

# Engineering Robust Server Software

## UNIX Daemons

# Test This Technology

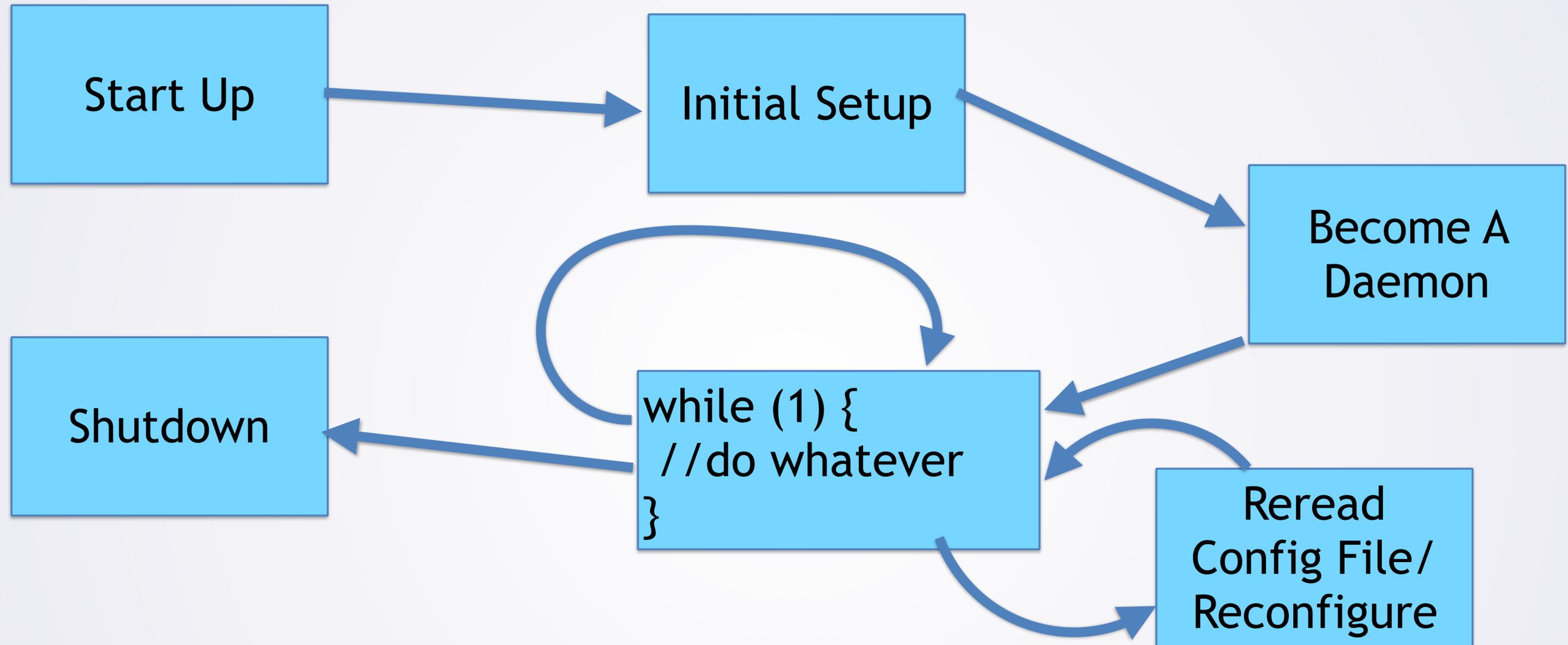
- Which of the following is answer "B"?
  - **A:** Clearly Wrong
  - **B:** This is the right answer. Pick me!
  - **C:** Kind of close
  - **D:** Right shape at least?

# Daemons

- Daemons: system services
  - Generally run startup -> shutdown
  - In the "background" no controlling tty
    - No stdin/stderr/stdout!
- Convention: names end in **d**
  - sshd, httpd, crond, ntpd, .....

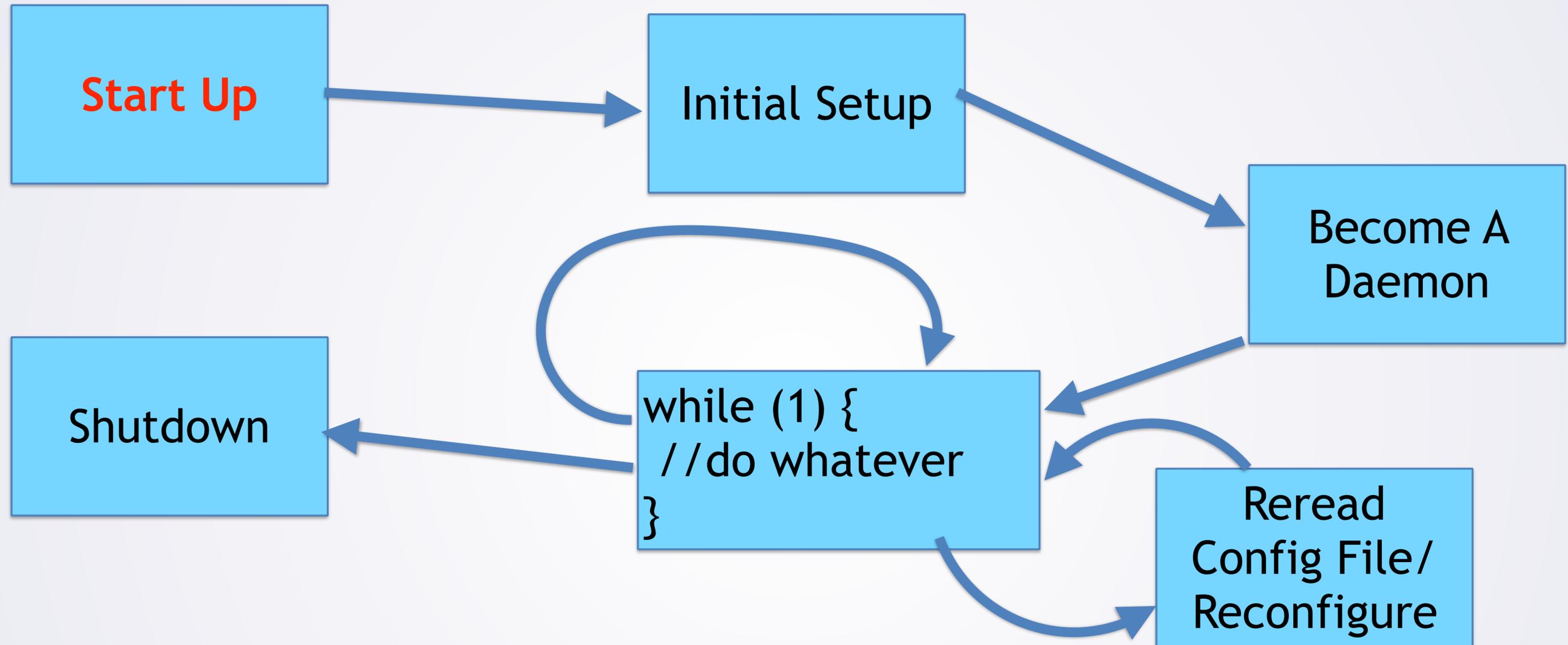


# Life Cycle



- General "life cycle" of a UNIX Daemon

# Life Cycle



- Often: started at system startup
  - Could also be started manually

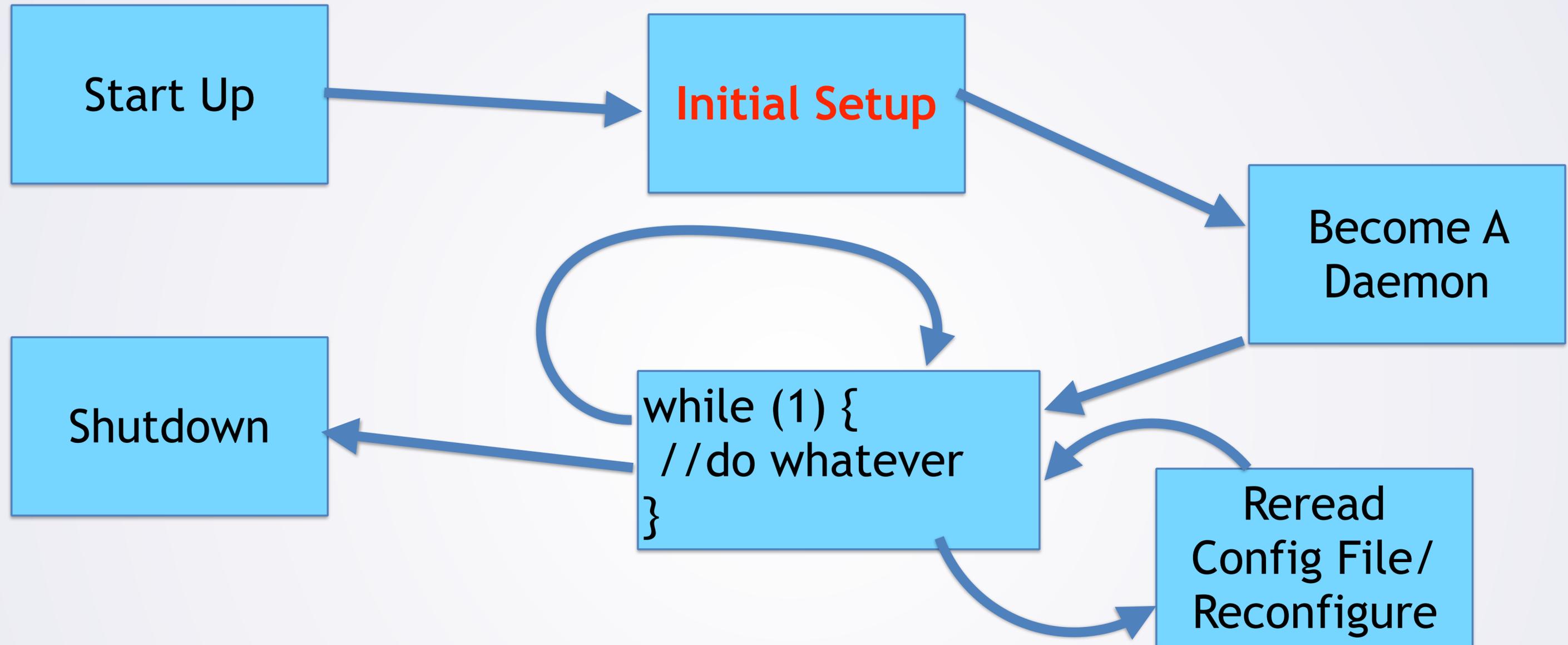
# System Startup

- Review:
  - Kernel spawns init (pid 1)
  - Init (on Linux, now "systemd") spawns other processes
- Init itself is a daemon
  - Reads config, runs forever,...
- Init's config files specify how to start/restart/stop system daemons
- Details depend on specific version
  - E.g., systemd is different from original init

# Init/Systemd Config

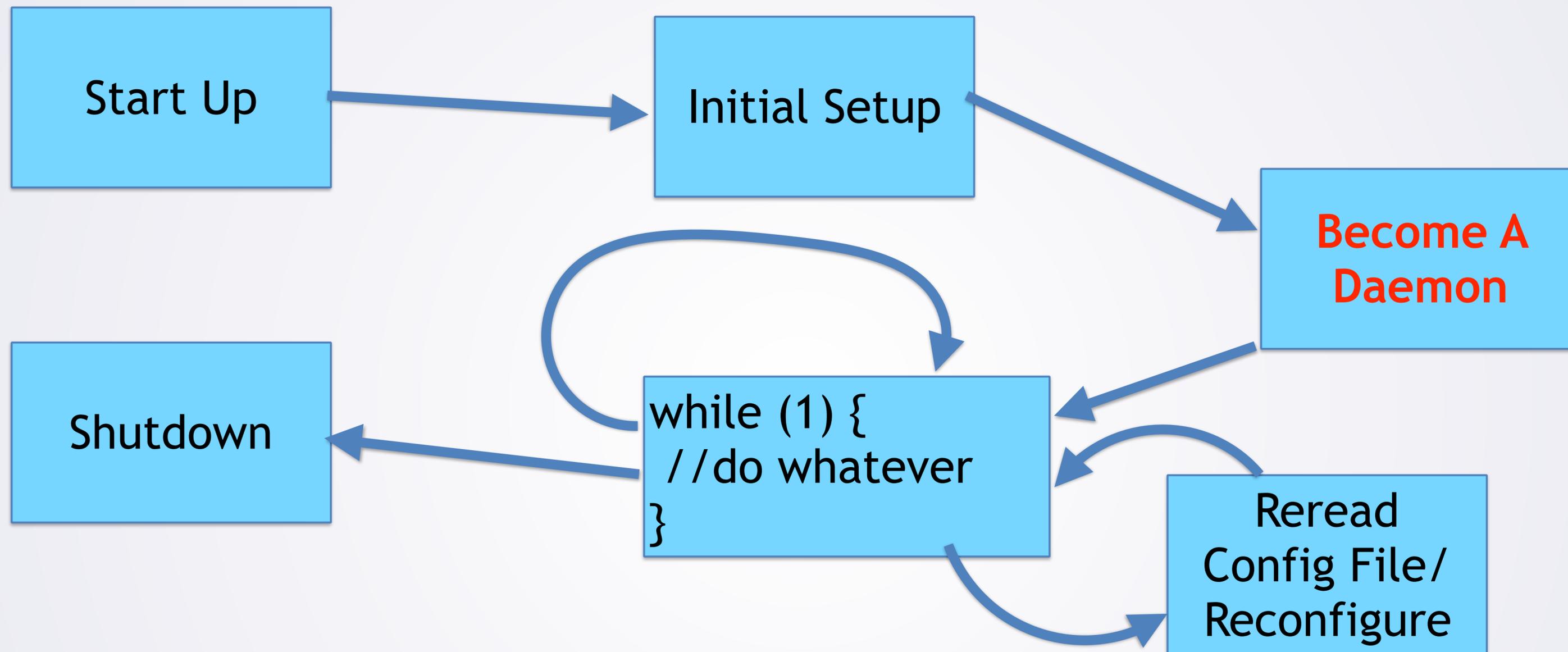
- Old way:
  - Numbered shell scripts, done in order
- Systemd (newer) way:
  - Units with dependencies
  - [https://access.redhat.com/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/7/html/System\\_Administrators\\_Guide/sect-Managing\\_Services\\_with\\_systemd-Unit\\_Files.html](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/sect-Managing_Services_with_systemd-Unit_Files.html)
- Can manually start/restart/status etc with **systemctl**
  - Can also control whether started automatically at boot

