Computer Science 340

Operating Systems

1995 EXAMINATION : ANSWERS AND NOTES.

Material in this font was added after marking the scripts.

It seems that an increasing proportion of stage 3 computer science students are illiterate to the extent of being unable to read the text on the front of the answer book. It says, on every answer book : "Begin the answer to a new question on a new page".

QUESTION 1.

(a) The file *must* have two components :

1 : The file attributes, which describe various properties of the file;
2 : The file data, where the contents of the file are stored.

The operating system must manage the attributes, so that it knows the properties of the file when it needs them to do things to the file. It need not know about the contents of the file, but it must manage the space where the contents are stored, in order to control disc use effectively.

To find the file, its name is sought in some sort of table maintained by the operating system and usually stored on the file volume with the files. From other information associated with the name in the file table it is possible to find the position on the disc of each component of the file. (Commonly the file attributes are stored in the table, together with the information needed to find the file data, but other sorts of file organisation are not uncommon.)

A Macintosh file has a third component :

3 : The resource branch, which contains various items used by the file but not properly part of the file data.

Not everyone stated that the system had to manage the file data !

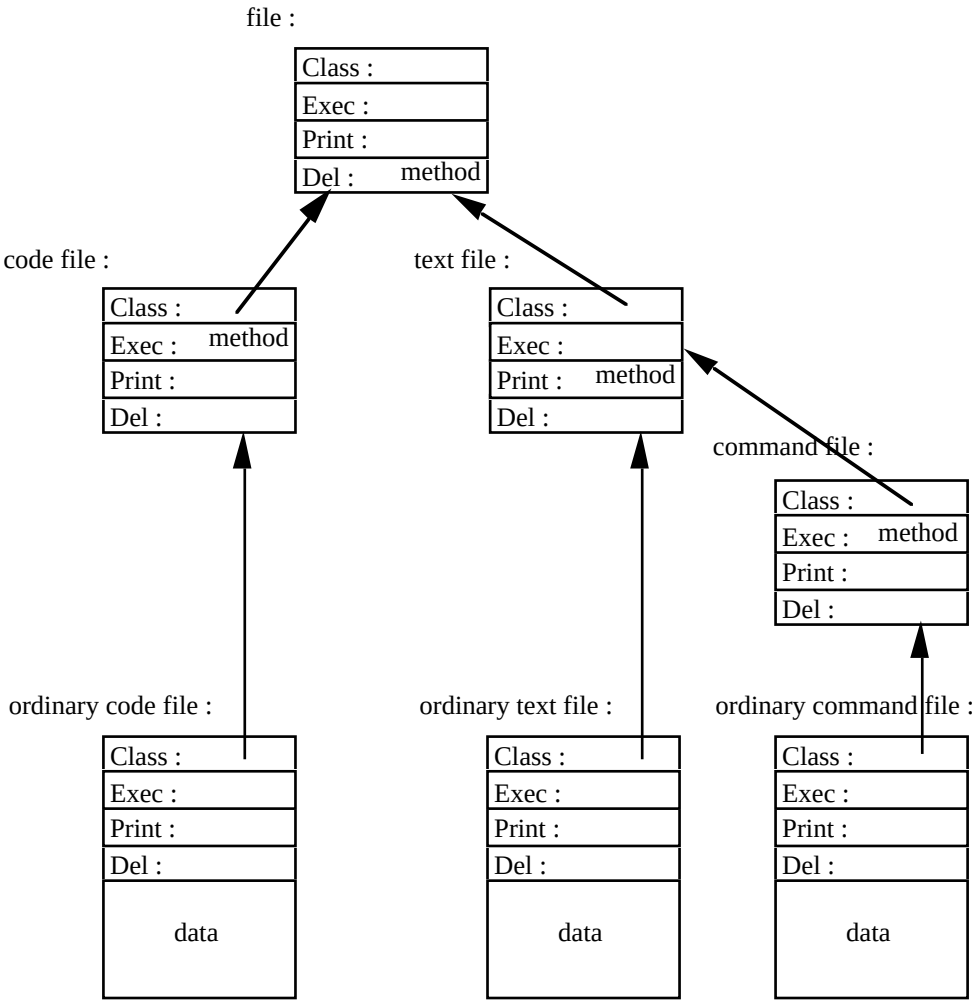
Several people answered that the Macintosh file was different because it appears on the screen as an icon. That got no marks, because isn't a property of the file. (It's a property of the system.)

As always, people didn't read the question. I asked for the components of a disc file; far too often, I was given the components of a disc system – file tables, directory structures ....

(b) The principle of an object-oriented file system is that each file should (in some sense) include methods for performing any applicable operation on itself. In practice, the same method can often be used for many files of the same type, so it is not sensible to store an individual copy of the instructions with each file. Instead, files are grouped into classes, with the methods stored in association with class structures rather than with individual files where possible.

This file organisation simplifies the user interface because it is no longer necessary to remember the details of any procedure used to perform an operation on a specific file. Any plausible instruction can be issued in respect of any file, and, if it's possible, the file itself will "know" how to perform the required action.

The diagram below illustrates how files of different types may share methods by inheritance. All files are deleted in the same way, so the delete operation appears as a property of the universal class file; all text files are printed similarly, but code files cannot sensibly be printed, so the print method must appear lower in the hierarchy; different methods of execution must be used for different sorts of executable file, so code files and command files have separate execution methods.



The important feature of an object-oriented file system isn't the classification or inheritance; it's that the methods are associated directly with the files. The structures are (very effective) means of implementing this important feature, and should certainly be included in "the ideas behind an object-oriented file system". (I gave marks for answers which stressed the structural aspect, provided that they made sense, and made some non-trivial reference to methods.)

Several people missed out the bit about the user interface.

An object-oriented file system is not the same as an object-oriented interface. The "select-and-click" interface design can be combined with a quite conventional file system (as in the Macintosh).

