Chapter 8
Main Memory

Zhi Wang
Florida State University
Contents

- Background
- Swapping
- Contiguous memory allocation
- Paging
- Segmentation
- OS examples
Objectives

• To provide a description of various ways to organize memory hardware

• To discuss various memory-management techniques, including **paging** and **segmentation**

• To provide a detailed description of the Intel Pentium
Background

- Program must be brought into memory for it to be run
- **Main memory** and **registers** are only storage CPU can directly access
  - register access in one CPU clock (or less)
  - main memory can take hundreds of cycles
  - cache sits between main memory and CPU registers
- Protection of memory is required to ensure correct operation
  - kernel/user space, between processes…
  - it can be based on **base + limit** registers
Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space.
Memory Protection with Base and Limit

CPU

address

≥

no

trap to operating system monitor—addressing error

base

yes

memory

base + limit

<

no
Address Binding

- Inconvenient to have first process address always at 0 (why?)
  - (shared) libraries, *nil* pointer dereference detection…

- Addresses are represented in different ways at different stages of a program’s life
  - **source code** addresses are usually **symbolic** (e.g., temp)
  - **compiler** binds symbols to **relocatable addresses**
    - e.g., “14 bytes from beginning of this module”
  - **linker** (or loader) binds relocatable addresses to **absolute addresses**
    - e.g., 0x0e74014
Address Binding

- Binding of instructions and data to memory can happen at three stages:
  - **compile time:**
    - if memory location is known, compiler can generate absolute code
    - usually used in the kernel and boot loader
    - must recompile code if starting location changes
  - **load time:**
    - the loader loads the program at some location, and fixes it
  - **execution time:**
    - binding is delayed until run time (demand paging)
    - code/data may be moved from one location to another (swapper)
Multi-step Processing of a User Program

1. Source program
2. Compiler or assembler
3. Object module
4. Linkage editor
5. Load module
6. Loader
7. In-memory binary memory image
8. Dynamic linking
9. System library
10. Dynamically loaded system library

Compile time: compile time
Load time: load time
Execution time (run time)
Logical vs. Physical Address Space

- **Logical/physical address separation** is central to memory management
  - **logical address** – address generated by the software
    - also referred to as virtual address
    - logical address space is all logical addresses generated by a program
  - **physical address** – address seen by the memory unit
    - physical address space is all physical addresses generated by a program
  - OS determines logical to physical address mapping using **MMU**
    - a logical address can be mapped to different physical address for different processes or different time of a process (swapping)
    - a block of physical memory can be mapped into many logical addresses (shared memory)
Memory-Management Unit

- MMU is hardware that maps **logical address** to **physical address** at run time
  - the user program deals with logical (virtual) addresses
    - it never sees the real physical addresses
  - MMU converts virtual address to physical address
    - CPU accesses memory using physical address
- Many different types of MMU
  - **segmentation** and **paging** are two typical types of MMU
  - e.g., MS-DOS on Intel 80x86 used 4 segments
    - memory allocated contiguously, segments used to protect OS
Contiguous Allocation

- A simple memory management scheme: contiguous allocation
- Main memory is partitioned into contiguous blocks
  - OS kernel residents in the low memory, user processes in high memory
  - each process is contained in **single contiguous section of memory**
- **Relocation registers** used to protect the kernel and user processes
  - **base register**: the smallest address for the partition
  - **limit register**: the maximum allowed offset
- a process cannot access other processes or the kernel
Relocation Register

- CPU
- logical address
- limit register: yes
- relocation register
- physical address
- memory
- trap: addressing error
Contiguous Allocation

- Multiple-partition contiguous allocation to support **multiprogramming**
  - degree of multiprogramming limited by number of partitions
- Kernel keeps track of free and allocated memory
  - when a process arrives, it is allocated memory from free memory large enough to accommodate it
  - process exiting frees its partition, adjacent free partitions are combined
Free Memory Allocation

- How to satisfy a request of size $n$ from a list of free memory blocks?
  - **first-fit**: allocate the first block that is big enough
  - **best-fit**: allocate the smallest block 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** is big problem for all three methods
  - first-fit and best-fit usually performs better than worst-fit
Fragmentation

