Memory Protections

Yu-Ju Huang
Slides adapted from Yunhao Zhang, Kevin A. Negy

P4

- Part1: Implement sleep system service
- Part2: memory protection
  - Setup PMP regions
  - Set user application to User mode
  - Kill user applications for exceptions

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.

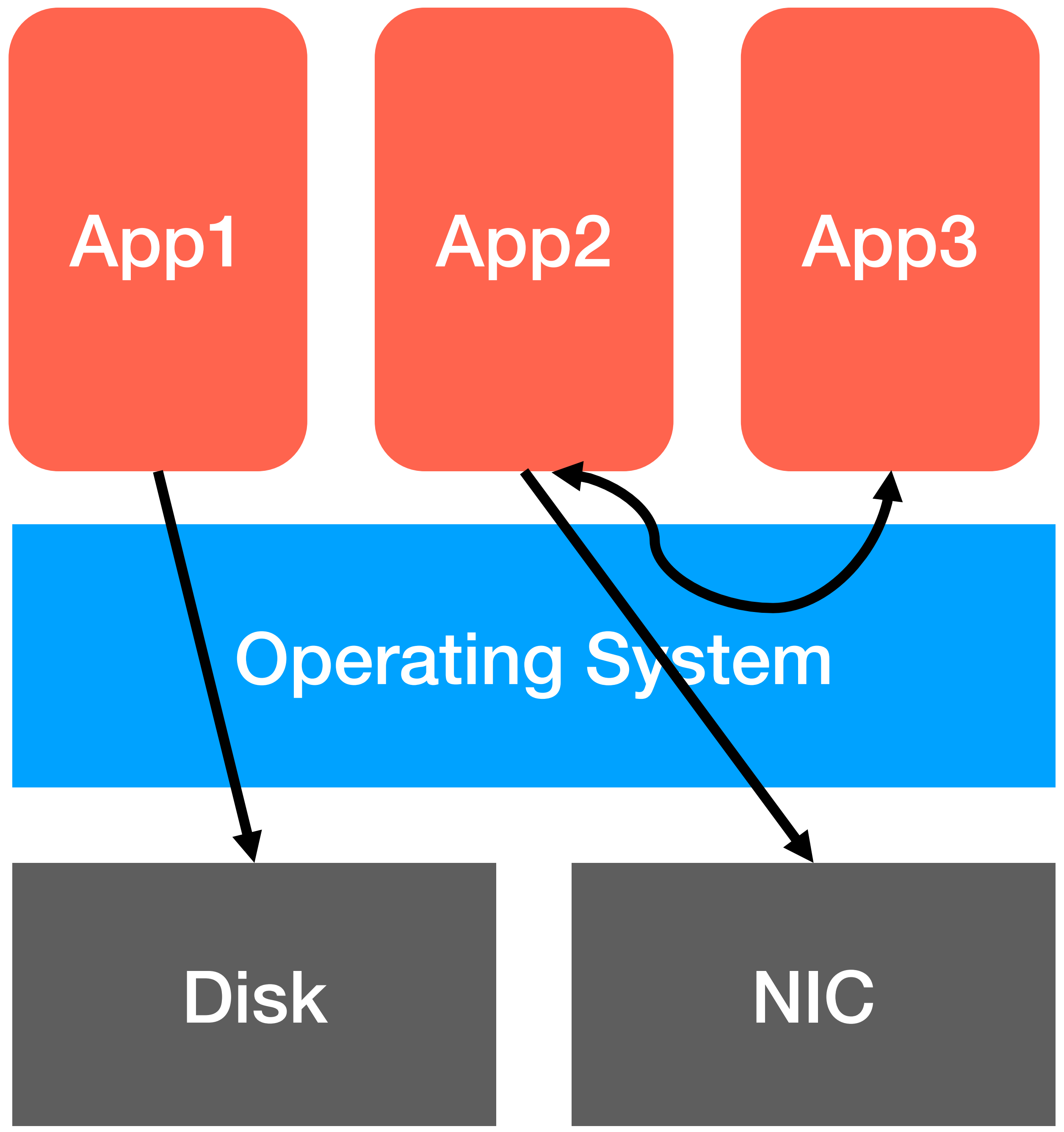
Table 22: `mcause` Register

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

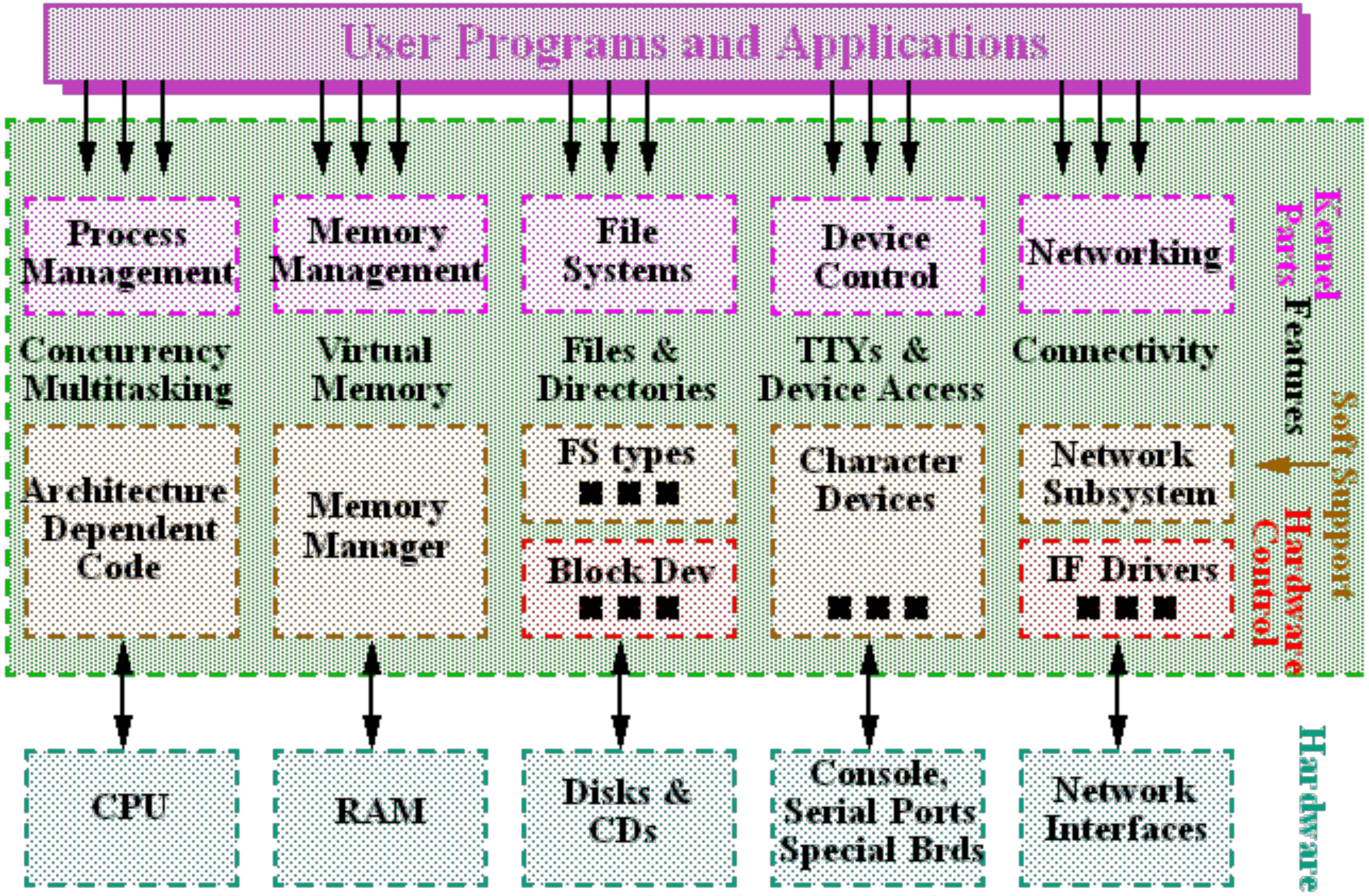
Table 23: `mcause` Exception Codes

Review: System Call

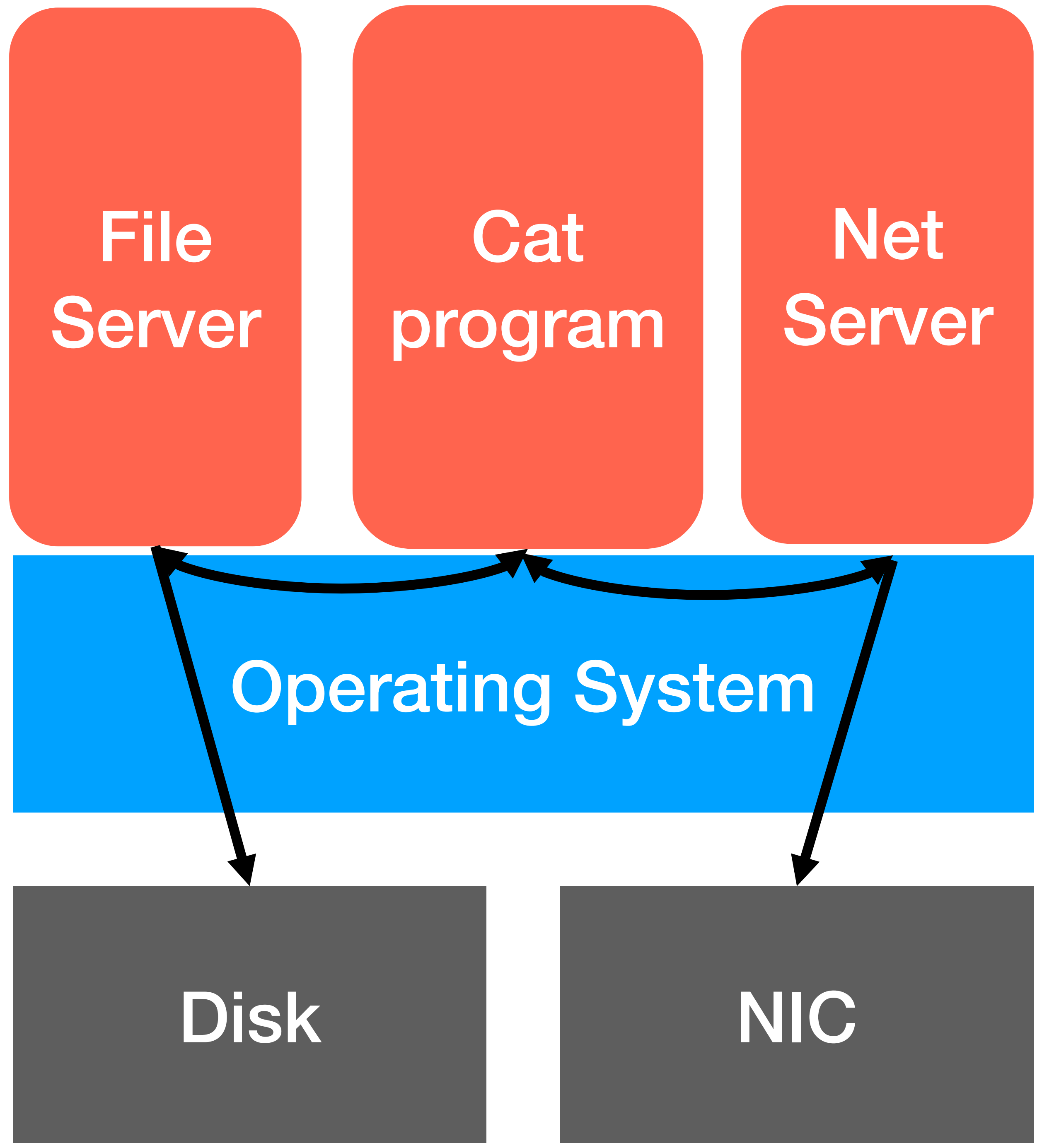
- A way for applications to request **services** from the OS.
  - E.g., read/write disks, access NICs, inter-process communication (IPC)
- How
  - Invoke OS kernel by **ECALL**



Monolithic kernel vs Microkernel



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



EGOS

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
```

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

Process #2

Apps/user/cat.c `main()`

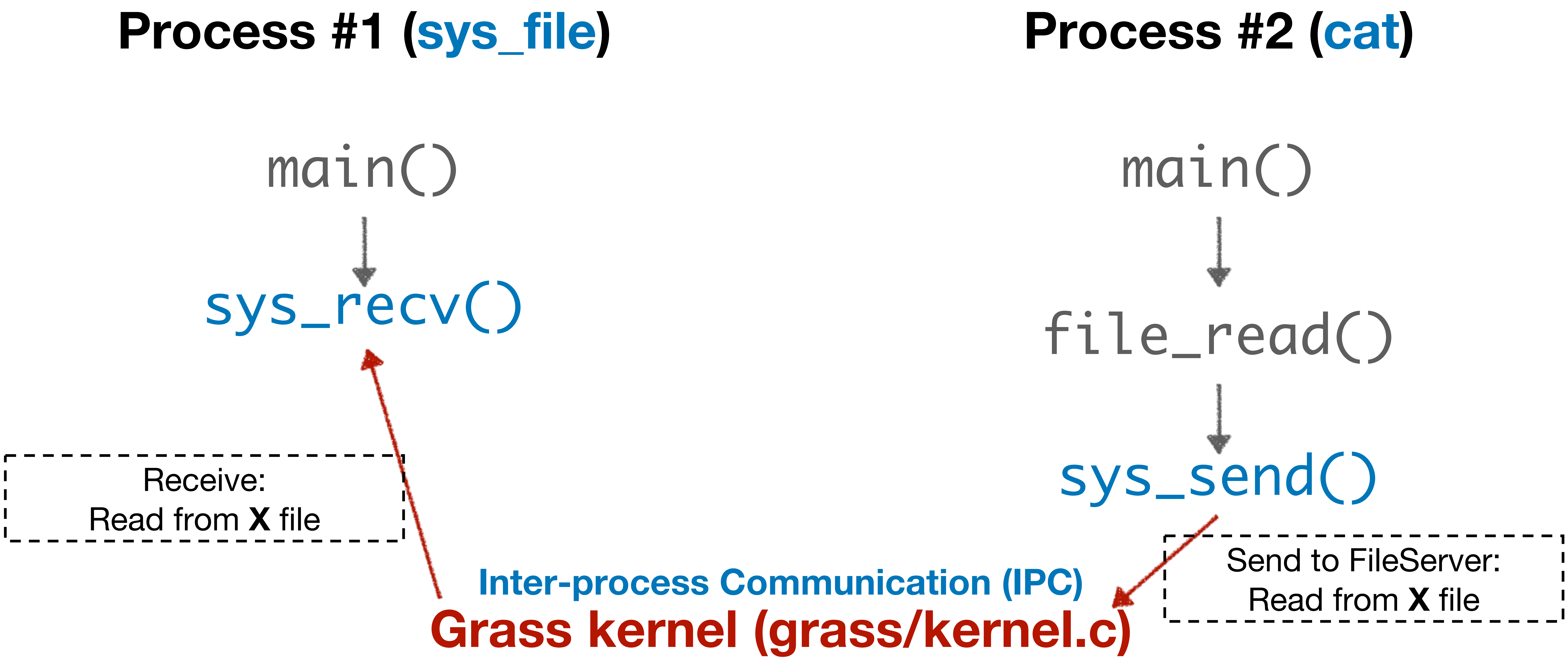
↓

library/servers/servers.c `file_read()`

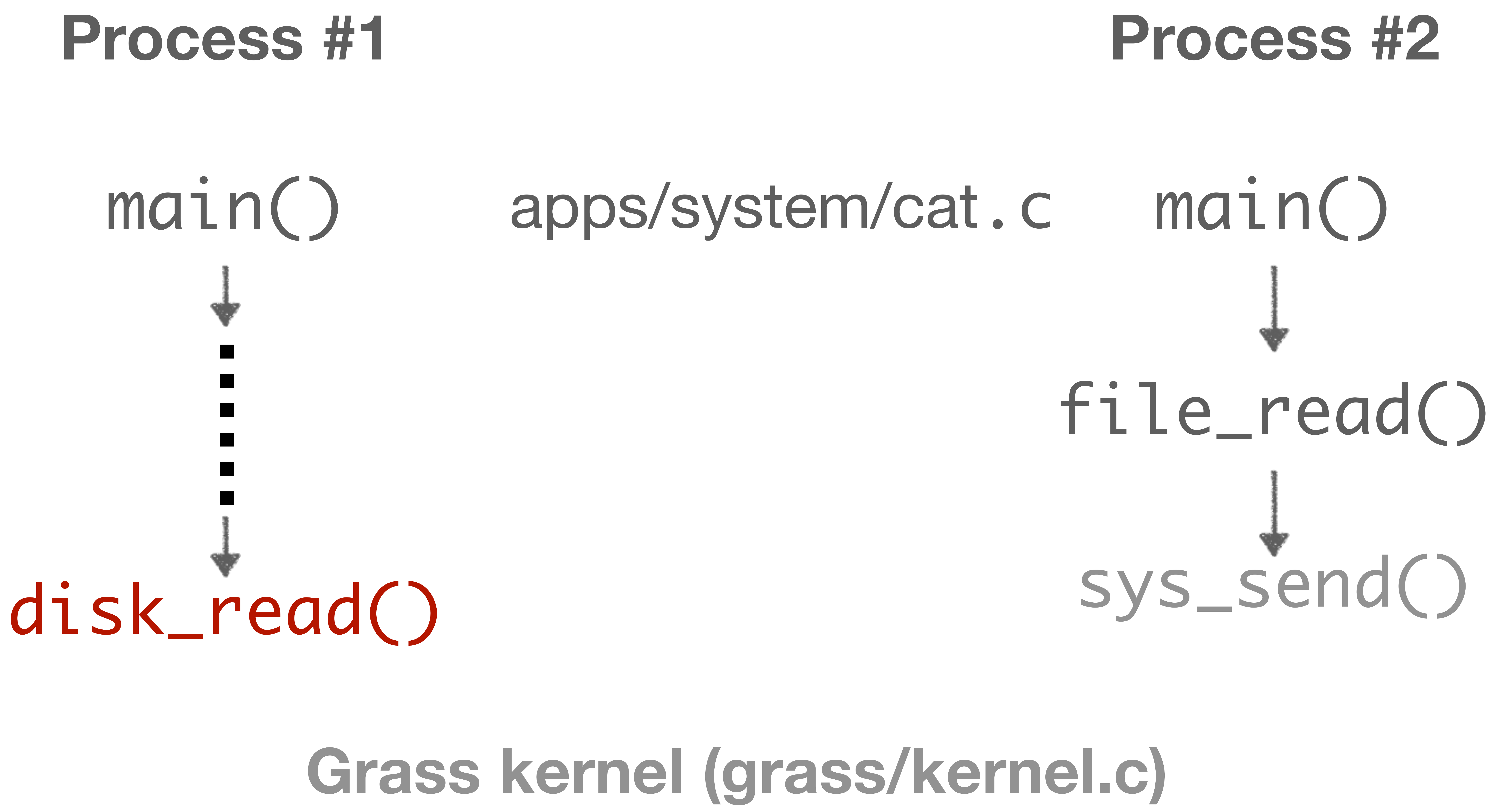
↓

grass/syscall.c `sys_send()`

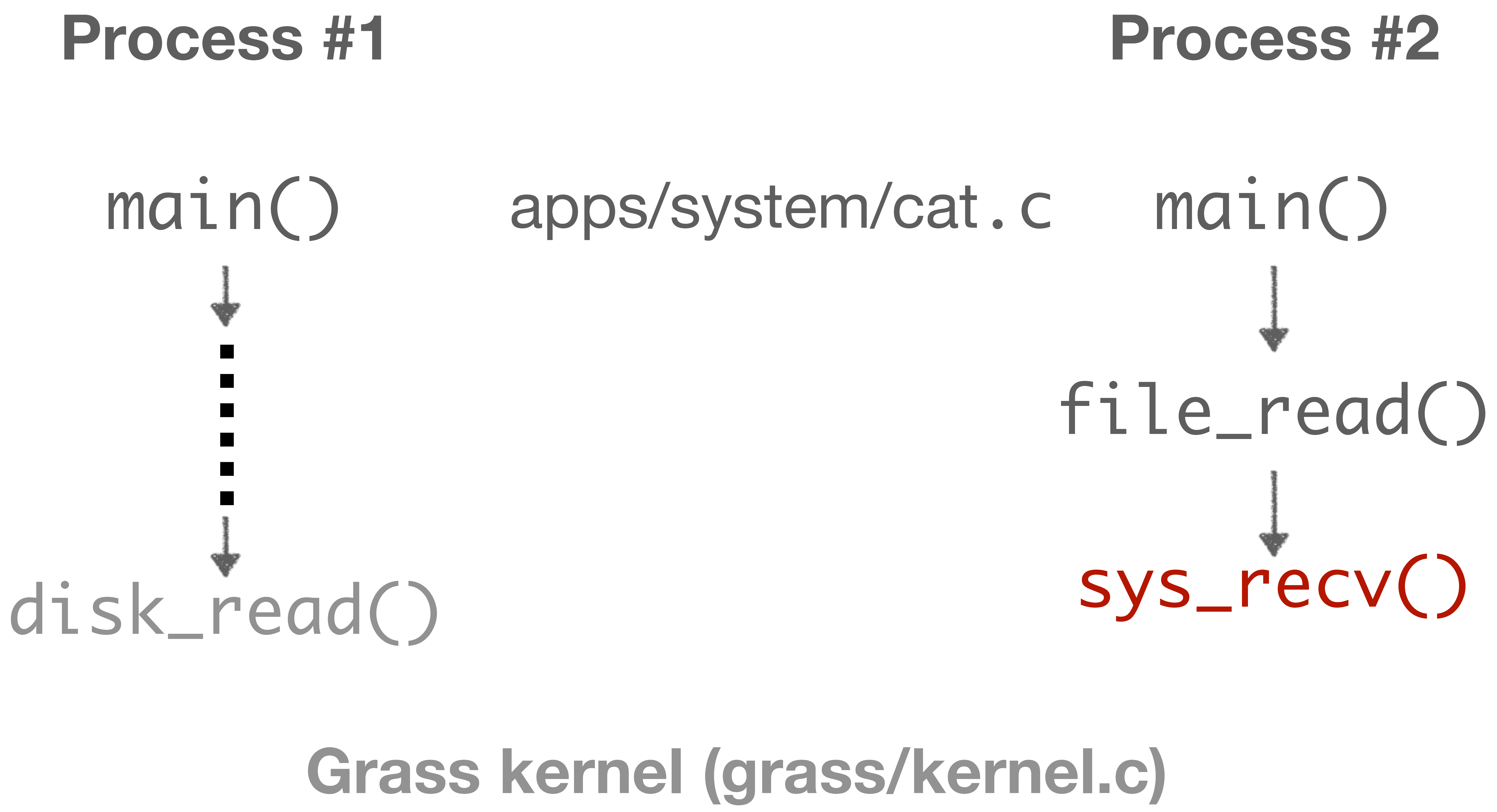
Step3. Kernel handles the IPC



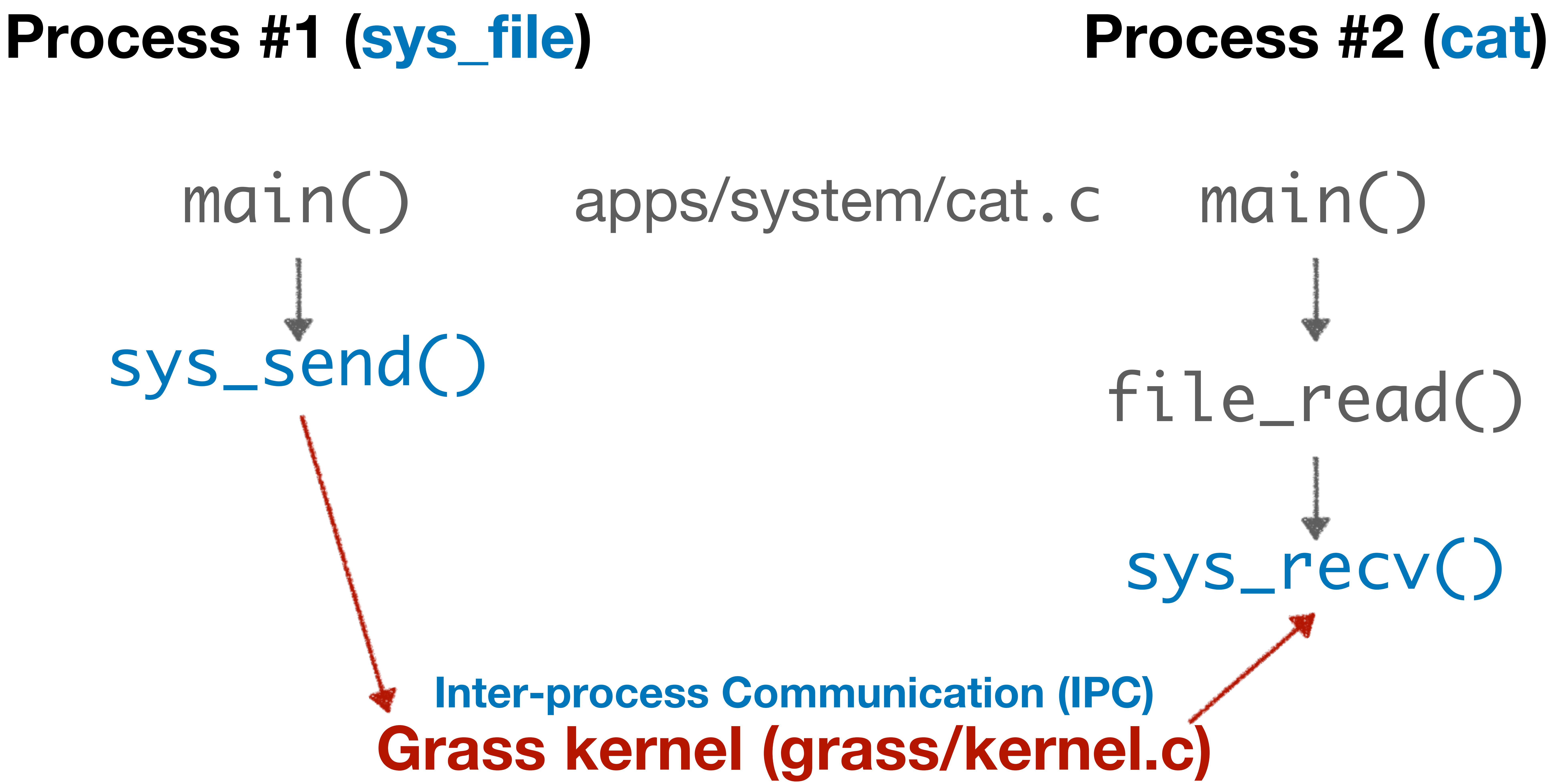
Step4a. File server reads file from disk



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

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

```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_i
    memcpy(SAVED_REGISTER_ADDR, proc_set[curr_proc_
           SAVED_REGISTER_SIZE);
}
```

```c
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 (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

P4

- Part1: Implement sleep system service
- Part2: memory protection
  - Setup PMP regions
  - Set user application to User mode
  - Kill user applications for exceptions

sleep system call

- sleep user API
  - sys_send(PROC_PROCESS, SLEEP, NTICKS)
- In PROC_PROCESS
  - grass->proc_sleep
- In kernel: 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()

Memory protection

- Machine mode can access all memory regions.
- OS specifies which regions can be accessed by user mode.
- In P4, you will specify 1 PMP regions for user mode
  - PMP stands for Physical Memory Protection
  - Read section 3.6 of the RISC-V reference manual

PMP entries

- Comprised of (at least) two parts:
  - a PMP address (one of pmpaddr0 – pmpaddr63)
  - a PMP configuration (one of pmpcfg0 - pmpcfg15)

PMP entries

- Comprised of (at least) two parts:
  - a PMP address (one of pmpaddr0 - pmpaddr15)
  - a PMP configuration (one of pmpcfg0 - pmpcfg15)
- Smallest PMP region you can protect is 4 bytes
  - RISC-V32 has 34 bit physical address, 32 bit registers (bottom two bits not stored in PMP)

PMP entries

- Comprised of (at least) two parts:
    - a PMP address (one of pmpaddr0 - pmpaddr15)
    - a PMP configuration (one of pmpcfg0 - pmpcfg15)
- Smallest PMP region you can protect is 4 bytes
    - RISC-V32 has 34 bit physical address, 32 bit registers (bottom two bits not stored in PMP)
- Different types of PMP configurations
    - e.g. TOR, NA4, NAPOT

How to read RISC-V **PMP figures?**

RISC-V32 physical memory address is 34 bits;
This register holds the first 32 bits [33 : 2].

Bit index (0 .. 31)

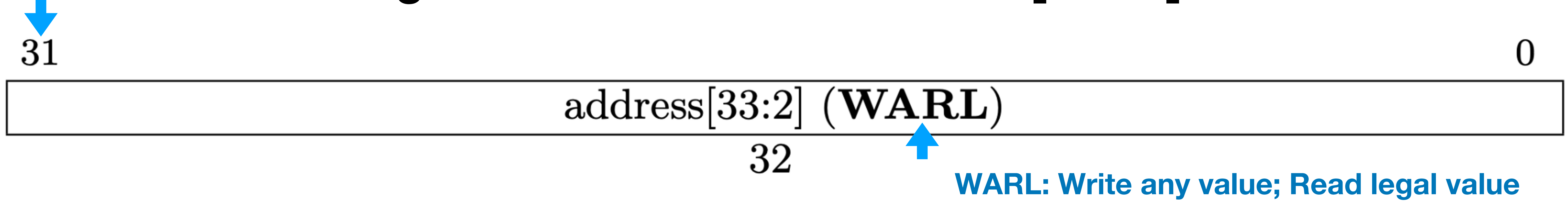


Figure 3.25: PMP address register format, RV32.

TOR PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space with WRX permission

TOR PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space with WRX permission
- A TOR (top of range) entry in `pmpaddr0` has special meaning
  - Protect range `0x0-pmpaddr0`

TOR PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space with WRX permission
- A TOR (top of range) entry in `pmpaddr0` has special meaning
  - Protect range `0x0-pmpaddr0`
- Convert physical address to PMP address
  - 0x1_0000_0000 (4GB) >> 2 == 0x4000_0000

7	6 5	4 3	2	1	0
L	0	A	X	W	R
1	2	2	1	1	1

Figure 34. PMP configuration register format.

