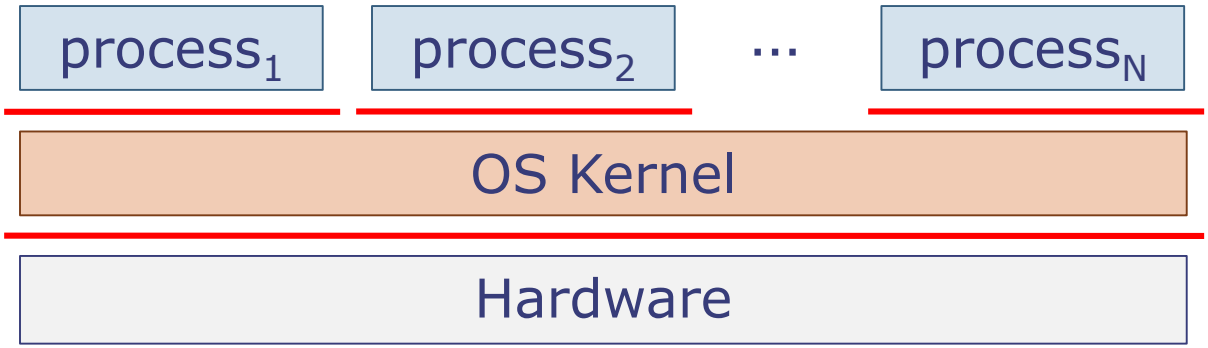
Operating Systems

Recitation

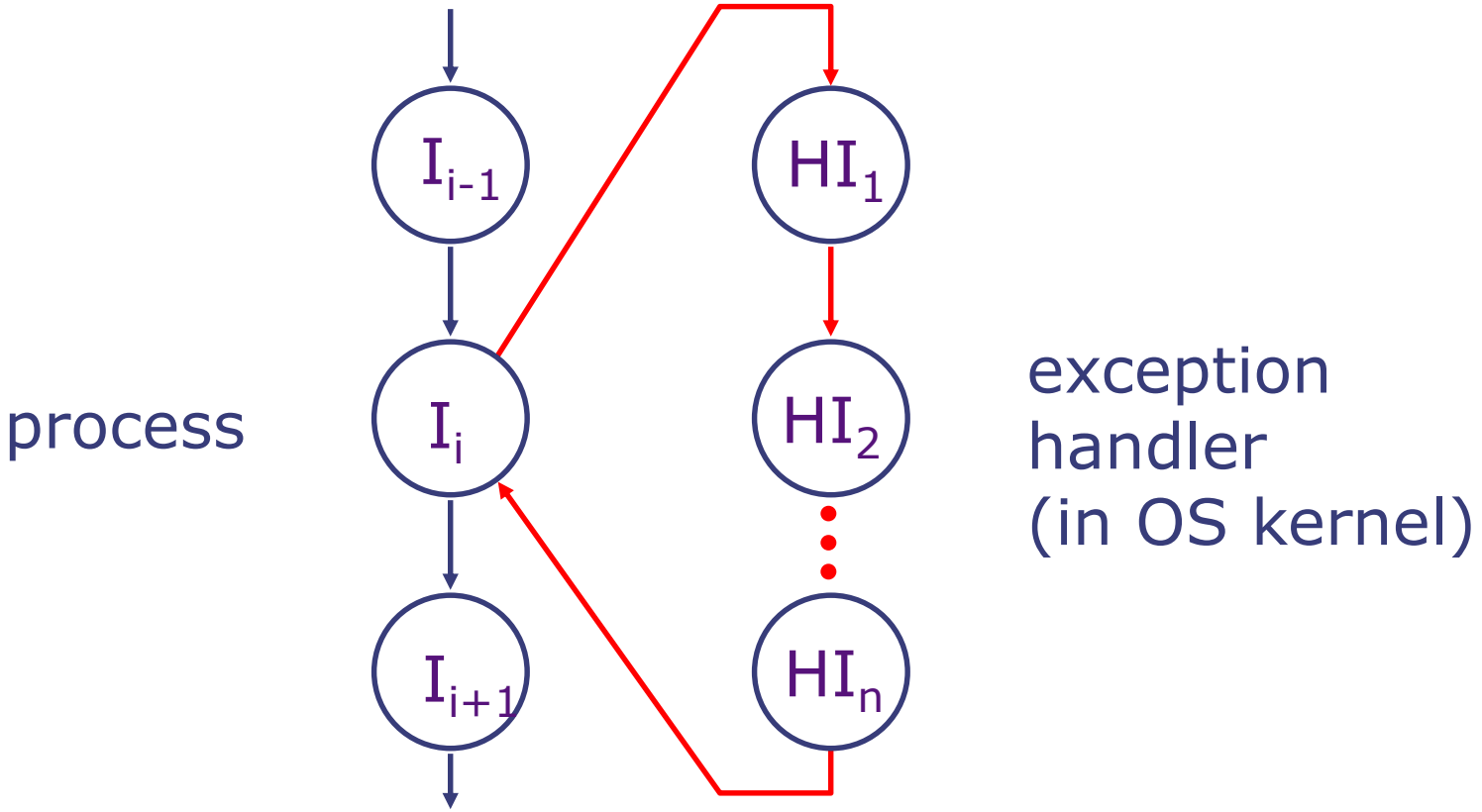
Review: Goals of Operating Systems



- Protection and privacy: Processes cannot access each other's data
- Abstraction: OS hides details of underlying hardware
  - e.g., processes open and access files instead of issuing raw commands to the disk
- Resource management: OS controls how processes share hardware (CPU, memory, disk, etc.)

Exceptions

- Exception: Event that needs to be processed by the OS kernel. The event is usually unexpected or rare.

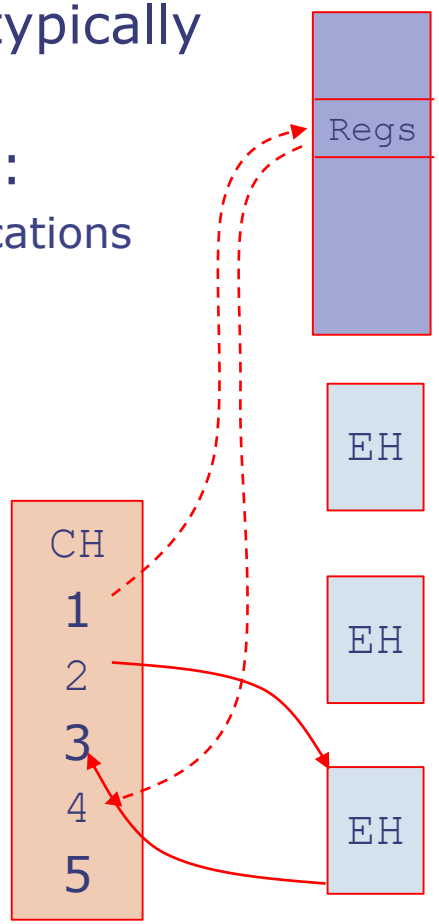


RISC-V Exception Handling

- RISC-V provides several privileged registers, called control and status registers (CSRs), e.g.,
  - mepc: exception PC
  - mcause: cause of the exception (interrupt, illegal instr, etc.)
  - mtvec: address of the exception handler
  - mstatus: status bits (privilege mode, interrupts enabled, etc.)
- RISC-V also provides privileged instructions, e.g.,
  - csrr and csrw to read/write CSRs
  - mret to return from the exception handler to the process
  - Trying to execute these instructions from user mode causes an exception → normal processes cannot take over the machine

Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)
- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes `mcause`, process state to the right EH to handle the specific exception/interrupt
  3. EH returns which process should run next (could be the same or a different one)
  4. CH loads x1-x31, mepc from memory for the right process
  5. CH executes `mret`, which sets pc to mepc, disables supervisor mode, enables interrupts



