System Call

System call is the interface between users and the OS for system services.

Case study: EGOS system call

- Define data structures
- Invoke a system call in an application
- Handle a system call in the OS kernel

Defining data structures

```
struct syscall {
    enum syscall_type type;
    struct sys_msg msg;
    int retval;
};
```

```
enum syscall_type {
        SYS_UNUSED,
        SYS_RECV,
        SYS_SEND,
        SYS_NCALLS
};
```

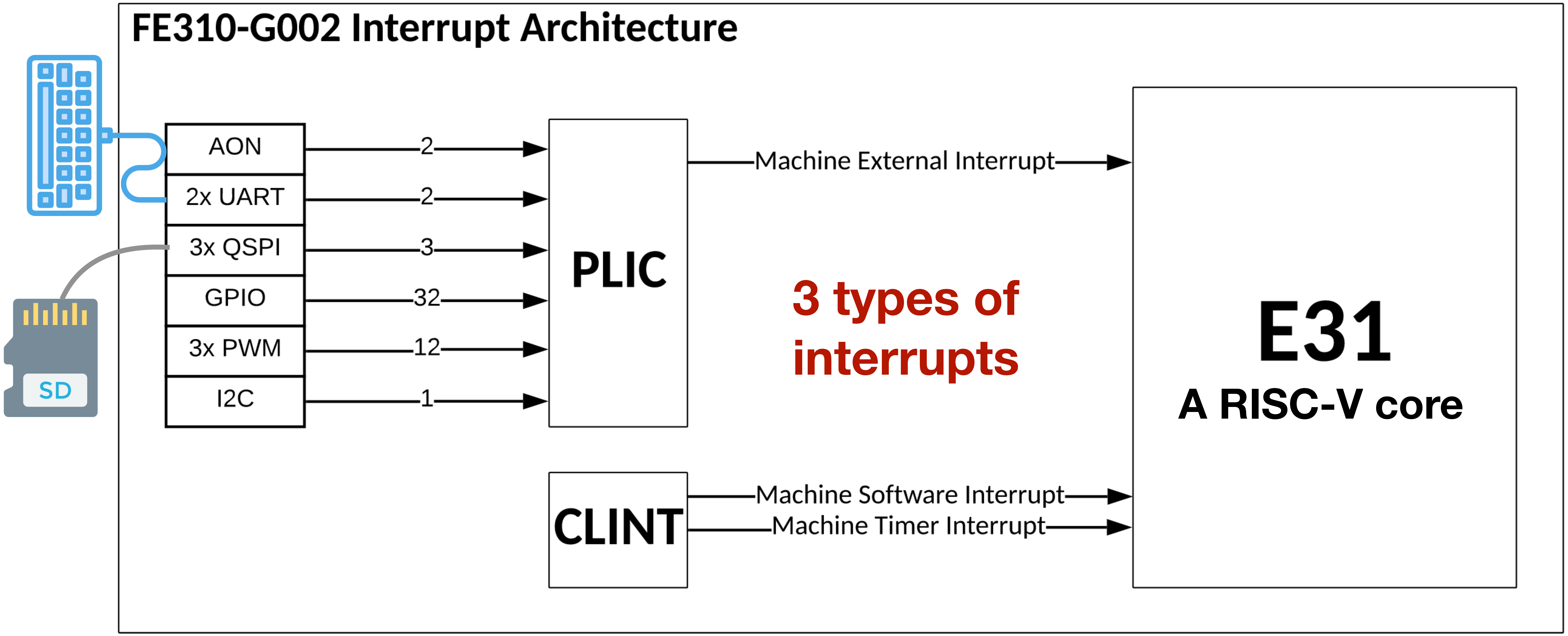
There are only 2 system calls in EGOS: send and receive messages between processes.

```
struct sys_msg {
    int sender;
    int receiver;
    char content[SYSCALL_MSG_LEN];
};
```

Case study: EGOS system call

- Define data structures
- ➔ Invoke a system call in an application
- Handle a system call in the OS kernel

Review: CPU support for interrupts



Page 38 of Sifive FE310 manual, v19p04
https://github.com/yhzhang0128/egos-2000/blob/timer_example/references/sifive-fe310-v19p04.pdf

CLINT: Core-Local Interrupt

Software → 0x2000000

Timer → 0x2004000, 0x200bff8

Address	Width	Attr.	Description
0x2000000	4B	RW	msip for hart 0
0x2004008			Reserved
…			
0x200bff7			
0x2004000	8B	RW	mtimecmp for hart 0
0x2004008			Reserved
…			
0x200bff7			
0x200bff8	8B	RW	mtime
0x200c000			Reserved

Page 38 of Sifive FE310 manual, v19p04
https://github.com/yhzhang0128/egos-2000/blob/timer_example/references/sifive-fe310-v19p04.pdf

Invoking SYS_SEND step#1

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;
```

OS and user application agree on a memory address for the system call data structure.

