Agenda

Inspecting process and system state

More Tips Here: https://www.cs.swarthmore.edu/~newhall/unixhelp/os_stats.php

File descriptors

Signals

Job Control

- Process Groups
- Sessions
- Terminal

Inspecting process and system information

Linux stores kernel state in two pseudo-file systems

/proc – runtime info about processes (`man 5 proc` for info)

/sys – info about the system state (devices, buses, drivers, etc.)

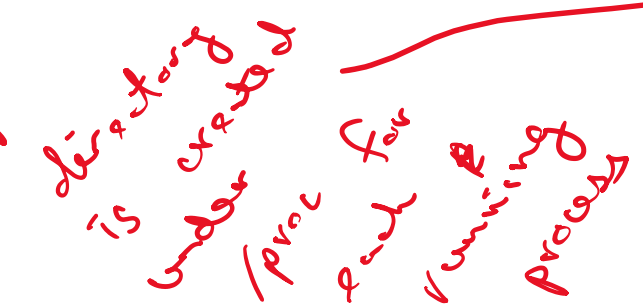
/proc creates a directory for each process

- /proc/<pid>/fd // File descriptors
- /proc/<pid>/status // pid, session id, process group id
- /proc/<pid>/XXX

/proc contains a directory for each process

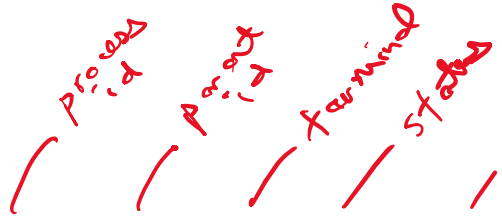
```
alinen@sutekh:~/cs355/os-devel/lectures/processes$ ps -A
  PID TTY          TIME CMD
    1 hvc0     00:00:00 init(Ubuntu)
    5 hvc0     00:00:12 init
    8 ?        00:00:00 SessionLeader
    9 ?        00:00:00 Relay(10)
   10 pts/0    00:00:00 bash
   41 ?        00:00:00 SessionLeader
   42 ?        00:00:00 Relay(43)
   43 pts/1    00:00:00 bash
  132 ?        00:00:00 SessionLeader
  133 ?        00:00:00 Relay(134)
  134 pts/2    00:00:00 bash
  345 ?        00:00:00 SessionLeader
  346 ?        00:00:00 Relay(347)
  347 pts/3    00:00:00 bash
  592 pts/0    00:00:00 ps
alinen@sutekh:~/cs355/os-devel/lectures/processes$
```

```
alinen@sutekh:/proc$ ls -l
total 0
dr-xr-xr-x  9 root   root              0 Feb 10 08:35 1
dr-xr-xr-x  9 alinen alinen            0 Feb 10 08:35 10
dr-xr-xr-x  9 root   root              0 Feb 10 09:04 132
dr-xr-xr-x  9 root   root              0 Feb 10 09:04 133
dr-xr-xr-x  9 alinen alinen            0 Feb 10 08:38 134
dr-xr-xr-x  9 root   root              0 Feb 10 09:11 345
dr-xr-xr-x  9 root   root              0 Feb 10 09:11 346
dr-xr-xr-x  9 alinen alinen            0 Feb 10 09:11 347
dr-xr-xr-x  9 root   root              0 Feb 10 09:04 41
dr-xr-xr-x  9 root   root              0 Feb 10 09:04 42
dr-xr-xr-x  9 alinen alinen            0 Feb 10 08:35 43
dr-xr-xr-x  9 root   root              0 Feb 10 08:36 5
dr-xr-xr-x  9 alinen alinen            0 Feb 10 11:06 598
dr-xr-xr-x  9 root   root              0 Feb 10 09:04 8
dr-xr-xr-x  9 root   root              0 Feb 10 09:04 9
dr-xr-xr-x  2 root   root              0 Feb 10 11:03 acpi
-r--r--r--  1 root   root              0 Feb 10 11:03 buddyinfo
dr-xr-xr-x  4 root   root              0 Feb 10 11:03 bus
```



Inspecting process state

Use **ps**



- ps -eo pid,ppid,tty,stat,cmd (Control what info is displayed)
- ps –efwH (Show process hierarchy)

