How to Launch A New Process (Step 1 of 2)

And a bit of armchair philosophy

Step 1: Clone your process.

```
pid_t fork(void);
```

Most things are cloned, e.g., memory (global vars, stack, heap, environment vars), current directory, file descriptor table, execution state,. . . Note: Both still running the same code!

Unlike sci-fi stories, Unix does not have a “who is the original” crisis! Each process has a process ID, assigned at birth and won’t change. The one with the old PID is the original, called “parent”; the one with the new PID is the clone, called “child”.

Both continue as though ‘fork’ returns. Child gets return value 0; parent gets return value = child’s PID (non-zero and not -1).

Sample code: fork-demo.c

How to Launch A New Process (Step 2 of 2)

Step 2: If wanted, the child can switch to running another program by one of the 'exec' family of system calls, e.g.,

```
execlp(path, arg0, arg1, ..., (char*)NULL)
```

Erased and replaced: code, data.

Preserved: PID, environment vars, current directory, file descriptor table (opt out individual FDs by marking "close on exec"), . . .

Why separate fork and exec:

- Some use cases don't need exec.
- Other use cases: Child can do interesting prep before exec—how file redirection and pipelining are done!

exec-demo.c

Digression: First Cause

More armchair philosophy

‘fork’ is the only way to launch new processes. All processes except one have parents. Who is the very first no-parent process?

In the beginning, kernel boots. After other preparations, kernel launches first process with PID 1, known as “init”. “init” launches system processes.

Some system processes are login screens, they exec into user shells or GUIs. User shells launch user processes upon user commands.

Useful Commands Concerning Processes

Useful programs to list your (and everyone's) processes:

- ps: List processes.
- pgrep: Find processes by name, users, etc.
- top: Periodically refreshed process list.
- htop: More features than top. (But not present on mathlab.)

Commands to force process termination (if you are allowed):

- kill: You need to provide specific PIDs.
- pkill: Find like pgrep, then kill.

See man pages! 'kill' is usually a shell built-in, try 'help kill' if bash.

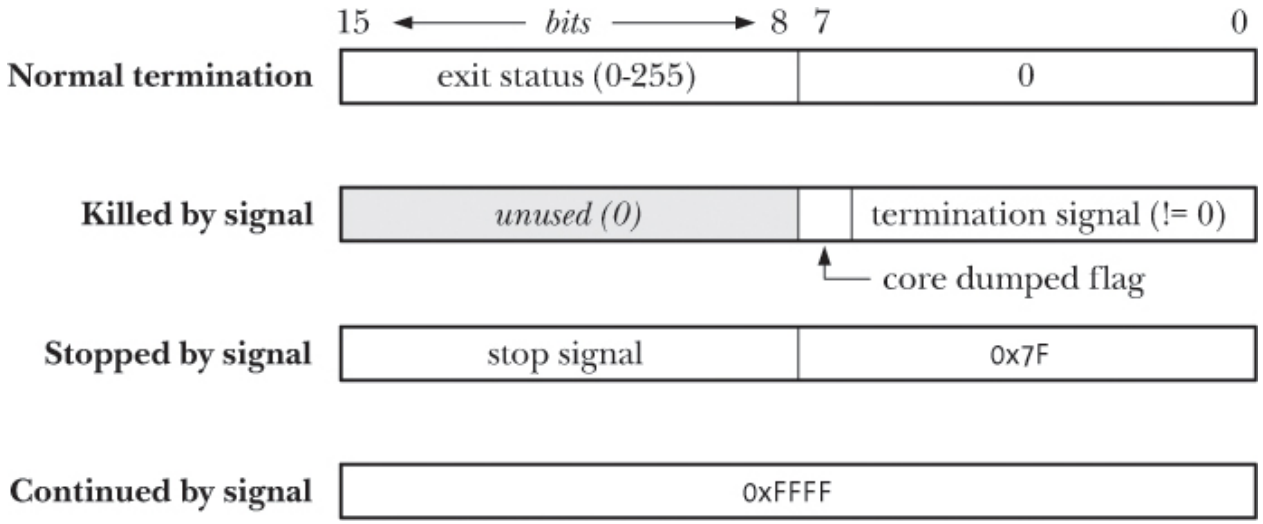
Waiting for Child

Wait for any child to terminate:
`pid_t wait(int *wstatus);`
`wstatus` is for child's exit code and/or why it died (e.g., killed).

To wait for a particular child, or don't hang waiting:
`pid_t waitpid(pid_t pid, int *wstatus, int options);`
`pid` > 0: Wait for particular child.
`pid` $= -1$: Wait for any child.
`options` = `WNOHANG`: Don't hang waiting. (Use 0 otherwise.)

(BTW after child terminates and after parent calls `wait`, parent can call `getrusage` to get child's resource usage.)

wstatus



Useful macros from sys/wait.h (see `man wait`):

- normal: `WIFEXITED(s)`, `WEXITSTATUS(s)`
  Example: wait-demo.c
- killed: `WIFSIGNALED(s)`, `WTERMSIG(s)`, `WCOREDUMP(s)`
- stopped, continued: `WIFSTOPPED(s)`, `WIFCONTINUED(s)`

Child Dies First vs Parent Dies First

Oliver Twist And Great Expectations And Zombies

If child dies, parent doesn't call 'wait', parent still running: "zombie", kernel still retains an entry in process table for child, but nothing is running. Process lists display "Z" for this.

Reason: Perhaps the parent will call 'wait' later.

If parent dies, child still running: "orphan", kernel resets child's parent PID to 1—'init' adopts child.

If child dies then: 'init' calls 'wait', no zombie!

If child is zombie, then parent finally dies: Adopt-wait combo move.

File Redirection How-To

- Before fork: Open the file. Let `fd` be the file descriptor. (If can't open, no point doing the rest.)
- Fork. Parent should close `fd`. Below are for child.
- `dup2(fd, i)` where `i` is 0, 1, or 2, as you need. Older programmers use `close(i)` then `dup(fd)`. Recall `dup` uses lowest-number free FD, which is `i` now.
- Close `fd` (or request close-on-exec when opening).
- Exec. The new program talks to `i` but that's the file you opened!

Sample code: file-redir.c

Pipes

`int pipe(int pipefd[2]);`
Creates a unidirectional pipe. `pipefd[0]` is FD for read end, `pipefd[1]` is FD for write end.

Exercise: What if you want two-way communication?

“Unnamed” pipe—no filename, does not exist in file system, even though you get file descriptors. The kernel performs internal data transfer when you write/read.

This is the pipe in “pipelines”. This is how shells do pipelining.

Pipelining example: piping.c

Pipe Hygiene

Usually just one process at write end, just another process at read end. Usually parent-child or siblings.

Usually also `dup`/`dup2` so stdin is read end, or stdout is write end.

Due to forking and dupping, intermediate state: multiple processes and FDs refer to write end, likewise for read end.

You should ASAP close the FDs you don't need. Reasons:

How does `read` know end of data (as opposed to no data yet) and return 0: Only when all write-end FDs are closed.

How does `write` know no audience: Only when all read-end FDs are closed.

Writer's Block And Broken Pipe

Sorry!

What if writer keeps writing, but reader reads too slowly or never?

Answer: Kernel has a buffer for unread data. But it's finite.

When filled up, the `write` call is blocked—hangs until the reader reads or closes the read end.

Writer's Block And Broken Pipe

Sorry!

What if writer keeps writing, but reader reads too slowly or never?

Answer: Kernel has a buffer for unread data. But it's finite.

When filled up, the `write` call is blocked—hangs until the reader reads or closes the read end.

What if the read end is closed: "Broken pipe".

Writer process receives a signal upon `write`. Discussed later, but by default process termination!

Reason: For example `yes | head -5`. `head` quits after 5 lines, no point running `yes` forever.

We'll discuss overriding this when discussing signals.

Non-Blocking Write/Read

Can request non-blocking I/O per file descriptor:

```
int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

(Could also set during ‘open’, but you don’t get your pipe from ‘open’ so meh.)

Then ‘read’ and ‘write’ won’t block, instead return -1 and set ‘errno’ to EAGAIN or EWOULDBLOCK.