Process Groups & Terminal Control

Computer Operating Systems, Spring 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/cis5480

- Congrats on finishing up Shredder & Penn-Vec; how did you feel about it?
  - Expect Style grading to be out by Sunday afternoon!

Administrivia

- Recitation
  - Thursday's @ 7PM – 8PM
  - Reviewing namely fork and exec. Then go over files and pipes.
  - But obviously, you all say what they go over if there's a consensus.

- Partners will be randomly assigned right after this lecture.
  - If you need to contact your partner, let us know and we can email both of you

- **Check-in Quiz** due before next lecture
  - Will release it tonight!

Poll Everywhere

pollev.com/cis5480

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Process Groups**
  - `setpgid()`
- Terminal Control
  - `tcsetpgrp()`
- SIGSTOP
- Project 1: Synch vs Asynch wait
  - `SIGCHLD`

Process Groups

- Process Groups: A way to associate processes together
  - Processes groups ***are never empty.***
- Convenient process & signal management:
  - If `SIGINT` is sent to a process via the keyboard, ***it is also sent to all processes within its group by the kernal.***
- When we create a process via `fork()`, the child and parent belong to same process group!
- Shell has the notion of a **job**: "commands" started interactively.
  - All processes within the same job are in the same group; ***let's see what this means.***
- Relevant for **`penn-shell`**

Process Group ID

- Process Group ID is set from an *initial PID!*
  - The PGID is equal to the PID of the first forked process in that job!
  - If the initial process (who's PID == PGID) is terminated, this PID still can't be reused.
    - ***That process ID will be reserved until the group is done***

```
int setpgid(pid_t pid, pid_t pgid);
```

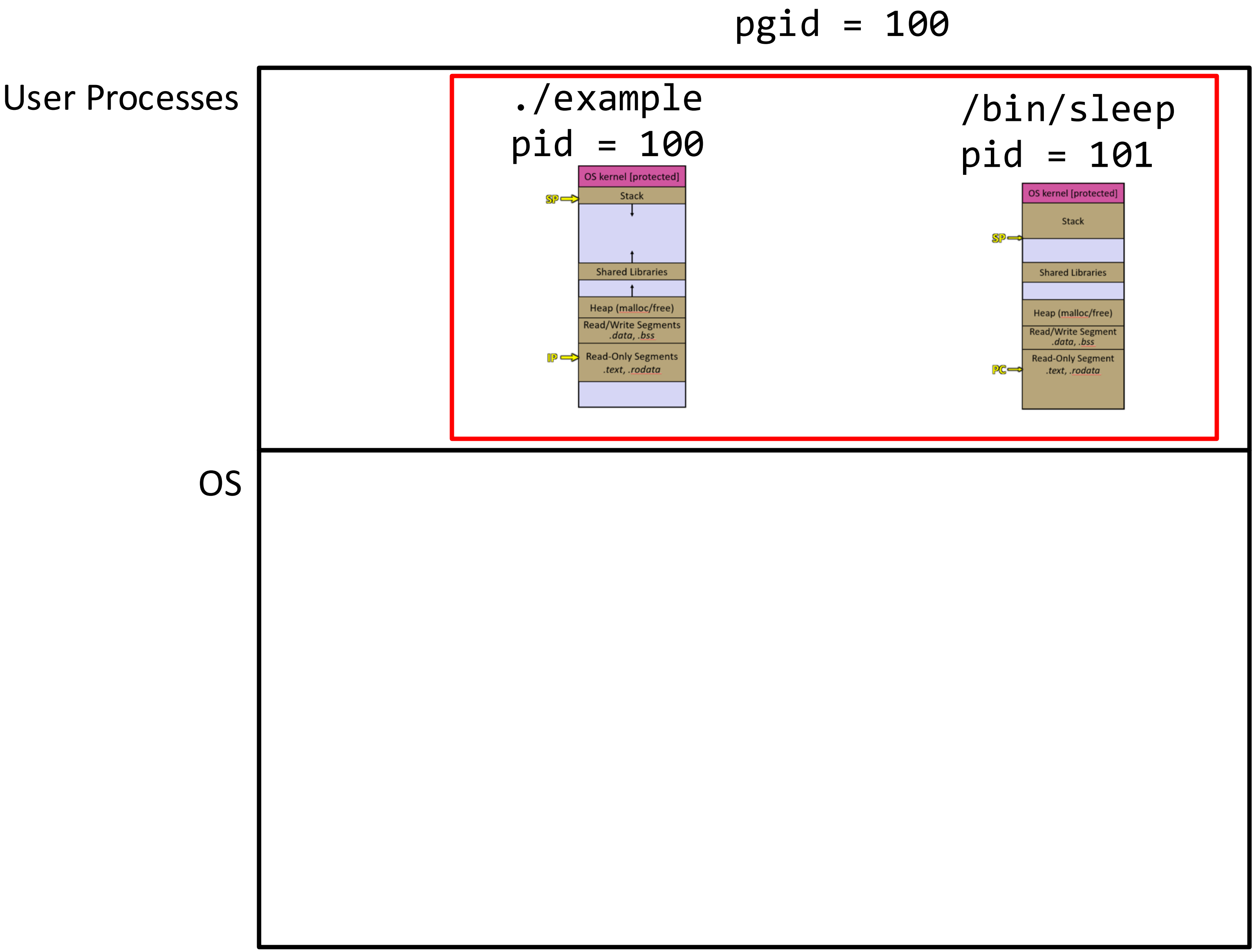
- The PGID of the process, *pid, is set* to *pgid*.
  - If *pid* is zero, then the process ID of the calling process is used.
- If *pgid* is zero, then the PGID of the process specified by *pid* is made the same as its process ID.

