Chapter VIII
The memory subsystem

VIII.1 Memory hierarchy

The programmer's view of the computer memory is that it is a flat, large storage space. Obviously we want it to be as fast as possible, since instruction fetch and data access take a significant part of a processor's cycle.

Unfortunately this is not possible with any known technology. We can make very fast memories by using expensive, static RAM circuits, but when we put too many of them together to create a large memory, its speed drops as the memory size increases. Not even supercomputers, where high cost is not a problem, can have a large, flat and fast memory.

At the same time we have technologies that can provide us with large, but relatively slow, semiconductor memory. This memory is called Dynamic RAM and each bit is stored in a single transistor in comparison with 6 for 1 bit of SRAM. We also have huge, but very slow compared to a processor, magnetic memory available to us, in the form of hard disks.

Locality

The principle of *locality* of memory references in computer programs comes handy here. It says that if a program accesses a particular memory address, it is likely that the next few accesses will be to nearby addresses (*spatial locality*), and also that the same address is likely to be accessed again within a short time (*temporal locality*). This is true for instruction fetches, and also for data reads and writes.

We can take advantage of the locality of references, to create a hierarchical memory with multiple levels of memory of different speed and size (and technology). At the top of the hierarchy, we have a fast but small memory which is directly connected to the processor. The memory sizes increase (but the speed drops) as we move to lower levels of the hierarchy further away from the processor. At the bottom, we have slow magnetic memory. The data held at a level close to the processor are a subset of the data held at any level further away. Assuming that the data is transfered between memory levels in a way that ensures that most processor accesses are handled at the top level, the whole memory system will appear as fast as the top memory level and as big as the bottom level.

Block

The minimum amount of data transfered between two adjacent memory levels is called a *block* or *line*. Although this could be as small as one word, the spatial locality principle suggests that larger blocks are better.

Hits, misses and rates

If the data requested by the processor is found at a memory level, we say that

we have a *hit* at that level. If not, we have a *miss*; the request is propagated to the next level down and the block containing the requested data is copied at this level when the data is found. This ensures that the next time this (or nearby) data is accessed there will be a hit at this level. The fraction of memory references found at a level is called *hit rate*.

Computer memory systems typically consist of a four-level hierarchy: Registers — Cache — Main Memory — Backing Store (i.e. disk). Note that data is moved between the cache, main memory and backing store transparently, but data movement between registers and the rest of memory is explicitly under the control of the program. It is up to the compiler to decide which data items should be moved to registers, and when, and the compiler must compile into the code the relevant load and store instructions.

In practice, many computers introduce another level into the hierarchy: they have *two* levels of cache; a small (perhaps 64Kbyte), very fast cache on the processor chip (called the *first level cache*), and a larger (perhaps 512Kbyte) *second level cache*, either on the processor chip or on separate chips, and intermediate in speed between the on-chip cache and the main memory.

VIII.2 Cache basics

The main issue in designing a cache is how to determine whether a data item is present in the cache and where it is stored.

Tags

As the storage space of a cache is much smaller than that of the main memory, each cache location can hold the contents of a number of different memory locations. Since data items are identified by their memory address, in order to ensure that a specific location in the cache indeed holds the required data, the address must be stored in a special field at the cache location together with the data. This special field is called the *tag*.

Valid bits

When a cache location is unoccupied, the tag field could still match to a requested memory address causing the program to malfunction. Therefore, a single bit *valid* field is also added in each location signifying whether the data held there is valid or not.

The cache organisation described above is called *fully-associative*. In order to find the referenced data, we have to compare the tag field of every cache location to the memory address requested by the processor. It is quite expensive to implement these parallel comparisons in circuits, therefore full-associativity is practical only for caches with a very small number of entries.

Direct-mapped caches are at the opposite extreme of cache organisation. Each data item can be stored only at one cache location. The location is determined by the memory address and the size of the cache as follows:
cache location = (memory address) MOD (cache size)

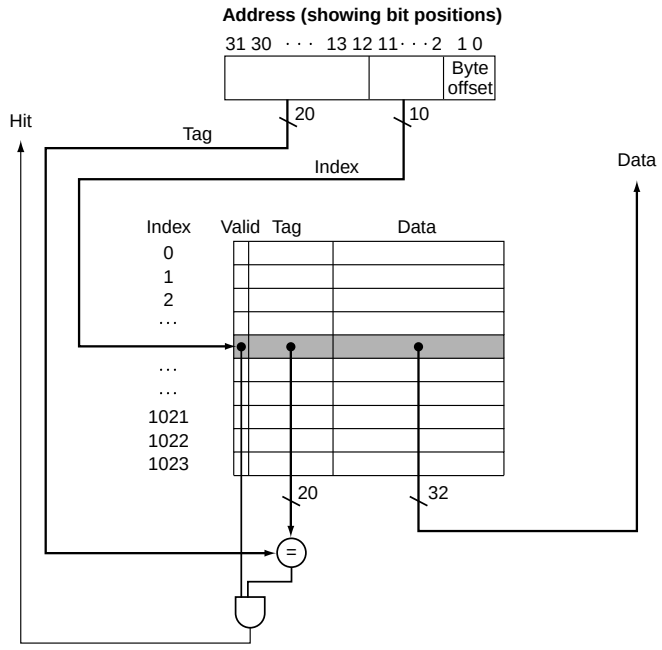

Figure VIII.1: Example of direct-mapped cache.