Use **top –u <username>**

Shows a configurable list of processes that updates dynamically

Use **strace**

Displays all system calls made by an application

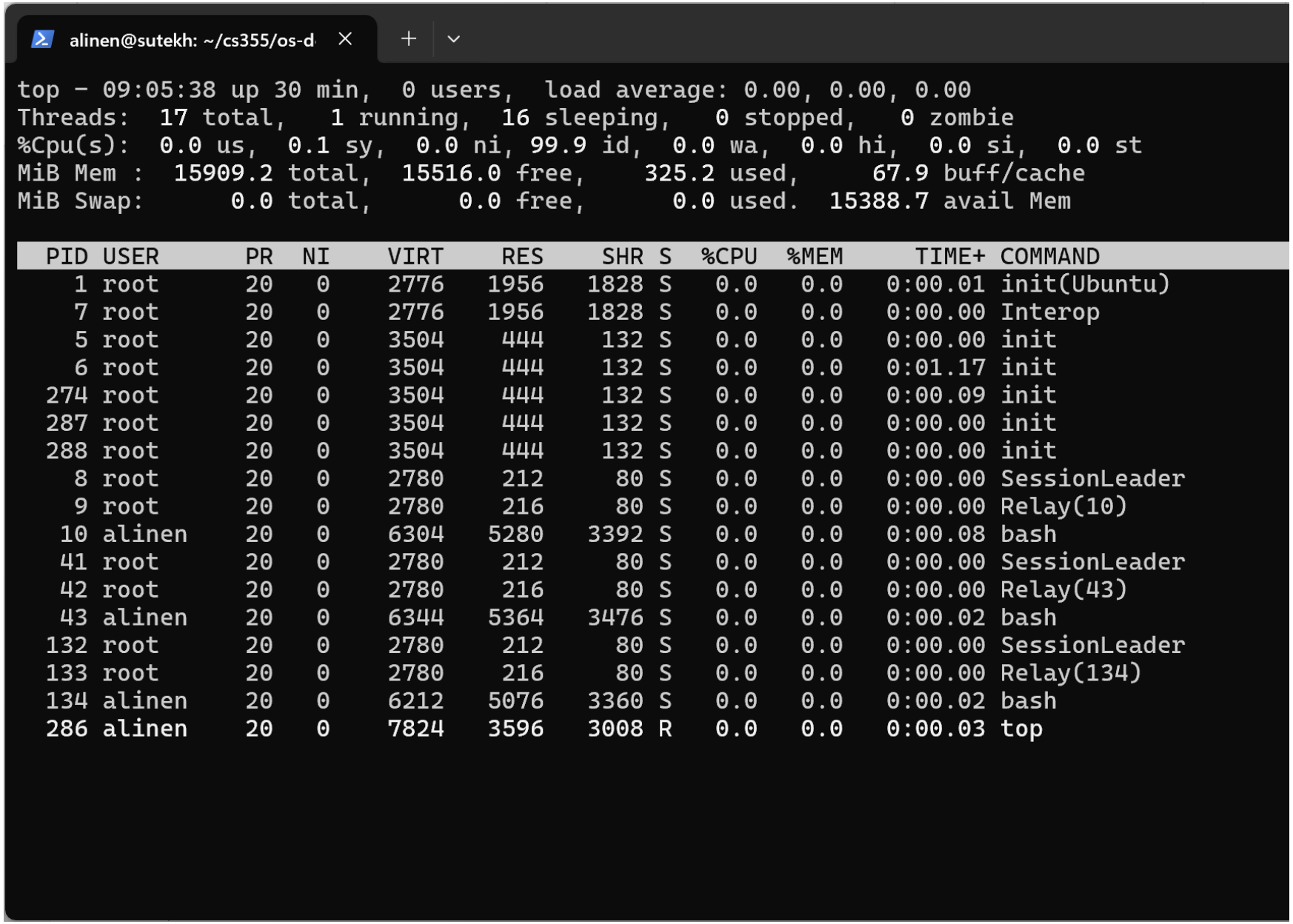
Displays any signals sent or received

top

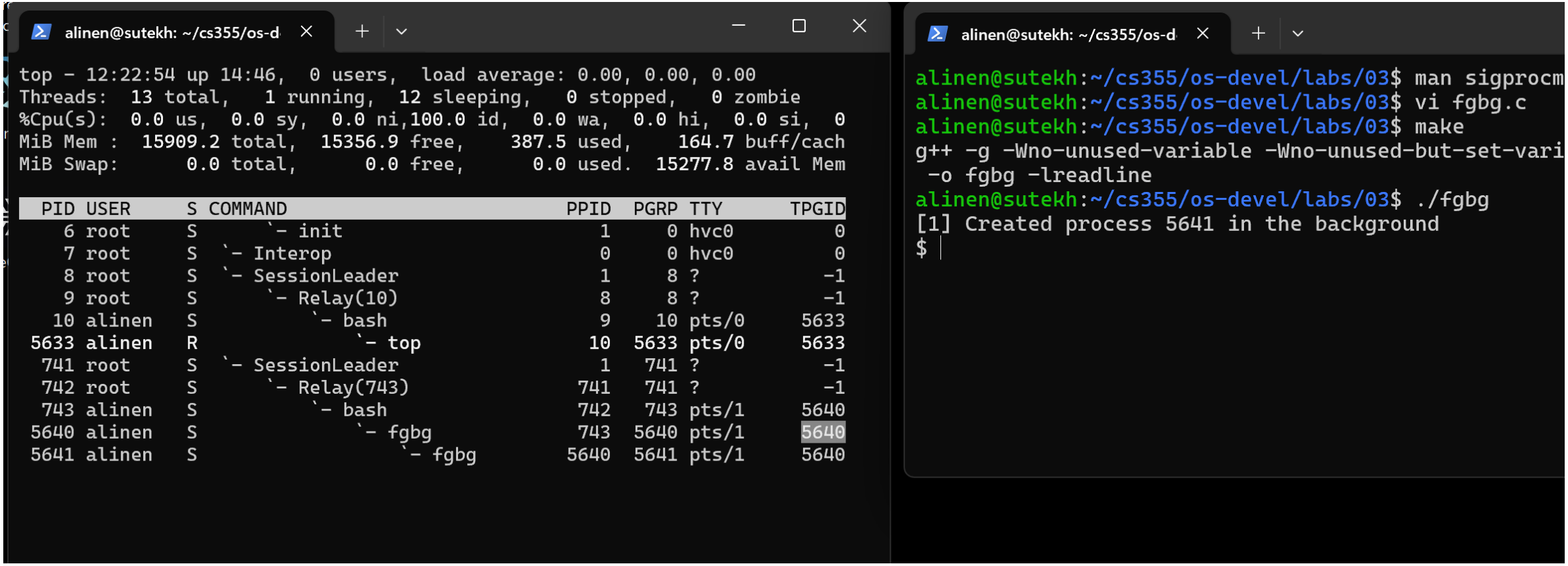
q to quit

f/F to filter/sort

v/V forest view



Demo: top H


alinen@sutekh: ~/cs355/os-d
top - 12:22:54 up 14:46, 0 users, load average: 0.00, 0.00, 0.00
Threads: 13 total, 1 running, 12 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0
MiB Mem : 15909.2 total, 15356.9 free, 387.5 used, 164.7 buff/cach
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 15277.8 avail Mem
PID USER S COMMAND PPID PGRP TTY TPGID
6 root S `- init 1 0 hvc0 0
7 root S `- Interop 0 0 hvc0 0
8 root S `- SessionLeader 1 8 ? -1
9 root S `- Relay(10) 8 8 ? -1
10 alinen S `- bash 9 10 pts/0 5633
5633 alinen R `- top 10 5633 pts/0 5633
741 root S `- SessionLeader 1 741 ? -1
742 root S `- Relay(743) 741 741 ? -1
743 alinen S `- bash 742 743 pts/1 5640
5640 alinen S `- fgbg 743 5640 pts/1 5640
5641 alinen S `- fgbg 5640 5641 pts/1 5640
alinen@sutekh: ~/cs355/os-d
alinen@sutekh:~/cs355/os-devel/labs/03$ man sigprocm
alinen@sutekh:~/cs355/os-devel/labs/03$ vi fgbg.c
alinen@sutekh:~/cs355/os-devel/labs/03$ make
g++ -g -Wno-unused-variable -Wno-unused-but-set-vari
-o fgbg -lreadline
alinen@sutekh:~/cs355/os-devel/labs/03$ ./fgbg
[1] Created process 5641 in the background
$

https://linux-audit.com/cheat-sheets/strace/

Demo: strace

Utility for tracking all system calls made by processes

To see all system calls made by a process and its children:

```
strace -f <exe>
```