Process Group ID

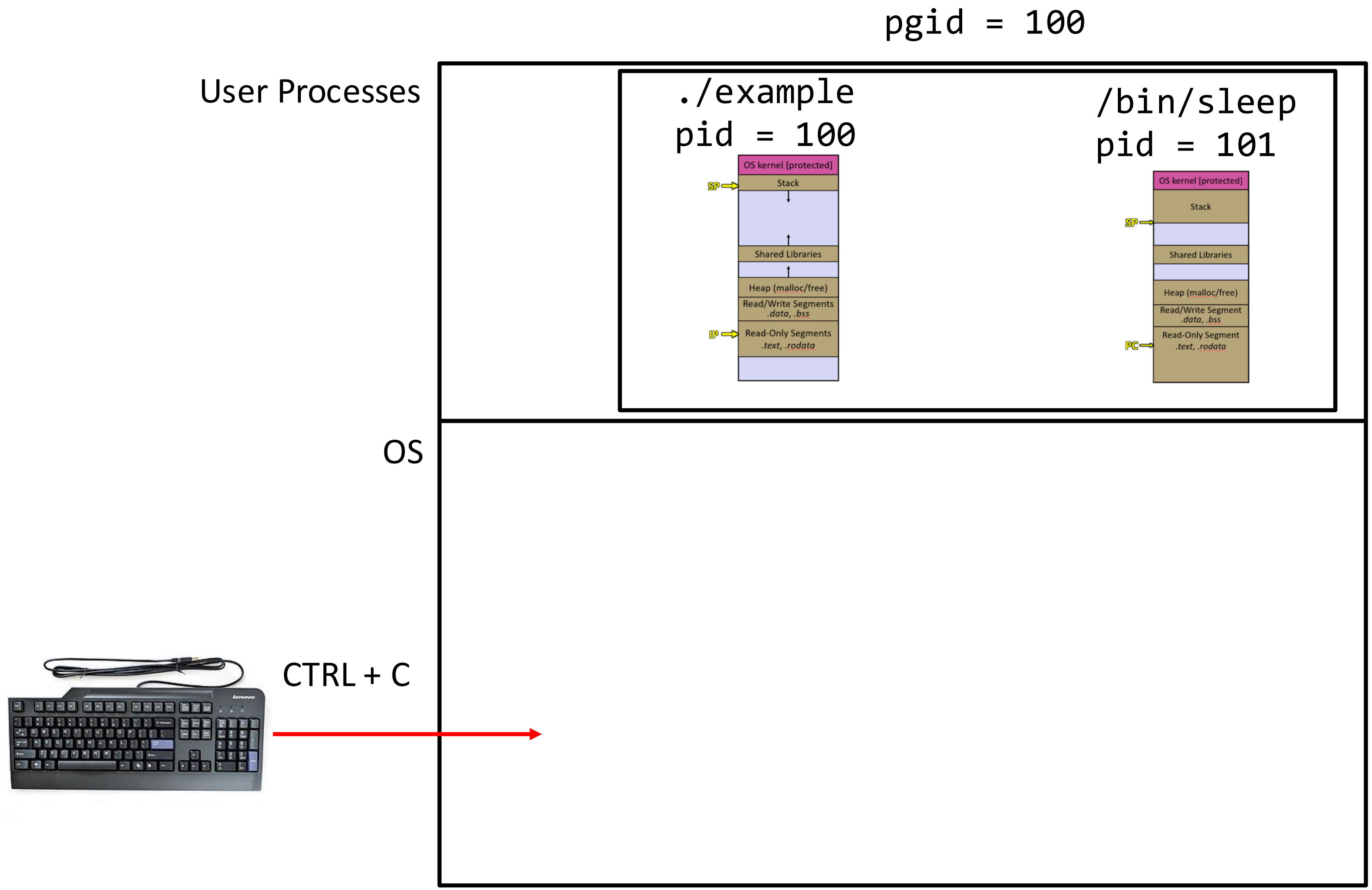
```
pid_t getpgid(pid_t pid);
```

- Returns the PGID of the process specified by *pid*.
  - returns -1 if error occurred.
- If *pid* is zero, the process ID of the calling process is used.

