15213 Lecture 14: Exceptions and Processes

Learning Objectives

- Recognize the difference between aborts and function failures.
- Identify program actions that can cause a synchronous exception.
- Describe what can happen during a trap, both immediately and ultimately.
- Explain how an application invokes helper functions in the kernel.
- Compare and contrast the OS's two tools for running multiple processes.

Getting Started

The directions for today's activity are on this sheet, but refer to accompanying programs that you'll need to download. To get set up, run these commands on a shark machine:

1. `$ wget http://www.cs.cmu.edu/~213/activities/lec14.tar`
2. `$ tar xf lec14.tar`
3. `$ cd lec14`

1 Exceptions

In this section, we'll be using an interactive calculator application as an example. Examine the `calc.c` file using your editor of choice or the `cat` command. Once you're comfortable with it, try building (`$ make calc`), running (`$ ./calc`), and using (e.g., `> 1+2`) it.

Programs appear to exhibit uninterrupted control flow through their local instruction stream. In reality, however, the hardware often interrupts them in response to special events and transfers control to the operating system; this is known as an **exception**. The kernel may respond by (eventually) transferring control back to the application[1], or it may decide to terminate the application, which is known as an **abort**. Since this case is easier to observe, let's look at some examples.

1.1 Asychronous Exceptions

Some exceptions arise due to an event that occurs *while* the program is executing, and cause the program to be paused at an arbitrary point in its execution while the kernel handles the event.

1. From within the `calc` application, try typing a word (e.g., `quit`). What does the program do after you hit enter? Did your input cause an abort?

2. This time, try pressing Ctrl-D to prematurely close the standard input stream. How does `calc` respond? Did your input cause an abort? How can you tell?

[1] Although it isn't always obvious that this has happened, we'll see later in the activity that an observant application can infer when it might have.

3. Now try pressing Ctrl-C. How does `calc` respond? Did your input cause an abort? How can you tell?

1.2 Synchronous Exceptions

Other exceptions arise occur in response to an action the program itself performs, and cause the OS to handle the event immediately at that particular point in the program. You've certainly already encountered the dreaded segmentation fault, but now we'll see another synchronous exception that reslts in a program abort.

4. This calculator accepts as valid input some math expressions that have an undefined mathematical result. Give one example and state what happens when you enter it.

5. The C standard permits implementations to abort when signed integer arithmetic wraps. Try testing addition and subtraction to see whether the shark machines abort in such a case. What do you find? *(Hint: Recall that 32-bit* TMax *is* $2,147,483,647$ *and* TMin *is* $-2,147,483,648$*.)*

6. *(advanced)* Now test multiplication and division. What do you find?

As you've discovered, x86-64 does not generate exceptions in the same integer arithmetic cases as some other architectures. In the cases where it does, it may surprise you to see an integer calculator printing `Floating point exception`. This is for historical reasons: Bell Labs initially developed Unix for the PDP-11, a computer that didn't even generate exceptions for undefined integer arithmetic results. As Unix was standardized across other platforms, the new abort was folded under an existing signal identifier[2].

[2]Similarly, the `Segmentation fault` message you know and love is left over from a time before paging, when virtual memory was managed using a scheme known as **memory segmentation**.

2 Special Exceptions: Traps

Not all exceptions represent error conditions. In fact, certain instructions **trap**, or deliberately cause a transfer of control to the operating system kernel. Traps are somewhat analogous to the function calls initiated by the `call` instruction; both often "return" to the following instruction once they finish. Alternatively, like a function, a system call might choose to prematurely exit the program rather than returning.

2.1 Breakpoints

In earlier activities, we saw examples of programs that manually caused the debugger to stop as if a breakpoint had been hit. Read `trap.s`, a short assembly program that triggers a breakpoint. Then try building (`$ make trap`) and running (`$ ./trap`) it.

7. Does this behavior look familiar? Besides as a trap, how would you classify this exception? *(Hint: Try to come up with* both *of the relevant categories of exception.)*

8. What happens when you run the program in the debugger (`$ gdb trap`)? Assuming the user `continue`s, would you still classify the exception in the same way?

9. *(advanced)* If you needed to implement an interactive debugger like GDB, what might you do to the program being debugged when the user asked you to create a breakpoint at a particular address?

2.2 System Calls

Another reason an application might trap is if it needs the operating system's help to perform some task. Many of the C library's functions are actually just thin wrappers around such **system calls**. This is especially true of those related to memory and I/O.

We can ask GDB to intercept system calls by creating a catchpoint. From within the same GDB session as before, try this now (`(gdb) catch syscall`) and rerun the program (`(gdb) r`).

10. Disassembling and continuing a couple of times, can you tell which instruction is trapping?

11. The runtime does some work before calling the `main()` function. The first system call it executes look familiar; what is it doing?

12. *(advanced)* Looking at the filenames passed to functions performing the `openat` system call, why are these files being opened? *(Hint: You may need to proceed past the first couple filenames before you start to recognize them.)*

3 Processes Scheduling

Now let's peek beneath the process abstraction and see how the operating system provides each process with its own notion of control flow.

3.1 Multicore

Take a look at `cores.c`, a program that spawns a user-specified number of child processes, each of which prints which CPU core[3] it's running on. When you're ready, build (`$ make cores`) and run (`$ ./cores`) it.

13. Try asking for 2 child processes. Do you notice anything about which CPUs the OS assigns them?

14. Notice that the program shows the number of schedulable cores. Now try requesting a number of children equal to one more than this number. What happens?

15. Looking at the output, do you see indication of whether the children were run **sequentially** (one after the other) or **concurrently** (overlapping in time)? (Note that the output is nondeterministic[4], so if the result doesn't match your hypothesis, try running the program once or twice more.)

3.2 Context Switching

Finally, glance over `timing.c`. This program prompts to ask whether it should spawn a single child process running a tight loop on the same core. Then, the parent constantly measures the amount of wall-clock time that has elapsed, and reports the minimum such time that has disappeared (above a certain detection threshold). Build (`$ make timing`) and run (`$ ./timing`) it.

16. Answer `n` when asked whether to spawn a child process, and notice whether the program experiences any lost time. Why do you think this did (or didn't) happen?

17. Run it again, this time answering `y`. What happens, and why?

[3] In the case of superscaler SMT (simultaneous multithreading) processors such as modern x86 CPUs, each physical core actually appears as two or more logical cores.

[4] The program uses a `sched_yield()` call to increase the chance of enlightening output: this library function causes a trap, during which the OS is allowed to give the core to a different process.