3 Memory management

Originally, several programs to be run on a single machine would have to be *overlaid*: each program would be given exclusive use of the memory, and would overwrite it predecessor when it was loaded and run. Whilst this makes serial batch processing possible, it does not support multiprogramming.

Memory management sets out to share memory space between processes, both protecting them from each other and allowing them to share when necessary. It should also make the addressing of memory as transparent as possible, disguising the existence of any memory management.

3.1 Partitioning

Straightforward first free-space allocation with a single flat address space requires each program to declare a size of memory which it needs, and loads programs into memory just after the end of the space already in use. This fails to provide any protection, makes no distinction between the kernel and users processes, makes no distinction between data and program, and worse than that allows holes to appear when processes terminate.

All memory (or rather address-space) allocation has to solve the same problems; the sizes and speeds involved are different, so the solutions are not the same. Recording which space is free might use a data structure (like a bitmap) which records the status of each component of the space; or it might keep a list of 'holes'. The bitmap makes it expensive to identify the sizes of holes, and find the holes of a given size; the list probably needs to be organised by address to make it possible to merge adjacent holes, but probably needs to be accessed by size for efficient allocation!

Better than this would be to *partition* the memory into areas that can each be allocated one to a process. Hardware support might be provided to prevent interference between processes by restricting each (user) process to accessing only its own partition. One such scheme would involve a *base register* and a *limit register* for each currently accessible partition, at the cost of performing an addition and a test on each memory access to determine whether the access was legal. Partitions might be *swapped*: the relevant parts of their data copied out of memory to disk, and resumed later, allowing more processes to be running that will fit in memory, although at a considerable cost in the time taken by the disk traffic.

However, if the partitions are of a fixed size space is wasted when smaller processes have to be fitted into larger partitions (*internal fragmentation*) and process can fail to find space to run even when there is enough unused memory. If the partitions are of variable size, repeated allocation and release of partitions can cause *holes* to form (*external fragmentation*) with the same effect. There are also policy

issues to be decided about which partition to use for a given process. Possible strategies include

First fit use first hole big enough;

Best fit search list for smallest hole big enough;

Worst fit search list for largest hole available;

Quick fit separate lists of commonly requested sizes.

Each has its advantages and its disadvantages.

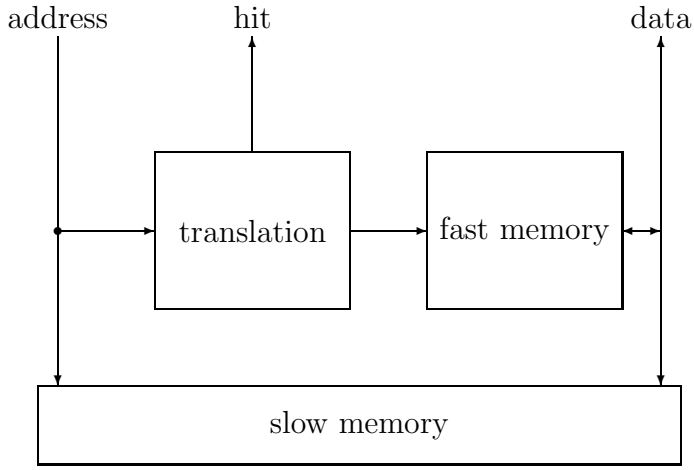
3.2 Caching

Memory design generally is plagued by the trade-off between speed and cost. Faster memory is more expensive (in physical resources, in power consumed, and therefore in money) so has to be smaller. Slower memory is cheaper, but performance would be lowered. An obvious solution is to provide a small amount of very fast memory backed up by quantities of slower memory, but to ensure that the most frequent accesses are always to fast memory. That way the average access time is very short, but the average cost is very low.

