CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania, Fall 2002

Lecture Note: Memory Management

Memory Management

- The memory management portion of the Operating System is responsible for the efficient usage of main memory, especially in a multiprogramming environment where processes contend for memory.
- It must also offer protection of one process address space from another (including protection of system address space from user processes).
- The memory subsystem should also provide programmers with a convenient logical or virtual address space, in which the low-level details of memory management are hidden.

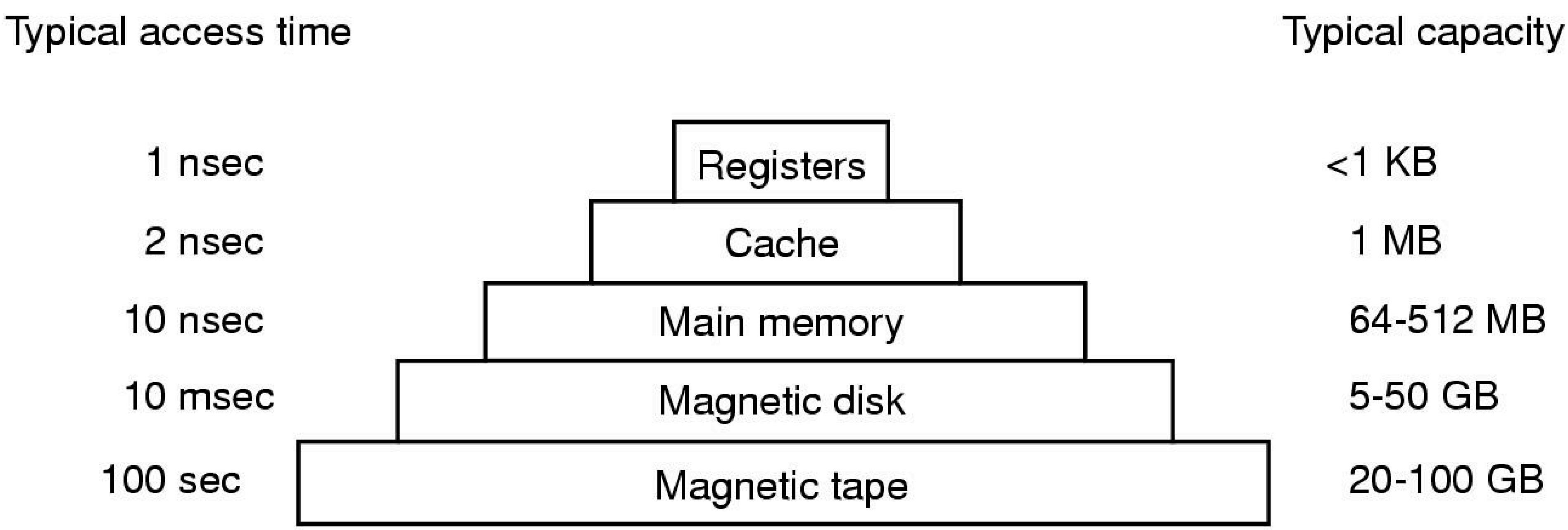
Sharing of Memory

Issues

- Allocation schemes
- Protection from each other
- Protecting OS code
- Translating logical addresses to physical
- Swapping programs
- What if physical memory is small: Virtual memory

Memory Hierarchy

- small amount of fast, expensive memory – cache
- some medium-speed, medium price main memory
- gigabytes of slow, cheap disk storage



Memory Management Strategies

1. Fetch Strategy:
   Determine when to load and how much to load at a time.
   E.g., demand fetching, anticipated fetching (pre-fetching).

2. Placement (or allocation) Strategy:
   Determine where information is to be placed.
   E.g., Best-Fit, First-Fit, Buddy-System.

3. Replacement Strategy:
   Determine which memory area is to be removed under contention conditions.
   E.g., LRU, FIFO.

Memory Management Evolution

- Variations
  1. Fixed Partitions
  2. Variable Partitions
  3. Segmentation
  4. Paging

Early computers
Relevant again: PDAs, smartcards

Modern PCs

- Criteria
  1. How efficiently can it be implemented?
  2. How effectively can the physical memory be utilized?

