Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Virtual Memory-Allocation and Thrashing

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Spring 2022

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- **Two major allocation schemes**
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation
  - For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool

Fixed Allocation

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Global vs. Local Allocation

- **Global replacement**
  - Process selects a replacement frame from the set of all frames; one process can take a frame from another

- **Local replacement**
  - Each process selects from only its own set of allocated frames

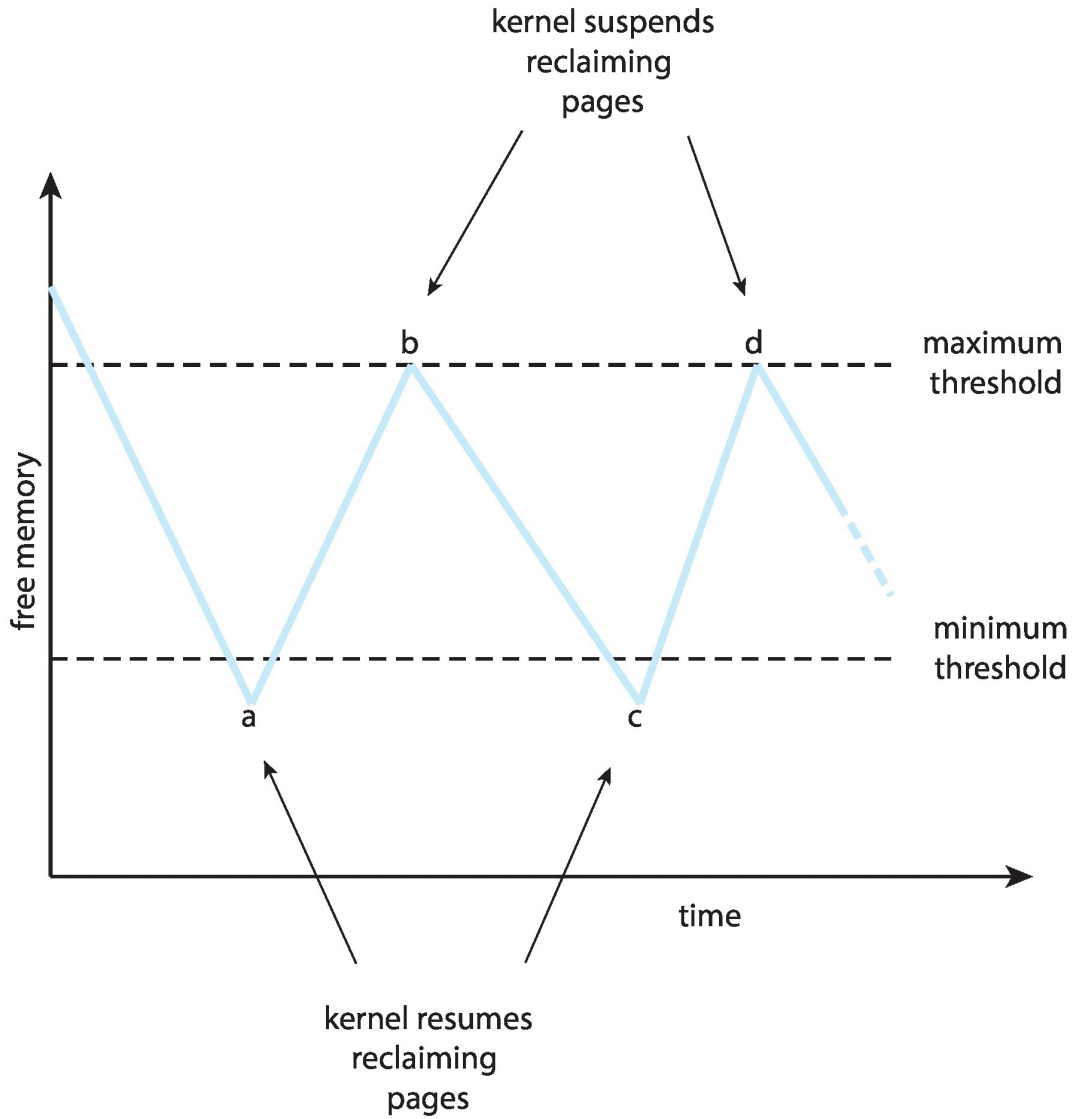
Global vs. Local Allocation (cont.)

