About Interrupts

CS 220

So, what is an interrupt, anyways, and why should you care? Interrupts are a type of exception. You've seen exceptions before; consider this bit of Java code:

```
Scanner scanner;
try {
   scanner = new Scanner("Secret Sauce Recipe");
}
catch (FileNotFoundException e) {
   System.out.println("Emergency: Can't find the Secret Sauce Recipe!!!");
   alertTheAuthorities(e);
   System.exit(1);
}
```

`Scanner`'s constructor takes a file name as a parameter. It's possible that the file can't be found when the constructor runs. If this happens, the constructor generates a `FileNotFoundException` and the code in the `catch` block will be run. Ordinarily, that `catch` block doesn't run. But, when the exception occurs, the flow of instruction execution is changed to the `catch` block. In fact, instruction flow "jumps" to the `catch` block.

Computer architects generally use the term "interrupt" to refer to exceptions that are generated externally to the CPU, namely I/O events. Data arriving over a network connection, for example, would cause an interrupt to be generated by the network adapter. When the CPU receives the interrupt, it transfers the flow of instruction execution (jumps) to an operating system routine that "services" the interrupt by moving the data from the network adapter to whatever thread will be using the data. When the routine that handled the interrupt finishes, execution returns to the original code (jumps back), at the point at which it was interrupted. Jumping to an interrupt service routine can be thought of as making an unplanned method call in order to handle something unexpected.

Why should you care about interrupts? In a world without interrupts, today's computer systems wouldn't be possible. Today's systems run hundreds of threads "simultaneously" and keep the CPU busy even though all the I/O devices surrounding it are super-slow in comparison. If a thread performed an I/O operation, for example reading keystrokes from a keyboard, the entire system would appear to "freeze" until those keystrokes were typed. Without interrupts, threads would have to be run one at a time, with the next thread not starting until the previous thread had finished. You wouldn't be able to watch a video *and* receive text messages on your phone.

With interrupts, the CPU can start an operation for a thread on a slow I/O device and then continue executing other threads — no freezes. When the I/O operation completes, the I/O device interrupts the CPU, which then jumps to an operating system routine to complete the I/O operation. With interrupts, the operating system can set a timer to expire in a fraction of a second and then turn over control of the CPU to a user thread. When the timer expires, the CPU runs an operating system routine which resets the timer and then transfer execution control to a different thread. Using this mechanism, the CPU appears to be running hundreds of threads simultaneously.

In Lab 5, you'll be writing interrupt service routines to handle events from a pushbutton switch (push and release events) and use a timer interrupt to provide the illusion of running multiple threads simultaneously.