Lecture 5 — Processes in UNIX

Jeff Zarnett

`jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

March 5, 2019

In UNIX a process may create other processes.

The creating process is the parent; newly-created is the child.

Every process has a parent, stretching back to `init`.

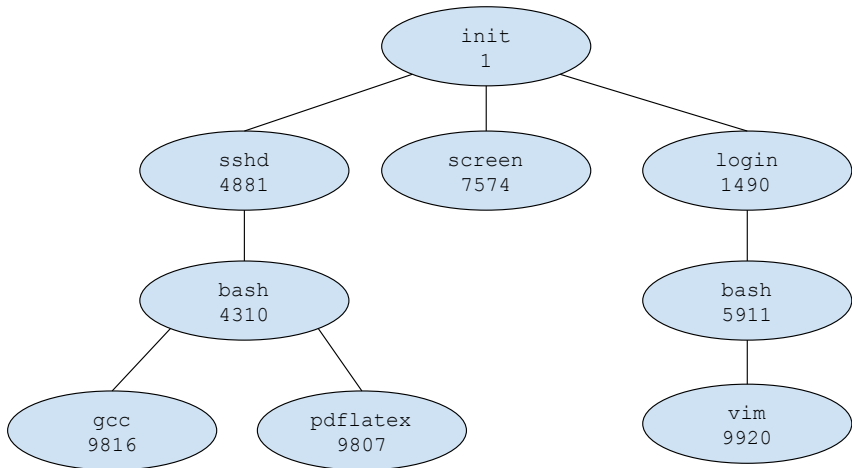Each process has a unique identifier in its process control block.

This is the pid (process ID).

For the most part, users will not need to know or think about the ID.

Exception 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.

We can obtain a list of processes with ps.

The diagram shows each user gets a login process.

The shell (bash) is spawned from login.

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.

It might finish on its own (e.g., `ls`).
  Or wait for the user to tell it to exit (`top`)

When it does, control goes back to the shell.
  You get presented with the prompt again (e.g., `jz@Loki:~/$`).

Must I 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.

Run in the Background

Option 1: tell the shell we want the task to run in the background.

To do that, add to the command the & symbol:

```
gcc fork.c &
```

Control returns almost immediately to the shell.
It is not waiting for gcc to finish.

You may see some output like `[1] 34429`.

This is the shell saying: child has been created; it has process ID 34429.

When the process is finished, there is another update:

```
[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) start up the service xyz.

It returns control to the console so I don't have to wait.

Next: `tail -f /var/log/xyz/console.log`

Watch the console log of the xyz service as it starts up.

The other alternative is the `screen` command.

While having something run in the background is nice, it does not work for interactive processes.

Example: text editing with `vi` and want to read e-mail with `pine`.

Could be done by saving and closing `vi`.

Or, start them in `screen` and switch between them.

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.

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.

Spawning Child Processes

In general, when a process spawns a child, the child will need resources.

The child may request them from the OS directly.

Or the parent can give some of its resources to the child.

The parent may partition resources amongst the children or allow its children to share.

Restrict a child process to only some subset of its parent's resources?

If so, cannot overload the system by spawning too many children.

At the time of initialization, the parent may pass the child some data.

Example: link from e-mail to browser.

Interesting note: child may be a duplicate or totally new.

Parent spawns the child process with the `fork` system call.

If waiting for the child process to finish, `wait`.
    Alternatively, carry on.

When the child process is finished, it returns a value with `exit`

The parent gets this as the return value of `wait` and may proceed.

Note: `fork` creates a new process as a copy of itself.

Both parent and child continue after that statement.

The call `fork` can return a value:
    A negative value means the fork failed.
    A zero value means this process is the child.
    A positive value: this is the parent; the value is the child `pid`.

After the `fork`, one of the processes may use the `exec` system call.

This will 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 a binary file into memory & starts execution.

At this point, the programs can go their separate ways.

Or the parent might want to wait for the child to finish.

```c
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;
}
```
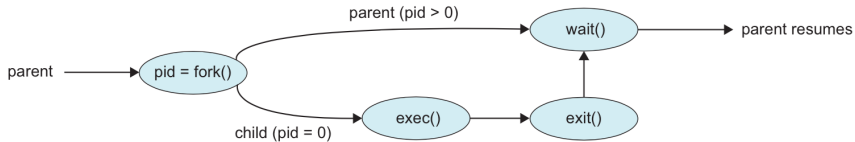
Thus, the output is:

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

Or, to represent this visually:

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.

Let us modify that code above to fork off a child process that will exit "abnormally" with an exit code of 1.

The wait function also returns the process ID of the child.

This is so that the parent can identify which of its children has terminated, though it is not used in this example.

Afterwards, the system will need to choose which process is going to run:

1. The parent process. The child is in the ready to run state.
2. The child process. The parent is in the ready to run state.
3. Another process. Both parent and child are in the ready to run state.

There is a task that can be split into parts 'A' and 'B'.

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.

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.

```c
int main( int argc, char** argv ) {
  int child_result;
  int parent_result;

  pid_t 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;
}
```

A short digression on a denial of service attack: the "fork bomb".

The idea is to call `fork` repeatedly.

Keep doing this until the system crashes (or no work can get done).

Exponential growth (2^n) processes after n calls.

A system can be configured to defend against this.

1. Limit total number of processes per user.

2. Limit rate of process spawning.

Note: do not attempt this on University computers!

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).

It is *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

A process may inform the OS it is prepared to handle a signal itself.

Example: doing some cleanup when 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.

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

On the command line: to send a signal, kill followed by a process ID.

Normally a command like kill 24601 will send SIGHUP to a process.

This 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 with SIGKILL:
  kill -9 24601.

The -9 parameter sends signal 9 (SIGKILL) rather than the default 1 (SIGHUP).

Some users are eager to jump to kill -9 whenever a process is stuck…