# Chapter 10

## Virtual Memory

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

# Background

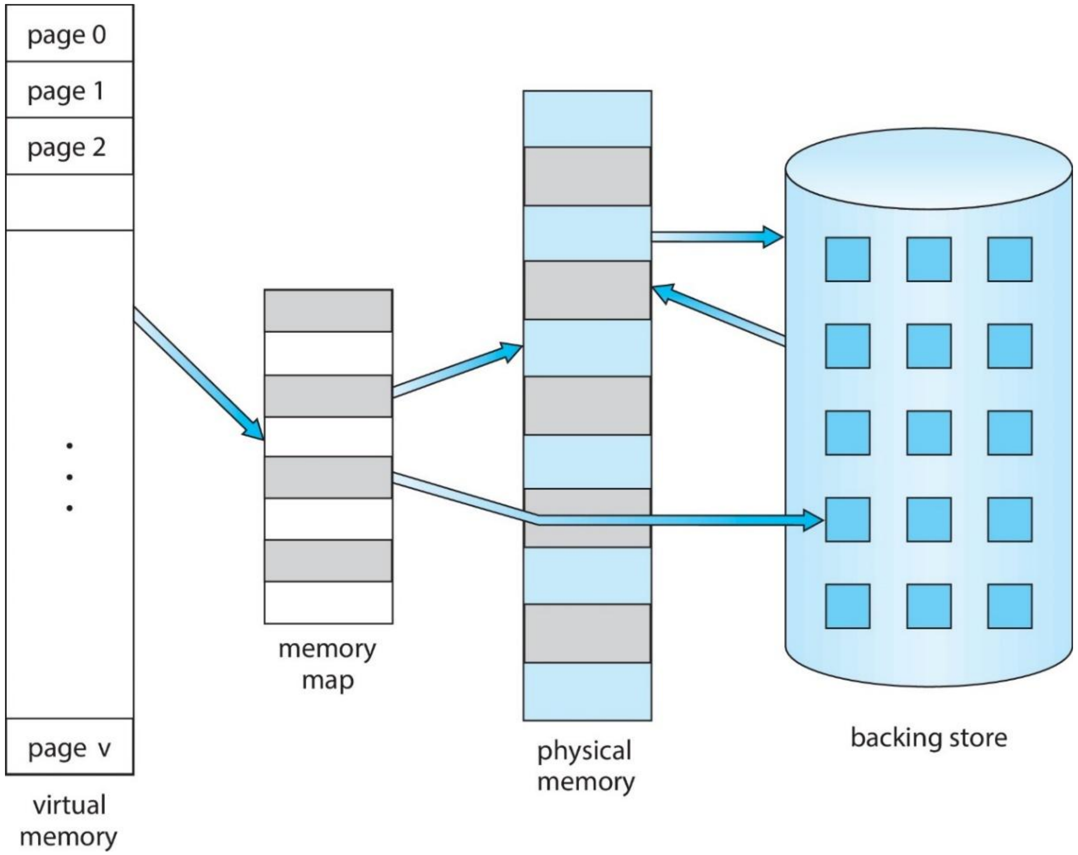
- Instructions should be in physical memory to be executed
  → In order to execute a program, should we load entire program in memory?
- Some parts are rarely used
  - Error handling codes
  - Arrays/lists larger than necessary
  - Rarely used routines
- Alternative
  - Executing program which is only partially in memory

# Background

- If we can run a program by loading in parts …
  - A program is not constrained by the amount of physical memory
  - More program can run at the same time
  - Less I/O is need to load or swap programs

# Virtual Memory

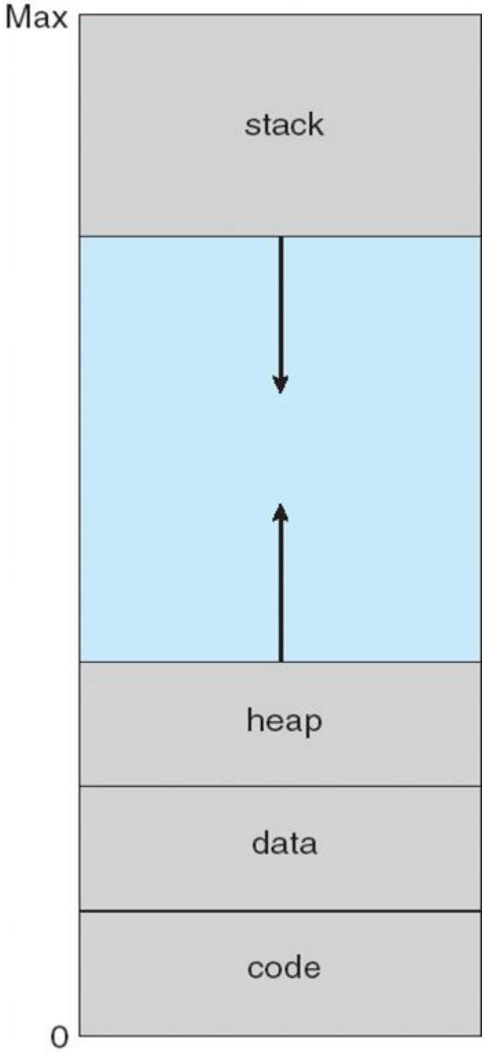
- **Virtual memory**: separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation (e.g., Copy-on-Write)
  - More programs running concurrently
  - Less I/O needed to load or swap processes




<Virtual memory that is larger than physical memory>

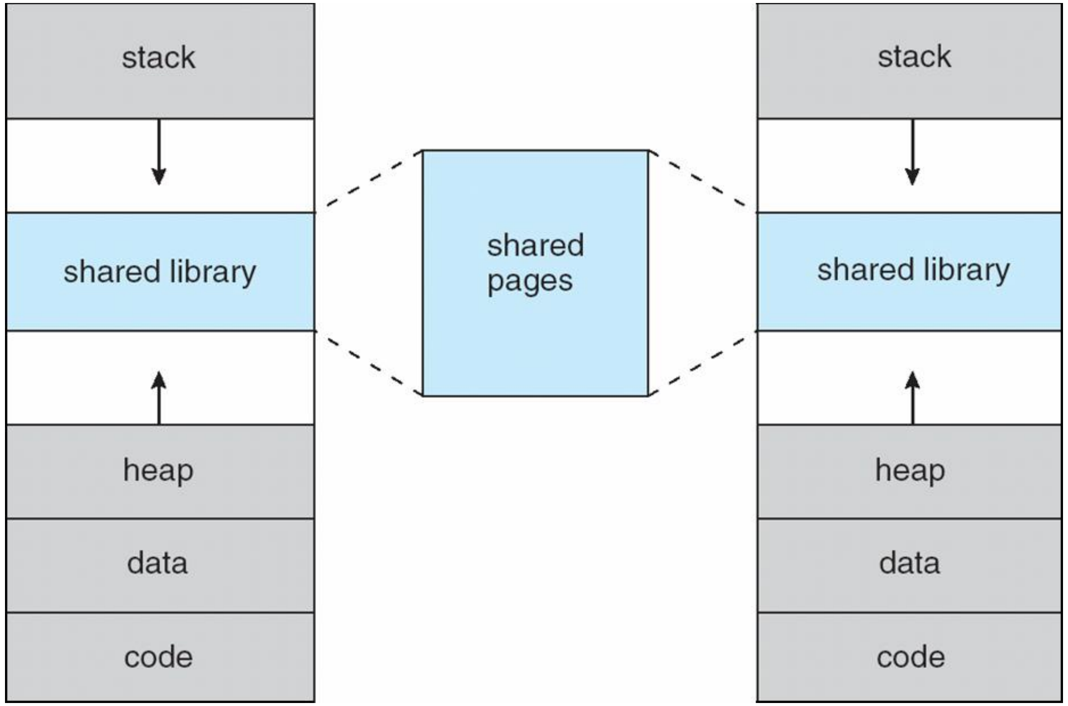
# Virtual Memory

- **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
  - Programmers don't have to concern about memory management
  - Virtual address space can be sparse
    - Holes can be filled as the stack or heap grow or if we wish to dynamically link libraries during program execution



# Virtual Memory

- Virtual memory allows files and memory to be shared by two or more processes through page sharing
  - System libraries shared via mapping into virtual address space
  - Shared memory by mapping pages read-write into virtual address space
  - Pages can be shared during `fork()`, speeding process creation