Example 1: Same PGID

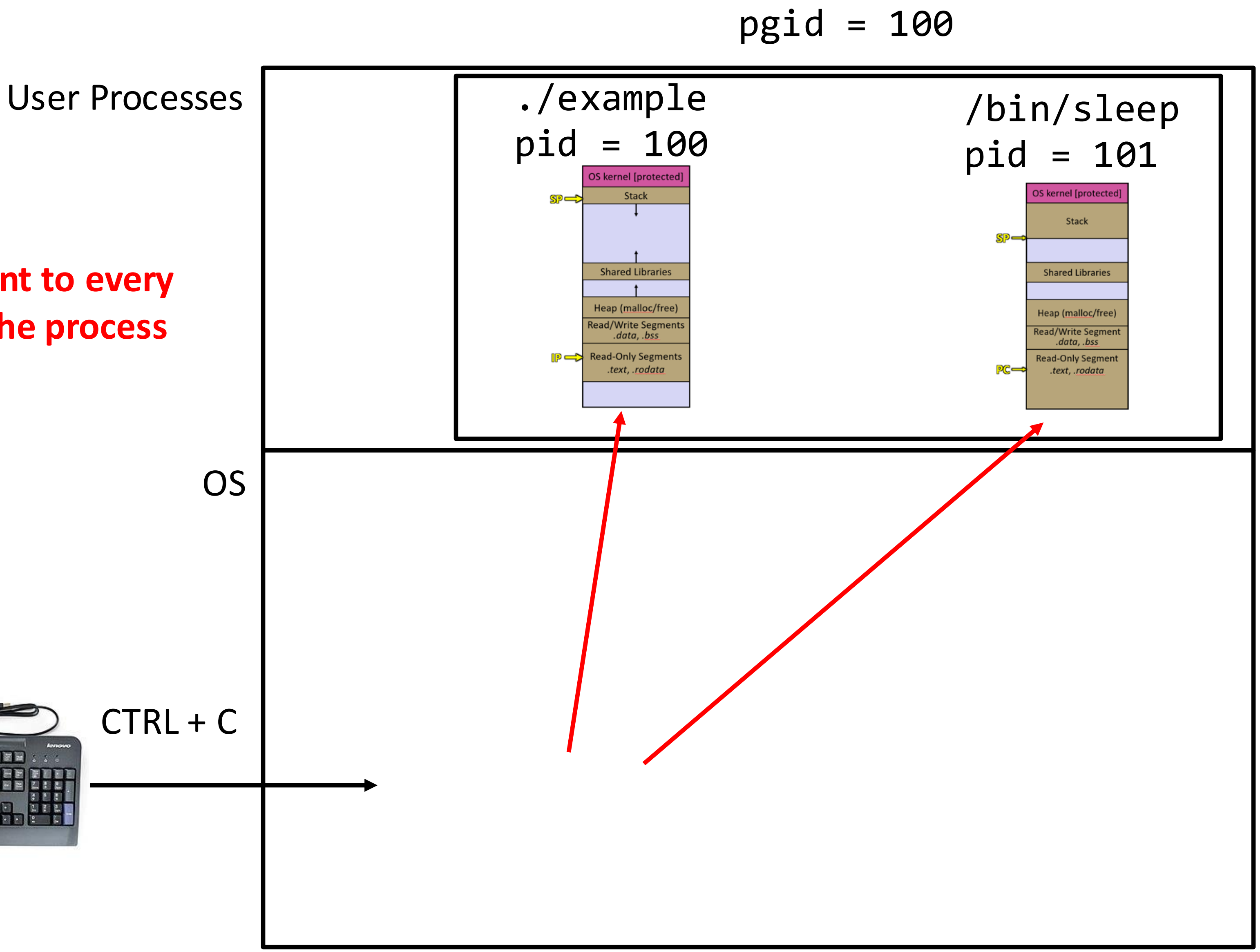


Example 1: Same PGID

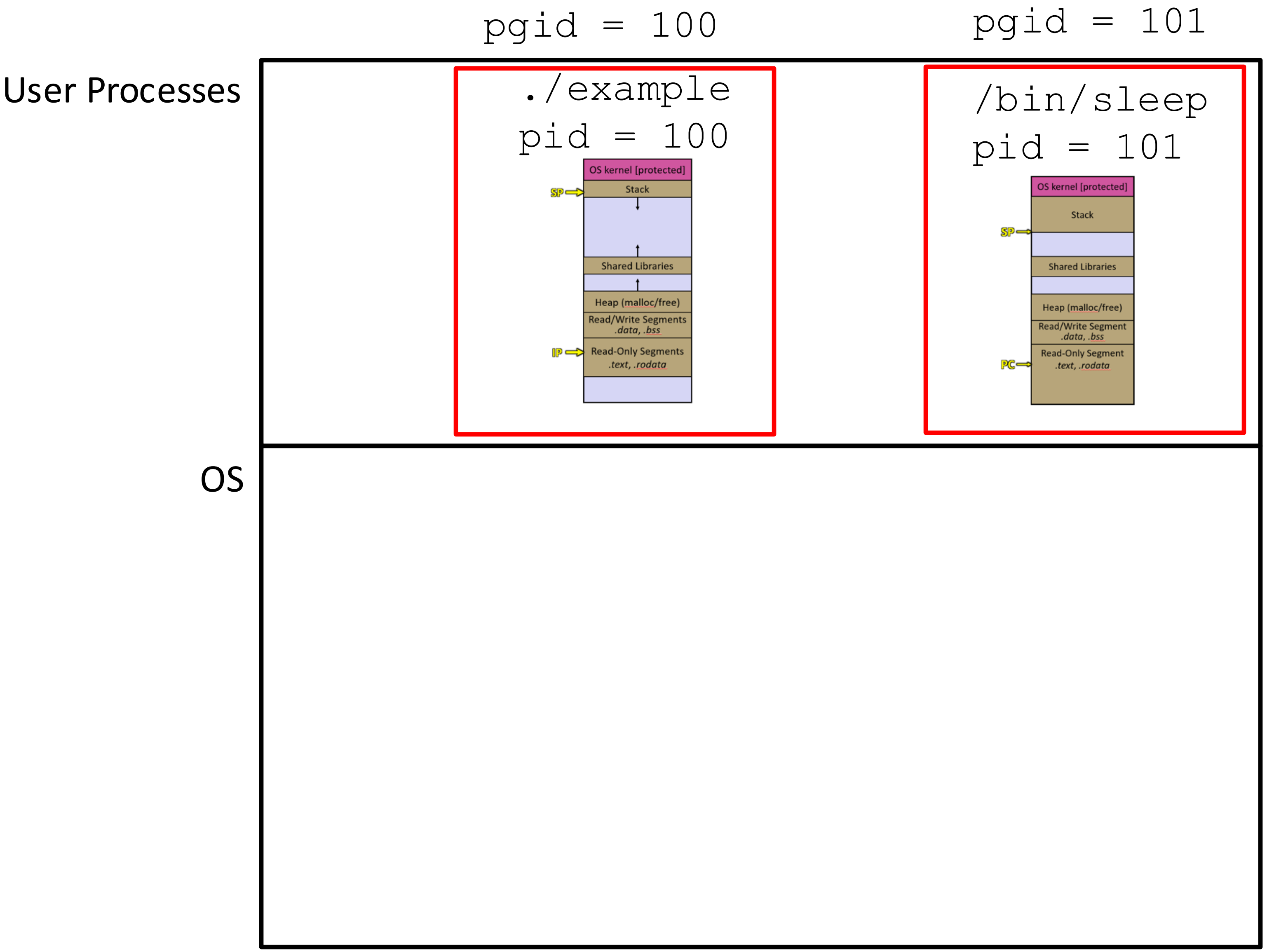


Example 1: Same PGID

SIGINT is sent to every Process in the process group

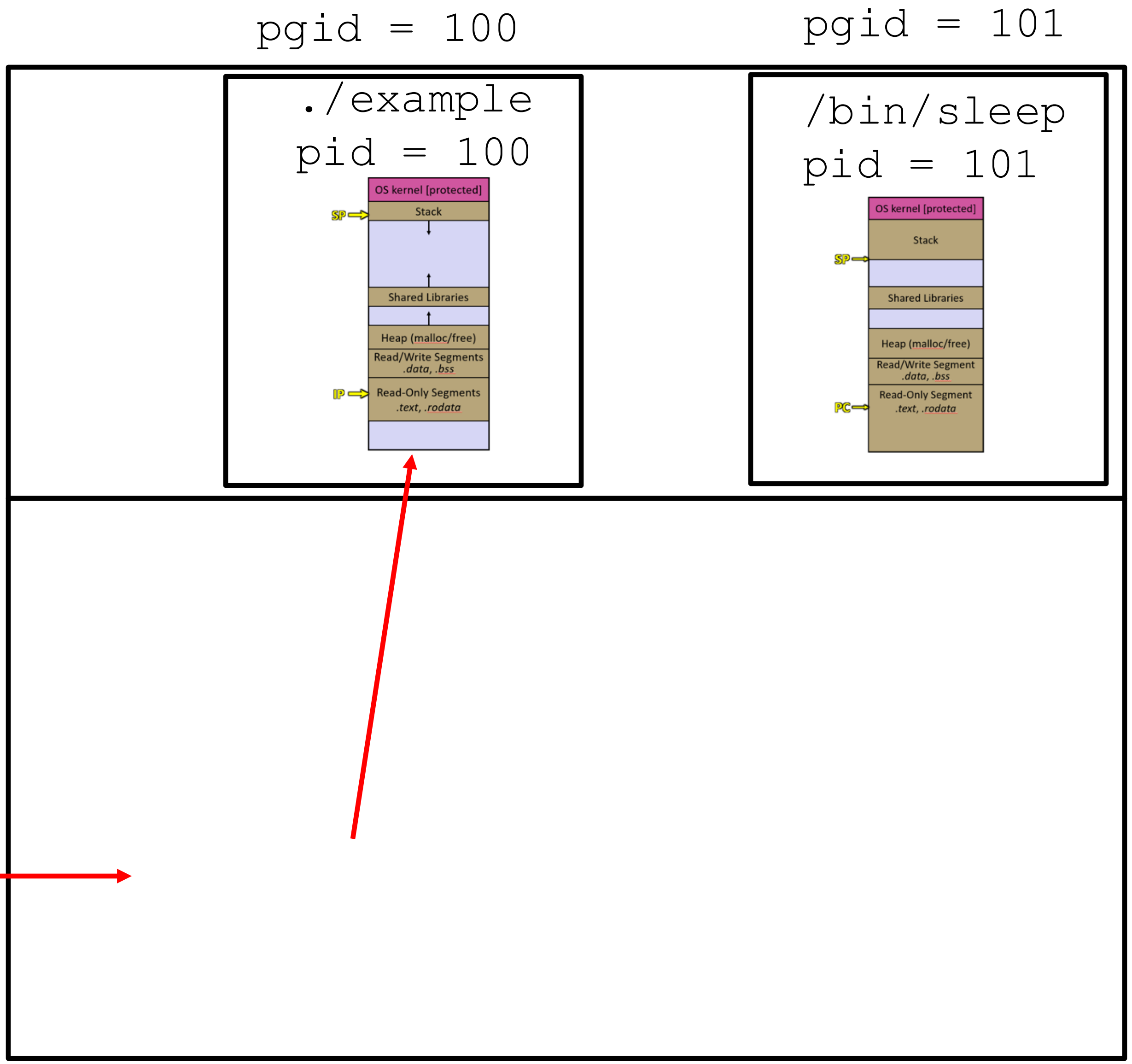


Example 2: Different PGIDs



Example 2: Different PGIDs

User Processes



SIGINT is sent to every Process in the process group

Child is in a separate group

OS

CTRL + C

waitpid & kill with PGIDs

- Instead of using a pid to refer to a singular process, you can pass in **-PGID** to **kill()** and **waitpid()**

```
