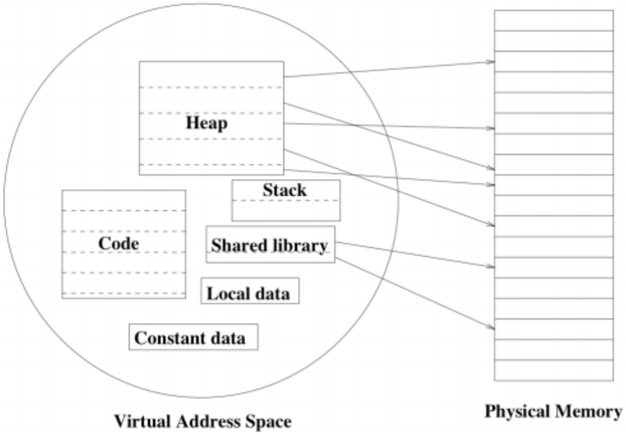
Last Class: Paging & Segmentation

- Paging: divide memory into fixed-sized pages, map to frames (OS view of memory)
- Segmentation: divide process into logical ‘segments’ (compiler view of memory)
- Combine paging and segmentation by paging individual segments



Background

- Up to now, the virtual address space of a process fit in memory, and we assumed it was all in memory.

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual Memory

- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

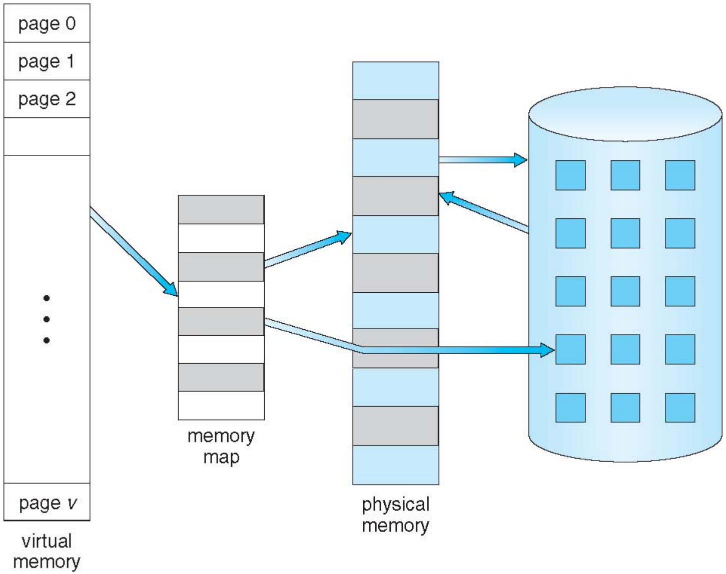
Virtual Memory (cont.)

- Virtual address space – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paged Virtual Memory

- Demand Paging uses a memory as a cache for the disk
- The page table indicates if the page is on disk or memory using a valid bit
- Once a page is brought from disk into memory, the OS updates the page table and the valid bit
- For efficiency reasons, memory accesses must reference pages that are in memory the vast majority of the time
    - Else the effective memory access time will approach that of the disk
- Key Idea: Locality --- the working set size of a process must fit in memory and must stay there (90/10 rule)

Demand Paged Virtual Memory



When to load a page?

- At process start time: the virtual address space must be no larger than the physical memory.
- Demand paging: OS loads a page the first time it is referenced.
  - May remove a page from memory to make room for new page
  - Process must give up the CPU while the page is being loaded
  - Page Fault: interrupt that occurs when an instruction references a page that is not in memory
- Pre-paging: OS guesses in advance which pages the process will need and pre-loads them into memory
  - Allows more overlap of CPU and I/O if the OS guesses correctly
  - If the OS is wrong => page fault
  - Errors may result in removing useful pages
  - Difficult to get right due to branches in code

