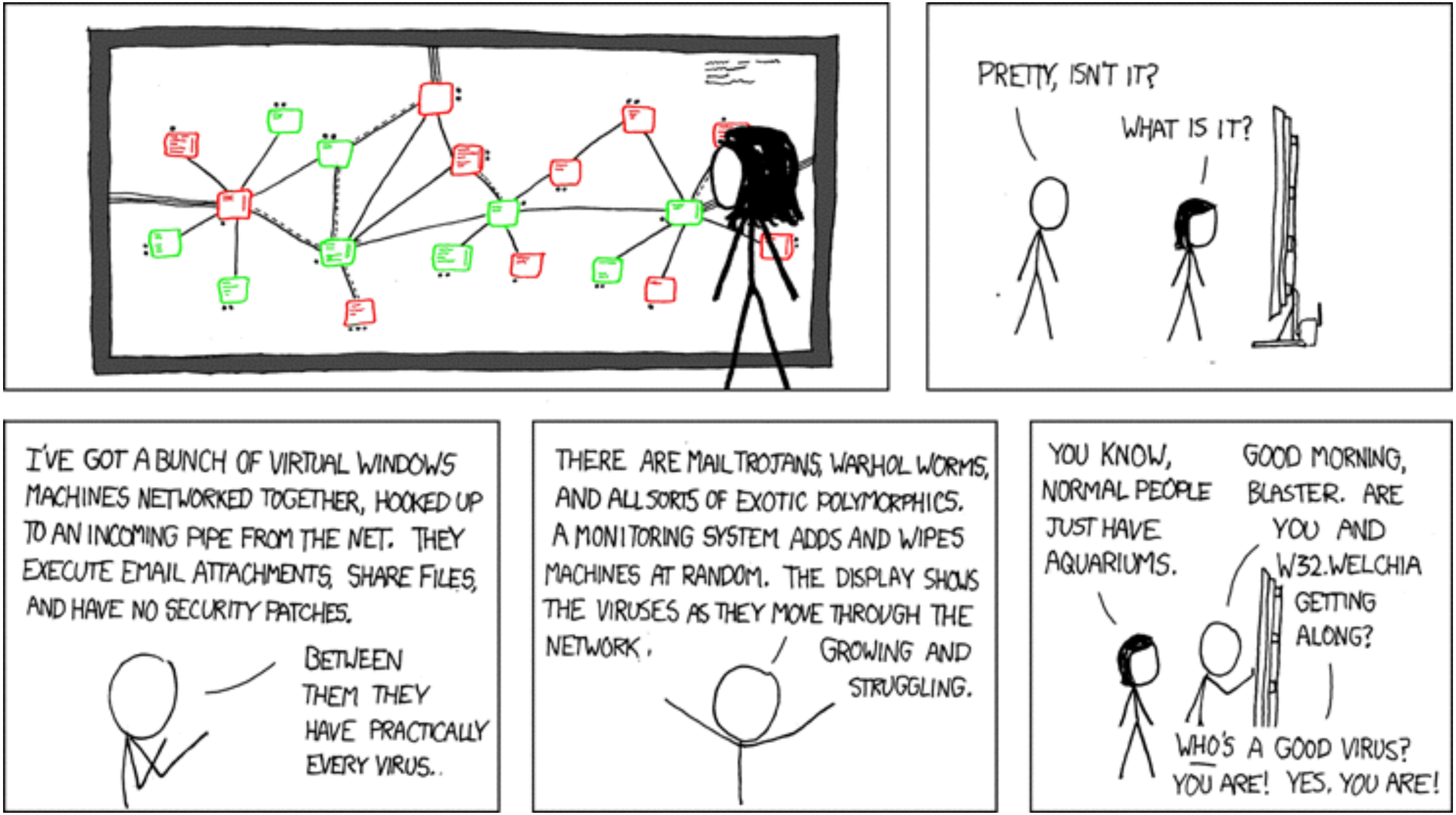
Start of Lecture: March 10, 2014



Reminders

- Assignment 2 is done!
- Assignment 3 has been released, due last day of classes
- Exercise 4 is due on March 18
- Paul Lu started talking about memory management (Chapter 8) and we'll continue from there

Some Thought Questions

- In the Semaphore section of the book it mentions that operations on a semaphore must be executed indivisibly (only one access to the semaphore at given point in time). How can such access be guaranteed?
  - Hardware atomic operations: test_and_set, compare_and_swap
- In a real system, CPU utilization should range from 40% to 90%. What factors cause the maximum utilization to be 90% rather than 100%?
  - context-switching has some overhead
  - lag/wait-time for using bus, memory stalls, etc.
  - though unlikely, on some systems might try to reduce wear-and-tear

Lecture 8: Main Memory

CMPUT 379, Section A1, Winter 2014
March 7 and 10

Objectives

- Provide a description of various ways of organizing memory hardware
- Explore various techniques of allocating memory to processes
- Discuss how paging works in contemporary computer systems

Why do we care about main memory?

