CGS 3763: Operating System Concepts
Spring 2006

Memory Management – Part 7

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cgs3763/spr2006

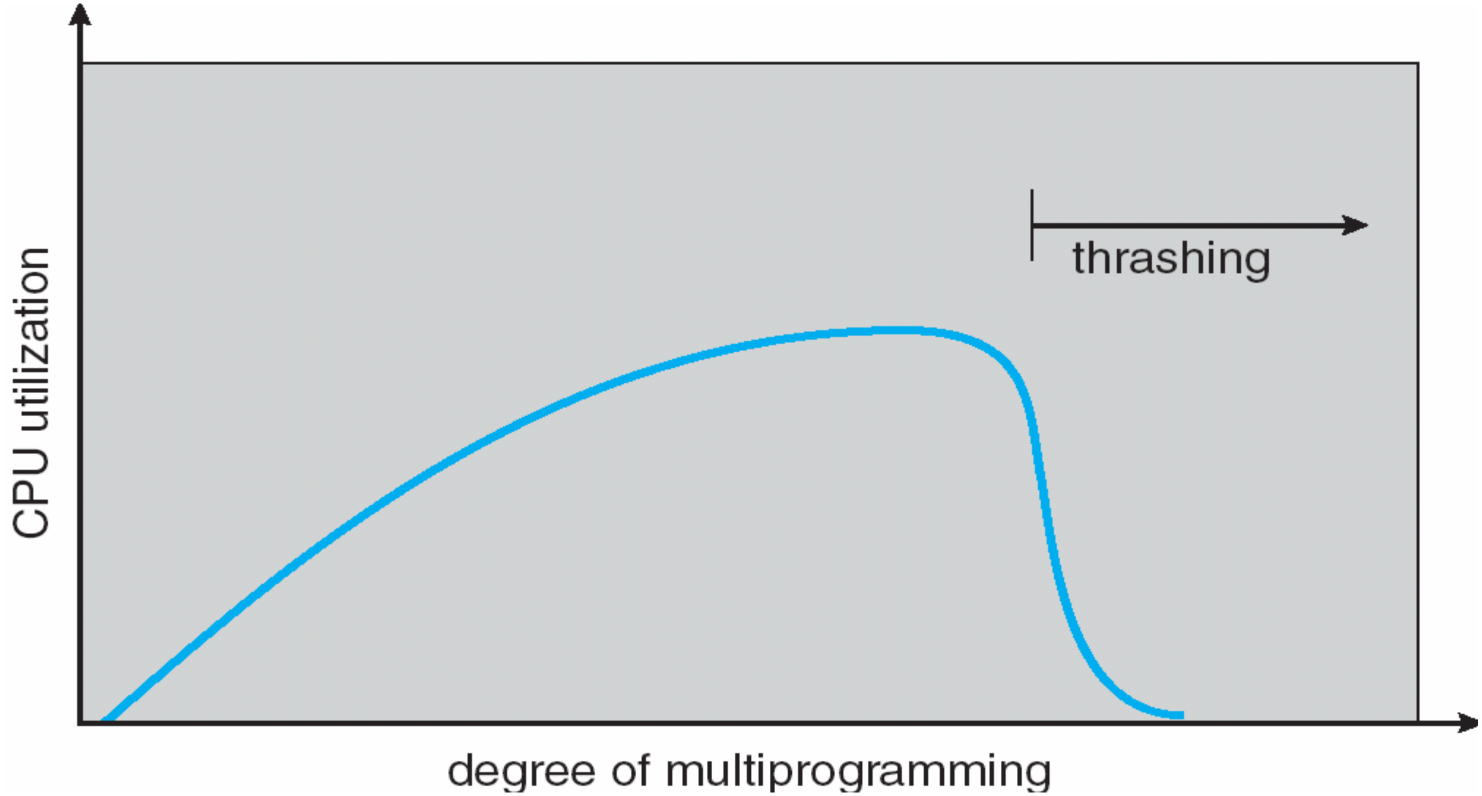
School of Electrical Engineering and Computer Science
University of Central Florida

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high.
- This leads to:
  - low CPU utilization
  - operating system thinks 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 without accomplishing any real activity. The process will spend most of the time in the queue for the paging device.

Thrashing (cont.)

