Recall: Starvation vs Deadlock
- **Starvation**: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- **Deadlock**: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
  - Thread B owns Res 2 and is waiting for Res 1

- Deadlock $\Rightarrow$ Starvation but not vice versa
  - Starvation can end (but doesn’t have to)
  - Deadlock can’t end without external intervention

Recall: Four requirements for 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 \( \{T_1, \ldots, T_n\} \) 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 \( T_1 \)

Recall: Ways of preventing deadlock
- **Force all threads to request resources in a particular order preventing any cyclic use of resources**
  - Example \((x.P, y.P, z.P, \ldots)\)
    - Make tasks request disk, then memory, then…
- **Banker’s algorithm**:
  - Allocate resources dynamically
    - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - Technique: pretend each request is granted, then run deadlock detection algorithm, and grant request if result is deadlock free (conservative!)
  - Keeps system in a “SAFE” state, i.e. there exists a sequence \( \{T_1, T_2, \ldots T_n\} \) with \( T_1 \) requesting all remaining resources, finishing, then \( T_2 \) requesting all remaining resources, etc.
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources
Can Priority Inversion cause Deadlock?

- Technically not – Consider this example:
  - 3 threads, T1, T2, T3 in priority order (T3 highest)
  - T1 grabs lock, T3 tries to acquire, then sleeps, T2 running
  - Will this make progress?
    » No, as long as T2 is running
    » But T2 could stop at any time and the problem would resolve itself... So, this is not a deadlock (it is a livelock)
  - Why is this a priority inversion?
    » T3 is prevented from running by T2

- How does priority donation help?
  - Briefly raising T1 to the same priority as T3 ⇒ T1 can run and release lock, allowing T3 to run
  - Does priority donation involve tacking lock away from T1?
    » NO! That would break semantics of the lock and potentially corrupt any information protected by lock!

Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
  - Demand paging
- Today: Translation

Virtualizing Resources

- Physical Reality: Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (starting today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    » Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don’t want different threads to even have access to each other’s memory if in different processes (protection)

Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process
Important Aspects of Memory Multiplexing

• 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

• 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)

• 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

Recall: Loading

Process view of memory

Physical addresses

Assume 4-byte words

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
</tr>
<tr>
<td>0x0300</td>
</tr>
<tr>
<td>0x0C00</td>
</tr>
<tr>
<td>0x2000</td>
</tr>
<tr>
<td>0x3000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data1: dw 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

Process view of memory

Physical addresses

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
</tr>
<tr>
<td>0x0300</td>
</tr>
<tr>
<td>0x0C00</td>
</tr>
<tr>
<td>0x2000</td>
</tr>
<tr>
<td>0x3000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process view of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>data1: dw 32</td>
</tr>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start: lw r1,0(data1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jal checkit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop: addi r1, r1, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bsz r1, loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checkit: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data1: dw 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start: lw r1,0(data1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jal checkit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop: addi r1, r1, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bsz r1, loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checkit: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
</tr>
</tbody>
</table>
Second copy of program from previous example

Process view of memory
Physical addresses
App X

data1: dw 32
start: lw r1,0(data1)
jal checkit
loop: addi r1, r1, -1
bnz r1, loop
checkit: ...

Need address translation!

Multi-step Processing of a Program for Execution

- 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., dynamic libs)
- 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

Recall: Uniprogramming

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

Valid 32-bit Addresses

Operating System
0xFFFF000000
Application
0x00000000
Multiprogramming (primitive stage)
- 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

Recall: General Address translation
- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the 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

Recall: Base and Bounds (was from CRAY-1)
- Could use base/bounds for dynamic address translation – translation happens at execution:
  - Alter address of every load/store by adding “base”
  - Generate error if address bigger than limit
- Gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Multiprogramming (Version with Protection)
- Can we protect programs from each other without translation?
  - Yes: use two special registers BaseAddr and LimitAddr to prevent user from straying outside designated area
    » Cause error if user tries to access an illegal address
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    » User not allowed to change base/limit registers
Issues with Simple B&B Method

- Fragmentation problem over time
  - Not every process is same size \( \Rightarrow \) 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

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

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)
**Administrivia**

- **Project 1**
  - Code Due tomorrow (3/8)
  - Final Report due Tuesday (3/12)

- **Midterm 1 Regrade Requests**
  - Due Monday (3/11)
  - Don’t just send request to send it! (We may regrade everything if it is a specious request for points – you might lose points…!)