- CPU is one main resource to share; memory is another key resource that the OS has to allocate
- There are many options for how we can store instructions before they are executed on the CPU, and for how we store the data that those instructions use
- Which ones should we use and what effects the choice?

What are some parameters in our choice?

- Do we store our entire program as one contiguous block of code that can be run, or in non-contiguous chunks?
- What are the repercussions of each choice?
- Where should we load the program in memory?
- What should we load in memory? The whole program?
- How do we protect programs from each other (and protect critical OS code)?

⋮

Review so far

- Many overhead issues for maintaining the location of a process in memory
- loader maps relocatable addresses to absolute addresses
- Memory can be separated into fixed or dynamic partitions
- **Absolute loading** — address specified at compile time and needs to be relinked to move (static partitioning)
- **Relocatable loading** — allocated exactly the required amount of memory when loaded with start address given then, not at compile time (dynamic partitioning)

Dynamic Memory Allocation

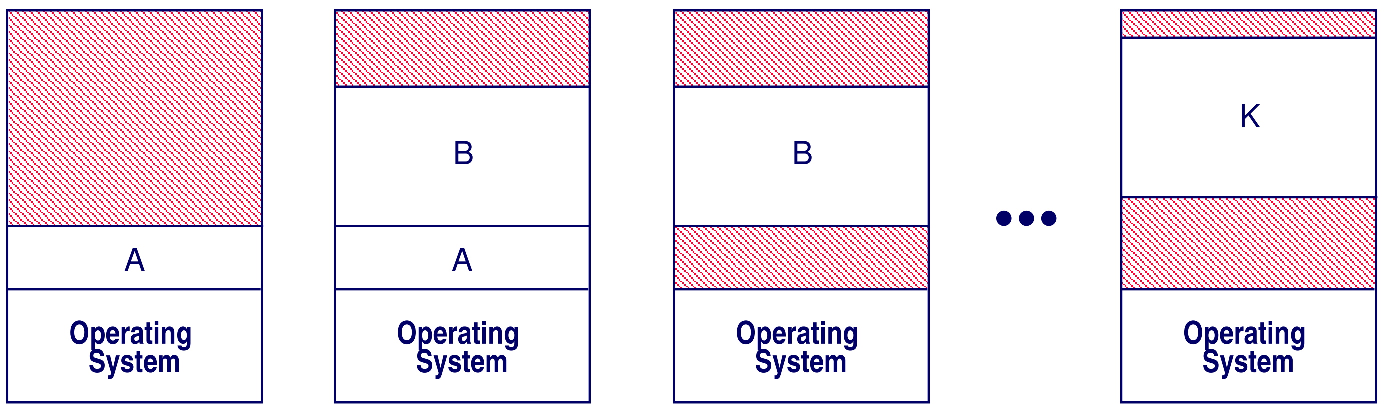
- **First-fit:** allocate the first hole that is big enough
- **Best-fit:** allocate the smallest hole that is big enough
- **Worst-fit:** allocate the largest hole
- Which of these do you think might be the best in terms of speed and memory availability?

Fragmentation

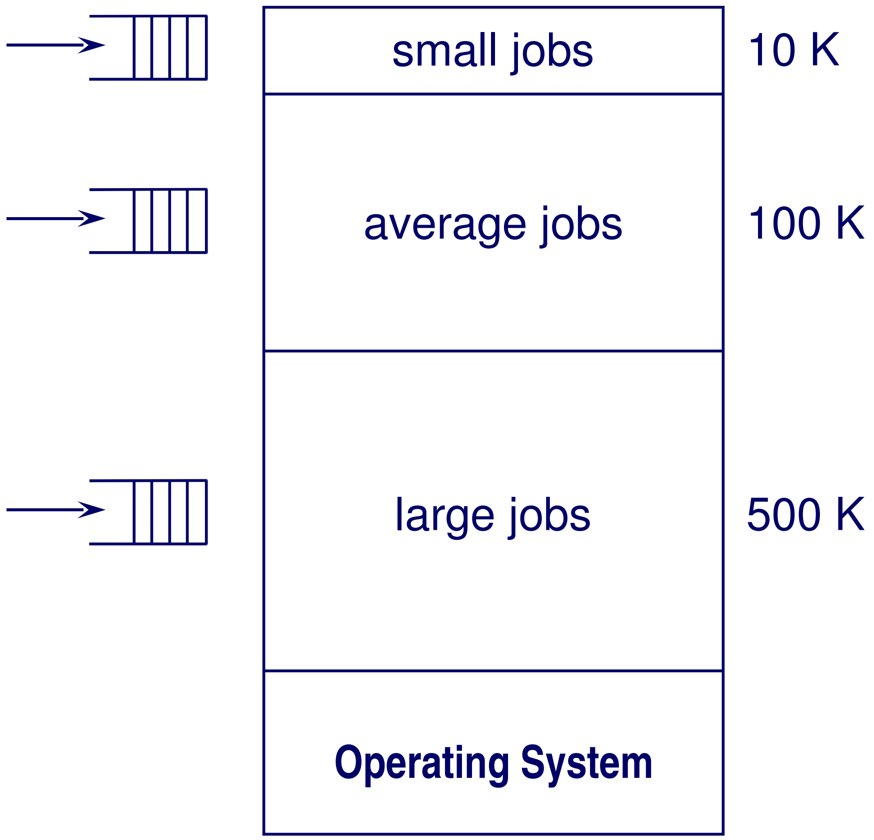
- **External Fragmentation** — waste of memory **between** partitions, from scattered non-contiguous free space
  - Total memory available is enough to satisfy requests, but usable
  - Using first-fit and continuous programs, statistical analysis reveals that can lose 50% of memory to fragmentation
- **Internal Fragmentation** — waste of memory **within** a partition, since size of partition larger than needed
  - Allocated memory is slightly larger than requested memory; this unused internal memory in the partition is internal fragmentation

