Outline

• Multi-step processing of programs
• Virtual to physical address translation
  • Segment mapping
  • Page tables
  • Multi-level tables
  • Inverted page table
Virtualizing Resources

- Physical reality: Different processes/threads share same hardware
  - Need to multiplex CPU (done)
  - Need to multiplex memory (this lecture)
  - Need to multiplex disk and devices (later in term)
Memory Multiplexing Goals

- **Protection**: Prevent processes/threads from accessing others’ private data
  - Protect kernel data from user programs
  - Protect programs from themselves
  - Give special access permissions to different data
    (Read-only, read-and-write, invisible to user programs, etc.)

- **Controlled overlap**: Allow processes to share data
  - E.g., communication across processes, shared libraries
Some Terminologies

- **Physical memory**: data storage medium
- **Address space**: set of memory addresses
- **Virtual address space**: set of addresses generated by program
- **Physical address space**: set of physical addresses available on physical memory
THE BASICS: Address/Address Space

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

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

- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
  - $2^{20}B = 2^{18}\text{KB} = 1\text{MB (megabyte)}$
  - $2^{32}B = 2^{12}\text{MB} = 2^{2}\text{GB (gigabyte)}$
  - $2^{64}B = 2^{34}\text{GB} = 2^{24}\text{TB (terabyte)} = 2^{14}\text{PB (petabyte)} = 2^{4}\text{EB (exabyte)}$

"Things" here usually means “bytes” (8 bits)
Recall:
Address Space Layout of C Programs