# Life Cycle



- Daemon may wish to do some setup while "normal" (\*) process
  - Read config files, open log files, bind/listen server socket, etc.
  - (\*)Some aspects of "normal" may be overridden by system

# Life Cycle



- A bunch of stuff has to happen to correctly run as a daemon
  - Requires introducing some new concepts

# Think, Pair, Share

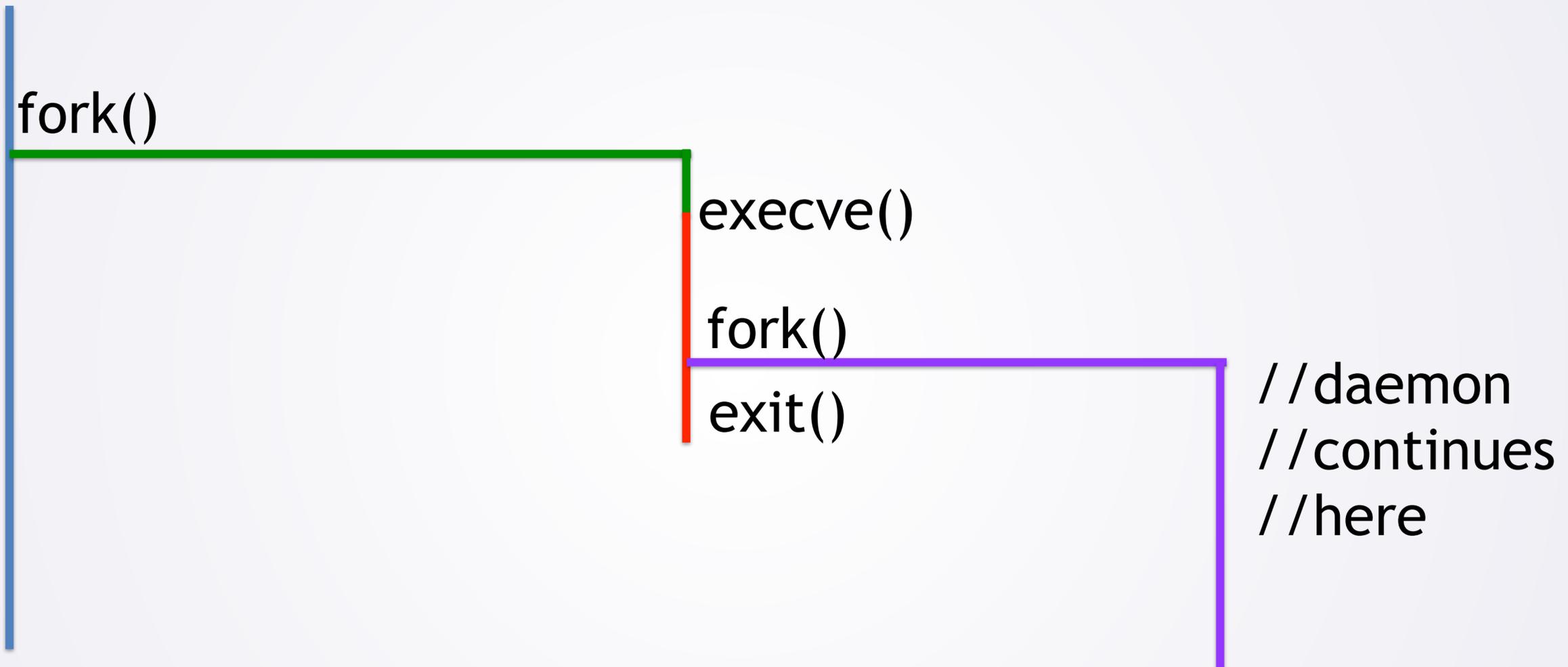
- What is special about daemons?
- What are the implications of these characteristics?
- How would you set things up?

# Becoming a Daemon

- Typically Required:
    - `fork()`, parent exits
    - Dissociate from controlling tty
    - Close `stdin/stderr/stdout`, open them to `/dev/null`
    - `chdir` to `"/"`
  - Good Ideas:
    - Clear `umask`
    - `fork` again -> not be session leader
- 
- daemon library call

# Becoming a Daemon

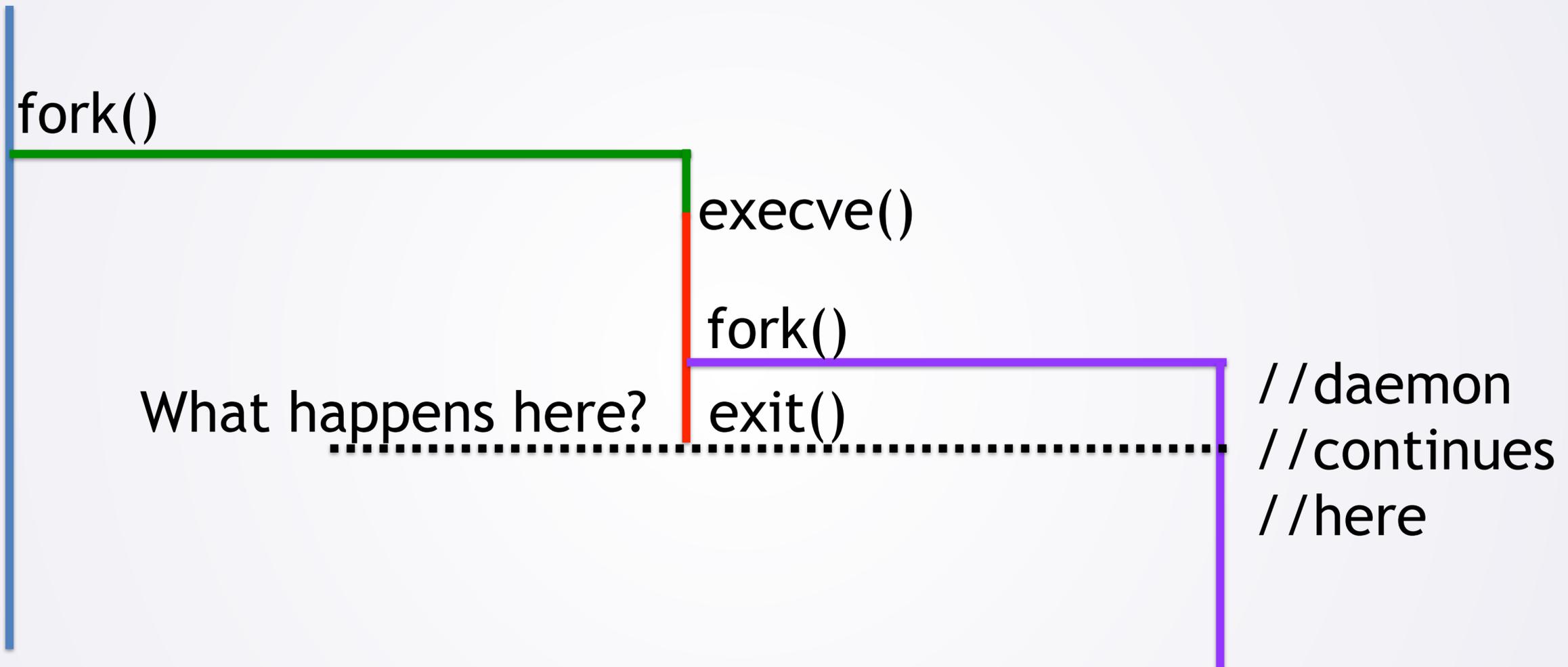
Whatever ran  
the daemon



- fork(), parent exits
  - Why?

# Becoming a Daemon

Whatever ran  
the daemon



- fork(), parent exits
  - Why?

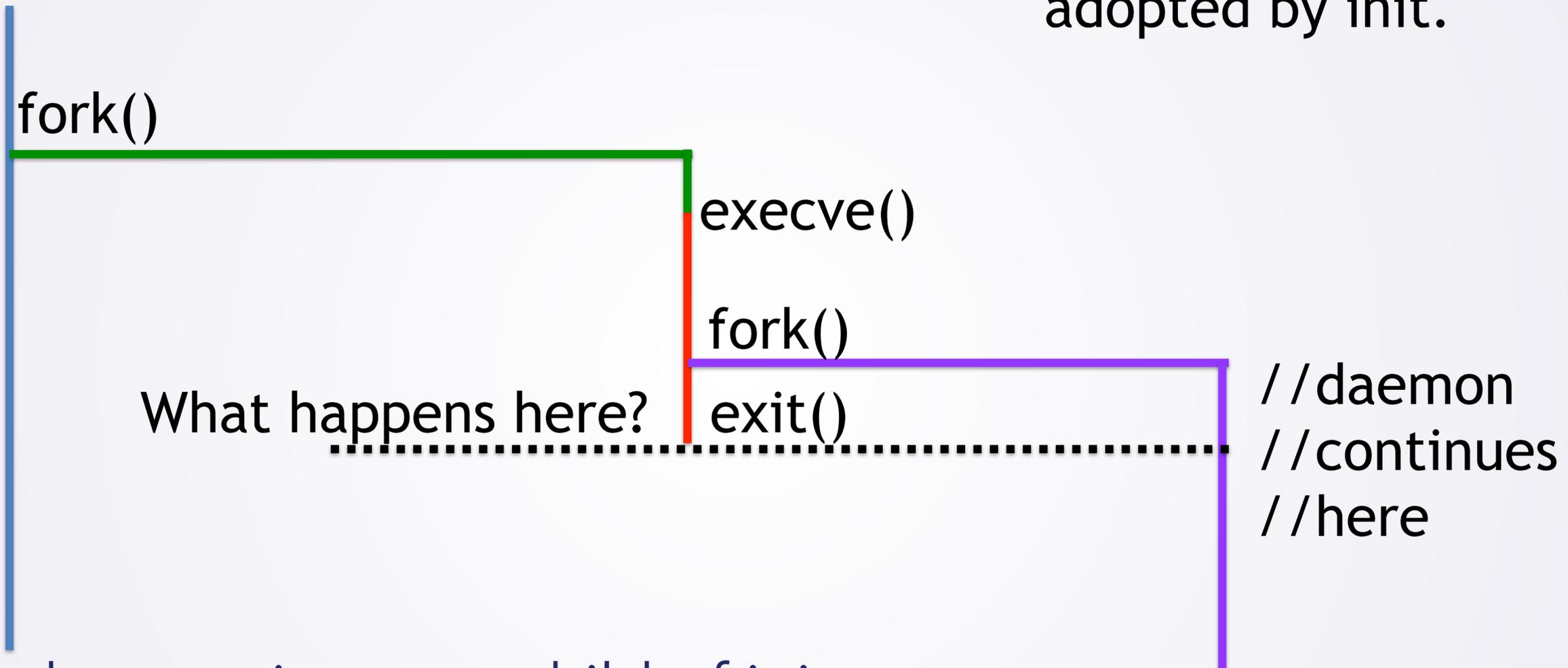
# What Happens Here?

- What happens when a process's parent exits?
  - **A:** The process also exits
  - **B:** The parent's call to exit blocks until all children exit
  - **C:** The process keeps running with no parent process
  - **D:** The process becomes a child of init

# Becoming a Daemon

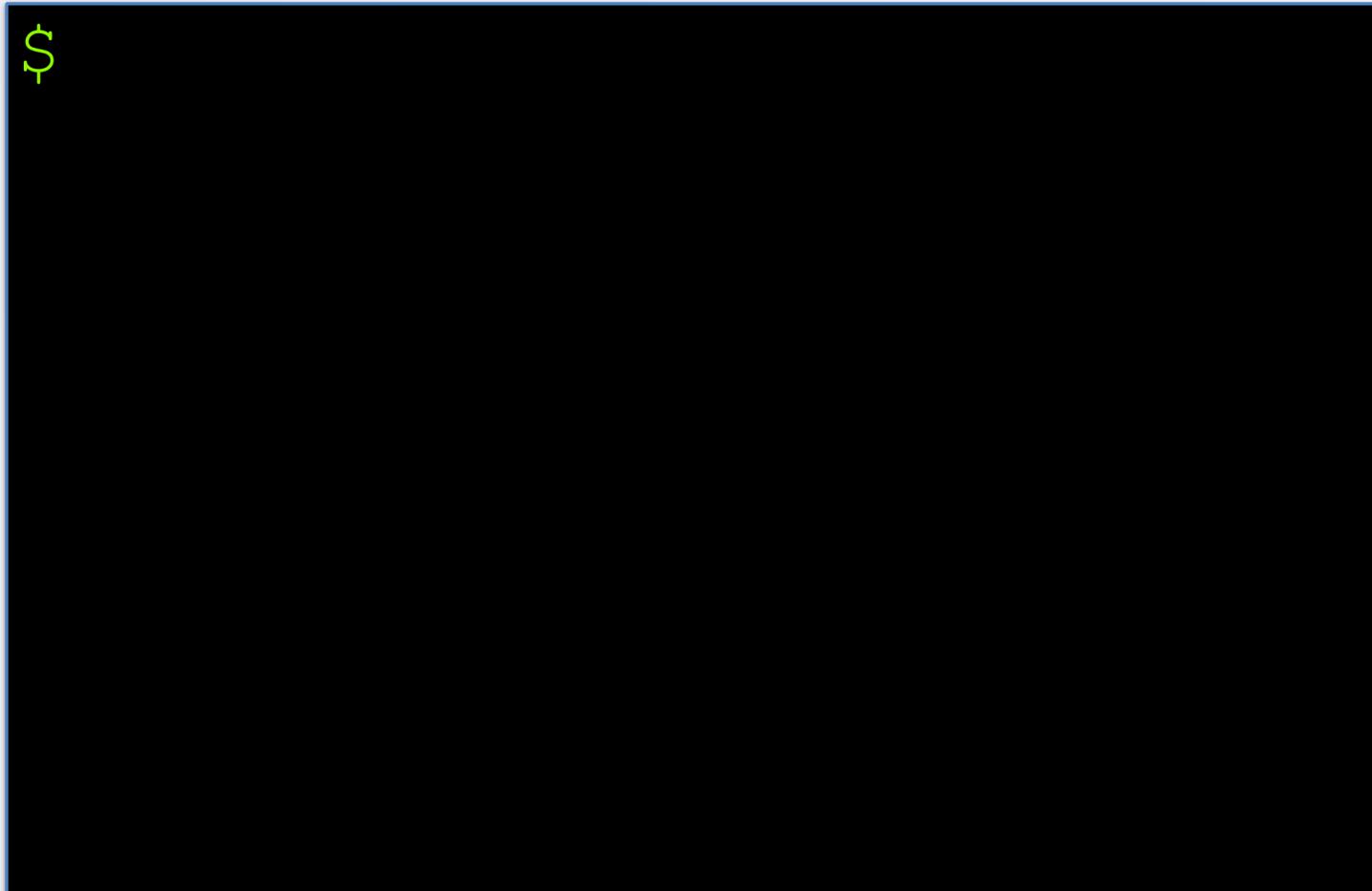
Whatever ran  
the daemon

This process is an orphan,  
adopted by init.



