Chapter 8: Main Memory
Main Memory

Main Memory

- Address Binding
- Dynamic Loading
- Dynamic Linking
- Paging

Primary Interest: Study the sequence of memory addresses generated by a process (a running program)
Main Memory

Background

- Main Memory: Large array of words (or bytes)
- CPU fetches Instructions from memory based on the value of the PC
- CPU uses the Load instruction fetch data from specific memory address
- CPU uses Store instruction to store data (from a register) into specific memory address

Instructions and data must be moved from their respective disk addresses to memory addresses before the CPU can operate on them
Main Memory

Processes

- OS takes a fixed segment of memory
- Each Process has a separate memory space
- Direct Memory access by CPU is slow
- To avoid CPU stall
  - Use Cache
- Protect OS from access by user processes:
- Protect user processes from each other:
  - Base register
    - Holds smallest memory address
  - Limit Register
    - Holds size of memory address range
- OS has unlimited access to user memory
  - Operates in kernel mode
  - Base and Limit registers loaded only by OS

Memory Unit

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OS</td>
</tr>
<tr>
<td>25600</td>
<td>Process -A</td>
</tr>
<tr>
<td>30004</td>
<td>Process -B</td>
</tr>
<tr>
<td>42094</td>
<td>Process -C</td>
</tr>
<tr>
<td>88000</td>
<td></td>
</tr>
<tr>
<td>102400</td>
<td></td>
</tr>
</tbody>
</table>
Unix Process Memory

The Unix Process memory is divided into logical segments

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max address</td>
<td>stack</td>
</tr>
<tr>
<td>Stack</td>
<td>heap</td>
</tr>
<tr>
<td>Dynamic storage</td>
<td>bss, data</td>
</tr>
<tr>
<td>Static variables</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>text</td>
</tr>
</tbody>
</table>
Main Memory
Address Protection

CPU compares each address generated in user mode with base and limit registers

CPU → limit register
      ↓
      <
      ↓
relocation register
      ↓
physical address
      → Memory

trap: addressing error address

logical address

+
Main Memory
Processing User Program

- Addresses in user program are typically symbolic
  - int count, num;
- Compiler & Assembler generate object file for each source file
- Linker generates a single executable object file by combining all object files for a program
  - The executable program
- Loader loads the executable object files into memory at fixed locations or specific locations determined by the OS
  - Part of the OS
- The running program
  - dynamically allocates memory (malloc)
  - gets space on stack during function call
Main Memory Unit
Address Binding

- Address binding of instructions and data to memory address:
  - Symbolic address $\rightarrow$ Relocatable address $\rightarrow$ Absolute address

  Compile Time

  Load Time
  or
  Execution Time
Address Binding
Compile Time

- External references (printf, scanf symbols) cannot be resolved in the Assembly stage
  - Compiler:
    - Puts an address of zero in the object code
    - Records external symbols and their location in a patch list
      - Linker will later resolve the references
  - Compiler does not have knowledge of the memory address
    - Assumes program starts from address 0
    - Records relocation info: “n bytes from start of module” in the object file

Compiler binds the symbolic addresses to relocatable addresses
Address Binding

Load Time

- Loader loads relocatable code into memory where it can be executed
  - Absolute Loader
    - Loads executable file into memory at fixed locations
  - Relocatable Loader
    - Loads executable file at fixed locations specified by the OS
  - Assembler and linker assume program starts at address 0

The Loader replaces the “0” addresses with the real start (Absolute) address
Address Binding
Execution Time

- If process can be moved during its execution from one segment to another
  - Binding must be delayed until run time

Need a special hardware to provide this capability:

**Memory Management Unit (MMU)**
Logical vs. Physical Address Space Concepts

- **Logical Address:**
  - Address generated by the CPU during compile time and load time
- **Physical Address**
  - Address loaded into the memory-address register of the memory unit during execution time

Address Binding

- **Compile–time and Load-time:**
  - Logical and physical address are identical
- **Execution-time:**
  - Logical address differs from Physical address
    - Logical address $\leftrightarrow$ Virtual address

- **Logical address space** = set of all logical addresses created by a program
- **Physical address space** = set of all physical addresses assoc with the logical addresses
Logical vs. Physical Address Space

Memory Management Unit

- Hardware device that maps virtual to physical address

- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

- The user program deals with *logical* addresses; it never sees the *real* physical addresses
Dynamic Relocation using a Relocation Register
Memory Management Unit

![Diagram showing dynamic relocation using a relocation register and a Memory Management Unit (MMU). The diagram illustrates the process of converting a logical address to a physical address, with the relocation register holding the relocation information.](image-url)
Memory Management

