Computer Science 340

Operating Systems

FIRST TEST, 1990 : SKETCH ANSWERS

Remarks in italic type were added after the test.

And here are a few to try :

Thanks to John Thornley for remarks on my "NOTE ON TERMINOLOGY" paragraph. John kindly sent me a copy of the entry in Chambers's dictionary pertaining to "application", which is immediately followed by another entry for "application program" (sic). It is very obvious that none of the meanings of "application" has anything to do with "application program". Now, I don't think that's the point John was hoping to make, but I thought it was good. (The Shorter Oxford dictionary, on my side for once, doesn't even mention anything to do with programming under "application".)

I remarked that you can't change the Macintosh system because it's in ROM; I should have added that you can specify alternative routines to use <u>instead of</u> the ROM versions if you want. (See "Inside Macintosh", volume 1, page 87 for further details if you're interested.)

I correct spelling and the like when I see mistakes (until I get tired). There are no marks for good spelling and grammar, but if no one ever tells you about your mistakes, you don't get a chance to improve your performance.

It is probably not a good idea to use the word "hack" or its derivatives in your answers; it is far too imprecise.

QUESTION 1.

This question didn't work out as I'd intended, largely because very few people read part (a). Most people mentioned a few relevant points; rather few presented them as anything recognisable as a list; and hardly anybody gave any sort of reasons why the points they'd mentioned led to greater secretiveness. As people hadn't advanced any arguments in part (a), they started part (b) at something of a disadvantage. I usually managed to fish up some marks, but a lot of them should have been thrown back in. (It took me about 13 hours - which will perhaps teach me to be more careful in future.) The arguments I'd expected were of the form :

<desirable feature> & <system constraint> → <restricted access>

The <system constraint> had to be something pertinent to an operating system which didn't apply to a monitor system - so would probably be something to do with people sharing a system. (I didn't particularly want the arguments presented so formally, but the parts should have been there somewhere.) For part (b), I wanted discussion of the parallel argument :

<desirable feature> & <changed system constraint> → ?

For example, in part (a) :

<safe disc files> & <all files on same disc> → <limited access to disc>

but, in part (b) :

<safe disc files> & <people have own floppy discs> → <free access to disc>.

(That's not complete : if you want to talk about distributed systems, some sort of limitation is still necessary. It's an example, not an answer.)

(a)

The desire for greater efficiency has led to a great emphasis on using all resources to the maximum, which in turn has caused system designers to work towards running many processes at once in a processor, so that it will almost always be possible to find some active process which can make use of a resource which would otherwise be idle. Clearly, each process must remain unaffected by the presence of the others; and the extra "secrecy" is introduced as a means of providing protection from interference.

1. People share the system, and must be protected from each other.
   Any person permitted to use the computer completely freely could interfere with other people's code and data.

2. The system must be protected from people.
   People permitted free access to the system's facilities could halt or corrupt the system.

3. Peripherals must be shared between many people.
   If people were allowed to drive them directly, there would be no easy way of ensuring that one person's - say - print run were not interspersed with others' output.

4. Private files must be protected.
   Even when people are not using the system, their data remain there; and free access to the disc would permit these to be stolen or spoilt.

5. Accounting data must be protected.
   People able to use all parts of the system could change their accounting entries, and thereby acquire unfair access to system resources.

   etc., etc., ……

I asked you to "explain why" : spell it out. It is true that the operating systems demand "total control of the environment" - but so, in a sense, did the monitor system. Why does that necessarily imply a reduction in access to things ?

I asked you to "list some reasons ..." and "show how ..." : quite a lot of people didn't "show how ...".

A few people mentioned protection from viruses and such things. These didn't exist when operating systems were first developed; in any case, as operating systems have so far not evolved any effective defence mechanisms against them, they can hardly be part of a reason for increased secretiveness.

The switch from batch to interactive working wasn't an important factor. That had a strong effect on things like scheduling policy and resource provision, but rather little on issues of broad system principles.

