Computer Science 340

Operating Systems

1993 EXAMINATION : ANSWERS AND NOTES.

Material in this font was added after marking the scripts.

These comments are less compendious than those I have added in previous years, because the increasing volume of marking has left me less time to record my views. I hope I've saved the useful bits.

A general point : perhaps because of the recent introduction of some operating systems material, and particularly Unix, into the 211 course, there was a noticeably increased tendency for Unix features to appear in the answers as though they were laws of nature. They are not; answers about Unix are not acceptable where general treatments are expected.

QUESTION 1.

(a) To support the MEMORY MANAGER's view : the system maintains a MEMORY MAP of some sort, through which it can find areas of memory for allocation to processes requiring memory. To support the PROCESS's view, it maintains ADDRESSING TABLES which link virtual addresses appearing in its programme with real data addresses in memory. The same structure also contains the data addresses in secondary memory which must be kept for the virtual memory swapping operations.

The MEMORY MANAGER sees primary and secondary memory as quite distinct, and is only concerned to allocate primary memory, and corresponding secondary memory where needed for swapping, to the processes which require it, and to oversee the transfer of data between primary and secondary memory. It sees only pages or segments; the *addresses* of allocated and free areas are important to it, but the *contents* of the memory is of no direct concern.

The PROCESS using the memory is not concerned with the distinction between primary and secondary memory, except insofar as it must remember addresses for its addressing units (segments or pages) on both media. The contents of the memory is vitally important; but the actual addresses are not interesting. The process sees only the memory model implemented, usually a flat or segmented memory; it has no notion of pages.

Addressing tables (pages or segments) appeared much more in the answers than system memory maps.

There was some confusion as to what the processes could see, with a significant number of people wanting to put the addressing tables in the system area, together with the memory maps. I accepted that, because I suspect that's a better way of looking at the memory management system anyway. Next time perhaps I'll change the specimen answer round too. (And the lectures, and the notes, ...)

(b) Important factors in determining the behaviour are the number and size of the segments (both determined by the value of the parameter N), the demand for memory in the system, and the pattern of access to the segments.

The pattern of access is illustrated by the diagram; each cell represents an element of the array, and the numbers in the cells show the order in which they are required in memory.

1, 2, 3, 4	5, 7	9, 11	13, 15	17, 19	21, 23

6, 8	$4N$+1, $4N$+2 $4N$+3, $4N$+4	$4N$+5, $4N$+7	$4N$+...	$4N$+...	$4N$+...
10, 12	$4N$+6, $4N$+8	$8N$+...	$8N$+...	$8N$+...	$8N$+...
14, 16	$4N$+...	$8N$+...	$12N$+...	$12N$+...	$12N$+...
18, 20	$4N$+...	$8N$+...	$12N$+...	$16N$+...	$16N$+...
22, 24	$4N$+...	$8N$+...	$12N$+...	$16N$+...	$20N$+...

It is clear that the first row of the matrix is in steady demand until the end is reached, after which there is a steady demand for the second row, and so on. The numbers down the left-hand side show that there is also another, quite different, pattern; each row in the matrix is required twice, but then ignored for a time depending on N.

For small N, both the number of segments and their size is small. Unless memory is very congested, it is quite likely that all segments can be brought into memory at once, and that the interval between the successive references to the lower rows is sufficiently short that they will still be available when needed again. If these conditions are satisfied, the algorithm will run smoothly.

As N increases or the system becomes more congested, the long gaps between references to the lower rows make it likely that the rows will be swapped out before they are required again, so greatly slowing down the algorithm, and further delaying the successive references : a memory fault is likely to be generated for each row once this pattern is established. The comparatively frequent references to the current "top" row are more likely to keep it in memory.

With yet larger values of N, even the top row may be swapped out between cycles, leading to even slower operation. An additional constraint is the possible difficulty in finding a large enough memory area to accommodate the large segments.

(If N becomes very large indeed, it may not be possible to allocate a sufficiently large segment at all; in practice, a segmented system must make provision for "paging" very large segments to make sure that this case can be handled.)

Plenty of marks were lost because people made unwarranted assumptions – for instance, that N was a large constant.

(c) The important factors in determining the behaviour of the algorithm in a paged virtual memory system are similar to those important in the segmented system, except that the size of the array is of much less direct importance, as it no longer determines the size of the memory requests.

