Computer Science 415.340

Operating systems

HOW TO DO IT

IMPLICATIONS OF MEMORY

What must we have in order to implement a memory manager ? The major implications of these requirements amount to a specification of hardware functions. It is unfortunate that in most cases we can't order specific hardware functions; the hardware is given, and there's often little enough evidence that it's been planned to suit the convenience of anyone but the hardware designers.

Recall that we have arrived here after a long process of analysis which began with the requirements of people who use computers. Whether or not the specific details of our arguments are correct is beside the point; the important thing is that some such process is possible, and we end up with a hardware specification which we believe will support an operating system designed to give good service. The hardware designers then tell us, after the manner of Henry Ford, that we can have any sort of memory we like so long as it's flat. (In fact, our arguments might very well be wrong, as we've compromised all the way along by accepting that we *have to* cope with a flat memory.)

That might be just a little cruel to the hardware designers, who are doubtless compassionate and well meaning people; but here's the introduction to a fairly recent paper[EXE22] *:*

The computer designer's objective is to produce a product that will succeed in the marketplace. This requires optimal use of available hardware and software technologies. If bottlenecks exist that prevent the exploitation of these technologies, they must be identified and eliminated.

To accomplish this objective, the computer designer has three choices:

(1) Do no analysis whatsoever, just design the machine from some cosmic understanding of what needs to be optimized.
(2) Do analysis based on analytic models, such as Petri nets and Markov chains.
(3) Do analysis based on experimental techniques, for example, trace-driven simulation of benchmark programs or hardware monitoring of an engineering prototype.

The first option can be rejected out of hand, although it is unfortunately still used. There are well-known examples of products that were designed without proper analysis that have not been successful in the marketplace. There is probably a larger number of lesser-known projects that were fortunately canceled before reaching the marketplace after the appropriate analysis suggested that cancelation made sense.

It's true that there's a reference to benchmark programmes – but those are commonly designed to test the performance of average software on existing machines, and therefore already embody the assumptions people make when using existing machines.

However, for what it's worth, here's an attempt to list our requirements. It isn't guaranteed to be complete or accurate; think critically about it. As always, we assume we begin with a flat memory implemented in hardware.

A linear array of addressable memory elements.

ADDRESSING.

To give each process an address space starting at zero, we need *base registers*.

If we are concerned about segments, then for protection against addressing errors, we need *limit* (or *range) registers*.

To permit flexible use of memory, we require *address mapping*; the hardware should be able to manage addressing via page or segment tables automatically.

For segmented memory management, hardware help with *list management* is useful, but not essential.

VIRTUAL MEMORY.

The basic requirement for a virtual memory is the machinery for recognising an *addressing fault*. The addressing table must include a *present bit*, which is inspected by the hardware on every reference to memory; if the address sought is not resident in memory, the virtual memory system must be brought into action.

That might be all there is to say; to some extent, it depends on your analysis of the structure of the system. Is memory management a single activity, or should we regard managing internal memory and virtual memory as essentially separate tasks ? If we favour the separate tasks model, we could take the position that the rest is up to the hardware to deal with – once the hardware has detected a reference to an absent address, the hardware had better do something about it. With that point of view, the virtual memory software becomes a service called on by the hardware. This is perhaps the better logical structure, as it then becomes very clear how to incorporate a hardware implementation of virtual memory. In effect, the higher level memory manager is a server providing memory on demand to processes which need it, drawing from an infinite pool; then a lower level manager supports the infinite (well, very large) memory abstraction by implementing virtual memory.

A more conventional view is that the virtual memory software is just another part of the memory manager, and the hardware functions only as a sort of switch which sets it in operation. This view concentrates more on what the parts of the systems do than on the nature of their interrelationships, and emphasises the close relationship between the two components, particularly obvious in their common concern with the addressing tables.

Either way, the virtual memory software is a part of the operating system, so we have to talk about it somewhere.

To implement a good memory replacement strategy, hardware help of several sorts is useful. The simplest is the *dirty bit*, a bit in the addressing tables marking a page or segment as having been changed. Other more elaborate schemes might use several dirty bits, or maintain a chain of pages or segments to implement a least-recently-used queue.

REFERENCE.

EXE22 : Y.N. Patt : "Experimental research in computer architecture", *IEEE Computer* **24#1**, 14 (January 1991).