I gave you three operations to use in your description so that you could show how different sorts of grouping could be implemented by defining appropriate classes of file. People who didn't do that received fewer marks; it's one of the more important properties of the object-oriented system.

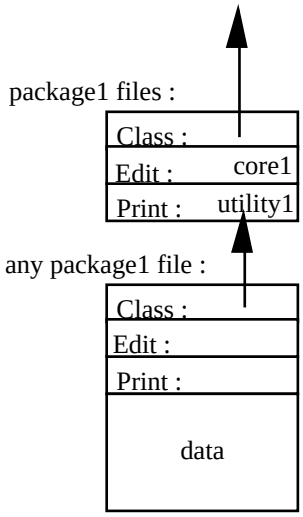
(c)

Likewise, people who didn't use the class structure in their answer to this part of the question got fewer marks. That isn't arbitrary; for separation of function it's important that the programme (in this case, the core) only needs to know the class to be associated with the files it produces. Changing the methods should be an operation on the class, not on the core programme or the files.

Rather similarly, I gave benefit of doubt in some cases where there was some suggestion that the core programme would know about the printing. The printing

utility is separate, so there's no reason at all why the core programme should be involved in a print request.

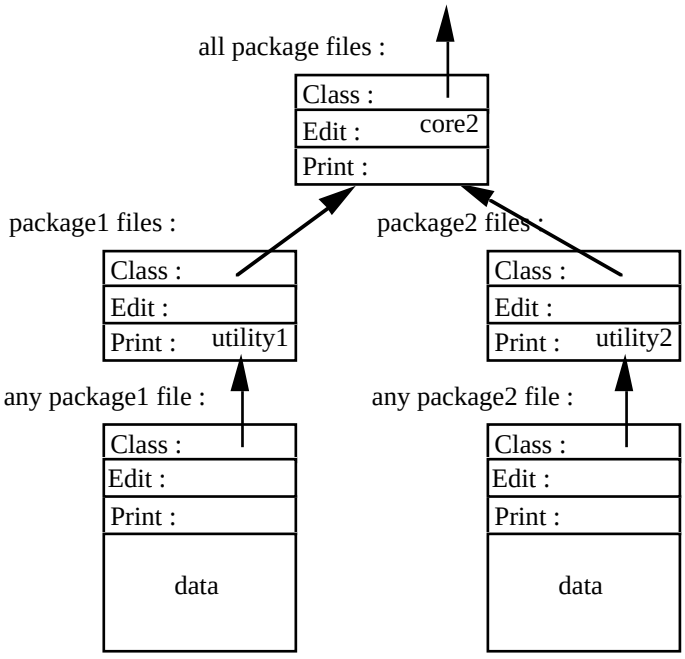
(i) A new class must be constructed, presumably as a subclass of some existing class, which identifies the core programme and utility as its editing and printing methods. The core programme must know the identity of this class (number, name, disc address, or whatever, as required by the system) so that it can make the link from its new files to the correct class.



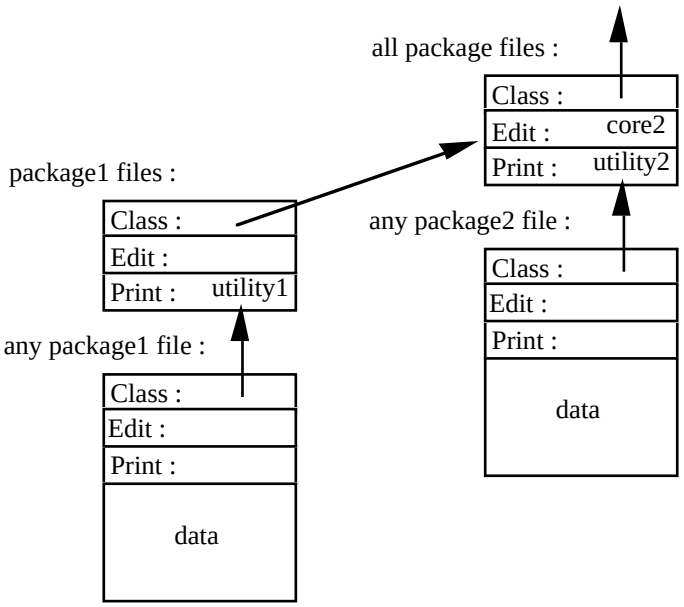
People thought of many interesting things which the core programme had to know, but very few thought of the class of the constructed file. As that is the one thing it needs to know which is unique to the object-oriented system, the others didn't get so many marks. (A possible alternative is the identities of the files which implement the methods, but the class is better.) I accepted (almost) any suggestion that the core programme had to know something to put into the file in some way to link it into the object system.

The Macintosh system uses the identity of (usually) the parent programme as the class, but that's a cheap way out, and doesn't lead to a satisfactory object-oriented system. How do you manage a hierarchy ?

(ii) The class structure must be reorganised so that there are two file classes, each inheriting the same core programme (core2), but with different printing utilities. New files will be associated with the new printing utility.



(There are several other possibilities, but I thought that one was the most obvious. Anything which preserves the principles of the object-oriented system will do. A rather neat alternative is to regard the old files as a special subset of the new files :



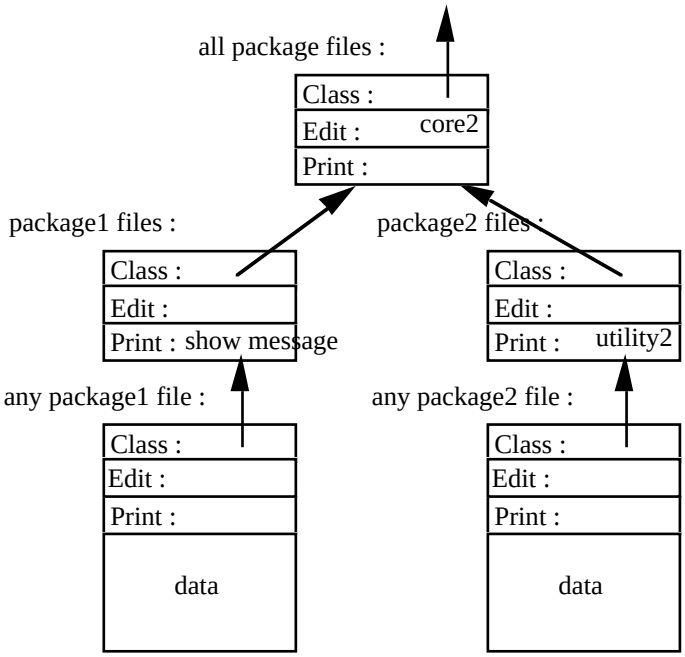
Once the old files have all gone, their branch of the tree can be removed, leading back to the original state.)