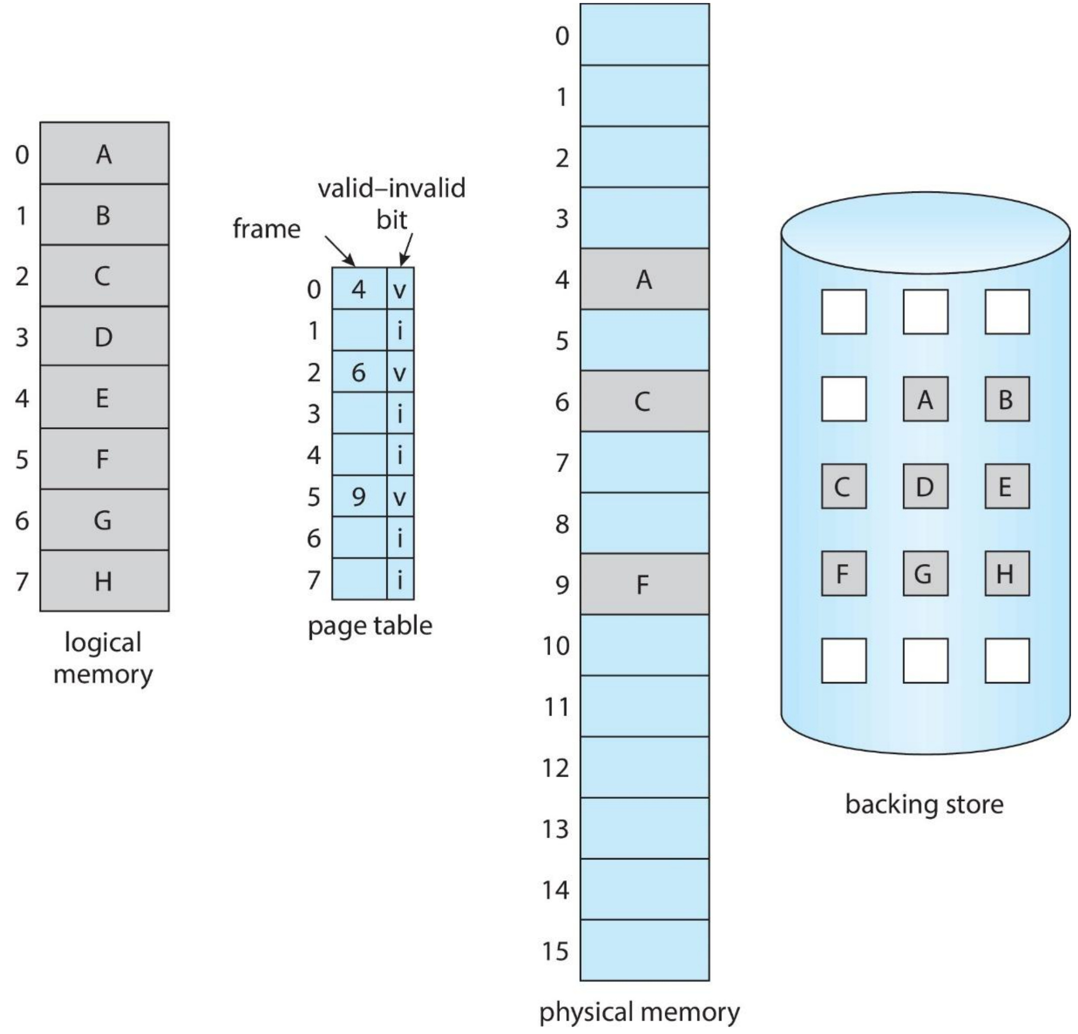

<Shared library using virtual memory>

# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

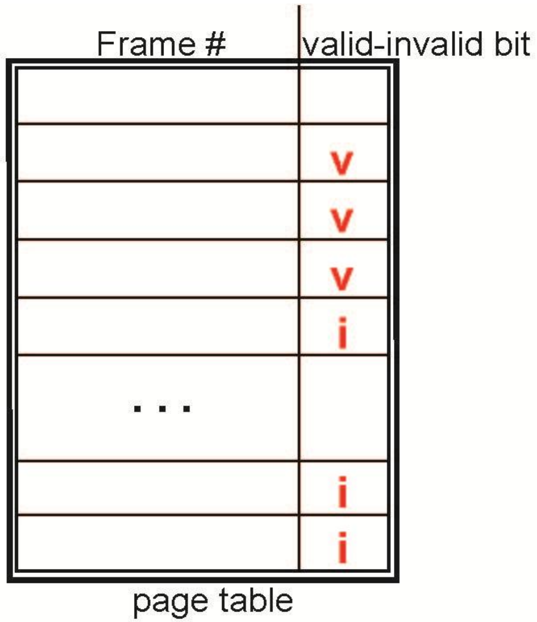
# Demand Paging

- **Demand Paging**: pages are loaded only when they are demanded during program execution
  - Similar to paging system with swapping
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
  - Requires H/W support to distinguish the page on memory or on disk



# Page Table with Valid-Invalid Bit

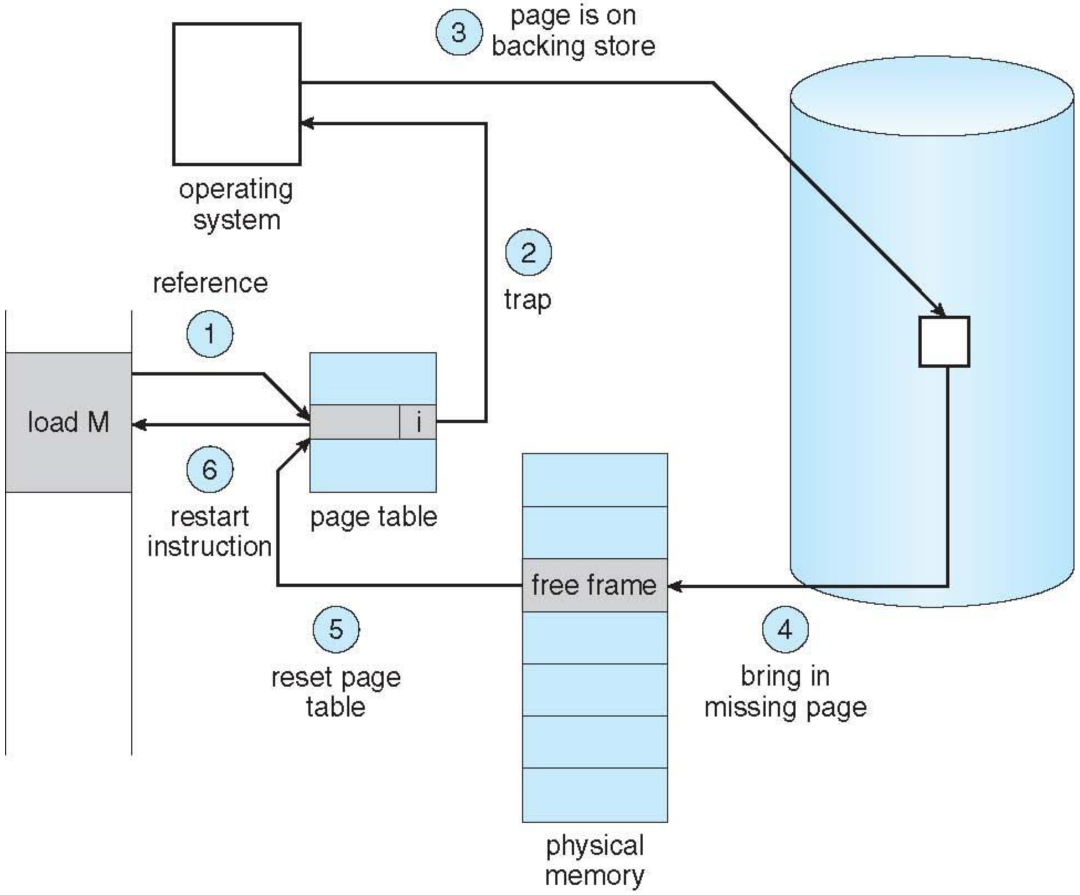
- Valid/invalid bit of each page
  - Valid: the page is valid and exists in physical memory
  - Invalid: the page is not valid (not in the valid logical address space of the process) or not loaded in physical memory




- If program tries to access ..
  - Valid page: execution proceeds normally
  - Invalid page: cause **page-fault** trap to OS

# Handling Page-Fault

- Handling page-fault
  - Check an internal table to determine whether the reference was valid or not
  - If the reference was invalid, terminates the process
  - If it was valid but we have not yet brought in that page, we page it in.
    - Find a free frame
    - Read desired page into the free frame
    - Modify internal table
    - Restart the instruction that caused the page-fault trap

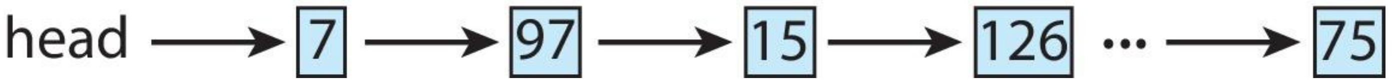


# Aspects of Demand Paging

- Extreme case: start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident → page fault
  - And for every other process pages on first access
  - Pure demand paging
- Actually, a given instruction could access multiple pages → multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of locality of reference
- Hardware support needed for demand paging
  - Page table with valid-invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart: ability to restart instruction after a page fault

# Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** → a pool of free frames for satisfying such requests.




- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** → the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

# Performance of Demand Paging

- Effective access time

  Effective access time = (1-***p***) * ***ma*** + ***p*** * <page fault time>

  - ***ma*** : memory access time (10~200 nano sec.)
  - ***p*** : probability of page fault

- Page fault time
  - Service page-fault interrupt → 1~100 μsec.
  - Read in the page → about 8 msecs
  - Restart the process → 1~100 μsec.

# Performance of Demand Paging

- Effective access time = (1-***p***) * ***ma*** + ***p*** * <page fault time>
- Example
  - Memory access time: 200 nano sec
  - Page-fault service time: 8 milliseconds
- Then…
  - Effective access time (in nano sec.)

    = (1-p) * 200 + 8,000,000 * p

    ≈ 200 + 7,999,800 * p
    - Proportional to page fault rate
    - Ex) p == 1/1000, effective access time = 8.2 μsec. (40 times)
  - Page fault rate should be kept low

# Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Ways to execute a program in file system
  - Option1: copy entire file into swap space at starting time
    - Usually swap space is faster than file system
  - Option2: initially, demand pages from files system and all subsequent paging can be done from swap space
    - Only needed pages are read from file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

OS
Intermediate

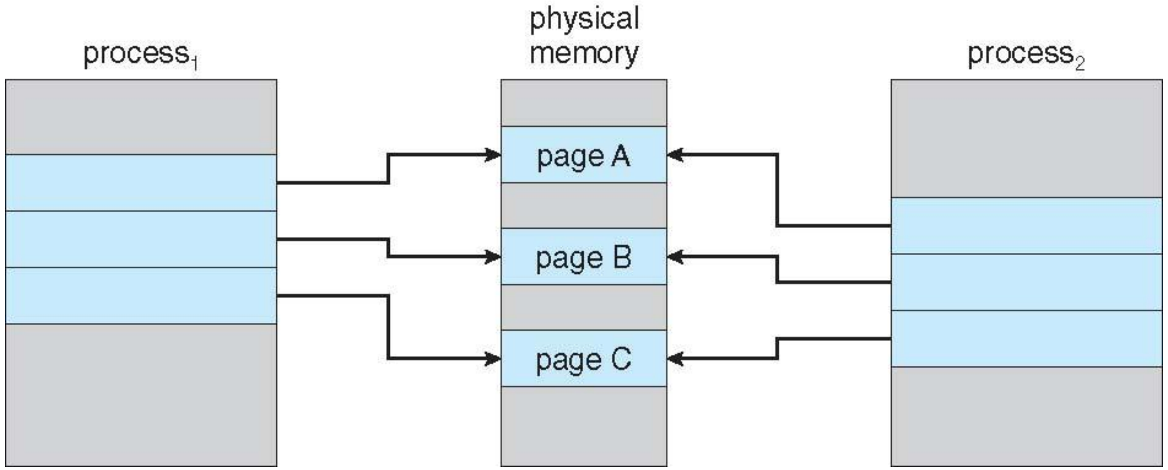
# Copy-on-Write

- `fork()` copies process
  - Duplicates pages belong to the parent
- **Copy-on-write (COW)**
  - When the process is created, pages are not actually duplicated but just shared.
  - Process creation time is reduced.
  - When either process writes to a shared page, a copy of the page is created.