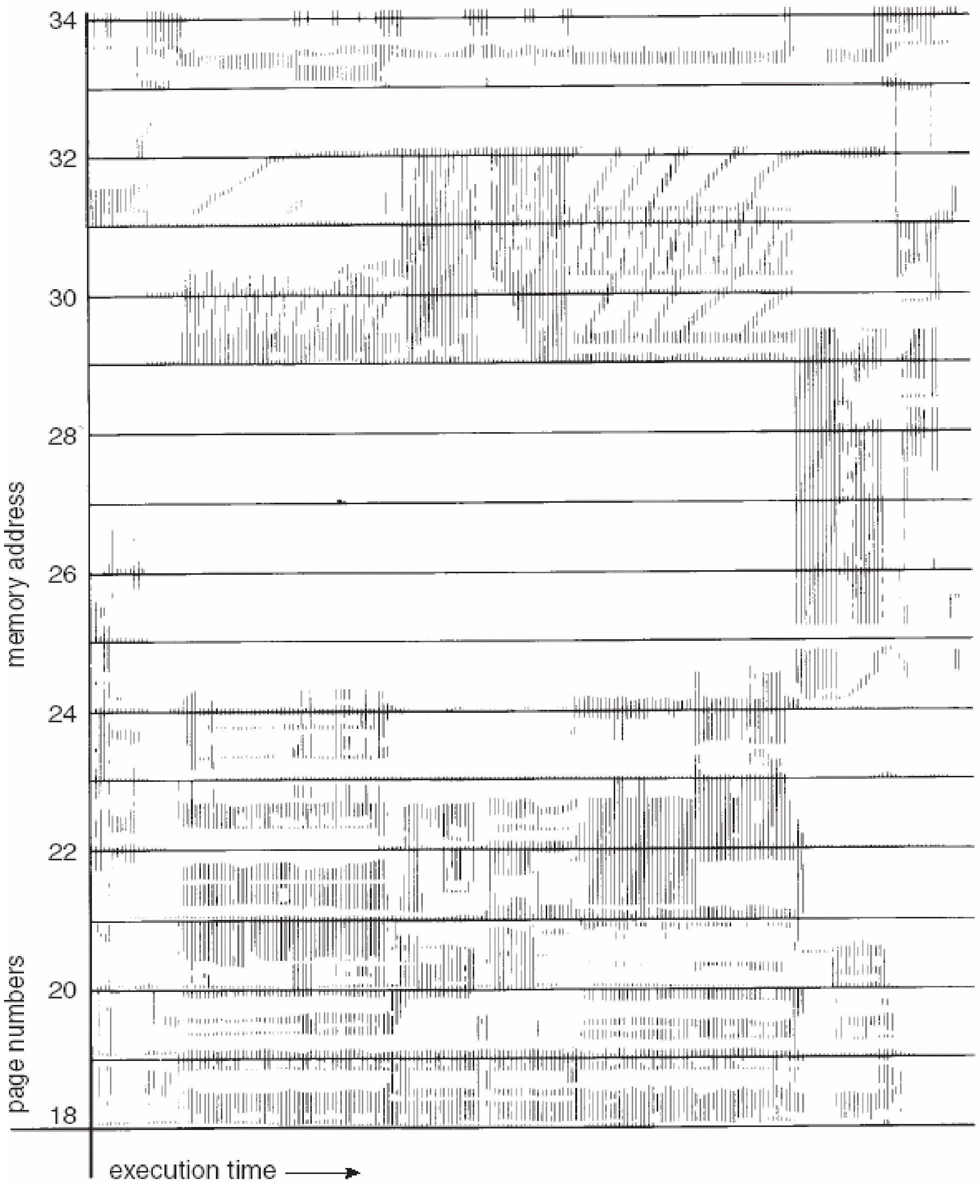


Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - A locality is a set of pages that are actively used together.
  - A program is generally composed of several different localities.
  - A process migrates from one locality to another during execution.
  - Localities may overlap.

