so evidence which may be useful for diagnosis is available.

Disadvantages : Slower processing overall - provided that there's something you could be doing while waiting. (And it may not matter if other processes can take up the time.)

Continue processing :

Advantages : Quicker if all goes well.

Disadvantages : It's difficult to guarantee that the context in which a request is issued will still exist when the result is received, without imposing constraints on the system which may nullify the advantages of parallel operation. If the request is made from a procedure at any but the lowest level of the dynamic stack then the procedure may have finished and the local context vanished by the time a response is returned. In extreme cases the process itself may have completed. This can be handled by standard procedures so long as all is well, but can be particularly messy if things go wrong.

I didn't insist that people answered with a 2x2 matrix; there are obvious complementary relationships between the entries !

Many people knew that there was something tricky about handling the result from an operation if processing has continued in the meantime, but rather few were able to say what the trickiness was. (I was a little suspicious of people who wrote "the context will be lost" without explaining what it meant.)

Some suggested that a process which continues to run can't react immediately to a fault; that's not necessarily so. The real problem is that it can be difficult to diagnose the fault.