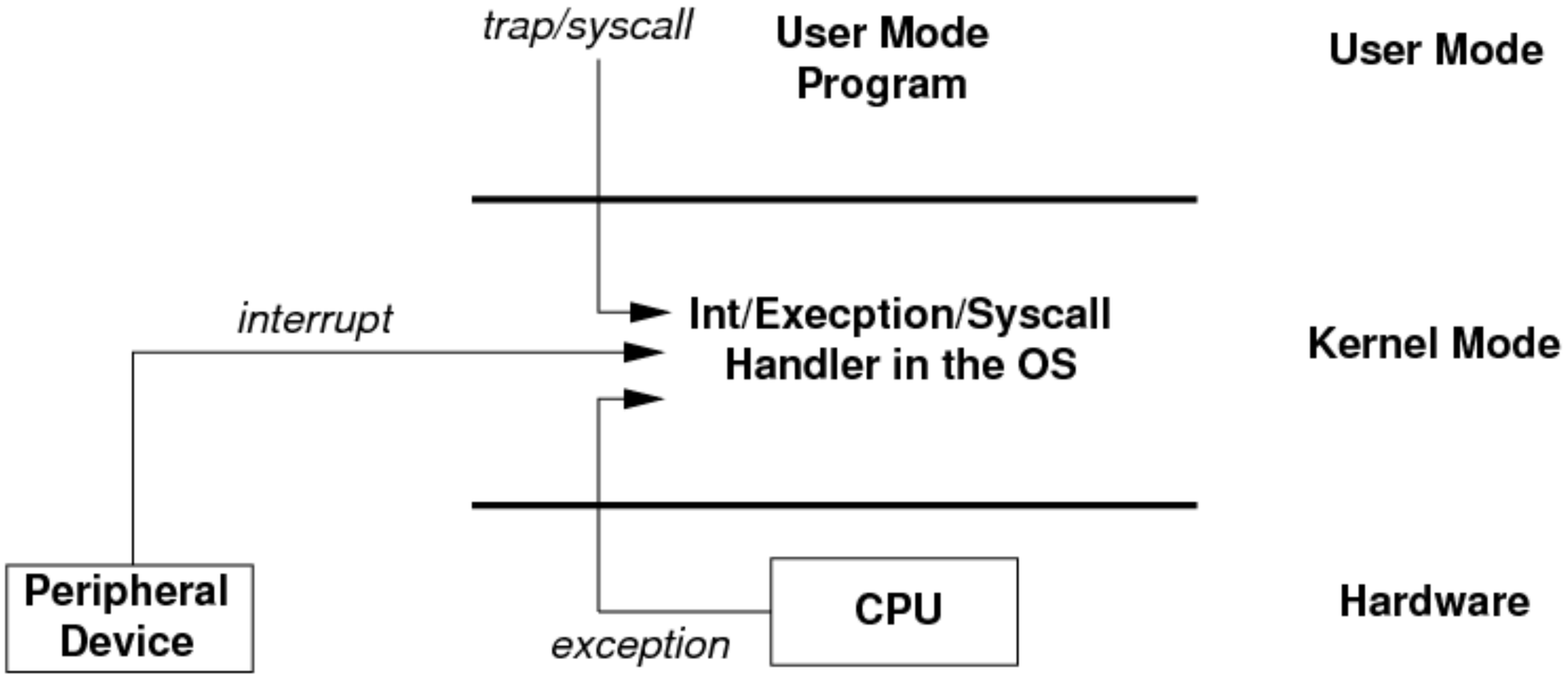
System Calls

Yu-Ju Huang, slides adapted from Yunhao Zhang

Privilege mode



https://minnie.tuhs.org/CompArch/Lectures/week05.html

Recap: Manage Hardware

CSR: control and status registers

- **Setting up timer**
  - **mtvec**: Machine Trap-Vector Base-Address Register
  - **mstatus**: Machine Status Register
    - bit#3: enable interrupt
  - **mie**: Machine Interrupt Enable Register
    - bit#7: enable timer interrupt

```c
void intr_init(uint core_id) {
    /* Initialize the timer */
    earth->timer_reset = timer_reset;
    mtimecmp_set(0x0FFFFFFFFFFFFFFFUL, core_id);

    /* Setup the interrupt/exception handling entry */
    void trap_entry(); /* (see grass/kernel.s) */
    asm("csrw mtvec, %0" ::"r"(trap_entry));
    INFO("Use direct mode and put the address of the trap_entry into mtvec");

    /* Enable timer interrupt */
    asm("csrw mip, %0" ::"r"(0));
    asm("csrs mie, %0" ::"r"(0x80));
    asm("csrs mstatus, %0" ::"r"(0x88));
}
```

```c
static void intr_entry(uint id) {
    if (id == INTR_ID_TIMER) {
        proc_yield();
        return;
    }

    ...

    FATAL("excp_entry: kernel got interrupt %d", id);
}

static void proc_yield() {
    /* Student's code goes here (Preemptive Scheduler). */
    if (!CORE_IDLE && curr_status == PROC_RUNNING) proc_set_runnable(curr_pid);
    int next_idx = MAX_NPROCESS;
    for (uint i = 1; i <= MAX_NPROCESS; i++) {
        struct process* p = &proc_set[(curr_proc_idx + i) % MAX_NPROCESS];
        if (p->status == PROC_PENDING_SYSCALL) proc_try_syscall(p);

        if (p->status == PROC_READY || p->status == PROC_RUNNABLE) {
            next_idx = (curr_proc_idx + i) % MAX_NPROCESS;
            break;
        }
    }

    ...

    curr_proc_idx = next_idx;
    proc_set_running(curr_pid);
}
```

```asm
trap_entry:
    /* Step1: acquire the kernel lock (only for P8)
     * Step2: switch to the kernel stack
     * Step3: save all the registers on the kernel stack
