Unix Signals

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae

Process Groups and Job Control

We start a long-running pipeline in a shell:

```
$ proc1.sh | proc2.sh  # we don't get the shell back while this runs
```

How can we do other work?

Process Groups and Job Control

We start a long-running pipeline in a shell:

```
$ proc1.sh | proc2.sh  # we don't get the shell back while this runs
```

How can we do other work?

Today: Use a modern terminal and open another tab/window/SSH connection, use a tmux (terminal multiplexer) session, etc.

Past: Use job control to put pipeline in the background and bring your shel
back to the foreground

Process Groups and Job Control

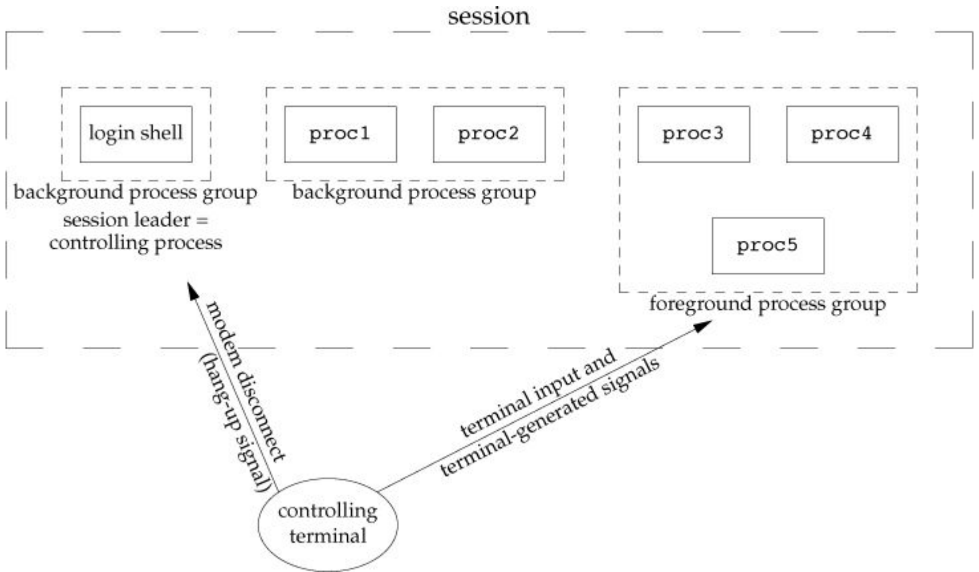
```
$ proc1 | proc2 &   # send pipeline to background
```

```
[1] 7106
```

```
$ proc3 | proc4 | proc5  # we have our shell, start another pipeline
```

`[1] 7106` refers to the job# and leading pid of the backgrounded pipeline. More job control:

- `jobs`: List all jobs
- Ctrl-Z: Suspend foreground job and send to the background
- `bg <job>`: Resume `<job>` in the background
- `fg <job>`: Bring backgrounded `<job>` into the foreground



Sending Signals

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

- Both return: 0 if OK, -1 on error
- If `pid < 0`, the signal is sent to the process group with `pgid == | pid |`
- If `pid = -1` ?

Terminal-generated signals

- Ctrl-C sends `SIGINT` to foreground process group
- Ctrl-\ sends `SIGQUIT` to foreground process group
- Ctrl-Z sends `SIGTSTP` to foreground process group

signal()

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Sets disposition of signum to handler, where handler can be:

- SIG_IGN: ignore the signal
- SIG_DFL: take the default action associated with the signal (see man 7 signal)
- a handler (function) of type sighandler_t: handler(signum) called to handle signal

Show portability issues between Mac and Linux for read1

Unreliable Signals: `read()`

What happens if a “slow” system call is interrupted by a signal?

- Slow underlying `read()` syscall gets interrupted. `errno` set to `EINTR`, causes `fgets()` to return `NULL`
- Hotfix?

Unreliable Signals: `read()`

What happens if a “slow” system call is interrupted by a signal?

- Slow underlying `read()` syscall gets interrupted. `errno` set to `EINTR`, causes `fgets()` to return `NULL`
- Hotfix: check `EINTR` and restart the syscall
  - …but this is annoying, most of the time we want the syscall to be restarted
  - need a way to indicate that slow syscalls should be restarted for us

Unreliable Signals: `read()`

What happens if a "slow" system call is interrupted by a signal?

- Slow underlying `read()` syscall gets interrupted. `errno` set to `EINTR`, causes `fgets()` to return `NULL`
- Hotfix: check `EINTR` and restart the syscall
  - …but this is annoying, most of the time we want the syscall to be restarted
  - need a way to indicate that slow syscalls should be restarted for us

Signals get lost

- Disposition set with Linux `signal()` resets after each signal
- Hotfix?

Unreliable Signals: `read()`

What happens if a "slow" system call is interrupted by a signal?

- Slow underlying `read()` syscall gets interrupted. `errno` set to `EINTR`, causes `fgets()` to return `NULL`
- Hotfix: check `EINTR` and restart the syscall
  - …but this is annoying, most of the time we want the syscall to be restarted
  - need a way to indicate that slow syscalls should be restarted for us

Signals get lost

- Disposition set with Linux `signal()` resets after each signal
- Hotfix: Set disposition again after detecting `EINTR`
  - ...but there's still a race condition: what if we get another signal before we set disposition?
  - need a way to indicate NOT to reset disposition

Reentrancy Issues

Can’t call certain function in asynchronous contexts

- Functions that use static data structures, `malloc()`, `free()`, standard I/O functions are unsafe!
  - Why is `printf()` not async-signal-safe?
  - Hint: recall std-io buffering (see also: `man 7 signal-safety`).
- Calling such functions in async manner could cause data corruption
- Check `man 7 signal-safety` for async-signal-safe functions

alarm()/pause()

```c
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
        // Returns: 0 or number of seconds until previously set alarm
int pause(void);
        // Returns: -1 with errno set to EINTR
```

`alarm()`: generate `SIGALRM` after `seconds`

`pause()`: suspend program execution indefinitely

Issues: check `sleep.c`

Portable Solution: `sigaction()`

See `sigaction.c`

An installed action stays installed until otherwise changed with `sigaction()`

`sigset_t sa_mask`: additional signals to block while `signo` is being handled with `sa_handler` → `signo` is blocked for you while in `sa_handler`

`int sa_flags`: handling options – some notable ones:

- `SA_INTERRUPT`: Don't automatically restart slow system call (default, there may not be a flag)
- `SA_RESTART`: Automatically restart slow system call
- `SA_NODEFER`: Don't block `signo` while in `sa_handler`
- `SA_RESETHAND`: Reset disposition of `signo` to `SIG_DFL`

More signal management

`sigprocmask()`: manipulate a process’s signal mask

`sigpending()`: retrieve a set of pending signals that are blocked from delivery

`sigsuspend()`: atomic `sigprocmask(SIG_SETMASK, ...)` + `pause()`, restores previous mask on interrupt