- Why does thrashing occur?
  Σ size of locality > total frame allocation for the process

Locality In A Memory-Reference Pattern



Working-Set Model

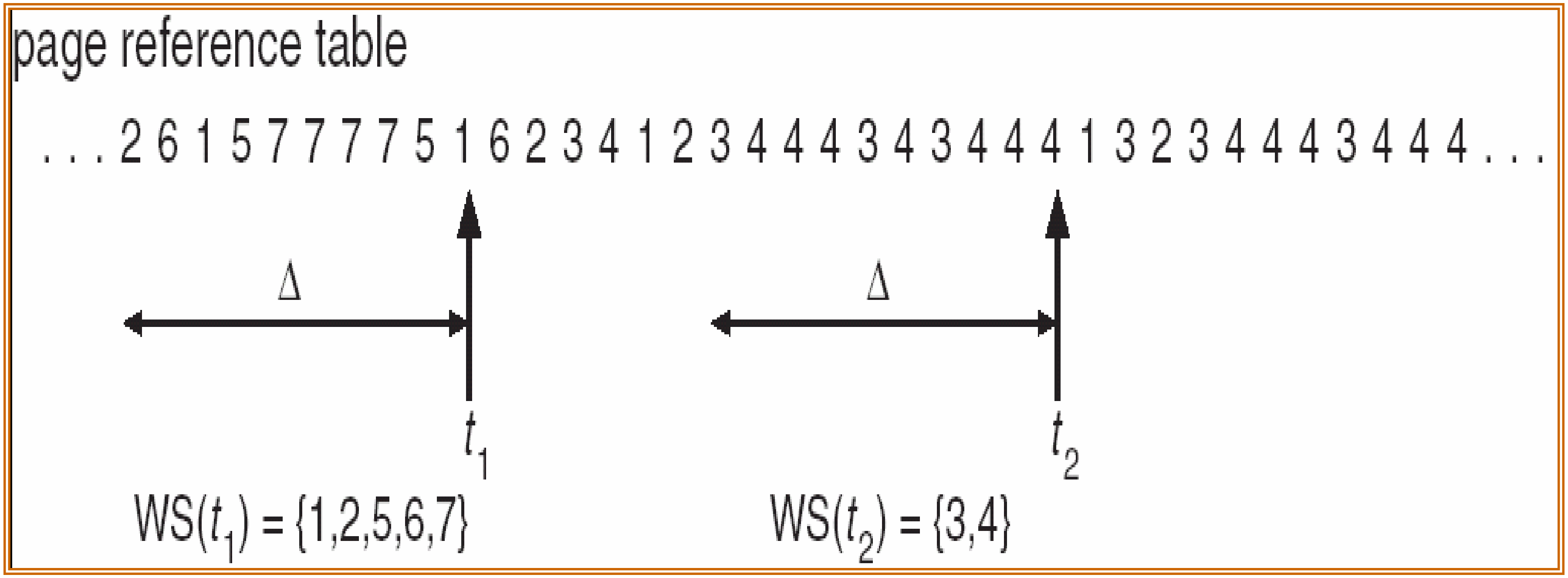
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references Example: 10,000 instructions.
- WSS_i (working set size of process P_i) =

  total number of pages referenced in the most recent Δ

  (time variant)

  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model Example



Assume Δ = 10

Working Set Model – Another Example

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- Δ = 3

reference string

		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
WS	1	7	7	7	2	2	2	0	0	2	2	2	0	2	2	2	2	2	1	1	1
	2		0	0	0	0	0	3	4	4	4	0	3	0	3	0	0	0	0	0	0
	3			1	1	1	3		3	0	3	3		3	1	1	1	1	7	7	7
WSS		1	2	3	3	3	3	2	3	3	3	3	2	3	3	3	3	3	3	3	3
		🟥	🟥	🟥	🟥		🟥		🟥	🟥	🟥	🟥		🟥	🟥		🟥		🟥		

13 page faults

Working-Set Model (cont.)

- Once Δ has been selected, using the working set model is simple.
- The OS monitors the working set of each process and allocates to that working set enough page frames to provide it with its working set size.
- If there are enough extra frames, another process can be initiated.
- If the sum of the working set sizes increases, exceeding the total number of available frames, the OS will select a process to suspend. The suspended process's pages are swapped out, and its frames are reallocated to other processes. The suspended process will be restarted later.
- The working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible, thus optimizing CPU utilization.

Keeping Track of the Working Set

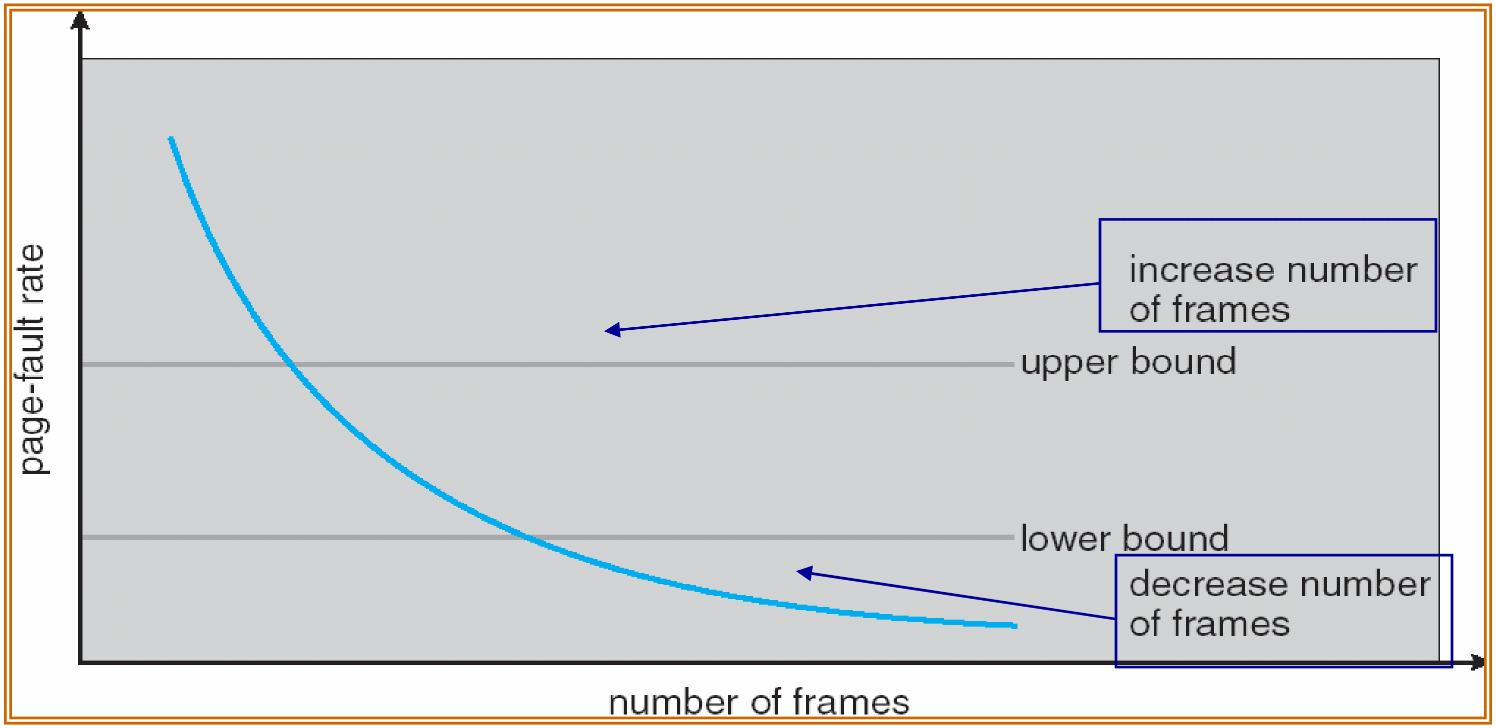
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- Why is this not completely accurate? Because you can't tell where, within an interval of 5000, a reference occurred.
- Improvement = 10 bits and interrupt every 1000 time units. The disadvantage to this approach is higher cost to service more frequent interrupts.

Page-Fault Frequency Scheme

- While the working set model is successful, and knowledge of the working set can be useful for prepaging (more later), it is a clumsy mechanism for controlling thrashing.
- A strategy that uses the page-fault frequency (PFF) is a more direct approach for controlling thrashing.
- Since thrashing exhibits a very high page fault rate, we need to control the page fault rate.
  - Too high a page fault rate implies that a process needs more page frames.
  - Too low a page fault rate implies that a process may have more page frames than it needs.
- Establish upper and lower bounds on the page fault rate.