When is fragmentation the worst?

- **External Fragmentation** severe in dynamic partition schemes
  - e.g. first-fit or best-fit strategies with variable sized programs



- **Internal Fragmentation** severe in static partition schemes
  - e.g. early fixed partitions with 3 partitions: small, medium, large

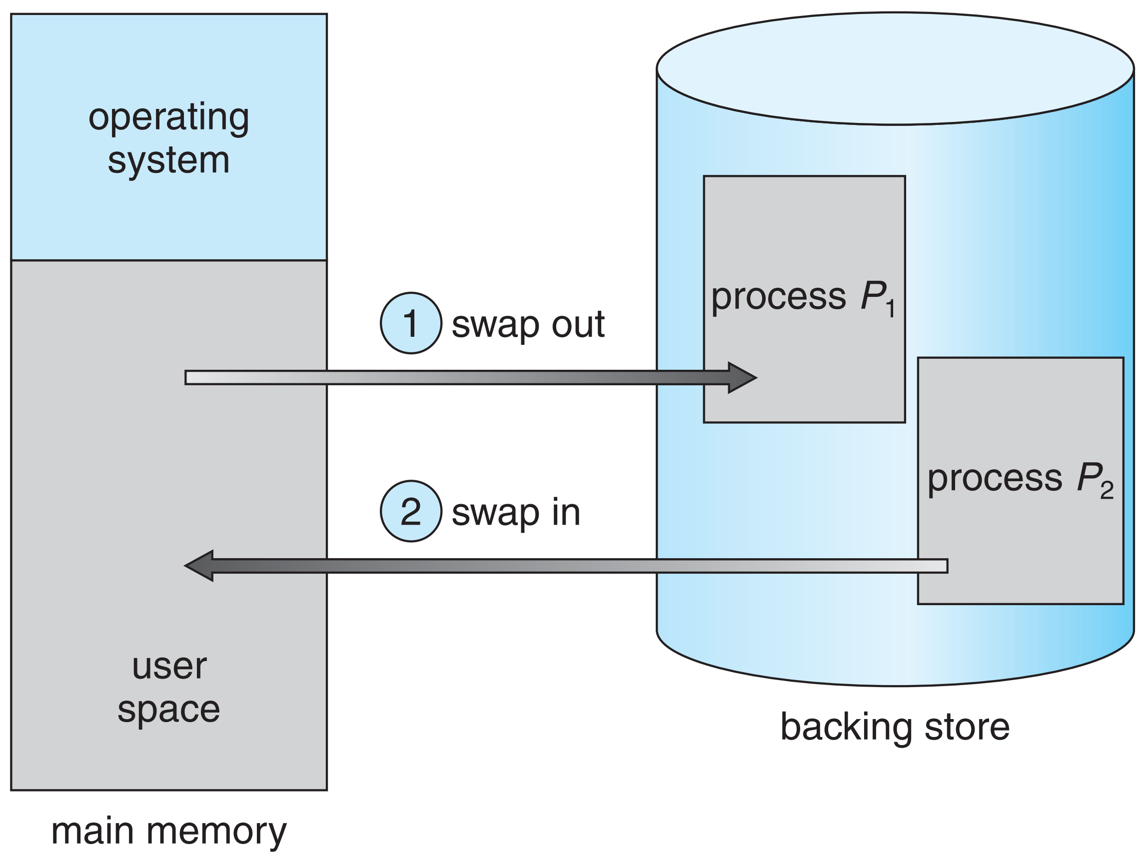


How do we fix fragmentation?

- How could you fix external fragmentation?
  - Compaction. What are the issues here?
- How could you fix internal fragmentation?
  - Allocate exactly the amount of memory requested by the process. What are the issues here?

What happens if there is not enough space in memory?

- **Swapping** processes out of memory (to disk) is the typical approach to handle space issues
- How do we pick which processes to swap?
- Where is the process swapped back into in memory?



Decisions in swapping

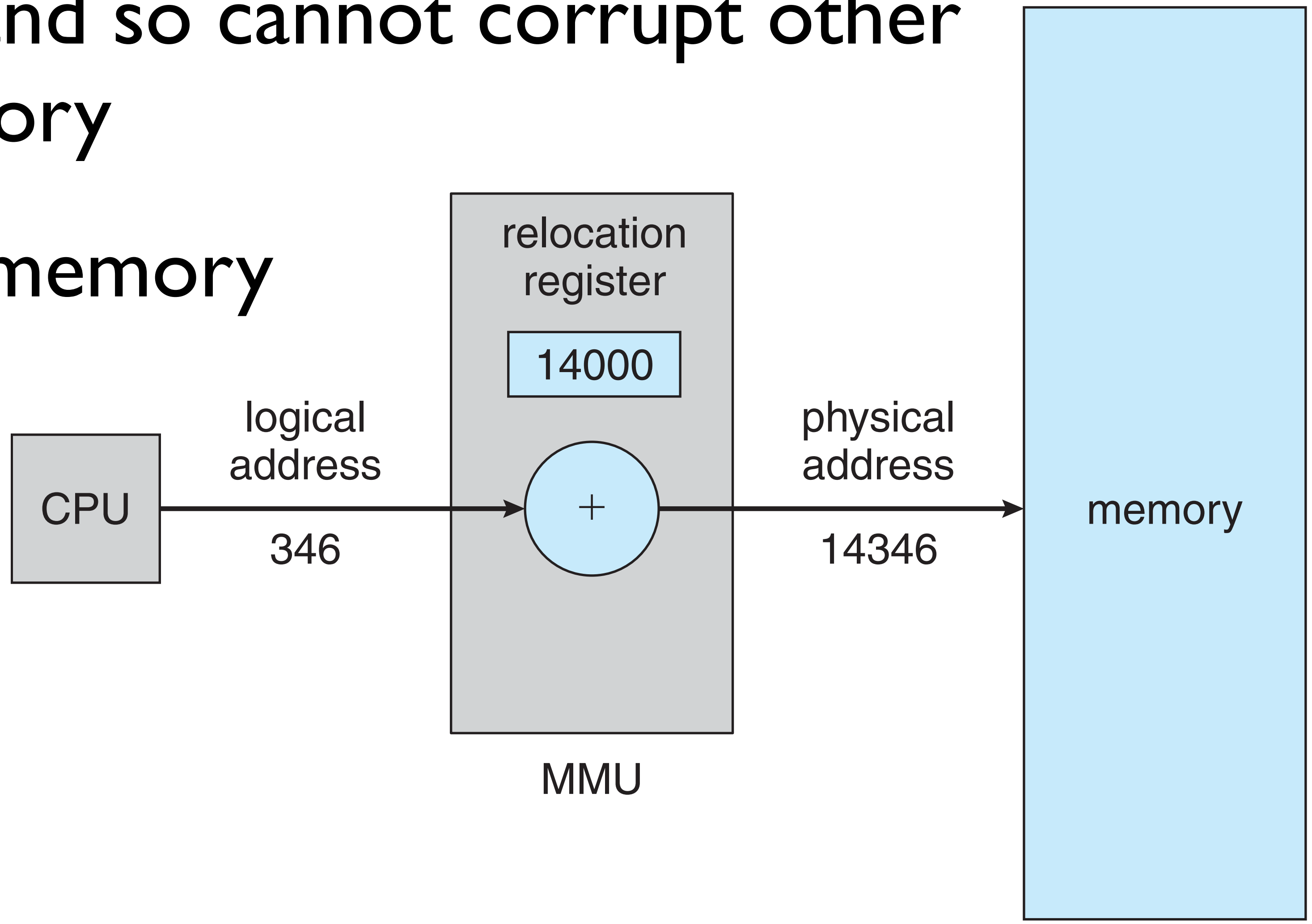
- Must select processes to swap out; some criteria that might make it more likely for a process to be swapped:
  - process is low priority
  - process is blocked/suspended
  - process has spent a larger amount of time spent in memory
- Criteria to swap a process back in: priority, time spent on swapping device
- Where to swap? e.g. same location if absolute loading
- How to create/manage swap space? e.g. swap partition

Context-Switch Time of Swapping

- Assume you have a 100MB process, with a transfer rate of 50MB/sec to hard disk
  - Swap out time of P1 = 2000 ms, swap in time of P2 = 2000 ms, so total context-switch time for swapping is 4 seconds!
- In modern system, we do not swap out an entire process and swap in a new one each time
  - swapping only used when available memory gets very low and then only parts of physical memory are swapped out
  - mobile systems do not typically do any swapping, instead asking apps to voluntarily relinquish allocated memory, terminates if not compliant