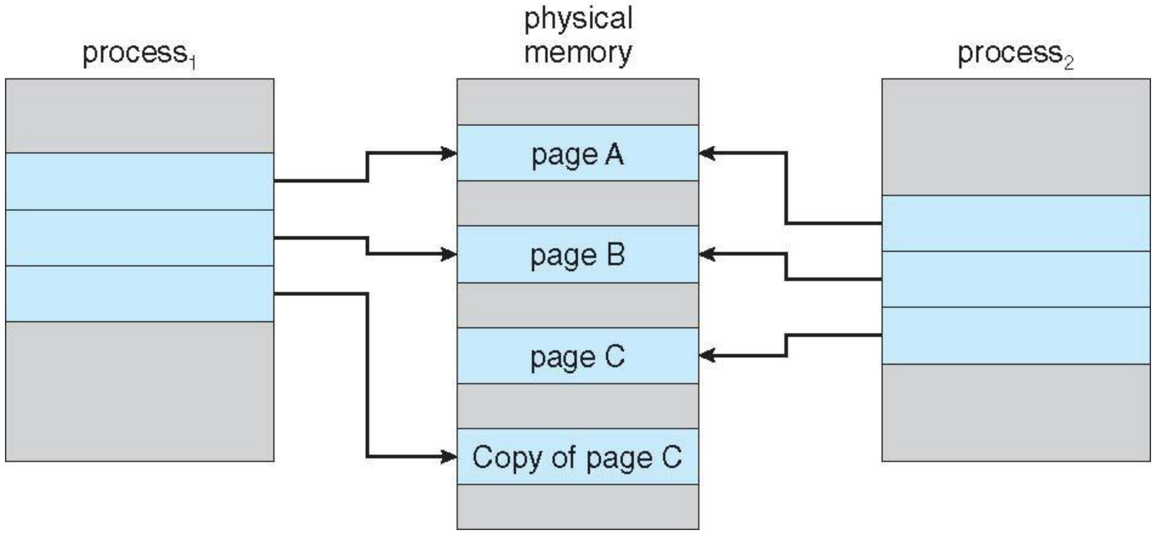
  cf. `vfork()` – logically shares memory with parent (obsolete)
- Many OS's provides a list of free frames for COW or stack/heap that can be expanded → Zero-fill-on-demand (ZFOD)
  - Zero-out pages before being assigned to a process

# Copy-on-Write

- Before P1 modifies page C




- After P1 modifies page C



# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

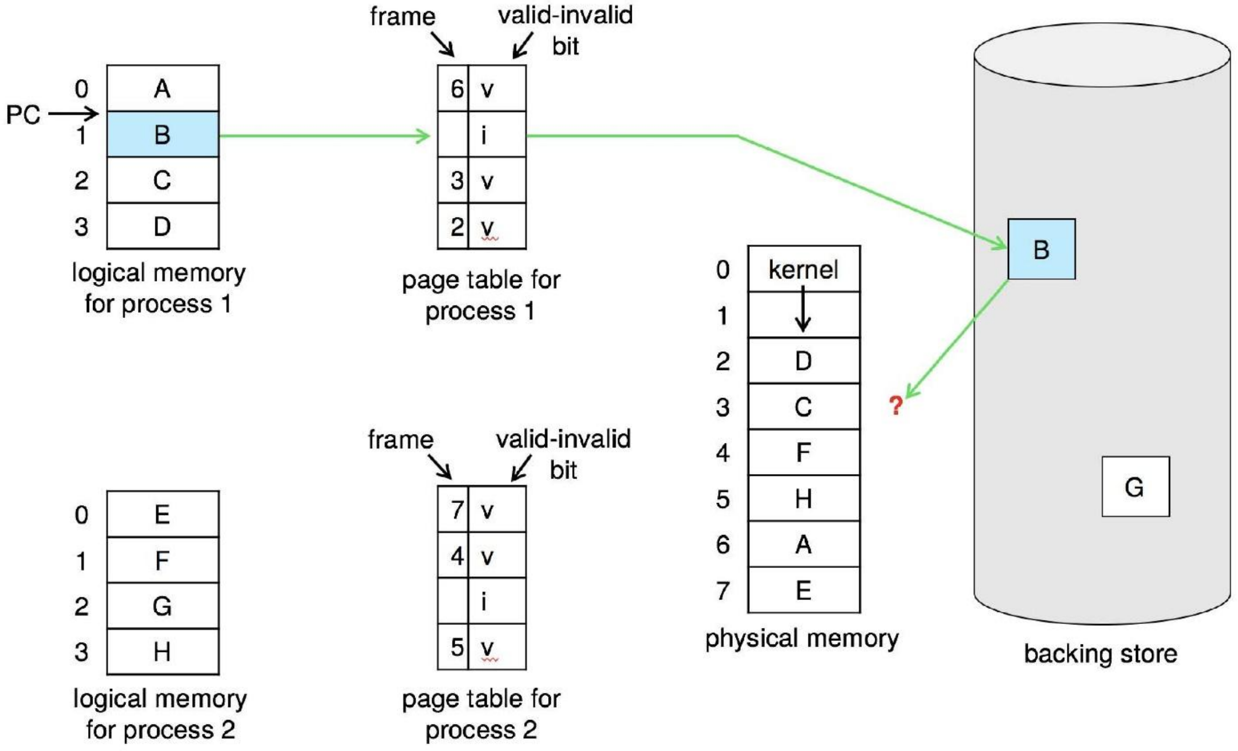
# Two Major Problems

- Two major problems in demand paging
  - Page-replacement algorithm
    - Want lowest page-fault rate on both first access and re-access
  - Frame-allocation algorithms
    - How many frames to give each process
    - Which frames to replace

- Even slight improvement can yield large gain in performance.

# Page Replacement

- Page replacement
  - If no frame is free at a page fault, we find a frame not being used currently, and swap out
  - Writing overhead can be reduced by modify-bit (or dirty-bit) for each frame

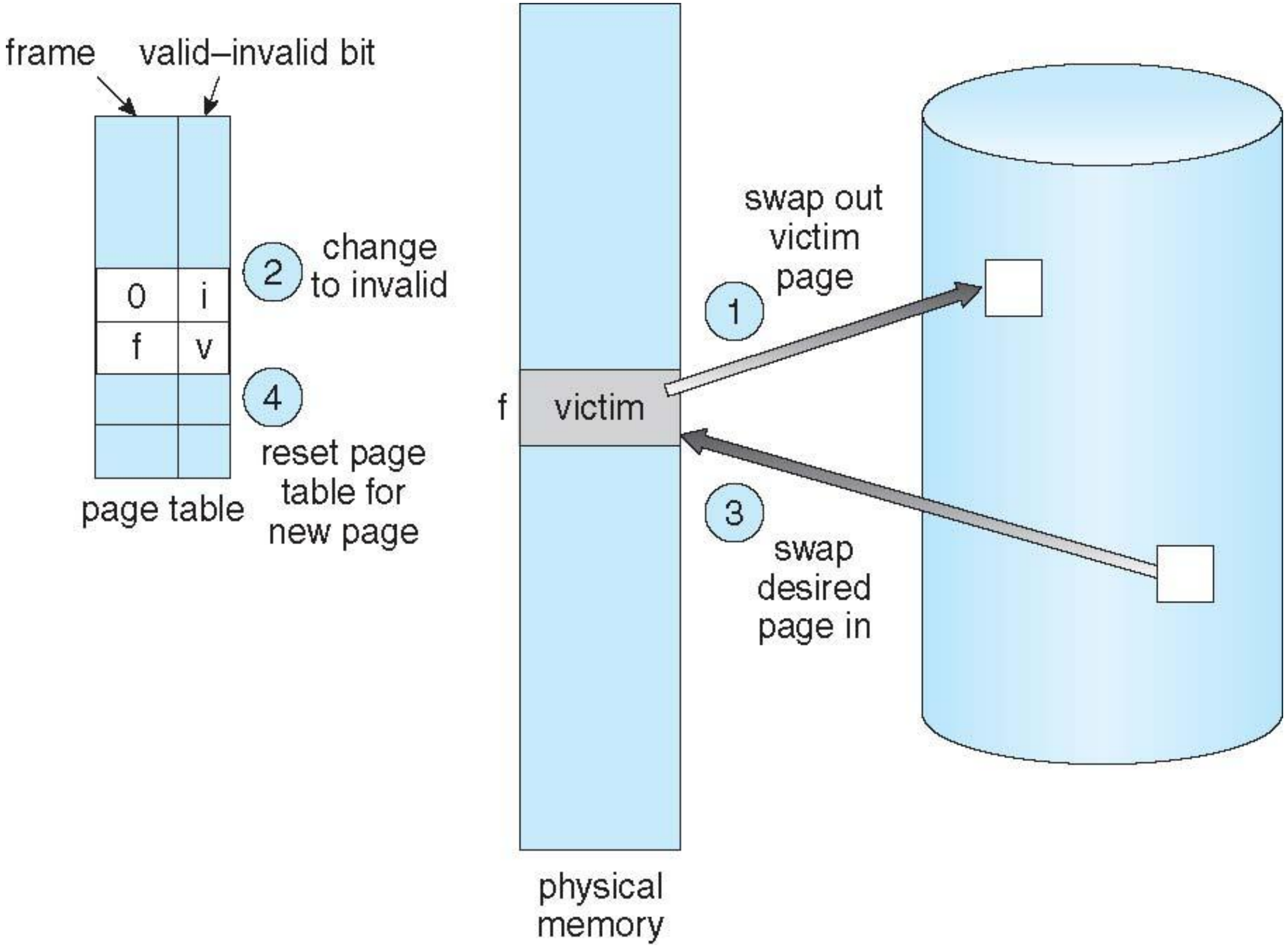


# Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   a. If there is a free frame, use it
   b. If there is no free frame, use a page replacement algorithm to select a victim frame
   c. Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