Dynamic Loading

- Load main Program into memory
- Store Routines in Relocatable Load format on disk
- As main program (or subsequently loaded routine) executes it calls other routines
  - The calling program checks if called routine is loaded?
    - Reference the called routine
  - Else:
    - Calling program calls Relocatable Linker Loader to load Routine into memory
    - Relocatable Linker Loader updates address space of calling program

Benefits:

- Better memory-space utilization; unused routine is never loaded
  - No need to load entire program and data into memory for process to execute
- Useful when large amounts of code are needed to handle infrequently occurring cases
Dynamic Linking
Dynamic Link Libraries

- A stub is included into the image of each library routine reference
  - Locates appropriate memory-resident library routine(s)
  - Indicates how to load the library routine(s)
- As user program executes, it calls system library routines:
  - The system call leads to execution of stub
    - The stub checks (via the OS) if the library routine is in memory
      - Otherwise referenced library routine is loaded
    - Stub replaces itself with the address of the routine, and executes the routine
- Linking postponed until execution time

Benefits:
- Dynamic linking is particularly useful for libraries
- No need to load entire copy of library into memory for each process to execute
Swapping:
- A process is swapped temporarily out of memory to a **backing store**, and then brought back into memory for continued execution
  - Round-Robin CPU Scheduling using time slice

**Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

- If Assembly or Load Time Address Binding:
  - The swapped process occupies same memory address
- Elif Execution-time binding:
  - The swapped process may occupy a different memory address

**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
  - Need longer time-slice
- OS: UNIX, Linux, and Windows adopt variant of swapping
Swapping

Disk serves as a Backing Store

1. swap out

2. swap in
Contiguous Memory Allocation

- Main memory usually divided into two partitions:
  - Lower memory
    - Usually Resident operating system
      - The interrupt vector usually occupies this partition
  - High Memory
    - Stores user processes
    - Each process is contained in a single contiguous section of memory

How do you allocate contiguous memory sections to processes waiting in the input queue?
Contiguous Memory Allocation
Memory Mapping and Protection

- MMU Maps logical address dynamically by adding value in relocation register
  - Relocation register
    - Smallest physical address
  - Limit register
    - Range of logical addresses

- Dispatcher loads values in:
  - Relocation register
  - Limit register
Contiguous Memory Allocation
Multiple-partition allocation

- The OS maintains information about
  - Allocated Partitions in memory
  - Free Partitions (hole) in memory
    - Set of Blocks of available memory
      - Blocks may not be contiguous memory locations
  - As process arrives from waiting input queue and needs memory:
    - OS searches for a hole large enough to accommodate it
      - Allocate enough block to accommodate process
      - Leave remaining holes for other processes
    - If hole is too large:
      - Split holes into two parts (Arriving processes, & Set of holes)
    - Else:
      - Wait until correct block size is found
  - When process terminates
    - OS returns its block of memory to set of holes
How do you satisfy a request of size $n$ from a list of free holes?
Dynamic Storage Allocation Problem

How to satisfy a request of size $n$ from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
  - Start the search at beginning of set of holes or where previous search ended

- **Best-fit**: Allocate the *smallest* hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole.

- **Worst-fit**: Allocate the *largest* hole
  - Must search entire list.
  - Produces the largest leftover hole.

First-fit and Best-fit may result in set of many tiny non-contiguous holes?  
**Fragmentations!**
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Memory is allocated in fixed sized blocks to avoid housekeeping overhead for tiny blocks
- 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 is done at execution time
    - Move program and data
    - Change contents of base register to reflect the new base address

Fragmentation also occurs on the backing store (disk)
Why is compaction solution not possible in this environment?
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Each process is divided into a set of small, fixed size partitions (Pages)
  - Divide Physical Memory into a large number of small, fixed-size partitions (Frames)
    - Page size = Frame size
    - 512bytes – 16K bytes
  - Whole Process is still loaded into memory
    - But Pages need not be in contiguous locations
Paging

- Address generated by CPU (Logical address) contains:
  - Page number and physical offset
  - Set up a Page table to translate logical address to Physical address

- **Page number** \( (p) \) – used as an index into a page table which contains base address of each page in physical memory

- **Page offset** \( (d) \) – combined with base address to define the physical memory address that is sent to the memory unit
Paging Example
Translation of Logical & Physical Memory

![Diagram showing page table and frame map]

- **Page Table**
  - Logical memory
  - page 0
  - page 1
  - page 2
  - page 3

- **Frame Map**
  - Frame number
  - 0: page 0
  - 1: page 1
  - 2: page 2
  - 3: page 3

