06. Memory Management

9th ed: Ch. 8, 9
10th ed: Ch. 9, 10
Objectives

• To describe the hardware required for memory protection
• To introduce the concepts of logical and physical addresses
• To discuss the problem of address binding
• To introduce the concept of segmentation
• To understand the problem of fragmentation
Outline

• Memory protection
• Memory allocation
Outline

• Memory protection
  • Address binding
  • Logical and physical addresses
  • Memory Management Unit (MMU)
  • Linking and loading

• Memory allocation
Memory management

• Will have many programs in memory simultaneously
  • Program code loaded from storage

• The CPU can only access registers and main memory directly
  • Register access in a single cycle, but memory access takes many cycles
  • Multiple levels of cache attempt to hide main memory latency (L1, L2, L3)

• Memory unit sees only a stream of
  • Address plus read request
  • Address plus data plus write request

• Need to protect memory accesses to prevent malicious or just buggy user programs corrupting other programs, including the kernel
Hardware address protection

- **Base** and **limit** registers define the logical address space
  - Base is the smallest legal address, e.g., 300040
  - Limit is the size of the range, e.g., 120900
  - Thus program can access addresses in the range [300040, 420940)

- CPU must check every user-mode memory access to ensure it is in that range
  - Exception raised to OS if not
Address binding

• Programs on disk are brought into memory to create running processes – but where in memory to put them given program code will refer to memory locations?
  • Consider a simple program and the assembly code it might generate
  • \([Rx]\) means the contents of memory at address \(Rx\)

\[
\begin{align*}
\text{int} & \ x, y; \\
\text{x} & = 5; \\
\text{y} & = x + 3; \\
\text{str} & \ #5, [Rx] ; \text{store 5 into x} \\
\text{ldr} & \ R1, [Rx] ; \text{load value of x from memory} \\
\text{add} & \ R2, R1, \#3 ; \text{and add 3 to it} \\
\text{str} & \ R2, [Ry] ; \text{and store result in y}
\end{align*}
\]

• Address binding happens at three different points
  • **Compile time**: If memory location known *a priori*, absolute code can be generated; requires recompilation if base location changes
  • **Load time**: Need to generate relocatable code if memory location is not known at compile time
  • **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another

• Bindings map one address space to another – requires hardware support
Logical vs physical addresses

• The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  • **Logical** (virtual) **address** – as generated by the CPU
  • **Physical address** – address seen by the memory unit
  • Identical in compile-time and load-time address-binding schemes
  • Differ in execution-time address-binding schemes

• The logical/physical address space is the set of all logical/physical addresses generated by a program

• Need hardware support to perform the mapping from logical to physical addresses at run time
Memory Management Unit (MMU)

• Hardware that maps logical to physical addresses at run time
• Conceptually simple scheme: replace base register with relocation register
• Add the value in the relocation register to every address generated by a user process at the time it is sent to memory
  • User programs deal with logical addresses, never seeing physical addresses
• Execution-time binding occurs when reference is made to location in memory
  • Logical address is bound to physical address by the MMU
Dynamic linking and loading

- Linking combines different object code modules to create a program
  - **Static linking** – all libraries and program code combined into the binary program image
  - **Dynamic linking** – postpone linking to execution time
- Dynamic linking is particularly useful for **system** or **shared** libraries
  - May need to track versions
- Calls replaced with a **stub**
  - A small piece of code to locate the appropriate in-memory routine
- Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes’ memory address, adding it if not
- Dynamic loading avoids loading routines until they’re called
  - Better memory usage as unused routines are never loaded
  - Requires they be compiled with relocatable addresses
  - Useful when large amounts of code are needed infrequently
- OS can help by providing libraries to implement dynamic loading
Outline

• Memory protection

• Memory allocation
  • Swapping
  • Dynamic allocation
  • Fragmentation
  • Compaction
  • Segmentation
Memory allocation

• Main memory must support both kernel and user processes
  • Limited resource, must allocate efficiently
  • **Contiguous allocation** is early method putting each process in one chunk of memory

• How to determine chunks?
  • Multiple fixed-sized partitions limits the degree of multiprogramming; prefer **variable partitioning**

• Main memory usually partitioned into two
  • Resident kernel, usually held in low memory alongside interrupt vectors
  • User processes then held in high memory, each in a single contiguous section