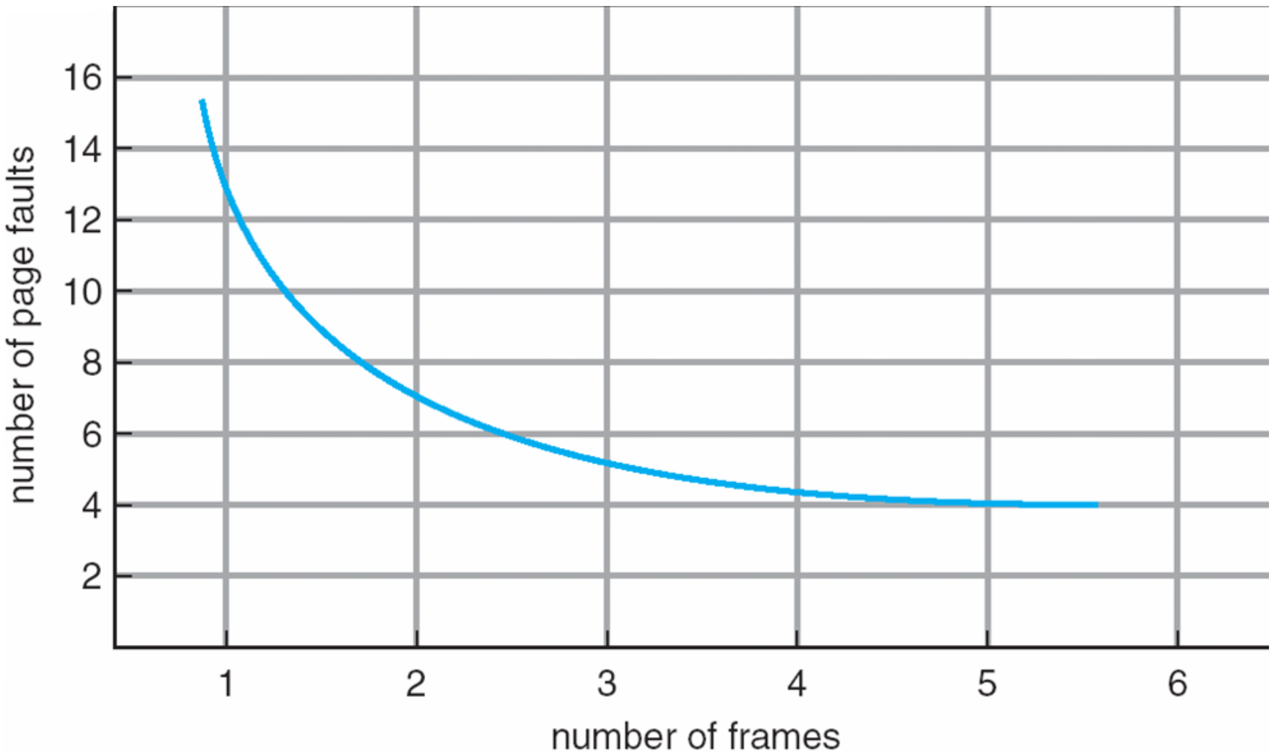
Note now potentially 2 page transfers for page fault → increasing EAT

# Page Replacement



# Page Faults vs. Number of Frames

- In general, the more frames, the fewer page faults

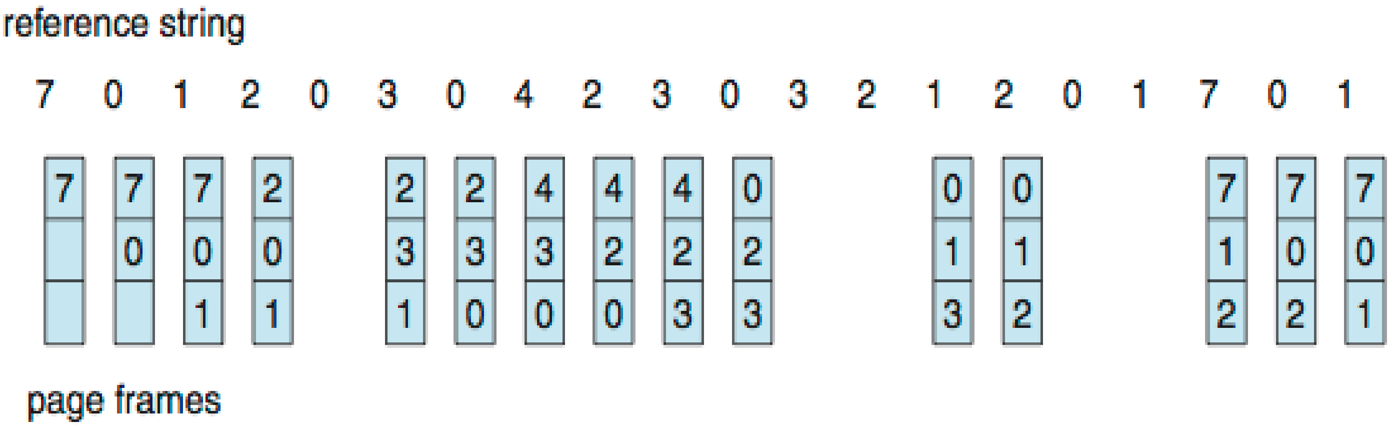


# Page Replacement Algorithms

- FIFO page replacement
- Optimal page replacement (in theory)
- Least-recently-used (LRU) page replacement
- LRU-approximation page replacement
- ETC.

# FIFO Page Replacement

- **First-in, first-out**: when a page should be replaced, the oldest page is chosen.
  - Easy, but not always good

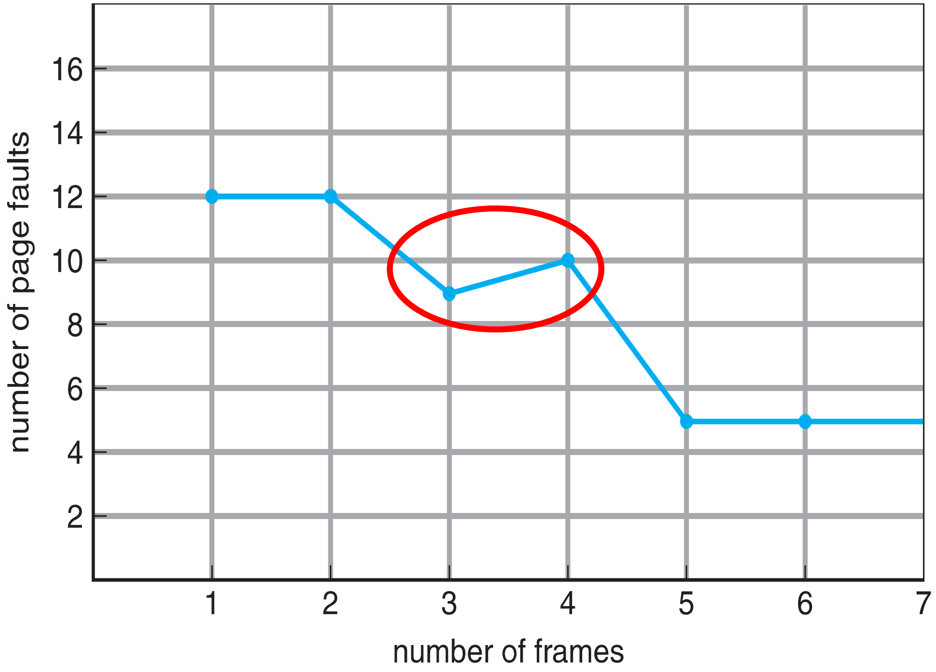



  - # of page faults: 15
- Problem: Belady's anomaly

# FIFO Page Replacement

- **Belady's anomaly**: # of faults for 4 frames is greater than # of faults for 3 frames

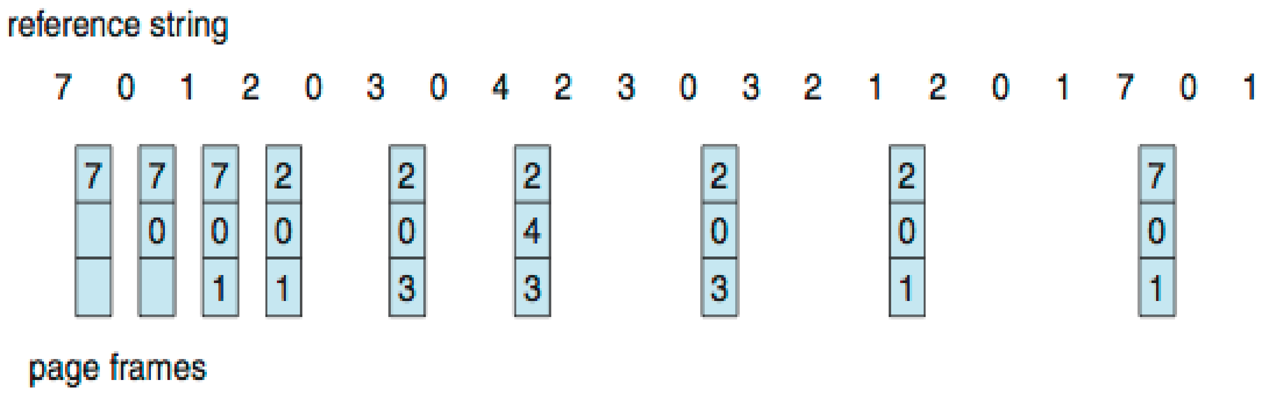
  (Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5)




Page-fault rate may increase as the number of frames increase.