Memory Protection

- Each logical address is translated into a physical address at runtime by the memory management unit (MMU)
- This abstraction ensures that the process cannot directly access memory, and so cannot corrupt other processes allocated memory
- Many options for virtual memory space and for mapping logical addresses to physical addresses

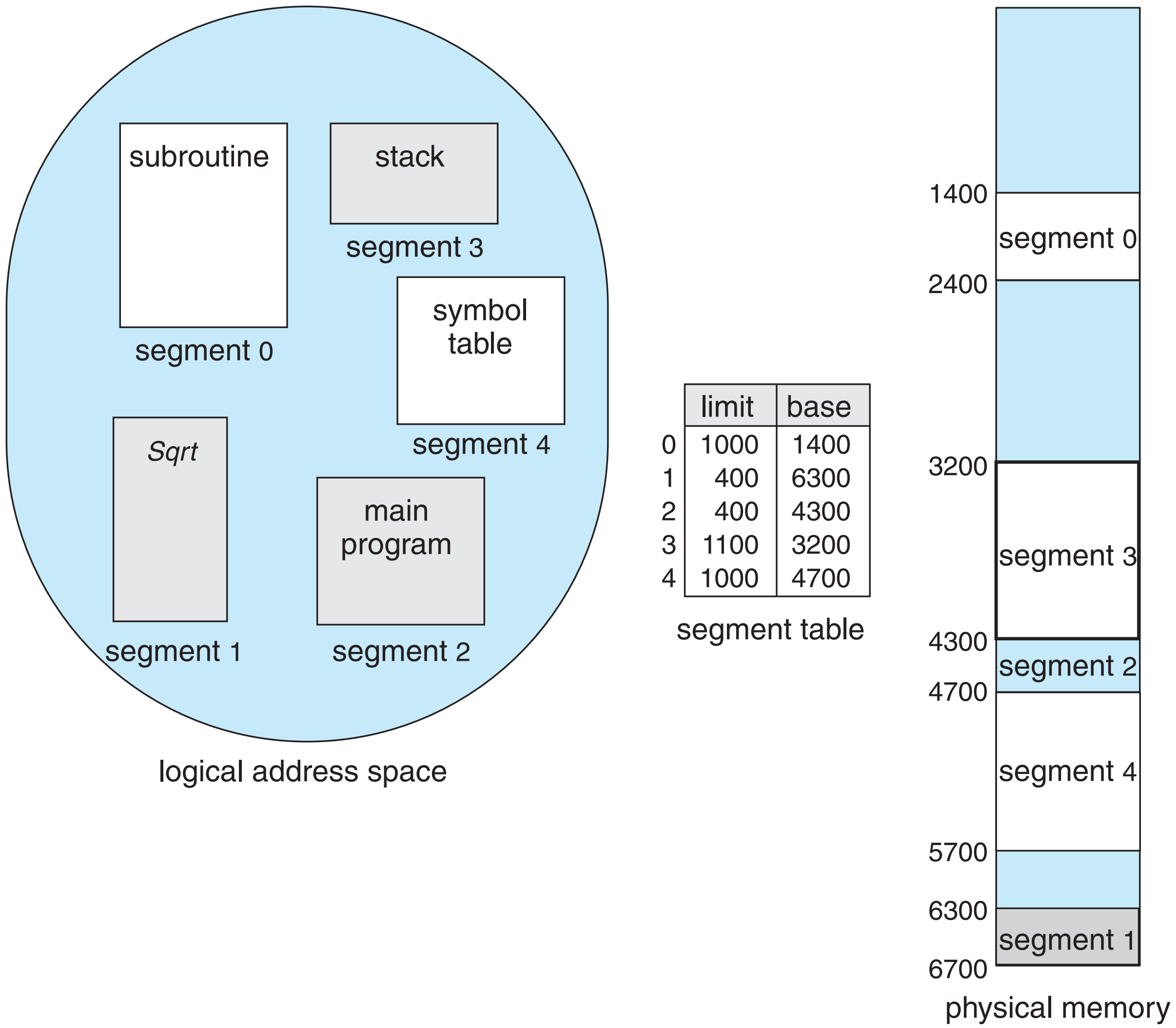


Review so far: video

https://www.youtube.com/watch?v=qdkxXygc3rE

What happens if our program does not have to be contiguous in memory?

