Evolution in memory management techniques

In early days a single program run on the machine. It used all the main memory available.

Since there is always need for more memory the programmer could plan *overlays*, i.e., static partitioning of program and data so that sections that were not needed at the same time could share the same physical addresses.

Then it was noticed that when a program was doing I/O, the CPU remained idle. Therefore, came *multiprogramming* whereby several programs were loaded together and when one program was stalled because of I/O another one could use the CPU.

The main questions to be solved were:

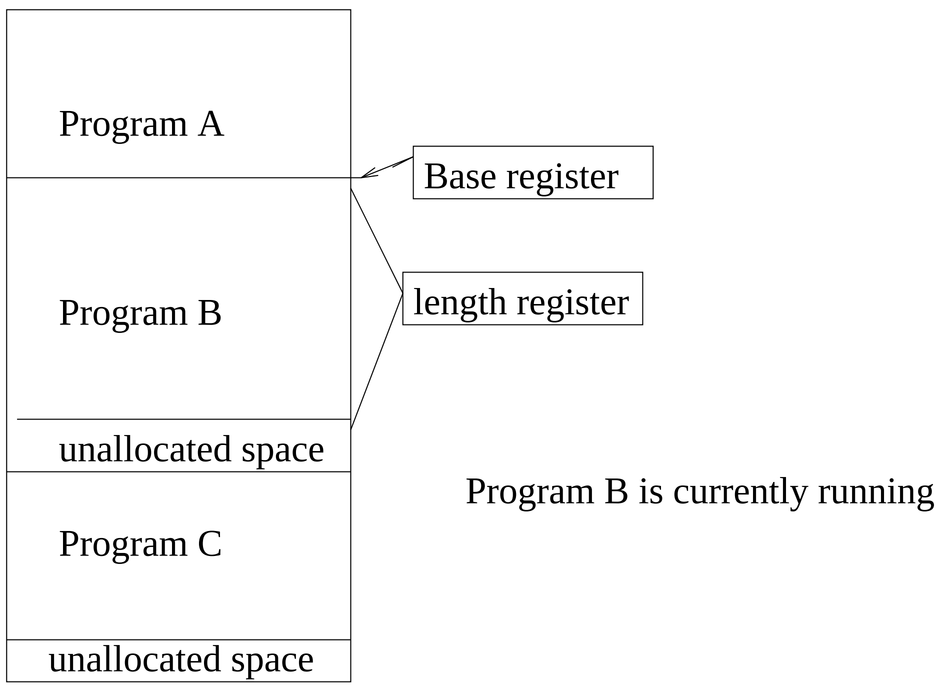
- How is one program protected from another
- How does one program ask for more memory
- When can a program be loaded in main memory

Relocation and length registers

Programs are compiled and linked w.r.t address 0.

The addresses that are generated need to be modified. In some sense an address is a *virtual address* that needs to be translated into a *physical address*

Hardware base and length registers delimit the physical address space
physical address = base register + virtual address
If (physical address) > (base address + length register)
then exception



Problems with relocation

Memory fragmentation (cf. unallocated space in previous slide)

Gets worse as time goes on (many small pieces)

Program must be allocated contiguously

Still requires overlays for large programs

Virtual memory: paging

Basic idea of *paging* (invented in early 60's at Manchester Univ.) is to divide a virtual address space in chunks of equal size, or *pages*

Divide physical memory in chunks of the same size, or *page frames*.

Provide a general mapping between virtual pages and page frames. This is a relocation mechanism whereby any virtual page can be stored in any physical frame (i.e., fully associative mapping).

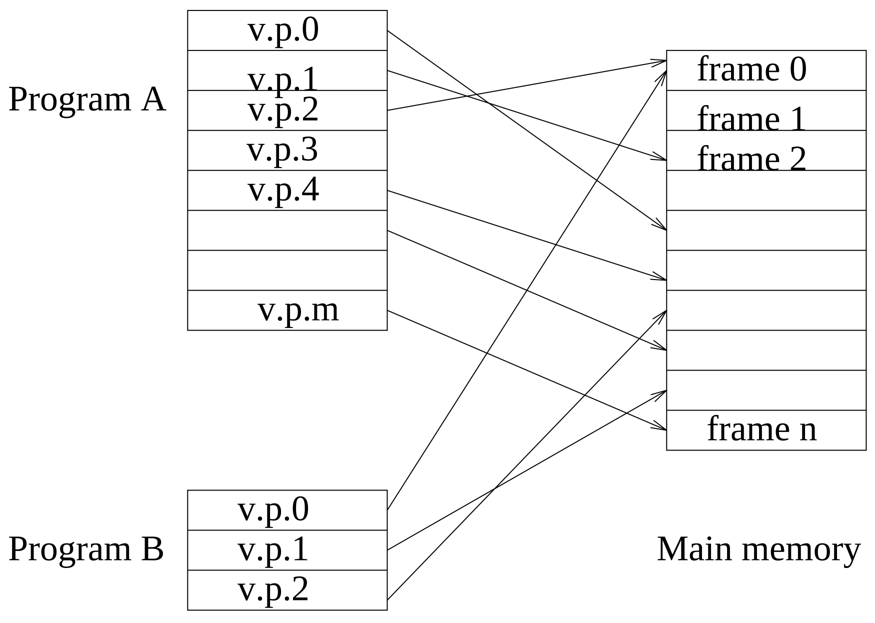
Note that the division into pages is arbitrary and has nothing to do with program semantics. There have been a few systems using *segmentation* where the virtual objects are of various sizes. Nowadays, the term segment connotes the idea of groups of pages.