Page-Fault Frequency Scheme

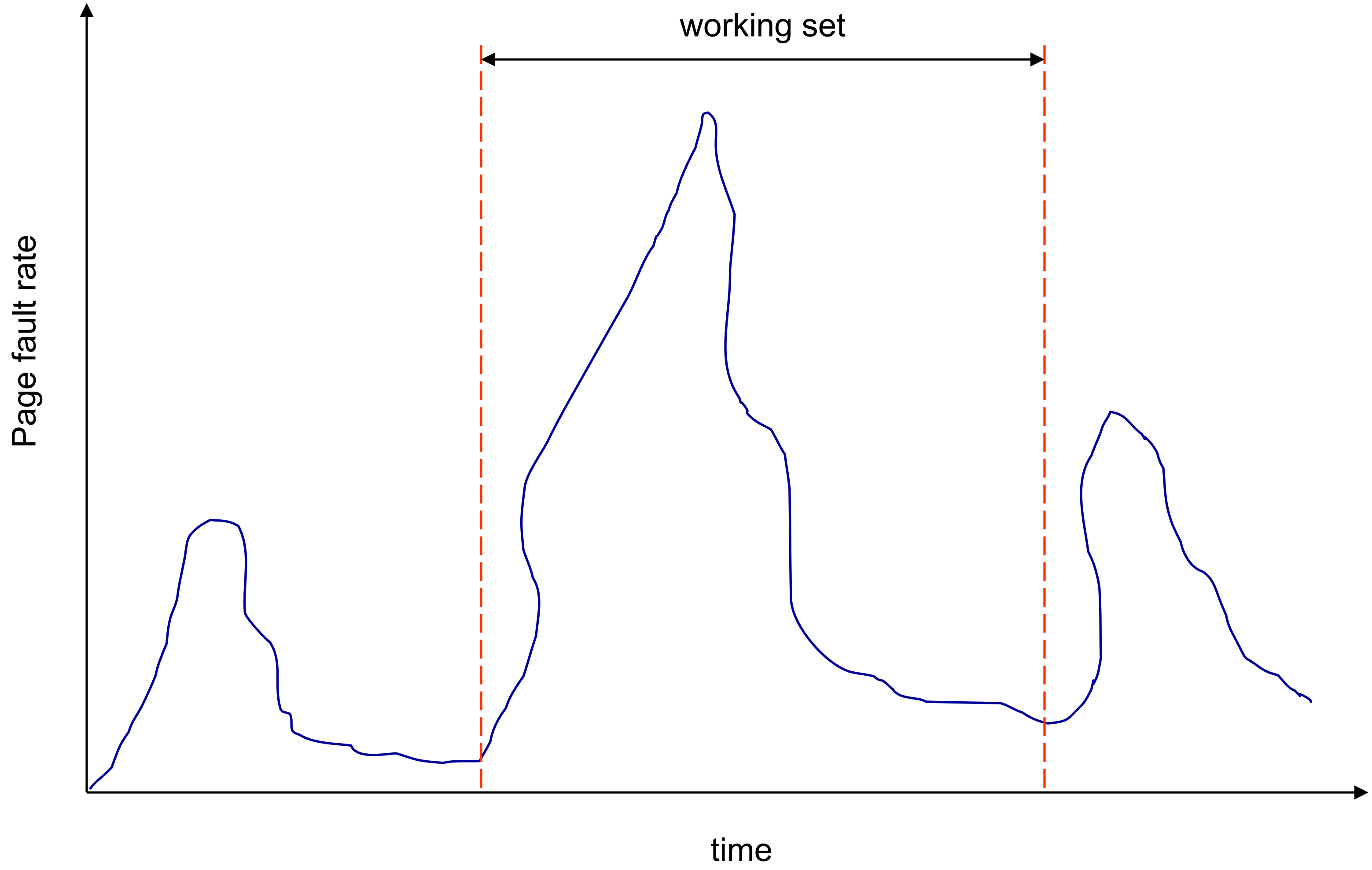
- Establish an “acceptable” page-fault rate
  - If the actual page fault rate is too low, process loses frames.
  - If the actual page fault rate is too high, process gains frames



Working Sets and Page Fault Rates

- There is a direct relationship between the working set of a process and its page fault rate.
- As shown in the example on page 7, typically the working set of a process changes over time as references to code and data sections move from one locality to another.
- Assuming that the process is not thrashing (i.e., it has a sufficient frame allocation), the page fault rate of the process will transition between peaks and valleys over time.
- This general behavior is illustrated on the next page.

Working Sets and Page Fault Rates (cont.)