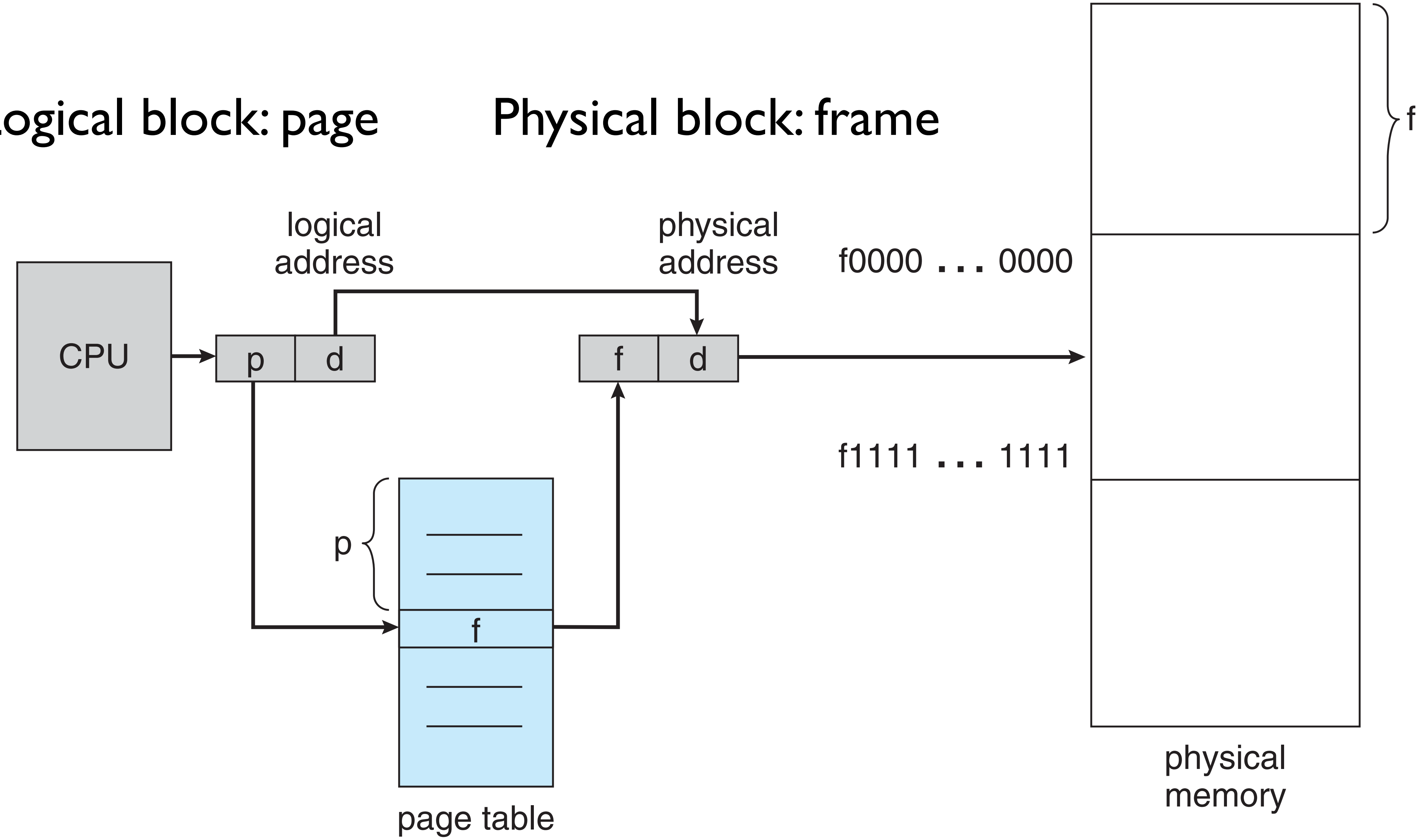
- Segmentation (which can actually be preferred to paging)
- Paging! Solves many issues due to the fact that we have uniform frame sizes
  - avoid external fragmentation
  - simplifies swapping
  - uniform structure makes it simpler to deign efficient access