which is equivalent to indexing the cache with some of the low bits of the memory address. Because part of the address has already been used to identify the cache location, the tag field needs to hold only the 'unused' part of the address, therefore the total amount of storage required in the cache is lower in comparison to fully-associative caches.

Figure VIII.1 (fig 7.7 in P&H 3/e) shows a 1Kword direct-mapped cache and the block diagram of the search mechanism: Bits 11-2 of the address are used as an index to read the cache line where the requested word may be stored. The tag field of the cache location is compared with bits 31-12 of the address and if they are equal and the valid bit is set, the access is declared a hit and the data can be used by the processor.

Write accesses to memory, which are much less frequent than reads, are more complicated, and different caches handle them in different ways. In a *write-through cache*, if the word being written to is in the cache, both the copy in the cache and the copy in main memory are written to at the same time. This obviously takes as long as a main memory access, but is simpler to implement than *write-back caches*, which store a block into the main memory only when it is thrown out of the cache. Caches also vary in their behaviour if a write access *misses*: some caches load the missing block into the cache; others do not.

VIII.3 Virtual memory

The principle of virtual memory is that those parts of the code and data areas of each active process which are actually being used are kept in main memory, while those parts not being accessed during the current phase of execution are kept on backing store, i.e. disk. The operation of the virtual memory system is *transparent* to the programmer — the transfer of parts of the program between main memory and disk happens automatically under control of the operating system, as needed. There are two main benefits associated with virtual memory: programs can run on computers with less memory than the total code and data of the program, and the main memory is shared efficiently and safely among different processes.

The compiler compiles the program assuming that the machine has a very large memory, all available to the program — the *virtual address space*. All addresses in the program refer to this space, i.e. are virtual addresses, typically 32 bits long. The program in execution uses virtual addresses to refer to data and to instructions — the program counter, stack pointer and pointers to other data items contain virtual addresses.

The real memory space of the computer is usually smaller than the virtual memory. The operating system keeps some chunks of the process code and data in main memory, leaving the rest on disk. The key to efficiency here is to keep the active parts of the program in main memory. This is very similar to the operation of caches.

Address translation

All addresses output by the processor as it fetches instructions or accesses data must be translated from virtual addresses to the real addresses (called *physical addresses*) of the locations at which the instructions or data currently lie. This is called *address translation*, and is done by a piece of hardware between the processor and memory called a *memory management unit* (MMU). The address translation can happen before or after accessing the cache; there are advantages and disadvantages to both approaches, but this is out of the scope of Inf2C.

Relocation

When a process is loaded into memory in order to run, the place it is loaded depends on what other processes are already active. The compiler cannot know in advance where in memory the process will be loaded. The address at which each instruction and each data item reside in memory will be different each time the process is loaded, and that leads to difficulties with instructions that contain absolute addresses, such as absolute jumps. Any instruction that refers to another instruction or to a data item using its absolute address (as opposed to a relative offset) will need to be modified in some way, to take account of the actual position in memory the process is loaded at. This modification is known as *relocation*, and is inconvenient for systems which do not use virtual memory.

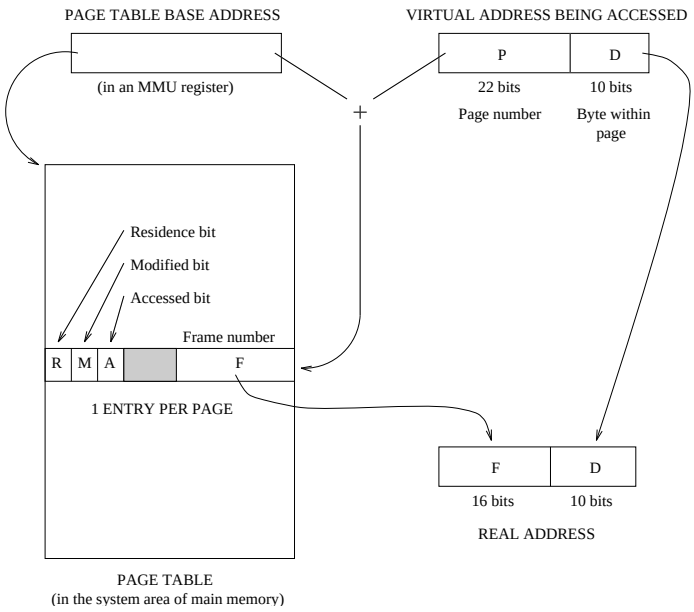

Figure VIII.2: Address translation.

