• HW#4 will be posted today.

• Exercise # 4 will be posted on Piazza today.
Learning Objectives

• Essential **requirement** of memory management system

• Tradeoffs between **fixed partitioning** and **dynamic partitioning**

• Calculate the placement in different algorithms for dynamic partitioning

• Explain why paging is desired to overcome limitations of fixed and dynamic partitioning

• Calculate **sizes** of page table, page table entries, etc. based on the given system description

• Identify the difference between **paging** and **segmentation**

• Identify **advantages** of paging over segmentation and vice versa
Memory Management

• Uniprogramming systems divide memory into OS area (kernel) and user area (running process)

• In a multiprogramming system, user memory is subdivided per process

• (That) memory needs to be allocated efficiently to pack as many processes into memory as possible

[maximize memory utilization]
Background

Main Memory

Program image in memory

Operating System

Process

Main Memory

Cache

CPU

Registers

Disk

Instructions

Data

Background
Need for Memory Management

- OS and hardware need to handle multiple processes in main memory

- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle

- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible

- In many systems, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes
Memory Management Requirements

- Memory management is intended to satisfy the following requirements:
  - Relocation
  - Protection
  - Sharing
  - Logical organization
  - Physical organization
Usually, a program resides on a disk as a binary executable file.
Relocation (II)

Where do addresses come from?

- **Compile time:** The compiler generates binary code and the exact physical location in memory starting from some fixed starting position (i.e., relative address). The OS does nothing. 
  
  \[ \text{[g++ mycode.cpp -o mycode.exe]} \]

- **Load time:** the OS loads the process and determines the process’ starting position. Once the process loads, it is supposed to not move in memory
  
  \[ \text{[./mycode.exe]} \]

- **Execution time:** CPU generates an address, and OS can move it around in memory as needed
Relocation (III)

- Programmers typically do **not know in advance** which other programs will be resident in main memory at the time of execution of their program

- Active processes need to be able to be **swapped** in and out of main memory in order to maximize processor utilization
  - [processes move around in main memory]

- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
  - may need to **relocate** the process to a different area of memory

**Relocation:** The ability of OS to execute processes **independently from their physical location** (after loading) in memory
Protection

- Each process should be protected against unwanted interference by other processes.
- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable (?)
- Memory references generated by a process must be checked at run-time
- satisfaction of the relocation requirement increases the difficulty of satisfying the protection requirement. Because the location of a program in main memory is unpredictable, it is impossible to check absolute addresses at compile time to assure protection.
Sharing

- Advantageous to allow each process access to the same copy of the program rather than have their own separate copy.
- Protection must have the **flexibility** to allow several processes to access the same portion of main memory.
- Memory management must allow **controlled access to shared areas** of memory without compromising protection.
- Mechanisms used to support relocation support sharing capabilities.
Logical Organization

- Memory is organized as linear
  - Programs often composed as modules that can be written and compiled independently

1. Modules can be written and compiled independently
2. With modest additional overhead, different degrees of protection can be given to different modules (read only, execute only)
3. It is possible to introduce mechanisms by which modules can be shared among processes.

How OS can deal with user programs and data?

- **Segmentation** is the tool that most readily satisfies requirements
How Does a Program Start Running?
The flow of information between main and secondary memory:

Step 1) OS (loader) copies a program from permanent storage into RAM [the program (bits) from disk]

Step 2) CPU’s Program Counter is then set to the starting address of the program and the program begins execution

Question? what if the program is too big? Memory available for a program+data may be insufficient
Physical Organization (II)

- **One solution:** programmer breaks code into pieces that fit into main memory (RAM) [independent]
- Pieces, called *overlays*, are loaded and unloaded by the program
- Does not require OS help

**Overlaying allows various modules to be assigned the same region of memory**

What are some programming challenges associated with overlaying?
Physical Organization (III)

Impractical and undesirable for two reasons:
- Programmer does not know how much space will be available
- Wastes programmer time

Solution: Virtual Memory

Allows the computer to “fake” a program into believing that its memory space is *larger* than physical memory (RAM)

Two memory “spaces”:
- **Virtual memory**: what the “program” sees
- **Physical memory**: what the “program” runs in

Mapping between two memories is the responsibility of memory management module [independent of programmer]
• Goals of memory management
• Partitioning
• Relocation
  – Need for logical and physical addresses
• Paging
• Segmentation
Partitioning

- An early method of managing memory
  - Pre-virtual memory
  - Not used much now

- Will clarify the later discussion of virtual memory if we look first at partitioning
  - Virtual Memory has evolved from the partitioning methods

If you had a chunk of memory that could be used for processes, how would you divide it up?
Fixed Partitioning
Fixed Partitioning

- Equal-size partitions
  - any process whose size is less than or equal to the partition size can be loaded into an available partition

- The operating system can swap out a process if all partitions are full and no process is in the Ready or Running state

memory has size 64 M
8 M is taken by OS
Disadvantages

- A program may be too big to fit in a partition
  - *program* needs to be designed with the use of overlays

- Main memory utilization is *inefficient*
  - any program, regardless of size, occupies an entire partition
    - *internal fragmentation*
      - wasted space due to the block of data loaded being smaller than the partition
Unequal Size Partitions

- Using unequal size partitions helps lessen the problems
  - programs up to 16M can be accommodated without overlays
  - partitions smaller than 8M allow smaller programs to be accommodated with less internal fragmentation
Placement Algorithm with Partitions

How do you decide where to put processes in memory?

- Equal-size partitions
  - Because all partitions are of equal size, it does not matter which partition is used

- Unequal-size partitions
  - Can assign each process to the smallest partition within which it will fit
    - A queue for each partition
    - Processes are assigned in such a way as to minimize the wasted memory within a partition
    - Advantage: minimize internal fragmentation
Memory Assignment

✓ Suppose no processes with a size between 12 and 16M at a certain point in time

✓ The 16M partition will remain unused, even though some smaller process could have been assigned to it.

✓ Any idea?
Placement Algorithm with Partitions

Employ a **single queue** for all processes. When it is time to load a process into main memory, the **smallest available** partition that will hold the process is selected.

![Diagram showing memory assignment for fixed partitioning](image)

**Figure 7.3** Memory Assignment for Fixed Partitioning
Disadvantages

- **Limited Concurrency**: The number of partitions specified at system generation time limits the number of active processes in the system.

- Small jobs will not utilize partition space efficiently.

How can we improve on it?
Dynamic Partitioning
Dynamic Partitioning

- Partitions are of **variable length** and **number**
- Process is allocated **exactly** as much memory as it requires no more
- This technique was used by IBM’s mainframe operating system, OS/MVT
Effect of Dynamic Partitioning

Process 1 needs 20M of memory, 2 needs 14, 3 needs 18, and 4 needs 8M.

External Fragmentation:
A lot of small holes at the end!

Figure 7.4  The Effect of Dynamic Partitioning
Dynamic Partitioning:
External Fragmentation and Compaction

External Fragmentation

- memory becomes more and more fragmented
- memory utilization declines

Compaction

- technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- free memory is together in one block
- time consuming and wastes CPU time
Compaction
Compaction
Compaction
Compaction
Where should we place a process with 14 MB of memory?