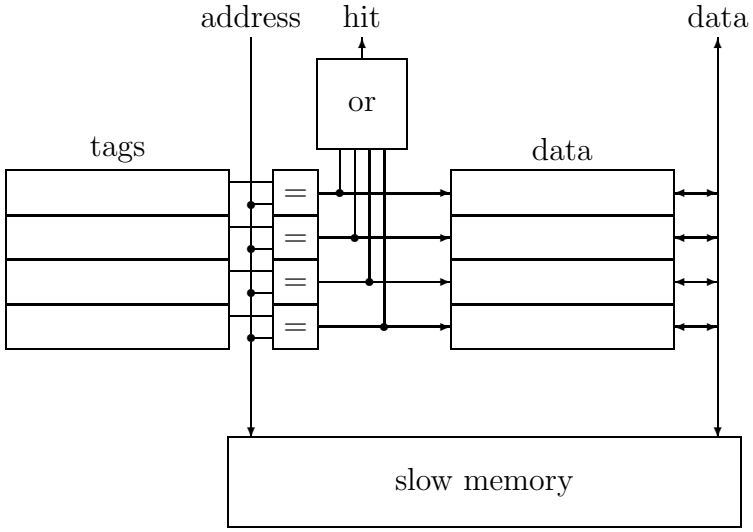
How to ensure a high hit rate? One scheme would be to decide in advance of running a program which of its data should be in fast memory. In essence this is what happens with processor registers against main memory: a compiler decides on the basis of local static analysis of the code which values should be in registers and which need to be pushed out to main memory. The consequence is code which fits a particular processor design.

Having the same inhomogeneity in main memory would be difficult to deal with, and would lead to programs having to be recompiled to fit different memories, as well as different processors. A better scheme is to explicit *dynamically* the locality of access in programs and use *caching*. Accesses to memory tend to be clustered, both is space and in time, so that locations that have recently been used, (and near those that have recently been used) are likely to be used again. A cache for a large slow memory keeps a copy of a small number of recently accessed areas of the address space, and intercepts accesses to those areas. It can serve those requests very quickly, and only hands off to main memory accesses to locations not kept in the cache. The result is an average performance which depends on the weighted average of the (short) access time c of the cache and the longer access time m of the main memory. The weighting is by the *hit rate* of the cache: if a proportion h of all accesses hit the cache, the average access time is $hc + (1 - h)m$. The higher the hit rate, the nearer this average is to c. In realistic cases, m/c will be about 10 or 100.

To implement a cache you need, in addition to the small fast memory, a way of translating memory addresses into the corresponding cache addresses. This translation has to be fast because it happens on every cache hit, but it has to have a size comparable to the cache, rather than the memory.



The translation is usually implemented by a contents-addressable memory that matches addresses accessed against the addresses (*tags*) which are cached. A *cache line* consists of one block of data, its address (*tag*), and the extra hardware needed to implement a CAM, in particular a tag comparator.



This analysis omits the work involved in keeping the cache up-to-date. If the contents of the cache are allowed to age, the hit rate will fall. Instead every cache miss, as well as resorting to main memory, brings the missed location into the cache. To do that a resident line needs to be evicted. Which victim to choose?

There is only a time about m to do this without impact on performance, so this has to be quite fast; conversely the penalty for inadvertently evicting something which will probably be accessed again soon is quite low, so it does not in practice matter all that much what you do.

3.3 Virtual memory

Virtual memory was invented by Tom Kilburn[3] for the Atlas computer built in Manchester (1958-1962) and Burroughs machines in the 1960s.

Virtual memory is often presented as a solution to the problem of fitting unreasonably large processes into reasonably sized memory by keeping on disk those parts of a process that do not fit and which are not in current use. Think of it the other way: virtual memory uses the disk or other backing store as the 'main memory' of the computer. Accesses to that would be unreasonably slow, so the real physical main memory is used as a cache for it.

So virtual memory is just a cache, no more to be said. Except that in this case the miss penalty is absolutely enormous. Accesses times for bulk RAM (c in this case) are measured in tens of nanoseconds; access times for rotating magnetic oxide disks (m) are measured in tens of milliseconds. This time m/c is 10^6 or more. It pays to take extreme care to avoid unnecessary misses.

3.4 Paging

The first layer of a VM system implements the translation part of the cache. For small enough virtual address spaces the tags and comparators are replaced by a table.

The address space is divided into *pages*: blocks of addresses each of which will fit one cache line. Bigger blocks improve the hit rate by increasing the number of 'nearby' accesses, and reduce the overhead of setting up a disk transfer.

If the block size is too big, the hit rate is damaged because (for a given amount of RAM) the number of lines in the cache goes down. Blocks that are too big also take longer to transfer, increasing the miss penalty.

