Memory Management – Thrashing, Segmentation and Paging

CS 416: Operating Systems Design, Spring 2011

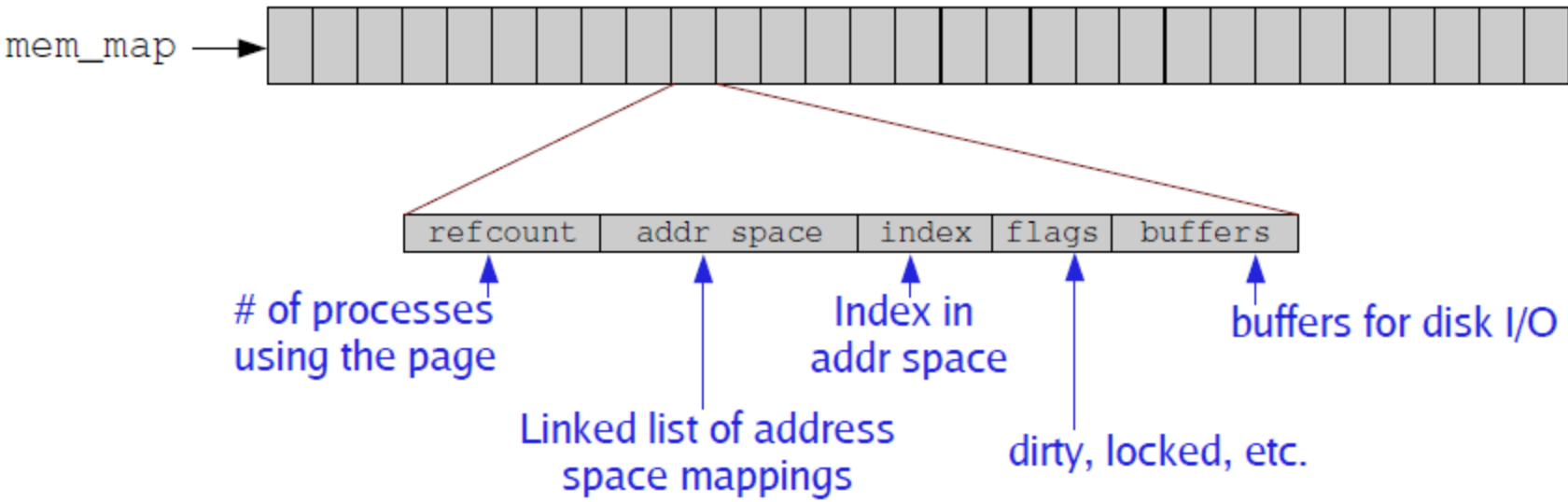
Department of Computer Science
Rutgers University

Rutgers Sakai: 01:198:416 Sp11
(https://sakai.rutgers.edu)

Summary of Page Eviction Algorithms

- OPT (MIN)
- Random
- FIFO (Use a List to maintain the pages as allocated)
  - Suffers from belady's anomaly
- LRU
  - Using a 32 bit timestamp (Not Efficient)
- LRU Approximations
  - Counter Implementation
  - Clock Implementation (Second Chance)
  - Counter + Clock (Nth Chance)

