Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Process is running program
  - Process must be in memory
- Input queue:
  collection of processes on the disk waiting to be brought into memory to run
- Program goes through several steps before being run

Binding of Instructions and Data to Memory

3 points in time

Compile time:
- Memory location fixed ➔ generate absolute code
- Must recompile code if starting location changes

Load time:
- Memory location not known ➔ generate relocatable code

Execution time:
- Process can be moved
- Code uses logical addresses
- Logical to physical translation done as part of execution
- Need hardware support (e.g., base and limit registers)
Logical vs. Physical Address Space

- Crucial separation for 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

  - relocation register:
    - Contains address to be added to every address generated by a user process at the time it is sent to memory

  - user process deals with logical addresses
    - it never sees the real physical addresses
Dynamic relocation using a register

CPU  \[\text{logical address} 345\] \[\text{MMU}\] \[\text{physical address 14346}\] \[\text{memory}\] relocation register

Dynamic Loading

- 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 process' memory space
- 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 process is larger than amount of memory allocated to it
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex

Overlays for a Two-Pass Assembler

Swapping

- process is moved from memory to backing store, moved back later for continued execution
  - Backing store:
    - fast large magnetic disk
    - must accommodate all processes
    - must provide direct access
  - Roll out, roll in
    - swapping variant for priority-based scheduling algorithms
    - lower-priority process is swapped out
    - higher-priority process loaded and executed
    - swap time is proportional to process size
  - Common on: UNIX, Linux, and Windows
Contiguous Allocation

- Main memory has two partitions:
  1. Resident operating system, in low memory, with interrupt vector
  2. User processes in high memory

- User process partition
  - Relocation-register scheme
    1. Protect user processes from each other, and from changing operating-system code and data.
    2. Holds smallest physical address
    3. Limit register contains range of logical addresses

Hardware Support for Relocation and Limit Registers
Contiguous Allocation (Cont.)

- Multiple user partition allocation
  - Hole – variable size block of available memory
  - New process is allocated memory from a hole
  - Operating system maintains information about:
    a) allocated partitions
    b) free partitions (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
- **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 better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation**
  - Many small holes
- **Internal Fragmentation**
  - Allocated memory larger than requested memory
  - Reduce external fragmentation by compaction
    - Move holes together into one large block
    - Requires dynamic relocation 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

- Process is allocated physical memory in pages
- Memory may be non-contiguous
- Physical memory is fixed-sized blocks called **frames** (size is power of 2: 512, 1024, … 4MB …)
- Logical memory is blocks of same size called **pages**
- To run a program of size $n$ pages, need to find $n$ free frames and load program
- Translate logical to physical address via page table
- Also: 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

### Address Translation Architecture

- Diagram showing the flow of logical to physical addresses through a page table.
Paging Example

Before allocation

After allocation

Free Frames
Address Translation Architecture

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 (PTLR)
  - indicates size of the page table
- doubles memory accesses:
  1. page table
  2. data/instruction
- solution:
  - special fast-lookup hardware cache (associative memory)
  - translation look-aside buffer (TLB)
  - if not found in TLB → consult page table

Paging Hardware With TLB
**Effective Access Time**

- Associative lookup time: $\varepsilon$
- Memory cycle time: $T$
- Hit ratio: $\alpha$
  - Percentage of times that a page number is found in TLB
  - Depends on TLB size
- Effective Access Time (EAT)

\[
EAT = (\tau + \varepsilon) \alpha + (2\tau + \varepsilon)(1 - \alpha)
\]

**Example:**

\[
EAT = (100\text{ns} + 20\text{ns}) \times 0.8 + (200\text{ns} + 20\text{ns}) \times 0.2
\]

\[
= 140\text{ns}
\]

---

**Memory Protection**

- Page table has bit for each frame
- "valid": page is legal, i.e. belongs to address space
- "invalid": page is not in the process' logical address space

---

**Valid (v) or Invalid (i) Bit In Page Table**
Page Table Structures

- 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 Paging Example

- 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
- page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- logical address is as follows:

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

$p_1$ is an index into the outer page table
$p_2$ is the displacement within the page of the outer page table
Two-Level Page-Table Scheme

Address-Translation Scheme

- two-level paging architecture

Hashed Page Tables

- Common in address spaces > 32 bits
- virtual page number is hashed into a page table
- 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 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:
  - virtual address of the page
  - information which process that owns the page
- decreases page table size
- increases search time
  - Use hash table to limit the search

Inverted Page Table Architecture
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 the 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
  - Pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

Segmentation

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

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

- Segment table: each table entry has:
  - base – starting physical address of segments in memory
  - limit – length of the segment

- Segment-table base register (STBR)
  - segment table’s location in memory.

- Segment-table length register (STLR)
  - segment number \( s \) is legal if \( s < \text{STLR} \)
Segmentation Architecture Issues

- Relocation:
  - dynamic, via segment table
- Sharing:
  - shared segment have same segment number
- Allocation:
  - first fit/best fit
  - external fragmentation
- Protection:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges

Segmentation Hardware

Example of Segmentation
Segmentation with Paging

- Segmentation problem:
  - external fragmentation
  - lengthy search times

- Solution: paged segmentation
  - segment-table entry contains
    - not base address of the segment
    - but base address of a page table for this segment