int kill(pid_t -pgid, int signal);
```

- Doing so for **kill()** will send the signal to ***all processes in the group***

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Doing so for **waitpid()** will wait for any process in the group

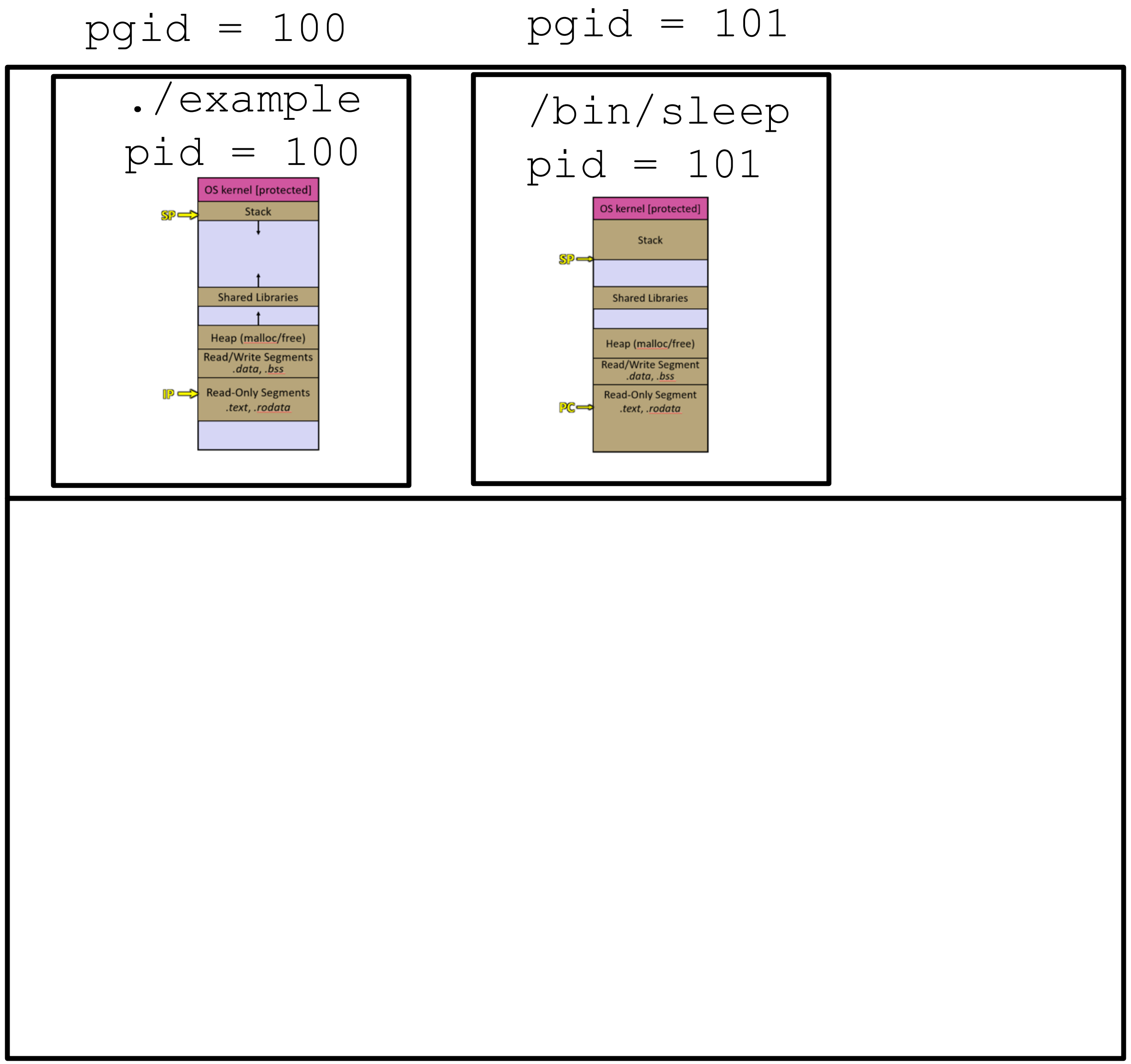
Wait; why does the PGID need to be negative?

You *may* find this useful for proj1: **penn-shell**

Example: pid vs –pgid

parent process on the left and a child process in its own group on the right

What if the parent forks a second child and adds it to the other child's group?



