SE 350: Operating Systems Winter 2023

Lecture 2 — Review of Computer Architecture

Jeff Zarnett *2022-11-08*

Computer Organization

Although a regular program like a word processor need not be concerned with the underlying hardware of the computer, this abstraction does not come for free: the operating system must be aware of these details and manage them for everyone. What is a program, anyway? You will know from your programming experience that a program is just a sequence of instructions and some data. Thus, to execute a program, we will need a few things:

1. **Main Memory** – a place where the instructions and data are stored;
2. **System Bus** – a way for instructions and data to travel between memory and the processor; and
3. **Processor** – that critical element of the system that actually executes the instructions.

Typically we have a fourth element: **Input/Output** (I/O). While not strictly necessary to run a program, a computer with no input or output devices is of, at best, limited usefulness.

Main Memory

Ideally, memory would be fast enough that the processor never has to wait for memory, large enough to hold all the data of the system, and inexpensive. Readers familiar with reality will recognize this as the *Iron Triangle*: "fast, good, cheap; pick two." The good news is that we do not have to make only one choice. We can have different levels of memory at different sizes, speed, and cost. You are presumably already familiar with this because a typical laptop has both RAM (Random Access Memory) and a hard drive (or solid state drive) and they are not only different sizes, but follow independent pricing schemes. Moving from 16 GB of RAM to 32 GB of RAM might cost the same as a change from a 2 TB hard drive to a 4 TB hard drive. So what we end up with is a hierarchy 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 μs	1000 GB
Backup Hard Disk Drive	10 ms	2 TB

Notably, the cache (pronounced like "cash") is very often broken down into different levels like L1 (Level 1), L2, L3. In any case, the trend is clear: fast memory is expensive and as we get further away from the CPU memory access gets slower, but it gets less expensive so we can have more of it.

The difference in access time is often quite dramatic. An analogy to put it in perspective: imagine I am the CPU and a particular book is the piece of data needed. If the data is in the cache, it is as if the book is on a bookshelf in my office. Thus, I can retrieve the book very quickly. If the data for the CPU on a magnetic hard disk, it is as if I would have to get the book from Library and Archives Canada in Ottawa (about 550 km away, according to Google Maps). And I would have to walk. Of course, the analogy is slightly tortured because the CPU doesn't go get the data; instead it must wait for it to arrive. If I ordered a book from Library and Archives Canada and someone had to walk it over, I would spend a lot of time waiting. What might I do in the meantime...?

Management of memory and caching will be a major topic to be examined later on.

System Bus(es)

As you can imagine, with every sort of communication using the same bus, contention for this resource is a major limiting factor in the operation of the computer. It turns out, the diagram shown earlier with the single bus is a simplification (though the original IBM PC did work like that). A modern system has numerous buses that work at different speeds and towards different functions. The study of buses and their functions and problems are not something we will focus on, but it deserves mention as a key part of the hardware.

The Processor

The Processor (or *CPU*, central processing unit) is the brain of the computer. It fetches instructions from memory, via the bus, decodes the instruction, then executes it. This fetch-decode-execute cycle will be repeated until the program finishes (... if the program finishes). The different steps may be executed in parallel: while one instruction is being executed, the next is being decoded, and the one after that is being fetched. We call this a *pipeline*, and the length and complexity of the pipeline is something we will not examine here, but is an interesting subject all its own. A processor's largest unit on which it operates is called a *word*. A 32-bit processor has a word size of 32 bits and a 64-bit processor has a word size of 64 bits.

CPU instructions are, obviously, specific to the processor. If you have any experience in writing assembly code, you have probably used some books that tell you all the instructions that can be issued and what operands, if any, those instructions take. In some CPU architectures, some operations are available only in "supervisor mode" and not in "user mode". An instruction that disables interrupts is an example of an instruction that would be available only in supervisor mode. Attempting to run it in user mode will be an error.

In addition to the hardware to decode and execute the different instructions, the CPU has some storage locations called *registers*. They may store data or instructions (they are both, after all, just a bunch of bits). Registers are a key concept in CPUs and management of those registers is partly the responsibility of the operating system. Registers are used to hold key variables and temporary results. Registers are often, but not always, word-sized. Let us examine some of the critical registers in processors.

Program Counter. A program is a sequence of instructions, and as you can imagine, to execute one correctly, we need to keep track of what instruction is next. After the instruction at that address is fetched, the program counter is incremented. That points it to the next instruction, or, at least, a guess at the next instruction. As you know, programs often have conditional instructions and loops, so the the program counter may be updated to hold a different address than simply the "next".

Status Register. The status register (sometimes called "program status word") is used as an array of bits to indicate various flags or properties, indicating the state of the processor. We can divide flags into two categories: arithmetic and non-arithmetic. The arithmetic flags are used to indicate mathematical outcomes, such as an overflow (e.g., the result of an operation was too large to fit in a register), or division by zero error. The non-arithmetic flags may be used to note that the CPU is running in supervisor mode rather than user mode, or that an assembly instruction was invalid. Complex CPUs may have more than one status register.

Instruction Register. The instruction most recently fetched will then be stored in the instruction register. Nothing complex or exciting here.

Stack Pointer. The CPU may have a specific register to indicate the location in memory that is at the top of the stack. You will remember that memory in an executing program is divided, logically, into two separate categories: the stack and the heap. It is convenient to maintain a handy reference to the top of the stack in memory.

General Purpose Registers. General purpose registers may store data as well as addresses. A typical arithmetic instruction like addition requires data to be loaded into these general purpose registers and stores the result in those general purpose registers. Usually the compiler determines how they are used, but in C you can exercise some measure of control using the `register` and `volatile` keywords.

This is not an exhaustive list, however. There may be others in a system like the memory address register, memory buffer register, I/O address register...

Program Execution

As mentioned earlier, a program to execute is just a set of instructions, telling the CPU what to do. The processor retrieves the next instruction from the instruction register and will decode it (and thus figure out what to do). According to [Sta14] we can categorize these actions into one of the following categories:

1. **Processor-Memory:** Transfer data from a processor to memory or vice versa (e.g, read an int from memory).
2. **Processor-I/O:** Transfer data to or from an I/O device (e.g., make the speaker beep).
3. **Data Processing:** Perform some arithmetic or logical operation on data (e.g., add two numbers).
4. **Control:** Alter the sequence of execution (e.g., go back to the start of the loop).

Interrupts

As discussed earlier, if I have ordered a book from Library and Archives Canada it will take quite a long time for someone to walk the book to my office in Waterloo, and I would spend a lot of time waiting. So in the meantime, it seems logical that I should do something else. There are two options for dealing with this situation. One is polling: that I can check periodically if the book has arrived. This approach is fairly wasteful (how often do I check? How much time will I spend checking?). The other solution is that I can receive a notification when the poor fellow who has walked the 550 km to my office has arrived. If he knocks on my office door, it will cause me to suspend what I am doing and collect the book. Or, to put it in one word: interrupts.

In that analogy I played the part of the CPU, and was interrupted and had to deal with that interruption. We have the same in computer systems. When an interrupt happens, the normal sequencing of the processor does not continue. Interrupts are, according to [Sta14], also something we can categorize into four buckets, based on where they are generated from:

1. **Program:** Something happens in the program (e.g., a division by zero).
2. **Timer:** A configured timer within the processor's time expires (e.g., to update the system clock).
3. **Input/Output:** An I/O controller signals (un)successful completion of an operation (e.g., read from disk).
4. **Hardware Failure:** Something happens in hardware (e.g., power failure).

Interrupts are primarily a way to improve processor utilization (i.e., the fraction of the time the CPU is doing useful work). CPU time is valuable and the CPU can and should do something else while it is awaiting the results of some other operation. Thus, the CPU might issue a read from memory instruction, work on something else, and then be interrupted when the data has arrived from memory.

