CSc 352

Processes

Russell Lewis (sub for Benjamin Dicken)

What is a Process?

- A **process** is a container for one instance of a program
  - If you run the same program multiple times, you get multiple processes

- A process is **isolated from other processes** on the same machine
  - Can’t access their memory
  - Can’t call their code
- A process is also isolated from **system resources**
  - No direct access to disk, network, keyboard, display
  - No ability to control other processes

The Simplest Virtual Machine

- Each process acts as if it is the *only program running on the machine*
  - Gobbles up CPU
  - Uses memory blindly

- Operating system's **"kernel"** makes this safe
  - Time-slicing
  - Virtual memory
    - Declares which addresses are allowed
    - Controls how to interpret the valid addresses

System Calls

- A **system call** is a way for the program to interact with the OS kernel
  - Not a function call, but often acts like one

- Through system calls, programs can:
  - Access files
  - Access devices
  - Talk on the network
  - Ask for virtual memory changes
  - Create / monitor / kill other processes
  - Ask the kernel to do *anything you can imagine*

Example System Calls

- Commonly used system calls:

  `open()/close()` - file handling

  `socket()` - first step in networking

  `chdir()` - change directory

  `pipe()` - prepares for pipelined processes

  `fork()/exec()` - create new programs

  `mmap()/munmap()` - change virtual memory

  `signal()/sigaction()` - send/receive “signals” between procs

- `man syscalls` - Linux has 100s of syscalls!

PID, PPID

- Each process has a **PID (process ID)**
  - Positive integer, usually small
  - Assigned when the process is created, never changed

    `getpid()` - returns the PID of your current process

- Each process has exactly one **parent process**
  - Stays the same forever, unless the parent dies (then set to 1)
  - Parent has special rights to control the process

    `getppid()` - returns the parent PID

ps

- The command-line tool `ps` shows the list of current processes
  - `ps` - Show processes related to current shell
  - `ps -f` - Also show PPID, other details
  - `ps -ef` - Show all processes on current box

ps

- The command-line tool `ps` shows the list of current processes
  - `ps` - Show processes related to current shell
  - `ps -f` - Also show PPID, other details
  - `ps -ef` - Show all processes on current box
- Let's log onto Lectura and do some **experiments together:**
  - Compare `ps` from different logins (including 2 from the same person)
  - Use `ps -ef` with `grep` to get information about one use
  - Lots of students run the same command - and we'll use `grep` to find all of them

`top`

- `top` is a handy tool for examining the current load on your machine
  - Run without arguments

- **Let's experiment!** Inside top, what do each of these commands do?

  `h` - ???

  `M` - ???

  `u` - ???

  `s` - ???

  `q` - ???

`top`

- `top` is a handy tool for examining the current load on your machine
  - Run without arguments

- **Let's experiment!** Inside top, what do each of these commands do?

  `h` - help

  `M` - sort by memory, not CPU usage

  `u` - limit list to a single user (type username)

  `s` - control update speed (type # o seconds)

  `q` - quit

`kill`

- `kill` allows you to terminate a running process
  - `kill PID` - Ordinary kill, works most times
  - `kill -9 PID` - Use only in emergencies

- In truth, `kill` is just sending a signal to a process
  - Signals are 1-bit messages or event notifications
  - `kill` sends SIGTERM to ask a process to terminate itself

- Can use `kill` to send any signal to a process
  - `kill -USR1 PID` - Sends USR1 to the process

`kill()` syscall

- `kill()` syscall is how you send signals from inside C code

```
int kill(pid_t pid, int sig);
```

- It isn't uncommon for some simple syscalls to have wrappers that are command-line utilities

`kill`, `wait`, `chroot`, `chown`, `chmod`, `mknod`, etc.

- Different man pages for the syscall and the command-line tools

`man 2 kill` - syscall

`man 1 kill` - command-line tool

`wait`

- `wait` blocks until a child of the current process has completed
- Useful if you kick off something in the background with &

```
sleep 10 &
wait
```

Creating Processes with the Shell

- What happens when you run a command on the shell?

```
ls -al foo bar baz
```

1. `bash` searches for the program (using your PATH variable)
2. New process created with `fork()`
3. Child process uses `exec()` to run the `ls` command
4. Parent process uses `wait()` to block until the child finishes

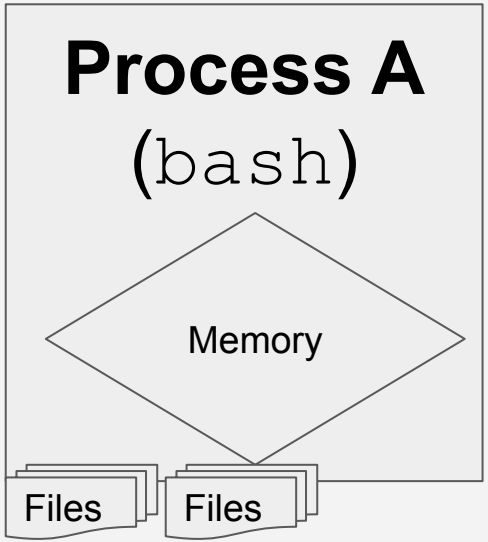
- How to run in the background (with `&`)? We simply skip step 4!

Why `fork()` and `exec()`?

- Why a two-step process for creating a new program?
  - `fork()` - creates the new process
  - `exec()` - replaces the process with a new program

- To understand this, let's start by looking at how `fork()` works…

`fork()` in Action



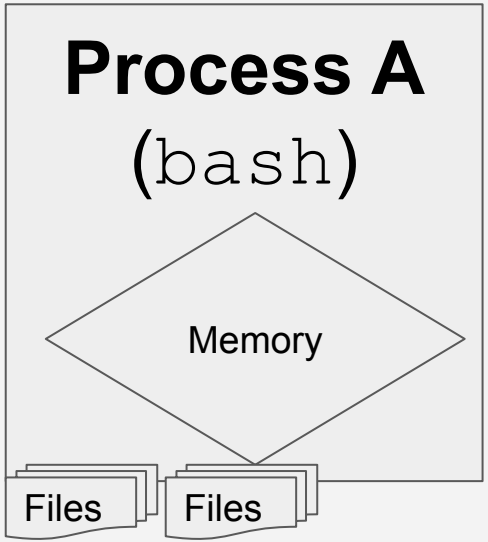
`bash` is running.

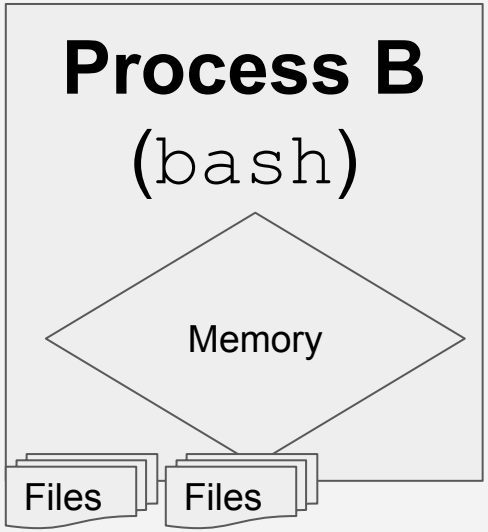
It has some memory (code+data).

It also has some open files.

(Actually, there is a **ton** of process state, that we won't detail here.)

`fork()` in Action

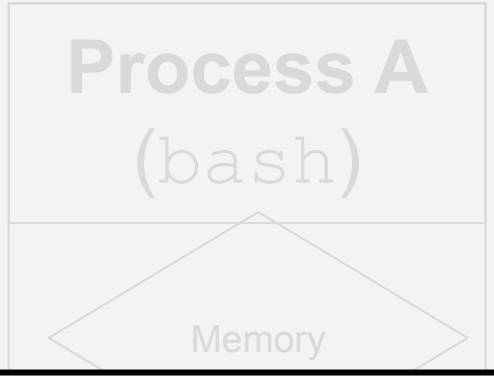


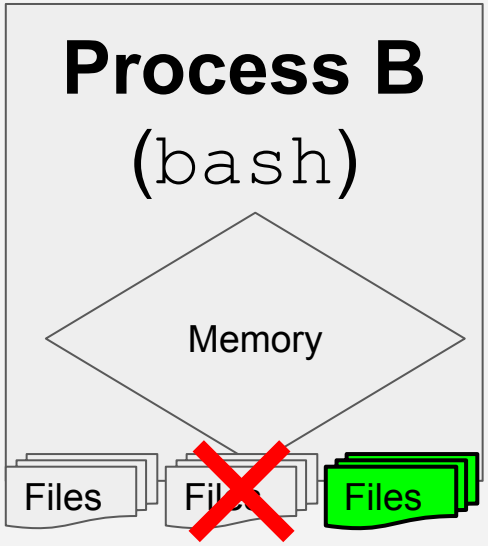


`bash` calls fork().

