Operating Systems: Virtual Memory – Part II

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 10)

Page-Buffering Algorithms

- Page buffering algorithms used in *conjunction* with page replacement algorithms to improve performance.
- Maintain a fixed minimum number of free frames (called free-frame pool) at all times. When page fault occurs
  - Select a victim frame (as before).
  - Read the new page into a free frame in the free frame pool, **before moving the victim page out.**
    - Results in the process causing the page fault restarted faster.
  - When convenient, evict victim page and add its frame to the free-frame pool.
- Note that the **frame numbers in the pool can vary**.

Allocation of Frames

- Various strategies/algorithms available to allocate frames to processes (after allocating frames to OS)
  - **Equal allocation -** Allocate free frames equally among processes
  - **Proportional allocation -** Allocate frames to each process according to its size
- Each process needs ***minimum*** **number of frames** to execute.
  - This is defined by the computer architecture.
- **Global Vs local replacement**
- Gobal 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

Thrashing

Thrashing – is a situation in which the system is busy swapping pages of a process in and out, instead of executing its instructions on the CPU.

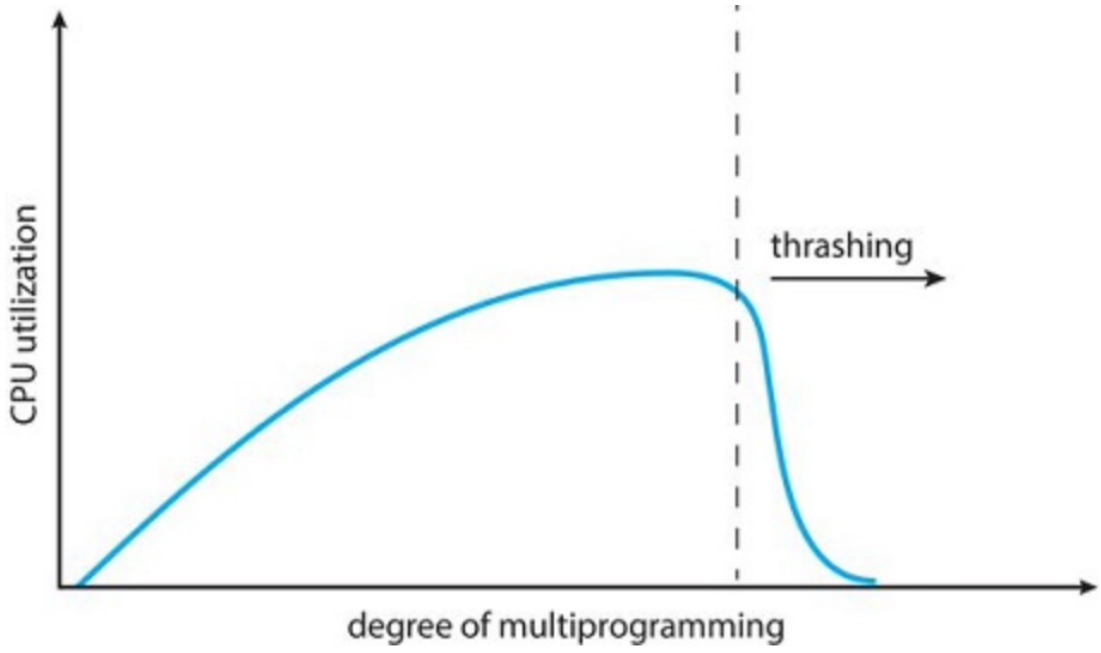
How does Thrashing occur in a system?

A process needs a certain of frames to support the active pages (pages in use). If it does not have the required number of frames, it will

- page fault to get the desired page in memory
  - As a result, it will replace a page in an **existing frame**
- But quickly needs to replace a page again from a frame to bring in the replaced page!

Thrashing

- As the thrashing processes wait for the pages to be swapped in and out, CPU utilization drops sharply.
- As CPU utilization plummets, OS thinking that it needs to increase the degree of multiprogramming.
- Another process added to the system, thus worsening the problem!



How to Prevent Thrashing? Cont…

To prevent thrashing, we must provide a process with as many frames as it needs for all its active pages.

How do we know how many frames it “needs”?

- Working-set Model
- Page Fault Frequency

Working-Set Model

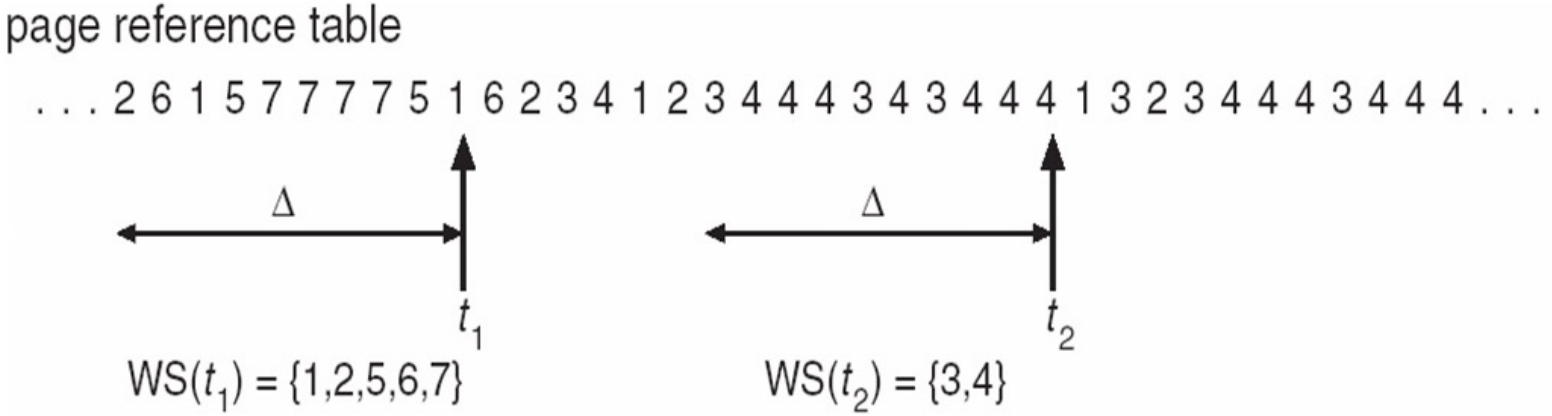
- **Working-set Model**: Based on locality model
  - **Locality model** – states that as a process executes, it moves from locality to locality.
  - A **locality** is a set of pages that are actively used together
- A program is generally composed of several different localities, which may overlap.
- To avoid thrashing enough frames should be allocated to accommodate the size of the current locality

Working-Set Model

- The working set model uses a parameter Δ for each process.
- Δ - the number of recent memory references
- working-set (WS) = set of recently accessed pages.
  - If a page is in active use, it will be in the working set.
  - If it is no longer being used, it will drop from the working set Δ time units after its last reference.

Working-Set Model Example

- For the given sequence of page references below
  - If Δ = 10 memory references
  - Working set (WS, set of recently accessed pages) at time t_1 is {1, 2, 5, 6, 7}.
  - By time t_2, the working set has changed to {3, 4}.



Working-Set Model

- WSS_i (working set size of Process P_i)
- m is the total number of available frames
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- If $D > m \Rightarrow$ Thrashing. Therefore, we need to suspend or swap out one of the processes.

Page Fault Frequency

- Thrashing has a high page fault rate.
- **Page Fault Frequency** - Establish “acceptable” page-fault frequency (PFF) rate and use local replacement policy
  - Page fault rate high => process needs more frames.
  - Page-fault rate is too low => process may have too many frames
  - Establish upper and lower bounds on the desired page-fault rate
    - Page fault rate > upper limit – allocate more frames to the process
    - Page-fault rate < lower limit - remove a frame from process

Allocating Kernel Memory

- So far, we have discussed about process' memory.
- Kernel memory is often allocated from a **free-memory pool** as:
  - Certain hardware devices interact directly with physical memory and may require memory in contiguous pages.
  - Memory needed for kernel data structures is of varying sizes (some smaller than half a pages).
    - Kernel must ensure that minimum memory is wasted due to fragmentation.
- Two strategies adopted for managing free memory that is assigned to kernel processes:
  - Buddy System
  - Slab Allocation

