C Tutorial: Playing with processes

Catching signals

Signals are a mechanim in Unix to alert, or poke, a process when one of several events happen. In some cases, these signals are generated to kill a program (SIGKILL) or notify the program that an illegal instruction, illegal memory access, invalid system call, or floating point exception took place. Other signals can be user-defined (SIGUSR1, SIGUSR2). When dealing with creating processes, the "child status has changed" signal, SIGCHLD, will be of interest to us. Take a look at the manual page for signal on your system (`man signal`) to see the full spectrum of signals that is available — they differ somewhat from system to system.

The program will usually exit upon receipt of a signal but it is possible to use the *signal* function to keep that from happening and either transfer control to a user-specified function when a specific signal is received or to ignore the signal.

We'll go through a few brief examples of using signals

Example 1: catching your own memory access errors

Here's a tiny program that does nothing but read memory location 0, which is an invalid memory location for user processes. This is the same as the classic bug of dereferencing a null pointer, which generally causes the shell to show you a `Segmentation fault` error message. Compile and run this program just to remind yourself of what happens.

```
/* an example that dereferences a null pointer */
/* Paul Krzyzanowski */

int
main(int argc, char **argv) {
        int a;

        a = *(int *)0;   /* read address 0 */
}
```

Download this file

Save this file by control-clicking or right clicking the download link and then saving it as `no-signal1.c`.

Compile this program via:

```
gcc -o no-signal1 no-signal1.c
```

If you don't have gcc, You may need to substitute the gcc command with cc or another name of your compiler.

Run the program:

```
./no-signal1
```

Now let's add a signal handler to trap the signal. In this case, the signal of interest is `SIGSEGV` – segmentation violation. See `man signal` for the full list of signals. We create a function called *catch* that will simply print a message and exit. To tell the operating system that it should transfer control to *catch* when the `SIGSEGV` signal is received, we add a call to:

```
signal(SIGSEGV, catch);
```

This call to *signal* should, of course, be made early on in the code before we anticipate getting the event. The function *catch* gets passed a parameter that contains the signal number. You can call *signal* multiple times to have different signals call the same function (or different functions). Try this code.

```
/* signal example #1: trap an illegal address reference */
/* Paul Krzyzanowski */

#include <stdio.h>      /* for printf */
#include <stdlib.h>     /* for exit */
#include <signal.h>     /* defines signals and the signal() function */

int
main(int argc, char **argv) {
        void catch(int);
        int a;

        signal(SIGSEGV, catch); /* catch SIGSEGV - segmentation violation */
        a = *(int *)0;  /* read address 0 */
}

void
catch(int snum) {
        printf("got signal %d\n", snum);
        exit(0);
}
```

Download this file

Save this file as `signal1.c`.

Compile this program via:

```
gcc -o signal1 signal1.c
```

Run the program:

```
./signal1
```

Example 2: Sending a signal to a process

Our next example involves sending an external signal to a process that's happily running and minding its own business. We will send it `SIGUSR1`, which is a user-defined signal via the *kill* command. The *kill* command is generally used to kill a process, and it does so by sending it a `SIGTERM` – the signal to terminate a process. That's just the default behavior and you can use the command to send other signals via the `-s` option. To send the `SIGUSR1` signal, you would simply run:

```
kill -s USR1 process_id
```

If you were programming this, you would code:

```
kill(SIGUSR1, process_id)
```

Here, the main function does three things:

1. It calls *signal* to tell the operating system to call the function *catch* if the process receives a `SIGUSR1` signal.
2. It prints a message with its own process ID number to tell the user how to send the *kill* command.
3. It then goes to a loop where it continually goes to sleep for one hour (3600 seconds)

Without receiving a signal, the program is essentially sleeping forever. When you run the program and send it a `SIGUSR1` signal via the *kill* command, however, the *catch* function is caught and prints a message. It then returns, which causes the program to go back to its endless for() loop.

```
/* signal example 2: catch an external signal */
/* Paul Krzyzanowski */

#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>

int
main(int argc, char **argv) {
        void catch(int);
        int waitstat;

        signal(SIGUSR1, catch);
        printf("Run this in another window: kill -s USR1 %d\n", getpid());
        for (;;) {
                printf("Going to sleep\n");
                sleep(3600);     /* for a good long time */
        }
}

void
catch(int snum) {
        printf("got signal %d\n", snum);
        /* on SunOS systems, we have to  reset the signal catcher */
        signal(SIGUSR1, catch); /* don't need this on Linux or OS X (but doesn't hurt) */
}
```

Download this file

Save it as `signal2.c` and compile it program via:

```
gcc -o signal2 signal2.c
```

Run the program:

```
./signal2
```

On Linux and Mac OS X systems, it is sufficient to have the *catch* function simply call the *printf* function and do nothing else. On SunOS systems, the signal has to be reset whenever it is caught, so you need to add the extra call to *signal*.

Example 3: detecting the termination of a child process

A parent process receives a `SIGCHLD` signal whenever the child process terminates. This allows a parent process to go off and do other things but still track the status of its child processes. What we do to accomplish this is set up a signal handler that gets called whenever the process gets a `SIGCHLD` signal and have that signal handler call *wait*. Normally, *wait* puts the process to sleep but in this case we already have that outstanding signal so *wait* simply picks up the status of a child and returns immediately. Calling *wait* on child processes is good programming practice since it ensures that the child process does not remain a zombie, which is a process that has terminated but is still taking up space in the process table on the chance that a parent process will need to call *wait* to get its exit status.

