CSCI 2330



TODO

- Assignment 5 assigned due November 20

Lab 4 graded

- If you haven’t turned in Lab 4 yet, send me an email when you turn it in

Overview

- Unix pipelines
- Signals

Unix pipelines

- A special case of shell redirection where the output of one process becomes the input of another
- <: redirect stdin
- >: redirect stdout (overwrite)
- >>: redirect stdout (append)
- |: pipes output from one process to input of other

Review some commands

- `cat <file>`: reads from `<file>`, outputs to stdout
- `ls`: lists files in current directory
- `grep <string>`: reads from stdin, outputs any line containing `<string>` to stdout
- `wc -l`: reads from stdin, outputs # of lines read to stdout

Pipeline examples

- `ls | grep fork`
- `cat forktest.c | grep fork`
- `ls | grep fork | wc -l`
- `cat forktest.c | grep "#include" | wc -l > count.txt`
- `cat forktest.c | grep "#include" | sort | uniq | wc -l`

How are pipelines implemented?

- `pipe()`: a system call that returns two file descriptors: a “read end” file descriptor and a “write end” file descriptor
  - The kernel pipes what’s written from the “write end” so that it can be read from the “read end”
- `dup2()` to replace one process’s stdout with the “write end” descriptor
- `dup2()` to replace the other process’s stdin with the “read end” descriptor

Takeaway

- `ls | grep fork`
- `cat forktest.c | grep fork`
- `ls | grep fork | wc -l`
- `cat forktest.c | grep "#include" | wc -l > count.txt`
- `cat forktest.c | grep "#include" | sort | uniq | wc -l`

Signals

- How the operating system notifies processes of events
- Possible sources:
  - Generated by something the process itself does
  - Generated by some external event
  - Sent by other processes

ID	*Name*	*Signal Description*	*Shell Shortcut*	*Default Action*	*Override?*
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	**No**
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	**No**
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	**No**
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes

signal system call

```
sighandler_t signal(int signum, sighandler_t handler);
```

- Override default signal action
- Handler can be the following dispositions:
  - `SIG_IGN`: ignore the signal
  - `SIG_DFL`: restore default handler
  - A custom function to call to handle the signal

obiwan

- Signal handlers demo!
- How can we kill it?