# Optimal Page Replacement

- Replace the page that will not be used for the longest period of time

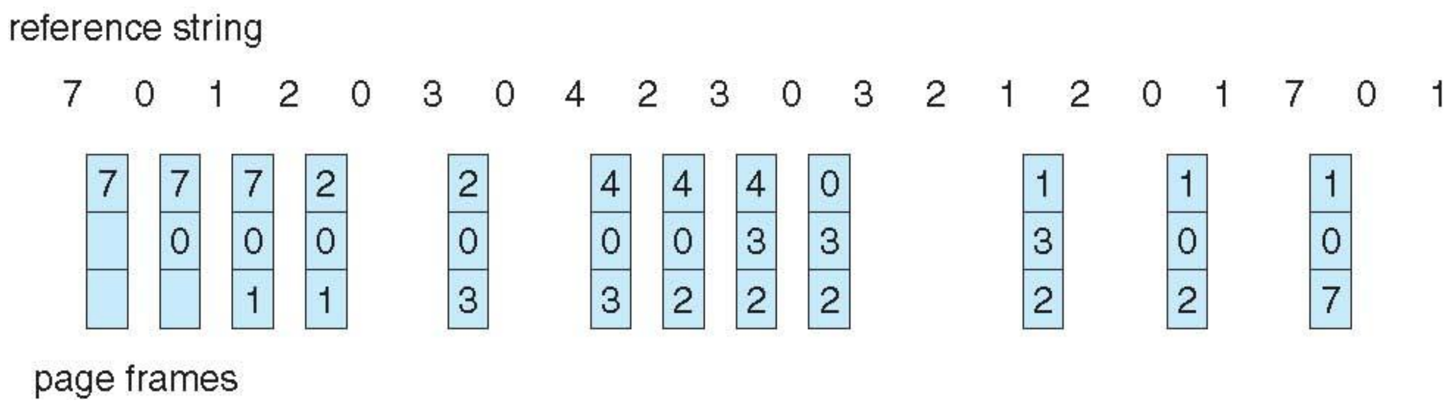



  - # of page faults: 9

- Problem: It requires future knowledge

# LRU Page Replacement

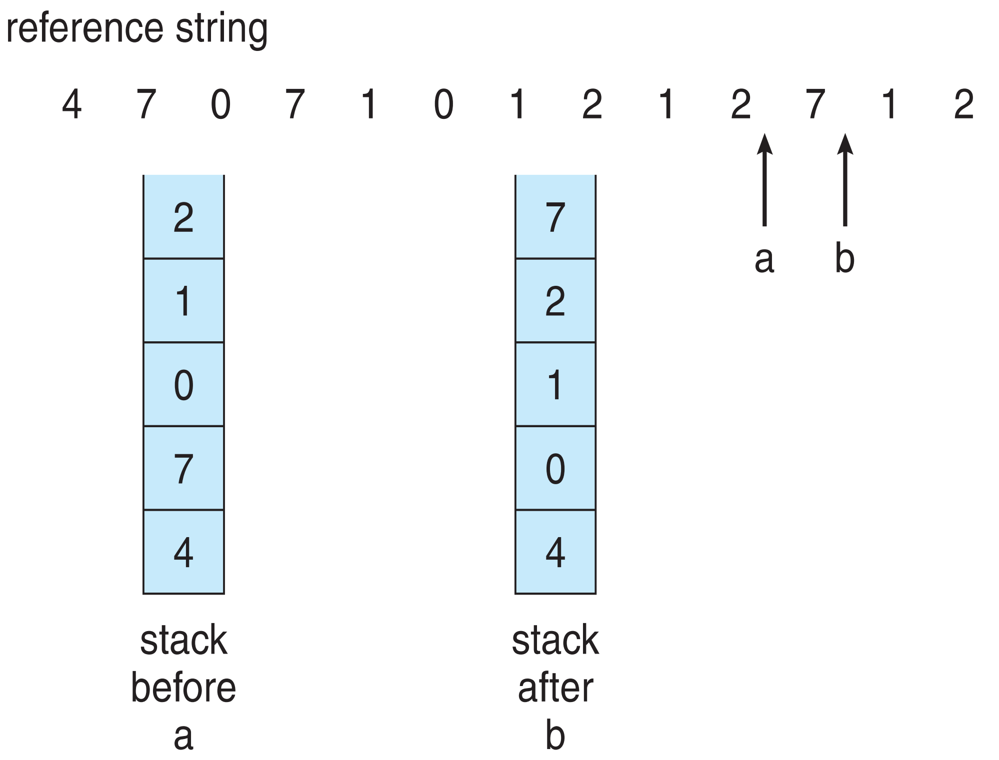
- **LRU (Least Recently Used)**: replace page that has not been used for longest period of time




  - # of page faults: 12
  - **LRU is considered to be good and used frequently**

# Implementation of LRU

- Using counter (logical clock)
  - Associate with each page-table entry a **time-of-used field**
  - Whenever a page is referenced, clock register is copied to its time-of-used field
- Using stack of page numbers
  - If a page is referenced, remove it and put on the top of the stack



# Stack Algorithm

- Does LRU cause the Belady's anomaly?
- Stack algorithm: an algorithm for which the set of pages in memory for $n$ frames is always a subset of the set of pages that would be in memory with $n$+1 frames.
  - Never exhibit Belady's anomaly
  - LRU is a stack algorithm

$\boldsymbol{n}$ frames

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

$\boldsymbol{n+1}$ frames

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

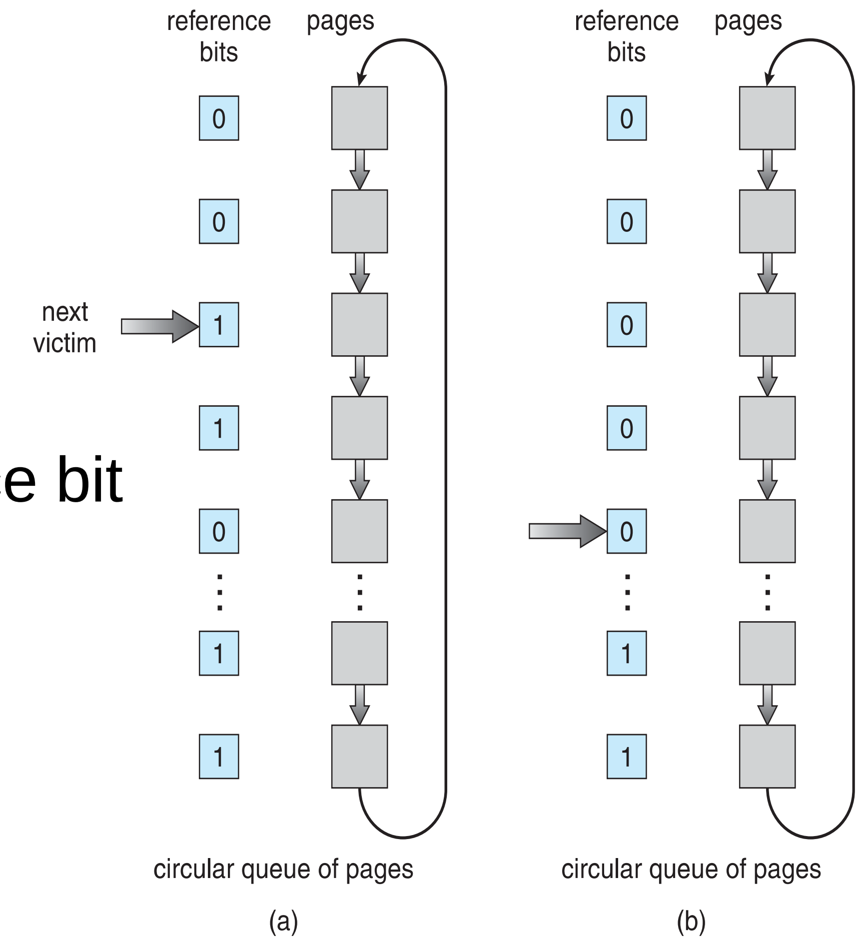
# LRU-Approximation Page Replacement

- Motivation
  - LRU algorithm is good, but few system provide sufficient supports for LRU
  - However, many systems support reference bit for each page
    - We can determine which pages have been referenced, but not their order.

- LRU-approximation algorithms
  - Additional-reference-bit algorithm
  - Second-chance algorithm

# LRU Approximation Algorithms

- **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



# Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used  but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

# 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

# Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- **Allocation of Frames**
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

# Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
  - How many frames does each process get?

- Minimum number of frames for each process
  - # of frames for each process decreases

    → page-fault rate is increases

    → performance degradation
  - Minimum # of frames should be large enough to hold all different pages that any single instruction can reference.

