Chapter 9: Virtual Memory Part II

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example:
  - PDP-11 – 16-bit mini-computer
    - 6 pages required to handle SS MOVE instruction:
    - Instruction is 6 bytes, might span 2 pages
    - The block of characters to move and the area to which it is to be moved can each also straddle two pages. (2 pages to handle *from*, 2 pages to handle *to*)
- ***Maximum*** of course is the number of total frames in the system

- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

– $s_i = \text{size of process } p_i$

– $S = \sum s_i$

– $m = \text{total number of frames}$

– $a_i = \text{allocation for } p_i = \dfrac{s_i}{S} \times m$

$$m = 62$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Priority Allocation

- Use a proportional allocation scheme wherein the ratio of frames depends on the priorities of processes or on a combination of size and priority

- If process P_i generates a page fault,
  - select for replacement one of its frames (local)
  - select for replacement a frame from a process with lower priority number (global)

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

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Non-Uniform Memory Access

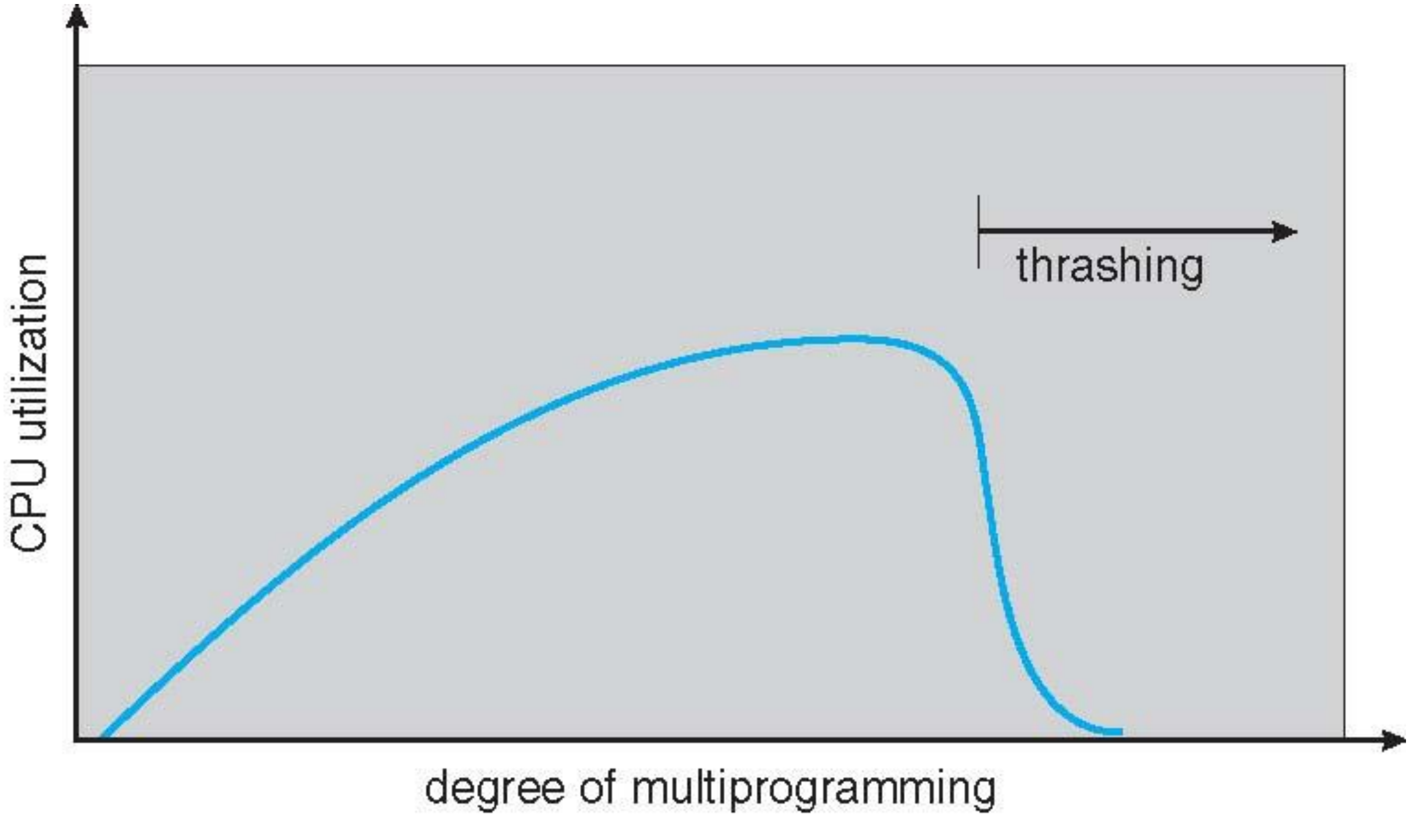
- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solaris creates **lgroups** (for "latency groups")
    - lgroup: Structure to track **CPU / Memory** low latency groups
    - Used by scheduler and pager
    - When it's possible, OS schedules all threads of a process and allocates all memory of the process within an lgroup

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 to replace the frame back
  - Scenario
    - Low CPU utilization
    - Operating system thinks that it needs to increase the degree of multiprogramming to increase the utilization
    - Another process is added to the system
    - Some running processes require more frames for execution and steal frames from the other processes
    - The processes that were stolen frames generate frequent page faults -> low CPU utilization

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)

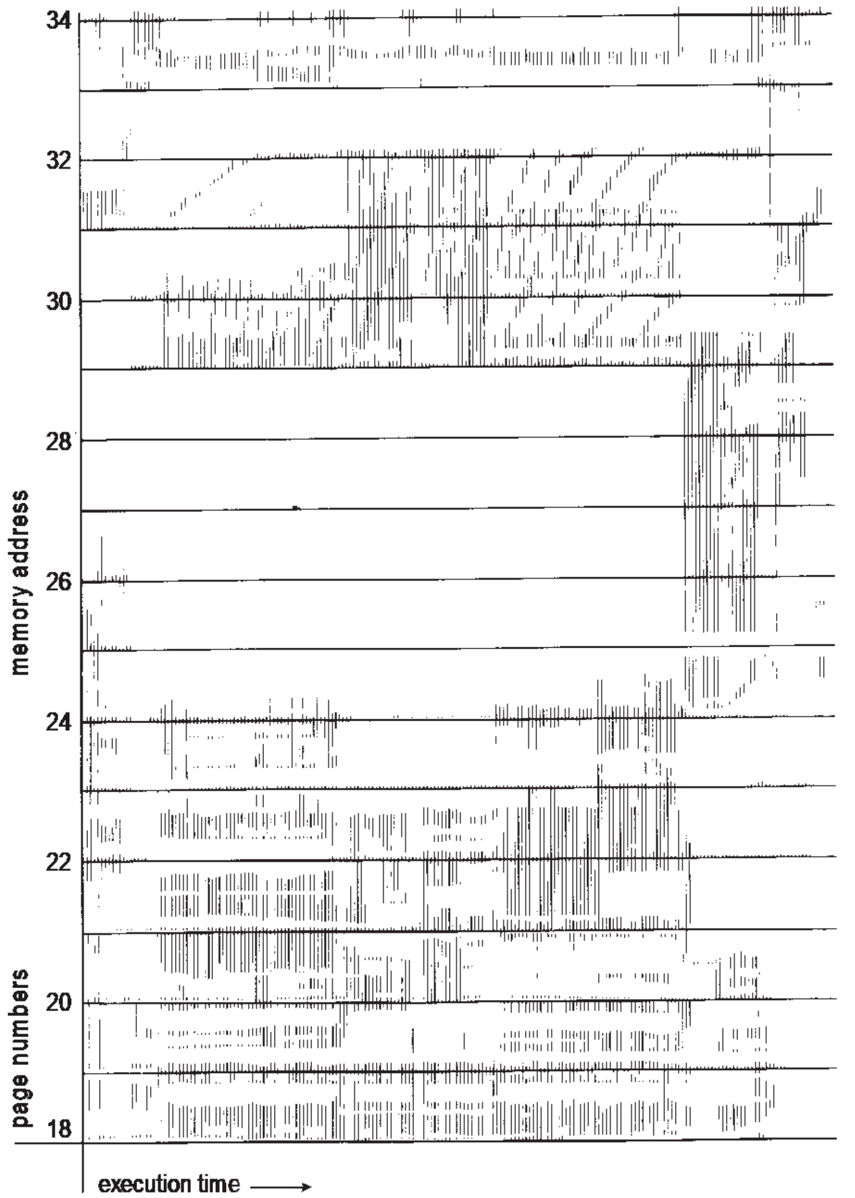


Demand Paging and Thrashing

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

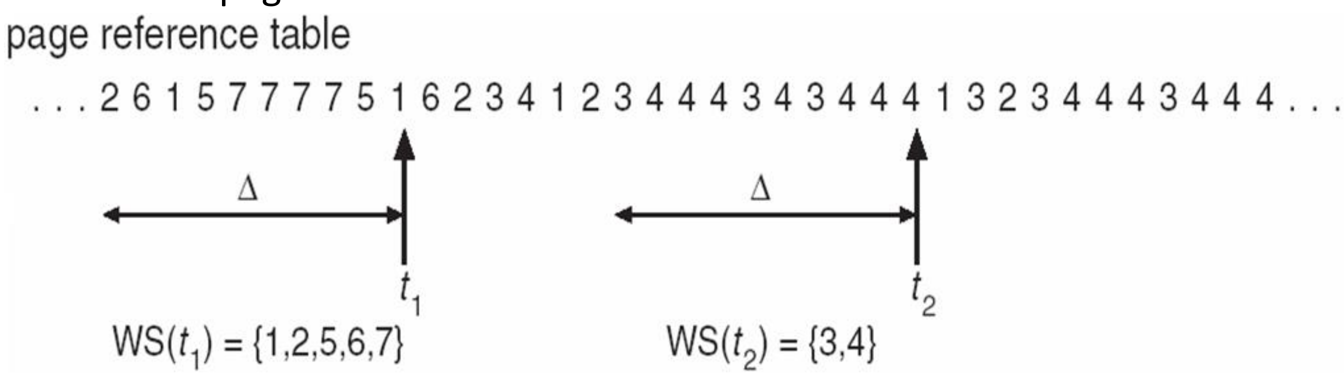
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit the effects of thrashing by using a local or priority replacement algorithm

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 (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
- Example: If $\Delta \equiv$ 10 page references



- $D = \Sigma\, WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing

m = total number of frame

- Policy if D > m, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with an interval timer + a reference bit
- Example: Assume Δ = 10,000 references, timer interrupts after every 5000 references (time units)
  - For each page, keep a reference bit and two in-memory bits
  - Whenever a timer interrupt occurs, copies (from reference to in-memory bits) and sets the values of all reference bits to 0
  - When a page fault occurs, if one of the three bits = 1 $\Rightarrow$ page in working set

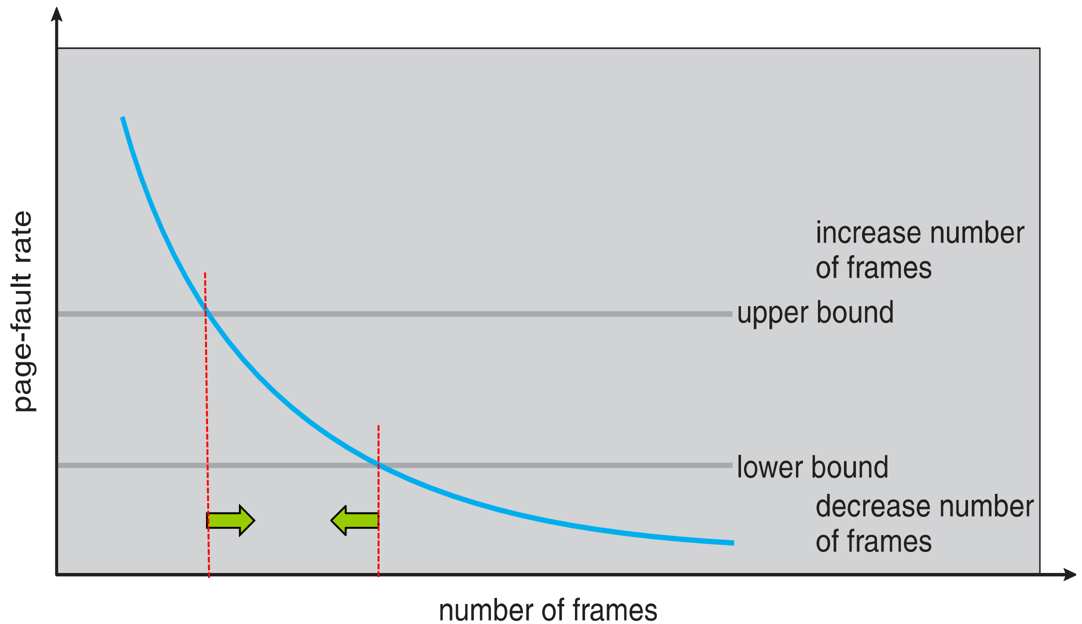
...	...	...
1 0 0 Page n	0 1 0 Page n	1 0 1 Page n
~5000 references Page n was referenced	~10000 references Page n was not referenced	~15000 references Page n was referenced

- Why is this not completely accurate?
  - Can’t tell which pages are older
- Improvement = 10 history bits and interrupt every 1000 references (time units)

1000 < 2^10 = 1024

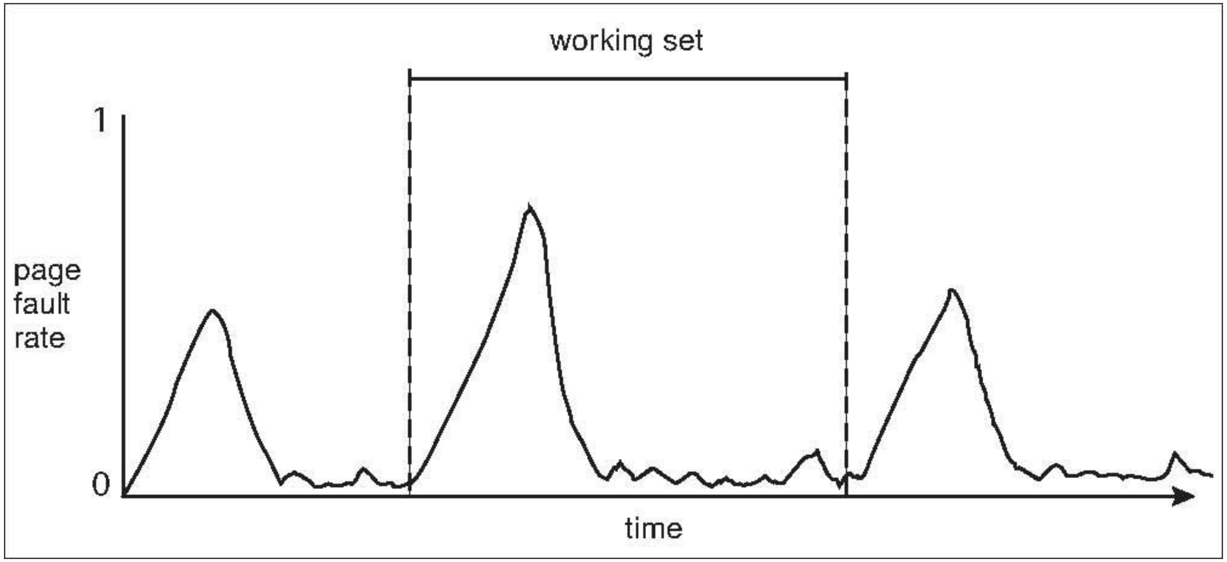
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate is too low, process loses frame
  - If actual rate is too high, process gains frame



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



Working Set and Page Replacement Exercise

- Reference string: **0 1 3 4 7   7 6 7 1 6   1 2 0 7 2**
- Example System
  - Pure demand paging
  - Dynamic frame allocation
    - Initial frame: 3 frames
    - At the start of every fifth memory access, frame is re-allocated by its working set – the previous five pages referenced
  - Frame deallocation priority
    - 1) not filled
    - 2) not included in a working set
    - 3) least recently accessed
- Optimal case

Working Set and Page Replacement Exercise

- Reference string: **0 1 2 1 3 3 2 7 6 3 7 8 4 8 2**
- Optimal Page Replacement
  - 8 Page Faults
  - Dynamically allocates 4 frames at 5th and 10th memory accesses

Page	**0**	**1**	**2**	**1**	**3**	**3**	**2**	**7**	**6**	**3**	**7**	**8**	**4**	**8**	**2**
Fault	F	F	F		F			F	F			F	F		
Frame 1	0	0	0	0	3	3	3	3	3	3	3	8	8	8	8
Frame 2		1	1	1	1	1	1	1	6	6	6	6	4	4	4
Frame 3			2	2	2	2	2	2	2	2	2	2	2	2	2
Frame 4								7	7	7	7	7	7	7	7

WS = {0, 1, 2, 3}

WS = {2, 3, 6, 7}

- Exercise: FIFO/LRU/Second Chance replacement?

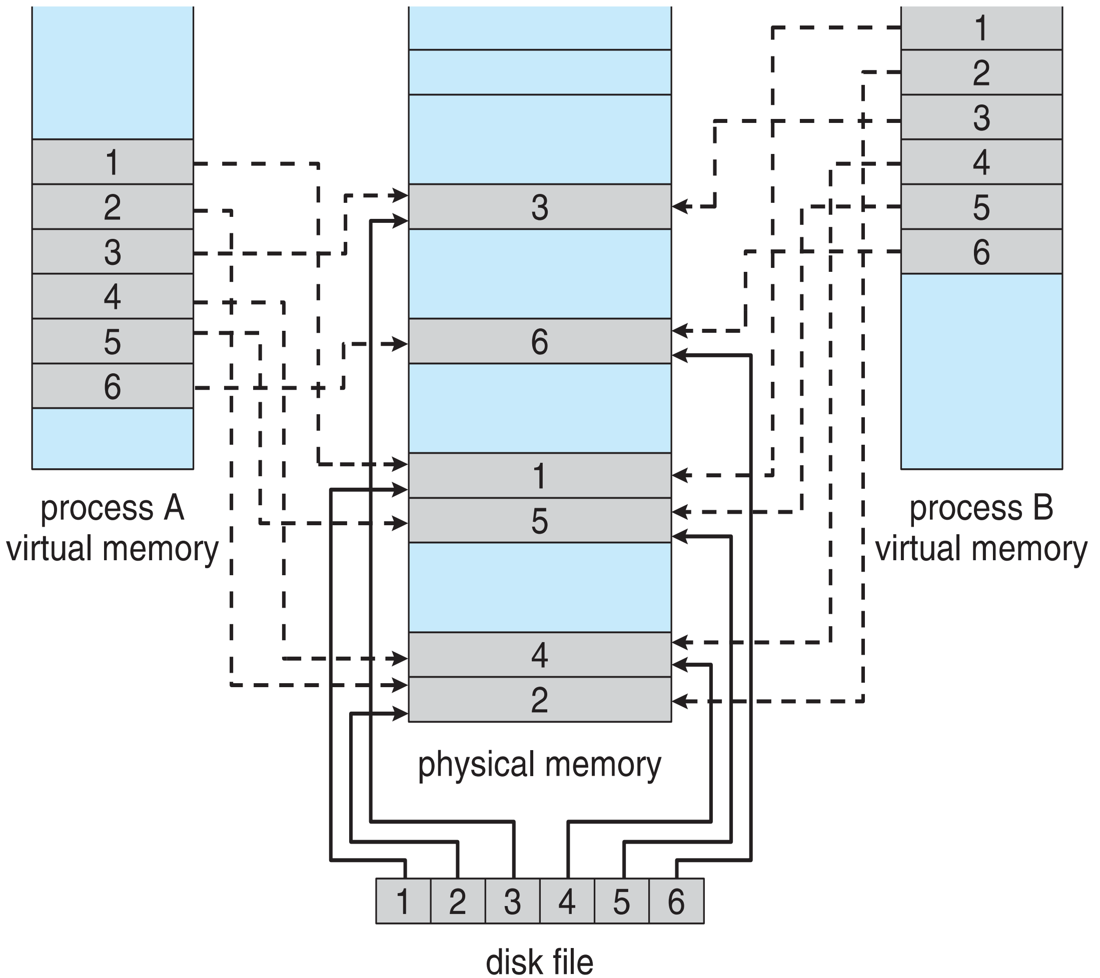
Memory-Mapped Files

- 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 from/to 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 the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

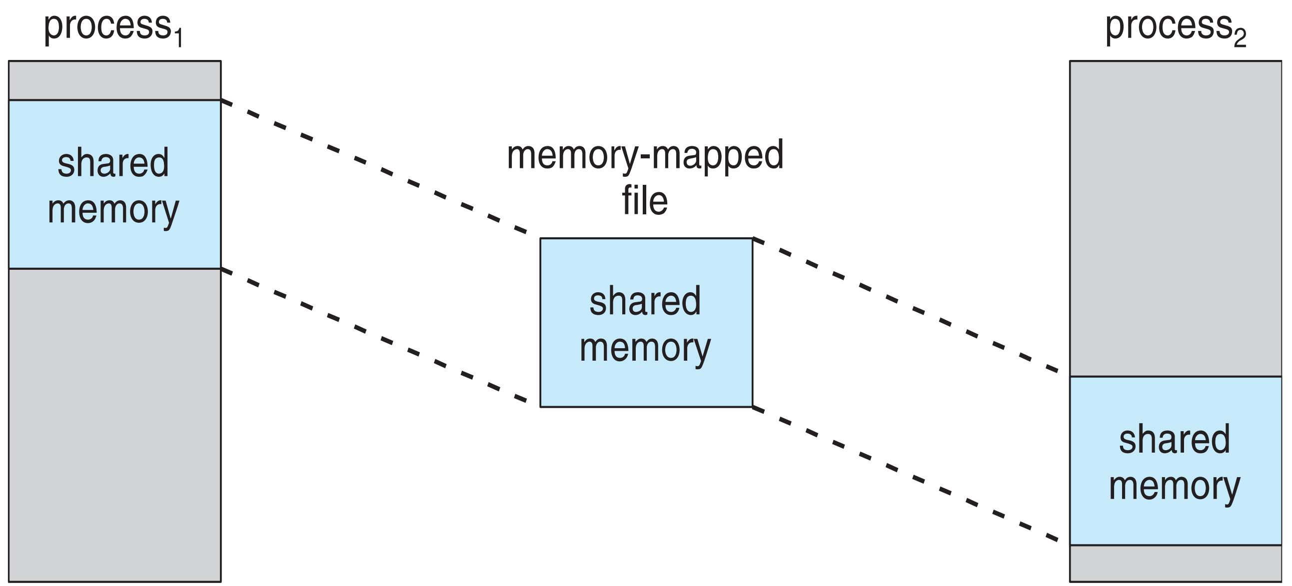
Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- In Solaris, for standard I/O (`open()`, `read()`, `write()`, `close()`), memory-maps files into kernel address space anyway
  - Process still does read() and write()
    - Copies data to and from kernel space and user space
  - Uses efficient memory subsystem instead of file subsystem
- Mmap system calls can also support copy-on-write (COW) functionality
- Memory mapped files can be used for shared memory between the communicating processes
  - Mapping the same file into their virtual address spaces

Memory Mapped Files



Shared Memory via Memory-Mapped I/O



Shared Memory in Windows API

- First create a **file mapping** for file to be mapped
  - Then establish a view of the mapped file in process's virtual address space
- Consider producer / consumer
  - Producer create shared-memory object using memory mapping features
    - Open file via `CreateFile()`, returning a `HANDLE`
    - Create mapping via `CreateFileMapping()` creating a **named shared-memory object**
    - Create view via `MapViewOfFile()`
  - Consumer establishes a view of the named shared-memory object
    - create a mapping to the existing named shared-memory object `OpenFileMapping()`
    - Create view via `MapViewOfFile()`
  - Sample code in Textbook

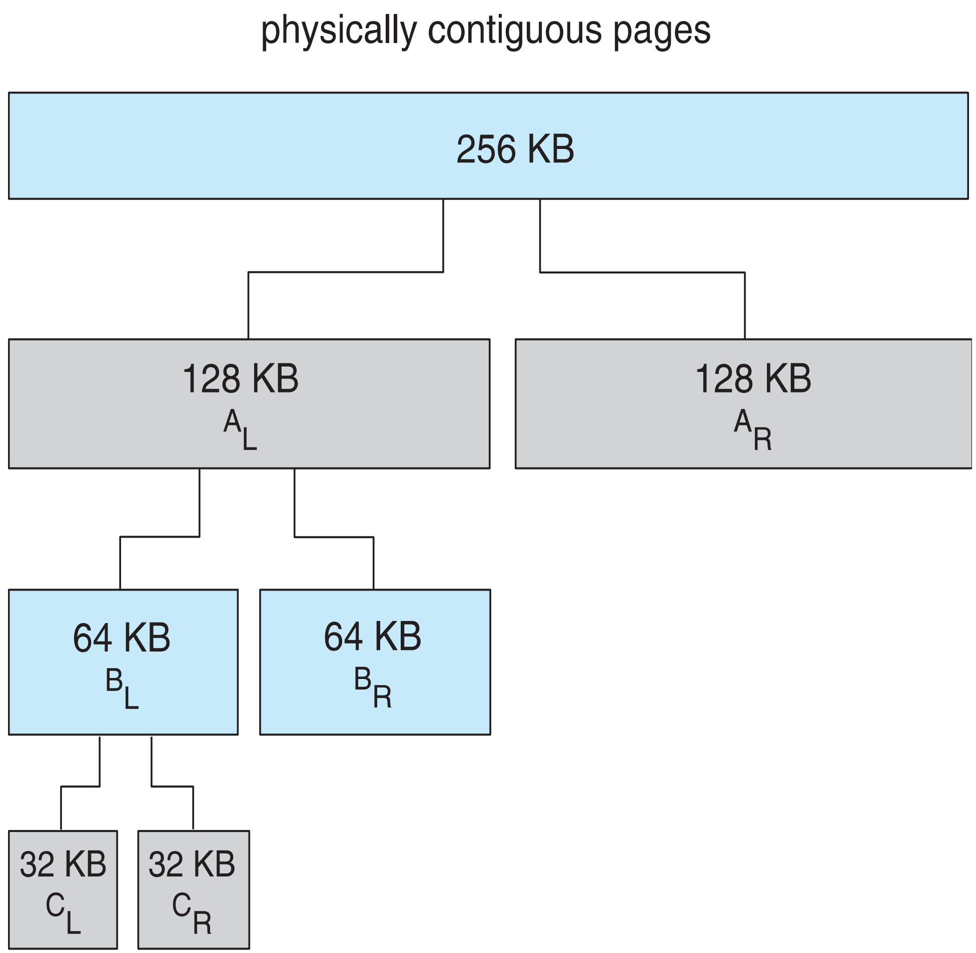
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
    - minimizing waste due to fragmentation
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O that directly interact with physical memory

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
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into $A_{L \text{ and }}$ A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

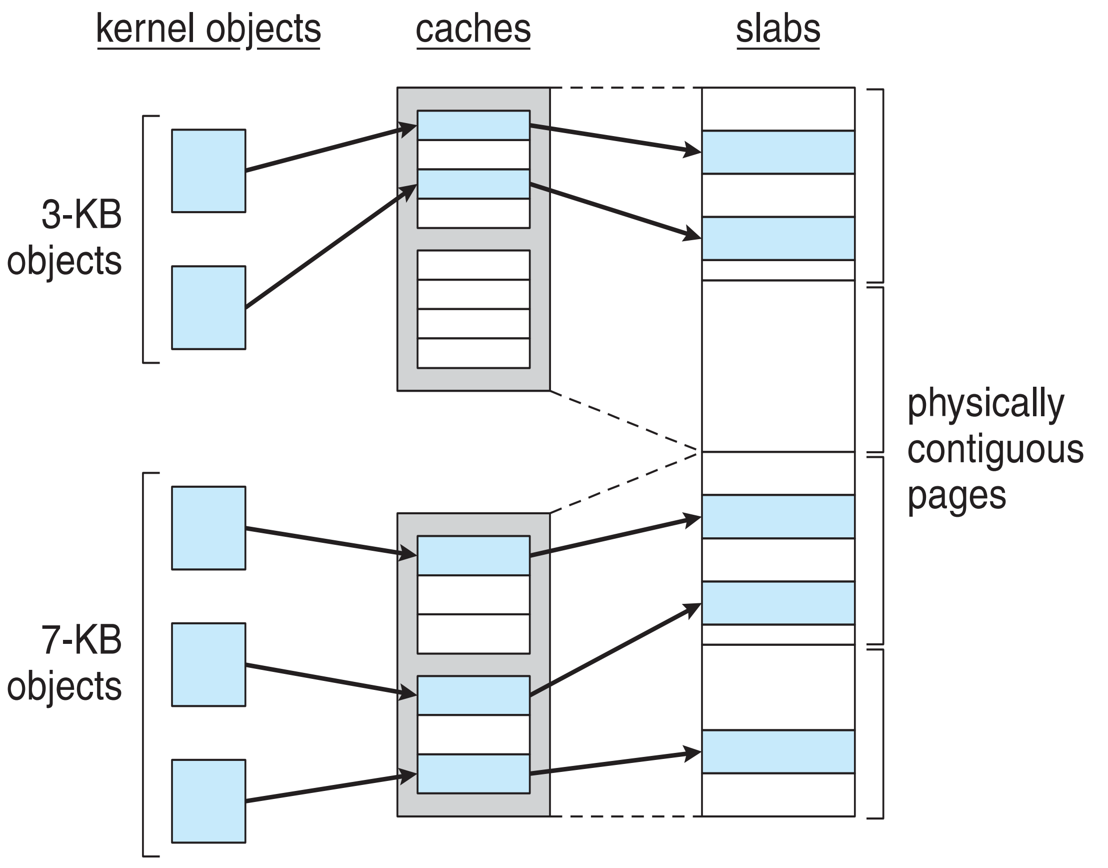
Buddy System Allocator



Slab Allocator

- Alternate strategy
- **Slab** is made up of one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure (i.e., all objects for the same type (size) are allocated from one cache)
    - Each cache filled with **objects** – instances of the data structure
- When cache is created, filled it with objects marked as **`free`**
- When structures are stored, objects marked as **`used`**
- If a slab is full of used objects, next object allocated from an empty slab
    - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB (performance-optimized SLAB) removes per-CPU queues, and move metadata stored with each slab to page structure

Other Considerations -- 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 the cost of $s * \alpha$ that saves page faults better or less than the cost of prepaging $s * (1- \alpha)$ for unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- Sometimes OS designers have a choice concerning the page size
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation ↑
  - Page table size ↓
  - **Resolution** (a small page size ⇒ high-resolution, a large page size ⇒ coarse resolution)
  - I/O overhead ↑
  - Number of page faults
  - Locality Spatial ↓ temporary ↑
  - 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

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 the process will spend a considerable amount of time resolving memory references in the page table rather than the TLB
- 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
  - This requires the operating system—not hardware— to manage the TLB (one of the fields in a TLB entry must indicate the size of the page frame)

Other Issues – Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page if the page size is 128 words
  - 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

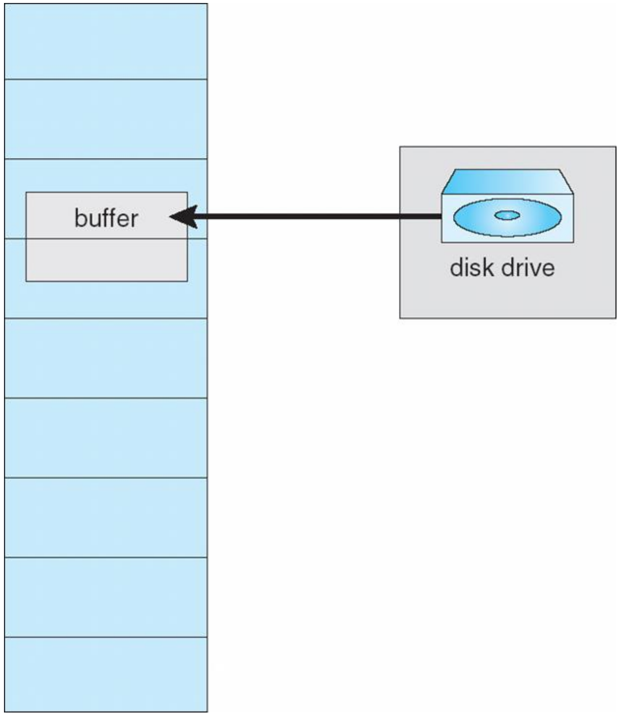
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

- Careful selection of data structures and programming structures required

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
- A **lock bit** is associated with every frame, If the frame is locked, it cannot be selected for replacement
- User processes such as database processes may also need to lock pages into memory
- **Pinning** of pages to lock into memory is fairly common, and most operating systems have a system call allowing an application to request that a region of its logical address space be pinned



Operating System Examples

- Windows
- Solaris

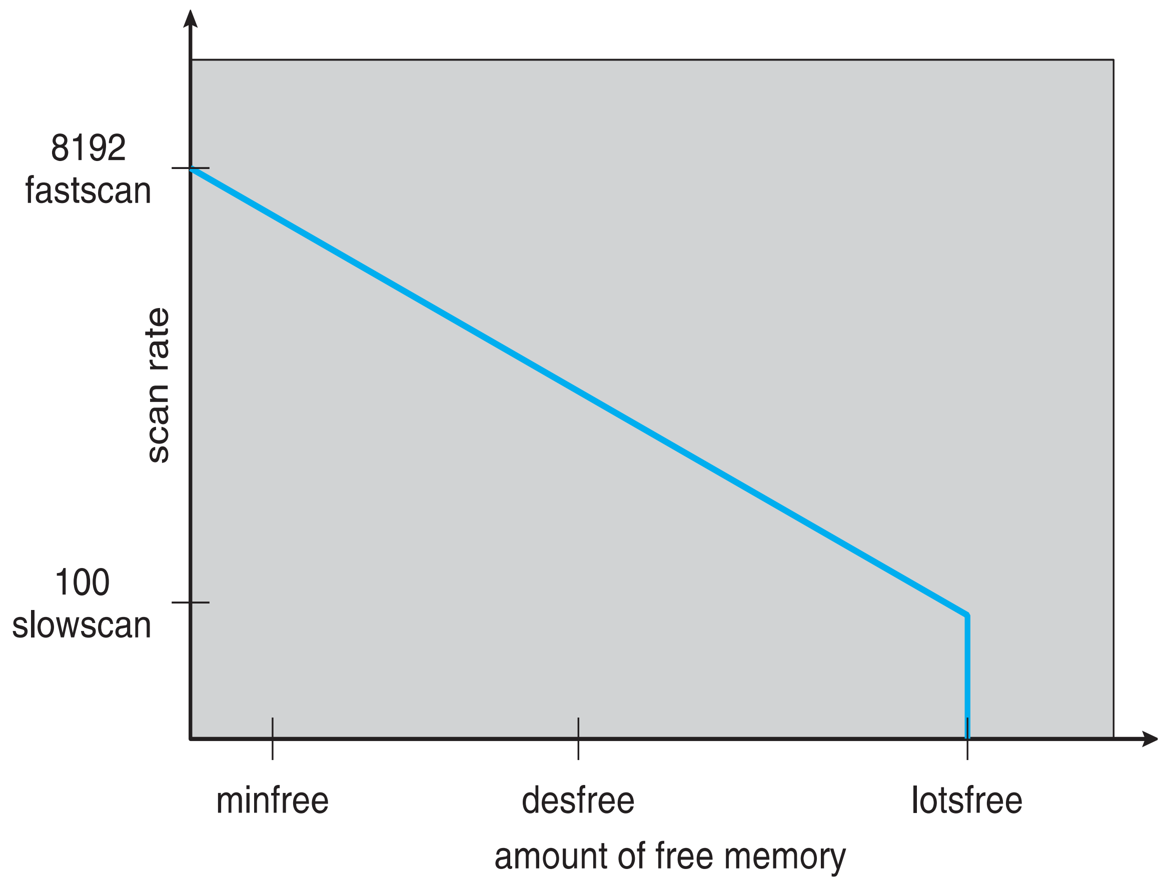
Windows

- 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
- Enhancements of the paging algorithm in a recent Solaris kernel
  - Pages belonging to libraries shared by several processes are skipped during the page-scanning process (no eviction)
  - Priority paging gives priority to process code pages over pages allocated to regular files

Solaris 2 Page Scanner



- When free memory falls below lotsfree, scanning occurs at slowscan (100) pages per second and progresses to fastscan (8192), depending on the amount of free memory available. Fastscan is typically set to the value (total physical pages)/2 pages per second.

End of Chapter 9