Physical Memory Map

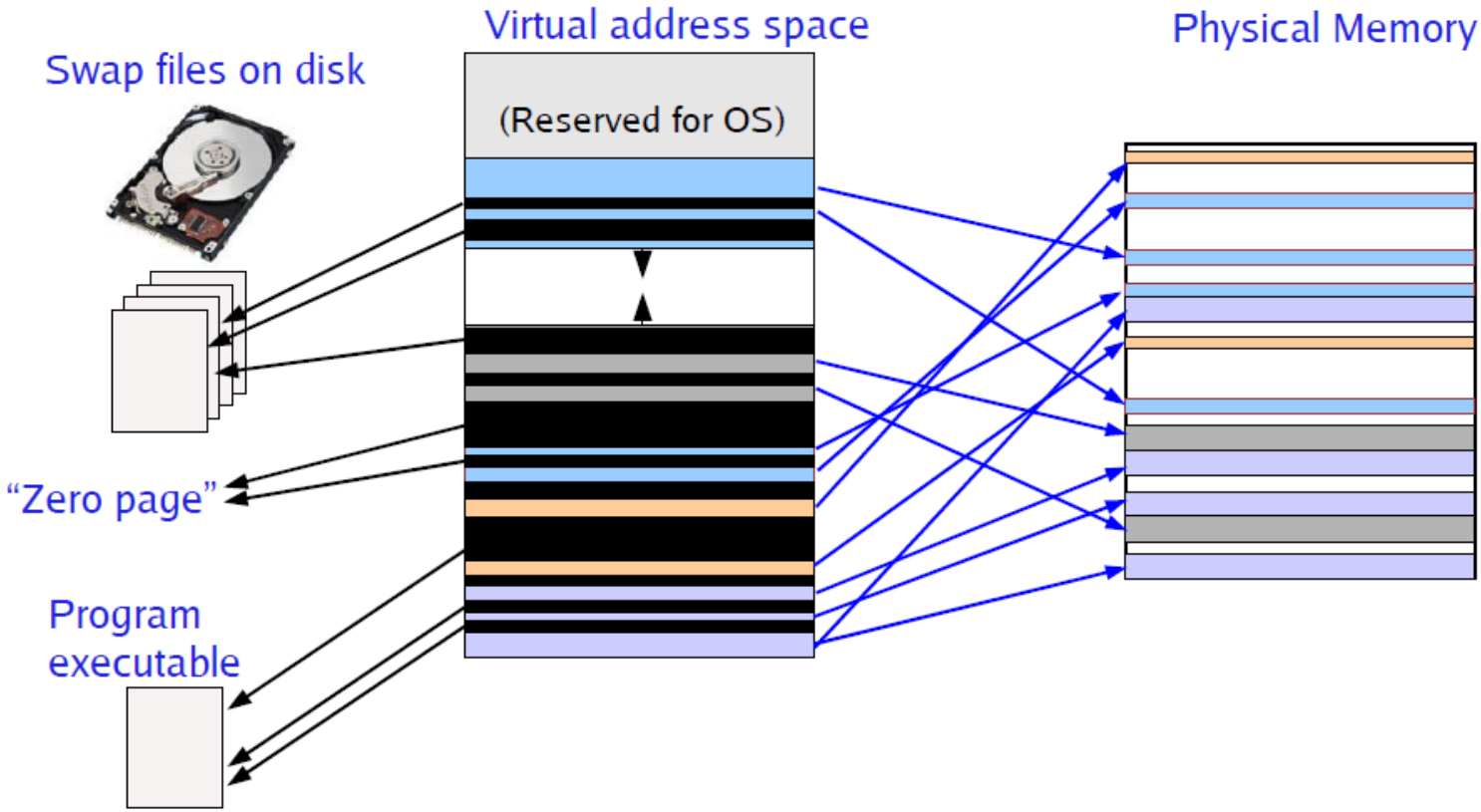


In Linux,

- 44 bytes of state maintained per page
  - If we have 4GB of RAM with each page being 4KB, we will have 2^20 pages
  - 44*2^20 pages = 44MB of storage

Swap Files

- What happens to the page that we choose to evict?
  - Depends on what kind of page it is and what state it's in!
- OS maintains one or more **swap files or partitions on disk**
  - Special data format for storing pages that have been swapped out



Page Eviction

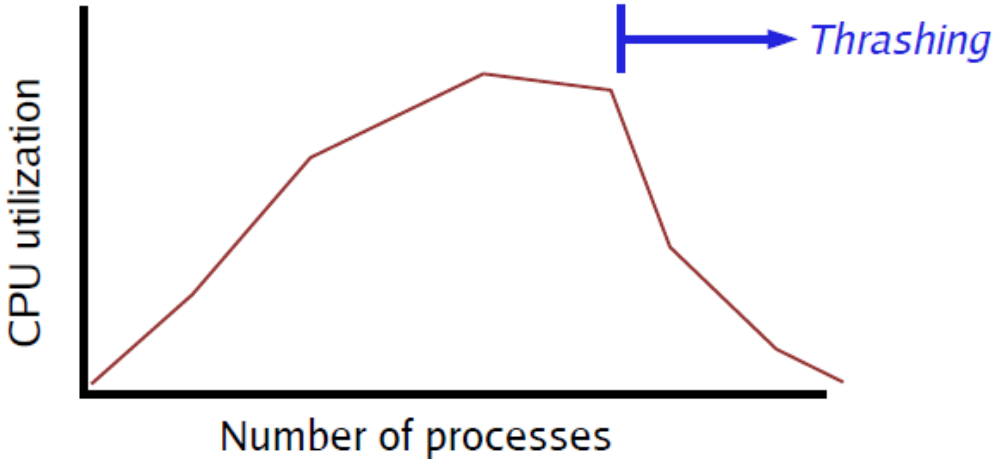
- How we evict a page depends on its type.
- Code page:
  - Just chuck it from memory – can recover it from the executable file on disk!
- Unmodified (*clean)* data page:
  - If the page has previously been swapped to disk, just chuck it from memory
    - *Assuming that page's backing store on disk has not been overwritten*
  - If the page has never been swapped to disk, allocate new swap space and write the page to it (This is just an optimization since swapping the page in is faster from swap space)
  - Exception: unmodified zero page – no need to write out to swap at all!
- Modified (*dirty) data page:*
  - If the page has previously been swapped to disk, write page out to the swap space
  - If the page has never been swapped to disk, allocate new swap space and write the page to it