Demo Code

- Log into Athena
- Clone the demo code:

```
git clone git@github.mit.edu:6004/os-demo.git
cd os-demo
```

- (Re-)Build the simulation:

```
make clean && make
```

- Run the simulation:

```
python run.py
```

- The demo code is incomplete, doesn't print anything yet
  - We will fix the code in this recitation
  - "Ctrl+C" to kill the simulation

Clarification

- The following code is for your understanding
- It uses C syntax we have not seen before
  - Structs
  - Pointers
  - Pointer arithmetic
- This content will not be on the quiz
  - No questions on this implementation of exception handlers (but you need to know the concepts)
  - We will never ask you to write C
  - We will never ask any questions that use complex C syntax (only simple C procedures with `int` and `unsigned int` like we've seen so far)

Common Exception Handler
RISC-V Assembly code

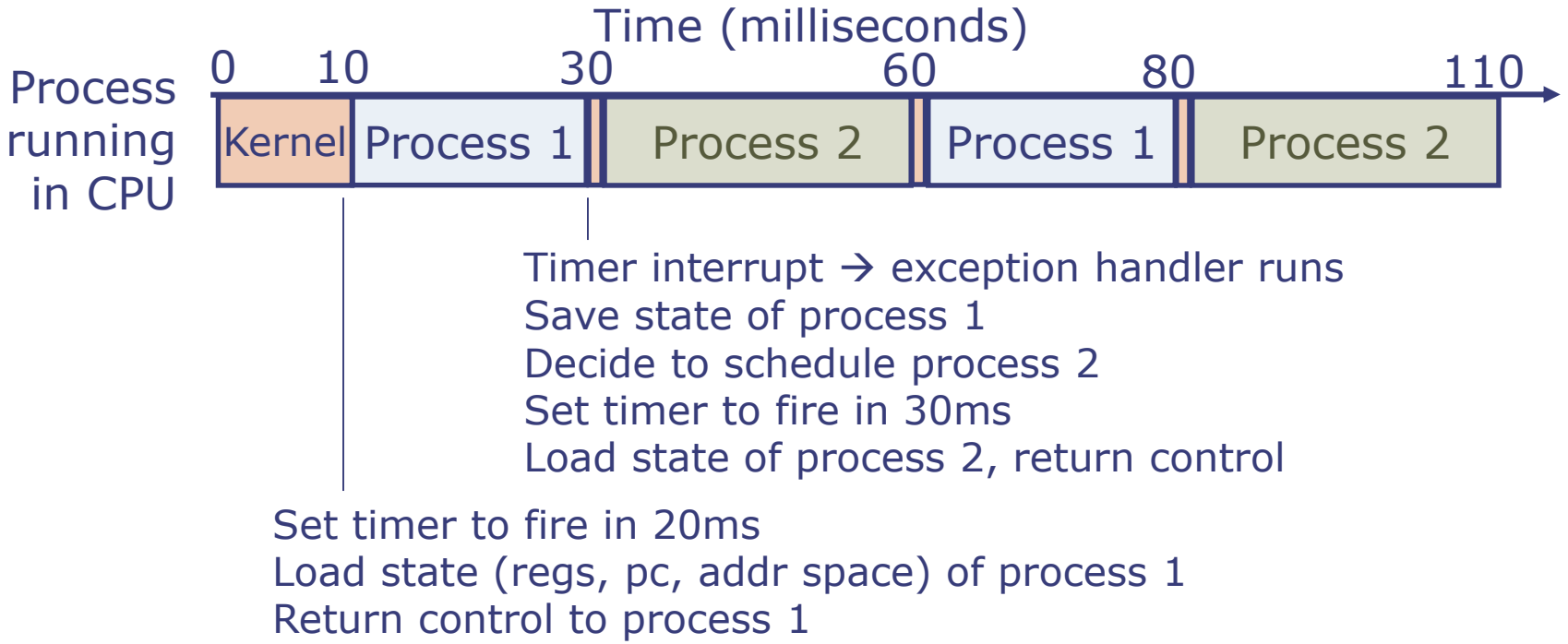
```
common_handler: // entry point for exception handler
  // save x1 to mscratch to free up a register
  csrw mscratch, x1 // write x1 to mscratch CSR
  // get the pointer for current process state
  lw x1, curProcState
  // save registers x2-x31
  sw x2, 8(x1)
  sw x3, 12(x1)
  ...
  sw x31, 124(x1)
  // now registers x2–x31 are free for the kernel
  // save original x1 (now in mscratch)
  csrr t0, mscratch
  sw t0, 4(x1)
  // finally, save mepc
  csrr t1, mepc
  sw t1, 0(x1)
```

Common Exception Handler *cont.*
Calling EH_Dispatcher and returning

```
common_handler:
  ... // we have saved the state of the process
  // call function to handle exception
  lw sp, kernelSp // use the kernel's stack
  mv a0, x1 // arg 0: address of process state
  csrr a1, mcause // arg 1: exception cause
  jal eh_dispatcher // calls the appropriate handler
  // returns address of state of process to schedule
  // restore return PC in mepc
  lw t0, 0(a0)
  csrw mepc, t0
  // restore x1-x31
  mv x1, a0
  lw x2, 8(x1); lw x3, 12(x1); ...; lw x31, 124(x1)
  lw x1, 4(x1) // restore x1 last
  mret // return control to program
```

Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time



EH Dispatcher (in C)
Dispatches to a specific handler based on "cause"

```
typedef struct {
  int pc;
  int regs[31];
  …
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {
  if (cause == TIMER_INTERRUPT)
    return interrupt_timer(curProc); // process scheduling
  else if (cause == 0x08)
    // system call, e.g, OS service "write" to file
    return syscall_eh(curProc);
  else if (cause < 0) // external interrupt
    ...
}
```

Exception Handler (in C)
Implements a round robin scheduler

- Schedule processes one after another, for a fixed amount of time

```
// an array to store state of all processes
ProcState procTbl[numProcs];

ProcState* interrupt_timer(ProcState* curProc) {
   // in C, this computes the index of curProc in procTbl
   int curPid = curProc - procTbl;
   // compute next pid accounting for roll over
   int nextPid = (curPid + 1) % numProcs;
   return &procTbl[nextPid];
}
```

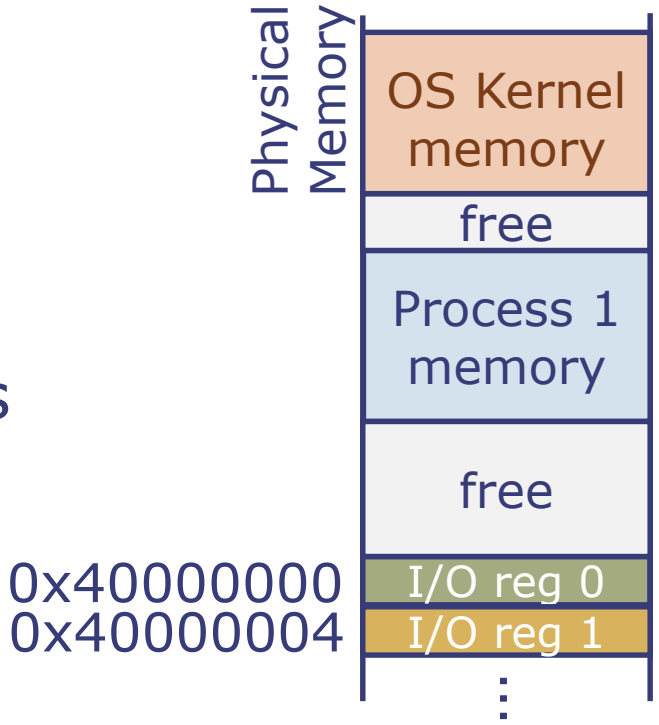
Case Study 2: I/O

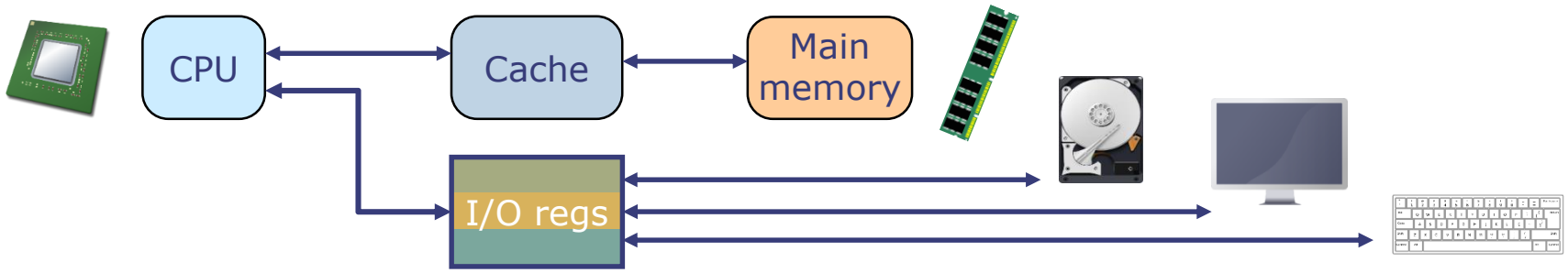
- So far, our OS does not interact with the outside world → Not very useful…

- We will use a display with memory-mapped I/O to print messages to the screen

Reminder: Memory-Mapped I/O

- I/O registers are mapped to specific memory locations
  - These locations are not accessible to normal processes, only to the OS kernel
- Processor accesses I/O registers with loads and stores





Coordinating I/O Transfers

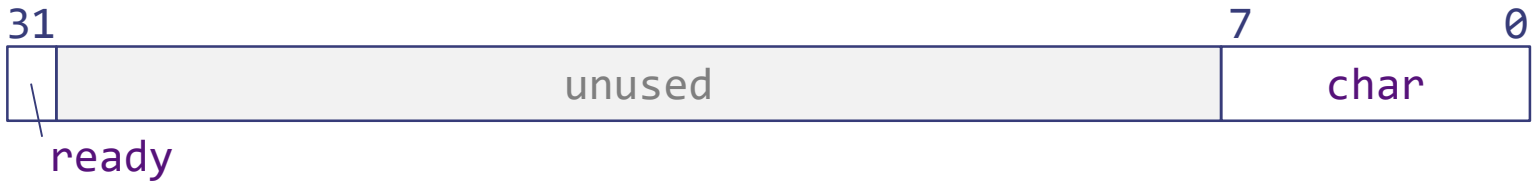
- Option 1: Polling (synchronous)
  - Processor periodically reads the register associated with a specific I/O device
- Option 2: Interrupts (asynchronous)
  - Processor initiates a request, then moves to other work
  - When the request is serviced, the I/O device interrupts the processor
- *Pros of each approach?*

Polling is simple

Interrupts let the processor do useful computation while request is serviced

Polling-Based MMIO Display

- Consider a simple character-based display
- Uses one memory-mapped I/O register with the following format:



  - The ready bit (bit 31) is set to 1 only when the display is ready to print a character
  - When the processor wants to display an 8-bit character, it writes it to char (bits 0-7) and sets the ready bit to 0
  - After the display has processed the character, it sets the ready bit to 1

Let's see a demo!

dispatcher.c

```
ProcState* eh_dispatcher(int cause, ProcState* curProc) {
  if (cause == TIMERINTERRUPT)
    return interrupt_timer(curProc); // process scheduling
  else if (cause == SYSCALL)
    // system call, e.g, OS service "write" to file
    curProc->pc += 4;
    return syscall_eh(curProc);
  else
    ...
}
ProcState* interrupt_timer(ProcState* curProc) {
  int nextPid = curProc->pid + 1;
  if (nextPid >= NUM_PROCS) nextPid = 1; // Proc0 is kernel
  print_string("Switching processes on timer interrupt\n");
  return &procTbl[nextPid];
}
```