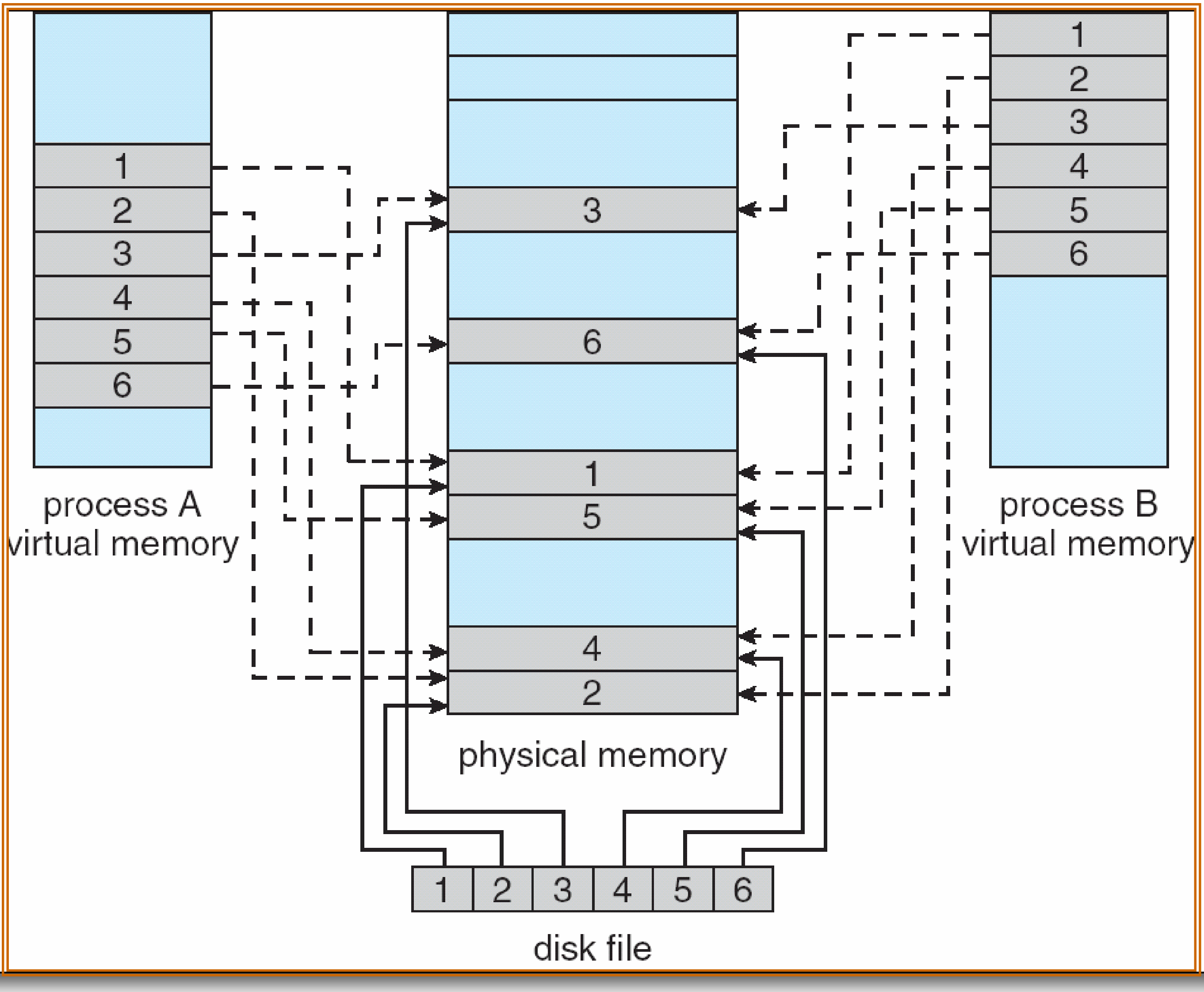
Working Sets and Page Fault Rates (cont.)

- A peak in the page fault rate occurs when demand paging begins in a new locality.
- Once, the working set of the new locality is in memory, the page fault rate falls.
- When the process moves to a new working set, the page fault rate rises towards a peak once again, returning to a lower rate once the new working set is in memory.
- The span of time between the start of one peak and the start of the next peak illustrates the transition from one locality to another (one working set to another).

Memory-Mapped Files

- Memory-mapped file I/O 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 physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **`read()`** **`write()`** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.