For sufficiently small N, it is likely that the whole array can be accommodated in a single page, so once the algorithm can start there is no obstacle to its finishing quickly.

As N increases, the difference between the two disciplines becomes less marked. For comparatively small N, each page will for a while contain more than one row of the matrix, so the number of requests may be reduced. As the length of the row becomes comparable to the page size, the difference will vanish, and for larger values of N the behaviour will be quite similar. Though several pages will now be needed for the current top row, each is frequently used while in memory; and for the elements in the lower diagonal requests for segments become requests for appropriate pages.

The paged systems escape one hindrance found in the segmented memory : there is no longer a requirement to find very large memory vacancies for large *N*.

Plenty of marks were lost because people made unwarranted assumptions – for instance, that pages were bigger than segments, that segments were bigger than pages.

QUESTION 2.

Lots of people got far too much credit for this question. There were many tiny parts, so any benefit of doubt (of which there was often plenty) made a big difference.

Line 1 :

Macintosh equivalent : The A window is at the top of the desktop.

Comparison : The Macintosh state is immediately visible : the active window identifies the current directory, and a selection of the contents of the directory is normally visible. The Unix state is not visible, though the current directory is accessible through the `pwd` instruction, and the files therein can be listed with `ls`.

Line 2 :

Macintosh equivalent : double click on the folder B within the A window; double click on the folder D within the B window. (OR : click on the folder B within the A window, then select `Open` from the file menu; click on the folder D within the B window, then select `Open` from the file menu; OR equivalents involving O , etc.)

Unix execution : execute the `cd` system programme; `cd` looks in the working directory for B; opens B as the new working directory; looks in the working directory for D; opens D as the new working directory.

Macintosh execution : locate the click within the A window to identify the parent directory; identify the click within the B icon or line to select the file within the directory; identify the function required by observing the double click or O or whatever; attempt to open B, find it's a directory, and display its window either by bringing it to the top of the stack if it's already open or by building it from the disc directory information. Then repeat for D within B.

Comparison :

for the system : the Unix implementation is much simpler. The input instruction is always in text form, and there is never any ambiguity about the destination of the instruction. The Macintosh system must interpret the actual instruction (the click) in terms of the current position of the cursor on the screen.

for the person : the Unix instruction is quicker if you can type and you know what to type, but its meaning is far from intuitively obvious. The Macintosh instruction is much easier to remember, but it takes at least two cycles of operation (and may be significantly more complicated if the icon of B or D isn't initially displayed within the parent window).

Line 3 :

Macintosh equivalent : drag X from D to the icon of C in A.

Unix execution : execute the `mv` programme; find `X` in the working directory; find `../../C` in the working directory (note that no special action is needed for the "name" `..` : it's in each directory, just like any other name); check that `C` is a directory; transfer the directory entry of `X` from `D` to `C`.

Macintosh execution : identify `X` and `D` by the position of the click as before; identify `C` from the position of the mouse-up; transfer the directory entry of `X` from `D` to `C`; redraw windows as required.

Comparison :

for the system : apart from differences in the detail of the way in which **X**, **C**, and **D** are identified, and the Macintosh's obligation to maintain the windows, the two operations are very similar.

for the person : the Unix instruction will always work without any preparation, but again its meaning is far from obvious. The Macintosh instruction is easier to remember, but you may have to prepare the ground by making the `C` icon (or the `C` window, if it's open) visible.

QUESTION 3.

This question was carefully constructed to be helpful. Part (a) is basic material about the supervisor call, part (b) leads you on to say what Unix has to do for its protection system. So in part (c) you obviously take the stuff you said in (a) and apply it to the stuff you said in (b). Don't you ?

(a) A supervisor call is a special form of procedure entry which can be intercepted by the operating system and which changes the operating mode of the hardware to supervisor mode, in which additional "dangerous" instructions are available. Its protective properties are connected with the combination of these two features : to use the "dangerous" instructions, a process must run in supervisor mode; to switch to supervisor mode, it must execute the supervisor call instruction; and executing the supervisor call instruction switches control to the operating system. The "dangerous" instructions are typically those giving a process access to operations which could affect other processes or the operating system.

Lots of answers including nothing about the "forbidden" instructions; some didn't mention supervisor mode. That left a supervisor call which didn't really do anything useful.

