CS 355
Operating Systems

Memory Management

Memory Management
• Responsible for the efficient usage of main memory, especially in a multiprogramming environment where processes contend for memory.
• 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.

Memory Management
• Ideally programmers want memory that is
  – large
  – fast
  – non volatile
• Memory hierarchy
  – small amount of fast, expensive memory – cache
  – some medium-speed, medium price main memory
  – gigabytes of slow, cheap disk storage
• Memory manager handles the memory hierarchy

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

Two Basic Memory Management Questions
• Is there multi-programming?

• What memory management features does the hardware support?
  – none
  – base registers
  – swapping
  – virtual memory/paging

Monoprogramming
• An OS that only runs one process at a time, simple?

• Where doe the OS go?

• Usual solution: OS starts at one end and user programs start at the other
Basic Memory Management
Monoprogramming without Swapping or Paging

Three simple ways of organizing memory - an operating system with one user process

Multiple Programs Without Memory Abstraction

Illustration of the relocation problem.

Base and Limit Registers

Share 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

Fixed Partitions

• 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 |

• Place only one process at a time in any partition.
• Processes assigned partition at load time.
• Incoming processes are queued until an adequate partition is available

Fixed Partitions

• Partition boundaries limit the available memory for each process.
• A process is either entirely in main memory or entirely on backing store (i.e., swapped in or swapped out).
• Should there be a queue per partition or one global queue?
Fixed Partitions
• 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
• A central goal of multiprogramming is to keep CPU busy while a process waits for I/O
• Number of processes constrained by memory size
• Tradeoff between memory size and CPU utilization
• Can we estimate the desired number of processes?
  – If each process spends 75% time waiting, how many processes would keep CPU busy all the time?
  – If waiting fraction is $p$ then CPU utilization is $1 - p^n$

Relocation and Protection
• Cannot be sure where program will be loaded in memory
  – address locations cannot be absolute
  – 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)
  – fixed partition requires that programs must be swapped into the same partition they are swapped out of, why?
• 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: Fixed Partitions
Memory allocation changes as:
  – processes come into memory
  – leave memory
Shaded regions are unused memory
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

Suppose, 180K free memory, and A needs 30K
OS: 20K
Free: 150K
A: 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?

Free: 60K
D: 20K
Free: 20K
B: 50K
Free: 30K
OS: 20K

Swapping: Variable Partitions

- Allocating space for growing data segment
- Allocating space for growing stack & data segment

More on swapping

- Need free space for dynamic allocation of memory (heaps) within the space allocated to a process
  - 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 00000001100111100011

Memory Management with Bit Maps

- Part of memory with 5 processes, 3 holes
  - Tick marks show allocation units
  - Shaded regions are free
- Corresponding bit map
- Same information as a list
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?

Memory Management with Linked Lists

Before X terminates

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After X terminates

Four neighbor combos for the terminating process X

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 — called Next-Fit.
- Assume holes of 20K and 15K, requests for 12K followed by 16K
- Assume holes of 20K and 15K, requests for 12K, followed by 14K, and 7K
- In practice, F-F is usually better than B-F, and F-F and B-F are better than W-F.

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

Buddy Systems

- Allocation algorithm that forms basis of Linux MM
- 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
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.

Buddy Systems

Memory Management Evolution

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

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

Modern PCs

Early computers
Relevant again: PDAs, smartcards