To only see signals:

```
strace -e 'trace=!all' -f <exe>
```

To monitor dup2 of files : `strace -e dup2 <exe>`

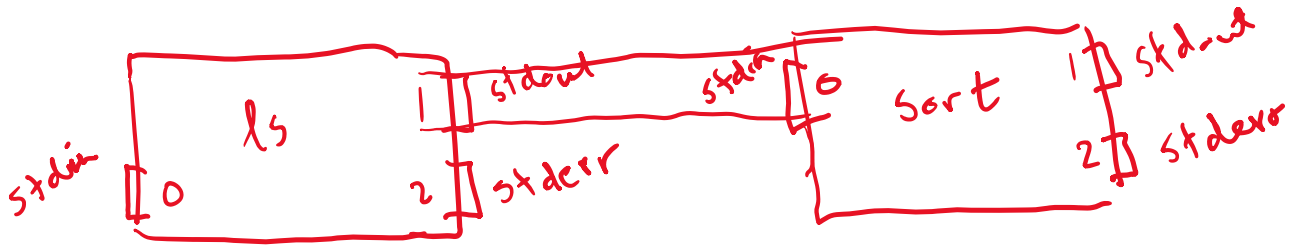
To see all file related activity: `strace -e trace=file <exe>`

Pipes and redirection

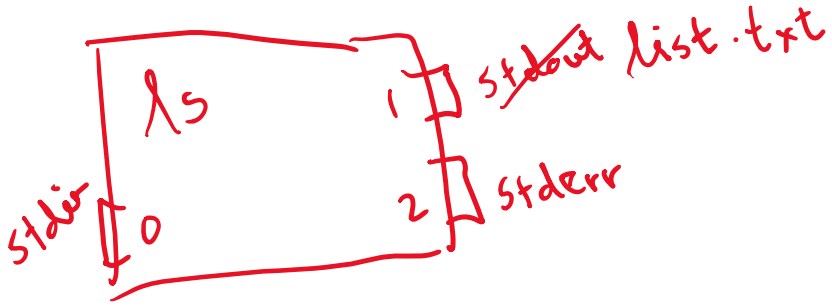
All processes have default file descriptors (fd) for stdin (0), stdout (1), stderr(2)

We can change the default input/output behavior by changing the file descriptors

ls | sort



ls > sort.txt



Demo: Creating a pipe

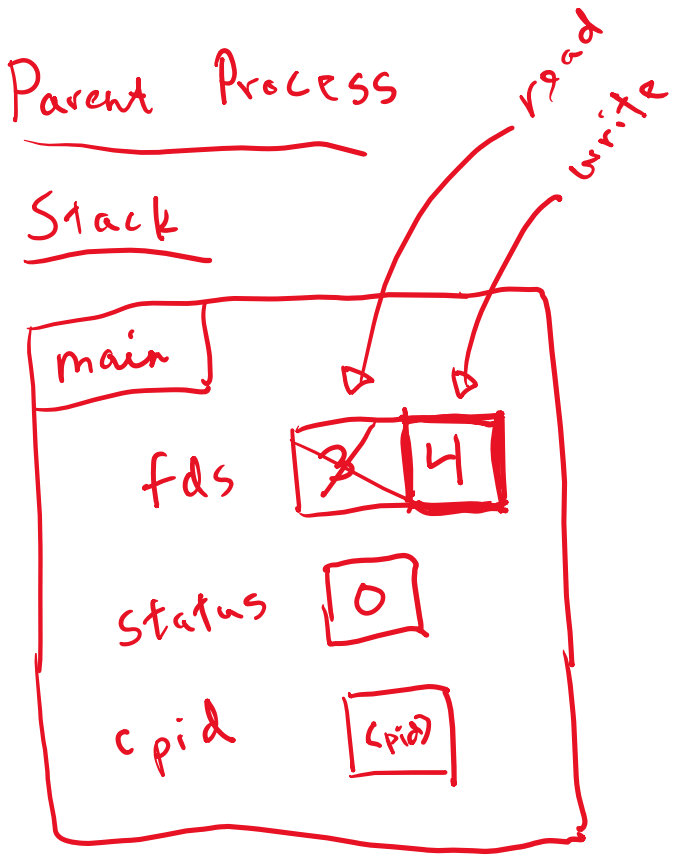
pipe is a system call
↖ initializes a pipe w/ a write end & a read end

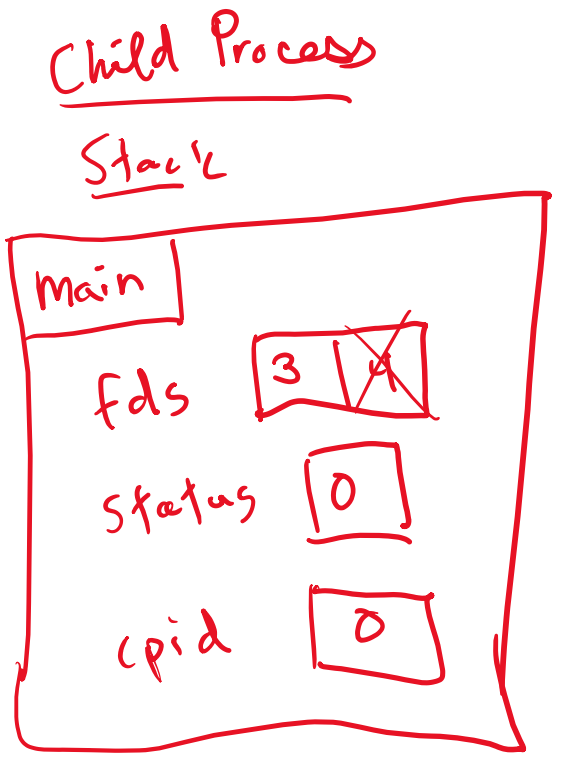
```
int pipefd[2]; // 0 -> read; 1 -> write

int status = pipe(pipefd);
checkerror(status, "pipe");

pid_t cpid = fork();
checkerror(cpid, "fork");

if (cpid == 0) {    //child
 close(pipefd[1]); // close unused write end
 while (read(pipefd[0], &buf, 1) > 0) {
  write(STDOUT_FILENO, &buf, 1);
 }
 write(STDOUT_FILENO, "\n", 1);
 close(pipefd[0]);
}
```

```
else {
  close(pipefd[0]); // close unused read end
  const char* message = "mario takes the pipe to world 5";
  write(pipefd[1], message, strlen(message));
  close(pipefd[1]);
  wait(NULL); // wait for child
 }
 exit(EXIT_SUCCESS);
}
```

Visualizing pipes

Parent writes to fds[1]
Child reads from fds[0]





