MEMORY MODELS

Meanwhile, back in the real world, we have decided that programmes require memory, and it's the operating system's job to provide the memory as required. How is it required ? From our investigations in the previous section, two conditions are clear : it is required at unpredictable times, and in unpredictable quantities. Another desirable feature noticed was that it should be possible to connect together different areas of memory to form more extensive structures. To gain a little insight into just what might be useful, it is interesting to investigate some existing *memory models* assumed by programming languages.

We use the term "memory model" (which we mentioned earlier in *PROGRAMMES*) in the sense of a mental model, which we have in mind (if we need it) as we design software. There is clearly a link with the notion of the system metaphor which we discussed earlier, but, as we shall see shortly, it has not usually been thought useful to simulate a "foreign" memory model given a hardware memory.

Memory models came into existence with high-level programming languages. Before they came along, people used assembly languages, or other means of programming which for the most part used a memory model indistinguishable from the computer's real memory. It's still a very useful organisation, and adequate for many purposes. We shall call it the *flat memory model*. High-level (or, at least, higher-level) languages began to appear in the 1950s, and a surprisingly diverse set of memory models turned up. The inventors of Cobol and Fortran simply assumed a memory organised like that, so their compilers compiled programmes accordingly, but Algol was based on a tree-like structure with branches which existed only while they were being executed, and Lisp relied on highly linked structures of arbitrary size.

There's nothing wrong with using a flat memory if that's what you need; but we may quibble with the approach. The assumption that you have to use a flat memory model because that's what's in the computer is practical enough, but if you know that some other memory model would serve your needs better it would be more useful to seek effective ways of supporting that model. In the long term, accepting the flat model is stultifying, because it means that you never try anything that won't fit a flat memory, so you never have any incentive to try to build other sorts of memory[SUP17]. We shall try to keep our feet firmly on the ground, but also investigate what sort of memory will be best suited to our aim of delivering good service to people trying to get work done.

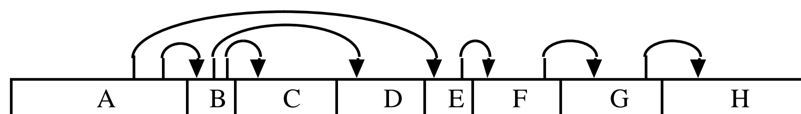
An example. We've already noticed that recursion was not popular – indeed, for many years, it was almost axiomatic that recursion was a difficult programming technique, and horrendously expensive to implement. There were even articles published[EXE13] on how to eliminate recursion from your source programmes. The argument was based on the proposition which can be roughly expressed as "flat memory, therefore recursion is bad". The proposition is fair enough, but a view from the software end makes it look different : "recursion is good, therefore flat memory is bad". Unfortunately, the software view was rarely taken into account, and programmers had to make do with the hardware they were given.

We are not alone in this view; here's a comment[EXE14] from 1967 : "Recursion is widely regarded at present as an interesting 'frill' in a programming system. There is also considerable prejudice against it, which is understandable since most machines are not designed to handle recursive procedures, and therefore do so inefficiently. But if the advantages of this way of programming come to be appreciated, machines will be designed to facilitate it.". Alan Kay, 30 years later[SUP12] (1997) : "The hardware manufacturers have given us, almost in every case, unbelievably bad designs and asked the software people to make them look reasonable. And, I think

that is completely backwards. The software people should be coming up with the most appropriate things and then forcing the hardware people to optimize those". What went wrong ?

WHAT PROGRAMMING LANGUAGES BELIEVE.

Here's an impression of the programmer's model of a Fortran programme. Each chunk is a "programme unit"; A is the main programme, and the other chunks are subroutines or functions ("subprogrammes").



The scopes of names (other than subprogramme names, which were known everywhere, and names of **common** blocks, which we shall ignore) were limited to the programme units in which the names were used, but the subprogrammes were simply regarded as bits of the programme which it was convenient to encapsulate separately. This model has sometimes been likened to a geographic view of the computer; each programme unit has its own territory, and getting in and out is comparatively difficult. With that model, **goto** becomes a very appropriate – and descriptive – instruction to use. Subprogramme entry was thought of as no more than an air journey to a different country, with the minor peculiarity that you had a return ticket. Perhaps because of that view, recursion wasn't usually even discussed; you don't think of using a small country recursively. This utilitarian view of programme structure is illustrated by the possibility of defining several different entry points (airports) for a subprogramme, and provision for passing labels as parameters, so that the return from a subprogramme could, in effect, be directed to a point distant from the subprogramme call.

