Computer Science 340

Operating Systems

TEST, 1995 : Answers

These are not the only possible answers, nor necessarily the best ones. They are certainly not the shortest ones; we have preferred to present fuller explanations rather than real answers. In most (maybe all, but we're playing safe) cases, you can get full marks with far less material than the answers given here.

Material in Helvetica type was added after marking the test.

QUESTION 1.

(a) **Necessity :** The user database is necessary to provide the information needed to separate the different users' activities and property. A user logging in to the system must be identified and given access to only such parts of the system as are appropriate – and must not have access to any other user's "property". The database is also necessary to provide some sort of continuity between sessions for people using the system; without that, it would be impossible to keep track of people's "property" in the system.

The security function of the userdata is important, but people who just said that didn't get all the marks. I wanted either two sorts of security (such as access and identification of the person's home directory in the file system), or security and something else (such as continuity in accounts).

Security : The user database itself contains a variety of information, much of which must not be directly accessible to any ordinary user, or unauthorised use of the system might be possible.

- Information not relevant to the operation of the computer system itself (names, addresses) may be open to reading and writing by the user concerned; some may be more widely available for reading (consider **finger** in Unix systems), though current obsessions with privacy may limit such access.

- Information of concern to users but required by the system (account balances, privileges) may be readable but not writable. (Passwords, in contrast, are likely to be writable but not readable.)

- Information of concern only to the system (disc addresses) may be inaccessible to the user, though as a matter of principle unnecessary concealment is better avoided.

The "discussion" of security requirements was commonly very brief indeed.

In a large proportion of the answers, there was no indication that different parts of the userdata had different security requirements.

It was possible to interpret the "security requirements" part of the question either as "what are the security considerations which must be taken into account in controlling access to the user database ?" (which I intended) or "what security information should be held in the user database ?" (which I didn't). I still think that my interpretation is the more obvious, but I gave marks for either.

Many answers included statements that no access to the userdata should be provided except through system calls. That's right as far as it goes, but the same goes for any file in the system. I wanted more information about what sort of access should be permitted.

(b) There are many possible answers to this part; see the course notes (*Requirements specification*, pp 40-41). Here's a possible answer :

Database items :

Home directory	Required to set the initial value of the working directory on logging in.
Current credit	Required to keep track of expenditure from session to session.

Active session items :

Working directory	Used to locate a file whenever a file name relative to the working directory must be resolved.
Terminal identity	Used to direct messages relevant to the job or user to the appropriate terminal.

The significant distinction is that between information stored between sessions, and information stored only within sessions.

There was a strong preference for username and password as components of the userdata. In hindsight, that's hardly surprising; if I'd thought about them beforehand, I'd probably have excluded them ! (- because they're rather too obvious). But I didn't, so you got the marks.

A few rather surprising things were discussed as components of the permanent or temporary userdata; examples were records of mail transactions, what processes were running, what files were in use, etc. While it's always impossible to assert that such information isn't in any userdata collection on any system anywhere, some things are more probable than others. Usually, if some system component is managed by a special part of the system, the userdata would not duplicate the information, though it may contain a link of some sort to some other part of the system. For example, the userdata may include a flag indicating that there is mail, and possibly a link to a mail file, but more specific information is more likely to be held by the mail system. Similarly, information on processes and files is more likely to be restricted to process and file tables; misunderstandings about processes are to some extent fair enough, because we haven't yet covered them in the course, but you should know about files. Lists of things are awkward, because they are variable in size, so explicit lists of processes or files in use, mentioned by several people, are unlikely; if you need to know, it's easy to search the file table or process table. (But the current directory has no other natural home, so is legitimately part of the temporary userdata.)

A few people found a loophole which I hadn't noticed in the question : they described items of information about users which were not permanently recorded, as requested – but which weren't about individuals either, which wasn't what I had in mind. (For example, a list of active users.) That was a fair cop, and I gave marks accordingly.

(c)

Attribute	Userdata	Preference files	Explanation
SIMILARITIES			
Persistence	Once set, preserved until reset.		The stored material sets parameters specific to the individual without the need for redefinition every time the item concerned is started.
DIFFERENCES			
Security	Maintained by the system.	None	The userdata file is used in a shared system, where other people may interfere with your property; preference files are used in personal systems, which are assumed to be used by individuals.
Linked to :	Operating system	Programmes	They do different things. The userdata must be available to the system before you log in, as it is needed for the identification; the preferences files are only needed when the programmes are started. (Note that personal systems can maintain system data – control panels, CONFIG.SYS and AUTOEXEC.BAT – while shared systems provide for personal configuration files – .login, .cshrc, .plan, etc.)

I was rather overconfident in expecting that everyone would know what I was talking about. While no one admitted it, it was clear that some people didn't know what preferences files were, and hadn't really used any user database facilities. I therefore accepted many plausible guesses which were not necessarily good ones.