(b) Associated with each file there is a set of protection indicators, which record the level of protection of a file with respect to reading, writing, and execution for access by the file's owner, other users in the owner's group, and anybody at all. Any attempt to use a file is rejected if the access mode required is not permitted for the user making the request.

To implement this principle, the first requirement is that there must be no possibility of file access without using the system's file management procedures.

It requires that the protection codes themselves are secure, and can be changed only by the file's owner; that the identity of someone attempting to gain access to the file be known; and that access to the file cannot be gained except through the system procedures.

In order to evaluate the protection level to be imposed, the operating system must know who owns the requesting process, who owns the requested file, and - if necessary - who belongs to the group of the file's owner.

The requester's identity comes from the requesting process's process control block; the ownership of the file comes from the file directory; group membership tables must be maintained by the system. The integrity of the file management procedures, as well as other system information mentioned later in this sequence, must be guaranteed : there must be no possibility of unauthorised access to the code in memory, which would permit someone to patch out identity checks.

> "Seen" was intended metaphorically. I was surprised that many people took it literally, and only told me about the **ls –l** display. Fortunately, that included most of the answer.

(c) First, to ensure that people cannot write their own file access code, input and output instructions must be among the "dangerous" set which can only be run in supervisor mode. This forces a supervisor call for any file access operation, which, as well as changing the processor mode, both guarantees that the system knows the identity, and therefore the owner, of the requesting process, and causes a switch from the process's memory space to the system's.

Some essentially administrative operations such as opening and closing files are guarded because they all require access to the file information stored on the disc. Others, such as those which seek information from memory tables, are not dependent on input or output, so in these cases supervisor calls must be forced in other ways. In practice, that isn't a big problem because the system routines operate in a different memory space, so supervisor calls are necessary for entry. The operating system can therefore ensure that the proper sequence of operations is observed.

For a request to open a file, the system finds the file's attributes from the file system, and checks access. If the requested access is available to all, no further check is necessary; otherwise, the system must determine whether the requester is the file's owner, and, if not, retrieve group information from the userdata files. The permitted access level can then be stored in the file table, or other convenient place, so that it need not be redetermined at every operation on the file. Checking permission after the file is opened is then simply a matter of inspecting the stored information.

Of course, none of this is much use if the requester can reach into the system code and patch out the security checks, which is why system routines have to work in a different memory space. MEMORY PROTECTION is therefore necessary, even apart from the need to force supervisor calls; this can only feasibly be managed by hardware, as every address used must be checked for validity.

The process control block and file tables, or other places where a record is kept of the permitted access modes for the file, must also be kept secure, as changes to these once access has been checked could override the security system. Memory protection will look after this, too. Information kept on the disc, such as the file directories, and the group membership tables must all be protected, at least against changing the items required by the protection scheme. It may be acceptable to permit read access to these values, and other parts of the structures may or may not require protection.

> Many people assumed that supervisor calls couldn't be interrupted. That's a Unix peculiarity which is quite unnecessary in the real world.

(d) The principle is that I must be unable to use any knowledge accessible to me to subvert the system. This can be upheld in two ways : either I must not have access to dangerous knowledge, or I must not be able to use it.

I must be able to acquire *some* knowledge of the file system. At the very least, I must be able to inspect the lists of files in my own directories. It is also reasonable that I should be able to see something of the system outside, if only to find out what's available; and the protection codes themselves imply that I can at least try to read and write other people's files. I must therefore have some, possibly filtered, access to other directories.

This does not mean that I can have normal read access to the directories. If I were able to read any directory, I would be able to find out file names and protection (which I can sometimes do anyway), and also information about file location. I can only be allowed read access if the information I can obtain is of no use to me in subverting the system.

As the mechanisms constrain me to using the system procedures to gain access to material on the disc, and the system procedures will be guided by the information in the directories, I will be unable to use any knowledge I have if I can't change the information.

It is clearly unsafe to permit me to write other people's directories, or I could change the protection codes (or, in Unix, the owner of the file) as I wished to permit me any sort of access I wanted. But it is also unsafe to let my write even my own directory. If I can read other people's directories, I can copy information about their files into my directory, and so gain whatever access I wish. And if I can't read other people's directories, I can just guess – make dummy disc addresses for files and go fishing.