Physical Frame Allocation

- How do we allocate physical memory across multiple processes?
  - When we evict a page, which process should we evict it from?
  - How do we ensure fairness?
  - How do we avoid one process hogging the entire memory of the system?
- Fixed-space algorithms
  - Per-process limit on the physical memory usage of each process
  - When a process reaches its limit, it evicts pages *from itself*
- Variable-space algorithms
  - Physical size of processes can grow and shrink over time
  - Allow processes to evict pages from other processes
- One process paging can impact performance of entire system!
  - One process that does a lot of paging will induce more disk I/O

Thrashing

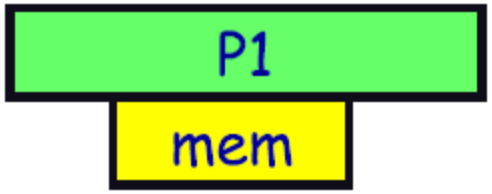
- As system becomes more loaded, spends more of its time paging
  - Eventually, no useful work gets done!
- System is overcommitted!
  - If the system has too little memory, the page replacement algorithm doesn't matter



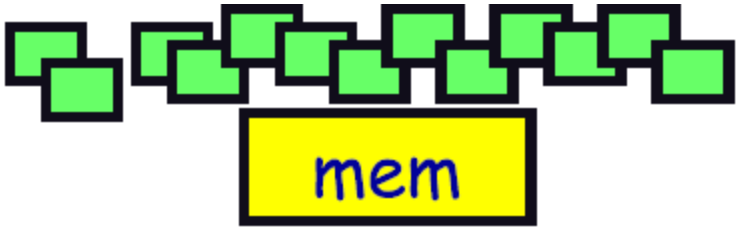
- Solutions?
  - Change scheduling priorities to “slow down” processes that are thrashing
  - Identify process that are hogging the system and kill them?

Reasons for Thrashing

- Process doesn't reuse memory, so caching doesn't work
  - (past != future)
- Process does reuse memory, but it does not "fit"



- Individually, all processes fit and reuse memory, but too many for system



  - This could be solved !

Dealing with Thrashing

Approach 1: Working set

- How much memory does the process need in order to make reasonable progress (its working set)?
- Only run processes whose memory requirements can be satisfied

Approach 2: Page Fault Frequency

- PFF = page faults / instructions executed
- If PFF rises above threshold, process needs more memory
  - Not enough memory on the system? Swap out.
- If PFF sinks below threshold, memory can be taken away

Working Set

- A process's *working set is the set of pages that it currently "needs"*
- Definition:
  - WS(P, t, w) = the set of pages that process P accessed in the time interval [t-w, t]
  - "w" is usually counted in terms of number of page references
    - *A page is in WS if it was referenced in the last w page references*
- Working set changes over the lifetime of the process
  - Periods of high locality exhibit **smaller working set**
  - Periods of low locality exhibit **larger working set**
- Basic idea: Give process enough memory for its working set
  - If WS is larger than physical memory allocated to process, it will tend to swap
  - If WS is smaller than memory allocated to process, it's wasteful
  - This amount of memory grows and shrinks over time

