Computer Architecture II
Memory Management Units

Syed Asad Alam
School of Computer Science and Statistics

Introduction

Reference Book
Operating System Concepts, A. Silberschatz, P. B. Galvin, G. Gagne
- Chapters 8 and 9
- Available on Blackboard
Introduction

- Memory → Central to operation of modern computer system
- Typical instruction cycle
  - Instruction fetch from memory
  - Instruction decode
  - Operands fetched from memory
  - Execution
  - Results stored on memory
- What does the memory unit see?
  - Stream of memory addresses

Role of MMU

- Translating addresses for instructions and operands
- Allocating memory for system and user processes
- Protecting memory space for system and users
- Sharing memory between user processes
- Cache

What is a Process?

- Program in execution → An active entity
Multiple Processes

- Protect OS from user processes (even valid for systems executing single user process)
- Separate per-process memory space

Hardware Address Protection

User Program Processing
**Address Binding**

- Systems load a user program in memory
- Symbolic addresses (for e.g., variables) in a program bound to relocatable (relative) addresses by compiler
- Relocatable addresses bound to absolute addresses by linker
- Three places where these bindings can take place
  - Compile time → Absolute code (Requires recompilation)
  - Load time → Compiler generates relocatable code (Requires reloading)
  - Execution time → Process can be moved, special hardware needed (Most common today)
- Dynamic relocation requires separating address generated by a processor and the one seen by the memory
  - Address generated by processor → Logical or Virtual address
  - Address seen by the memory → Physical address
- Address mapping/translation handled by MMU

**Basic Hardware for Separate Address Spaces**

**Dynamic Loading**

- Programs can be large
- If the whole program is to be loaded as a process → Limits the number of active processes in memory
- Not all parts of the program used with the same frequency → Subroutines to handle error or the occasional dumping of data on standard output
- Keep subroutines on disk in a relocatable load format
- Main program loaded into memory and other routines called when needed (A process being divided into multiple small processes)
- Referred to as → Dynamic Loading
- User dependent
- Related concepts → Dynamic linking and shared libraries
Allocation of Memory to Processes

- Multiple user processes can reside in memory
- Processes can be moved from disk to memory when ready or moved back when not needed
- Need to allocate available memory to processes
- Two ways:
  - Contiguous memory allocation
  - Non-contiguous memory allocation

Contiguous Memory Allocation

- Each process \(\rightarrow\) Single section of memory
- Section of memory \(\rightarrow\) Contiguous to the section for next process
- Two types\(^1\):
  - Fixed-partition method \(\rightarrow\) Internal fragmentation, limited process size and degree of multiprogramming
  - Variable-partition method \(\rightarrow\) External fragmentation, difficult to implement
- Used by IBM System/360 Operating System

Non-Contiguous Memory Allocation

- Memory allocated to various process not contiguous \(\rightarrow\) Allocate physical memory to processes where ever it is available
- Reduces memory wastage
- Involves address translation
- Two types
  - Segmentation
  - Paging
- Can be combined
Segmentation

- The programmer’s view of a program

<segment-number, offset>

Separate Segments

- An example of separate segments by a compiler:
  - The code
  - Global variables
  - Heap
  - Stack
  - Standard libraries
- 8086 to 80286 used a segmented memory model
Intro

Paging

Segmentation Example

Problems of Previous Approaches

- External fragmentation
- Need of compaction
- Fitting memory chunks of various sizes onto the backing store
Basic Method

- Physical memory → Fixed-sized frames
- Logical memory → Fixed-size pages (A process is made up of multiple pages)
- Size of frames = Size of pages
- Backing store → Fixed-sized blocks
- Size of blocks = n× Sizes of frames

Advantages/Disadvantages

- Separate logical and physical address space
- Large logical address space (greater than physical address space)
- No external segmentation (but possible internal fragmentation)
- Easy swapping of pages/frames
- Requires a page table to translate logical into physical address, eats up memory

Paging Hardware
Mapping Between Process, Page and Memory

Simple Example

Non-Contiguous Allocation
Internal Fragmentation

Assume:
- Page size: 2,048 bytes
- Process: 72,766 bytes
- How many pages? → 35 pages plus 1,086 bytes
- How many frames? 36 frames
- Worst case: n pages +1 byte → n + 1 frames

Page Size

- Avg. internal fragmentation → One-half page per process
- Consequence?
- Small pages sizes desirable
- Results in higher overhead for each PTE (page table entry)
- Typical page sizes → 4KB – 8KB

IA32:
- \(2^{20}(4\text{GB}) \rightarrow 2^{20}(\text{pages}) \times 2^{12}(4\text{KB})\) pages

