



COMS3010A

Operating Systems

Lab 3 - Singals

Instructors

COMS3010A Lecturer:

Mr. Branden Ingram branden.ingram@wits.ac.za

COMS3010A Tutors:

Mr. Joshua Bruton 1699140@students.wits.ac.za

Mr. Mark Alence 1676701@students.wits.ac.za

Mr. Tshepo Nkambule 1611821@students.wits.ac.za

Mr. Dylan Brandt 1352906@students.wits.ac.za

Mr. Skhumbuzo Dube 1608194@students.wits.ac.za

Consultation Times

Questions should be firstly posted on the moodle question forum, for the Lecturer and the Tutors to answer. If further explanation is required consultation times can be organised.

1 Introduction

This is a small lab that lets you explore signals in a Linux environment. You need access to a Unix machine and know how to use the C compiler. If you are on Windows I suggest - <https://docs.microsoft.com/en-us/windows/wsl/install-win10> the Windows Subsystem for Linux. The small examples that you will implement are not rocket science but if you have done them once, it will hopefully help you understand the operations of the operating system and how it can control processes. Note - the examples that are given in this lab are probably nothing that you will ever use in regular programming. Some are even things that you should never do; the examples are here to help you gain a better understanding of what is going on under the hood i.e. how the operating system can interact with running programs.

2 Signals

Signals are primarily used by the operating system to signal to processes that something has happened that probably needs some attention. It could also be used in between processes or even inside a process to raise an exception. If you open a shell you can list all possible signals by using the command “kill”. As you see there are quite many, but don’t be afraid, you do not have to learn them by heart.

- kill -l

You might have used the “kill” command when you wanted to kill a process but the command will be able to send any signal to a process. When no signal is given the “SIGTERM” signal is sent to the process. You might have learned to write “kill -9” when you really wanted to kill something; what is signal number 9 called? When you hit “ctrl-c” in a terminal window, the shell will send a “INT”(interrupt) signal to the foreground processes. There is some magic taking place when you’re running a program in a shell and the “ctrl-c” will

terminate the program but not the shell; all will be crystal clear in a couple of weeks when we look into threads. To learn a bit more about how signals work we will write small examples and see how they behave.

3 catching a signal

Normally when you run a program the process inherits the default signal handlers of its creator. The process can then set its own signal handlers to change the behavior of the process. One example is when a process wants to do some final things before terminating when it receives the “SIGTERM” signal.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int volatile count;

void handler(int sig){
    printf("signal %d ouch that hurt \n" , sig);
    count++;
}

int main(){
    struct sigaction sa;
    int pid = getpid();
    printf("ok, let's go ,kill me (%d) if you can!\n " , pid);
    sa.sa_handler = handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if(sigaction(SIGINT, &sa, NULL) != 0){
        return(1);
    }

    while(count != 4){

    }

    printf(" I've had enough! \n");
    return(0);
}
```

Copy this program, call it “test1.c”, compile and execute it in a terminal. In this terminal try to kill it by hitting ctrl-c. The program does not do very much besides trying to stay alive. If we take a look at how it achieves this we see that it uses a “sigaction” structure and a call to “sigaction”. The structure holds some parameters to control what will happen. We will set the “sa_handler” property to point to our own handler procedure. We will then clear the “sa_mask” entry because we do not want to block any other signals when we’re in the handler. The interesting thing is the call to “sigaction”, here we pass three arguments: the signal we want to handle, a pointer to the sigaction structure and a null pointer (that we don’t have to bother about now). This is the library call that will change our signal table and when a “INT” signal is sent we will be able to do what we want. The signal handler itself will in this example not do anything useful, only print a message so that we know that it was invoked. Note that one should not use a system library call such as printf inside a handler since this might be in conflict with an ongoing library call. We do so in this exercise to keep things as simple as possible. Run the program and do some experiments; can you kill it by sending a “kill -SIGINT” from another terminal.

What happens if you send it a “SIGTERM” signal? Look up “kill” and “sigaction” using the “man” command. There is probably a lot more here than you would ever want to know but you should get the habit of reading the “man” pages. This is the simple way to catch a signal; using a slightly different technique we can get some more information on what is going on.

3.1 catch and throw

In Java you’re quite used to declaring what exception methods could cause and you could always trap them in an exception handler to possibly do something else. When you’re programming in C there is no such support in the language, but it turns out that you can use signals to handle exceptions. Let’s catch any exception caused by division by zero. The handler will simply print an error message and then exit but we could of course have done something to save the situation. Create another c file and call it “test2.c”, within it you will be adding the following segments of code.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig){
    printf("signal %d was caught \n", sig);
    exit(1);
    return;
}
```

