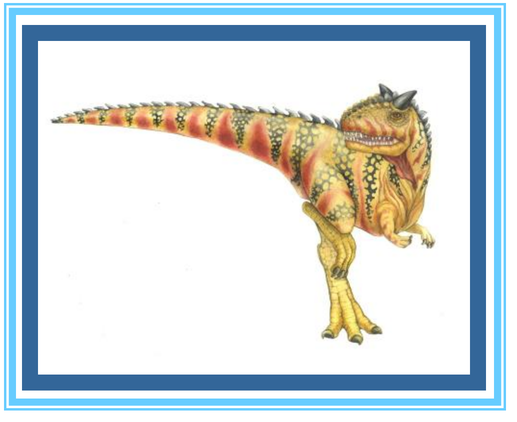
Chapter 9: Virtual Memory (contd.)



- **Slides mostly borrowed from Silberschatz & Galvin**
  - With occasional modifications from us

Virtual memory: recap

- Virtual memory
  - Only part of the program needs to be in memory for execution
  - Logical address space much larger than physical address space

- Demand paging – bring a page (from disk) to physical memory only when needed

- If CPU accesses a page that is not in physical memory: Page Fault
  - Find free frame in memory
  - If no free frame, select victim page/frame to replace
  - Swap required page into frame via scheduled disk operation
  - Update page tables of this process

Algorithms for Virtual Memory

- **Page-replacement algorithm**
  - Which page to replace when a new page is to be swapped in?

- **Frame-allocation algorithm**
  - How many frames to allocate to each process?
  - Which frames can be allocated to pages of a process?

Page Replacement algorithms: a closer look

Page Replacement Algorithms: evaluation

- How to evaluate a page replacement algorithm?

- Evaluate algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

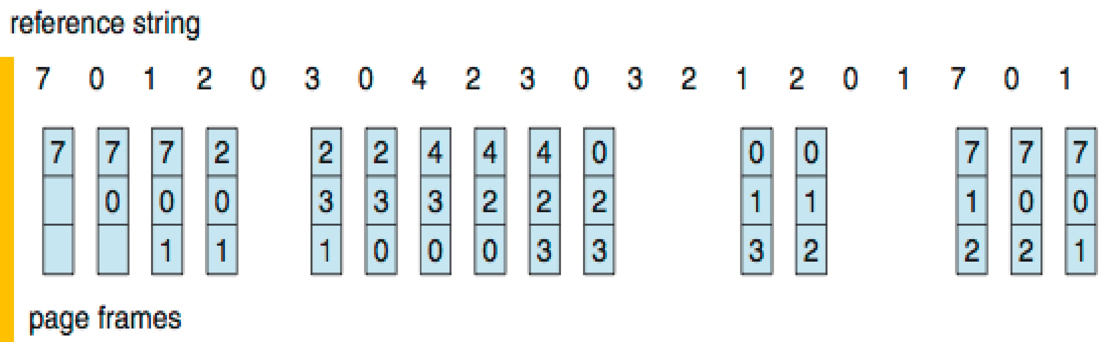
- In our examples, the reference string of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

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**
- Assume 3 frames (3 pages can be in memory at a time per process)

2 replaces 7 since 7 was brought into mem first (among all mem-resident pages)



FIFO algorithm with 3 frames gives 15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

