Pipe (Fin), Process Groups, Terminal Control!

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra
Khoi Dinh
Sydney Fitzgerald
Rania Souissi
Jeffrey Oduman
Abdullah Goher
Monisha Krothapalli
Veer Kakar
Kevin Xue
Shreya Mukunthan
Daniel Price
Wesley Liu
Leo Zhang

pollev.com/tqm

- Any questions/comments/concerns? (Especially about pipe)

Administrivia

- Second Assignment (penn-shell)
  - Milestone 0 due Friday 9/11 // you have everything you need for this
  - Milestone 1 due Friday 9/18 // you will have everything you need after this lecture
  - Full assignment due Wednesday 9/23 // after Today's lecture
  - Basic shell implementation
    - System calls, processes, I/O redirection, pipes, job control, and more!
  - Autograders are posted

- Information about interviews will be posted this week!

Lecture Outline

- **pipe2**
- Process Groups
  - `setpgid()`
- Terminal Control
  - `tcsetpgrp()`
- `SIGSTOP`
- Project 1: Sync vs Async wait
  - `SIGCHLD`

pipe2

```
int pipe2(int pipefd[2], int flags);
```

- Still creates a pipe, like `pipe`, but we can now specify behavior!
- Flags
  - O_CLOEXEC, your new friend Also a flag for open!
  - Enables the close-on-exec flag for the file descriptors
  - ***FDs with this flag enabled will be closed when that process execs***
    - Only closes the references from the ***exec'ing*** process!
    - File descriptors that are ***dup2'd*** with these FDs are not closed!
- Requires `#define _GNU_SOURCE`
  - Check the man page!
  - `pipe2()` is Linux-specific

pipe2

```
int pipe2(int pipefd[2], int flags);
```

- Here's an equivalent macro, for those not on Linux machines.

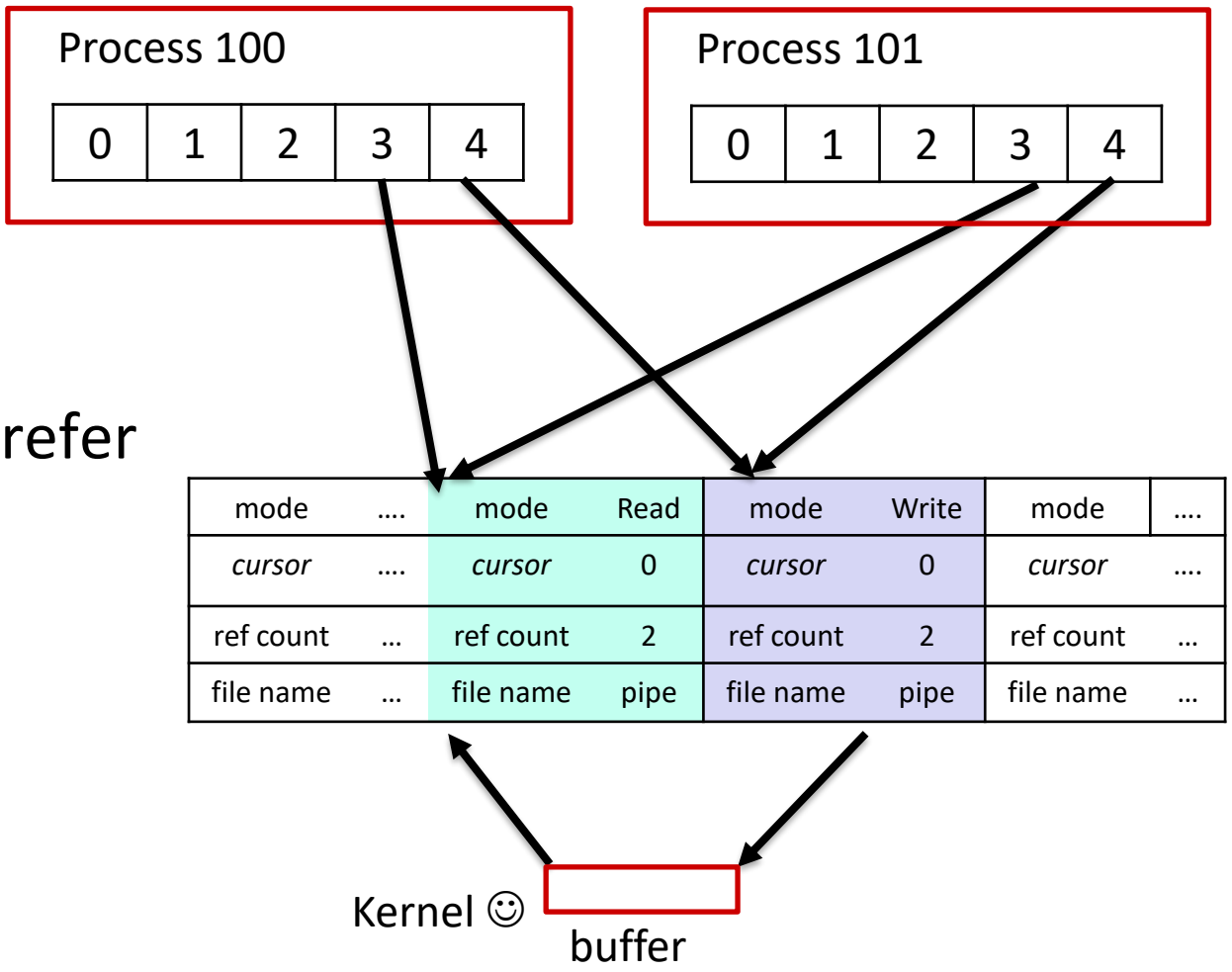
```
#define pipe2(FD, FLAG) \
pipe((FD)); \
fcntl((FD)[0], F_SETFD, FLAG); \
fcntl((FD)[1], F_SETFD, FLAG);
```

O_CLOEXEC Behavior

```
int pipe_fds[2];
pipe2(&pipe_fds, O_CLOEXEC);
pid_t cat_pid = fork();

if(cat_pid == 0){
    execvp(…);
}
// parent does some stuff.
```

- Prior to the execvp, both processes refer to the same pipe!

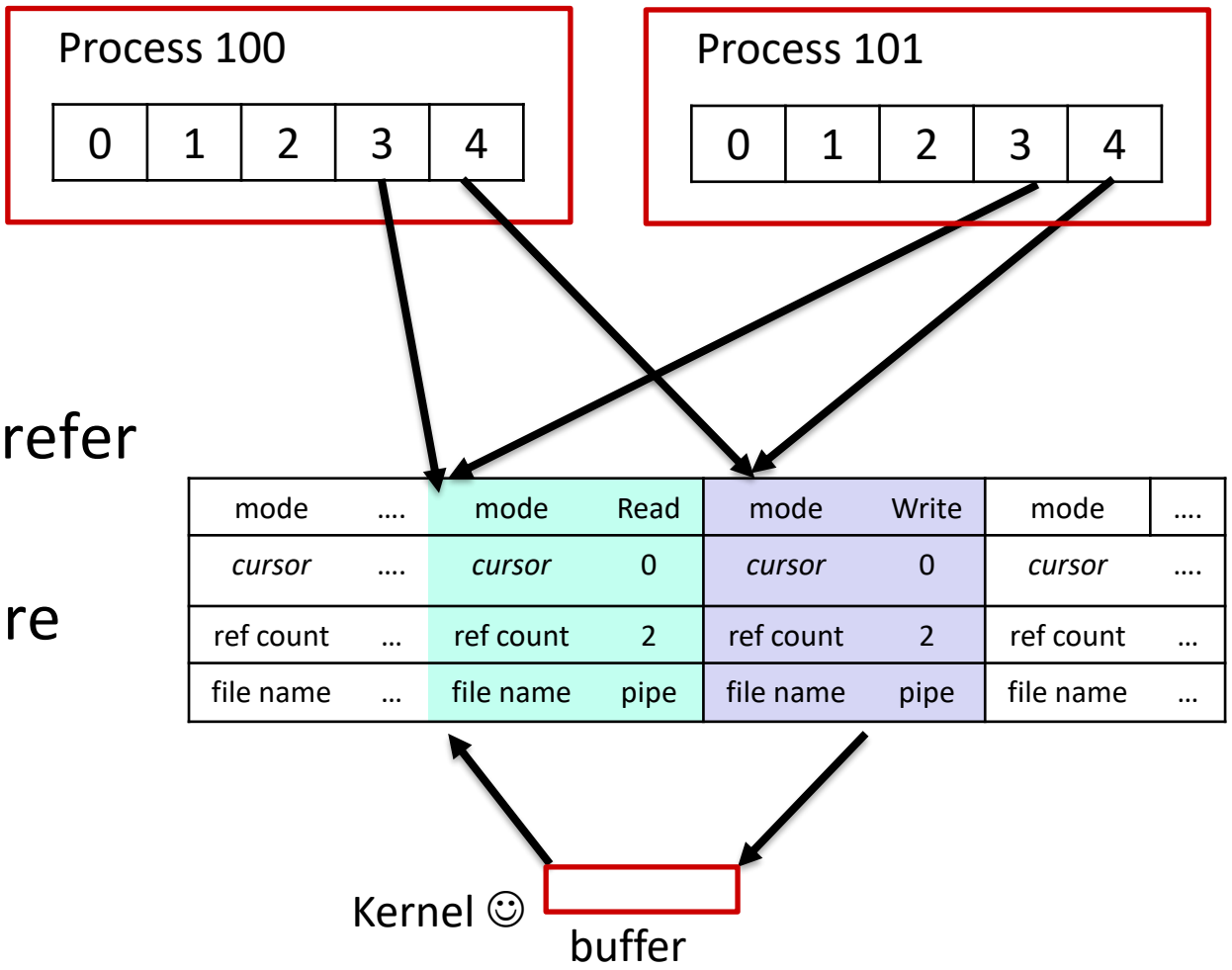


O_CLOEXEC Behavior

```
int pipe_fds[2];
pipe2(&pipe_fds, O_CLOEXEC);
pid_t cat_pid = fork();

if(cat_pid == 0){
    execvp(…);
}
// parent does some stuff.
```

- Prior to the execvp, both processes refer to the same pipe!
- Once the child execs, the pipe_fds are closed!

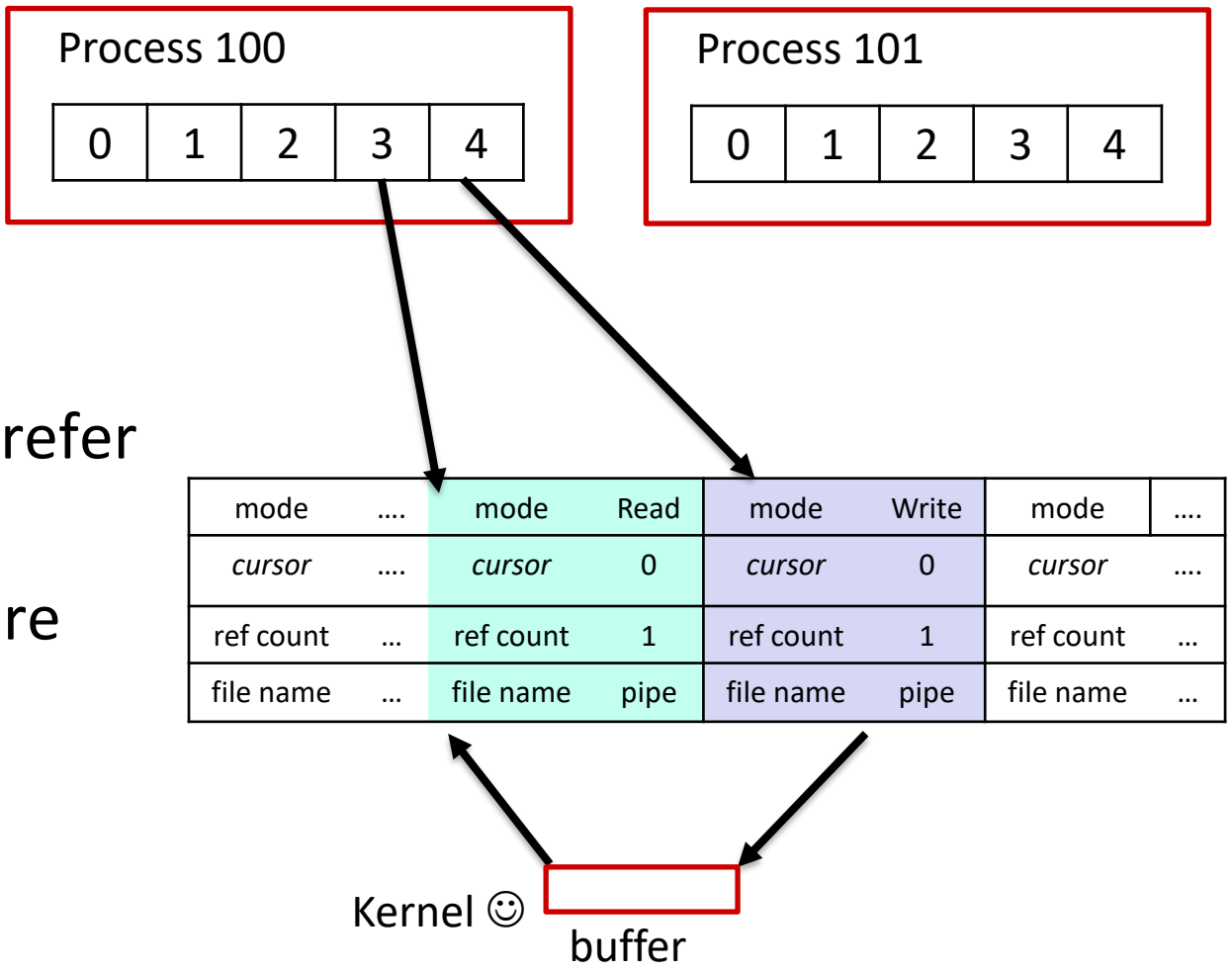


O_CLOEXEC Behavior

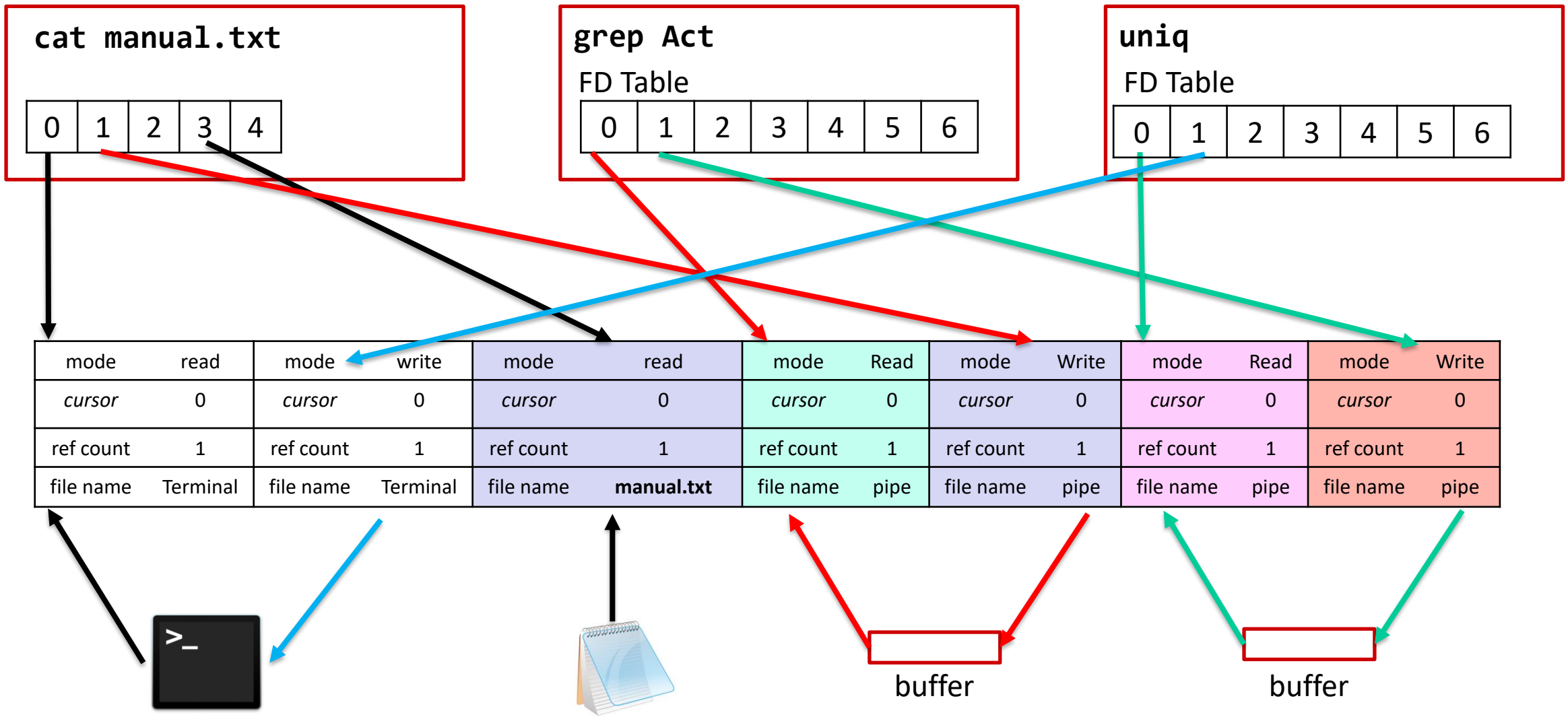
```
int pipe_fds[2];
pipe2(&pipe_fds, O_CLOEXEC);
pid_t cat_pid = fork();

if(cat_pid == 0){
    execvp(…);
}
// parent does some stuff.
```

- Prior to the execvp, both processes refer to the same pipe!
- Once the child execs, the pipe_fds are closed!