Allocation type	Performance (e.g., execution time)	Throughput (utilization)
Global	can vary greatly	greater throughput, so more common
Local	more consistent per-process performance	possibly underutilized memory

Reclaiming Pages

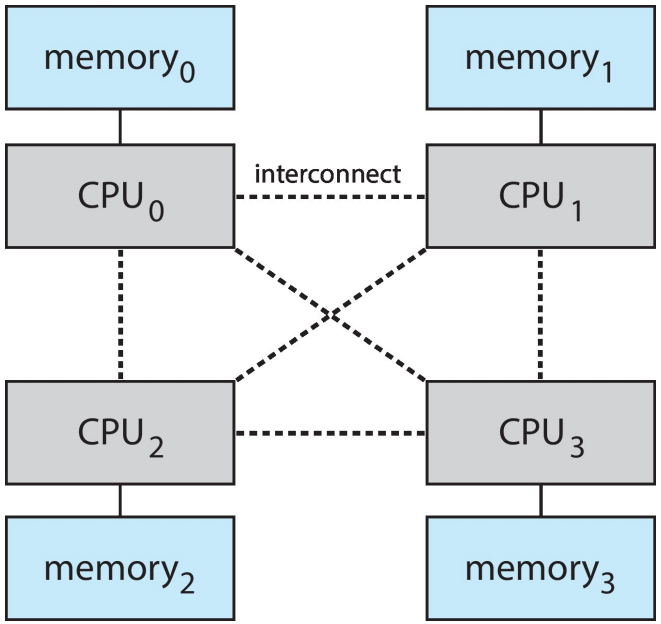
- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list.
- We begin selecting pages for replacement when the list falls below a certain threshold.
  - Rather than waiting for the list to drop to zero.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example



Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



Non-Uniform Memory Access (cont.)

- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible.
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Thrashing

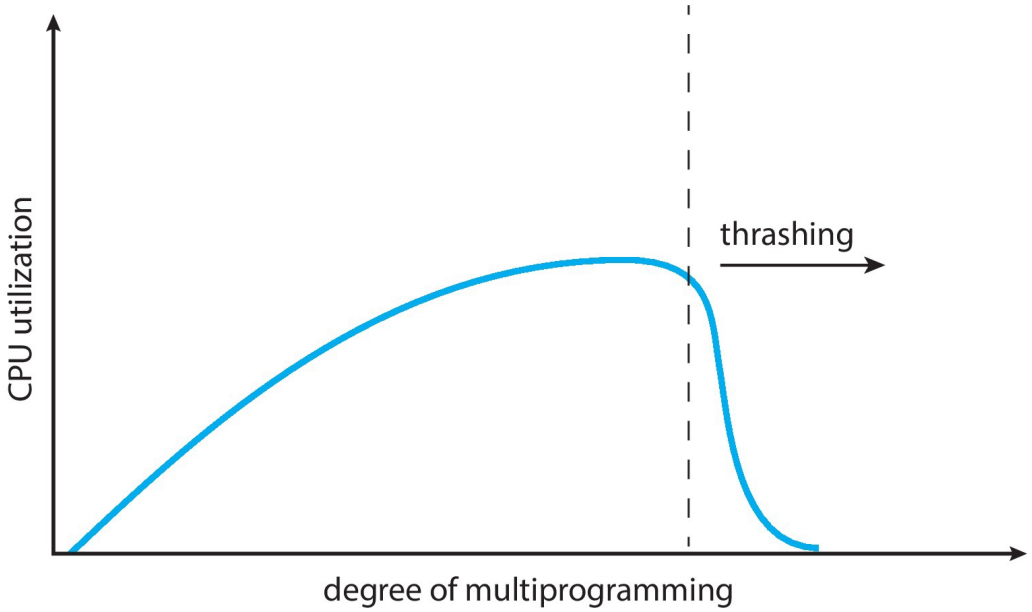
- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

Thrashing (cont.)

- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system

Thrashing (cont.)

- **Thrashing.** A process is busy swapping pages in and out



Demand Paging and Thrashing

- Why does demand paging work?
  - **Locality model**
    - A locality is a set of pages that are actively used together.
    - Process migrates from one locality to another
    - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size
- Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern

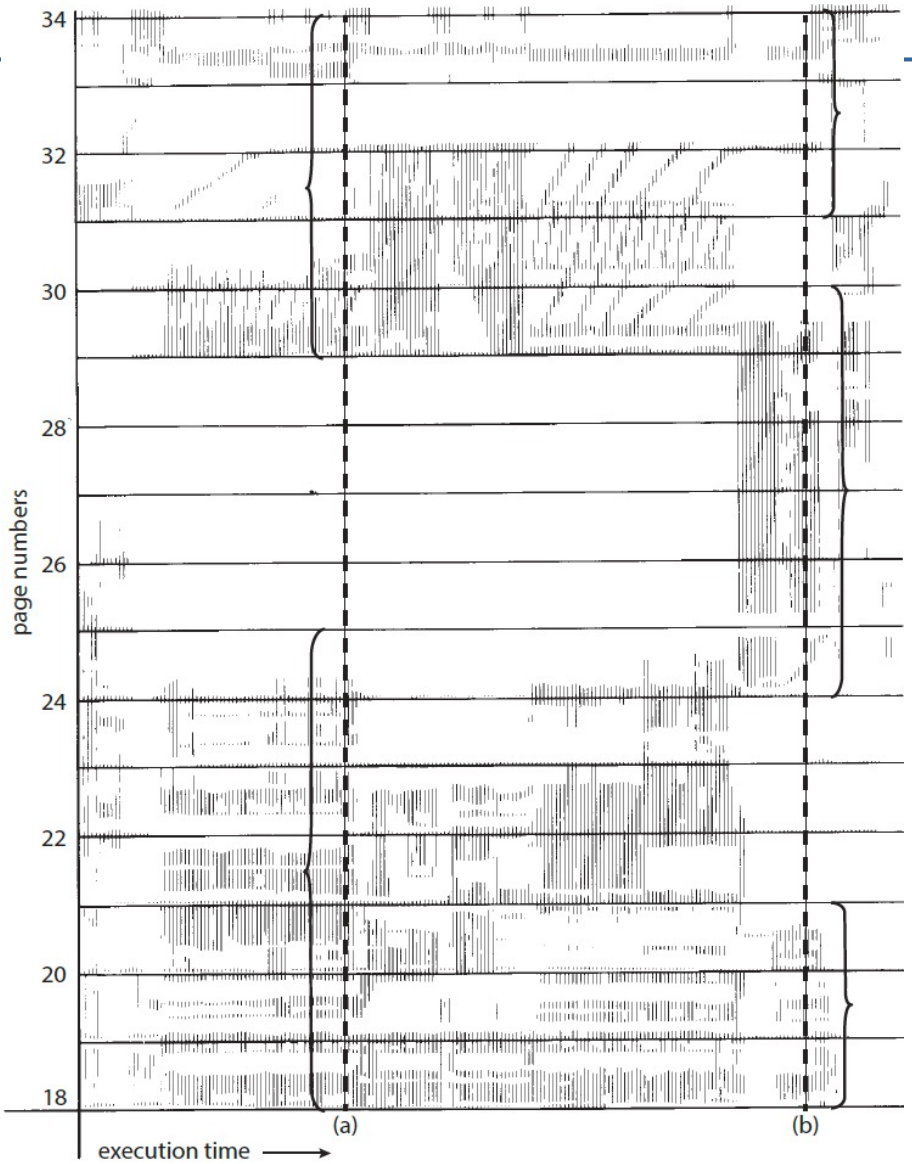
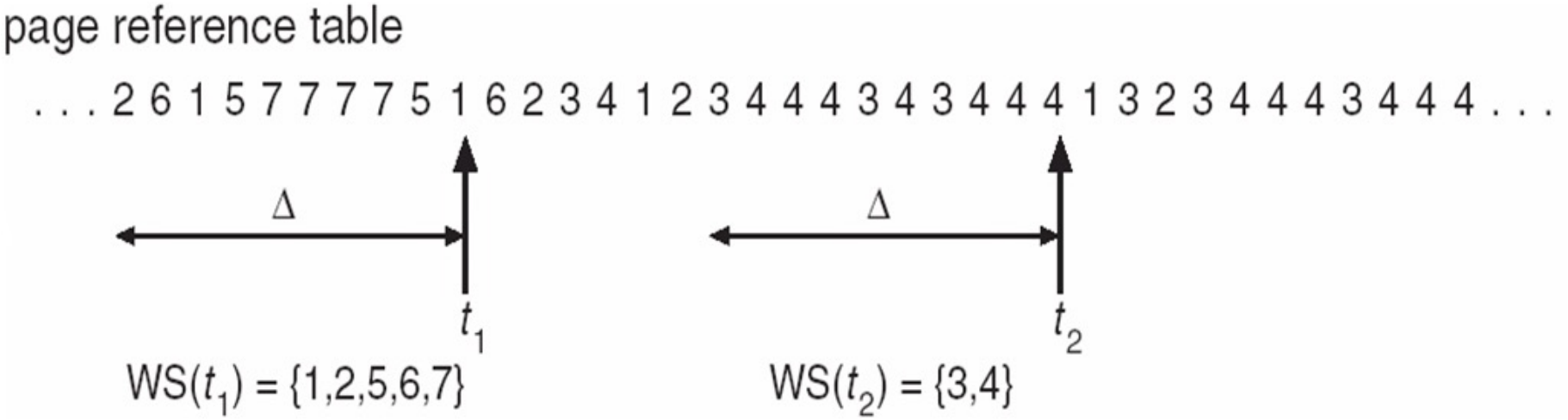


