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.)

---

**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
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

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

<table>
<thead>
<tr>
<th>Process running in CPU</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>Process 1</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

- Timer interrupt → exception handler runs
  - Save state of process 1
  - Decide to schedule process 2
  - Set timer to fire in 30ms
  - Load state of process 2, return control

- Set timer to fire in 20ms
- Load state (regs, pc, addr space) of process 1
- Return control to process 1
Problem 1 ★

The curProcState is a structure in memory that stores the state of the current process at a known memory location. The common handler process stores all the registers (and pc) from the current process and stores them at the location pointed to by curProcState. It then calls the eh_dispatcher which deals with the exception and returns the procState memory location of the new process that must continue executing. Let us see how to implement process scheduling (switching to next process at every timer interrupt) in code with these handlers. Assume the following ProcState structure, and that the eh_dispatcher uses addresses to this in order to store process state:

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

ProcState procTbl[numProcs]; // global process state table
```

(A) First, write the function signature of the eh_dispatcher function, based on what arguments it takes from the common_handler code (see comments):

```c
ProcState* eh_dispatcher(ProcState* curProc, int cause) { ... }
```

We know that the eh_dispatcher function returns a ProcState* by definition. From the common_handler code, we know that the first argument is the location of the current process state, therefore the first argument is ProcState* curProc, the second argument is the cause number, therefore int cause. Reiterate that for all of this problem, they do not need to know the exact the C syntax, just the overall ideas.

(B) The eh_dispatcher function checks if the cause is TIMER_INTERRUPT, and if it is, then it knows it is time to switch to the next process. To do so, it calls interrupt_timer with the current process state to get the next process state.

```c
ProcState* eh_dispatcher(ProcState* curProc, int cause) {
    if (cause == TIMER_INTERRUPT)
        return interrupt_timer(curProc); // process scheduling
    else if (...)
        ...
}
```

(C) Finally, implement interrupt_timer, which finds increments the process index and returns the new process state address.

```c
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];
}
```
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

Polling-Based MMIO Display

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

<table>
<thead>
<tr>
<th>31</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>char</td>
<td></td>
</tr>
</tbody>
</table>

  - 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
Problem 2 ★
In this problem, we implement a system call that does simple polling based text display I/O. In order to use this I/O, the process makes a system call (using the ecall instruction) with the call number SYS_print (= 0x13) set in a7 (to indicate it is trying to write to display) and sets the address of the string to be printed as the first argument (in a0).

```
start:
    la a0, hello_string
    li a7, 0x13
    ecall

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

(A) First, extend our eh_dispatcher to check if the cause is SYSCALL (i.e. the cause number that informs whether this is a system call initiated through ecall). If it is, we call syscall_eh with the current process state, which handles the system calls. **Hint: Does eh_dispatcher need to do something with the current process pc as well?**

```
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
        ...
}
```

We need to increment the pc because the execution needs to resumed from the next instruction after the ecall since the ecall has already been processed.

(B) Next, syscall_eh needs to check the appropriate register to check the type of syscall being performed to figure out which action to do (read/write from I/O, file I/O, etc). Assume that the following print_string function exists that prints directly to the display.

```
void print_string(char* s);

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_print) {
        print_string((char*)va_to_pa(curProc,curProc->regs[REG_a0]));
    } else if { ... }
}
```

The syscall number is stored in a7. The va_to_pa function walks the page table and performs address translation that we will learn about in the Virtual Memory lecture.
(C) Implement the print_string function using that print_char function that takes a character and prints it to the display. Assume the following print_char function exists.

```c
void print_char(char x);
void print_string(char* s) {
    while (*s != 0) {
        print_char(*s);
        s++;
    }
}
```

All we are doing is incrementing s till the null-termination is reached and printing the current character. Strings are stored with a zero at the end to mark their end.

(D) Finally, implement the print_char function that uses the register from the last slide to put text on the terminal. Assume that the global variable display_mmio stores the address of where the I/O register from the previous slide resides.

```c
void print_char(int x) {
    while (*display_mmio < 0x80000000); // wait until ready
    *display_mmio = x; // write to the mmio register
}
```

We need to wait till the display is ready to receive a character by making sure the MSB is set. Once it is, clear the MSB and set the lowest 8 bits to the character to be printed. The second line does both these actions at once.
Problem 3 ★

Let us see how to emulate non-existent instructions. For example, this problem will implement the multiply instruction, which does not exist in the RV32I instruction set, but is found in the RV32IM extension.

\[
\text{mul x1, x2, x3}; \ // \ \text{reg[x1] <= reg[x2] * reg[x3]};
\]

When the processor reaches this instruction, it dispatches an exception with the cause number 0x02.

(A) Extend the eh_dispatcher to handle this condition. Is should call illegal_eh with the curProc which does the handling of the instruction.

```c
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 == 0x02) // illegal instruction
        return illegal_eh(curProc);
}
```

(B) Note that the curProc->pc stores the pc value at which the illegal instruction occurred. Use the load_mem function with this pc value to get the bits of the instruction. Then, decode the fields from the instruction. Check the opcode to check if it is a multiply, then finally, perform the multiply and store the result in the destination register.

```c
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;

    // check opcode & function codes
    if ((inst & MASK_MUL) == MATCH_MUL) {
        // is MUL, 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
    }
    return curProc;
}
```
Problem 4 (Fall 2020 Quiz 3)
Consider the following two processes running RISC-V programs labeled with virtual addresses. All pseudoinstructions in these programs translate into a single RISC-V instruction.

program for process A:
\[ \text{.} = 0x50 \]
\[ \text{li a0, 0x250} \quad // \text{load address of string to be printed} \]
\[ \text{li a7, 0x13} \quad // \text{set system call number for print_string} \]
\[ \text{ecall} \]
\[ \text{mv a0, a7} \]
\[ \text{ret} \]
\[ \text{.} = 0x250 \]
\[ \text{string:} \]
\[ \text{.ascii "printing from program A"} \]

program for process B:
\[ \text{.} = 0x550 \]
\[ \text{li a0, 13} \]
\[ \text{li a1, 7} \]
\[ \text{rem a0, a0, a1} \quad // \text{rem rd, rs1, rs2 stores rs1 modulo rs2 into rd} \]
\[ \text{li a2, 0x700} \]
\[ \text{mv a0, a2} \]
\[ \text{sw a0, 0(a2)} \]
\[ \text{ret} \]

For each of the following questions (A-F), either provide the requested answer, or if there is not enough information to determine the answer, then write CAN’T TELL. You must also provide an explanation for your answers. Assume all memory accesses are legal for parts (A) through (D). Also, assume that neither process A nor B has run yet.

(A) (3 points) The OS schedules process B first, but the processor does not support the \text{rem} instruction, so the OS emulates it in software. What are the values of a0 and pc (in virtual address) when the common handler returns to process B after emulating the \text{rem} instruction?

\[ \text{a0: } 6 \]
\[ \text{pc: } 0x55C \]

Explanation:

\text{After the OS emulates rem and returns control, a0 stores the result of rem, 13 mod 7 = 6} 

\text{The OS returns control to the instruction following rem, at pc = 0x550 + 3*4 = 0x55C}
(B) (3 points) Just prior to process B executing \texttt{li \ a2, \ 0x700}, a timer interrupt occurs and the OS switches to process A. What are the values of a0 and pc (in virtual address) when the common handler returns to process A?

\begin{itemize}
  \item \textbf{a0:} \underline{Can’t tell}
  \item \textbf{pc:} \underline{0x50}
\end{itemize}

\textbf{Explanation:}

Since A hasn’t run yet, the pc is the pc of the first instruction, and a0 is at its initial value, which is unknown ($a0 = 0$ is also correct if you explain that you assume all registers start initialized to 0).

(C) (3 points) After process A executes \texttt{ecall}, what is the value of pc (in virtual address) when the common handler returns to process A?

\begin{itemize}
  \item \textbf{pc:} \underline{0x5C}
\end{itemize}

\textbf{Explanation:}

The OS returns control to the instruction following \texttt{ecall}, at \texttt{pc = 0x50 + 3*4 = 0x5C}
(D) (3 points) Just prior to process A executing the \texttt{mv} instruction, a second timer interrupt occurs and the OS switches to process B. What are the values of a0 and pc (in virtual address) when the common handler returns to process B?

\begin{align*}
a0: & \_6 \_ \\pc: & 0\times55C\_ \\
\end{align*}

Explanation:

The OS returns control to the \texttt{li} a2, 0x700 instruction, where the interrupt happened. This is at pc = 0x550 + 4*3 = 0x55C, same as (A); a0 is also the same as in (A), since process B has not modified it.