Fixed Partitions

1 Divide all physical memory into a fixed set of contiguous partitions. E.g., early IBM 360 models.

```
+---------+
|  12K    |   Queue for waiting processes
+---------+
|  2K     |     ....
+---------+
|  6K     |     ....
+---------+
| OS: 2K  |
+---------+
```

2 Place only one process at a time in any partition.

3 Bind physical to virtual address during loading, not during execution.

4 Partition boundaries limit the available memory for each process.

5 A process is either entirely in main memory or entirely on backing store (i.e., swapped in or swapped out).

6 A process may only be swapped into the same partition from which it was swapped out (why?)

7 It can only simulate smaller, not larger, virtual space than physical space.

8 No sharing between processes.

9 Should there be a single queue per partition or one global queue?

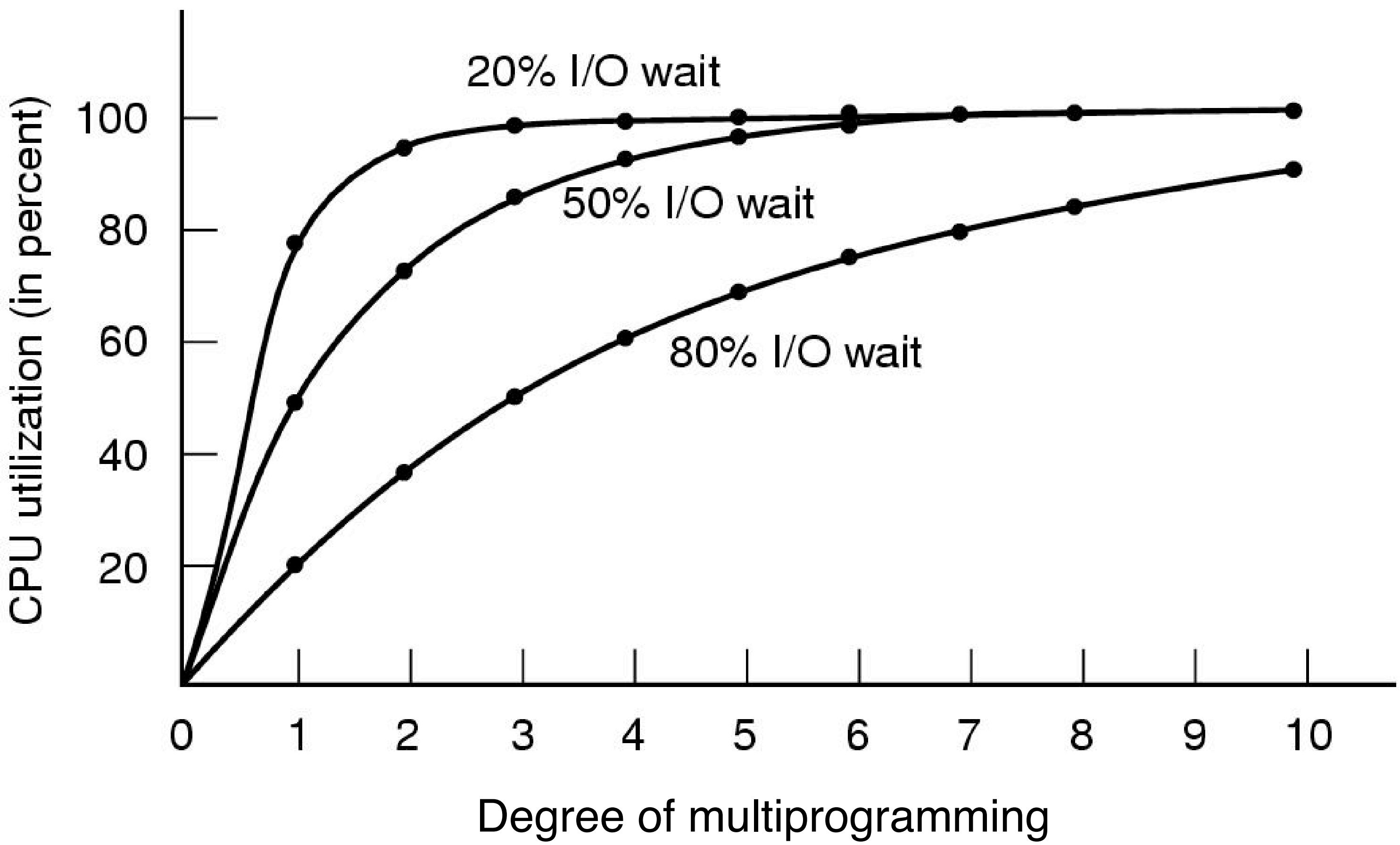
10 Memory space wasted:

- Internal fragmentation: memory which is internal to a partition, but not used.
- External fragmentation: a partition is unused and available, but too small for any waiting job.

Effect of Multiprogramming

- Recall: A central goal of multiprogramming is to keep CPU busy while one process waits for an I/O
- Number of processes constrained by memory size
- Tradeoff between memory size and CPU utilization
- Can we have estimate of desired number of processes? If each process spends 75% time waiting, how many processes would keep CPU busy all the time?
- If each process spends .75 fraction waiting, then assuming independence, probability that N processes will all wait at the same time is $.75^N$ (this equals .05 for N = 10). So effective CPU utilization is $1 - .75^N$
- If waiting fraction is p then CPU utilization is $1 – p^N$
- This is only a crude estimate, but a useful guide

CPU Utilization Curve



Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute, and some scheme for mapping compile-time (logical) addresses to run-time (physical) addresses needed
  - must keep a program out of other processes' partitions (protection)

- Simplest scheme: Loader performs relocation (feasible only for fixed partitions)

- Use base and limit registers in the hardware
  - Logical addresses added to base value to map to physical addr
  - Logical addresses larger than limit value is an error
  - Frequently used, so special hardware required

Swapping

- Swapper decides which processes should be in main memory
- How to allocate memory?
- For now, assume the entire memory needed by a process is allocated in a single block
- Suppose, 180K free memory, and A needs 30K

Swapping

- B requests 50K
- C requests 20K
- D requests 20K
- A exits
- C exits
- Memory is fragmented
- Should OS compact it to make free memory contiguous?

More on swapping

- There should be some free space for dynamic allocation of memory (heaps) within the space allocated to a process
  - In modern systems, stack grows downwards and heap grows upwards, with fixed space for compiled code
- With variable partitions, OS must keep track of memory that is free
  - Bitmaps (arrays)
  - Linked lists
- Classical tradeoffs: space required vs time for (de)allocation

Managing Free Space

- Bit-map
  - Suppose memory is divided in chunks of 10K
  - Maintain a vector of 0/1's that specifies availability
  - i-th bit tells whether i-th chunk is free
  - For the current example: 20 bits

    00000011 00111110 0011

Managing Free Space: Linked Lists

- Each record has
  - Process ID/ Free (H: hole)
  - Start location
  - Size
  - Pointer to Next record
- Current state

(H,2,3),(B,5,5),(H,10,2),(D,12,2),(H,14,6)

How should we update the list when B leaves?

Managing Free Space: Linked Lists

- Current state

(H,2,3),(B,5,5),(H,10,2),(D,12,2),(H,14,6)

- PCB for a process can have a pointer into the corresponding record
- When a process terminates, neighboring blocks need to be examined
  - Doubly-linked lists

Allocation Strategy

- Suppose a new process requests 15K, which hole should it use?
- First-fit: 30K hole
- Best-fit: 20K hole
- Worst-fit: 60K hole

Allocation strategies

- Let $\{H_i \mid i = 1,\ldots,n\}$ be unused blocks and k be the size of a requested block.
- **First-Fit**
  - Select the first H_i such that size $(H_i) \geq k$.
  - That is, select the first block that is big enough
- **Best-Fit**
  - Select H_i such that size $(H_i) \geq k$ and, if size $(H_j) \geq k$ then size $(H_j) \geq$ size (H_i) for $i \neq j$.
  - That is, select the smallest block that is big enough.
- **Worst-Fit**
  - Select H_i such that size $(H_i) \geq k$, and if size$(H_j) \geq k$ then size$(H_j) \geq$ size(H_i) for $i \neq j$. (idea: to produce the largest left-over block.)
- **Buddy System**

Best-fit vs. First-fit

- Both could leave many small and useless holes.
- To shorten search time for First-Fit, start the next search at the next hole following the previously selected hole.
- Best-Fit performs better: Assume holes of 20K and 15K, requests for 12K followed by 16K can be satisfied only by best-fit
- First-Fit performs better: Assume holes of 20K and 15K, requests for 12K, followed by 14K, and 7K, can be satisfied only by first-fit
- In practice,
  F-F is usually better than B-F, and
  F-F and B-F are better than W-F.

Buddy Systems

- Allocation algorithm that forms basis of Linux memory management
- Suppose we have 128 units (128 pages or 128K)
- Each request is rounded up to powers of 2
- Initially a single hole of size 128
- Suppose, A needs 6 units, request rounded up to 8
- Smallest hole available: 128. Successively halved till hole of size 8 is created
- At this point, holes of sizes 8, 16, 32, 64
- Next request by B for 5 units: hole of size 8 allocated
- Next request by C for 24 units: hole of size 32 allocated

Buddy Systems