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[]) {
    int *values;
    int i;
    values = (int *)malloc(sizeof(int)*5);
    for (i = 0; i < 5; i++)
        values[i] = i;
    return 0;
}
```
Recall: What Happens During Program Execution?

- Execution sequence
  - Fetch instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/memory
  - PC ← Next(PC)
  - Repeat

Data references:
Memory access on load/store instructions

Instruction references:
Memory access on every instruction

E.g., function calls, return, branches, etc.
Multi-Step Processing of Programs

- **Compiler**
  - Generate **object file** for each source code containing information about that source code
  - Has **incomplete** information, code can reference things from other codes
  - Doesn't know **addresses of external objects** when compiling each files
    - E.g., where is `printf` routine
  - Doesn't know where things it's compiling will go in memory

- **Linker**
  - Combines object files to **one single object file**
  - Arranges new **memory organization** for all pieces to fit together
  - Changes **addresses** for program to run under new organization
Multi-Step Processing of Programs (cont.)

- Originally all programs were **statically linked**
  - All external references are fully resolved, and program is complete
  - + Program startup is fast because it doesn’t need any further processing
  - − Object file becomes too large as it includes copy of all referenced libraries
  - − Physical memory is wasted, copies of same library exists in multiple programs
  - − To use new versions of libraries, entire program needs to be linked again

- Modern OS’s support **shared libraries** and **dynamic linking**
  - All processes share single copy of library code in physical memory
  - Each process will have its own copy of library global and static variables
  - On program startup, **dynamic linker** is invoked
  - If shared library is not currently in memory, it is brought into memory
  - Dynamic linker binds region of program’s virtual address to shared library
  - − Program startup could be slow because of extra processing at runtime
Side Note:
Shared Library Address Space

- **Problem**: shared libraries can't use **absolute addresses** for data references (why?)
  - Because different processes could bind same library to **different virtual address regions**

- **Solution**: shared libraries are compiled to be **Position-Independent Code (PIC)**
  - Code executes properly regardless of its absolute address

- **Data references from PIC** are made indirectly through **Global Offset Tables (GOT)**
  - GOT is located at fixed offset from code
  - GOT has one entry per global variable containing absolute address of the variable
  - Each variable is accessed using PC-relative offset to corresponding GOT entry
  - Each process has its own GOT

- **Instruction references** are made indirectly through **Procedure Linkage Table (PLT)** and GOT
Uniprogramming (No Translation or Protection)

- There is always only one program running at a time
- Program always runs at same place in physical memory
  - Virtual address space = physical address space
- Program can access any physical address

- Program is given illusion of dedicated machine by literally giving it one
Multiprogramming
(No Translation or Protection)

- To prevent address overlap between processes, loader/linker adjust addresses while programs are loaded into memory (loads, stores, jumps)
  - Virtual address = physical address

- Bugs in any program can cause other programs (including OS) to crash
Multiprogramming (Version with Protection)

• Can we protect programs from each other *without translation*?

Yes: use two special registers BaseAddr and LimitAddr
  • Prevent application from straying outside designated area
  • If application tries to access an illegal address, raise exception

• During switch, kernel loads new base/limit from PCB
  • User is not allowed to change base/limit registers
- **Address translation**: Map addresses from one address space to another
- Processor uses virtual addresses while memory uses physical addresses
  - Virtual address ≠ physical address
Ups and Downs of Virtual to Physical Address Translation

- + Code can be written, compiled, linked, loaded independently as if it has total unrestricted control of entire memory range (illusion)
  - Regardless of behavior or memory usage of any other program
- + OS can provide protection by mapping different virtual address spaces to different physical memory regions
  - If thread A cannot access thread B’s data, no way for A to adversely affect B
- + OS can allow memory sharing by mapping different virtual address regions to the same physical memory region

- − Address translation adds performance overhead
- − Address translation needs extra hardware support
  - Extra hardware consumes area and power
Recall: Address Translation with Base and Bound (B&B)

- Application is given illusion of running on its own dedicated machine, with memory starting at 0x00000000
- Program are mapped to continuous region of memory
- Virtual addresses do not change if program is relocated to different region of physical memory
Issues with B&B Method

- **Fragmentation** problem over time
  - Not every process is same size ⇒ memory becomes fragmented
- Missing support for inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
- Missing support for sparse address space for each process
  - Would like to have multiple segments (e.g., code, data, stack)
Multi-Segment Model

- Segment map resides in processor
  - Base is added to offset to generate physical address
- For each contiguous segment of physical memory there is one entry
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead
    - E.g., `mov ax, es:[bx]`