There is a grey area around facilities provided in a shared system for specifying various system parameters which you wish to be set for your sessions – such as .login and .cshrc files in Unix, which live in your directory but are executed by the system. Because there is explicit provision in the system for using information from these files while starting the session, I'd classify them as part of the userdata which was held in your directory because it was convenient and satisfied the security requirements. I accepted answers from people who took the view that these were preference files, and gave any justification.

I should have been a little more careful in phrasing the question, because it allows answers with very little substance – "information to be stored by both methods would be basically the same", for example. I gave some marks (up to half) for answers of that sort, but required something rather more significant to the operating system for full marks.

These comments add up to evidence that this wasn't a satisfactory part of the question. That's quite right, and I apologise, but it's rather late to do anything about it. Many answers didn't fit my expectations at all, so I tried to give marks corresponding to my impression of the appropriateness and completeness of the answer. If you said reasonably sensible things about two features of the files, you got most of the marks.

QUESTION 2.

Throughout this question it was clear that many people haven't worked out what is done by the operating system and what is done by the programme. I find that rather alarming. Many people had operating systems changing the contents of windows or files. This is impractible, even if it were desirable; you'd have to add extensions to the operating system every time you acquired a new programme. It's the programme's business to decide what's in the file or on the screen, then, once decided, it's the operating system's business to get it there. Similarly, while we want consistency in the interpretation of input to a programme, only the programme knows how to do it, so the operating system is restricted to passing on low level information – "That was a double-click" rather than "That was a select operation". While there really is no hard-and-fast line,

the principle is that the operating system should provide general facilities, while the programmes look after specific requirements. One would not expect that the operating system would have to worry about the details of copying and pasting (for most systems, it's impossible, because they know nothing of the structure within a window); but it should certainly be responsible for transporting the data from programme to programme. Consider : if you were writing ClarisWorks (replace with the name of any other programme running in a GUI system if that offends you), would you expect the operating system to format your windows for you ?

If you're asked for examples of good design, and explicitly told what to do if you can't think of any, it isn't good policy to give examples of bad design.

Not everyone got the facts right in their examples; I didn't worry much about that, unless the "facts" were obviously silly.

(a) **Consistency** is important because it reduces the size of the system genie – the body of knowledge needed to use the system effectively. If a principle applies consistently throughout the system, one need not learn different conventions for different parts of the system.

A good example of consistent design is the uniform design of windows in the Macintosh system. Almost all conform to the standard pattern, and the various components of the window almost always work in the same way, so that in almost all circumstances one can move the window, move the material in the window, change the size of the window, and close the window using standard methods.

Consistency was often interpreted rather restrictively, sometimes to the extent of associating it with the fact that "vi X" will always edit X (which isn't really true in any useful sense : try it using a directory name for X). While it's certainly consistent, that isn't what is commonly meant by consistency in the context of GUI. (It's more a matter of functionality than consistency : you'd normally expect to speak of consistency between two potentially different things.) For example, that double-clicking a file icon always opens the file isn't an example of consistency – it just shows that the file-open operation is encoded as a double-click. You get an example of consistency if double-clicking anything at all does something quite like opening it.

I was interested in an assertion that Unix had achieved its "power and popularity" "just because [of] the consistency of" its convention that "everything is treated as a file, a directory is a file, and [a] print device, [and] a terminal are also files". I remain unsure of the "just because", but there's something in the rest of it, though I'd have preferred the assertion to have been expressed in terms of streams. Whether it's a good example is debatable; it's the sort of consistency you can achieve by refusing to acknowledge any differences – on the level of "all files are character streams", however much structure they may have. A Macintosh or Windows analogy would be a convention which treated text and graphics files identically as character streams, and I don't think that would be regarded as a good consistent user interface.

ANECDOTE : In one answer, the example given for consistency was <Shift-Insert>, which always means "paste" in Windows. No, it doesn't, I thought, it's <Shift-Insert> in Turbo-C++, but usually <Control-V>. But, being comparatively inexperienced in Windows, I tried it. The answer – so far as I could check it – is perfectly correct, but there's nothing visible on the screen to tell you so. I can't find <Shift-Insert> in the manuals, either. Consistency : good; self-explanatoriness: dreadful.

Self-explanatoriness is important because it makes the system easier to learn. If a part of the system is self-explanatory, it offers clues which help one to use it effectively.

A good example of a really self-explanatory operating system is very hard to find. (Self-explanatory programmes are easier to find, but not what we asked for.) The big difficulty is that "self-explanatory" means different things depending on where you begin. A really self-explanatory system would explain itself from the beginning to an "average" person (hermaphrodite, skin patches of all known shades, speaks Esperanto or Ido, etc., and – fortunately – non-existent). GUI systems commonly have standard ways of giving help which can be used to make it easy to start using new programmes and techniques, but they rely on knowing a great deal about the system conventions before you start. Here are two answers, one appropriate for a specialised (speaking and reading English) average person, and one appropriate to someone who has already learned to use a GUI.

