SE 350: Operating Systems Winter 2023

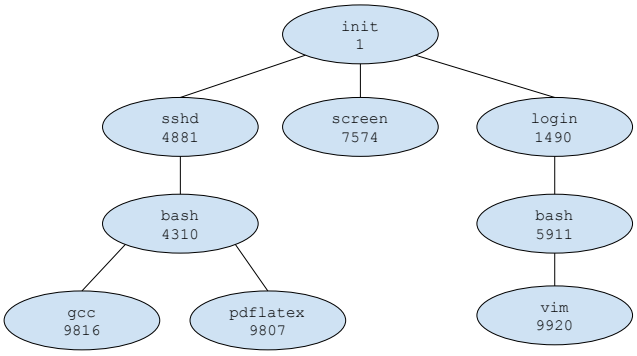
Lecture 5 — Processes in UNIX

Jeff Zarnett *2022-10-16*

The Process in UNIX

Earlier on, we mentioned that in UNIX, a process may create other processes. The creating process is the parent and the newly-created process(es) is (are) its child(ren). Every process has a parent, stretching back to the `init` process (or `launchd`) at the root of the tree.

Each process has a unique identifier in its process control block, and in UNIX we call this the `pid` (process ID). For the most part, users will not need to know or think about the ID of a process except when trying to terminate one that's gotten stuck (`kill -9 24601`). The `init` process always gets a pid of 1. I don't recommend trying to kill `init`. In most cases, `init` will just laugh off your attempt ("'tis but a scratch!") but you might end up rebooting the system or causing a crash.



A tree of processes in a Linux system.

In a UNIX system, we can obtain a list of processes at any time with the `ps` command. The diagram illustrates that each user, when logging in, spawns a `login` process (and an administrator can log people out by killing their `login` process... not that this is necessarily a good way to do it). The user's shell (these days, almost always `bash`, the Bourne Again Shell) is then spawned from `login`. That shell provides the command line interface where the user can enter a command.

When you issue a command, like `ls` or `top` (table of processes), the new process is created and the shell will `wait` on that process to finish (in the case of `ls`) or for the user to tell it to exit (`top`); when it does, control goes back to the shell and you get presented with the prompt again (e.g., `jz@Loki:~/$`). This would, on its face, seem kind of limiting – do I have to log in to the system in a second terminal window to run two things at a time? The answer is no, and there are two ways to get around it.

The first thing we can do is tell the shell we want the task to run in the background. To do that, add to the command the & symbol, like so:

```
gcc fork.c &
```

This will return control almost immediately to the shell (as it will not be waiting for the `gcc` command to finish). You may see some output like `[1] 34429` which is the shell saying the child has been created and it has process ID 34429. When the process is finished, there is another update, looking something like:

```
[1]+ Done gcc fork.c
```

Notably, any console output that the `gcc` command would generate will still appear on the console where the background task was created. Maybe you want that but maybe you want to put the output in a log file, with a command like `cat fork.c > logfile.txt &`. (Telling `gcc` to be silent is a somewhat more complex operation.)

A common example of a command I use involving the &:
`sudo service xyz start &`
This will (with super user permissions - that's the purpose of `sudo`) start up the service `xyz` but return control to the console so I don't have to wait for the `xyz` service to be started to enter my next command. This is good, because the next thing I'd like to do is `tail -f /var/log/xyz/console.log` which will allow me to watch the console log of the `xyz` service as it starts up to see if there are any errors.

The other alternative to get something to run in the background is with the `screen` command. While having something run in the background is nice, it does not work for interactive processes. Suppose you are working on some code in `vi` and you would like to pause that for a minute and write an e-mail (with `pine` or whatever the cool kids use for command line e-mail these days). One approach is to save and exit `vi` and open up `pine`. The other would be to start up each of these in `screen` and switch between them.

Thus instead of just opening `vi fork.c` I can issue the command `screen vi fork.c` and this spawns `screen` and takes me right to editing the file. The key difference is that I can "detach" from this screen and go back to the command line that spawned it. And if I log out, `screen` keeps running with the `vi` inside it. If I have multiple screens running, I can just "reattach" to the one I want to use next. To get a full understanding of `screen`, try the command `man screen` and the user manual will appear to give you some information and instructions about how to use this. Or you can use Google.

