Computer Science 340

Operating Systems

1997 EXAMINATION : ANSWERS AND NOTES.

Material in this font was added after marking the scripts.

NOTICE

These answers are offered now as a service. We believe that they are correct, but it is not impossible that before marking is complete we will have had reason to change our views. Wait for the final version which we will make available after completing the marking before complaining.

GENERAL.

It is sad, but clearly true, that a significant proportion of the student in the class are unable to read the clearly printed instructions on the cover of the examination answer book, where it says (among other things) :

Begin the answer to a new question on a new page.

and

Write in the margin the number of each question attempted. Do not use the margin for anything else.

Some of the instructions we don't care about, but those are important. I have wasted a lot of time scrabbling through answer books looking for practically invisible beginning of questions, and it is not conducive to the benevolent frame of mind which tends to easy marks. It wastes more time checking that the marks have been correctly transcribed to the cover, because the question totals can't be put at the top of the first page of the question.

If you are concerned about the ecological implications of wasting the occasional half side of paper, we sympathise, but as few answer book are completely used anyway the difference is tiny. Do not emulate one student this year who had the effronery to write "Save the trees" before an answer which started in the middle of the page, while leaving blank the reverse sides of all the pages in the first answer book and using a second answer book for a single page. Incidentally, the instructions also say :

Use both sides of each leaf.

QUESTION 1.

(a)

For the system input : the UIMS must receive input from the input devices, carry out any interpretation necessary to convert the input into the signals required by the software, and direct the signals to the correct processes.

Note that the question was about what the UIMS does when signals arrive – not what the people have to do to use the system. Not everyone made this distinction.

It was also about the transfer of information between the system and user; moving the pointer in response to mouse signals is significant, but I wanted a lot more.

Not many people noticed the "converting the input" bit – that covers things like double clicks, clicking on components of the window other than the active part (close box, maximise, etc.), use of menus, and other administrative activities.

For the system output : the UIMS must be able to deal with instructions which it receives from several processes running in the system, and display them correctly, maintaining the required relationships between windows on the screen and satisfying any timing constraints.

Note that the question was about what the UIMS does when signals arrive – not what the processes have to do to use the system. Not everyone made this distinction.

(b)

(i)

The system must maintain a list of windows and their dimensions and positions in order of "height".

- and (of course) which processes own them.

I got a lot of extraneous information about this one. You don't need to know where the mouse is to redraw the windows.

Some answers went into detail about keeping lists of damaged regions which would have to be redrawn, and other means of managing window maintenance. I didn't really want those – but to make those work you still need the basic ordered list I gave as the answer.

(ii)

Starting from the bottom of the list and working to the top, send requests to the processes to redraw their windows.

Nobody found a simpler way. Several people reported more complicated ways, and got rather fewer marks.

A few people explicitly gave the reversed order. I assumed it was a natural slip under examination conditions. More to the point, the principle was there, and you'd realise what was wrong as soon as you tried it.

Despite my saying that the processes had to redraw the windows, several people said that the simplest way was for the UIMS to keep bitmaps of all the windows.

One script gave a more complicated answer (beginning "that area(s) affected are worked out ...", which proves that it's more complicated), with the correct answer – labelled "horrible ... yucky" – opposite on the page headed "**Notes Only – Will NOT Be Marked**". What would you have done ? My big mistake was to look at the notes page. (I generally try not to.)

(iii)

First, it must search down the list of windows until it finds one which includes the position of the screen pointer when the click was detected.

Then, if this window was not the top window, it must move the item to the head of the list, and request the process to redraw its window.

Several people just said "find the window at the point of the click". What if more than one window includes that point ?

A common answer was something like "work out which is the new top window, then do <answer to (ii)>". You only need to redraw the new top window.

(c)

Command file	Script file
Sequence of textual instructions	
Executed as a "programme" by the system	
Can be written as text	
Direct copy of normal input	Interpreted copy of normal input

Familiar language	Unfamiliar language
No change required to a programme run by the file.	A programme run from a script file must be *scriptable* – it must be able to understand the script language as well as the normal GUI language.

Several answers mentioned that you would write a command file but record a script. You can equally record a command file, but you don't necessarily need a special programme to do it – just keep a log of the input.

There was a fairly common view that the command files were simpler. Examples given didn't convince me; accidents of implementation make some of the scripts look more complicated, but that has little to do with the nature of the files.