- Compaction implies the need for a dynamic relocation capability. [That is, it must be possible to move a program from one region to another in main memory without invalidating the memory references in the program]
✓ Operating system must decide which free block to allocate to a process

✓ Because memory compaction is time consuming, the OS designer must be clever in deciding how to assign processes to memory

**Best-fit**
- chooses the block that is closest in size to the request

**First-fit**
- begins to scan memory from the beginning and chooses the first available block that is large enough

**Next-fit**
- begins to scan memory from the location of the last placement and chooses the next available block that is large enough
Dynamic Partitioning Placement Algorithm (I)

- **Best-fit algorithm**
  - Chooses the block that is closest in size to the request
  - Since smallest block is found for process, the smallest amount of fragmentation is left
Dynamic Partitioning Placement Algorithm (II)

• First-fit algorithm
  - Scans memory from the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block
Dynamic Partitioning Placement Algorithm (III)

• Next-fit
  
  – Scans memory from the location of the last placement
  
  – More often allocate a block of memory at the end of memory where the largest block is found
  
  – The largest block of memory is broken up into smaller blocks
  
  – Compaction is required to obtain a large block at the end of memory
    Memory compaction must be done more often
Memory Configuration Example

16 MB?

Which partition each would best fit, next-fit, and first-fit algorithm pick?

Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block
Due data of HW#4 is on **Sunday June 19**.

Due data of Exercise #4 is on **Monday (June 13)**.
Buddy System
Fixed Partitioning:

✓ Fixed number of running processes
✓ Space (memory) inefficiency

Dynamic Partitioning:

✓ More complex to maintain
✓ Overhead of compaction

Can we resolve these problems and propose a much more effective partitioning algorithm?
Buddy System

- Comprised of fixed and dynamic partitioning schemes
- Space available for allocation is treated as a single block [but partition size cannot be arbitrary, only power of 2]
- Memory blocks are available of size $2^K$ words, $L \leq K \leq U$, where
  - $2^L = \text{smallest size block that is allocated}$
  - $2^U = \text{largest size block that is allocated};$ generally $2^U$ is the size of the entire memory available for allocation
Buddy System

• An instance of dynamic partitioning

• Satisfies requests *in units sized as power of 2*

• Request rounded up to next highest power of 2

• When *smaller allocation* needed than is available, current chunk *split* into two buddies of *next-lower power of 2*

• Continue until appropriate sized chunk available

• To begin, the entire space is available for allocation
A Few Things to Remember

- Memorize these numbers if you do not ready know them:

  - 1K = \( 2^{10} \)
  - 1M = \( 2^{20} \)
  - 1G = \( 2^{30} \)
  - 1T = \( 2^{40} \)

- Note: 1K is not 1000 in the context of memory management
Observations

• There are exactly 2 blocks of size $2^9$ in a block of size $2^{10}$

• There are exactly 2 blocks of size $2^8$ in a block of size $2^9$

• If you want memory of certain size, say $2^i$, you can select a block of size (that is a power of 2) such that
  
  – It is large enough $2^i \geq 2^L$
  – It uses more space than it wastes.
Buddy Algorithm: Example (I)

Memory = 1 MB = 2^20 B = 2^10 KB = 2*2^9

- Process A needs memory 100 KB = 2^7
- Process B needs memory 240 KB = 2^8
- Process C needs memory 64 KB = 2^6
- Process D needs memory 256 KB = 2^8
- Process B removed
- Process A removed
- Process E needs memory 75 KB = 2^7
- Process C removed
- Process E removed
- Process D removed
We know:

1024

512 512

256 256

128 128

64 64

32 32
### Buddy System Example

<table>
<thead>
<tr>
<th>Request 100 K</th>
<th>A = 128K</th>
<th>128K</th>
<th>256K</th>
<th>512K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request 240 K</td>
<td>A = 128K</td>
<td>128K</td>
<td>B = 256K</td>
<td>512K</td>
</tr>
<tr>
<td>Request 64 K</td>
<td>A = 128K</td>
<td>c = 64K</td>
<td>64K</td>
<td>B = 256K</td>
</tr>
<tr>
<td>Request 256 K</td>
<td>A = 128K</td>
<td>c = 64K</td>
<td>64K</td>
<td>B = 256K</td>
</tr>
<tr>
<td>Release B</td>
<td>A = 128K</td>
<td>c = 64K</td>
<td>64K</td>
<td>256K</td>
</tr>
<tr>
<td>Release A</td>
<td>128K</td>
<td>c = 64K</td>
<td>64K</td>
<td>256K</td>
</tr>
<tr>
<td>Request 75 K</td>
<td>E = 128K</td>
<td>c = 64K</td>
<td>64K</td>
<td>256K</td>
</tr>
<tr>
<td>Release C</td>
<td>E = 128K</td>
<td>128K</td>
<td>256K</td>
<td>D = 256K</td>
</tr>
<tr>
<td>Release E</td>
<td>512K</td>
<td>D = 256K</td>
<td>256K</td>
<td></td>
</tr>
<tr>
<td>Release D</td>
<td>1M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.6 Example of Buddy System**
Figure 7.7 Tree Representation of Buddy System
Buddy Algorithm: Example (II)

512 KB of Memory

• Process A needs memory 45 KB in size
• Process B needs memory 70 KB in size
• Process C needs memory 50 KB in size
• Process D needs memory 90 KB in size

• Process C removed
• Process A removed
• Process B removed
• Process D removed
Example

512 KB of Memory (physically contiguous area)

Alloc A 45 KB
Alloc B 70 KB
Alloc C 50 KB
Alloc D 90 KB
Free C
Free A
Free B
Free D
Outline

• Goals of memory management
• Partitioning
• Relocation
• Need for logical and physical addresses
• Paging
• Segmentation
Relocation

• When program loaded into memory the actual (absolute) memory locations are determined

• A process may occupy different partitions which means different absolute memory locations during execution (from swapping)

• Compaction will also cause a program to occupy a different partition which means different absolute memory locations

The location of instructions and data are not fixed in memory and frequently changes

[change when process is swapped or shifted]
**Addresses**

**Logical**
- reference to a memory location independent of the current assignment of data to memory
- Translation must be made to physical address

**Relative**
- address is expressed as a location relative to some known point (a particular example of logical address)

**Physical or Absolute**
- actual location in main memory
## Program Addresses and Memory

### Program

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>12</td>
</tr>
<tr>
<td>Mov</td>
<td>8</td>
</tr>
<tr>
<td>…</td>
<td>4</td>
</tr>
<tr>
<td>Jump 8</td>
<td>0</td>
</tr>
</tbody>
</table>

### RAM

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>44</td>
</tr>
</tbody>
</table>

**Physical addresses of RAM:**
# Program Addresses and Memory

## Program 1
- **Add**: 12
- **Mov**: 8
- **...**: 4
- **Jump 8**: 0

## Program 2
- **Cmp**: 12
- **Sub**: 8
- **...**: 4
- **Jump 12**: 0

## RAM
- **Jump 8**: 0
- **Jump 12**: 0
- **Add**: 12
- **Mov**: 8
- **Sub**: 8
- **Cmp**: 12
- **...**: 4
- **...**: 4
- **...**: 4
- **...**: 4
- **...**: 4
- **...**: 4
- **...**: 4
- **...**: 4
- **40**
- **44**
- **36**
- **32**
- **28**
- **24**
- **20**
- **16**
- **12**
- **8**
- **4**
- **0**

**Physical addresses of RAM**
We need logical address space concept, that is different than the physical RAM (main memory) addresses.

A program uses logical addresses.

Set of logical addresses used by the program is its logical address space
  - Logical address space can be, for example, [0, max_address]

Logical address space has to be mapped somewhere in physical memory
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.
Simple example of hardware translation of addresses

• When a process is assigned to the running state, a base register (in CPU) gets loaded with the **starting** physical address of the process

• A bound register gets loaded with the process's **ending** physical address

• When a **relative addresses** is encountered, it is **added** with the content of the **base register** to obtain the **physical address** which is compared with the content of the bound register

• This provides hardware protection: each process can only access memory within its process image
Address Translation:

starting address in memory for this program

Base register → Adder → Process control block
Bounds register → Comparator → Process image in main memory

Absolute address → Interrupt to operating system

ending location in memory for this program

Figure 7.8   Hardware Support for Relocation
Registers Used during Execution

• **Base** register
  – Starting address for the process

• **Bounds** register
  – Ending location of the process

• These values are set when the process is loaded or when the process is swapped in
Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address.