- Goal: Set up a PMP region for the lowest ... with WRX permission

8. Encoding of A field in PMP configuration registers.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

- A TOR (top of range) entry in `pmpaddr0` ha
  - Protect range `0x0-pmpaddr0`
- Convert physical address to PMP address
  - 0x1_0000_0000 (4GB) >> 2 == 0x4000_0000
- Configuration: TOR, readable, writable, executable.
  - cfg = 0b01111 = 0x0f

```
asm("csrw pmpaddr0, %0" : : "r" (0x40000000));
asm("csrw pmpcfg0, %0" : : "r" (0xF));
```

NAPOT PMP example

- Goal: Setup PMP NAPOT region 0x20400000 - 0x20800000 with r/w/- permission

NAPOT PMP example

- Goal: Setup PMP NAPOT region 0x20400000 - 0x20800000 with r/w/- permission
- Convert base physical address to PMP address
  - 0x20400000 >> 2 == 0x08100000

NAPOT PMP example

- Goal: Setup PMP NAPOT region 0x20400000 with r/w/- permission
- Convert base physical address to PMP addres
  - 0x20400000 >> 2 == 0x08100000
- Calculate the alignment
  - 0x20800000 - 0x20400000 = 0x00400000 = 2^22
  - 2^22 -> 0111_1111_1111_1111_1111

pmpaddr	pmpcfg.A	Match type and size
yyyy…yyyy	NA4	4-byte NAPOT range
yyyy…yyy0	NAPOT	8-byte NAPOT range
yyyy…yy01	NAPOT	16-byte NAPOT range
yyyy…y011	NAPOT	32-byte NAPOT range
…	…	…
yy01…1111	NAPOT	2^{XLEN}-byte NAPOT range
y011…1111	NAPOT	2^{XLEN+1}-byte NAPOT range
0111…1111	NAPOT	2^{XLEN+2}-byte NAPOT range
1111…1111	NAPOT	2^{XLEN+3}-byte NAPOT range

NAPOT PMP example

- Goal: Setup PMP NAPOT region 0x20400000 - 0x
  r/w/- permission
- Convert base physical address to PMP address
  - 0x20400000 >> 2 == 0x08100000
- Calculate the alignment
  - 0x20800000 - 0x20400000 = 0x00400000 = 2^22
  - 2^22 -> 0111_1111_1111_1111_1111 = 0x7ffff
- Calculate pmpaddr
  - 0x08100000 | 0x7ffff = 0x0817ffff

pmpaddr	pmpcfg.A	Match type and size
yyyy…yyyy	NA4	4-byte NAPOT range
yyyy…yyy0	NAPOT	8-byte NAPOT range
yyyy…yy01	NAPOT	16-byte NAPOT range
yyyy…y011	NAPOT	32-byte NAPOT range
…	…	…
yy01…1111	NAPOT	2^{XLEN}-byte NAPOT range
y011…1111	NAPOT	2^{XLEN+1}-byte NAPOT range
0111…1111	NAPOT	2^{XLEN+2}-byte NAPOT range
1111…1111	NAPOT	2^{XLEN+3}-byte NAPOT range

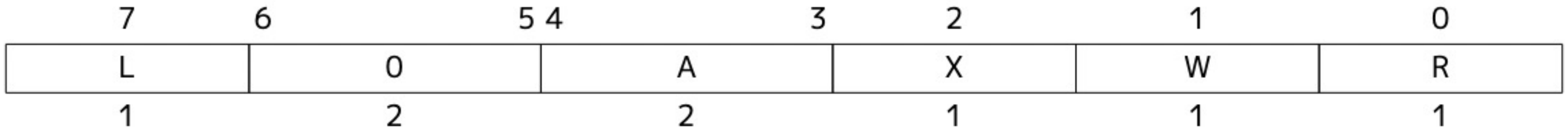


Figure 34. PMP configuration register format.

le

8. Encoding of A field in PMP configuration registers.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

permission

- Convert base physical address to PMP address
  - 0x20400000 >> 2 == 0x08100000
- Calculate the alignment
  - 0x20800000 - 0x20400000 = 0x00400000 = 2^22
  - 2^22 -> 0111_1111_1111_1111_1111 = 0x7ffff
- Calculate pmpaddr
  - 0x08100000 | 0x7ffff = 0x0817ffff
- Configuration: NAPOT, readable, writable.
  - cfg = 0b11011 = 0x1b

```
asm("csrw pmpaddr1, %0" : : "r" (0x0817ffff));
asm("csrw pmpcfg0, %0" : : "r" (0x1b << 8));
```

Switching privilege level

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mstatus.MPIE`.
- The pc is set to the value of `mepc`.

31	30 … 23	22	21	20	19	18	17
SD	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV
1	8	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

Set user application to User mode

- In proc_yield, before switching backing to user application, set the mstatus.MPP

```
void proc_yield() {
    ….
    FIND_NEXT
    ….
    if (curr_pid >= GPID_USER_START) {
        SET mstatus.MPP
    }
    ….

}
```

Kill user applications for exceptions

```
if (curr_pid >= GPID_USER_START) {

    // kill the process

}
```

```
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);
}
```

Demo

Submission: git patches

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

Today

- Memory protection
- P4
- Mid-term evaluation!