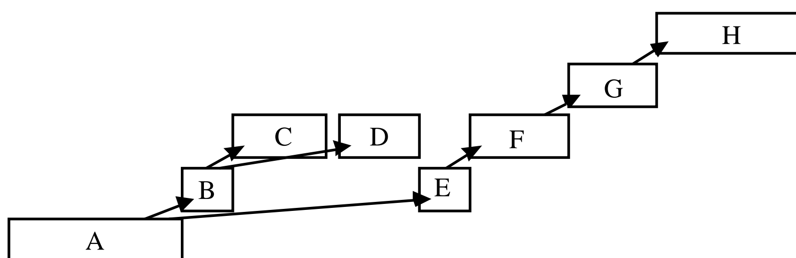
Be-fair-to-Fortran section : The Fortran specification itself did not directly prescribe any sort of memory in detail, and much of the Fortran style was determined by assumptions made by the compiler writers, and thereafter assumed to be a natural part of Fortran by people who used it. The specification did explicitly say that data in memory local to subroutines were not guaranteed to be preserved when the subroutines were inactive; but most compilers implemented static memory allocation, so that the contents of memory persisted between subroutine calls, and many programmers came to rely on this "feature". Recursion was also banned, by custom if not by edict, whether for the reason mentioned above or because no one knew how to do it efficiently. (In one textbook[EXE12], the factorial function is used as an example of functions and subroutines – both of which compute the factorial by iteration, which is fair enough, but the book doesn't mention recursion even as a possibility !)

Be-fair-to-Cobol section : We've already mentioned that in Cobol, alone among the early languages, there was provision for the declaration of tree-structured records, implying some recognition that the simple flat memory, and the arrays implemented in Fortran and Algol, were not sufficient. Even so, the form of the declarations makes it clear that it was the programmer's responsibility to look after the details of mapping the components of the structure onto the flat memory, and no help from the system was expected.

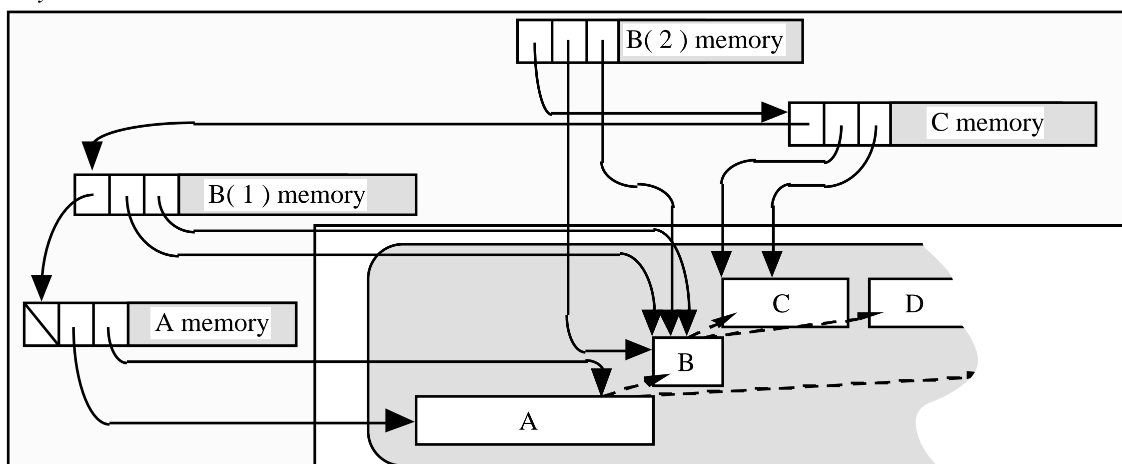
Algol was different. Its inventors assumed that memory could be acquired and released by a programme as it was executed. In particular, as each procedure (strictly, each block, but procedure will do for our purposes) was entered, a new piece of

memory, sometimes called the *activation record*, was acquired for the procedure's local storage requirements and certain housekeeping functions. Because of this model, Algol implementations (if they worked at all) had no trouble with recursion, which was forbidden in Cobol and Fortran. Because of the Algol semantics, the activation records were always structured as a stack, so we shall call this the *stack memory model.* (We shall use the common name "stack" for the structure, though it's not precise. It is assumed that ordinary push and pop operations are available, but also that items on the stack are accessible at all times and can be modified internally in situ.)

Here is an attempt to depict the Algol programmer's view of a programme similar to that of the Fortran example. The same programme units appear, but they are now clearly structured as a tree, expressed in the programme source code by containment – so the text of procedure C is completely within the text of procedure B. There is no expected relationship between the positions of the memory areas for the different programme units, though in practice they might well be laid out in much the same way as with the corresponding Fortran programme.



