Recall: Four requirements for occurrence of Deadlock

- **Mutual exclusion**
  - Only one thread at a time can use a resource.
- **Hold and wait**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
  - There exists a set \{T_1, ..., T_n\} of waiting threads
    - \(T_1\) is waiting for a resource that is held by \(T_2\)
    - \(T_2\) is waiting for a resource that is held by \(T_3\)
    - \(\ldots\)
    - \(T_n\) is waiting for a resource that is held by \(T_1\)

Virtualizing Resources

- **Protection:**
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs
    - Programs protected from themselves
- **Translation:**
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
    - Side effects:
      - Can be used to avoid overlap
      - Can be used to give uniform view of memory to programs
- **Controlled overlap:**
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
Alternative View: Interposing on Process Behavior

- OS interposes on process’ I/O operations
  - How? All I/O happens via system calls.

- OS interposes on process’ CPU usage
  - How? Interrupt lets OS preempt current thread

- Question: How can the OS interpose on process’ memory accesses?
  - Too slow for the OS to interpose every memory access
  - Translation: hardware support to accelerate the common case
  - Page fault: uncommon cases trap to the OS to handle

Recall: Four Fundamental OS Concepts

- Thread: Execution Context
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack

- Address space (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine
    (in which case programs operate in a virtual address space)

- Process: an instance of a running program
  - Protected Address Space + One or more Threads

- Dual mode operation / Protection
  - Only the “system” has the ability to access certain resources
  - Combined with translation, isolates programs from each other and the OS from programs

THE BASICS: Address/Address Space

- What is $2^{10}$ bytes (where a byte is abbreviated as “B”)?
  - $2^{10} \text{B} = 1024 \text{B} = 1 \text{ KB}$ (for memory, $1 \text{K} = 1024$, not 1000)

- How many bits to address each byte of 4KB page?
  - $4 \text{KB} = 4 \times 1 \text{KB} = 4 \times 2^{10} = 2^{12} = 12 \text{ bits}$

- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
  - Use $2^k$

Address Space, Process Virtual Address Space

- Definition: Set of accessible addresses and the state associated with them
  - $2^{32} = \sim 4 \text{ billion bytes}$ on a 32-bit machine

- How many 32-bit numbers fit in this address space?
  - 32-bits = 4 bytes, so $2^{32} / 4 = 2^{30} = \sim 1 \text{ billion}$

- What happens when processor reads or writes to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - …
Recall: Process Address Space: typical structure

Processor registers

PC:
0x000...

SP:
0xFFF...

Code Segment
Static Data
Heap
Stack Segment
sbrk syscall

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at the same place in physical memory since only one application at a time
  - Application can access any physical address

Application

– Application given illusion of dedicated machine by giving it reality of a dedicated machine

Recall: Primitive Multiprogramming

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads

- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
  - Everything adjusted to memory location of program
  - Translation done by a linker-loader (relocation)
  - Common in early days (… till Windows 3.x, 95?)

- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

Binding of Instructions and Data to Memory

Process view of memory

Physical addresses

Assume 4 byte words
0x300 = 4 * 0x80
0x80 = 0000 1100 0000
0x300 = 0011 0000 0000

0x0000 0000
0x00200000
0x0000 0000
0x0300 00000020
0x0900 0000 0000 0280
0x0904 0000 0000 0820
0x0908 0000 0000 2021FFFF
0x090C 0000 0000 0242
0x0A00

**Binding of Instructions and Data to Memory**

- **Process view of memory**
  - **Physical addresses**:
    - `0x0300`: `00000000` to `00000020` (data1)
    - `0x0900`: `00000020` to `00000060`
    - `0x1300`: `00000060` to `000000FF`
    - `0x0A00`: `00000020` to `0000002F`

- **Physical Memory**
  - `0x0000`: `00000000`
  - `0x0300`: `00000020`
  - `0x0900`: `00000060`
  - `0x1300`: `000000FF`

**Second copy of program from previous example**

- **Process view of memory**
  - **Physical addresses**:
    - `0x0300`: `00000000` to `00000020` (data1)
    - `0x0900`: `00000020` to `00000060`
    - `0x1300`: `00000060` to `000000FF`
    - `0x0A00`: `00000020` to `0000002F`

- **Physical Memory**
  - `0x0000`: `00000000`
  - `0x0300`: `00000020`
  - `0x0900`: `00000060`
  - `0x1300`: `000000FF`

**Need address translation!**

**From Program to Process**

- Preparation of a program for execution involves components at:
  - Compile time (i.e., "gcc")
  - Link/Load time (UNIX "ld" does link)
  - Execution time (e.g., dynamiclibs)

- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system

- **Dynamic Libraries**
  - Linking postponed until execution
  - Small piece of code (i.e. the stub), locates appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine

**Second copy of program from previous example**

- **Process view of memory**
  - **Physical addresses**:
    - `0x1300`: `00000000` to `00000020` (data1)
    - `0x1900`: `00000020` to `00000060`
    - `0x1904`: `00000060` to `000000FF`
    - `0x1A00`: `00000020` to `0000002F`

- **Physical Memory**
  - `0x0000`: `00000000`
  - `0x0300`: `00000020`
  - `0x0900`: `00000060`
  - `0x1300`: `000000FF`

- **App X**
  - `0x0000`: `00000000`
  - `0x0300`: `00000020`
  - `0x0900`: `00000060`
  - `0x1300`: `000000FF`

- **One of many possible translations!**

- Where does translation take place? Compile time, Link/Load time, or Execution time?
## Administrivia

- **Midterm 2:** Wednesday 3/15 from 8-10PM  
  - A week from tomorrow!!!  
  - All material up to Lecture 16 technically in bounds
- **Homework 4** coming out  
  - Released tomorrow, Wednesday 3/08
- **Project 2 design document** due this Friday!

## Administrivia (Con’t)

- **You need to know your units as CS/Engineering students!**
- **Units of Time:**  
  - Millisecond: 1ms $\Rightarrow 10^{-3}$ s  
  - Microsecond: 1μs $\Rightarrow 10^{-6}$ s  
  - Nanosecond: 1ns $\Rightarrow 10^{-9}$ s
- **Integer Sizes:**  
  - Bit: 1b  
  - Byte: 1B $\Rightarrow 8$ bits  
  - Word: $W$ (depends. Could be 16b, 32b, 64b)
- **Units of Space (memory), sometimes called the “binary system”**  
  - Kilo: 1KB $\Rightarrow 1024$ bytes $\Rightarrow 2^{10}$ bytes $\Rightarrow 1024 = 1.0 \times 10^3$  
  - Mega: 1MB $\Rightarrow 1024^2$ bytes $\Rightarrow 2^{20}$ bytes $\Rightarrow 1,048,576 = 1.0 \times 10^6$  
  - Giga: 1GB $\Rightarrow 1024^3$ bytes $\Rightarrow 2^{30}$ bytes $\Rightarrow 1,073,741,824 = 1.1 \times 10^9$  
  - Tera: 1TB $\Rightarrow 1024^4$ bytes $\Rightarrow 2^{40}$ bytes $\Rightarrow 1,099,511,627,760 = 1.1 \times 10^{12}$  
  - Peta: 1PB $\Rightarrow 1024^5$ bytes $\Rightarrow 2^{50}$ bytes $\Rightarrow 1,125,899,906,842,624 = 1.1 \times 10^{15}$  
  - Exa: 1EB $\Rightarrow 1024^6$ bytes $\Rightarrow 2^{60}$ bytes $\Rightarrow 1,152,921,504,606,846,976 = 1.2 \times 10^{18}$
- **Units of Bandwidth, Space on disk/etc, Everything else…, sometimes called the “decimal system”**  
  - Kilo: 1KB/s $\Rightarrow 10^3$ bytes/s, 1KB $\Rightarrow 10^3$ bytes  
  - Mega: 1MB/s $\Rightarrow 10^6$ bytes/s, 1MB $\Rightarrow 10^6$ bytes  
  - Giga: 1GB/s $\Rightarrow 10^9$ bytes/s, 1GB $\Rightarrow 10^9$ bytes  
  - Tera: 1TB/s $\Rightarrow 10^{12}$ bytes/s, 1TB $\Rightarrow 10^{12}$ bytes  
  - Peta: 1PB/s $\Rightarrow 10^{15}$ bytes/s, 1PB $\Rightarrow 10^{15}$ bytes  
  - Exa: 1EB/s $\Rightarrow 10^{18}$ bytes/s, 1EB $\Rightarrow 10^{18}$ bytes

