Communicating with I/O devices

Operating Systems: I/O and System Calls

Accessing I/O registers

- Option 1: Use special instructions
  - e.g., `in` and `out` instructions in x86
  - Inflexible, adds instructions → used rarely

- Option 2: Memory-mapped I/O (MMIO)
  - I/O registers are mapped to physical memory locations
  - Processor accesses them with loads and stores
  - These loads and stores should not be cached!

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
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)

Demo code

- Clone the demo code into your VM:
  ```bash
git clone git@github.mit.edu:6004/os-demo.git ~/(os-demo)
```
- (Re-)Build the simulation:
  ```bash
make clean && make
```
- Run the simulation:
  ```bash
python run.py
```
- The demo code is buggy; doesn’t print anything yet
  - We will fix the code in this recitation
  - “Ctrl+C” to kill the simulation

Example 1: Polling-based I/O

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

```
+------------------+
| 31                  |
| 30                  |
| 29                  |
| 28                  |
| 27                  |
| 26                  |
| 25                  |
| 24                  |
| 23                  |
| 22                  |
| 21                  |
| 20                  |
| 19                  |
| 18                  |
| 17                  |
| 16                  |
| 15                  |
| 14                  |
| 13                  |
| 12                  |
| 11                  |
| 10                  |
| 09                  |
| 08                  |
| 07                  |
| 06                  |
| 05                  |
| 04                  |
| 03                  |
| 02                  |
| 01                  |
| 00                  |
+------------------+
|     unused      |
|     7           |
|     char       |
```

- 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!
```
print.c: fixing print_char

```c
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
    }
}
```

Example 2: Interrupt-based I/O

- Consider a simple keyboard that uses a single I/O register with a similar format:

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

- On a keystroke, the keyboard writes the typed character in char, sets full bit to 1, and raises a keyboard interrupt
- Interrupt handler reads char and sets full bit to 0, so the keyboard can deliver future keystrokes
- This works fine because keyboard is very slow compared to CPU. Faster devices use more sophisticated mechanisms (e.g., network cards write packets to main memory and use I/O registers only to indicate status of transfer)

dispatcher.c

```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
        ...
}
```

Not implemented in this demo

- Keyboard Interrupt Handler needed;
  - reads from the register and processes the keystroke
  - clears the full bit [31] of the register
  - returns a process to resume

Communicating with the OS

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

```
Virtual Processor   Virtual Memory   Files   Sockets Other services
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- 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!

**dispatcher.c:**
**adding print system call (1)**

```c
// Modification 1 (Wrong):
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*) curProc->regs[REG_a0]);
    } else {
        curProc->regs[REG_a0] = -128;
    }
}
```

**dispatcher.c:**
**adding print system call (2)**

```c
// Modification 2 (Correct):
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;
    }
}
```

**System Calls: Example**

```
proc1.user.S
.

.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
.

.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

```

```c
// Modification 2 (Correct):
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;
    }
}
```
Typical System Calls

- Accessing files (sys_open/close/read/write/…)
- Using network connections (sys_bind/listen/accept/…)
- Managing memory (sys_mmap/munmap/mprotect/…)
- Getting information about the system or process (sys_gettimeofday/getpid/getuid/…)
- Waiting for a certain event (sys_wait/sleep/yield/…)
- Creating and interrupting other processes (sys_fork/exec/kill/…)
- ... and many more!

Programs rarely invoke system calls directly. Instead, they are used by library/language routines

Some of these system calls may block the process!

Process Life Cycle: The Full Picture

- OS maintains a list of all processes and their status
  {ready, executing, waiting}
  - A process is scheduled to run for a specified amount of CPU time or until completion
  - If a process invokes a system call that cannot be satisfied immediately (e.g., a file read that needs to access disk), it is blocked and put in the waiting state
  - When the waiting condition has been satisfied, the waiting process is woken up and put in the ready list

Summary

<table>
<thead>
<tr>
<th>ABI</th>
<th>Physical Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processor</td>
</tr>
<tr>
<td>ISA</td>
<td>Supervisor mode</td>
</tr>
</tbody>
</table>

- System calls (ecall) are typically slow
  - Transfer to the common exception handler
  - Invoke the right system call handler
  - Save / restore user process’s registers

- fastcall: system call as a function call
  (+) Less overhead
    - no common handler, less saving/restoring of registers
  (-) Security / Protection Broken !!
    - User processes can jump anywhere in the kernel, bypassing protection mechanisms;

Takehome problem