Many people said something about how to manage the printing programme, but didn't say what should be done with the new core programme.

An example of not using the class structure : "All files that were created by the old core programmes must have [their] methods changed to the methods for the new one ...". Another, quite common : "Any requests to the old package must be redirected to the new package" – with no indication of how that is to be achieved.

It's curious that many people who enthusiastically commended the possibility of associating different methods with different files in part (b) busily tried to avoid doing just that here.

(iii) The same structure must be maintained to cater for the old files, but their printing method must be replaced by something. It is sensible to insert a small programme explaining what's happened, and giving advice. ("Edit the file with the new core programme, and store it again.")



Pictures are not always worth a thousand words, but in this part of the question they were often worth a few marks. Some people stuck to verbal descriptions, and left me unsure of whether they understood what they were doing; a simple diagram would have made it clear – though perhaps that wasn't what they wanted !

Many people wrote statements such as "the core programme must be inserted in the class hierarchy". If that means that the core programme is identified as the editing method of a class in the hierarchy, that's right, but why didn't you say so ?

Others wanted to define classes of printable and editable files; the point about the object-oriented system is that *all* files are potentially both printable and editable, because it must always be possible to deal with print and edit requests. If in fact a file can't be printed, then its "print" method must be some sort of error indicator, but it's still there.

A surprising number of people insisted that all the files should be reformatted, including those off-line. They didn't say how that was to be managed. It is important to understand that an operating system must be designd to deal with reality.

Another answer to this part can be obtained by defining "something" in my answer in a different way : you could construct a new method which used the core programme to change the old files into new files, then print them using the new printer. Some people suggested that, and got the marks. I'm not sure whether that's better because it's automatic, or whether my way is better because there might always be someone who wanted the old format for some special reason.

One suggestion : "But why not keep the old one ? It can't take up all that much space...". Well, *I* don't know – perhaps the ancient printer finally broke down. How about I set the questions, and you answer them ?

QUESTION 2.

(a) Four values for the action parameter are conventionally distinguished, two (*get* and *put*) for data transfer and two (*control* and *status*) for administration.

Get requests transfer of data from the stream identified in the `fileidentifier` into the `data` area. The returned value of the `resultdescriptor` shows what happened during the operation.

Put requests transfer of data from the `data` area to the stream identified in the `fileidentifier`. The returned value of the `resultdescriptor` shows what happened during the operation.

Control requests the execution of some instruction on the stream (or the file or device associated with the stream) identified in the `fileidentifier`. Details of the instruction may be defined in some structure passed through the `data` parameter. The returned value of the `resultdescriptor` shows what happened during the operation.

Status requests information about the stream (or the file or device associated with the stream) identified in the `fileidentifier`. Details of the request may be defined in some structure passed through the `data` parameter, and the result may be returned through the same parameter. The returned value of the `resultdescriptor` shows what happened during the operation.

Different interpretations for different devices are achievable by associating procedures which handle the requests with the different device types in the device table. To deal with any of its requests, `DOIO` first finds the device associated with the stream using the file information block, and then uses the get, put, control, or status procedure supplied for the device in the device table.

I didn't insist on the set of operations given in the answer, but I did insist on a complete set – which does require that there should be means for issuing arbitrary instruction and requesting arbitrary information.

Most people (not all) got *get* and *put* (or *read* and *write*); many fewer got *control* and *status*, or anything equivalent. Instead, they started on a long list, often beginning *seek*, *open*, *close*, ... There are two reasons why the list won't do : first, it's hard to give a complete list, because it depends on what devices you're using; and, second, it can't possibly cater for all future eventualities, so it's a bad basis on which to build a stable system. Apart from that, *open* is questionable anyway, if you're implementing DOIO with a device table : it can't be handled by the ordinary method because the file information block isn't complete. I'd have accepted it if anyone had noticed that and commented accordingly, but no one did. (Consider the implementation : if the first parameter isn't always some sort of index into the process's file table, you first have to check the action parameter to find out what to do – then, if it isn't *open* (or one or two others which are related) you have to find the file information block, and the device descriptor, and *then* use the action parameter. Yes, it's possible, but it's very messy, and we'd rather not design systems that way. It would, of course, be nice if people who do design systems agreed with that point of view.)

I was puzzled by several answers in which certain possible actions were listed in this part, then another set (sometimes a better set) were used in part (b).

There seems to be a widespread belief that a *status* request needs no data. While that would perhaps be possible if the whole device status could be held in a single variable which could be returned through the resultdescriptor, that's hardly a reasonable general assumption to make, and it also ties up the resultdescriptor so that there's no channel for reporting real errors. It's more plausible to use a structure for the data, in which fields can be set to identify the sort of information required and to receive the results.

Many answers didn't include anything about the device table or device descriptors. I wanted some reference to the device table, or something equivalent, for full marks; if the descriptors were not collected together in some structure, they wouldn't be much use.

(b) (i)

```
calculate the sector number;
if the sector isn't in the sector buffer
then  begin
      if another sector is in the buffer
      then  if it has been changed
            then  put the resident sector back to the disc;
      get the new sector;
      end;
change the sector;
put the sector.
```

(The final *put* is necessary to ensure that the byte is written to the disc.)

(Apologies for the spelling mistake.)

Why did so many people waste time on telling me how to open a file when the question states that the file "is open for input and output" ?

- and I'm concerned that so many people don't know that you can't write a single byte into a disc sector. Apart from its being general knowledge, I talked about it in the lectures, and it's in the notes (for example, *Implementation*, page 55).

(ii) The instruction requires a random-access write operation. Given the `action`s defined above, this must be performed by a *control* action to find the correct position in the file, followed by a *put* operation.

```
DOIO( XYZ, control, { seek byte 3376 }, result );
```

The request is handled by the device's *control* procedure. It must execute this part of the operation :

```
calculate sector number;
if sector isn't in the sector buffer
then  begin
      if another sector is in the buffer
      then  if it has been changed
            then  put the resident sector back to the disc;
      get the new sector;
      end;

DOIO( XYZ, put, 67, result );.
```

The request is handled by the device's *put* procedure. It inserts the value given (67) into the current file position as determined by the preceding *control* operation.

```
DOIO( XYZ, control, ( flush buffer }, result );
```

The final *control* action is necessary to ensure that the altered sector is copied to the disc.

(And this one should say "What sorts of DOIO call ...".)

The details again depend on the operations defined, but full marks required a plausible sequence of DOIO operations, and an answer to "how would they be executed ?" including explicit mention of calculating which sector was required, reading from the disc, inserting the byte in the buffer, and writing the disc.

QUESTION 3.

(a) Different languages and other systems want to structure memory in different ways; such a memory structure is called a memory model. A memory model is rather more than a handy way to describe the operation of part of a programme. It amounts to an assumption about how the computer hardware works, so it must apply to everything that goes on. For example, Algol-like languages rely on a stack model of memory.

A description of any memory model is acceptable, provided that it shows what sort of address is used in the programmes, and how this is identified with a specific memory location in use.

Software support for "foreign" memory models is possible, but exceedingly expensive, because it requires action at every attempt to use memory. Software memory support is therefore usually restricted to services directly connected with the native addressing model of the computer hardware.

For the explanation of a memory model, I wanted two parts : that programmes wanted memory organised in different ways, and that the model was an assumption about the hardware. There were various ways of making these statements.

A purely verbal and qualitative description of a memory model got half marks. Because of that, most answers got half marks. In view of the intimate connection between memory models and addressing, I'd expected that a description of the model would naturally include at least a description of the address implied, and probably a note on how to map it onto a flat memory.

There is no such thing as a paged memory model; the memory model is the programme's view of memory, and paging is carefully designed to make no difference to the programme's view of memory.

There was some confusion between memory models, memory management techniques, and virtual memory systems. I did want an answer about a memory model; I'd gone to some pains to emphasise (in lectures and notes) that the memory model was determined by what the software wanted, so answers which only considered hardware or memory management didn't get many marks. Some people tried to have it both ways : "The memory model is how the memory is seen by the O/S and by the process ...".

Several people suggested that you couldn't run several memory models at once because they would conflict in some way, or they would be very hard to manage. That's not so; each programme has its own memory model, and it's no harder to keep it confined than to it is to stop processes straying into each others' areas.

In fact, you *can* run different memory models at once – that's why the question didn't ask why system *could* not support different models. You can do it with software, by interpreting, or with hardware, given a sufficiently microprogrammable processor. All have been done. You can have a different memory model for each process if you like.

(b)

Several people gave very peculiar answers, and – as they didn't explain their reasoning, as requested – got no marks. It is very hard to work out whether there's any sense in silly answers without some sort of clue.

Several people wrote things equivalent to assumptions that A's code included all the other code. That's not satisfactory; the programme is described as "Pascal-like".

Many answers were based on the assumption that all the sizes were identical. There is absolutely no basis for that assumption in the question. Why would I write

down all those Cis and Mis if they didn't mean anything ? (Well, maybe don't answer that.)

(i) All memory must be allocated separately. The total is $\sum_i(M_i + C_i)$.

Almost everyone got this one. People who didn't explain the reasoning only got half marks.

(ii) As there is no virtual memory, space must be provided for all code segments, but local memory need only be allocated for segments currently in use. The minimum requirement is :

$$\sum_i(C_i) + M_a + \max((M_b + \max(M_c, M_d)), (M_e + M_f)).$$

Quite a number of people didn't seem to think it necessary to provide space for the code of the leafier procedures all the time. They didn't say where the code came from when they wanted it. If they'd suggested that some sort of overlay system was being used, it might have made sense of some of the answers, but not all – though as a segmented model was specified it would be hard to justify !

- except for a few who explained carefully that it was swapped out, despite the clear statement that the system was "without virtual memory". Not many marks there.

If you sometimes want A bytes of memory, and you sometimes want B bytes, then the minimum total memory with which you can manage is the MAXIMUM of A and B. Not everyone worked that out. I assumed that the mistake was a slip of the pen, and gave benefit of doubt, but sometimes there wasn't much doubt.

(c) The compiler must be able to produce code to request local memory on entry to a procedure, and release it to the system when leaving a procedure. Possible forms of the functions are **allocate(size)** (which may return a segment descriptor) and **release(segmentdescriptor)**.

This part was almost trivial – if I hadn't been pushed for time when setting the examination, it would have been harder. Nevertheless, many people got low or no marks. (It was sometimes clear that people who had the right answer were worried because it was too short, and tried to pad it out with other material. That was entirely my fault; I'll try not to let it happen again.)

That was usually because they didn't answer the question. I explicitly mentioned "application programme interface" and "used by the programme" to emphasise that I didn't mean the system memory management details – but I got them, frequently.