present bit

Valid-Invalid Bit

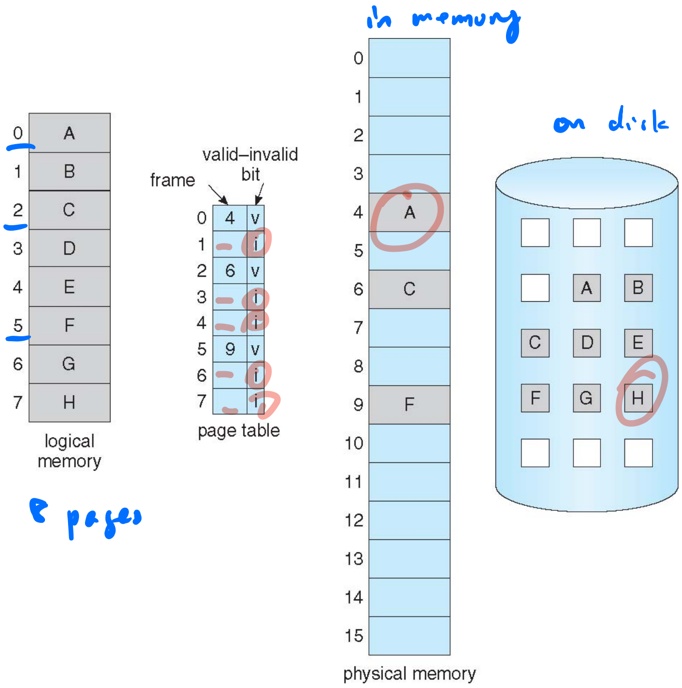
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – memory resident, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
2	v
6	v
4	v
–	i
. . .	
–	i
–	i

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



Page Fault

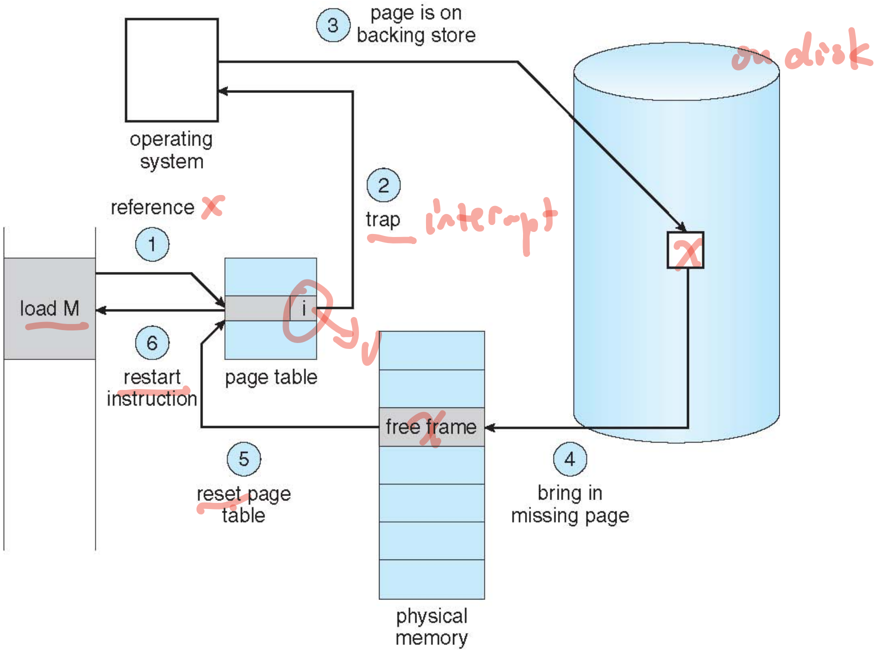
reference page 0 → hit
page 7 (i) → page fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Stages in Demand Paging

- When referenced, if the page is not in memory, trap to the OS
- The OS checks that the address is valid. If so, it
  1. Select a page to replace (page replacement algorithm)
  2. Invalidates the old page in the page table
  3. Starts loading new page into memory from disk
  4. Context switches to another process while I/O is being done
  5. Gets interrupt that page is loaded in memory
  6. Update the page table entry
  7. Continues faulting process