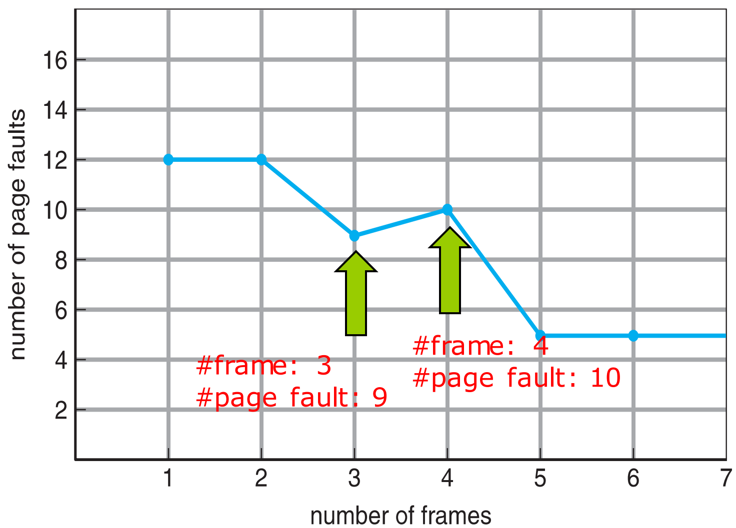
Problem with FIFO: Belady's Anomoly

- Consider reference string: **1,2,3,4,1,2,5,1,2,3,4,5**

- With 3 frames per process, 9 page faults
- With 4 frames per process, 10 page faults

Problem with FIFO: Belady's Anomoly

- Consider reference string: **1,2,3,4,1,2,5,1,2,3,4,5**



- Adding more frames can cause more page faults!
  - **Belady's Anomaly**

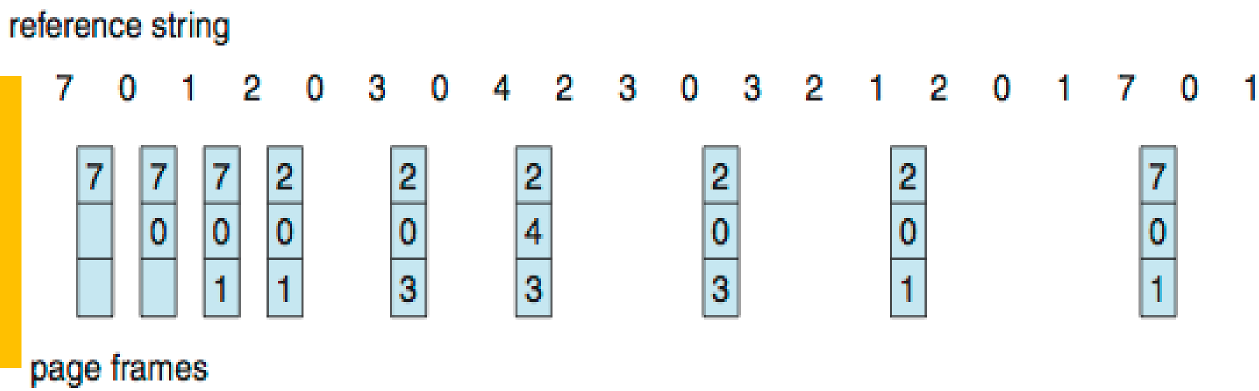
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least 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
- Not commonly used due to complexity

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 page faults is optimal for the example

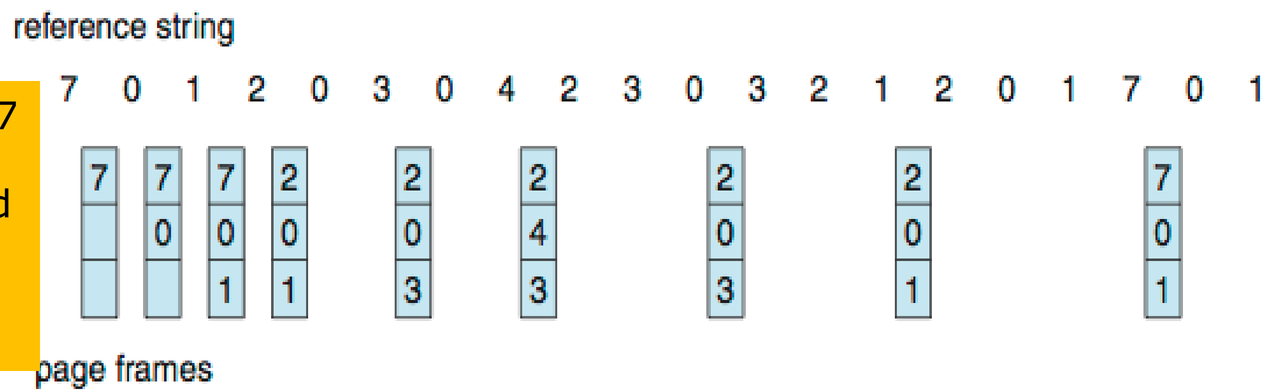
2 replaces 7 (that will not be used for longest period of time)



Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 page faults is optimal for the example

2 replaces 7 (that will not be used for longest period of time)

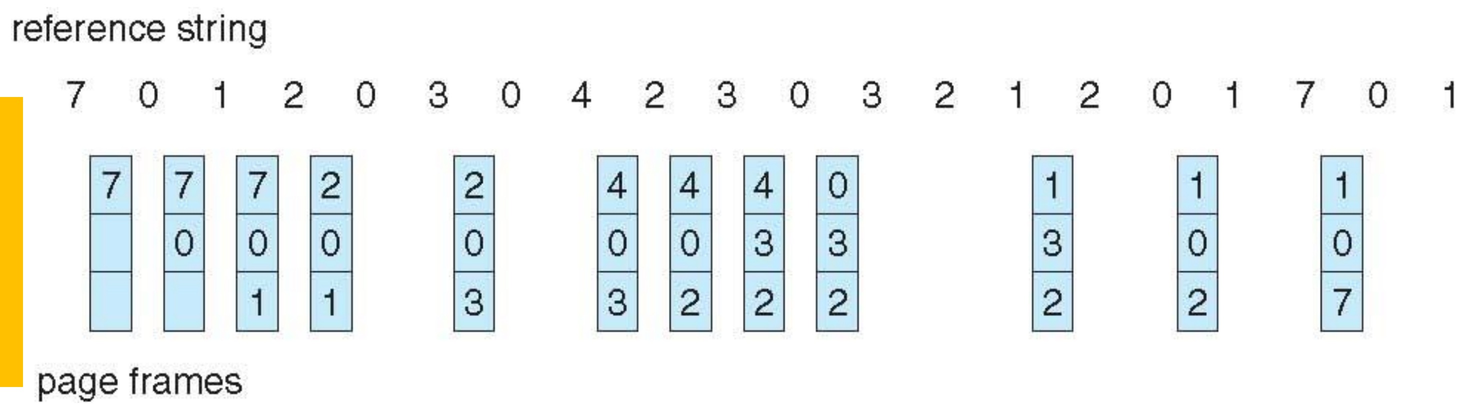


- Problem: You don’t know the future memory references
- Still useful: To measure how well another algorithm performs, compare that algorithm to the optimal algorithm

Least Recently Used (LRU) Algorithm

- 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

2 replaces 7 (that hasn't been used for longest period of past time)



- 12 page faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement? Will be discussed shortly ...

Stack Algorithms

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- **stack algorithms:** the set of pages in a memory with n frames is always a subset of the set of pages that would be in a memory with n+1 frames

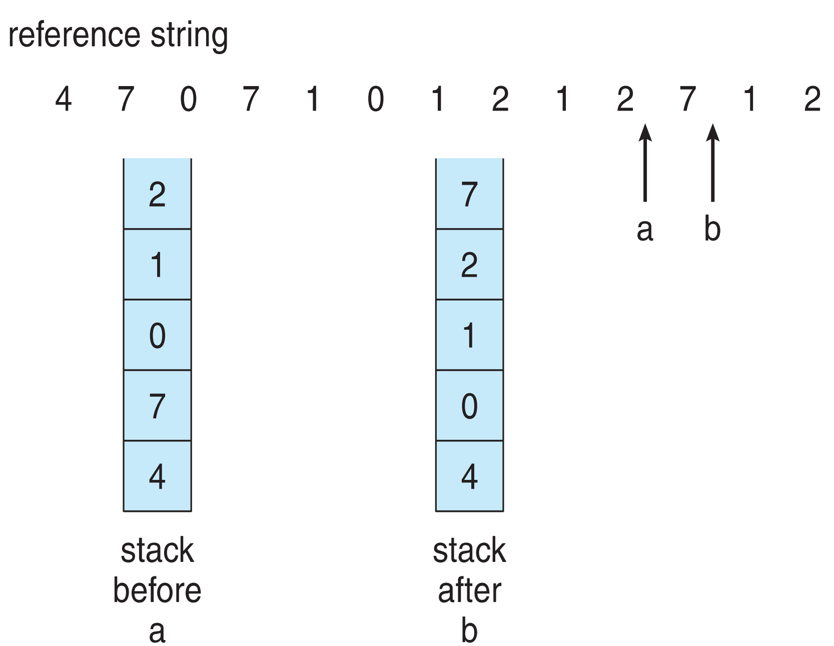
Implementation and approximations of LRU algorithm