• Relocation registers used to protect
  • User processes from each other, and
  • OS code and data from being modified

• Can then allow actions such as kernel code being transient and kernel changing size
Swapping

• When physical memory requested exceeds physical memory in machine, temporarily swap processes out
  • Move processes from main memory to storage

• Significant performance impact
  • Time to transfer process to/from storage directly proportional to the amount of memory swapped
  • Context switches can thus become very expensive
  • E.g., 100MB process with storage transfer rate of 50MB/s

• Swapping default disabled
  • Enabled only while allocated memory exceeds threshold
  • Plus consider pending I/O to or from process memory space
  • System maintains a ready queue of ready-to-run processes with memory images on disk

• Must swapped out processes be swapped into the same physical addresses?
  • Depends on address binding method
Multiple variable-partition allocation

• **Holes**, blocks of available memory of various size are scattered throughout memory
  • When a process arrives, it is allocated memory from a hole large enough to accommodate it
  • Process exiting frees its partition, adjacent free partitions combined

• OS maintains information about:
  • allocated partitions and
  • free partitions (holes)
Dynamic allocation problem

• How to satisfy a request of size from a list of free holes?

• **First-fit**, allocate the first hole that is big enough

• **Best-fit**, allocate the smallest hole that is big enough
  • Requires searching entire list, unless maintained ordered by size
  • Produces the smallest leftover hole

• **Worst-fit**, allocate the largest hole
  • Also requires searching entire list, producing the largest leftover hole

• First-fit and best-fit better than worst-fit in terms of speed and storage utilization
Fragmentation

• Fragmentation results in memory being unused and unusable

• **External Fragmentation**
  • Occurs when free memory exists to satisfy a request but it is not contiguous
  • Can eventually result in blocking as insufficient contiguous memory to swap any process in

• **Internal Fragmentation**
  • Occurs when allocated memory is slightly larger than requested memory
  • Memory internal to a partition, but unused

• Analysis of first-fit indicates that for $N$ blocks allocated, 0.5 $N$ blocks lost to fragmentation
Compaction

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
- Compaction is possible only if
  - relocation is dynamic, and
  - done at execution time
- I/O problem
  - Pin job in memory while involved in I/O
  - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems
Segmentation

• Memory-management scheme supporting user view of memory
  • View a program as a collection of segments, logical program units such as the program, a procedure, an object, an array, etc

• Accessing memory requires user program to specify
  • Segment name (number) and
  • Offset within segment
Segmentation hardware

• Logical address is now a pair \(<\text{segment-number}, \text{offset}>\)

• **Segment table** maps to physical addresses via entries having
  • **Base**, the starting physical address where the segment resides
  • **Limit**, specifying the length of the segment

• **Segment-table base register (STBR)** points to the segment table’s location in memory

• **Segment-table length register (STLR)** indicates number of segments used by a program;
  Segment number \(s\) is legal if \(s < \text{STLR}\)

• Protection provided by associating with each entry in segment table
  • Validation bit indicating legal / illegal segment
  • Read/Write/Execute privileges
  • Associated with segments so code sharing occurs at segment level

• Segments vary in length so memory allocation is a dynamic storage-allocation problem
Sharing segments is subtle

- Consider jumps within shared code
  - Specified as a condition and a transfer address \(< segment-number, offset >\)
  - \(segment-number\) is (of course) this one
- So all programs sharing this segment must use the same number to refer to it
  - The difficulty of finding a common shared segment number grows as the number of users sharing a segment
  - Thus, specify branches as PC-relative or relative to a register containing the current segment number
  - Read only segments containing no pointers may be shared with different segment numbers

- Wasteful to store common information on shared segment in each process segment table
  - Also dangerous as can get out of sync between processes
- Assign each segment a unique **System Segment Number** (SSN)
  - **Process Segment Table** then maps from a **Process Segment Number** (PSN) to SSN

---

06. Memory Management
Summary

• Memory protection
  • Address binding
  • Logical and physical addresses
  • Memory Management Unit (MMU)
  • Linking and loading

• Memory allocation
  • Swapping
  • Dynamic allocation
  • Fragmentation
  • Compaction
  • Segmentation