CHAPTER 8: MEMORY MANAGEMENT
Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium
Background

- Program must be brought into memory and placed within a process for it to be run.
  - **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program.

- Main memory and registers are only storage CPU can access directly.
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles
  - **Cache** sits between main memory and CPU registers

- Protection of memory required to ensure correct operation
Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:**
  - If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes.

- **Load time:**
  - Must 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. Need hardware support for address maps (e.g., *base* and *limit registers*).
A pair of **base** and **limit** registers define the logical address space.
Multistep Processing of a User Program
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
  - **Logical address** — generated by the CPU; also referred to as virtual address
  - **Physical address** — address seen by the memory unit

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.

- Logical (virtual) and physical addresses differ in execution-time address-binding scheme.
Memory-Management Unit (MMU)

- 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
Dynamic Loading

- A routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
- Implemented through program design
Dynamic Linking

- Linking postponed until execution time

- Small piece of code, stub, used to locate the appropriate memory-resident library routine

- Stub replaces itself with the address of the routine, and executes the routine

- Operating system needed to check if routine is in processes’ memory address

- Dynamic linking is particularly useful for libraries
Overlays

- Keep in memory only those instructions and data that are needed at any given time.

- Needed when a process is larger than amount of memory allocated to it.

- Implemented by users, no special support needed from operating system, programming design of overlay structure is complex
Overlays for a Two-Pass Assembler

The diagram illustrates the layout of overlays for a two-pass assembler. The diagram includes:

- **Symbol Table**: 20K
- **Common Routines**: 30K
- **Overlay Driver**: 10K

The diagram shows that the assembler passes 1 and 2 occupy space from 70K to 80K.
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

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

- **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
Swapping

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
Schematic View of Swapping

1. Swap out
2. Swap in
Contiguous Allocation

- Main memory usually can be divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically
Hardware support for relocation and limit registers
Multiple-partition allocation

- **Hole** – block of available memory; holes of various sizes are scattered throughout memory.

- When a process arrives, it is allocated memory from a hole large enough to accommodate it.

- Operating system maintains information about:
  a) allocated partitions
  b) free partitions (hole)
Memory Allocation

How to satisfy a request of size $n$ 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; must search entire list, unless ordered by size. Produces the smallest leftover hole.

- **Worst-fit**: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit are usually better than worst-fit in terms of speed and storage utilization.
Fragmentation

- **External Fragmentation**
  - total memory space satisfies 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
Fragmentation (cont.)

- 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

- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).

- Divide logical memory into blocks of same size called **pages**.
Paging

- Keep track of all free frames

- To run a program of size $n$ pages, need to find $n$ free frames and load program

- Set up a page table to translate logical to physical addresses

- Internal fragmentation
Address Translation Scheme

- Address generated by CPU is divided into:
  - **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

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(d)</td>
</tr>
<tr>
<td>(m - n)</td>
<td>(n)</td>
</tr>
</tbody>
</table>

- For given logical address space \(2^m\) and page size \(2^n\)
Paging hardware
Paging Model of logical and physical memory
Paging Example

logical memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
</tbody>
</table>

page table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

physical memory
Free Frames

Free-frame list

14
13
18
20
15

page 0
page 1
page 2
page 3

new process

(a)

Free-frame list

15

page 0
page 1
page 2
page 3

new process

(b)

new-process page table

0 14
1 13
2 18
3 20

21
Implementation of Page Table

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PRLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
Associative Memory

- Associative memory – parallel search

<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>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address translation \((p, d)\)

- If \(p\) is in associative register, get frame \#\ out
- Otherwise get frame \#\ from page table in memory
Paging Hardware With TLB
Effective Access Time

- **Associative Lookup** = \( \varepsilon \) time unit
- Assume memory cycle time is 1 microsecond

- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

- Hit ratio = \( \alpha \)

- **Effective Access Time** (EAT)
  \[
  \text{EAT} = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\
  = 2 + \varepsilon - \alpha
  \]
Memory Protection

- Memory protection implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process’ logical address space
Valid (v) or Invalid (i) Bit In A Page

Table
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.

- **Private code and data**
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.
Sharing of code in a paging environment
Structure of Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables
Hierarchical Page Tables

- Break up the logical address space into multiple page tables

- A simple technique is a two-level page table
Two-Level Page-Table Scheme
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset

- Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>

where $p_i$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table
Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture
# Three-level Paging Scheme

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$d$</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Hashed Page Tables

- Common in address spaces > 32 bits

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.

- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Hashed Page Table
Inverted Page Table

- One entry for each real page of memory

- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

- Use hash table to limit the search to one — or at most a few — page-table entries
Inverted Page Table Architecture
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays
User’s View of a Program
Logical View of Segmentation

user space

physical memory space
Segmentation Architecture

- Logical address consists of a two tuple:
  \(<\text{segment-number}, \text{offset}>\),

- **Segment table** — maps two-dimensional physical addresses; each table entry has:
  - base — contains the starting physical address where the segments reside in memory
  - limit — specifies 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 \( s < \text{STLR} \)
Segmentation Architecture (Cont.)

- **Relocation.**
  - dynamic
  - by segment table

- **Sharing.**
  - shared segments
  - same segment number

- **Allocation.**
  - first fit/best fit
  - external fragmentation
Protection. With each entry in segment table associate:

- validation bit = 0 ⇒ illegal segment
- read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level

Since segments vary in length, memory allocation is a dynamic storage-allocation problem

A segmentation example is shown in the following diagram
Segmentation hardware
Example of Segmentation
Sharing of Segments

Logical memory process $P_1$

Logical memory process $P_2$

Segment table process $P_1$

Segment table process $P_2$

Sharing of Segments

Segment 0

Segment 1

Data 1

Data 2

Editor

Physical memory

Segment 0

Segment 1

Limit | Base
---|---
25286 | 43062
4425 | 68348

Limit | Base
---|---
25286 | 43062
8850 | 90003
Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
Logical to Physical Address Translation in Pentium

![Diagram showing the process of logical to physical address translation.]

**Diagram:**
- **CPU** → **segmentation unit** → **paging unit** → **physical memory**
  - Logical address: CPU
  - Linear address: Segmentation unit
  - Physical address: Paging unit

**Table:*

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>10</td>
</tr>
<tr>
<td>$p_2$</td>
<td>10</td>
</tr>
<tr>
<td>$d$</td>
<td>12</td>
</tr>
</tbody>
</table>
Intel Pentium Segmentation

logical address

selector  offset

descriptor table

segment descriptor

32-bit linear address
Pentium Paging Architecture
Linear Address in Linux

Broken into four parts:

<table>
<thead>
<tr>
<th>global directory</th>
<th>middle directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
</table>
Three-level Paging in Linux
END OF CHAPTER 8