Let's see how pipe2 changes our code…



How would we implement these?

- **`cmd1 | cmd2`** creates a pipe so that the `stdout` of **`cmd1`** is redirected to the `stdin` of **`cmd2`**
  - E.g. **`history | grep valgrind`**

- **`cmd < file`** redirects `stdin` to instead read from the specified file
  - E.g. **`./penn-shell < script.txt`**

- **`cmd > file`** redirects the `stdout` of a command to be written to the specified file
  - E.g. **`grep -r kill > out.txt`**

How would we implement these?

- To use < and >, you would have to open these files on behalf of the executable, and then dup2 STDIN or STDOUT.

```
cat manual.txt > copy_manual.txt
```

Here, the output from *cat* that would normally go to STDOUT, now needs to be written to this new file that we must make or ***clobber.***

If it already exists, we just overwrite what is there.

Poll Everywhere

pollev.com/tqm

`cat manual.txt > copy_manual.txt`

To make this a possibility, what should the arguments to open be? Check the ***man*** Page…

```
char *output = "copy_manual.txt";

int copy_fd = open(output, ???????, 0644);
```

"Here, the output from *cat* that would normally go to STDOUT,
now needs to be written to this new file, ***we must make*** or ***clobber (rewrite from scratch).***"

Lecture Outline

- pipe2
- **Process Groups**
  - **`setpgid()`**
- Terminal Control
  - `tcsetpgrp()`
- `SIGSTOP`
- Project 1: Sync vs Async 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 kernel***
- 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.
  - How do we set a process' PGID to its own PID?

```
setpgid(0, 0);
```

Process Group ID

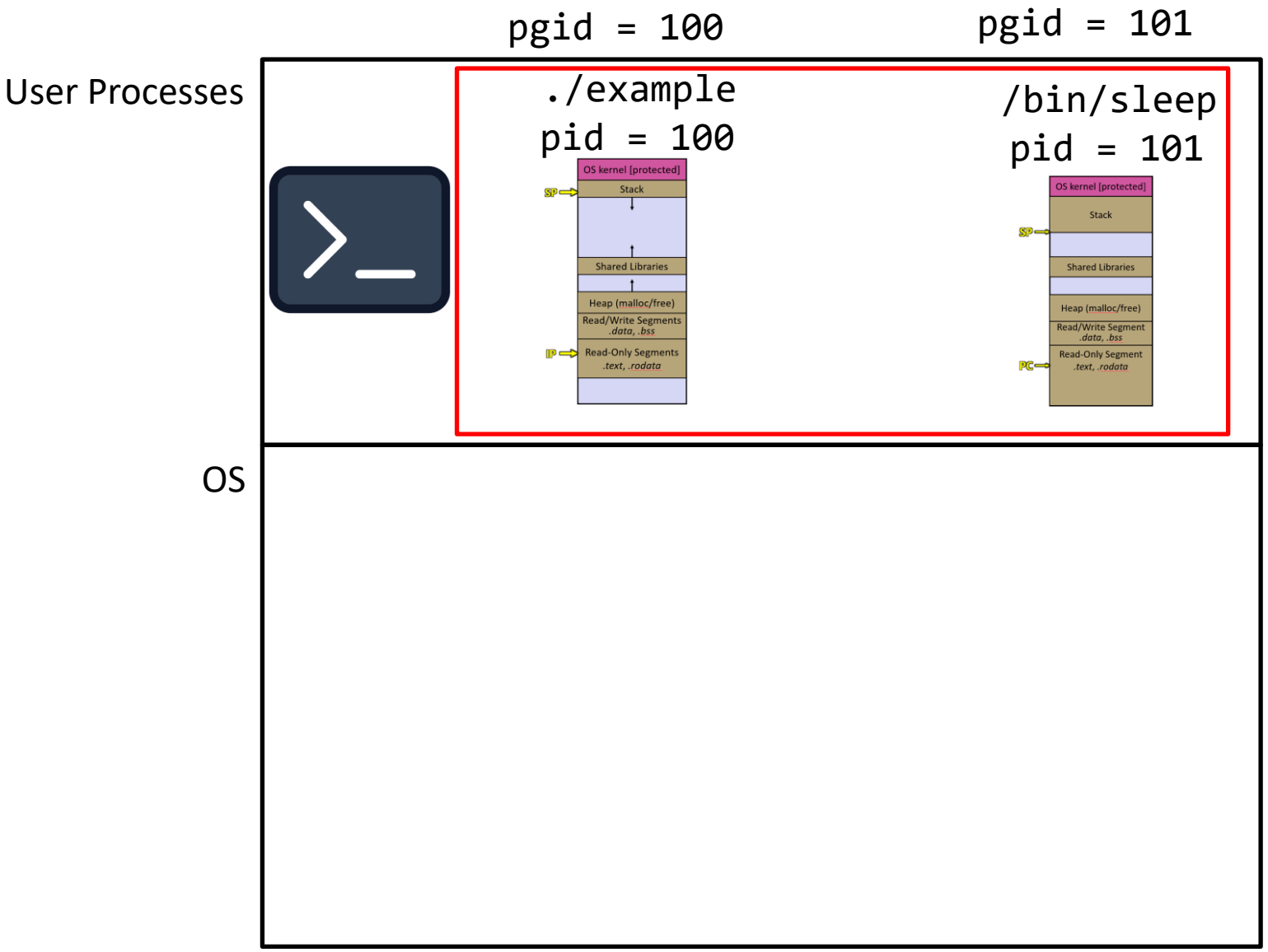
```
pid_t getpgid(pid_t pid);
```

- 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

But why change process groups?