Paged memory is divided into *frames*, each capable of holding a single block, together with a *page table* which maps the number of a page to the number of the frame in which it will be found. This is potentially much bigger, but also faster, than a CAM. A CAM needs one tag and one comparator per frame; but a page table needs one frame address per page. The number of frames depends on the

[3]Tom Kilburn CBE, FRS (1921–2001), published in *One-Level Storage System*, T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner, IRE Trans. Electronic Computers April 1962

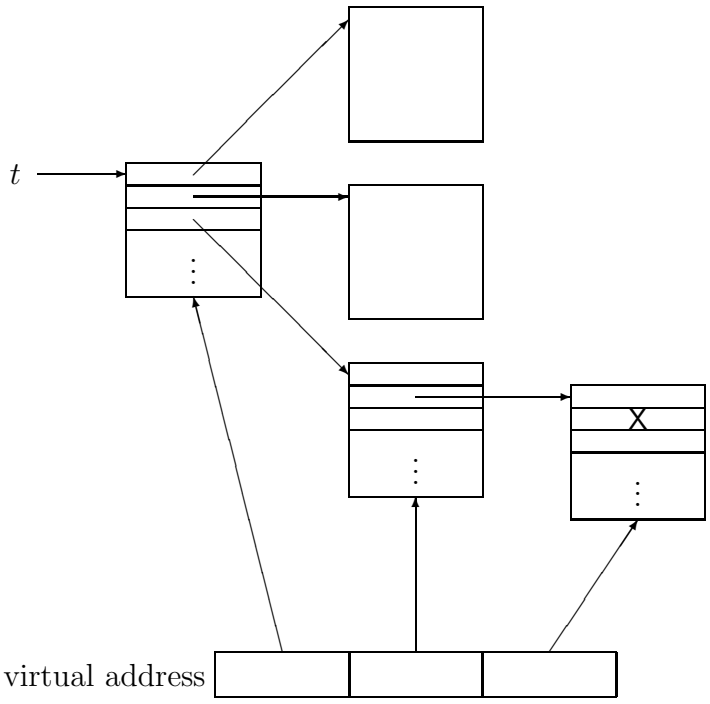
size of the paged RAM; the number of pages depends on the size of the virtual address space.

Suppose the block size is b, and the page table is an array at t. To address location a you need to find location $a \bmod b$ in page a/b. This page is stored in frame $[t + (a/b)]$ so the addressed location is $[t + (a/b)] \times b + (a \bmod b)$. Of course b will be a power of two, and the arithmetic consists of dividing a word up into parts and putting it back together.

Cache hits have to be fast: every access to memory now goes through the page table as well as the real physical main memory. One possibility would be to keep the page table in registers; however that involves having a large number of registers, and costs the processor additional time on a context switch, when the page table for one process has to be swapped for that of another.

Another possibility would be to put the page table in memory, with a single register identifying the base address of the table. This helps context switching, but would appear to double the cost of every memory access.

Worse than that, as the size of the virtual address space grows so does the size of the page table. However a process will likely only use a small part of a huge virtual address space. Two-level page tables split the page number into two parts:



Not all of the second-level page tables need be present, but the cost is now three physical memory accesses per virtual memory access. A possible strategy is to cache the translation from page number to frame number; a cache for this is called a *translation lookaside buffer*.

As the virtual address space grows, even a multi-level page table would become too expensive. If there are p bits in a physical address, v bits in a virtual address, and b bits to select a location within a page, the page table occupies a space which grows as $p2^{v-b}$ which is exponential in v. The only rational solution is to keep an *inverted page table* which maps frames to pages. The inverted table is only $v2^{p-b}$ in size, but looking up a page in this would involve a long (software) search. A *tlb* can make accesses which hit the *tlb* efficient; misses can be sped up by keeping the inverted page table in a hash table.

3.5 Paging with virtual memory

When the number of pages in use exceeds the number of frames, some will need to be absent from main memory. The page table must contain both frame addresses for pages present in memory, and disk addresses for pages absent from memory but held on disk.

When an access is made to a location in an absent page there is a cache miss: these are called *page faults*. Since the disk access part of the miss is going to be catastrophically expensive anyway, there is no point providing much hardware support for handling the miss directly. instead a page fault usually causes a call to an interrupt routine to handle the miss. That routine starts bringing in the missing page, and (eventually) resumes execution. In the meanwhile there is probably time to run other processes. One thing which the hardware does need to support is the resumption of the instruction which caused the page fault. Particularly in the case of processors which can make several memory accesses in one instruction it is necessary to be able to restart an instruction, probably by ensuring that the state can be returned to that at the start of the instruction.

3.5.1 Handling page faults

On a page fault, before the current process can continue, the missing page must be fetched from disk into RAM. Doing this may require an existing page to be evicted to free a page frame for the new page.

Since writes to disk are so expensive, not every write to virtual memory can update the disk, so writes are also cached in the RAM and the pages present in RAM may be *dirty*. Dirty pages are ones that have been written while they have been present, and are therefore both different from and more recent than their disk copies. If a dirty page is evicted it must first be written back to disk.