A description of a mythical system, suitable for rank beginners :

> The most prominent feature of the keyboard is a large green button clearly labelled HELP. At any time in a session, pressing the button displays, probably as a menu (ideally on a secondary screen, so that your current position is not obscured), a list of the actions available to you and an explanation of what they are and how to do them.
>
> (That isn't a joke. If you know nothing about computer systems, that's about the only way that a system can tell you what you can do.)
>
> The system is good because help is always available, and does not interfere with what you are doing. There are no constraints on the information presented, so the help can be as detailed as required. The special button is independent of any other actions, so is never unsafe; the help is not forced upon you, so as you become more expert you can stop using it. The button is green because that conventionally means that it's safe – people are (or should be) wary of pressing red buttons.

A description of a less mythical system, suitable for those with some experience :

> The implementation of windows in the Macintosh system is an example of good design.
>
> The standard window provides features with which it can be identified, moved, resized and reshaped, closed, and made active. Where necessary, scroll bars can be included to move the material of interest under the window.
>
> This is a good example because, once you know the conventions, they work reliably. Each of the features has a simple and clearly defined function, is readily recognisable, and almost always works. (The exceptions are usually due to the software developers, not the system.) Most important, they work in a direct and obvious way; to resize a window, you move its bottom right corner to where you want it to be, and so on. (The scroll bars are not quite so obvious : moving the position marker moves the text in the opposite direction.) If a feature is omitted (usually scroll bars), it is clear that the corresponding operation isn't available. It is also a good example because almost all windows conform to the specification; this is a consequence of good(ish) software support from the manufacturers.

Someone suggested touch-screen systems as a good example of self-explanatoriness. I agree : most of those I've seen are in fact very good examples – they are very explicit in their instructions. On the other hand, I don't think I've seen one used as an operating system interface.

A common example of self-explanatoriness was the "Trash" icon on the Macintosh. (I think the "Delete" key on the PC is even more self-explanatory ! – but that's by the way.) Someone very properly pointed out that the self-explanatoriness (can anyone suggest a better name for that property ?) didn't extend to emptying the trash. That didn't matter on the old systems, where the trash was automatically cleared quite regularly (which is a reasonable continuation of the metaphor for city-dwellers), but on later versions it isn't automatic.

Another (less common – only once, so far as I remember) suggestion was the "filing cabinet" icon used for the Windows file manager, and the answer included a remark that it might be less easily understood by those unaccustomed to an office environment. That's a significant point, and it applies to some degree to the Macintosh interface too. (It was inherited from the Star through the Lisa; the office emphasis is diluted, but the desk is still there.) It is something of a surprise that this office (or, at least, clerical) metaphor is so readily accepted by people who don't have that sort of background.

A common type of answer for an example of a well designed self-explanatory feature was "... the logout command in Unix which does what it implies ...". Which, of course, is perfectly true. It wasn't what I had in mind (see my specimen), but it fits the question and gets the marks.

(b) There are very many possible answers at the detailed level, but the main issues are concerned with the establishment of communication between the processes, and with the display of the pasted material : if you copy graphics material into a text file, who is going to display it ? I'll give full marks for evidence of reasonable understanding of either of these issues and its implications. Three possible answers follow.

I got a little tired of reading that "a picture is worth a thousand words" (usually mildly misquoted). I know it's in the notes, but in a qualified form (*Requirements Specifications*, page 21). Even if it were true, it wouldn't necessarily be useful unless you could guarantee that the words were relevant. Have you ever read any reviews of art exhibitions ? It would be more interesting if you could prove that "for any set of 1000 or fewer words, it is possible to compose a picture which has the same semantic content and which can be understood by anyone who sees it". Send your proofs to me.

There is nothing in the question to suggest that the two processes are (or are not) running at the same time. I would expect you to be aware from your experience that material once copied on the Macintosh (for example) remains in the clipboard until it's replaced by something else or the system is restarted. (I am not as accustomed to Windows, but an experiment which I have just performed shows that saved text certainly survives the programme from which it was saved, so some similar structure must be available.)

So far as I remember, no one seemed to notice that the question isn't simple, and very few commented on the problems of displaying the copied material. The reason why we urge you to think about what's happening as you use computers is that, if you do so carefully, you will see that there are more things going on than appears on the surface.

Rather few people addressed the question of different sorts of data – some explicitly stated that they assumed all the data were similar, despite the explicit reference in the question to "text and graphical material".

A significant proportion of the answers contained nothing that wasn't already available in the separate programmes; cutting and pasting were mentioned, but nothing to do with movement of material from process to process.

Several answers contained statements of the form "you can't do <x> because it would get lost" – for example, "if you copy text to the desktop it won't be part of a file". (That isn't a direct quotation.) If you don't want it to get lost, then it's up to you to decide what should be done with it – that's part of the question. The Macintosh operating system is not some sacred object which must not be changed for fear of thunderbolts; this part of the question is about designing an extension to the operating system, so do so.

There were some odd almost-inconsistencies. For example, a statement in (i) that the system had to notice the copy instruction and copy data from the display into a special buffer would be accompanied by a statement in (ii) that the system should provide a procedure which the programmer could use to copy data into the special buffer.