To be more precise, we should identify that as the *static* structure of the programme. When the programme is running, it also has a *dynamic* structure in which the sequence of procedure entries is encoded; Fortran has no equivalent of this notion. Here is an impression, incorporating lots of implementation assumptions which could be made differently, of the dynamic structure at a point during execution at which the main programme A has called B, which has called C, which has recursively called B :

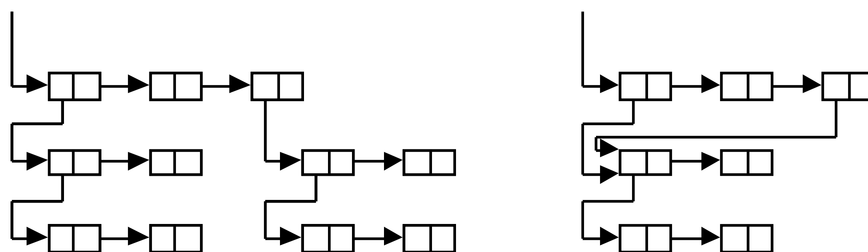


The stack of activation records is shown as scattered about a bit to emphasise the independence of the memory chunks used. It is implemented as a number of logically separate memory areas each (except the lowest) linked to the item below it in the stack. Notice the profusion of links; each activation record is linked to the record of the procedure which called it, and to the code of the procedure it is executing, and to the point within the code which has been reached during execution. Not all these need be implemented as explicit pointers, but it is clear that there is some basis for our assertion that facilities for linking memory blocks will be useful. This diagram illustrates another difference between the Fortran and Algol models : data storage in Algol is within the activation records, so is clearly temporary, while in Fortran data storage is thought of as included within the memory areas of the programme units.

A point of particular interest for memory management is the appearance in the Algol dynamic structure of an arbitrary number of memory chunks, which are assigned and released as the execution proceeds. This was a very new idea indeed, and did nothing to increase the popularity of Algol in the 1960s, but we shall see that it has become an important structure in practice.

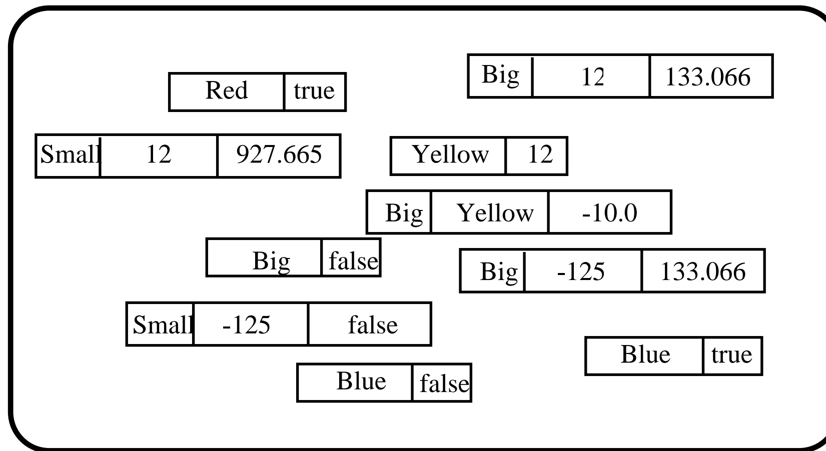
The Burroughs B6700 machine was designed (around 1970) to execute Algol-like programmes. It had an execution architecture which implemented activation records, and a "segmented" memory organisation modelled very closely on the diagram above. Its Fortran compiler had a special switch which you had to set if you wanted your programme's memory to be allocated in a flat memory space; if you didn't set the switch, Fortran programme units were allocated just as though they were Algol procedures. (You could even use them recursively.) It was alarming that, without the switch, many Fortran programmes which would run on other machines refused to work on the B6700, which detected their attempts to address memory outside the allocated address space. Burroughs had, perhaps naively, relied on the Fortran specification : see the remark above. We don't remember whether they managed to prevent recursion in switched mode.

Lisp was different too. In Lisp, recursion is not so much permitted as enforced. In the semantics, it is assumed that, as in Algol, you can always get more memory when you want it, but the memory isn't tied up in activation records; instead, a lot of it ends up connected together in complex linked structures, so we shall call this the *linked memory model*. Here's a picture of typical (small) list structures reproduced from an article[EXE15] by John McCarthy, the inventor of Lisp :



While it isn't very informative, it shows the arbitrary structures which can easily be built up in a Lisp programme. The small memory chunks illustrated are typical of the structural representation of Lisp; data storage (property lists) and auxiliary structures (such as symbol tables) are also implemented as linked structures.

For a final example, consider Linda[IMP13]. Linda is not a programming language, but a scheme which uses a common memory area to provide communication between different programmes running simultaneously in a computer system; we shall discuss it further in the chapter reasonably entitled *LINDA*. It is based on a "tuple memory", which is simply an area in which we can store a number of data structures called tuples. The tuples are not thought of as connected, but the memory can be searched for tuples of a given pattern. We shall call this the *heap memory model*. Other names for this sort of memory are *associative memory* and *content-addressable memory*, both of which make the point that the aim is to identify the memory datum you want by some property of what is stored there, not by its physical memory address. Here's an impression of a Linda memory :



The details of the different models are not very important for our purposes, but it is interesting to compare the types of address used in the different models. The form of the address is the most characteristic feature of a memory model – which is just what you would expect for a top-down analysis, because it should be the requirements of the software which determine the memory model, not the reverse.