Intel x86 General-Purpose Registers

16-Bit Segment Addresses (Every 16 bytes in memory)

Segment registers specify which paragraph boundary begins a segment. Segment registers do not contain memory addresses per se!

You the programmer do not change code segments directly. "Long jump" instructions alter CS as needed.

Segments need not be all the same size, and they may overlap.

Much of memory is taken up by the operating system and various buffers and tables dedicated to its use.

20-Bit Memory Addresses

0FFFFFFH (IMB)
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

Physical Address Space

Virtual Address Space
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

Virtual Address Space

Physical Address Space

Might be shared
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

Virtual Address Space

Physical Address Space

Seg. ID = 0

Seg. ID = 1

Shared with Other Apps

Space for Other Apps

Might be shared
Example of Segment Translation (16-bit 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>0xFFFF</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

- Fetch \texttt{0x0240}
  - Virtual segment number? 0, offset? \texttt{0x240}
  - Physical address? Base: \texttt{0x4000}, so physical address: \texttt{0x4240}
  - Fetch instruction at \texttt{0x4240}, get \texttt{“la $a0, varx”}
  - Move \texttt{0x4050} to \texttt{$a0}, move PC+4 to PC

- Fetch \texttt{0x244}, translated to physical address: \texttt{0x4244}, get \texttt{“jal strlen”}
  - Move \texttt{0x0248} to \texttt{$ra} (return address!), move \texttt{0x0360} to PC

- Fetch \texttt{0x360}, translated to physical address: \texttt{0x4360}, get \texttt{“li $v0, 0”}
  - Move \texttt{0x0000} to \texttt{$v0}, move PC+4 to PC

- Fetch \texttt{0x364}, translated to physical address \texttt{0x4364}, get \texttt{“lb $t0, ($a0)”}
  - Since \texttt{$a0} is \texttt{0x4050}, try to load byte from \texttt{0x4050}
  - Translate \texttt{0x4050 (1000 0000 0011 0000)}: virtual segment #? 1, offset? \texttt{0x50}
  - Physical address? Base: \texttt{0x4800}, physical address: \texttt{0x4850}
  - Load byte from \texttt{0x4850} to \texttt{$t0}, move PC+4 to PC
Observations About Segmentation

- Virtual address space has holes
  - Segmentation is efficient for sparse address spaces
  - If program tries to access gaps, trap to kernel

- When is it OK to address outside valid range?
  - This is how stack and heap 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)
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

• Allocate physical memory in fixed-size chunks ("pages")
  
  • Can use simple *bit map* 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, big pages could lead to internal fragmentation
    
    • Typically have small pages (1K-16K)
  
• Consequently, each segment needs multiple pages
Implementation of Paging

- Page resides in physical memory
- Contains physical page and permission for each virtual page
- Offset from virtual address gets copied to physical address
  - E.g., **10-bit** offset ⇒ **1024-byte** pages
- Virtual page # is all remaining bits
  - E.g., **32-bits** v-address and **10-bit** offset ⇒ 22 bits for v-page # ⇒ 2²² entries in page table
- Physical page # is copied from table into physical address
Simple Page Table Example
(4-Byte Pages)
What About Sharing? (Single Pages)
Paging Example

Virtual Memory

Page Table

Physical Memory

stack

heap

data

code

page# offset

1111 1111 1111 0000

1100 0000

1000 0000

0100 0000

0000 0000

11111 11101 11100 11101 11110 null null

10010 10000

0011 0000

0001 0000

0000 0000

1110 0000

0111 000

0101 000

0011 000

0000 000
What happens if the stack grows to 1110 0000?
Paging Example (cont.)

Virtual Memory

- stack
- heap
- data
- code

Page Table

Physical Memory

Allocate new pages where room!

Challenge: Table size equal to # of pages in virtual memory!
Page Table Discussion

• What needs to be switched on a context switch?
  • Page table pointer and page table size

• Pros and cons?
  • + Simple memory allocation
  • + Easy to share
  • − Inefficient for sparse address spaces
    • There are too many unused page table entries
    • What if page size is very small?
      • With 1KB pages, we need $2^{22}$ (~4 million) table entries!
    • What if page size is too big?
      • Wastes space inside of page (internal fragmentation)
Two-Level Page Tables

- Tables fixed size (e.g., 1024 entries)
  - On context-switch: save single Page Table Pointer 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
Two-Level Paging Example

Virtual Memory

Physical Memory

Page Tables (level 1)

Page Tables (level 2)
Two-Level Paging Example (cont.)

In best case, total size of page tables $\approx$ number of pages used by program $\text{virtual memory}$. Requires two additional memory access!
Multi-level Translation: Segments + Pages

- What must be saved/restored on context switch?
  - Contents of top-level segment registers
Multilevel Paged Segmentation (x86)

- Global Descriptor Table (segment table)
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions

- Context switch
  - Change global descriptor table register (GDTR, pointer to global descriptor table)

- Multilevel page table
  - 32-bit: two level page table (per segment)
  - 64-bit: four level page table (per segment)
x86 32-bit Virtual Address

- 4KB pages; each level of page table fits in one page
x86 64-bit Virtual Address

- Fourth level table maps 2MB, and third level table maps 1GB of data
- If 2 MB covered by a fourth level table is contiguous in physical memory, then entry one third level can directly point to this region instead of pointing to a forth level page table
What About Sharing? (Entire Segment)

Process A's Virtual Address

| Seg # | Page # | Offset |

Segment Map