First answer. This answer maintains a functional system, in that activities of one process don't affect those of another. It is assumed that this principle should be maintained. It is also assumed that the conversions will ensure that the pasted material either can be displayed (or otherwise handled, depending on the programme) by the destination programme or can safely be ignored, so no provision for keeping track of the originating programme is needed.

(Further post-marking comments apply to all answers, so they're presented at the end. What's that got to do with postmarks ? Not much, but the comments are frank. Non-philatelists (phobatelists ?) who don't understand that bit may ignore it without loss of information.)

(i) **Identify the task.** This is neither copy nor paste – these operations must be implemented by the programmes looking after the windows. The operating system's job is to transport information between source and destination. To do so, it must be able to communicate both the copied information itself (so some global repository must be provided – that's the clipboard in a Macintosh system), and control information to tell the destination programme what's happening. The copied information must include all attributes – shapes, sizes, positions, typefaces, layouts, etc. – needed to reproduce the original.

(ii) **Application programme interface.**

The programmer of the source programme must be able to specify (probably by calling some system procedure) what material is to be exported, what sort of material it is, and how it is to be displayed. This requires that the system designers establish (and document, and publish) a common format which must be observed by all programmers, and – of course – provide the procedures.

The programmer of the destination programme must be able to determine that there is imported material to be used, and from where it is available, and it must be possible to get the material (probably by calling some system procedure) if required. The incoming information must be sufficiently detailed and well structured that the destination programme can determine whether it has the facilities to display it as specified, or whether some details (such as typefaces) can be ignored without damaging the information significantly, or whether there is no chance of displaying the information as given.

(iii) **Step-by-step.**

(Items labelled SP, DP, and OS happen in the Source Process, Destination Process, and Operating System, respectively.)

1 : Some material is selected and copied (SP). The selected material must be exported (SP, OS). (It would be neater to manage the export only when the process is suspended, but that would require a "you are about to be suspended" signal (OS) to the process. We assume that there isn't one.) The export must perform any conversion needed to express the material according to the established conventions (SP), and copy the material to some global store (OS). (If it didn't, the material would be lost if the source process were stopped. That's why a mechanism which defers the export until the paste is required won't work.)

2 : The source process is suspended (OS).

3 : The destination process is resumed (OS).

4 : An imported paste is requested (DP).

5 : The material is retrieved from the global store (OS), converted to local form if necessary (DP), and inserted into the local data (DP) as appropriate.

Second answer. This is Macintosh-like, so far as process functionality is concerned. It doesn't maintain a functional system, in that copy-and-paste operations in different programmes can (and frequently do) interfere with each other because local and global operations use the same store. It is assumed that this interference doesn't matter. As before, it is also assumed that no provision for keeping track of the originating programme is needed.

(i) **Identify the task.** As before.

(ii) **Application programme interface.**

The programmers must have access (probably by calling some system procedure) to a global store for copied data, and must always use it. As before, this requires that the system designers establish (and document, and publish) a common format which must be observed by all programmers, and – of course – provide the procedures.

(iii) **Step-by-step.**

(Items labelled SP, DP, and OS happen in the Source Process, Destination Process, and Operating System, respectively.)

1 : Some material is selected and copied (SP). The selected material is placed in the global store (SP and OS). The export must perform any conversion needed to express the material according to the established conventions (SP).

2 : The source process is suspended (OS).

3 : The destination process is resumed (OS).

4 : A paste is requested (DP).

5 : The material is retrieved from the global store (OS), converted to local form (DP), and inserted into the local data (DP), as appropriate.

Third answer. This concentrates on the display issue. As in the second answer, it is assumed that preserving functional processes isn't important. It is also assumed that all pasted material should be satisfactorily displayed, even if the destination programme doesn't know how to do it. This can be achieved in two ways : either we can insist that all copied material be "compiled" into some low-level form which can express any material and which can be displayed by a standard module of some sort (translate into postscript or bit map), or the destination programme must be able to call on the original source programme – the only entity it knows to be capable of the job – to manage the display. Here we assume the second of these possibilities, which is more complicated, but better in that it is more likely that the copied material can retain its original structure rather than be converted into something like a bit map.

(i) **Identify the task.** As before.

(ii) **Application programme interface.**

The programmers must have access (probably by calling some system procedure) to a global store for copied data, and must always use it. As before, this requires that the system designers establish (and document, and publish) a common format which must be observed by all programmers, and – of course – provide the procedures. To ensure that material can always be displayed, the stored data must include the identity of the source programme so that it can be called upon if necessary to manage the display.

There must be provision in the operating system for the destination programme to call on the original programme to generate an appropriate display of the pasted material. Perhaps the minimum requirement is for the source programme to accept the copied material from the destination programme and return a bit map; the programmer of the source programme must be able to make system calls which will implement this transaction, and the system conventions must include a requirement that programmes be able to handle such requests and a standard protocol for the requests themselves. (It's a minimum requirement, because it only handles the simplest static case. It won't handle animated material, for example.)