The OS creates a new process. It is **identical** to the old.

`fork()` in Action

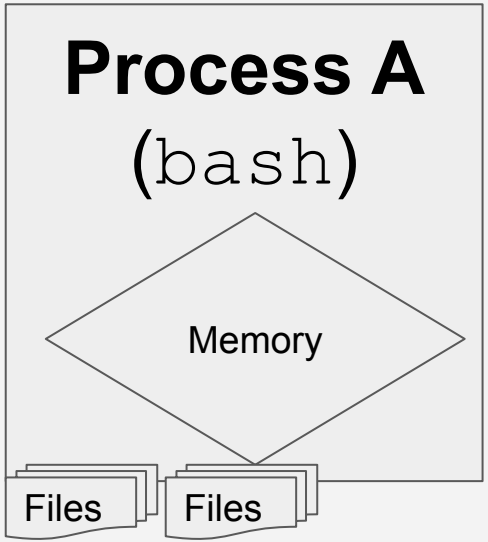


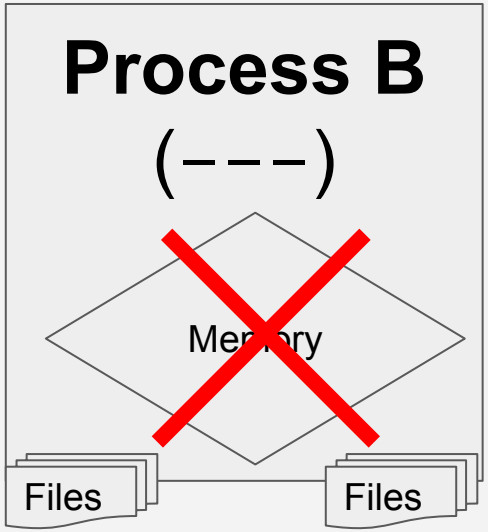


`bash` can now close some of the files, and open new ones. (This is how we handle redirection, and pipelines.)

In general, bash can change many things about the process.

`fork()` in Action

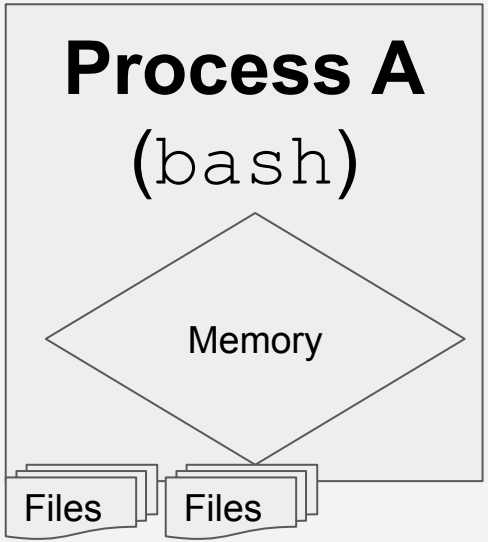


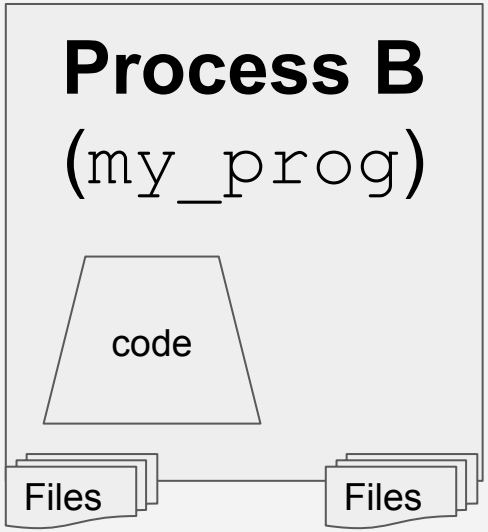


`bash` (in the child) calls `exec()`.