Swap Space

- What happens when a page is removed from memory?
  - If the page contained code, we could simply remove it since it can be reloaded from the disk
  - If the page contained data, we need to save the data so that it can be reloaded if the process it belongs to refers to it again
  - Swap space: a portion of the disk is reserved for storing pages that are evicted from memory
- At any given time, a page of virtual memory might exist in one or more of:
  - The file system HD
  - Physical memory mem
  - Swap space HD

Performance of Demand Paging

no PF
PF

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Theoretically, a process could access a new page with each instruction
- Fortunately, processes typically exhibit *locality of reference*
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

no PF

EAT = $(1 – p)$ x memory access (page already in mem)
+ p (page fault overhead
+ swap page out
+ swap page in) disk accesses

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

disk access

What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc.
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

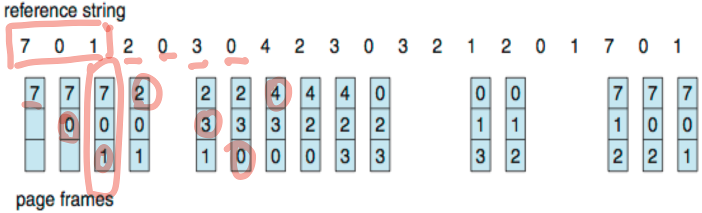
Page Replacement Algorithms

- On a page fault, we need to choose a page to evict
- FIFO: First-In, First-Out. Throw out the oldest page. Simple to implement, but the OS can easily throw out a page that is being accessed frequently
- MIN: (a.k.a. OPT). Throw out the page that will not be accessed for the longest time
- LRU: Least Recently Used. Approximation of MIN that works well if the recent past is a good predictor of the future. Throw out the page that has not been used in the longest time.

First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

cold-start miss a.k.a. compulsory miss



15 page faults

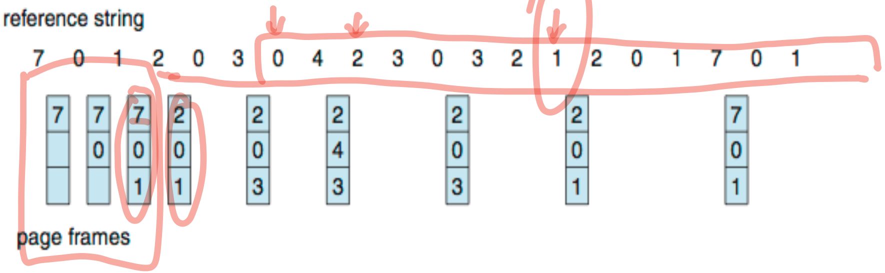
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - Belady's Anomaly
- How to track ages of pages?
  - Just use a FIFO queue

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can’t read the future

future

- Used for measuring how well your algorithm performs

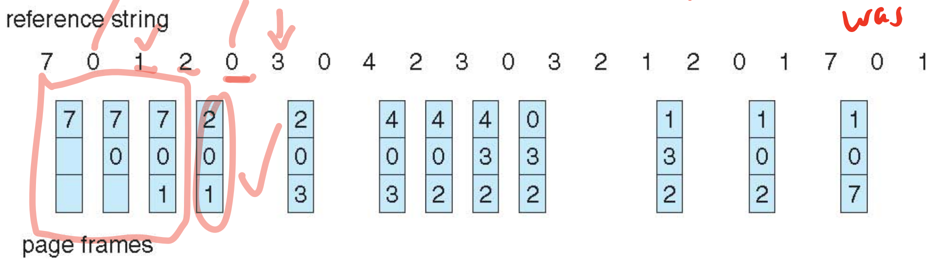


Least Recently Used (LRU) Algorithm

FIFO: time when a page was loaded

LRU: time when a page was referenced

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



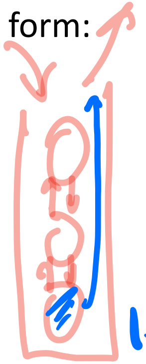
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