For the **flat** model, memory is a single array, so all that is required to identify an object in memory is the array index – which is, of course, the traditional memory address.

In a **stack** memory, an address has two components – one to identify the activation record, usually related to its position on the stack, and one to give the position of the item within the activation record. This two-component address is very helpful in implementing features like recursion, as it is easy to construct two or more activation records which differ only in their positions on the stack.

In **Lisp**, data are represented as *atoms*, and the only structure is a thing called an *S-expression*. An S-expression has two components, a head and a tail (historically called CAR and CDR, a fact we shall henceforth ignore). All addresses are composed of list names and head and tail operators, which act just like a set of instructions to travel to the addressed object starting from a base point in the list : *hd(hd(tl(tl(tl(hd(tl(hd(hd(tl(tl(list))))))))))).*

A **Linda** address is just a partly defined tuple. An "address" *(xxx, ?i, 25)* will address any triple in the common memory area which has the string *xxx* as its first component, any integer as the second component, and the number *25* as the third. If the pattern matches several triples, one is selected at random. With the memory configuration shown in the diagram above, therefore, *(?s, 12)* will identify the single tuple *(Yellow, 12)*, *(Big, ?i, 133.066)* identifies two tuples, while *(?s, ?i, ?l)* matches only *(Small, –125, false)*.

We describe these models here not so much because of their own intrinsic interest, but because it is important to realise that different models exist. Ideally, we would hope to be able to use the memory model of our choice, and it is clearly the job of the operating system to provide the required service.

In all the models, except the flat model, memory is treated as a resource which can be acquired in chunks of arbitrary size. This is not a big surprise; it is precisely what we foresaw in *HOW PROGRAMMES USE STORAGE*. It should be clear even from the sketchy account which we have presented that our expectations are thoroughly borne out in practice, and that our introduction of the notion of segments is no more than a reflection of the way in which ordinary programmes naturally use memory. Segments are particularly significant in that they are common to the different models. While it might be unreasonable or impracticable to implement all the memory models in detail in a single system, it is much more feasible to provide support for segments which the different language processors can then exploit to implement their own memory models.

Of course, we have not presented an exhaustive review of all possible memory models. We've chosen to describe these examples because they are all fairly simple, but

cover quite a wide range of structures. Generally, any programming language presents some memory model to programmers who use it, so there will be memory models appropriate to functional languages, object-oriented languages, logic programming languages, spreadsheets, graphical programming systems, and so on – indeed, any programme which you can use to build up structures, give names to objects, store and retrieve data, or other such operation will provide means of associating objects, naming them, or moving them about which amount to a memory model. Among all these varieties, though, we know of no model which introduces anything alarmingly different from those we have described.

We would be surprised to find anything radically new. It is in principle just what we expected from our earlier discussion, and in practice it covers the range of possibilities : you can identify an object by saying where it is (specifying an array index or pointer), giving it a name (a structure, as in Pascal or C), or describing its nature (associative memory, as in Linda). When programming, we use them all quite naturally : to give something a name in a computer, you use a symbol table, in which you may search for a name in an array using the description of its nature (the character string), and find the position of the object from a parallel array. The methods we've described are combinations of these elementary operations; if you can think of another way, we'd like to know.

BUT THERE MUST BE A *REAL* ADDRESS ?

If that's what you're thinking, and if it means that, despite all this silly messing about with weird ideas about what addresses are, they must all be translated into a conventional index into a flat memory before they can be used, then you've missed the point. And you're wrong.

The point is that, even if you're right, if some non-flat address is the natural one to use in the programme which is being executed, then this translation into a conventional index is probably a waste of time; if the address which the programme most easily produces could be used directly, then we'd save time. It would therefore make sense to investigate ways in which we could use different memory models for different programmes.

And you're wrong because not all memories are linear arrays. A store called a bubble memory[EXE8] was at one time seen as the successor to mechanically rotating discs; data were encoded as streams of "magnetic bubbles" driven round a cyclic path, and monitored as they passed a particular point in the path. The heap memory model could easily be implemented with this device by simply watching the output until an acceptable pattern appeared. And if you think that's ridiculous, you might like to know that ICL didn't, and they made a lot of money out of their "Content Addressable File Store"[EXE9] (CAFS), which did practically the same trick simultaneously in parallel on every track of a hard disc drive; you could search a whole file for a pattern in one revolution of the disc.

WHAT WE DON'T WANT.