(iii) **Step-by-step.**

(Items labelled SP, DP, and OS happen in the Source Process, Destination Process, and Operating System, respectively.)

1 : Some material is selected and copied (SP). The selected material is placed in the global store (OS). The export must perform any conversion needed to express the material according to the established conventions (SP); this includes labelling with the identity of the source programme.

2 : The source process is suspended (OS).

3 : The destination process is resumed (OS).

4 : A paste is requested (DP).

5 : The material is retrieved from the global store (OS), converted to local form, and inserted into the local data, as appropriate (DP).

6 : For display, the destination programme may recognise that it is unable to deal with the display (DP), and call upon the original programme to construct a display of the pasted material (SP – but not the same process) and return to the destination programme (again using some standardised form) for incorporation in the local display (DP).

(i)

Defining a task is not the same as *describing* it. In a significant number of answers, material obviously originally intended as the answer to subpart (i) had been renumbered (iii), where it was often quite a good answer. To define a task is to decide what must be done; once you know that, you can work out an implementation, which in turn determines the course of events. This part of the question was intended to proceed through that sequence.

Several answers included phrases like "the operating system would have to work out which window things are being copied from". That's upside down : it's the process which deals with whatever you've selected on the window, works out what it is, and tells the operating system to accept some chunk of data which is to be transmitted (or, correspondingly, requests whatever has been transmitted). Just think of how you'd write the programme for (say) a word processor system.

(This has nothing to do with the question, but it illustrates the point. The incomprehensibility of graphical displays at the operating systems level is a serious obstacle to people who construct equivalents of GUIs for blind people. The sensible way to build such an equivalent – called a Screen Reader – is to collect the information – not the picture – presented on the screen, and then to present this in a manner more easily understood by blind people. This is where you want the thousand words, not the picture, and there is no way to work out what the words are. Screen readers have to manage by trying to interpret the bitmap of the screen. There are moves to extend the API for graphics packages so that they can export some semantically useful information as well as useless bitmaps.)

A few people commented on various forms of protection checking – in effect, to determine whether it was permissible to copy into the destination window. That's a reasonable factor to consider, and I gave a mark. (Fewer worried about permission to copy from the source window – as you presumably have permission to look at the material, that's probably reasonable too.)

(ii)

Many answers included things which had nothing to do with the problem of transport of information between processes. The system would have to tell the programmes about mouse clicks and positions on the screen even without the transfer operation.

Rather few people talked about the application programme interface. If you were trying to write the code for a programme to run in such a system, isn't that what you'd want ? How else would you get the copied material out of your programme ?

Operating systems are not omniscient, and should not be expected to be. The only entity which can sensibly check whether it can accept pasted material is the destination programme. If the operating system must know, then there must be facilities for the operating system to ask the destination programme. If this were not so, how could you ever introduce a new programme into the system ?

(iii)

Many answers included instructions on how to do it, but didn't say what the operating system was doing. Bearing in mind that 340 is about operating systems, that wasn't enough. That might

in some cases be because people hadn't really found anything for the operating system to do in the earlier parts.

Some people rely on magic. (Most operating systems don't work that way.) It appears from their descriptions that the operating system knows before it leaves the source process that it will be asked to paste something in the destination process. Please think about what you're writing.

There are also miraculous but unspecified entities (fairies, perhaps) which do things for you. "The material from the buffer is inserted into the text ..." : inserted how ? – by what ? (And note the common assumption that it's text, despite the specification of "text and graphical material" in the question.)

(c)

I had rather hoped that people would consider topics like consistency and self-explanatoriness in answering this part of the question as a way of answering the "what should" questions. Most people ignored the "what should" questions instead.

I was surprised that most people just wrote down an answer without any discussion. There is no simple answer; like most properties of user interfaces, you have to balance considerations like consistency and self-explanatoriness to reach a compromise answer. (Even if there were a simple answer and you wrote it down, how would I know it wasn't just a lucky guess ? Compare "should one drive on the right or left side of the road ?". I didn't record the answers, but I guess that the number of people answering part (i) by saying that the icon should appear was of the same order as the number saying that the contents should appear; that proves nothing, but suggests that the answer wasn't trivially obvious.) Simple statements that "<x> should happen", without any justification, were not acceptable; I took them as indications that you couldn't see the problems. Answers of the form "<x> should happen ...." with some explanation or discussion were accepted, though not necessarily with full marks, because they give me some reason to assume that you're not just guessing.

A surprising number of people demonstrated the futility of user interface design by obviously believing that what you see on the front of a Macintosh computer was a representation of the operating system, and *not* a representation of a desktop ! Phrases like "you can't paste text onto the desktop because it's the operating system" (not a direct quotation) were not uncommon. Considering that the operating system is carefully designed to simulate a desktop, that sounds like failure. If you can't think of it as if it were a desktop, it isn't working. (That's a bit unfair, because you're experts, and are likely to have a more advanced system genie than most people; on the other hand, though, being experts, you know that you're supposed to interpret actions on the screen in terms of the desktop metaphor.)