A surprising number of answers suggested that a script file for a graphical system would contain screen coordinates. It doesn't work; the actions have to be translated into words which use objects' names; that's one of the harder bits.

It is the last point which complicates the implementation of script languages. If a programme only understands GUI instructions, there is nothing that the operating systems can do to provide these instructions. Even keeping an exact record of the screen operations used in a sample session won't do, because the resulting actions might depend on the graphical environment, and there is no general way to guarantee that the environment will be the same every time the script is run.

Very few answers made this point. Many worried about conveying graphical information, but hardly any expressed concern that you might not be able to communicate any information at all.

QUESTION 2.

(a)

This was supposed to be an easy bit so that everyone could get some marks. I thought you'd know this from stage II. I was wrong. That made a mess of much of the question for quite a number of people.

A supervisor call is a special form of procedure entry with which a process can communicate with the operating system under carefully controlled conditions.

I asked you to explain the nature of a supervisor call, not to describe what it's used for. I accepted answers which resembled my specimen, and one or two others where interesting ideas about what happens in a supervisor call were expressed. I didn't accept answers which said that a supervisor call was used to manage dangerous operations or to get out of a process's memory space; those are consequences of the way the call is set up, not the nature of the call itself. (The nature of a car is to provide a means of personal transport for a few people, not to go shopping, to waste petrol, and to endanger pedestrians.) Any answer which begins "A supervisor call is used to ..." is going in the wrong direction. (But I didn't stop reading.)

A large number of people think that a "supervisor call is called only by the operating system". Doesn't that destroy the point of it ?

The supervisor call causes a transfer of control to a location which is determined by the system (hardware or software), not the running process.

The supervisor call changes the operating mode of the hardware to supervisor mode, in which additional "dangerous" instructions are available.

(b)

It is *not* sufficient merely to describe a page table; I asked for an explanation. The difference is not trivial, for a description of a page table doesn't say what happens if a page number outside the defined range is used.

Any address generated by a process running in a virtual memory system must be mapped to a real address using some form of address table.

Only two things can go wrong : you can try to use a chunk that isn't there, or you can try to get outside the bounds of a chunk. Nothing else can go wrong; I expected something like that in the answer. People who vaguely wrote about poorly defined "means of checking" without any further details got fewer marks.

With a paged system, provided that pages are allocated as a whole (so that processes will never share pages unless the sharing has been established deliberately), safety is assured if attempts by the process to use page numbers not present in its page table are trapped.

You can't get outside the page.

With a segmented system, a similar argument applies, but it is also necessary to record the length of each segment and to trap any attempt to address a location outside the range of the segment bounds.

You might try to get outside the segment.

In either case, it is clearly necessary that the process should not be able to write to its own address table.

That means that you can't implement memory protection without memory protection.

I asked for an explanation of a *guarantee*. I didn't expect a formal proof (and didn't get one), but to justify a guarantee you do have to say something about the mechanism, and why you're sure it will always work. Statements like "the system monitors memory addresses to check whether they exceed the bounds" (not actually a quotation) are not much more than restatements of the question.

This is a quotation (apart from some punctuation which I've corrected) : "Therefore if the OS manages the physical memory correctly there should be no problem guaranteeing that a process is confined to its own memory space". That's all; it doesn't go on to say "This is how to manage memory correctly ...". So it amounts to saying that the operating system will work if the operating system works.

Either a large proportion of people have got pages and segments inverted ("all segments are the same size"), or people who've got it wrong were specially keen to tell me about it.

(c)

To ensure secure access, it must be guaranteed that a process cannot operate a device without using operating system procedures. It is therefore necessary that the instructions used to drive the devices are not available to the process. For a system with explicit device-management operations in the code, these instructions can be made executable only in supervisor mode. For a system using memory-mapped input-output instructions, the system must ensure that the "address" of the device never appears in a process's address table.

The process is therefore forced to use a system procedure to operate the device. It can only enter the system routines through a supervisor call, and the system can guarantee to catch all such calls.

Far too many students believe in magic. Answers like "Devices can only be accessed by system calls" were common, as though it were a law of nature. (It isn't.) It is also not essential for a supervisor call to be uninterruptible.

Given memory protection but no supervisor call, a process is permanently confined to its own address space, so any device operation must be accomplished by operators within that space. Either they have to be usable, which gives no protection, or they are not, which gives no device operation. Neither of these is acceptable.