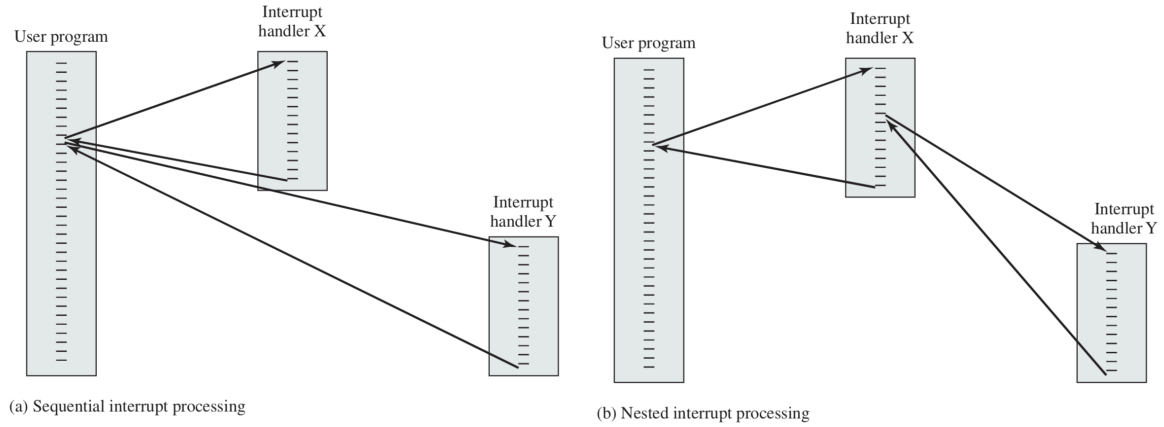
When an interrupt takes place, the processor might choose to ignore it, but this is rare. Interruptions usually contain important information. Almost always, we will want to *handle* (deal with) the interrupt in some way.

Consider an analogy with which you are surely familiar: a professor is at the front of the lecture hall droning on about some subject. A student raises her hand to ask a question. The professor can ignore this, but that is generally a bad idea. The professor should pause what he is doing (pause execution), remembering where he was (save state), take the question and answer it (handle the interrupt), and then resume from where he left off (restore the saved state and continue execution).

The operating system is responsible for storing the state of the program being interrupted, handling the interrupt, and then restoring the state of the program that was interrupted.

Sometimes the CPU is in the middle of something where an interruption would be bad. Thus, interrupts can be disabled. This is like the professor saying that all questions should be saved until the end of the lecture. Once interrupts are enabled again, the interruptions can be handled. So all the questions that students have saved can be answered. Interrupts tend to have a priority, so if multiple interrupts are pending, the highest priority one will be dealt with first.

There can also be multiple interrupts happening in a short period of time: suppose interrupt 1 occurs and then interrupt 2 occurs while interrupt 1 is still being handled. Often, but not always, interrupt handlers themselves disable interrupts so a subsequent interrupt will be handled after the current one is dealt with. Or, interrupt 2 may itself interrupt the first. Consider the diagrams below:



Left: Sequential interrupt handling. Right: Nested interrupt handling. [Sta14]

A combination of these different techniques may be used. A low priority interrupt handler may be suspended to deal with a high priority interrupt. Suppose while the high priority interrupt is being handled, a medium priority interrupt occurs. The high priority interrupt handler will be allowed to complete its work before the medium priority interrupt is dealt with.

Storing and Restoring State. We mentioned already that the operating system is responsible for storing the state of the program (or interrupt handler) being interrupted when an interrupt occurs. To do so, the state of the CPU is what must be stored. The values of the various registers (e.g., program counter, instruction register, etc.) are stored by pushing them onto the stack. When the interrupt is finished, those values are popped off the stack and loaded into the registers again. After that, program execution continues (almost) as if the interruption never took place.

Multiprogramming. Thus far we have assumed that there is a program running and that same program is the one that resumes at the end of the interrupt handling. This is, however, not always the case. Given that we have stored the state of the interrupted program, we can resume it any time we like, and run a different program now instead. But which program should we run next? We will examine the subject when we talk about scheduling.

I/O Communication

That fourth major element of the modern computer system, I/O, is something we have thus far not examined. Now we will consider three strategies for I/O operations, each of which we can examine briefly. In the examples we will assume the operation to be completed is a read, but writing is the same, just in the opposite direction.

Programmed I/O. This strategy is really just the name for polling. The processor issues an I/O command and is responsible for checking when the I/O operation is complete. This usually requires checking the I/O device's status register or memory location. The I/O device updates this when it has finished the operation. Polling is, however, an inefficient way to get things done. The CPU is either waiting around doing nothing or checking the status register. Once the status has been checked and indicates finished, the CPU can copy the data into memory.

Interrupt-Driven I/O. Interrupts are a much more efficient way to get things done. The CPU issues the read or write and eventually, when the operation is completed, the I/O module issues an interrupt to indicate it is finished. At that point the CPU will copy the data into memory.

In both the interrupt and programmed I/O strategy, the CPU is involved twice: it initiates the read (or write) and then collects the result and does something with it. The read has three parties: the device, the CPU, and memory. What if we could eliminate the middleman (the CPU) from this equation?

Direct Memory Access (DMA). The most efficient way to handle a large block of data is with DMA. The advantage to DMA is that the CPU is involved in setting up the operation but does not have to manage things. It introduces the I/O device and memory to one another and says "you figure it out". The CPU will do some set up, indicating:

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

This data is sent to the DMA module (a delegate). After that, the CPU can go on to do other work and the I/O device will interact directly with memory. Hence, the name, Direct Memory Access. Note that the CPU may have to contend with the DMA operation for the bus, but even if this is the case, it is more efficient to use DMA than either interrupt or programmed I/O [Sta14].

It's a Trap!

Operating systems run, as previously discussed, on interrupts. In addition to the interrupts that will be generated by hardware and devices (e.g., a keyboard signalling that the F1 key has been pressed), there are also interrupts generated in software. These are often referred to as a *trap* (or, sometimes, an exception). The trap is usually generated either by an error like an invalid instruction or from a user program request.

If it is simply an error the operating system will decide how to deal with it, and in desktop/laptop OSes, the usual strategy is sending the exception to the program that caused it, and this is usually fatal to the offending program. Your programming experience will tell you that you can sometimes deal with an exception (perhaps through the language equivalent of the Java/C# `try-catch-finally` syntax), but often an exception is unhandled and terminates the program.

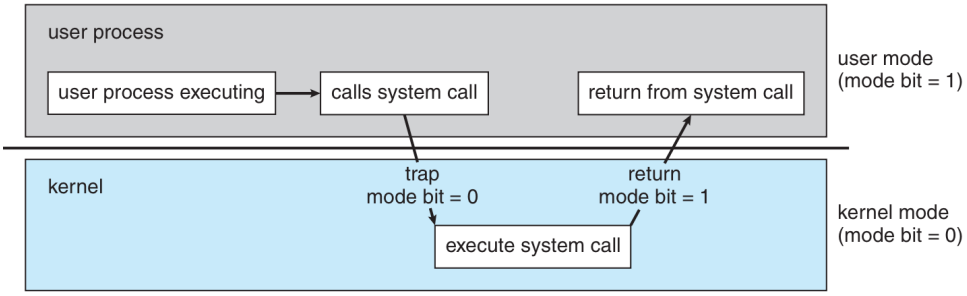
The more interesting case is the intentional use of the trap: this is how a user program gets the operating system's attention. When a user program is running, the operating system is not; we might even say it is "sleeping". If the

program running needs the operating system to do something, it needs to wake up the OS: interrupt its sleep. When the trap occurs, the interrupt handler (part of the OS) is going to run to deal with the request.

