Terminal Control & Proj1 demo

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

pollev.com/tqm

- How was project 0?
  Are you excited for more C and project 1? ☺

Administrivia

- Peer Evaluation: out now, due Friday 9/22 @ 11:59 pm
  - Please do it, it shouldn't take long
  - Mostly completion, don't just say "this is fine" for everything

- Project 1 is out now
  - The milestone is due Wed 9/27 @ 11:59 pm
    late deadline: 11:59 pm on Sun, Oct 01
  - Project is due 11:59 pm on Wed, Oct 11
    late deadline 11:59 pm on Sun, Oct 15
  - Demo of project 1 in the last half of this class
  - After this class, should have everything needed to complete the project

Administrivia

- Recitation 3
  - After lecture today, going over pipes and redirection
  - Should help with finishing the project milestone
- Partner sign up
  - If you do not have a partner, we will begin random pairing the remaining students sometime tonight
- Check-in Quiz 3 Due in ~1 week

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Process Groups Revisited**
  - `setpgid()`
- Terminal Control
  - `tcsetpgrp()`
- Project 1: Synch vs Asynch wait
  - `SIGCHLD`
- Project 1 Demo & Q&A

Process Groups

- Processes are associated together into Process Groups.
  - A process always is in a process group
- Allows for convenient process & signal management:
  - If ctrl + C (SIGINT) is sent to a process via the keyboard, it is also sent to all processes within its group
- When we create a process with fork(), the child belongs to the same process group as the parent
- Shell has the notion of a **job**: "commands" started interactively. All processes in a job are in the same group
- Relevant for proj1: **`penn-shell`**

Process Group ID

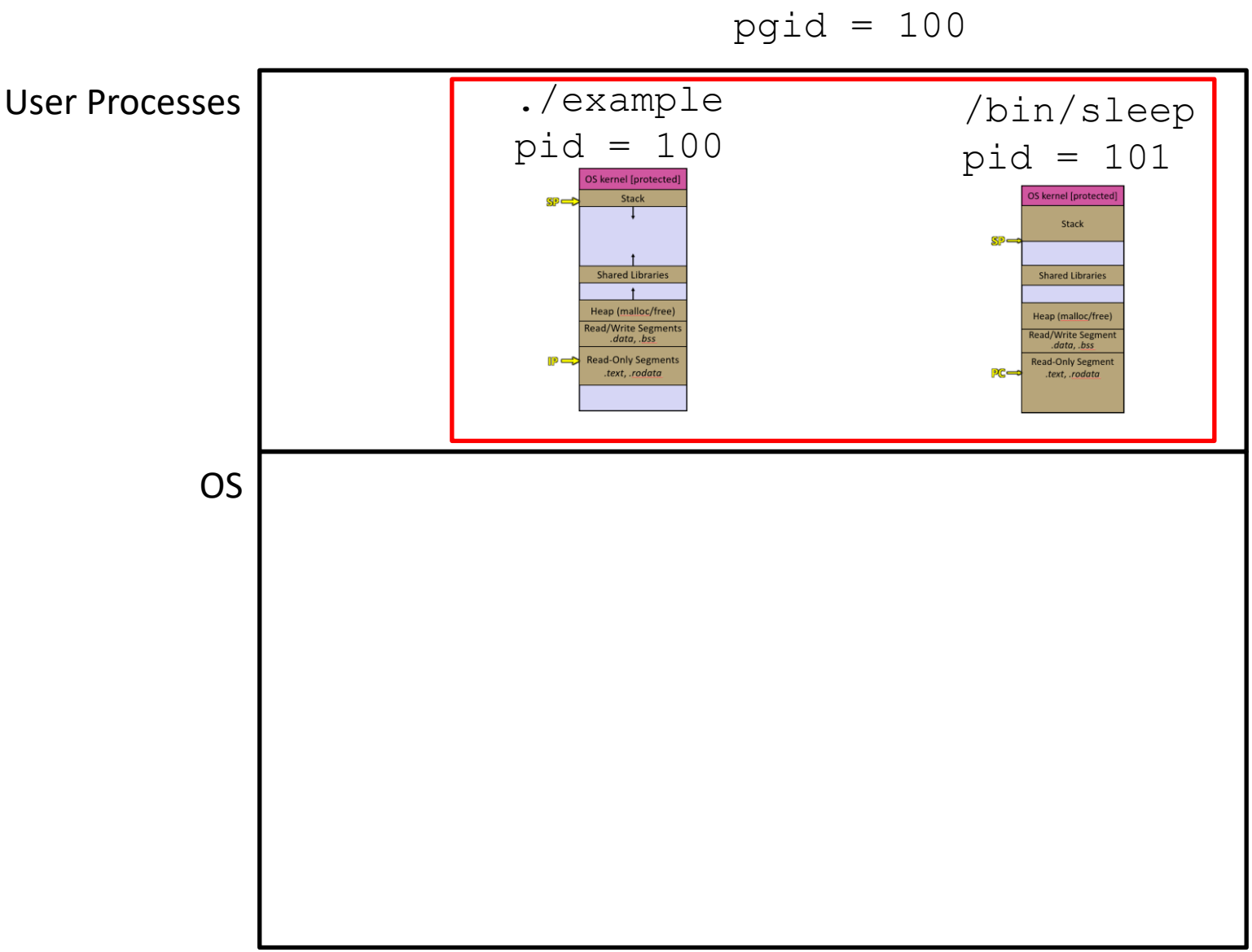
- The process group ID is equal to a process ID
  - The process ID of the first process to exist in the group
  - If a process group "leader" terminates, can its process ID be reused by another process? Even if the old group is still going?
  - Answer: no, that process ID will be reserved until the group is done

- ```
  int setpgid(pid_t pid, pid_t pgid);
  ```
- Sets page group id of the specified process to the new value
  - Only works if pgid specifies an existing process group
  - Or if pgid == pid, creates a new process group of that id
  ```

Process Group ID

