Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Virtual Memory

Schedule

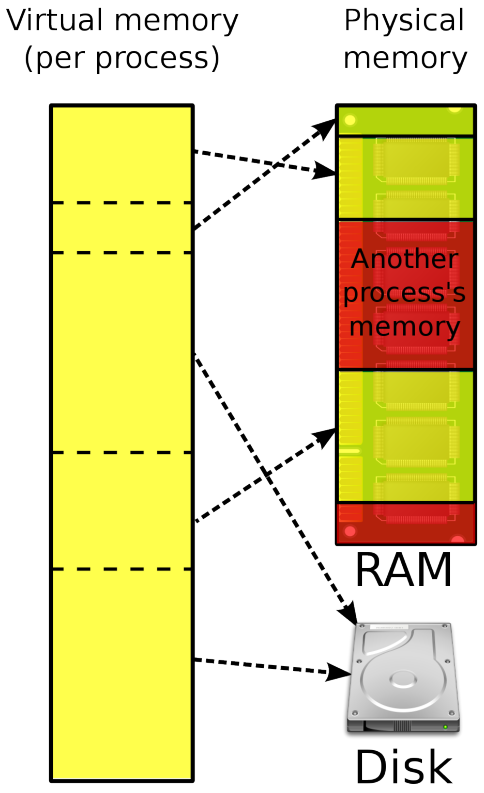
- **Today**
  - Chapter 6 – Virtual memory
- **Wednesday**
  - Chapter 7 – Input / Output Systems
- **Friday (March 30th)**
  - **Quiz #5** – Chapter 6 (Cache & Virtual Memory)

Recap – Virtual Memory

Virtual Memory is a BIG LIE!

- We **lie** to your application and tell it that the system is simple:
  - Physical memory is infinite! (or at least huge)
  - You can access *all* of physical memory
  - Your program starts at *memory address zero*
  - Your memory address is *contiguous* and *in-order*
  - Your memory is *only RAM* (main memory)