accessed time

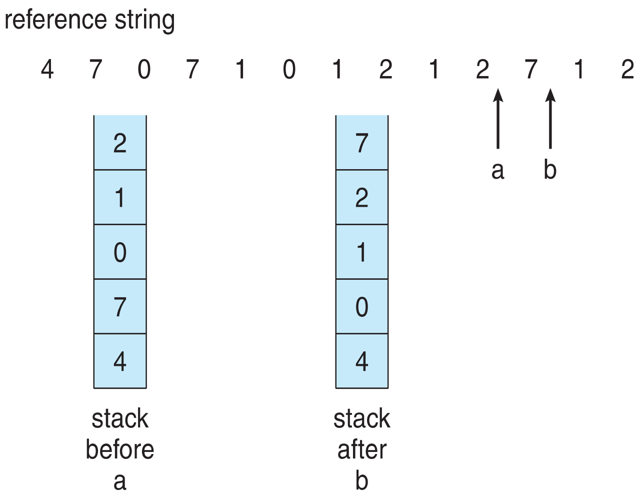
timestamp

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement



least recent

Use Of A Stack to Record Most Recent Page References



Adding Memory: FIFO is not a stack algorithm

- Does adding memory always reduce the number of page faults?

FIFO:

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	A	A	A	D	D	D	E			E	E	
frame 2		B	B	B	A	A	A			C	C	
frame 3			C	C	C	B	B			B	D	
frame 1	A	A	A	A			E	E	E	E	D	D
frame 2		B	B	B			B	A	A	A	A	E
frame 3			C	C			C	C	B	B	B	B
frame 4				D			D	D	D	C	C	C

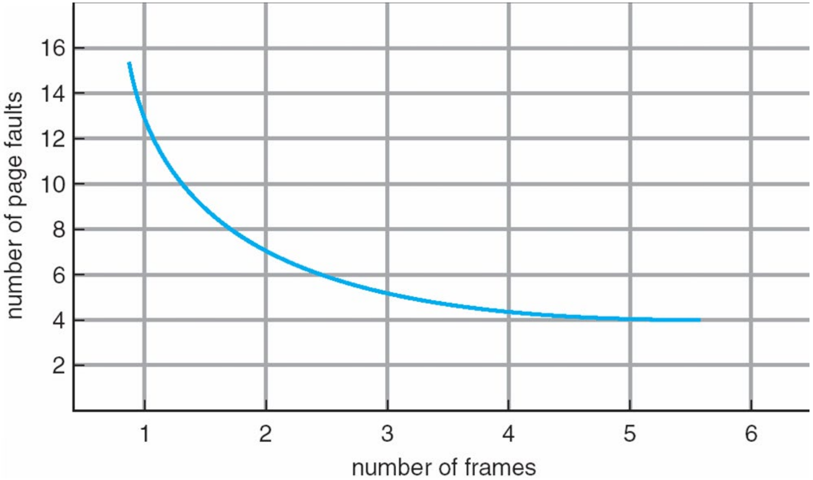
9 PF

10 PF

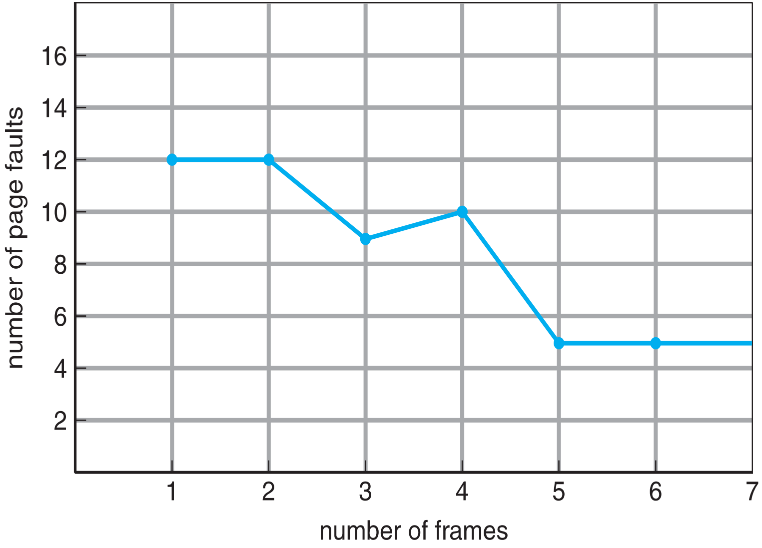
Adding Memory: FIFO

- Does adding memory always reduce the number of page faults?
- Belady’s Anomaly: Adding page frames may actually cause more page faults with certain types of page replacement algorithms (such as FIFO).