Figure 10.21 Locality in a memory-reference pattern.

Working-Set Model

- $\Delta \equiv$ working-set window
  - a fixed number of page references (e.g., 10,000 instructions)

- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)



Working-Set Model (cont.)

- if Δ too small, WSS_i will not encompass entire locality
- if Δ too large, WSS_i will encompass several localities
- if $\Delta = \infty \Rightarrow WSS_i$ will encompass entire program

- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

- if $D > m \Rightarrow$ Thrashing
- **Policy:** if $D >$ m, then suspend or swap out one of the processes

Working-Set Model (cont.)

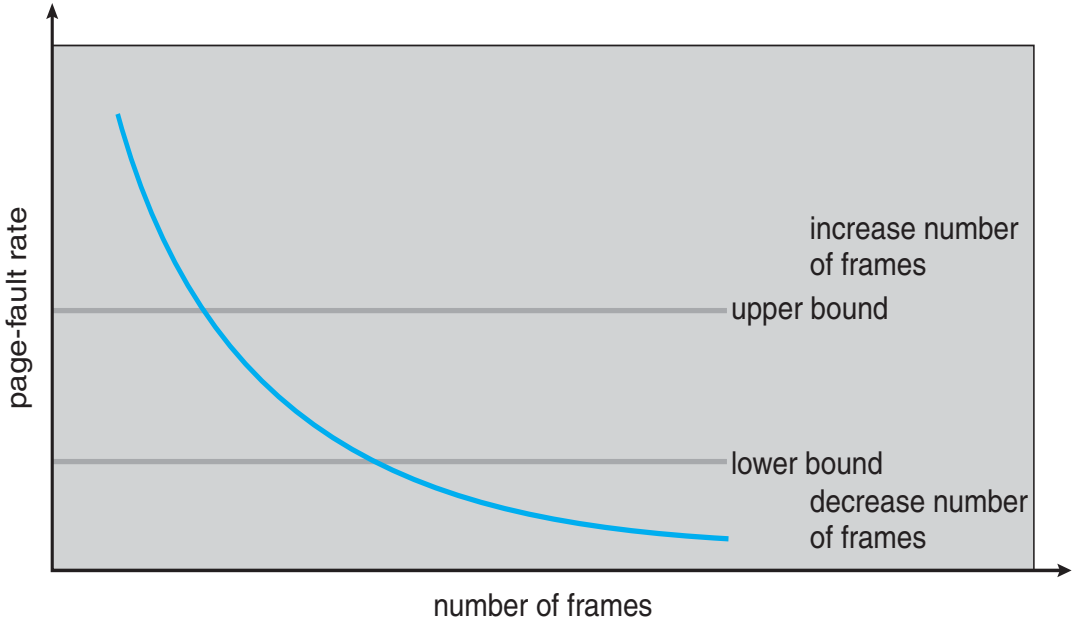
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.
- Thus, it optimizes CPU utilization.
- The difficulty with the working-set model is keeping track of the working set.

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy 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?
- Improvement = 10 bits and interrupt every 1000 time units

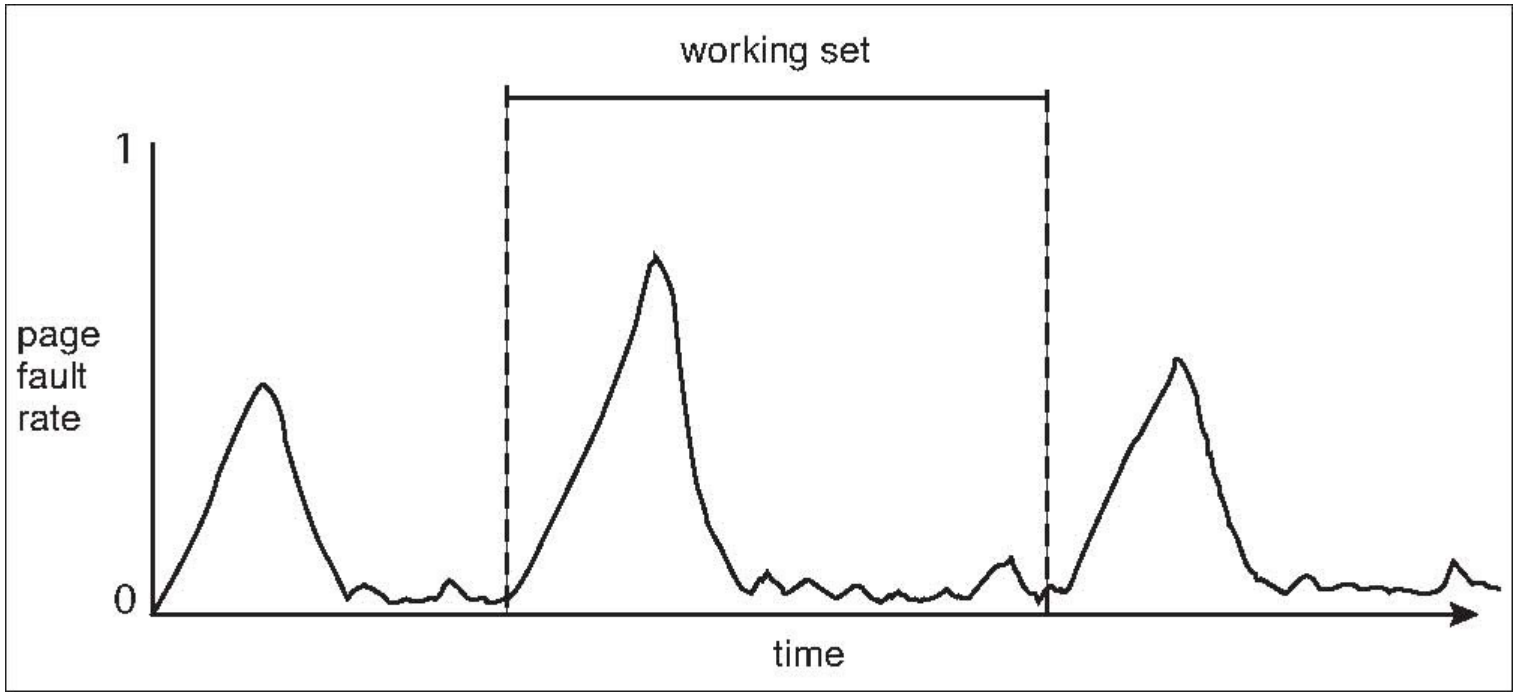
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Allocating Kernel Memory