Memory Mapped Files



Allocating Kernel Memory

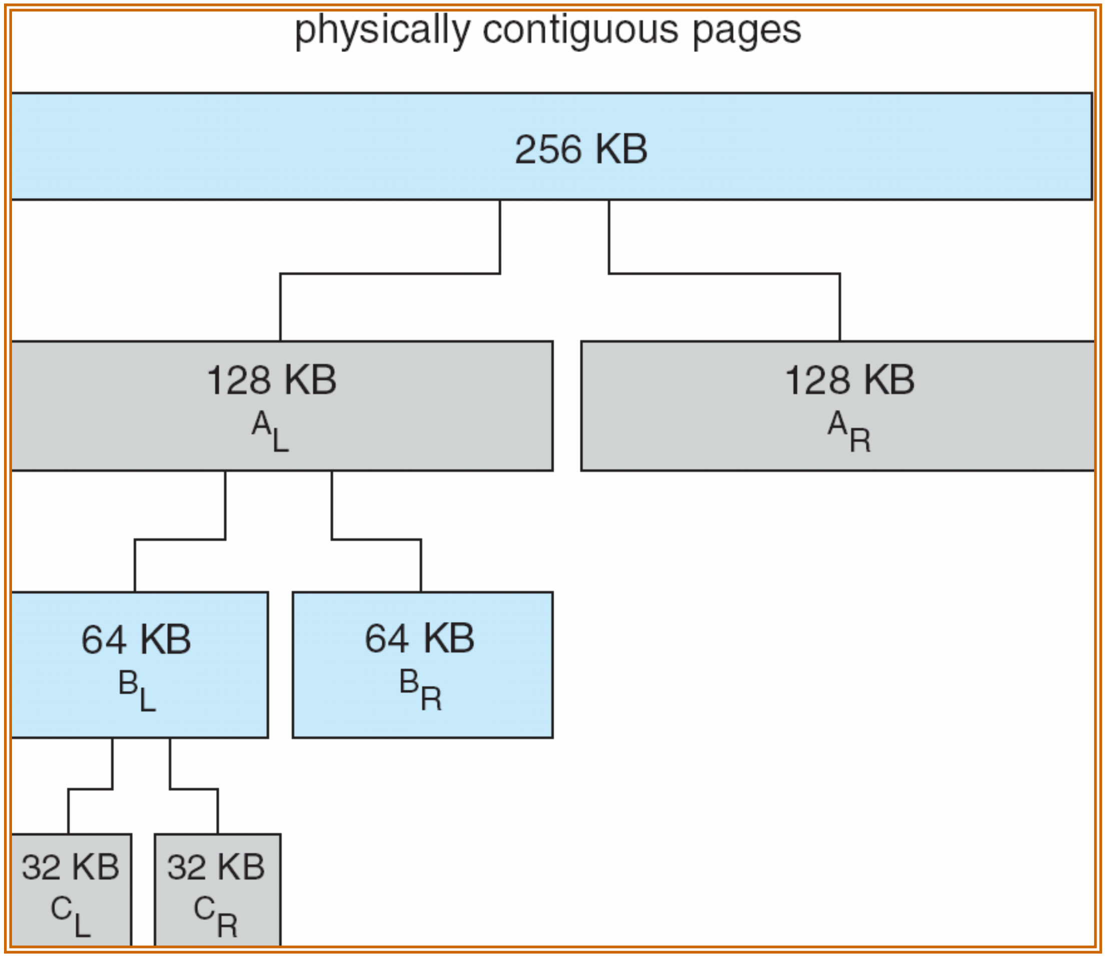
- When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernet.
- Most likely, the free pages are scattered throughout the physical memory (they are not contiguous pages).
- Kernel memory is treated differently from user memory.
- Often allocated from a free-memory pool different from that used to satisfy normal user-mode requests. There are two primary reasons for doing this:

1. Kernel requests memory for structures of varying sizes (often less than one page in size).
2. Some kernel memory needs to be contiguous as some hardware devices interact directly with physical memory – without the benefit of a virtual memory.

Buddy System

- The “buddy system” allocates memory for the kernel from a fixed-size segment consisting of physically-contiguous pages.
- Memory allocated from this segment 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
- The example on the next page illustrates a kernel request for 21 KB of memory from an original segment of 256 KB.

Buddy System Allocator



Since the next power of 2 division would be a 16 KB split which is too small to satisfy the require, the 21 KB request is allocated from one of these two buddies.