The notion of a memory model is useful provided that it doesn't get mixed up with anything else. We make this point explicitly, because experience shows that memory models are easily confused with memory management techniques. We urge you to bear in mind that a memory model is something you want to use when writing a programme – so operations intended to be invisible to the programme cannot be memory models. In particular, to anticipate two topics which will appear later :

- There is no such thing as a *paged memory* model; paging is a technique for allocating memory in small chunks to make it easier to pack programmes into the available memory space. It is carefully designed to make no difference to the programme's view of memory.

- Similarly, there is no such thing as a *virtual memory* model. Virtual memory is a technique for increasing the apparent memory size by permitting some disc space to

be used as if it were memory. This also is carefully designed to make no difference to the programme's view of memory.

The real test of whether or not a new memory model is involved is to look at the means of addressing; in both the cases mentioned, the memory management technique makes no difference at all to the nature of the address used.

WHAT WE HAVE.

We have some hardware. Most hardware implementations of memory are still quite close to the flat model, despite the example of the Burroughs (now Unisys) segmented memory (and a later rather similar architecture implemented by ICL). As well as these segmented memories, well adapted to the stack memory model, there are Lisp machines which presumably do something appropriate. (Other implementations of "segmented memory" are often slightly tarted-up flat memories : beware.)

It is worth paying some attention to segmented memory, because, even though implementations of pure segmented memory are few, the ideas turn up in many other contexts, as we observed in the previous discussion. The "real" memory of such an implementation is a flat memory, because that's the only sort of memory we know how to build sufficiently cheaply, but segmentation is imposed at a very low level by hardware. A segment is defined by a *segment descriptor*, which is a hardware-recognised data structure containing an address in the flat memory (the segment origin) and the segment length. Other items might conveniently be added (for example, protection fields) but they are not part of the segmented addressing scheme proper. This descriptor is in effect the name of the segment; to identify a memory word, a displacement within the segment is necessary, so the address is a pair (segment descriptor, displacement).

It is important to realise that a segment descriptor is *not* just a pointer; it is an entry into a new and independent address space, quite separate from any other address space in the system. The segment length is, in the abstract model, unnecessary, as no location in one segment can ever interfere with any other segment. In practice, we don't know any way to implement segments except as pointers and displacements, so the segment length is essential to guarantee that segments' address spaces remain distinct. It is the presence of the length in the descriptor which made it possible for the Burroughs system previously described to check for array bounds errors at every array reference, with no waste of time. Implementations of "segmentation" where the "segments" can overlap arbitrarily are among the aforementioned "tarted-up flat memories". Observe, though, that the objection is to the arbitrariness; properly nested segments have their uses.

There is a catch here. Earlier, we defined a segment in terms of the semantics of the programme; now we're using the same word to refer to a hardware feature. There is no law of nature which constrains software writers to identify one with the other – so a hardware segment might or might not be used to contain precisely one semantic segment. The situation is not simplified in any way by the imprecise definition of the semantic segment; the notion of a chunk of memory of arbitrary size holding code or data which naturally form a unit could be applied to anything from a byte to a programme. We shall try to ensure that the sense of the word is clear when we use it, but be warned that it isn't a very well defined term.

Notice that memory caches, virtual memory, and other such techniques don't affect the principle : they are primarily ways of implementing bigger and faster memories, but, while they sometimes introduce machinery which can be used to advantage to simulate other varieties, they don't change the underlying model.

THE OPERATING SYSTEM'S JOB.

If the operating system has a place here at all, then it is to provide a *memory service* which will make memory available as and when it is required, and look after low-level administration. This fits in well with our description of the operating system's task as the provision of services which everyone wants but no one wants to do. So far as the higher level operations are concerned, the operating system must make the required memory model work using the actual hardware available. There are two parts to this : implementing the addressing mechanism, and providing support services.

Implementing the addressing mechanism is a matter of mapping one sort of address onto another sort of address. For example, to implement an Algol-like language on a machine with flat addressing, the two-component Algol addresses { <which activation record>, <displacement within activation record> } must be converted into conventional single-component addresses.

This conversion must be performed every time a memory address is evaluated. It is therefore significantly expensive if carried out in software, so this service is in practice rarely, if ever, provided by an operating system. Instead, any software requiring an unusual memory model is expected to be responsible for its own sort of implementation (provide an interpreter, as is common with Lisp; map it in the compiler, as in most implementations of Pascal, etc.), thereby guaranteeing incompatibility with any other software running on the machine. For these reasons, it is not practicable for the operating system to do its proper job; the hardware memory to a great extent dictates the software memory models which can be effectively used, and "foreign" models are rarely available. (We might compare this restriction with the limited provision for file structure in most implementations of file systems : the flat memory is in many ways analogous to the "flat disc".)