Demo: pipes

```
alinen@sutekh:~/cs355/os-devel/lectures/processes$ ls -l /proc/1581/fd
total 0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 1 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 2 -> /dev/pts/0
lr-x------ 1 alinen alinen 64 Feb  4 20:15 3 -> 'pipe:[5344]'
l-wx------ 1 alinen alinen 64 Feb  4 20:15 4 -> 'pipe:[5344]'
alinen@sutekh:~/cs355/os-devel/lectures/processes$ ls -l /proc/1582/fd
total 0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 1 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 2 -> /dev/pts/0
lr-x------ 1 alinen alinen 64 Feb  4 20:15 3 -> 'pipe:[5344]'
l-wx------ 1 alinen alinen 64 Feb  4 20:15 4 -> 'pipe:[5344]'
alinen@sutekh:~/cs355/os-devel/lectures/processes$ ls -l /proc/1581/fd
total 0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 1 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 2 -> /dev/pts/0
l-wx------ 1 alinen alinen 64 Feb  4 20:15 4 -> 'pipe:[5344]'
alinen@sutekh:~/cs355/os-devel/lectures/processes$ ls -l /proc/1582/fd
total 0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 1 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb  4 20:15 2 -> /dev/pts/0
lr-x------ 1 alinen alinen 64 Feb  4 20:15 3 -> 'pipe:[5344]'
alinen@sutekh:~/cs355/os-devel/lectures/processes$
```

Demo: Redirecting output with dup2

```
void waitkey()
{
  printf("Press any key to continue"); getc(stdin);
  printf("\n");
}

int main(int argc, char *argv[])
{
  printf("Run `ls -l /proc/%d/fd` to see the file descriptors "
     "for this process.\n", getpid());
  waitkey();

  int fd = open("list.txt", O_RDONLY);  waitkey();

  dup2(fd, STDERR_FILENO);  waitkey();

  close(fd);  waitkey();
  return 0;
}
```

```
$ ./dup
Run `ls -l /proc/639/fd` to see the file
descriptors for this process.
Press any key to continue

Press any key to continue

Press any key to continue

Press any key to continue
```

Visualizing dup2

```
int fd = open("list.txt", O_RDONLY);
dup2(fd, STDERR_FILENO);
close(fd);
```

Demo: Dup2

```
alinen@sutekh:~$ ls -l /proc/639/fd
total 0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 1 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 2 -> /dev/pts/0
alinen@sutekh:~$ ls -l /proc/639/fd
total 0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 1 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 2 -> /dev/pts/0
lr-x------ 1 alinen alinen 64 Feb 10 11:14 3 -> /home/alinen/cs355/os-devel/lectures/processes/list.txt
alinen@sutekh:~$ ls -l /proc/639/fd
total 0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 1 -> /dev/pts/0
lr-x------ 1 alinen alinen 64 Feb 10 11:14 2 -> /home/alinen/cs355/os-devel/lectures/processes/list.txt
lr-x------ 1 alinen alinen 64 Feb 10 11:14 3 -> /home/alinen/cs355/os-devel/lectures/processes/list.txt
alinen@sutekh:~$ ls -l /proc/639/fd
total 0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 0 -> /dev/pts/0
lrwx------ 1 alinen alinen 64 Feb 10 11:14 1 -> /dev/pts/0
lr-x------ 1 alinen alinen 64 Feb 10 11:14 2 -> /home/alinen/cs355/os-devel/lectures/processes/list.txt
alinen@sutekh:~$
```

NOTE: The special files in fd are **symbolic links**. A symbolic link refers to a file without copying it.

Signals

Signals are a type of software interrupt. A small message to tell a process that some event has happened

How signals work:

1. OS sends a signal to a process
   - On behalf of another process that called the `kill` syscall
   - As the result of some event (NULL pointer dereference)
2. A process receives a signal
   Asynchronous: signalee doesn't know when it will get one
   Signals are pending before a process recieves it
3. A signal interrupts the receiving process, which then runs **signal handler** code
   - default handlers for each signal type in OS
   - programmer can also add signal handler code

Examples: Signal Types

segfault!

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Invalid memory reference (e.g. NULL ptr)
18	SIGCONT	Continue	Resume process (e.g. bg/fg shell commands)
19	SIGSTP	Stop	Stop process (e.g. ctrl-z from keyboard)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signals can be sent explicitly (by a user) or implicitly (as a side-effect of program doing something, e.g. dereferencing a NULL ptr causes `SIGSEGV`)

`**man 7 signal**` for more information

Example: Cntrl-z, bg

```
$  ./inf_loop
^Z
[1]+  Stopped  ./inf_loop
$ ps w
  PID TTY      STAT   TIME COMMAND
28850 pts/2    Ss     0:00 bash
29011 pts/2    T      0:03 ./inf_loop
29021 pts/2    R+     0:00 ps w
$ bg
./inf_loop
$ ps w
  PID TTY      STAT   TIME COMMAND
28850 pts/2    Ss     0:00 bash
29011 pts/2    R      0:03 ./inf_loop
29105 pts/2    R+     0:00 ps w
```

Ctrl-z sends a SIGTSTP signal to every process running in the foreground, process is STOPPED

bg: sends SIGCONT signal and process runs in the background (shell continues)

`ps w` STAT field values:

First letter:

- S: sleeping
- T: stopped
- R: running

Second letter:

- s: session leader
- +: foreground process

Slide by Tia Newhall

Example: Cntrl-c, fg

```
$  ./inf_loop
^Z
[1]+  Stopped  ./inf_loop
$ ps w
  PID TTY      STAT   TIME COMMAND
28850 pts/2    Ss     0:00 bash
29011 pts/2    T      0:03 ./inf_loop
29021 pts/2    R+     0:00 ps w
$ fg
./inf_loop
^C
$ ps w
  PID TTY      STAT   TIME COMMAND
28850 pts/2    Ss     0:00 bash
29105 pts/2    R+     0:00 ps w
```

fg: sends SIGCONT and moves process into the foreground (shell waits)

Ctrl-c sends a SIGINT signal to every process running in the foreground (process will exit)

`ps w` STAT field values:

First letter:

- S: sleeping
- T: stopped
- R: running

Second letter:

- s: session leader
- +: foreground process

Slide by Tia Newhall

Sending signals

Signals can be sent using the **kill** command

Examples:

Shell command:

```
$ kill -9 1234    # send SIGKILL signal to process 1234
$ kill -L         # List all signals
```

System call:

```
kill(1234, SIGKILL); // send SIGKILL to process 1234
```

Example: Sending signals

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main() {

  int pid = 0;
  printf("Enter a process id: ");
  scanf(" %d", &pid);

  int signal_id = 0;
  printf("Enter a signal id: ");
  scanf(" %d", &signal_id);

  kill(pid, signal_id);
}
```

Receiving signals

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Possible ways to react:
  - ***Ignore*** the signal (do nothing)
    not all signals can be ignored (e.g. SIGKILL, SIGSTP)
  - ***Terminate*** the process on receipt of signal, possibly with **core** file
  - ***Catch*** the signal by executing a user-level function called signal handler

Signal Handlers

To catch a signal, a program needs to register a signal handler

Modifies the default action associated with the receipt of a particular signal

A process can register a function (e.g. the signal handler) to be invoked when a signal is received

`#include <sys/signal.h>` lists signal handling functions

To define a fn ptr type: typedef void (*handler_t)(int);

Writing your own signal handler