The memory in the child process is thrown away…

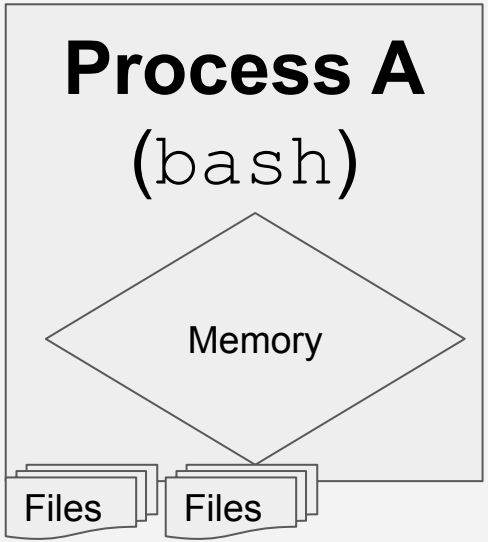
`fork()` in Action

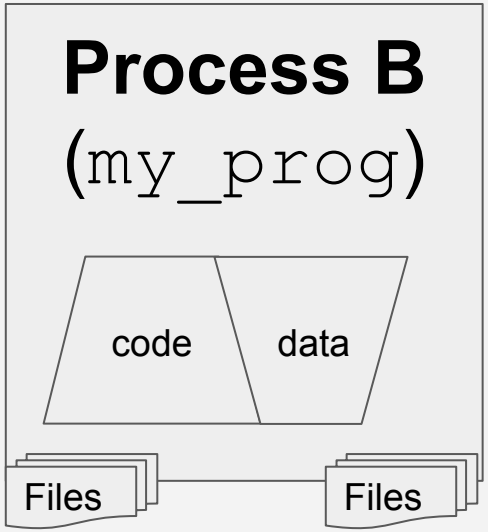




…the code for my_prog is loaded into the child process memory…

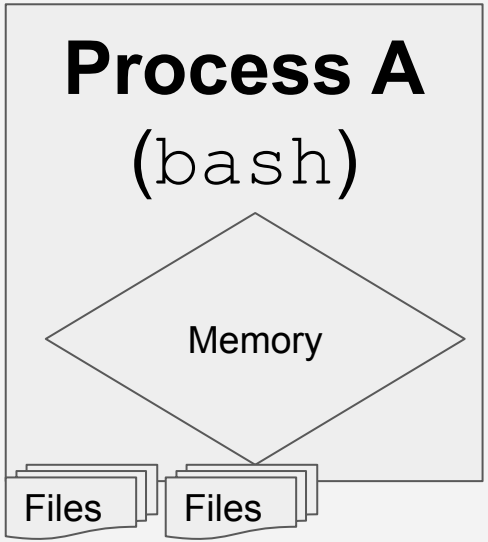
`fork()` in Action

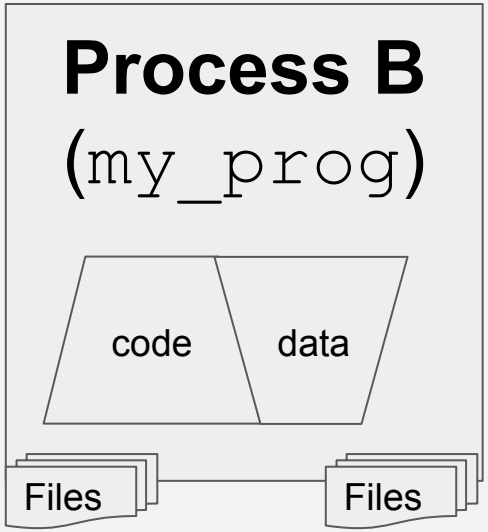




…the data is initialized, and the new program starts to run.

`fork()` in Action





Note that exec() did **not** change the files (or many other things about basic process state).

This is how bash sets up a new proc.

Pipelines

- What if there are multiple processes in a pipeline?

```
ls -al | grep foo | cut -f3 -d' '
```

- One child process per program in the pipeline
- Use `pipe()` syscall to create pipelines from each to the next
- `bash` waits on the *last* in the pipeline to finish

Why `fork()` and `exec()`?

- `fork()` and `exec()` are separate operations
  - Allows the code to modify the child process before the new code takes over
  - Vastly simplifies the parameters to `exec()` !