Week 06
Signals, C Programs, SetUID, SetGID, and Fun Stuff

Melvyn Ian Drag

October 10, 2019

Abstract

We'll learn about signal handling, C Programs, ssh, setuid, setgid, and talk about the exam.

1 SIGTSTP and SIGINT

In nix a common thing you do is send signals to processes. You will sometimes want to send information to a running program to make it do things. Signals are how we do that.

The most common signals you'll want to send are **SIGINT** and **SIGTSTP**

We'll see what these do by running a small python program. Create this program and run it on your computer:

```python
import time

i=0

while True:
        i+=1
        print(i)
        time.sleep(1)
```

This program will run forever and hog up our terminal. How do we make it stop? We can send the **SIGINT** signal by typing "CTRL+C". Note that it stops. Another thing you might want to do is just pause the program for a bit. You pause the program by sending it a **SIGTSTP**. This is done by typing CTRL+Z. To restart the program that you paused you can send it a **SIGCONT** command. To send a **SIGCONT** command is a little different, you must type

```
user@machine$ kill -18 $PYTHON_PID # where $PYTHON_PID is the
   PID of the process you stopped.
```

1.1   Verify the signals

1. type *pidof python* (or *python3*)

2. Run *runForever.py*

3. Send **SIGINT** with the keyboard

4. type *pidof python* (or *python3*)

5. There should be no pid there corresponding to your process.

6. Run *runForever.py*

7. Send **SIGTSTP** with the keyboard

8. type *pidof python* (or *python3*)

9. There **should** be a pid corresponding to your process.

10. Restart the process with *kill -18 $ PID*

11. Type *jobs*

12. Type *fg $JOB_ID*

13. Kill the process so it doesn't hog memory.

2   jobs

Note I used the *jobs* command above. That command lists the processes started by the current shell. If you start a new shell and type *jobs* there will be no output, because your new shell didn't do anything yet.

3   kill

You use the *kill* command to send a signal to a process. I showed you one example wherein we sent an 18 to a process. You can see all of the signals available to send by typing

```
user@machine$ kill -l
```

Using the kill command you can send these signals in three ways:

- Using the signal name e.g **CONT**

- Using 'SIG' + the signal name e.g. **SIGCONT**

- Using the numeric code for the signal e.g. 18

In this lecture we're going to focus on the following signals:

1. SIGINT

2. SIGKILL

3. SIGTERM

4. SIGSTOP

5. SIGTSTP

6. SIGSEGV

though there are more signals, this should be enough to get you started. And if you want to learn more you of course can go get a book. We're going to look at when you use each one of the above and how programs handle them when they are received. I think these are the most interesting of the 64 signals.

Let's test the signals

```
user@machine$ python3 runForever.py & # ampersand puts it in
   the background
user@machine$ jobs
[1]+Running   python3 runForever.py &
user@machine$ kill -20 %1 # or kill -20 $PIDOF_PYTHON. kill
   takes job ids or pids
user@machine$ jobs
#stopped
user@machine$ kill -18 %1
user@machine$ jobs
# it's running again
user@machine$ fg 1
user@machine$ [type ctrl + z]
```

We just sent **SIGCONT** and **SIGTSTP** using the numeric codes for them. Remember that we got the numeric codes by looking at the output of *kill -l*. You'll note that, while the command is called *kill*, it does more than kill. It can send all kinds of signals.

There is a signal related to **SIGTSTP** called **SIGSTOP**. The difference between them is that **STOP** cannot be ignored and typically comes from a program. **TSTP** can be ignored. Depending on how much you've used linux, you may or may not have been in a situation where you were mashing *CTRL+C* over and over again, but the program wouldn't stop! By the end of tonight we'll write some code showing how that happens. Just know, that by the design of the Linux OS, **STOP** signals cannot be ignored.

```
user@machine$ python runForever.py &
user@machine$ jobs
[1]+ Running python runForever.py &
user@machine$ kill -STOP %1
user@machine$ jobs
# it's stopped
```

```
user@machine$ kill -18 %1
user@machine$ jobs
# it's running
user@machine$ fg 1
user@machine$ [type ctrl + z]
```

4   fg and bg

You'll note I've used *fg* a bit. This command takes a job id and puts it in the foreground. When you continue a job with **SIGCONT** it starts in the background. You have to bring it back into the foreground with *fg* so you can interact with it e.g. send it a **SIGINT**. *bg* has the opposite effect of putting a job in the background, like we did with &

5   INT, TERM, KILL

INT is sent when you hit 'CTRL+Z' on your keyboard. It interupt the process and makes it stop. 'SIGINT' can also be sent will *kill*. **SIGTERM** and **SIGINT** are approximately the same thing, differences in their behavior are left up to the application developer. You should now send **INT** and **TERM** to the *runForever* program and you will see that both end the process. **KILL** is like **INT** or **TERM** but it cannot be ignored.

