System Calls
Key concepts from previous lectures

• Control and status registers
• Memory-mapped I/O
• Interrupt and exception handling
• Privilege levels
• Memory protection
Recap: Manage Hardware

• Manage CPU: Control and Status Register (CSR)

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

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

- Manage IO: Memory-Mapped I/O (MMIO)
  - Timer
  - Software interrupt

**mtimecmp_set()** writes 8 bytes to

**mtime_get()** reads 8 bytes from

<table>
<thead>
<tr>
<th>Address</th>
<th>Width</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000000</td>
<td>4B</td>
<td>RW</td>
<td>msip for hart 0</td>
</tr>
<tr>
<td>0x2004008</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200bff7</td>
<td>8B</td>
<td>RW</td>
<td>mtimecmp for hart 0</td>
</tr>
<tr>
<td>0x2004000</td>
<td>8B</td>
<td>RW</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x2004008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200bff7</td>
<td>8B</td>
<td>RW</td>
<td>mtime</td>
</tr>
<tr>
<td>0x200bff8</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>0x200c000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recap: Manage Hardware

- Manage IO: Memory-Mapped I/O (MMIO)
- Timer
- Software interrupt

<table>
<thead>
<tr>
<th>Address</th>
<th>Width</th>
<th>Attr.</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000000</td>
<td>4B</td>
<td>RW</td>
<td>msip for hart 0</td>
<td>MSIP Registers (1 bit wide)</td>
</tr>
<tr>
<td>0x2004008</td>
<td></td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200bff7</td>
<td>8B</td>
<td>RW</td>
<td>mtimecmp for hart 0</td>
<td>MTIMECMP Registers</td>
</tr>
<tr>
<td>0x2004000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200bfff</td>
<td></td>
<td></td>
<td></td>
<td>Timer Register</td>
</tr>
<tr>
<td>0x200c000</td>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 24: CLINT Register Map

9.2 MSIP Registers

Machine-mode software interrupts are generated by writing to the memory-mapped control register msip. Each msip register is a 32-bit wide WARL register where the upper 31 bits are tied to
Recap: Privilege Levels

https://minnie.tuhs.org/CompArch/Lectures/week05.html
Recap: Memory Protection

- Memory region

- Configuration
  - R/W/X
  - Address matching: TOR, NAPOT, etc
Today: System Call

- A way for applications to request **services** from the OS.
  - E.g., read/write disks, access NICs, inter-process communication (IPC)

- How
  - Invoke OS kernel by SWI or ECALL
Monolithic kernel vs Microkernel

https://ece-research.unm.edu/jimp/310/slides/linux_driver1.html
Agenda

- A high-level picture of system calls
  - A concrete implementation of system calls
  - Starting P2: Invoking system calls with ecall
apps/user/cat.c

make qemu

[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 scheduling and messaging, system call, file system, shell, 7 user commands and the mkfs/mkrom tools.

→ /home/yunhao
int main(int argc, char** argv) {
    if (argc == 1) {
        INFO("usage: cat [FILE]".);
        return -1;
    }

    /* Get the inode number of the file */
    int file_ino = dir_lookup(grass->workdir_ino, argv[1]);
    if (file_ino < 0) {
        INFO("cat: file %s not found", argv[1]);
        return -1;
    }

    /* Read and print the first block of the file */
    char buf[ BLOCK_SIZE ];
    file_read(file_ino, 0, buf);
    printf("%s", buf);
    if (buf[strlen(buf) - 1] != '\n') printf("\n");
    return 0;
}
Step1. File server waits for requests

Process #1

apps/system/sys_file.c  main()

grass/syscall.c  sys_recv()
Step 2. Cat sends a request for file content

Process #2

Apps/user/cat.c  main()

library/servers/servers.c  file_read()

grass/syscall.c  sys_send()
Step 3. Kernel handles the IPC

Process #1 (sys_file)

main()

sys_recv()

Receive:
Read from X file

Inter-process Communication (IPC)
Grass kernel (grass/kernel.c)

Process #2 (cat)

main()

file_read()

sys_send()

Send to FileServer:
Read from X file
Step 4a. File server reads file from disk

Process #1

main()

\[\text{disk\_read(\()\text{grass/kernel.c}\text{)}\]

Process #2

apps/system/cat.c

\[\text{main(\()\]

\[\text{file\_read(\()\]

\[\text{sys\_send(\()\]

Grass kernel (grass/kernel.c)
Step4b. Cat waits for the file content

Process #1

main()  
↓  
disk_read()  

Process #2

apps/system/cat.c

main()

↓

file_read()

↓

sys_recv()

Grass kernel (grass/kernel.c)
Step 5. File server returns the file content

Process #1 (sys_file)

main()  

sys_send()  

Inter-process Communication (IPC)

Grass kernel (grass/kernel.c)

Process #2 (cat)

main()  

file_read()  

sys_recv()  

apps/system/cat.c
• A high-level picture of system calls

arrow\ 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

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
    }
}
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;
}
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 calls system call handler
Kernel **blocks** the file server process

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

```c
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;
}
```
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

```c
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

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

→ 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
P2 – Three Tasks

• Use `ecall` to handle system calls
• Assign privilege level to processes
• Add memory protect to four memory regions
Virtual Memory

• What: allowing multiple processes to use the same (virtual) address
  • Both Cat and FileServer can use address 0xaabb

• How
  1. OS buffers memory for application processes
  2. Page table
Back to our Cat & SysFile

Process #1 (sys_file)

main()  
sys_recv()

Receive:
Read from X file into BUF

Inter-process Communication (IPC)
Grass kernel (grass/kernel.c)

Process #2 (cat)

main()  
file_read()  
sys_send()

Send:
Read from X file into BUF
Software TLB

static void proc_send(struct syscall *sc) {
    ...  
    /* Copy message from sender to kernel stack */  
    struct sys_msg tmp;  
    earth->mmu_switch(curr_PID);  
    memcpy(&tmp, &sc->msg, sizeof(tmp));  
    
    /* Copy message from kernel stack to receiver */  
    earth->mmu_switch(receiver);  
    memcpy(&sc->msg, &tmp, sizeof(tmp));  
    
    /* Set receiver process as runnable */  
    proc_set Runnable(receiver);  
    ...
}

int soft_tlb_switch(int pid) {
    ...
    /* Unmap curr_vm_pid from the user address space */  
    for (int i = 0; i < NFRAMES; i++)  
        if (table[i].use && table[i].pid == curr_vm_pid)  
            paging_write(i, table[i].page_no);

    /* Map pid to the user address space */  
    for (int i = 0; i < NFRAMES; i++)  
        if (table[i].use && table[i].pid == pid)  
            memcpy((void*)(table[i].page_no << 12), paging_read(i, 0), PAGE_SIZE);
    ...
}
Homework

• P2 is due on March 15th.

• Please remember to fill up the mid-term evaluation!

• Come to office hours for help.