- Our daemon is now a child of init
- Some shells kill their children when they exit
- Daemon is guaranteed to not be a **process group leader**

# Process Groups



shell

- To understand process groups, let us think about some commands..

# Process Groups

```
$ find / -name xyz > tmp
```



- To understand process groups, let us think about some commands..

# Process Groups

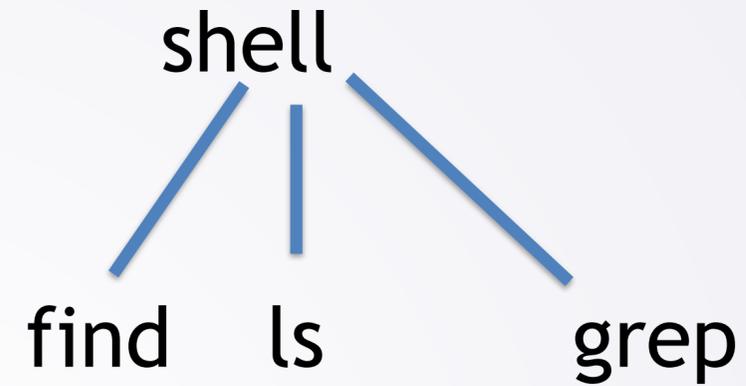
```
$ find / -name xyz > tmp  
^Z  
$ bg
```



- To understand process groups, let us think about some commands..

# Process Groups

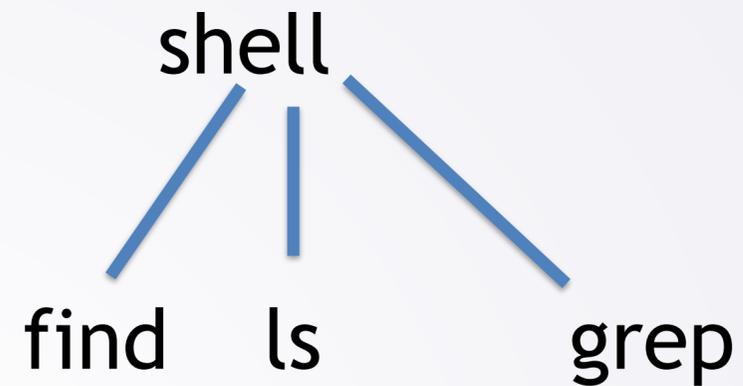
```
$ find / -name xyz > tmp  
^Z  
$ bg  
$ ls *x* | grep y
```



- To understand process groups, let us think about some commands..

# Process Groups

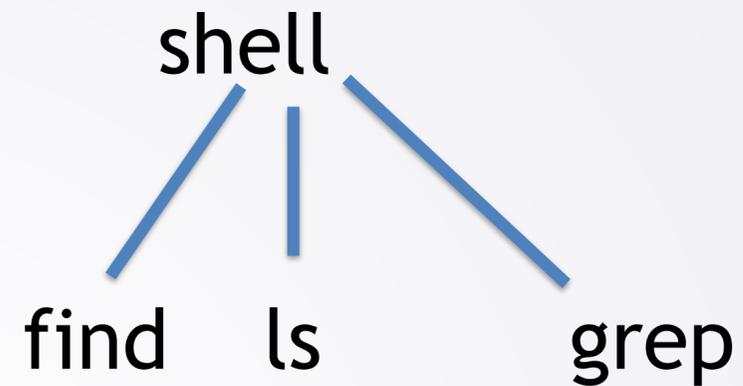
```
$ find / -name xyz > tmp
^Z
$ bg
$ ls *x* | grep y
^C
```



- To understand process groups, let us think about some commands..
  - Which process(es) to kill when I type ^C?

# Process Groups

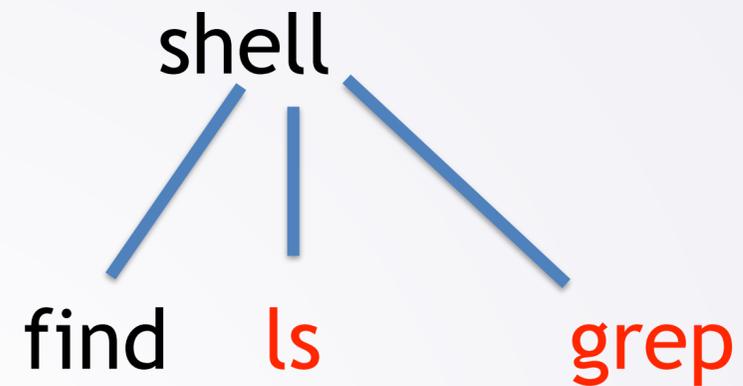
```
$ find / -name xyz > tmp
^Z
$ bg
$ ls *x* | grep y
^C
```



- Which processes should be killed here with ^C?
  - **A:** find, ls, and grep
  - **B:** ls, and grep
  - **C:** find
  - **D:** all four

# Process Groups

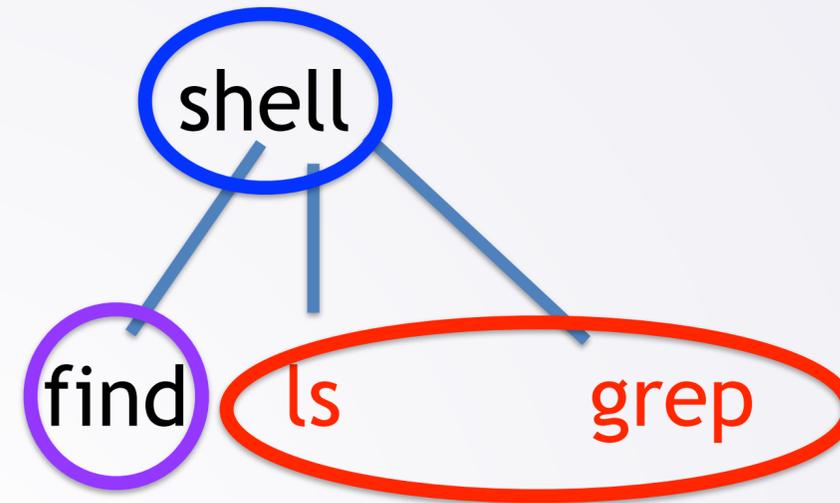
```
$ find / -name xyz > tmp  
^Z  
$ bg  
$ ls *x* | grep y  
^C
```



- To understand process groups, let us think about some commands..
  - Which process(es) to kill when I type ^C? **ls + grep**

# Process Groups

```
$ find / -name xyz > tmp
^Z
$ bg
$ ls *x* | grep y
^C
```

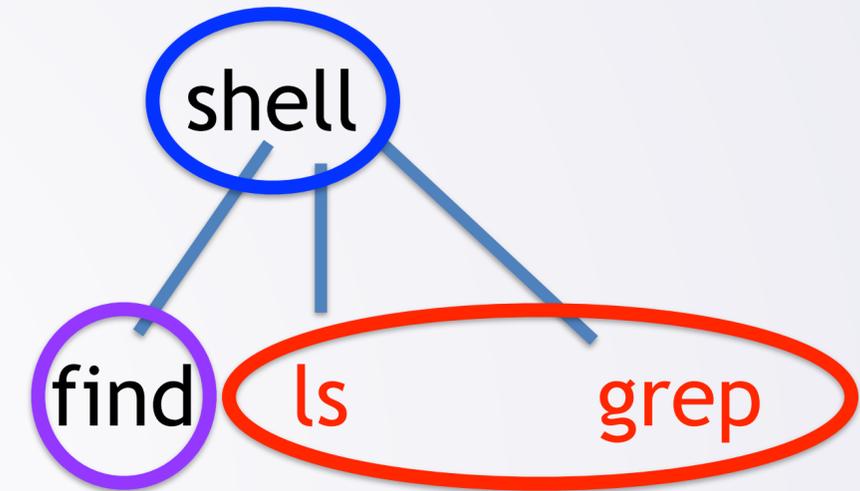


- Related processes organized into "process groups"
  - E.g., one command pipeline = one process group

# Process Groups

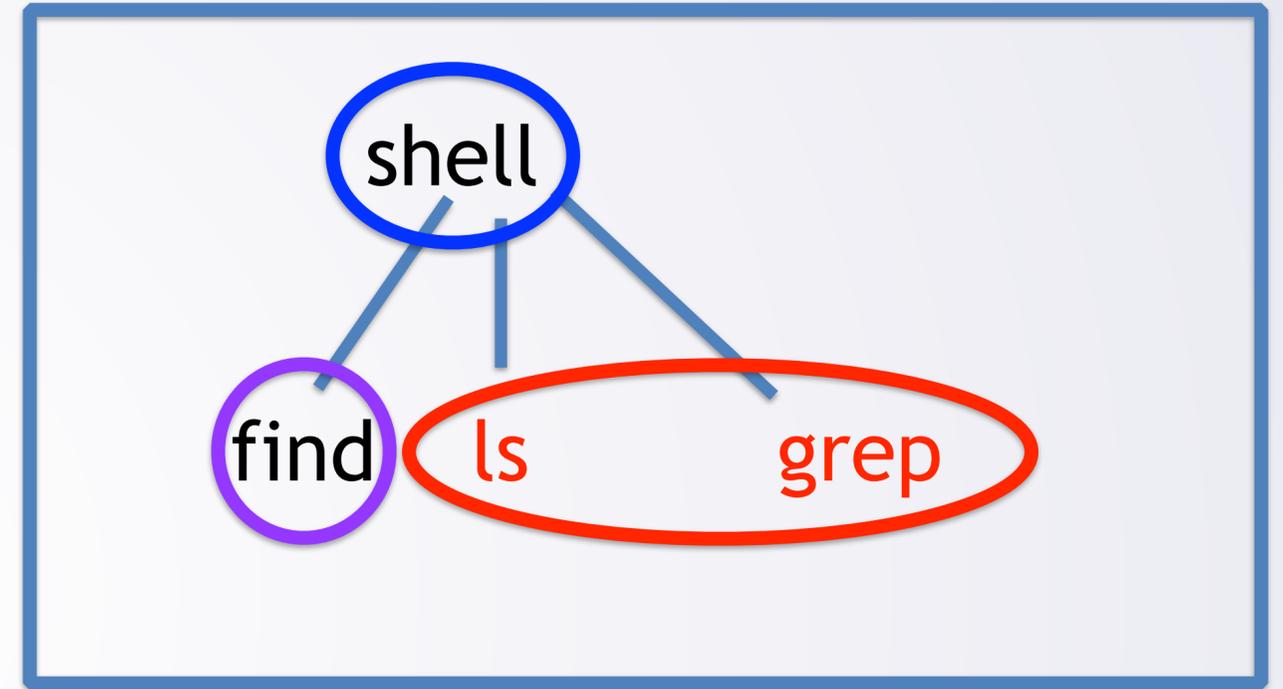
- Process groups recognized by kernel
  - Various operations are applied to process groups
    - What receive signals from  $^C$ ,  $^Z$ ,  $^\backslash$
    - Foreground/background of processes
- Background process groups stop if attempt to read/write terminal
  - Resumed when brought to foreground
- Ok, that's the basics of process groups...
  - ...but what do they have to do with becoming a daemon?

