Operating Systems:

Chap 5: I/O Software and Interrupts

CS400

Week 12: 31st March

Spring 2020

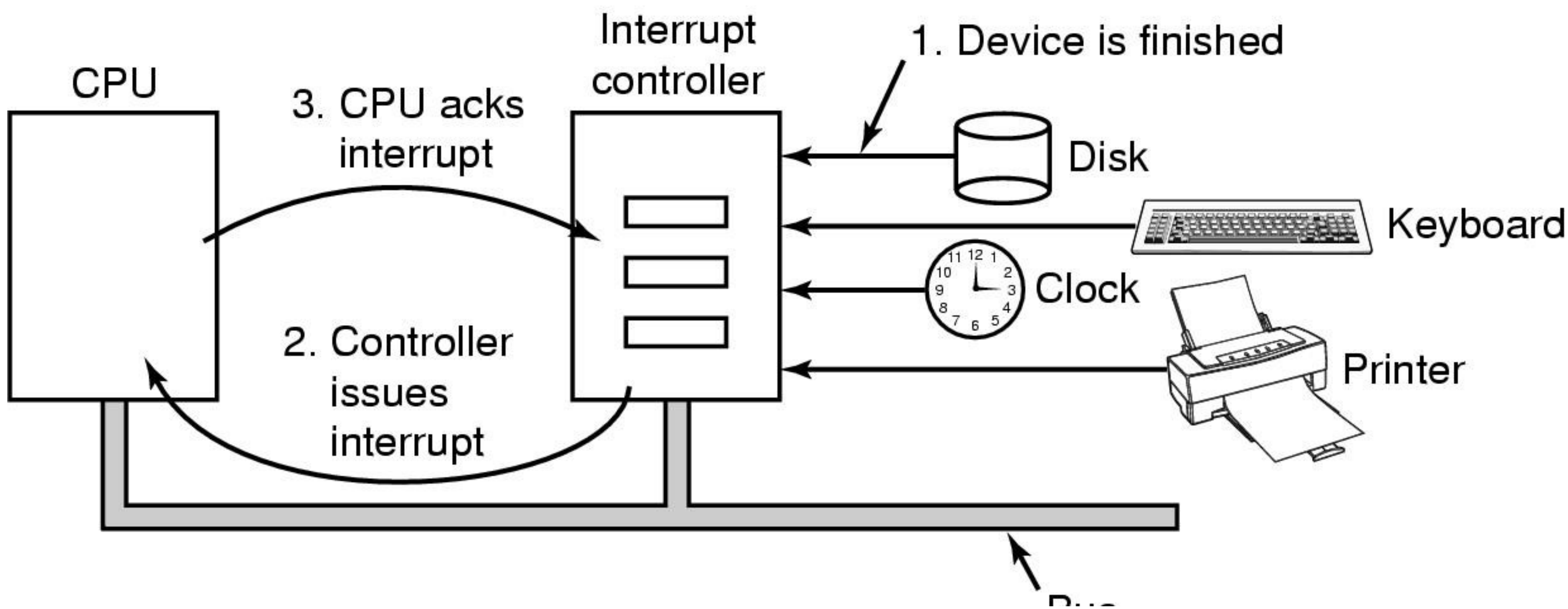
Oliver BONHAM-CARTER

Participation 5: Instructions at the end of slides.

Three common ways that I/O can be performed

- Programmed I/O
  - *Code to accept and process data*
- I/O using DMA
  - Direct Memory Access-based use of devices
- **Interrupt-Driven I/O**
  - **Exceptions and other *unusual* input, output**

Example of Interrupts



How interrupts happens. Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires

Programmed I/O (*Polling*), 0

- Polling is the process where the computer or controlling device waits for an external device to check for its readiness or state, often with low-level hardware.

Programmed I/O (*Polling*), 1

- Used when device and controller are relatively quick to process an I/O operation
- Device driver
  - Gains access to device
  - Initiates I/O operation
  - Loops testing for completion of I/O operation
  - If there are more I/O operations, repeat

Programmed I/O (*Polling*), 2

- Used in following kinds of cases
  - Service interrupt time is greater than device response time
  - Device has no interrupt capability
  - Embedded systems where CPU has nothing else to do

Programming and Interrupts: Keyboard & Mouse (1)

- Keyboard & mouse buttons implemented as 128-bit read-only register
  - One bit for each key and mouse button
  - *0 = “up”; 1 = “down”*
- Mouse “wheels” implemented as pair of counters
  - One click per unit of motion in each of *x* and *y* directions
- Clock interrupt every 10 msec
  - Reads keyboard register, compares to previous copy
  - Determines key & button transitions up or down
  - Decodes transition stream to form character and button sequence
  - Reads and compares mouse counters to form motion sequence

Programming and Interrupts: Keyboard & Mouse (2)

- Clock interrupt every 10 msec
  - Reads keyboard register, compares to previous copy
  - Determines key & button transitions up or down
  - Decodes transition stream to form character and button sequence
  - Reads and compares mouse counters to form motion sequence

Programming and Interrupts: Other examples (3)

- Check status of device
- Read from disk or boot device at boot time
    - No OS present, hence no interrupt handlers
    - Needed for bootstrap loading of the inner portions of kernel
    - Bootstrapping: automatically executed by the processor when turning on the computer. The bootstrap loader reads the hard drives boot sector to continue the process of loading the computer's operating system.
- External sensors or controllers
    - Real-time control systems

So, What’s an Interrupt?

- A signal to the processor emitted by hardware or software indicating an event that needs immediate attention.
- An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.
- The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event.
- This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities.

Hardware Interrupts

- Used by devices to communicate
- Require OS for support
- Alerting signals that are sent to the processor from an device
- Ex: pressing a key on the keyboard or moving the mouse cause interrupts forcing processor to read the keystroke or mouse position.

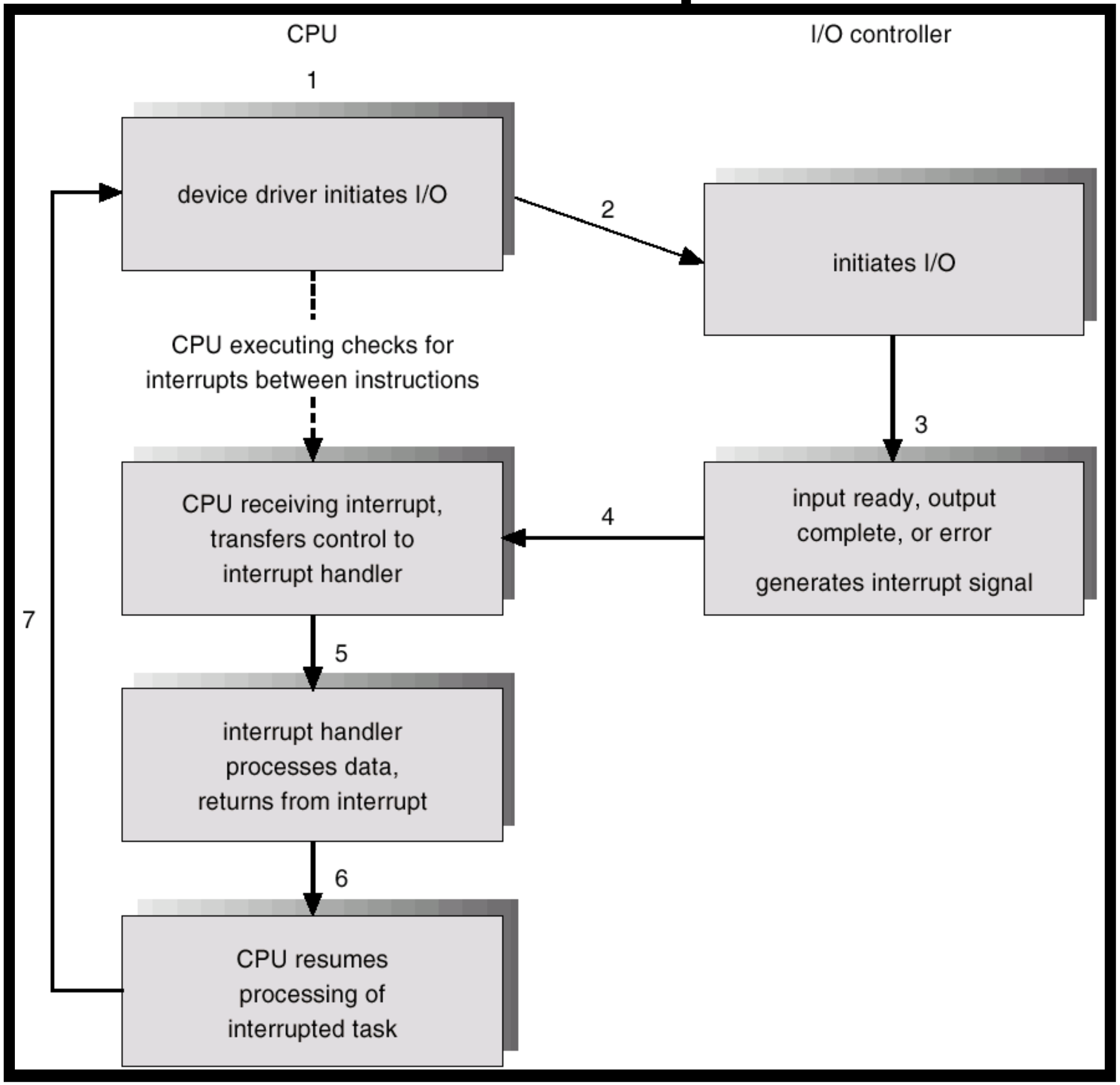
Software Interrupts

- Caused either by:
  - (A) an exceptional condition in the processor itself,
  - (B) special instruction in the instruction set which causes an interrupt when it is executed.
- Part A: *Traps* or *exceptions* and are used for errors or events occurring during program execution that are so exceptional, that they cannot be handled within the program itself.
- Ex: catching divide-by-zero errors

Interrupt Handling

- Interrupts occur on I/O events
    - operation completion
    - Error or change of status
    - Programmed in DMA command chain
- Interrupt
    - stops CPU from continuing with current work
    - Saves some context
    - restarts CPU with new address & stack
        - Set up by the interrupt vector
        - Target is the interrupt handler

Interrupts


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for
interrupts between instructions
3
CPU receiving interrupt,
transfers control to
interrupt handler
4
input ready, output
complete, or error
generates interrupt signal
7
5
interrupt handler
processes data,
returns from interrupt
6
CPU resumes
processing of
interrupted task

Interrupts Request Lines (IRQs)

- Every device is assigned an IRQ
  - Used when raising an interrupt
  - Interrupt handler can identify the interrupting device
- Assigning IRQs
  - In older and simpler hardware, physically by wires and contacts on device or bus
  - In most modern PCs, etc., assigned dynamically at boot time

Programming Interrupts in C

- #include<signal.h>
  - A resource in the include file that contains pre-written (standard) code for basic signal handling.
  - http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html
  - "*Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-2001, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.*"
- Standards for coding interrupts in C.

Handling Interrupts in Linux

- **Terminology**
  - Interrupt context – kernel operating not on behalf of any process
  - Process context – kernel operating on behalf of a particular process
  - User context – process executing in user virtual memory
- **Interrupt Service Routine (ISR), also called Interrupt Handler**
  - The function that is invoked when an interrupt is raised
  - Identified by IRQ
  - Operates on Interrupt stack (as of Linux kernel 2.6)
    - One interrupt stack per processor; approx 4-8 kbytes

Handling Interrupts in Linux

- ***Top half*** – does minimal, time-critical work necessary
  - Acknowledge interrupt, reset device, copy buffer or registers, etc.
  - Interrupts (usually) disabled on current processor
- ***Bottom half*** – the part of the ISR that can be deferred to more convenient time
  - Completes I/O processing; does most of the work
  - Interrupts enabled (usually)
  - Communicates with processes
  - Possibly in a kernel thread (or even a user thread!)

What is the *Top* and *Bottom* Half?
http://www.makelinux.net/ldd3/chp-10-sect-4.shtml

Programming Tips

- Interrupts must be carefully and cautiously handled, mainly because carelessly written interrupts can lead to some mysterious run-time errors.
- These errors are difficult to uncover and understand since the controller might enter into an undefined state, report invalid data, halt, reset, or otherwise behave in an incomprehensible manner.

Examples of Interrupt Signals

- **SIGQUIT**: Terminal quit signal.
- **SIGSEGV**: Invalid memory reference.
- **SIGSTOP**: Stop executing (cannot be caught or ignored).
- **SIGTERM**: Termination signal.
- **SIGTSTP**: Terminal stop signal.
- **SIGTTIN**: Background process attempting read.
- **SIGTTOU**: Background process attempting write.
- **SIGUSR1**: User-defined signal 1.
- **SIGUSR2**: User-defined signal 2.

Let’s Code!

Up Next!

Commands to Run
From (Linux) Bash

- Build the container :
  - `docker build -t gccdev .`
- Run the container :
  - `docker run -it gccdev`
- Mount local drive and run container :
  - `docker run -it --mount type=bind,source=$PWD,target=/home/gccdev gccdev`

Note: the directory where you run this becomes your local directory in the container.

Using a Docker container with one terminal for your coding? Please run your signalHandler program in the background using the “&” argument. Read on to see how. Please ask if you have questions. Read more: https://linuxhandbook.com/run-process-background

05_part/src/signalHandler.c

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
// compile: gcc -o signalHandler signalHandler.c

void sig_handler(int signalType)
{
    if (signalType == SIGUSR1) // first type of signal to handle
        printf("  received SIGUSR1\n");
    else if (signalType == SIGKILL) // second type of signal to handle
        printf("  received SIGKILL\n");
    else if (signalType == SIGSTOP)// third type of signal to handle
        printf("  received SIGSTOP\n");
}

int main(void)
{
    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("\n  Cannot catch SIGUSR1\n"); // catch first type of signal
    if (signal(SIGKILL, sig_handler) == SIG_ERR)
        printf("\n  Cannot catch SIGKILL\n"); // catch second type of signal
    if (signal(SIGSTOP, sig_handler) == SIG_ERR)
        printf("\n  Cannot catch SIGSTOP\n"); // catch third type of signal

    // A long long wait so that we can easily issue a signal to this process
    while(1)
        sleep(1);
    return 0;
}
```

Compile

- We will spend some time to investigate how interrupts are communicated across two Linux terminals
- Locate the code: 05_part/src/`signalHandler.c`
  - Compile: gcc -o signalHandler signalHandler.c
  - Run executable from first terminal: ./signalHandler

Run

- Running: ./signalHandler
- Run in the background: ./SignalHandler &
  - If you run the program in the background, then you will see the responses from your code in your terminal when you run the `kill` commands. (up next).

```
./signalHandler

 Cannot catch SIGKILL

 Cannot catch SIGSTOP
```

From New Terminal, Find PID

- Next open a second terminal.
- We need to find process ID (number) of “signalHandler” in the OS:
  - ps -aux | grep signalHandler
  - (Scans all processes and finds the one called “signalHandler”)

```
$ ps -aux | grep signalHandler
obonham+ 24457  0.0  0.0   2480   708 pts/2    S+   22:53   0:00 ./signalHandler
obonham+ 24609  0.0  0.0   9028   988 pts/3    S+   22:57   0:00 grep --color=auto signalHandler
```

obonham+ **24457** 0.0 0.0 2480 708 pts/2 S+ 22:53 0:00 ./signalHandler

The Kernel’s **Process ID**
Note, this number will be different each time you run this program.
Currently, the PID is 24457

Study Code

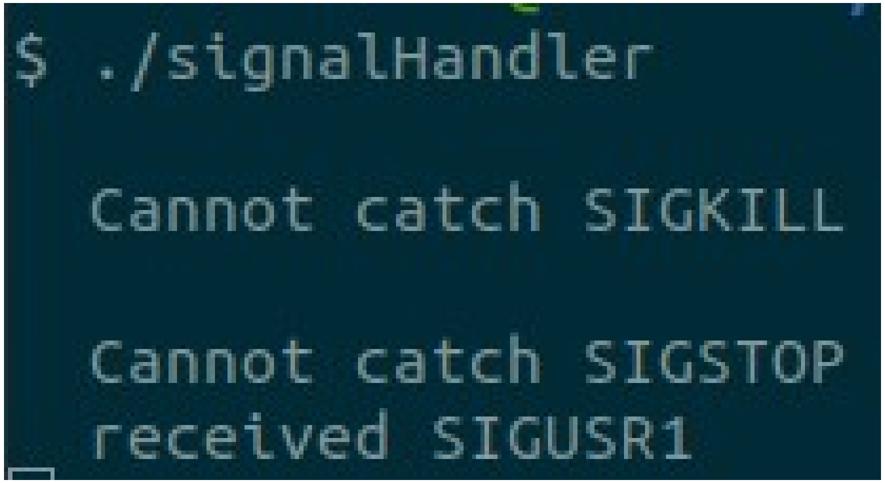
- In the signalHandler.c code, there is a block;

```
void sig_handler(int signalType)
{
   if (signalType == SIGUSR1) // first type of signal to handle
      printf("  received SIGUSR1\n");
   else if (signalType == SIGKILL) // second type of signal to
handle
      printf("  received SIGKILL\n");
   else if (signalType == SIGSTOP)// third type of signal to handle
      printf("  received SIGSTOP\n");
}
```

- Interrupts are being handled by the code.

Experiment

- From the second terminal (that you just used to find the pid) type the following command.
  - kill -s SIGUSR1 24457 # signal, pid



- What did you observe?

Participation 5: Hack Your Code

- Now, try adding the signal handler for SIGUSR2 to your code and rerun. What happens when you send this signal from the second terminal?

Other signals to try out in your code. Do they do anything?

```
| Signal      | Output                        |
|-------------+-------------------------------|
| SIGKILL     | Killed: 9                     |
| SIGQUIT     | Quit: 3                       |
| SIGILL      | Illegal instruction: 4        |
| SIGABRT     | Abort trap: 6                 |
| SIGFPE      | Floating point exception: 8   |
| SIGPIPE     | (no output)                   |
| SIGALAR     | Alarm clock: 14               |
| SIGUSR1     | User defined signal 1: 30     |
| SIGUSR2     | User defined signal 2: 31     |
|-------------+-------------------------------|
```

Participation 5: Instructions

- Go find other code to implement at; https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/19/lec.html
- Leave your modified `signalHandler.c` and other code from the website in your source directory: **05_part/scr.**
- Describe your experience in **`05_part/writing/reflections.md`**
- General Participation Repository
  - https://classroom.github.com/a/S8lbI9Z5

Submit by Friday 3rd April at midnight