Given a supervisor call but no memory protection, it is possible in principle to make sensitive operations accessible only in supervisor mode, but it is impossible to prevent any process getting into supervisor mode; all it has to do is plant a branch back to itself at the destination of the supervisor call.

QUESTION 3.

I am amazed. Not very many people attempted the question, and of those who did most managed to do rather badly. Part of the reason was perhaps my unfamiliarity with semesters during the course; my reorganisation of my notes for a year's course into the rather shorter time

available in a semester sometimes worked and sometimes didn't, so occasionally topics were treated rather more briefly than I'd hoped.

Having said that, though, the material needed for this question is all in the distributed notes, and the topic of memory management was covered at stage II; I'd expected the question to be popular. Certainly for anyone who knew what it was about, this question was a lot of easy marks (which a few people got) as compared with some of the others.

(a)

Very few of the very few answers mentioned segments and pages. The pattern (if there was any coherent answer at all) was to think up something that sounded a bit like a chunk of memory, and to describe it briefly, and not very well. Argument was notably absent.

In designing a programme, we use top-down analytical methods which decompose the problem into successively smaller components until we reach a level of detail at which we can translate the specification into code and data definitions. By this means, we define logical units of code and data which are related by the requirements of the software, and are therefore likely to be handled as chunks; these are the programme's *segments*.

The addresses used in a computer's memory are strings of binary digits. From the hardware point of view, it is very easy to divide the memory into chunks of the same size, 2^n addresses, by regarding the last n bits of the address as an index into a *page*, with the rest of the address identifying the page number.

(b)

(i)

(Memory address)	(Disc address)	(Present)	(Dirty)
segment 1	disc sector of M	yes	?
?	disc sector of A	no	?
?	disc sector of B	no	?

(ii)

Step 1 : A is brought into memory. There is an available segment which is large enough, so nothing need be swapped out.

(Memory address)	(Disc address)	(Present)	(Dirty)
segment 1	disc sector of M	yes	?
segment 2	disc sector of A	yes	no
?	disc sector of B	no	?

Step 2 : A's memory is changed. The dirty bit is switched on.

(Memory address)	(Disc address)	(Present)	(Dirty)
segment 1	disc sector of M	yes	?
segment 2	disc sector of A	yes	yes
?	disc sector of B	no	?

Step 4 : B must be brought into memory. There is no available segment, but the least recently used is segment 2, currently occupied by A. (Segment 1 is currently in use as M is in execution.) A's dirty bit is set, so A must be written to disc, after which B can be loaded.

(Memory address)	(Disc address)	(Present)	(Dirty)
segment 1	disc sector of M	yes	?

segment 2	disc sector of A	no	yes
segment 2	disc sector of B	yes	no

Step 6 : A must be brought into memory. As before, segment 2, must be used, but this time B's dirty bit is not set, so A can be loaded immediately.

(Memory address)	(Disc address)	(Present)	(Dirty)
segment 1	disc sector of M	yes	?
segment 2	disc sector of A	yes	no
segment 2	disc sector of B	no	no

QUESTION 4.

If Question 4 was too hard, this was too easy. (It was too easy anyway.) Many people tried it, and most got quite good marks. I don't mind the marks, but they were too easy to get. My mistake was not to tie it down well enough – I should have asked you to draw the structures you recommended, and to justify them. I didn't accept suggestions that seemed to be unworkable for any reason, but I gave some benefit of some doubts which I would have preferred not to do.

(a) (i)

Several (not very many) people misunderstood what I meant by "components". One clarified his difficulty by asking if I meant that the names could be arbitrarily long or that they referred to files with a number of "forks". Well, the question does say "file names may have arbitrarily many components", which is what I meant. Note that I do sometimes make mistakes, but I also try quite hard to use words precisely. It is also the terminology I used during the lectures, though as you're not required to go to the lectures that doesn't count for much. It isn't prominent in the course notes, though it's a bit hard to believe that anyone who's as much as glanced at the chapter "NAMES OF FILES" would misunderstand. The misinterpretations also practically destroys the question, leaving nothing particular to answer. If you got the wrong question, I marked it, as well as I could, but gave a maximum of half marks.

Any plausible directory will do, but it has to work.