# Process Groups, Sessions, Controlling TTY



- Process groups relate to **sessions**
  - Sessions relate to **controlling ttys**
  - Daemons cannot have a controlling tty

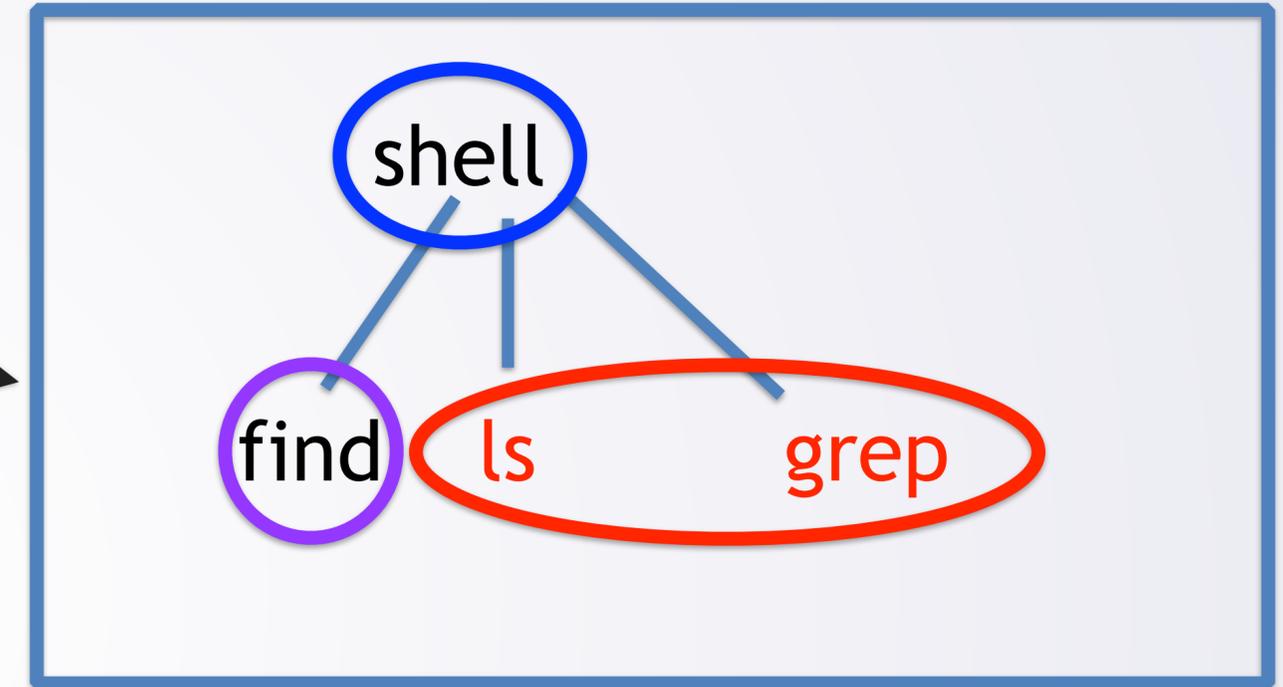
# Process Groups, Sessions, Controlling TTY



- The processes are all in one session
  - Session leader is the shell

# Process Groups, Sessions, Controlling TTY

```
$ find / -name xyz > tmp  
^Z  
$ bg  
$ ls *x* | grep y  
^C
```



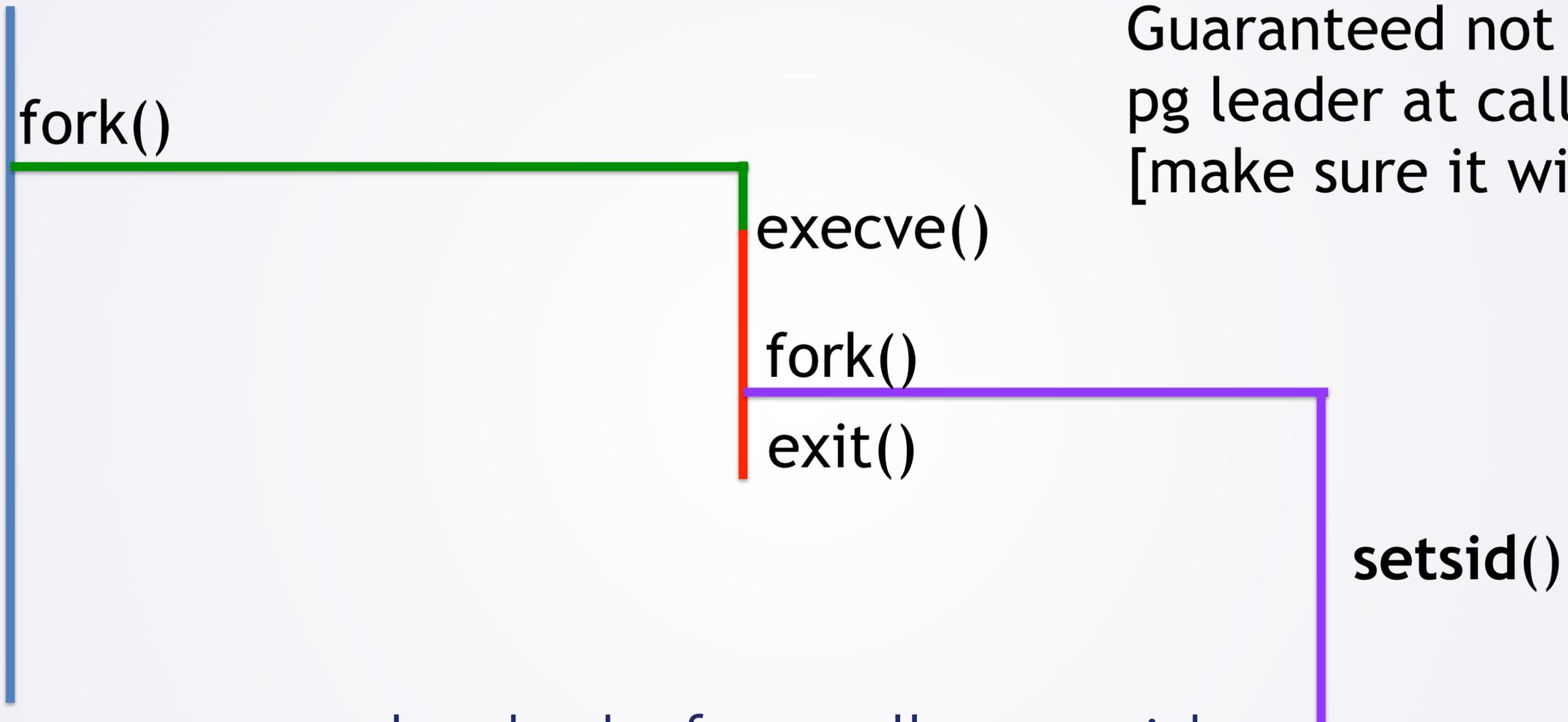
- Session has a controlling tty
  - The terminal that "owns" the processes

# New Sessions

- A process can start a new session by calling **setsid()**
  - Process must **NOT** be a process group leader
    - If caller is pg leader, fails.
  - On success:
    - Calling process is process group leader (of a new group)
    - Calling process is session leader (of a new session)
    - Newly created session has no controlling tty

# Becoming a Daemon

Whatever ran  
the daemon



- Daemon not pg leader before call to setsid

# Quick check up

- Which of the following is NOT true of a process that just successfully called `setsid()`
  - **A:** It is a process group leader
  - **B:** It is a session leader
  - **C:** It has a controlling TTY
  - **D:** None of the above is false (all are true)

# Becoming a Daemon

- Typically Required:
    - **fork(), parent exits**
    - **Dissociate from controlling tty**
    - Close stdin/stderr/stdout, open them to /dev/null
    - chdir to "/"
  - Good Ideas:
    - Clear umask
    - fork again -> not be session leader
- 
- daemon library call

# Point stdin/err/out at /dev/null

- Do not want stdin/err/out associated with old terminal
  - Generally do not want associated with a normal file either
- open /dev/null
  - Use dup2 to close stdin/err/out, and duplicate to fd of /dev/null

# Chdir to "/"

- Do not want to keep other directory "busy"
  - If cwd of a process is a directory, it is "in use"
- Change working directory to "/"
  - Will always be in use anyways

# Becoming a Daemon

- Typically Required:

- **fork(), parent exits**
- **Dissociate from controlling tty**
- **Close stdin/stderr/stdout, open them to /dev/null**
- **chdir to "/"**

} daemon library call

- Good Ideas:

- Clear umask
- fork again -> not be session leader

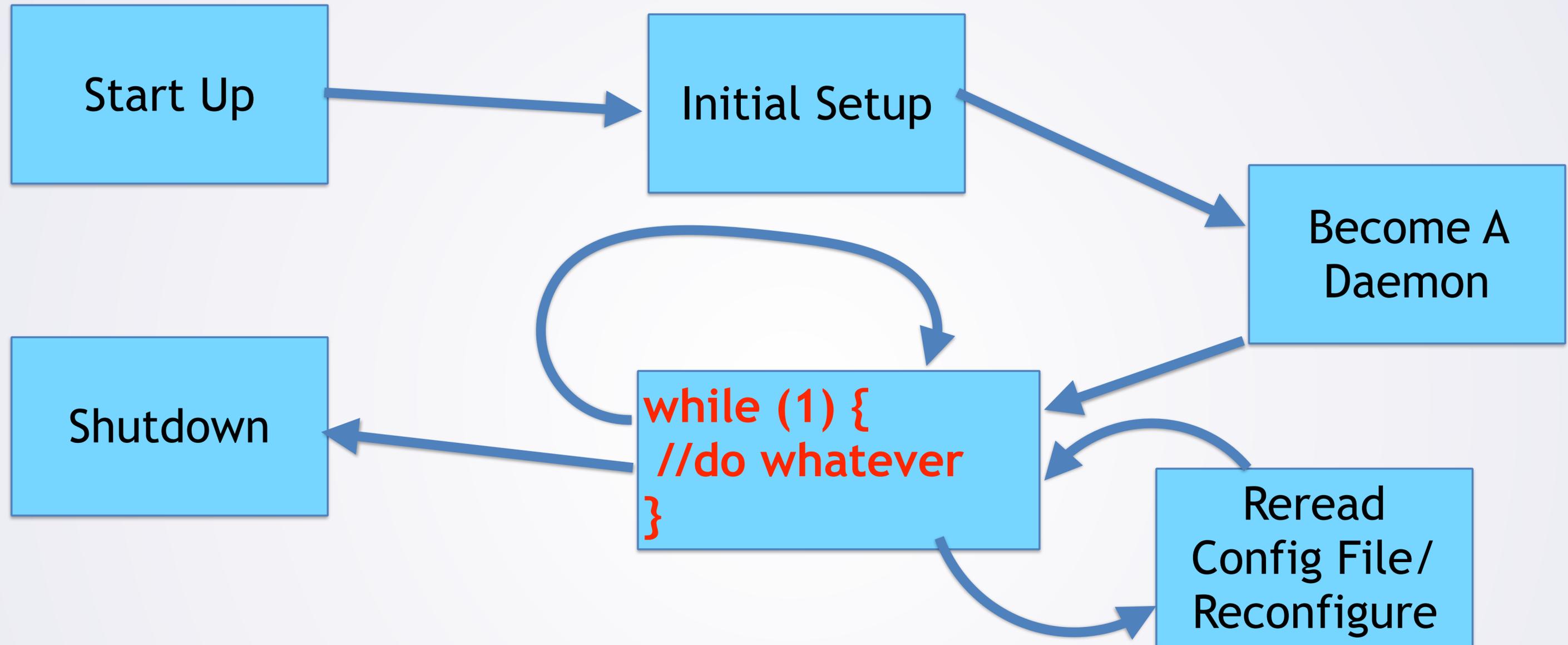
# Umask

- Processes have a "umask"—file creation mask
  - Affects the permissions of files that are created
  - Try to create with `mode`?
    - Actually `get mode & ~umask`
- Why?
  - Security: set default permissions to prevent
- Alter umask with `umask` system call (see `man umask(2)`).
  - Specify new mask.
- `umask (0) => clear umask (get exactly mod you request)`

# fork() Again, Do Not Be a Session Leader

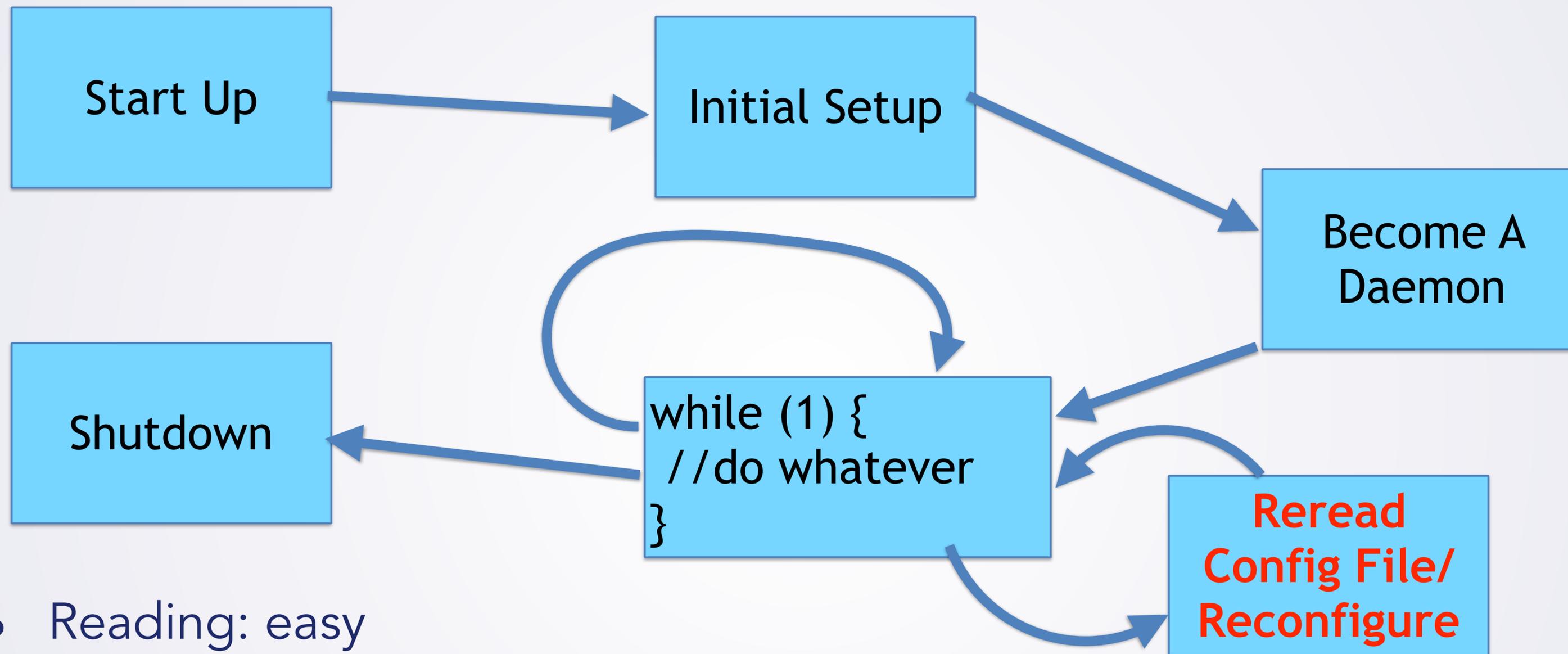
- May be a good idea to fork one more time
  - (How many forks do we need?!?!)
- Another fork() => new process is not session leader
  - Made it session leader to not have controlling tty
  - Now does not have...
- Why?
  - If a session leader without a controlling tty opens a tty...
  - That tty will **become the session's controlling tty** :(
  - Non-session leaders cannot gain a controlling tty

# Life Cycle



- Ok, now we are a daemon!
- Time to do useful stuff... forever...?
- Delve into this "stuff" shortly

# Life Cycle



- Reading: easy
- Re-configure: maybe tricky (depends on what to do...)
- How do we know **when** to reconfigure?