- Treated differently from user memory

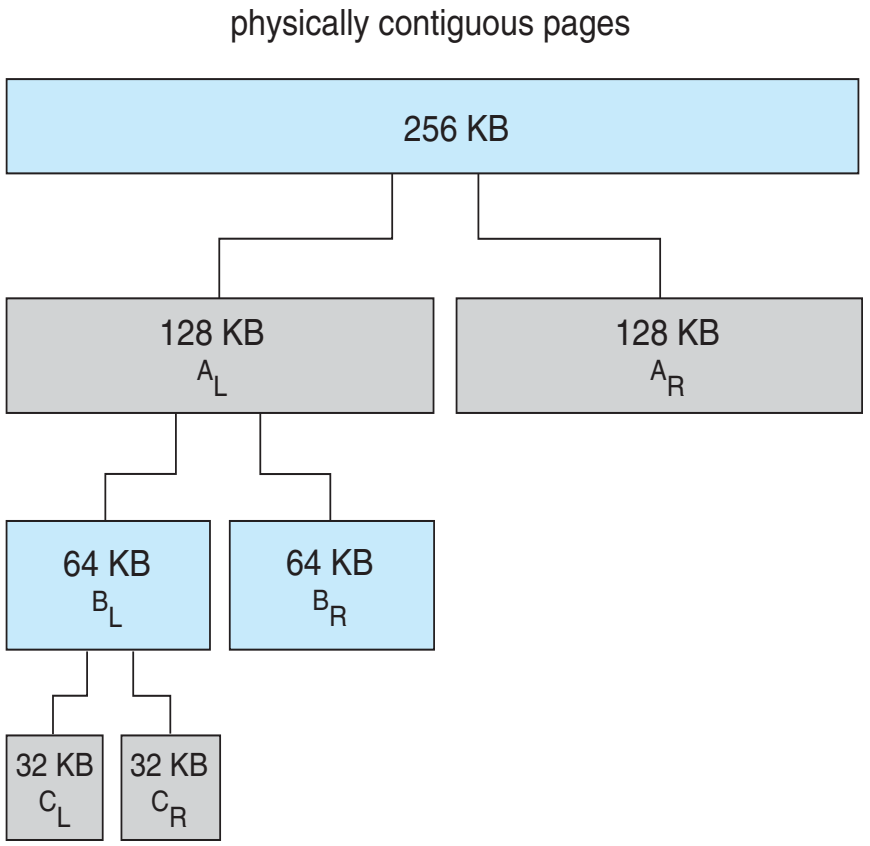
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - i.e., for device I/O

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

Buddy System

- 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



Buddy System

- Advantage
  - Quickly **coalesce** unused chunks into larger chunk
- Disadvantage
  - Fragmentation
  - In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation.