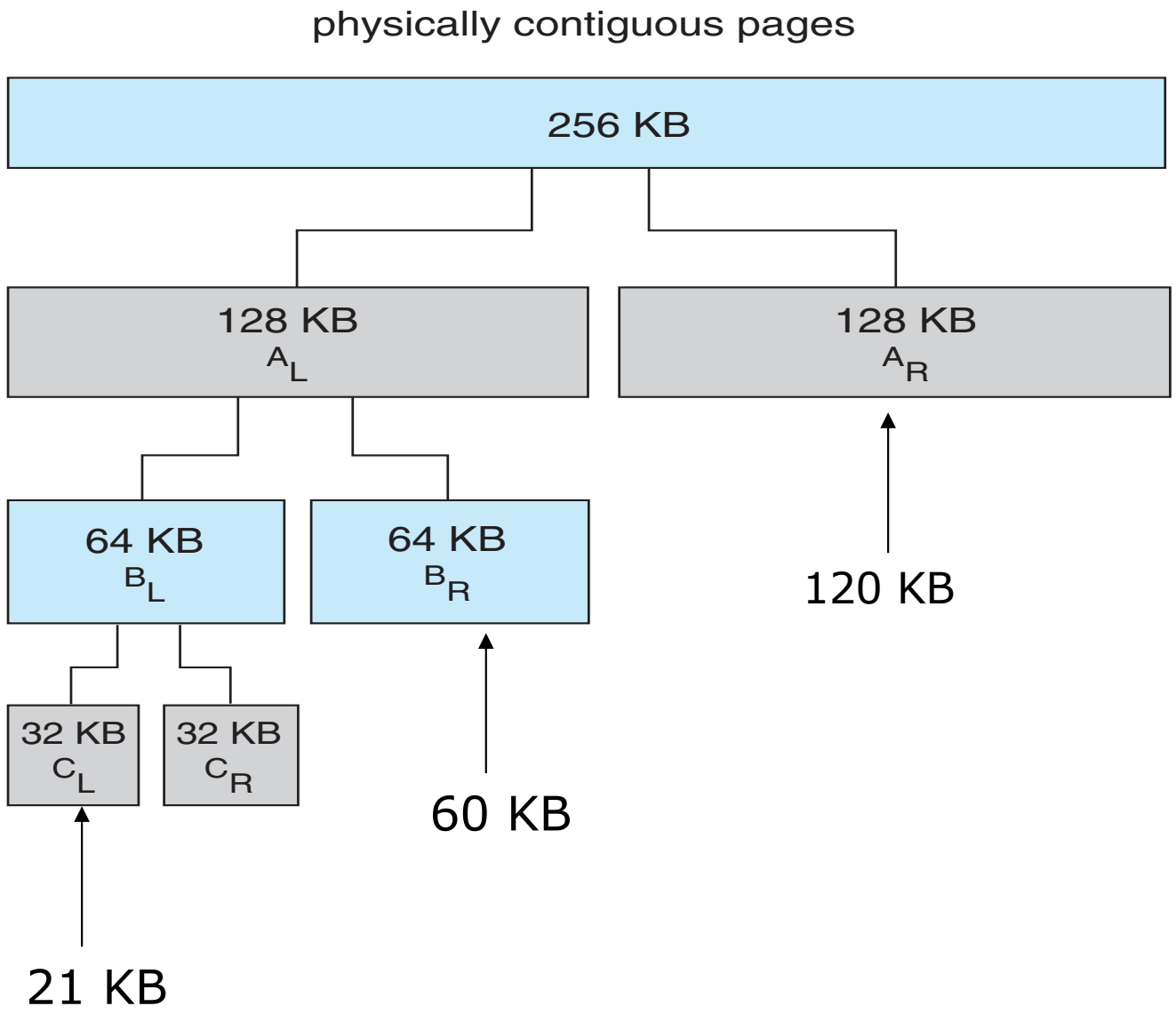
Buddy System

- 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 appropriately sized chunk available
- Advantage – quickly coalesce unused chunks into larger chunk (**note that only buddies can be coalesced**)
- **Disadvantage – internal fragmentation**

Buddy System Example

- For example, assume 256KB chunk available, kernel makes the following requests
  - request 21KB,
  - request 60 KB, and
  - request 120KB
- Rounding the request of 21KB to the closest power of 2 > 21, we get segment of size 32KB. Therefore, request 21KB is satisfied by memory segment C_L (see the tree in next slide).
- Other requests (60KB and 120KB) are satisfied in a similar way.
- If request 60KB and 120KB are released, we cannot coalesce, these segments as they were not buddies – that is they did not result from the same partition.
- However, if request 21KB is released later, then all the segments can coalesce to form the original 256KB segment.