Privileged mode print function

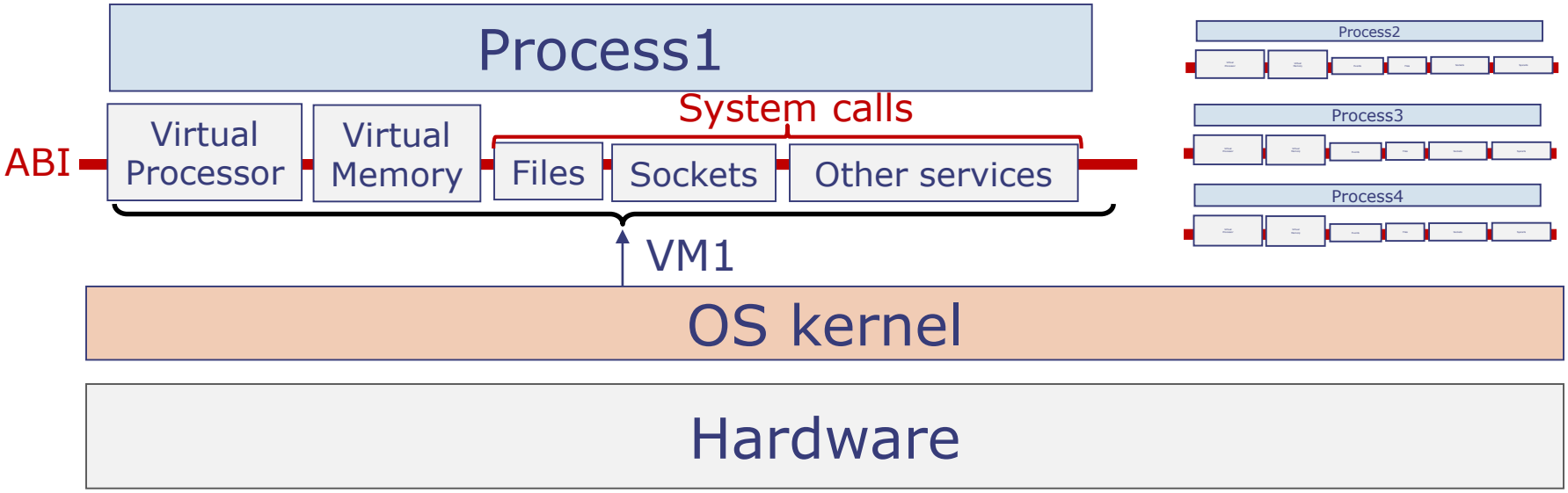
print.c: Fixing print_char

```
void print_string(char* s) {
  // Privileged mode function for printing a string
  // iterate through a character array and call print_char
}

// Modification 1 (Wrong):
void print_char(int x) {
  // character display using mmio
  *display_mmio = x; // write to the mmio register
}
// Modification 2 (Correct):
void print_char(int x) {
  // polling-based character display using mmio
  while (*display_mmio < 0x80000000); // wait until ready
  *display_mmio = x; // write to the mmio register
}
```

Case Study 3: System Calls

- The OS kernel lets processes invoke system services (e.g., access files) via system calls



- Processes invoke system calls by executing an instruction that causes an exception
  - Same mechanism as before!

System Calls in RISC-V

- `ecall` instruction causes an exception, sets `mcause` CSR to a particular value
- ABI defines how process and kernel pass arguments and results
- Typically, similar conventions as a function call:
  - System call number in a7
  - Other arguments in a0-a6
  - Results in a0-a1 (or in memory)
  - All registers are preserved (treated as callee-saved)

Let's see a demo!

