Recall: Starvation vs Deadlock

- 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 ⇒ 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 \( \{T_1, ..., T_n\} \) 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 \( T_3 \)
    » ...
    » \( T_n \) 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,...)
    » 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, substituting \( ([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) \) for \( ([\text{Request}_{\text{node}}] \leq [\text{Avail}]) \)
    » Grant request if result is deadlock free (conservative!)
    » Keeps system in a "SAFE" state, i.e. there exists a sequence \( \{T_1, T_2, ..., T_n\} \) 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
Recall: Address translation

- Address Space:
  - All the addresses and states 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 essential to implementing 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: General Address Translation

Simple Base and Bounds (CRAY-1)

- Could use base/limit for dynamic address translation - translation happens at execution:
  - Alter address of every load/store by adding “base”
  - Generate error if address bigger than limit
- This 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

Issues with Simple B&B Method

- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program
  - E.g.: Code, Data, Stack
- 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)

Example: Four Segments (16 bit addresses)
Example of segment translation

Let's simulate a bit of this code to see what happens (PC=0x240):
1. Fetch 0x240. 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 0x244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248    $ra (return address!), Move 0x0360    PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000    $v0, Move PC+4    PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

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 segments than will 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
- 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)
  - Consequently: need multiple pages/segment

How to Implement Paging?

- 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)

| Virtual Memory | | Physical Memory |
|----------------|----------------|
| 0000 0110      | 0000 1110      |
| 0000 1001      | 0000 0101      |
| 0000 0100      | 0000 0100      |
| 0000 0101      | 0000 0101      |

Check Perm

AccessError

Page Table Size

Page Table Ptr

Virtual Address:

Offset

Page Table #

Physical Address

Offset

Page Table #
What about Sharing?

Virtual Address (Process A):

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

Virtual Address (Process B):

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

This physical page appears in address space of both processes.

What happens if stack grows to 1110 0000?
**Summary: Simple Page Table**

Virtual memory view | Physical memory view
---|---
0000 0000 | 0000 0000
0001 0000 | 0001 0000
0101 000 | 0101 000
0111 000 | 0111 000
1110 0000 | 1110 0000

Allocate new pages where room!

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

- 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 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

**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
  - Con: 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!
  - Con: 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 combining paging and segmentation?
  - Segments with pages inside them?
  - Need some sort of multi-level translation

**Summary: Two-Level Paging**

Virtual memory view | Page Tables (level 1) | Physical memory view (level 2)
---|---|---
0000 0000 | 0001 0000 | 0000 0000
0001 0000 | 0001 0000 | 0000 0000
0101 000 | 0101 000 | 0101 000
0111 000 | 0111 000 | 0111 000
1110 0000 | 1110 0000 | 1110 0000

Allocate new pages where room!
**Summary: Two-Level Paging**

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

- **Physical memory view**
  - Heap
  - Data
  - Code

**Physical memory view (level 2)**

- **Page Tables**
  - Level 1: 10000000
  - Level 2: 00000000

**Virtual memory view (level 2)**

- **Page Table**
  - Stack
  - Heap
  - Data

**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):**
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

**Multi-level Translation Analysis**

- **Pros:**
  - Only need to allocate as many page table entries as we need for application
  - In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)
- **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
Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    » Much of process space may be out on disk or not in use

- Answer: use a hash table
  - Called an “Inverted Page Table”
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces

- Cons: Complexity of managing hash changes
  - Often in hardware!

X86 Segment Descriptors (32-bit Protected Mode)

- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
  - There are 6 registers: SS, CS, DS, ES, FS, GS

- What is in a segment register?
  - A pointer to the actual segment description:
    - Segment selector [13 bits]
      - G/L: Global vs local descriptor tables
      - RPL: Descriptor Privilege Level
    - G/L selects between GDT and LDT tables (global vs local descriptor tables)

- Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
  - Includes length of table (for < 2^13 entries)

- Descriptor format (64 bits):
  - Base address (39 bits)
  - Offset (13 bits)

  - G: Granularity of segment (0: 16bit, 1: 4KiB unit)
  - DB: Default operand size (0: 16bit, 1: 32bit)
  - A: Available for use by software
  - P: Segment present
  - DPL: Descriptor Privilege Level
  - S: Type (0: segment, 1: system)
  - Type: Code, Data, Segment