kill system call

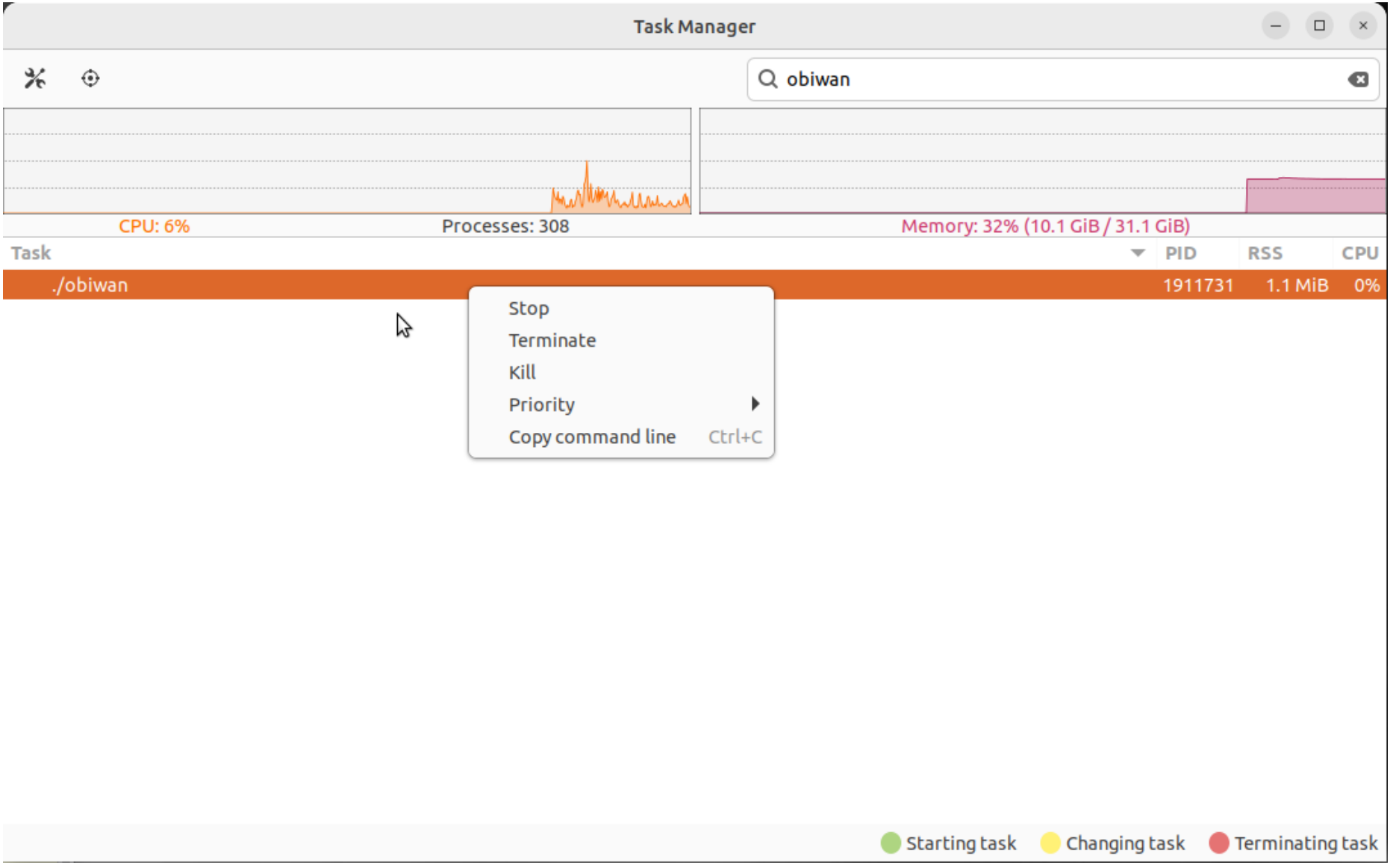
```
int kill(pid_t pid, int signum);
```

- Name is misleading
- This is the system call to send *any* signal to a process (not just SIGKILL)

kill command

- Thin wrapper around the kill system call
- `kill 12345 # send SIGTERM to pid 12345`
- `kill -INT 12345 # send SIGINT to pid 12345`
- `kill -KILL 12345 # send SIGKILL to pid 12345`
- `kill -9 12345 # send SIGKILL to pid 12345`

ID	*Name*	*Signal Description*	*Shell Shortcut*	*Default Action*	*Override?*
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	**No**
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	**No**
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	**No**
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes


Task Manager
obiwan
CPU: 6%
Processes: 308
Memory: 32% (10.1 GiB / 31.1 GiB)
Task
PID
RSS
CPU
./obiwan
1911731
1.1 MiB
0%
Stop
Terminate
Kill
Priority
Copy command line
Ctrl+C
Starting task
Changing task
Terminating task

Notes on SIGSEGV

- You can handle SIGSEGV!
  - You virtually never should
  - The faulting instruction is restarted
- You can also send SIGSEGV via kill!
  - But why!?!?

Remember: signal is a syscall

- The kernel maintains the process's signal handlers/dispositions
- This occurs in something called a *signal disposition table*
- This table is inherited by children on `fork()`
  - All custom signal handlers are set to default after `exec()`
  - *But ignored signals remain ignored*

Our working model for a process

- Registers
- Memory
- PID
- File descriptor table
- Signal disposition table 🆕

How are signals delivered?

