To support multiprogramming, we need “Protection”

- Kernel vs. user mode
- What is an address space?
- How to implement it?

<table>
<thead>
<tr>
<th>Physical memory</th>
<th>Abstraction: virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>No protection</td>
<td>Each program isolated from all others and from the OS</td>
</tr>
<tr>
<td>Limited size</td>
<td>Illusion of “infinite” memory</td>
</tr>
<tr>
<td>Sharing visible to programs</td>
<td>Transparent --- can’t tell if memory is shared</td>
</tr>
</tbody>
</table>
The big picture

- To support multiprogramming with protection, we need:
  - dual mode operations
  - translation between virtual address space and physical memory
- How to implement the translation?

Address translation

- Goals
  - implicit translation on every memory reference
  - should be very fast
  - protected from user’s faults
- Options
  - Base and Bounds
  - Segmentation
  - Paging
  - Multilevel translation
  - Paged page tables
Base and Bounds

Each program loaded into contiguous regions of physical memory. Hardware cost: 2 registers, adder, comparator.

Base and Bounds (cont’d)

- Built in Cray-1
- A program can only access physical memory in \([\text{base, base+bound}]\)
- On a context switch: save/restore base, bound registers
- Pros: Simple
- Cons: fragmentation; hard to share (code but not data and stack); complex memory allocation
Segmentation

- **Motivation**
  - separate the virtual address space into several segments so that we can share some of them if necessary
- **A segment is a region of logically contiguous memory**
- **Main idea: generalize base and bounds by allowing a table of base&bound pairs**
  
  (assume 2 bit segment ID, 12 bit segment offset)

<table>
<thead>
<tr>
<th>virtual segment #</th>
<th>physical segment start</th>
<th>segment size</th>
</tr>
</thead>
<tbody>
<tr>
<td>code (00)</td>
<td>0x4000</td>
<td>0x700</td>
</tr>
<tr>
<td>data (01)</td>
<td>0</td>
<td>0x500</td>
</tr>
<tr>
<td>- (10)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>stack (11)</td>
<td>0x2000</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

Segmentation (cont’d)

- Have a table of (seg, size)
- Protection: each entry has
  - (nil,read,write)
- On a context switch: save/restore the table or a pointer to the table in kernel memory
- Pros: efficient, easy to share
- Cons: complex management and fragmentation within a segment
Segmentation example

(assume 2 bit segment ID, 12 bit segment offset)

<table>
<thead>
<tr>
<th>v-segment #</th>
<th>p-segment start</th>
<th>segment size</th>
</tr>
</thead>
<tbody>
<tr>
<td>code 00</td>
<td>0x4000</td>
<td>0x700</td>
</tr>
<tr>
<td>data 01</td>
<td>0</td>
<td>0x500</td>
</tr>
<tr>
<td>- 10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>stack 11</td>
<td>0x2000</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

Virtual memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6ff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14ff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3ff</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

physical memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4ff</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Segmentation example (cont’d)

Virtual memory for strlen(x)

Main: 240
244
248
24c
... strlen: 360
... 420
... x: 1108
... physical memory for strlen(x)

Main: 4240
4244
4248
424c
... strlen: 4360
... 4420
...
Paging

- **Motivations**
  - both branch bounds and segmentation still require fancy memory management (e.g., first fit, best fit, re-shuffling to coalesce free fragments if no single free space is big enough for a new segment)
  - can we find something simple and easy

- **Solution**
  - allocate physical memory in terms of fixed size chunks of memory, or **pages**.
  - Simpler because it allows use of a bitmap

$$00111110000001100 \quad \text{--- each bit represents one page of physical memory}$$

1 means allocated, 0 means unallocated

Paging (cont’d)

- Use a page table to translate
- Various bits in each entry
- Context switch: similar to the segmentation scheme
- What should be the page size?
- Pros: simple allocation, easy to share
- Cons: big page table and cannot deal with internal fragmentation easily
Paging example

Segmentation with paging

Each segment has its own page table!
Two-level paging

A logical address (on 32-bit machine with 4K page size) is divided into:
- a page number consisting of 20 bits.
- a page offset consisting of 12 bits.

Since the page table is paged, the page number is further divided into:
- a 10-bit page number.
- a 10-bit page offset.

Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table.
Segmentation with paging - Intel 386

As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.
How many PTEs do we need?

- Worst case for 32-bit address machine
  - \# of processes \times 2^{20} (if page size is 4096 bytes)

- What about 64-bit address machine?
  - \# of processes \times 2^{52}

Summary: virtual memory mapping

- What?
  - separate the programmer’s view of memory from the system’s view

- How?
  - translate every memory operation using table (page table, segment table).
  - Speed: cache frequently used translations

- Result?
  - each user has a private address space
  - programs run independently of actual physical memory addresses used, and actual memory size
  - protection: check that they only access their own memory
Summary (cont’d)

- **Goal:** multiprogramming with protection + illusion of “infinite” memory

- **Today’s lecture so far:**
  - HW-based approach for protection: dual mode operation + address space
  - Address translation: virtual address -> physical address

- **Future topics**
  - how to make address translation faster? use cache (TLB)
  - demand paged virtual memory

- **The rest of today’s lecture:**
  - x86 specifics & project overview

Debugging as engineering

- **Much of your time in this course will be spent debugging**
  - In industry, 50% of software dev is debugging
  - Even more for kernel development

- **How do you reduce time spent debugging?**
  - Produce working code with smallest effort

- **Optimize a process involving you, code, computer**
**Debugging as science**

- Understanding -> design -> code
  - not the opposite
- Form a hypothesis that explains the bug
  - Which tests work, which don’t. Why?
  - Add tests to narrow possible outcomes
- Use best practices
  - Always walk through your code line by line
  - Module tests - narrow scope of where problem is
  - Develop code in stages, with dummy replacements for later functionality

---

**x86 abstract model**

- I/O: Communicating data to and from devices
- CPU: Logic for performing computation
- Memory: Storage
x86 CPU/memory interaction

- Memory stores instruction and data
- CPU interprets instructions

x86 implementation

- EIP points to next instruction
- Incremented after each instruction
- x86 instructions are not fixed length
- EIP modified by CALL, RET, JMP, and conditional JMP
x86 general purpose registers (GPR)

<table>
<thead>
<tr>
<th>16-bits</th>
<th>8-bits</th>
<th>8-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AX</td>
<td>AL</td>
</tr>
<tr>
<td>EBX</td>
<td>BX</td>
<td>BH</td>
</tr>
<tr>
<td>ECX</td>
<td>CX</td>
<td>CH</td>
</tr>
<tr>
<td>EDX</td>
<td>DX</td>
<td>DH</td>
</tr>
<tr>
<td>EDI</td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>ESI</td>
<td></td>
</tr>
</tbody>
</table>

- Temporary registers
- Contents may be changed by instructions
- Contents not changed by interrupts / exceptions / traps
- EDI/ESI used by string ops but also as GPR

x86 memory models

- **Real mode** with segmentation (16-bit mode)
  - Used by early OSes
  - All x86 still boots in Real Mode for “compatibility” reasons
  - You can only use 1MB memory (4-bit segment + 16-bit address)
    \[
    \text{PhysicalAddress} = \text{segment} \times 16 + \text{offset}
    \]

- **Protected mode** w. segmentation & paging (32-bit)
  - 4GB memory
  - Segmentation done via GDT (Global Descriptor Table)
    * A code segment descriptor holding a base address
    * A data segment descriptor holding a base address
    * A TSS segment descriptor …
x86 segmentation registers

- 8086 registers 16-bit w/20-bit bus addresses

- Solution: segment registers
  - CS: code segment, EIP
  - SS: stack segment, ESP and EBP
  - DS: data segment, register mem ops
  - ES: string segment, string ops

- Linear address computation:
  - EIP => CS:EIP = 0x8000:0x1000 = 0x81000
  - ESP => SS:ESP = 0xF800:0x1000 = 0xF9000
  - (EAX) => DS:EAX = 0xC123:0x1000 = 0xC2230

x86 real mode

- 8086 16-bit with 20-bit address bus
- Stored in segment registers CS, DS, ES, FS

- Logical address: segment:offset

- Physical address: segment*0x10 + offset
x86: the runtime stack

- Additional (temporary) storage
- Stack registers --- 32-bits long
- ESP – stack pointer
- EBP – base pointer

x86 EFLAGS register
Using EFLAGS register

- Lots of conditional jumps
  en.wikibooks.org/wiki/X86_Assembly/Control_Flow

```
    mov $5, %ecx
    mov $5, %edx
    cmp %ecx, %edx # ZF = 1
    je equal
    ...
    equal:
    ...
```

**x86 assembly**

We will use AT&T syntax

```c
int main(void)
{
    return f(3) + 1;
}

int f(int x)
{
    return x + 4;
}
```

```
_main:
    pushl %ebp        # prologue
    movl %esp, %ebp
    pushl $3           # body
    call _f
    addl $1, %eax
    movl %ebp, %esp
    popl %ebp
    ret

_f:
    pushl %ebp        # don’t clobber registers
    movl %esp, %ebp
    pushl %ebx         # access argument
    addl $4, %ebx
    movl %ebx, %eax
    popl %ebx          # restore
    movl %ebp, %esp   # epilogue
    popl %ebp
    ret
```
x86 memory layout

CS422/522 Lab 1: Bootloader & Physical Memory Management (due 9/22/2015)

- Learn how to use git
- Part 1: PC Bootstrap
  - x86 assembly & QEMU & BIOS
- Part 2: Bootloader
  - Learn how to use QEMU & GDB & read ELF file
- Part 3: Physical Memory Management
  - The MATIntro Layer
  - The MATInit Layer
  - The MATOp Layer
- Enrichment (optional)