# Common Approach: SIGHUP

- Many daemons interpret the **signal** SIGHUP to mean "reconfigure"
- What are signals?
  - When OS wants to send *asynchronous* notification to process, send signal.
  - Many different signals (each numbered): SIGSEGV, SIGABRT, SIGHUP,...
  - Default actions: terminate (possibly w/ core dump), ignore, stop, continue
    - See man -S7 signal for specifics
  - Signals can also be blocked
- Programs can change behavior with **sigaction**
  - Default, ignore, or programmer-defined function

# Using sigaction

```
struct sigaction sigterm_action;
sigterm_action.sa_handler = my_function;
sigterm_action.sa_flags = some_flags; //e.g. SA_RESTART
if(sigemptyset(&sigterm_action.sa_mask) != 0) {
    //handle error
}
//use sigaddset to add other signals to sa_mask
if(sigaction(SIGHUP, &sigterm_action, NULL) != 0) {
    //handle error
}
```

- Basic structure of using sigaction to setup a signal handler

# Using sigaction

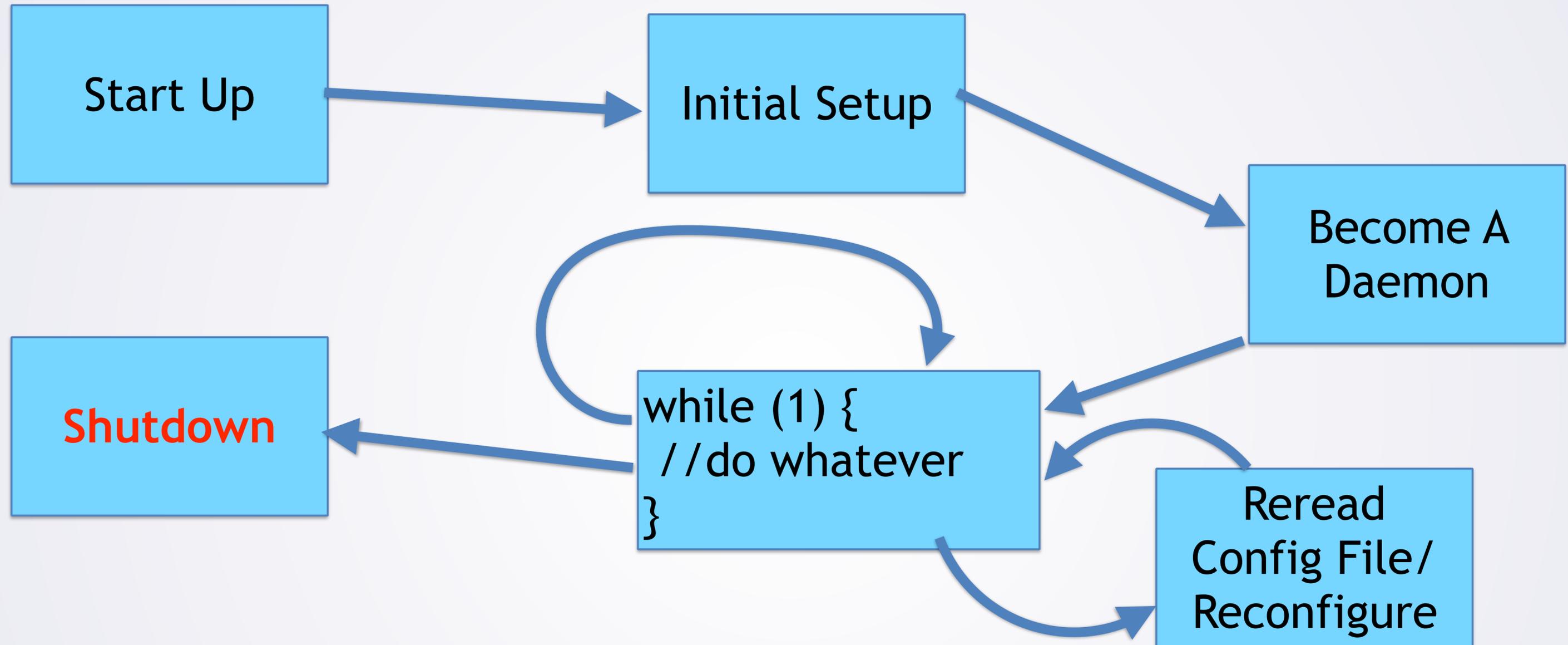
```
struct sigaction sigterm_action;
sigterm_action.sa_handler = my_function;
sigterm_action.sa_flags = some_flags; //e.g. SA_RESTART
if(sigemptyset(&sigterm_action.sa_mask) != 0) {
    //handle error
}
//use sigaddset to add other signals to sa_mask
if(sigaction(SIGHUP, &sigterm_action, NULL) != 0) {
    //handle error
}
```

- What is the type of sa\_handler in sigaction?
  - **A:** int sa\_handler
  - **B:** int \* sa\_handler
  - **C:** void \* sa\_handler
  - **D:** void (\* sa\_handler) int

# Signal Handler

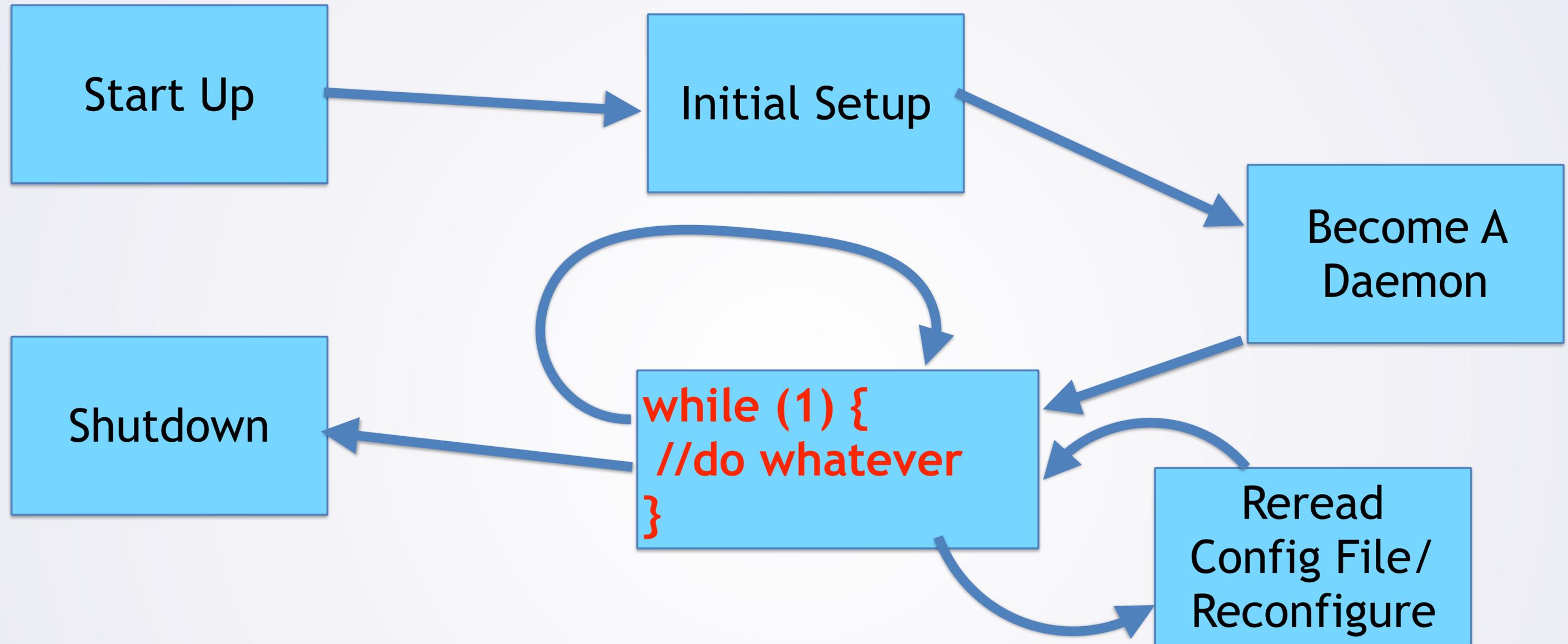
- Signal handler ("my\_function") looks like
  - `void my_function (int signal_number) { ... }`
- You have to be careful what you call/do in it
  - Program may be interrupted in the middle of something
  - Similar problems/ideas to data races with multiple threads
  - Some functions are defined as safe to call in signal handler

# Life Cycle



- Shut down daemon by sending signal
  - kill system call sends signal to a process

# Life Cycle



- Now let us go back and revisit the "stuff" that the daemon does

# Accept, Process, Respond [mostly]

```
while (true) {  
    req = accept_incoming_request();  
    resp = process_request(req);  
    send_response(req, resp);  
}
```

This might take many forms:

- accept() network socket
- read from FIFO/pipe
- read from DB table

- Not strictly a rule (may communicate both ways<sup>...</sup>, etc)
- But a good "general formula" to start from

# 650 Review: Sockets

```
while (true) {  
    req = accept_incoming_request();  
    resp = process_request(req);  
    send_response(req, resp);  
}
```

This might take many forms:

- **accept()** network socket
- read from FIFO/pipe
- read from DB table

...

- Speaking of **accept()**
  - Who can remind us about sockets from 650?

# Accept, Process, Respond [mostly]

```
while (true) {  
    req = accept_incoming_request();  
    resp = process_request(req);  
    send_response(req, resp);  
}
```

This might take many forms:

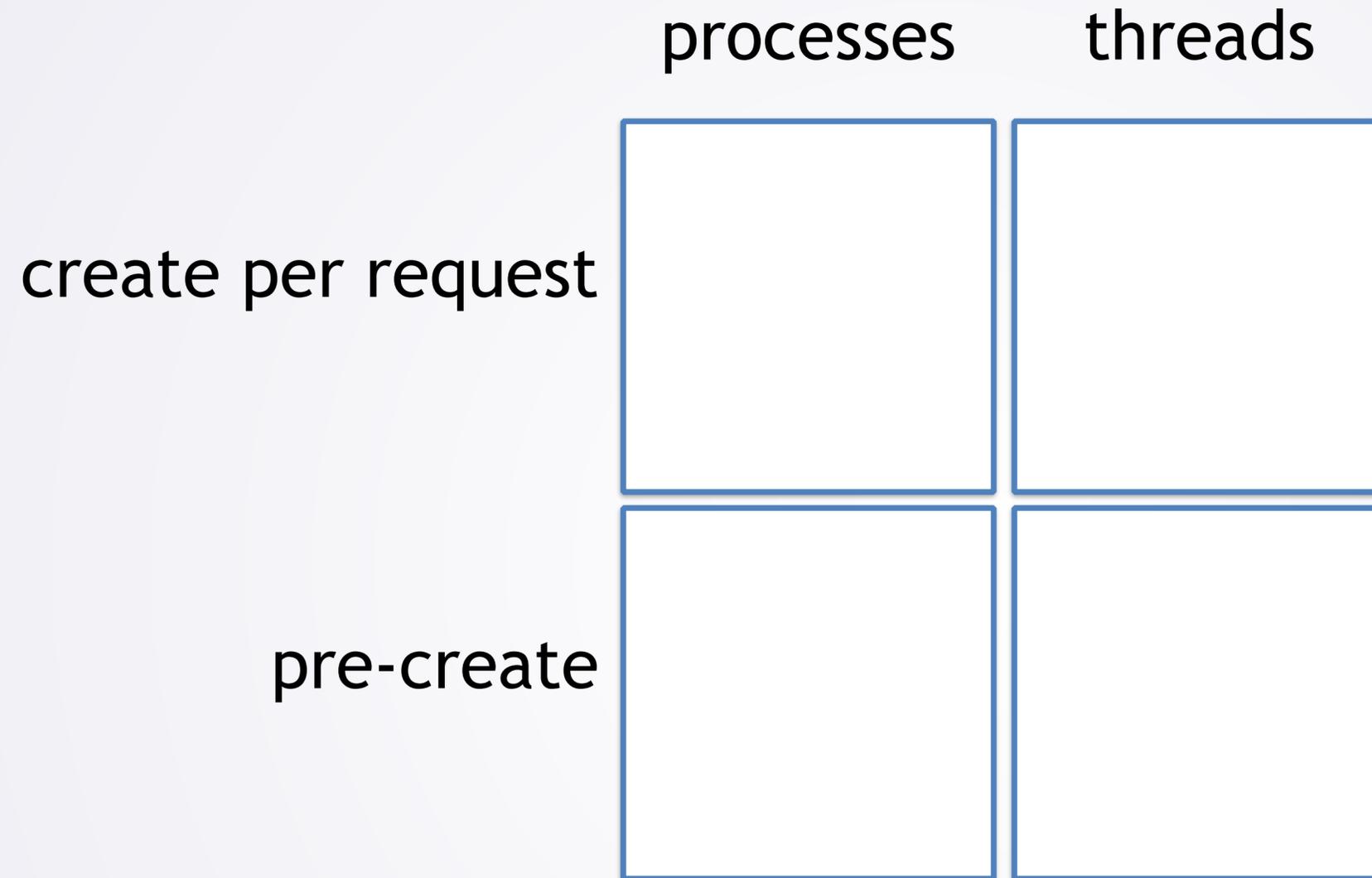
- accept() network socket
- read from FIFO/pipe
- read from DB table

- As noted last time: probably want some **parallelism**...
  - What would this parallelism look like?

# Think, Pair, Share

- What ways might we structure this parallelism?
  - How do we run code in parallel (hint: 2 ways)?
  - How could we put these to use?

# Parallelism



- What does this parallelism look like?
  - 4 main options

# fork per-request

```
while (true) {
    req = accept_incoming_request();
    pid_t p = fork();
    if (p < 0) { /*handle error */ }
    else if (p == 0) {
        resp = process_request(req);
        send_response(req, resp);
        exit(EXIT_SUCCESS);
    }
    //cleanup: close/free req
    //need to wait for p w/o blocking
}
```

Pros and cons?