LRU Algorithm (Cont.)

- 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
  - Disadvantage: Search through table needed

LRU Algorithm (Cont.)

- Stack implementation
  - Keep a stack of page numbers in a double link form
  - When a page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - Advantage: No search for replacement
  - Disadvantage: But each update more expensive



LRU Approximation Algorithms: Basics

- 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

LRU Approximation Algorithms

- **Additional reference bits algorithm**
  - Record the reference bits at regular intervals
  - Say we keep 8 bits (a byte) for each page in a table in memory
  - At regular intervals, a timer interrupt causes the OS to shift the reference bit for each page into msb and shift other bits to right
    - These 8-bits contain history of page reference for last 8 time periods
    - Page with lowest number is the LRU page

LRU Approximation Algorithms

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - 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

Give the page a second chance

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (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

Frame Allocation algorithms

Allocation of Frames

- Each process needs a ***minimum*** number of frames
  - Example: in IBM 370, minimum 6 pages
- ***Maximum*** of course is total frames in the system

- Two major allocation schemes
  - Fixed allocation
  - Priority allocation
- Many variations

Fixed Allocation

- Equal allocation – e.g., if there are 103 frames (after allocating frames for the OS) and 2 processes, give each process 51 frames
  - Keep some frames as free frame buffer pool

- 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 = 103
s1 = 10
s2 = 127
*a1 = (10/137)*103 = 7*
*a2 = (127/127)*103 = 95*

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size of processes
- If process $\boldsymbol{P_i}$ generates a page fault:
  - Select for replacement one of its own frames
  - Or, select for replacement a frame from a process with lower priority