     * Step4: call kernel_entry()
     * Step5: restore all the registers
     * Step6: switch back to the process stack
     * Step7: release the kernel lock (only for P8)
     * Step8: invoke mret and return to the process context */

    /* Step1 */
    /* Student's code goes here (Multicore & Locks). */
    /* Acquire the kernel lock and make sure not to modify any registers, */
    /* so you may need to use sscratch just like how Step2 uses mscratch. */

    /* Student's code ends here. */

    /* Step2 */
    csrw mscratch, sp
    li sp, 0x80400000

    /* Step3 */
    addi sp, sp, -128 /* sp == SAVED_REGISTER_ADDR */
    sw a0,  0(sp)
    ...
    sw t0,  120(sp)   /* t0 holds the value of the old sp before trap_entry */

    /* Step4 */
    csrr a0, mcause
    call kernel_entry

    /* Step5 */
    lw a0,  0(sp)
    ...
    lw tp,  116(sp)

    /* Step6 */
    lw sp,  120(sp)

    mret
```

```c
void kernel_entry(uint mcause) {
    /* With the kernel lock, only one core can enter this point at any time */
    asm("csrr %0, mhartid" : "=r"(core_in_kernel));

    /* Save process context */
    asm("csrr %0, mepc" : "=r"(proc_set[curr_proc_idx].mepc));
    memcpy(proc_set[curr_proc_idx].saved_register, SAVED_REGISTER_ADDR,
           SAVED_REGISTER_SIZE);

    (mcause & (1 << 31)) ? intr_entry(mcause & 0x3FF) : excp_entry(mcause);

    /* Restore process context */
    asm("csrw mepc, %0" ::"r"(proc_set[curr_proc_idx].mepc));
    memcpy(SAVED_REGISTER_ADDR, proc_set[curr_proc_idx].saved_register,
           SAVED_REGISTER_SIZE);
}
```

mcause

Machine Cause Register			
CSR	`mcause`		
Bits	**Field Name**	**Attr.**	**Description**
[9:0]	Exception Code	WLRL	A code identifying the last exception.
[30:10]	Reserved	WLRL	
31	Interrupt	WARL	1 if the trap was caused by an interrupt; 0 otherwise.

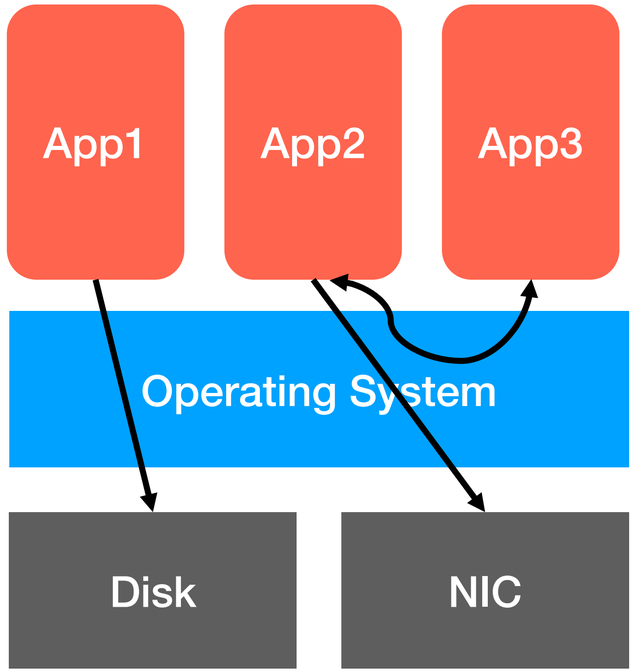
Table 22: `mcause` Register

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
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

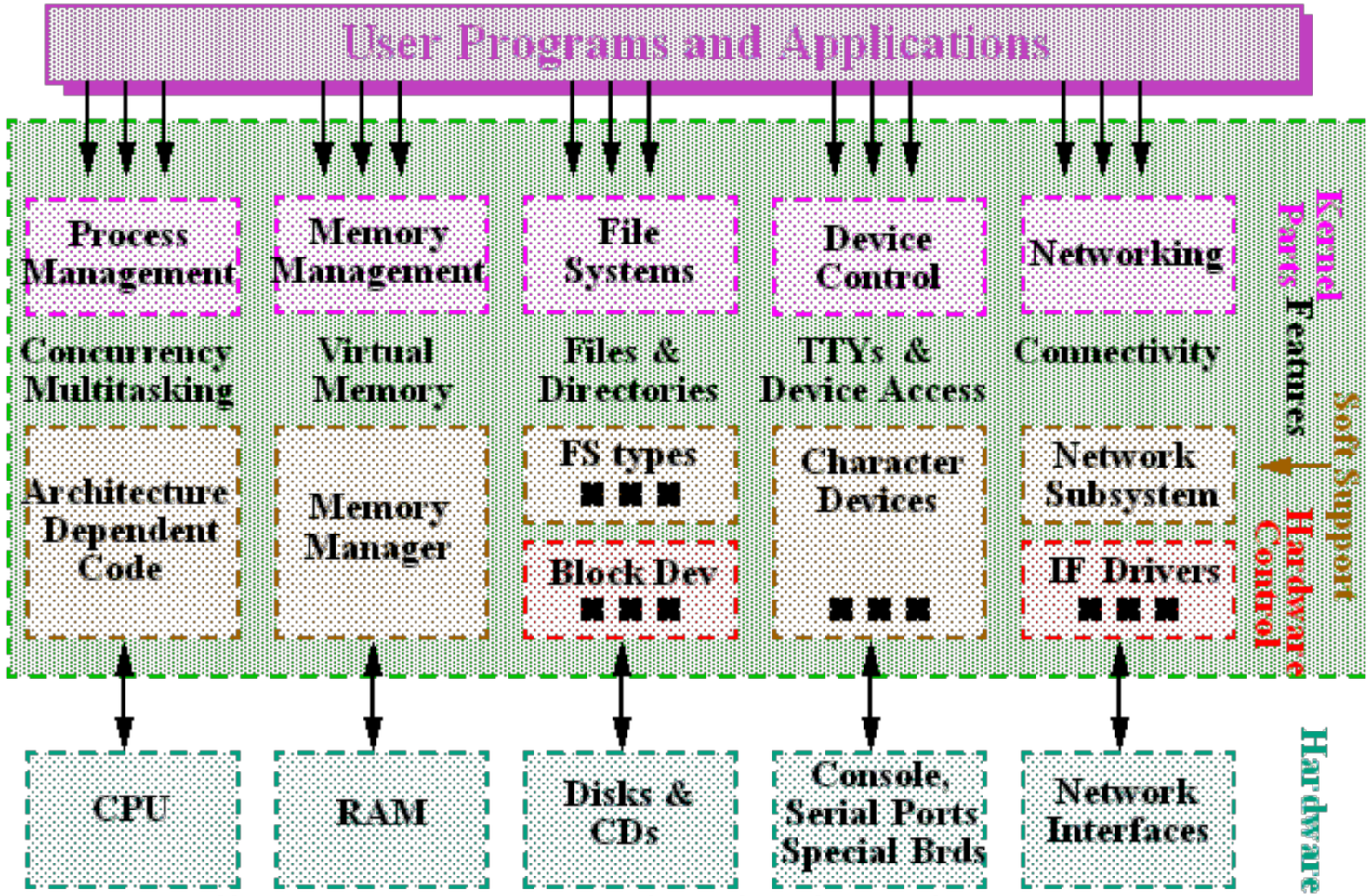
Table 23: `mcause` Exception Codes

Today: System Call

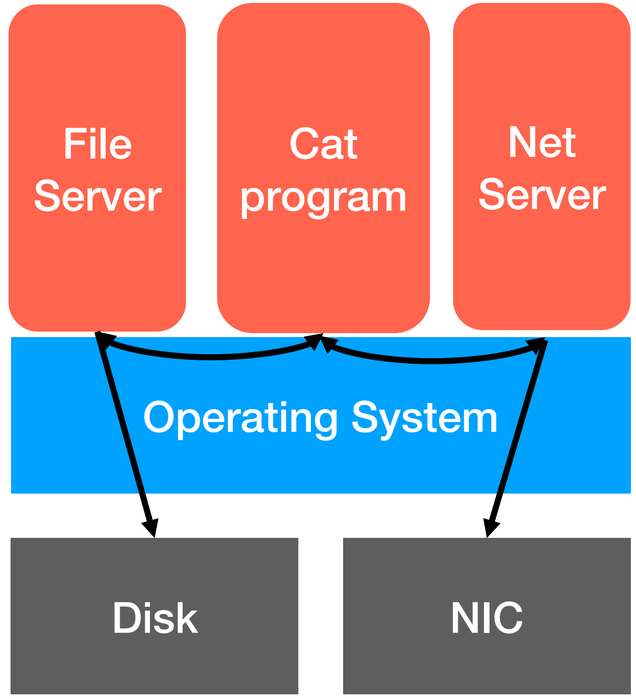
- A way for applications to request **services** from the OS.
  - E.g., read/write disks, access NICs, inter-process communication (IPC)
- How
  - Invoke OS kernel by **ECALL**