```
static void sys_invoke() {
    *((int*)0x2000000) = 1;
}

int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

Invoking SYS_SEND step#2

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;

static void sys_invoke() {
    *((int*)0x2000000) = 1;
}
// The sys_send function takes 3 parameters
int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

Invoking SYS_SEND step#3

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;

static void sys_invoke() {
    *((int*)0x2000000) = 1;
}

int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;
    // Prepare the system call data structure
    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

Invoking SYS_SEND step#4

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;

static void sys_invoke() {
    *((int*)0x2000000) = 1;  // Trigger a software interrupt
}

int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

Case study: EGOS system call

- Define data structures
- Invoke a system call in an application
- ➔ Handle a system call in the OS kernel

Software interrupt is #3

Interrupt Exception Codes		
Interrupt	**Exception Code**	**Description**
1	0–2	Reserved
1	3	Machine software interrupt
1	4–6	Reserved
1	7	Machine timer interrupt
1	8–10	Reserved
1	11	Machine external interrupt
1	≥ 12	Reserved

Page 42 of Sifive FE310 manual, v19p04
https://github.com/yhzhang0128/egos-2000/blob/timer_example/references/sifive-fe310-v19p04.pdf

Review: kernel ≈ 3 handlers

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

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 3) { syscall_handler(); }
        if (id == 7) { timer_handler(); } // scheduler
    } else {
        fault_handler();
    }
}
```

`syscall_handler` for `SYS_SEND`

- Possibility #1 (blocking)
  - Wait until the receiver calls `sys_recv()`
  - Similar to the wait in P1's semaphore implementation
- Possibility #2 (non-blocking)
  - Maintain a message queue in TCB (i.e., `struct process`)
  - Add the message to receiver's message queue and return

EGOS case study summary

- Define data structures
  - ~16 lines in grass/syscall.h
- Invoke a system call in an application
  - ~32 lines in grass/syscall.c
- Handle a system call in the OS kernel
  - ~75 lines in grass/scheduler.c

Go and read the code!

Homework

- P3 has been released on CMSx.
  - Software interrupt is NOT the usual way of invoking system call and there is a special `ecall` instruction.
  - You will implement this usual way in P3.

Understanding `ecall`

- ➔ Review RISC-V function call
- Review interrupt handler call
- Understand the RISC-V instruction `ecall`

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)
```

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)
```

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)
```

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)
```

Understanding `ecall`

- Review RISC-V function call
- ➔ Review interrupt handler call
- Understand the RISC-V instruction `ecall`

Problem #1

If an interrupt happens during `main()`, the compiler didn't know about it.

i.e., compiler cannot store/resume registers with the `main()` stack.

Address problem #1

<some user function>:

    . . .
    ~~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 stack
    . . .
    Restore all registers
    Return to some_user_function() with ra

Problem #2

How to restore the value of `ra`?

The problem is explained with 2 bullets in the next slide.

How to restore the value of ra?

<some user function>:

. . .

~~Store caller-saved registers on the stack~~

Call handler (set ra to the address of ➡)

~~Restore caller-saved registers~~

➡ . . .

2. Previously, the ra register was restored here.

<handler>:

Store all registers on the stack

. . .

Restore all registers

Return to some_user_function() with ra

1. The ra register is needed here for ret.

Address problem #2: the mepc CSR

<some user function>:

. . .

~~Store caller-saved registers on the stack~~

Call handler (set mepc to the address of ➡)

~~Restore caller-saved registers~~

➡ . . .

<handler>:

Store all registers on the stack

. . .

Restore all registers, including the ra register

Return to some_user_function() with mepc

Understanding `ecall`

- Review RISC-V function call
- Review interrupt handler call
- ➔ Understand the RISC-V instruction `ecall`

Call handler with a special instruction

```
<some user function>:
    . . .
    ecall // Triggers an exception, CPU calls handler
    . . .

<handler>:
    . . .
    // handle the system call
    // set mepc+=4, i.e, the instruction after ecall
    mret   // return to some user function with mepc
```

Summary of `ecall`

- Review RISC-V function call
  - separating caller-saved and callee-saved registers
- Review interrupt handler call
  - address problem #1: save all registers on the callee stack
  - address problem #2: use `mepc + mret` instead of `ra + ret`
- Understand the RISC-V instruction `ecall`
  - a special instruction triggering an exception as system call

Note

We haven’t talked about privilege levels:
It is possible to provide system services without protection.

And we will talk about protections next week.

Homework

- P3 has been released on CMSx.
  - You will implement
    - system call, memory protection and exception handling
  - We give one RISC-V board to each team, see post on Ed.
- Next lecture: memory protection and translation