Paging

Allows virtual address spaces larger than physical address spaces (recall, e.g., that the stack is in the high addresses of virtual space)

Allows sharing of physical space between different programs w/o as much fragmentation

Allows page sharing between different programs (not always simple; cf. CSE 451)



Note m can be greater than n

Programs A and B share frame 0

v.p.3 on program A has no corresponding frame but v.p.4 has.

Mapping - Page tables

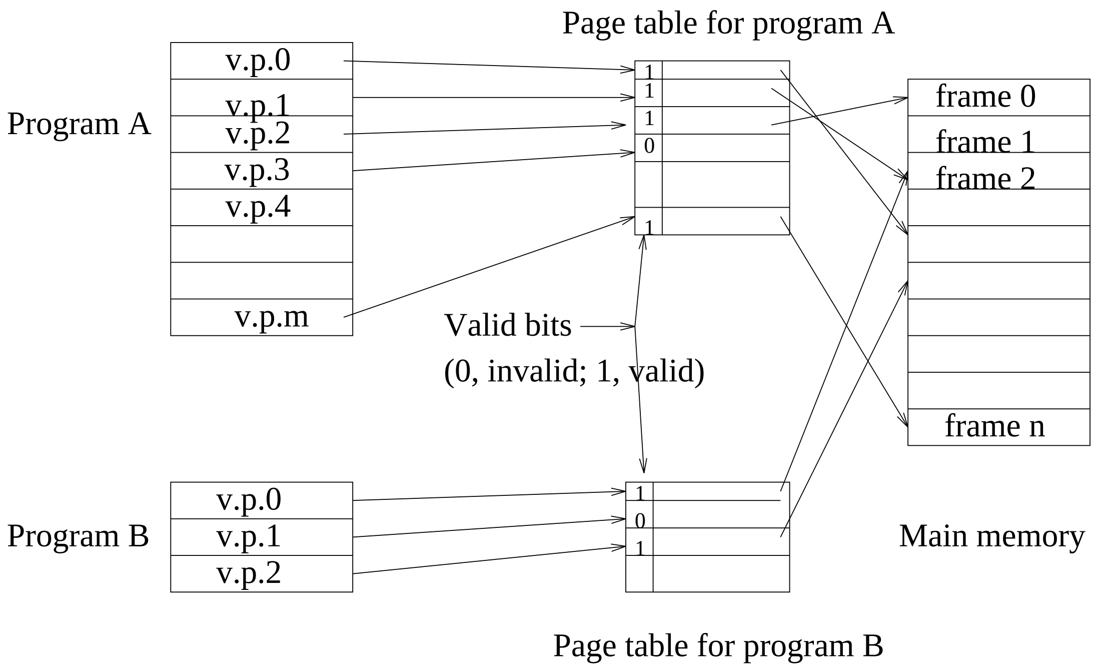
The mapping information for each program is kept in a *page table*

A *page table entry* PTE indicates the mapping of the virtual page to the physical page.

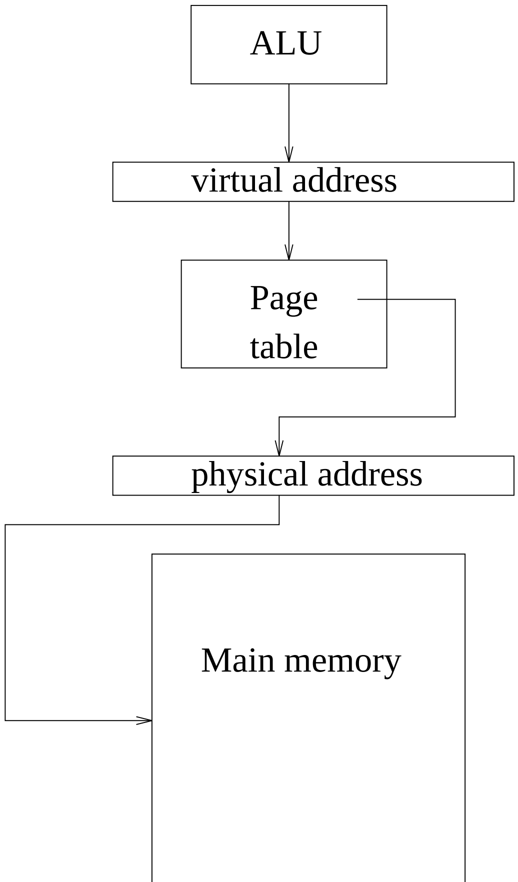
A *valid bit* indicates whether the information is current (i.e., whether the virtual page is currently mapped into a physical frame)