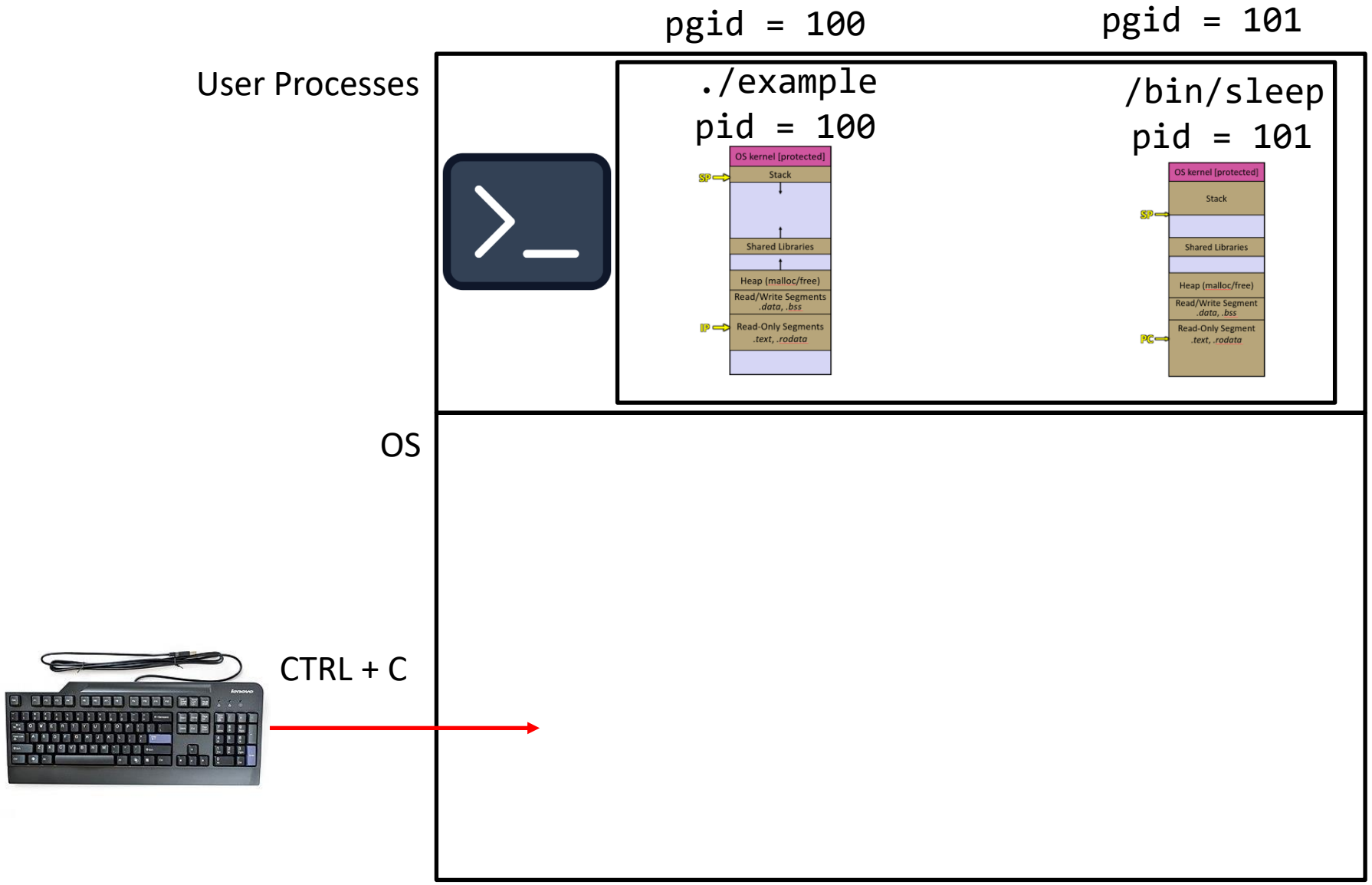
Changing process groups allows *you* to ***control*** who gets what signal and ***by what means.***

Example 1: Same PGID



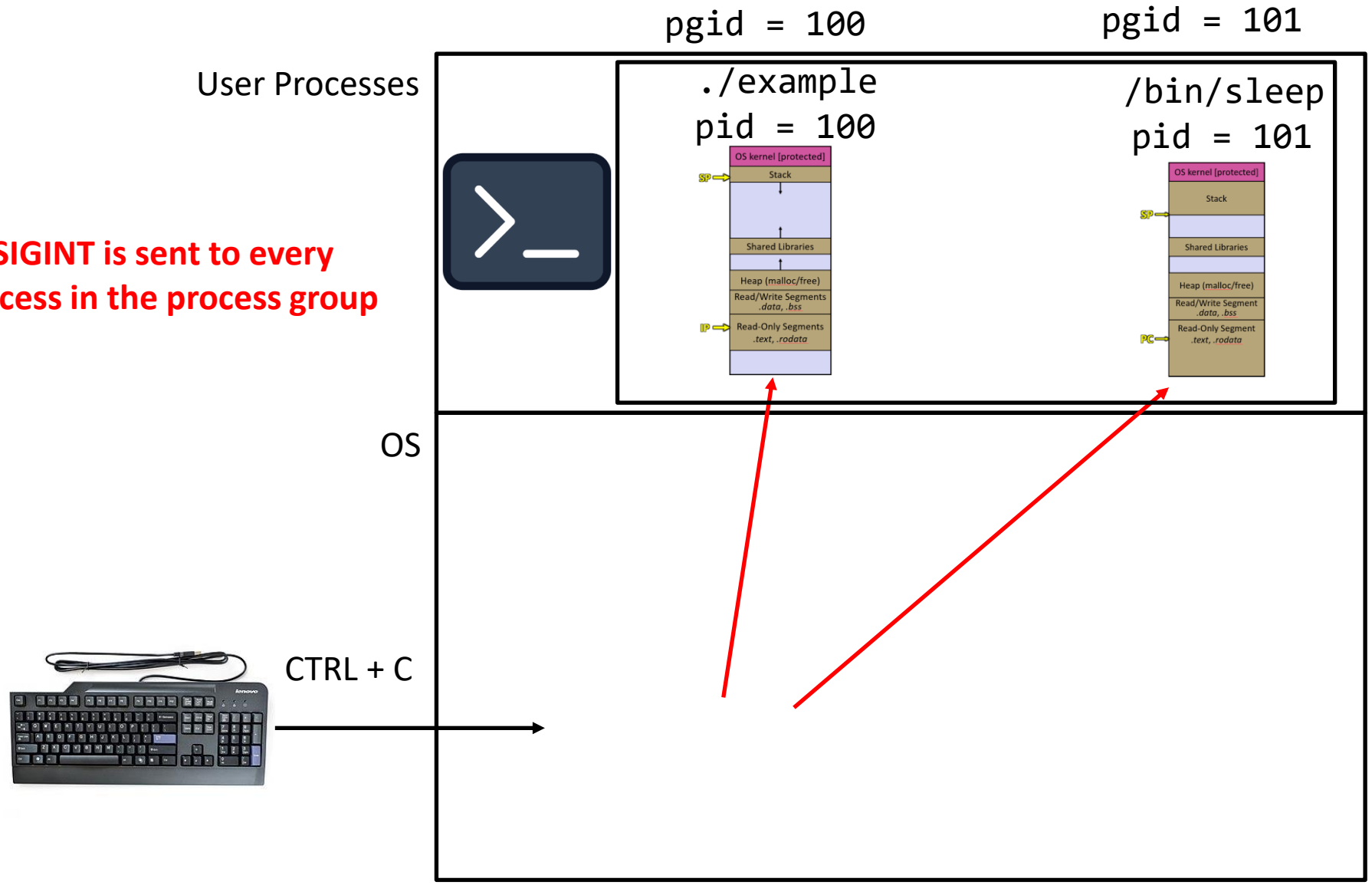
Both processes running in shell.

Example 1: Same PGID



Both processes running in shell.

