Virtual Memory

In the early days of computers, memories were small and expensive.
The IBM 650, the leading scientific computer of the late 1950s, had only 2000 words of memory.

One of the first ALGOL 60 compilers was written for a computer with only 1024 words of memory.

In those days the programmer spent a lot of time trying to squeeze programs into the tiny memory.

The traditional solution to this problem was to use a secondary memory, such as disk.

The programmer divided the program up into a number of pieces, called **overlays**, each of which could fit in the memory.

To run the program, the first overlay was brought in to memory and its code ran for a while. It then read in the next overlay and that code ran for a while, and so on. The programmer was responsible for breaking the program into overlays, deciding where in the secondary memory each overlay was to be kept, arranging for the transport of overlays between main memory and secondary memory, and in general managing the whole overlay process without any help from the computer.

Virtual Memory

In 1961 a group of researchers in Manchester, England, proposed a method for performing the overlay process automatically, without the programmer even knowing that it was happening.

This method, now called **virtual memory**, had the obvious advantage of freeing the programmer from a lot of annoying bookkeeping.

By the early 1970s virtual memory had become available on most computers.

Now even single-chip computers have highly sophisticated virtual memory systems.

Virtual Memory – Paging

Paging

The idea put forth by the Manchester group was to separate the concepts of address space and memory locations.

Consider, as an example, a typical computer of that era, which might have had a 16-bit address field in its instructions and 4096 words of memory.

A program on this computer could potentially address 65536 words of memory ($65536 = 2^{16}$) .

Therefore, the **address space** for this computer consists of the addresses 0, 1, 2, ..., 65535.

As in our example, many computers of the day had fewer than 65535 words of memory.

218

Virtual Memory – Paging

Before virtual memory was invented, people programming this example machine would have made a distinction between the addresses below 4096 and those equal to or above 4096.

Although rarely stated in so many words, these two parts were regarded as the **useful address space** and the **useless address space** (useless because they did not correspond to actual memory locations).

Prior to the contribution of the Manchester group, people did not make a distinction between address space and memory locations, because the hardware enforced a one-to-one correspondence between them.

Separating the address space and the memory locations means:

At any instant of time, 4096 locations could be directly accessed, but they need not correspond to addresses 0 to 4095.

219

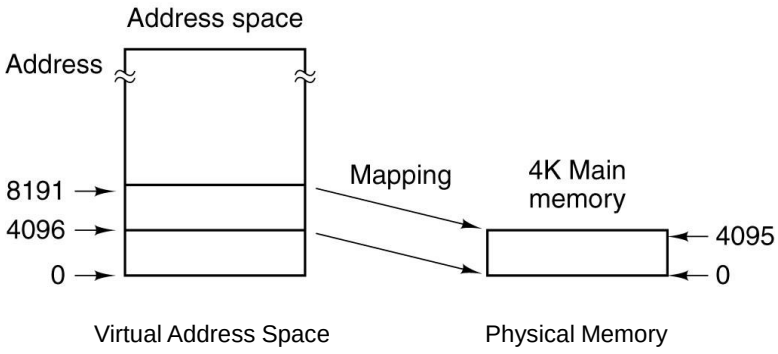
Virtual Memory – Paging

We could, for example, ‘‘tell’’ the computer that henceforth whenever address 4096 is referenced, the memory word at location 0 is to be used. Whenever address 4097 is referenced, the memory word at location 1 is to be used; whenever address 8191 is referenced, the memory word at location 4095 is to be used, and so forth.

In other words, we have defined a mapping from the address space onto the actual memory locations, as shown below.

220

Virtual Memory - Paging



A mapping in which virtual addresses 4096 to 8191 are mapped onto memory locations 0 to 4095.

221

Virtual Memory – Paging

A 4-KB machine without virtual memory simply has a fixed mapping between the addresses 0 to 4095 and the 4096 words of memory.

An interesting question is: ‘‘What happens if a program branches to an address between 8192 and 12287?’’

On a machine without virtual memory, the program generates an error and terminates.

On a machine with virtual memory, the following sequence of steps occur:

1. The contents of main memory is saved on disk.
2. Words 8192 to 12287 are located on the hard disk.
3. Words 8192 to 12287 are loaded into memory.
4. The address map is changed to map addresses 8192 to 12287 onto memory locations 0 to 4095.
5. Execution continues as though nothing unusual had happened.

This technique for automatic overlaying is called **paging** and the chunks of program read in from disk are called **pages**.

222

Virtual Memory - Paging

To avoid confusion, the addresses referred to by the program are collectively called the **virtual address space**, and the physical memory locations, the **physical address space**.

A **memory map** or **page table** specifies for each virtual address what the corresponding physical address is.

Paging gives the programmer the illusion of a large, continuous, linear main memory, the same size as the virtual address space.

Whenever an address is referenced, the proper instruction or data word appears to be immediately available.

Because the programmer can program without reference to the paging mechanism, it is said to be **transparent**.

223

Virtual Memory - Paging

One essential requirement for a virtual memory is a disk on which to keep the whole program and all the data.

