Lecture 2 — Interrupts & System Calls

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

June 5, 2018

A regular program like a word processor need not be concerned with the underlying hardware of the computer.

The OS must be aware of the details and manage them.

What is a program? Instructions and data.

To Execute a Program

To execute a program we need:

1. **Main Memory**
2. **System Bus**
3. **Processor**

Of course, this is the minimal set.

Central Processing Unit (CPU)

The processor (CPU) is the brain of the computer.

Fetch instructions, decode them, execute them.

Fetch-decode-execute cycle repeated until the program finishes.

Different steps may be completed in parallel (pipeline).

Processors' largest unit is the word.
32-bit computer $\rightarrow$ 32-bit word. 64-bit computer $\rightarrow$ 64-bit word.

Central Processing Unit (CPU)

CPU instructions are specific to the processor.

Written assembly? You know the books.

Some operations only available in supervisor mode.
    Attempting to run it in user mode is an error.

Central Processing Unit (CPU)

CPUs have storage locations: registers.

They may store data or instructions.

Management of registers is partly the role of the OS.

Let us examine a few of the critical registers.

CPU Registers

A few of the registers in a typical CPU:

- **Program Counter** – Next instruction.
- **Status Register** – Array of bits to indicate flags.
- **Instruction Register** – Instruction most recently fetched.
- **Stack Pointer** – Top of the stack.
- **General Purpose Registers** – Store data, addresses, etc.

Interrupts

The CPU needs data, but it takes a variable amount of time to get it.
Sometimes this means the equivalent of walking a book from Ottawa.

In the meantime, I should do something else.

Polling: check periodically if the book has arrived.

Interrupts: get a notification when the book is here.

If someone knocks on my door, I pause what I'm doing and get the book.

Sources of Interrupts

We can put interrupts into four categories, based on their origin:

1. **Program.**
2. **Timer.**
3. **Input/Output.**
4. **Hardware Failure.**

Interrupts

Interrupts are a way to improve processor utilization.

CPU time is valuable!

When an interrupt take place, the CPU might ignore it (rarely).

More commonly: we need to handle it in some way.

Analogy: professor in a lecture; student has a question.

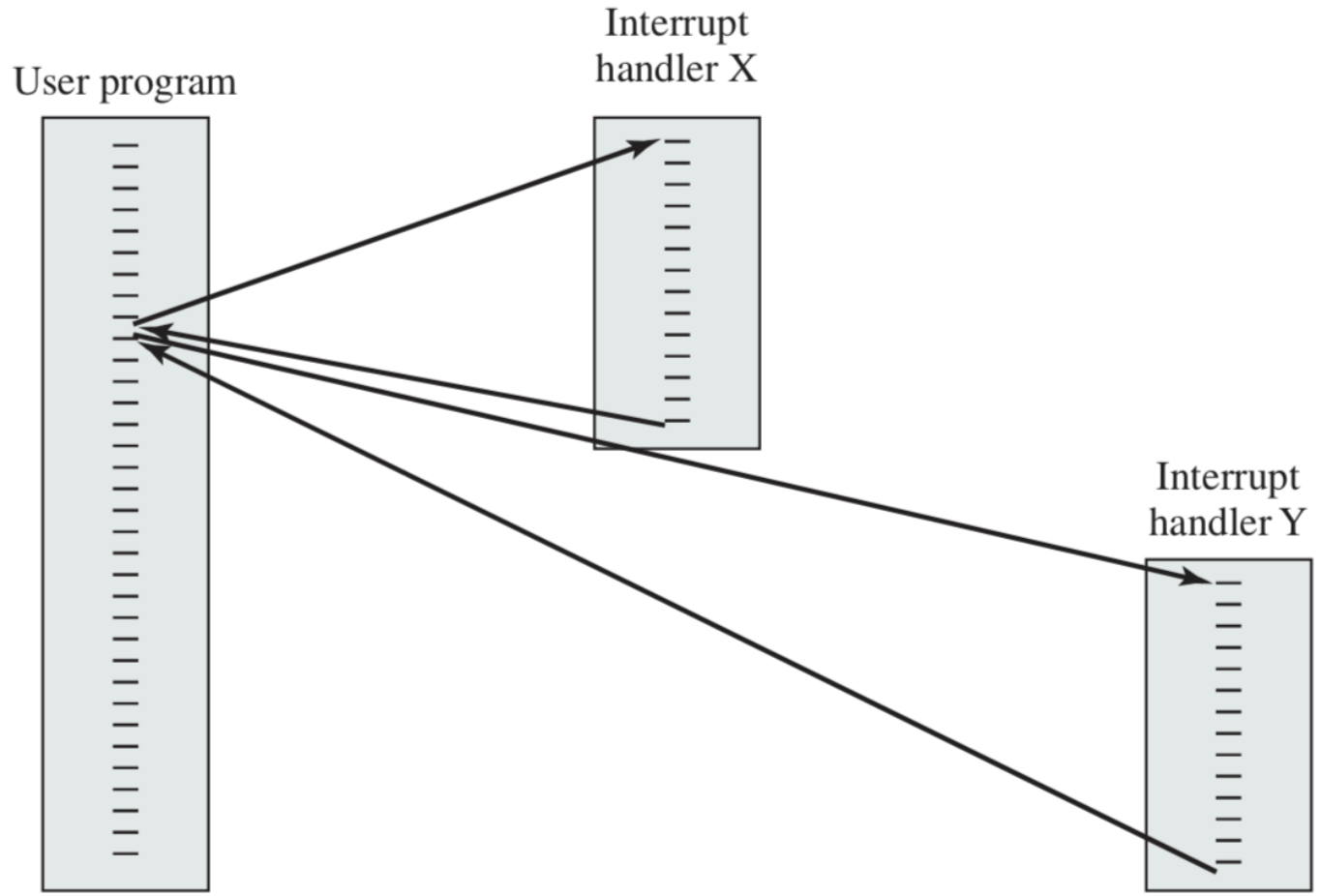
Interrupts

The OS: stores the state, handles the interrupt, and restores the state.

Sometimes the CPU is in the middle of something uninterruptible.
    Interrupts may be disabled (temporarily).

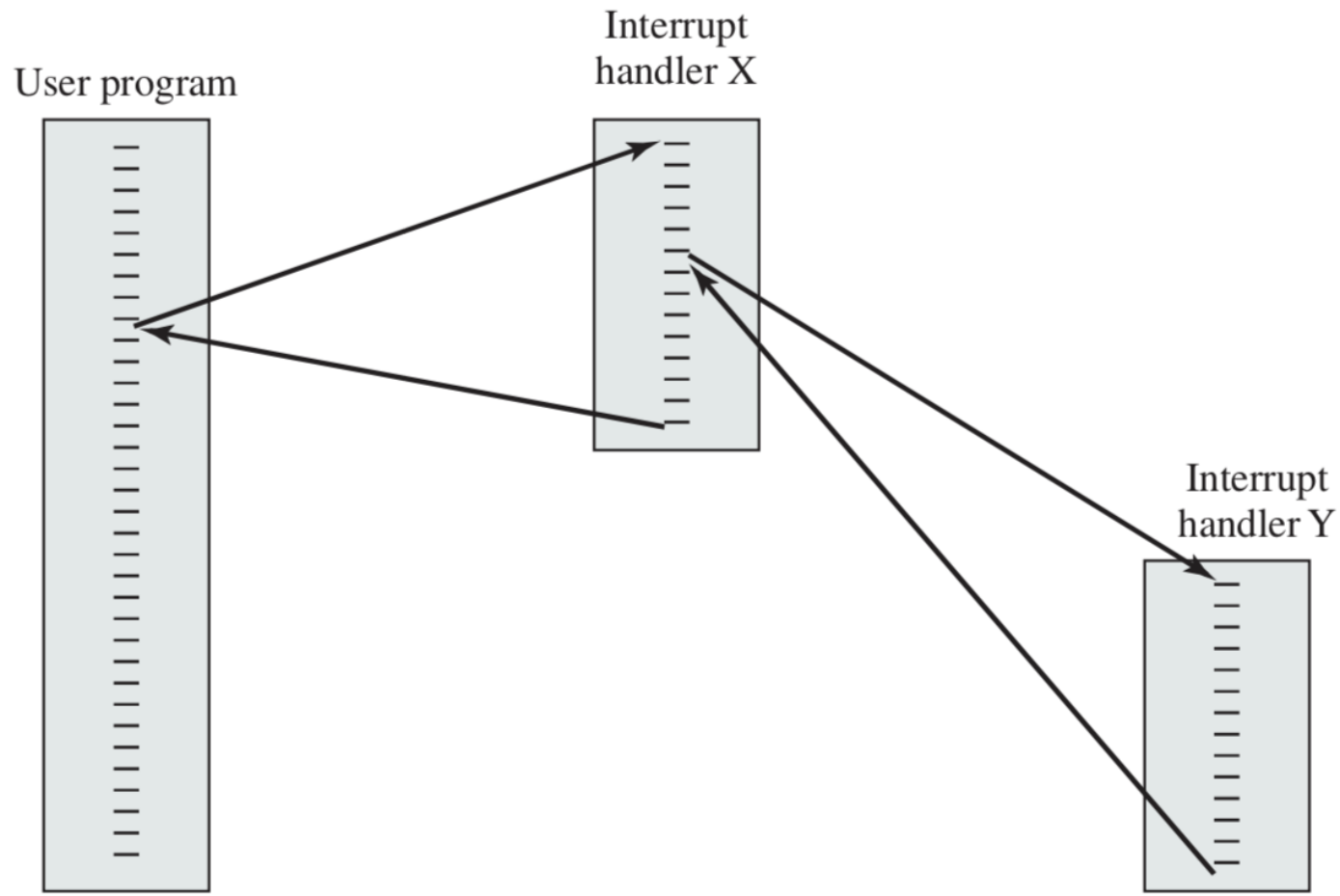
Interrupts can have different priorities.