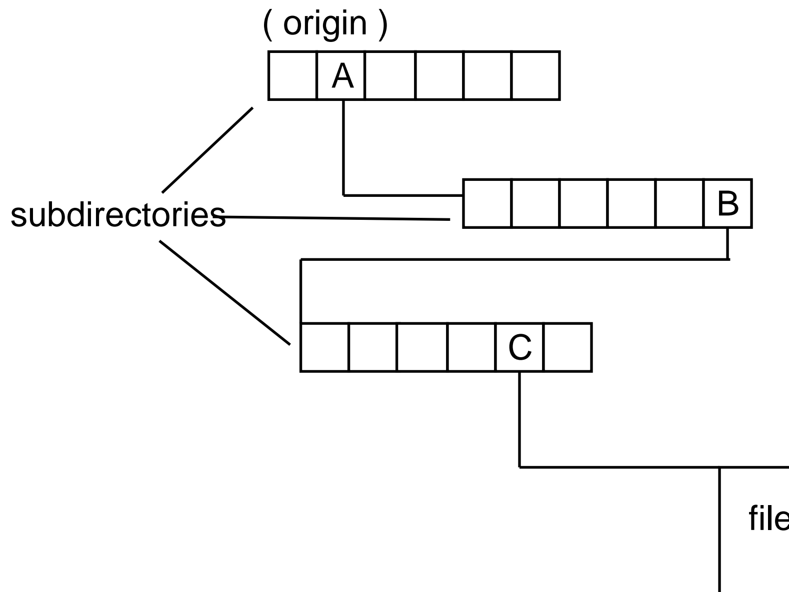
And that means that it has to be described in enough detail to convince me that it will work, and that you know how it will work. "Using a linked list system combined with an index table files can be given an unlimited number of properties" – well, yes, but that could mean a number of things, and the somewhat vague accompanying diagram wasn't much help.

If you want to use a flat directory, you have to say how you accommodate the file names of arbitrary length.

Two people gave interesting answers based on the Linda system.

A hierarchic directory can be used. It is organised as a sequence of named subdirectories, each containing links to further subdirectories or to files. The names of the subdirectories are the components of the file names. Each subdirectory therefore corresponds to a subset of the components, with the final component choosing the file itself.

A file called A/B/C will be linked into a structure looking something like this :



A diagram is a big help. Many people might have lost marks because I couldn't understand their verbal descriptions.

(ii)

The directory must deal with (at least) names, attributes, and links to the items which it manages. These data can conveniently be kept within the subdirectories – so, for example, the entry in subdirectory B for the link to subdirectory C could include :

name : C
attributes –
    type : directory
    protection : ....
    date : ......
link : disc address of C

Alternatively, the attributes can be stored with the file, which must then have a structure in which the attributes can be stored and retrieved :

header –
    type : directory
    protection : ....
    date : ......
data : contents of C

Considering that all our students are experienced programmers who've been using moderately structured languages for years, why is it that almost all of them describe structures with unstructured text rather than the common notation for structures which I've used above ?

(b)

(i)

The main advantage of using a volume directory is that there is likely to be much less overhead in finding and opening files when searching the directory. If separate files are used for each subdirectory, then, to find a file with several components to its name, each component implies another disc search for a new subdirectory. With a single directory for the disc volume, there is much more chance that the required directory will be in memory when required, so the disc traffic is much reduced.

It said "Discuss ..". I wanted a bit more than a simple statement that a volume directory would be faster because it was compact.

A directory tree in a single file is no *easier* to search than one distributed over many files; the structure is exactly the same. It's quicker, but that's different.

One advantage given was that it was easy to delete the entire directory at once. I'm still not quite sure whether that was intended seriously.

(ii)

A general linked tree structure is appropriate. Whatever structure is chosen, it must be very flexible, so that subdirectories of arbitrarily large or small size can be used without unduly wasting memory. It should be possible to insert items at arbitrary points to maintain lists in alphabetical order of name. If multiple links to files or directories are desired (and why shouldn't they be ?) it helps if structures for subdirectories are independent.

A structure composed of items of this form is appropriate :

> link to next entry in this subdirectory;
> this entry (as in part (iii) above).

The root of the tree is identified as a pointer to the first entry at the lowest level of the tree.

I expected something which would obviously work. In particular, it had to accommodate an arbitrary number of files – so "entries consisting of a name and a set of pointers" wasn't enough.

The most common defect was the lack of any suggestion of how to cope with unpredictable and arbitrary growth of subdirectories.

A few people suggested a hash table, which is all right as a mechanism but difficult for anything but searching given the whole pathname – so relative addressing (almost universally used in practice) and names including wild characters are harder to manage. The question does specify "A conventional file directory", so such things are expected.