Once the page which caused the fault is present, the process which caused the fault can be made ready, and the instruction which caused the fault restarted.

Acceptable performance demands that hardware support is absolutely necessary for handling virtual memory accesses, delivering the results of hits, and pre-

empting instructions on misses. In contrast, page eviction and replacement are relatively leisurely activities which only happen when a disk transfer will happen anyway, and can be implemented in code. There must of course be a guarantee that this code will always be present in RAM!

3.5.2 Page replacement policy

The penalty for a page fault is so great that the page replacement policy can afford to expend considerable time on minimising page faults.

A complete analysis of the future pattern of memory accesses would allow an optimal choice, but of course that is in general not possible. Although you would like to choose on the basis of future accesses, all you can know is the pattern of accesses in the past.[4]

Not Required for Longest would be good, but the pattern of accesses will depend on what a program does. If you could know that, in general, you would not need to run the program.

Least Frequently Used would be a good approximation, at least assuming that the pattern of past behaviour is any guide to future accesses.

Least Recently Used is a good compromise to aim for, but requires hardware support to maintain (on page hits) a data structure that records how recently each page frame was accessed.

Not Recently Used is a common approximation to LRU, which requires less support. A single bit per frame records whether that frame has been accessed since a clock tick; all these bits are cleared on a timer interrupt. Eviction then chooses in some way a page that has not been touched in the current period.

The clock algorithm goes around the page frames in turn, like a clock hand. If the currently selected page has not been accessed recently it is evicted; if it has been accessed the access bit is cleared and the clock hand moves on to the next page and repeats the check.

In the absence of all else, *random* choice of pages to be evicted is not so good, but neither is it impossibly bad.

It is tempting to try something really simple like *First In First Out* which evicts the page that has been in its frame for longest. This needs no hit-time data maintenance, but has quite bad properties. In particular it would be good if the eviction policy was *monotonic*. A monotonic policy is one which causes fewer page faults if it is given more page frames to use. FIFO is not even monotonic, whereas LRU is.

[4] Søren Kierkegaard: Livet forstås baglæns – men må leves forlæns.

3.6 Problems with virtual memory

The *working set* of a process over a given interval of time is the collection of pages which it is accesses in that time. Less formally, you can talk about the 'current' working set so long as it is stable over short time. If the working set of a process fits in RAM, and the working set changes slowly over time, then the page fault rate is low and performance of virtual memory is good.

If the available memory is smaller than the working set of a process, even only slightly smaller, this causes *thrashing*. Repeatedly pages are evicted just before they are needed, and very large number of faults happen. In the extreme, almost every memory access causes a page fault, and the machine behaves as though every access to memory were an access to disk. The constant by which this slows a program down is so great that (apart from the noise from the disk) it is barely distinguishable from a program that has stopped.

3.7 Segmentation

Virtual memory offers the freedom to be profligate with address space. Sixty-four bit address spaces (and larger) are not for filling! Accessing every one of 2^{64} locations in an address space at a rate of one location per nanosecond would take almost six hundred years. One possibility opened up by large address spaces is to allocate some (higher order) address bits by logical intent rather than physical need. Some bits might be allocated to the process identifier, but the address space of a process might equally be segmented into logical sub-spaces. There might be segments for data and segments for code; each procedure or library might be given its own segment(s), simplifying the business of linking code (or rather delaying the work until run time).

Exercises

3.1 A particular program is run on a very small virtual memory system which has five pages. The program accesses them in the order:

0 1 2 3 0 1 4 0 1 2 3 4

starting with an empty page table. What happens if the VM system is able to use:

1. three page frames, with FIFO replacement;
2. four page frames, with FIFO replacement;
3. three page frames, with LRU replacement;

4. four page frames, with LRU replacement;
5. four page frames, with optimal replacement?

3.2 Prove that LRU page replacement is monotonic.

3.3 Suppose a, b and c are three arrays of integers in a C program. Why, when N is large, might

```
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        c[i][j] = a[i][j] + b[i][j];
```

take a thousand times longer than

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i][j] = a[i][j] + b[i][j];
```

on the same machine?

Hint: this is a question about virtual memory.

3.4 The system call *fork* creates a new process with (initially) the same values in all locations in its address space. On page 10, this is described as though it copies the contents of the memory of the parent. The child is quite likely to overwrite itself with an *exec* call almost immediately.

How might virtual memory be modified and exploited to make this copying cheaper, and what are the consequences for the implementation of the virtual memory system?