- ```
  pid_t getpgid(pid_t pid);
  ```
- Gets the process group id of the specified process
- If `0` is passed in, get the group ID of the calling process
- `-1` returned on error
  ```

CTRL +C, same group

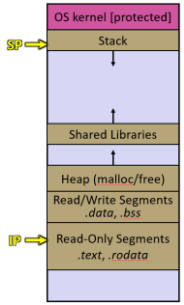


CTRL +C, same group

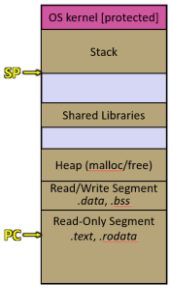
pgid = 100

User Processes

./example
pid = 100



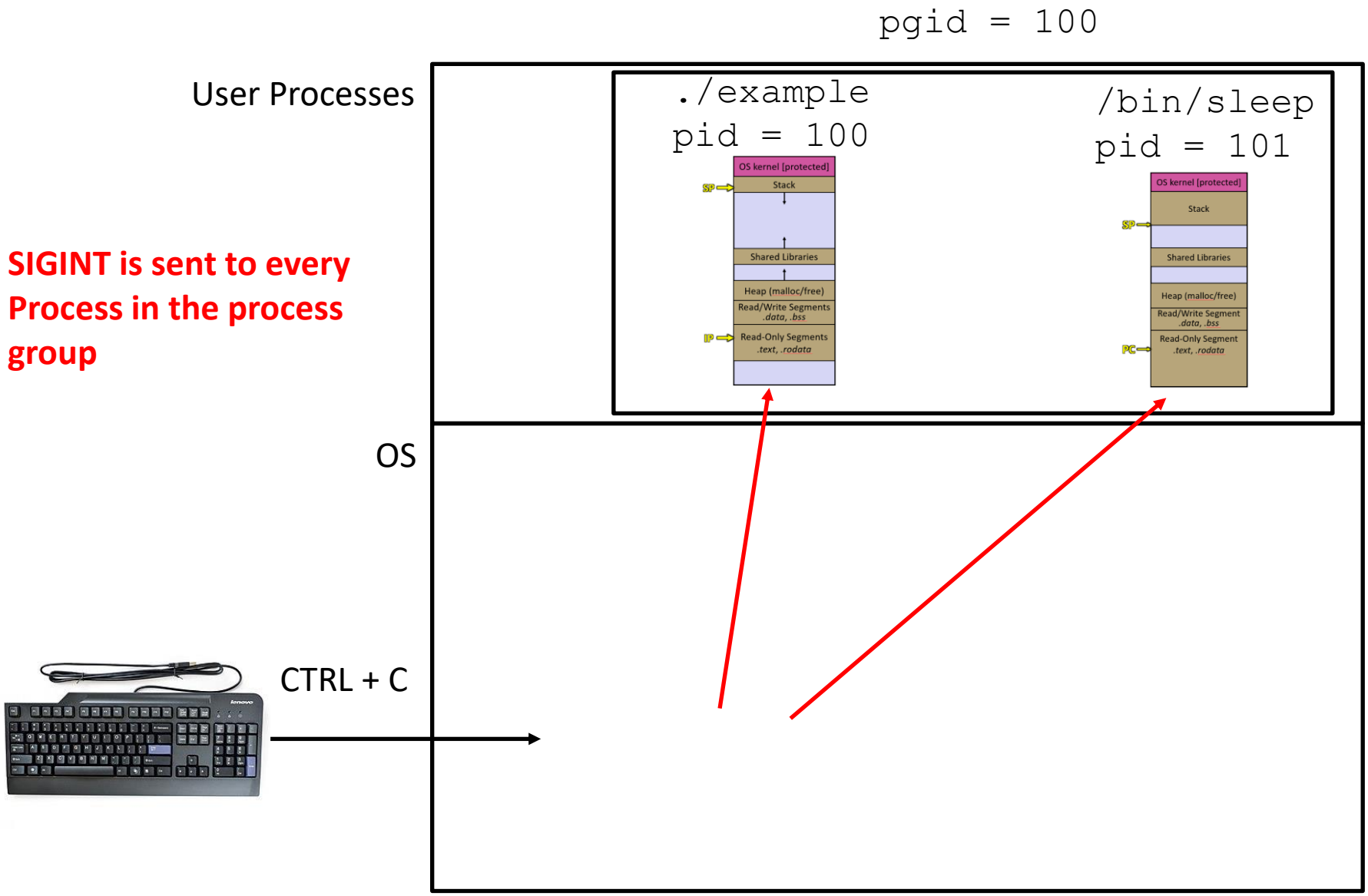
/bin/sleep
pid = 101



OS

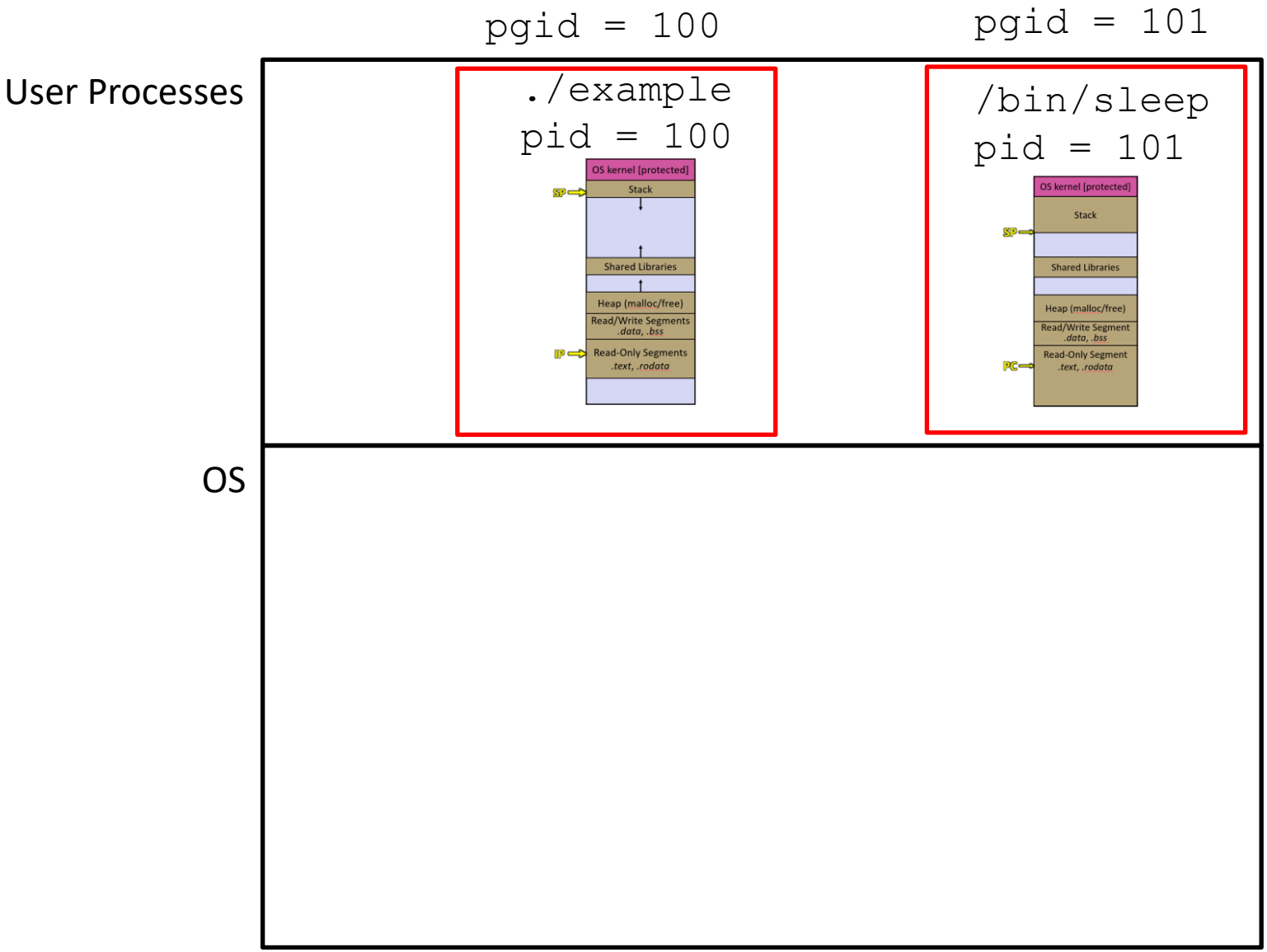
CTRL + C

CTRL +C, same group

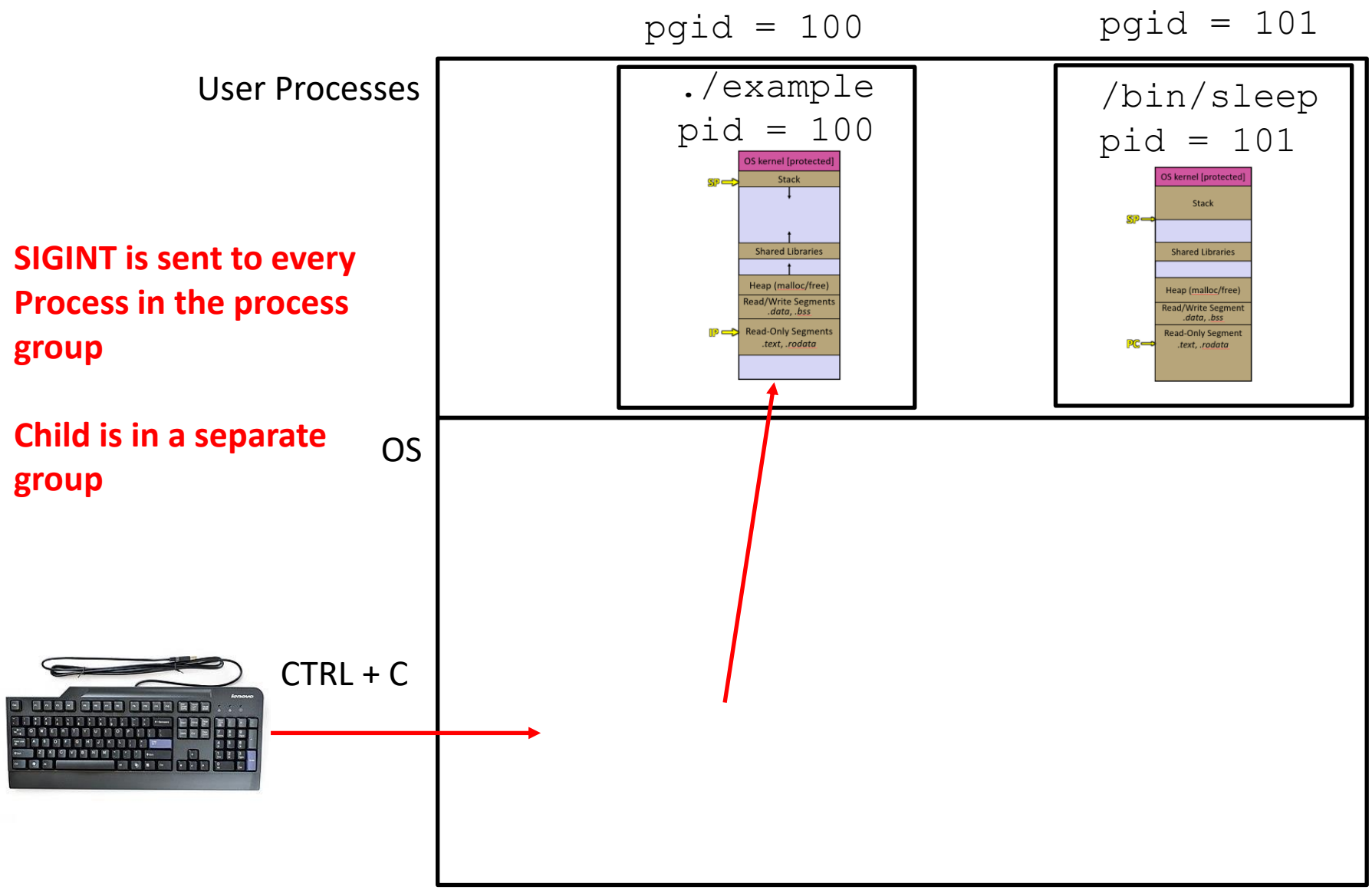


GAP SLIDE: MOVING ON TO DIFFERENT EXAMPLE

CTRL +C, different group



CTRL +C, different group



GAP SLIDE: MOVING ON TO DIFFERENT EXAMPLE

Process Groups: utility

- Can pass in **`-PGID`** (negative PGID) to **`kill()`** and **`waitpid()`**
- Doing so for **`kill()`** will send the signal to all processes in the group
- Doing so for **`waitpid()`** will wait for any process in the group
- You may find this useful for proj1: **`penn-shell`**

Diagram: signals between process groups

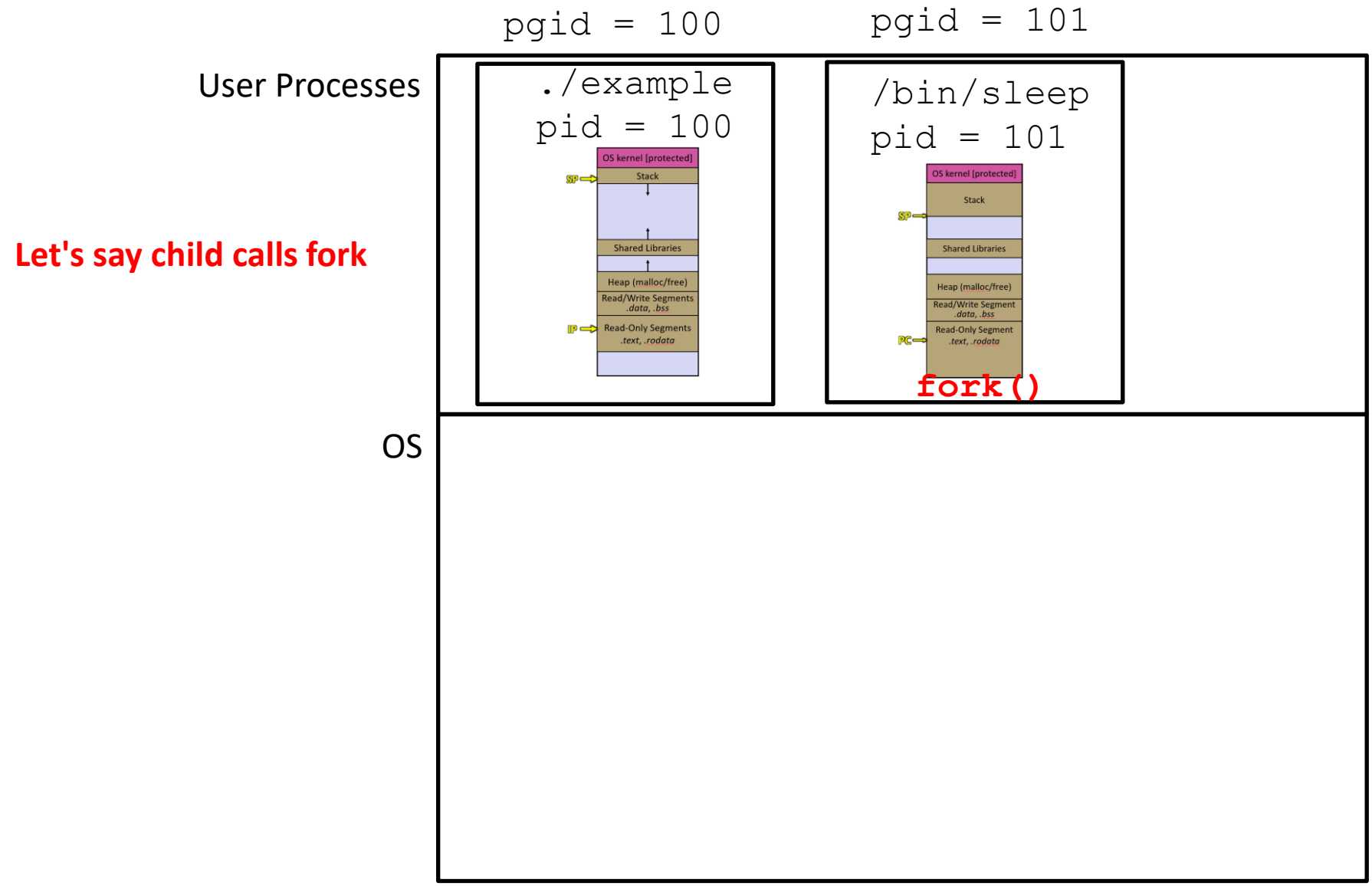