Global vs. Local Allocation

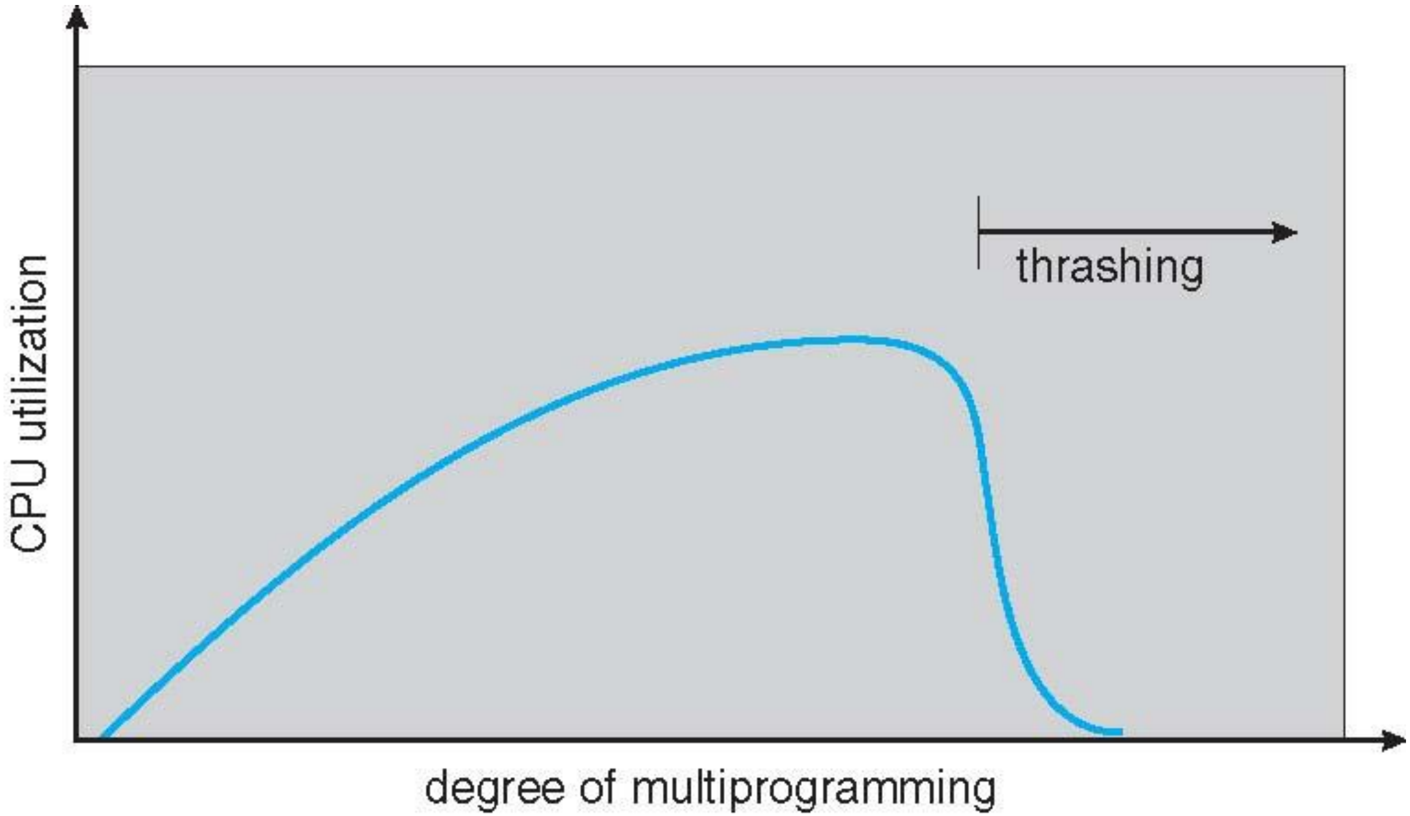
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly (since a process cannot control its own page fault rate)
  - But greater throughput; so more common
  - Also good for considering priority of processes
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing (an undesirable effect of page replacement) and how to avoid it

Thrashing

- If a process does not have “enough” pages in the memory, the page-fault rate is very high
  - Page fault to get page
  - Replace existing page (of the same process)
  - But quickly need replaced page back, ...
  - This leads to:
    - Low CPU utilization
    - Operating system may think that it needs to increase the degree of multiprogramming
    - Another process added to the system – aggravation
- **Thrashing:** a process is busy swapping pages in and out (no useful work gets done)

Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another (e.g., a function call)
  - Localities may overlap
  - We need to allocate enough frames to a process to accommodate its current locality

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another (e.g., a function call)
  - Localities may overlap
  - We need to allocate enough frames to a process to accommodate its current locality (which can get defined by the program structure)

```
for i = 1 to 128:
    for j = 1 for 128:
            A[i][j]++
```

```
for j = 1 to 128:
    for i = 1 for 128:
            A[i][j]++
```

Consider 512 byte pages and 4 bytes per element of array A

Each row of the matrix A is stored in one page

=> 128 vs. 128x128 page faults

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another (e.g., a function call)
  - Localities may overlap
  - We need to allocate enough frames to a process to accommodate its current locality (which can get defined by the program structure)

```
for i = 1 to 128:
    for j = 1 for 128:
            A[i][j]++
```

```
for j = 1 to 128:
    for i = 1 for 128:
            A[i][j]++
```