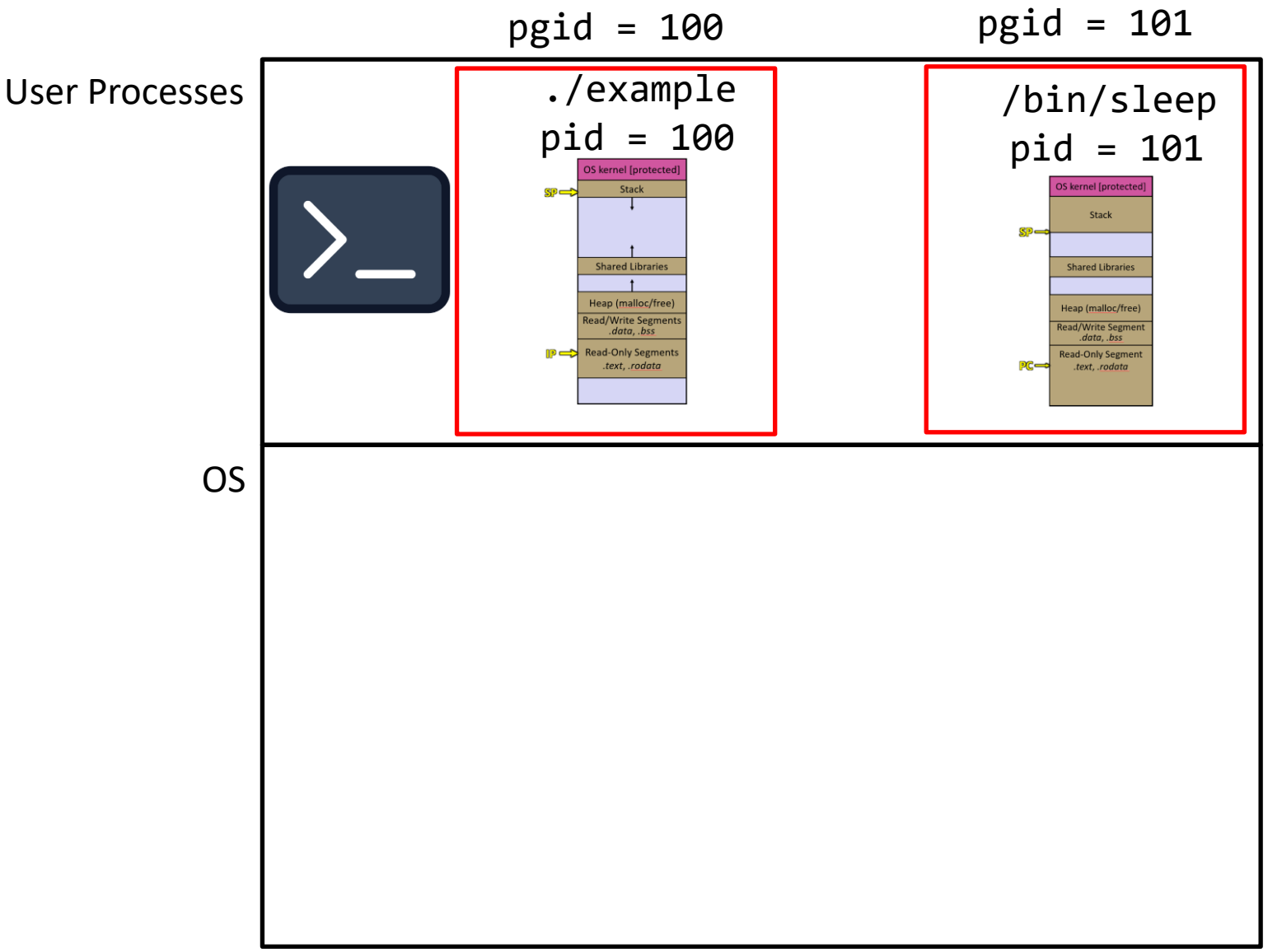
Example 1: Same PGID



SIGINT is sent to every Process in the process group

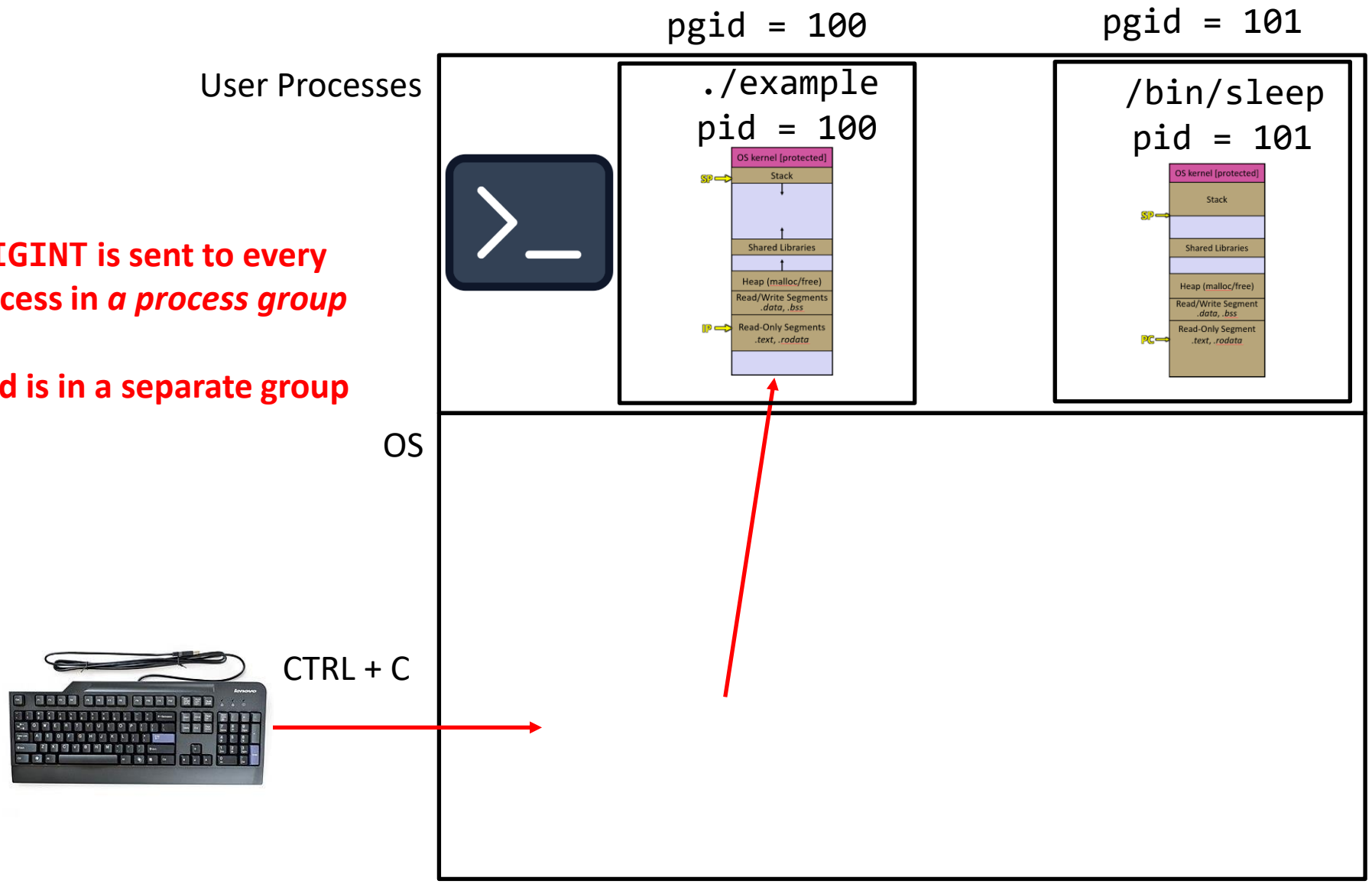
Both processes running in shell.

Example 2: Different PGIDs



Both processes running in shell.

Example 2: Different PGIDs

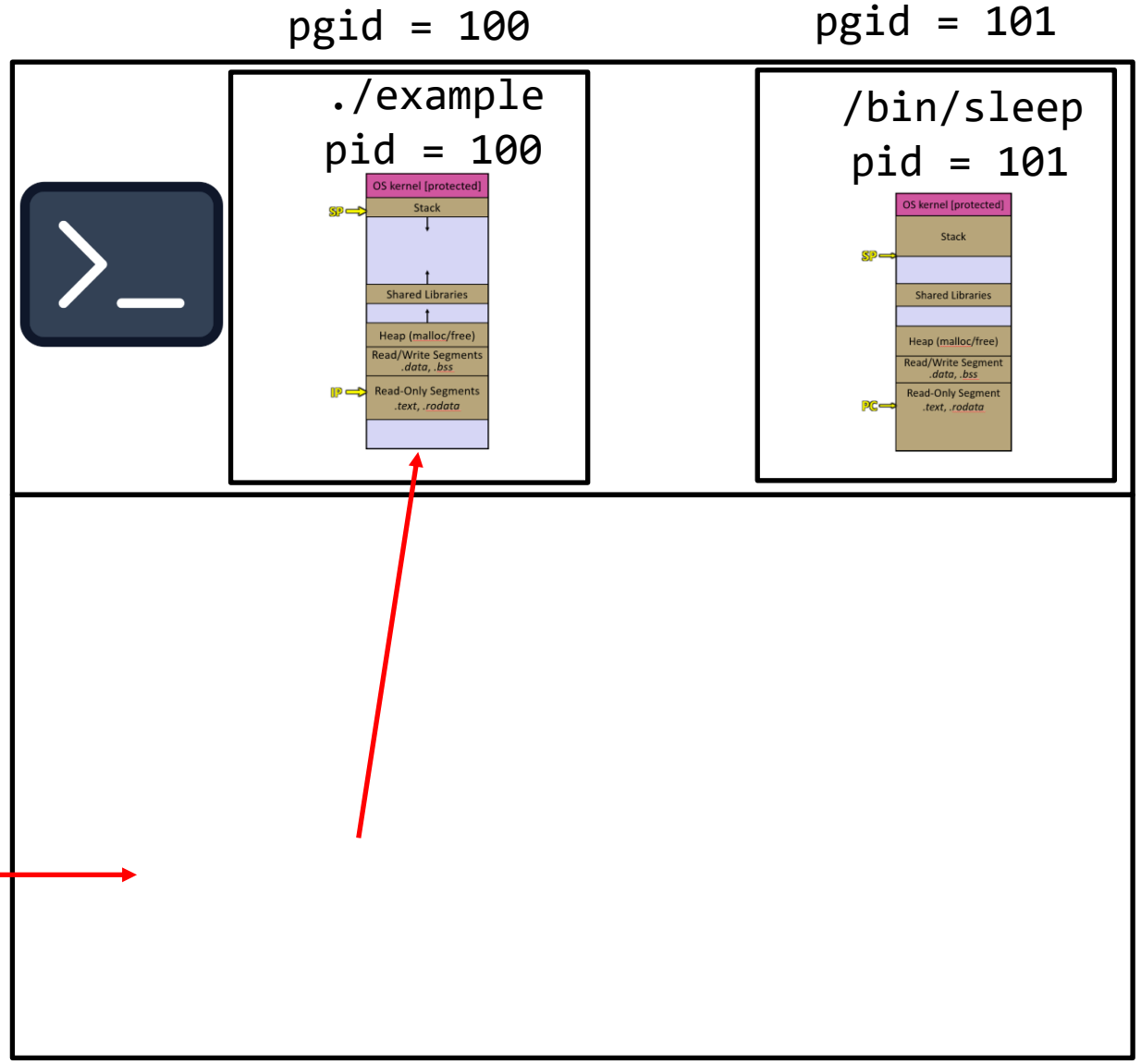


SIGINT is sent to every process in *a process group*

child is in a separate group

Both processes running in shell.