- **External fragmentation**
  - unusable memory between allocated memory blocks
    - total amount of free memory space is larger than a request
    - the request cannot be fulfilled because the free memory is not contiguous
  - external fragmentation can be reduced by **compaction**
    - shuffle memory contents to place all free memory in one large block
    - program needs to be relocatable at runtime

- **Internal fragmentation**
  - memory allocated may be larger than the requested size
  - this size difference is memory *internal to a partition*, but not being used
  - Sophisticated algorithms are designed to avoid fragmentation
    - none of the first-/best-/worst-fit can be considered sophisticated
Paging

- Paging maps logical address space to physical spaces in **pages**
  - divide physical memory into fixed-sized blocks called **frames**
    - frame/page size is a power of 2, between 512 bytes to 16 Mbytes
    - kernel keeps track of all free frames
  - divide logical memory into blocks of the same size called **pages**
    - page table maps logical pages to physical frames
      - logical address space is contiguous (visible to program)
      - physical address space could be non-contiguous (visible to hardware)
      - each process has its own page table
- Does paging have external/internal fragmentation problems?
Paging: Address Translation

- A logical address is divided into:
  - **page number** \(p\)
    - used as an index into a page table
    - page table entry contains the corresponding *physical* frame number
  - **page offset** \(d\)
    - offset within the page/frame
    - combined with frame number to get the physical address

\[
\begin{array}{c|c}
\text{page number} & \text{page offset} \\
\hline
p & d \\
\hline
m - n \text{ bits} & n \text{ bits}
\end{array}
\]

\(m\) bit logical address space, \(n\) bit page size
Paging Hardware
Paging Example

logical memory

page 0
page 1
page 2
page 3

page table

0 1
1 4
2 3
3 7

frame number

0 page 0
1
2 page 2
3
4 page 1
5
6
7 page 3

physical memory
$m = 4$ and $n = 2$  32-byte memory and 4-byte pages
Free Frames

Before allocation

After allocation
Paging: Internal Fragmentation

- Paging has **no external fragmentation**, but **internal fragmentation**
  - e.g., page size: 2,048, program size: 72,766 (35 pages + 1,086 bytes)
    - internal fragmentation: 2,048 - 1,086 = 962
  - worst case internal fragmentation: **1 frame – 1 byte**
    - average internal fragmentation: 1 / 2 frame size
- Small frame sizes more desirable than large frame size?
  - memory becomes larger, and page table takes memory
  - page sizes actually grow over time
    - 4KB ➔ 2MB ➔ 4MB ➔ 1GB ➔ 2GB
    - why we need 2GB frames?
One-level Page Table

- One big page table maps logical address to physical address
  - the page table should be kept in main memory
- **page-table base register** (PTBR) points to the page table
  - does PTBR contain physical or logical address?
- **page-table length register** (PTLR) indicates the size of the page table
- Every data/instruction access requires **two memory accesses**
  - one for the page table and one for the data / instruction
  - CPU can cache the translation to avoid one memory access (**TLB**)
TLB

- TLB (translation look-aside buffer) caches the address translation
  - if page number is in the TLB, no need to access the page table
  - if page number is not in the TLB, need to replace one TLB entry
- TLB usually use a fast-lookup hardware cache called **associative memory**
- TLB is usually small, 64 to 1024 entries

- TLB and context switch
  - each process has its own page table
    - switching process needs to switch page table
  - **TLB must be consistent with page table**
    - flush TLB at every context switch, or,
      - tag TLB entries with **address-space identifier** (ASID) that uniquely identifies a process
  - some TLB entries can be **shared** by processes, and fixed in the TLB
    - e.g., TLB entries for the kernel
Associative Memory

- Associative memory: memory that supports parallel search
- Associative memory is not addressed by “addresses”, but contents
  - if p is in associative memory’s key, return frame# (value) directly
  - think of hash tables
Paging Hardware With TLB

The diagram illustrates the process of accessing physical memory through the logical address space. The CPU provides a logical address, which is then broken down into page and frame numbers. These numbers are used to look up the page and frame in the TLB (Translation Lookaside Buffer). If a TLB hit occurs, the physical address is directly provided from the TLB to physical memory. However, if a TLB miss occurs, the page number and frame number are used to access the page table, which provides the physical address.
Effective Access Time