> *Is it really impracticable ? That depends on the cost and inconvenience of using an interpreter. The speed penalty is generally estimated to be around a factor of ten, which sounds severe, but it is not very long since people were delighted to use processors with one tenth the speed of today's models. The popularity of Java, designed to run on a simulated machine, shows that people will accept reduced speed – and the fact that in most routine computer use far more effort is spent on maintaining the pretty user interface than on useful work does suggest that some of the arguments for the retention of a flat memory model are somewhat overstated.*
>
> *It is true that there will always be cases where the full speed of the processor can be used to advantage, but, if our analysis is accepted, it would not be unreasonable to lean on the hardware manufacturers to cause them to build machines with microprogrammable memory models. It's certainly possible in principle, and it would open the way to operating systems which really could provide a memory model to fit a programme's requirements. The memory model required would become one of the code file's attributes, to be handled by the system in much the same way as it handles, say, memory allocation. Of course, it might well be that the hardware manufacturers are doing very nicely, thank you, and don't really want to listen.*

What is left for the operating system to do ? Quite enough. Even if only one memory model is accessible, it has to be used, so support services are necessary. Basic memory management, which provides memory from the system resource as requested by other parts of the operating system, is standard in all but the tiniest systems. Whether or not memory should be provided piecemeal to processes is a decision which must be taken as part of the system design; early systems did not usually support such flexibility, and it does increase the possibility of poor performance in the forms of thrashing and deadlock. Because of the considerable advantages of flexible data structures and recursion and the

increasing size of memories, it is now common practice to make this service available to programmes, so that they can acquire memory at any time. (In Unix, the function **alloc( )** and its relations.) This policy gives some support to software implementing approximations to more elaborate memory models. It is obviously also necessary to reclaim memory when it is no longer needed so that it can be reused for some other purpose. More ambitious systems might also offer additional facilities such as shared memory services, or automatic garbage collection. It is this collection of services which is handled by the memory manager in most operating systems.

The mention of "shared memory" raises a point of nomenclature. In common speech, we use the word "share" in two different ways : two people sharing a house have equal access to its facilities at all times (subject to physical constraints which prevent two material objects occupying the same space at once), but two people sharing a pie each take separate portions. Processes share memory in both these senses, without the physical constraints. We think that the sense of the word "share" is adequately implied by the context whenever we have used it in our discussion, but if you have difficulties in interpretation ask yourself if you might have picked the wrong meaning.

SPECIFYING THE MEMORY MANAGER'S APPLICATION PROGRAMMER INTERFACE.

The memory manager is specified by describing the services which it must offer. We've just listed a set of possible services, but we found them essentially by guesswork. Can we be more systematic in our analysis ? If we can, then we should, because it gives us a better chance of getting a useful memory service. Here are some criteria which we must take into account.

First, the available memory must be allocated for use as required in such a way as to satisfy the criterion of a functional system – so memory must only be accessible to processes authorised to use it (to avoid interference between processes), it must be made available as requested (so that processes will not be prematurely stopped for lack of memory), and recovered when no longer needed (or the pool of available memory will dwindle, and processes will unexpectedly run out of memory).

Second, we should determine what sorts of request for memory must be satisfied. Memory may be requested by parts of the operating system (for example, to load new programmes) and by ordinary processes (for example, for data storage); different rules might be appropriate for requests from these two sources. It is convenient to express the definition as a set of functions, each describing some operation as seen by the programme requiring the service. This defines an interface between the memory manager and its clients; now, provided that the memory manager implements the interface as described, no one else need know how it works. Some examples follow. They should be taken as illustrations only; a great deal of work is needed to produce a satisfactory precise definition.

SOME NOTATION : In the examples, the description

get : size   area

identifies **get** *as a function which accepts a* size *as argument, and returns an* area *as its value.*

In all cases, one would also expect that a result descriptor would be returned in some way so that any noteworthy consequence of executing the function (such as an error) can be communicated to the calling environment.

The constraints on memory management depend on the nature of the environment. In particular, there is a difference in emphasis between operating systems primarily

designed to support a single process and those which provide multiprogramming facilities, in which many activities can reside in memory at the same time. If many processes must be accommodated, which is now the case in all common systems, the system must keep track of all memory use to ensure that no request for memory is unnecessarily refused, and that the processes do not interfere with each other. If only one process is present, the main concern is to identify the boundaries within which it must work, given which it can do what it wants. Systems of this sort used to be common, but are now mainly found in small special-purpose computer applications.

A system designed to run single processes, at its simplest, provides just those memory management functions which the process must have in order to use the space efficiently. Typical functions are :

lowerlimit : address (Find the first memory location available for use by the running process.)
upperlimit : address (Find the last memory location available for use by the running process.)
resetlowerlimit : address (Change the first available memory location.)
resetupperlimit : address (Change the last available memory location.)