Graph of Page Faults Versus The Number of Frames



FIFO Illustrating Belady's Anomaly



Adding Memory: LRU

Stack property: A memory of size N+1 naturally includes the contents of a memory of size N.

LRU:

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	**A***	A	A	**D***	D	D	**E***	E	E	**C***	C	C
frame 2		**B***	B	B	**A***	A	A	A	A	A	**D***	D
frame 3			**C***	C	C	**B***	B	B	B	B	B	B
frame 1	**A***	A	A	A	A	A	A	A	A	A	A	**E***
frame 2		**B***	B	B	B	B	B	B	B	B	B	B
frame 3			**C***	C	C	C	**E***	E	E	E	**D***	D
frame 4				**D***	D	D	D	D	D	**C***	C	C

is a subset of

9 PF

8 PF

- With LRU, increasing the number of frames always decreases the number of page faults.

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- Second-chance algorithm
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- Lease Frequently Used (LFU) Algorithm: replaces page with smallest count
- Most Frequently Used (MFU) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

We skipped this part.

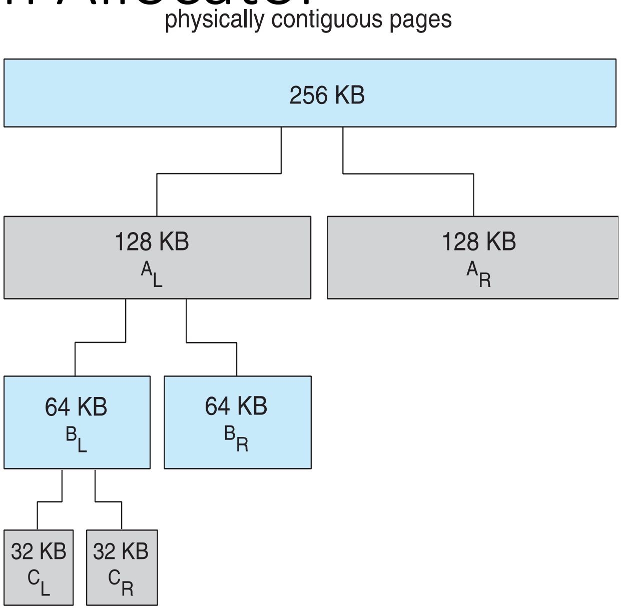
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

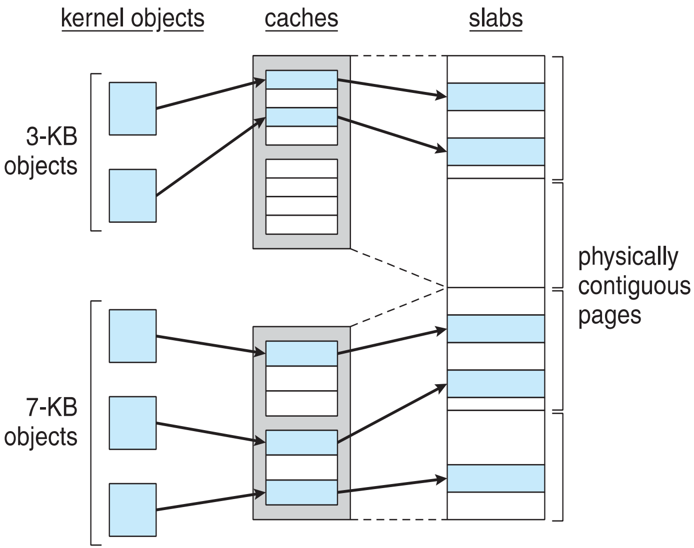
Buddy System Allocator



Slab Allocator

- Alternate strategy
- Slab is one or more physically contiguous pages
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with objects – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
  - One thing you learned in this lecture
  - One thing you didn’t understand