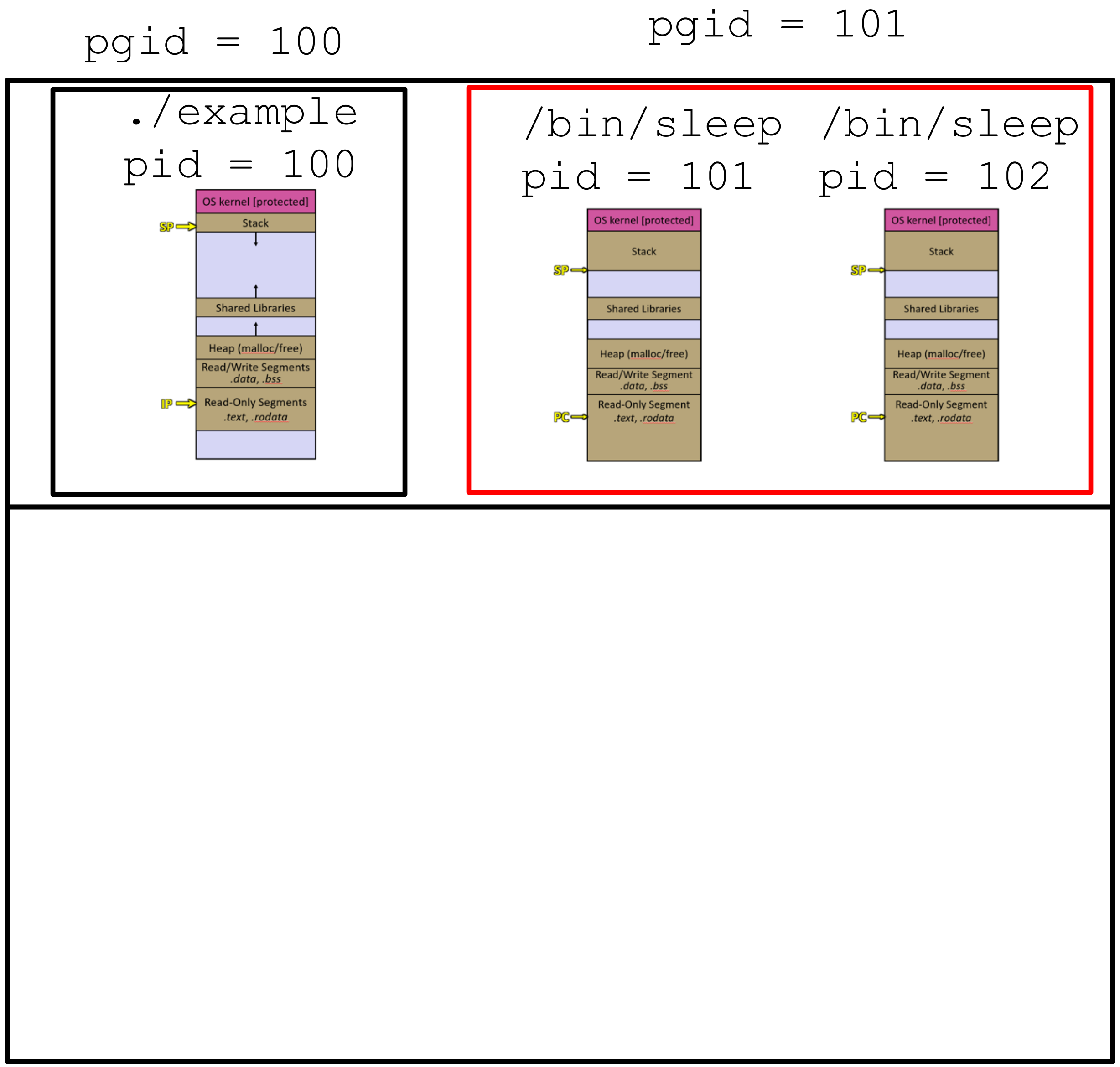
Example: pid vs –pgid

User Processes

parent process on the left and a child process in its own group on the right

What if the parent forks a second child and adds it to the other child’s group?