System Calls: Example

proc1.user.S

```
. = 0x0
start:
    // do this main loop forever
    mv t0, zero
    li t1, 0x1000
loop:
    addi t0, t0, 1
    blt t0, t1, next
    mv t0, zero
    la a0, hello_string
    li a7, 0x13
    ecall
next:
    j loop

hello_string:
    .ascii "Hello from process 1!\n\0"
```

proc2.user.S

```
. = 0x0
start:
    // do this main loop forever
    mv t0, zero
    li t1, 0x4000
loop:
    addi t0, t0, 1
    blt t0, t1, next
    mv t0, zero
    la a0, hello_string
    li a7, 0x13
    ecall
next:
    j loop

hello_string:
    .ascii "Hello from process 2!\n\0"
```

”print” system call for user processes 1 and 2

```
SYS_print = 0x13
```

dispatcher.c:
adding print system call (2)

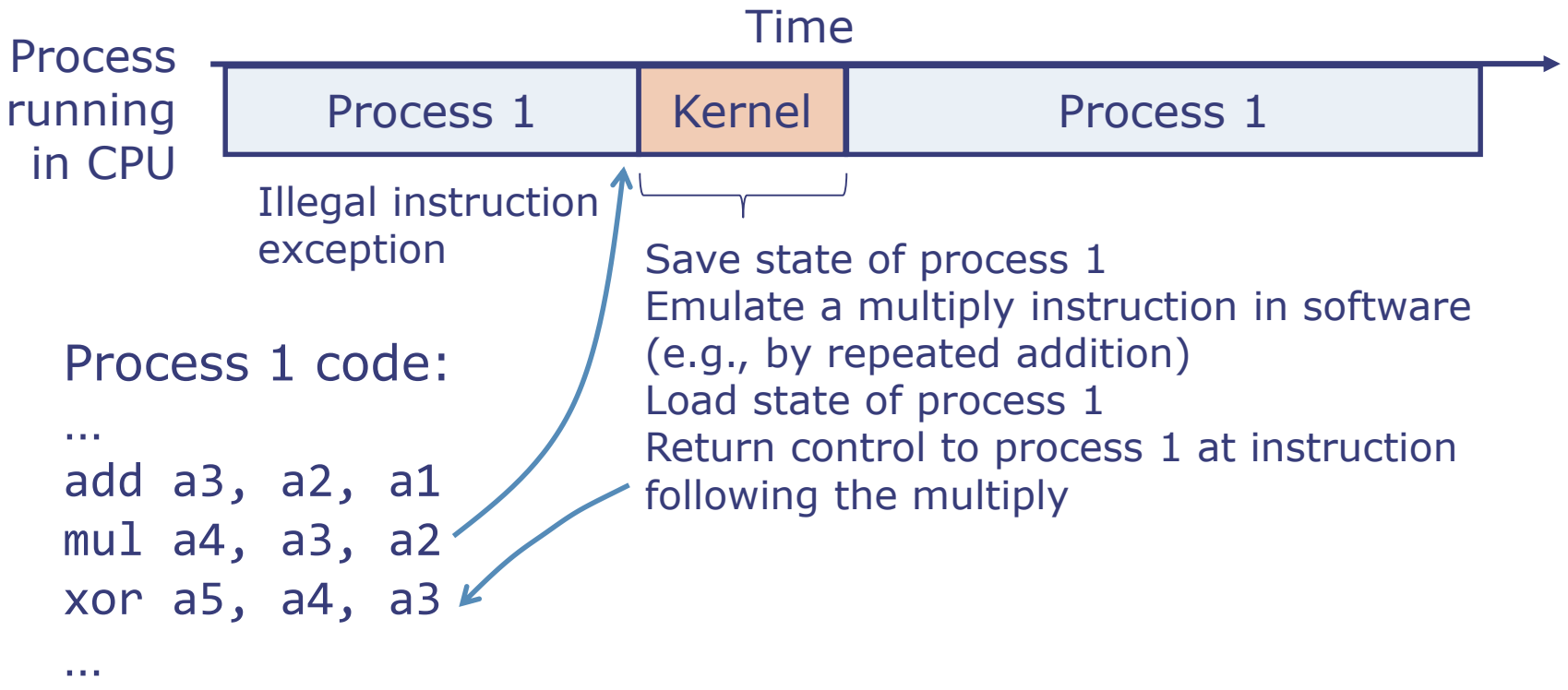
```
ProcState* syscall_eh(ProcState* curProc) {
  // Privileged mode function for printing a string
  // iterate through a character array and call print_char
  int syscall = curProc->regs[REG_a7];
  if (syscall == SYS_getpid) {
     curProc->regs[REG_a0] = curProc->pid;
  } else if (syscall == SYS_print) {
 print_string((char*)va_to_pa(curProc,curProc->regs[REG_a0]));
  } else {
    curProc->regs[REG_a0] = -128;
  }
}
```

Need to translate virtual (per-process) address to physical address
More on Lecture 18!

Case Study 4: Emulating Instructions
Enabled by illegal instruction exceptions

- mul x1, x2, x3 is an instruction in the RISC-V 'M' extension (x1 := x2 * x3)
  - If 'M' is not implemented, this is an illegal instruction
- What happens if we run code from an RV32IM machine on an RV32I machine?
  - mul causes an illegal instruction exception (mcause = 2)
- The exception handler can emulate the instruction and return to the program at mepc+4
  - Requires incrementing mepc by 4 before mret

Emulating Unsupported Instructions



- Result: Program believes it is executing in a RV32IM processor, when it's actually running in a RV32I

EH Dispatcher (in C)
Modified to handle unsupported instructions

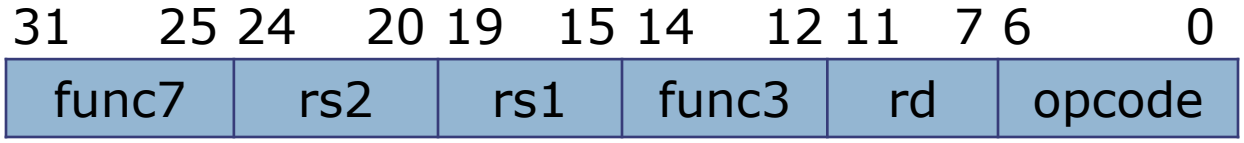
```
typedef struct {
  int pc;
  int regs[31];
  …
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {
  if (cause == TIMER_INTERRUPT)
    return interrupt_timer(curProc); // process scheduling
  if (cause == 0x02) // illegal instruction
    return illegal_eh(curProc);
  else if (cause == 0x08)
    // system call, e.g, OS service “write” to file
    return syscall_eh(curProc);
  else if (cause < 0) // external interrupt
    ...
}
```

Illegal Instruction Exception Handler

```
ProcState* illegal_eh(ProcState* curProc) {
    // load_mem fetches instruction
    int inst = load_mem(curProc->pc);
    // check opcode & function codes
    if ((inst & MASK_MUL) == MATCH_MUL) {
      // is MUL, extract rd, rs1, rs2 from inst
      int rd = (inst >> 7) & 0x01F;
      int rs1 = ...; int rs2 = ...;
      // emulate regs[rd] = regs[rs1] * regs[rs2]
      curProc->regs[rd] = multiply(curProc->regs[rs1],
                                   curProc->regs[rs2]);
      curProc->pc = curProc->pc + 4; // resume at pc+4
    } else abort();
    return curProc; }
```

Instruction decoding in software



R-type instruction

```
ProcState* illegal_eh(ProcState* curProc) {
   // load_mem fetches instruction
   int inst = load_mem(curProc->pc);
   ...
   // extract rd, rs1, rs2 from inst
   int rd = (inst >> 7) & 0x01F;
   int rs1 = (inst >> 15) & 0x01F;
   int rs2 = (inst >> 20) & 0x01F;
   ...
}
```