```
signal(int signum, handler_t *handler);
```

function ptr's type is determined by its return value & parameters

- `handler` is a function, indicated with a **function pointer**
  - When program receives signal, it jumps to start executing the `handler` function.
  - When the `handler` done executing, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Example: Simple signal handler

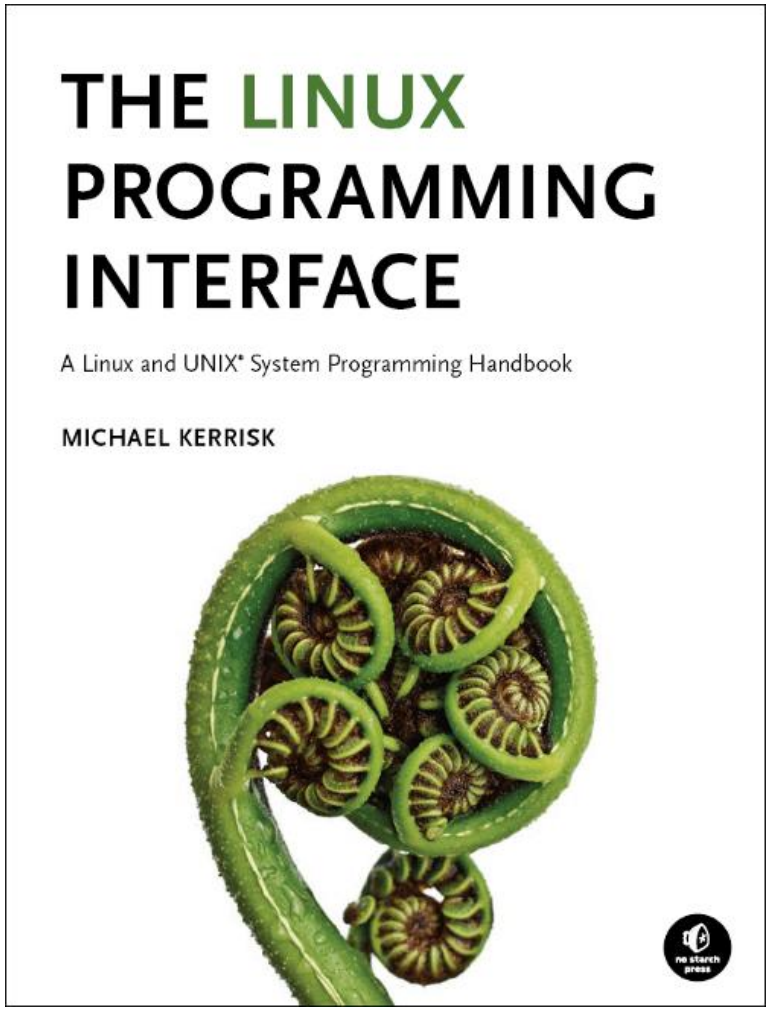
```
// signal handler function: called when process receives SIGINT
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/wait.h>

void int_handler(int sig) {
  printf("Proc %d received signal %d\n",getpid(), sig);
  exit(0);
}

void main() {
  signal(SIGINT, int_handler);
  while(1);
}
```

Job Control

Chapter 34 from TLPI

The Linux Programming Interface, by Michael Kerrisk, No Starch Press, 2010

- Lots of examples
- https://www.man7.org/tlpi/



Job Control Overview

Modern shells support **job control** features that allow the user to start, pause and stop processes running at the command line

First introduced in 1980 on BSD (Kerrisk 2010)

A **job** is a command, typed at the command line. It can contain pipes, file redirections, and multiple commands, e.g.

ls –l | wc –w

./compute_stats > output.txt

./generate_data | ./compute_stats > output.txt

A job can run in the **foreground** (receives terminal input) or **background** (Demo)

Terminal

The **terminal** corresponds to the window where we type in commands and see output

When you run a process in the foreground, the file descriptors – stdout (1), stdin (0), stderr (2) – are configured to write to the terminal

Writing to the terminal involves writing to a virtual file (**/dev/tty**) that represents the interface to our keyboard/mouse/monitor

Example: Job Control at the terminal

$ **echo $$**	*Display the PID of the shell*
400	
$ **find / 2> /dev/null | wc -l &**	*Creates 2 processes in background group*
[1] 659	
$ **sort < longlist | uniq -c**	*Creates 2 processes in foreground group*

tty

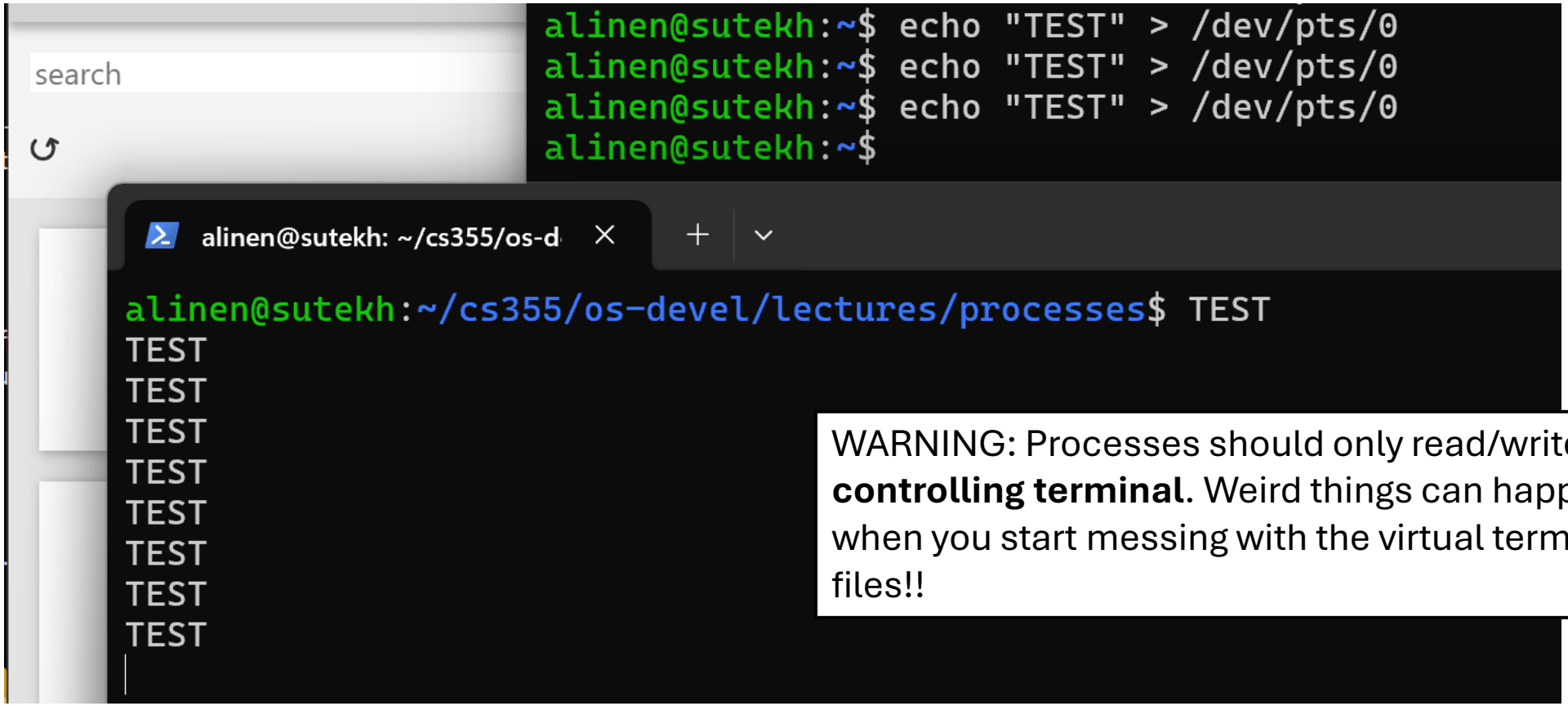
tty = Teletypewriter

Each terminal gets a tty for reading and writing

