Memory Management

Chapter 7
Memory Management

- A task carried out by the OS and hardware to accommodate multiple processes in main memory
- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle
- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible
- The kernel typically occupies some fixed portion of main memory and the rest is shared by multiple processes
Memory Management Requirements

- Relocation

  - programmer cannot know where the program will be placed in memory when it is executed
  - a process may be (usually) relocated in main memory due to swapping
  - swapping enables the OS to have a larger pool of ready-to-execute processes
  - memory references in code (for both instructions and data) must be translated to actual physical memory address
Memory Management Requirements

- Protection
  - processes should not be able to reference memory locations in another process without permission
  - impossible to check addresses at compile time in programs since the program could be relocated
  - address references must be checked at run time by hardware
Memory Management Requirements

- Sharing
  - must allow several processes to access a common portion of main memory without compromising protection
  - better to allow each process to access the same copy of the program rather than have their own separate copy
  - cooperating processes may need to share access to the same data structure
Memory Management Requirements

- **Logical Organization**
  - Users write programs in modules with different characteristics
    - Instruction modules are execute-only
    - Data modules are either read-only or read/write
    - Some modules are private, others are public
  - To effectively deal with user programs, the **OS and hardware** should support a basic form of module to provide the required protection and sharing.
Memory Management

Requirements

- Physical Organization
  - Secondary memory is the long term store for programs and data while main memory holds program and data currently in use.
  - Moving information between these two levels of memory is a major concern of memory management (OS).
    - It is highly inefficient to leave this responsibility to the application programmer.
In this chapter we study the simpler case where there is no virtual memory.

An executing process must be loaded entirely in main memory (assuming overlay is not allowed).

Although the following simple memory management techniques are not used in modern OS, they lay the ground for a proper discussion of virtual memory (next chapter):

- fixed partitioning
- dynamic partitioning
- simple paging
- simple segmentation
Fixed Partitioning

- Partition main memory into a set of non-overlapping regions called partitions

- Partitions can be of equal or unequal sizes
Fixed Partitioning

- any process whose size is less than or equal to a partition size can be loaded into the partition.
- if all partitions are occupied, the operating system can swap a process out of a partition.
- a program may be too large to fit in a partition. The programmer must then design the program with overlays (to replace what is already stored).
  - when the module needed is not present the user program must load that module into the program’s partition, overlaying whatever program or data are there (The overlay replaces what is already stored).
  - we don’t need to assume overlay in this class. If a process cannot fit into any partition, it cannot be executed.
Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This causes internal fragmentation.

Fixed-size partitions were used in early OSs.

Unequal-size partitions relieve this problem but they still remain...
Placement Algorithm with Partitions

- **Equal-size** partitions
  - If there is an available partition, a process can be loaded into that partition
    - because all partitions are of equal size, it does not matter which partition is used
    - practically, it can first allocate the available slot at a lower address (In real world, placement cannot be random)
  - If all partitions are occupied by blocked processes, choose one process to swap out to make room for the new process
Placement Algorithm with Partitions

- **Unequal-size** partitions: use of multiple queues
  - Assign each process to the smallest partition within which it will fit
  - A queue for each partition size
  - Tries to minimize internal fragmentation
  - Problem: some queues will be empty if no processes within a size range is present (thus reducing multi-programming)
Placement Algorithm with Partitions

- **Unequal-size** partitions: use of a single queue
  - When it's time to load a process into main memory, the smallest available partition that will hold the process is selected
  - Increases the level of multiprogramming at the expense of internal fragmentation
Dynamic Partitioning

- Partitions are of various length (thus the number of partitions also varies)
- Each process is allocated exactly as much memory as it requires

Whether can this dynamic partition cause fragmentation?
Dynamic Partitioning

- Partitions are of various length (thus the number of partitions also varies)
- Each process is allocated exactly as much memory as it requires
- Eventually holes are formed in main memory. This is called **external fragmentation**
A hole of 4M is left after loading 3 processes: not enough room for another process.