- The resulting address is compared with the value in the bounds register.

- If the address is not within bounds, an interrupt is generated to the operating system.
Logical and physical addresses

CPU

<table>
<thead>
<tr>
<th>base</th>
<th>limit</th>
<th>bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>32</td>
<td>56</td>
</tr>
</tbody>
</table>

PC

IR mov r1, M[28]

Main Memory (RAM)

MO[28+base]
MO[28+24] M[52]

int x
int y;
cmp ..
mov r1, M[28]
mov r2, M[24]
add r1, r2, r3
jmp 16
mov ..

logical addresses

28 int x
24 int y;
20 cmp ..
16 mov r1, M[28]
12 mov r2, M[24]
08 add r1, r2, r3
04 jmp 16
00 mov ..
Outline

- Goals of memory management
- Partitioning
- Relocation
  - Need for logical and physical addresses
- Paging
- Segmentation
Virtual Memory

- **Virtual memory** – program uses virtual memory which can be partially loaded into physical memory

- **Benefits:**
  - Only *part of the program* needs to be in memory for execution
    - more concurrent programs
  - Logical address space can therefore be *much larger than physical address space*
    - execute programs larger than RAM size
  - Easy *sharing* of address spaces by several processes
    - Library or a memory segment can be shared
  - Allows for more *efficient process creation*
Paging

- Fixed-size partitioning → internal fragmentation
- Variable-size partitioning → external fragmentation
- Partition memory into equal fixed-size chunks (termed frames) that are relatively small
- Process is also divided into small fixed-size chunks (termed pages) of the same size

No external fragmentation + little internal fragmentation (only last page of process)
Paging

- Physical address space of a process can be noncontiguous
  - Physical address space will also be **noncontiguous**.

- Divide physical memory into small equal fixed-size blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)

- Divide logical memory (process) into blocks of same size called **pages**

- 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
- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page
RAM (Physical Memory)

- a frame (size = $2^x$)
- physical memory: set of fixed sized frames

Program: set of pages

Page size = Frame size
Assignment of Process to Free Frames

There are not sufficient unused *continuous* frames to hold the process D. Does this makes it impossible to load D?

Process A consists of 4 pages
Process B consists of 3 pages
Process C consists of 4 pages
Process B is suspended and is swapped out of main memory.
Process D consists of 5 pages
Paging

Each page can be mapped to any frame.

A program

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Page table

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RAM</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Load

0 mapped to 1
1 mapped to 4
2 mapped to 2
3 mapped to 7
4 mapped to 9
5 mapped to 6
Page 0
Page 1
Page 2
Page 3
Page 4
...
page n-2
page n-1

Virtual memory

Page 2
unavail
Page 0
unavail
Page 3
Page 1

Page 0
Page 1
Page 2
Page 3
Page 4
page n-2
Page n-1

Physical memory

move pages

all pages of program sitting on physical Disk
One page table for each process + One for free frame

Figure 7.9 Assignment of Process Pages to Free Frames

Free frame list that are currently unoccupied and available for pages
The page number is used as an index into the page table. The page table contains the base address (frame number) of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
Page 0 is assigned physical frame 1.
Page 1 is assigned physical frame 4.
Page 2 is assigned physical frame 3.
Page 3 is assigned physical frame 7.
Page Table

- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Processor must know how to access for the current process
- Used by processor to produce a physical address
✓ The OS now needs to maintain (in main memory) a page table for each process

✓ Each entry of a page table consists of the frame number where the corresponding page is physically located

✓ The page table is indexed by the page number to obtain the frame number

✓ A free frame list, available for pages, is maintained
Address Translation Scheme in Paging

- Assume Logical Addresses are \( m \) bits. Then logical address space is \( 2^m \) bytes.
- Assume page size is \( 2^n \) bytes.