Quite a number of answers included "reasons" for restricting access which applied just as well to monitor systems. I did ask for reasons why operating systems were ***more*** *secretive.*

Several people were, very commendably, concerned to help programmers who were worried about having to code low-level details of input-output routines. Unfortunately, that isn't the point : procedure libraries for practically everything have been around for a very long time, and no one using a monitor system ever needed to worry about such matters. The problem is the people who insist on fiddling about with the system, because they think that they can do better. Sometimes they're quite correct - but by no means always.

The issue of standards was raised in a few answers, usually with the idea that if you can fiddle about with the system to any extent then standards would be threatened, leading to a proliferation of different and incompatible systems.

Some people took a sense of "secretive" which I hadn't intended, and wrote essays on security systems; "restrictive" might have been a better word. (I wouldn't have called security systems "secretive" either - perhaps "defensive".) (It was perhaps unfortunate that the test immediately followed the section of the course on protection and security; I'd set the question before, some years ago, without encountering the misunderstandings.) I think the meaning is clear enough in the context of the preceding sentence - and you have to take the context into account, if only to complete the meaning of "more secretive". (Than what ?) Despite their having nothing whatever to do with the first part of the question, I'd have accepted the security essays, but they didn't address the "more secretive" question either.

(On a more positive note, I remark that it was most refreshing to detect your attitudes of horror and reproof towards those who would dare to try to steal other people's data, and otherwise misuse the system. Perhaps there's hope for the computer industry after all.)

The difference is a matter of scale. Security is usually thought of as operating at the level of file access and programme execution, roughly corresponding to system instructions; the question is about individual system procedures. You can be as careful as you like about file access codes and that sort of thing, but it's not much use if people can get at the system routines which read and write files and bend them to leave out the security checks.

A view which turned up in various guises from time to time was that people should be kept out of the operating system because they don't need to be in. It's an argument which I find familiar, because I use it myself : as about 99% of the population of New Zealand don't need cars, they shouldn't be allowed to have them. My argument is not usually taken seriously - so why should yours be ? The point is that people may want to do things they don't need to do, and require good reasons why the things in question are actively dangerous before accepting restrictions.

If you want to impose restrictions, you have to justify them; and, if you can't justify them, you've no business to impose them.

In practice, it appears that you have to find a stronger justification than that widespread car ownership costs the country many human lives every year, as well as fantastic sums in road maintenance and importing cars, fuel, and so on.

(b)

1. People share the system, and must be protected from each other.
   The degree of sharing can be made very much less if each person has a separate computer. Sharing cannot be completely abolished, as people must exchange data to work together, but the focus of the problems moves from sharing processors and memory to the communications system.

2. The system must be protected from people.
   Still true; but a person will only be able to affect a limited part of the system. It becomes important to find ways of localising the effects of a breakdown.

3. Peripherals must be shared between many people.
   Not necessarily true, except for very expensive machinery; but people are still going to want some sort of system software to drive the equipment easily and effectively without their having to know all about it. Security is easier to implement; the need for convenience remains unchanged (and may even be a more pressing requirement than in the past, as people's expectations become greater).

4. Private files must be protected.
   Less of a problem when people can take their multi-megabyte files home on a floppy disc; though there is still a need for central archiving (if only to ensure that not all the firm's data are at people's homes on floppy discs) and that still needs protecting. But it's a more restricted environment, and you can enforce stricter rules.

5. Accounting data must be protected.
   What accounting data ? It's very hard to maintain an accounting system at all on, say, a system composed of many microcomputers linked by communications lines. In any case, the cost of computing is such that accounting in the traditional sense is hardly worth while.

I had expected people to discuss a set of communicating microcomputers ("many smaller machines, which may communicate through a network"). If you weren't going to talk *about that, you should have said what you were going to talk about, and at least suggested why you thought that was more appropriate than my suggestion.*

My quick answer to part (b) is "no". If the basic reason for secrecy is that many people sharing one machine is dangerous, then arguments for secrecy will break down once people start to have their own unshared machines. That doesn't mean that the services to be provided by the system have changed; it may mean that we provide them in different ways.