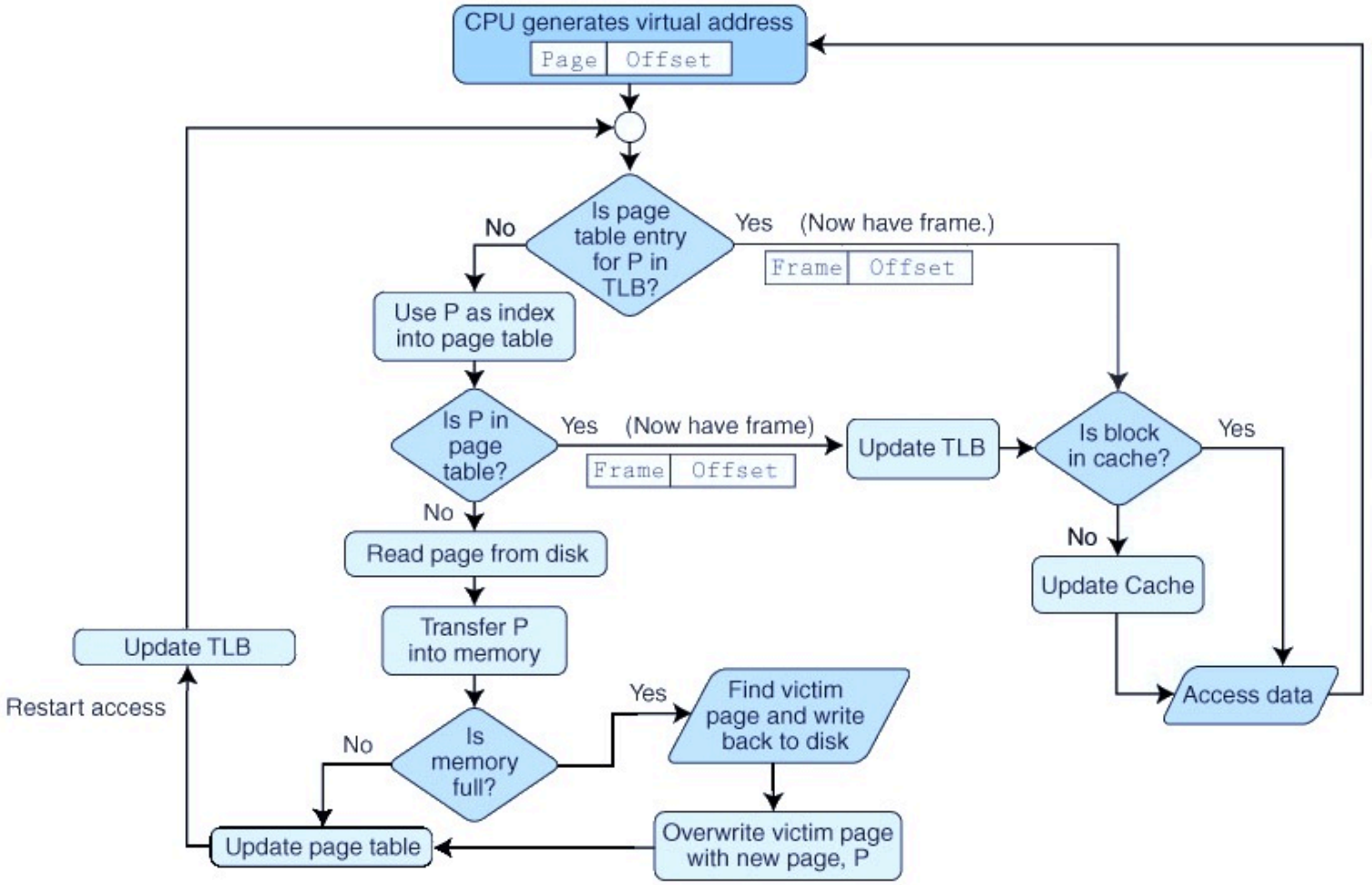
What the System Really Does



Recap – Why use Virtual Memory?

- We want to run multiple programs on the computer concurrently *(multitasking)*
  - Each program needs its own separate memory region, so physical resources must be divided
  - The amount of memory each program takes could vary dynamically over time (and the user could run a different mix of apps at once)
- We want to use multiple types of storage (main memory, disk) to increase performance and capacity
- We don't want the programmer to worry about this
  - Make the processor architect handle these details

TLB + Page Table + Main Memory + Cache



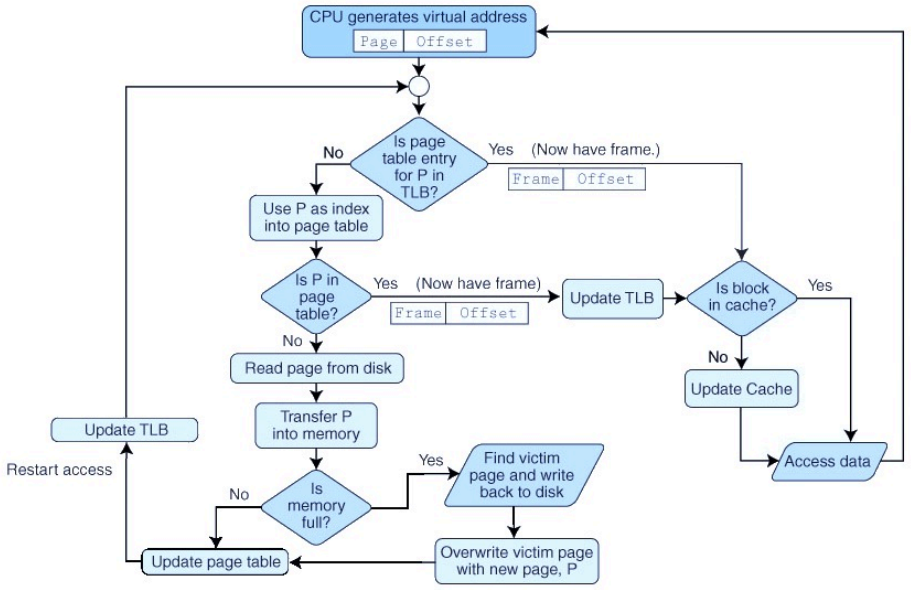
Example – Access Time

- Suppose we have a virtual memory (VM) system with a TLB, cache, and page table. Also assume:
  - A TLB hit takes 10ns, and has a hit ratio of 92%
  - A cache hit takes 20ns (hit ratio 98%)
  - A physical memory reference takes 45ns (page fault rate 0.025%)
  - A disk reference takes 150ms (including loading the page table and TLB)
- For a page fault, the page is loaded from disk and TLB is updated, and memory access restarts
- **How long does it take to access a word if it is not in physical memory?**

Example – Access Time

- Page in TLB (10ns) → no
- Page in physical memory (45ns) → no
- Read page from disk into memory (150ms) → restart
  - Assumes a free page
- Page in TLB (10ns) → yes
- Block in cache (20ns) → no
- Load block from physical memory (45ns) → done
- Total: 10ns + 45ns + 150ms + 10ns + 20ns + 45ns = **150,000,130ns**
  - Aka a **long time**

How long does it take to access a word if it is not in physical memory?



Example – Access Time

- **How frequently does this happen? (i.e. how often must we go to disk?)**
  - TLB (10ns, 92%), Cache (20ns, 98%), Memory (45ns, 1-0.025%), Disk (150ms)
- Process
  - Page in TLB → 92%
  - Page not in TLB (requires access to page table) → 8%
  - Page not in page table (i.e. not in memory) → 0.025%
  - Page not in TLB **and** not in page table:
    - 8% * 0.025% = .08 * .00025 = 0.00002 (or 0.002%)
- **What is the *effective access time* for a page fault?**
  - .00002 * 150,000,130ns = 3,000.0026ns

Segmentation and Fragmentation

Segmentation

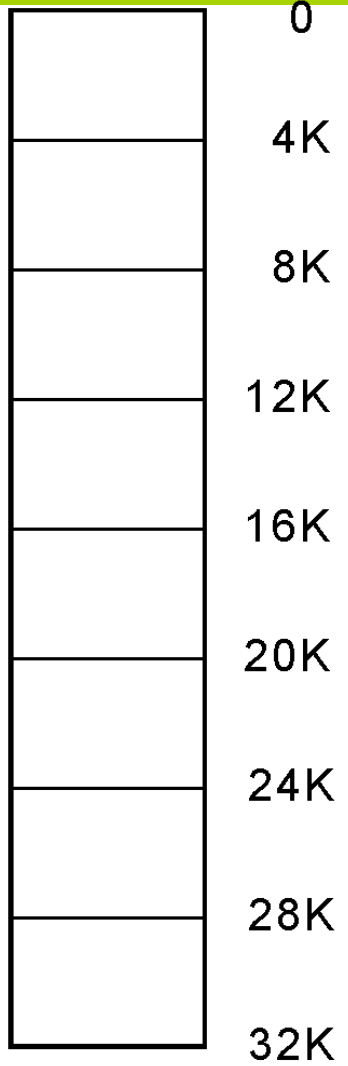
- Alternate way to implement virtual memory instead of pages: **segmentation**
- Idea: Instead of dividing memory into *equal-sized pages*, virtual address space is divided into *variable-length segments (typically under the control of the programmer)*
- A segment is located through its entry in a **segment table**
  - Starting address of segment in main memory
  - Size of segment
- Page fault? Operating system searches for a location in memory large enough to hold the segment that is retrieved from disk.

Fragmentation

- Both paging and segmentation can cause **fragmentation**
- Paging is subject to **internal fragmentation**
  - A process may not need the entire range of addresses contained *within* the page
  - There may be many pages containing unused fragments of memory
- Segmentation is subject to **external fragmentation**
  - Contiguous chunks of memory become broken up as segments are allocated and deallocated over time
  - Fragmentation is “outside” the segment

Fragmentation

- Example computer
  - 32K main memory
  - Divided into 8 page frames of 4K each
- The numbers at the right are memory frame addresses



Internal Fragmentation

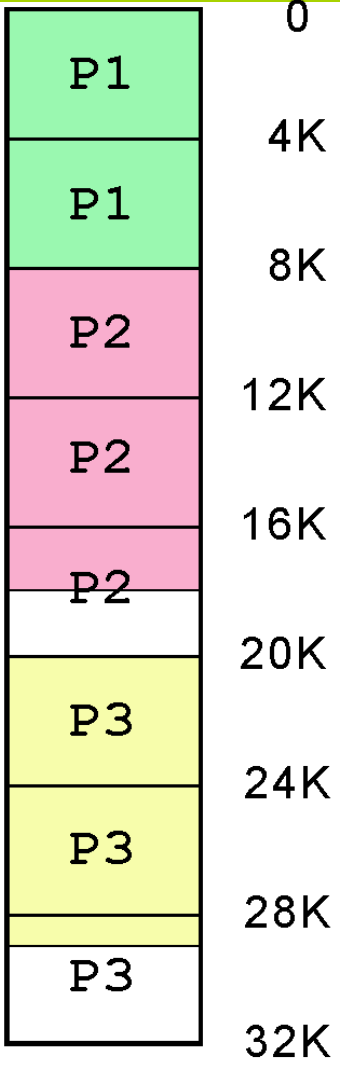
- Suppose there are four processes waiting to be loaded into the system with memory requirements as shown in the table
- All together, these processes require 31K of memory
  - **This should all fit, right?**

Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

Internal Fragmentation

- When the first three processes are loaded, memory looks like this:
- All of the frames are occupied by only three of the processes

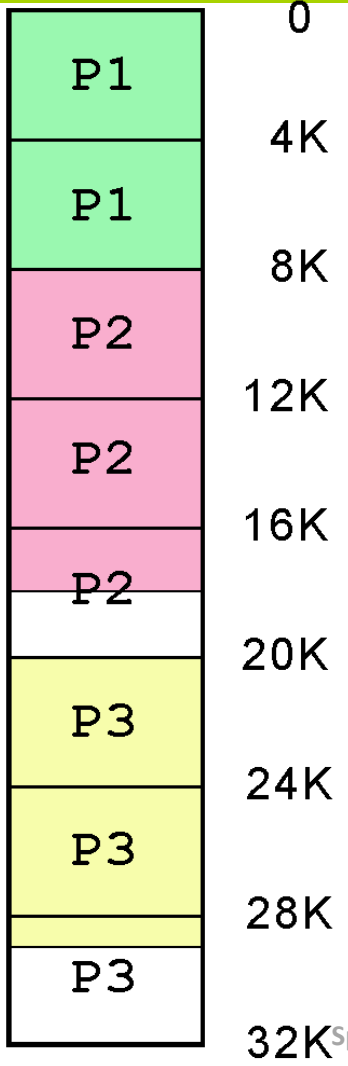
P1	8K
P2	10K
P3	9K
P4	4K



Internal Fragmentation

- P4 has to wait for one of the other three processes to terminate,
  - There are no unallocated frames available
  - But there *is* enough free bytes in memory, we just can't use them!
- This is an example of **internal fragmentation**

P1	8K
P2	10K
P3	9K
P4	4K



External Fragmentation

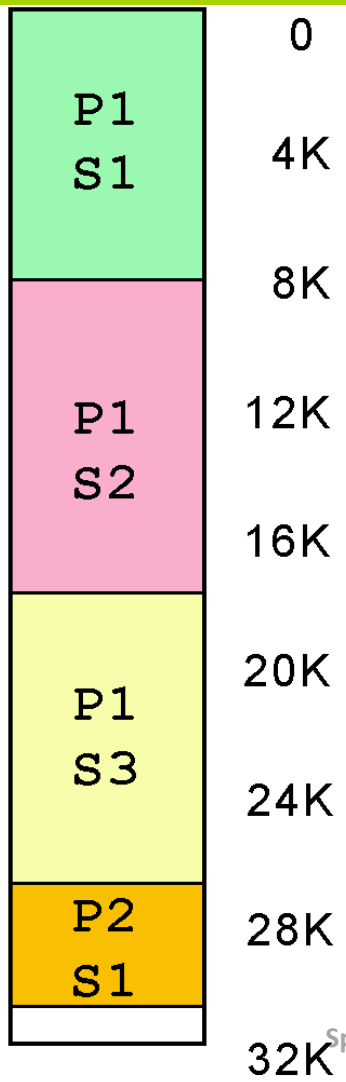
- Suppose that instead of frames, our 32K system uses **segmentation**
- The memory segments of two processes is shown in the table at the right
  - 42K of total segments with these processes
- The segments can be allocated anywhere in memory

Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

External Fragmentation

- All of the segments of P1 and one of the segments of P2 are loaded as shown at the right.
- Segment S2 of process P2 requires 11K of memory, and there is only 1K free, so it must wait