Estimating the Working Set

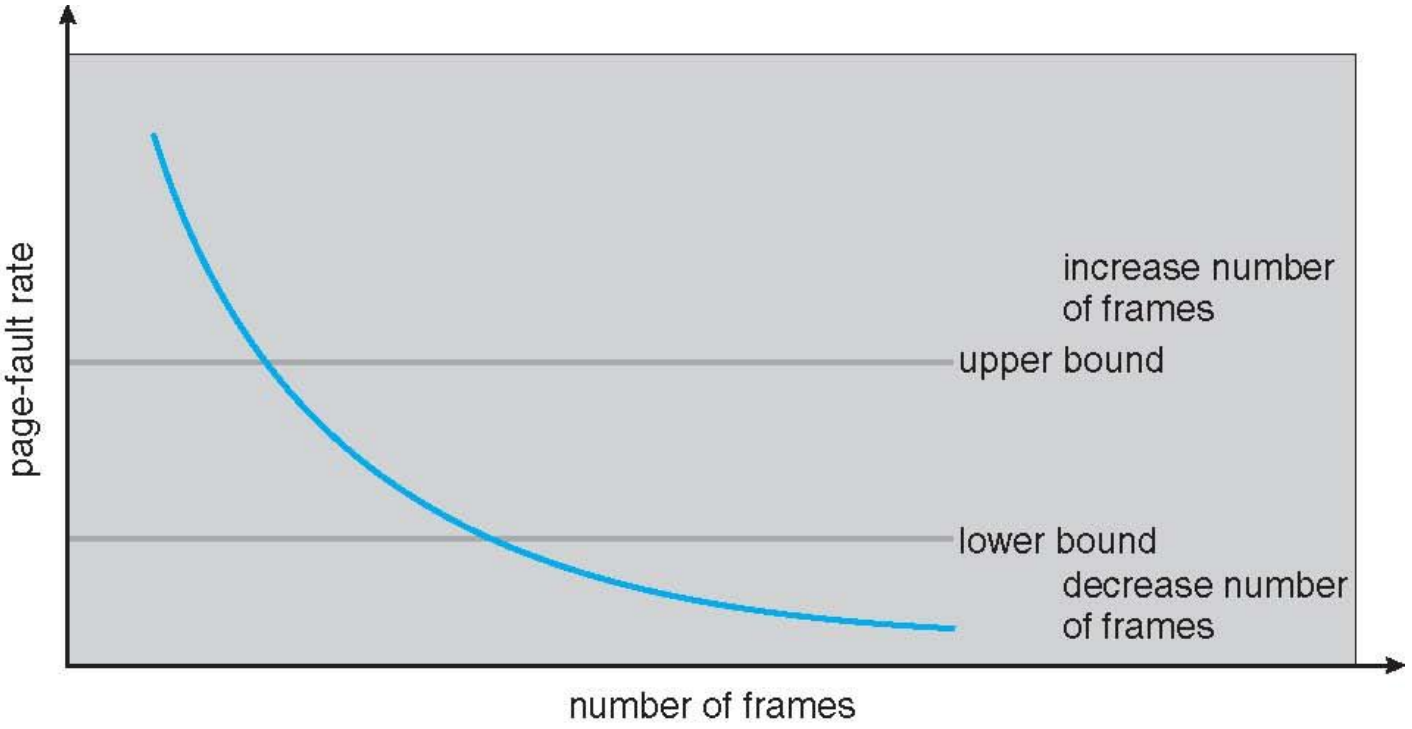
- How do we determine the working set of a process?
- Simple approach
  - Approximate with interval timer + a reference bit
- Example: t = 10,000 instructions
  - Interrupts after every 5000 instructions.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts, shift the bits to right and copy the reference bit value onto the high order bit and sets the values of all reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
  - Not sure when exactly in the last 5000 time units was this page accessed
- Improvement = 10 bits and interrupt every 1000 instructions.

Working Set

- Now that we know the working set, how do we allocate memory?
  - If working sets for all processes fit in physical memory, done!
  - Otherwise, reduce memory allocation of larger processes
    - Idea: Big processes will swap anyway, so let the small jobs run.
  - Very similar to shortest-job-first scheduling: give smaller processes better chance of fitting in memory
- How do we decide the working set limit T?
  - If T is too large, very few processes will fit in memory
  - If T is too small, system will spend more time swapping