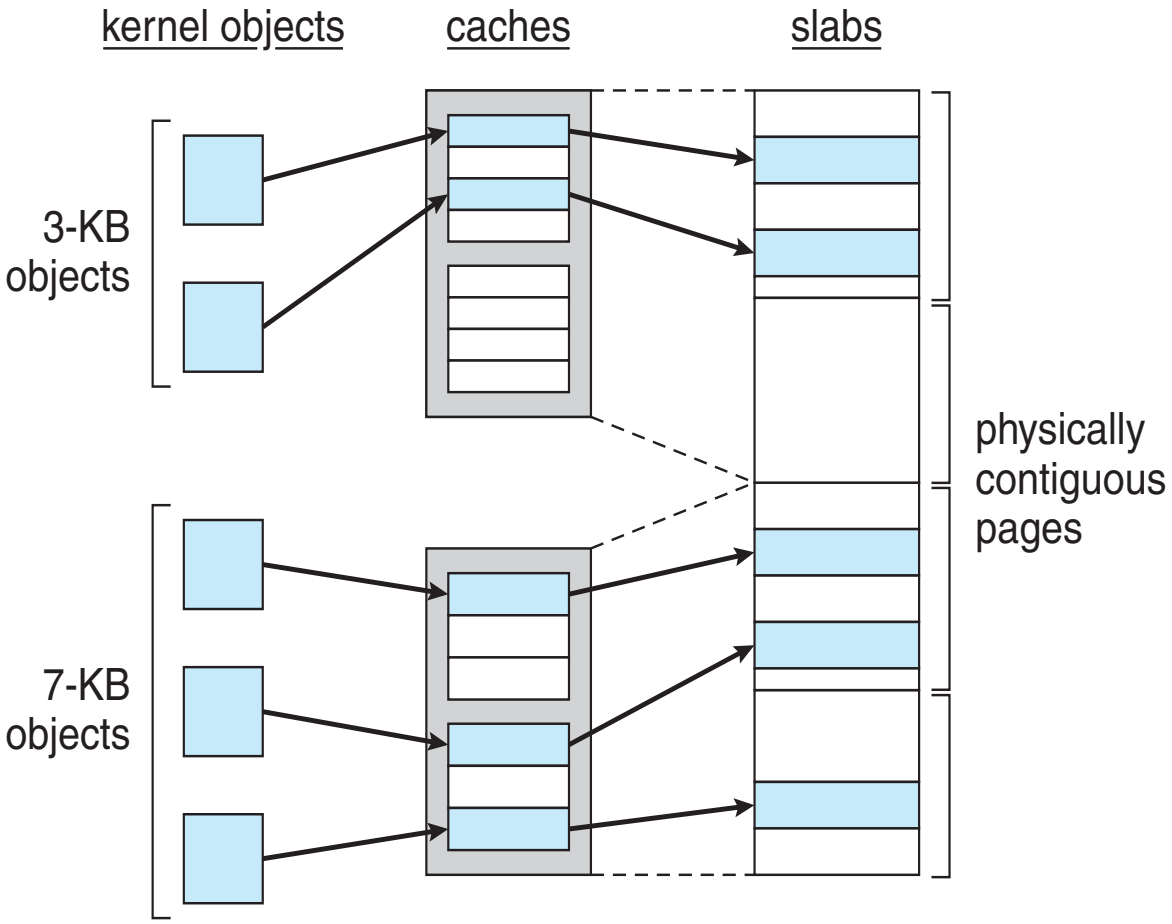
Slab Allocator

- 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

Slab Allocator

- 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



Slab Allocator in Linux

- For example process descriptor is of type struct task_struct
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free struct task_struct

Slab Allocator in Linux

- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various Oses.

- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure.

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking