System Calls

Key concepts from last lecture

- Control and status registers
- Interrupt and exception handling
- Kernel $\approx$ 3 handlers

Recap: During OS initialization

```
void kernel() {
    ...
}

void os_init() {
    ...
    // Register kernel() as the
    // interrupt/exception handler.
    asm("csrw mtvec, %0" ::”r"(kernel));
    ...
}
```

Recap: Kernel ≈ 3 handlers

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

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 7) { yield(); } // P1: multi-threading
    } else {
        // P2: system call and memory protection
        if (id == 8 || id == 11) { syscall_handler(); }
                            else { fault_handler(); }
    }
}
```

- ➡ A high-level picture of system calls
- A concrete implementation of system calls
- Starting P2: Invoking system calls with ecall

Consider `apps/user/cat.c`

```
make qemu                                                        ⌥⌘2
[INFO] App file size: 0x00002770 bytes
[INFO] App memory size: 0x00002fc8 bytes
[SUCCESS] Enter kernel process GPID_FILE
[INFO] sys_proc receives: Finish GPID_FILE initialization
[INFO] Load kernel process #3: sys_dir
[INFO] App file size: 0x00000fa4 bytes
[INFO] App memory size: 0x00001bb0 bytes
[SUCCESS] Enter kernel process GPID_DIR
[INFO] sys_proc receives: Finish GPID_DIR initialization
[INFO] Load kernel process #4: sys_shell
[INFO] App file size: 0x000006d0 bytes
[INFO] App memory size: 0x00000ed0 bytes
[CRITICAL] Welcome to the egos-2000 shell!
➜ /home/yunhao cat README
With only 2000 lines of code, egos-2000 implements boot loader, microSD driver,
tty driver, memory paging, address translation, interrupt handling, process sche
duling and messaging, system call, file system, shell, 7 user commands and the `
mkfs/mkrom` tools.
➜ /home/yunhao
```

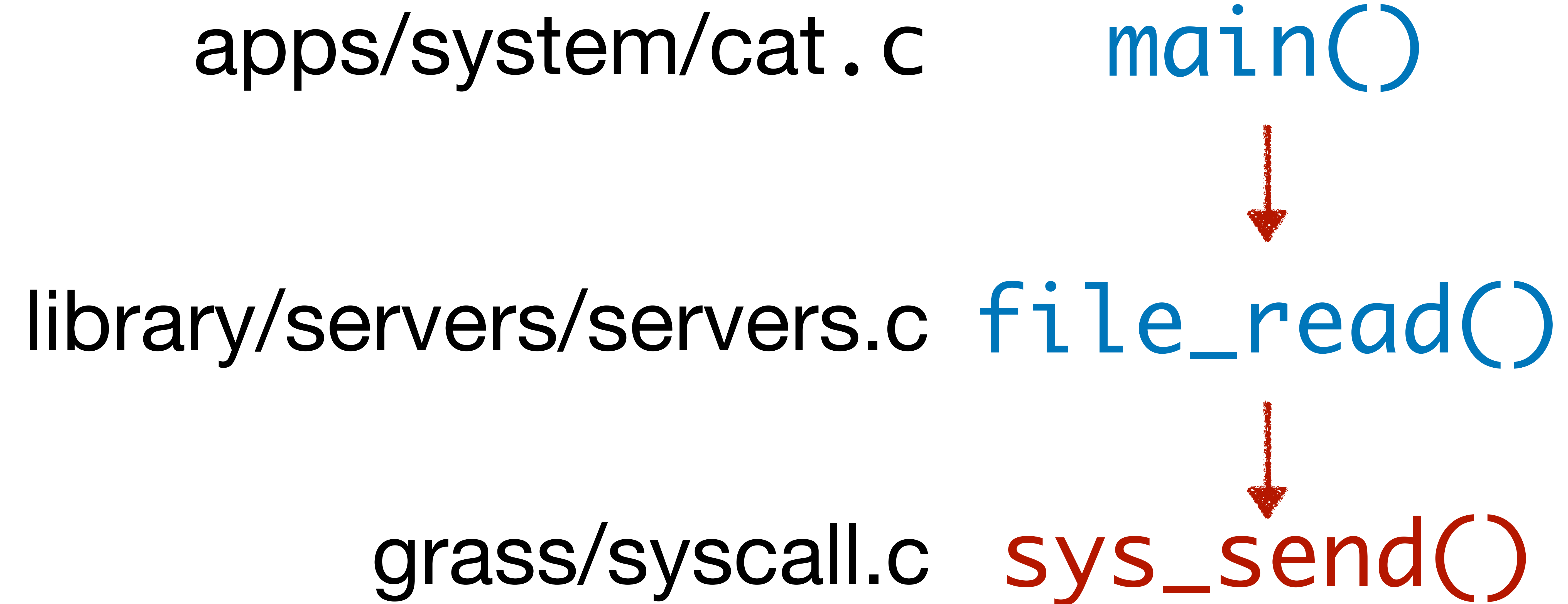
Cat invokes `file_read()`

```
13    int main(int argc, char** argv) {
14        if (argc == 1) {
15            INFO("usage: cat [FILE]");
16            return -1;
17        }
18
19        /* Get the inode number of the file */
20        int file_ino = dir_lookup(grass->workdir_ino, argv[1]);
21        if (file_ino < 0) {
22            INFO("cat: file %s not found", argv[1]);
23            return -1;
24        }
25
26        /* Read and print the first block of the file */
27        char buf[BLOCK_SIZE];
28        file_read(file_ino, 0, buf);
29        printf("%s", buf);
30        if (buf[strlen(buf) - 1] != '\n') printf("\r\n");
31
32        return 0;
33    }
```

Step1. File server waits for requests

Process #1

Step2. Cat sends a request for file content



Step3. Kernel handles the IPC

Process #1 (sys_file)

```
main()
  ↓
sys_recv()
```

Process #2 (cat)

```
main()
  ↓
file_read()
  ↓
sys_send()
```

Step4a. File server reads file from disk

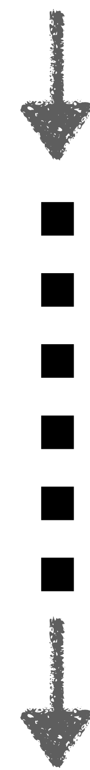
Process #1

Process #2

main()

apps/system/cat.c

main()



file_read()

disk_read()

sys_send()

Grass kernel (grass/kernel.c)

Step4b. Cat waits for the file content

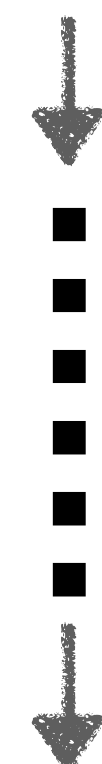
Process #1

Process #2

main()

apps/system/cat.c

main()



file_read()

sys_recv()

disk_read()

Grass kernel (grass/kernel.c)

Step5. File server returns the file content

Process #1 (sys_file)

Process #2 (cat)

main() → sys_send()

apps/system/cat.c

main() → file_read() → sys_recv()

Inter-process Communication (IPC)

Grass kernel (grass/kernel.c)

- A high-level picture of system calls
- ➡ A concrete implementation of system calls
- Starting P2: Invoking system calls with ecall

Data structures for system calls

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

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

See header file grass/syscall.h

File server invoking sys_recv

```
static void sys_invoke() {
    *((int*)0x2000000) = 1;  // Trigger a software interrupt
}                            // which is interrupt #3

int sys_recv(int* sender, char* buf, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_RECV;
    sys_invoke();
    memcpy(buf, sc->msg.content, size);
    if (sender) *sender = sc->msg.sender;
    return sc->retval;
}
```

Kernel calls system call handler

```
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(); }
    } else {
        fault_handler();
    }
}
```

Kernel blocks the file server process

```
void syscall_handler() {
    ...
    // Block the file server process
    // until it receives a message,
    // which is similar to sema_dec() in P1.
    ...
}
```

App invoking sys_send

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

a well-known memory address

```
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;
}
```

App invoking sys_send

```
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;
}
```

App invoking sys_send

```
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;
}
```

App invoking sys_send

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

static void sys_invoke() {
    *((int*)0x2000000) = 1;  // Trigger a software interrupt
}                            // which is interrupt #3

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;
}
```

Kernel calls system call handler

```
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(); }
    } else {
        fault_handler();
    }
}
```

Kernel unblocks the file server process

```
void syscall_handler() {
    ...
    // Copy the message from the memory
    // of cat to the memory of sys_file.
    // Then unblock the file server process.
    ...
}
```

File server is unblocked

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

int sys_recv(int* sender, char* buf, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_RECV;
    sys_invoke();
    memcpy(buf, sc->msg.content, size);
    if (sender) *sender = sc->msg.sender;
    return sc->retval;
}
```

Parse the message as a request

```
char buf[SYSCALL_MSG_LEN];

while (1) {
    int sender, r;
    struct file_request *req = (void*)buf;
    r = grass->sys_recv(&sender, buf, SYSCALL_MSG_LEN);

    switch (req->type) {
    case FILE_READ:
        ... // read a file from disk
    case FILE_WRITE:
        ... // write to a file on disk
    }
}
```

File server handles the request

```
char buf[SYSCALL_MSG_LEN];

while (1) {
    int sender, r;
    struct file_request *req = (void*)buf;
    r = grass->sys_recv(&sender, buf, SYSCALL_MSG_LEN);

    switch (req->type) {
    case FILE_READ:
        ... // read a file from disk (project P3 & P4)
    case FILE_WRITE:
        ... // write to a file on disk (project P3 & P4)
    }
}
```

- A high-level picture of system calls
- A concrete implementation of system calls

➔ Starting P2: Invoking system calls with ecall

Modify sys_invoke

```
// Previously
static void sys_invoke() {
    *((int*)0x2000000) = 1;   // Trigger a software interrupt
}                             // which is interrupt #3

// Now
static void sys_invoke() {
    asm(“ecall”);             // Trigger an ecall exception
}                             // which is exception #8 / #11
```

System call is exception #8, #11

	Interrupt Exception Codes		
	Interrupt	**Exception Code**	**Description**
Interrupts	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
Exceptions	0	0	Instruction address misaligned
	0	1	Instruction access fault
	0	2	Illegal instruction
	0	3	Breakpoint
	0	4	Load address misaligned
	0	5	Load access fault
	0	6	Store/AMO address misaligned
	0	7	Store/AMO access fault
	0	8	Environment call from U-mode
	0	9–10	Reserved
	0	11	Environment call from M-mode
	0	≥ 12	Reserved

Page 42 of Sifive FE310 manual, v19p04

https://github.com/yhzhang0128/egos-2000/blob/main/references/sifive-fe310-v19p04.pdf

Next lecture: Memory protection

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

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        // P1: multi-threading
        if (id == 7) { yield(); }
    } else {
        // P2: system call and memory protection
        if (id == 8 || id == 11) { syscall_handler(); }
                            else { fault_handler(); }
    }
}
```

Homework

- P2 is due on Oct 20
- Handle system calls using `ecall`
- Next lecture: privilege levels and protection
- Please remember to fill up the mid-term evaluation!