I can therefore not be permitted under any circumstances to write directories. Provided that the rest of the protection system works, then, there's no particular reason why I should be forbidden to read them.

Therefore : if I have access to a directory, I may be permitted to read it, but must not be permitted to write it.

The last three words of the question were the most important, but widely ignored. Most answers were stated as if received by revelation.

QUESTION 4.

Another old problem turned up again. Words like *backup* and *archive* have been used in many different ways. Having gone to some trouble to define them fairly carefully in lectures and in the notes, it's not unreasonable for me to expect that you will either stick to my conventions or – if you really must speak in IBM or whatever – give me a translation.

(a) **System backup** is a system-wide mechanism designed to provide protection from accidents, which aims to ensure that a system file store can be returned to its state as it was at some - preferably recent - known time. Its main advantage to the system's users is that it limits the damage which can be done by a system collapse to the work done since the last backup operation.

File generations are intended to offer protection against erroneous alteration or removal of files. It ensures that previous versions of files are automatically saved, so that they can be retrieved in case later versions are in any way damaged. The main advantage of this technique is that it provides an "undo" function for any operation which results in gross changes to a file.

A **file archive** is a long term, off-line store for selected files. It provides safety against system failure, as does a backup system, but adds the advantage of long term storage and control over storage time. Backup systems only guarantee to restore the system to its state at the most recent backup; even though older versions of files are often retrievable from an incremental backup system, the total life time is not defined and backup times are not related to the times of significant changes to the files. An archive system provides facilities for better file management, and can reduce the demand for the system disc by providing storage elsewhere.

(b) A **discretionary archive** is an independent file system, with little formal connection with the conventional file system. It maintains directory information on line so that it can respond sensibly to people's instructions. Several generations of a file can commonly coexist in the archive. On receiving a request to archive a file, the file is copied immediately to a holding area (so that the owner can now delete it or change it with impunity); all other requests are saved for execution when the archive job is run, typically at night. Being a separate file system, it is administered by a set of instructions completely separate from those used for the ordinary file system. It need have essentially no interaction with the conventional file system.

Necessary services are archive and retrieve requests, and some way to get a directory of your archived files.

An **automatic archive** acts as an extension to the conventional system. It decides for itself when a file is to be archived, on the basis of some criterion such as time since the file was last used or amount of disc space available. It must interact closely with the conventional file system, both to acquire the information it needs to decide which files should be archived, and because the archived files should appear in the system directory in the usual way. In addition, as it is in effect an extension to the conventional system, it must obey the conventional system instructions as far as possible, so that operations (copy, delete, rename, etc.) requesting actions on archived files must as far as possible be implemented correctly - or, at least, be reported in appropriate error or warning messages.

There are no strictly necessary services other than those implied in the preceding paragraph, as archiving is automatic, and retrieval can be triggered by attempts to use files which are currently only stored on the archive medium. On the other hand, some services are useful, such as explicit archive and retrieve requests. Without requests of such types, an automatic archive system is much less useful for file management, and becomes much more like a backup system. (Of course, there are other means to attain the same ends – it is easy enough to guarantee that a copy will be saved by making a copy under a different name – but it is much more satisfactory to provide this service if it's needed than to leave it to the whim of the file's owner.)

(c) (This was a real question. When it was set, the Computer Science Department had just begun to operate such a backup discipline, but without an index.)

The function of the file archive is to provide a long term off-line store for selected files; this function would be satisfied by the eternal backup if the "selected" requirement could be met. It is not met automatically, as, though the backup saves *all* the files, it only does so at certain times, so one can no longer decide to save copy of a file in the archive at arbitrary times.

The archive directory tells you what files you have in the archive, how many versions there are, and when they were archived. An index to an eternal backup system would need to maintain at least that information. Unfortunately, there is no easy way to limit the information to the files which you would have archived had you been able to do so, and it is almost certain that there will be far more files saved than you want. A simple directory is therefore unlikely to be adequate. and some means of searching the information would be needed. Obvious facilities which could be provided are substring searches and time limits on searches; others can be imagined (list the contents of any directories called "bbb", list any files made by MacWrite, etc.) could turn out to be useful in practice, but are less obviously essential.

This is another case where the question should have helped you. In (a) you defined backup and archives; now you can use those definitions to assess how closely the eternal backup satisfies the requirements for archive, and to discuss the importance of any defects.

