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 \(\Rightarrow\) 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 \(T_3\)
    » …
    » \(T_n\) is waiting for a resource that is held by \(T_1\)

---

Can Priority-Inversion cause deadlock?

- Technically not – Consider this example:
  - 3 threads, \(T_1, T_2, T_3\) in priority order (\(T_3\) highest)
  - \(T_1\) grabs lock, \(T_3\) tries to acquire, then sleeps, \(T_2\) running
  - Will this make progress?
    » No, as long as \(T_2\) is running
    » But \(T_2\) 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?
  » \(T_3\) is prevented from running by \(T_2\)

- How does priority donation help?
  - Briefly raising \(T_1\) to the same priority as \(T_3\) \(\Rightarrow\) \(T_1\) can run and release lock, allowing \(T_3\) to run
  - Does priority donation involve taking lock away from \(T_1\)?
    » NO! That would break semantics of the lock!
Recall: Important Aspects of Memory Multiplexing

- **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

- **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

Assume 4-byte words

0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

Assume 4-byte words

0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

Assume 4-byte words

0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

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, stub, used to locate 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

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

**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
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
    - If user tries to access an illegal address, cause an error
    - During switch, kernel loads new base/limit from PCB (Process Control Block)
      - User not allowed to change base/limit registers

0x00000000 0xFFFFFFFF
Application1
0x00020000
BaseAddr=0x20000
LimitAddr=0x10000
Application2

Administrivia

- Midterm coming up in 1 week!
  - Wednesday 10/14, 6:30-9:30pm
  - In 145/155 Dwinelle
  - No class that day, extra office hours
- Details
  - Intend this to be a 2-hour exam in 3 hour slot
  - 1 page of hand-written notes, both sides
  - Closed book
  - Topics will include the material from that Monday
- Review Session
  - This Sunday, 2-4 PM, 306 Soda Hall

Recall: Address translation

- 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 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 Simple Example:
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)

<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 | 0x0000 |
| Seg ID = 1 | 0x4000 |
| 0x4800 | 0x5C00 |
| 0x0000 | 0x4800 |
| 0x0000 | 0x5C00 |

Physical Address Space

Example of segment translation

0x240 main: la $a0, varx
0x244 jal strlen
...
0x360 strlen: li $v0, 0 ;count
0x364 loop: lb $t0, ($a0)
0x368 beq $r0, $t1, done
...
0x4050 varx dw 0x314159

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 0x3248 → $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 more 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

Recall: General Address Translation

<table>
<thead>
<tr>
<th>Prog 1 Virtual Address Space 1</th>
<th>Prog 2 Virtual Address Space 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code</strong></td>
<td><strong>Code</strong></td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td><strong>Data</strong></td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td><strong>Heap</strong></td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td><strong>Stack</strong></td>
</tr>
</tbody>
</table>

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?

- 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

Virtual Memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Page Table</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x04</td>
<td>0x01</td>
<td>0x00000100</td>
</tr>
<tr>
<td>0x08</td>
<td>0x02</td>
<td>0x00000200</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x03</td>
<td>0x00000300</td>
</tr>
</tbody>
</table>

Virtual Address (Process A):

- Page Table Pointer A
  - 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):

- Page Table Pointer B
  - Page #0: V,R
  - Page #1: V,R
  - Page #2: V,R,W
  - Page #3: N
  - Page #4: V,R,W
  - Page #5: V,R,W

What about Sharing?

This physical page appears in address space of both processes.

Memory Layout for Linux 32-bit

Summary: Simple Page Table
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 can we handle sparseness?
  - Multi-level page table
- How about combining paging and segmentation?
  - Segments with pages inside them?
  - Also need some sort of multi-level translation

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
Summary: Two-Level Paging

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):

  What about Sharing (Complete Segment)?

- What must be saved/restored on context switch?
  - 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

**Pros:**
- 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

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

Inverted Page Table

Offset
Virtual Page #
Hash Table
Offset
Physical Page #

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

Making it real:

X86 Memory model with segmentation (16/32-bit)

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 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 < 213) entries
- Descriptor format (64 bits):
  - Granularity of segment (0: 16bit, 1: 4KiB unit)
  - Default operand size (0: 16bit, 1: 32bit)
  - Freely available for use by software
  - Segment present
  - Descriptor Privilege Level
  - System Segment (0: System, 1: code or data)
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”
  - P: 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 (Priviledge 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 $\rightarrow$ 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)

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!

Recall: User $\rightarrow$ 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 $\rightarrow$ 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

User $\rightarrow$ 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