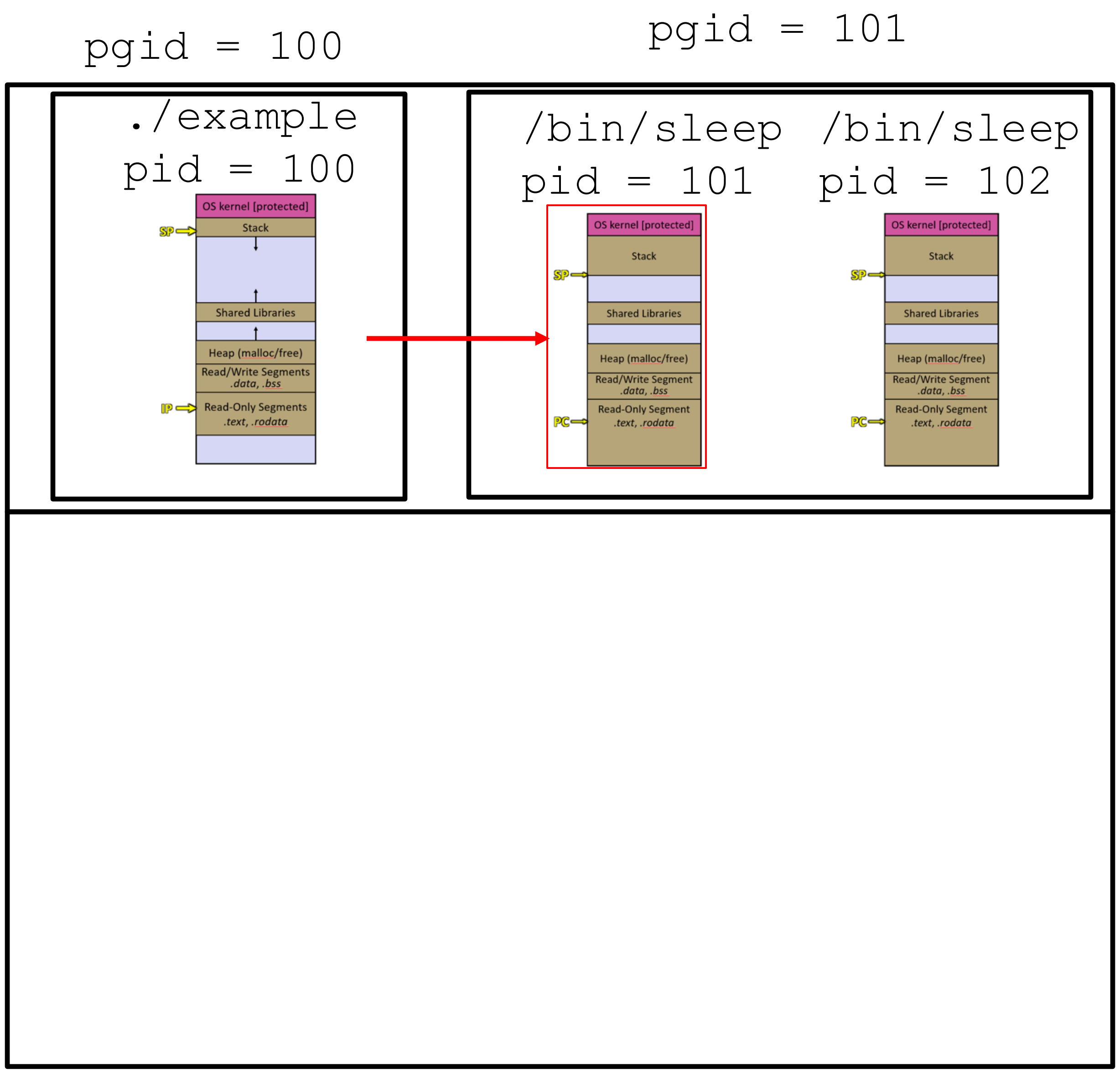
OS



Example: pid vs –pgid

User Processes

OS



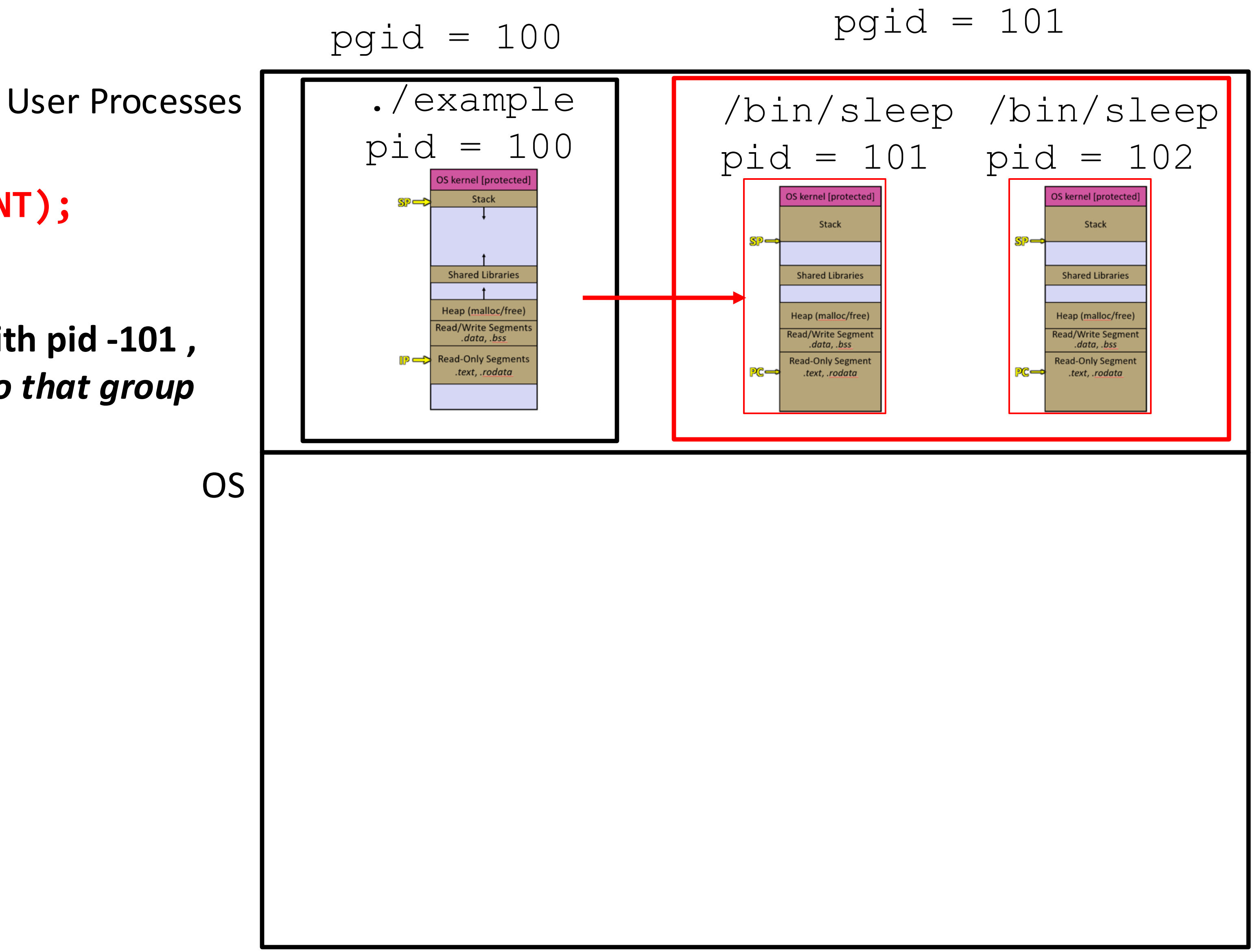
`kill(101, SIGINT);`

If the parent calls kill with pid 101 , *only* the child with that pid receives the signal

Example: pid vs –pgid

`kill(-101, SIGINT);`

If the parent calls kill with pid -101 , *all children belonging to that group are killed*



Demo: pgrpg_signals.c

- See code demo: `pgrp_signals.c`
  - Handler registered for SIGINT in both child and parent
  - Parent puts child in its own group
  - CTRL + C is input -> parent signal handler is invoked -> parent relays the signal to the child

  - What happens if we don't call kill in parent handler?
  - What happens if we then don't put child in its own group?

Lecture Outline

- Process Groups
  - `setpgid()`
- **Terminal Control**
  - `tcsetpgrp()`
- SIGSTOP
- Project 1: Synch vs Asynch wait
  - `SIGCHLD`

What if the child tried to use the terminal?

- Demo!
  - Let’s try to write a program so that the child does “cat”
    - (read from stdin, echo it to stdout until EOF)
    - First let’s see what cat is supposed to do.

What if the child tried to use the terminal?

- Demo!
  - Let's try to write a program so that the child does "cat"
    - (read from stdin, echo it to stdout until EOF)
    - First let's see what cat is supposed to do.

It doesn't work.

Let's try to peel back the layers to see why it doesn't.

Sessions

- A **Session** is a collection of process groups
  - A session can be attached to a controlling terminal
    - However, ***only one process group within the session can have control of the terminal***
  - Or not attached to any terminal (daemon's)

- You can think of a session as mostly associated with a "login" or instance of a terminal application. Each login/terminal is a singular session

- Within a session (that has a controlling terminal) there are
  - Background processes
    - These do not have have access to the terminal, and can not read from it.
  - ***Foreground processes***
    - ***These can read and write to their hearts content.***

Foreground Process Groups

- Foreground process groups (i.e., Foreground Jobs) can read from STDIN and the processes in that group receive the signals from the keyboard
- A foreground group (the shell truly) can make another group the foreground with the function:

```
int tcsetpgrp(int fd, pid_t pgrp);
```

  - **`fd`** is a file descriptor associated with the controlling terminal (stdin)
  - Sets the process group specified by **`pgrp`** to be the foreground process group
    - Essentially, this process group (or job from the perspective of the shell), is the star of the show.
  - `-1` returned on error, `0` when successful

Background in the shell

- To start a background job in the shell (and in penn-shell) run the command with a & at the end.
  - **`sleep 10 &`**
- While a command is running in the background, we can run other commands in the shell
  - So, while another command is using the terminal for Input, the background jobs/processes can not.
- Can use the **`jobs`** command to see the status of the jobs we have started

Process Groups and Controlling the Terminal

```
int setpgid(pid_t pid, pid_t pgid);
```

- When you make a process have it’s own process group, ***it no longer has the ability to read from the terminal.***
  - *It no longer is the process group who* **controls the terminal…**
- So, yes, jobs need to have their own groups, but they also need to navigate control of the terminal.
  - (This ***kinda*** makes sense. You don’t want 100 processes trying to read the terminal at the same time. What if what is in the terminal isn’t for them? (aka, what if it is your super secret password that you’re typing in?)