# Advantages of Fork-per-request

- Which would be an advantage of **fork-per-request**?
  - **A:** Low overhead
  - **B:** Easy to share state between requests
  - **C:** Isolation between requests
  - **D:** None of the above

# Fork-per-request Pros/Cons

- Pros:
  - Simplicity: avoid difficulties of multi-threaded programming
  - Isolation between requests : separate processes
- Cons:
  - No ability to share state between/across requests
  - fork() latency on critical path
  - Creates arbitrary number of processes

# Pre-fork

```
for (int i = 0; i < n_procs; i++) {  
    pid_t p = fork();  
    if (p < 0) { /* handle error */}  
    else if (p == 0) {  
        request_loop(); //never returns  
        abort(); //unreachable  
    }  
    else {  
        children.push_back(p);  
    }  
}
```

**//What happens here depends...**

# pre-fork request loop

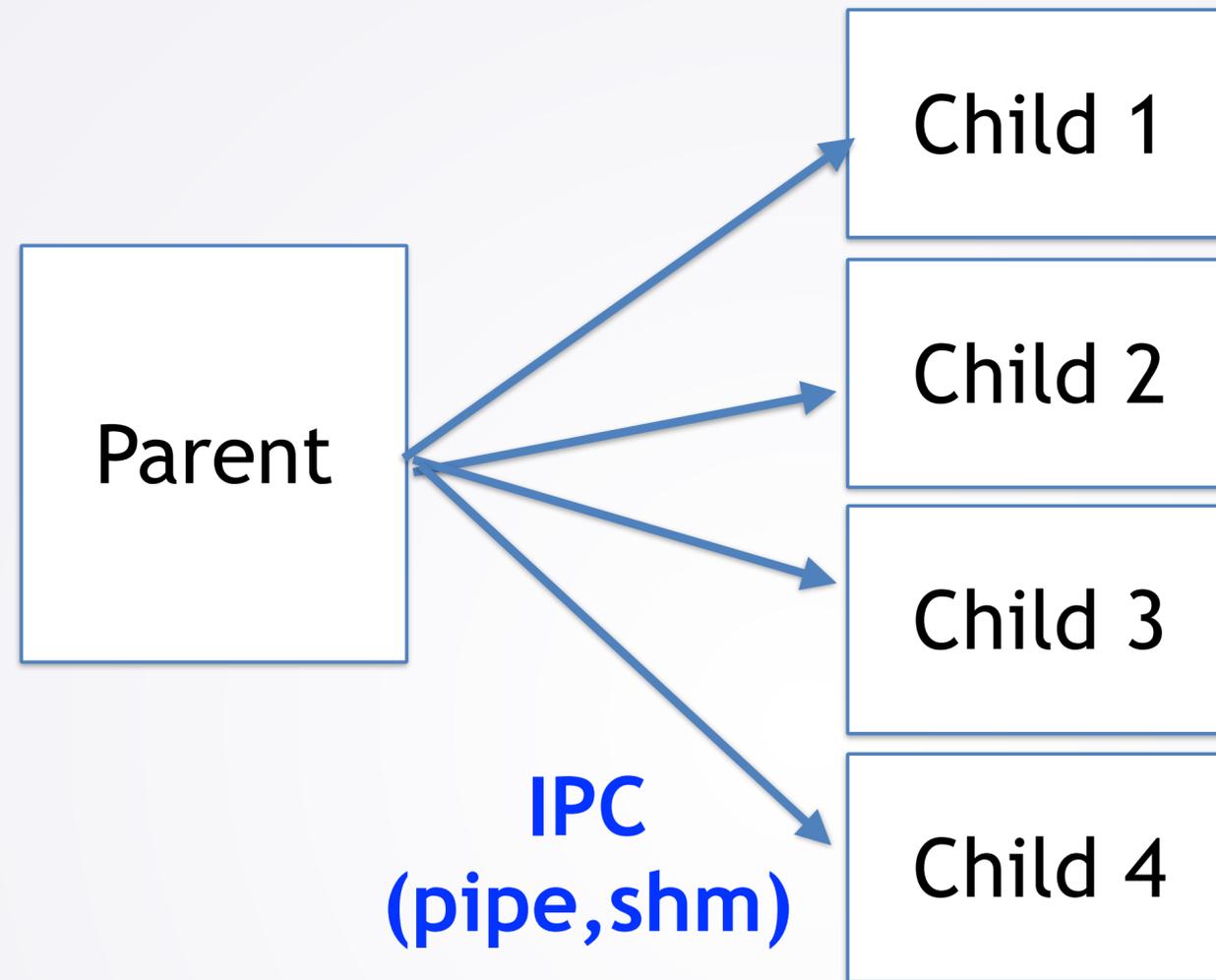
```
void request_loop(void) {  
    while (true) {  
        req = accept_incoming_request();  
        resp = process_request(req);  
        send_response(req, resp);  
    }  
}
```

- How does this work across multiple processes?
  - ...it depends...

# Remind Us About... IPC

- Who can remind us about interprocess communication?
  - What approaches do you know?
  - How do they work?

# Parent Dispatches Requests



- One approach: parent dispatches requests:
  - Requests come into parent
  - Parent chooses a child and sends it via IPC (pipe, shared memory,..)

# Pre-fork

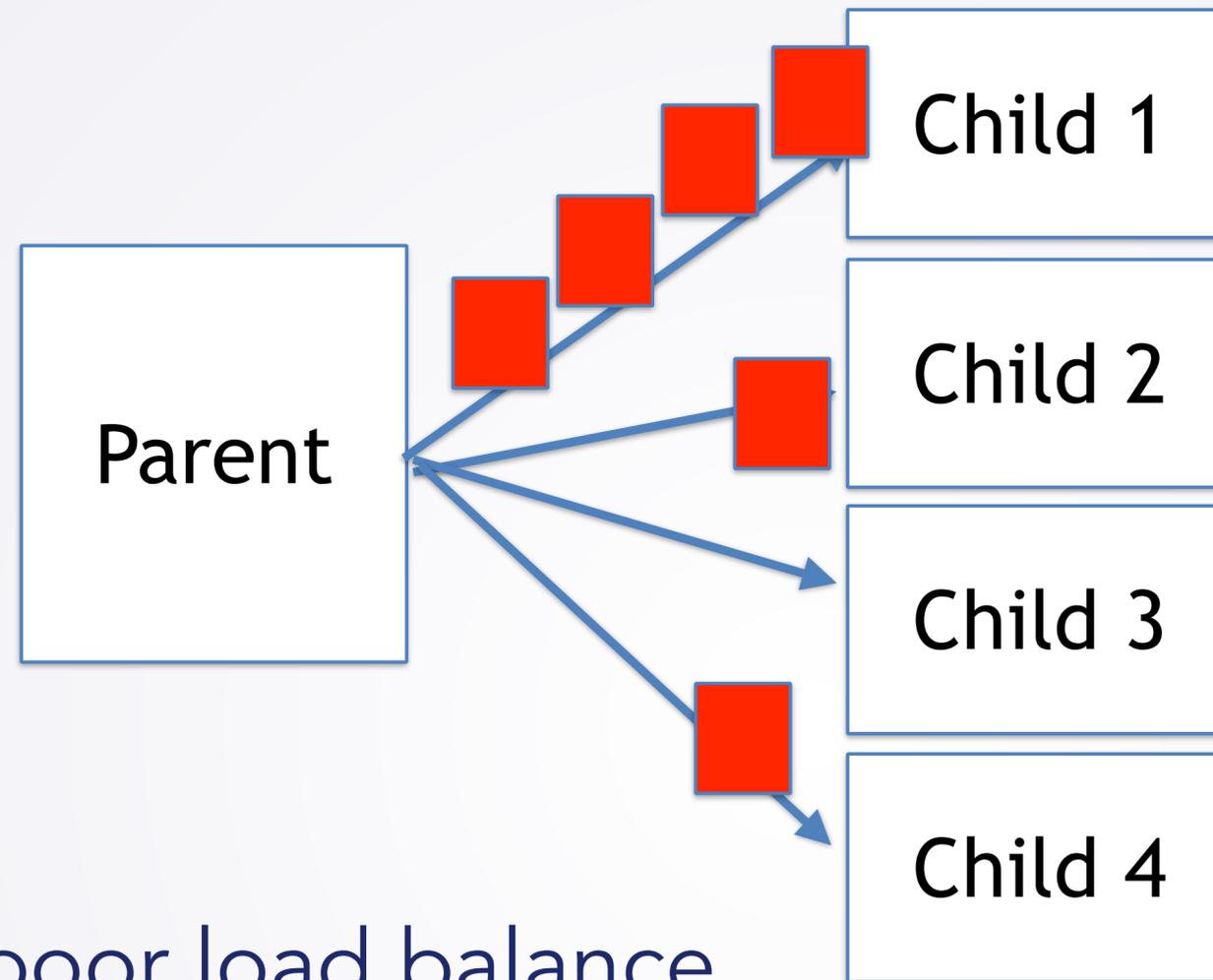
```
for (int i = 0; i < n_procs; i++) {  
    pid_t p = fork();  
    if (p < 0) { /* handle error */ }  
    else if (p == 0) {  
        request_loop(); //never returns  
        abort(); //unreachable  
    }  
    else {  
        children.push_back(p);  
    }  
}
```

Need to add code  
to setup  
IPC here  
(before fork())

Need to record  
IPC info in  
data structure

**request\_dispatch\_loop(); //accept, send to child**

# Load Balancing



- System has poor load balance
  - Child 1 is overloaded, Child 3 is underloaded
- What if we dispatch round-robin (1, 2, 3, 4, 1, 2, 3, 4),...
  - Requests have different latency -> may still have poor balance

# Networking (or other fd-based reqs)

- Previous approach not great for network sockets
  - Can't easily send a socket over a pipe (fd is just a number)
- Common approach: each process calls `accept()` on same server socket
  - Create socket, bind, listen before `fork()`
  - Have each process call `accept()`
  - Safe under Linux, cannot find POSIX guarantees
- Not best for performance
  - Preferable [on Linux]: have each process make own socket
  - Use `SO_REUSEPORT` socket option: all can bind to same port
    - If interested: <https://lwn.net/Articles/542629/>

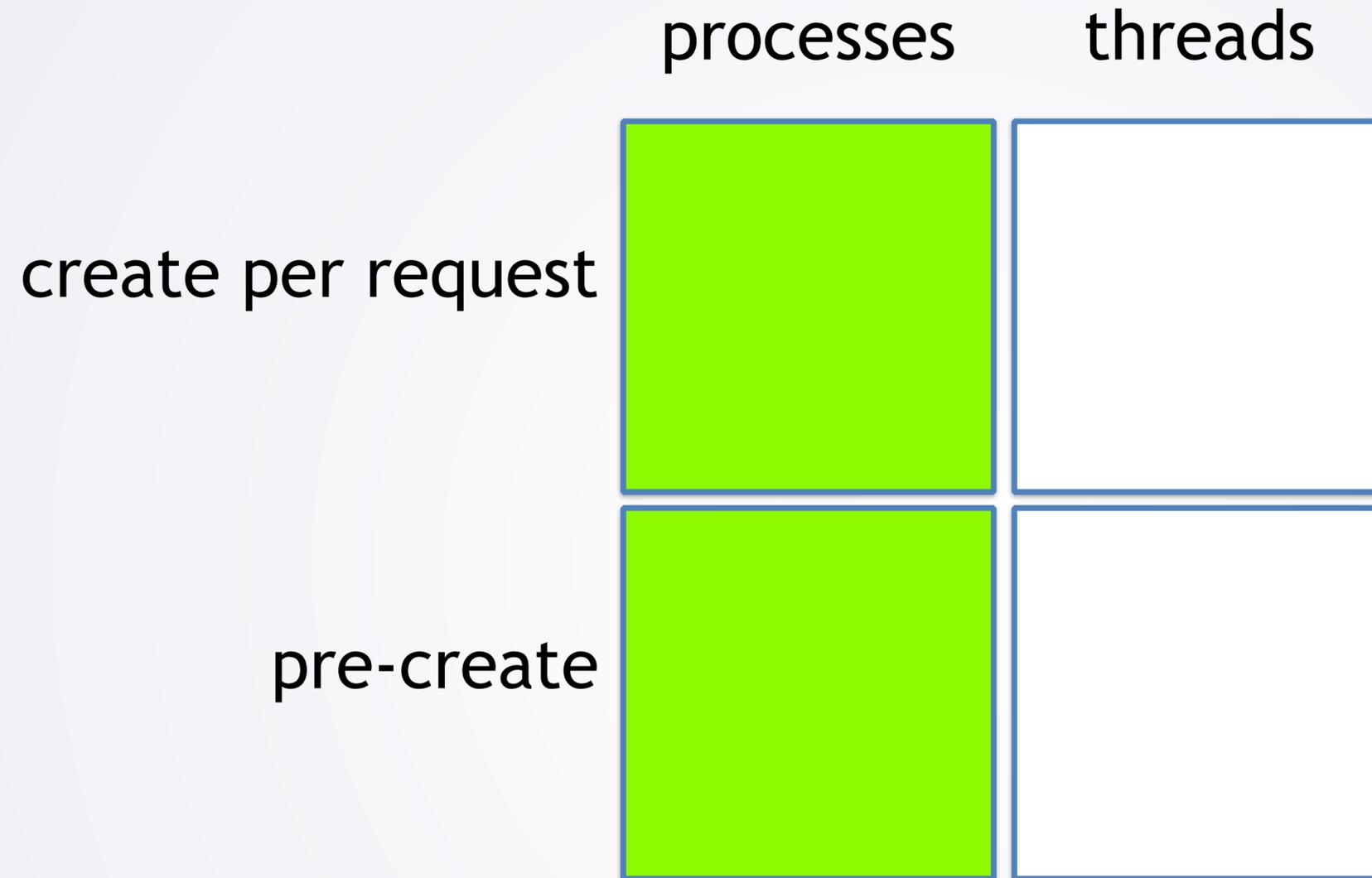
# Advantage of Pre-Forking

- Which would be an advantage of **pre-forking** relative to fork-per-request?
  - **A:** Lower overhead
  - **B:** Easier to share state between requests
  - **C:** Stronger isolation between requests
  - **D:** None of the above

# pre-fork

- Pros:
  - Simplicity: avoid difficulties of multi-threaded programming
  - Some isolation between requests
  - Choose number of processes (can even adjust dynamically...)
  - fork() overhead only once at startup=> lower overhead
- Cons:
  - No ability to share state between/across requests
  - Not as much isolation as per-request forking
  - [Some forms] More likely to need explicit load balancing

# Parallelism



- Talked about process-based approaches (forking)
- Now let's talk about the thread-based ones.