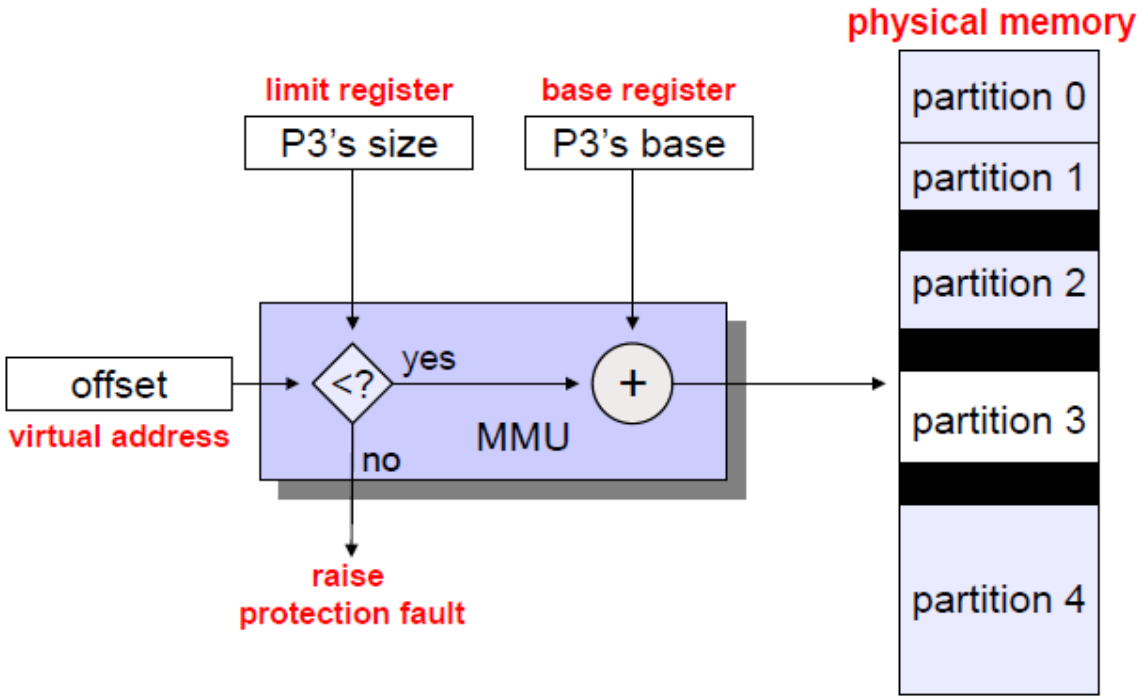
Page-Fault Frequency Scheme

- Page Fault Rate = (#Page Faults)/No of Executed Instructions
- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame (or is swapped out)



Variable Partitions – Remember ?