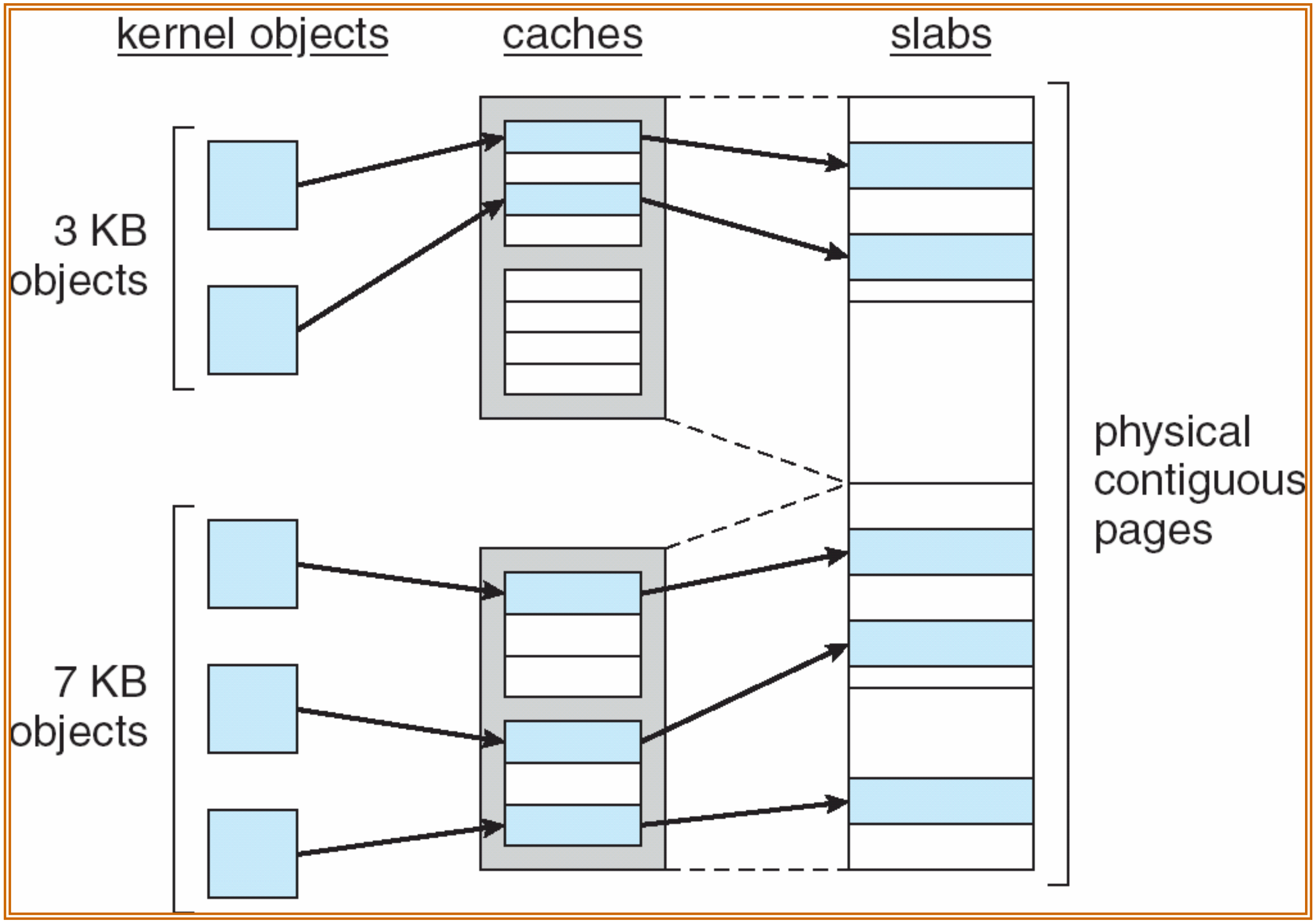
Buddy System (cont.)

- The advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing.
- In the previous example, when the kernel releases the C_L unit it was allocated (let's assume that C_L was the segment allocated to the kernel), the system will coalesce C_L and C_R into a 64 KB segment B_L. Assuming no further allocations occurred, the B_L and B_R would be coalesced into form a 128 KB segment. Eventually, the original 256 KB segment would be reconstructed.
- The obvious drawback to the buddy systems it that rounding up to the next higher power of 2 is very likely to cause internal fragmentation within the allocated segments. For example, a 33 KB request can only be satisfied with a 64 KB segments. In fact, there is no guarantee that less than 50% of the allocated segment will be wasted due to internal fragmentation.

Slab Allocator

- An alternative to the buddy system which solves the internal fragmentation problem.
- A **slab** is one or more physically contiguous pages.
- A **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 and fast memory request satisfaction.

Slab Allocation



Other Issues -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - 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 $s * \alpha$ save pages faults > or < than the cost of prepaging
$s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

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
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Issues – Program Structure

- Program structure
  - `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

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked 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

Reason Why Frames Used For I/O Must Be In Memory