It is important to synchronize the data in memory with the data on the disk. When changes are made to the copy in main memory, they should also be reflected on the disk (eventually).

The virtual address space is broken up into a number of equal-sized pages.

Page sizes ranging from 512 Bytes to 64 KBytes per page are common at present, although sizes as large as 4 MBytes are used occasionally.

The page size is always a power of 2, for example, 2^k , so that all the addresses can be represented in *k* bits.

The physical address space is broken up into pieces in a similar way, each piece being the same size as a page, so that each piece of main memory is capable of holding exactly one page.

These pieces of main memory into which the pages go are called **page frames**. In our previous example, the main memory contains a single page frame, only.

224

Implementation of Paging

One possible way to divide up the first 64 KB of a virtual
address space is as 16 pages of 4-KB each.

The virtual memory is administered by means of a page table with as many entries as there are pages in the virtual address space.

In a 64 KB virtual address space addresses between 0 and 65535 are used by programs and are generated using Indexing, indirect addressing, and all the usual techniques.

Page	Virtual addresses
≈	≈
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

225

Implementation of Paging

A physical memory consisting of eight 4-KB page frames.

This memory might be limited to 32 KB because

(1) this is the extent on the physical memory on the machine

(2) the rest of the memory is allocated to other programs.

Page frame	Bottom 32K of main memory Physical addresses
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

32 KB of main memory, say, may be divided into eight page-frames of 4 KB each.

226

Virtual Memory - Paging

Now consider how a 32-bit virtual address can be mapped onto a physical main-memory address.

Every computer with virtual memory has a device for doing the virtual-to-physical mapping. This device is called the **MMU** (**Memory Management Unit**).

Since our sample MMU maps from a 32-bit virtual address to a 15-bit physical address, it needs a 32-bit input register and a 15-bit output register.

When the MMU is presented with a 32-bit virtual address, it separates the address into a 20-bit virtual page number and a 12-bit offset within the page (because the pages in our example are 4K and 2^{12} = 4K).

227

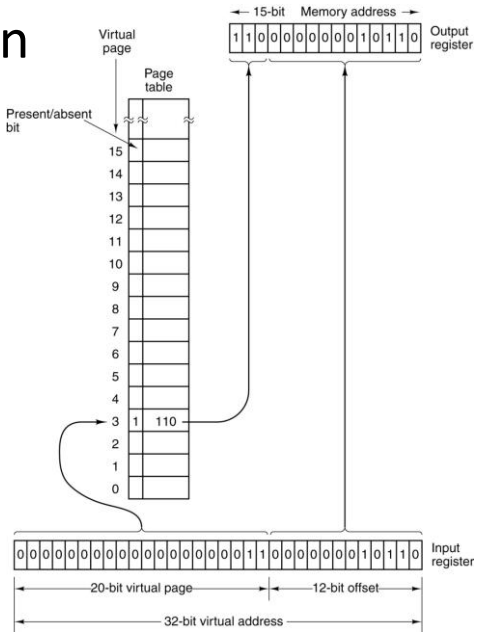
Implementation of Paging

The virtual page number is used as an index into the page table to find the entry for the page referenced

Here, entry 3 of the page table is selected.

The MMU checks if the page is already in the memory by looking at the **present/absent bit** in the page-table entry.

Here, the bit is 1, meaning the page is currently in memory.



228

Virtual Memory - Paging

The next step is to take the page-frame value from the selected entry (6 in this case (i.e., 110)) and copy it into the upper 3 bits of the 15-bit output register.

Three bits are needed because there are eight page frames (2^3) in physical memory.

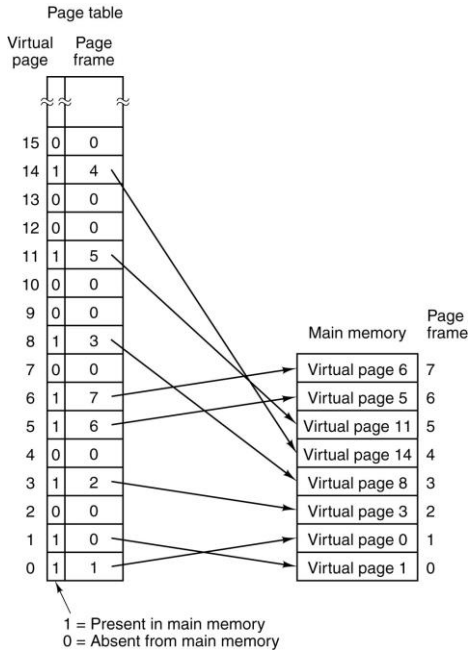
In parallel with this operation, the low-order 12 bits of the virtual address (the page-offset field) are copied into the low-order 12 bits of the output register, as shown.

This 15-bit address is now sent to the cache or memory for lookup.

229

A possible mapping between virtual pages and physical page frames.

Virtual page 0 is in page frame 1. Virtual page 1 is in page frame 0.
Virtual page 2 is not in main memory.
Virtual page 3 is in page frame 2.
Virtual page 4 is not in main memory.
Virtual page 5 is in page frame 6, and so on.



230