Several people based their answers on the principle that an icon represents an activity. I can't assert that this principle is wrong, but I'm not aware of any common system in which it's used. Icons usually represent objects; activities are more commonly represented by windows. It's sometimes possible to reduce windows to icons when they aren't being used, but then they're at best suspended activities. No one has taken issue with our assertion in the notes (*Requirements Specification*, page 20) that icons represent objects; if you have examples which contradict (or, for that matter, reinforce) the notes, please tell us – we can't keep track of everything !

(i) The answer depends on which principle is taken to prevail in determining the behaviour.

- If the WYSIWYG principle prevails : The picture of the icon should be copied. And that's like the other copy-and-paste, so consistent.

- If the principle that the icon stands for its denotation prevails : then the icon stands for the file (or whatever), and operations on the icon imply operations on the file, so pasting the icon should paste the file. If so, does it mean all the contents (= insert the contents as instructed), or will a file pointer do (which makes sense for multimedia and hypermedia files, and can be handled as in the third answer to (b)) ?

- Practically : what if the icon denotes a directory, or a device ? (That might sometimes be all right for multimedia, when a pointer can be used, but is unlikely to be all right for ordinary files.)

Someone answered "both". My immediate reaction was "no" (followed quickly by "ha ha"), but that was wrong. In effect, "both" is exactly what happens if you copy an icon (by dragging) from a desktop into a window representing a directory on a different medium. (And see the comment on Windows below.) This emphasises yet further that the right answer depends on the circumstances.

In another good answer, it was suggested that the two possibilities should be distinguished by different means of selection (clicking the icon or drawing a selection box round it).

Yet another : it's an object-oriented interface, so the window should decide what it ought to do.

Several people said that the icon should appear in the document and convert itself into the file contents when clicked (which is quite a good answer), but didn't say why (which made it a bad answer). Some explanation is clearly necessary; there are not very many cases where a file's icon is replaced by the file contents, so it isn't very consistent.

Others asserted that icons couldn't appear in text. They didn't say why not. Perhaps that's because there's no reason why not; icons in text are quite common in hypertext documents (which represent one of the cases mentioned in the previous comment).

What I didn't know when I set the question was that Windows will (at least sometimes) let you do exactly what I described. (Select a file in the file manager (which isn't quite the desktop, but the same sort of thing), select "Copy" from the file menu, and use "Copy to clipboard".) When you paste it into a text document, the icon appears in the document; double-clicking the icon opens whatever the icon represents in the usual way. This doesn't prove much, nor does it necessarily define the only possible way to do it, but it does prove that people who answered that it couldn't be done were wrong.

(ii) For consistency – the inverse of (i).

- If the icon picture is pasted into the file in (i), then file information should form a new icon picture. It isn't very clear what the icon should do, as it has nothing obvious to represent; it could sensibly be entered into the same directory as the source file as a "file" with no contents and no type. Unless there were some other operations on such an entity, it couldn't do much, but you could copy it back to some other file some time. It becomes a version of the Macintosh "scrapbook", integrated with the file system instead of as a rather untidy separate entity.

- If the file contents are pasted in (i), then file information should be placed in a new file represented on the desktop by an icon. Presumably, and as in a file copy, the new file should be in the same directory as the original, and the icon should be similar to that of the parent item, as it's the same sort of material.

- If file pointers are used in (i), there's no inverse, unless it's a pointer that's moved.

Generally, the operating system must make provision to receive the information and do something sensible with it, which will presumably imply a new file of some sort. (Typically perhaps a dialogue to determine the context of the resulting thing – which directory it sits in, what type of file, etc.)

At least two people answered that it shouldn't be allowed, because the desktop was the operating system's window. But the whole point of the system metaphor is that you shouldn't think about the operating system : if I want to tear a page out of a book, or photocopy it, and put the page or copy on the desktop, why shouldn't I ? (- leaving out questions of vandalism and the ownership of the book.)

Few people included any account of what happened to the moved information and how the system maintained it on the desktop. It must be linked into something somehow – you can't just

write it on the screen and leave it there. I didn't worry about the omission for marking, and it isn't implied by analogy with part (i), but the situation is rather different – in part (i) it was clear that the transplanted material would have to be linked to the rest of the material in the window in some way determined by the programme managing the window, but in this part there is no such obvious answer.

(iii) In most cases, a fairly consistent system is achievable, but the biggest difficulty is in resolving the contending possibilities for the treatment of an icon. This is a real ambiguity in GUI systems which (to the best of our knowledge) is never sorted out. The anomalies are very clear in the Macintosh system, where you can cut and paste, or edit, the file name as text, but you can't do anything similar with the icon. As the icon is, in essence, just a pictorial file name, this is quite inconsistent.