• Assume TLB lookup takes $\varepsilon$ time unit with a hit ratio of $\alpha$
  • $\varepsilon$ can be < 10% of memory access time
  • hit ratio – percentage of page translation that is found in the TLB

• Effective access time (EAT): $(m + \varepsilon) \alpha + (2m + \varepsilon)(1 - \alpha)$
  • $m$ is main memory access time
  • assume TLB and page table access is not parallel

• e.g., $\alpha = 80\%$, $\varepsilon = 20\text{ns}$, $m = 100\text{ns}$: $\text{EAT} = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$
• e.g., $\alpha = 98\%$, $\varepsilon = 20\text{ns}$, $m = 140\text{ns}$: $\text{EAT} = 0.98 \times 160 + 0.02 \times 300 = 162.8\text{ns}$
Memory Protection

- Each page table entry has a **present** (aka. valid) bit
  - present: the page has a valid physical frame, thus can be accessed
- Each page table entry contains some protection bits
  - **kernel/user**, **read/write**, **execution**?, **kernel-execution**?
  - why do we need them?
- Any violations of memory protection result in a trap to the kernel
Memory Protection

- Page table:
  - Frames 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
  - Pages 0, 1, 2, 3, 4, 5, 6, 7, 8, ...
  - Frame number:
    - 0: Frame 2, Valid
    - 1: Frame 3, Valid
    - 2: Frame 4, Valid
    - 3: Frame 7, Valid
    - 4: Frame 8, Valid
    - 5: Frame 9, Valid
    - 6: Frame 0, Invalid
    - 7: Frame 0, Invalid

- Page table structure:
  - 00000
    - page 0
    - page 1
    - page 2
    - page 3
    - page 4
    - page 5
    - 10,468
    - 12,287
Page Sharing

- Paging allows to share memory between processes
  - e.g., one copy of **code** shared by **all processes of the same program**
    - text editors, compilers, browser..
  - shared memory can be used for **inter-process communication**
  - shared libraries
- Each process can, of course, have its private code and data
Page Sharing

process $P_1$

page table for $P_1$

ed 1
ed 2
ed 3

ed 1
ed 2
ed 3
data 2

process $P_2$

page table for $P_2$

ed 1
ed 2
ed 3
data 2

process $P_3$

page table for $P_3$

ed 1
ed 2
ed 3
data 3

0
1
data 1
2
data 3
3
ed 1
4
ed 2
5
6
ed 3
7
data 2
8
9
10
11
Page Table

- One-level page table can consume lots of memory for page table
  - e.g., 32-bit logical address space and 4KB page size
    - page table would have 1 million entries ($2^{32} / 2^{12}$)
    - if each entry is 4 bytes $\rightarrow$ 4 MB of memory for page table alone
  - each process requires its own page table
  - page table must be **physically contiguous**
- To reduce memory consumption of page tables:
  - **hierarchical page table**
  - **hashed page table**
  - **inverted page table**
Hierarchical Page Tables

- Break up the logical address space into **multiple-level** of page tables
  - e.g., two-level page table
  - first-level page table contains the frame# for second-level page tables
    - “page” the page table
- Why hierarchical page table can save memory for page table?
Two-Level Page Table
Two-Level Paging

• A logical address is divided into:
  • a page directory number (first level page table)
  • a page table number (2nd level page table)
  • a page offset

• Example: 2-level paging in 32-bit Intel CPUs
  • 32-bit address space, 4KB page size
  • 10-bit page directory number, 10-bit page table number
  • each page table entry is 4 bytes, one frame contains 1024 entries ($2^{10}$)
Address-Translation Scheme

logical address
\[ p_1 \ p_2 \ d \]

\[ p_1 \quad \text{outer page table} \]

\[ p_2 \quad \text{page of page table} \]

\[ d \]
64-bit Logical Address Space

- 64-bit logical address space requires more levels of paging
  - two-level paging is not sufficient for 64-bit logical address space
    - if page size is 4 KB ($2^{12}$), outer page table has $2^{42}$ entries, inner page tables have $2^{10}$ 4-byte entries
  - one solution is to add more levels of page tables
    - e.g., three levels of paging: 1st level page table is $2^{34}$ bytes in size
    - and possibly 4 memory access to get to one physical memory location
- usually not support full 64-bit virtual address space
  - AMD-64 supports 48-bit
    - canonical form: 48 through 63 of valid virtual address must be copies of bit 47
64-bit Logical Address Space