Show Me The Code!

The workflow in UNIX is as follows. First, the parent spawns the child process with the `fork` system call. If it is interested in waiting for the child process to finish, it will use the system call `wait`, in which case the parent will be awaiting the completion of the child process. When the child process is finished, it returns a value with the `exit` system call. The parent process will then get this as the return value of the `wait` call and may proceed.

What does `fork` do? It creates a new process; it makes a copy of itself. The parent and child continue execution after the `fork` statement. If `fork` returns a negative number, the `fork` system call failed. If it returns 0, the process that got the 0 back is the child. If it returns a positive value, that is the process ID of the child.

After the `fork`, one of the processes may use the `exec` system call, or one of its variants, to replace its memory space with a new program. There's no rule that says this must happen; a child can continue to be a clone of its parent if it wishes. The `exec` invocation loads the binary file into memory and starts execution [SGG13]. At this point, the programs can go their separate ways, or the parent might want to wait for the child to finish. The parent is then blocked, waiting for the child process to execute.

Let's put this all together in an actual C-code example adapted from [SGG13]:

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

int main( int argc, char** argv ) {
  pid_t pid;
  int childStatus;

  /* fork a child process */
  pid = fork();

  if (pid < 0) {
    /* error occurred */
    fprintf(stderr, "Fork␣Failed");
```

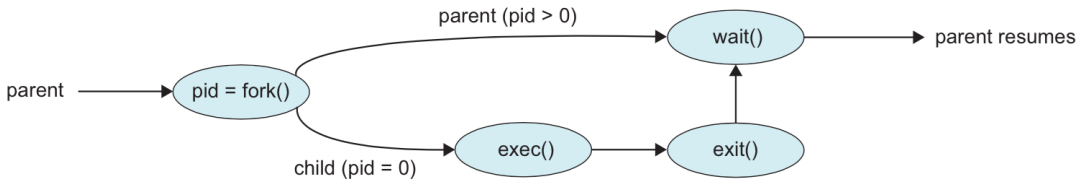
```
    return 1;

 } else if (pid == 0) {
    /* child process */
    execlp("/bin/ls","ls",NULL);

  } else {
    /* parent process */
    /* parent will wait for the child to complete */
    wait(&childStatus);
    printf("Child Complete with status: %i \n", childStatus);

  }

  return 0;
}
```

When executed, this code starts up and attempts to spawn a child process. Let us assume that the `fork` command succeeds and we do not enter the error-occurred block. After the fork there are now two processes at the statement `if ( pid < 0 )` . The child process calls `execlp`, replacing itself with the `ls` (list directory contents) command. The parent process will go to the `wait` statement and wait for the child process to complete. The child process runs `ls`, listing the contents of the directory. Then it finishes. The parent process, finally, prints "Child Complete" to the console.

Thus, the output is:

```
jz@Freyja:~/fork$ ./fork
fork   fork.c
Child Complete with status: 0
jz@Freyja:~/fork$
```

Or, to represent this visually:



Process creation with the `fork` system call [SGG13].

What about termination? On the assumption that the process is terminating normally and not being killed, the system call for that is `exit`. If the program itself has no explicit call to `exit`, the `return` statement at the end of `main` will have the same effect.

Use of Fork Design Problem. It's not necessary for a child to replace itself with another one. Let us do an example of how we might write a program where both parts are in the same source file.

There is a task that can be split into parts 'A' and 'B'. You may assume there are implementations for the functions `int execute_A()` and `int execute_B()` that work correctly (and are not shown). If either of the execution functions returns a non-zero value, this indicates an error.

Use `fork()` to create a child process. The child process should call function `execute_B()` and return the result to the parent. The parent process should call `execute_A()` and collect its result. The parent should then collect

the result of the child using `wait()` and then produce the console output described in the next paragraph. If no errors occurred, `main` should return 0; otherwise it should return -1.

If an error occurs, it should be reported to the console including the error number (e.g., "Error 7 Occurred."). If more than one error occurs, report both errors. If both functions return zero, it means all is well and the program should print "Completed." to the console.

Use of Fork Design Solution The solution:

```
int main( int argc, char** argv ) {

  pid_t pid;
  int child_result;
  int parent_result;

  pid = fork();

  if ( pid < 0 ) { /* Fork Failed */
    return -1;
  } else if ( pid == 0 ) { /* Child */
    return execute_B();
  } else { /* Parent */
    parent_result = execute_A();
    wait( &child_result );
  }

  if ( child_result == 0 && parent_result == 0 ) {
    printf( "Completed.\n" );
    return 0;
  }

  if ( child_result != 0 ) {
    printf( "Error %d Occurred.\n", child_result);
  }
  if ( parent_result != 0 ) {
    printf( "Error %d Occurred.\n", parent_result);
  }

  return -1;
}
```

The Fork Bomb

We will take a slight digression from the general topic of processes to discuss a denial-of-service attack against UNIX systems that is called the "Fork Bomb". The idea behind the attack is to call `fork` repeatedly until the number of processes spawned is too high for the system to manage and it either crashes or is so slow that no useful work can get done. As you can imagine, this is caused by repeatedly calling `fork`. Each time a program does so, there are now two processes running, each of which calls `fork`. There are then 2^n processes after each of n invocations and this exponential growth will soon crash up against the limits of the system.

A system configured to defend against this may impose limits on (1) the total number of processes a user may create; and (2) the rate at which a user may spawn a new process.

Note: do not attempt this on anything other than your personal computer at home. It is a denial of service attack and would certainly count as misuse of resources on any system. Accordingly, trying to do it will very likely result in a ban. Getting banned from using university computer resources is not conducive to completing your degree.

Signals

UNIX systems use signals to indicate events (e.g., the `Ctrl-C` on the console). Signals also are things like exceptions (division by zero, segmentation fault), etc. A signal may be *synchronous* if the signal occurs as a result of the program execution (e.g., dividing by zero); it is *asynchronous* if it comes from outside the process (e.g., the user pressing `Ctrl-C` or one process or thread sending a signal to another). Signals are, in the end, interrupts with a certain integer ID.

By default, the kernel will handle any signal that is sent to a process with the default handler. The behaviour of the default handler may be to ignore the signal, but some signals (segmentation fault) will result in termination of the process.

Here are some of the many signals described in the POSIX.1-1990 standard:

Signal	**Comment**	**Value**	**Default Action**
`SIGHUP`	Hangup detected	1	Terminate process
`SIGINT`	Keyboard interrupt (`Ctrl-C`)	2	Terminate process
`SIGQUIT`	Quit from keyboard	3	Terminate process, dump debug info
`SIGILL`	Illegal instruction	4	Terminate process, dump debug info
`SIGKILL`	Kill signal	9	Terminate process
`SIGSEGV`	Segmentation fault (invalid memory reference)	11	Terminate process, dump debug info
`SIGTERM`	Termination signal	15	Terminate process
`SIGCHLD`	Child stopped or terminated	20,17,18	Ignore
`SIGCONT`	Continue if stopped	19,18,25	Continue the process if stopped
`SIGSTOP`	Stop process	18,20,24	Stop process

Alternatively, a process may inform the operating system it is prepared to handle the signal itself (such as doing some cleanup when the `Ctrl-C` is received instead of just dying). In any event, a signal needs to be handled, even if the handling is to ignore it. Note that the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

On the command line, the command to send a signal is also`kill` followed by a process ID. Normally a command like `kill 24601` will send `SIGHUP` to a process, which will, by default, kill the process. The process has an opportunity to clean things up if it wants to. If the process is still stuck, you can "force" kill the process by sending `SIGKILL` with the command `kill -9 24601`. The `-9` parameter says to send signal 9 (`SIGKILL`) rather than the default 1 (`SIGHUP`). Some users are eager to jump to `kill -9` whenever a process is stuck, but it's usually worthwhile to attempt a less severe killing (`SIGHUP` or `Ctrl-C`) so that the process can at least try to clean up.

A signal is, interestingly, a form of inter-process communication: you are sending a signal from one process to another, even if it's your terminal doing the sending. When we have more than one process, we might want them to communicate, so we should look into how to make that happen.

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.