Notice an elegant feature : if copying the icon means "copy contents", then copying and pasting an icon from desktop to desktop copies the file, thereby integrating data copy-and-paste with file copy. Maybe that's a step towards a smaller system genie ?

I asked for a comment on the achievable consistency – not the consistency of the answers you'd given to (i) and (ii).

QUESTION 3.

This question really wasn't done well.

(a) Any new processes are going to have to be given capabilities to perform basic tasks such as creating files when they are started. The capabilities could be inherited from the process which created the new process or from the capabilities of the owner of the process. In some cases it makes sense for the capabilities to be explicitly stated, giving the creating process control over the rights of the new process. In other cases, such as being able to create files, it makes sense to have this right passed on implicitly. The capability to create files should only provide the right to create files in directories accessible to the owner of the process. Since the create files capability is either the same as the owner's or is a reduced version of these the safety of file directories is guaranteed.

Apart from the substantial number of people who refused to talk about capabilities in their answer, most people did not seem to understand how capabilities function or why any system would bother to use them. A common answer was that before creating a file the process would ask the system for the capability to perform this task. Then the capability would be passed back to the system along with the request to create the file. Possibly a confusion between the capability to create objects of this type, and the capability of working with this particular object. The only problem with this approach is that it doesn't use the qualities of capabilities. If the system is going to be asked to provide a capability as the capability is about to be used, what is the difference from a system which checks a subject's rights to create a file without capabilities? In both cases some other method of protection is being used to provide the basic rights.

Some people correctly talked of the user database as being one possible source for original capability information. Unfortunately most then reverted to the discussion outlined above. A better answer would be that the capabilities stored in the user database would automatically be given to any processes started by that user.

More than one person gave an answer about object-oriented file systems. The word "object" means something completely different when we are talking about "subjects" and "objects" and security.

(b) Capabilities are created by concatenating the object identification with the required access rights and then concatenating an encrypted checksum of this data to stop tampering. In Amœba the object identification is two numbers, one identifying the server (48 bits) and the other number identifying the object (24 bits). The access rights are stored as 8 bits. The encryption algorithm must be secret, it generates an extra 48 bits of check. When a capability is presented to perform a task, the encryption algorithm recalculates the identification and the rights and checks this against the stored checksum. Only if the answers agree is the

task carried out. This means that capabilities cannot be altered. Any change in the capability requires a corresponding change in the encrypted field, and the encryption algorithm is supposed to 1) be secret and 2) be "very difficult" to reverse engineer. This method does not stop somebody else getting hold of your capability and using it. But the passing around of capabilities is one of the "advantages" of using them.

Not many people appeared to have read the notes, or if they had, they certainly didn't understand them. A common misunderstanding was that files were encrypted rather than capabilities. This led people to believe that it didn't matter if files were read because they couldn't be understood. The capability system is designed to allow access only to those subjects with the required capability.

(c) Even though it is possible to find a valid capability it is very unlikely. We assume that the encryption algorithm gives a very good distribution of numbers, every bit sequence is equally as likely. Then if the attempted capability forger can generate, transmit and verify the result of a capability every microsecond it would take 2^{48} microseconds just to generate all of the possible values.

2^{48} microseconds ~ 2^{28} seconds
2^{28} seconds > 256 million seconds > 4.2 million minutes = 70000 hours > 2800 days
(actually 3258 days)

So on average it would take more than 1000 days to "guess" the required capability. Which is a long enough period of time to hope that no one would try to do this. Two further points need to be made. If the capability system didn't warn the system's administrator that someone was attempting to breach security, then the design is sorely amiss. Secondly as machines get faster we are going to have to increase the number of bits. We could postulate a million fold increase in speed, which reduces the average testing time to about 2 minutes. In fact later versions of Amœba have doubled the size of the capabilities.

We intended the simple calculation to be more than stating the obvious (2^{48} different combinations). Most people made statements such as "it would take forever to find the correct sequence of bits"; as you can see from the answer we really wanted an estimate of how long. By now you should have some idea of how fast computers work. In the answer we have assumed a very fast machine, because this shows the greatest weakness to the technique. Even if your estimate differed by several factors of 10 you would have got the marks.

Some people commented on how easy it would be for the system to change the encrypted information, thus making it more difficult to guess the right answer. Unfortunately it is not a trivial matter to alter capabilities, because all legitimate users have to get the new capability, otherwise their legal access is denied.

QUESTION 4.

(a) (i) Slower access because of having to continually read directories, especially if the required files are deep in a file hierarchy. There may be a file cache which holds recent file blocks; this would speed up access to frequently referenced directories. Has very low memory overhead. Easy to maintain consistency as directories are written back to disk frequently. Little difficulty in adding or removing file systems.

(ii) Slows down startup, but all file descriptor searches are then entirely in memory and hence fast. (In a virtual memory system there would be some slow down for rarely referenced directories as they will have been paged out.) Depending on the size of the system, this could be a very large amount of information. For many systems it would be an acceptable size (50 users with 100 files, each file directory entry of size 100 bytes gives half a Meg, and then double that for all the system files (on UNIX anyway)). Since all references for directory information are made in memory we need to ensure that changes are reflected to disk. More difficult to add file systems (all the directory information for the new file system must be loaded). Maintaining consistency between shared file systems is much more difficult.