Eventually each process is blocked. The OS swaps out process 2 (for example) to bring in process 4 (8 MB).
Dynamic Partitioning: an example

- Another hole of 6M is created
- Eventually each process is blocked. The OS swaps out process 1 (for example) to bring in again process 2 and another hole of 6M is created...
- Then how to address these holes?
Compaction

- Compaction:
  - Dynamic partitions that are spread out in memory are "compacted" by shifting them to fill any intervening holes
  - Can help address external fragmentation

- Compaction requires support for relocation
  - Relocation turns out to be critical in multiprogramming machines

- Overhead sucks
  - Compaction is pure overhead and relatively time-consuming
Placement Algorithm

- Used to decide which free block to allocate to a process
- Goal: to reduce usage of compaction (time consuming)
- Possible algorithms:
  - **Best-fit**: choose smallest hole
  - **First-fit**: choose first hole from beginning
  - **Next-fit**: choose first hole from last placement

Example of memory configuration before and after allocation of **16 M Process**
Placement Algorithm:

- Next-fit often leads to allocation of the largest block at the end of memory.
- First-fit favors allocation near the beginning: tends to create less fragmentation than Next-fit.
- Best-fit searches for smallest block: the fragment left behind is small as possible.
  - Main memory quickly forms holes too small to hold any process: compaction generally needs to be done more often.
Internal vs External Fragmentation

- **Internal fragmentation:**
  - when some portion *inside* a fixed-sized partition is not fully used

- **External fragmentation:**
  - when portions between dynamic-sized partitions is too small to be used
Buddy System

- A reasonable compromise to overcome disadvantages of both fixed and variable partitioning schemes

- Memory blocks are available in size of $2^K$ where $L \leq K \leq U$ and where
  - $2^L$ = smallest size of allocable block
  - $2^U$ = largest size of allocable block (generally, the entire memory available)
Buddy System

- We start with the entire block of size $2^U$
- When a request of size $S$ is made:
  - If $2^{U-1} < S \leq 2^U$ then allocate the entire block of size $2^U$
  - Else, split this block into two buddies, each of size $2^{U-1}$
  - If $2^{U-2} < S \leq 2^{U-1}$ then allocate one of the 2 buddies
  - Otherwise one of the 2 buddies is split again
- This process is repeated until the smallest block greater or equal to $S$ is generated
- Two buddies are coalesced whenever both of them become unallocated
## Example of Buddy System

<table>
<thead>
<tr>
<th>Request</th>
<th>1 Mbyte block</th>
<th>1 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 K</td>
<td>A = 128 K</td>
<td>256 K</td>
</tr>
<tr>
<td>240 K</td>
<td>A = 128 K</td>
<td>B = 256 K</td>
</tr>
<tr>
<td>64 K</td>
<td>A = 128 K</td>
<td>C = 64 K</td>
</tr>
<tr>
<td>256 K</td>
<td>A = 128 K</td>
<td>C = 64 K</td>
</tr>
<tr>
<td></td>
<td>A = 128 K</td>
<td>C = 64 K</td>
</tr>
<tr>
<td></td>
<td>128 K</td>
<td>C = 64 K</td>
</tr>
<tr>
<td></td>
<td>E = 128 K</td>
<td>C = 64 K</td>
</tr>
<tr>
<td></td>
<td>E = 128 K</td>
<td>128 K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512 K</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Buddy System

Tree representation of buddy system before “Release B”
Address Types

- Because of swapping and compaction, a process may occupy different main memory locations during its lifetime.
- Hence physical memory references by a process cannot be fixed.
- This problem is solved by distinguishing between logical address and physical address.
Address Types

- A physical address (absolute address) is a physical location in main memory.
- A logical address is a reference to a memory location independent of the physical structure/organization of memory.
- Compilers produce code in which all memory references are logical addresses.
- A relative address is an example of logical address in which the address is expressed as a location relative to some known point in the program (e.g., the beginning).
Address Translation

- Relative address is the most frequent type of logical address used in program modules (i.e., executable files)
- Such modules are loaded in main memory with all memory references in relative form
- Physical addresses are calculated “on the fly” as the instructions are executed
- For adequate performance, the translation from relative to physical address must be done by hardware
Address Translation Example

- When a process is assigned to the running state, a **base register** (in CPU) gets loaded with the starting physical address of the process.
- A **bound register** gets loaded with the process’s ending physical address.
- When a relative addresses is encountered, it is added with the content of the base register to obtain the absolute address which is compared with the content of the bound register.
- This provides security protection: each process can only access memory within its process image.
Paging

- Main memory is divided into equal fixed-sized chunks (of relatively small size), called frames (e.g., similar to partitions in the partitioning approaches)
- Trick: each process is also divided into chunks of the same size called pages
- The process pages can thus be assigned to the available frames in main memory
- Consequence: a process does not need to occupy a contiguous portion of memory
Example of process loading

- Now suppose to accommodate Process D with five pages, process B is swapped out.
When process B is swapped out, the pager loads a new process D consisting of 5 pages.

Process D does not occupy a contiguous portion of memory.