Consider 512 byte pages and 4 bytes per element of array A

Each row of the matrix A is stored in one page

=> 128 vs. 128x128 page faults

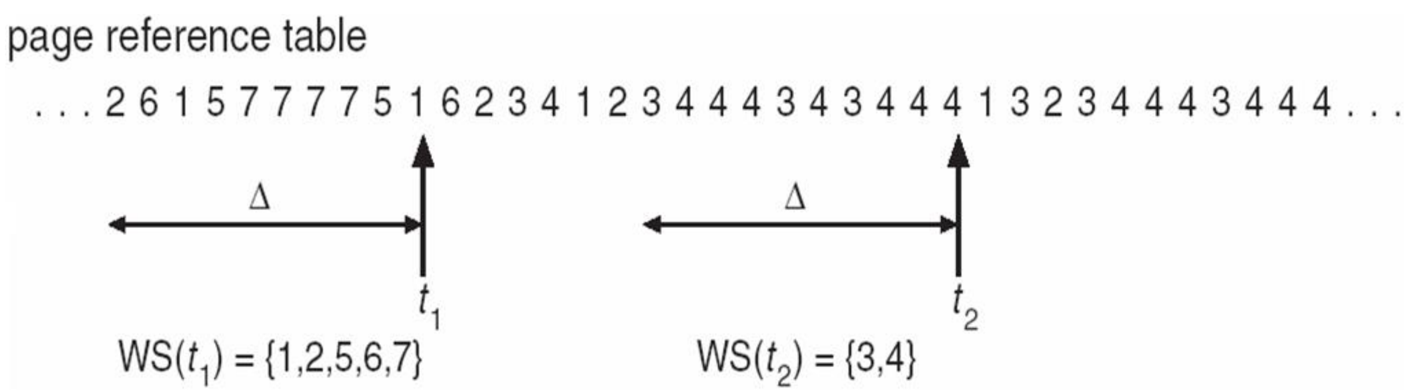
- Why does thrashing occur?

Σ size of locality across all processes > total memory size

Working-Set Model

- Δ : working-set window : a certain number of page references
  Example: all page references made in 10,000 instructions

- WS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small, will not encompass entire locality
  - if Δ too large, will encompass several localities
  - if Δ = infinity, will encompass entire program



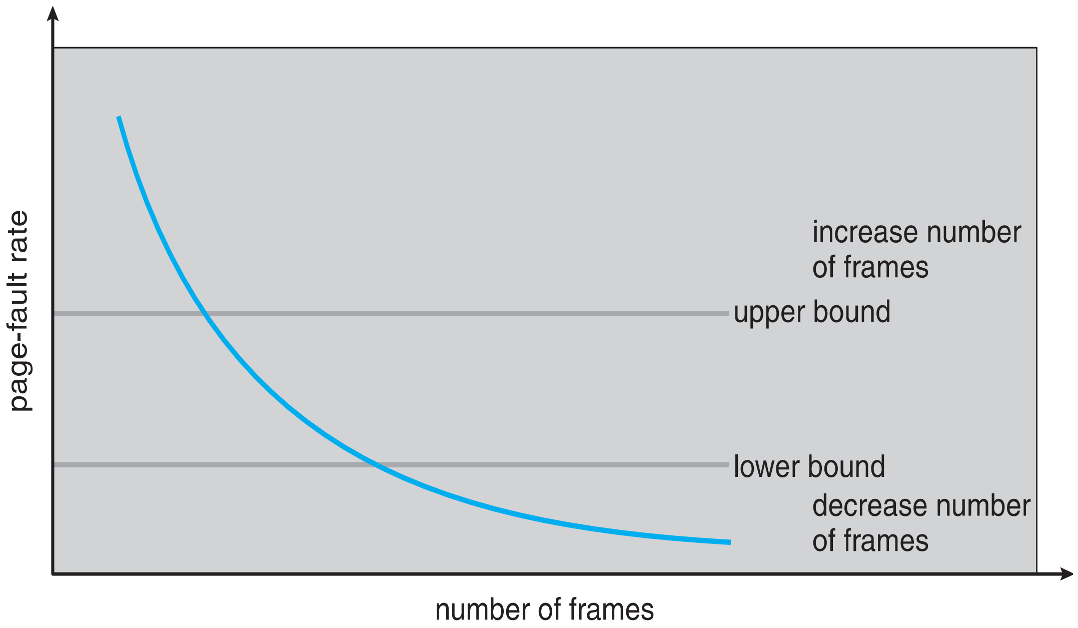
Working-Set Model

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

- $D = \Sigma\ WS_i$: total demand for frames across all P_i (an approximation of locality)
- if $D > m$ --> Thrashing
- Policy: if D > m, then suspend or swap out one of the processes

