ICOM5007
OPERATING SYSTEMS
Programming

Memory Management

Objectives

• To provide a detailed description of various ways of organizing memory hardware
• To discuss various memory-management techniques, including paging and segmentation
• To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background (1)

• Program must be brought (from disk) into memory and placed within a process for it to be run..
• Execution of a program requires constant access to main memory:
  – instructions are fetched from memory & decoded
  – data operands may be fetched from or written back to memory
• Main memory and registers are only storage CPU can access directly
  – ISA instructions accept memory addresses but no disk addresses…
  – data must be moved to memory in order to be used by the CPU

Background (2)

• Register access in one CPU clock (or less)
  – one or more operations per clock tick
• Main memory can take many cycles
  – CPU stalls frequently, hurting performance
• Cache sits between main memory and CPU registers with the goal to reduce time of memory accesses
• Proper management of memory is required to ensure correct operation of the several processes that may be in memory concurrently.
Memory Hierarchy

Base and Limit Registers (1)
- Ensure correct memory operations to protect the OS from accesses from other processes and between processes
  - make sure each process has a separate memory space
  - determine range of valid addresses that a process accesses
- A simple approach to guarantee memory protection among processes is to use registers to delimit the address space area of processes.

Base and Limit Registers (2)
- This can be achieved by a pair of base and limit (the size of the allocated block) registers
- each process has its own values for both registers
- the physical address space of a process goes from memory word at address base to the memory word at address base+limit-1
- Valid addresses for a particular process must satisfy:
  \[ \text{base} \leq \text{address} < \text{base}+\text{limit} \]

Base and Limit Registers (3)
- Each process keeps, as part of its context, the values of base and limit registers.
- Protect user process to access memory areas outside its physical address space
  - any attempt by a program to access the OS or other program’s space generates an error trapped by the OS (fatal error)
- These are properly loaded whenever process goes to Running state
Base and Limit Address Translation

Address Translation Details

- The **logical address space** has addresses in range 0 .. limit-1
- Addresses in range 0 .. limit-1 are referred to **logical addresses**
- Any memory reference through a logical address needs to be properly mapped to a physical memory:

  ```java
  Address p(LogicalAddress a, Register b, Register limit) throws IAE {
      if (a >= limit) throw new IAE(...);
      return b+a;
  }
  ```

Binding of Instructions and Data to Memory

- Programs usually reside on disk as executable files
  - to execute, must be brought into memory and placed within a process
  - may be moved between disk and memory during execution (**input queue**)
- Allow processes to reside in any part of physical memory
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location is known a priori, absolute code can be generated; must recompile code if starting location changes
  - **Load time**: relocatable code if memory location is not known at compile time; starting address changes only reload code
  - **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) Used by most modern OS
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 used in memory address register
- 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

- All routines are kept on disk
- 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 (1)

- Some OS only support **static linking**
  - Libraries treated as object module combined by the loader into the binary program image
- Linking postponed until execution time **dynamic linking**
- 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

Dynamic Linking (2)

- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Operating system needed to check if routine is in processes’ memory address

Swapping (1)

- A process can be swapped temporarily out of memory to a backing store, and 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 (2)

- 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)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
Schematic View of Swapping

Contiguous Allocation; Partitioning

- Main memory usually 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**

Hardware Support for Relocation and Limit Registers

- **Limit register** contains **range of logical addresses** – each logical address must be less than the limit register
- **MMU** maps logical address **dynamically**

Contiguous Allocation; Partitioning

- **base** relocation register
- **limit register**
- **physical address**
- **trap: addressing error**
- CPU
  - logical address
    - yes
    - no
    - physical address
  - memory

1. swap out
2. swap in
Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - **Hole** block of available memory; various size 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)

Dynamic Storage-Allocation Problem

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

Fragmentation (1)

- **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

Fragmentation (2)

- 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 (1)

- 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 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**

Paging (2)

- 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
- Generates **internal fragmentation**.
- No **external fragmentation**.

Address Translation Scheme (1)

- Consider an address of length $m$ bits.
- 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>$m-n$ bits</td>
<td>$n$ bits</td>
</tr>
</tbody>
</table>

Address Translation Scheme (2)

- The logical address corresponds to
- logical address space of **size $2^m$**
- a page table of $2^{m-n}$ entries
- page **size is $2^n$**
Paging Hardware

Frame number: $f = 2^n$

Frame $f$

Size = $2^n$

Paging Model of Logical and Physical Memory

Paging Example

32-byte memory and 4-byte pages
Implementation of Page Table (1)

- 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 cache called associative memory or translation look-aside buffers (TLBs).

Implementation of Page Table (2)

- The two memory access problem can be solved by the use of a special fast-lookup cache called associative memory or translation look-aside buffers (TLBs)
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

- Associative memory – parallel search
  - 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 (1)

- TLB Lookup = $\varepsilon$ time unit ~ 0.2 microsecond (\(\mu s\))
- Assume memory access time is 1 \(\mu s\)
- Hit ratio – percentage of times that a page number is found in the TLB; ratio related to number of associative registers
- Hit ratio = $\alpha$

Effective Access Time (EAT)

\[
EAT = (\text{mem access} + \varepsilon) \alpha + (2 \text{ mem access} + \varepsilon)(1 - \alpha)
\]

\[
\approx 2.2 - \alpha \ \mu s
\]

Effective Access Time (2)

- Effective Access Time (EAT)
  
  \[
  EAT = (\text{mem access} + \varepsilon) \alpha + (2 \text{ mem access} + \varepsilon)(1 - \alpha)
  \]

  \[
  EAT = (1 + 0.2) \alpha + (2 + 0.2)(1 - \alpha) \ \mu s
  \]

  \[
  \approx 2.2 - \alpha \ \mu s
  \]

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

<table>
<thead>
<tr>
<th>Frame Number</th>
<th>Valid–Invalid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>v</td>
</tr>
<tr>
<td>1</td>
<td>v</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>3</td>
<td>v</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
</tr>
<tr>
<td>5</td>
<td>v</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td>7</td>
<td>i</td>
</tr>
</tbody>
</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 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.
  - The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example

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 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

Two-Level Paging Example

- $p_i$ 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

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

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

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.

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.
Segmentation (1)

- Users view of memory from the perspective of programs, which consists of several units.
- Each such unit may have different requirements of memory.
- A separate memory segment can be assigned to each such unit.
- Some systems support that user view of memory by supporting segmentation.
- A program is a collection of segments - logical units such as:
  - main program

Segmentation (2)

- A program is a collection of segments - logical units such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local/global variables
  - common block
  - stack
  - symbol table
  - arrays

User’s View of a Program

Process’s memory area as conceived by the users (programmers).
Logical View of Segmentation

- For simplicity of implementation, each process’s segment is identified by a number.
- Segments are created by the compiler of the program.
- Logical addresses have the form: <segment-number, offset>

Segmentation Architecture (1)

- Logical address consists of a two tuple:
  - <segment-number, 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 current length of the segment

Segmentation Architecture (2)

- 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 < STLR

Segmentation Architecture (Cont.)

- 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.
**Segmentation Hardware**

- Logical to physical address translation under segmentation

---

**Example of Segmentation**

- 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

Pentium Paging Architecture

Intel Pentium Segmentation

Linear Address in Linux

Broken into four parts:

global directory  middle directory  page table  offset
Three-level Paging in Linux