A special signal is signal 9. This signal cannot be blocked or ignored. You can always send signal -9, no matter what is happening this signal will work. It cannot be blocked or ignored! In any Linux system administration book you read you will see thei **warning: 'NEVER USE -9¡**. We have just seen that processes can deliberately ignore or block signals if interruption would cause some serious system harm. But sometimes, we just want the darned process to stop! You can use SIGKILL under very dire circumstamstances, because issuing tihs signal could cause file corruption!

Actually, thats what they say. I use **SIGKILL** sometimes and don't worry about it. My reason for doing so is boring to explain. I encourage you to read the following posts to develop your own opinion of whether or not to use SIGKILL/KILL/9

- https://www.reddit.com/r/linux/comments/4b1mwh/do_not_use_sigkill/

- https://unix.stackexchange.com/questions/281439/why-should-i-not-use-kill-9-sigkill

- http://turnoff.us/geek/dont-sigkill/

- https://unix.stackexchange.com/questions/8916/when-should-i-not-kill-9-a-process

Whether you use the following code or not in your daily life, on your own laptop . . .

```
user@machine$ kill -9 $pid
```

when you are in the office make sure you use **SIGINT** and **SIGTERM** first or your colleagues might look down on you as a reckless person.

6   Exercise

Have students experiment sending the different signals to runForever.py until I'm sure they can send them all. Try to send signals with CTRL+C, CTRL+Z and then on the command line with *kill*

7   C Programming

We are going to write some C programs now. C is a language that is very intimately related to Linux. It looks and feels somewhat like other languages like

1. Java

2. C#

3. C++

4. But has many similarities to others. . .

So if you have any experience with those languages, you'll more or less be able to read C. Example program:

```c
#include <stdio.h>
int main(int argCount, char** args){
        char * h = "hello";
        char * w = "world";
        printf("%s %s", h, w);
}
```

To compile and run this code you:

```
user@machine$ gcc helloworld.c -o helloworld # compile the code
user@machine$ ./helloworld # run the code
```

8   Writing Custom Signal Handlers With C

We will change the behavior now. Run the code. From a separate terminal window, send kill signals. Then inspect the return code of the running process and see how we customized the return valu based on which signal we received. Remember that in our first assignment we looked at $? to see how a program ended. You can return different values not just fro mbash scripts, as we've done with 'return' and 'exit' statements up until now. Your software can be programmed to return values too.

Returning to our discussion of signals, signals can be caught, ignored, or blocked.

Most important thing for these three samples

GO SLOOOOOOOW

Open two Terminals and show how you send various signals to the process using *kill*

When appropriate (i.e. TSTP or INT) also show you can use CTRL+C and CTRL+Z

8.1   01_signal_handler.c

This code shows the Linux programs can choose how to handle signals.

We will change write come code to change signal handling behavior now. Open '01*.c' on your computer and have a look at it. This is a small C program that registers two signal handlers. Remember, all these linux command line tools we are using like mv, cp, etc. are all simple C programs like the one we are writing now. They all register signal handlers like we are going to do now, and that's how the programs know what to do when you type CTRL+C or CTRL+Z on the keyboard, or what to do when you send a command with *kill*

modify the program, compile, run. Allow students a few minutes to change the messages returned by SIGINT and SIGTERM.

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


static volatile int keep_running_int = 1;
static volatile int keep_running_term = 1;

static void sig_handler_int(int i)
{
    (void) i;
    keep_running_int = 0;
}

static void sig_handler_term(int i)
{
    (void) i;
    keep_running_term = 0;
}

int main(void)
{
    signal(SIGINT,  sig_handler_int);
    signal(SIGTERM, sig_handler_term);
```

```c
    while ( ( keep_running_int) && ( keep_running_term ) ) {
        puts("Still running...");
    }

    if( !keep_running_term ){
        puts("Received SIGTERM!");
        return 1;
    }
    else if ( !keep_running_int ){
        puts("Received SIGINT!");
        return 2;
    }

    return 0;
}
```

8.2 How to compile?

Go to the SignalHandlers directory and type:

```
user@machine$ gcc 01_signal_handler.c -o 01.out #compile
user@machine$ ./01.out #run
```

8.3 02_ignore.c

This example shows that we can ignore a signal! We can tell a program to ignore a signal.
    What happens if you tell a program to ignore a signal that is unignorable?

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

static volatile int keep_running = 1;

static void sig_handler(int i)
{
    (void) i;
    keep_running = 0;
}

int main(void)
{
    signal( SIGINT, sig_handler );
    signal( SIGTERM, SIG_IGN );
```

```
    while ( keep_running ) {
        puts("Still running...");
    }

    if ( !keep_running ){
        puts("Received SIGINT!");
        return 1;
    }

    return 0;
}
```

8.4   03_block.c

This example shows that programs can temporarily refuse to handle signals. This is done with blocking. This program temporarily blocks a signal - can you see which one?

Try to send the signal while it is blocked and see what happens!!

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

#define STRANGE_ERROR 1
#define SHOULDNT_SEE_THIS 2
#define SLEEP_TIME 5

int main( int argc, char** argv ){
    int i;
    sigset_t intmask;
    int repeatfactor;
    double y = 0.0;

    if(( sigemptyset(&intmask) == -1 ) ||( sigaddset(&intmask,
       SIGINT) == -1 )){
        perror("Failed to set up the signal mask");
        return STRANGE_ERROR;
    }
    while(1){
        printf("Entering BLOCK state\n");
        if( sigprocmask(SIG_BLOCK, &intmask, NULL) == -1 ){
            break;
        }
        fprintf(stderr, "SIGINT signal blocked\n");
        sleep(SLEEP_TIME);
```

```
        printf("Entering UNBLOCK state\n");
        if( sigprocmask(SIG_UNBLOCK, &intmask, NULL) == -1 ){
            break;
        }
        fprintf(stderr, "SIGINT signal unblocked\n");
        sleep(SLEEP_TIME);
    }
    perror("Failed to change signal mask!");
    return SHOULDNT_SEE_THIS;
}
```

9 Summary

What can happen with signals?

- They can be handled

- They can be ignored

- They can be blocked

10 What are signals?

Signals are a form of interprocess communication used in Linux. There are many interprocess communication mechanisms, this is just one of them. If you've never heard of IPC, just know it is all around you when you are using computers. For example, consider the case when you are on your telephone and you're listening to music. When a phone call comes in, the music is probably paused and the phone app is brought up on the screen. The operating system on your phone needed an IPC mechanism to detect that the phone process was in some active state and that other interfering processes need to take action while the call is active. Depending on how the software is written on your phone, the music might resume playing when you hang up, and the music app might come to the foreground on your screen. Or maybe it stays paused and you have to manually reopen the app and hit 'play' again. Or maybe something else happens.

IPC might also be used if you have two processes on a device - maybe you have one process listening for radio transmissions and another on that turns a motor. If the radio receiver receives a signal, maybe you want another process to turn a motor and open your garage door or something.

There are many ways to do IPC, even on Linux there are a bunch of IPC mechanisms. Signals is just one of them.

11 ssh and break

ssh is a command line tool you use to access remote machines. This tool has many uses, and I use it every day. I'm very surprised that so many of you don't know it, but that's fine, it's a pleasure to show you!! I can't imagine how life was before ssh! It's truly one of the most common protocols/tools that Linux people use.

I'm going to show you how to connect to my raspberry pi at my house.

Take 20 minutes for everyone to do the following. Students need to install software, create an ssh key, the fork and pr my repo. Should take a while. Then they need to connect to the server.

1. have students install openssh-client

2. have students generate an ssh key with ssh-keygen

3. create a repo called ssh-keys

4. have students make a pr with their ssh keys to that repo - add the contents of your id_rsa.pub to the Readme.md

5. create a user on the rpi

6. put all the public ssh keys in the authorized_keys file

7. have students try to connect with username@ip_for_rpi

8. have the students make a text file on the raspberry pi

For your midterm you're going to use ssh to connect to a remote machine and configure a postgres database on it. Next week in Lecture I'll show you how databases work.

12 SetUID and SetGID

12.1 TL;DR

In the next few minutes we're going to write a program that you can run as a non-root user and delete files belonging to root. This shouldn't happen! But it can if you know this cool trick.

12.2 Introduction

There are some special permissions that can be assigned to executables in Linux. You can trick your operating system into running a process as root, even though you are not root. The idea being, maybe you have a very trusted program that needs to do some root stuff. You are absolutely sure that the root stuff it does will not destroy your system, so you want to allow other users to do the root stuff even though they aren't root. ¡Flesh this out with some more discussion until it is clear¿.

12.3   war story

Show Sparcus.com

A use case that I had was I wrote a very secure docker container and wanted anyone to be able to build and launch it, connected to a few system resources. I needed root permissions to make this happen, but I didn't want to drop into root all the time I wanted to do this safe thing, so I wrote a little program that would do some "rootly stuff", set the permissions and ownership of the file appropriately, and now I can run it as an unprivileged user.

If you call setuid(0) in your program, it changes you to the root user from the eyes of the operating system and lets you do rootly stuff.

12.4   uid

In Linux, all users have a user id. You can get your user id from the command line by typing 'id'. We haven't talked about this uid yet, and we certainly haven't used it, but it has been there all along. Today we will see what it is and we'll use it.

Try this:

```
$ id
[your id]
$ sudo su -
$ id
[you see user id is 0]
```

So now we know what the uid of the root user is. Check out this program

```c
/**
 * What a great example this is.
 * You see calls to 4 cool Unix functions!
 *    1. set_uid() - sets the user id
 *    2. set_gid() - sets the group id
 *    3. fork() - makes a copy of a process
 *    4. execve() - creates a new process
 *
 * I don't expect you to understand the details.
 * Alot of things have to go unsaid about this program.
 * I just want you to see that we can run this root command
   from
 * a non root user!!
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

char* const root_command[4] = { "/bin/sh", "-c", "cat
   /root/hello.txt"};
```

```c
int main(void)
{
    int uid_of_person_running_code = getuid();

    printf("My UID is: %d. My GID is: %d\n", getuid(),
        getgid());

    if ( setuid(0) ) // Note the return value 0 means good!
    {
        perror("setuid");
        return 1;
    }

    printf("My UID is: %d. My GID is: %d\n", getuid(),
        getgid());

    // Good Linux stuff ahead!
    // fork() call followed by execve()!
    pid_t pid = fork();
    if (pid){
        execve( root_command[0], root_command, NULL );
    }

    //Time to drop back to regular user privileges
    setuid(uid_of_person_running_code);

    printf("My UID is: %d. My GID is: %d\n", getuid(),
        getgid());

    return 0;
}
```

We see that with this program we make a few calls to getuid() and a few calls to setuid(). You don't have to worry too much about this program - if you don't know C it will maybe scare you a bit. That's why I've chosen this simple and illustrative program. The main idea is that you see some source code here that calls these mysterious functions getuid() and setuid()! We can change our user id while the program is running to trick the computer into thinking you are root!

'getuid' returns the user id of the user running the program.

We will first compile this normally and run it. You will see it fails and it exits with the value 1.

```
$ gcc setuid.c -o setuid
$ ./setuid
...
```

```
error msg
$ echo $?
1
```

To make this work, we will have to change the ownership, the group, and the permissions.

```
$ sudo chown root:root setuid
$ sudo chmod +s setuid
$ ls -l setuid
$ ./setuid
```

So you see that both user and group have an 's' where there used to be an 'x'. This means that the program can setuid() and setgid(). For this example we only wanted to setuid(), so we can just set u+s.

```
$sudo chmod g-s setuid
$ls -l setuid
$./setuid
```

Notice if we make group have '+s' but take the privilege away from user,

```
$sudo chmod u-s,g+s setuid
$ls -l setuid
$./setuid
[fails]
```

Also, rememberwhen we were setting permissions using the octal notation? You can do that for this as well. There is no setAllID, there isonly setuid and setgid So set uid is a 4, setgid is a 2, but what does the 1 correspond to? Not a big deal, I'll leave it here for you to read if you're curious! https://askubuntu.com/questions/432699/what-is-the-t-letter-in-the-output-of-ls-ld-tmp . So getting back to octal numbers and setuid/setgid we can set the permissions to r-sr-xr-x by saying 'chmod 4555'. We can set the permissions to 'r-xr-sr-x' with 'chmod 2555'. Etc.

13   Homework

Discuss the homework in detail

14   Exam

Discuss the exam in detail

15   Show Reading Materials

If time, have class download the vim book. Make them skim it and learn something. They can all leave after they've learned and showed me they've learned 3 things about vim.