# Allocation Algorithms

- Equal allocation
  - Split m frames among n processes → m/n frames for each process

- Proportional allocation
  - Allocate available memory to each process according to its size

$$a_i = s_i/S * m$$

    - $a_i$: # of frames allocated to process pi
    - $s_i$: size of process pi
    - $S = \sum s_i$

# Global vs. Local Allocation

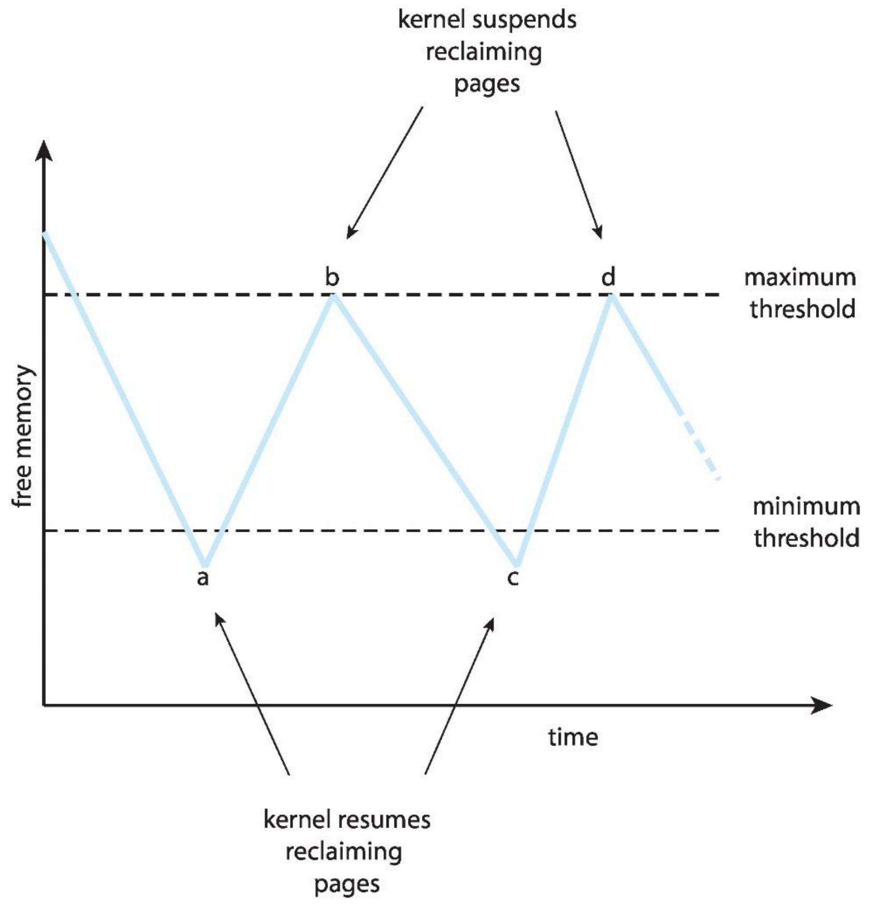
- **Global replacement**: a process can select a replacement frame from the set of all frames, including frames allocated to other processes
  - A process cannot control its own page-fault rate

- **Local replacement**: # of frames for a process does not change
  - Less used pages of memory can’t be used by other process

→ global replacement is more common method.

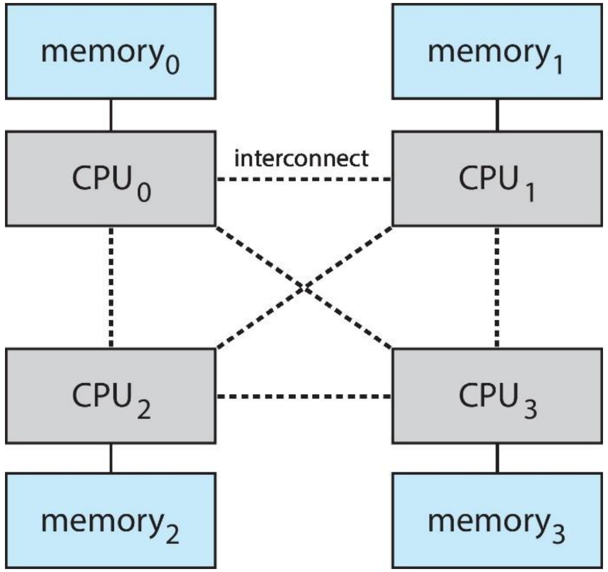
# Reclaiming Pages

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



# Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA**: speed of access to memory varies
- 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
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup



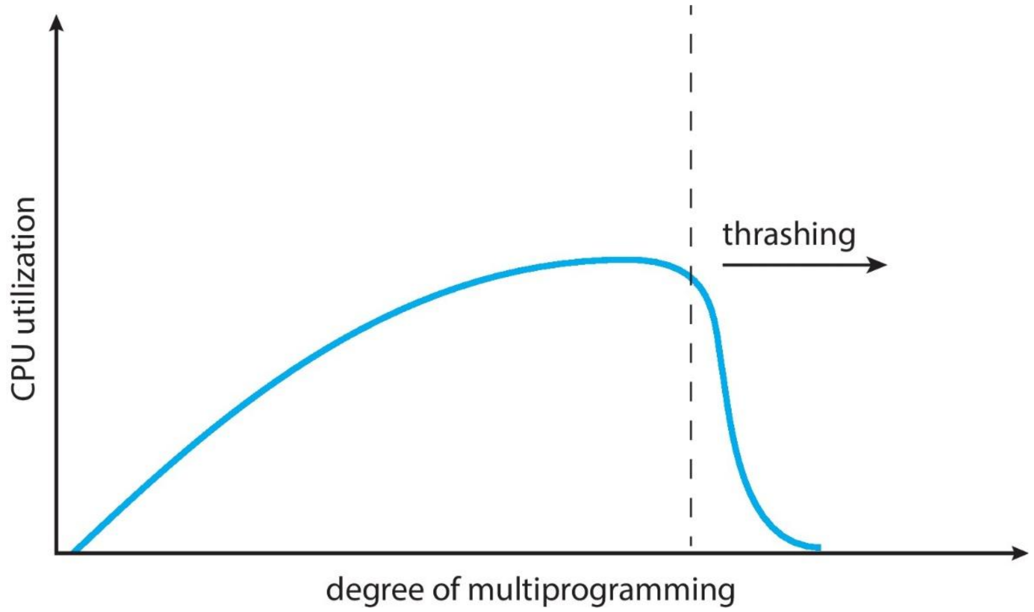

NUMA multiprocessing architecture

# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

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

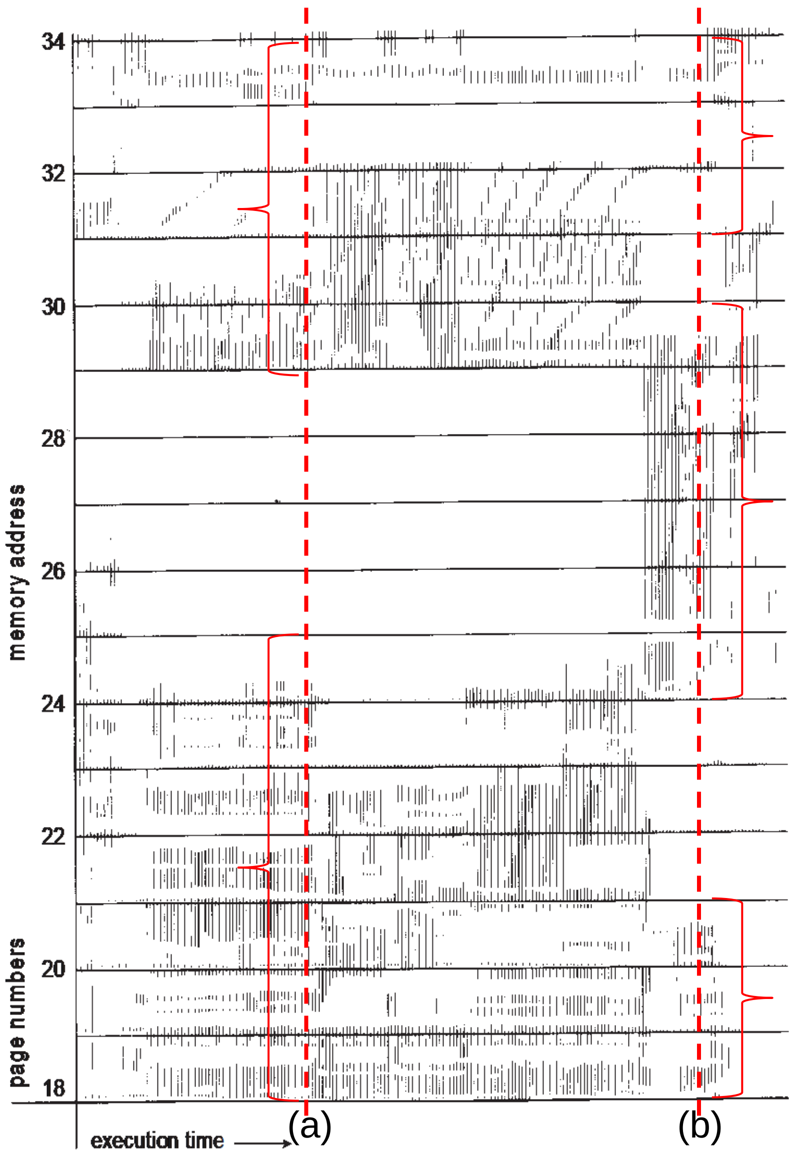



- **Thrashing**: A process is busy swapping pages in and out

# Demand Paging and Thrashing

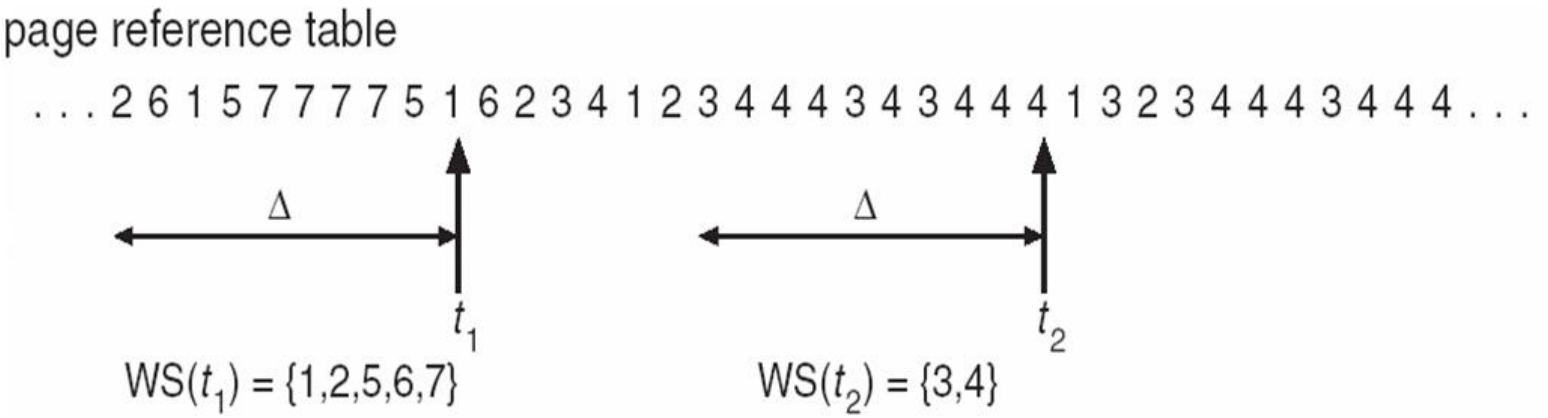
- To prevent thrashing, a process must be provided with as many frames as it needs.
  - → How to know how many frames it needs?