If there is a reference to a page with the valid bit off, we have a *page fault* (needs to go to secondary memory to fetch the page)

There are also a *dirty bit* (has the page been modified) and bits for protection (who is allowed to read, write or execute the page)



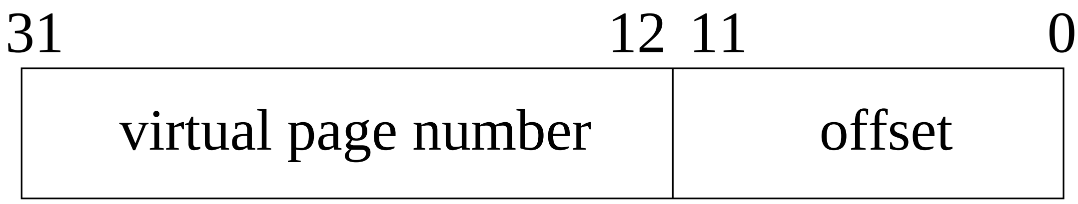
From virtual adress to memory location (highly abstracted)



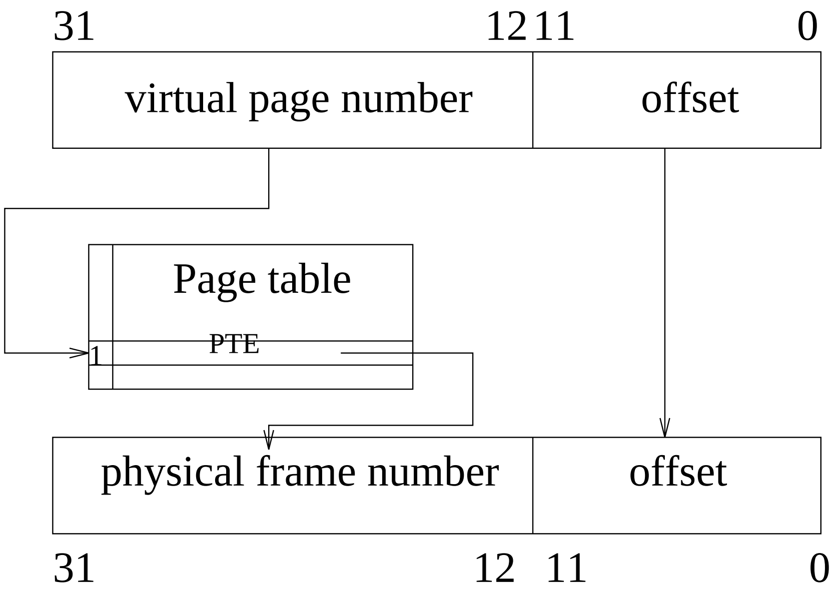
Virtual address translation

Page size is always a power of 2: so the virtual address is divided into a virtual page number and an offset.

For example for a page size of 4 KB



To translate from Virtual to physical address, index the page table with the page number. The contents of the PTE (if valid) give the frame number.



Paging systems

Recall that every address issued by the CPU is a *virtual address*

In order to access memory that virtual address must be translated into (mapped to) a physical address

That translation is done on a program by program basis. That is, each program must have its own set of *page tables*

The virtual address of program A and the same virtual address of program B *in general* will map to two different physical addresses.

Page faults and Context-switching

When a virtual address has no corresponding physical address mapping (i.e., the valid bit in the corresponding PTE is off), we have a *page fault*

On a page fault, the missing page must be fetched from secondary memory (disk). It takes of the order of milliseconds. Hence page sizes must be a few KB's to amortize the fetching cost. Because the page fault time is so large, the program (process, thread) that incurs a page fault is interrupted. The O.S. takes over and schedules another program. This is called *context-switching*.

How to choose the page size

Small page

- Advantage: takes less time to fetch from disk but...as we'll see fetching a page of size $2x$ takes less than twice the time to fetch a page of size x
- Drawback: large page tables. But can use multilevel page tasbles

Large page

- Advantage: better throughput from disk; smaller page tables
- Drawback: fragmentation

Current trends: page sizes 4K or 8K

Some O.S. allow several page sizes (e.g., one enormous one, and a "normal" one)

Basic questions on how to handle page faults and some "top level" answers

Questions:

1. When do we bring the contents of a page in main memory.?
2. Where do we put it?
3. How do we know it's there?
4. What happens if main memory is full and we want to bring in something new?

Answers:

1. The first time there is a page fault for that page. This is called *demand paging*
2. The mapping is fully-associative
3. There is a valid bit in the PTE entry
4. One page will be kicked out to liberate a frame (replacement algorithms will be studied in detail CSE 451)