Almost everyone said that (i) was more memory efficient and slower than (ii). More marks were given for better explanations. Only a few people mentioned the differences at start-up and no one mentioned sharing the file system between several machines and similar difficulties.

Several people got confused about the protection aspects of these approaches. Because we mentioned writing back to disk if changed in (i) some thought that (ii) didn't do this. Others thought that (ii) was more secure because it was maintained both in memory and on disk.

Unfortunately some people thought that all files (not just directories) were read into memory as the system started. Please read the questions carefully.

(b) In this answer I am assuming that apart from the length of the filenames we have a typical hierarchically structured file system, with arbitrary nesting, and that directory names are treated as other file names.

Advantages for the user – no restriction on descriptive naming of files. The filename could include notes the user might want associated with the file which are not of the same type as the file contents. The user could include arbitrary keywords, making the filename a useful search tool.

Disadvantages for the user – if long filenames have been used then having to remember and type them or select them from a list (the pathnames could be even worse). Slower response time, as searching for a particular file could take much longer, e.g. using a "find" command to find a file with a particular string of characters in its name. The whole idea of having a filename is to give a convenient handle to retrieve the information the file contains; filenames which are too long violate this principle.

From the system's point of view there are two different things going on. Because we are allowing filenames of **any** length we can no longer store the names in constant sized arrays. This is a separate issue from having flexible length filenames. Flexible length filenames can be stored in constant sized arrays (as long as we have a maximum length) or in variable sized arrays (even with a maximum length). This latter method is how some UNIX systems do it. So some of the following advantages and difficulties occur in systems with restrictions on the length of filenames.

Advantages for the system. It is possible that with flexible length filenames, the average length of a filename will be smaller than a preset inflexible maximum size, thus saving space.

Disadvantages for the system. Without a maximum size, all file table entries, whether on secondary storage or in memory have to be able to grow to an unlimited extent. Displays of filenames will have to be flexible, being able to cope with screenfuls of information for single filenames, or else abbreviating them. This means the displayed sections of different filenames may not be unique. (This consequence for the system is also a disadvantage for the user.)

Not only do directory structures have to grow (and contract) as the number of files changes as with a limited filename length system, but a directory holding the information for only one file could be of any size.

Regardless of how we implement our directory entries, maintaining the name space after file moves and renames will be more complicated. With fixed size names, directory entries could simply be reused, or marked as empty for later use.

Very few people thought deeply about this question. A typical answer said it was great for the users because they could use whatever name they wanted, and that it was bad for the system because of the vast amount of space the names would take up. More analysis of the situation was required to get good marks. As an example you should have asked yourself, why is it an advantage for the users to be able to call files anything they like, how might this facility be used?

Some people couldn't quite believe the question. They talked about how it wouldn't work because the system would have limits on the size of filenames and these limits would be overflowed by names that were too large. In a similar vein were the answers which talked about running out of memory (both primary and secondary) with huge filenames. Since we don't normally worry about running out of space with our files (of course it does happen, but our system administrators give us warning, and we apply for a bigger disk drive) why should we worry about running out of space with our filenames?

(c) The advantage of (almost) unrestricted naming can still be achieved with a maximum length (the length of an average sentence would suffice). Rarely do I need more than the 31 characters the Mac file system allows. A real limit of 1 or 2 hundred characters should be ample.

For adding notes about the file, a separate note system could be added. The note could be a (usually) small file, accessible from the directory entry. (Perhaps a third file fork – data, resource, notes.) The Macintosh allows you to add notes to files, however these are stored in the desktop file and they are lost when you rebuild the desktop.

In a similar manner keyword lists could be associated with the file in a separate structure.

Hiearchical directories were a very popular solution to this problem. There was nothing in the question which indicated that we didn't already have them. In fact directories are mentioned throughout the question and traversing directories is explicitly referred to. The advantages and disadvantages most people referred to also exist under a hierarchical directory structure. It is true that the organisation and naming of pathnames leading to a file could include the information from a long filename, however, some of the uses we may put a long filename to would not be appropriate for parts of a complete pathname.

Another common answer was to use two filenames, one for the system to use, which has a limited length, and an unlimited length filename for the user to use. Some of these methods were impossible, most didn't solve the problem and others solved problems but without mentioning the new problems they caused, in particular non-unique naming of files.

Related to these were the answers which compressed the filename. First of all the number of bits required to store compressed data grows as the data grows (with some exceptions). Secondly, searching for information may get slower (it depends on the relative speeds of decrypting and i/o).

The question does not assume any particular method of implementing the long filenames. Hence a peculiar implementation was not a satisfactory answer to this section. All implementations people came up with suffered from some of the disadvantages mentioned in part (b).

Robert Sheehan and Alan Creak,
July, 1995.