I wasn't expecting answers in terms of distributed operating systems. I got some.

I certainly wasn't expecting answers about timesharing systems with lots of terminals, but I got a lot of them. I didn't think I'd have to spell it out; after all, you've been using a network of linked microcomputers for two years, so most of you, at least, should be much more familiar with that than with timesharing systems.

An excellent point : with the development of more graphical WIMPs interfaces, there's a sense in which the operating system is becoming more secretive than ever !

If you have a network of essentially independent microcomputers, it is very close to impossible to enforce any sort of behaviour anywhere, so the best you can hope for is helpful protective measures.

QUESTION 2.

THERE IS ONLY ONE "E" IN THE WORD "ARGUMENT". Several people even managed to get another one in when supposedly copying from the transcript in the test paper.

*TERMINOLOGY : the **shell** is the system programme which interprets instructions and runs programmes to perform tasks as required; a file containing instructions to be interpreted by the shell is a **script**. Some people seem to have the words mixed.*

(a) plus determines whether its single argument is or is not a "+".

The "+" must immediately (apart from intervening spaces) follow the "`plus`". I accepted mentions of "the argument" or "first argument".

I interpreted "no more than one line" fairly liberally - I used my own handwriting as a gauge. This may be the first time ever that anyone has gained any benefit from my handwriting.

(b)

: This instruction does nothing. It is present only to ensure that the first non-blank character in the file isn't #, which would cause the script to be executed by the C shell rather than the Bourne shell.

This absurd situation is a consequence of the strange decision to make the (totally obscure) C shell selection instruction identical with the character which introduces a comment.

CRITIQUE : The provision of comments is good; the ability to choose the interpreting shell is good; the mechanism with which the choice is communicated is ridiculous.

`if` A conditional instruction which branches according to the value of the expression which follows it. In this case, the expression is the name of a programme (`test`), and the value is the value returned by the programme.

The first argument to test is given as `$1`; the shell replaces this with the script's first argument before entering test. In this case, test will normally have three arguments - the value substituted for `$1`, "`=`", and "+".

CRITIQUE : The provision of a conditional instruction is good; it is a little inconvenient, but not necessarily bad, to use a separate programme to evaluate expressions rather than building an expression interpreter into the shell (as in the C shell); it is unsatisfactory to be forced to separate terms in expressions with spaces after test when apparently analogous constructions used elsewhere in the system must not include spaces. (For example, in assignments.)

`then` ... `else` ... `fi`   Satisfactory.

`echo` Fair enough, but a clearer identification of the following string would be better. If the \ before the ' is omitted, you get a very odd error message - "unexpected end of file".

I asked you to "explain HOW it works". That implies some sort of attempt to work out what's going on when the shell script is executed. Instead, I got a lot of rather extended answers to (a), which usually added little enough to the (a) answers I'd already read. I expected at least some comment on each of the three instructions in the script (`:`*,* `if`*, and* `echo` *), and - in the spirit of "how it works" - some comment about things like* `test` *being a programme which returns a value, and parameter substitution. It's true that the explanation has to stop at some level, but as the question is about the shell, it's reasonable to expect an answer in terms of what the shell does - which is to interpret the shell script and execute programmes accordingly. I didn't expect any comment on the internal organisation of* `test` *and* `echo`*. You may be interested to learn that I deliberately reduced my original plus to the banal remnant which appeared in the test so that you wouldn't be tempted to spend too much time on this part of the question.*

Only one or two people commented on the `:` *instruction.*

I'd expected (and intended) criticism of the shell language and facilities; I was quite surprised to get criticism of `plus` *itself. I suppose that's connected with the difference in interpretation of "how it works". The criticism of* `plus` *was, of course, a perfectly satisfactory response in terms of the question, and I accepted it as such.*

The features of `plus` *are the things which are there : the absence of things which are not there is not to the point. I therefore gave little credit for complaints that the user interface was bad, or that there was no help provided. As the programme was obviously not meant to be used except as an example for purposes of this test, they weren't really valid criticisms anyway, however good the principles they embodied.*

I'm surprised how many people didn't take up the invitation to criticise; presumably they all find both the `plus` *script and the shell neither desirable nor undesirable.*

The file `test` *has to be executable by the shell, or the shell will ignore it when searching for a test to execute.*

The shell "variables" (which includes the parameters) aren't really variables at all : they are macros, expanded by substitution before the instructions are executed.