We need to install the handler and do so by registering it under the “SIGFPE” signal. Then we call a not so good procedure that will divide by zero (note, it’s integer division so it will generate a fault).

```
int not_so_good(){
    int x = 0;
    return 1 % x;
}

int main(){
    struct sigaction sa;
    printf("Ok, let's go I'll catch my own error. \n");
    sa.sa_handler = handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);

    / and now we catch... FPE signals /
    sigaction(SIGFPE, &sa , NULL);

    not_so_good();

    printf(" Will probably not write this. \n");
    return(0);
}
```

You might ask yourself how a division by zero is detected as a fault and how it is turned into a signal to the user process. The fault is first detected by the hardware; when the instruction is executed an exception is raised. The CPU will then look in an “Interrupt Descriptor Table” (IDT) and jump to a location in memory that

hopefully contains some code that will take care of the problem. This code is part of the kernel so the kernel decides what to do. In the case of a division by zero the user process that generated the fault must of course be interrupted. If the process has registered its own “SIGFPE” handler, as in the case above, control is passed to this function; the default procedure is to kill the process.

4 who do you think you are

If we use the simple version of signal handler then there is not much information to go on; a signal has been sent but we don’t know much more. We can however use a slightly more elaborate call that gives us some more information. To use this version we set a flag in the sigaction structure. Create a new file called “test3.c” and complete the following code segments.

```
int main(){
    struct sigaction sa;
    int pid = getpid();

    printf("Ok, let's gokill me (%d).\n" ,pid);

    / we're using the more elaborated sigaction handler /
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = /* INSERT HERE */
    sig emptyset(&sa.sa_mask);

    if(sigaction(/* INSERT HERE */, &sa ,NULL) != 0){
        return(1);
    }

    while(!done){

    }

    printf("Told you so! \n");
    return(0);
}
```

Nothing strange there, but now the handler will be passing three arguments: the signal number, a pointer to a “siginfo_t” structure and a pointer to a context that we can ignore.

```
int volatile done;

void anotherhandler(int sig, siginfo_t *siginfo, void context){
    printf("signal %d was caught \n", sig);
    printf("your UID is %d\n", siginfo>si_uid);
    printf("your PID is %d\n", siginfo>si_pid);
    done = 1 ;
}
```

The “siginfo_t” structure contains information about the process that sent the signal. If you start this program in one shell and then try to kill it from another shell (using “kill -SIGINT”) you should hopefully be able to tell which process that tried to kill you.

5 Summary

Signals is the mechanism that the kernel uses to inform processes about exceptions in the normal execution. If the user process has not stated otherwise, the kernel will decide what to do. The user process can register a signal handler to decide what to do. The kernel will then pass control to a specified procedure. Signals can also be used in between processes, to send notifications to, or control processes. The signals themselves contain no information, but the kernel can provide more information about who sent the signal and possibly why.

Academic Integrity

There is a zero-tolerance policy regarding plagiarism in the School. Refer to the General Undergraduate Course Outline for Computer Science for more information. Failure to adhere to this policy will have severe repercussions.

During assessments:

- You may not use any materials that aren't explicitly allowed, including the Internet and your own/other people's source code.
- You may not access anyone else's Sakai, Moodle or MSL account.

Offenders will receive 0 for that component, may receive FCM for the course, and/or may be taken to the legal office.