Page sizes among architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Smallest page size</th>
<th>Larger page sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit x86</td>
<td>4 KB</td>
<td>4 MB in PSE mode, 2 MB in PME mode[28]</td>
</tr>
<tr>
<td>x86-64 [29]</td>
<td>4 KB</td>
<td>2 MB, 1 GB (only when the CPU has PDPES flag)</td>
</tr>
<tr>
<td>IA-64 [30] [31]</td>
<td>4 KB</td>
<td>8 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 256 MB[32]</td>
</tr>
<tr>
<td>Power ISA[33]</td>
<td>4 KB</td>
<td>64 KB, 16 MB, 16 GB</td>
</tr>
<tr>
<td>SPARC V9 with SPARC Reference Model[34]</td>
<td>4 KB</td>
<td>256 KB, 16 MB</td>
</tr>
<tr>
<td>UltraSPARC</td>
<td>4 KB</td>
<td>64 KB, 512 KB (optional), 4 MB, 32 MB (optional), 256 MB (optional), 2 GB (optional), 16 GB (optional)</td>
</tr>
<tr>
<td>AXP64[35]</td>
<td>4 KB</td>
<td>64 KB, 1 MB (&quot;section&quot;), 16 MB (&quot;supersection&quot;) (defined by a particular implementation)</td>
</tr>
</tbody>
</table>

4 en.wikipedia.org
IA32 Virtual Address Space and Physical Memory

- Each process → 4GB virtual address space
- Not fully utilized but divided into pages
- Programmers view → Memory as single space (not aware of other processes in memory)
- Actual view → User program scattered throughout physical memory
- Pages mapped by MMU to real physical frames in memory
- Virtual pages may be:
  - Not allocated/mapped
  - Allocated in physical memory
  - Allocated on paging disk

Typical Windows 7 process memory usage: Word (43 MB), Firefox (27 MB)...

Memory Cruncher

- Consider the following program outline

```c
#define GB (1024*1024*1024)
char *p = malloc(4*GB); // just moves internal OS pointer
for (size_t i = 0; i < 4*GB; i += PAGESIZE, p += PAGESIZE)
    *p = 0; // access causes physical memory to be allocated
```

- A more complete version of memorycruncher.cpp and memorycruncher_gcc.cpp is on the CSU34021 website and Blackboard
- Designed to run as a x86 or x64 process
- size_t is the size of an address [x86 32 bits, x64 64 bits]
- PAGESIZE is 4KB

Memory Cruncher – Windows

- What is the largest contiguous memory block that can be allocated?
- Windows x86
  - 4GB virtual address space, bottom 2GB for user and top 2GB for OS
  - malloc() allocates approx 1371 MB contiguous memory block
  - To enable more → Right click on project name
    [Properties][linker][System][EnableLargeAddresses] to allow malloc() to allocate approx 2032 contiguous memory block
- Windows x64
  - A much larger contiguous memory block
  - Allocating a block much greater than size of physical memory [16GB] results in PC becoming extremely unresponsive [had to reboot by turning off power]
  - RUN with caution
Memory Cruncher – Windows

x86

```
```

x86 with large addresses

```
```

Memory Cruncher – Windows

x64

```
```

Memory Cruncher – Linux

- x86 → Default large address enable
- Compile with → gcc -o mc memorycruncher_gcc.cpp -m32
- Execute with → ./mc

```
```
Page Tables

- Total PTEs → $2^{20} = 1,048,576$
- Each PTE → 32-bits or 4-bytes
- Total for each PT → 4 MB
- Impractical for modern OS with many processes running at the same time

Hierarchical Paging

- To reduce the size of page table structure → Hierarchical Paging or N-level page table

Address Translation

- Logical address
- Page number
- Page offset
- P1
- P2
- P3
How Does it Work?

- Page table base register (PTB) → Primary/outer page table
- A Valid PTE → Secondary/inner page/page of page table
- Size of page table → 4KB

Generic MMU Operation

- When MMU accesses a PTE it checks the Valid bit
- If accessing a primary PTE and V == 0 (i.e. PTE is invalid)
  - No physical memory allocated for corresponding secondary page table
- If accessing a secondary PTE and V == 0
  - No physical memory allocated for referenced page [i.e. virtual address NOT mapped to physical memory]
- In both cases a “page fault” occurs, the instruction is aborted and the MMU interrupts the CPU