This diagram illustrates the translation of logical memory pages to physical memory frames.
Paging Example
Translation of Logical & Physical Memory

logical memory

page table
physical memory

Operating Systems CS 33211
Managing Pages and Frames
OS support

- OS keeps track of frames in Physical memory
  - Data Structure: Frame Table
    - A bit map
      - 1 indicates frame is available
      - 0 indicates frame is allocated to a page
    - Page ID of Process
  - To find an available frame for a page:
    - Look for the 1\textsuperscript{st} 1-bit in bit map
    - Allocate the page to the physical memory address assoc with the 1\textsuperscript{st} 1-bit

Before Allocation

After Allocation
Basic Requirements

- Every access to memory goes thru’ paging map
  - Need efficient translation for table look up
    - How about using [high speed] registers?
      - Size could be a limiting factor
  - Else:
    - Use registers + Page Table in Memory:
      → Base register points to page table held in memory
      → Introduces a second memory access per page (inefficient)
- Need to resolve internal fragmentation of physical memory
Managing Pages and Frames
Hardware support using Cache

- **Cache** allows:
  - Efficient use of “Registers and Page Table in Memory”

- **Cache** [Associative memory register] – Do parallel search

  ![Address translation table]

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address translation (A´, A´´)
  - If A´ is in cache, get frame # out
  - Otherwise get frame # from page table in memory

- But how do you get frame number in cache?
Managing Pages and Frames
Cache: Translation Look-aside Buffer

TLB Hit?  

TLB Hit Ratio?

Large Logical address space → large pages in the page table?
Each entry in the page table is associated with a set of bits:

- **Read-write bit, Read-only, Valid-invalid** etc..

  - **Valid-invalid bit**
    - If bit is set to “1” (valid), the associated page is in the process’ logical address space, and is thus a legal page
    - If bit is set to “0” (invalid) the page is not in the process’ logical address space
Valid (v) or invalid (i) bit in Page Table

- Frame number:
  - 0: v
  - 1: v
  - 2: v
  - 3: v
  - 4: v
  - 5: v
  - 6: i
  - 7: i

- Valid–invalid bit:
  - 0: v
  - 1: i
  - 2: v
  - 3: v

- Pages:
  - page 0
  - page 1
  - page 2
  - page 3
  - page 4
  - page 5
  - page 6
  - page 7
  - page 8
  - page 9
  - page n
Structuring Page Table

2-Level Page-table

- Large Logical address space $\rightarrow$ large pages in the page table $\rightarrow$ Increased table loop up
  - Modern Computers Address Space:
    - $2^{32}$ to $2^{64}$
    - Up to one million entries in table

- One Solution:
  - Divide and Conquer!
    - 2-Level Page Table
2-Level Paging Algorithm
Page-Table is Paged
2-Level Paging Example

- Consider a logical address (on 32-bit machine with 4K page size) divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Suppose the page table is paged: the page number is sub-divided into:
  - a 10-bit page number (leading page number bits)
  - a 10-bit page offset (trailing page number bits)

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>P₂</td>
</tr>
</tbody>
</table>

- where P₁ is an index into the outer page table
- and P₂ is the displacement within the page of the outer page table
2-Level Page-Table 32-bit Architecture
Logical Address Translation
Segmentation

- What’s a programmer’s typical view of a memory?
  - main() → routines, variables, arrays, objects, data structures, etc...
- Segmentation – Supports user’s view of memory
  - Basic Idea: Divide process into separate segments in memory
    - Each segment serves different purpose
      - Code, Static Data, function, object
        - Segments may have different sizes
      - The set of Segments that make up a process may not be in contiguous memory locations
      - Space within a segment is contiguous
  - Each segment has protection bits
    - Read-only segment (code)
    - Read-write segments (data)
    - Allows processes to share code and data
Segment Addresses

- Logical address [generated by compiler]:
  - Segment number
  - Offset from beginning of the segment

- What is stored in the instruction?
  - Simple method:
    - Leading bits specify the segment
    - Trailing bits specify offset
  - Implicit segment specification
    - Segment is selected implicitly by instruction being executed (code vs. data)
  - Explicit segment specification:
    - Instruction prefix can request that a specific segment be used (386/486)
Implementing segments

- A segment table keeps track of every segment in a particular process
  - Each entry contains
    - base address where segments resides in memory
    - limit (length of segment)
  - Associated with protection bits
    - Read vs. read/write, sharing allowed
  - Requires additional hardware support
    - Base and limit registers or
    - Segment table base register (points to a segment table in PCB)
Implementing Segments

Virtual address

Physical address

Access physical memory

Trap: Address error

Segment  offset

base  limit

offset  base
Segmentation Example
Sharing of Segments