CPE-2600 FALL 2024

Systems Programming

Lab 10 Signals

Introduction

The purpose of this assignment is for you to explore the use of signals as an inter-process communication technique. You will learn how to register a signal handler as well as how to send, suspend, and view pending signals.

Work on the assignment is to be completed individually. You are welcome to collaborate with class members, but the submitted assignment must be your work alone.

The assignment is to be completed in GitHub Classroom. The assignment is here: https://classroom.github.com/a/H_7Ef-0a. Follow the link to accept the assignment then clone your new repository. All work is to be done on the main branch.

Background and References

Signals used in inter-process communication are used a notification mechanism between processes. Signals are unique in the sense that there isn't necessarily any data associated with the signal; rather, the receiver of a signal is only notified that it happened. There are, however, different signal types (integer values) to indicate different notifications.

Think of it as being similar to the case where someone taps you on your shoulder to get your attention. If you want to know why he/she is getting your attention, you have to ask.

In programming, signals work in a similar way; a process registers a handler for the signal. This handler is invoked when the signal is received by the process. Once the signal handler completes (returns), execution continues exactly where it was prior to the receipt of the signal.

If there is no handler registered, the operating system takes a default action.

- Signal Manual Page - https://man7.org/linux/man-pages/man7/signal.7.html
- Sending a Signal with the `kill` system call - https://man7.org/linux/man-pages/man2/kill.2.html

Note the similarity of the kill system call with the kill command-line command that was demonstrated in lecture which is documented here: https://man7.org/linux/man-pages/man1/kill.1.html

Project Description

Part 1: Signal Research

Research POSIX signals, be able to answer the following questions:

- What is a signal disposition?
- What is a signal handler? What is it used for?
- Name and describe each of the five (5) default dispositions?
- Name and describe one way to ***programmatically*** send a signal to a process. Be able to give an example (the code) to send a signal.
- Name and describe one way to send a signal to a process ***from the command line***. Be able to give an example (the command, key combination, etc.) to send a signal.

Each signal has a corresponding type. Research POSIX signal types. For ***EACH*** of the following signal types:

- SIGINT
- SIGTERM
- SIGUSR1
- SIGKILL
- SIGSTOP

Be able to:

- Name and describe the signal
- Define the default disposition taken by the operating system if a process does not define a signal handler
- Can the disposition be overridden by a signal handler? Why do you think this is the case?

Part 2: Working with a Signal Handler

The source file **signal_handler.c** (in assignment repository) contains the source for a program that registers a signal handler.

Compile and run the signal handler code.

- Determine two (2) ways to send the SIGINT signal to the process created for the running program.
    - ***TIP:*** Research the `kill` command
    - ***TIP:*** Some signals can be sent using key combinations from the command line (CTRL + other characters).
- Be able to describe how you sent SIGINT to the process and the behavior of the process when SIGINT is handled.

Modify the code so that it does ***NOT*** exit inside the signal handler. - Compile and run the program and send SIGINT to the process. Be able to describe the behavior. - Determine how to make the process exit. ***TIP:*** Research SIGKILL

- Be sure to update comments at the top of the source file and commit your changes to the file.

Part 3: Signals Sent From the Operating System

There are times when a running program performs an operation that is either not allowed or asks the operating system to send a signal to notify the process of the status of the system.

Using SIGALRM

One example of notification sent by the operating system is SIGALRM, which is sent as the result of a user calling the `alarm` function.

Research the `alarm` system call and the SIGALRM signal. - Write a program (signal_alarm.c) that schedules an alarm to be sent to after 5 seconds. Then write a signal handler to print out that the signal was received.

Handling SIGSEGV

Consider the following program (also included in **signal_segfault.c**):

```c
#include <stdio.h>

int main (int argc, char* argv[]) {
    // Declare a null pointer
    int* i = NULL;

    // Dereference the null pointer
    printf("The value of i is: %d\n", *i);

    // Return to exit the program
    return 0;
}
```

The de-reference of `i` causes a segmentation fault when the program attempts to load memory from a NULL pointer. The segmentation fault is actually the result of a signal sent by the operating system. Here, the signal is called SIGSEGV - SEGV stands for segmentation violation.

Research the SIGSEGV signal - Modify **signal_segfault.c** to install a handler for SIGSEGV. In your handler print a message that a segmentation fault was received, then return without performing any other action. - Run the program and observe the results. - What do you observe? Why is this happening? ***TIP:*** Note that when ***any*** signal is handled by a signal handler, execution of the program returns ***exactly*** where it left off before the signal was

received. In this case, execution continues by re-running the statement that de-references the NULL pointer.

- Be sure to update comments at the top of the source file and commit your changes to the file.

Part 3: Getting Details from a Received Signal

While the `signal` system call is one way to register a signal handler with the operating system, it does not allow the signal handler to receive a lot of information about the received signal (just the signal number). What if the signal handler needs more information, for example what process sent the signal. For that, there is another system call `sigaction` that can be used to register a signal handler that receives a structure of values for each received signal.

The `sigaction` system call allows the process to customize a lot about the signal handler. It can set a signal to be completely ignored (not allowed for SIGSTOP or SIGKILL), retrieve the memory address that caused a fault (e.g. for SIGSEGV), get the process identifier that sent the signal, among other things.

Research the `sigaction` system call. - Write a program (signal_sigaction.c) that uses `sigaction` to register a handler for the SIGUSR1 signal. - After registering, have the program wait in an infinite loop (see previous examples). The program doesn't need to print anything inside the loop. - In the signal handler registered using `sigaction`, print out the process identifier of the sender, then return. - Send the SIGUSR1 to the process and observe the output. - Write and record in comments of the program a command that can be used to send SIGUSR1 to the process.

HINT: Pay attention to the `sa_flags` field on the `sigaction` structure. Read about `SA_SIGINFO` flag.

Part 4: Sending Data with a Signal

While using `sigaction` it is possible to retrieve information about a received signal, for example: - the address of the violating memory access in the case of SIGSEGV - process identifiers for the sending process

What if a process wants to send data along with a signal, for example when using one of the user signals (e.g. SIGUSR1)? The `kill` system call doesn't allow for data to be sent with the signal. To send data along with the signal a process can use the `sigqueue` system call https://man7.org/linux/man-pages/man3/sigqueue.3.html

Research the `sigaction` system call. - Write a program (recv_signal.c) that uses `sigaction` to register a handler for the SIGUSR1 signal. - After registering, have the program wait in an infinite loop (see previous examples). The program doesn't need to print anything inside the loop. - In the signal handler registered using `sigaction`, retrieve the `sival_int` and print out this value, then return. - Write a second program (send_signal.c) that sends

SIGUSR1 along with a random integer (see https://man7.org/linux/man-pages/man3/srand.3.html) to the process using `sigqueue`. Print this number in the sending program before sending SIGUSR1. - ***NOTE:*** To send a number that is sufficiently random make sure to seed the random number generator with `srand`. See the manual page for more information. Using the `time` function is a sufficient way to seed the generator (https://www.tutorialspoint.com/c_standard_library/c_function_srand.htm)) - ***NOTE:*** The process identifier assigned to a program changes every time a program is executed. You will need to figure out how to send the message to the correct process. - ***HINT:*** Utilize the command line and `atoi` to convert a command line string to a number

HINT: Pay attention to the `sa_flags` field on the `sigaction` structure. Read about `SA_SIGINFO` flag.

NOTE: You'll be creating two (2) programs for this exercise. Make sure you specify the `-o` parameter on `gcc` to get different executable names for each program.

For example:

```
gcc -o send_signal send_signal.c
gcc -o recv_signal recv_signal.c
```

Part 5: Signal Tennis

This part is OPTIONAL and worth 20 extra credit points on the assignment.

Your implementation must have all of the base features along with the challenge features of scorekeeping and randomness. Scorekeeping may be tennis, table tennis, pickle ball, or badminton rules - your choice. You must demo for extra credit.

Template files are not in the repository. Create and add files to repository for implementation.

Given what you learned about signals, use this knowledge to write two (2) programs to play signal tennis. These two programs will send signals back and forth to each other to simulate a tennis ball. The signal to use for the ball is up to you. Note this activity is substantially similar to Part 4, so maybe start there.

Development Requirements

- The first program (the receiving program):
    1. Installs a signal handler for the 'ball' signal, the signal handler should determine the process that sent the signal (the other player)
    2. Waits for a random amount of time between 1 and 2 seconds
    3. Sends the signal (the ball) back to the sender
    4. For each step (receiving signal, sending signal), the program should print out a status message to indicate what it is doing. Also play the system bell ('' or '\007') to simulate a ball being hit.

- The second program (the serving program):
    1. Installs a signal handler for the 'ball' signal, the signal handler should determine the process that sent the signal (the other player)
    2. Waits for a random amount of time between 1 and 2 seconds
    3. Sends the signal (the ball) back to the sender
    4. For each step, the program should print out a status message to indicate what it is doing and play the system bell when sending the signal (hitting the ball)
    5. After the server sets up the signal handler, it should serve the ball (send the signal) to the other player.

Run the receiving program first and then run the serving program to play the game. The game must exit cleanly (without crashing or requiring CTRL+C) after a successful volley (sending the signal 'ball' back and forth) of 10 exchanges.

Here are some things to consider:

- How does the serving know the process identifier of the other player?
    - HINT:
        1. Run the receiving process first
        2. Figure out the process identifier using `ps` or have the receiving process print its pid to the console.
        3. Utilize the command line on the serving program to give it the process identifier of the receiving process and `atoi` to convert a string to a number
- How do you end the game?
    - HINT: Use `sigqueue` to send the signal 'ball' and include the current count of exchanges.

Challenge

Signal tennis does not have the features of regular tennis. If you accept the challenge can you use the signaling functions (e.g. `sigqueue`, `kill`, `sigaction`, etc.) to:

- Keep score using traditional tennis scoring: https://www.usta.com/en/home/improve/tips-and-instruction/national/tennis-scoring-rules.html
    - This will require some sort of randomness or heuristic to determine if the player succeeds or fails to return the ball
    - An additional signal or value sent with the signal will be needed to tell the other player that the ball wasn't successfully returned
    - This will also require the serving process to serve again after each point. You are free to use the same program to always perform the serve.
- End the game when one player looses using traditional tennis scoring

- Only implement a single game

Code Structure

Code must follow style guidelines as defined in the course material.

Hints and Tips

Testing and Debugging

Debugging

See the course debugging tips for using `gdb` and `valgrind` to help with debugging.

Your program must be free of run-time errors and memory leaks.

Deliverables

When you are ready to submit your assignment prepare your repository:

- Make sure your name, assignment name, and section number are all files in your submission - in comment block of source file(s) and/or at the top of your report file(s).
- Make sure you have completed all activities and added all new program source files to repository.
- Make sure your assignment code is commented thoroughly.
- Make sure all files are committed and pushed to the main branch of your repository.
- Tag your repo with the tag `vFinal`

NOTE: Do not forget to 'add', 'commit', and 'push' all new files and changes to your repository before submitting.

To submit, copy the URL for your repository and submit the link to associated Canvas assignment.

You may be asked to demo one or all of the programs.