Page-Fault Frequency

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



Some other issues about Paging

Other Issues – Page Size

- Sometimes OS designers have a choice about page size
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - Resolution
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time as processes need more memory

Other Issues – Prepaging

- Prepaging
  - Prepage all or some of the pages a process will need, before they are referenced
  - To reduce the large number of page faults that occurs at process startup
  - But if prepaged pages are unused, I/O and memory was wasted

- Assume *s* pages are prepaged and *α* of the pages is used
  - Is cost of ***s * α*** save pages faults > or < than the cost of prepaging ***s * (1- α)*** unnecessary pages?
  - ***α*** near zero --> prepaging loses; hence need to select which pages to prepage intelligently

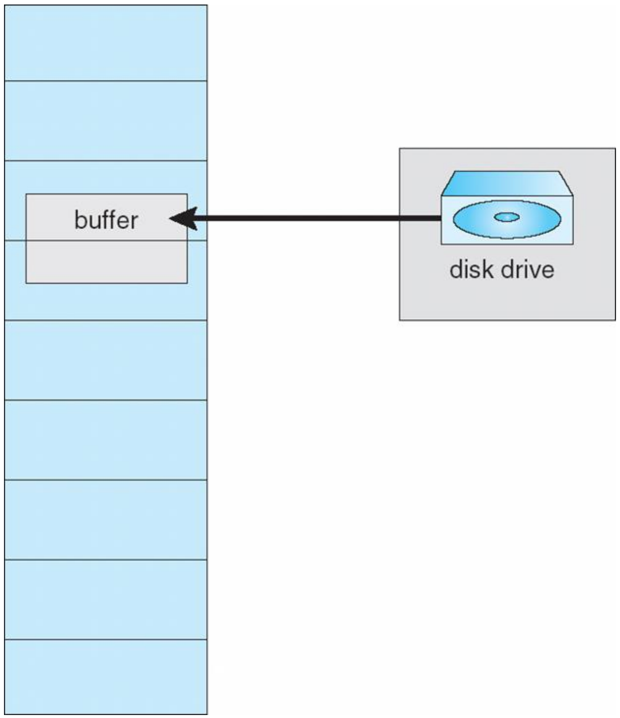
Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- **Pinning** of pages to lock into 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



Allocating kernel memory

Allocating Kernel Memory

- When some module of the kernel asks for memory
  - Some OS may treat such a request differently from a user process asking for memory
  - Memory may be allocated from a free-memory pool different from the list of frames used for user-processes
- Why some OS opt of different treatment?
  - Kernel requests memory for structures of varying sizes; must use memory conservatively (e.g., if demand << page size)
  - Some kernel memory needs to be contiguous
    - E.g., some hardware devices interact directly with physical memory assuming the memory to be contiguous
- Two approaches
  - Buddy system
  - Slab allocation

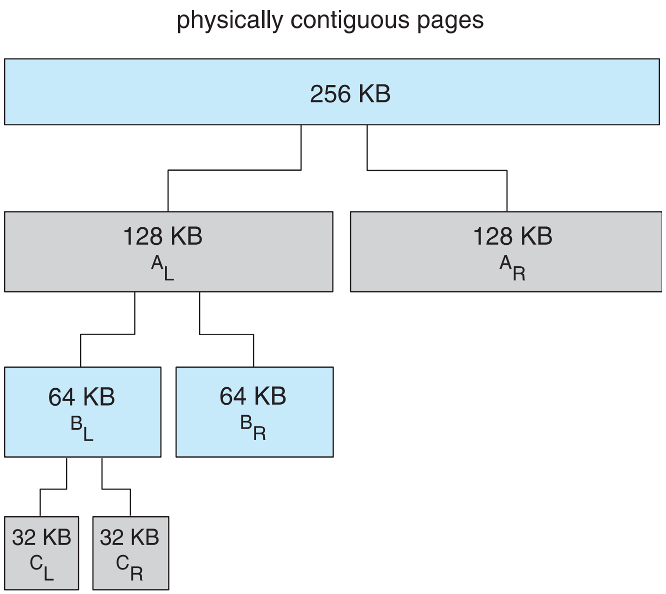
Buddy System

- 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 appropriate sized chunk available