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>V/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 1</td>
<td>Bound 1</td>
<td>N</td>
</tr>
<tr>
<td>Base 2</td>
<td>Bound 2</td>
<td>V</td>
</tr>
<tr>
<td>Base 3</td>
<td>Bound 3</td>
<td>N</td>
</tr>
<tr>
<td>Base 4</td>
<td>Bound 4</td>
<td>V</td>
</tr>
</tbody>
</table>

Process B's Virtual Address

| Seg # | Page # | Offset |

Segment Map

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>V/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 1</td>
<td>Bound 1</td>
<td>N</td>
</tr>
<tr>
<td>Base 2</td>
<td>Bound 2</td>
<td>V</td>
</tr>
<tr>
<td>Base 3</td>
<td>Bound 3</td>
<td>N</td>
</tr>
<tr>
<td>Base 4</td>
<td>Bound 4</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P-Page #</th>
<th>Access</th>
<th>V/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>R</td>
<td>V</td>
</tr>
<tr>
<td>7</td>
<td>R/W</td>
<td>V</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Where is Memory Sharing Used?

- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments

- User-level system libraries (execute only)

- Shared segments between different processes
  - Must map page into same place in address space!

- In Linux, all processes share “kernel region” (before Meltdown patch!)
  - Every process has same page table entries
  - Process cannot access it at user level
  - On user to kernel mode 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?
32-bit Linux Memory Layout (Pre-Meltdown patch!)

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

program break brk
start_brk
Random brk offset

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char "gonzo = "God's own prototype";

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0xc0000000 == TASK_SIZE
0x08048000
Page Table Entry

- What is in each 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

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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 bit: Page has been modified recently
- L: L=1 ⇒ 4MB page
How PTE is Used?

- **Demand paging** (more on this later)
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid

- **Copy on write**
  - UNIX fork gives copy of parent address space to child
  - How to do this cheaply?
    - Make copy of parent's page tables
    - Mark entries in both sets of page tables as read-only
    - On write, page fault happens, OS creates two copies

- **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
Address Translation:
Hardware vs. Software

• Implement page tables in **hardware**
  • Generate “Page Fault” if encounter invalid PTE
    • Fault handler will decide what to do (more on this later)
  • Pros and cons?
    • + Relatively fast (but still many memory accesses!)
    • − Inflexible, complex hardware

• Implement page tables in **software**
  • Pros and cons?
    • + Very flexible
    • − Every translation must invoke Fault!

• In fact, we need a way to cache translations for either case!
Multi-level Translation Analysis

• Pros?
  • Allocate only as many page table entries as needed for application
    • In other words, sparse address spaces are easy
  • Easy memory allocation (bit-map 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, we can make each table to fit exactly one page
  • Two (or more, if >2 levels) lookups per reference
    • Seems very expensive!
In all previous methods (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 smaller.

Inverted page table fixes this problem by using a hash table.

- Size of hash table is related to size of physical memory not virtual address space.
- Very attractive option for 64-bit address spaces (e.g., PowerPC, UltraSPARC, IA64).

Cons?
- Complexity of managing hash chains: Often in hardware!
- Poor cache locality of page table.
Inverted Paging Example (cont.)

Virtual Memory

- stack
- heap
- data
- code

Physical Memory

- stack
- heap
- data
- code

Inverted Table
Hash(v-page#) = p-page#

- h(11111) = 11101
- h(11110) = 11100
- h(11101) = 10111
- h(11100) = 10110
- h(10010) = 10000
- h(10001) = 01111
- h(10000) = 01110
- h(01011) = 01101
- h(01010) = 01100
- h(01001) = 01011
- h(01000) = 01010
- h(00011) = 00101
- h(00010) = 00100
- h(00001) = 00011
- h(00000) = 00010

Total size of page table ≈ number of pages used by program in physical memory
# Address Translation Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Fast context switching: Segment mapping maintained by CPU</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Single-level paging</td>
<td>No external fragmentation, fast easy allocation</td>
<td>Large table size ~ virtual memory</td>
</tr>
<tr>
<td>Multi-level translation</td>
<td>Table size ~ # of pages in virtual memory, fast easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Inverted Table</td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
</tbody>
</table>
Some Simple Security Measures

• Address space randomization
  • Random start address makes it much harder for attacker to cause jump to code that it seeks to take over
  • Stack & Heap can start anywhere, so randomize placement
  • This requires position independent code (PIC)

• Kernel address space isolation
  • Don’t map entire kernel space into each process
  • Meltdown patch ⇒ map none of kernel into user mode!
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 # from virtual address mapped through page table to physical page #
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
Summary (2/2)

• **Multi-level tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space

• **Inverted page table**
  - Use of hash-table to hold translation entries
  - Size of page table ~ size of physical memory rather than size of virtual memory
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Ousterhout, Culler, Stoica, Silberschatz, Joseph, and Canny