Example 2: Different PGIDs



Why is the signal only sent to one of the process groups we created?

Both processes running in shell.

`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

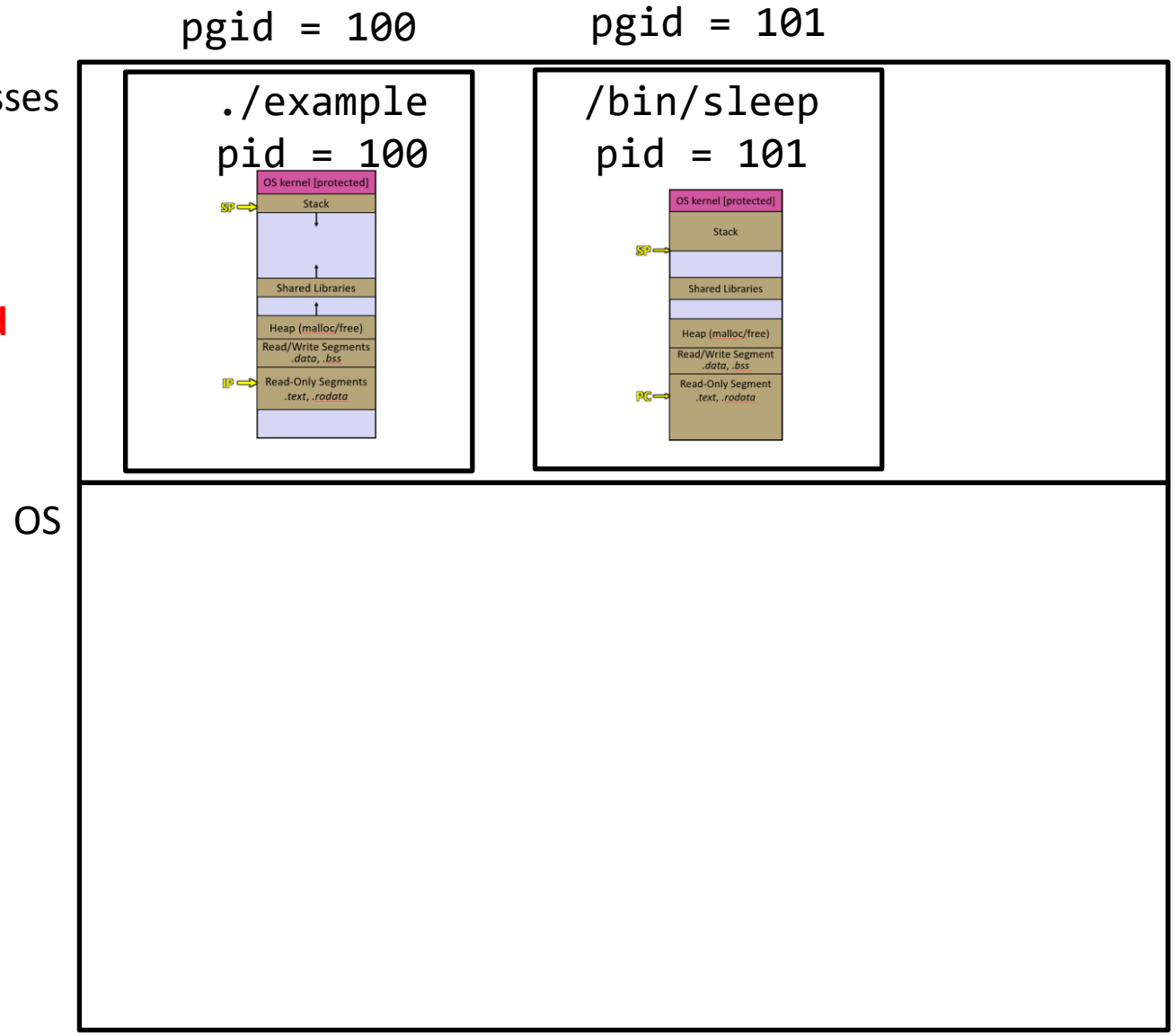
Wait - why does the PGID need to be negative?

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

Example: pid vs –pgid

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

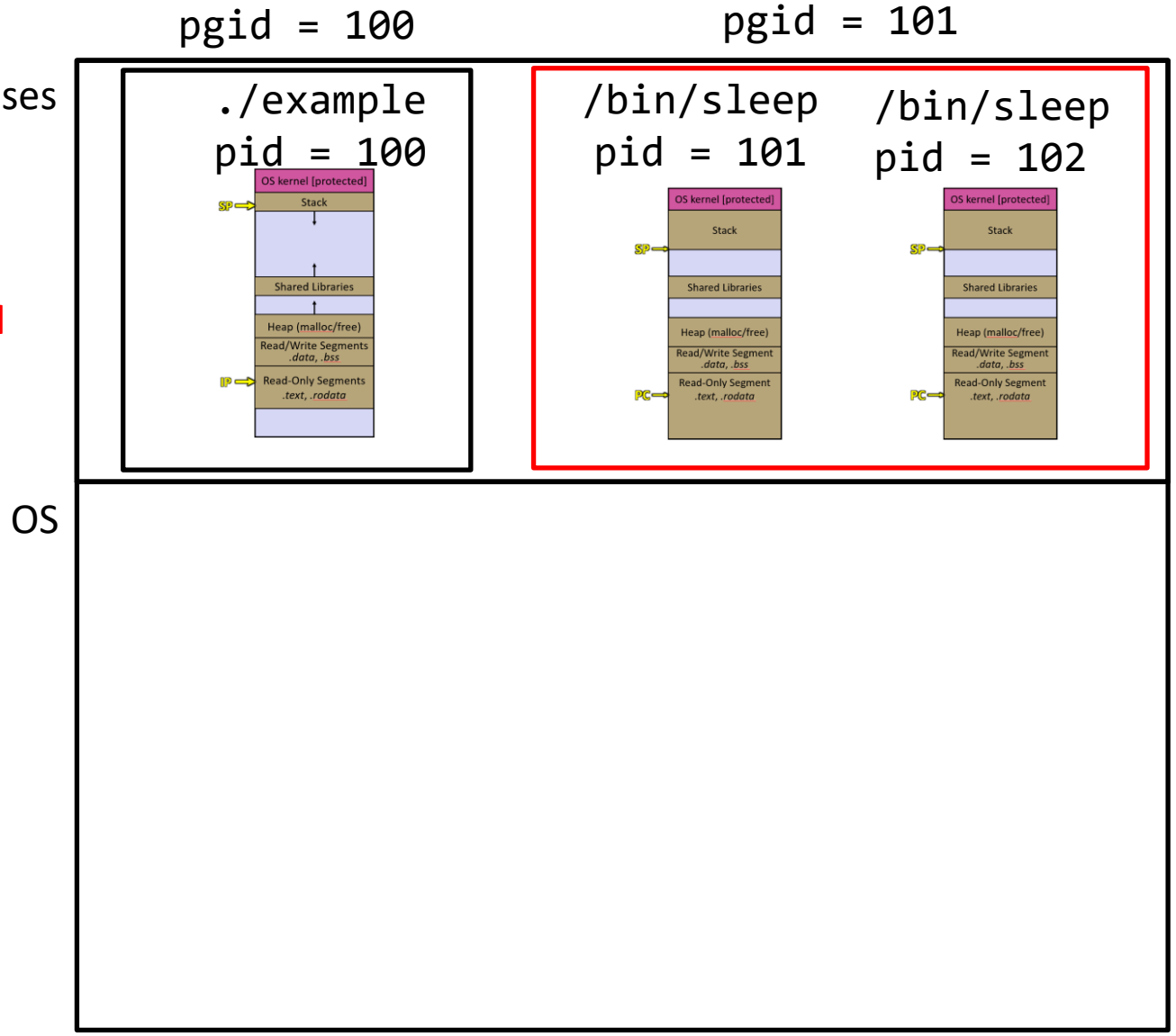
What if the parent forks a second child and adds it to the other child's group?



