CS110 Spring 2021
Pre-Lecture 5:
Control Flow, Exceptions, & Context Switches

Principles of Computer Systems
Stanford University, Dept. Of Computer Science
Lecturer: Roz Cyrus
Content adapted from material by Jerry Cain
Reading Material

Bryant & O'Hallaron: Sections 1 - 4 of: Chapter 1 (reader) or 8 (full textbook)
Processes Recap

- **A process** is an instance of a program in execution, versus a **program**, which is the code and data. Processes are active, whereas programs are passive.

- A program always runs in the **context** of some process. This context has state that the program needs in order to properly run.

- Processes provide two key abstractions to an application:
  - 1. A private address space that provides **the illusion that our program has exclusive use of memory** (discussed in the last lecture)
  - 2. An independent logical control flow that provides **the illusion that our program has exclusive use of the processor** (discussed in lecture 5)
Introducing Control Flow

- The processor’s program counter (which contains the next address to be executed) assumes a sequence of addresses that each correspond to an instruction.

- Each transition from one instruction to another is a control transfer, and a sequence of such control transfers is called the flow of control (control flow) of the processor.
  - A “smooth” control flow runs contiguous (adjacent in memory) instructions. Abrupt changes in this flow could occur from something program-related, such as jumps, function calls and returns.
  - Other abrupt changes to the system may not have to do with the running program at all. Examples:
    - A hardware timer going off
    - Network packets arrive and must be stored in memory
    - Another program is ready to receive data
    - Parent processes (explained later) are notified when their children processes terminate
  - These kinds of abrupt changes are called exceptional control flow (keyword: “exception”)
Control Flow (continued)

- Exceptional control flow occurs at all levels of a computer system. Examples:
  - **Hardware level**: events detected by hardware trigger abrupt control transfers to exception handlers.
  - **OS level**: the kernel transfers control from one user process to another via context switches (explained in a later slide).
  - **Application level**: a process can send a signal to another process, which abruptly transfers control to a signal handler in the recipient.

    A signal is a small message that notifies a process that an event of some type occurred. Signals are often sent by the kernel, but they can be sent from other processes as well. Signals and signal handlers will be explained in greater detail in the next lecture.
 Exceptions

- An **exception** is an abrupt change in the control flow in response to some change in the processor’s state.

  A change in the processor’s state (an **event**) triggers an abrupt control transfer (an exception) from the program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program (to either the same instruction or the next one) or aborts.
Exceptions (continued)

- Each type of possible exception in a system is assigned a unique nonnegative integer exception number. x86-64 systems: up to 256 exception types, numbered 0 - 255.
- Some numbers are assigned by designers of the processor. Examples:
  - Divide by zero (exception number 0)
  - Arithmetic overflows
  - Page faults (exception number 14)
  - Memory access violations
- Other numbers are assigned by the designers of the kernel. (e.g. system calls)
- An exception table stores the address of the handler code for each exception. It is initialized at boot time. Exception control handlers run in kernel mode, meaning they have complete access to all system resources. Once hardware triggers the exception, the handler runs in software and then optionally returns to the interrupted program.
## Exceptions (continued)

### Types of exceptions

<table>
<thead>
<tr>
<th>Class</th>
<th>Cause</th>
<th>Async/sync</th>
<th>Return behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Signal from I/O device that is external to the processor</td>
<td>Async</td>
<td><strong>Always</strong> returns to <strong>next</strong> instruction</td>
</tr>
<tr>
<td>Trap</td>
<td><strong>Intentional</strong> exception as a result of executing an instruction. <strong>Includes system calls</strong></td>
<td>Sync</td>
<td><strong>Always</strong> returns to <strong>next</strong> instruction</td>
</tr>
<tr>
<td>Fault</td>
<td>Potentially recoverable error (e.g. a page fault)</td>
<td>Sync</td>
<td><strong>Might</strong> return to <strong>current</strong> instruction</td>
</tr>
<tr>
<td>Abort</td>
<td>Nonrecoverable error</td>
<td>Sync</td>
<td><strong>Never</strong> returns</td>
</tr>
</tbody>
</table>
More About System Calls

- The most important use of traps is to provide a procedure-like interface between user programs and the kernel, known as a **system call** (explained in the previous lecture). [View a list of Linux system calls]
  - Regular functions run in **user mode**, which restricts the types of instructions they can execute.
  - A system call runs in **kernel mode**, which allows it to execute privileged instructions and access a stack defined in the kernel.
- User programs often need to request services from the kernel such as handing a file (**read**, **write**, **open**, etc.) and creating a new process (**fork**).
- To allow controlled access to such kernel services, processors provide a special **syscall n** instruction that user programs can execute when they want to request service **n**.

![Diagram of system call process]

1. Application makes a system call
2. Control passes to handler
3. Trap handler runs
4. Handler returns to instruction following the syscall
Recap: “Exclusive” Memory Use

System Calls Summary (continued)

- A system call uses an interrupt to transfer control to the OS kernel. The user can only call a set of well-defined system calls, and there is little room for a security breach.
- Once the kernel is running a system call, it is in complete control of the system, and can access the necessary resources to fulfill the system call's needs.
- After a system call, the kernel returns control to the user program.
Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar in the right image represents a portion of the logical control flow for a process. Note how the execution of the three logical flows is interleaved. Each process takes turns using the processor; each runs for a while then is temporarily suspended while others get their turn.

Two flows run concurrently if their execution overlap in time. Multiple flows executing concurrently is concurrency. This is independent of # of cores. Which processes are concurrent?

This notion of a process taking turns with other processes is also called multitasking.

There may be tens, hundreds, or thousands of processes "running" at once, but on a single-core system, only one can run at a time, and this is coordinated by the OS. On multi-core machines (like most modern computers), multiple programs can literally run at the same time, one per core. Parallel flows are a subset of concurrent flows: flows running concurrently on different processor cores or computers. This is true multiprocessing.

Each time period that a process executes a portion of its flow is called a time slice (roughly 20 milliseconds).
The operating system kernel implements multitasking using a higher-level form of exceptional control flow known as a **context switch**. The decision to preempt the current process and restart a previously preempted process is called **scheduling**.

A context switch (1) Saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.

Recall that the kernel maintains a **context** for each process, which is state that the kernel needs in order to restart a preempted process. The context is stored in a **process control block** (one per process) which stores a lot of process-related information, including:

- Contents of general-purpose and floating-point registers
- Program counter
- User and kernel stack
- Status registers
- Code and in-memory data
- A file descriptor table
Context Switches

In the below example, the kernel runs some code for process A in user mode. It then reaches the `read` syscall, which requests data from disk to be loaded into memory, but since this will take a while, the kernel context switches from A to B: the kernel first does work on A’s behalf in kernel mode, then stays in kernel mode and does some work for B. The B’s user code is run in user mode until the disk interrupts and says, “hey! I have data ready for process A!” Shortly after, the kernel will get tired of running B and will context switch to A (first wrapping up B in kernel mode, then some kernel mode work for A). It will then proceed with A’s code in user mode, picking up right after the `read` call. A will continue happily along until the next exception occurs.
Lecture 5 will dive into multiprocessing. Yay!