What is in a Page Table Entry?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only

- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

- Page Frame Number
  - (Physical Page Number)
  - Free (OS)
  - O: User accessible
  - L: Present (same as "valid" bit in other architectures)
  - W: Writeable
  - U: User accessible
  - PWT: Page write transparent: external cache write-through
  - PCD: Page cache disabled (page cannot be cached)
  - A: Accessed: page has been accessed recently
  - D: Dirty (PTE only): page has been modified recently
  - L: L=1=4MB page (directory only).
  - Bottom 22 bits of virtual address serve as offset
Examples of how to use a PTE

- **How do we use the PTE?**
  - Invalid PTE can imply different things:
    - Region of address space is actually invalid or
    - Page/directory is just somewhere else than memory
  - Validity checked first
    - OS can use other (say) 31 bits for location info

- **Usage Example: Demand Paging**
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid

- **Usage Example: Copy on Write**
  - UNIX fork gives copy of parent address space to child
  - Address spaces disconnected after child created
  - How to do this cheaply?
    - Make copy of parent's page tables (point at same memory)
    - Mark entries in both sets of page tables as read-only
    - Page fault on write creates two copies

- **Usage Example: Zero Fill On Demand**
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

How is the translation accomplished?

- **What, exactly happens inside MMU?**
  - One possibility: Hardware Tree Traversal
    - For each virtual address, takes page table base pointer and traverses the page table in hardware
    - Generates a "Page Fault" if it encounters invalid PTE
    - Fault handler will decide what to do
    - More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
  - Another possibility: Software
    - Each traversal done in software
    - Pros: Very flexible
    - Cons: Every translation must invoke Fault!
  - In fact, need way to cache translations for either case!

Recall: Dual-Mode Operation

- Can a process modify its own translation tables?
  - NO!
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Privileged Level) from 0 - 3
    - PLO has full access, PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel processors only use PLO ("kernel") and PL3 ("user")

How to get from Kernel→User

- **What does the kernel do to create a new user process?**
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - Point at code in memory so program can execute
    - Possibly point at statically initialized data
  - Run Program:
    - Set machine registers
    - Set hardware pointer to translation table
    - Set processor status word for user mode
    - Jump to start of program
- **How does kernel switch between processes?**
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)
Recall: User→Kernel (System Call)
- Can’t let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?

• System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    » No! Only specific ones.
  - System call ID encoded into system call instruction
    » Index forces well-defined interface with kernel

System Call Continued
• What are some system calls?
  - I/O: open, close, read, write, lseek
  - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ...
  - Process: fork, exit, wait (like join)
  - Network: socket create, set options
• Are system calls constant across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
• What happens at beginning of system call?
  » On entry to kernel, sets system to kernel mode
  » Handler address fetched from table/Handler started
• System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel mem
    » User addresses must be translated!
    » Kernel has different view of memory than user
  - Every Argument must be explicitly checked!

User→Kernel (Exceptions: Traps and Interrupts)
• A system call instruction causes a synchronous exception (or “trap”)
  - In fact, often called a software “trap” instruction
• Other sources of Synchronous Exceptions (“Trap”):
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
• Interrupts are Asynchronous Exceptions
  - Examples: timer, disk ready, network, etc....
  - Interrupts can be disabled, traps cannot!
• On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
• Actual handler typically saves registers, other CPU state, and switches to kernel stack

Closing thought: Protection without Hardware
• Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
• Protection via Strong Typing
  - Restrict programming language so that you can’t express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
• Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can’t step out of bounds
    » Compiler puts in checks for every “dangerous” operation (loads, stores, etc.). Again, need special loader.
    » Alternative, compiler generates “proof” that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)
Summary (1/2)

- **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

- **Inverted page table**
  - Size of page table related to physical memory size

Summary (2/2)

- **PTE: Page Table Entries**
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)

- **Dual-Mode**
  - Kernel/User distinction: User restricted
  - User → Kernel: System calls, Traps, or Interrupts
  - Inter-process communication: shared memory, or through kernel (system calls)

- **Exceptions**
  - Synchronous Exceptions: Traps (including system calls)
  - Asynchronous Exceptions: Interrupts