Monolithic kernel vs Microkernel



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

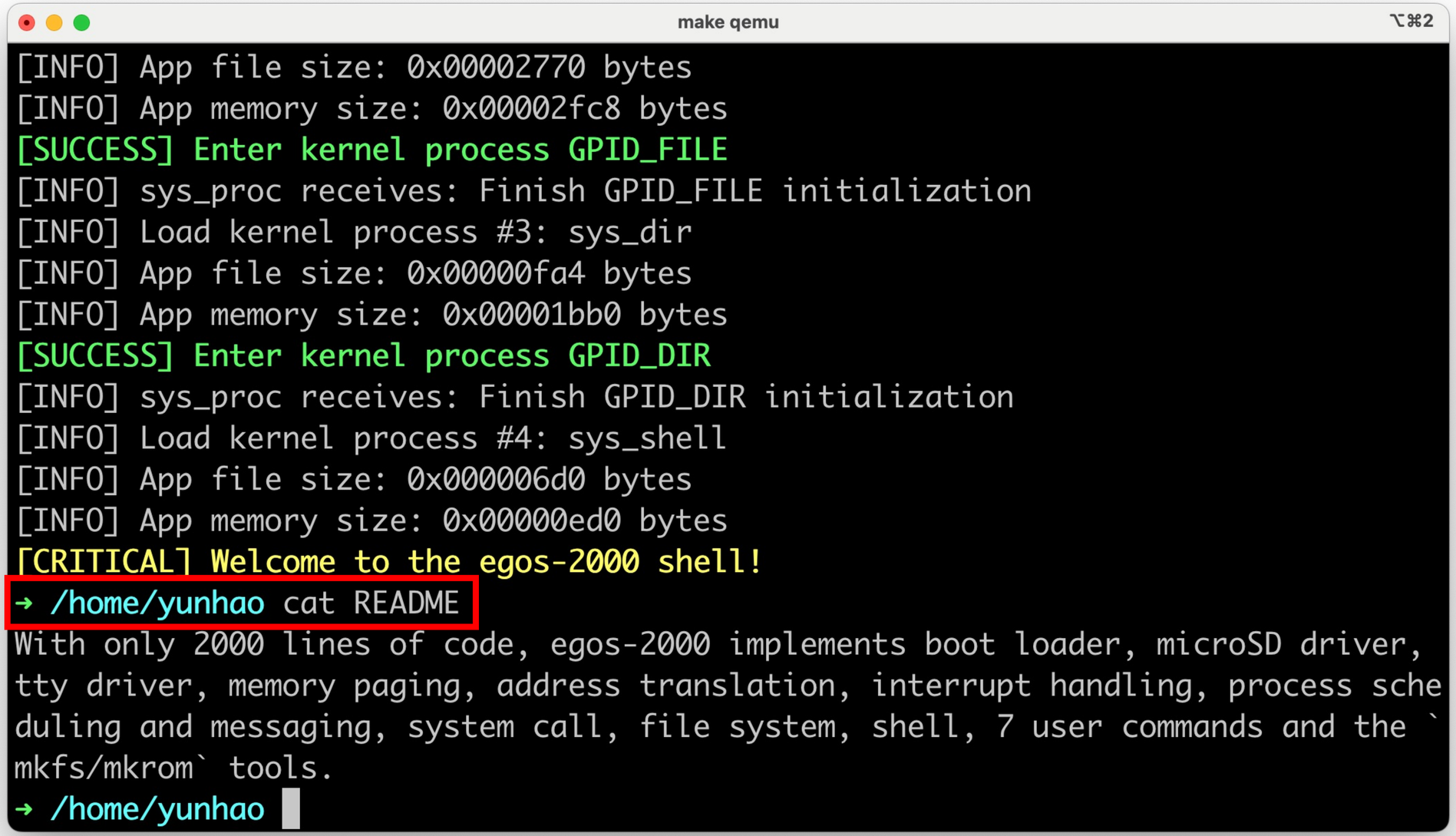


EGOS

Agenda

- A high-level picture of system calls
- A concrete implementation of system calls
- P4: Implement **sleep** system calls

apps/user/cat.c



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  }
```

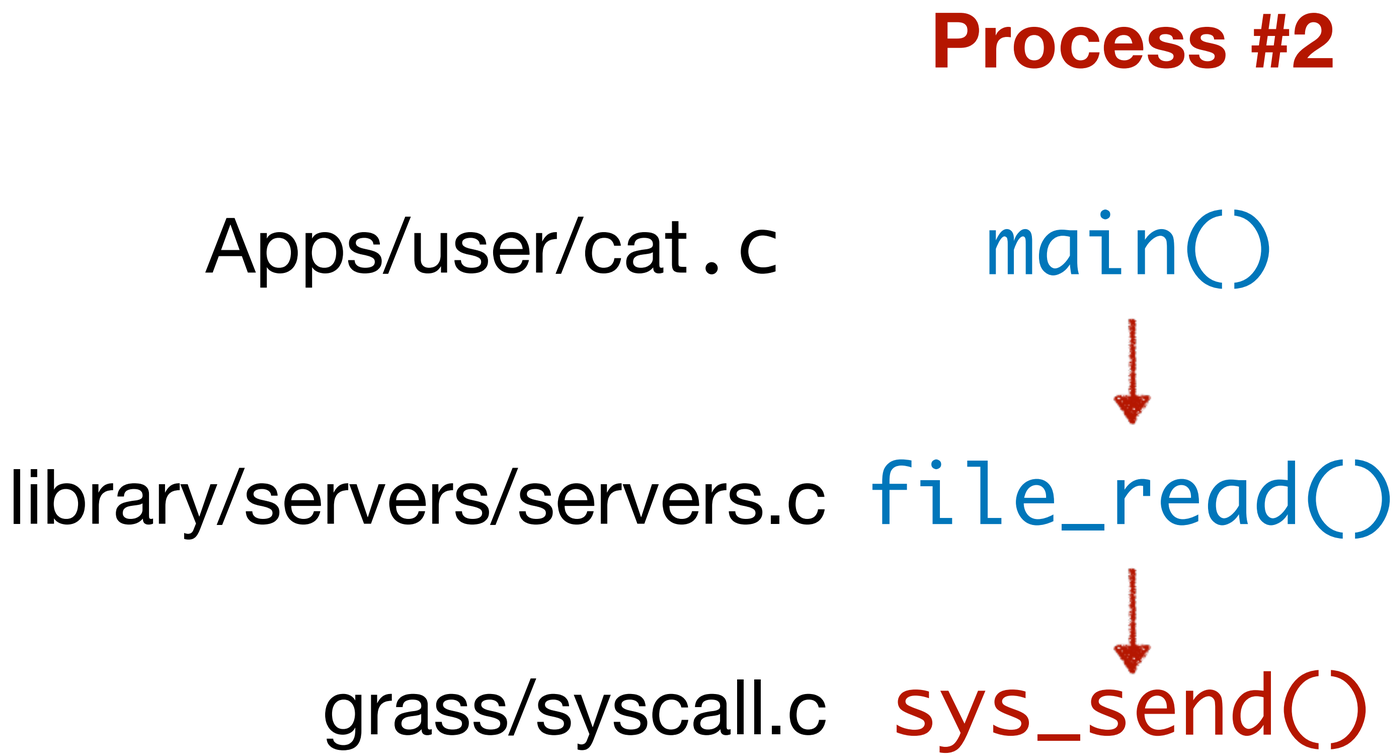
Step1. File server waits for requests

Process #1

apps/system/sys_file.c `main()`

grass/syscall.c `sys_recv()`

Step2. Cat sends a request for file content



Step3. Kernel handles the IPC

Process #1 (sys_file)

```
main()
  ↓
sys_recv()
```

Receive:
Read from **X** file

Process #2 (cat)

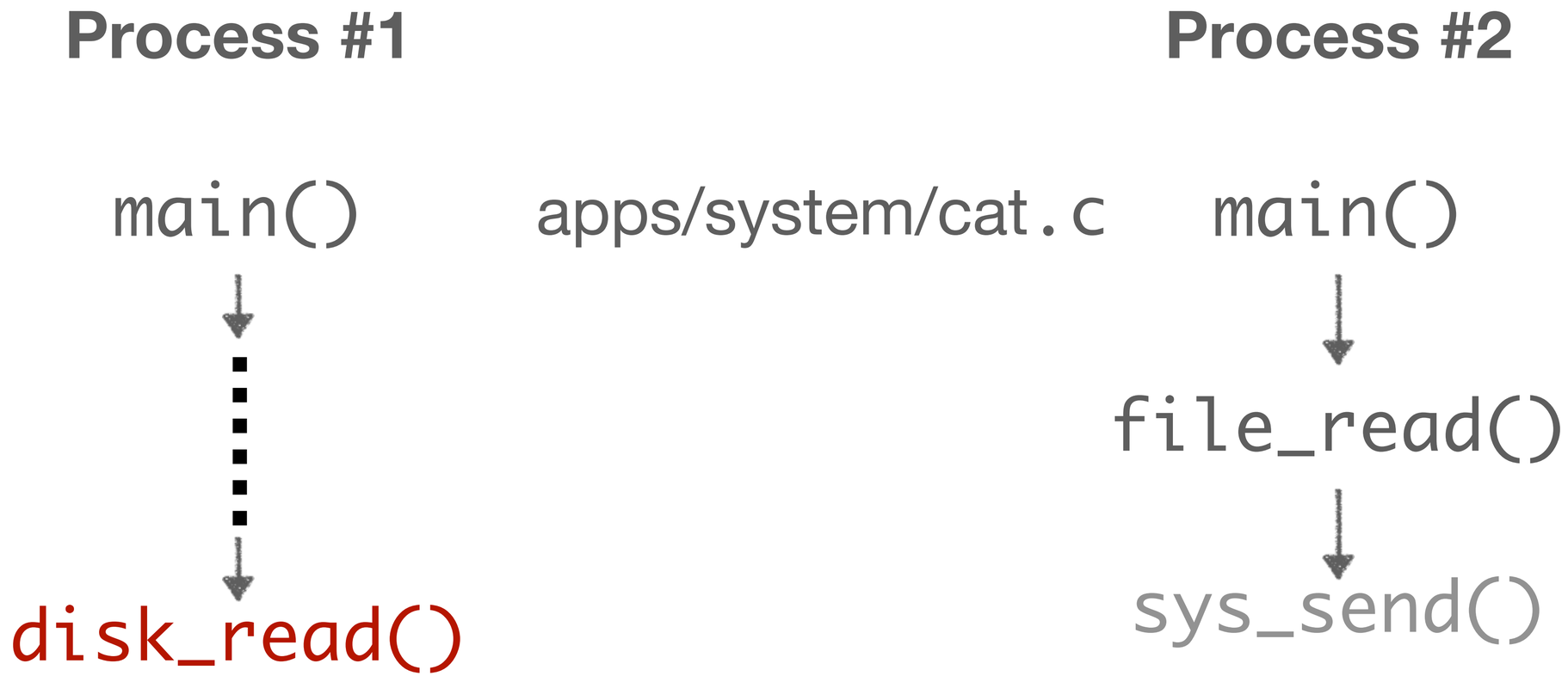
```
main()
  ↓
file_read()
  ↓
sys_send()
```

Send to FileServer:
Read from **X** file

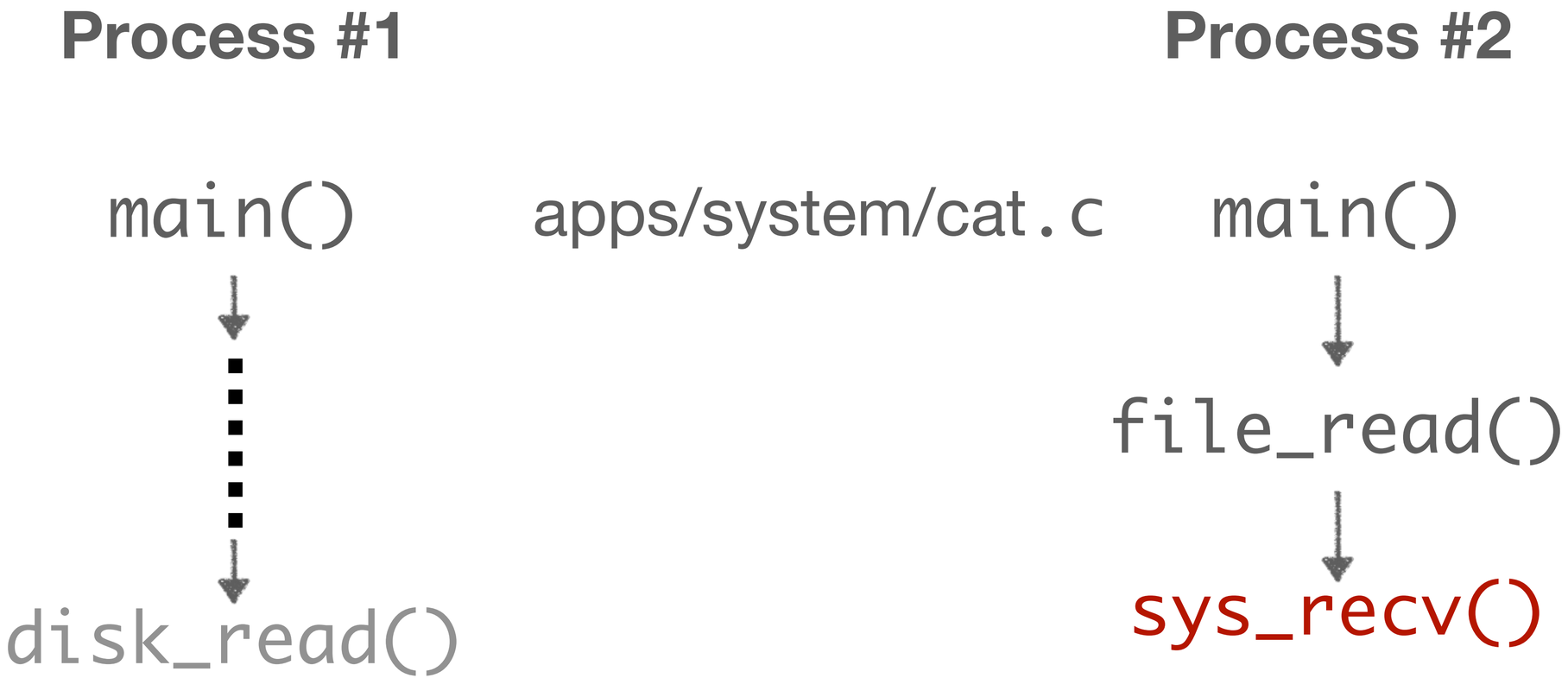
Inter-process Communication (IPC)

Grass kernel (grass/kernel.c)

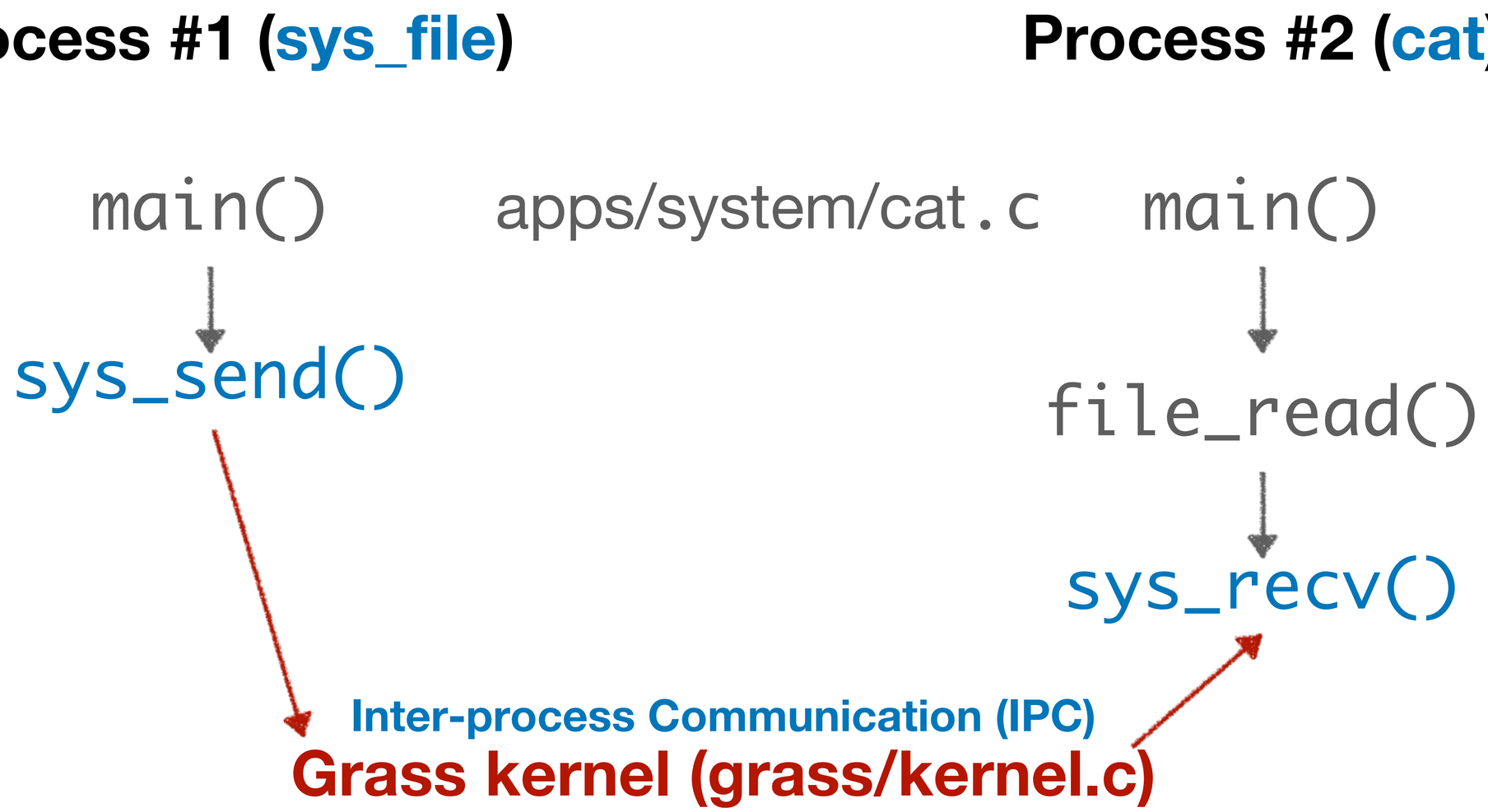
Step4a. File server reads file from disk



Step4b. Cat waits for the file content



Step5. File server returns the file content



- A high-level picture of system calls
- A concrete implementation of system calls
- P4: Implement **sleep** system calls

Data structures for system calls

```
enum syscall_type {
    SYS_UNUSED,
    SYS_RECV, /* 1 */
    SYS_SEND, /* 2 */
};
```

```
struct syscall {
    enum syscall_type type; /* SYS_SEND or SYS_RECV */
    int sender;             /* sender process ID    */
    int receiver;           /* receiver process ID  */
    char content[SYSCALL_MSG_LEN];
    enum { PENDING, DONE } status;
};
```

library/syscall/syscall.h

File server invoking sys_recv

```
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
    }
}
```

apps/system/sys_file.c

sys_recv

```
void sys_send(int receiver, char* msg, uint size) {
    sc->type     = SYS_SEND;
    sc->receiver = receiver;
    memcpy(sc->content, msg, size);
    asm("ecall");
}

void sys_recv(int from, int* sender, char* buf, uint size) {
    sc->sender = from;
    sc->type   = SYS_RECV;
    asm("ecall");
    memcpy(buf, sc->content, size);
    if (sender) *sender = sc->sender;
}
```

library/syscall/syscall.c

Kernel system call handler

```
void kernel_entry(uint mcause) {
    /* With the kernel lock, only one core can enter this point at any time */
    asm("csrr %0, mhartid" : "=r"(core_in_kernel));

    /* Save process context */
    asm("csrr %0, mepc" : "=r"(proc_set[curr_proc_idx].mepc));
    memcpy(proc_set[curr_proc_idx].saved_register, SAVED_REGISTER_ADDR,
           SAVED_REGISTER_SIZE);

    (mcause & (1 << 31)) ? intr_entry(mcause & 0x3FF) : excp_entry(mcause);

    /* Restore process context */
    asm("csrw mepc, %0" ::"r"(proc_set[curr_proc_i
    memcpy(SAVED_REGISTER_ADDR, proc_set[curr_proc_
           SAVED_REGISTER_SIZE);
}
```

```
static void excp_entry(uint id) {
    if (id >= EXCP_ID_ECALL_U && id <= EXCP_ID_ECALL_M) {
        /* Copy the system call arguments from user space to the kernel */
        memcpy(&proc_set[curr_proc_idx].syscall, (void*)syscall_paddr,
               sizeof(struct syscall));

        proc_set[curr_proc_idx].mepc += 4;
        proc_set[curr_proc_idx].syscall.status = PENDING;
        proc_try_syscall(&proc_set[curr_proc_idx]);
        proc_yield();
        return;
    }

    /* Student's code goes here (system call and memory exception).
     * Kill the process if curr_pid is a user application */

    /* Student's code ends here. */
    FATAL("excp_entry: kernel got exception %d", id);
}
```

Handle system call

```
static void proc_yield() {
    /* Student's code goes here (Preemptive Scheduler). */
    if (!CORE_IDLE && curr_status == PROC_RUNNING) proc_set_runnable(curr_pid);
    int next_idx = MAX_NPROCESS;
    for (uint i = 1; i <= MAX_NPROCESS; i++) {
        struct process* p = &proc_set[(curr_proc_idx + i) % MAX_NPROCESS];
        if (p->status == PROC_PENDING_SYSCALL) proc_try_syscall(p);

        if (p->status == PROC_READY || p->status == PROC_RUNNABLE) {
            next_idx = (curr_proc_idx + i) % MAX_NPROCESS;
            break;
        }
    }

    ...

    curr_proc_idx = next_idx;
    proc_set_running(curr_pid);
}
```

App invoking sys_send

```
void sys_send(int receiver, char* msg, uint size) {
    sc->type     = SYS_SEND;
    sc->receiver = receiver;
    memcpy(sc->content, msg, size);
    asm("ecall");
}

void sys_recv(int from, int* sender, char* buf, uint size) {
    sc->sender = from;
    sc->type   = SYS_RECV;
    asm("ecall");
    memcpy(buf, sc->content, size);
    if (sender) *sender = sc->sender;
}
```

library/syscall/syscall.c

Kernel system call handler

```
void kernel_entry(uint mcause) {
    /* With the kernel lock, only one core can enter this point at any time */
    asm("csrr %0, mhartid" : "=r"(core_in_kernel));

    /* Save process context */
    asm("csrr %0, mepc" : "=r"(proc_set[curr_proc_idx].mepc));
    memcpy(proc_set[curr_proc_idx].saved_register, SAVED_REGISTER_ADDR,
           SAVED_REGISTER_SIZE);

    (mcause & (1 << 31)) ? intr_entry(mcause & 0x3FF) : excp_entry(mcause);

    /* Restore process context */
    asm("csrw mepc, %0" ::"r"(proc_set[curr_proc_i
    memcpy(SAVED_REGISTER_ADDR, proc_set[curr_proc_
           SAVED_REGISTER_SIZE);
}
```

```
static void excp_entry(uint id) {
    if (id >= EXCP_ID_ECALL_U && id <= EXCP_ID_ECALL_M) {
        /* Copy the system call arguments from user space to the kernel */
        memcpy(&proc_set[curr_proc_idx].syscall, (void*)syscall_paddr,
               sizeof(struct syscall));

        proc_set[curr_proc_idx].mepc += 4;
        proc_set[curr_proc_idx].syscall.status = PENDING;
        proc_try_syscall(&proc_set[curr_proc_idx]);
        proc_yield();
        return;
    }

    /* Student's code goes here (system call and memory exception).
     * Kill the process if curr_pid is a user application */

    /* Student's code ends here. */
    FATAL("excp_entry: kernel got exception %d", id);
}
```

Handle system call

```
static void proc_yield() {
    /* Student's code goes here (Preemptive Scheduler). */
    if (!CORE_IDLE && curr_status == PROC_RUNNING) proc_set_runnable(curr_pid);
    int next_idx = MAX_NPROCESS;
    for (uint i = 1; i <= MAX_NPROCESS; i++) {
        struct process* p = &proc_set[(curr_proc_idx + i) % MAX_NPROCESS];
        if (p->status == PROC_PENDING_SYSCALL) proc_try_syscall(p);

        if (p->status == PROC_READY || p->status == PROC_RUNNABLE) {
            next_idx = (curr_proc_idx + i) % MAX_NPROCESS;
            break;
        }
    }

    ...

    curr_proc_idx = next_idx;
    proc_set_running(curr_pid);
}
```

Handle system call (2)

```
static int proc_try_send(struct process* sender) {
    for (uint i = 0; i < MAX_NPROCESS; i++) {
        struct process* dst = &proc_set[i];
        if (dst->pid == sender->syscall.receiver &&
            dst->status != PROC_UNUSED) {

            dst->syscall.status = DONE;
            dst->syscall.sender = sender->pid;
            /* Copy the system call arguments within the kernel PCB */
            memcpy(dst->syscall.content, sender->syscall.content,
                   SYSCALL_MSG_LEN);
            return 0;
        }
    }
    FATAL("proc_try_send: process %d sending to unknown process %d",
          sender->pid, sender->syscall.receiver);
}

static int proc_try_recv(struct process* receiver) {
    if (receiver->syscall.status == PENDING) return -1;
    memcpy((void*)syscall_paddr, &receiver->syscall, sizeof(struct syscall));
    return 0;
}
```

File Server is unblocked

```
void sys_send(int receiver, char* msg, uint size) {
    sc->type     = SYS_SEND;
    sc->receiver = receiver;
    memcpy(sc->content, msg, size);
    asm("ecall");
}

void sys_recv(int from, int* sender, char* buf, uint size) {
    sc->sender = from;
    sc->type   = SYS_RECV;
    asm("ecall");
    memcpy(buf, sc->content, size);
    if (sender) *sender = sc->sender;
}
```

library/syscall/syscall.c

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 P5 & P6)
    case FILE_WRITE:
        ... // write to a file on disk (project P5 & P6)
    }
}
```

- A high-level picture of system calls
- A concrete implementation of system calls
- P4: Implement **sleep** system calls

P4

- Part1: Implement sleep system service
- Part2: memory protection

sleep system service

- sleep user API
- sleep system call

sleep system call

- sleep user API
- sleep system call
  - sys_send(PROC_PROCESS, SLEEP, NTICKS)
  - grass->proc_sleep

```
while (1) {
    ...
    grass->sys_recv(GPID_ALL, &sender, buf, SYSCALL_MSG_LEN);

    switch (req->type) {
    case PROC_SPAWN:
        reply->type = app_spawn(req);
        ...
        grass->sys_send(GPID_SHELL, (void*)reply, sizeof(*reply));
        break;
    case PROC_EXIT:
        ...
        break;
    ......
}
```

apps/system/sys_proc.c

sleep system call

- sleep user API
- sleep system call
  - sys_send(PROC_PROCESS, SLEEP, NTICKS)
  - grass->proc_sleep
- proc_sleep

sleep system call

- sleep user API
- sleep system call
  - sys_send(PROC_PROCESS, SLEEP, NTICKS)
  - grass->proc_sleep
- proc_sleep
  - Add sleep_time to struct process
  - When the scheduler kicks in, check if the sleep time has elapsed and change the state to runnable.
    - Using mtime_get()

Demo

git patches

- git format-patch BASE_COMMIT_NUMBER
- git am -3 XXX.patch

Today

- System call
- P4 part1: implement sleep system call
- Next week: P4 part2 – memory protection
- Mid-term evaluation!

- P3 (5411 required, 4411 optional) release today, due in a month
- P4 release today, due in three weeks