External fragmentation?  
Internal fragmentation?
Example of process loading

- When process B is swapped out, the pager loads a new process D consisting of 5 pages.
- Process D does not occupy a contiguous portion of memory.
- There is no external fragmentation.
- Internal fragmentation consist only of the last page of each process.
Page Tables

- The OS now needs to maintain (in main memory) a page table for each process.
- Each entry of a page table consists of the frame number where the corresponding page is physically located (in the main memory).
- The page table is indexed by the page number to obtain the frame number.
- A free frame list, available for pages, is maintained.
Summary and issue correction
Discussed on Oct 25 and 30

- Midterm Exam
- Syllabus Correction
  - No grace period on projects
  - HW grades will be averaged based on the number of problems
- Anonymous Survey
Logical address in paging

- Within each program, each **logical address** must consist of a **page number** and an **offset** within the page.

- A CPU register always holds the starting physical address of the **page table** of the currently running process.

- Presented with the logical address (page number, offset) the processor accesses the page table to obtain the **physical address** (frame number, offset).
Logical-to-physical address translation in paging

- Given a relative address 1502 of an instruction, where is the instruction physically located?

- Phase 1: Divide logical address, and find page # and offset.

- Phase 2: Logical-to-physical translation
Logical-to-Physical Address Translation in Paging

16-bit logical address

6-bit page #  10-bit offset

00000010111101111110

Process page table

000101
000110
011001

(a) Partitioning

(b) Paging (page size = 1K)

Relative address = 1502
0000010111011110

Logical address = Page# = 1, Offset = 478
0000010111011110

16-bit physical address

00011100011110111110
To facilitate efficient logical-physical address translation, the page size is always a power of 2.

E.g., if 16 bits addresses are used and page size = 1K, we need 10 bits for offset and have 6 bits available for page number.

Then the 16 bit address obtained with the 10 least significant bit as offset and 6 most significant bit as page number is a location relative to the beginning of the process.
Logical-to-Physical Address Translation in Paging

- By using a page size of a power of 2, the pages are **invisible** to the programmer, compiler/assembler, and the linker.

- Address translation at run-time is then easy to implement in hardware.
  - Logical address \((n,m)\) gets translated to physical address \((k,m)\) by indexing the page table and appending the same offset \(m\) to the frame number \(k\).
Simple Segmentation

- Each program is subdivided into blocks of non-equal size called segments.
- When a process is dynamically loaded into the partitions of the main memory, its different segments can be located in any partitions.
- Each segment is fully packed with instructions/data: no internal fragmentation.
- There is external fragmentation.
Simple Segmentation

- In contrast with paging, segmentation is visible to the programmer
  - provided as a convenience to organize logically programs (e.g., data in one segment, code in another segment)
  - must be aware of segment size limit
- The OS maintains a segment table for each process. Each entry contains:
  - the starting physical addresses of that segment
  - the length of that segment (for protection) \(\Rightarrow\) this is different from paging.
Logical-to-Physical Address Translation in Segmentation

Presented with a logical address (segment index, offset) = (n,m), the CPU can use the segment index n to obtain the partition index k and the length l of that segment using the table.

The physical address is obtained by adding m to k (in contrast with paging).

The hardware also compares the offset m with the length l of that segment to determine if the address is valid.
Logical-to-Physical Address Translation in Segmentation
Comparison of simple segmentation and paging

Two major differences!
Simple Segmentation and Paging Comparison

- Segmentation suffers from external fragmentation, whereas paging only yield a small internal fragmentation.
- Segmentation is visible to the programmer whereas paging is transparent.
- Segmentation requires more complicated hardware for address translation.
- Segmentation can be viewed as commodity offered to the programmer to organize logically a program into segments and using different kinds of protection (e.g., execute-only for code but read-write for data).
  - for this we need to use protection bits in segment table entries.
<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Partitioning</td>
<td>Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.</td>
<td>Simple to implement; little operating system overhead.</td>
<td>Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.</td>
</tr>
<tr>
<td>Dynamic Partitioning</td>
<td>Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.</td>
<td>No internal fragmentation; more efficient use of main memory.</td>
<td>Inefficient use of processor due to the need for compaction to counter external fragmentation.</td>
</tr>
<tr>
<td>Simple Paging</td>
<td>Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.</td>
<td>No external fragmentation.</td>
<td>A small amount of internal fragmentation.</td>
</tr>
<tr>
<td>Simple Segmentation</td>
<td>Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.</td>
<td>No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.</td>
<td>External fragmentation.</td>
</tr>
</tbody>
</table>
Homework Assignment 5
Chapter 7

- Problems 7.6, 7.7, 7.12 and 7.14.

Detail is presented on course website: http://inside.mines.edu/~hzhang/Courses/CSCI442/assignment.html

- Write down your full name clearly.
- Note: Problems are NOT Review Questions in the textbook!

Turn in hard copy solutions before the beginning of the class.