```
$ ps w
 PID TTY      STAT   TIME COMMAND
  10 pts/0    Ss     0:00 -bash
  43 pts/1    Ss+    0:00 -bash
 134 pts/2    Ss+    0:00 -bash
 347 pts/3    Ss+    0:00 -bash
 738 pts/0    R+     0:00 ps w
```

Demo: tty

Writing to the console corresponds to writing to the console's TTY



Who owns the terminal?

Foreground process groups (i.e., Foreground Jobs)

- Can read from STDIN
- Receive signals from the keyboard (e.g., CTRL + C)

To move a process from the background to the foreground, you "give" the background process the terminal

```
int tcsetpgrp(int fd, pid_t pgrp) // fd = STDIN_FILENO
```

Watch out!

Only the foreground process can give the terminal to another process group

- 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

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

In your shell, the parent process should call **tcsetgpgrp** to move processes from the foreground and background => the shell should ignore the signals SIGTTIN and SIGTTOU(More later)

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

Sessions and process groups

Each job belongs to the same **process group**

- By default, the process group ID (**PGID**) corresponds to the **process group leader.** The process group leader is typically the parent when using fork and exec.

A **session** is a collection of process groups

- The session id (**SID**) corresponds to the process ID of the session leader.
- Associated with a login or a terminal
- The session leader is typically the shell program that is run when the terminal starts

All processes in a session have a single **controlling terminal**

Within a session, a single process group is the **foreground process group**. Any number of process groups can be a **background process group**

Example: Session and process groups

```
$ echo $$
400
$ find / 2> /dev/null | wc -l &
[1] 659
$ sort < longlist | uniq -c
```

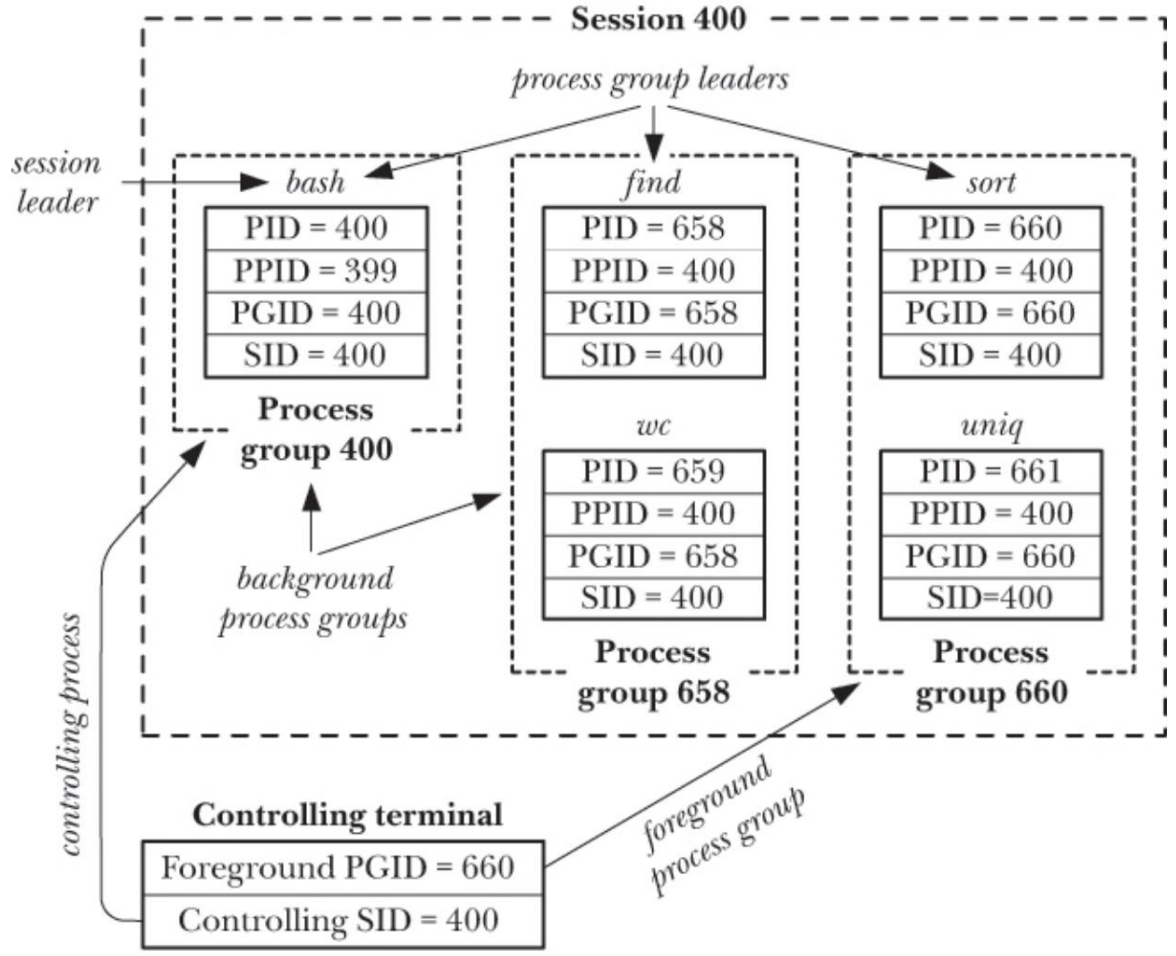


Figure 34-1: Relationships between process groups, sessions, and the controlling terminal

Example: Session and process groups