- Locality model
  - Locality: set of pages actively used together
  - A program is generally composed of several localities



# Working-Set Model

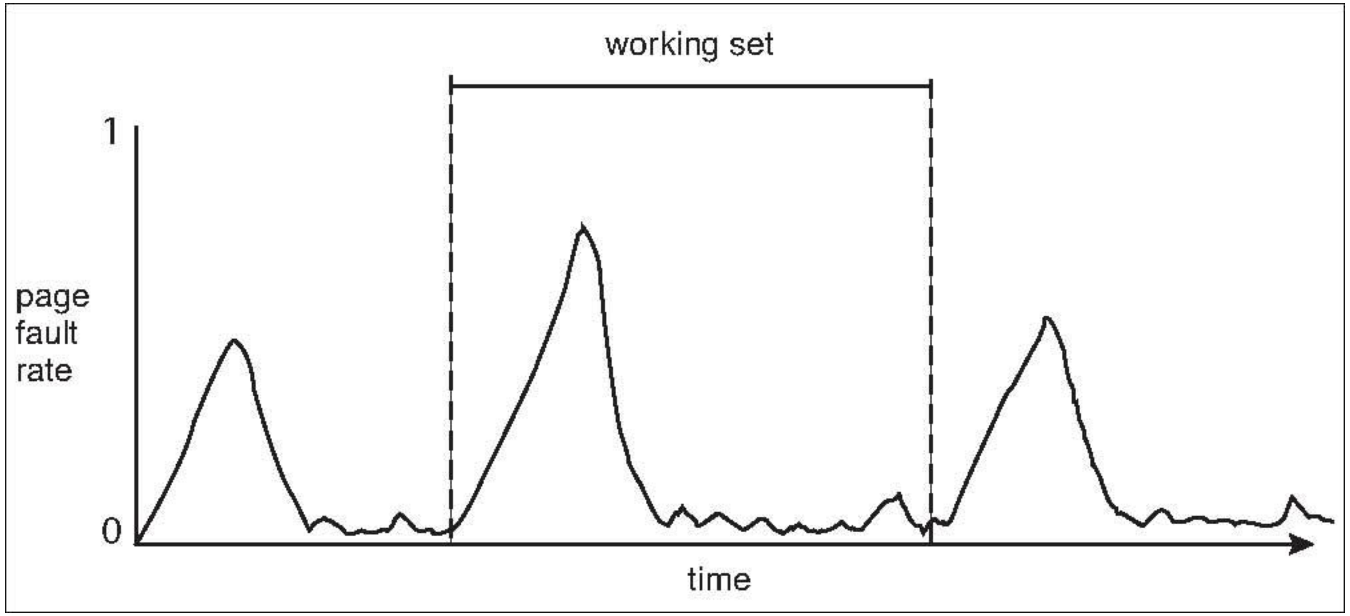
- **Working set**: set of pages in the most recent Δ page references
  - Parameter Δ: **working-set window**




  - $WSS_i$: working set size of process $p_i$
  - **Process $p_i$ needs $WSS_i$ frames**
- If total demand is greater than # of available frames, thrashing will occur.

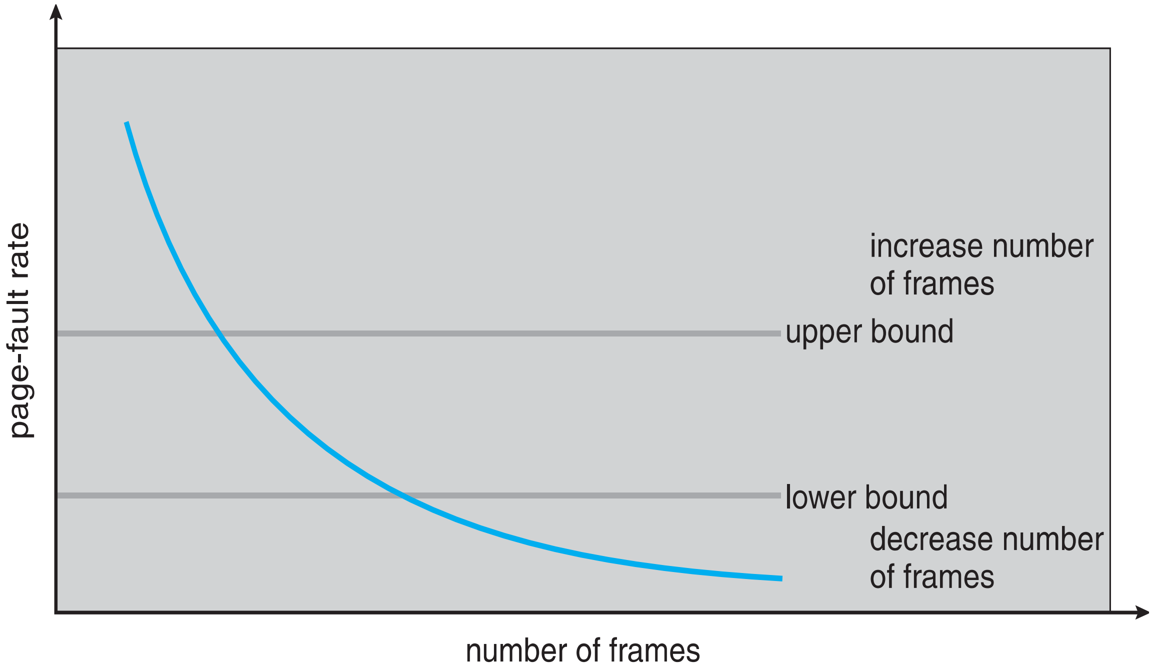
# Working Sets and Page Fault Rates

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



# Page-Fault Frequency

- Alternative method to control trashing: control degree of multiprogramming by page-fault frequency (PFF)
  - If PFF of a process is too high, allocate more frame
  - If PFF of a process is too low, remove a frame from it



# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- **Allocating Kernel Memory**
- Other Considerations
- The Linux Virtual Memory System

# Allocating Kernel Memory

- Allocation of kernel memory requires special handling
  - Kernel requests memory for data structures of varying sizes
    - Many OS’s do not subject kernel code/data to the paging system
  - Certain H/W devices interact directly with physical memory
    - Memory should reside in physically contiguous pages.

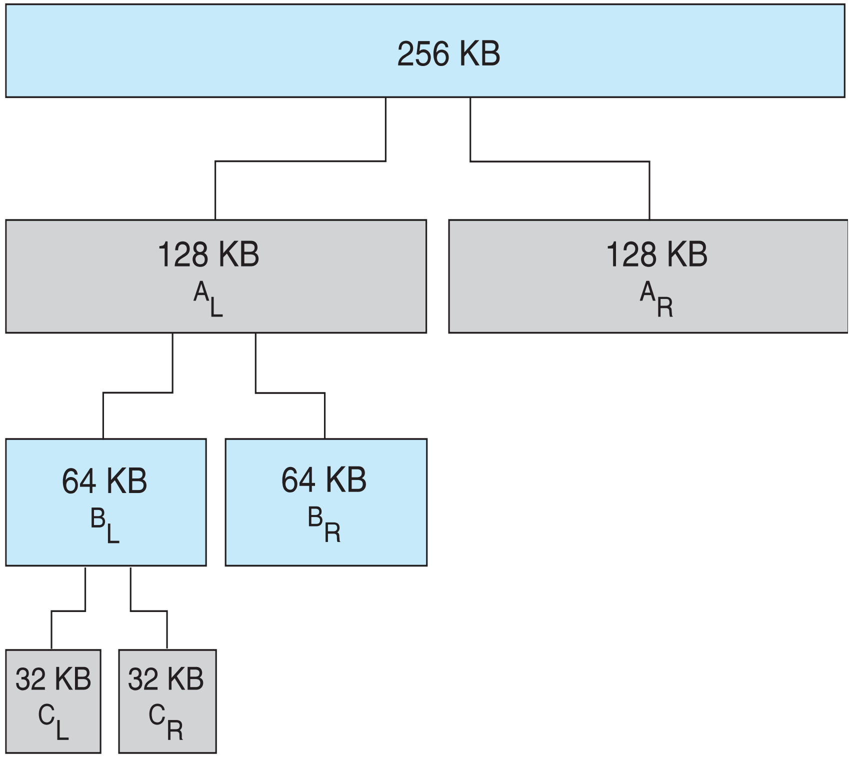
- Strategies for kernel memory allocation
  - Buddy system
  - Slab allocation

# Buddy System

- **Buddy system**: allocates memory from a fixed-size segment consisting of physically contiguous pages
  - Power-of-2 allocator

