



Systems Programming in C— Processes and Signals

In this second article in the series, let us try to understand the concept of processes, loading a program and handling signals in the context of Linux-like operating systems. These concepts are implicitly used even by those who are not programmers, so understanding them will help you get more out of Linux.

A process is defined as a program in execution. We all use the *ps* or *top* commands to get a list of running processes and more information about them. But how do you create a process using the C language interface in Linux? Use the *fork()* system call, as in the *myfork.c* code below.

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

int global =20;
main() {
    int localvar = 30, ret;
    pid_t pid; /* See the Note below */
    ret = fork ();
    if (ret == 0) /*Child Code */ {
        global++; localvar ++;
        exit (0);
    }
    else /*Parent Code */ {
        printf ("My Child's PID =%d, My Id =%d\n", ret, getpid
());
    }
    printf ("My PID= %d, global=%d, local=%d\n", getpid (),
global, localvar);
    exit (0);
}
```



Note: C data-types are normally not used in order to avoid OS or architecture-specific implementation differences. Most data-types used in such programs are available as *typedefs* in the *sys/types.h* header file.

Now what will be the output of the above code? Two lines will be printed, by the two *printfs*, and the values of *localvar* and *global* will be 20 and 30, and not 21 and 31.

And why is this so? The *fork* call creates a new process, which is a copy of the parent. This child process runs independently in a separate address space. It starts executing, not from the *main* function, but after the *fork* call (*ret = fork()*). The *fork* function is different from other calls; it is called once, and returns twice—once in the parent, and once in the child.

Code in each process can distinguish which of the two it is by checking the return value—*fork* returns 0 to the child and a non-zero value (the child's process identifier) to the parent.

The statements incrementing the two variables are actually executed in the child, but since it runs in a separate address space, the variables are not shared. Thus, the last *printf* statement, executed by the parent process, uses the parent's copy of the variables, which are not incremented. So the output is 20 and 30.

The *fork* call is how processes are created and used by most programs. Let's look at a common example—executing an external program from a C program. See the *call_wc.c* code, which you can get from the archive at www.linuxforu.com. In that code, invoke the program *wc* via the function *execlp*; *wc* reads the file *myfile*. The final output will be the number of characters, words, and lines of *myfile*. This is the case when *execlp* is successful—you won't get the last *write* executed. (If *execlp* fails, then the line is executed.) The *execlp* function loads the executable *wc*, passing command-line arguments (*myfile*, *NULL*) to it. *NULL* here means a null pointer, which is used to mark the end of the argument array. The *main* function of a standard C program receives the number of command-line arguments in *argc*, and an array of pointers to individual arguments, represented as strings, in *argv*, as seen in the typical signature *int main (int argc, char *argv [])*.

Now what does *execlp* do? Briefly, it copies into memory the image of the '*wc*' program, overwriting the current process, and sets up the execution environment for the new program. Since the 'old' code (of the current process) no longer exists after a successful invocation of *execlp*, on successful completion, the function *never* returns. This is why you should not get the last-

line output in `call_wc.c`, if `execlp` succeeds.

When end-users run C programs from a Linux shell, the shell creates the argument array and invokes the loader—just as we have done earlier. In general, user-level commands that are run in a shell are classified into three types: external commands, internal commands and scripts. You can use the Bash command `type` to find what type a particular command is—`type ls` will show you that it is external, and `type cd` is a shell built-in. When the shell encounters an external command (like `ls` or `wc`), it calls `fork()` to create a copy of itself; and in the child process, it calls `execlp` with the external command (`wc`) as the argument, like we have seen above. (Internal commands should be implemented as function calls in the shell. For shell scripts, the loader checks the `#!` first line of the script and calls an appropriate interpreter.) Thus, even as ordinary users, we implicitly use `fork` and `exec` calls everyday!

Please note that the original system call that loads a program is `execve`. Its prototype is:

```
int execve (const char *filename, char *const argv [], char *const
envp []);
```

There are a number of library functions (`execl`, `execlp`, `execle`, `execv` and `execvp`) built on top of it. I have used `execlp` to simplify our discussion. In these function names, *l* means list, and *p*, path. For functions with *l*, you can provide arguments in the form of separate character pointers (as used above); for functions with *v*, arguments are an array of character pointers. Obviously, the second is more flexible, since normally you do not know how many arguments the user will enter. The letter *p* in the name means the function is expected to find the executable by searching a standard path list, normally the `PATH` environment variable, which has paths separated by a colon.

You can even change the environment variables available to the loaded program by using the `envp` parameter of the `exec` calls. Please refer to the manual page for more details.

More concepts: Zombies, orphans and *wait*

To quote Wikipedia, “When a process ends, all memory and resources associated with it are deallocated, so they can be used by other processes. However, the process' entry in the process table remains; this entry is needed to let the parent process read the child's exit status. When a (child) process completes execution, but its entry is still in the process table because its parent has not retrieved the child's exit status, the child is called a *zombie* or *defunct* process. If the parent executes the `wait` system call and collects the exit status of the child, the zombie entry is removed from the process table.”

An *orphan* process is one that is still executing, but whose parent has died. These do not become zombies; instead, they are adopted by `init` (process ID 1), which *waits* on them and reads their exit status.

How does a parent process *wait* for children to complete? See the `mywait.c` code, in the downloaded archive. Compile the code with `gcc mywait.c -Wall` (it's always preferable to compile

with `-Wall` to enable all warnings). Run it with arguments, like `./a.out 10 20` and you will get the output *My child returned 30*—as expected, which is the sum of the two arguments. The parent *waits* for the child to exit; when the child process terminates, the `wait` call returns, and the exit status of the child is stored in `exitstatus`. Later, it is printed using the macro `WEXITSTATUS`. This is because `wait` also updates the variable at the integer pointer (ours is `&exitstatus`) with more information, like whether the child process terminated abnormally, whether a memory image (core file) has been generated, and so on. To extract only the 1-byte return value (so only numbers up to 255 are possible), use the `WEXITSTATUS` macro.

We can check the return status of the last executed command by echoing the shell variable `$?` and this is implemented using `wait`.

The sample code in `myzombie.c` (in the downloaded archive) will create a zombie or defunct process. Compile the code, and run it in the background (`./a.out &`). Then run the `ps` command to see the defunct entry, like the following:

```
bash-3.00$ ./a.out&
[1] 24849
bash-3.00$ Before fork
```

```
bash-3.00$ ps
  PID TTY          TIME CMD
 17631 pts/1        00:00:00 bash
 24849 pts/1        00:00:00 a.out
 24850 pts/1        00:00:00 a.out <defunct>
 24856 pts/1        00:00:00 ps
bash-3.00$
```

Zombies will be cleared once the parent terminates, so when you write code for a server, you need to add code that removes zombies, since they can slow the system down. (Typically, you need to register a signal handler for the death-of-child signal, and take appropriate action when it happens. You will understand this after reading the signal section below.)

Signal handling

To understand signal handling, let's start with some questions.

Quite often, we use the `kill` command to terminate processes, like, `kill -9 processid`. What is `-9`, and what does `kill` do internally?

Consider trivial code like that below, which, when compiled and executed, runs in an infinite loop:

```
#include<stdio.h>
int main () {
    while (1);
}
```

Press `CTRL-C`, and notice that the process is terminated.

Look at faulty code like that below. The global variable `p` is initialised to 0, which is an invalid address—and you can try to assign a value 10, to store at this invalid memory address.

