Put Projects P0-P2 Together

— the first 3 steps of building an operating system
Step 1-3 of building an OS

- Step #1: understand computer architecture
  - memory and context
  - function call and calling convention
- Step #2: understand interrupt and exception
- Step #3: understand context-switch and multi-threading
Before building an OS, you have

Computer

Hardware documents
Two important documents

From the **CPU** vendor

From the **computer** vendor
What are the **registers and instructions** supporting an operating system?

How to **control devices**?
Required reading

• A reading assignment is released on CMSx
  • RISC-V and SiFive documents
  • because they are simpler and shorter than Intel/Dell
Chapter 4 of the memory map

CPU debug @0000_0000 (ignore this for building an OS)

Device control @0200_0000 (ignore this for building an OS)

Boot ROM @0200_0000

Main memory @0x8000_0000

(main memory ≤2GB in this architecture)
provides a **hello world**

**Step#1**: compile the hello-world program in Linux

**Step#2**: copy the compiled code to the boot ROM with special tools provided by SiFive

- CPU debug
- Device control
- Boot ROM
- Main memory
Enter the context of hello world

Step#3: Press the boot button on the computer

Step#4: CPU set instruction pointer to the beginning of boot ROM

Step#5: An `li` instruction sets the stack pointer to main memory
hello-world prints to screen

Step#6: a call instruction calls main(), which calls printf()

Step#7: during printf(), store instructions will send data to the screen

Step#8: the screen shows “Hello World!”

CPU debug
Device control
Boot ROM
Main memory
Step 1-3 of building an OS

➡️ Step #1: understand computer architecture
  ✓ memory and context
    • function call and calling convention
  • Step #2: understand interrupt and exception
  • Step #3: understand context-switch and multi-threading
## Calling convention in RISC-V

Page 137, table 25.1 of RISC-V manual, volume1

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5</td>
<td>t0</td>
<td>Temporary/alternate link register</td>
<td>Caller</td>
</tr>
<tr>
<td>x6–7</td>
<td>t1–2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>

References:

Function call step #1

<main>::
    ...
    Store caller-saved registers on the stack
    Call printf (set ra to the address of )
    ➡️ Restore caller-saved registers
    ...

<printf>::
    Store callee-saved registers on the stack
    ...
    Restore callee-saved registers
    Return to main() (set pc to ra)
<main>:
  ...
  Store **caller-saved** registers on the stack
  Call printf (set ra to the address of ➤)
  ➤Restore caller-saved registers
  ...

<printf>:
  Store callee-saved registers on the stack
  ...
  Restore callee-saved registers
  Return to main() (set pc to ra)
Function call step#3

<main>:
  ...
  Store **caller-saved** registers on the stack
  Call printf (set ra to the address of)
  ➤ Restore caller-saved registers
  ...

<printf>:
  Store **callee-saved** registers on the stack
  ...
  Restore callee-saved registers
  Return to main() (set pc to ra)
<main>:
  ...
  Store **caller-saved** registers on the stack
  Call printf (set ra to the address of )
  Restore caller-saved registers
  ...

<printf>:
  Store **callee-saved** registers on the stack
  ...
  Restore **callee-saved** registers
  Return to main() (set pc to ra)
Function call step #5

<main>:

... Store caller-saved registers on the stack
Call printf (set ra to the address of )

Restore caller-saved registers
...

<printf>:

Store callee-saved registers on the stack
...

Restore callee-saved registers
Return to main() (set pc to ra)
Function call step #6

<main>:
   ...  
   Store caller-saved registers on the stack
   Call printf (set ra to the address of)
   ➡️ Restore caller-saved registers
   ...  

<printf>:
   Store callee-saved registers on the stack
   ...  
   Restore callee-saved registers
   Return to main() (set pc to ra)
Step 1-3 of building an OS

✿ Step #1: understand computer architecture
  ✿ memory and context
  ✿ function call and calling convention

• Step #2: understand interrupt and exception
• Step #3: understand context-switch and multi-threading
Can we do more than hello world?

Yes, add a simple timer handler.
Step 1-3 of building an OS

- Step #1: understand computer architecture
- **Step #2: understand interrupt and exception**
  - control and status registers (CSR)
  - inserting a call to the handler function
- Step #3: understand context-switch and multi-threading
Control and status registers (CSR)

- There are many registers other than the 32 user-level ones:
  - `misa`: 32-bit or 64-bit?
  - `mhartid`: the core ID number
  - `mstatus`: the machine status
  - `mtvec`, `mie`, `mtime`, `mtimecmp`: interrupt handling
Recap: timer interrupt