- Logical 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>
<tr>
<td>( m ) bits</td>
<td></td>
</tr>
</tbody>
</table>
Virtual Memory has 16-bit address.

Page size = 1 KB = $2^{10}$ Bytes

What is a Logical address of relative 1502?
Logical Address in Paging

Logical address = page # + offset within page

Relative address = 1502

0000010111011110

Logical address =
Page# = 1, Offset = 478

0000010111011110

User process
(2,700 bytes)

(a) Partitioning

Page 0

Page 1

Page 2

(b) Paging
(page size = 1K)

Internal fragmentation

16-bit address

Page size = 1KB
= 2^{10} Bytes

program can consist of a maximum 64 pages.
Paging Hardware: Address Translation
Paging Example

Page size = 4 bytes = 2^2

Number of frames = 7

4 bit logical address

Page number  Offset inside page

32 byte memory

LA = 5
PA = ?
5 is 0101
PA = 11001

LA = 11
PA = ?
11 is 1011
PA = 00111

LA = 13
PA = ?
13 is 1101
PA = 01001
16 bit logical address
page size = 4096 bytes
(offset is 12 bits)
15 bit physical address
translate 0010000000000100?

```
<table>
<thead>
<tr>
<th>p#</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>f#</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>0000000000100</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>frame number</th>
<th>Present/not Present bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
Address Translation: Example (II)

- Virtual Memory has 3 bits:
  - \(2^3 = 8\) logical addresses
  - page size = \(2^2 = 4\)
- 1 bit for page # and 2 bits for offset

Each entry is used to map 4 addresses (page size addresses)
Logical Address in Paging

• We can not use physical addresses in paging

• All addresses are logical addresses

• Within each program, each logical address” must consist of a page number + an offset within the page

• A CPU register always holds the starting physical address of the page table of the currently running process

• Presented with the logical address (page number, offset) the processor accesses the page table to obtain the physical address (frame number, offset)
Logical-to-Physical Address Translation: Paging

Diagram:

- 16-bit logical address
  - 6-bit page #
  - 10-bit offset
- Logical Address: 000000101110111110
- Process page table:
  - Page 0: 000101
  - Page 1: 000110
  - Page 2: 011001
- Physical address: 000110011110111110
- 16-bit physical address

(a) Paging
Each process has its own page table

Each page table entry contains the frame number of the corresponding page in main memory

Two extra bits are needed to indicate:
- A bit is needed to indicate whether the page is in main memory or not (Present bit)
- Whether the contents of the page has been altered since it was last loaded (Modified bit)
- Other Control bits
  - Depends on OS
    - Used if protection is managed at the page level
      - E.g., read/only
      - E.g., kernel only

Why would we want a modified bit?
Modify bit is needed to indicate if the page has been altered since it was last loaded into main memory.

If no change has been made, the page does not have to be written to the disk when it needs to be swapped out [saving IO time].
Example: Paging

Virtual memory = 32G
Page size = 4K
Physical memory = 8G
Number of processes = 256
Assume that a page table entry: 4 bytes

Number of pages we have?

Size of page table?

Size of all page tables together?

Size of a page table entry in physical?
Example: Paging

Virtual Address (35 bits because of 32G virtual memory)
Number of pages we have?
8M (because we have 23 bits for page#)
Size of page table?
32M
Size of all page tables together:
32M * 256 = 8G
Physical Address (33 bits)
P (1bit)
M (1bit)
Frame#: 21 bits
Need a min of 23 bits for page table entry
4bytes gives me 32 bits, I have 9 for other control bits

The page table itself may be so large!
Page tables are subject to paging!
Page tables are also stored in virtual memory
Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into the page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. (The logical memory shown is simply for visual reference). Let us consider an example:
The CPU generates the logical address necessary to reference the letter g, or "page 1, offset 2": 0102
The CPU generates the logical address necessary to reference the letter g, or "page 1, offset 2" : 0102
The page number (p) of the logical address is then used to reference the page table to obtain the physical memory base address.
