Computer Science 340

Operating Systems

SECOND TEST : ANSWERS AND SOME OTHER BITS

Answers look like this;
comments (added after the test) look like this.

QUESTION 1.

Assignment 2 strikes again. The specimen answers are those I wrote down before the test; the answers you gave weren't always the same, but some of them were Macintosh equivalents of my answers, which I had to accept - sometimes. I've tried to distinguish between differences of interpretation, and differences of principle.

For example, the Macintosh interprets opening a file as an operation you perform on an existing file, as is clear both from its file manager procedures and from the choice it usually offers in dialogue between "Open" to open an existing file, and "New" to make a new one. The alternative is that "Open X" should return you an open X if it's at all possible - which includes making a new one if necessary. (Think of the Pascal rewrite function.) That's a difference of interpretation, and I accepted either view.

On the other hand, any interpretation of "open" which excludes multiple simultaneous opening, for reading at least, is unrealistic for a general-purpose operating system. That's a difference in principle. (Someone explicitly mentioned the possibility of exclusive opening, which is fair enough.)

(a)

- Find the file attributes, check that access is permitted.
- Construct the file information block.
- Allocate buffer space.
- Report the operation complete.

Very few people mentioned anything to do with the file information block (terminology varies; I accepted anything plausible). After assignment 2, I'd have thought that ParamBlockRecs would be forever engraved on your memories.

Quite a few people think that part of the open operation is to load a file into memory. If they mean that the first block (or something of the sort) is loaded, it makes some sense, though unless you know that the file will be used sequentially the effort might be wasted, but usually they seemed to mean the complete file. This seems highly unlikely; apart from the obvious question of having enough memory, unless you know before you start that you'll want all of the file, it's a waste of time. Part of the misapprehension may be a consequence of your lack of contact with any but toy files; files measured in megabytes are common, and tens or hundreds of megabytes not out of the way for large systems. Add to that many programmes' need to use several files simultaneously, and the constraints of operating within a shared system, and it should be clear that holding complete files in primary memory as a matter of course isn't a viable general policy. Disc caches may approach this condition by chance, but can't guarantee it.

A disadvantage someone suggested was that the existence of files which were not represented in a directory was counterintuitive, and violated the system illusion. That's an excellent point, which I hadn't recognised before. Marking tests isn't all bad !

(b)

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

A file can be opened even if it is already open - but some care is necessary. There's no difficulty in having a file open by many programmes at once so long as they're all reading the file. There are difficulties when one or more programmes want to write to a file opened by other programmes, but even these can be managed subject to certain constraints.

(c)

It is **possible** to use such a file because all operations on it are performed in terms of the file information block, not the file directory entry.
An **advantage** is that it is possible to use temporary files without the overhead of searching file directories.
A **disadvantage** is that if the programme or system fails without closing the file, then all information is likely to be lost.

The point of the question is that the operations on the file are administered using the information in the programme's file information block; the directory is not involved at all, except in opening and closing the file. The directory isn't involved in allocating disc space, either; that's a matter for direct transactions between the disc writing software and the disc space manager.

Several people devised ingenious stratagems to search for a file which wasn't in a directory - inspect memory, special hardware devices, search remote machines. Generally, if you can't find a disc file in some sort of directory, then you can't find it. The first sentence was intended to tell you how you got the file; the question is about how you can use it. A lot of people seem not to have read the first sentence at all.

*It isn't a disadvantage that you might not be able to find the file again; it is **certain** that you won't be able to find the file again. Where could the information be kept ?*

The file would normally be kept on the disc. (In fact, whether or not it's on the disc is irrelevant : the question was about directories.) Many people seemed to expect it to live in primary (or virtual) memory, which will presumably grow out of the side of your computer as and when required. Why in the world should anyone provide lots of special additional software for special cases when the existing stuff will do the job perfectly well ? ("Ram disc" doesn't count - it's rigged up to look just like a disc, with directories and all. See device independence.) There were occasional hints that people might have been thinking of the "log-structured file system" - but that's still a copy of the original disc system, with directories as usual.

(i)

When the file is closed, the system must :

Determine the external name of the file;
Search for the directory;
Enter the new file name, and any required attributes, into the directory;
Link the directory entry to the file by putting its address in the directory.

(ii)

If the file is not closed at the end of the programme, the system must :

Return to the system the disc area allocated to the file.

Despite the explicit statement to the contrary in the first sentence of part (c), a number of people gave an answer amounting to "behave as in (i), finding an external name from somewhere".

Your experience probably suggests that it's unnecessary to take special action to close a file, but I'm far from sure that it's the operating system, as opposed to the compiler, that brings this about. From memory, I think that both C and Pascal undertake to close any files that are left open when the programme ends. If you want to try some experiments, write a little programme which makes a file and does something with it, then somehow stop the programme before it ends. Then look for the file. The only language within my reach at the moment is Zortech C on MS-DOS; with that, the file vanishes. Be careful with the Macintosh - a file can end up not in use but open, and can then be something of an embarrassment.

What OUGHT to happen ? That depends on what you think is important. If disc space is in short supply, you don't want to use any unless you have to, and it's reasonable to insist that anyone requiring a permanent file should ask for it explicitly. On the other hand, you could argue that as the memory space available to a programme becomes bigger and bigger, there's very litle need to make temporary files anyway, so it's reasonable to assume that all files are intended to be permanent. You could also take into account the effect on the system illusion, mentioned earlier.

Some suggested that the system should ask whether or not the file should be saved. That only works if there's someone there to ask. Even if there is, the last thing I want when I've finished doing something with a programme is a lot of silly questions to answer : OF COURSE I want to save the file at the end of an editing session (all (?) Macintosh word processing things); OF COURSE I want to print the stupid file in a perfectly standard format (after selecting PRINT). If I didn't, I'd have said so. A good interface will do obvious things automatically, but make it easy to change the specification if you need to.

QUESTION 2.

a) Nothing can take the processor away from a process which is currently running in kernel mode (interrupts run in the context of the process). It will continue to run until it leaves kernel mode or puts itself to sleep.

b) The interrupt is handled whilst in kernel mode.
The interrupt conveys information that some other process should now run (i.e. the currently running process does not have the best priority).
This causes a flag to be set.
The interrupt handler finishes and returns to the process running in kernel mode.
The flag is tested as the running process is about to return to user mode.
If it is set the dispatcher is called to restart another process.

c) We can use a simple lock because the kernel cannot be preempted.

```
lock:
      while lock_x do
            sleep waiting for lock_x;
      lock_x := true;
```

Mutual exclusion is guaranteed by the kernel.

QUESTION 3.

a) No busy wait.

b) The process still runs, therefore it takes up processor time and memory.
No guarantee that a waiting process will get the resource, indefinite postponement.
No record of processes waiting for the resource.

c) The process waits in a queue. Only running when the resource is available.
The system makes sure that the most worthy process gets the resource next.
The queue is a complete record of processes waiting for the resource.

d) Holding one resource whilst waiting on another can lead to deadlock. If the conditional wait returns without the resource we could release the first resource and try for both of them again.