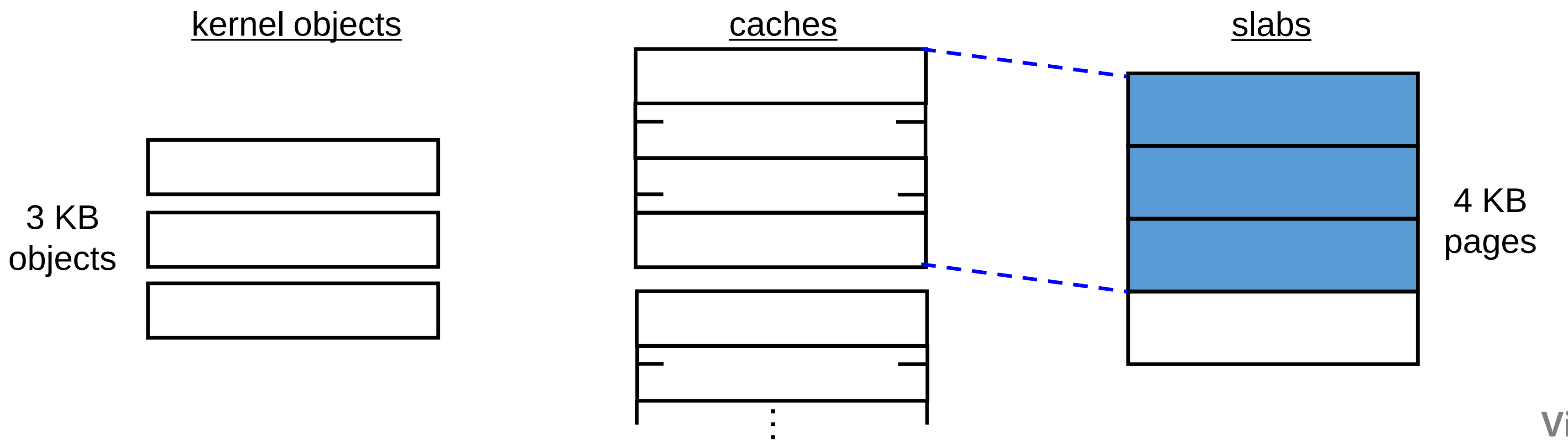
    Ex) Initially 256 KB is available,
    21 KB was requested
  - Advantage: easy to combine adjacent buddies
  - Disadvantage: internal fragmentation

physically contiguous pages



# Slab Allocation

- Motivation: mismatch between allocation size and requested size
  - Page-size granularity vs. byte-size granularity
  - Applied since Solaris 2.4 and Linux 2.2
- Cache for each unique kernel data structure
  - A slab is made up of one or more physically contiguous pages
  - A cache consists of one or more slabs



# Slab Allocation

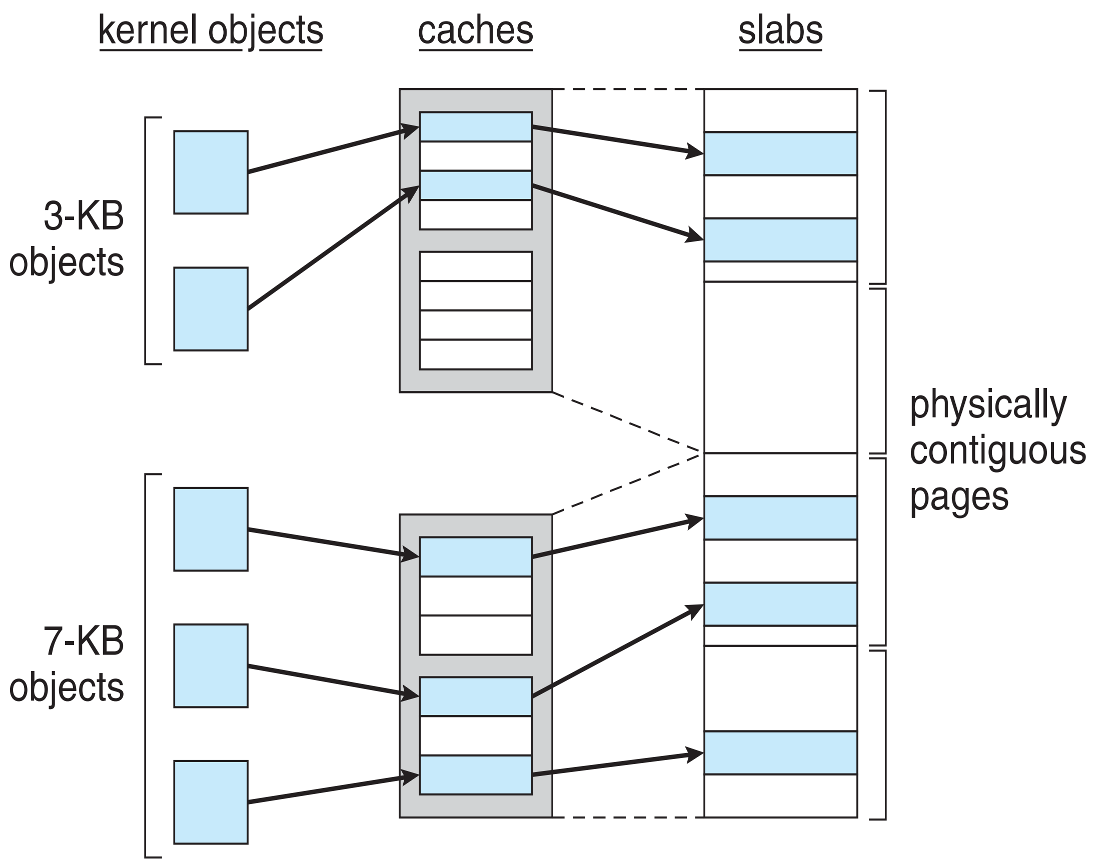
- Single cache is for each unique kernel data structure
  - Each cache filled with **objects**: instantiations of the data structure.

  Ex) cache for process descriptor, cache for file objects, cache for semaphore, …
- 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 is allocated.

# Slab Allocation

- Benefits
  - No memory waste due to fragmentation
  - Memory requests can be satisfied quickly

  → Suitable for data structures that are allocated and deallocated frequently.



# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

# Other Considerations

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

# Prepaging

- A problem of pure demand paging: a large number of page faults

- **Prepaging**: bring all pages that will be needed at one time to reduce page faults.

  Ex) working-set model

  - Important issue: cost of prepaging vs. cost of servicing corresponding page faults

# Page Size

- Issues about page size

| | smaller page | larger page |
|---|---|---|
| **Size of page table** | large | small |
| **Memory utilization** | better | worse |
| **I/O latency** | large | small |
| **Locality** | good | bad |
| **Page fault** | many | few |

  - Historical trend: page size is getting larger

# TLB Reach

- To improve TLB hit ratio, size of TLB should be increased.
  → but associate memory is expensive, power hungry

- TLB reach: amount of memory accessible from TLB
  - TLB reach = <# of entries in TLB> * <page size>

- TLB reach can increase by increasing page size
  - However, with large page, fragmentation also increases.
  
  → S/W managed TLB (OS support several different page sizes)
  
  Ex) UltraSparc, MIPS, Alpha
  
  Cf) PowerPC, Pentium: H/W managed TLB

# Inverted Page Tables

- Inverted page table reduces amount of physical memory needed to memory translation
- However, it no longer contains complete information about logical address space of a process
  - Demand paging requires complete information about logical address space to process page fault
- Remedy: maintaining external page table for each process
  - External page table can be paged out and in

# Program Structure

- User don't have to know about nature of memory. But, if user knows underlying demand paging, performance can be improved
  - Ex) `int[128,128] data;`
    - Each row is stored in one page

* Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

128 x 128 = 16,384 page faults

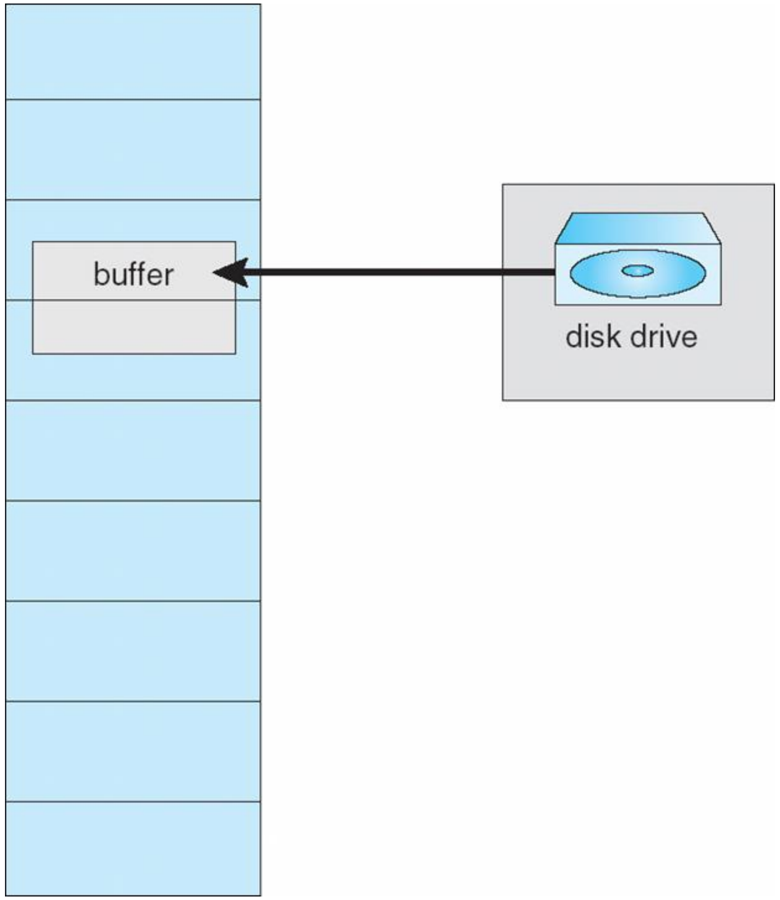
* Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

128 page faults

# I/O interlock

- **I/O Interlock**: Pages must sometimes be locked in memory
- Consider I/O: Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



# Agenda

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- The Linux Virtual Memory System

# The Linux Virtual Memory System

- Please refer to OSTEP!