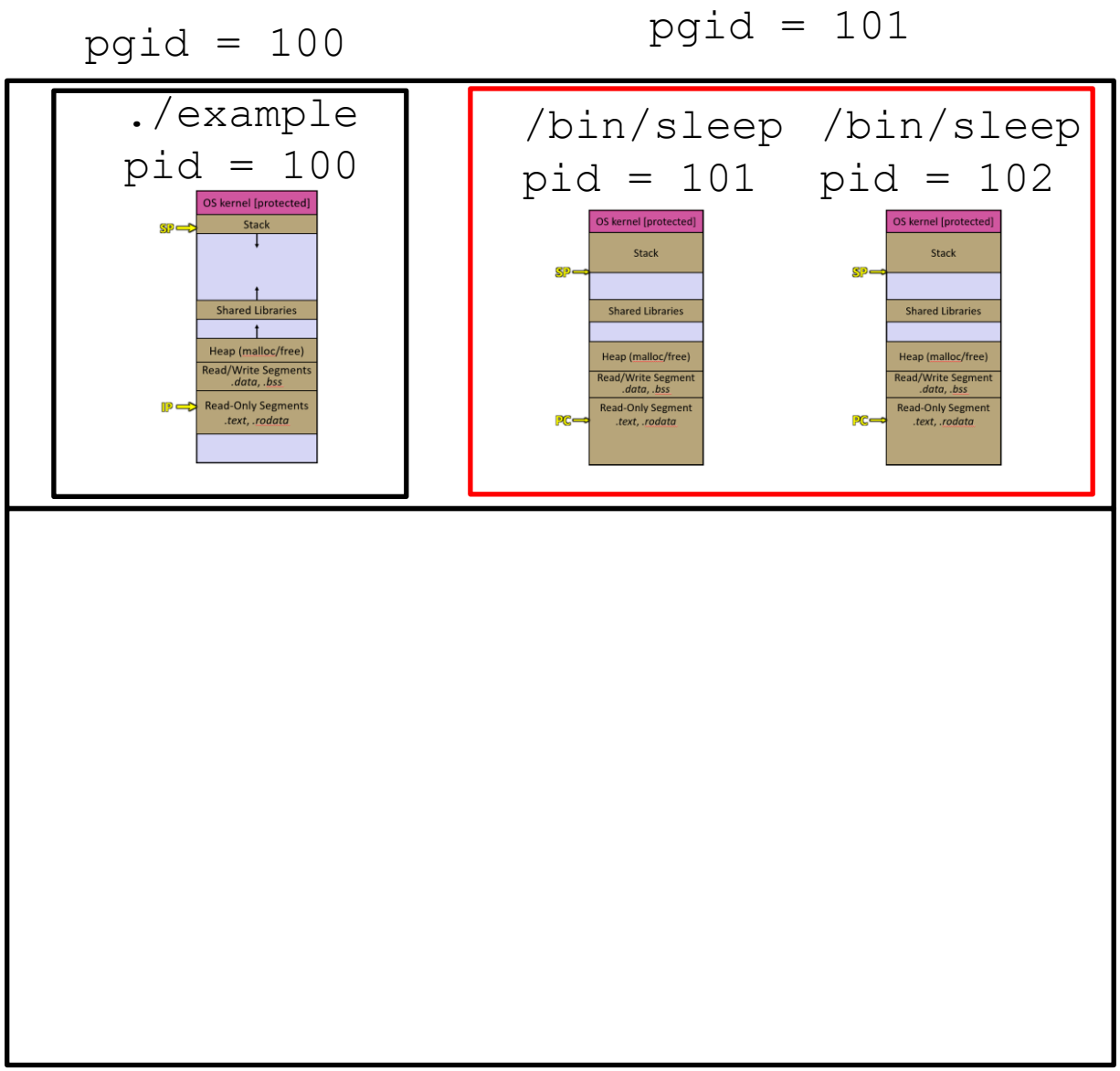