There were some compromise answers – several people suggested **allocate** and **release**, but then spoilt it by saying that **allocate** (for example) "retrieve[s] the segment from somewhere (probably the hard disk)", with something corresponding for **release**. They got about half marks, possibly modified by other comments indicating more or less understanding.

QUESTION 4

a) Complexity

Simple locks are the simplest of the three. They require a single bit for the lock and some way of guaranteeing exclusive access by disabling context switches or indivisible instructions. They are accessed with lock and unlock operations. They may or may not have an associated queue of waiting processes.

Semaphores consist of an integer counter and the wait, signal and initialize operations. They are more complex than simple locks because the counter can take on many different values. All implementations (that I know of) include queues to hold waiting processes.

Monitors are the most complex of the three. A monitor consists of protected data and procedures with initialization routines as well as condition variables with associated queues. There is also a guard routine which catches all requests to enter the monitor and ensures that only one process at a time is running in the monitor.

Safety

Semaphores and simple locks both make it easy for mistakes to occur. Resources are only protected if they are used properly. Programmers must call lock or wait before using a resource and must signal or unlock it after finishing with it. There is nothing to stop a process keeping a resource indefinitely. Both locks and semaphores must be correctly initialized by an explicit initialization before any operations are performed.

Monitors were designed to minimise such possible mistakes. The only way to use the protected resource is via the monitor. Exclusive access is guaranteed by the monitor. As long as the monitor is written correctly the resource cannot be held onto indefinitely by the requesting process. The initialization routine is called automatically when the monitor is created, therefore no process can access it before the initialization is complete.

So why semaphores?

Semaphores are more general than simple locks, they can easily be used as simple locks and can also be used to coordinate multiple copies of similar resources. They make it particularly easy to implement producer/consumer processes with ring buffers.

While monitors are much safer, they have to be supported by the language the programs are written in, whereas semaphores are language independent. Semaphores are also more flexible than monitors. A monitor enforces exclusive access by making sure that only one process is running within the monitor at a time. This is sometimes an unnecessary restriction.

The use of semaphores to safeguard resources can be made as safe as monitors *(houses?)* by insisting that a particular process or subsystem is in charge of the resource. All requests to use the resource must be handled by this process which takes the place of the monitor. The semaphores are only used inside this process.

b) The wait on s2 inside the Wait routine stops any access to the general semaphore because s1 provides exclusive access to the general semaphore and it hasn't been released yet.

The solution is to release s1 before waiting on s2, as in:

```
wait(s1);
counter := counter – 1;
if counter < 0 then begin
        signal(s1);
        wait(s2)
end else
        signal(s1);
```

There is another problem with multiple calls to the General Wait. It is possible for a second process to call the General Wait and if it is lucky (following a couple of General Signals) to proceed before the process waiting on s2. This can be solved by serializing access to the General Wait routine by another binary semaphore.

s1 should be initialised to 1 and s2 should be initialised to 0. The counter variable is the value of the general semaphore.

c) Message passing is concerned with sending information from one process to another. This is the basis for the producer/consumer problem as well. We want to make sure no information is lost from the producer in the same way that we want to make sure no messages are lost when we do a send. We also don't want the same message being received twice, which is also a requirement of the producer/consumer problem.

The message passing system itself will provide these properties. There are two main ways it is done. Either the sender is blocked until the receiver can take the message or there is buffering of messages by the message passing system.

A semaphore solution to the same problem, can use analagous methods. This is still a little more complicated than with message passing because the shared buffer has to be explicitly

protected. Providing exclusive access to the buffer is insufficient to solve the problem, the producers and consumers must be synchronized to ensure that data is not lost or duplicated. Results can be buffered, and in this case producers need to be blocked when the buffer is full and consumers need to be blocked when the buffer is empty.

If the message passing scheme uses a form of indirect naming such as mailboxes then coping with multiple producers and consumers is trivial. The mailbox receives one copy of every result and passes it on exactly once regardless of the number of producers and consumers.

QUESTION 5.

(a) The access matrix for a system defines the permitted mode of access by any subject to any object.

A capability is held by a subject; it defines the subject's mode of access to a specific object. The level of access defined is granted to any subject who (or which) presents the capability.

An access control list is associated with an object, and defines the mode of access to the object for every subject. Any attempt to gain access to the object is checked against the list; if the desired mode is listed for the subject making the request, access is permitted.

Each capability defines the modes of access to a specific object available to any subject holding the capability. It therefore defines elements of the row (or column) of the access matrix corresponding to the object, but just which elements are so defined is determined by the way in which the capabilities have been distributed. The collection of capabilities held by a subject defines the subject's column (or row) of the access matrix.

Each access control list defines the mode of access of every subject to the object with which it is associated. It also defines elements of the row (or column) of the access matrix corresponding to the object, but in this case individual elements can be determined precisely and cannot be changed by external agencies.

This was essentially bookwork, but it seemed to be easy to make mistakes.

(b) *(Three possible answers for different capability models. I've given the same list of operations in each case, though some are unnecessary in specific cases, and therefore not required by the question. Comments are collected at the end.)*

Variable model, hardware implementation :

(i) A capability is a hardware-implemented data type. It is composed of an identification field, which determines the object to which it applies, and a list of privileges. In addition, all addressable memory (primary, secondary, etc.) locations and the corresponding processor registers incorporate a capability bit which is turned off if the location contains ordinary data but is turned on if the location contains a capability.

Capabilities are kept secure by the hardware, which is so constructed that new capabilities may only be formed while in supervisor mode.

(ii)

Operation	Implementation
New capability	Any system component which constructs or sets up an object of protected type (files, directories, devices, etc.) must, while running in supervisor mode, construct a capability variable including an identifier for the object and a complete set of privileges. This is converted into a capability using a "make capability" hardware operator, and can then be returned to the subject performing the action.
Copy	Conventional load and store operations may be used; the hardware must not change the capability bit. *Not required.*
Transfer	No special implementation is needed. A capability may be transferred from subject to subject in a message, in a file, through shared memory, etc. *Not required.*
Reduction	A hardware operator is required. It must accept the initial capability and a specification of the privileges to be removed, and construct a new capability representing the new set of privileges.
Validate and Test	A hardware test operator is required for validation. Ordinary inspection is not a sensitive operation, and need not be protected : the capability is disentangled to give object identity and privileges, and obvious tests are conducted. If the encoding is complex, or the object identities are difficult to interpret, a system procedure can be provided
Withdraw	There is no mechanism for withdrawal with capabilities implemented as variables. *Not required.*

Variable model, cryptographic implementation :

(i) A capability is an ordinary data structure, which can be implemented as a software data type. It is composed of an identification field, which determines the object to which it applies, and a list of privileges. In addition, it has a check field which contains a bit pattern which, taken with the identification and privilege fields, can be tested for validity by some cryptographic technique.

Capabilities are kept secure by the cryptographic technique. Only one pattern of the check bits can validate the combination of bits in the earlier fields, and the function which determines the check field is designed to be very hard to discover from examples of its results. With a large check field (say, 48 bits), trial-and-error methods would take too long to be worthwhile..

(ii)

Operation	Implementation
New capability	Any system component which constructs or sets up an object of protected type (files, directories, devices, etc.) must, while running in supervisor mode, construct a capability variable including an identifier for the object and a complete set of privileges. This is converted into a capability using a "make capability" function, and can then be returned to the subject performing the action.
Copy	Conventional load and store operations may be used; the capability is an ordinary variable. *Not required.*
Transfer	No special implementation is needed. A capability may be transferred from subject to subject in a message, in a file, through shared memory, etc. *Not required.*
Reduction	A supervisor call is required. It must accept the initial capability and a specification of the privileges to be removed, and construct a new capability representing the new set of privileges, which it returns to the caller.
Validate and Test	A supervisor call is required for validation. Ordinary inspection is not a sensitive operation, and need not be protected : the capability is disentangled to give object identity and privileges, and obvious tests are conducted. If the encoding is complex, or the object identities are difficult to interpret, a system procedure can be provided.
Withdraw	There is no mechanism for withdrawal with capabilities implemented as variables. *Not required.*

Privilege model :

(i) A capability is represented (for people) as an entry in the userdata system, or (for other sorts of subject) as an entry in some other system table. The entry for a subject lists the capabilities which have been acquired by the subject, identifyng the object concerned and the level of access permitted.

Security is achieved as with any other part of the system data; the information is not present in any process's memory, and access is only possible through supervisor calls.

(ii)

Operation	Implementation
New capability	Whenever a system component constructs or sets up an object of protected type (files, directories, devices, etc.), the operating system constructs a new entry in the capability list of the subject performing the action. The entry includes an identifier for the object and a complete set of privileges.
Copy	No local copying operation is required. *Not required.*
Transfer	A system call is used. It is given the identity of the recipient, the identity of the object protected, and a list of the privileges which are to be transferred, Either it constructs a new capability in the recipient's capability list, or, if the recipient already held a capability for the object, adds any new privileges acquired.
Reduction	This operation is included in the transfer operation. *Not required.*
Validate and Test	A supervisor call can be provided to control access to the capability list, but there is no reason why the operation itself should require any special protection.
Withdraw	A supervisor call can be used to request that the system withdraw specified privileges from identified subjects. The ability to withdraw privileges may itself be a capability, or it may be restricted to the original owner.

(i) Some people didn't say how the capabilities were represented, but nevertheless told me how they should be kept secure. If I thought that I could make sense of it, I gave marks, but more often I couldn't, and didn't.

Just saying that a capability is "represented as ... a long stream of bits" isn't enough. The structure is part of the representation too. Neither is it sufficient to say "Capabilities are a structure given to a subject" without saying what the structure is. (Both genuine quotations.)

A fashionable answer was along the lines of "One type of capability system is the Unix file protection system ...". I call it fashionable, because many people used it, and it has no rational basis. They all got no marks for this part of the question.

(ii) Full marks for any two operations, reasonably described. Descriptions were more important than names : six names of operations wasn't quite full marks. I did want to know that you'd done more than learn the list of names.

(c) The two methods are in many ways complementary.

Capabilities excel in their ability to pass permission from one subject to another. Some way to do so is essential if (for example) a protected file must be used by a system utility to perform some task for a subject authorised to use the file. The subject can pass (a suitably reduced version of) the required capability to the system utility. Similarly, it is sensible in a large organisation to permit a subject to pass on an appropriate selection of privileges to an underling for certain purposes, which is harder to manage if permission must always be sought from a central agency.

Access control lists can be made much more specific, and can be controlled by the object's owner at any time; changes are implemented instantly. Access control lists may be used to restrict access to certain members of groups even if all members of the groups have been given capabilities, or to impose short-term restrictions in special cases (such as a general restriction to read-only access while the owner require write access).

Answers which amounted to "two methods are better than one", without any discussion, got few marks. Any plausible example of a case where having two systems really did something useful got at least half marks; with some sort of discussion, full marks.

Not all answers made much sense. One suggested that having both systems could be a guard against people stealing capabilities; he didn't say how he proposed to distinguish a stolen capability from one which had been passed on legally.

QUESTION 6

a) A remote procedure call is a means of sending a request from one process, possibly across a network to another process. From the point of view of the programs at both ends of the transaction the calls look exactly like local procedure calls.

The first process (the client) makes an ordinary local procedure call to a stub procedure which has been linked into the program. The stub is responsible for making the connection to the remote process (the server). The client stub has to determine where the server process is and then packages up the request (marshals the parameters) in a way which can be understood by a corresponding stub procedure associated with the server. This request message is then sent to the server's machine via the network.

At the server's machine the message is passed on to the server's stub procedure. After unpacking the message and determining which local procedure is to be called, it makes an ordinary local procedure call to the server procedure.

