Chapter 8: Main Memory

We now know how OS organizes and coordinate a group of processes executing concurrently.

- Dealt with synchronization issues arising from concurrency.
- Dealt with scheduling issues that impact fairness, utilization.

Now we get into the non-computational resources of the machine.

- Memory
- Disks
- Start with memory.

OS as a Resource Manager

<table>
<thead>
<tr>
<th>OS Area of Responsibility</th>
<th>CPU</th>
<th>Memory</th>
<th>I/O Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>CPU task size, stack size</td>
<td>MMU, Memory hierarchy (registers, cache, main, secondary)</td>
<td>Disk, tape, and drives, backup, Keyboard, monitor, printer, joystick, etc.</td>
</tr>
<tr>
<td>OS key concept and data structures</td>
<td>Process PCB Queues</td>
<td>Logical address space</td>
<td>Running next</td>
</tr>
<tr>
<td>Issues</td>
<td>Scheduling, Synchronization, Deadlock</td>
<td>Address translation, Translation</td>
<td>Running next</td>
</tr>
<tr>
<td>Performance</td>
<td>THP, CPU utilization</td>
<td>Page fault rate, Effective access time</td>
<td>Running next</td>
</tr>
</tbody>
</table>

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the 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
- Hardware manages cache, not OS or user
- Protection of memory required to ensure correct operation

Simple

- Simplest task we need to tackle.
  - How to keep processes’ memory areas separate.

  First attempt:
  - Define a start location and length of the memory footprint of a process.

  Machine can at least make sure the processes don’t wander outside the range this defines.

Base and Limit Registers

- A pair of base and limit registers define the logical address space

Base: 0
Limit: 12,000

Logical Address: 3000:0

Physical Address: 2980000
Hardware address protection

Process: “Where am I?”

- How does the process get assigned to location in memory?
- How to know where code/data will be at runtime.
- Problem of defining how addresses are bound to code/data.

 Multistep Processing of a User Program

Binding of Instructions and Data to Memory

- 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)

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

- 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
- Different from dynamic linking

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
- System also known as shared libraries

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

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

**OS Memory Mgmt Scheme #1: Contiguous Allocation**

- Main memory usually 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**

- 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

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

**Contiguous Allocation (Cont)**

- Multiple-partition allocation
  - Hole – block of available memory; holes of 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)

<table>
<thead>
<tr>
<th>OS process 5</th>
<th>OS process 5</th>
<th>OS process 5</th>
<th>OS process 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>process 5</td>
<td>process 5</td>
<td>process 9</td>
<td>process 9</td>
</tr>
<tr>
<td>process 2</td>
<td>process 2</td>
<td>process 2</td>
<td>process 2</td>
</tr>
</tbody>
</table>
Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

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

Approach to memory management that eliminates external fragmentation:
- Does not eliminate internal fragmentation.
- Some processes will consume more memory than they need. Holes have well defined bounds due to fixed page size though.
- This approach is used all over the place. So popular, hardware support exists in modern CPUs to support it.

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:

  \[
  \text{Physical address} = \text{Page number} \times \text{Page size} + \text{Page offset}
  \]

  
  \[
  n \quad m - n
  \]

  
  For given logical address space \(2^m\) and page size \(2^n\).
Paging Hardware

Paging Model of Logical and Physical Memory

Paging Example

Free Frames

Implementation of Page Table

Hardware to the rescue

- 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).
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry—uniquely identifies each process to provide address-space protection for that process.
TLBs

- Think of TLBs like a specialized cache for page tables.
- First, try to look up a page in the TLB.
  - Very fast if there is a hit — on die, register or L1 access speeds
- If not in TLB, miss and go out to main memory to get the entry from the page table.
  - Store result in TLB.
  - Temporal and spatial locality says that we are likely to reuse this later and have a hit.
  - Amortizes slow memory access in miss over multiple hits.

Associative Memory

- Associative memory – parallel search

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

TLBs

- Pentium
- Note layout and proximity of TLB to both caches and bus interface to main memory.

Effective Access Time

- Associative Lookup = $\epsilon$ 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)
  \[
  EAT = (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha) = 2 + \epsilon - \alpha
  \]

  *Hit:* time to access memory since we know address

  *Miss:* time to access page table in memory, then time to access what we wanted in the first place

Example

- 70% probability of a hit
- ~100 ns memory read
- ~5 ns TLB lookup
- \[(5 + 100)*0.7 + (5 + 200)*0.3 = 135\) ns EAT

  Not bad. Better than having to go out to memory every time to look up page table entries.

  *(Timings based on approximate Xeon ratings)*
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

Also possible to accomplish with page table length register: more memory efficient.

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

Shared Pages Example

Protection issues

- Shared pages impact multiple processes.
- OS can tag pages as read-only when dynamic libraries are loaded.
  - Shared library support is in the OS, not user space.
- Allows OS to ensure that processes do not make bad changes to shared pages.

Structure of the 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

- 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:

  \[
  \begin{array}{c|c|c}
  \text{page number} & \text{page offset} \\
  \hline
  p_1 & p_2 & d \\
  \end{array}
  \]

  where \( p_1 \) is an index into the outer page table, and \( p_2 \) is the displacement within the page of the outer page 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>

Hierarchy and TLBs

- 2008 whitepaper on Nehalem from Intel ("First the Tick, Now the Tock")
- One big update is the introduction of a two-level TLB hierarchy
  - Adds a 512 entry second level TLB
- This is an interesting look at current trends in CPU architectures.
  - Clearly paging is very important, and TLB assistance critical for performance if they're advertising it as a major new feature of year '08 architectures.
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

OS Memory Mgmt Scheme #3: 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

Segmentation Architecture

- 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 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 < 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
  - A segmentation example is shown in the following diagram

Segmentation Hardware

Example of Segmentation

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

<table>
<thead>
<tr>
<th>CPU</th>
<th>logical address</th>
<th>segmentation unit</th>
<th>linear address</th>
<th>paging unit</th>
<th>physical address</th>
<th>physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intel Pentium Segmentation

logical address

<table>
<thead>
<tr>
<th>selector</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Segment descriptor

32-bit linear address

Intel Nehalem Example

Instruction TLB

Data TLB

Pentium Paging Architecture

<table>
<thead>
<tr>
<th>logical address</th>
<th>page directory</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page directory</td>
<td>page table</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4-kB page</td>
<td></td>
</tr>
<tr>
<td>CPD register</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linear Address in Linux

Broken into four parts:

- global directory
- middle directory
- page table
- offset

Three-level Paging in Linux

<table>
<thead>
<tr>
<th>linear address</th>
<th>page directory</th>
<th>middle directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>global directory</td>
<td></td>
<td>middle directory</td>
<td>page table</td>
<td></td>
</tr>
<tr>
<td>global directory entry</td>
<td></td>
<td>middle directory entry</td>
<td>page table entry</td>
<td></td>
</tr>
<tr>
<td>CPD register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
End of Chapter 8