- **Midterm 2: Thursday 4/4**
  - Ok, this is a few weeks and after Spring Break
  - Will definitely include Scheduling material (lecture 10)
  - Up to and including some material from lecture 17
  - Probably try to have a Midterm review in early part of that week…. Stay tuned
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>

Example of Segment Translation (16bit address)

<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>

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

1. Fetch 0x240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
2. Fetch 0x244 (0000 0011 0100 0000). Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360 (0011 0010 0000 0000). Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x04050 → $a0, Move PC+4→PC

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

1. Fetch 0x240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
2. Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x04050 → $a0, Move PC+4→PC
3. Fetch 0x360 (0011 0010 0000 0000). Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x04050 → $a0, Move PC+4→PC
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 → $a0, Move PC+4→PC

2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

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

4. Fetch 0x0364. Translated to Physical=0x4364. Get "lb $t0, ($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Example of Segment Translation (16bit address)

<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>

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    » If it does, trap to kernel and dump core
- When it is OK to address outside valid range?
  - This is how the stack and heap are 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)
  - Shared segment could be read-only or read-write
- What must be saved/restored 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 into memory?

- Extreme form of Context Switch: Swapping
  - In order 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
Recall: General Address Translation

Virtual Address Space 1

Code
Data
Heap
Stack

Prog 1

Virtual
Address
Space 1

Translation Map 1

Translation Map 2

Physical Address Space

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)
  – Consequently: need multiple pages/segment

How to Implement Paging?

Virtual Address: Offset

Page Table (One per process)

– Resides in physical memory
– Contains physical page and permission for each virtual page
  » Permissions include: Valid bits, Read, Write, etc

Virtual address mapping

– Offset from Virtual address copied to Physical Address
  » Example: 10 bit offset ⇒ 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)
What about Sharing?

Virtual Address (Process A):

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
<th>V, R, W</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td></td>
<td>V, R</td>
</tr>
<tr>
<td>page #1</td>
<td></td>
<td>V, R</td>
</tr>
<tr>
<td>page #2</td>
<td></td>
<td>V, R, W</td>
</tr>
<tr>
<td>page #3</td>
<td></td>
<td>V, R, W</td>
</tr>
<tr>
<td>page #4</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td></td>
<td>V, R, W</td>
</tr>
</tbody>
</table>

Virtual Address (Process B):

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
<th>V, R, W</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td></td>
<td>V, R</td>
</tr>
<tr>
<td>page #1</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>page #2</td>
<td></td>
<td>V, R, W</td>
</tr>
<tr>
<td>page #3</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>page #4</td>
<td></td>
<td>V, R</td>
</tr>
</tbody>
</table>

Shared Page

This physical page appears in address space of both processes.

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

Summary: Paging

What happens if stack grows to 1110 0000?

Summary: Paging
Summary: Paging

- Virtual memory view:
  - Stack
  - Heap
  - Data

- Physical memory view:
  - Stack
  - Heap
  - Data

- Page Table:
  - Page table pointer and limit

Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- Analysis
  - Pros
    - Simple memory allocation
    - Easy to share
  - Cons: What if address space is sparse?
    - E.g., on UNIX, code starts at 0, stack starts at \((2^{31}-1)\)
    - With 1K pages, need 2 million page table entries!
  - Cons: What if table really big?
    - Not all pages used all the time ⇒ would be nice to have working set of page table in memory

- How about multi-level paging or combining paging and segmentation?

Fix for sparse address space:
The two-level page table

- Virtual Address: 10 bits
  - Virtual P1 index
  - Virtual P2 index
- Offset: 12 bits
- Physical Address: 12 bits
- Physical Page #: 10 bits

- PageTablePtr

- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don’t need every 2\(^{nd}\)-level table
  - Even when exist, 2\(^{nd}\)-level tables can reside on disk if not in use

Summary: Two-Level Paging

Virtual memory view:
- Stack
- Heap
- Data

Page Table (level 1):
- Stack
- Heap
- Data

Page Tables (level 2):
- Stack
- Heap
- Data

Physical memory view:
- Stack
- Heap
- Data
Summary: Two-Level Paging

Virtual memory view:
- Stack
- Heap
- Data
- Code

Physical memory view:
- Stack
- Heap
- Data
- Code

Virtual memory view:
- 1001 0000 (0x90)
- 1000 0000 (0x80)

Page Tables:
- Level 1
- Level 2

Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

Virtual Address:
- Virtual Seg #
- Virtual Page #
- Offset

Physical Address:
- Physical Page #
- Offset

Cons:
- One pointer per page (typically 4K – 16K pages today)
- Page tables need to be contiguous
  » However, previous example keeps tables to exactly one page in size
- Two (or more, if >2 levels) lookups per reference
  » Seems very expensive!
Summary

- **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

- **Multi-Level Tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space