

# Lecture 2 — Review of Computer Architecture

Jeff Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

December 14, 2022

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

What is a program? Instructions and data.



To execute a program we need:

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

Of course, this is the minimal set.

Ideally, memory would be:

- Fast enough that the processor never has to wait;
- Large enough to hold all the data;
- Inexpensive.

The **Iron Triangle**: “fast, good, cheap; pick two.”

Good news: we can have different levels of memory.

Let us compare the various levels I might have in my laptop from 2019:

Memory Level	Access Time	Total Capacity
Register	1 ns	< 1 KB
Cache	2 ns	16 MB
Main Memory (RAM)	10 ns	64 GB
Solid State Hard Disk	250 $\mu$ s	1000 GB
Backup Hard Disk Drive	10 ms	2 TB

I am the CPU and a particular book is the piece of data needed.

If the data is in the cache: the book is on a bookshelf in my office.

If the data for the CPU on a magnetic hard disk, I have to get the book from Library and Archives Canada in Ottawa (550 km away).

And I would have to walk.

The CPU doesn't go get the data; instead it must wait for it to arrive.

What might I do in the meantime...?

Every sort of communication using the same bus.  
Contention for this resource is a limiting factor.

The original IBM PC did work like that.  
A modern system has numerous buses.



# Central Processing Unit (CPU)



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

Fetch instructions, decode them, execute them.

Okay, there's a bit more to it than this.

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 → 32-bit word. 64-bit computer → 64-bit word.

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.

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.

A few of the registers in a typical CPU:

- **Program Counter**
- **Status Register**
- **Instruction Register**
- **Stack Pointer**
- **General Purpose Registers**

Program is a sequence of instructions. We can categorize them as:

- 1 Processor-Memory**
- 2 Processor-I/O**
- 3 Data Processing**
- 4 Control**

If I ordered a book from Ottawa, it takes a long time to arrive.

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.

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

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



Interrupts are a way to improve processor utilization.

CPU time is valuable!

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



Interrupts are a way to improve processor utilization.

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

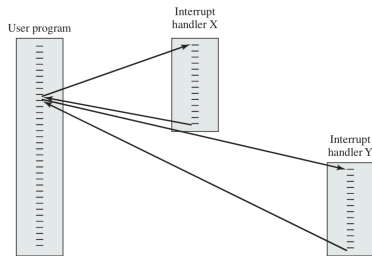
Analogy: professor in a lecture; student has a question.

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

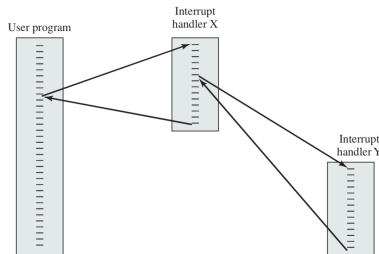
Sometimes the CPU is in the middle of something uninterruptible.  
Interrupts may be disabled (temporarily).

Interrupts can have different priorities.

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



(a) Sequential interrupt processing



(b) Nested interrupt processing

They may be sequential (left) or nested (right), or a combination.

# Storing and Restoring State

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



The state must be stored.

But what is state? Values of registers.

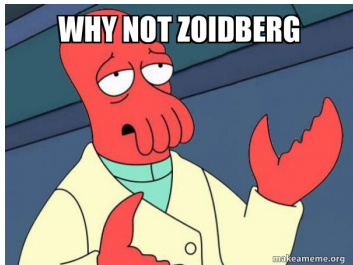
Where? Push them onto the stack.

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?



We will come to this in scheduling.

Three major strategies for communication:

- 1 Programmed I/O.**
- 2 Interrupt Driven I/O.**
- 3 Direct Memory Access (DMA).**



CPU does setup – data sent to DMA module.



Then the CPU can go on.

Delegation requires setting up:

- 1 The operation to perform (read or write)
- 2 The source
- 3 The destination
- 4 How much data is to be transferred

The I/O device will interact directly with memory.

Some services run automatically, without user intervention.

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



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.

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.

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.

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.

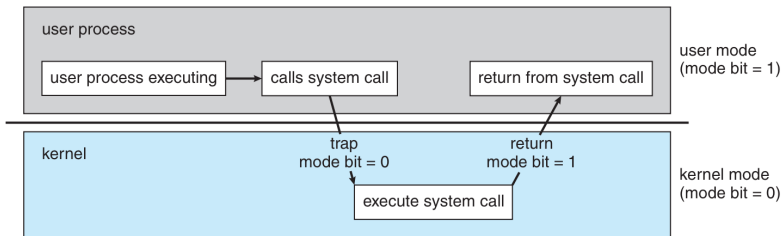
When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode.

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:



# User Mode and Kernel Mode: Motivation

Actually, though: “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.

# User Mode and Kernel Mode: Motivation



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.

Here's the function we use for reading data from a file:

---

```
ssize_t read( int file_descriptor, void *buffer, size_t count );
```

---

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.

Example:

---

```
int bytes_read = read( file, buffer, num_bytes );
```

---

Note that read returns the number of bytes successfully read.

# They Elected Me To Lead, Not To Read

This is a system call, and system calls have documentation.

Finding and reading this information is a key skill for systems programming.

Google (or other search engine of your choice) is your friend.

Good sources: [man7.org](http://man7.org), [linux.die.net](http://linux.die.net), or the website of the code library!



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.

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.

---

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

void readfile( int fd );

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

---

---

```
void readfile( int fd ) {
    int buf_size = 256;
    char* buffer = malloc( buf_size );
    while ( 1 ) {
        memset( buffer, 0, buf_size );
        int bytes_read = read( fd, buffer, buf_size - 1 );
        if ( bytes_read == 0 ) {
            break;
        }
        printf("%s", buffer);
    }
    printf("\nEnd_of_File.\n");
    free( buffer );
}
```

---

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.