# Threads

- Similar code to forking:
  - Replace fork with `pthread_create`
  - Communication? Simpler: naturally share memory
    - Easier to have shared queue of requests for pre-created threads
- Have to deal with multi-threaded programming
  - Harder parts come exactly when we get benefits from MT over MP
    - Shared state

# Thread Per Request

- Pros:
  - Shared State
- Cons:
  - Complexities of multi-threaded programming
  - No isolation
  - No limit on number of threads created (may be highly inefficient)
  - Overhead of `pthread_create()` is on critical path

# Pre-Create Threads

- Pros:
  - Shared State
  - Probably easier load balancing
  - Overhead for `pthread_create` up front
  - Shared State
  - Easy to control (and adjust) number of threads
- Cons:
  - No isolation
  - Complexities of multi-threaded programming

# Which To Pick?

- Which one to pick?
  - Depends on what you need to do
- You should understand the tradeoffs of each option
- Think carefully/critically as you design your server

# UNIX: Users, Permissions, Capabilities

- Important considerations for UNIX Daemons
  - What user does it run as?
    - What if it needs \*some\* **privileged** operations?
  - Relatedly: file permissions/ownership
- Now:
  - Users: uid manipulation, setuid programs
  - File permissions
  - Capabilities

# UNIX Users

- You are used to running as a "normal" user
  - But now you have "root" on a machine..
- root is the privileged user: uid 0
  - Aka "super user"
- Can perform operations that normal users cannot
  - Load kernel modules
  - Adjust system settings
  - Listen on privileged ports (< 1024)
  - Change to other users...
  - ...

# ROOT IS DANGEROUS

- Anything running as root is **DANGEROUS**
  - Can do anything to the system
    - Add accounts, change password
    - Setup key loggers
    - Hide its own existence
  - 650 later this semester: will make rootkit
- Want to minimize what happens as root
  - When possible, run as "nobody"

# setuid(): switch users

- Do privileged operations, then switch users
  - `setuid(...);`
- Example:
  - Start as root
  - bind to/listen on privileged port
  - `setuid(...)`
- Useful if all privileged operations are needed **at start**

# Real, Effective, Saved UID

- There are three UIDs for each process:
  - **Real** user id: the user id of the user who ran it
  - **Effective** user id: the user id currently used for permission checking
  - **Saved set-user-id**: remembers "set-user-id" on suid binaries
- Set-user-id binaries:
  - File permissions that specify to switch euid at the start of execution
  - This is what lets programs like sudo, su, etc work

# UID Example

```
int main(void) {
    uid_t temp = getuid();
    printf("uid: %d\n", getuid());
    printf("euid: %d\n", geteuid());
    seteuid(temp);
    printf("uid: %d\n", getuid());
    printf("euid: %d\n", geteuid());
    seteuid(0);    fails (EPERM)
    printf("uid: %d\n", getuid());
    printf("euid: %d\n", geteuid());
    return EXIT_SUCCESS;
}
```

```
uid: 1001
euid: 1001
uid: 1001
euid: 1001
uid: 1001
euid: 1001
```

- Compile, run as user 1001

# UID Example

```
int main(void) {
    uid_t temp = getuid();
    printf("uid: %d\n", getuid());
    printf("euid: %d\n", geteuid());
    seteuid(temp);
    printf("uid: %d\n", getuid());
    printf("euid: %d\n", geteuid());
    seteuid(0); Succeeds: Saved-set-user-id is 0
    printf("uid: %d\n", getuid());
    printf("euid: %d\n", geteuid());
    return EXIT_SUCCESS;
}
```

```
uid: 1001
euid: 0
uid: 1001
euid: 1001
uid: 1001
euid: 0
```

- `sudo chown root.root a.out`
- `sudo chmod u+s a.out` //make program suid: **USE WITH CAUTION!**

# UID Example

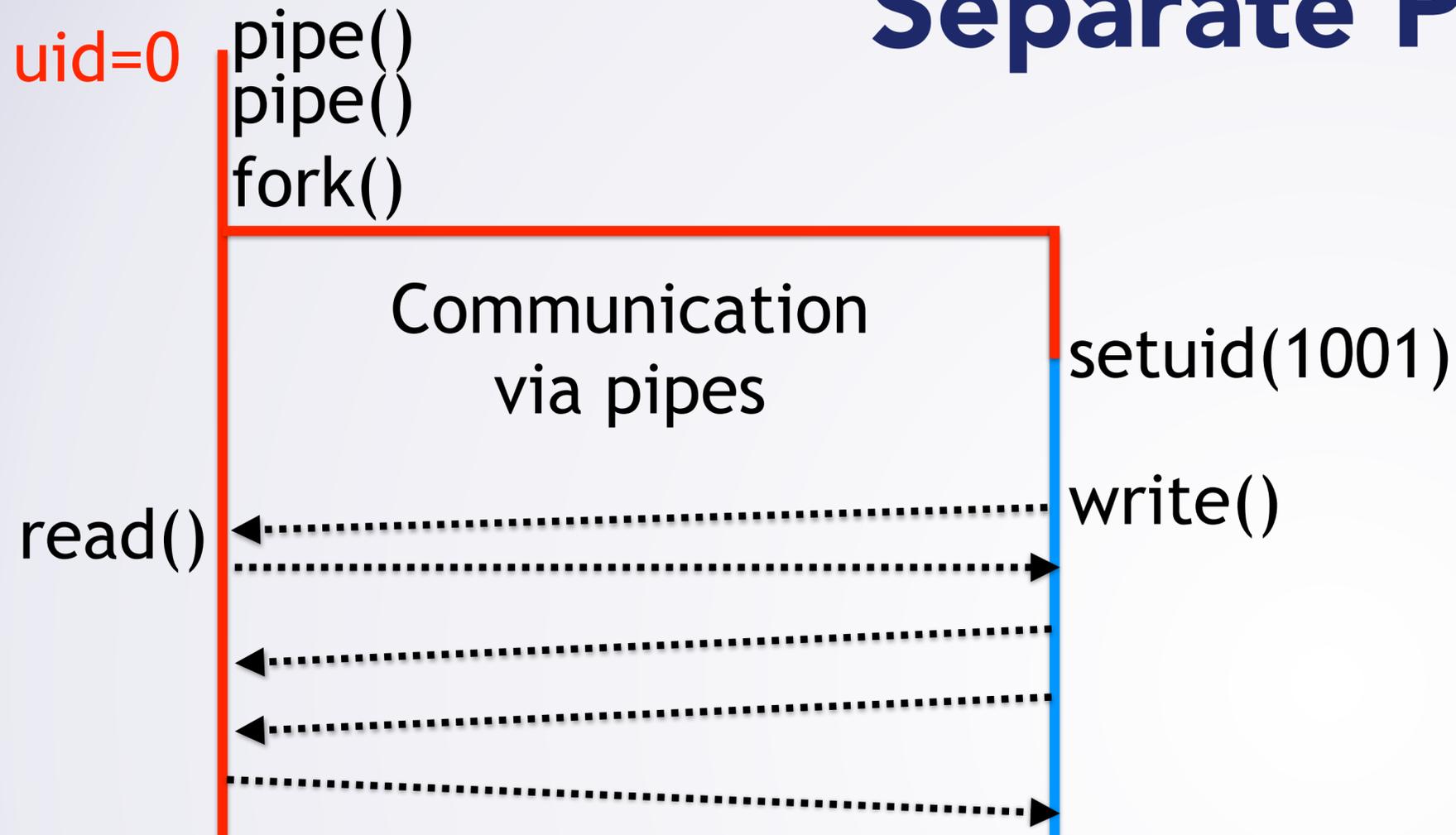
```
int main(void) {  
    uid_t temp = getuid();  
    //Dangerous: root permissions  
    seteuid(temp);  
    //Safer: user 1001 permissions  
    seteuid(0);  
    //Dangerous again  
    return EXIT_SUCCESS;  
}
```

- This program is safer when euid is not 0
- Not completely safe: arbitrary code exploit can seteuid(0)

# Think, Pair, Share

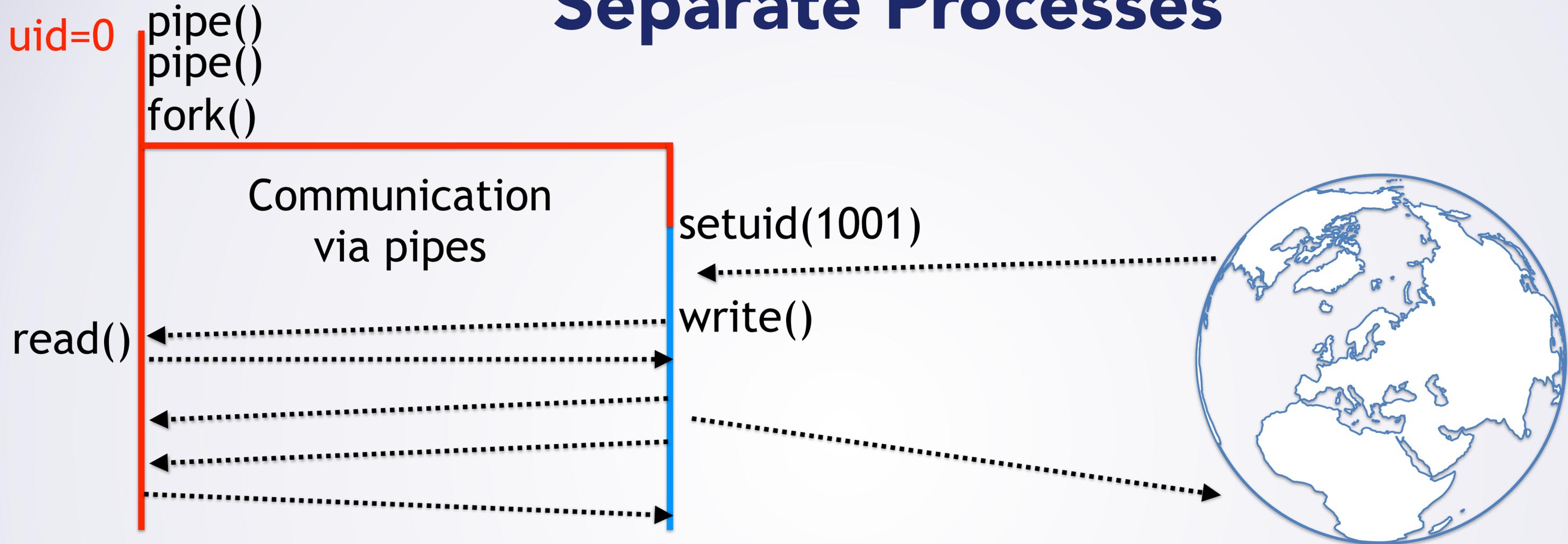
- How could we make things safer?

# Separate Processes



- Privileged process can fork
  - New process can completely drop privileges (call `setuid()` to change all uids)
- Communication can be done with your favorite IPC

# Separate Processes



- Unprivileged Process: Interacts with outside world
  - Sends request to privileged process as needed
    - What does this sound like? (a couple familiar ideas...)

# Linux Capabilities

- Linux (since 2.2) has the concept of **capabilities**
  - Divides root's super-user powers into sub-abilities (~40)
  - Example: CAP\_NET\_BIND\_SERVICE — bind to port < 1024
- Why useful?
  - If all you need is to bind a privilege port, can have
  - Without ability to do other things (load modules, change permissions,...)
- Executables can be granted individual capabilities
  - Rather than full set-uid status

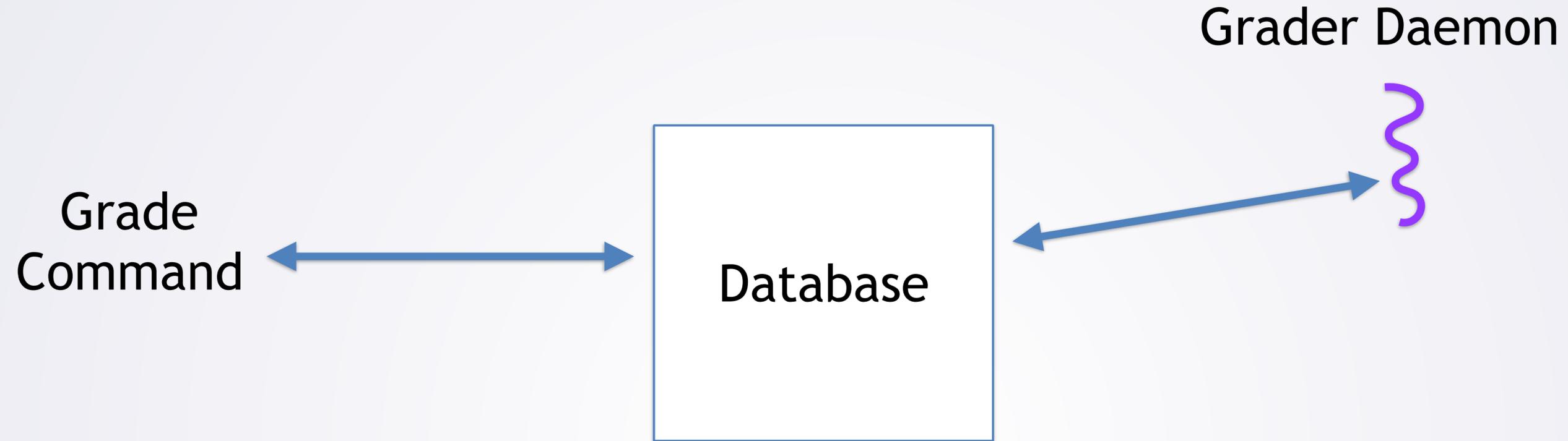
# Other User/Permissions Things

- Similar concepts/system calls apply/exist for group ids
  - Programs can be "set group id"
- There is also a "file system user id"—not so common to use