The first two functions (which might in practice be implemented simply as reserved memory locations) provide the process with information it might need; the others permit the process to adjust the apparent size of the available area, typically so that it can leave material in memory but protect it from subsequent processes which would otherwise overwrite it. This mechanism is often used when special routines for handling non-standard input or output devices are required; they can be loaded into memory, then the bounds can be reset so that they are safe from later memory allocations. An example is the MS-DOS *Keep Process* function (sometimes still known by the name of an earlier version, *Terminate but Stay Resident*, or TSR).

In the case of a multiprogramming system, we are concerned with sharing out the memory between several processes which wish to use it. Here's a first guess at the functions we might need for a flat model :

The operating system might require :

get : size area (Acquire a chunk of memory of the specified size.)
release : area (Return a chunk of memory to the system pool.)
resize : area × size area (Extend or truncate an existing chunk.)

and for public use it might offer :

resize : size (Extend or truncate the address space. In a strict flat memory model, a running process knows only about a single simple address space.)

For comparison, here is a guess at the sort of memory management functions one might require in order to implement a stack model :

newstack : stack (Set up a new stack.)
demolish : stack (Destroy an unwanted stack.)
push : size × stack stack (Acquire a chunk of memory of the specified size, pushed onto the named stack.)
pop : stack chunk × stack (Remove the chunk from the top of the stack.)
index : stack × displacement chunk (Give access to the chunk at a specified position on the stack.)

For public use :

push, **pop**, and **index**, roughly as above but with the identity of the stack restricted to the current stack.

Returning now to the flat model, why do we choose those functions ? Because, at the lowest level, **get** and **release** cover all possible requirements. The only inherent structure of the memory is that imposed by the addressing machinery, which is to say that of an array of many functionally identical cells. All we can do is cut it up into slices. The **resize** function is not strictly necessary, and can create certain problems in implementation, but if it isn't available all processes begin by requesting the maximum area which they might possibly require, just in case the worst happens, and you can argue that you gain more from the function than you lose.

The public **resize** function allows a process to change only its own memory requirements; this is obviously a necessary restriction, and one example of the way in which an operating system presents a *virtual machine* to the processes which it runs.

This is not an exhaustive account of all desirable memory management functions – for example, later in this chapter we introduce one function which we shall eventually find very useful. That is some sort of **share** function which a process can use if it must share an area of memory with some other process. The functions we have mentioned are those which are purely to do with acquiring and releasing memory; others turn up in various contexts where there are other considerations as well, and we shall deal with those, if necessary, as they arise.

WHAT DIFFERENCE DOES IT MAKE ?

If you use a memory model which is appropriate to the work you're doing, it makes the work easier. You might still be able to write your programmes using a suitable model, but the software which prepares your programmes for execution then has to translate your requirements into terms of the model provided by the system – which is, as we've seen, inevitably that provided by the hardware. Consider what can happen when a programme which (like most programmes) uses procedures is prepared for execution in a flat memory. First you compile each of the pieces of the programme into a relocatable form; then you collect your pieces, and any other pieces you might want to include from the system or other software libraries; then you run some sort of linker to tie them together; then perhaps there's a loader to take care of final adjustments.

Why do you need the linker ? To map the relocatable pieces into different places in the flat memory. If you were using a system with a memory model more appropriate to the structure of the language and the way it's used, such as a stack memory, you might be able to do without it. If you use a very simple block-structured language in which parts of the programme can't see the insides of other parts, all the parts are autonomous, and all you need as a "linker" is a little table saying how to find each of the parts. The addresses (if there are any) within each part can start at zero without any confusion. It isn't in fact a lot harder to allow controlled access to the insides of the parts – the activation records – if you need it (as you do if you want to use global variables) but the address of any entity outside the local addressing range has to include both the name of the addressed part and an address within it.

There is a way to avoid most of the complications of linking and loading : stop worrying about separate procedures and compile the whole programme every time. Everything is then under control of the compiler, it can allocate addresses as it goes, and we end up with a single code file which can be loaded quite simply into memory and run. It works; very many compilers are implemented that way. Of course, it does mean that when anything goes wrong you have to recompile the whole lot, so that you save much of the cost of linking and loading at the expense of much more compiling, which is the really hard bit. The result is also less flexible; facilities such as subroutine libraries and interlanguage binding are harder to implement – and so is switching to some other manufacturer's compiler, which might or might not be relevant. You might not want these valuable facilities, but without an appropriate memory model implemented in the system you're that much less likely to have the choice.

Nevertheless, the world has so far given no sign of bending to our wishes, so we must make the best of what we have. By far the most common machine architecture presents us with a flat memory, and we'll assume that henceforth unless otherwise

specified. With the flat memory, we'll allow processes to own more than one chunk of memory where it makes sense, but we won't try to control or assist the process in how it uses the chunks; that's realistic in terms of today's operating systems.