In this sample program, we essentially copy the example we used for wait but now, instead of waiting for the child to die, the parent sleeps for 10 seconds and then exits. In reality, of course, you will probably have the parent do useful work, such as getting and processing input from a user or doing other task.

The child prints a message annoucing itself, sleeps for three seconds, and then annouces that it's exiting. The parent process gets the `SIGCHLD` signal when the child process exits, which causes the *catch* function to be called. This function prints a message that it detected the death of a child and prints its exit status. It then returns, at which point the parent exits. Note that the parent's *sleep* was interrupted.

```
/* signal3: parent creates a child and detects its termination with a signal */
/*  The child prints an "I'm the child message and exits after 3 seconds */
/* Paul Krzyzanowski */

#include <stdlib.h>     /* needed to define exit() */
#include <unistd.h>     /* needed for fork() and getpid() */
#include <signal.h>     /* needed for signal */
#include <stdio.h>      /* needed for printf() */

int
main(int argc, char **argv) {
        void catch(int);        /* signal handler */
        void child(void);       /* the child calls this */
        void parent(int pid);   /* the parent calls this */
        int pid;        /* process ID */

        signal(SIGCHLD, catch); /* detect child termination */

        switch (pid = fork()) {
        case 0:         /* a fork returns 0 to the child */
                child();
                break;

        default:        /* a fork returns a pid to the parent */
                parent(pid);
                break;
```

```
        case -1:        /* something went wrong */
                perror("fork");
                exit(1);
        }
        exit(0);
}

void
child(void) {
        printf("        child: I'm the child\n");
        sleep(3);       /* do nothing for 3 seconds */
        printf("        child: I'm exiting\n");
        exit(123);      /* exit with a return code of 123 */
}

void
parent(int pid) {
        printf("parent: I'm the parent\n");
        sleep(10);      /* do nothing for five seconds */
        printf("parent: exiting\n");
}

void
catch(int snum) {
        int pid;
        int status;

        pid = wait(&status);
        printf("parent: child process exited with value %d\n", WEXITSTATUS(status));
}
```

Download this file

Save it as `signal3.c` and compile it program via:

```
gcc -o signal3 signal3.c
```

Run the program:

```
./signal3
```

Example 4: detecting the termination of multiple child processes

Our final example is a small variation of the previous one. Instead of forking off one child, the parent will fork off four processes (or change NUMPROCS to whatever you want). As soon as the parent forks off a child, it loops back to create another one until it has created NUMPROCS of them. When a process is created, the variable `nprocs` is incremented to count the number of processes that we forked.

The parent then goes into a sleep loop:

```
while (nprocs != 0) {
        printf("parent: sleeping\n");
        sleep(60);      /* do nothing for a minute */
}
```

This looks like it will loop forever but two additional things are happening:

1. Whenever the parent gets the `SIGCHLD` signal and the *catch* funciton is called, the call to *sleep* is interrupted, so that *sleep* immediately returns. /li>
2. The signal handler, *catch*, decrements the count of child processes, `nprocs`, whenever it is called.

```
/* signal4: parent creates multiple child processes and detects
   their termination via a signal
   Each child prints an "I'm the child" message and exits after n seconds,
   where n is the sequence in which it was forked.

   Paul Krzyzanowski
*/

#include <stdlib.h>      /* needed to define exit() */
#include <unistd.h>      /* needed for fork() and getpid() */
#include <signal.h>      /* needed for signal */
#include <stdio.h>       /* needed for printf() */

#define NUMPROCS 4       /* number of processes to fork */
int nprocs;              /* number of child processes */

int
main(int argc, char **argv) {
        void catch(int);         /* signal handler */
        void child(int n);       /* the child calls this */
        void parent(int pid);    /* the parent calls this */
        int pid;         /* process ID */
        int i;

        signal(SIGCHLD, catch); /* detect child termination */

        for (i=0; i < NUMPROCS; i++) {
                switch (pid = fork()) {
                case 0:          /* a fork returns 0 to the child */
                        child(i);        /* child() never returns; the function exits */
                        break;
                case -1:         /* something went wrong */
                        perror("fork");
                        exit(1);
                default:         /* parent just loops to create more kids */
                        nprocs++;        /* count # of processes that we forked */
                        break;
                }
        }
        printf("parent: going to sleep\n");

        /* do nothing forever; remember that a signal wakes us out of sleeep */
        while (nprocs != 0) {
                printf("parent: sleeping\n");
```

```
                sleep(60);      /* do nothing for a minute */
        }
        printf("parent: exiting\n");
        exit(0);
}

void
child(int n) {
        printf("\tchild[%d]: child pid=%d, sleeping for %d seconds\n", n, getpid(), n);
        sleep(n);       /* do nothing for n seconds */
        printf("\tchild[%d]: I'm exiting\n", n);
        exit(100+n);    /* exit with a return code of 100+n */
}

void
catch(int snum) {
        int pid;
        int status;

        pid = wait(&status);
        printf("parent: child process pid=%d exited with value %d\n",
                pid, WEXITSTATUS(status));
        nprocs--;
        signal(SIGCHLD, catch); /* reset the siganl (for SunOS) */
}
```

Download this file

Save it as `signal4.c` and compile it program via:

```
gcc -o signal4 signal4.c
```

Run the program:

```
./signal4
```

Note: the *signal* system call has been replaced with a more robust version, called *sigaction*. If you plan on doing extensive programming using signals, be sure to read up on this. The *signal* that we're using is really a library wrapper over *sigaction*