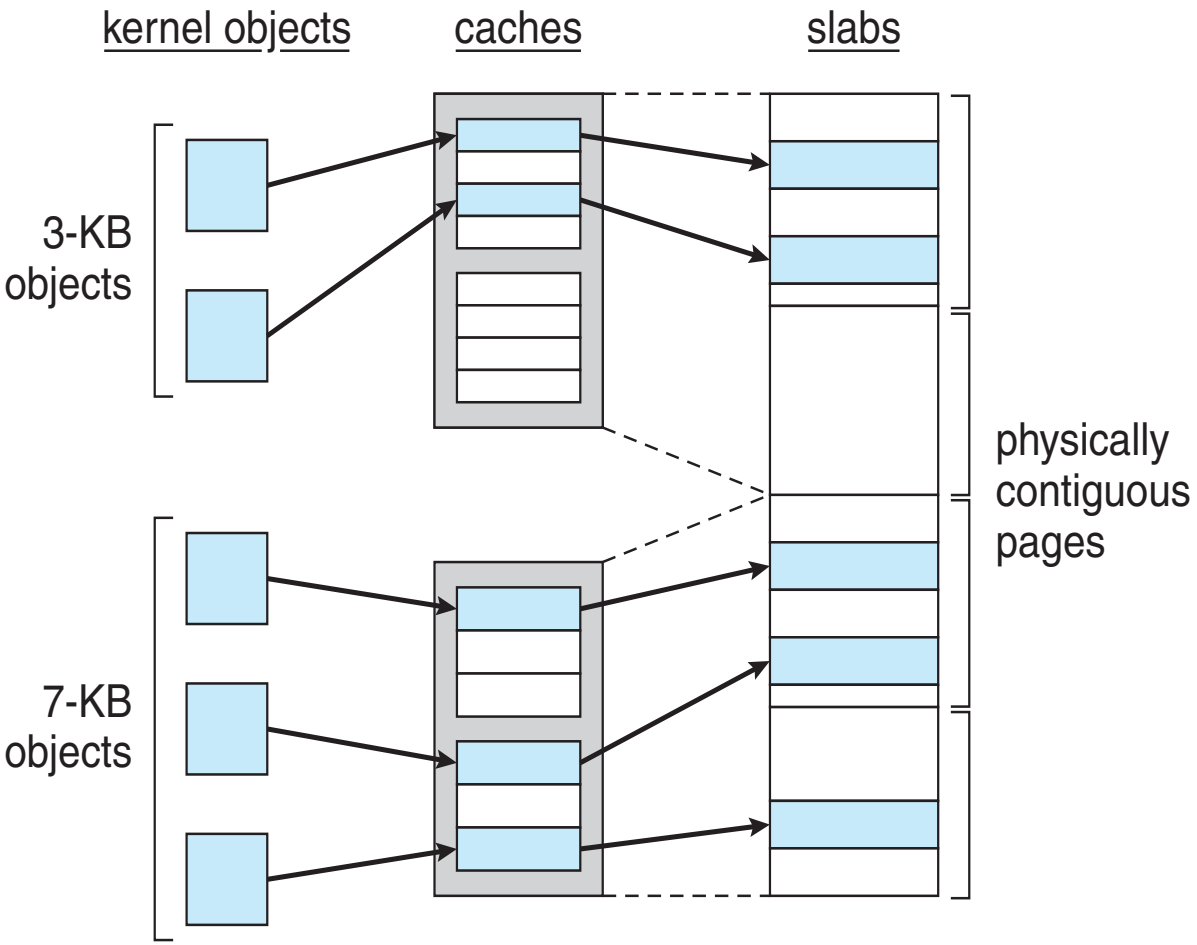
Buddy System Example Cont…



Slab Allocation

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs.
  - Single cache for each unique kernel data structure (e.g.: separate cache for program descriptors, semaphores, file objects etc.)
  - Each cache filled with **objects** – instantiations of the kernel data structure the cache represents.