SHARING MEMORY.

When drawing up our preliminary specification for processes, we didn't say much about storage except that there had to be some, and we didn't say anything at all about memory specifically, because we hadn't invented it yet. Now we have invented memory to cater for the practicalities of the programme's requirements for storage, we should ask whether it is useful for any other requirements. It turns out that for one purpose at least memory is very convenient; this is the requirement for communication between processes. This is an interesting development, because it runs contrary to many of our assumptions about processes' use of storage. In particular, we usually assume that a processes' memory must be carefully protected from access by any other process, in the interests of keeping our functional system pure; now we want to seek out ways in which processes can share memory in the interests of communication. The apparent conflict arises because memory itself is not a fundamental idea; it's just a convenient invention, and here we are using it for two quite different purposes.

Just as there are occasions when it is useful for processes to share files, so it is sometimes valuable for processes to have simultaneous access to common areas of memory. Sometimes sharing memory is just a convenient way to save memory space or to reduce the load on the system, but for some purposes it is an essential basis for communication between processes. In all cases, though, processes will wish to share semantically significant chunks – which is to say, segments.

Processes which share memory might or might not be active at the same time. Shared memory has become particularly important as a means of communication between concurrently running processes, but some sort of general access to data areas provided by the operating system has been common since monitor system days. Indeed, that sentence is almost a pun (unintended, we assure you), because a popular way to implement the shared memory was by an extension of the Fortran **common** area. Fortran **common** was intended to be a means of providing a global memory accessible to all units of a Fortran programme, and consisted simply of an area of memory which all programme units could address. For convenience, this was usually set aside at some standard memory location, typically the high address end; then, once the convention is established, it only needs a commitment from the operating system not to use the high end of memory and the **common** material left by one process becomes accessible to the next.

Whether or not this – or, for that matter, any sort of shared memory – is in all respects a good idea is quite a different question, and we shall address it later. It is much clearer that if we're going to share memory then just leaving a vacant space which any process can use however it wants is not a good way to do it; even within Fortran, **common** gave problems. That's why we suggested that a share function, administered by the operating system, would be appropriate.

Sharing memory is *convenient* when different processes use the same code or data. Code is the more common case, for many people might wish to use various public programmes at the same time, and they all have to use the operating system code. It might be *essential* when several processes cooperate to complete a task by working in parallel, for it might be the only means by which information can be exchanged between the processes at an adequate rate. We shall discuss the question further in the chapter *MEMORY MANAGEMENT : THE PROCESSES' VIEW.*

COMPARE :

Silberschatz and Galvin[INT4] : Section 8.6.

REFERENCES.

SUP17 : H.W. Lawson : "Salvation from system complexity", *IEEE Computer* **31#2**, 120-119 (sic) (February, 1998).

EXE8 : R. Bernhard : "Bubbles take on disks", *IEEE Spectrum* **17#5**, 30-33 (May, 1980) (quoted in *Files and databases : an introduction* (P.D. Smith, G.M. Barnes (Addison-Wesley, 1987), page 33).

EXE9 : R.W. Mitchell : "Content addressable file store", *Proceedings of the Online Database Technology Conference, London* (April, 1976) (quoted in *Files and databases : an introduction* (P.D. Smith, G.M Barnes (Addison-Wesley, 1987), page 34).

EXE12 : C.B. Kreitzberg, B. Shneiderman : *Fortran programming a spiral approach* (Harcourt Brace Jovanovich, 1975), page 288.

EXE13 : M.A. Auslander, H.R. Strong : "Systematic recursion removal", *Comm.ACM* **21**, 127 (1978).

EXE14 : D.W. Barron : *Recursive techniques in programming* (Macdonald/Elsevier, 1968), Preface.

EXE15 : J. McCarthy : "Recursive functions of symbolic expressions and their computation by machine", *Comm.ACM* **3**, 184 (1960), reprinted in *Programming Systems and Languages* (ed. S. Rosen, McGraw-Hill, 1967).

IMP13 : S. Ahuja, N. Carriero, D. Gelernter : "Linda and friends", *IEEE Computer* **19#8**, 26 (August 1986).

QUESTIONS.

How could you implement various memory models on dissimilar hardware memory devices ?

Why didn't we propose public **get** and **release** functions for the flat memory manager ?

In a multiprogramming system which offers a flat memory model to processes, is it possible, or sensible, to share memory between two processes by simply overlapping their memory areas ? What is the minimum requirement for sharing memory between three processes ?

Is there any sense in providing a **share** function in a system which runs only one process at a time ? (HINT : yes.)

What sort of memory model is appropriate to an overlay system ? to Prolog ? to a functional language ? to a spreadsheet ?

What sorts of memory model are implied by these features of high-level languages : Pascal scope rules; Pascal pointers; recursion; array dimensions determined during execution; strings ?