A remarkable number of answers began by saying that an additional archive was absolutely necessary because you would never be able to find the file you wanted in the vast backup. That was the whole point of the question about an index.

QUESTION 7.

This question was very badly done – which was quite surprising, because the first part had been set before, and I haven't changed the answer much. A few people had obviously learnt the answer, and got good marks. Judging by the number of part (b) answers which began with instructions to open the file, it may have been because you didn't read the question.

(a) The **DOIO** procedure executes a standard algorithm in all cases, with specialisations for specific operations and devices as determined by individual devices procedures accessible from the device table. The basic execution pattern of the **DOIO** procedure is :

1. Find the `file identifier` in the file table to determine the device.

2. Check that the device (or file, on a multiple device) is available.

3. If the action can be executed immediately (such as a request for status information), do it, and return with the result encoded in the `data` area or the `result descriptor`, as appropriate.

4. Construct an *input-output request block* identifying the process requesting the action, the `action` required, and containing any other relevant information such as data addresses, and attach the IORB to the device request queue.

5. For an asynchronous operation, or if it is known that the device will not respond, return with an appropriate result code.

6. For a synchronous operation, suspend the process.

7. Wait for the device to respond.

8. For an asynchronous operation, change the value of the `result descriptor` to its final value.

9. For a synchronous operation, return with an appropriate result code.

(b) (i)

```
calculate sector number;
if sector isn't in the sector buffer
then  begin
      if another sector is in the buffer
      then  write the resident sector back to the disc;
      read the new sector;
      end;
change sector;
write sector.
```

(The final "write" is necessary to ensure that the byte is written to the disc.)

(ii) read sector :

```
doio( XYZ, control, { setsector ??? }, result );
doio( XYZ, read, buffer, result );.
```

write sector (each time):

```
doio( XYZ, write, buffer, result );
```

(That assumes that the sector number has been retained since the preceding read instruction. If the file is in use by any other process, it may also be necessary to set the sector number before writing the buffer.)

QUESTION 8.

Uninterruptible supervisor mode turned up again. It wasn't any use here either.

(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).

"Put the running process at the end of the dispatcher queue, get the new process from the front" is a summary of a common answer. It got few marks. (You could work it out from the question of (b), anyway.) I wanted rather more detail for marks.

(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 the same for both queues, so on average a process will receive the same share of processor time whichever queue it is in. The lower frequency of the Other queue cuts down its context-switching overhead by a factor of 10.

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

The product of frequency and time-slice is ten times as much for the Initial queue as for the Other, so on average the processor time received by processes in the queues will be in that proportion. As before, the lower frequency of the Other queue cuts down its context-switching overhead by a factor of 10.

Many people thought that the size of the timeslice or the selection frequency was sufficient to determine the share of the processor time. It wasn't. Several others argued correctly that you need a 1:10 ratio of timeslice lengths to satisfy the administrative overhead requirement, but then reduced the "other" queue's timeslice by a factor of 10 in part (ii).

The reference to "other processes" in part (i) misled a few people. It was intended to be read in much the same way as part (ii), but we didn't notice the possible misinterpretation before the examination. Fortunately, the misinterpretation led to an answer which (should have) contained the expected answer.

(c) The algorithm described guarantees that every queue will be inspected within a finite time, so that, even if work is arriving faster than the system can handle it, each queued process will be executed within some finite time. With the strict priority queue system, a steady stream of work in the high priority queues can hold up lower priority queues – in theory – for ever.

In practice, a random algorithm which selected processes from the queues at with relative probabilities according to their priorities would almost certainly be just about as good – but because it's random, you can't prove that it really would avoid starvation. The real objection is the unprovability, not the performance.

In practice, whatever the scheduling mechanism the rate of arrival of work on average must be within the processor's capacity, or you'd buy a new one (or another one). This requirement makes sure that even the random algorithm will process everything eventually, but a strict priority system can still lead to unsatisfactorily long delays at times of heavy loading.

The quotation was meant to be from part (b), but again we missed the fact that the original had been changed. No one seemed to be worried by that.

There were two parts to the answer – one on the significance of non-random, which required comment on what a random procedure would do, and one on the "strict priority" algorithm. Many people missed the first; I think just one person commented on the connection between non-randomness and provability.