Slab Allocation



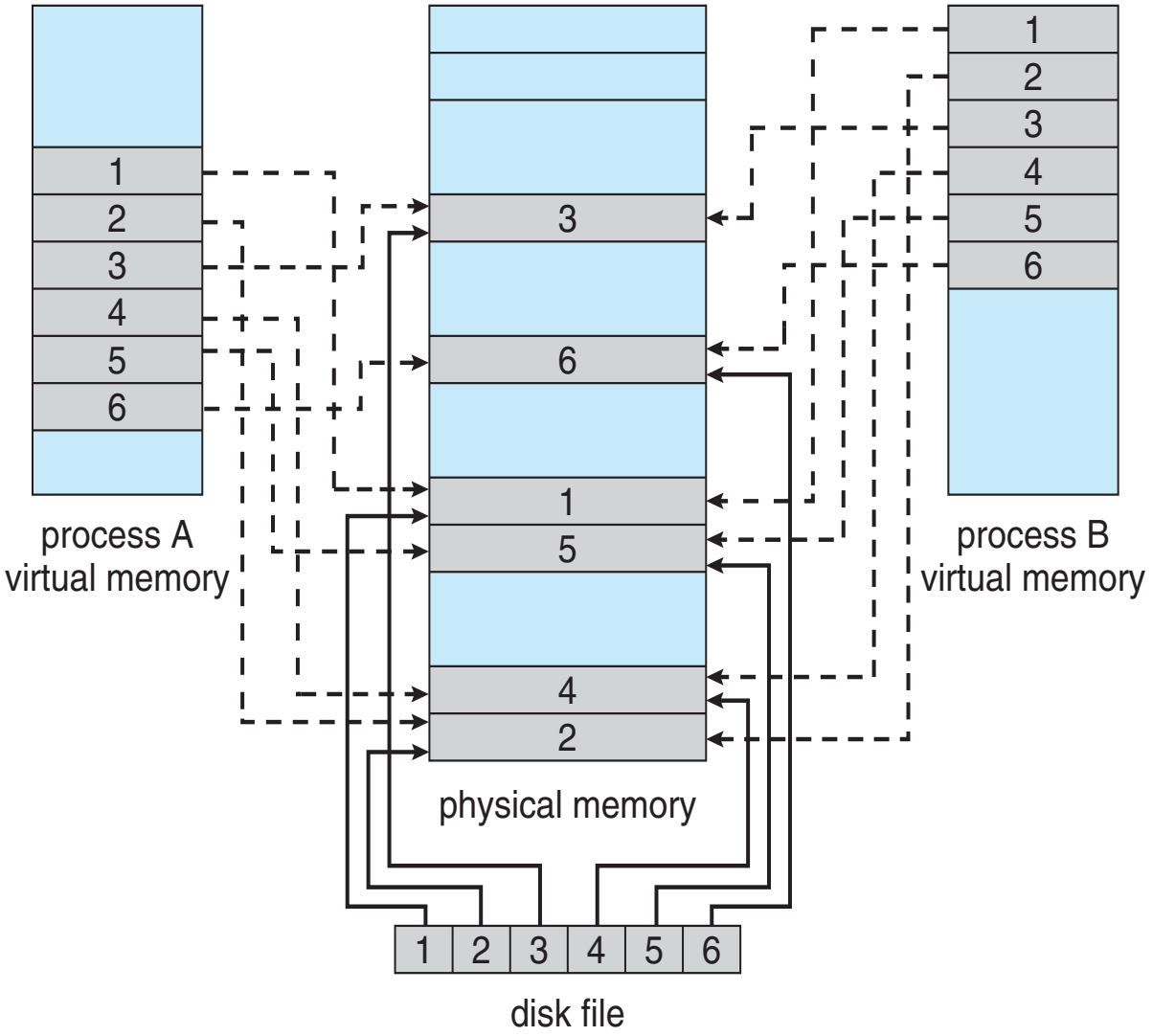
Slab Allocation

- When cache is created, it is filled with objects marked as **`free`**
- Objects assigned from cache are marked as **`used`.**
- If a slab is full of used objects, the next free object is allocated from a partial slab if available.
- If no empty slabs, new slab allocated from contiguous physical pages and assigned to a cache.
  - memory for the object is allocated from this slab.
- **Benefits include**
  - no fragmentation
  - fast memory request satisfaction (as no allocation and deallocation of memory)
- Slab Allocation used in Solaris, and now used by various Operating Systems (e.g., Linux)

Memory-Mapped Files

- Memory-mapped file allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a frame in memory
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map to the same file allowing the pages in memory to be shared
- Data written back to disk periodically and/or at file `close()` time
- Some Oses (e.g Solaris) uses memory mapped files for standard I/O.

Memory Mapped Files



Copy-on-Write

- Consider the fork() system call to create a new child process.
  - It creates a copy of the parent's address space for the child.
  - As most fork() calls are followed by exec() system, the above steps is unnecessary.
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied.