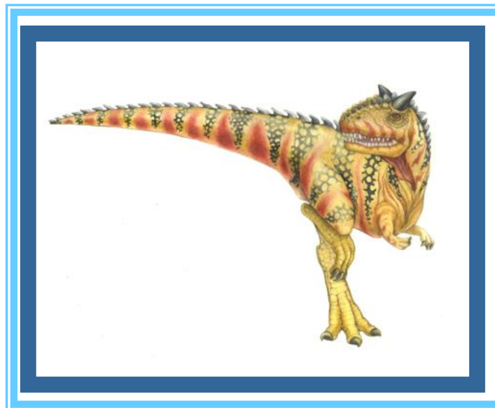
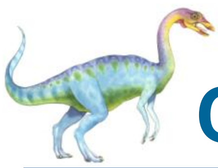
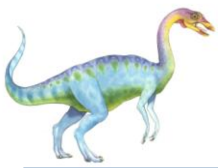
Chapter 9: Virtual-Memory Management





Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory



Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames

Background

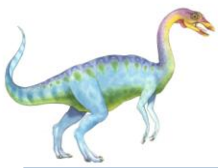
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

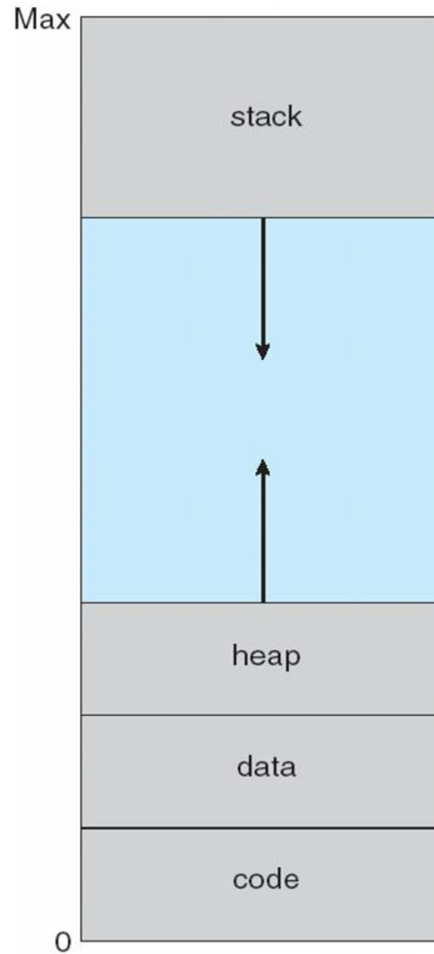
page 0
page 1
page 2
page v
virtual memory

memory map

physical memory



Virtual-address Space



Shared Library Using Virtual Memory

stack

shared library

heap

data

code

shared pages

stack

shared library

heap

data

code

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space

program A

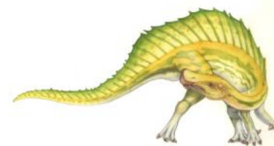
program B

main memory

swap out

swap in

0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23



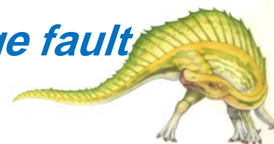
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

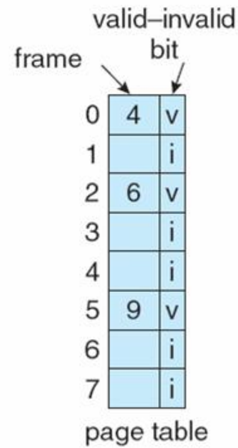
Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

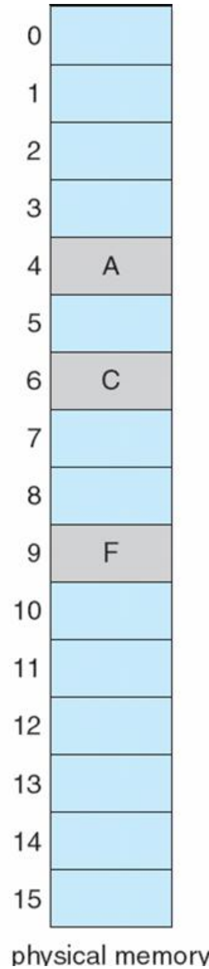
page table

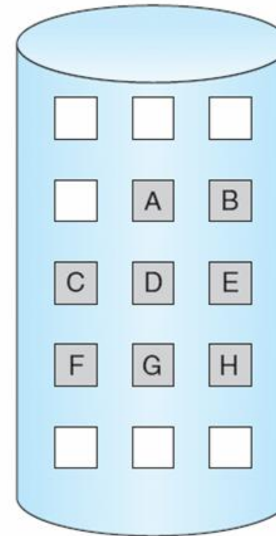
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ ***page fault***

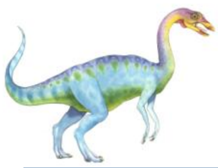


Page Table When Some Pages Are Not in Main Memory





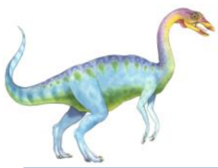




Page Fault

- Access to a page marked invalid causes interrupt to the operating system called a **page fault**

1. Operating system looks at internal table (in PCB) to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault



Steps in Handling a Page Fault

3 page is on backing store

operating system

2 trap

reference

1

load M

i

6 restart instruction

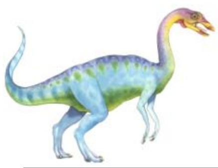
page table

free frame

5 reset page table

4 bring in missing page

physical memory



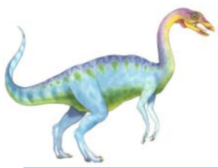
Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$ { p is the probability of a page fault }
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT) with Page Fault

  EAT = $(1 - p)$ * memory access + p * page fault time

- Effective Access Time (EAT) with no Page Fault

  EAT = memory access



Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

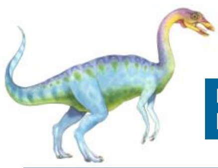
  - Memory-Mapped Files (later)

Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

  If either process writes to a shared page, a copy of the shared page is created

- COW allows more efficient process creation as only modified pages are copied



Before Process 1 Modifies Page C

process$_1$

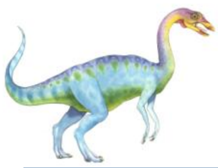
physical memory

process$_2$

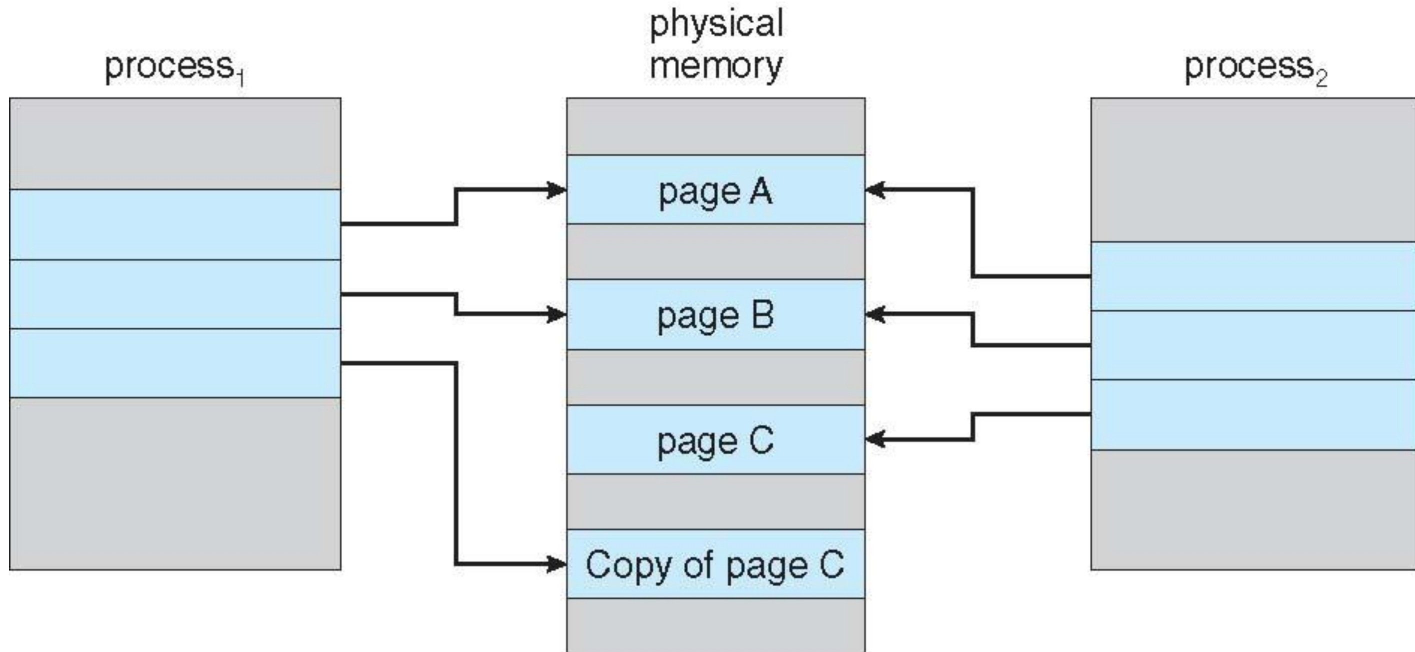
page A

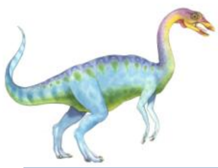
page B

page C



After Process 1 Modifies Page C



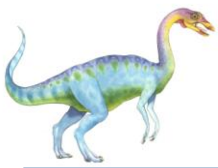


Page Replacement

What happens if there is no free frame?

The solution:

- ***Page replacement*** – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
  - Same page may be brought into memory several times



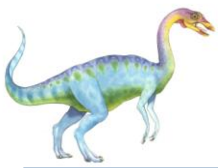
Basic Page Replacement

1. Find the location of the desired page on disk

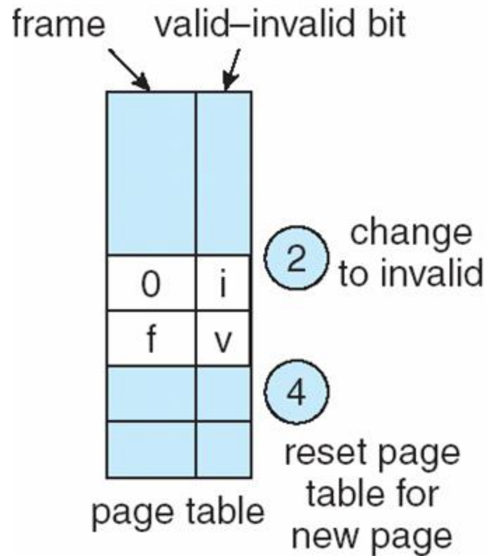
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

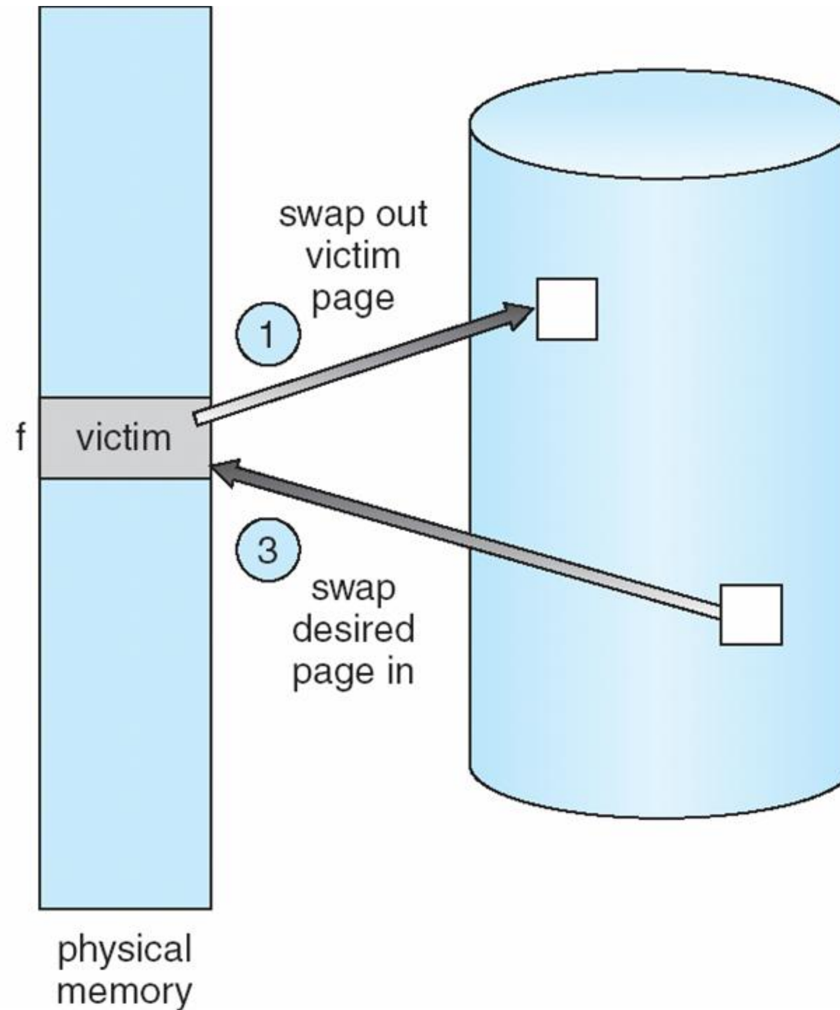
3. Bring the desired page into the (newly) free frame; update the page and frame tables

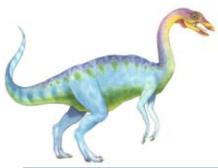
4. Restart the process



Page Replacement







Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (***reference string***) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

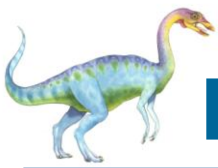
Graph of Page Faults Versus The Number of Frames

number of page faults

16
14
12
10
8
6
4
2

1 2 3 4 5 6

number of frames



First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

	Frame		
1	1	4	5
2	2	1	3
3	3	2	4

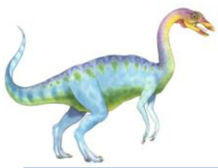
9 page faults

- 4 frames

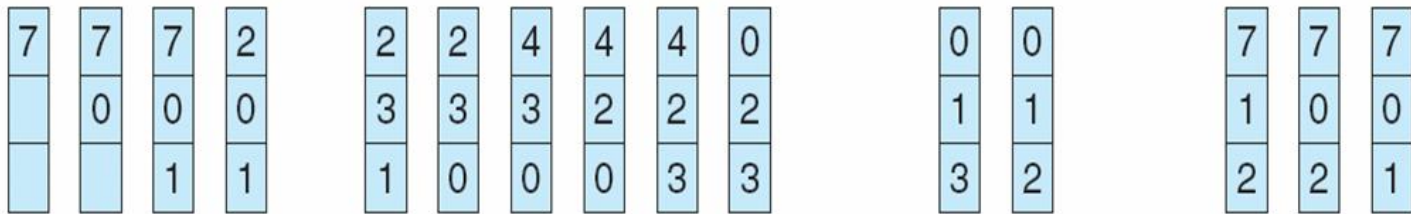
	Frame		
1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

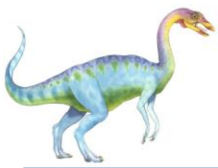
10 page faults

- Belady's Anomaly: more frames $\Rightarrow$ more page faults



FIFO Page Replacement

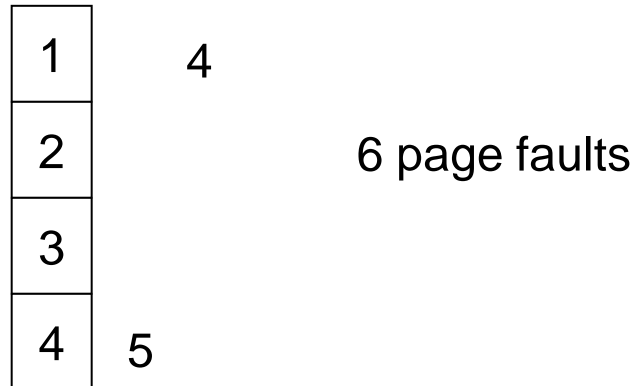




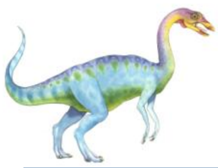
Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

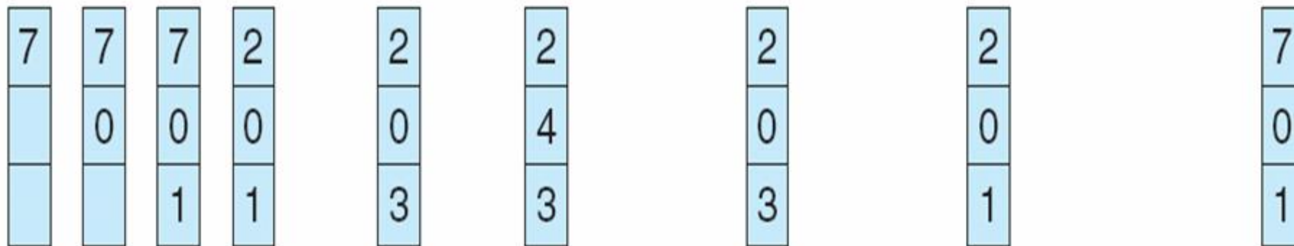
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- Used for measuring how well your algorithm performs



Optimal Page Replacement

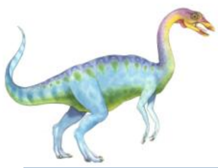


Least Recently Used (LRU) Algorithm

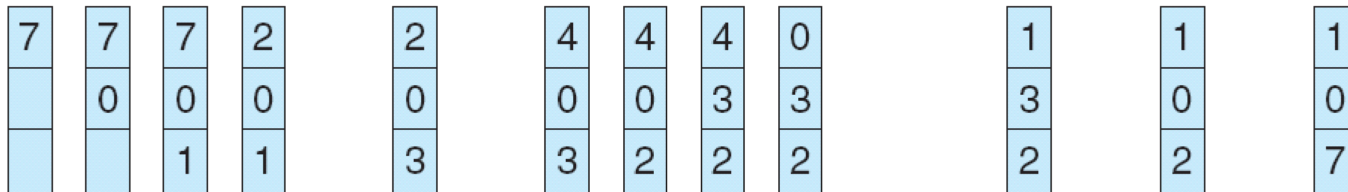
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

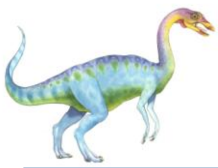
1	1	1	1	5
2	2	2	2	2
3	5	5	4	4
4	4	3	3	3

- Replace page that has not been used for the longest period of time
- 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 determine which are to change



LRU Page Replacement





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

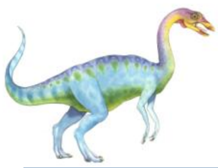
Fixed Allocation

- 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 $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

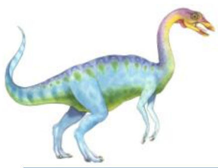
Priority Allocation

- 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



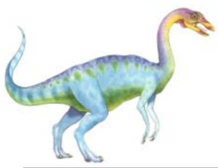
Global vs. Local Replacement

- **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

- **Thrashing** ≡ a process is busy swapping pages in and out
- If a process does not have "enough" frames, the page-fault rate is very high.
- **Cause of trashing**:
  - OS monitors CPU utilization , If low CPU utilization
  - operating system increase the degree of multiprogramming by adding another process to the system
  - *A global page replacement algorithm* is used.
  - A trashing is appear and processes waiting for the paging device, the CPU utilization decreases.
  - To limit the effects of trashing by using *a local replacement algorithm*.

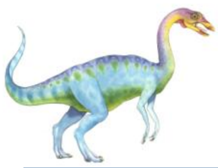


Thrashing (Cont.)

thrashing

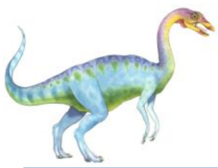
CPU utilization

degree of multiprogramming



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

1
2
3
4
5
6
process A
virtual memory

3
6
1
5
4
2
physical memory

1
2
3
4
5
6
process B
virtual memory

1 2 3 4 5 6
disk file

Memory-Mapped Shared Memory in Windows

process$_1$

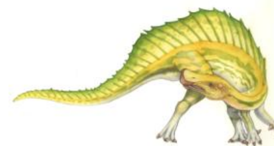
shared memory

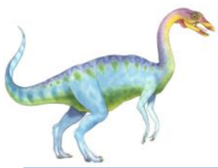
memory-mapped file

shared memory

process$_2$

shared memory





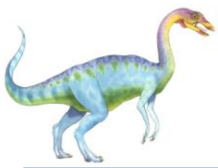
Allocating Kernel Memory

- 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

- There is two strategies for managing free memory assigned to kernel processes:
  - 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

physically contiguous pages

256 KB

128 KB A_L

128 KB A_R

64 KB B_L

64 KB B_R

32 KB C_L

32 KB C_R

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
- The *slab-allocation algorithm* uses caches to store kernel objects:
  - When cache created, filled with objects marked as **free**
  - When a new kernel structure is needed, the allocator assign free object from cache.
  - 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 from contiguous physical pages
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation

kernel objects

caches

slabs

3 KB objects

7 KB objects

physical contiguous pages

End of Chapter 9