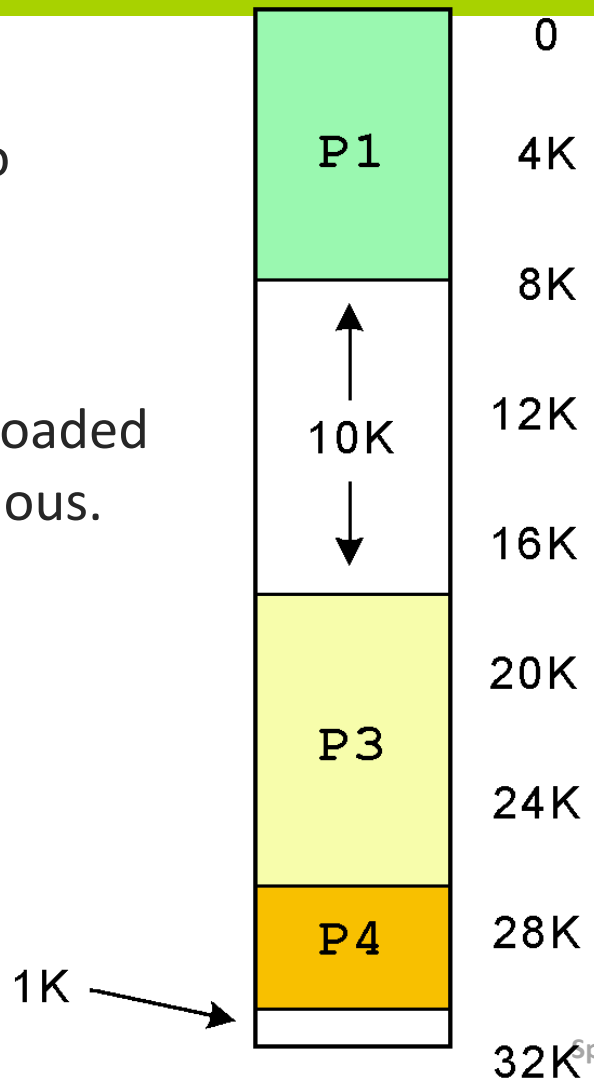
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



External Fragmentation

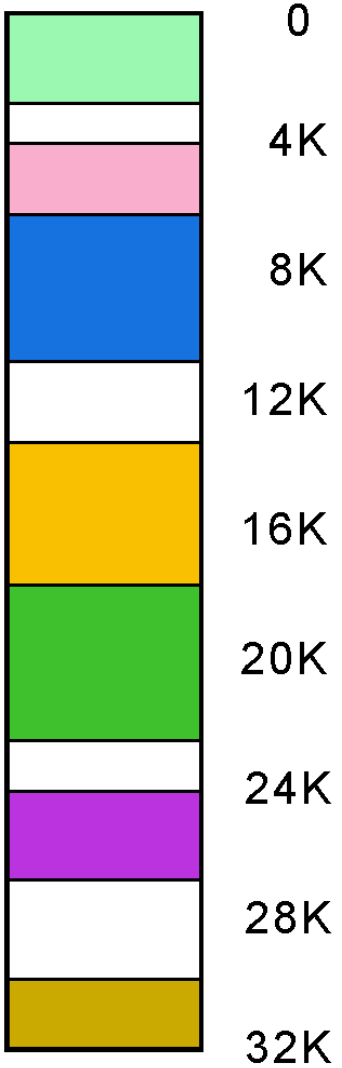
- Eventually, Segment 2 of Process 1 is no longer needed, so it is unloaded
  - 11K of free memory now available
- But, Segment 2 of Process 2 cannot be loaded because the free memory is not contiguous.

P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



External Fragmentation

- Over time, the problem gets worse, resulting in small unusable blocks scattered throughout physical memory
- This is an example of **external fragmentation**
- Eventually, this memory is recovered through **compaction**, and the process starts over



Chapter 6 Summary

- **Done with Chapter 6!**
- Computer memory is organized in a hierarchy
  - Smallest, fastest memory at the top
  - Largest, slowest memory at the bottom
- Cache
  - Gives faster access to main memory
  - Cache maps blocks of main memory to blocks of cache memory
- Virtual memory
  - Uses disk storage to give the illusion of having a large main memory
  - Virtual memory maps page frames to virtual pages

Chapter 6 Summary

- There are three general types of cache:
  Direct mapped, Fully associative, and Set associative
- Need replacement policies (i.e. which pages to evict?) for
  - Fully associative cache
  - Set associative cache
  - Virtual memory
- Replacement policies include LRU (least recently used), FIFO (first-in, first-out), or random replacement
  - Need to take into account what to do with dirty blocks
- All virtual memory must deal with fragmentation
  - Internal fragmentation for paged memory
  - External fragmentation for segmented memory