```
#include<stdio.h>
int *p;
int main () {
    *p=10;
    printf ("The value of p =%d", *p);
    return 0;
}
```

If you compile and run that, on most Linux systems, you will get a *Segmentation fault (core dumped)* error. How is this error output on screen?

In all three cases, some form of event notification (signal) to a process took place. In the first case, *kill* requested the OS to send a notification, denoted by the number 9, to a process. In the second, the user's *CTRL-C* terminated the process. In the third case, a memory access violation occurred. Basically, exceptional conditions are reported by *signals*, and these are some examples of such conditions. Other general examples can be the expiration of a timer, unexpected errors while executing a system call, or termination of a child process. Standard signals are defined for these types of events, like *SIGSEGV*, *SIGINT* and *SIGUSR1*, etc. Refer to the manual page (*man 7 signal*) for a list of signals.

When a process receives a signal, the action is performed by a *signal handler* function. Such functions are automatically linked to your code for actions in response to OS signals—such as to stop the process (the process will exist, but will not be scheduled), to terminate the process with a memory image saved to disk (called core dumped) and to ignore the signal. In the above examples, the *default signal handler* performed the actions like printing '*Segmentation fault (core dumped)*' (as a result of the *SIGSEGV* signal—memory fault); termination of the process by the *SIGKILL* signal (denoted by the *kill* parameter -9); or termination of the process by the *SIGINT* signal (*CTRL-C*).


So to write your own signal handlers, simply register your own function for a signal; the default signal handler is then no longer invoked, but your function is called when the specified signal is received by the process. Whenever a signal is delivered to the process, the code flow jumps to the signal handler, and after it executes, returns to the code where it was interrupted. For example, if you modify your earlier infinite-loop example to add a signal handler for *CTRL-C*, the resulting code is in the file *inctrl.c*. Compile and run it, and press *CTRL-C*. You will see that the process is not terminated, but prints the following output:

```
bash-3.00$ gcc inctrl.c
bash-3.00$ ./a.out
I have received 2
I have received 2
Quit (core dumped)
bash-3.00$
```

What I did was press *CTRL-C* twice, and then *CTRL-*. The *SIGINT* signal handler prevents *CTRL-C* from being useful—after each execution, control again returns to the *while* loop. The third time, I pressed *CTRL-* and the process was terminated with a core dump, since this is the default action for *CTRL-* and we have not put in a custom handler for that signal. You can use the *gdb* debugger to view the core file; you should find something like, '*Core was generated by ./a.out. Program terminated with signal 3, Quit.*' This means the program terminated due to Signal 3, which on my system is *SIGQUIT*, generated by my pressing *CTRL-*.

For another example, look at *chldsig.c* in the code archive. Compile and run it in the background, like *./a.out&* and you will see that the line '*Received death of Child Signal*' is printed 10 times. The *SIGCHLD* signal is delivered to a parent process on the death of a child process. On receiving *SIGCHLD*, your signal handler in the parent executes.

Please note that not all signals may be handled, like *SIGKILL*, *SIGSTOP*, etc. You may even ignore or block a signal. Improper handling can even cause race conditions in your code. Once you get familiar with the signal interface, you would probably like to use the *sigaction* function instead of *signal*—it gives you finer control.

In this article, we have learnt the basic concepts of process, loading programs, and signal handling. In the next article, we will look at Inter-Process Communication techniques like pipes, FIFO, message queues, etc. I hope you found this useful as an introduction, and will explore the topic a lot more. 

References

- [1] Bach, Maurice J: 'The Design of the Unix Operating System', PHI
- [2] Stevens, W Richard: 'Unix Network Programming: Interprocess Communication', Vol-2, 2nd Edition, Pearson Education
- [3] Stevens, W Richard and Rago, Stephen A: 'Advanced Programming in the Unix Environment', 2nd Edition, Pearson Education
- [4] Linux Manual Pages
- [5] You can download the source code of this article from http://www.linuxforu.com/article_source_code/july12/system_programming.zip

Acknowledgement

I would like to thank Tanmoy Bandyopadhyay and Vikas Nagpal for their help in reviewing this article.

By: Swati Mukhopadhyay

The author is having more than 12 years of experience in academics and corporate training. Her interest areas are Digital logic, Computer architecture, Computer Networking, Data Structures, Linux and programming languages like C, C++. For any queries or feedback, she can be reached at swati.mukerjee@gmail.com or <http://swati.mukhopadhyay.wordpress.com/>.