"An appropriate structure ... could be a FAT file structure" is all right, provided that you go on to describe the structure. You do have to convince me that you know what you're talking about; this course is still rather more than an introduction to Microsoft vocabulary. (But wait for corporate sponsorship.)

QUESTION 5.

(a)

A spin lock is a simple loop which repeatedly tests the value of some variable waiting for the state of the variable to indicate the resource is available. They are avoided forgeneral resource allocation because they tie up the processor, stopping other processes from running on that processor until the process has completed its timeslice (or found the resource available). There is also no control over which of several waiting processes will get the resource when it is unlocked.

If the resource is currently unavailable a spinlock on a single processor causes no more useful work to be performed in the system until the waiting process is not running. With a multiprocessor the other processors can still carry on (and hopefully release the lock).

(b)

If the resource becomes available shortly (which may happen, since the current holder is running) there is no need for a context switch away from the waiting process. Context switches are expensive and the hope is that the resource will be freed in less cycles than a context switch would consume.

(c)

If the resource is available it is possible for more than one process to mark it as busy, if they both check at the same time.

There is something queued to use the resource. Just after the resource is marked as available but before the process from the queue has marked it as busy another process calls lock and finds the resource available.

There is nothing on the queue, the holdingProcess is not runnng. As the currentProcess is about to go to sleep the holdingProcess starts and calls unlock. It sees nothing on the queue and so the requesting process is asleep possibly indefinitely.

QUESTION 6.

(a)

This is good if the file is to be accessed from different machines because the copies can be stored locally. It works well if the files are read only. But the file system has to know this file is duplicated and where all the copies are when changes need to be made to the files. The information about all copies must be stored with the directory entries of each copy. They could be stored in a separate table but in that case we could use method c) below. It takes up lots more disk space.

(b)

Similar to (a) but this saves disk space. Each directory needs to know about the others when updating. If it is a distributed system we can't just duplicate the directory entries unless the location information is valid system wide.

(c)

There is no need to update directory entries whenever the file is altered. The directory entries don't need to know about each other. The table has to be valid system wide, regardless of the number of disk devices and their positions. The table needs to keep track of how many entries refer to this file.

QUESTION 7.

(a)

(i)

A simple round-robin scheduler would be adequate. With each process in turn being allowed to run. This ensures fairness. Interactive response can be provided by having a small enough time slice.

The interactive response is ruined if the number of ready processes grows too large.

(ii)

Keep one ready queue. Allow each processor to choose the next process from the queue.The queue has to be kept consistent by enforcing mutual exclusion.

(b)

(i)

Response times should be good, since the foreground process gets immediate attention. The foreground process is privileged, regardless of how CPU intensive it is, it runs whenever it can.It can be argued that this is a good thing, because the foreground process is the one the user is most concerned with. Background tasks, such as printing may be held up a very long time. It is particularly bad for IO intensive background processes because whenever they wake up they go to the end of the queue and may have a long wait to run again if all processes in the FCFS queue are long CPU intensive tasks and the user doesn't do much so that they aren't preempted. If the user has a lot of CPU intensive tasks it is good for them because when the foreground process is not running they can run (or one of them can) without the overhead of repeated context switching.

(ii)

No. It would need to be extended to cope with multiple foreground processes. We would need some form of preemption on the foreground tasks (which wasn't necessary in the single user situation). We then have a standard two queue situation with foreground tasks always having priority over the background ones. The behaviour of the background tasks is highly undeterministic. (Actually processing could be more evenly distributed than with a single user because there would be more preemption of background tasks.)

QUESTION 8.

(a)

Increased flexibility.

A process can communicate with several processes via the same mailbox.

Two processes can communicate with each other via more than one mailbox.

A process doesn’t need to know the name of a process it is communicating with, just the name of the mailbox. e.g. In the case of a client/server relationship, even if the server changes the client code doesn’t have to be altered.

(b)

(i)

Multiple threads. Each thread can call receive and the first thread to receive a message can carry on. It will either have to terminate the other waiting thread or set a flag it can check so that it ignores the message it receives. Care will have to be taken to avoid race conditions.

(ii)

Two processes are used to catch messages, one from mailbox A and one from mailbox B. They both write the messages to a third mailbox which the original process waits to receive from. The second message is ignored.

(c)

The receiving process doesn’t provide any name for the process (or mailbox). It just indicates to the message passing system that it is waiting for any message. Which is exactly the behaviour we want in this case.