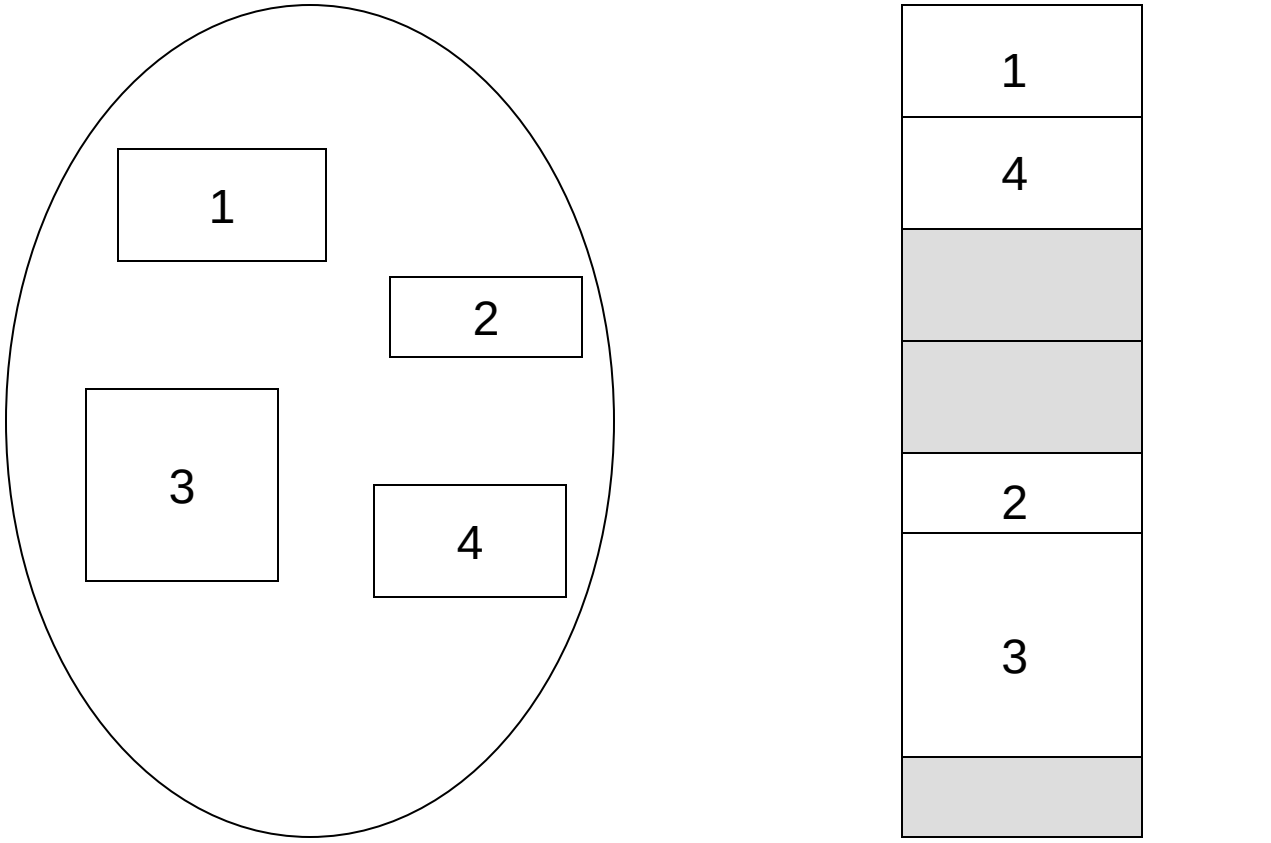
- Allow variable sized partitions
  - Now requires both and *base* and a *limit* register
- Problem with segmentation: external fragmentation
  - Holes left in physical memory when segments are destroyed



Segmentation

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - functions
  - local variables, global variables,
  - common block,
  - stack,
  - heap
  - symbol table, arrays

Logical View of Segmentation

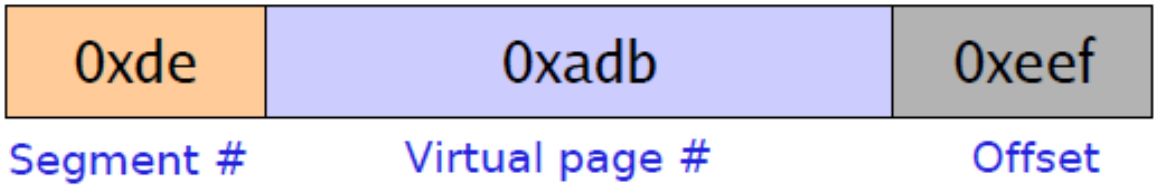


Why use segments?