Virtual memory removes the need for relocation, as all addresses in the program are virtual and guaranteed to be unique to that program.

Paging

The virtual memory space is divided into a large number of equally sized chunks called *pages*. Page sizes vary between computer systems, but typical sizes are 1K and 4K bytes, and the 32 bit virtual address therefore consists of two fields, the *page number* (the most significant 22 bits) and the *page offset*, the number of the byte within the page (the least significant 10 bits for 1K byte pages). The main memory of the machine is also divided up into chunks, of the same size as pages, called *page frames*, so the physical address, which might be 26 bits long, will consist of a 16 bit frame number and a 10 bit byte address within the frame. The OS maintains the currently needed parts of the program code and data areas in physical memory by loading the required pages of the program's virtual memory space into a set of (not necessarily contiguous) page frames in physical memory.

VIII.3.1 Address translation

Figure VIII.2 shows the address translation mechanism in the memory management unit (MMU). The *page table* contains information about each page in the virtual memory space used by the running process. The table itself is in physical memory, in that part of the memory reserved for use by the operating system. In the MMU there is a register containing the (physical) address of the

start of the page table. When a virtual address is presented to the MMU for translation, the page number `P` is added to the page table base address, to access a table entry describing the location of the page. The `F` field of that table entry is simply the number of the frame in physical memory containing the page, and so if it is concatenated with the lower 10 bits (`D`) of the virtual address, we get the corresponding physical memory address.

What happens if the process tries to access a page which is not held in main memory, but is on disk? There is a bit in the page table entry, the `R` bit, which is set only if the page is in main memory. If the `R` bit for the page being accessed is not set, a *page fault* exception occurs, interrupting the process and invoking the OS page fault handler. The handler fetches the missing page from disk and loads it into an unused frame in main memory, then sets the `R` bit and the `F` field in the page table entry. The interrupted process can then resume. This technique of loading pages into memory when they are first accessed is called *demand paging*.

Page replacement

Each process in a multi-tasking system has allocated to it a certain fraction of all the page frames in the physical memory space. What happens if a page fault occurs and all the process's page frames are already in use? The OS must pick one of the page frames for re-use, and load the new page on top of the page currently in that frame, replacing it. If the replaced page is a data page, and any location in the page has been written to since the page was loaded into the frame, the old page must be written out to disk, updating the copy on disk, before the new page is loaded over it. A bit in the page table entry for the page, the `M` (Modified, alternatively called *dirty*) bit, is set if a write has been performed to the page since it was loaded into the frame — if the `M` bit is unset, the page can be overwritten without being saved to disk, as the previous copy on disk is still valid.

If the page which was overwritten is accessed again by the process, it has to be reloaded from disk (into any available frame), so ideally the OS should choose pages for replacement which will not be accessed again for a long time. Using the locality principle, the OS approximates this criterion by replacing pages which have not been accessed recently. This is achieved by using a further bit in the page table entry for each page, the `A` (Accessed) bit[1]. Whenever any address in a page is accessed, the `A` bit is set. All the `A` bits in the page table are reset periodically by the OS. Thus any page which has an unset `A` bit has not been used since the last reset of the `A` bits, and is a good candidate for replacement with the incoming page.

[1] Called *reference* or *use* bit in P&H.

VIII.3.2 Efficiency in address translation

Address translation in a paged virtual memory system involves an access to the page table for every access by the process to memory. The page tables themselves are held in the main memory of the machine, so whenever a process accesses memory, the system must make two memory accesses, one for the page table, and one to the process's requested location. Since main memory speed is a major limiting factor determining the speed of a computer, this would slow the entire computer down by a factor of up to two.

To solve this problem, the MMU contains a fast memory which holds the necessary information from the page table entries of the most recently accessed pages. This fast memory is called the *translation lookaside buffer* (TLB) and typically holds 16 to 512 entries. Each entry contains the page number of the virtual page, the `M` (Modified) bit from the page table entry for that page, and the frame number of the memory frame holding the page. When the processor outputs a virtual memory address to the MMU, the page number is simultaneously compared against the page numbers of all the TLB entries. If one of the TLB entries matches the requested virtual address, the frame number of the page is available immediately from that entry, and address translation is very fast. If there is no match in the TLB, the page table lookup must proceed by accessing the 'full' page table, kept in main memory, and the information is loaded into the TLB, replacing a TLB entry which has not been used recently[2].

[2]Hardware associated with the TLB can keep track of which entries have been used most recently.