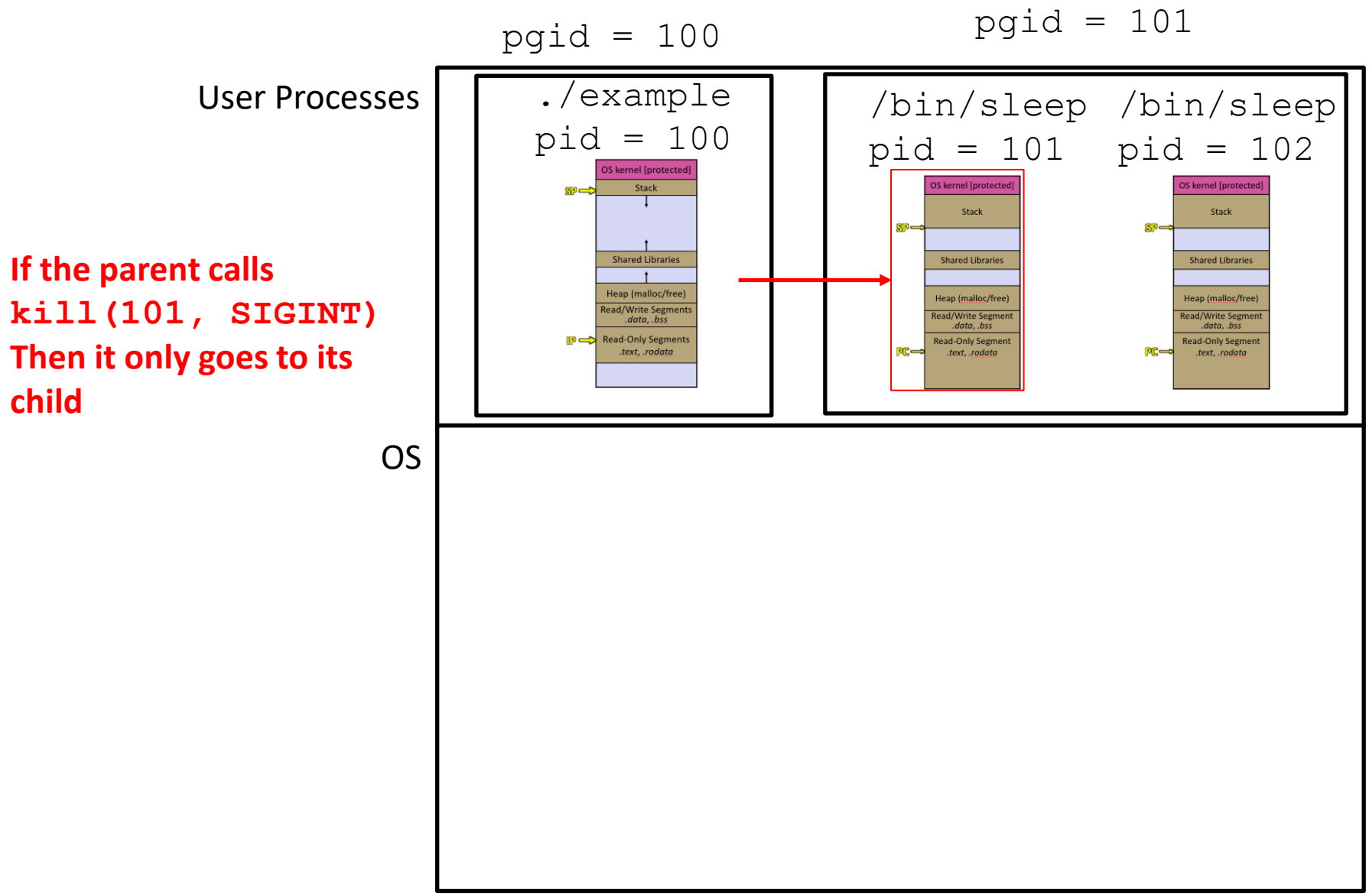
Diagram: signals between process groups



Let's say child calls fork

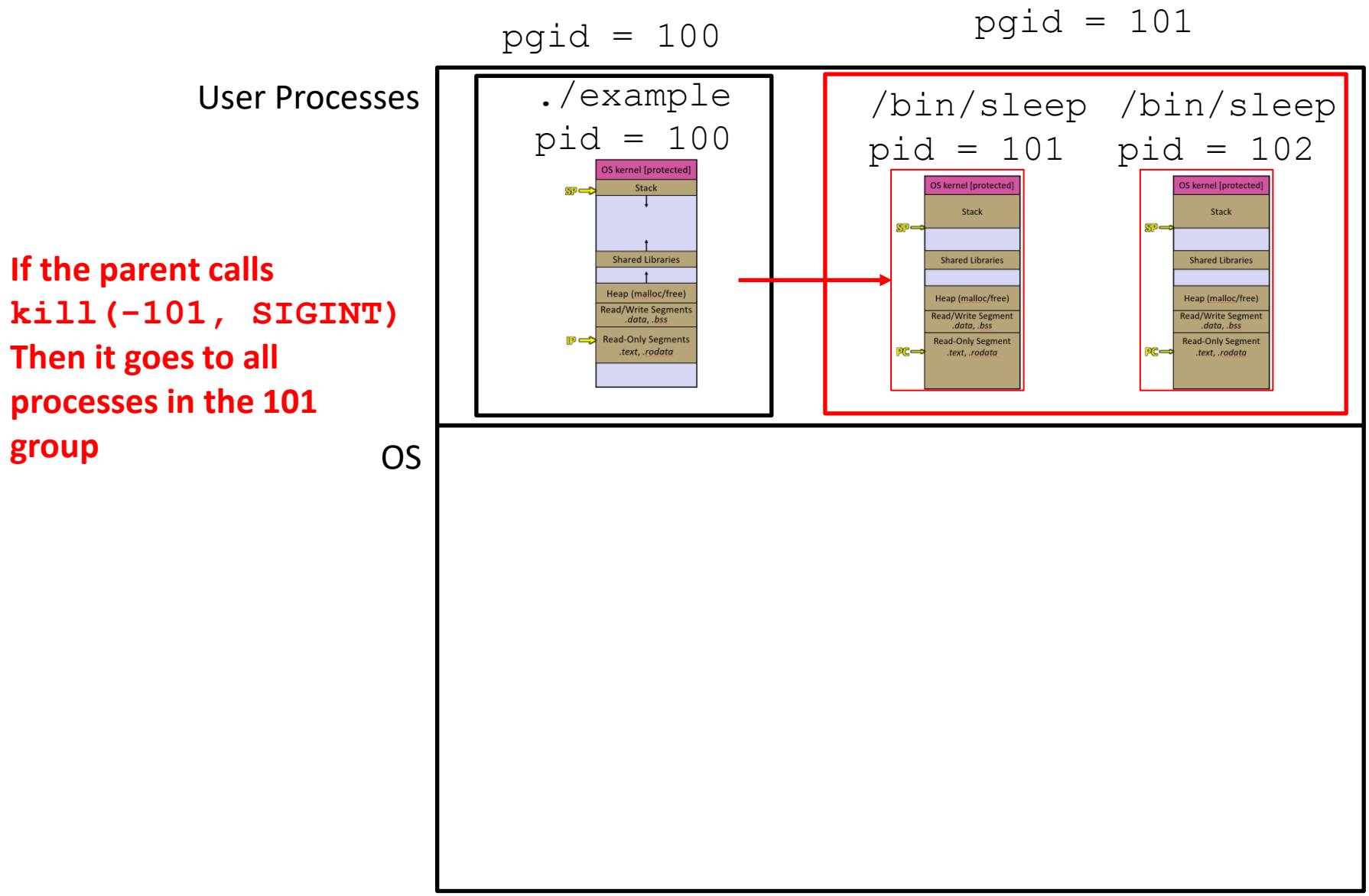
New child (grandchild of initial process) would be in the child's group

Diagram: signals between process groups



If the parent calls `kill(101, SIGINT)` Then it only goes to its child

Diagram: signals between process groups



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 Revisited
  - `setpgid()`
- **Terminal Control**
  - `tcsetpgrp()`
- Project 1: Synch vs Asynch wait
  - `SIGCHLD`
- Project 1 Demo & Q&A

What if the child tried to use the terminal?

- Demo!
  - Modify the `pgrp_signals.c` so that the child does “cat” (read from stdin, echo it to stdout until EOF)
  - it does not work?

Sessions

- A **Session** is a collection of process groups
  - A session can be attached to a controlling 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 session
- Within a session (that has a controlling terminal) there are
  - Background processes
  - Foreground processes

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 (e.g., CTRL + C)

- A foreground group can make another group the foreground with the function:
- ```
  int tcsetpgrp(int fd, pid_t pgrp);
  ```
  - **`fd`** is a file descriptor associated with the terminal (stdin)
  - Sets the process group specified by **`pgrp`** to be the foreground process group
  - `-1` returned on error, `0` when successful
  ```

Background Process

- If a background process tries to read from `stdin`, it gets sent the signal `SIGTTIN`

- If a background process tries to take control of the terminal with `tcsetgpgrp`, then the group gets sent `SIGTTOU`, which will stop the process group

- Writing to `stdout` from the background is ok, but can be configured so that background processes get `SIGTTOU`

Demo: tc.c

- See code demo: `tc.c`
  - Fixed our process group code so that it can run cat ☺
  - Parent can print to stdout even if has given away the terminal
  - How can we make the parent take back the terminal control?

Poll Everywhere

pollev.com/tqm

- 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

- 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 Revisited
  - **`setpgid()`**
- Terminal Control
  - `tcsetpgrp()`
- **Project 1: Synch vs Asynch wait**
  - **`SIGCHLD`**
- Project 1 Demo & Q&A

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

- Can use the `jobs` command to see the status of the jobs we have started

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

- Discuss: what does this do?
- How does this 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[]) {
  signal(SIGCHLD, handler);
  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
  - 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

Lecture Outline

- Process Groups Revisited
  - `setpgid()`
- Terminal Control
  - `tcsetpgrp()`
- Project 1: Synch vs Asynch wait
  - `SIGCHLD`
- **Project 1 Demo & Q&A**