```
$ echo $$
400
$ find / 2> /dev/null | wc -l &
[1] 659
$ sort < longlist | uniq -c
```

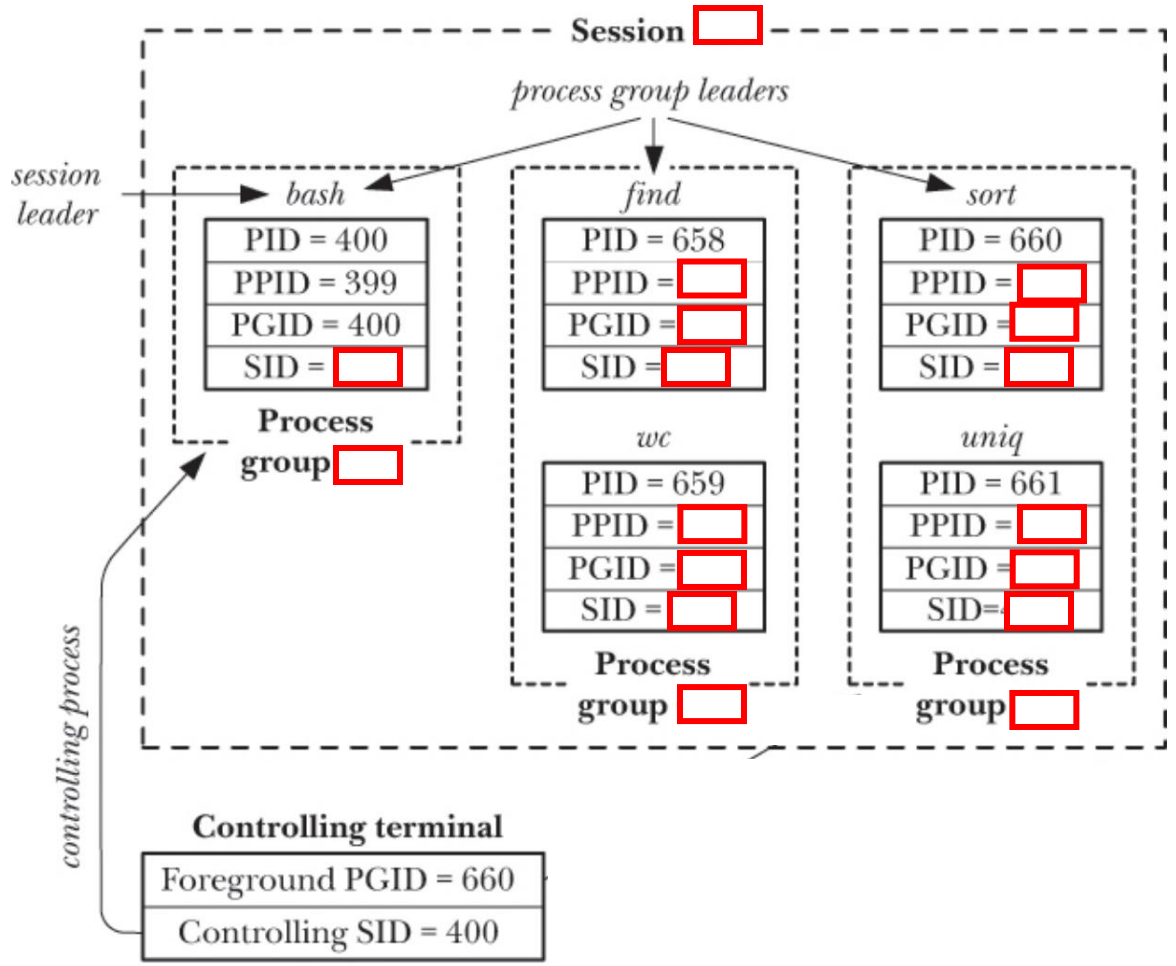


Figure 34-1: Relationships between process groups, sessions, and the controlling terminal

Example: Job control in the shell

```
$ grep -r SIGHUP /usr/src/linux >x &
[1] 18932
$ sleep 60 &
[2] 18934
$ jobs
[1]- Running      grep -r SIGHUP /usr/src/linux >x &
[2]+ Running       sleep 60 &
$ fg %1
grep -r SIGHUP /usr/src/linux >x

[1]- Done         grep -r SIGHUP /usr/src/linux >x
[2]+ Done          sleep 60
$
```

Job 1: process running grep has PID 18932

Job 2: process running sleep has PID 18934

Job 1: move to foreground

Completed jobs are reported when finished

Process groups

When a signal is sent, it sent to the entire process group by default

When we create a process with fork(), the child belongs to
the same process group as the parent

The process group ID is equal to a process ID

- The process ID of the first process to exist in the group

Question: 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

Setting the process group

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

- Only works if pgid specifies an existing process group
- Or if pgid == pid, creates a new process group of that id

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

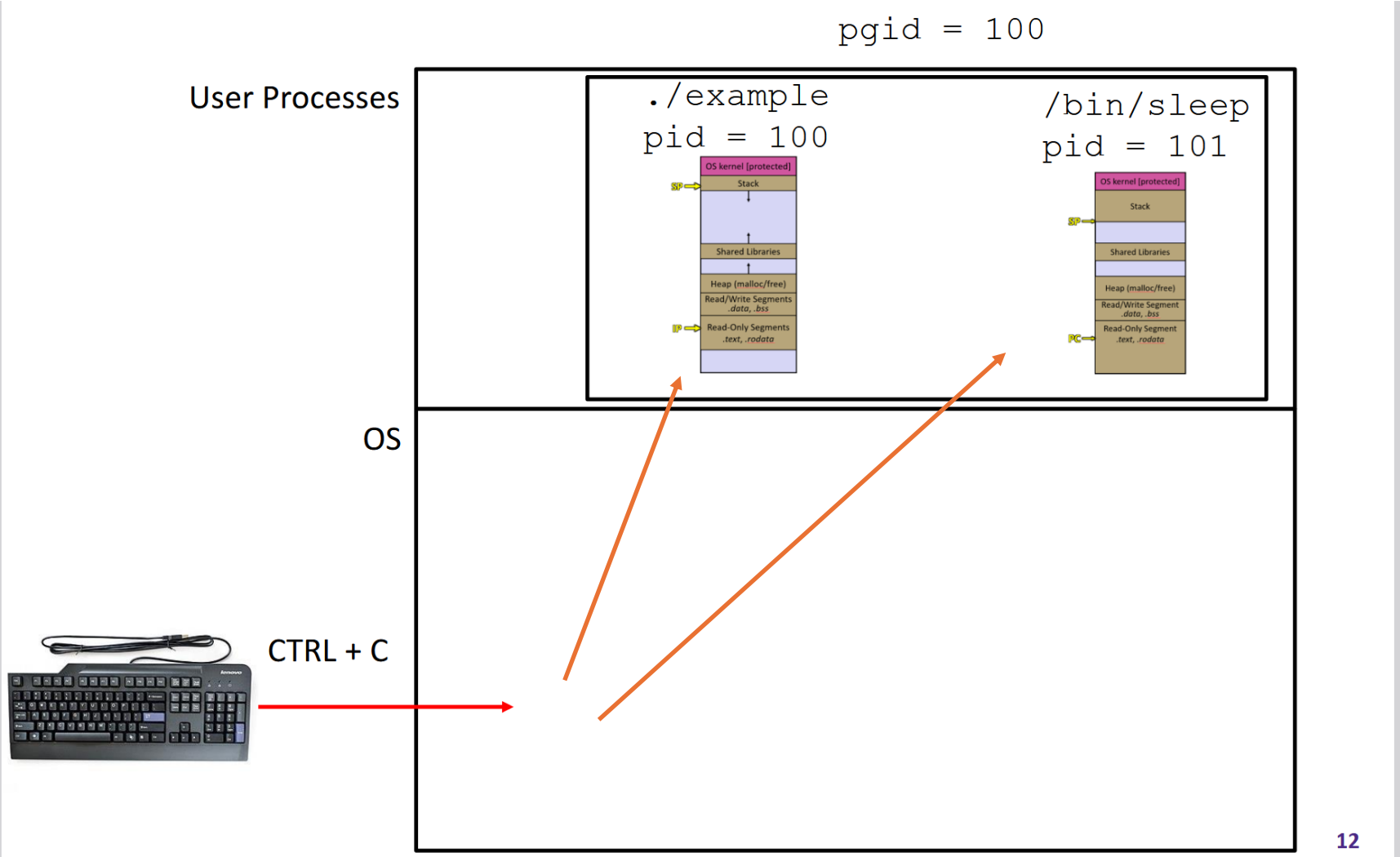
- Gets the process group id of the specified process
- If 0 is passed in, get the group ID of the calling process

Process group example

// Putting the child in its own process group

```
 int pid = fork();
 if (pid == 0)   {
   setpgid(0, 0); // put child in their own group; can also do setpgid(getpid(), getpid())
   while(1) { pause(); }
 }
 else   {
   setpgid(pid, pid); // put child in their own group

   std::string command;
   std::cin >> command;
   while (command != "quit") std::cin >> command;

   kill(pid, SIGTERM); // make sure to kill child
}
```

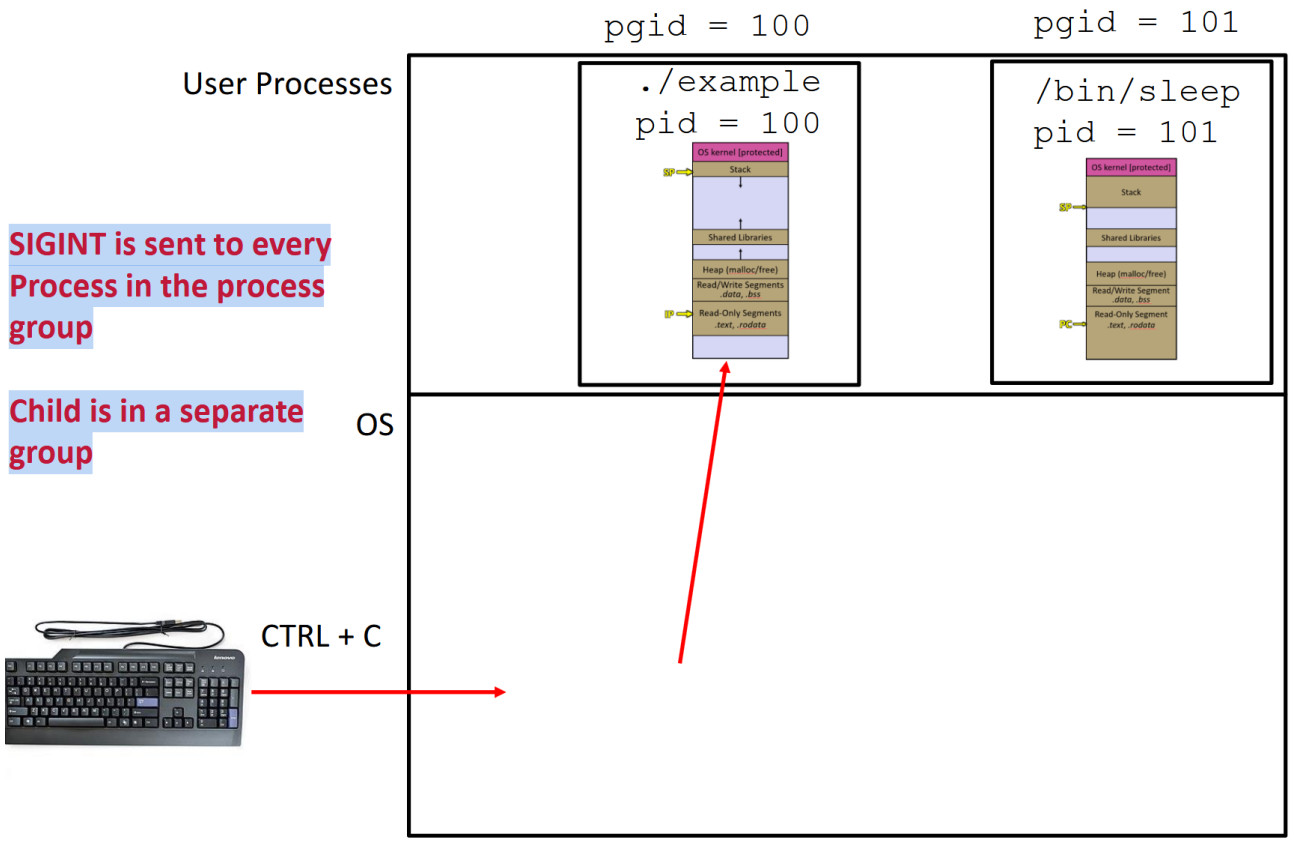
Example: Ctrl-C Same Group



SIGINT is sent to every Process in the process group

Example: Ctrl-C different groups

CTRL +C, different group



SIGINT is sent to every Process in the process group

Child is in a separate group

Waiting on processes

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