- How to register an interrupt handler?
  - write the address of handler function to $\text{mtvec}$
- How to set a timer?
  - write ($\text{mtime} + \text{quantum}$) to $\text{mtimecmp}$
- How to enable timer interrupt?
  - set certain bit of $\text{mstatus}$ and $\text{mie}$ to 1
int quantum = 50000;

void handler() {
    earth->tty_info("Got timer interrupt.");
    mtimecmp_set(mtime_get() + quantum);
}

int main() {
    earth->tty_success("A timer interrupt example.");

    asm("csrw mtvec, %0 ::"r"(handler));
    mtimecmp_set(mtime_get() + quantum);  // Set a timer

    int mstatus, mie;
    asm("csrr %0, mstatus" : "=r"(mstatus));
    asm("csrw mstatus, %0 ::"r"(mstatus | 0x8));
    asm("csrr %0, mie" : "=r"(mie));
    asm("csrw mie, %0 ::"r"(mie | 0x80));

    while(1);
}
Question

How does the RISC-V processor
insert a call to the handler function?
Recall the main-printf example

<some user function>: // instead of main()

  . . .
  Store caller-saved registers on the stack
  Call handler (set ra to the address of)
  Restore caller-saved registers
  . . .

<handler>: // instead of printf()

  Store callee-saved registers on the stack
  . . .
  Restore callee-saved registers
  Return to some_user_function() with ra
Intuition: CPU inserts these code

<some user function>:

...  
Store caller-saved registers on the stack  
Call handler (set ra to the address of )  
Restore caller-saved registers  
...

<handler>:

Store callee-saved registers on the stack  
...  
Restore callee-saved registers  
Return to some_user_function() with ra
Cleanup these code

<some user function>:
  . . .
  Store caller-saved registers on the stack
  Call handler (set ra to the address of <handler>)
  Restore caller-saved registers
  . . .

<handler>:
  Store all registers on the stack
  . . .
  Restore all registers
  Return to some_user_function() with ra
Handler returns to the same context

<some user function>:
  ...
  Call handler (set ra to the address of )
  ...

<handler>:
  Store all registers on the stack
  ...
  Restore all registers
  Return to some_user_function() with ra
Question
How does the handler function switch to the context to a different thread?
First, replacing `ra` with CSR `mepc`

```plaintext
<some user function>:
  ...
  // mepc: machine exception program counter
  Call handler (set mepc to the address of )
  ...

<handler>:
  Store all registers on the stack
  ...
  Restore all registers
  Return to some_user_function() with mepc
```
Then, switch context with **mepc**

<some user function>:

...  
Call handler (set mepc to the address of another thread)  
...

<handler>:

Store all registers on the stack  
...

Set mepc to the code section of another thread  
Restore all registers  
Switch to another thread with mepc
Brief summary

• The interrupt handler function
  • Stores all register on stack, instead of callee-saved
  • Uses mret and mepc instead of ret and ra
• This is why, in the demo code, there is one line:
  • void handler() __attribute__((interrupt));
  • telling the compiler this function is an interrupt hander
A demo using **mepc** and **mret**

```c
void thread0() { while(1) { // print something green } }
void thread1() { while(1) { // print something yellow } }

int next_thread = 0;
void handler() {
    next_thread = 1 - next_thread;
    asm("csrw mepc, %0 ::"r"((next_thread == 0)? thread0 : thread1));

    mtimecmp_set(mtime_get() + quantum); // reset timer
    asm("li sp, 0x80002000"); // set stack pointer
    asm("mret"); // forget previous thread and start a new thread
}
```
Demo on a RISC-V board

https://github.com/yhzhang0128/egos-2000/tree/timer_example/grass

demo code in microSD card

earth layer code in boot ROM

1. Load demo from microSD
2. Print strings to the screen
Step 1-3 of building an OS

- Step #1: understand computer architecture
- Step #2: understand interrupt and exception
  - control and status registers (CSR)
  - inserting a function call to the handler
- Step #3: understand context-switch and multi-threading
Can we do more than timer handler?

Yes, add thread init/create/switch/exit just like P1.
Step 1-3 of building an OS

- Step #1: understand computer architecture
- Step #2: understand interrupt and exception
- **Step #3: understand context-switch and multi-threading**
  - skipped in this lecture since you just finished P1
  - read the grass kernel (358 lines of code)
Homework

• Read
  • the two CPU documents posted on CMSx
  • the 358 lines of code mentioned in the previous slide
• P2 is due on Oct 12.
• No class on Oct. 7 — enjoy your fall break.
After the fall break

- CS4411 has 12 lectures and we have finished half.
  - Step #1: understand computer architecture
  - Step #2: understand interrupt and exception
  - Step #3: understand context-switch and multi-threading
  - Step #4: understand privilege levels
  - Step #5: understand I/O devices
  - Step #6: understand file systems