When the server has completed it returns to the server stub, which packages up the results and sends them back via a message to the client stub. The client stub unpacks the result and passes it back to the procedure which made the original call.

b) Advantages of RPC

Easier for the application programmer to use. No need to work out the identification of the server, nor to construct sends and receives. Since normal procedure calls block the calling procedure until the called procedure returns, any remote procedure call also blocks the caller. Thus the state of the caller is unchanged when the result returns and there is no further need to synchronize the caller with the service.

This means greater safety as there is less chance to make a mistake. There is no need to worry about different architectures or system conventions.

Of course the work still has to be done somewhere. Someone (possibly not the programmer of the client or the server) has to carefully design the interface which is to be used by the RPC. This interface is commonly used by a stub compiler which creates the stub procedures which are linked in to the client and server.

The extra checking which the compiler does also helps to reduce the possibility of mistakes.

Disadvantages of RPC

It is less flexible to use RPCs. It is impossible for the thread of control which makes the procedure call to continue until after the procedure call has returned.

The implementation of RPCs is more complicated. As we have seen, interfaces need to be made available across the system, stub compilers need to compile the stubs, which then have to be linked into the client and server.

c) This answer ignores the problems of different data formats and the marshalling necessary to hide these differences.

There are no problems with the *operation* and *outBytes* parameters. Their values get passed in the obvious way, from the client to the server.

outAddress and *result* are the difficult ones. *outAddress* refers to an address in local memory which is totally meaningless in the remote environment and *result* would be passed by reference within a local procedure call.

All possible solutions send data representing the actual contents of memory between the server and the client, it becomes a question of how much data needs to move and when the moves are made. It is possible to copy all reference parameters to the server when the remote procedure call is made and send the altered copies back to the client on the return. If the RPC interface given to the stub compiler is designed to allow the programmer to specify that this parameter will only be used one way, then only one transfer has to occur. In the case of the *result* parameter the copy only needs to go from the server to the client. With the *outAddress* parameter the copy needs to go from the client to the server.

The client stub determines the number of bytes to send from *outAddress* by inspecting the *outBytes* parameter. The server stub has to allocate space for this number of bytes and fill the space in with the received values.

The server stub also has to allocate space for the remote *result* parameter (this parameter did not have to be sent to the server). On completion of the operation the values stored in this variable have to be sent back to the client. The server stub knows how many bytes to send from the type of the variable.

Back at the client the local *result* parameter gets filled in with the returned values.

QUESTION 7.

	How to do it :	*GUI comment :*
`cd B`	Search the current working directory (A) to find B; Make B the working directory.	The necessary information is the identity of the desired new working directory, and the fact that a change in working directory is required. To identify the directory, make B's entry visible in the A directory's window, and select it; the identity is retrieved from the position of the pointer when the selection operation is received, and the system's map of the screen. To identify the instruction, carry out some standard operation (double-click, select a menu item, etc.); the combination of action and context completes the identification.
`cp X Y`	A new file descriptor (directory entry) for Y is constructed; The file X is copied, and the copy associated with the new file descriptor; The file attributes are set in Y's file descriptor.	The original item, the new name, and the request for a copy are needed. The original item and request can be handled in much the same way as for the previous instruction. The new name requires more, because it isn't already there, so can't easily be made visible, and must be entered at the keyboard. (Neither Macintosh nor Windows provides a simple copy instruction, presumably because neither Apple nor Microsoft can think of a neat way to get the new file name. Windows provides copy as a menu item, and gets the new name through a dialogue box; in the Macintosh system, there is no explicit copy, and you have to make do by first duplicating the file then changing its name.)
`mv Z ..`	The file Z is sought in the working directory (now B); it is not found; an error is reported.	The information needed is the nature of the instruction, the object to be moved, and the destination. This is well suited to a GUI, because the two objects required are already present – but in this case there is no Z. *This instruction can therefore not be issued with a GUI*, because you can't issue an instruction about object Z unless you can find it, and you can't find it if it isn't there. This is an advantage of the GUI approach.
`mv X ..`	The file X is sought in the working directory (still B), and found; The entry .. is sought in the working directory, and found (it points to Directory A); A new file descriptor is constructed in Directory A, pointed at the file body of X, and filled in with the correct details; The X descriptor in Directory B is deleted.	The instruction is the same as in the previous example, but this time all the information is there. It may be necessary to ensure that objects (windows or icons) representing X and its desired new position (A) are both visible (trivial with the Windows file manager, may require manipulation if B's window obscures A's on the Macintosh). The instruction is represented by dragging the X icon from its old position to its new position. The sequence (mousedown on X icon in B window – mouseup in A window) completes the identification. In the Macintosh system, the parent directory cannot be automatically indicated by the mouse movement; in the Windows file manager, the parent of an open directory is shown (curiously enough, as ..), so a direct equivalent of the text instruction is available.

`rm *`	Each entry in the working directory is identified in turn and `rm` is applied to it; If the entry is a file, it is deleted, and the space which it occupied is returned to the system (unless there were other links to the file).	The system requires the nature of the instruction, and the selection of operands. The instruction is delete files – not directories. The operands are all files in the working directory. The delete instruction is fairly easy – drag to "Trash" on the Macintosh, delete key with MS-DOS, menu alternative for both – but it doesn't distinguish between files and directories. Selection is therefore necessary. The selection is harder; selecting all entries in a directory is reasonably easy, but selecting only files isn't. In the general case, you would probably have to pick them out one by one. Group selections can be built up item by item by controlling the context of the mouse operations; a click on an item while a specified key (shift on Macintosh, control in Windows) adds (or removes) an item to (or from) a selected group. (In the example, it's easy, because it happens that there's only one file left in B, but that's accidental.)

There were no parts in the usual sense to this question. Two responses were required for each of five instructions, so I've used the instructions as headings below.