Buddy System Allocator

- 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 division into C_L and C_R of 32KB each – one used to satisfy request

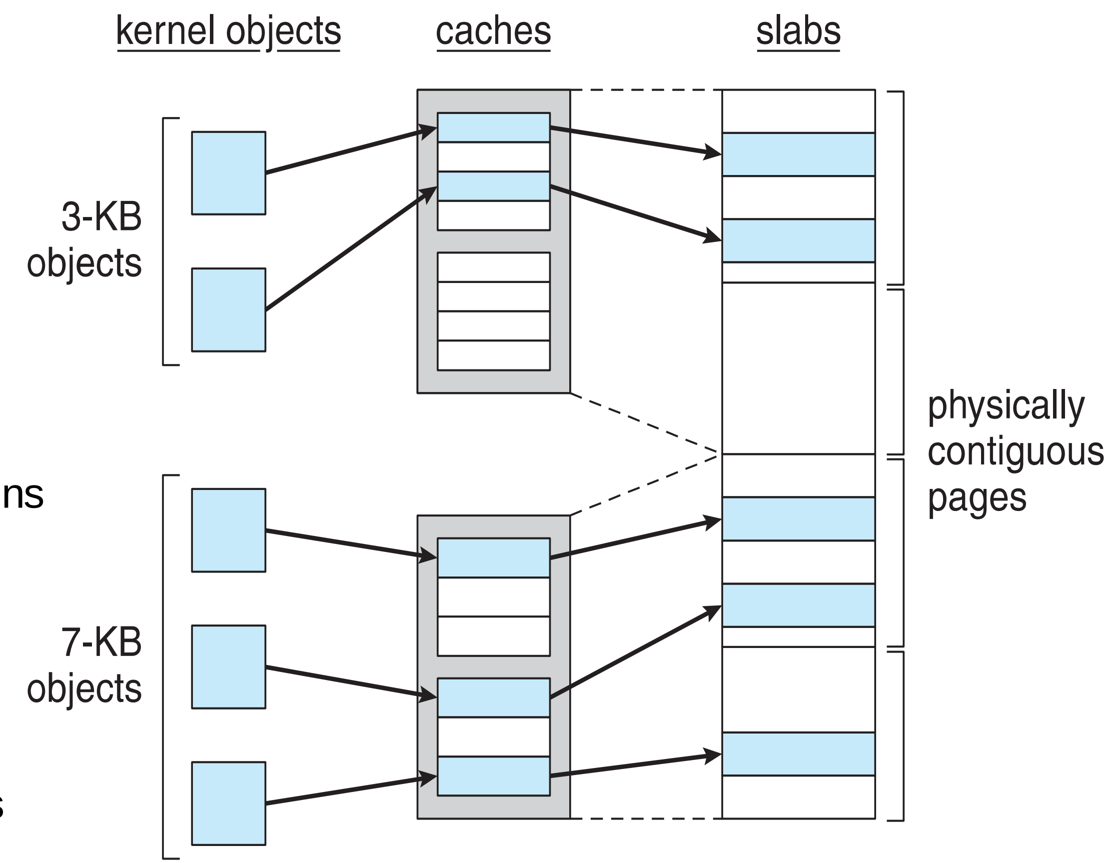


- Advantage – can quickly **coalesce** unused chunks into larger chunk
- Disadvantage – fragmentation within allocated segment

Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs

- A single cache for each unique kernel data structure
  - Each cache filled with **objects** that are instantiations of the data structure
  - Examples
    - a cache for process descriptors
    - a cache for semaphores
    - a cache for file objects



Slab Allocation

- When cache created, filled with objects marked as **free**
- When a new object for a kernel data structure is needed, a free object allocated and marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits:
  - no fragmentation
  - fast memory request satisfaction