Evolution in memory management techniques

- In early days, single program ran on the whole machine
  - used all the memory available
- Even so, there was often not enough memory to hold data and program for the entire run
  - use of overlays, i.e., static partitioning of program and data so that parts that were not needed at the same time could share the same memory addresses
- Soon, it was noticed that I/O was much more time consuming than processing, hence the advent of *multiprogramming*

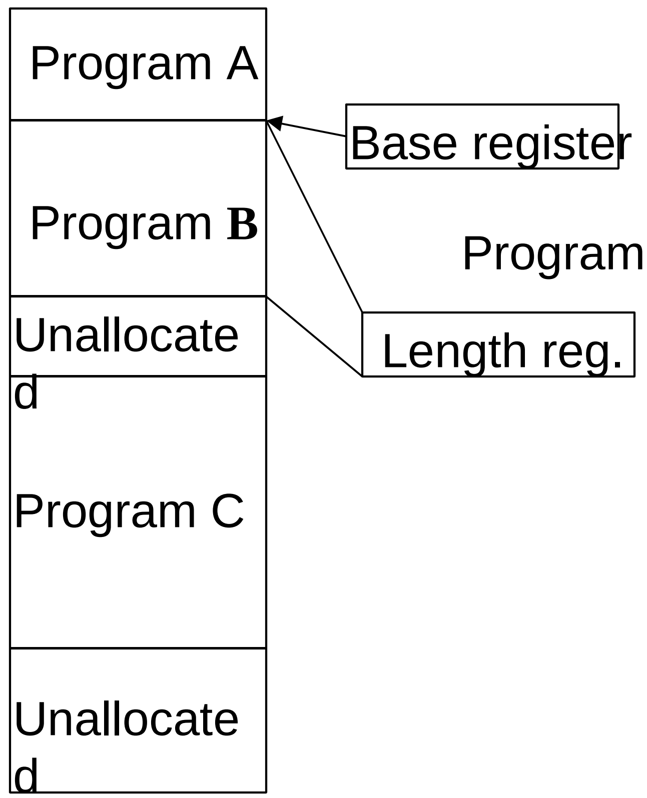
Multiprogramming: issues in memory management

- Multiprogramming
  - Several programs are resident in main memory at the same time
  - When one program executes and needs I/O, it relinquishes CPU to another program
- Some important questions from the memory management viewpoint:
  - How is one program protected from another?
  - How does one program ask for more memory?
  - How can a program be loaded in main memory?

Multiprogramming: early implementations

- Programs are compiled and linked wrt to address 0
- Addresses that are generated by the CPU need to be modified
  - **A generated address is a *virtual address***
  - The virtual address is translated into a *real* or *physical address*
- In early implementations, use of a base and length registers
  - physical address = base register contents + virtual address
  - if physical address > (base register contents + length register) then we have an exception

Relocation and length registers



Program B is executing

Note; fragmentation (unallocated memory) gets worse as time goes on (more small pieces)

Program must be allocated in continuous memory locations

Still requires overlays for large programs

Virtual memory: paging

- Basic idea first proposed and implemented at the University of Manchester in the early 60's.
- Basic idea is to divide the virtual space into chunks of the same size, or *(virtual) pages* and divide also the physical memory into *physical pages* or *frames*
- Provide a general (fully-associative) mapping between virtual pages and frames
  - This is a relocation mechanism whereby any virtual page can be stored in any physical frame

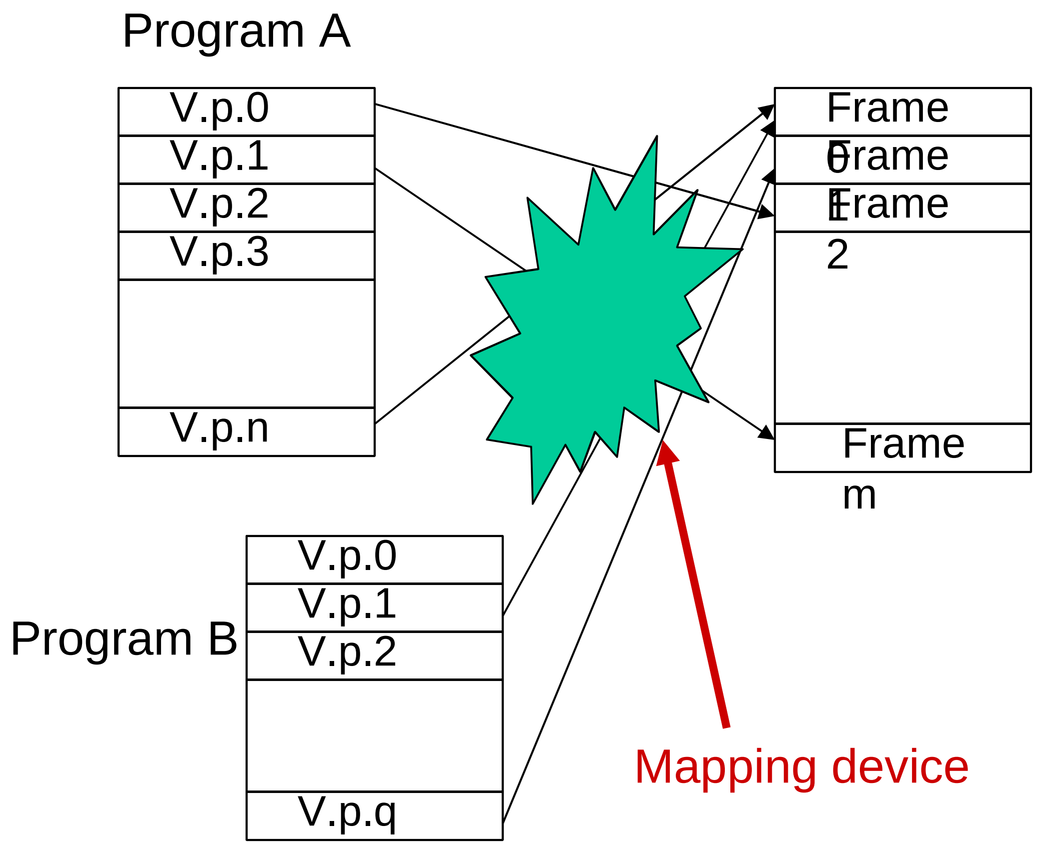
Paging and segmentation

- Division in equal size pages is arbitrary
  - division in segments corresponding to semantic entities (objects), e.g., function text, data arrays etc. may make more sense but…
  - implementation of segments of different sizes is not as easy (although it has been done, most notably in the Burroughs series of machines)
- Nowadays, segmentation has the connotation of groups of pages

Paging

- Allows virtual address space larger than physical memory
  - recall that the stack starts at the largest possible virtual address and grows towards lower addresses while code starts at low addresses
- Allows sharing of physical memory between programs (multiprogramming) without as much fragmentation
  - physical memory allocated to a program does not need to be continuous; only an integer number of pages
- Allows sharing of pages between programs (not always simple, cf. CSE 451)

Illustration of paging



Physical memory

Note: In general n, q >> m

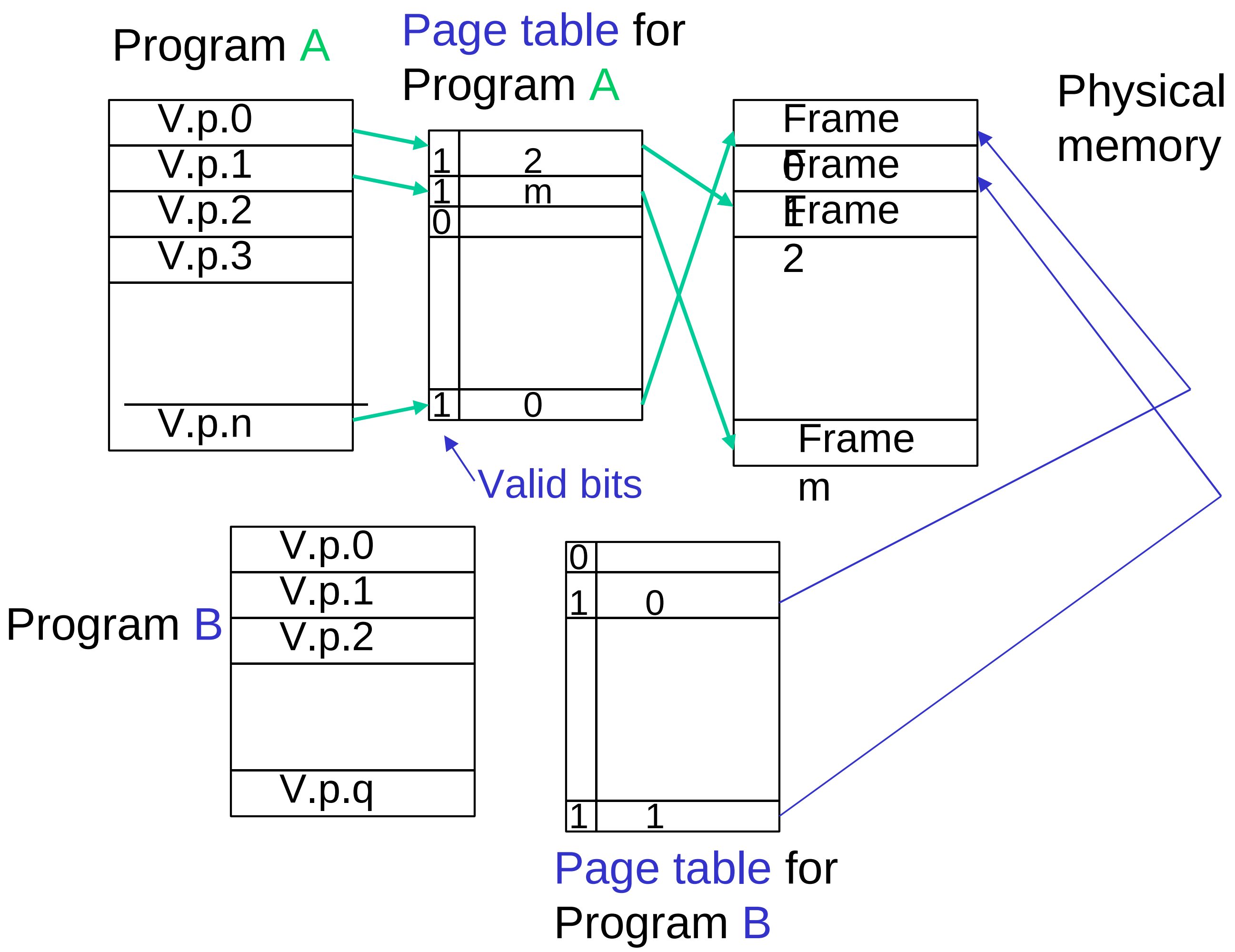
Programs A and B share frame 0 but with different virtual page numbers

Not all virtual pages of a program are mapped at a given time

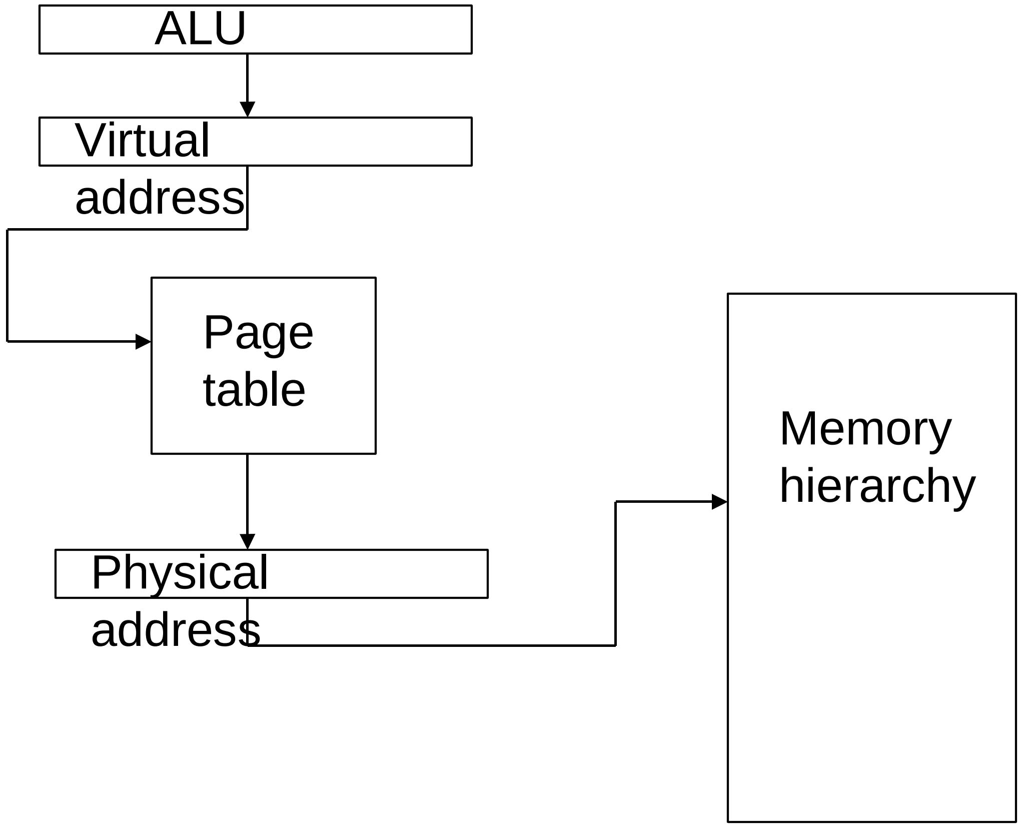
Mapping device: Page table

- Mapping info. 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 mapping is current or not
- If there is a memory reference (recall that a reference is a *virtual address*) to a page with the valid bit off in its corresponding PTE, we have a *page fault*
  - this means we'll have to go to disk to fetch the page
- The PTE also contains a *dirty* bit to indicate whether the page has been modified since it was fetched

Illustration of page table



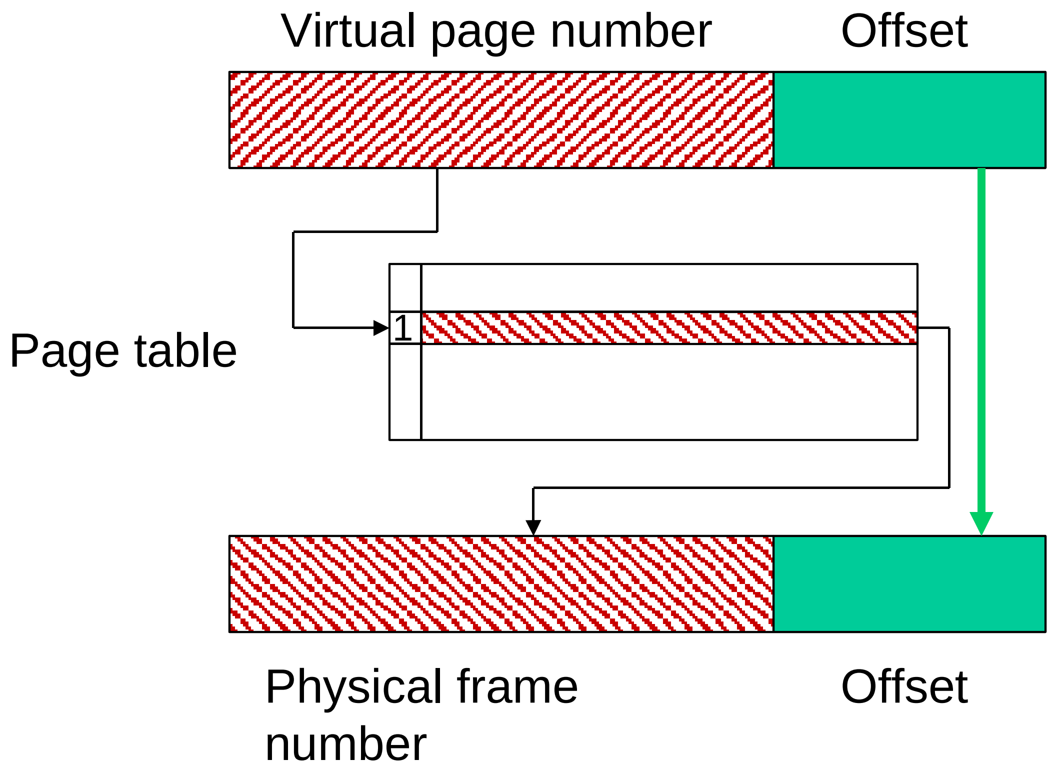
From virtual address to memory location (highly abstracted)



Virtual address translation

- Page size is always a power of 2
  - Typical page sizes: 4 KB, 8 KB
- A virtual address consists of a virtual page number and an offset within the page
  - For example, with a 4KB page size the virtual address will have a page number and an offset between 0 and 4K -1
  - By analogy with a fully-associative cache, the offset is the displacement field, the virtual page number is the tag.
  - Thus for a 4KB page, offset will be 12 bits and virtual page number is 20 bits
- The physical address will have a frame number and the same offset as the virtual address it is translated from

Virtual address translation (ct'd)



Paging system summary (so far)

- Addresses generated by the CPU are virtual addresses
- In order to access the memory hierarchy, these addresses must be translated into physical addresses
- That translation is done on a program per program basis. Each program must have its own page table
- The virtual address of program A and the same virtual address in program B will, in general, map to two different physical addresses

Page faults

- When a virtual address has no corresponding physical address mapping (valid bit is off in the PTE) we have a *page fault*
- On a page fault
  - the faulting page must be fetched from disk (takes milliseconds)
  - the whole page (e.g., 4 or 8KB) must be fetched (amortize the cost of disk access)
  - because the program is going to be idle during that page fetch, the CPU better be used by another program. On a page fault, the state of the faulting program is saved and the O.S. takes over. This is called *context-switching*

Page size choices

- Small pages (e.g., 512 bytes in the Vax)
  - Pros: takes less time to fetch from disk but as we'll see fetching a page of size $2x$ takes less than twice the time of fetching a page of size x; better utilization of pages (less fragmentation)
  - Con: page tables are large but one can use multilevel pages
- Large pages. Pros and cons converse from small pages
- Current trends
  - Page size 4 KB or 8KB.
  - Possibility of two pages sizes, one normal (4KB) and one very large, e.g. 256KB for applications such as graphics.

Top level questions relative to paging systems

- When do we bring a page in main memory?
- Where do we put it?
- How do we know it’s there?
- What happens if main memory is full

Top level answers relative to paging systems

- When do we bring a page in main memory?
  - When there is a page fault for that page, i.e., on demand
- Where do we put it?
  - No restriction; mapping is fully-associative
- How do we know it’s there?
  - The corresponding PTE entry has its valid bit on
- What happens if main memory is full
  - We have to replace one of the virtual pages currently mapped. Replacement algorithms can be sophisticated (cf. CSE 451) since we have a context-switch and hence plenty of time