Already we saw the concept of user mode vs. supervisor mode instructions: some instructions are not available in user mode. Supervisor mode, also called kernel 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 operating system every time.

Modern processors keep track of what mode they are in with the mode bit. This was not the case for some older processors and some current processors have more than two modes, but we will restrict ourselves to dual-mode operation with a mode bit. Thus we can see at a glance which mode the system is in. 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 [SGG13].

Suppose a text editor wants to output data to a printer. Management of I/O devices like printers is the job of the OS, so to send the data, the text editor must ask the OS to step in, as in the diagram below:



Transition from user to supervisor (kernel) mode [SGG13].

So to print out the data, the program will prepare the data for printing. Then it calls the system call. You may think of this as being just like a normal function call, except it involves the operating system. This triggers the operating system (with a trap). The operating system responds and executes the system call and dispatches that data to the printer. When this job is done, operation goes back to user mode and the program returns from the system call.

Motivation for Dual Mode Operation

Why do we have user and supervisor modes, anyway? As Uncle Ben told Spiderman, "with great power comes great responsibility". Many of the reasons are the same as why we have user accounts and administrator accounts: we want to protect the system and its integrity against errant and malicious users.

An example: multiple programs might be trying to use the same I/O device at once. If Program 1 tries to read from disk, it will take time for that request to be serviced. During that time, if Program 2 wants to read from the same disk, the operating system will force Program 2 to wait its turn. Without the OS to enforce this, it would be up to the author(s) of Program 2 to check if the disk is currently in use and to wait patiently for it to become available. That may work if everybody plays nicely, but without someone to enforce the rules, sooner or later there will be a program that does something nasty, like cancel another program's read request and perform its read first.

This doesn't come for free, of course: there is a definite performance trade-off. Switching from user mode to kernel mode requires some instructions and some time. It would be faster if everything ran in kernel mode because we would spend no time switching. Despite this, the performance hit for the mode switch is judged worthwhile for the security and integrity benefits it provides.

Example: Reading from Disk

Let us examine in some more detail what is actually happening in a system call. This example is from [Tan08] and will use C code to perform a read on a UNIX system. The definition of the function we want to use is:

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

The specification says the function `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. Here is an example:

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

(The `read` function returns the number of bytes successfully read; it is normally equal to `num_bytes` but might be smaller if the end of the file is reached. Here we are storing it in a variable `bytes_read`.)

This is a system call, and system calls have documentation. Finding and reading this information is a key skill for systems programming. If you want to know what the parameters to a function are, what the return value means, any assumptions made in the implementation – read these. Google (or other search engine of your choice) is your friend: usually searching for the name of the function you want to use (along with C, as in, the programming language) gives you everything you need. Good sources include: man7.org, linux.die.net, or the website of the code library.

In preparation for invocation of `read` the parameters are pushed on the stack. This is the normal way in which a procedure is called in C++. Then the `read` procedure is called and this is just the normal instruction to enter another function. The `read` function will put its identifier (the system call number) in a predefined location (typically a register). Then it executes the `trap` instruction, activating the OS.

When the `trap` occurs, the OS takes over and control switches from user mode to kernel mode. Control transfers to a predefined memory location within the kernel (the trap handler). The trap handler then runs and examines the request: it checks the identifier that was put in the register earlier. Based on that, it knows what system call request handler should execute: the one to read from a file. That routine executes. When it is finished, control will be returned to the `read` function; we exit the kernel and return to user mode.

Back in user mode, the `read` call finishes and returns, and control goes back to the user program.

Let's consider a more complex example. This uses some Linux specific items, but is a quick example of a program that reads a file and prints it out:

```
#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 );
}
```

Summary: Invoking a System Call

To summarize, 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.

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.