Computer Science 340

Operating Systems

SECOND TEST, 1993 : Answer to question 1

Material in *italic* type was added after marking the test.

REMARK 1 : After many comments from you that the questions in the first test were too wordy, I deliberately made these questions brief. The experiment cannot be described as successful. People still answered questions which I hadn't asked – if anything, rather more frequently than I've noticed before, though one experiment can't support generalisation.

REMARK 2 : I assume that you are all well aware of my views on the course material, from lectures and distributed notes. (If you are not so aware, then I'm entitled to wonder why not.) You will also be aware that I don't always agree with the textbook. If you've read a bit further, you may also have noticed that different textbooks don't always agree either. It is not impossible that you don't agree with me. That doesn't bother me, provided that you have given some thought to the matter and you know why you don't agree with me. I am even ready to give marks (and often have given marks) in tests and examinations for answers which are quite different from mine, provided that they make sense.

But I am giving the marks, and I have to assume that questions have been answered in my way unless you make it clear that your ideas are different, and that you know what you're doing. If you just come out with answers which I believe to be wrong or incomplete ("a file is a named collection of data"), I shall mark them as wrong or incomplete. If you want to stick with that definition, convince me. ("I do not believe that the structure of the file or the distinction between the file and its attributes is significant because ....")

However, in fact I know that you all (well, almost all) agree with me, or you'd have asked me to justify my opinion. Hardly anyone has visited me, or sent mail, to query issues raised in lectures or notes, or taken advantage of the pause which I try to insert in all the lectures if there's time.

(a)

Files are persistent arrays of data, while **streams** are temporal sequences of data.

(That answer is sufficient for the question; more completely, and allowing for Macintosh-style files and possible further developments :

```
file = record
     attributes : set of values;
     data : array of
          component : array of record;
     end;
```

No, I don't suppose it would be accepted by a Pascal compiler, but it shows the structure.)

File operations : **read** : file, index → datum; **write** : datum, file, index → file.

(Or, more fully :

read : file, component, index → datum;
write : datum, file, component, index → file.

read attribute : file, attribute name → datum;
set attribute : datum, file, attribute name → file.

)

> Notice that **open** and **close** are not strictly operations on the file; they are primarily operations on the file information block.

Stream operations : **get** : stream → datum; **put** : datum, stream →.

> (A pure stream cannot have attributes, because there's nowhere to put them. Of course, there's no reason why you can't define things which must appear in the stream – such as headers – to carry information about the stream, but then you have to provide machinery to collect this information and save it somewhere, and you have something more complicated than a simple stream.)

Files and streams are not usually distinguished in programming languages, all being called "files". The nature of the correspondence between files in programming languages and the abstractions defined above depends on the sort of "file" concerned, which is usually identified with some device.

- For stream-like devices, such as keyboards and screens : **get** and **put** (commonly disguised as **read** and **write**) work as expected, so the correspondence with abstract streams is quite good. The main difference, particularly with input streams, is that real streams are characterised by absolute time as well as by sequence, so if data are "got" too slowly some may be lost.

- For file-like devices, such as discs : the correspondence is good in principle in that the operations are available, but is often obscured by implementation mechanisms – so that, rather than providing an explicit **read**(filename, index, destination) instruction, the system may distinguish the index administration as a **seek** operation followed by automatic sequential increments instead of leaving it to the programme. The system must therefore maintain the index variable for itself.

In asking you to consider abstract data types, I wasn't expecting a terribly formal set of definitions, but I was expecting you to take a little care in your definitions. Please think about what you're writing.

If I'd asked you to "distinguish between real and integer as abstract data types", I'd have expected statements of what reals and integers were and what sort of operations could be applied to them. ("Real numbers can take on any numerical scalar values whatever, integers are limited to whole numbers. Standard operations on real numbers include addition, subtraction, multiplication, and division; integers can be added, subtracted, and multiplied." No, it isn't a mathematical specification, but it's appropriate for the context.) If I'd then asked for comment on how actual implementations matched these types, I'd have expected something about whether reals and integers in programmes followed the ideal rules set out as the definitions. ("Real numbers in computer languages can only take on a limited number of values and the arithmetic is approximate, because of the finite precision of the representation", and maybe something about overflow, DIV and MOD operations, and so on.)

If a file is a set of records, how can you identify an individual record ? You must have at least an array to associate a record number with each record. (You can identify individuals in sets, but only by content – which isn't how we usually think of files. If anyone had answered in that way, I'd have accepted it, because it's quite close to indexed files.)

*If a file is a set, or even an array, of records, then how can it sustain a **seek** operation ? There's nothing in the file to record the effect of **seek**. Conversely, if you want to apply a **seek** operation to a file, then the file must be at least a set or array of records and a "next record number".*

I was lax in not defining the nature of the file operations for which I asked you. I had not intended to include delete, rename, change protection, and so on. I accepted these operations if they were presented. It was something of a surprise to find several answers giving this list without read and write.

Why were there several statements to the effect that "a file has a fixed length" ? That used to be so when files had to be stored contiguously on the disc, but it's certainly an uncommon restriction now.