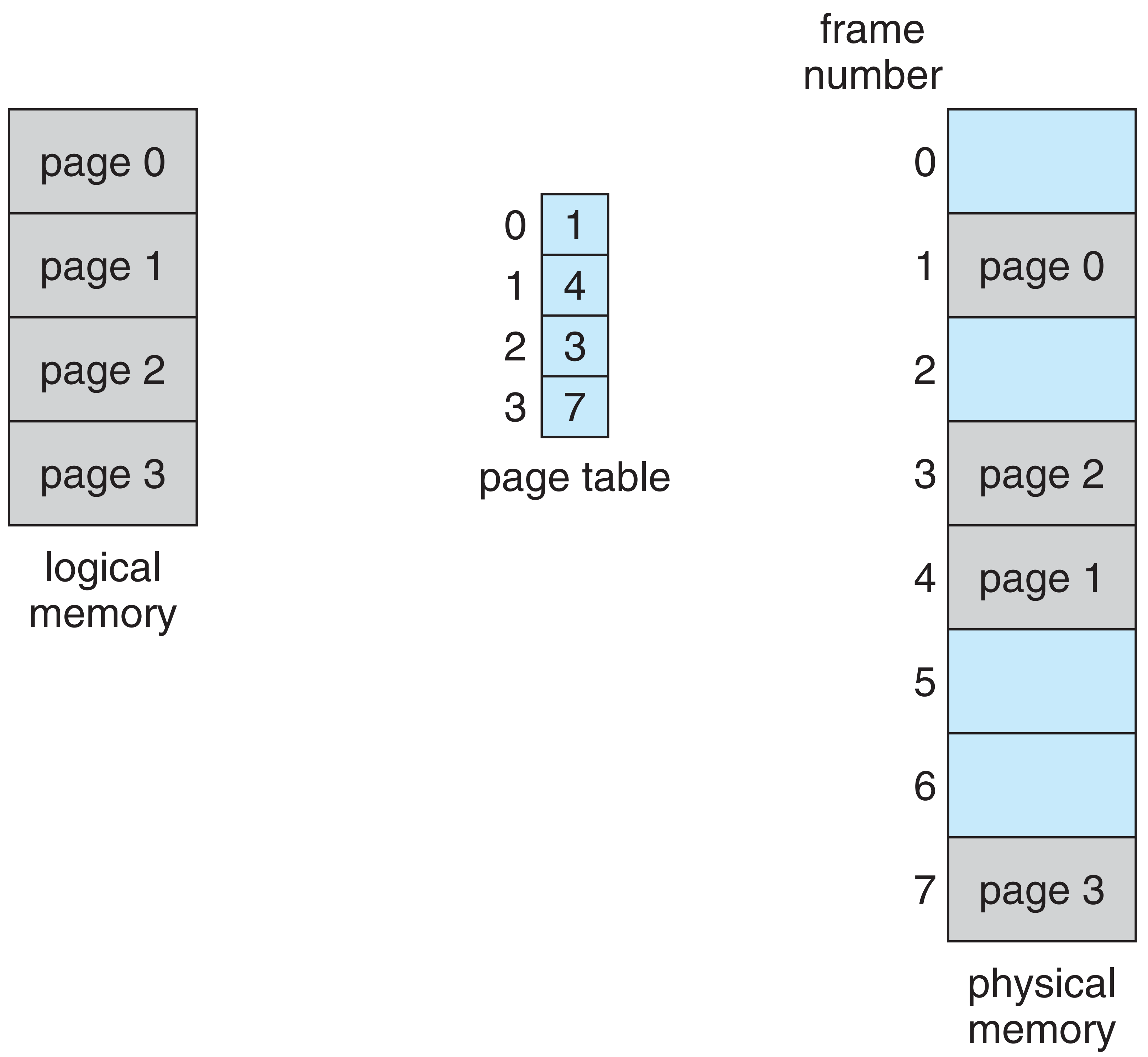
Segmentation

Paging

Logical block: page   Physical block: frame



Example: logical pages versus physical frames



Address Translation Scheme

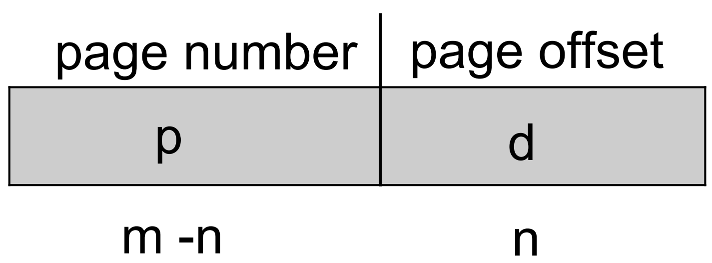
- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

