CS61 Scribe Notes **Fall 2014**

Lecture 21: Process Interruption and Signals

Scribes: Ansel Duff, Lukas Gemar, Jesse Chen

21.1 Utilization

Review and Motivation: By using a loop to wait for a child process to exit, our code has bad utilization since it uses polling - this can be seen in waittimeout.c.

21.1.1 Definition

Utilization is the fraction of some resource devoted to useful work. To quantify this concept, think of it as a number between 0 and 1 where a bigger number signifies higher utilization (and bigger is better).

21.1.2 Core Inquiry

Who decides what is useful?

Consider the example of the kernel - useful work for the kernel is when processes are running, since that is its goal. However, the kernel cannot examine process code holistically to determine its utility like we can.

From our perspective in the program waittimeout.c, we are simply running a busy loop, which we can evaluate to be not very useful (although we're using it because we don't want race conditions).

21.1.3 Example - waitblock.c

In this program we consider a blocking model, which we can implement using the interrupt functionality, as opposed to a polling model.

In waitblock.c, the parent process gets delivered a signal when a child process dies. While its running, a parent wakes up very quickly after the child dies - this is good utilization. Since the wait program and its child are both blocked, the overall cpu usage by the kernel is more efficient and utilization is higher, since less useless work is being done. This is another aspect of utilization functionality, that less useless work occurs.

21.1.4 Race Conditions

This is the phenomenon in which scheduling of executions leads to an incorrect outcome.

In the blocking code, the race condition happens since there exists the possibility that the child exits before we call sleep - if this happens, the interrupt will never happen and the parent will sleep for the whole sleep time even though the child has exited.

21.2 Signalling

Motivation: Consider the case in which we want to wake up a process or stop blocking when an event occurs. We cannot detect the event and sleep atomically.

21.2.1 Definition

Signals are models of hardware interrupts, used in linux to stop faulty processes. A simple example would be infinite loops.

Signals are not reliable; conceptually, they are more like booleans than counts. If a signal is delivered and another version of the same signal is waiting, the second signal is thrown away.

Atomic code is code that executes indivisibly, without interruption.

21.2.2 Sleep-wakeup Race

The scenario described above (motivation) is the problem known as the sleep-wakeup race. What are some possible solutions to this problem?

1. Put interrupts ″ on ice ″ to block signals?
2. Use a system call?

We can consider the propositions as follows;

1. There is a system call that attempts to do this, called SIGBLOCK. It takes a mask, a set of all signals that have been blocked so far; while a signal is blocked, it will not be delivered. Different signals can be masked at different times - however, every process can always be killed. For more information, check out the man page.

   Why doesn't this work? Although we can unblock the signal right before we sleep, the race condition still exists, just with less probability.
2. What about PSELECT? SELECT is a generic way to go to sleep for a certain amount of time - PSELECT takes a signal mask that allows us to atomically unblock a set of signals; this combines the functionalities of checking an event and going to sleep to solve the sleep-wakeup problem.

21.2.3 Example - waitblocksafe.c

This code demonstrates using pipes to solves the issues above; to get around the race condition, we make sure we sleep before we wake up in every case.

21.3 Interrupts

How can we deal with interrupts in the shell?

21.3.1 Process

1. Set up the foreground pipeline
2. If (shell received the ctrl+C command) { then kill the fg pipeline; // wakeup equivalent }
3. // if the user pushes ctrl+C here there is a problem (race-condition)
4. Put fg pipeline in foreground;
5. Wait for foreground pipeline; // sleep equivalent

How can we avoid the race condition? If we reorder lines 2 and 4, the child will get the ctrl+C and the problem will be solved.

21.4 Zombie Processes

21.4.1 Definition

This phenomenon occurs when a child process that was never waited for is still taking up resources. It may die, and the parent hasn't waited yet. Why does this take up resources?

Unix guarantees that when a child process dies, its parent waiting on it will get to access information about its death and its status upon death. Since the child died, unix keeps storing its death info until the parent claims it, but if the parent does not wait on it, then this is a zombie process.

21.4.2 General Info

There is always a process that waits for a spawned process, even if the parent process has died; this is what we will code for shell (a program whose job it is to run other programs).

There is an uber process called init (it has process id 1), and when the parent of a child dies, the system reparents the child to init.

The "pstree" command shows you a tree of all processes and their children on the system.

21.5 Networking

The general concept/value is when one machine is using the resources of other machines.

21.5.1 Terms to Know

Client - this is the ″ active party ″, the one that sends requests
Server - this is the ″ passive party ″, the one that replies to the requests with a response

21.5.2 Concepts

Clients access some information that the server is able to provide.

Port 80 - reserved for unencrypted web servers (http)
Port 443 - reserved for encrypted web servers (https)

Servers should be long lived, so they can respond to a many questions

21.5.3 Sockets

How can we represent the idea of messages between channels? One intuition is that this might be well represented by a pipe.

However in this case we need a file descriptor that represents a future channel - the system call we need is socket, and it creates an edgepoint for communication - it's a network file descriptor. Conceptually, the socket call creates an endpoint and the connect call binds them together.

The server needs a well-known endpoint, after which the client may seek it out and connect to it.

21.5.4 Connect

Three step process on client side:

1. look up address
2. create a socket
3. connect

Server side:

1. Create socket and set up options on that socket
2. Bind socket to a port
3. Listen for new connections

21.5.5 Accept

We do not use listening sockets for connections, since accept() solves the race condition problem that might arise.

accept() takes in a listening file descriptor and returns a new file descriptor for the connection – the connected socket is then bound to the client

21.6 Denial of Service Attack

One user of a service can prevent others from using a service - the attacker isn't gaining things directly, but prevents other people from accessing the service

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*