We may also have multiple interrupts in a short period of time:



(a) Sequential interrupt processing

They may be sequential…



(b) Nested interrupt processing

Or nested. Or a combination.

Storing and Restoring State

The OS must store the program state when an interrupt occurs.

The state must be stored.

State: values of registers.

Push them onto the stack.

Interrupt finished: restore the state (pop off the stack).

Then execution continues.

That is saving and restoring the same program.

Why not restore a different program?

The Operating System might! It's not up to us.

Invoking System Calls

Some services run automatically, without user intervention.

In other cases, we want specifically to invoke them. How?

Admiral Ackbar Says:

Trap

Operating systems run on the basis of interrupts.

A trap is a software-generated interrupt.

Generated by an error (invalid instruction) or user program request.

Trap

If it is an error, the OS will decide what to do.

Usual strategy: give the error to the program.

The program can decide what to do if it can handle it.

Often times, the program doesn't handle it and just dies.

User Mode and Kernel Mode

Already we saw user mode vs. supervisor (kernel) mode instructions.

Supervisor mode allows all instructions and operations.

Even something seemingly simple like reading from disk or writing to console output requires privileged instructions.

These are common operations, but they involve the OS every time.

User Mode and Kernel Mode

Modern processors track what mode they are in with the mode bit.

At boot up, the computer starts up in kernel mode as the operating system is started and loaded.

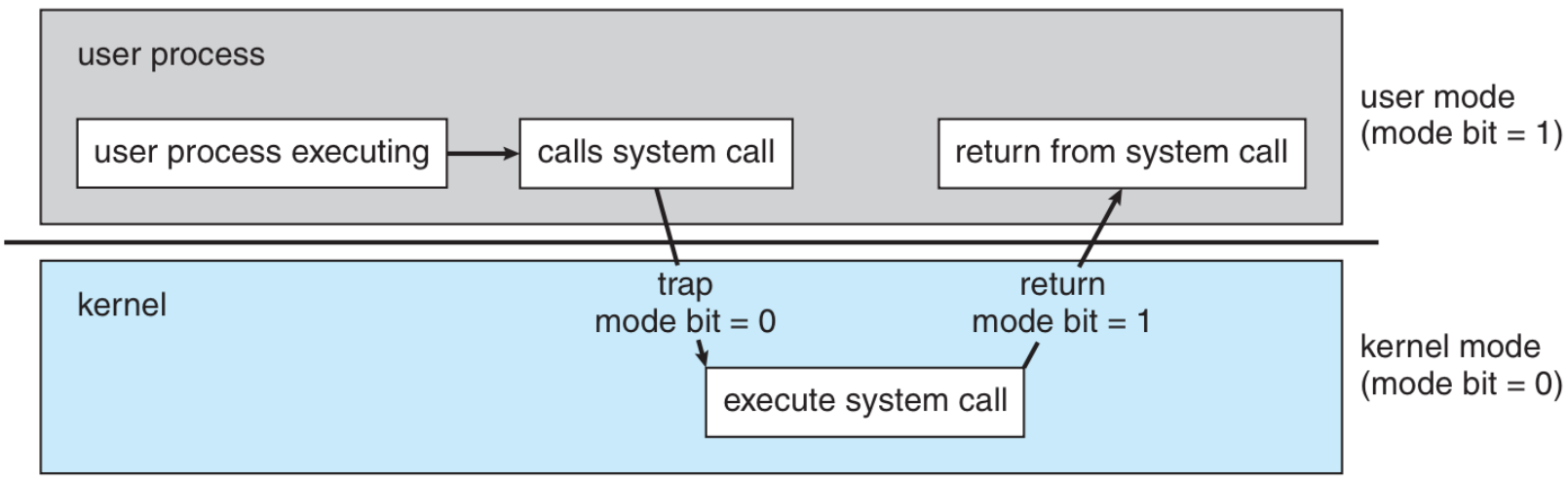
User programs are always started in user mode.

When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode.

When it is finished the system goes back to user mode before the user program resumes.

Example: Text Editor Printing

Suppose a text editor wants to output data to a printer.



User Mode and Kernel Mode: Motivation

Why do we have user and supervisor modes, anyway?

Uncle Ben to Spiderman: “with great power comes great responsibility”.

Same as why we have user accounts and administrator accounts.

To protect the system & its integrity against errant and malicious users.

User Mode and Kernel Mode: Motivation

Multiple programs might be trying to use the same I/O device at once.

Program 1 tries to read from disk. This takes some time.

If Program 2 wants to read from the same disk, the operating system forces Program 2 to wait its turn.

Without the OS, it would be up to the author(s) of Program 2 to check and wait patiently for it to become available.

Works if everyone plays nicely.

Without enforcement of the rules, a program will do something nasty.

User Mode and Kernel Mode: Motivation

There is a definite performance trade-off.

Switching from user to kernel mode takes time.

The performance hit is worth it for the security.

Example: Reading from Disk

C code to perform a read on a UNIX system.

`read` takes three parameters:

1. the file (a file descriptor, from a previous call to open);
2. where to read the data to; and
3. how many bytes to read.

```
int bytesRead = read( file, buffer, numBytes );
```

Note that `read` returns the number of bytes successfully read.

Example: Reading from Disk

In preparation for a call to `read` the parameters are pushed on the stack.
This is the normal way in which a procedure is called in C(++).

`read` is called; the normal instruction to enter another function.

The `read` function will put its identifier in a predefined location.

Then it executes the `trap` instruction, activating the OS.

Example: Reading from Disk

The OS takes over and control switches to kernel mode.

Control transfers to a predefined memory location within the kernel.

The trap handler examines the request: it checks the identifier.

Now it knows what system call request handler should execute: read.

That routine executes.

When it is finished, control will be returned to the `read` function.
Exit the kernel and return to user mode.

`read` finishes and returns, and control goes back to the user program.

System Call Complex Example

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void readfile( FILE* file );

int main( int argc, char** argv ) {
    if ( argc != 2 ) {
        printf("Usage: %s <filename>\n", argv[0]);
        return -1;
    }
    FILE* f = fopen( argv[1], "r");
    if ( f == NULL ) {
        printf("Unable to open file! %s is invalid name?\n", argv[1] );
        return -1;
    }
    readfile( f );
    fclose( f );
    return 0;
}
```

System Call Complex Example

```
void readfile( FILE* file ) {
    char** line = malloc( sizeof( char* ) );
    ssize_t* length = malloc( sizeof( ssize_t ) );

    while ( 1 ) {
        int read = getline(line, length, file);
        if (read == -1 ) {
            break;
        }
        printf("%s", *line);
    }
    printf("\nEnd of File.\n");
    free( length );
    free( line );
}
```

System Call Summary

The steps, arranged chronologically, when invoking a system call are:

1. The user program pushes arguments onto the stack.
2. The user program invokes the system call.
3. The system call puts its identifier in the designated location.
4. The system call issues the `trap` instruction.
5. The OS responds to the interrupt and examines the identifier in the designated location.
6. The OS runs the system call handler that matches the identifier.
7. When the handler is finished, control exits the kernel and goes back to the system call (in user mode).
8. The system call returns control to the user program.