CS 61C | OS & I/O
Summer 2020 | Discussion 10: July 22, 2019

## 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Polling and interrupts are only relevant concepts for low level programming.

False. Similar concepts apply to almost all types of applications, including web apps, mobile apps, distributed systems, and so on.

1.2 Memory-mapped IO only works with polling.

False. The implementation backing the memory mapping can use interrupt-driven IO (for example, reading files).

1.3 Responsibilities of the OS include loading programs, handling services, multiplexing resources, and combining programs together for efficiency.

False. While the OS is responsible for loading programs, handling services (such as the network stack and the file system), and multiplexing resources for multiple programs, it is actually responsible for isolating programs from each other so that a given program doesn't interfere with another program's memory or execution.

1.4 The purpose of supervisor mode is to isolate certain instructions and routines from user programs.

True. In the case that a program is buggy or malicious, supervisor mode limits the impact of the program on the computer, since the OS maintains control over all the resources.

1.5 User programs call into OS routines using system calls.

True. System calls, or syscalls, allow user programs to execute the OS routine in supervisor mode before switching back to user mode.

## 2 Polling & Interrupts

2.1 Fill out this table that compares polling and interrupts.

| **Operation** | **Definition** | **Pro/Good for** | **Con** |
|---|---|---|---|
| Polling | Forces the hardware to wait on ready bit (alternatively, if timing of device is known, the ready bit can be polled at the frequency of the device). | • Low Latency<br>• Low overhead when data is available<br>• Good For: devices that are always busy or when you can't make progress until the device replies | • Can't do anything else while polling<br>• Can't sleep while polling (CPU always at full speed) |
| Interrupts | Hardware fires an "exception" when it becomes ready. CPU changes PC register to execute code in the interrupt handler when this occurs. | • Can do useful work while waiting for response<br>• Can wait on many things at once<br>• Good for: Devices that take a long time to respond, especially if you can do other work while waiting. | • Nondeterministic when interrupt occurs<br>• interrupt handler has some overhead (e.g. saves all registers, flush pipeline, etc.)<br>• Higher latency per event<br>• Worse throughput |

## 3 Memory Mapped I/O

3.1 For this question, the following addresses correspond to registers in some I/O devices and not regular user memory.

- `0xFFFF0000`—Receiver Control: LSB is the ready bit (in the context of polling), there may be other bits set that we don't need right now.
- `0xFFFF0004`—Receiver Data: Received data stored at lowest byte.
- `0xFFFF0008`—Transmitter Control: LSB is the ready bit (in the context of polling), there may be other bit set that we don't need right now.
- `0xFFFF000C`—Transmitter Data: Transmitted data stored at lowest byte.

Recall that receiver will only have data for us when the corresponding ready bit is 1, and that we can only write data to the transmitter when its ready bit is 1. Write RISC-V code that reads byte from the receiver (busy-waiting if necessary) and writes that byte to the transmitter (busy-waiting if necessary).

```
                lui t0 0xffff0
receive_wait:   lw t1 0(t0)
                andi t1 t1 1             # poll on ready of receiver
                beq t1 x0 receive_wait
                lb t2 4(t0)              # load data
```

```
transmit_wait:  lw t1 8(t0)             # poll on ready of transmitter
                andi t1 t1 1
                beq t1 x0 transmit_wait # write to transmitter
                sb t2 12(t0)
```

## 4 Forking

One of the many responsibilities of the OS is to load new programs, and in order to do this it creates a new process and loads in the program to execute. In Linux, the system call to create a new process is fork(). fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. In the parent process, fork() returns the process ID of the child or -1 if the fork has failed. In the child process, it returns 0.

Use this information to complete the code block below, which creates a child process to change the value of y while the parent process changes the value of x.

```
int x = 10;
int y = 0;
int pid = __________;
if(___________){
    y++
}
else{
    x--;
}
```

```
int x = 10;
int y = 0;
int pid = fork();
if(pid == 0){
    y++
}
else{
    x--;
}
```