## Multiprogramming with Protection

- Can we protect programs from each other without translation?  
  - **Yes:** Base and Bound!  
  - Used by, e.g., Cray-1 supercomputer

## Recall: Base and Bound (No Translation)

- **Still protects OS and isolates program**  
  - **Requires relocating loader**  
  - **No addition on address path**

### Original Program

```
0000... code
Static Data
heap
stack
1000...
```

### New Program

```
0000... code
Static Data
heap
stack
1000...
```

### Bound

```
Bound = 0x30000
Base = 0x20000
0x00000000
```

### Program address

```
Program address = 0x1010...
```

### Base and Bound

```
1100... < Base
```

```
1000... > Bound
```

### Code Segment

```
1000... Base
```

```
1100... Bound
```

```
0000... code
Static Data
heap
stack
```

```
0000... code
Static Data
heap
stack
```

```
0000... code
Static Data
heap
stack
```
Recall: General Address translation

- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

![Address Translation Diagram]

Recall: Base and Bound (with Translation)

- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

![Base and Bound Diagram]

Issues with Simple B&B Method

- Fragmentation problem over time
  - Not every process is same size ⇒ memory becomes fragmented over time
- Missing support for sparse address space
  - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

![Issues with Simple B&B Diagram]

More Flexible Segmentation

- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

![More Flexible Segmentation Diagram]
**Implementation of Multi-Segment Model**

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: `mov [es:bx],ax`
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well

**Intel x86 Special Registers**

- Typical Segment Register
  - Current Priority is RPL of Code Segment (CS)
- Segmentation can’t be just “turned off”
  - What if we just want to use paging?
  - Set base and bound to all of memory, in all segments

**Example: Four Segments (16 bit addresses)**

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Base Limit</th>
<th>Virtual Address Format</th>
<th>Seg ID</th>
<th>Base Limit</th>
<th>Virtual Address Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000 0x0800</td>
<td>0x0000 0x04000 0x08000 0xC000</td>
<td>1</td>
<td>0x4800 0x1400</td>
<td>0x0000 0x48000 0x14000</td>
</tr>
<tr>
<td>2</td>
<td>0xF000 0x1000</td>
<td>0x0000 0x80000 0x10000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x0000 0x3000</td>
<td>0x0000 0x00000 0x30000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example: Four Segments (16 bit addresses)**

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Base Limit</th>
<th>Virtual Address Format</th>
<th>Seg ID</th>
<th>Base Limit</th>
<th>Virtual Address Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000 0x0800</td>
<td>0x0000 0x40000 0x08000</td>
<td>1</td>
<td>0x4800 0x1400</td>
<td>0x0000 0x48000 0x14000</td>
</tr>
<tr>
<td>2</td>
<td>0xF000 0x1000</td>
<td>0x0000 0x80000 0x10000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x0000 0x3000</td>
<td>0x0000 0x00000 0x30000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format:

- Seg ID = 0
- Seg ID = 1
- Seg ID = 2
- Seg ID = 3

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"

Example of Segment Translation (16bit address)

```
0x0240 main: la $a0, varx
0x0244 jal strlen
0x0360 strlen: li $v0, 0 ;count
0x0364 loop: lb $t0, ($a0)
0x0368 beq $r0, $t0, done
0x0450 varx dw 0x314159
```

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
Let’s simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 ( 0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050
   Move PC+4

2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248
   Move 0x0360
   Move PC

3. Fetch 0x0360. Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x0000
   Move PC+4

Observations about Segmentation

- Translation on every instruction fetch, load or store
- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
- When it is OK to address outside valid range?
  - This is how the stack (and heap?) allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
- What must be saved/restore on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called “swapping”)

What if not all segments fit in memory?