<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

- In hashed page table, virtual page# is hashed into a frame#
  - the page table contains a chain of elements hashing to the same location
  - each element contains: page#, frame#, and a pointer to the next element
    - virtual page numbers are compared in this chain searching for a match
    - if a match is found, the corresponding frame# is returned
- Hashed page table is common in address spaces > 32 bits
Hashed Page Table
Inverted Page Table

- Inverted page table tracks allocation of physical frame to a process
  - one entry for each physical frame → fixed amount of memory for page table
  - each entry has the **process id** and the **page#** (virtual address)
- Sounds like a brilliant idea?
  - to translate a virtual address, it is necessary to search the (whole) page table
    - can use TLB to accelerate access, TLB miss could be very expensive
  - how to implement shared memory?
    - a physical frame can only be mapped into one process!
Inverted Page Table
Segmentation

- Segmentation supports user view of a program
  - a program is a collection of segments
    - main program
    - function
    - local variables, global variables
    - stack
  - each segment can be mapped to physical blocks
User’s View of a Program

- subroutine
- stack
- symbol table
- Sqrt
- main program

logical address
Segmentation

user space

physical memory space
Segmentation

• In segmentation: a **logical** address consists of a tuple `<segment#, offset>`,

• Segment table maps segments to physical memory
  • each segment table entry has:
    • **base**: the starting physical address where the segments reside in memory
    • **limit**: the maximum offset of the segment
    • **memory protection** bits: present/read/write/execution
  • segment-table base register (STBR) points to the segment table
  • segment-table length register (STLR) indicates number of segments
Segmentation Hardware

CPU \rightarrow s, d \rightarrow s \rightarrow segment table \rightarrow yes/no

- yes: \rightarrow + \rightarrow physical memory
- no: \rightarrow trap: addressing error
Example of Segmentation
Swapping

- **Swapping** extends physical memory with backing disks
  - a process can be swapped temporarily out of memory to a backing store
    - backing store is usually a (fast) disk
  - the process will be brought back into memory for continued execution
    - does the process need to be swapped back in to same physical address?
- Swapping is usually only initiated under memory pressure
- Context switch time can become very high due to swapping
  - if the next process to be run is not in memory, need to swap it in
  - disk I/O has high latency
Swapping

1. swap out
2. swap in
Example: Intel Pentium

- Pentium supports both **segmentation** and **segmentation with paging**
  - each segment can be 4 GB
  - two segment tables, each can contain 8K entries
    - local descriptor table (LDT): per process
    - global descriptor table (GDT): shared
- Three address types:
  - CPU generates **logical address**: segment selector + offset
    - segment selector: index into LDT/GDT
    - segmentation unit converts logical address to **linear address**
  - paging unit converts linear address to **physical address**
    - pages sizes can be 4 KB, 2MB, 4 MB
Intel Pentium

![Diagram of address translation]

<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>

Logical address → Segmentation unit → Linear address → Paging unit → Physical address → Physical memory
Intel Pentium

Diagram:
- Logical address
- Selector
- Offset
- Descriptor table
- Segment descriptor
- 32-bit linear address
Intel Pentium: Paging

(logical address)

31  22  21  12  11  0

page directory  page table  offset

CR3 register

page directory  4-MB page

4-KB page

Diagram showing the paging structure in an Intel Pentium processor.
Linux Support for Intel Pentium

- Linux uses only 6 segments
  - kernel code, kernel data, user code, user data
  - task-state segment (TSS), default LDT segment
- Linux only uses two of four possible modes
  - kernel: ring 0, user space: ring 3
- Uses a generic four-level paging for 32-bit and 64-bit systems
  - for two-level paging, middle and upper directories are omitted
  - older kernels have three-level generic paging
Three-level Paging in Linux
End of Chapter 8