A small, but non-zero, number of people ignored the directory structure given in the question,and therefore spent time discussing general issues which I'd tried to preserve them from. Perhaps this was a special case of people simply ignoring most of the question. An alarming number of answers included comments to the effect that there was insufficient information, and that it had therefore been assumed that directory B was within directory A, or that X was the only file in directory B, or that initially the working directory was A, or that the A window was open and a B icon was visible, or that the Unix instructions were executed. (I think all those were real examples; I'm not going back to check.)

Contrariwise, others went to the opposite extreme, and reduced rm * to rm Y (or, in several cases, to rm <nothing at all> – presumably they lost count somewhere). Even if there wasn't much to do, the questions are about the execution of rm * and its GUI equivalent.

Some general comments on the two responses :

(i) Many people found things to say which I didn't include in my answer (security checks were common), or missed out things I'd included (searching directories). I gave two marks for anything about the same size as my answers or bigger. (And, of course, correct.)

A curious answer turned up several times. It was "Run a programme called **cd** (or whatever) with parameters B (or whatever)" – without any comment on what **cd** was supposed to do. I suppose you could argue that the answer fitted the question, but only by asserting that what **cd** does isn't part of the file system, which I don't believe. I could also quibble that "Run a programme called **cd**" assumes a certain sort of implementation which isn't necessarily general, but I didn't.

(ii) You were asked to *explain* how the information needed to identify the instructions was acquired. Some people just told me what to do – for example, "double click on the folder labeled B" was given as a complete answer to this part for the first operation. What's the information needed by the system ? How is it acquired from the action ? It was also common for the instruction given to be specific to some system (either Macintosh or Windows), with no attempt to give a general answer. As I don't do half-marks, I had to give either 0 or 1 for such "answers"; I decided that I had to give 1, so some people got rather too many marks for this question. They were too many, because I don't want to give stage 3 marks for knowing how to use a Macintosh.

Very few people took the trouble to identify the information needed – so even fewer were able to tell me in detail how it was acquired. Hardly anyone thought that the instruction itself was information.

Having given me the details of how the system took notice of clicks and mouse positions in the first example, some people seemed to think that would do. It wouldn't.

An alternative was to describe the GUI operations, then to add a note at the bottom saying that all the information came from the mouse movements. I gave a few marks, but really wanted more specific details for the separate instructions.

Several people told me that there is now a "move" instruction for MS-DOS; thanks.

cd B To make a directory into the working directory isn't the same as opening it; many directories may be open, but only one is the working directory. Systems usually have some sort of pointer which identifies the working directory.

cp X Y This instruction, interpreted in the context given, can only mean copy as a new file Y in the B directory.

A surprising number of people didn't know how to copy a file in Macintosh or Windows without opening it and using "Save as ...".

mv Z .. Even people who hadn't mentioned searching directories elsewhere mentioned it for this one. But they still didn't mention it for the others.

In part (ii) I expected (but didn't always get) a comment pointing out that it was *impossible* to give the instruction.

Quite a number of people copied the file data as well as doing things with directories in (i). Why ? Unless the new copy is made on another disc (when mv doesn't usually work anyway), there's no point – you don't need the old copy any more.

mv X .. I wanted some mention of removing the original directory entry for full marks in (i).

rm * Considering that the information is given in the question, I wanted some comment about not deleting directories. In particular, answers to (ii) recommending "select all" or equivalents got no marks unless appropriately qualified.

Many people said, or implied, that programme, or procedure, **rm** was executed with ***** as parameter. It made no difference to the marks, but I'm surprised that few people apparently knew that wild characters such as ***** are interpreted by the Unix shell, not by the individual procedures. Having the shell do it is the only reasonable way to guarantee consistent interpretation.

QUESTION 8.

(a) (Several activities may depend on the clock. The sequence described here is the activity related to the dispatcher, and is executed after the other activities if a time slice ends at the clock interrupt.)

{ Stop the running process. }
1. Interrupt the running process (= start the clock interrupt handler).
2. Save the process's current state (machine registers, etc.) in the process's memory or PCB.
3. Set the PCB state to *ready*, attach it to the end of the ready queue.

{ Enter the dispatcher to start the new process. }
4. Remove the PCB from the front of the ready queue, set its state to *running*.

5. Set the current state from the saved state in the new running PCB or process's memory.
6. Reset the programme counter from the running PCB (= branch to the new process).

(b) (i)

Initial : { Frequency 10; time-slice 10; destination Other }
Other : { Frequency 1; time-slice 100; destination Other }

The product of frequency and time-slice is equal for both queues, so on average a process will receive the same share of processor time whichever queue it is in. The smaller frequency of the Other queue cuts down its context-switching frequency correspondingly by a factor of 10.

(ii)

Initial : { Frequency 100; time-slice 10; destination Other }
Other : { Frequency 1; time-slice 100; destination Other }

The processes in the Initial queue receive 10 milliseconds execution 100 times as often as processes in the Other queue receive 100 miliseconds of processing As before, the lower frequency of the Other queue cuts down its context-switching overhead by a factor of 10.

(c) Context switching is expensive in time, and adds nothing to the productivity of the system so far as raw processing is concerned. It's used because it makes timesharing possible, which can help in the productivity of people using the system, but if we can get away without it without compromising the people's productivity we would like to do so.

Some context switching is inevitable, as many processes' resource requests cannot be handled immediately, and if we are to continue to use the system some other process must take over. We would still like to cut down as far as possible on accidental context switches.

If the timeslice length has been well chosen, most processes will stop because of resource requests before their timeslices are complete. In such a case, the occurrence of a clock interrupt suggests that the process concerned is engaged on some solid processing without making additional requests, and is therefore ideally suited to use rather longer timeslices efficiently. By increasing the timeslice length and decreasing the dispatching frequency correspondingly, the same amount of processing is carried out, but the system overheads connected with context switches are reduced.