

Chapter 9: Virtual Memory

Administrivia

- Read Chapter 10.

Outline

- Frame allocation mechanisms.
- Thrashing.
- Working set and page fault frequency models.
- Kernel memory allocation mechanisms.
- Design issues: pre-paging, page size, TLB reach, program design, locking pages in memory.
- Examples: Windows XP and Solaris.

Allocation of Frames

- Each process needs *minimum* number of pages
- 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*
- Two major allocation schemes
 - fixed allocation
 - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
 - s_i = size of process p_i
 - $S = \sum s_i$
 - m = total number of frames
 - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
 - select for replacement one of its frames
 - select for replacement a frame from a process with lower priority number

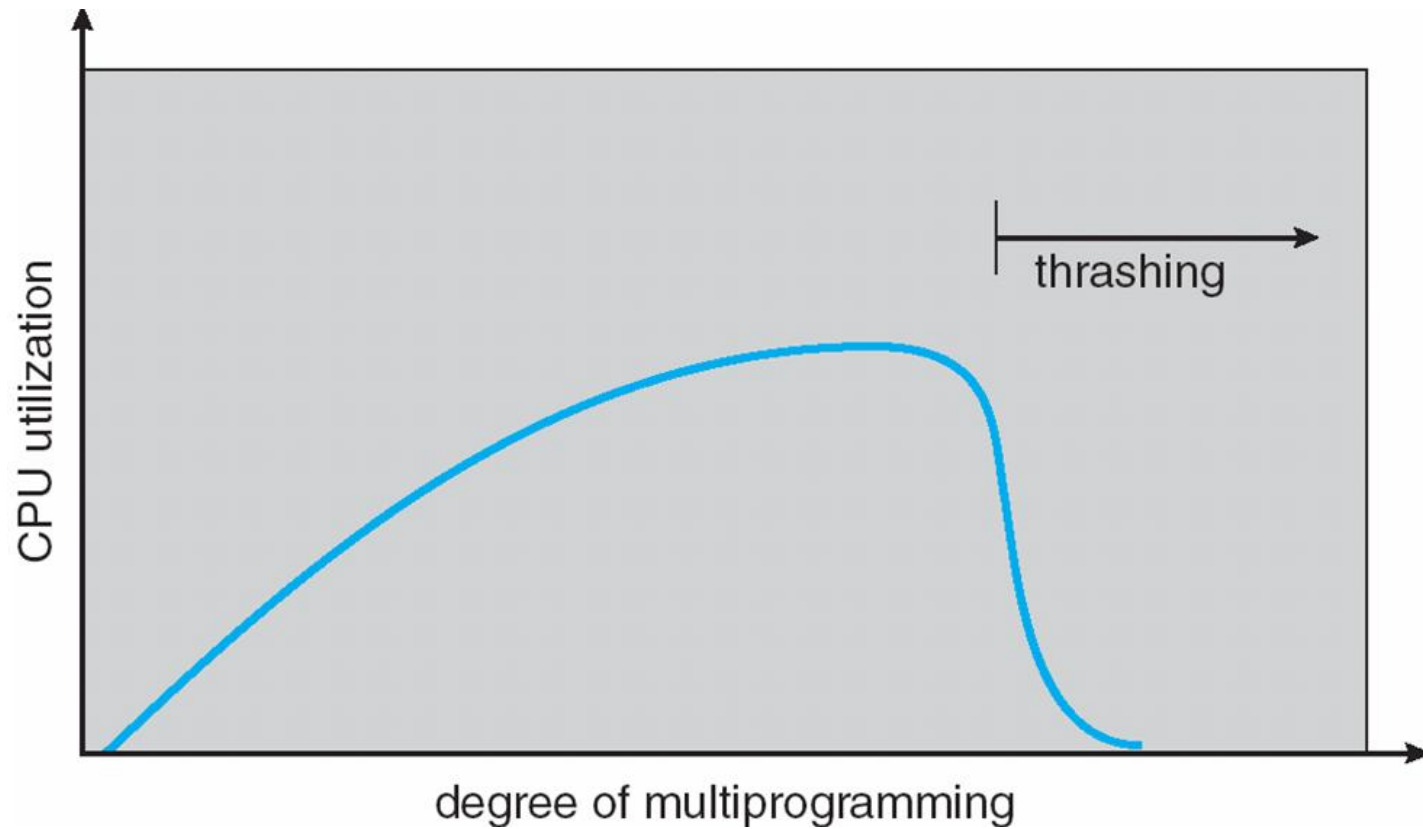
Global vs. Local Allocation

- **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

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** \equiv a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work?
Locality model
 - Process migrates from one locality to another
 - Localities may overlap

- Why does thrashing occur?
 Σ size of locality > total memory size

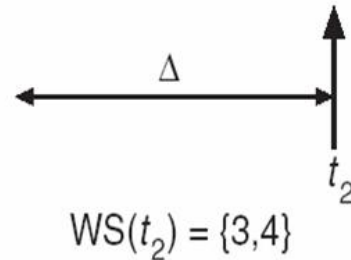
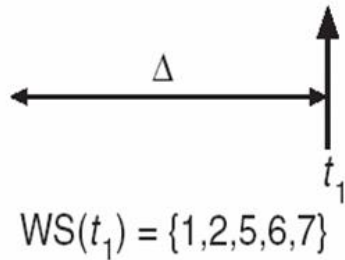
Working-Set Model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instruction
- WSS_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 $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes

Working-set model

page reference table

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

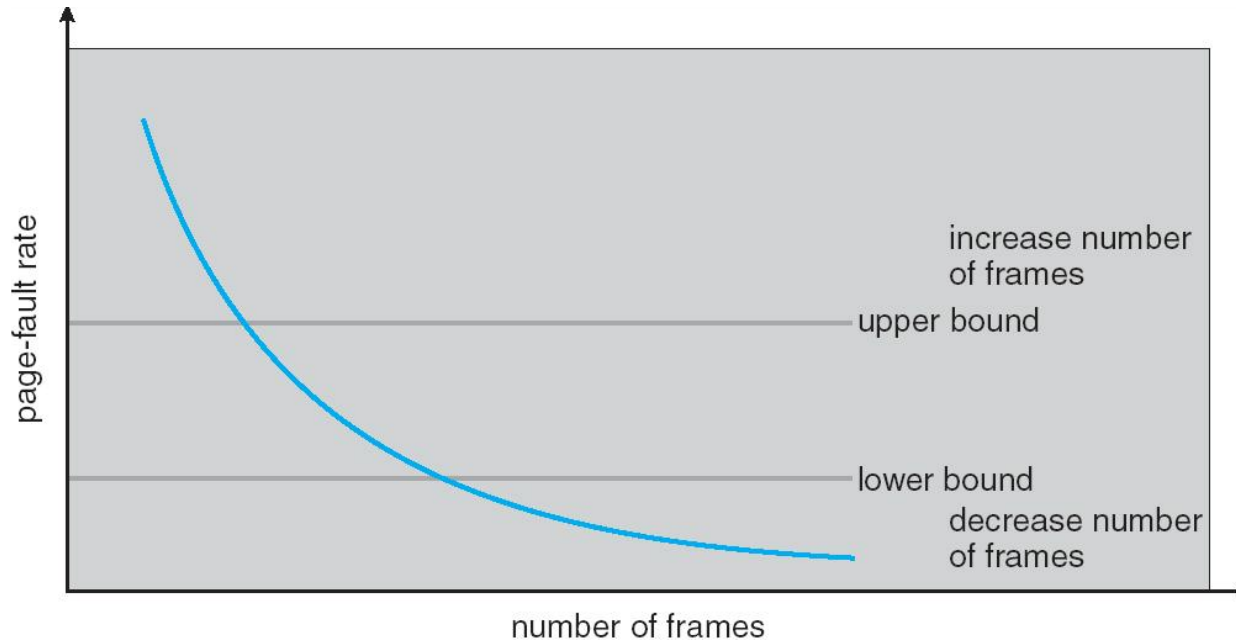


Keeping Track of the Working Set

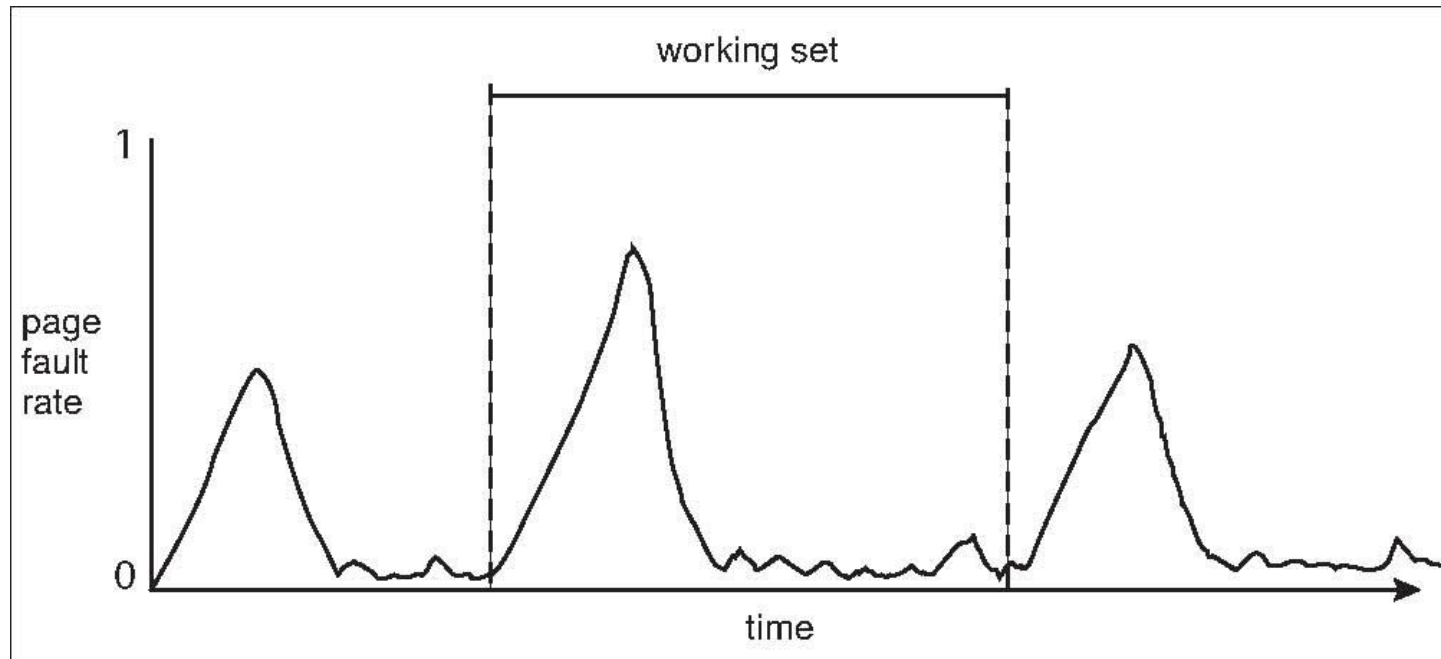
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame



Working Sets and Page Fault Rates



Allocating Kernel Memory

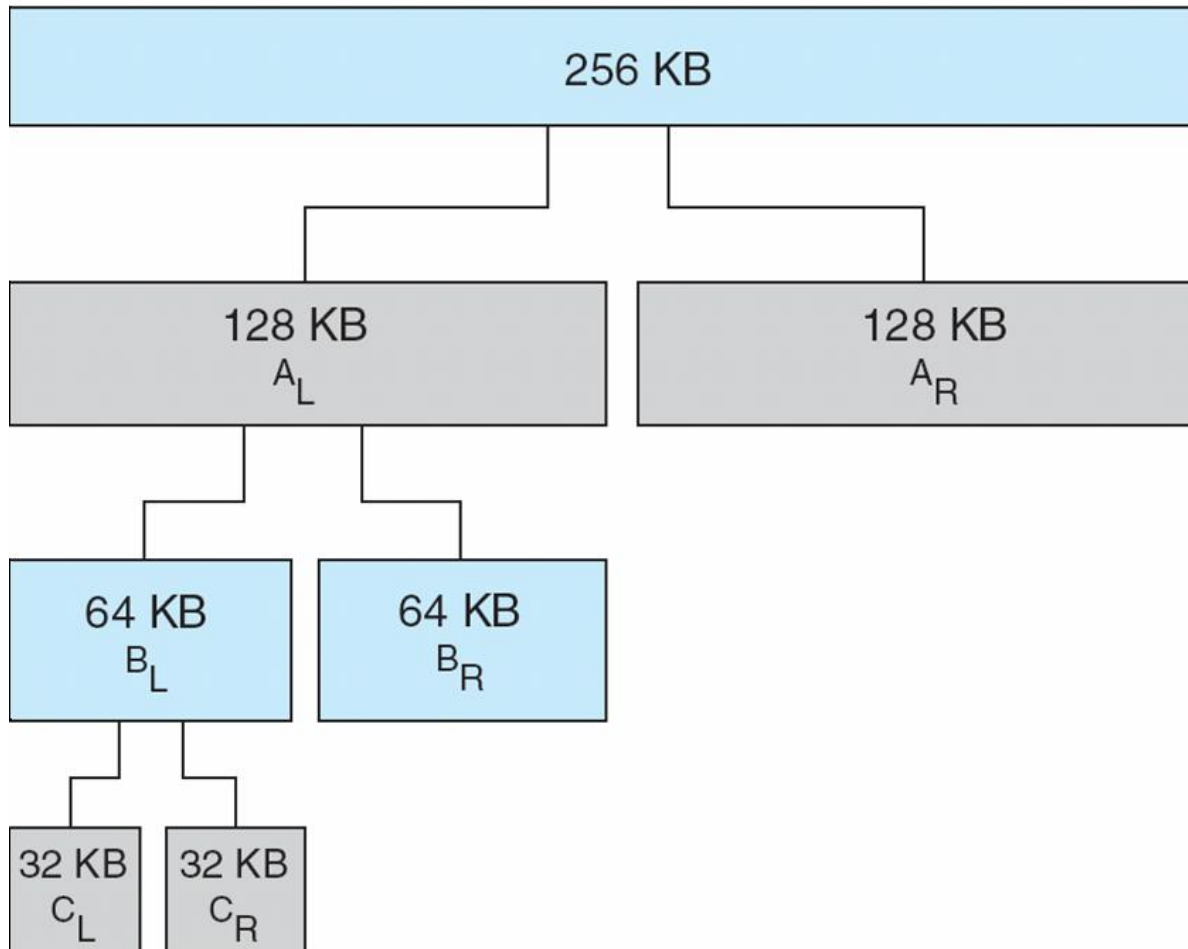
- Treated differently from user memory
- Often allocated from a free-memory pool
 - Kernel requests memory for structures of varying sizes
 - Some kernel memory needs to be contiguous

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

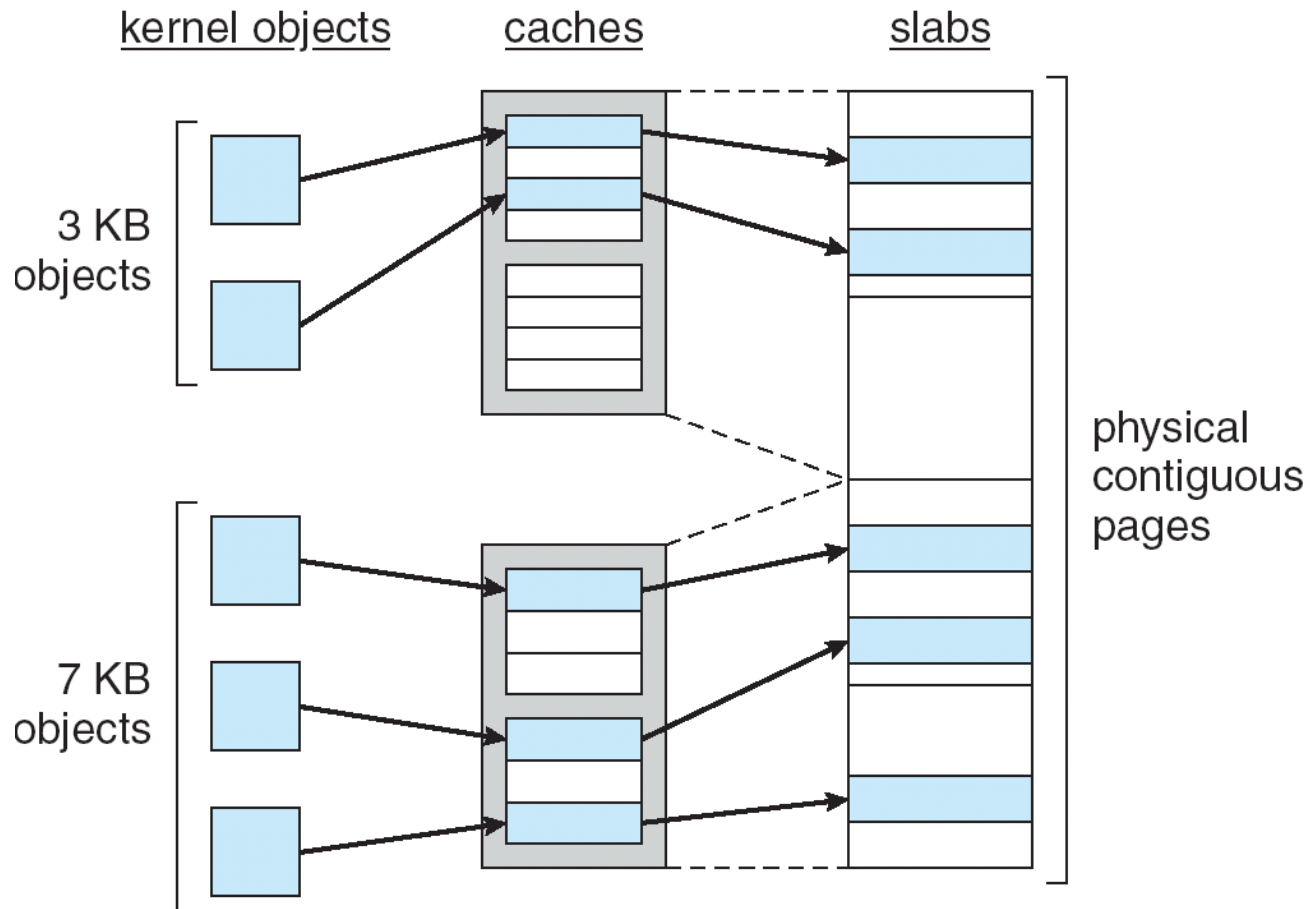
physically contiguous pages



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
- 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



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
- $TLB\ Reach = (TLB\ Size) \times (Page\ Size)$
- 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

Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner