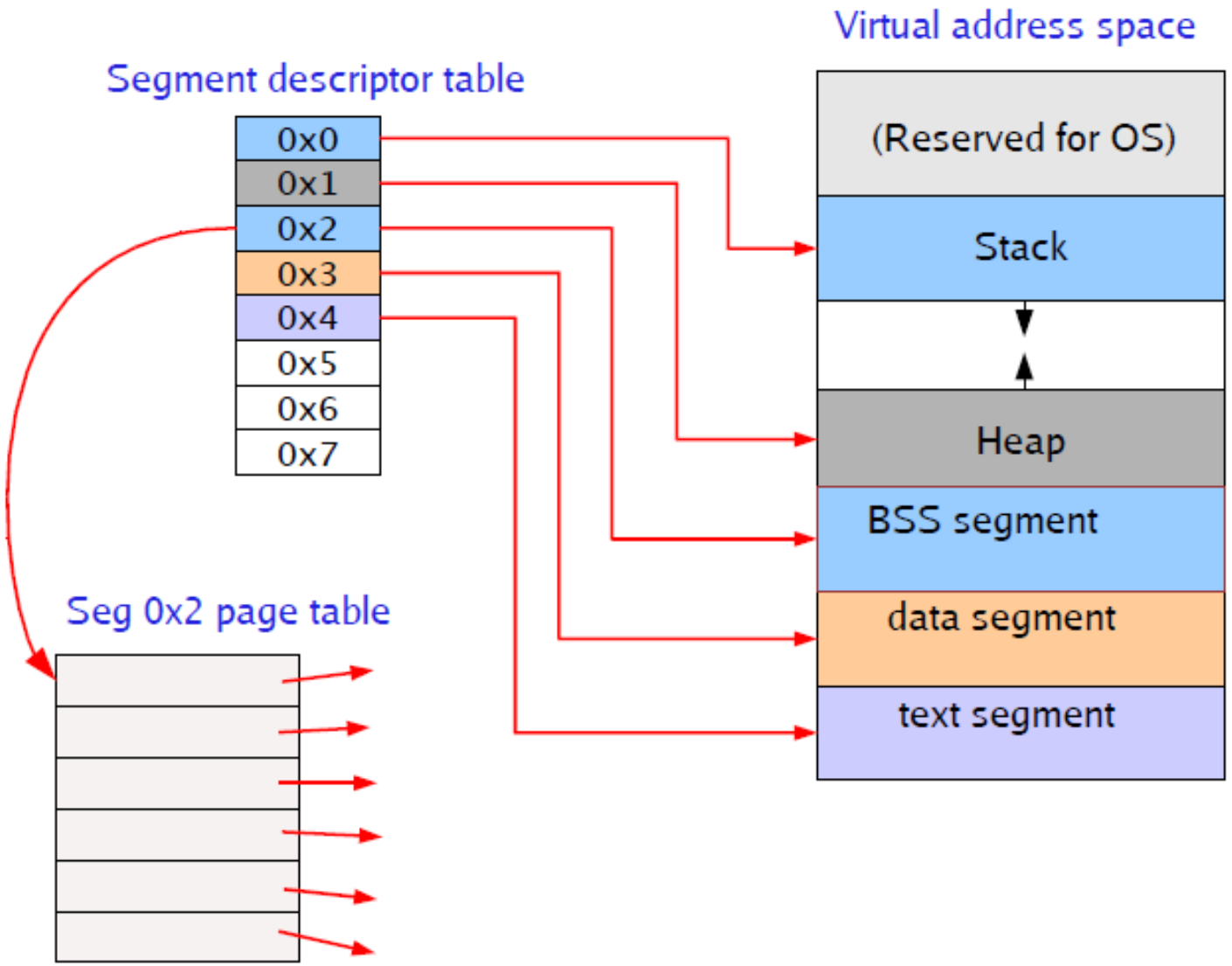
- Segments cleanly separate different areas of memory
  - e.g., Code, data, heap, stack, shared memory regions, etc.
  - Use different segment registers to refer to each portion of the address space
- Allows hardware to enforce protection on a segment as a whole
  - e.g., Segment descriptor can mark the entire code segment as read only
- Note that using page tables can accomplish the same result...
  - But requires the ***OS to carefully maintain*** page table entries for entire "segments"

Combined Segmentation and Paging

- A segment is a contiguous span of *virtual addresses*
  - ... rather than physical addresses, as on the previous slide
  - Described by a *segment descriptor*
  - Segment descr has total segment size, access rights, base virtual address
- Each segment can have a corresponding page table!
  - Segment broken into pages internally
  - Can use either one- or two-level paging on each segment
- Virtual address now looks like:
  - Segment number may be part of the address, or stored in a separate register
  - Value of current segment register used to determine which segment to access