- Extreme form of Context Switch: Swapping
  - To make room for next process, some or all of the previous process is moved to disk
    - Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory
Problems with Segmentation

• Must fit variable-sized chunks into physical memory
• May move processes multiple times to fit everything
• Limited options for swapping to disk
• Fragmentation: wasted space
  – External: free gaps between allocated chunks
  – Internal: don’t need all memory within allocated chunks

Paging: Physical Memory in Fixed Size Chunks

• Solution to fragmentation from segments?
  – Allocate physical memory in fixed size chunks ("pages")
  – Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation: 00110001110001101 ... 110010
    » Each bit represents page of physical memory
      1 => allocated, 0 => free

• Should pages be as big as our previous segments?
  – No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  – Consequentially: need multiple pages/segment

Recall: General Address Translation

Code
Data
Heap
Stack

Prog 1
Virtual Address Space 1

Translation Map 1

Translation Map 2

Physical Address Space

How to Implement Simple Paging?

Virtual Address: Value Page # Offset

PageTablePtr

PageTableSize

PageTable

Check Perm

Physical Address

Access Error

Access Error

• Page Table (One per process)
  – Resides in physical memory
  – Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
• Virtual address mapping
  – Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset \( \rightarrow \) 1024-byte pages
  – Virtual page # is all remaining bits
    » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    » Physical page # copied from table into physical address
  – Check Page Table bounds and permissions
**Simple Page Table Example**

Example (4 byte pages)

<table>
<thead>
<tr>
<th>Virtual Memory</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 abcd</td>
<td>0x00 0000</td>
</tr>
<tr>
<td>0x04 efgh</td>
<td>0x04 0000</td>
</tr>
<tr>
<td>0x08 ijkl</td>
<td>0x08 0000</td>
</tr>
<tr>
<td>0x0C aefg</td>
<td>0x0C 0000</td>
</tr>
</tbody>
</table>

**What about Sharing?**

Virtual Address (Process A):

- PageTablePtrA
- page #0: V,R
- page #1: V,R
- page #2: V,R,W
- page #3: V,R,W
- page #4: N
- page #5: V,R,W

Virtual Address (Process B):

- PageTablePtrB
- page #0: V,R
- page #1: N
- page #2: V,R,W
- page #3: N
- page #4: N
- page #5: V,R,W

Shared Page

This physical page appears in address space of both processes

**Where is page sharing used?**

- The “kernel region” of every process has the same page table entries
  - The process cannot access it at user level
  - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?
- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  - Can actually share objects directly between processes
    » Must map page into same place in address space!
  - This is a limited form of the sharing that threads have within a single process

**Memory Layout for Linux 32-bit (Pre-Meltdown patch!)**

- Kernel space
- User code
- Stack (grows down)
- Memory Mapping Segment
  - File mappings (including system libraries) and anonymous mappings
  - Example: /lib/x86_64-linux-gnu
- Data Segment
  - Static data initialized by the programmer: Example: static char *p = "Bob's new prototype";
  - Text Segment
    - Maps the binary image of the program

http://static.duarte.s.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png
Some simple security measures

- Address Space Randomization
  - Position-Independent Code \(\Rightarrow\) can place user code anywhere in address space
    » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don’t map whole kernel space into each process, switch to kernel page table
  - Meltdown \(\Rightarrow\) map none of kernel into user mode!

Summary: Paging
Conclusion

• Segment Mapping
  – Segment registers within processor
  – Segment ID associated with each access
    » Often comes from portion of virtual address
    » Can come from bits in instruction instead (x86)
  – Each segment contains base and limit information
    » Offset (rest of address) adjusted by adding base

• Page Tables
  – Memory divided into fixed-sized chunks of memory
  – Virtual page number from virtual address mapped through page table to physical page number
  – Offset of virtual address same as physical address
  – Large page tables can be placed into virtual memory

• Next Time: Multi-Level Tables
  – Virtual address mapped to series of tables
  – Permit sparse population of address space