Not many people seemed to be thinking in terms of `test` *as an independent programme returning a value to the shell. Notice in particular that the shell does not perform any tests on its own account : the* `if` *instruction must be given something equivalent to a logical value. Likewise, the shell doesn't display anything : it executes the programme* `echo`*.*

Some other interesting features mentioned are listed below. Not all got a lot of marks, but they may repay careful thought.

Good features :

- `if` *..* `fi` *construct is good.*
- *The script works. (Actually an unwarranted conclusion unless you know what it was supposed to do.)*
- *The ability to execute programmes and find out what they did,*

Bad features :

- *It's useless. (Not true : it was used to set a test question, and therefore perfectly fulfilled its purpose.)*
- *The messages aren't very helpful. (Irrelevant : see previous point.)*
- *It doesn't check that there's no local file called `test`.*
- *The + sign isn't in quotation marks. (Why should it be ? Why is it different from any other token in the programme ?)*
- *The script doesn't set an explicit "search path" (that's a list of directories which should be inspected when any file is sought). By making sure the local directory wasn't in the search path, the troubles with test could be avoided.*

(c)

The key is the **rm** instruction about half way down the transcript : this removes a file called `test`.

plus # 1, # 2, # 3 :	In each case, the local `test` is used. Whatever it does, it tries to execute something called `a`, which doesn't exist, and always returns a false result. (In fact, `test` is a "shell script" containing the single line "a word".)
plus # 4 :	Now there is no local executable file called `test`, so the system `test` is used. As there is no argument for `plus`, the shell replaces `$1` by nothing at all, and `test` cannot find a sensible expression to evaluate.
plus # 5, # 6 :	`plus` works as described in (b).

Some of you seem to have a rather curious "system genie" which allows a file called `test` to interfere with the the shell's execution of a supposed built-in test function. Yes, of course you could write a shell to work that way if you wanted to, but it would be an odd thing to want.

There was some suggestion that the "missing argument" in the first and fourth uses of `plus` was somehow wrong. That depends on your point of view. It's certainly not what was intended when `plus` was written, but so far as the shell is concerned it's perfectly legal. The argument "variables" are just like any other, except that they might be given values in a couple of rather unusual ways (from arguments, or by read).

Almost everybody recognised that there was an extra file called `test` in the local directory. Not very many explained how this file produced the observed effects. Several answers included phrases like "if the test fails", which didn't seem to me to be an adequate substitute for "as the script `test` returns a false value after execution".

The explanation should include some interpretation of the error messages displayed.

The message "`test: a: not found`" doesn't mean that `test` can't be found; it means that `test` can't find `a`.

One person interpreted the message "`test: a: not found`" as meaning that `test` was unable to find a colon. That's a perfectly plausible interpretation of the message; unfortunately, it's wrong. It illustrates how easy it is to generate misleading error messages. (NOTE : I'm not getting at the person who made the mistake - not many people even tried.)

Another, who clearly regards me as a very devious character indeed, asserted that I had given you the transcript in reverse order ! The theory advanced was that the first three tries at `plus` failed because there was no `test`, so must have been executed after the rm `test`. I can only remark that I may set slightly curly questions, but I don't set dishonest ones : if I say something's a transcript, then it's a transcript. (It was, in fact, cut-and-pasted from a Unix window into a MacWrite window.)

A number of people were uncertain as to just what had failed in the first three invocations of `plus`*. In fact, nothing had failed : the script* `test` *was executed, and returned the value false.*

QUESTION 3.