Example: pid vs –pgid

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

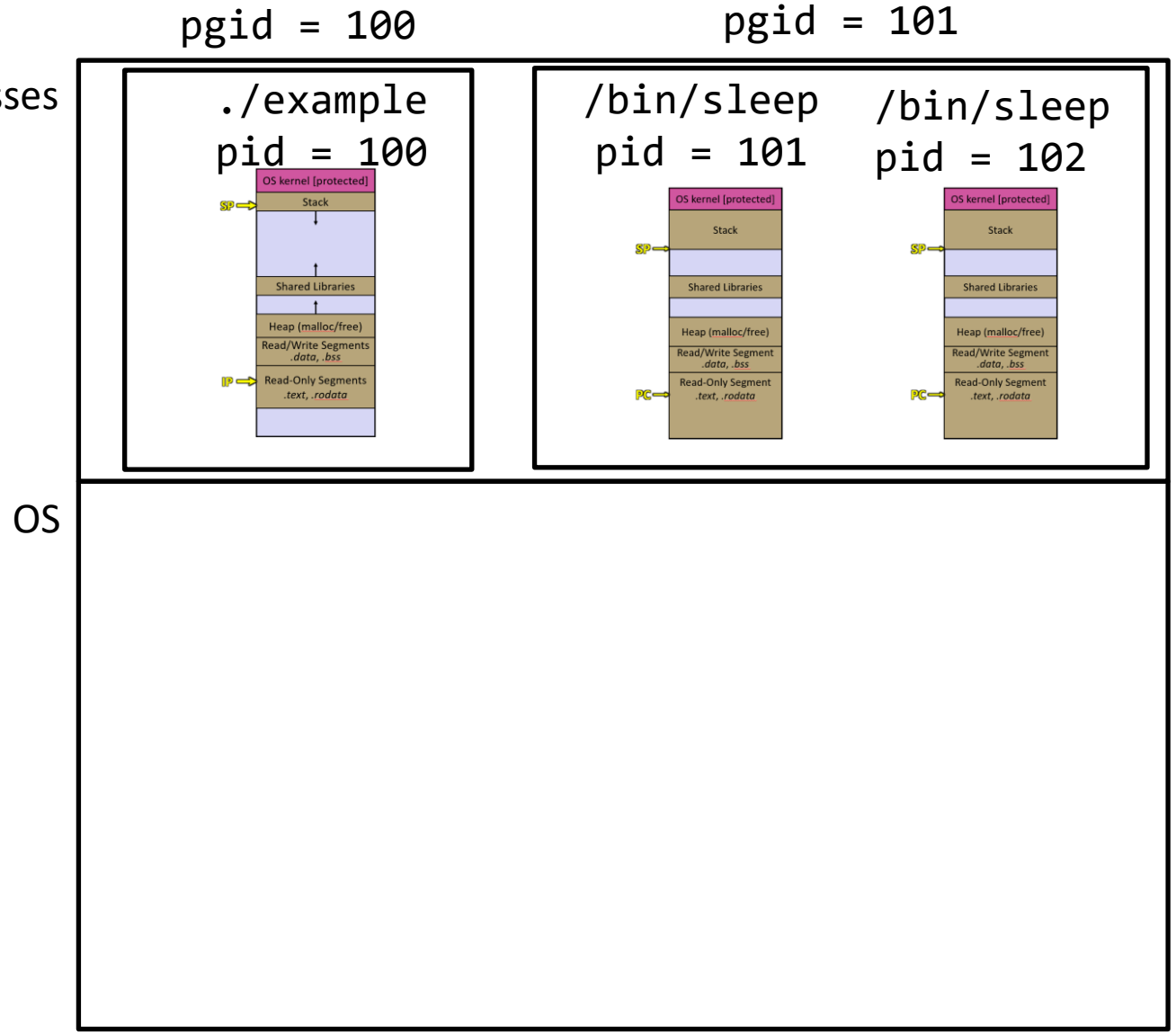
What if the parent forks a second child and adds it to the other child's group?



Example: pid vs –pgid

```
kill(101, SIGINT);
```

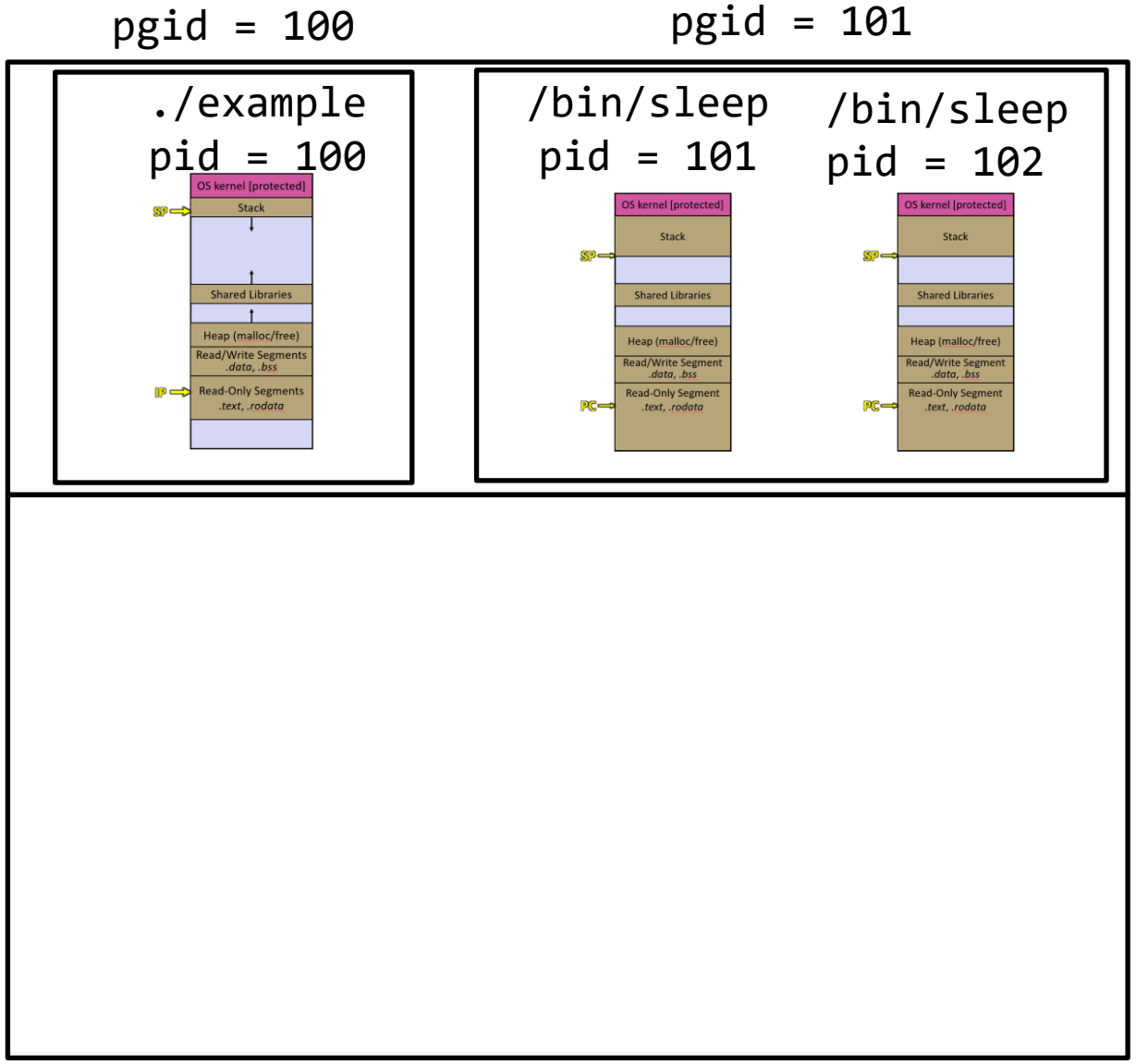
If the parent calls kill with pid 101, *only* the child with that pid receives the signal!



