# CSSE2310 — 4.2

Processes

# Abstraction

A process is:

- An instance of a program in execution.
  - There is one `/opt/local/bin/bash` on moss.
  - There could be many executing instances of bash
- An abstraction of a computer
  - Interactions (Eg with files) are via system calls to the kernel.
  - Other processes' resources are not visible.
  - Processes' memory is [1] not accessible by other processes
  - Other processes' cpu activity shouldn't influence your's.
  - Indirectly aware of other non-related processes
    - Eg Files on the filesystem.

[1] generally

# How?

Separating:

- Resources (eg open files)
  - Each process can have their own table of resources.
  - At lower level, files are identified by integers.
- Memory
  - Virtual memory (next week)
- CPU activity (From Week 4)
  - Whenever a cpu switches to kernel mode, registers are saved.
  - (possibly much later) when the process is put on the cpu, registers are restored.
  - The process is none the wiser.

# Process states

From the kernel point of view, a process could be in one of the following states:

- Running — The process is currently executing on the CPU
- Ready — The process could run but isn't (something else has the CPU).
- Blocked — Process is not ready because it is waiting for something.
  - eg: waiting to read from a file
  - sleep?
- Ended(?) — Process has exited or terminated and needs to be cleaned up
  - May have become a zombie

# The circle of life

Fork discussion

# fork()

`fork()`

- Asks the kernel to create a new (child) process
- Called in one process (the parent)
- Returns in two processes (two processes remember calling it).
- Parent gets the PID of the child.
  - or if no child was created (and sets errno).
- Child gets 0.

```
pid_t pid = fork();
if (pid) {
    // parent
} else {
    // child
}
```

Memory aid: A parent can have many children (so it needs to know who is new). The child process only has one parent (and can `getppid()`).

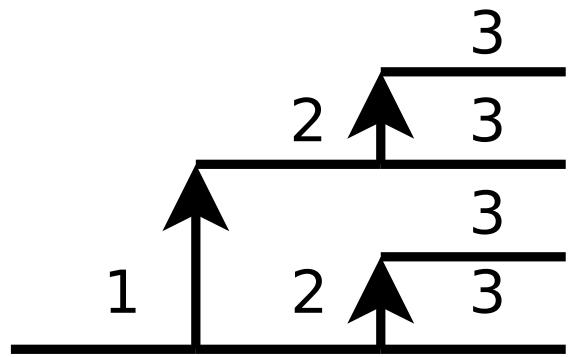
# Memory starts as a copy

```
void X(void) {
    int p=5;
    if (fork()) {
        ++p;
    } else {
        --p;
    }
    printf("%d\n", p);
}

int main(int argc, char** argv) {
    X();
    printf("Done");
    return 0;
}
```

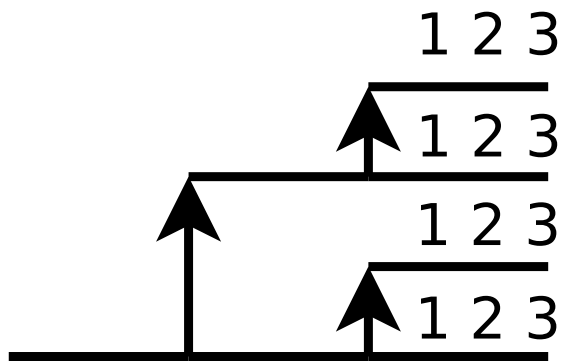
See `f1.c`

Child processes are processes and can fork as well.
See `f2.c`.



# Watch out for buffering

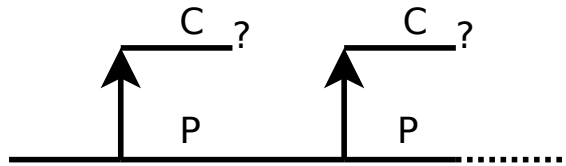
See `f3.c` vs `f4.c`.



# More care

The parent process could fork to make workers

See `f5.c`



# Hmmm

```
for (int i=0; i<20; ++i) {
    if (!fork()) {
        printf("Child\n");
    }
}
```

# fork bomb

The example on the previous slide will terminate eventually. Not always the case.
Difficult to stop:

- Limited number of simultaneous processes per user.
  - `ulimit -u` on bash
- Use `kill` or `pkill`?
  - In most cases[2] that needs to fork again.
  - Which you can't do
  - How do you know which process IDs to kill?[3]
- Can you do anything quickly enough?
  - Your useful programs will be competing with all those clones.

[2] Unless your shell has a built in `kill` (bash doesn't)
[3] `kill -9 -1`?

# Fork bomb consequences

Depends on the system configuration. On moss:

- May slow *your* processes down.
- *Probably* won't affect other users
- "But moss is being slow"
  - Moss almost always I/O bound.
  - Unless sysadmins say otherwise, Someone else fork bombing themselves is unlikely to be causing problems for you.

If you manage to do this to yourself.
And it doesn't die down on its own. Contact the helpdesk.

# Ending a process

`exit()` is a system call which ends the *current process*. ie you can't use it to end a different process.
Useful things:

- The parameter to exit is the "exit status" of the process.
- Any open output streams are flushed
- exit hooks are executed
  - `int atexit(void (*function)(void));`

Don't want those things? (eg if you think you have inherited unflushed buffers)

- `man _exit()`

Program crashes (or receives a signal[4])

- `exit` doesn't happen and there is no exit status.

[4] Signals are later