"PRIVILEGE" IS SPELT "PRIVILEGE".

NOTE : I have a nasty feeling that I may occasionally have applied my reflex correction of "...edge" to "...ege" in cases where it wasn't valid - such as "knowledge". If I've done that to you, I apologise.

A PASSWORD system requires either (preferably both)

(i) that the operating system, at least, be able to maintain private files; or (preferably and)

(ii) that a standard encryption procedure be available to those parts of the operating system which need to use it, but not to people in general.

A private file system is impossible to add on to a system which doesn't provide it, by software changes alone. If you can by any means write an assembly language programme which will read from any selected part of the file system, then there is no way to prevent the programme from working, short of hardware changes.

It is easy to provide an encryption procedure; it is harder to control access to it. The critical facility needed is memory protection; the procedure must be in memory to be executed, but it must be invisible to any but those parts of the system which need it. If it is possible for a programme to branch to any arbitrarily selected address in memory, this protection cannot be guaranteed.

A CAPABILITY system requires

(i) a source of capabilities; and

(ii) means of handling them safely - in particular, it must be possible to transfer capabilities, and to change them, subject to constraints.

Given file and memory protection as described for the password system, it is reasonably straightforward to add capability-based security of the system-managed "privilege" variety. Provided that all operations on capabilities are managed by the operating system and are inaccessible to the public, except through controlled supervisor calls or some similar route, then the system is secure.

The usual problem - a lot of people didn't read the question. I didn't want dissertations on access control lists, and I didn't want to know any details of how you proposed to implement the protection and security system.

I should perhaps have been a bit more precise in wording the question, but I thought that the stress on "facilities which must be provided" in the assignment would have shown you what I meant. It's a matter of top-down design. Imagine yourself as providing security services to other parts of the system, and try to determine what special things you must be given by the existing system in order to build the new part on top of it. The result is a list of things which must be implemented at the next level down. The question is therefore "what we can't construct for ourselves, but can't manage without" rather than "what would be quite nice to have". There's also an implied bottom level; I didn't give marks for things which aren't special to protection and security, like files.

I'd hoped that adding a reference to "abstractions" might help; I'm not sure that it did. It's used in the context of information hiding - anything which you can use but where the details are looked after by someone else can be thought of as an abstraction. It includes data structures, procedures, services, ...

It would have been easier for me if the answers had been presented in an order which was in some way related to the question. It would probably have been more profitable for you, too, as I may have failed to unearth significant information from the rubble. I understand that the literary fashion for stream-of-consciousness writing has passed.

LIST SOME FACILITIES :

> *A lot of answers were hints as to how I should go about implementing the system. I was sometimes able to retrieve references to encryption, or to the use of files which weren't open to public view, and I sometimes felt justified in interpreting these as evidence of understanding that encryption and secure files were things which should be provided by the system.*
>
> *Some of the suggestions were things which the new protection and security system should be implementing, not things that must be available before it can start. For example, a list of people permitted to use the system is something that can be added - but there must be a guaranteed way of making a private file if it's to be any use.*
>
> *Encrypted passwords are useless unless there's some way to keep people out of other people's property.*
>
> *A lot of people used terms like identification and file access privileges, often suggesting that they should be in some sense "provided" by the system. As identification and the definition of access privileges are the function of passwords and capabilities, these answers were often circular.*
>
> *Some suggestions not included in my specimen answer :*
>
> > *The ability to store lots of useful information in the file system.*
> >
> > *A way of adding instructions to the user interface, and of receiving information therefrom.*
> >
> > *A means of turning off the screen echo, so passwords entered at the keyboard don't appear on the screen.*
> >
> > *Education for the people who use the system - excellent in principle, but unfortunately not responsive to the question.*
> >
> > *Memory management : yes, we need it, but it hasn't anything directly to do with passwords.*

ADDITIONAL FACILITIES :

> *You do not need an access matrix to implement capabilities.*

Alan Creak,
May 1990.