Page Fault Handling

- OS must resolve page fault by performing one OR more of the following actions:
  - Allocating a page of physical memory for use as a secondary page table [from an OS maintained list of free memory pages]
  - Allocating a page of physical memory for the referenced page
  - Updating the associated page table entry/entries
  - Reading code or initialized data from disk to initialize the page contents [context switches to another process while waiting]
  - Signaling an access violation [e.g. writing to a read-only code page]
  - Restarting [or continuing] the faulting instruction
**x86 Architecture**

- x86 → Segmentation + Paging
- Segment size → Up to 4GB
- Number of segments per process → Up to 16K
- Logical address space → 2 partitions
  - Private → Up to 8K segments (local descriptor table)
  - Shared → Up to 8K segments (global descriptor table)
  - Each entry in LDT/GDT → 8 bytes
- 6 segment registers
- 6 8-byte microprogram registers → LDT/GDT

**x86 Architecture – Segmentation**

- Page size → 4KB/4MB
- Two level paging scheme
  - $p_1$ → Index into the primary/outermost table (page directory)
  - $p_2$ → Index into the secondary/innermost table (page table)
  - $d$ → Address offset
x86 Architecture – PAE

- PAE → Page address extension
- Physical address space → Larger than 4GB
- Three level paging scheme
- PDE/PTE → 64-bits
- Total address width → 36-bits (64GB physical memory)
- Not supported by 32-bit Windows desktop operating system but supported by Linux/Mac
- Process → 4GB

x64

- 48-bit virtual address space
- Page sizes → 4KB, 2MB, 1GB
- Four levels of page hierarchy
- PAE → 52-bit physical address (4096TB)
More About Paging

Shared Pages

- Advantage of paging \(\rightarrow\) Sharing common code
- How can we share code?
- Re-entrant code \(\rightarrow\) Reusable routine that can be invoked, interrupted and re-invoked with unique data
- Example \(\rightarrow\) Text editor
- Advantage?

Assume a system:
- 40 users
- Text editor \(\rightarrow\) 150KB of code and 50KB of data space
- Total \(\rightarrow\) 8,000KB
- Advantage of sharing \(\rightarrow\) 150KB + 50x40KB = 2,150KB
Hardware Support for Paging

- Simple case → Page table implemented as a set of dedicated registers
- Example → DEC PDP-11
  - Address space → 16-bits
  - Page size → 8KB
  - Page table → 8 entries
- Limitation → Only feasible if page table reasonably small
- Solution → PTBR (CR3) with page tables in memory

Translation Look-Aside Buffer (TLB)

- Advantage of PTBR → Changing page table require changing one register
- Disadvantage of PTBR → Time required to access a user memory location
  - Two memory access required to access a byte → One for page table entry and one for the byte
- Solution → Special, small and fast lookup hardware cache → Translation look-aside buffer (TLB)

What is TLB?

- Associative, high-speed, m-entry memory
- Provides direct mappings for the m most recently accessed virtual pages
- Each entry → Two parts
  - Key/Tag → Page number
  - Value → Frame number
- General operation
  - Virtual address presented to TLB → Page number matched against all entries
  - If an entry matches → Read the frame number (TLB Hit)
  - If not → Memory reference to the page table → Page walk (TLB Miss) and add entry to TLB (replace based on LRU or other mechanisms)
  - Page walk → Hardware state machines (x86/x64) or Interrupts/ordinary instructions (RISCs)
- TLB lookup part of instruction pipeline → No performance penalty
- Separate instruction and data address TLBs
- TLBs typically small → Between 32 and 1,024 entries
Basic Hardware for TLB

![Diagram of TLB](image)

Context/Process Switch

- What happens on a context/process switch?
  - TLBs looked up by virtual addresses
  - Same virtual address for multiple processes
  - Cannot allow new process to access memory of old process
- Solution:
  - Flush TLB → Update CR3 (80386)
  - Invalidate individual entries → INVLPG
  - Associate a PID with TLB entry → Address-space identifiers (ASID)
  - Address space protection
  - Sharing of TLB between processes
  - PCID (Intel), ASID (ARM)

Example: Pentium-M

- Four different TLBs
  - Instruction TLB for 4K pages
    - 128 entries, 4-way set associative
  - Instruction TLB for large pages
    - 2 entries, fully associative
  - Data TLB for 4K pages
    - 128 entries, 4-way set associative
  - Data TLB for large pages
    - 8 entries, 4-way set associative
  - LRU replacement policy
You are now able to:
- Explain the concept and benefits of virtual memory
- Identify and explain different memory allocation techniques to processes, like segmentation and paging
- Explain different aspects of paging like different page sizes, multi-level address translation and shared pages
- Explain the operation of a TLB