This physical memory base address is then combined with the logical address offset to produce the physical memory address needed to reference the letter “g” in physical memory.
Currently running process is process 1 (P1)
Important to Remember

• With paging and segmentation
  – Process ALWAYS issues ONLY LOGICAL ADDRESS
  – It CAN NOT issue PHYSICAL ADDRESS
  – Address translation is a must

• In most situations, it is important that the address translation be done O(ns) so that there is no extra delay in accessing the page.
A Few Observations with Paging

• With paging, a process need **NOT** be contiguous in memory

• **Does page table need to be contiguous?** Why?

• Remember the question. We will come back to problems caused by this in Chapter 8.
Paging vs Fixed Partitioning

- With paging, the partitions are rather small
- Program may occupy more than one partition
- these patrons need not be contiguous
Outline

- Goals of memory management
- Partitioning
- Relocation
  - Need for logical and physical addresses
- Paging
- Segmentation
A program can be subdivided into segments
- may vary in length
- there is a maximum segment length

Addressing consists of two parts:
- segment number
- an offset

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning

Eliminates internal fragmentation
Segmentation

- All segments of all programs do NOT have to be of the same length
  - There is a maximum segment length

-Segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

  [suffers from external fragmentation but less]
In contrast with paging, segmentation is visible to the programmer—provided as a convenience to organize logically programs (ex: data in one segment, code in another segment)—must be aware of segment size limit

The OS maintains a segment table for each process.

Each entry contains:
- the starting physical addresses of that segment.
- the length of that segment (for protection)
Segmentation

Logical address space

- Subroutine
- Stack
- Symbol table
- Main program

Segment 0
Segment 1
Segment 2
Segment 3
Segment 4

Starting physical addresses

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>4700</td>
</tr>
</tbody>
</table>

Length

Physical memory

1400
2400
3200
4300
5700
6300
6700
Logical-to-Physical Address Translation: Segmentation

16-bit logical address

4-bit segment # 12-bit offset

000100101111110000

Process segment table

Length Base
0 001011101110 000001000000000000
1 01110011110 0010000000100000

(b) Segmentation

16-bit physical address
Each entry of the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
A logical address consists of two parts: a segment number, $s$, and an offset into that segment, $d$. The segment number is used as an index into the segment table. The offset $d$ of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If the offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
Consider the following 3 examples: The CPU generates logical address 1200. This is divided into segment 1, offset 200.
Segmentation Hardware

<table>
<thead>
<tr>
<th>Segment</th>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4700</td>
</tr>
</tbody>
</table>

Segment 1 is accessed in the table, and the offset, 200, is compared to the limit, 400, to see if it is less than the limit.
The comparison passes, so the offset is added to the base to produce the physical memory address $6300 + 200 = 6500$. 
Example 3: The CPU generates logical address 2500. This is divided into segment 2, offset 500.
The comparison fails, so a trap to the operating system is generated.
Segmentation vs Dynamic Portioning

- A process is broken up into a number of small pieces so external fragmentation should be less.

- Program may occupy more than one partition

- These partitions need not be contiguous

Disadvantages:
- Programmer must be aware of the maximum segment size limitation
- No simple relationship between logical and physical addresses.
Memory Management

- one of the most important and complex tasks of an operating system
- needs to be treated as a resource to be allocated to and shared among a number of active processes
- desirable to maintain as many processes in main memory as possible
- desirable to free programmers from size restriction in program development
- basic tools are paging and segmentation (possible to combine)
  - paging – small fixed-sized pages
  - segmentation – pieces of varying size