# Case Study: ECE 551 Grader System

- Everyone's favorite piece of server software!
  - Very interesting from a system design perspective
- Requirements:
  - Run arbitrary (student) code w/o security risk
    - Not concerned about things you could do at shell
    - Concerned about access to grades/grader
  - Simple/low overhead commands [do not require password each time]
  - Interface with git

# Grading



- Student runs grade command
- Grader daemon responsible for grading
- Database holds state

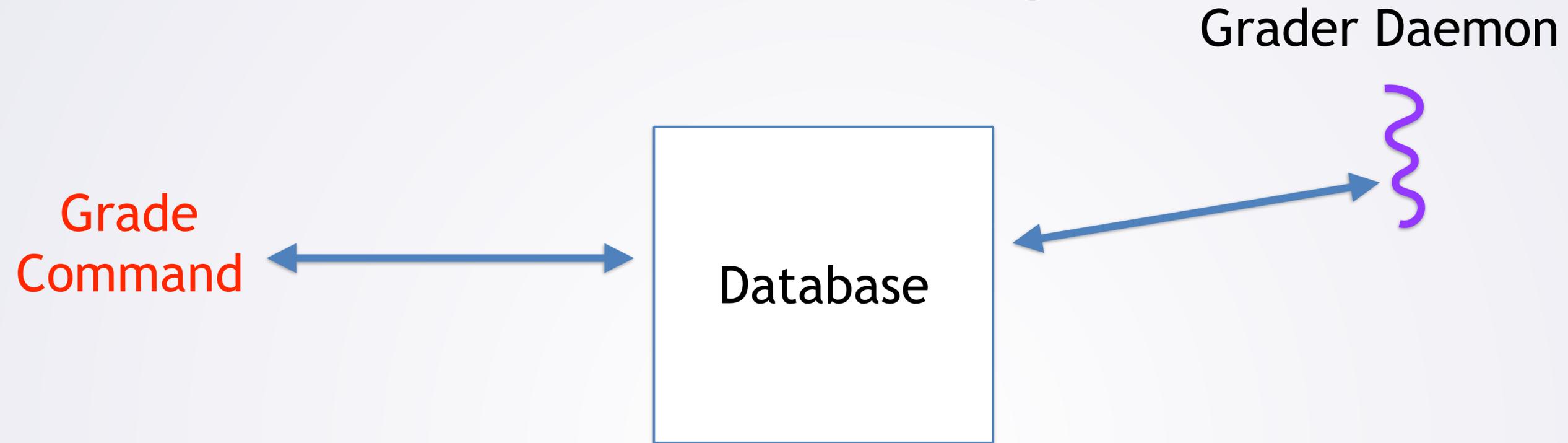
# Think, Pair, Share

Grader Daemon



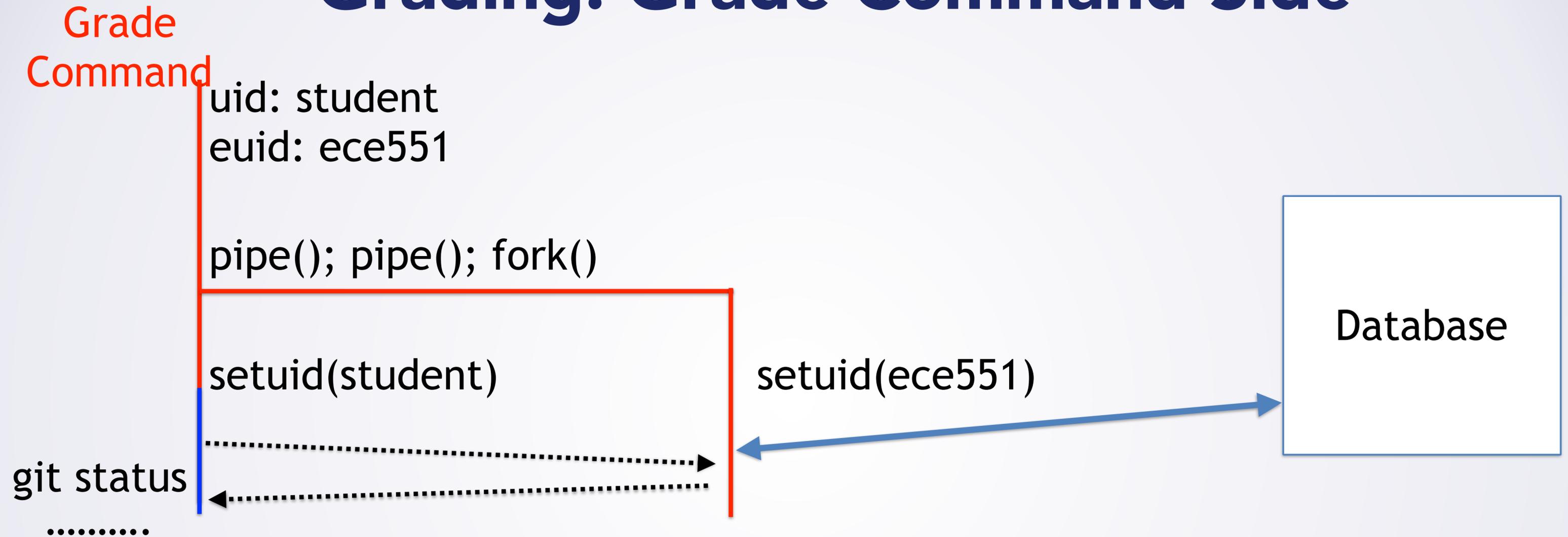
- What could possibly go wrong here?
  - What security issues was I worried about when I designed this?

# Grading



- Grade command: runs as student
  - But accesses database
- How do we prevent student from accessing db directly?

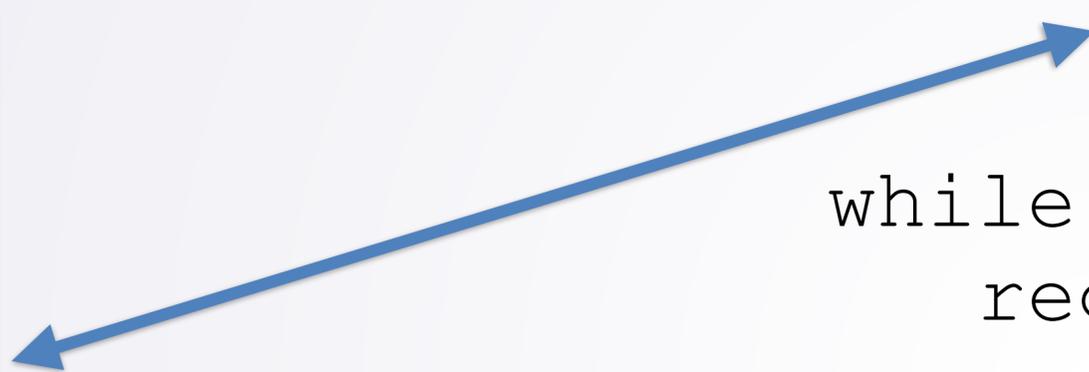
# Grading: Grade Command Side



- grade is setuid ece551
  - Sets up pair of pipes, then fork()s
  - One process becomes "student side", other "ece551 side"
  - Communication over pipes with Google Protocol Buffers

# Grading: Daemon Side

## Grader Daemon



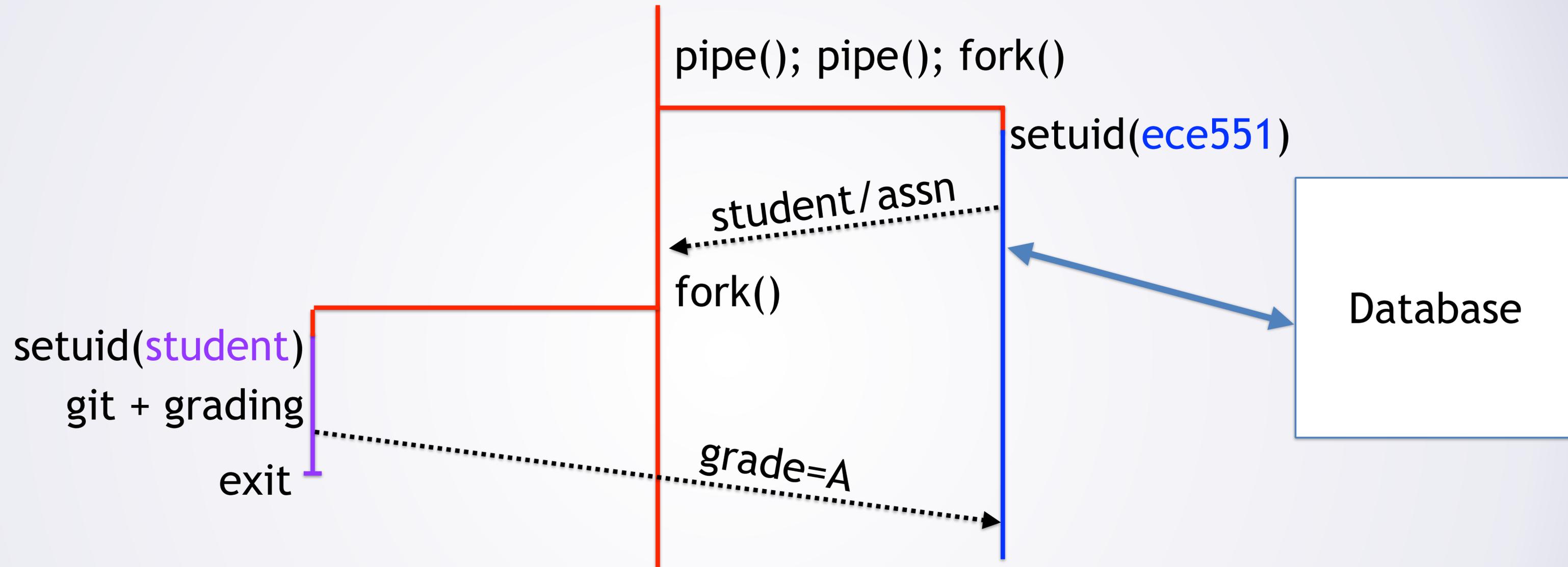
```
while (true) {  
    req = accept_incoming_request();  
    resp = process_request(req);  
    send_response(req, resp);  
}
```

- Same structure we've seen before
  - Accept request: from database **Run as ece551**
  - Process: grade it **Run as...?**
  - Send request: update DB **Run as ece551**

# Doing Actual Grading

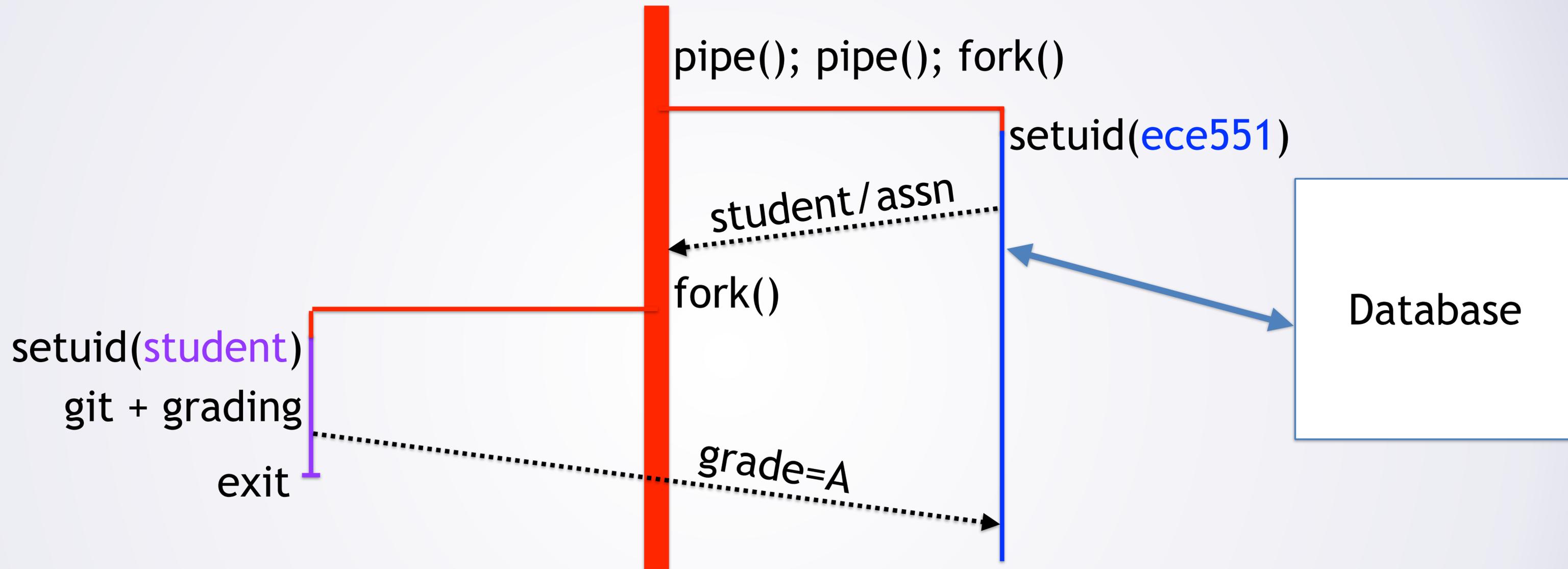
- Need to access student repository: push + pull
  - Want student repo permissions restricted to only student
  - ...so run as student?
- Want actual student code to be run as **nobody**
  - Minimal permissions
  - ...but also need code to not be able to read grader files [answers,etc]
- So to process a request:
  - git pull [as student]
  - run code/grader [as nobody—without grader readable by nobody?]
  - git push [as student]

# Graderd



- graderd runs as **root!**
  - How does the grading get done as nobody? Another program

# Question



- Why does this process need to run as root?
  - **A:** So it can write /var/log/grader.log
  - **B:** So its children can setuid to any student
  - **C:** So it can call daemon()
  - **D:** So it can access the database

# Minimal Privileges

- Run with **the lowest privilege** as possible
  - The lower the privilege, the less damage you can do
- **Separate** code that needs different privileges
  - **Communicate** over well defined API
  - **Restrict** requests that can be sent to privileged code
  - Privileged code must **distrust** less privileged code
- Could go even further:
  - Separate across machines...
  - We'll discuss this idea later!