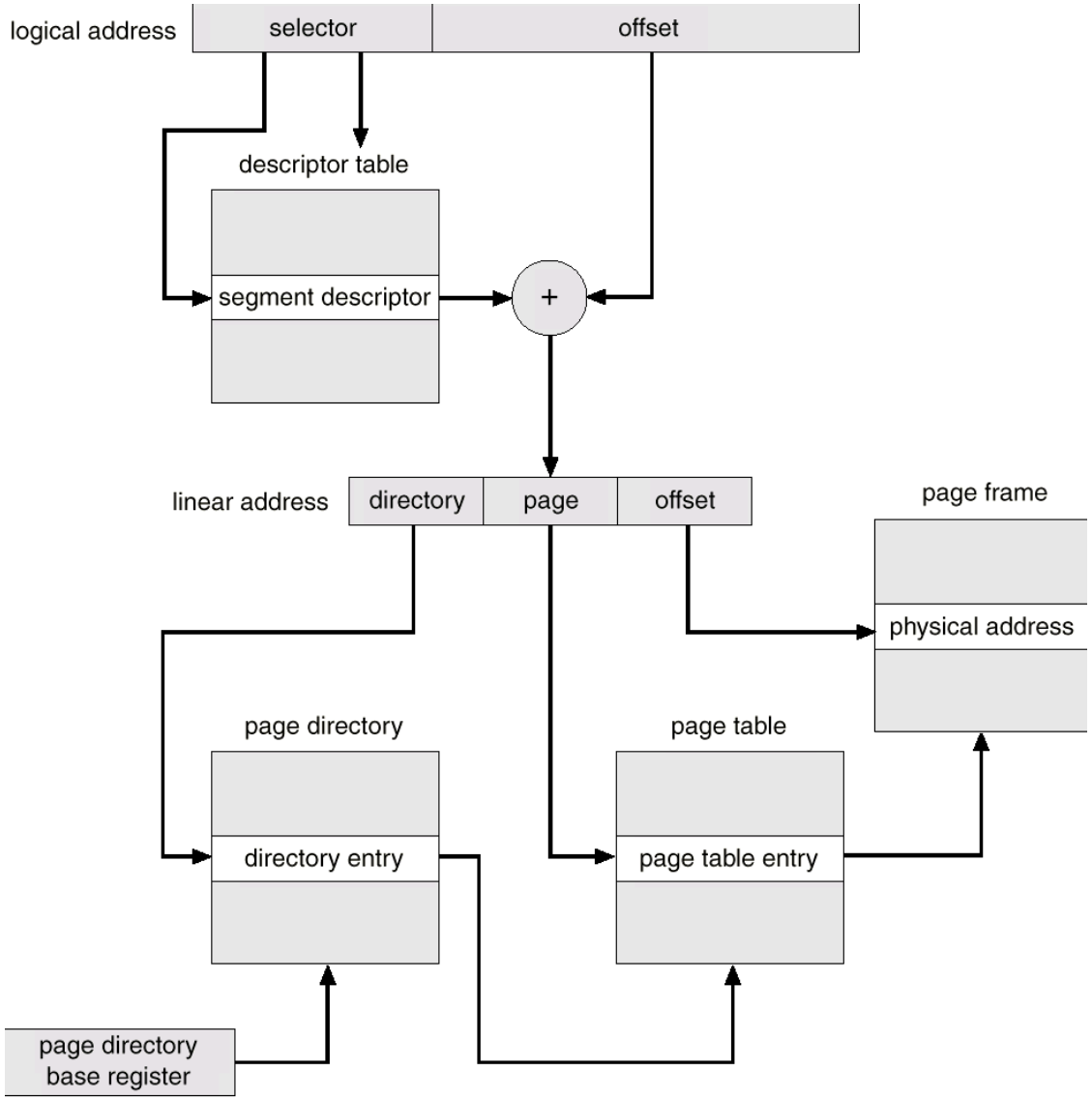
Virtual address space with segments



Intel X86 Segments

- Multiple *segment registers*
  - CS: Code Segment
  - DS: Data Segment
  - SS: Stack Segment
  - Also ES, FS, and GS ... “other” segments
- Each instruction uses one of these segment registers
  - For example, instruction fetch implicitly uses segment pointed to by CS
  - Push/pop instructions implicitly use segment pointed to by SS
- Segment descriptor information:
  - Virtual base address and size of segment
  - Segment access rights (read, write, execute)
- All segments share the same linear address space!
  - This means there is **one set of page tables for all segments in a process**
  - Segments can overlap in linear address space, too.

Intel X86 address translation



Logical/Virtual address

↓

Linear address

↓

Physical address

Do we need Segmentation and Paging ?

- Short answer: You don't – just adds overhead
  - Most Operating systems use "flat mode" – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
- Linux x86: One segment for user code, another segment for user data
  - Both segments cover the same virtual address range! (0 ... 3GB)
  - Another pair of segments for kernel virtual addresses (3GB ... 4GB)

Summary

- Virtual memory is a way of introducing another level in our memory hierarchy in order to abstract away the amount of memory actually available on a particular system
  - This is incredibly important for "ease-of-programming"
  - Imagine having to explicitly check for size of physical memory and manage it in each and every one of your programs
- It's also useful to prevent fragmentation in multi-programming environments
- Can be implemented using paging (sometime segmentation or both)
- Page fault is expensive so can't have too many of them
  - Important to implement good page replacement policy
- Have to watch out for thrashing!!