Kernel and Privilege Levels
Review: kernel ≈ 3 handlers

```c
void kernel() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 3) { syscall(); }
        if (id == 7) { yield(); }
    } else {
        fault();
    }
}
```
First step of P2, system call

```c
void kernel() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 3) { syscall(); }
        if (id == 7) { yield(); }
    } else {
        fault();
        // Exceptions 8-11 are triggered by ecall
        if (id >= 8 && id <= 11) { syscall(); }
    }
}
```
Understanding `ecall`

- Review RISC-V function call
  - Understand interrupt handler call
  - Understand the RISC-V instruction `ecall`
Say `main()` calls `printf()`

`<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 #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)
<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>x5–7</td>
<td>t0–2</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>

Function call step #2

<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)
<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>x5–7</td>
<td>t0–2</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>

Function call step #3

<main>:

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

<stdio>: 
Store callee-saved registers on the stack
... Restore callee-saved registers
Return to main() (set pc to ra)
<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>x5–7</td>
<td>t0–2</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>

Function call step #4

<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)
In particular, \texttt{ra} is restored here

\begin{verbatim}
<main>:
  ...
  Store caller-saved registers on the stack
  Call printf (set ra to the address of \texttt{printf})
  \textbf{Restore the ra register}
  ...

<printf>:
  Store callee-saved registers on the stack
  ...
  Restore callee-saved registers
  Return to main() (set pc to ra)
\end{verbatim}
Understanding \texttt{ecall}

- Review RISC-V \texttt{function call}

- Understand \texttt{interrupt handler call}

- Understand the RISC-V instruction \texttt{ecall}
Problem #1
If an interrupt happens during `main()`, the CPU will call `handler()`, but the compiler can’t predict it and store registers on `main()` stack.
Address problem #1

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

<handler>:
  Store ALL registers on the handler stack
  ...
  Restore ALL registers
  Return to main() with ra
If `main()` calls `handler()` directly

**<main>:**

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

**<handler>:**

```
Store ALL registers on the handler stack
... 
Restore ALL registers
Return to main() with ra
```
But **`handler()`** is inserted by CPU

<main>:

... Store caller-saved registers on the stack
CPU inserts a call to handler
  (set `ra` to the address of )
Restore caller-saved registers

- - - `ra` will be a different value at this point!

<handler>:
Store ALL registers on the handler stack
...
Restore ALL registers
Return to main() with `ra`
Problem #2
How to keep the value of $ra$?
Introducing the **mepc** CSR

<main>:

... Store caller-saved registers on the stack

CPU inserts a call to handler

| set mepc to the address of → |

Restores caller-saved registers

| → · · · ra is still the same value at this point! |

<handler>:

Store ALL registers on the handler stack

... Restore ALL registers

Return to main() with mepc
Understanding *ecall*

- Review RISC-V function call
- Understand interrupt handler call

→ Understand the RISC-V instruction *ecall*
Call `handler()` by triggering an exception

<some user function>:

```
...  

ecall // Triggers exception 8 or 11
  ... // CPU inserts a call to handler
```

<handler>:

```
...  

// handle the system call
// read value of mepc (the value of )
// write value+4 to mepc (the next instruction)
mret // similar to ret but uses mepc instead of ra
```
Summary of `ecall`

- Review the RISC-V calling convention
- Understand interrupt handler call
  - Solve problem #1: save all registers on the handler stack
  - Solve problem #2: use `mepc/mret` instead of `ra/ret`
- Understand the `ecall` instruction
  - triggering `exception` 8 or 11 for system call
Adding privilege levels

8.2.1 Interrupt Entry and Exit

When an interrupt occurs:

- The value of mstatus.MIE is copied into mcause.MPIE, and then mstatus.MIE is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in mstatus.MPP.
- The current pc is copied into the mepc register, and then pc is set to the value specified by mtvec as defined by the mtvec.MODE described in Table 19.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. Interrupts can be re-enabled by explicitly setting mstatus.MIE or by executing an MRET instruction to exit the handler. When an MRET instruction is executed, the following occurs:

- The privilege mode is set to the value encoded in mstatus.MPP.
- The global interrupt enable, mstatus.MIE, is set to the value of mcause.MPIE.
- The pc is set to the value of mepc.
When an interrupt occurs:

- The value of mstatus.MIE is copied into mcause.MPIE, and then mstatus.MIE is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in mstatus.MPP.
- The current pc is copied into the mepc register, and then pc is set to the value specified by mtvec as defined by the mtvec.MODE described in Table 19.

![Machine-mode status register (mstatus) for RV32.](image)
When invoking **mret**

When an **MRET** instruction is executed, the following occurs:

- **The privilege mode is set to the value encoded in** $\text{mstatus.MPP}$.
- **The global interrupt enable,** $\text{mstatus.MIE}$, **is set to the value of** $\text{mcause.MPIE}$.
- **The pc is set to the value of** $\text{mepc}$.

<table>
<thead>
<tr>
<th>SD</th>
<th>WPRI</th>
<th>TSR</th>
<th>TW</th>
<th>TVM</th>
<th>MXR</th>
<th>SUM</th>
<th>MPRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.6: Machine-mode status register (**mstatus**) for RV32.
Switching privilege level

Kernel can modify these 2 bits

- The privilege mode is set to the value encoded in \texttt{mstatus.MPP}.
- The global interrupt enable, \texttt{mstatus.MIE}, is set to the value of \texttt{mcause.MPIE}.
- The \texttt{pc} is set to the value of \texttt{mepc}.

Figure 3.6: Machine-mode status register (\texttt{mstatus}) for RV32.
In proc_yield()

static void proc_yield() {
    . . .
    if (curr_pid >= GPID_USER_START) {
        /* Modify mstatus.MPP to user mode */
        . . .
    } else {
        /* Modify mstatus.MPP to machine mode */
        . . .
    }
    . . .
}
Homework

• Handle system call with the `ecall` instruction
  • i.e., `asm("ecall")`

• P2 will be due on Mar 24, but likely extended next week