Virtual Memory and Page Tables

for the optional part of P3
Agenda

- What is the problem?
  - What is virtualization?
  - Implement virtual memory
    - mechanism #1: software TLB + PMP
    - mechanism #2: page table translation
  - Further discussion: how to read a code repository?
Put the code at address 0x .... and stack at address 0x ....

compiles

Adobe

Ps

Put the code at address 0x .... and stack at address 0x ....

compiles

Google

Chrome
OS suggests a standard layout

- 32-bit Windows standard layout
- It is good to follow this standard, but also possible not to follow
- There are many standards in the systems industry
  - POSIX, ISO OSI, etc.
Good to follow the memory layout standard!
But the problem is

- The code section (program image) starts at \textit{0x0040_0000}
- In \textit{physical memory}, the OS can only put the code section of \textit{one process} at this address
  - But OS runs \textit{many processes}!
- Introduce \textit{virtual memory} will help
Agenda

• What is the problem?

What is virtualization?

• Implement virtual memory
  • mechanism #1: software TLB + PMP
  • mechanism #2: page table translation

• Further discussion: how to read a code repository?
An important concept:

One-to-many virtualization
A computer has 3 mandatory pieces
Scheduler is **virtualizing** the CPU

one physical CPU
→ an illusion of many virtual CPU
Virtual Memory

one physical memory

→ an illusion of many virtual memory

Virtual memory address space #1

Virtual memory address space #2

Virtualize
File system is **virtualizing** the Disk

one **physical** disk

→ an illusion of

many **virtual** disks (files)
Take-away

operating system = virtual CPU + virtual memory + virtual disk

All are one-to-many virtualization here.
Further topics: 3 Types of Virtualization

- **One-to-many** virtualization
  - e.g., operating system

- **Many-to-one** virtualization
  - e.g., RAID, Spark

- **A-to-B** virtualization
  - e.g., VMware Workstation, Windows Subsystem Linux
Agenda

• What is the problem?
• What is virtualization?

Implement virtual memory

• mechanism #1: software TLB + PMP
• mechanism #2: page table translation

• Further discussion: how to read a code repository?
// Allocate a physical memory page
int (*mmu_alloc)(int* frame_no, void** cached_addr);

// Free a physical memory page
int (*mmu_free)(int pid);

// The physical memory roughly looks like:

OS code, stack  Metadata for alloc/free  Pages to be allocated or freed
Virtual Memory Interface #2

// Map a virtual page in the address space of pid
// to a physical page (here called frame);
// Useful when creating a new process, such as zoom
int (*mmu_map)(int pid, int page_no, int frame_no);

// Switch the address space to pid;
// Useful when switching the context to a process
int (*mmu_switch)(int pid);
Just a **brief recap** of last week’s class

- What is the problem?
- What is virtualization?
- Implement virtual memory
  - mechanism #1: software TLB + PMP
  - mechanism #2: page table translation
- Further discussion: how to read a code repository?
RUNNING and Runnable

- Page * 3: code/data/heap of the RUNNING process
  
  0x0800_5000

- Page * 2: stack of the RUNNING process
  
  0x8000_0000

- All pages of all Runnable processes
  
  0x8000_4000

Memory buffer
When creating a process

Allocate 5 pages in the memory buffer and load the code/data of the new process. `mmu_alloc()` and `mmu_map()` are involved.
mmu\_switch() step1 in yield()

Write these 5 pages of the previously running process to the memory buffer.

All pages of all RUNNABLE processes
Load the 5 pages of the next running process from the memory buffer.

mmu_switch() step2 in yield()
Agenda

• What is the problem?
• What is virtualization?
• Implement virtual memory
  • mechanism #1: software TLB + PMP
  ➔ mechanism #2: page table translation
• Further discussion: how to read a code repository?
For every process

Allocate 5 pages for code, stack, etc.
And in addition, allocate some more pages, say 3, as page tables.
Example: 0x8000_1234 → 0xabcd_1234

0x8000_1234 is the virtual address of the process
0xabcd_1234 is the physical address in the memory
Break down address 0x8000_1234

VPN[1] is 0x200, or 10_0000_0000 in binary
VPN[0] is 0x001, or 00_0000_0001 in binary
Offset is 0x234

Translate to 0xabcd_1234

Page 79 of RISC-V manual, volume2, v1.10
Translate Step #1

<table>
<thead>
<tr>
<th></th>
<th>VPN[1]</th>
<th>VPN[0]</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 4.16: Sv32 virtual address.

VPN[1] is 0x200, or 10_0000_0000 in binary
VPN[0] is 0x001, or 00_0000_0001 in binary
Offset is 0x234

Translate to 0xabcd_1234

The page table base CSR (satp) stores the physical address of
Translate Step #2

VPN[1] is 0x200, or 10_0000_0000 in binary
VPN[0] is 0x001, or 00_0000_0001 in binary
Offset is 0x234

Translate to 0xabcd_1234

Entry 0x200 of page table #1 stores the physical address of page table #2
Translate Step #3

Figure 4.16: Sv32 virtual address.

VPN[1] is 0x200, or 10_0000_0000 in binary
VPN[0] is 0x001, or 00_0000_0001 in binary
Offset is 0x234

Translate to 0xabcd_1234

Entry 0x001 of page table #2 stores the physical address 0xabcd_1000
Translate Step #4

**Figure 4.16: Sv32 virtual address.**

VPN[1] is 0x200, or 10_0000_0000 in binary  
VPN[0] is 0x001, or 00_0000_0001 in binary  
Offset is 0x234

Translate to **0xabcd_1234**

0xabcd_1000 plus offset 0x234 gives **0xabcd_1234**
Page table #3 is not used in this example

VPN[1] is 0x200, or 10_0000_0000 in binary
VPN[0] is 0x001, or 00_0000_0001 in binary
Offset is 0x234

Translate to 0xabcd_1234

But page table #3 may be used when translating some other virtual addresses
Homework

• Read section 4.1.11 and 4.3
  • 4.1.11 introduces the satp register
  • 4.3 introduces the Sv32 translation process
• P3 will be due on Nov 4.
• Next lecture: disk driver and file system
Agenda

- What is the problem?
- What is virtualization?
- Implement virtual memory
  - mechanism #1: software TLB + PMP
  - mechanism #2: page table translation

➡️ Further discussion: how to read a code repository?