Background Process

- If a background process tries to read from `stdin`, the OS sends the signal `SIGTTIN` to the background process
  - The Disposition of SIGTTIN is to suspend/stop the program.
  - **Check it out for yourself: cat &**
- If a process in the background background calls `tcsetpgrp()`, the OS will send the entire process group a SIGTTOU signal.
  - If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent…might be important…
- Writing to **stdout** from the background is ok, but can be configured so that background processes get **SIGTTOU**
  - **The Dispositon of SIGTTOU is to Stop the program.**
  - Check it out for yourself: `cat file.txt &` (this is totally fine.)

Let’s try to fix our code from before!

- See code demo: `cat.c`
  - Let’s try to fix our process group code so that it can run cat ☺
    - Remember, printing to the terminal is fine. It’s reading that causes the issues.
    - So, we’ll go ahead and see if this holds true!

  - ***How can we make the parent take back the terminal control?***
    - ***If a process is done running in the foreground, then penn-shell should resume control.***

Poll Everywhere

pollev.com/cis5480

```
pid_t pid = fork();
if (pid == 0) {
    char* args[] = {"cat", NULL};
    execvp(args[0], args);
    exit(EXIT_FAILURE);

}

// put the child in its own process group
if (setpgid(pid, pid) == -1) {
  perror("setpgid\n");
  exit(EXIT_FAILURE);
}

// give terminal to the child
if(tcsetpgrp(STDIN_FILENO, pid) == -1) {
  perror("tcsetpgrp\n");
  exit(EXIT_FAILURE);
}

printf("starting to wait\n");

int wstatus;
waitpid(pid, &wstatus, 0);
```

Is there a race condition here?

Poll Everywhere

pollev.com/cis5480

```c
pid_t pid = fork();
if (pid == 0) {
    char* args[] = {"cat", NULL};
    execvp(args[0], args);
    exit(EXIT_FAILURE);

}

// put the child in its own process group
if (setpgid(pid, pid) == -1) {
  perror("setpgid\n");
  exit(EXIT_FAILURE);
}

// give terminal to the child
if(tcsetpgrp(STDIN_FILENO, pid) == -1) {
  perror("tcsetpgrp\n");
  exit(EXIT_FAILURE);
}

printf("starting to wait\n");

int wstatus;
waitpid(pid, &wstatus, 0);
```

Is there a race condition here?

YES! YOU CAN NOT CHANGE THE PGID AFTER A PROGRAM HAS EXECED!

By the time a process (or a set of them for a job) have exec'd they should all have their PGID set up.

This is PARAMOUNT to setting up the stage for multiple pipes as well. They should all be a part of the same group prior to execi'ng.

Race Condition: `setpgid();`

- You can not change the PGID of a process after it has been exec'd.
  - Trying to do so will result in a failed setgpid with error: EACCES
- This is because we are at the mercy of the schedular
  - We don't know if a child will be exec'd before the parent can change it's PGID.
- To be safe, we must call PGID from both the parent and the child.

```c
// put the child in its own process group
if (setpgid(pid, pid) == -1) {
    perror("setpgid\n");
    exit(EXIT_FAILURE);
}
```

```c
if (pid == 0) {
    // child
    // reads from the terminal and
    // prints what it reads until EOF
    setpgid(0, 0); //sets it's pgid to be it's own pid.
    char* args[] = {"cat", NULL};
    execvp(args[0], args);
    exit(EXIT_FAILURE);
}
```

Caveat: remember, the initial process is the one who is the PGID of the entire group. You must keep track of this.

Poll Everywhere

pollev.com/cis5480

- What is the intention of this code? Does it do what it intends to do? How can we fix it?

```c
int main() {
  while (true) {
    fprintf(stderr, "give command: ");
    char c;
    ssize_t bytes  = read(STDIN_FILENO, &c, 1);
    if (bytes == -1) {
      perror("read\n");
      exit(EXIT_FAILURE);
    } else if (bytes == 0) {
      break;
    }

    if (c == 'c') {
      pid_t pid = fork();

      if (pid == 0) {
        // child
        // reads from the terminal and
        // prints what it reads until EOF
        char* args[] = {"cat", NULL};
        execvp(args[0], args);
        exit(EXIT_FAILURE);
      }
      // parent

      // put the child in its own process group
      if (setpgid(pid, pid) == -1) {
        perror("setpgid\n");
        exit(EXIT_FAILURE);
      }

      // give terminal to the child
      if(tcsetpgrp(STDIN_FILENO, pid) == -1) {
        perror("tcsetpgrp\n");
        exit(EXIT_FAILURE);
      }
      printf("starting to wait\n");

      int wstatus;
      waitpid(pid, &wstatus, 0);
    } else if (c == 's') {
      printf("sleeping...\n");
      sleep(5);
      printf("awake\n");
    } else if (c == 'p') {
      printf("HOWDY\n");
    }
  }
```

Demo: tc_loop.c

- See code demo: `tc_loop.c`
  - The code from the poll
  - Let's try to fix it…

  - **How can we make the parent take back the terminal control?**

Lecture Outline

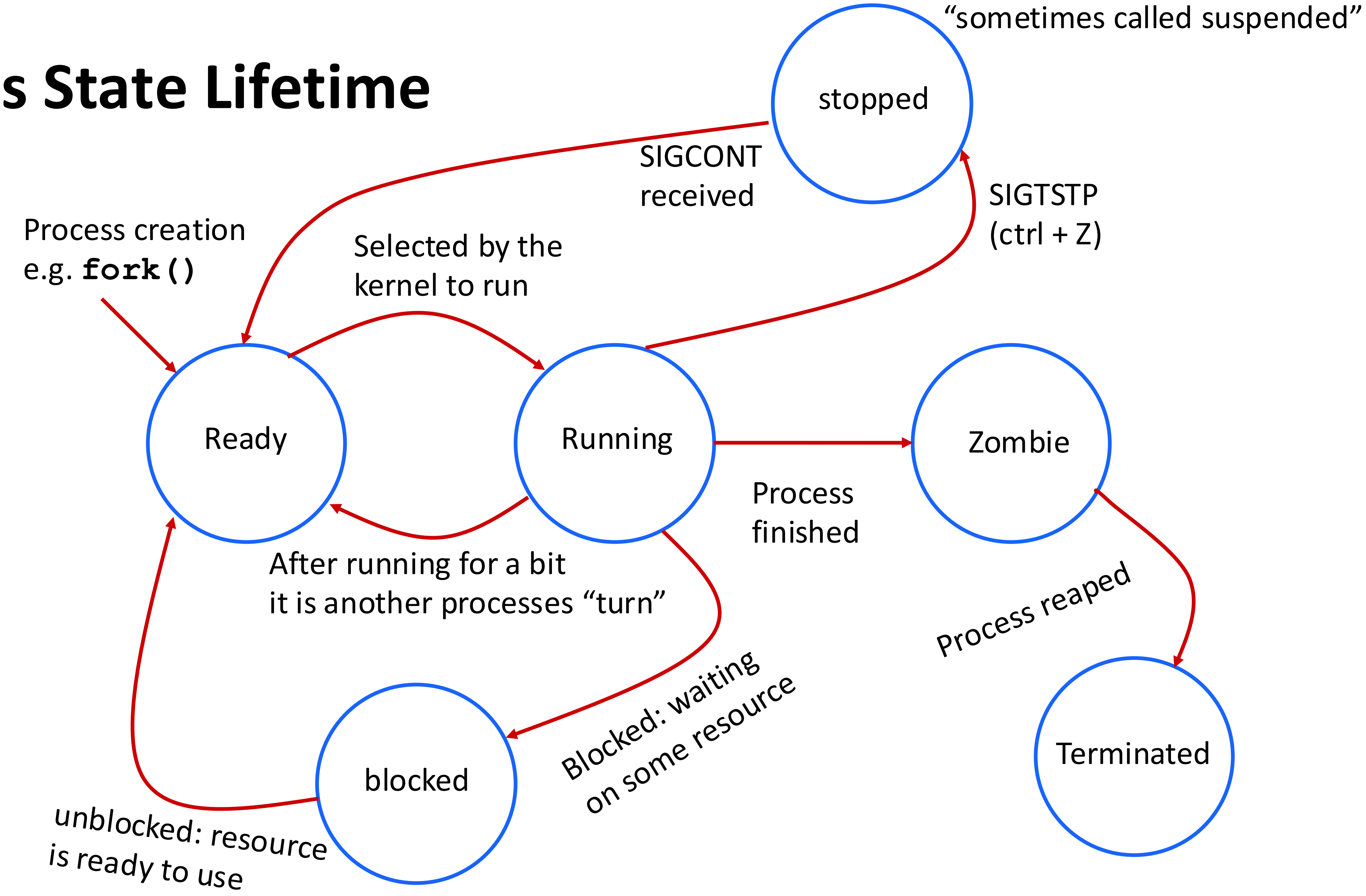
- Process Groups
  - **`setpgid()`**
- Terminal Control
  - `tcsetpgrp()`
- **SIGSTOP**
- Project 1: Synch vs Asynch wait
  - `SIGCHLD`

Stopped Jobs

- Processes can be in a state slightly different than being blocked. *// This is relevant for* `penn-shell`
  - When a process gets the signal `SIGSTOP`, the process will not run on the CPU until it is resumed by the `SIGCONT` signal
  - Other signals can still stop a program by default, like `SIGTSTP` or `SIGTTOU`

- Demo:
  - In terminal: `ping google.com`
  - Hit `CTRL + Z` to stop
  - Command: "`jobs`" to see that it is still there, just stopped
  - Can type either "`%<job_num>`" or "`fg`" to resume it

Process State Lifetime



Lecture Outline

- Process Groups
  - **`setpgid()`**
- Terminal Control
  - `tcsetpgrp()`
- SIGSTOP
- **Project 1: Synch vs Asynch wait**
  - **`SIGCHLD`**

Penn-shell

- Part of what you do in HW1 (after the milestone) is to make a shell that manages process groups in the foreground and background

- This means your code will have to handle multiple process groups at once, keeping track of the state of all of them.

- Need to maintain a linked list of the current jobs to handle job control

"Normal" approach Pseudo Code

- Discuss: what does this do?

- Is there a flaw in this?
  Not in correctness but maybe
  - Responsiveness
  - Resource utilization
  - etc.

```
int main(int argc, char* argv[]) {
  while(...) {
    printf(PROMPT);

    getline(&user_input);

    pid = fork_exec(user_input);

    waitpid(pid, &wstatus, 0);

    for (pid_t p : background) {
       // check status of background
       waitpid(p, &wstatus, WNOHANG);
       // if there is an update,
       // need to update the lists…
    }
    // re-prompt user
  }
}
```

Analysis: "Normal"

- The “normal”: check background processes before re-prompting the user
  - may not be responsive to background processes finishing
  - Consider we have many background processes then the user runs `sleep 1000000` in the foreground...
  - those background processes will not be reaped until foreground finishes

"Polling" approach Pseudo Code

- Discuss: what does this do?
- How does this compare to the previous attempt?

```
int main(int argc, char* argv[]) {
  while(...) {
    printf(PROMPT);
    getline(&user_input);
    pid = fork_exec(user_input);

    while (waitpid(pid, &wstatus, WNOHANG) == 0) {
      for (pid_t p : background) {
        // check status of background
        waitpid(p, &wstatus, WNOHANG);
        // if there is an update,
        // need to update the lists…
      }
    }
    // re-prompt user
  }
}
```

Analysis: Polling

- Polling is a term used to describe when we check to see if something is ready, but do not block if it is not ready

- This approach is more responsive than the previous one…
- but it busy waits… consuming CPU cycles…

Aside: SIGCHLD

- This approach registers `SIGCHLD` as a handler, `SIGCHLD` is a signal that is sent when a child process stops or is terminated
  - Is ignored by default

"async" approach Pseudo Code

```
void handler(int signo) {
  for (pid_t p : background) {
    // check status of background
    waitpid(p, &wstatus, WNOHANG);
    // if there is an update,
    // need to update the lists…
  }
}
int main(int argc, char* argv[]) {
  //setting stuff up…
  sigaction(SIGCHLD, &sigact_handler, NULL);
  while(...) {
    printf(PROMPT);
    getline(&user_input);
    pid = fork_exec(user_input);
    waitpid(pid, &wstatus, 0);
    // re-prompt user
  }
}
```

- Discuss: what does this do?
- How does this compare to the previous attempt?

Analysis: Async

- This approach registers `SIGCHLD` as a handler, `SIGCHLD` is a signal that is sent when a child process stops or is terminated
  - Is ignored by default

- This allows us to respond quickly to the background children terminating
- No busy waiting! Main process instead is mostly blocked waiting on the foreground job
- Must use signal handlers and handle critical sections ☺
- Handling this ASYNC is your extra credit
  pass the normal autograder first PLEASE

Reminder: `sigsuspend()`

- Another way to approach handling async is to use `sigsuspend()`
  - May be a little harder to reason about; I find it to be a bit more intuitive…
  - Don't have to do much in the signal handler if this is the case!

You finally have everything you need for shell. Yay.