CS 24: Introduction to Computing Systems

Execeptional Control Flow

The next part of the course is all about understanding how the system transfers control between different programs. We'll learn that the *kernel* of the operating system is responsible for overseeing this process.

Motivation

It can be a bit hard to see why these concepts are important at first, but we believe the following four motivating questions (all of which we will answer using the new concepts we'll learn this week) do a good job of explaining why this stuff is important.

- Your computer has a lot of programs all running at once, but there are only a few CPUs. How is this possible?
- When you access memory that your program hasn't been cleared to use, something causes a Segmentation Fault. What is going on there?
- Sometimes when a program is misbehaving, you kill it using control panel or `kill`. How does that work?
- How do `fopen` and `fread` actually access files?

So far, we've seen control flow *within* a program (e.g., jumps, call/return), but what about switching *between* running programs? What about responding to the user, network, or disk? We accomplish this using *exceptional* control flow (ECF). There are several types of ECF that run from very low-level to very high-level.

Exceptions

An *exception* is a change in control flow in response to a system event. We consider all of the below types of ECF to be exceptions, but there are specific types which are worth getting to know individually.

- **Interrupts.** An *interrupt* occurs when an event external to the processor needs to be responded to. For example, if the disk retrieves data for a program, it alerts the processor that the data has arrived via an interrupt. One of the most important interrupts, as we will see later, is the *hardware timer interrupt* which triggers every few milliseconds to give the kernel control of the processor.
- **Faults.** A *fault* occurs when an unintentional interruption occurs internal to the processor. For example, if the `idiv` instruction tries to divide by zero, the processor needs to alert someone (who?) that an error has occurred. You've likely seen the message `Floating point exception (core dumped)` over the course of the projects so far–that's exactly what's going on there! Some faults are recoverable (e.g., the program tries to read data that is not currently loaded), and some are unrecoverable (e.g., divide-by-zero).
- **Traps.** A *trap* occurs when the program intentionally interrupts execution to ask the kernel to do something. The most important example of a trap is when a user process invokes a system call to ask the kernel to provide functionality or a resource that the user process cannot execute directly. This is how numerous functions you probably assumed were "magic" work: interacting with the file system, starting/killing programs, etc.

A commonality across all types of exceptions is that the user program is not responsible for handling them and they should get resolved opaquely without the program even knowing they're happening. Someone's code has to handle all of these types of exceptions, and, because they often have to access system resources to be resolved, it needs to be privileged. This privileged code is a part of the operating system called the *kernel*.

Multiprocessing

Another example of ECF, which is built on top of exceptions, is when one program suspends execution and another begins running. *Multiprocessing* is necessary, because we expect all the running programs on our computer to make progress and respond to the user; so, we can't just run them serially. The kernel is responsible for providing an implementation of this idea. The actual switching between running programs is called *context switching* and involves saving the registers for the currently executing program and switching to the new program's address space.

Signals

When an exception or other important system event occurs, the kernel notifies the receiving process via a small message called a *signal*. Programs usually do one of the following three things in response to signals.

- **Ignore.** The process can just ignore the signal and keep executing.
- **Terminate.** The process can immediately terminate the program.
- **Catch.** The process can "catch" the signal and perform a custom action as a result.

All signals have a default response (e.g., `SIGSEGV` (segmentation faults) generally exit the program with an error code).

Answering the Motivating Questions

Now that we have a sense of the available types of control flow, let's answer the questions we started with.

- Your computer has a lot of programs all running at once, but there are only a few CPUs. How is this possible?

  > Periodically, the timer sends an interrupt to each processor, causing them to go from userspace into the kernel. The kernel then chooses a new process to run, loads its state, and goes back into userspace (now executing the new process).

- When you access memory that your program hasn't been cleared to use, something causes a Segmentation Fault. What is going on there?

  > The CPU's page mapping hardware will fail to find the address and deliver a page fault. This causes the CPU to transfer control to the kernel, which responds with its page fault handler. On Linux, the kernel responds to this by delivering a `SIGSEGV` to the process. By default, a `SIGSEGV` causes the process to terminate. The kernel reports the specific signal causing the process to exit to its parent (e.g. `bash`), which might respond by printing "segmentation fault"

- Sometimes when a program is misbehaving, you kill it using control panel or `kill`. How does that work?

  > `kill` and similar programs can send `SIGTERM` to a process which generally terminates the program.

- How do `fopen` and `fread` actually access files?

  > `fopen` and `fread` call the syscalls `open(2)` and `read(2)`, respectively. The kernel handles open by talking to the filesystem to find the specified file and creating a "file descriptor" that refers to its record of the open file. When the kernel gets a "read" syscall, it looks up the specified file descriptor and goes back to the specific filesystem code to actually retrieve the desired number of bytes of the file from disk. The disk sends an interrupt when it is done retrieving the data from disk.