Several answers included words something like "... a stream is stored ...". It isn't. Once it's stored, it stops being a stream, just as a river when prevented from flowing becomes a lake.

A stream can be a stream of anything – it doesn't have to be bytes.

Many people had some difficulty in responding to the second sentence of this part because they hadn't been clear enough in their answers to the first sentence, so hadn't anything specific to talk about. You may care to look at your response to the first sentence, and ask yourself whether it looks like a discussion of abstract data types. If you have difficulty in answering that question, see whether you can find the definitions of the abstract data types. You should at least find a description of the data structure.

The second sentence was specifically about how files were implemented in programming languages. That's significant, because the operating system has to provide the supporting software which compilers use to link the view of files seen in the languages with the physical implementation on whatever device is involved. For example, if you thought that ***seek*** *was an operation, you would have to say how the file pointer was advanced on a read or write operation.*

In Pascal and C, files are not implemented as arrays, because there's nothing which corresponds to an indexed variable in the languages' constructs for dealing with files. Cobol has an index variable which you can set; Fortran provides true random read and write operations as well as serial reads and writes where the index is automatically managed.

In some answers, there was an emphasis on relationships between streams and files, often stressing that a file was always in effect read or written as a stream. That's true enough, but a matter of implementation : one reason for concentrating on the abstract data types was to avoid such connections. (You can implement real arithmetic using integer hardware, but that has nothing to do with the real data type.)

(b)

Paged :

flat – single numeric address.

The programme sees memory as a single continuous address space.

Internal fragmentation – some pages aren't full.

Segmented :

segmented – two-component address.

The programme sees memory as a number of continuous address spaces, so to identify a specific point it must both identify the address space required and specify the displacement within the selected space.

External fragmentation – arbitrarily sized segments don't pack together well.

Overlay :

trunk-and-branches – single numeric address, with overlay block number to resolve ambiguity.

The programme sees memory as a number of continuous address spaces, much as in a segmented system, but in this case there is one address space which can be used at any time without formality, while for the others identifying the address space required and specifying the position with the address space (which is not quite a displacement) must be carried out as separate operations. ("Trunk-and-branches" because you can always climb the trunk, but you can only climb one branch at a time.)

Occupancy of overlay area determined by size of subroutines; also constrained by requirements imposed to avoid calling between overlay blocks.

This part begins, "Describe the memory models ...". That seemed to me to be simple, precise, and straight to the point. So why did hardly anyone do it ? You can hardly plead ignorance – there are more than four pages specifically about memory models in my notes (Support for execution, pages 27-31 –

and memory models for flat and segmented memories are discussed on pages 33-35), and I talked about it during the lectures. The second sentence of this part was about efficiency in using memory. That's not the same as efficiency in using time. In designing your system, you may reasonably seek ways in which you can trade these against each other to achieve some optimal overall system efficiency, but that's another matter. If I were to ask you about the efficiency of a vacuum cleaner, I'd expect you to tell me whether or not it was an efficient cleaner. While it might then be reasonable to add, "... but it takes a very long time", the primary concern is with its efficiency at cleaning.

The memory model is the view of memory seen by the programme – which is essentially a matter of how memory is addressed. The model itself has nothing to do with implementation. The machinery of paging and segmentation (addressing tables, etc.) is quite invisible to the programme. (Ideally,) it's the operating system's job to make the model work using the available hardware, but that wasn't relevant to the first requirement in this part where you were asked to describe the memory models.

Several people mentioned the stack model. It has no particular relevance to the question, as it's associated with certain sorts of programme structure or execution model (nested procedures, recursive execution) which weren't mentioned. It's quite like the segmented model, except that some or all of the segments are arranged in a stack, which makes it special. The stack model is rather easy to implement if you have hardware which supports a segmented model.

Some (but certainly not all) mentions of stacks might have been the result of a misunderstanding. I had intended the question to be about the memory models which inevitably appear when you implement paged (etc.) memory; it could (just) be read alternatively as requiring a description of implementing other memory models within paged (etc.) memories. I may have missed some which are answered in the way I didn't expect, as I didn't discover the ambiguity until quite late in the marking. If you think you've been hard done by for this reason, and also think that you can persuade me that your answer fits the alternative interpretation better, then please bring your answer to me and we'll negotiate.

Several people thought that the overlay technique was based on a stack memory model. There is no stack. If there were, there would be some sort of pop operation, but an overlay system maintains no memory of what was in the overlay area before the current occupant.

A common comment was that a segmented memory conforms to the flat memory model. I was puzzled by this for some time, until one answer explained that the model was flat because each of the segments had its own flat memory. That's a bit like saying that a tower block is a one-storey house because each floor only has one storey.

I had missed the point that page tables and overlay management software occupies space, which could reasonably be considered as wasted.

Appeals to virtual memory considerations to demonstrate inefficiency in paged and segmented systems are not valid; paging and segmentation can be (and are) used as memory allocation techniques even when virtual memory is not used.

Alan Creak,
August, 1993.