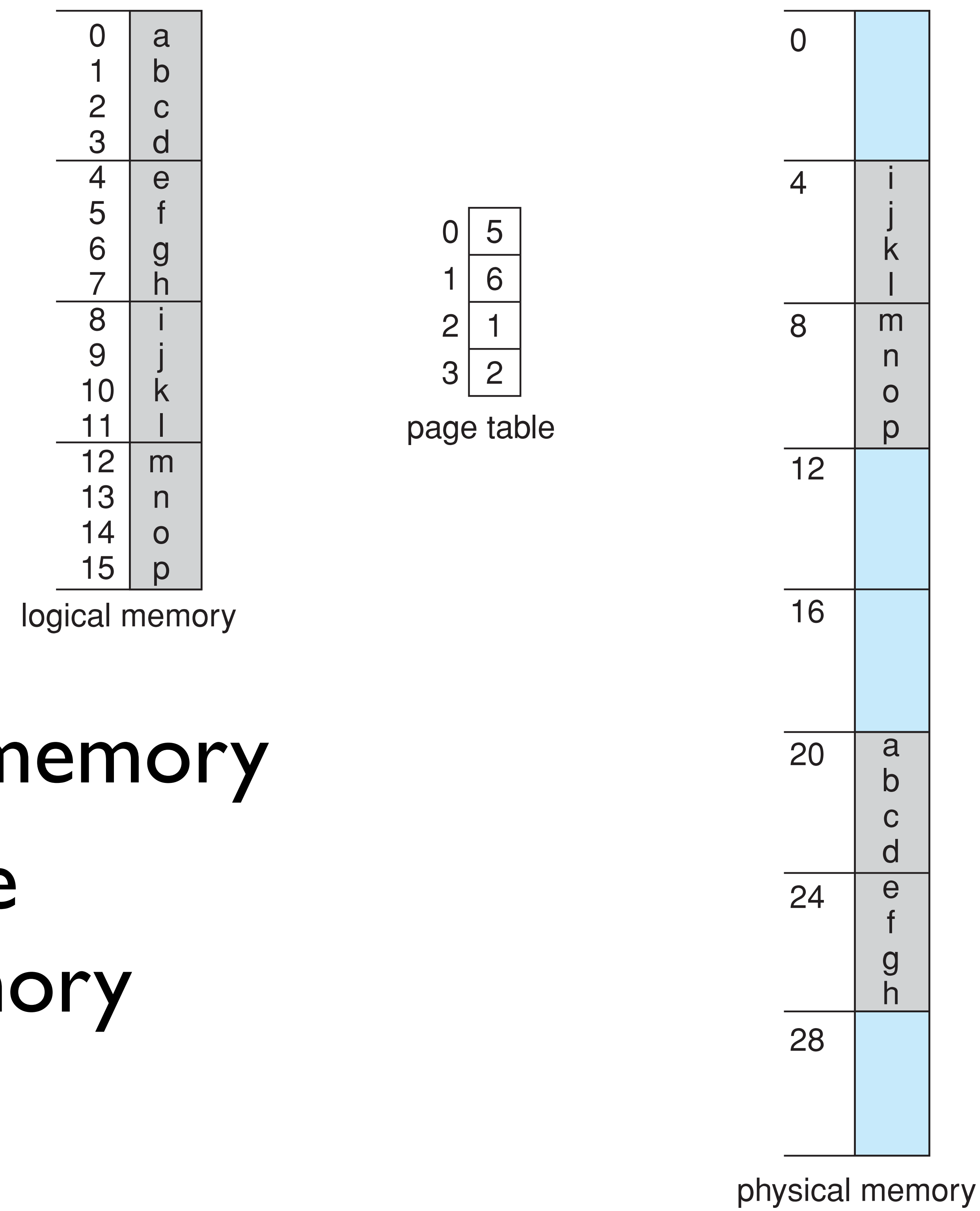
page number	page offset
p	d
m -n	n

  - For given logical address space 2^m and page size 2^n

Paging Example

$n = 2$ and $m = 4$





2^n = 16-byte logical memory

2^m = 4-byte page size

32-byte physical memory

Video Break: brought to you by another fantastic classmate!

https://www.youtube.com/watch?v=RtWbpyjJqrU&feature=youtube_gdata_player

Internal fragmentation in paging

- Let page-size = 2048 bytes, process size = 72,766 bytes
- Process uses 35 pages + 1086 bytes
- Internal fragmentation = 2048 - 1086 = 962 bytes
- Worst case fragmentation = 1 frame - 1 byte
- Average fragmentation = 1/2 frame size
- What if the distribution is not uniform? What if Gaussian distributed with mean at 3/4 frame?

Trade-off in page sizes

- Small pages sizes (e.g. VAX has 512 byte pages) have more overhead but less fragmentation
- **table fragmentation**: waste of storage due to excessively large tables
- Large page sizes (e.g. MIPS 10000 has 16 MB pages) have less overhead but more **internal fragmentation**
- How do we pick a page size?

Exercise: calculate optimal page size

- Size of page table = number of pages = size of virtual memory address space / size of page
  - e.g. for 4KB pages, number of pages = 2^32 / 2^12 = 2^20 = 1 MB
- **No internal fragmentation:** page-size = 1 byte
  - each process needs double the amount of memory to store page locations
- **No overhead:** page-size = physical memory size
  - can only have 1 process in memory
- **Objective**: internal fragmentation? table fragmentation?
  - many objectives possible that we might want to optimize
  - can also use experimental (statistical) analysis and make heuristics

Theorem for optimal page size (according to one objective)

- Let $c1$ = cost of losing a word to table fragmentation and $c2$ =cost of losing a word to internal fragmentation
- Assume that each program begins on a page boundary.
- If the average program size $s0$ is much larger than the page size z, then the optimal page size is approximately $\sqrt{2(c1/c2)s0}$

Proof of optimal page size (according to one objective)

Revealed in class. First complete the exercise of figuring it out on your own

Support of proofs from our honourable former Prime Minister

http://www.youtube.com/watch?v=aX6XMIIdkRU