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

- pipe2
- Process Groups
  - `setpgid()`
- **Terminal Control**
  - **`tcsetpgrp()`**
- `SIGSTOP`
- Project 1: Sync vs Async 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:
  - ***Foreground processes***
    - ***These can read and write to their hearts content***
  - Background processes
    - These do not have access to the terminal and ***can not read from it***

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_FILENO)
  - 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 its own process group, ***it can no longer read from the terminal***
  - **It is no longer in the process group who controls the terminal**

- Jobs need to have their own groups, but they also need to navigate control of the terminal
  - Don’t want 100 processes trying to read the terminal at the same time. What if the data in the terminal isn’t for them?

Background Process

- If a **background process** tries to read from `stdin`, the OS sends the signal `SIGTTIN` to the background process
  - Default disposition: suspend/stop the process
  - **Try it:** `cat &`

- If a **background process** calls `tcsetpgrp()`, the OS will send the *entire* process group a `SIGTTOU` signal
  - Default disposition: suspend/stop the process
  - 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…

Background Process

- If a **background process** tries to write to `stdout`, that is okay!
  - Can be configured so that background processes get `SIGTTOU` instead
  - **Try it**: `cat file.txt &`

Let's try to fix our code from before!

- 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

Discuss

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

if (setpgid(pid, pid) == -1) {
  perror("setpgid\n");
  exit(EXIT_FAILURE);
}

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? If so, where would it be?

Discuss

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

if (setpgid(pid, pid) == -1) {
  perror("setpgid\n");
  exit(EXIT_FAILURE);
}

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? If so, where would it be?

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 scheduler
  - 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

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

```
if (setpgid(pid, pid) == -1) {
  perror("setpgid\n");
  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.

Discuss

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

```
13 int main() {
14   while (true) {
15     fprintf(stderr, "give command: ");
16     char c;
17     ssize_t bytes  = read(STDIN_FILENO, &c, 1);
18     if (bytes == -1) {
19       perror("read\n");
20       exit(EXIT_FAILURE);
21     } else if (bytes == 0) {
22       break;
23     }
24
25     if (c == 'c') {
26       pid_t pid = fork();
27
28       if (pid == 0) {
29         // child
30         // reads from the terminal and
31         // prints what it reads until EOF
32         char* args[] = {"cat", NULL};
33         execvp(args[0], args);
34         exit(EXIT_FAILURE);
35       }
36       // parent
37
```

```
36       // parent
37
38       // put the child in its own process group
39       if (setpgid(pid, pid) == -1) {
40         perror("setpgid\n");
41         exit(EXIT_FAILURE);
42       }
43
44       // give terminal to the child
45       if(tcsetpgrp(STDIN_FILENO, pid) == -1) {
46         perror("tcsetpgrp\n");
47         exit(EXIT_FAILURE);
48       }
49       printf("starting to wait\n");
50
51       int wstatus;
52       waitpid(pid, &wstatus, 0);
53     } else if (c == 's') {
54       printf("sleeping...\n");
55       sleep(5);
56       printf("awake\n");
57     } else if (c == 'p') {
58       printf("HOWDY\n");
59     }
60   }
```

Demo: tc_loop.c

- 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

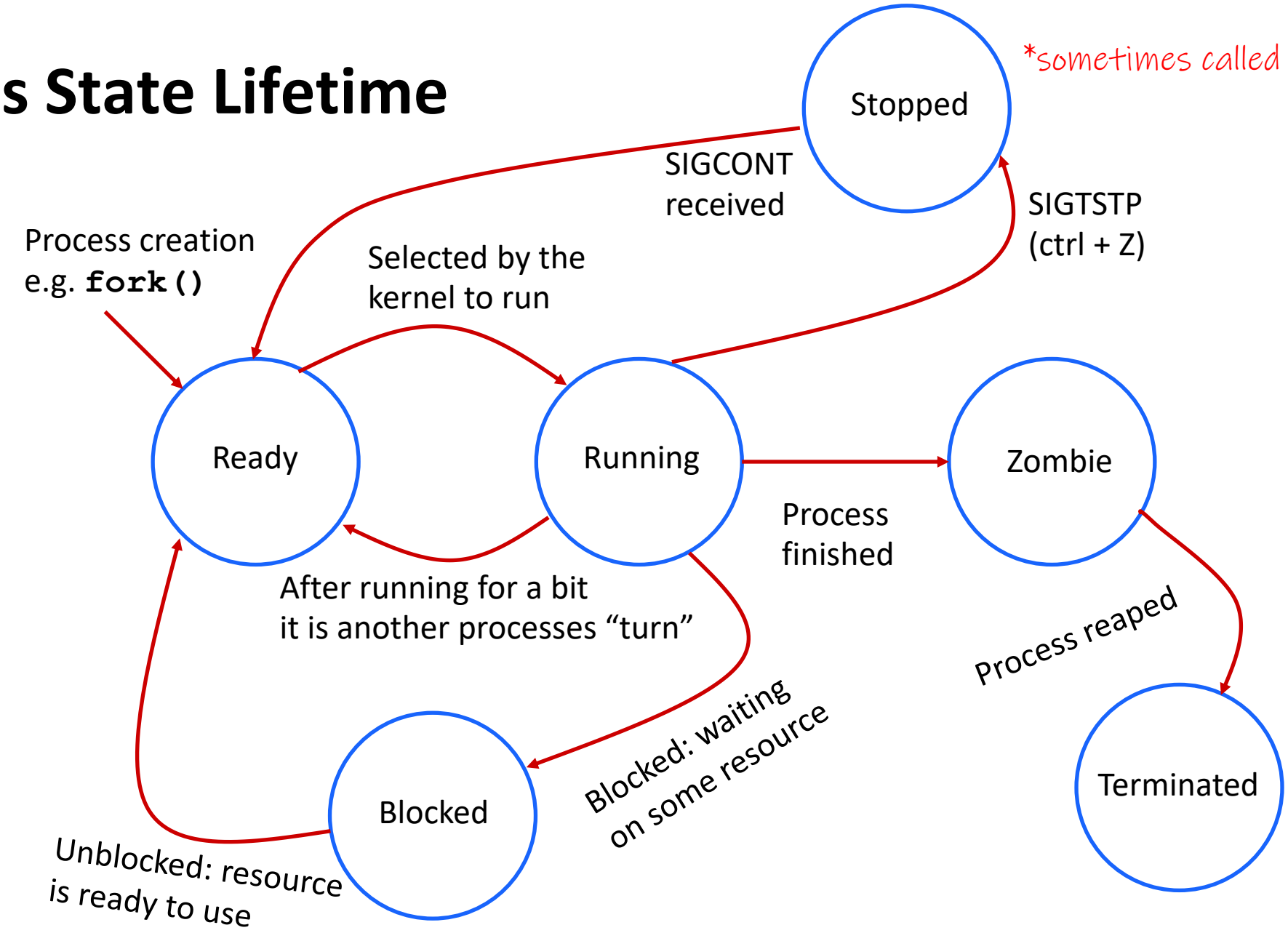
- pipe2
- Process Groups
  - `setpgid()`
- Terminal Control
  - `tcsetpgrp()`
- **SIGSTOP**
- Project 1: Sync vs Async wait
  - `SIGCHLD`

Stopped Jobs

- Processes can be in a state slightly different than being blocked called **stopped**
  - 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

- pipe2
- Process Groups
  - `setpgid()`
- Terminal Control
  - `tcsetpgrp()`
- `SIGSTOP`
- **Project 1: Sync vs Async wait**
  - **`SIGCHLD`**

Penn-shell

- Part of what you do in Project 1 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 list of the current jobs to handle job control

Discuss

- What does this pseudo-code of the “normal approach” 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!

Discuss

- What does this pseudo-code of the “polling approach” do?
- How does it 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… not good…

Aside: SIGCHLD

- This approach registers `SIGCHLD` as a handler, `SIGCHLD` is a signal that is sent when a child process stops or is terminated
  - Default disposition: ignored

Discuss

- What does this pseudo-code of the “async approach” do?
- How does it compare to the previous attempt?

```
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
  }
}
```

Analysis: Async

- This approach registers `SIGCHLD` as a handler, `SIGCHLD` is a signal that is sent when a child process stops or is terminated
  - Default disposition: ignored
- 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**

*Eventually ASYNC will be the standard assignment as it's the most useful to implement.

Reminder: `sigsuspend()`

- Another way to approach handling async using `sigsuspend()` or `sigwait()`
  - May be a little harder to reason about but some find it to be a bit more intuitive
  - Forces you not to call `waitpid` unless you need to.
    - Optimal shell will have one function with `waitpid` inside of it, called only when necessary
  - Don't have to do much in the signal handler if this is the case!

You finally have everything you need for shell! Yay!!!!!!!!!!!!!!!!!!!!