SE350: Operating Systems

Lecture 10: Address Translation
Outline

• Multi-step processing of programs
• Virtual to physical address translation
  • Segment mapping
  • Page tables
  • Multi-level tables
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
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

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
Protection with Address Translation

- **Address translation**: Map addresses from one address space to another
- Processor uses virtual addresses while memory uses physical addresses
  - Virtual address $\neq$ 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)
• 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.

Address pointed to is SS:SP

Address pointed to is ES:DI

Address pointed to is DS:SI

Next instruction executed is at CS:IP

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:

物理地址空间：

虚拟地址空间：

虚拟地址格式：
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

Physical Address Space

Shared with Other Apps

Space for Other Apps

Might be shared
Example of Segment Translation (16b address)

0x0240 main: la $a0, varx
0x0244 jal strlen
...
0x0360 strlen: li $v0, 0 ;count
0x0364 loop: lb $t0, ($a0)
0x0368 beq $r0,$t0, done
...
0x4050 varx dw 0x314159

Seg ID # | Base  | Limit  |
---------|-------|--------|
0 (code) | 0x4000 | 0x0800 |
1 (data) | 0x4800 | 0x1400 |
2 (shared)| 0xF000 | 0x1000 |
3 (stack)| 0x0000 | 0x3000 |

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

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

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

- Fetch 0x0364, translated to physical address 0x4364, get “lb $t0, ($a0)”
  - Since $a0 is 0x4050, try to load byte from 0x4050
  - Translate 0x4050 (0100 0000 0101 000): virtual segment #? 1, offset? 0x50
  - Physical address? Base: 0x4800, physical address: 0x4850
  - Load byte from 0x4850 to $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
Multi-Segment Model

- 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 # ⇒ **2^{22}** entries in page table
- Physical page # is copied from table into physical address
Simple Page Table Example (4-Byte Pages)

Virtual Memory

Page Table

Physical Memory

0x00

0x04

0x06?

0x08

0x09?

0x00 0000 0000

0x04 0000 0100

0x06 0000 0100

0x08 0000 1000

0x09? 0000 1000

0x0C 0000 1100

0x0E! 0000 0100

0x10 0001 0000

0x00 0000 0000

0x04 0000 1100

0x08 0000 0100

0x0E! 0011 0000

0x05! 0000 0101

0x04 0000 0100

0x08 0000 1100

0x0E! 0000 0100

0x05! 0000 0101
What about Sharing?

Process A’s Virtual Address

<table>
<thead>
<tr>
<th>V-Page #</th>
<th>Offset</th>
<th>Page Table Pointer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>P-Page #</th>
<th>Access</th>
<th>V/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>R</td>
<td>V</td>
</tr>
<tr>
<td>5</td>
<td>R/W</td>
<td>V</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>P-Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-Page 1</td>
</tr>
<tr>
<td>P-Page 2</td>
</tr>
<tr>
<td>P-Page 3</td>
</tr>
<tr>
<td>P-Page 4</td>
</tr>
<tr>
<td>P-Page 5</td>
</tr>
</tbody>
</table>

Process B’s Virtual Address

<table>
<thead>
<tr>
<th>V-Page #</th>
<th>Offset</th>
<th>Page Table Pointer</th>
</tr>
</thead>
</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>N</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>P-Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-Page 1</td>
</tr>
<tr>
<td>P-Page 2</td>
</tr>
<tr>
<td>P-Page 3</td>
</tr>
<tr>
<td>P-Page 4</td>
</tr>
<tr>
<td>P-Page 5</td>
</tr>
<tr>
<td>..</td>
</tr>
</tbody>
</table>
Summary: Paging

Virtual Memory

- stack
- heap
- data
- code

Page Table

- 1111
- 1110
- 11101
- 11100
- 11101
- 11100
- null

Physical Memory

- stack
- heap
- data
- code
What happens if stack grows to 1110 0000?
Summary: Paging

Virtual Memory

- stack
- heap
- data
- code

Page Table

Physical Memory

- stack
- data

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

Allocate new pages where room!
Page Table Discussion

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

• Pros
  • Simple memory allocation
  • Easy to share

• Con:
  • What if page size is very small?
    • With 1K pages, we need $2^{22}$ (≈4 million) table entries!
  • What if table too big?
    • Wastes space inside of page (internal fragmentation)

• How about multi-level paging or combining paging and segmentation?
Two-Level Page Table

- 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
Summary: Paging

Virtual Memory

- stack
- heap
- data
- code

Physical Memory

- stack
- heap
- data
- code

Page Tables

Page Table (level 1)

Page Table (level 2)
In best case, total size of page tables $\approx$ number of pages used by program 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
What about Sharing?
(Complete Segment)

Process A’s Virtual Address

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

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>

Segment Map

<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>
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, previous example keeps tables to exactly one page in size
  • Two (or more, if >2 levels) lookups per reference
    • Seems very expensive!
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:
  • Address same format previous slide (10, 10, 12-bit offset)
  • Intermediate page tables called “Directories”

<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 (PTE only): page has been modified recently
• L: $L=1 \Rightarrow 4\text{MB page (directory only)}$

• Bottom 22 bits of virtual address serve as offset
How is Translation Accomplished?

- What, exactly happens inside MMU?

- One possibility: Hardware Tree Traversal
  - For each virtual address traverses the page table in hardware
  - Generates “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 physical memory
  - Must be restricted somehow

- For 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
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 # 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

- Multi-level tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
Questions?
Acknowledgment

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