- The kernel delivers a signal to a process
- But if you, say, press Ctrl+C, nothing *immediately* happens
- The process doesn’t act on the signal until it *receives* it

Pending signals

- After the signal is sent, before it is received, the signal is pending
- Signals can only be processed by actively running processes
- The kernel maintains a bit vector, where the ith bit is 1 if signal #i is pending
  - E.g., if SIGINT is pending, then bit 2 is 1

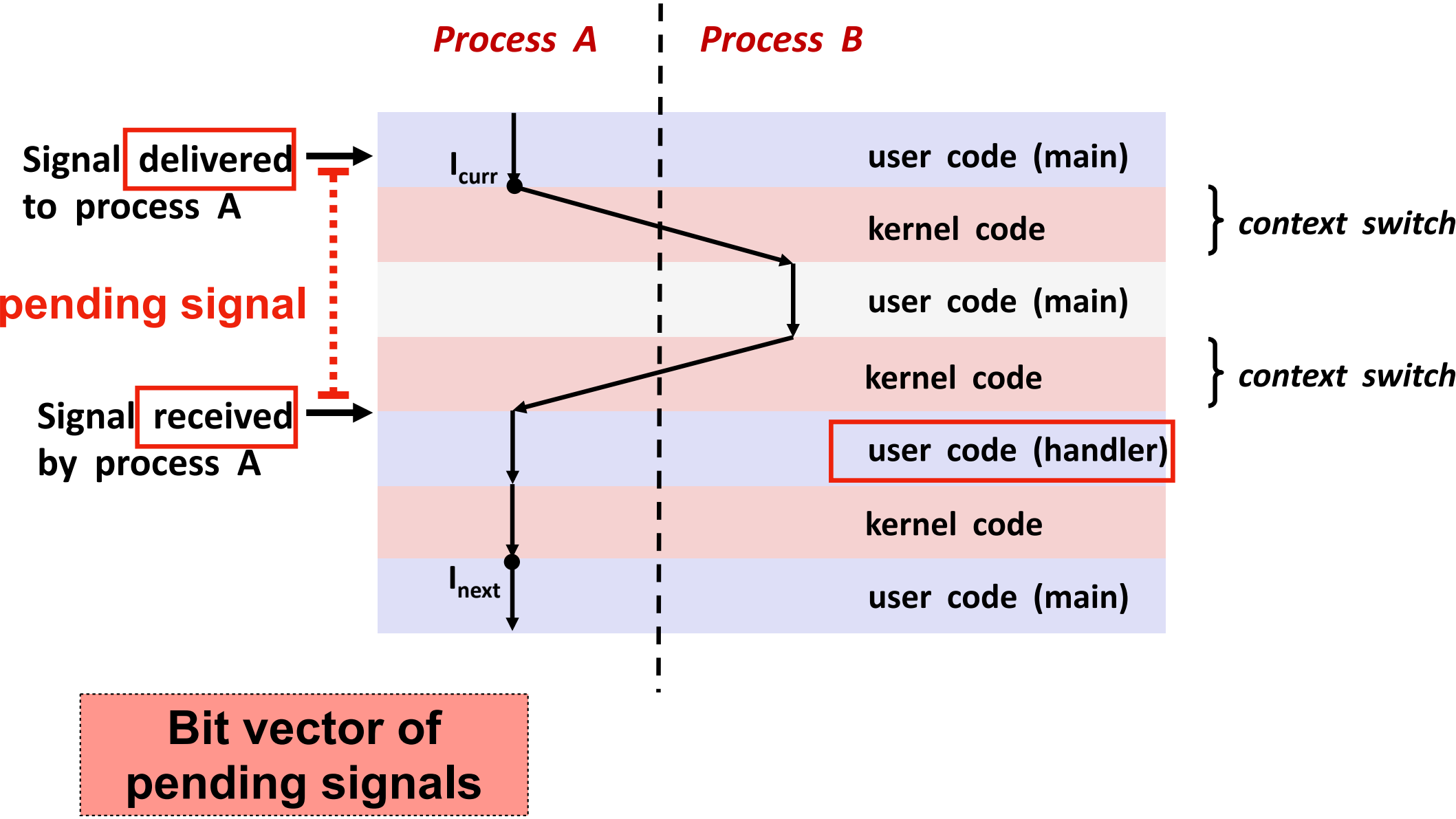
Bit vector of pending signals

- Consequence: only one signal can be pending at any time
- If you send a second SIGINT before the first is received, then the second is lost
- Therefore, received signals may be less than sent signals

ID	*Name*	*Signal Description*	*Shell Shortcut*	*Default Action*	*Override?*
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	**No**
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	**No**
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	**No**
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes

Signals received on context switch

- Even if the receiving process is currently executing, it will not immediately process the signal
- The process receives the signals the next time it is switched in…


Process A
Process B
Signal delivered to process A
pending signal
Signal received by process A
I_{curr}
I_{next}
user code (main)
kernel code
user code (main)
kernel code
user code (handler)
kernel code
user code (main)
context switch
context switch
Bit vector of pending signals

Our working model for a process

- Registers
- Memory
- PID
- File descriptor table
- Signal disposition table
- Pending signal bit vector 🆕

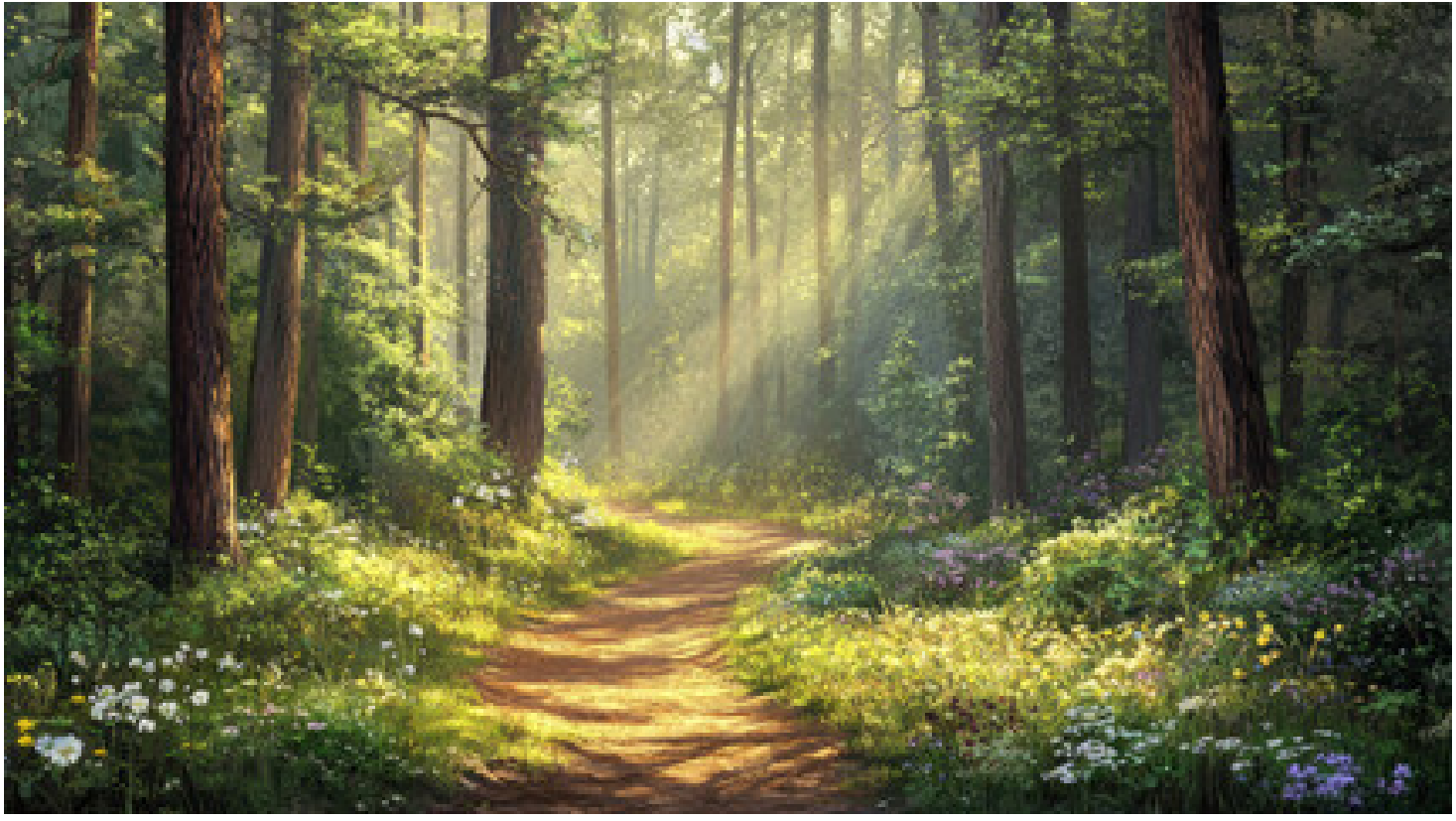
Exceptions to the rule

- Signals that are generated by synchronous exceptions do not need a context switch
- They are delivered immediately
- Examples:
  - SIGSEGV
  - SIGFPE

Order of signals not guaranteed

- When a process is switched in, the kernel scans the bit vector for 1 bits
- When multiple signals are pending, Linux begins by checking the lowest bits first
  - Hence, on Linux, the lowest signal numbers are received first
- Other Unix operating systems may have different behavior

Where do processes go when they die?



Where do processes go when they die?

- Nowhere at first!
- The kernel keeps them around because their parent may try to reap them
  - Thus, their PID cannot be reused until they are reaped (their parent might call `waitpid()` on them)
- If you don't call `waitpid()` on your dead children, you will "leak" PIDs
  - Think calling `malloc()` without ever calling `free()`

- These dead processes waiting to be reaped have a name….



Let’s revisit forktest…

- Let’s wait to reap our dead child
- Does killing it work?


YOU CANNOT KILL
WHAT IS ALREADY
DEAD

What if the parent never reaps them?

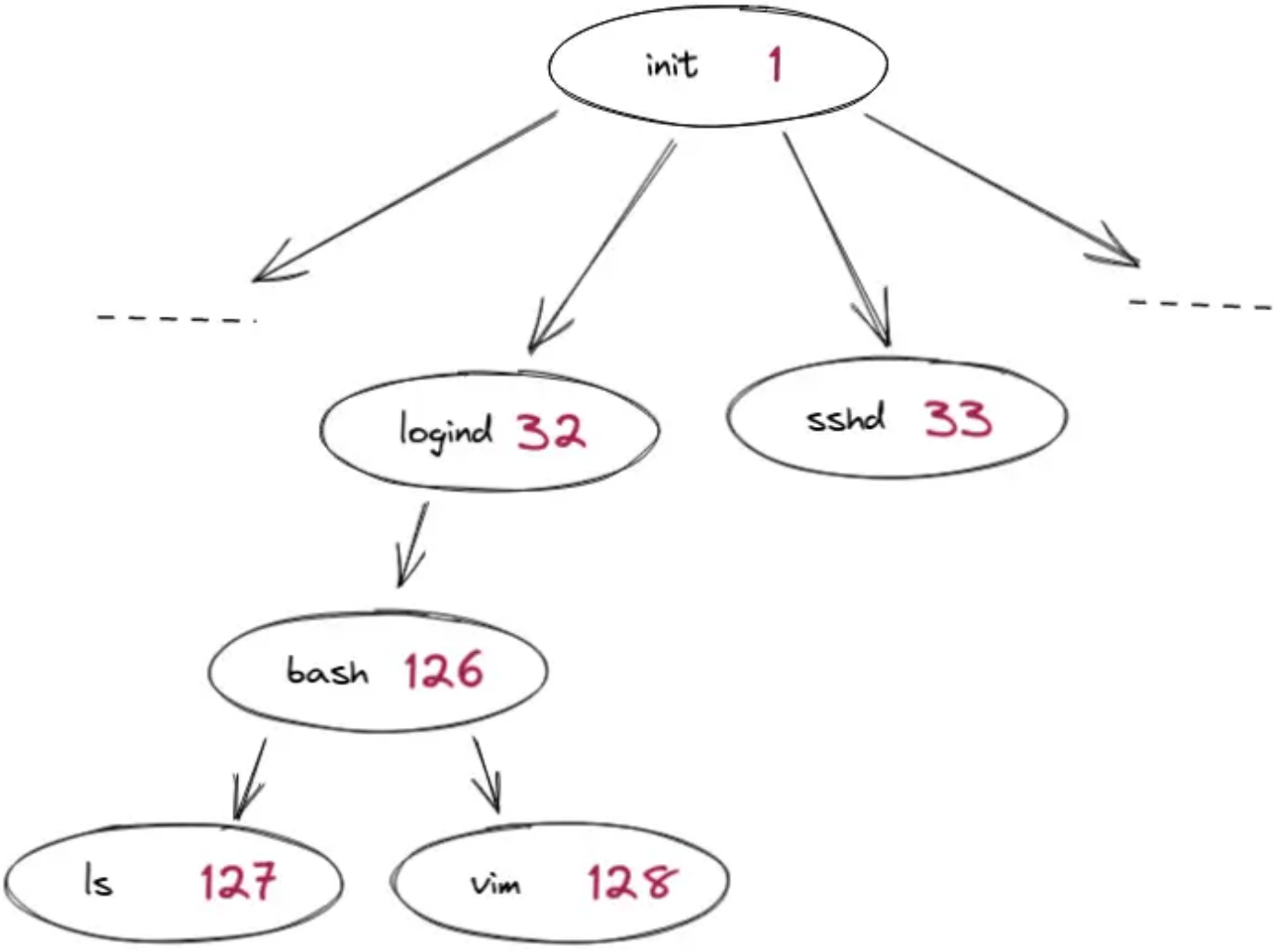
- The parent dies before reaping them?
- For instance, if a process dies without free()ing memory, we learned the kernel frees it
- What happens to zombie children when their parent dies?
- We need to discuss more broadly what happens to children processes when the parent dies…

Orphans

- When your parent dies, you become an *orphan*
- Orphan processes are all adopted by a daemon…

Daemons

- A *daemon* is a process that runs continuously in the background without a GUI
- There are many daemons:
  - E.g., `sshd` (the ssh server on krebs you sign into)
  - Or `logind` (a daemon for handling user sessions)
- But there is another we have seen…


init 1
logind 32
sshd 33
bash 126
ls 127
vim 128

Orphans

- So if you are living, your new parent becomes init (pid 1)
- But same for if you are dead (a zombie)
- If you are dead, the init process immediately reaps you after adopting you

```
while (true) {
    Print command prompt.
    Read command line from user.
    Parse command line.
    If command is built-in, execute it.
    Else, fork process
        in child:
            Execute requested command with exec
                                (never returns)
        in parent:
            if foreground:
                Wait for child to complete with waitpid
}
```

How to reap background jobs?

Signals!

- But how do we reap background processes?
- How do we know when they’ve died?
- Which of these signals seems like it may be useful…

ID	*Name*	*Signal Description*	*Shell Shortcut*	*Default Action*	*Override?*
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	**No**
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	**No**
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	**No**
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes

Signals!

- When a background child dies, we will receive SIGCHLD
- We can reap it there…

```c
int main() {
    pid_t pid;

    signal(SIGCHLD, sigchd_handler); // install signal handler

    while (1) {
        // print prompt, read cmd from user, etc.
        if ((pid = fork()) == 0) {
            exec(...); // child: run target program
        }
        // parent: wait for child to exit if foreground
    }
    return 0;
}
```

```c
void sigchld_handler(int sig) {
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        // reaped child with process ID pid
    }
}
```

Why waitpid() in a loop?

- Remember that pending signals are tracked in a bitvector
- If more than one of the same signal arrives in short succession, only one of that signal will be received
- This is true of SIGCHLD just like any signal
- Thus, one SIGCHLD may correspond to the death of more than one child

Why waitpid() in a loop?

- `waitpid()` only reaps one child at once
- Since we can have more than one process running in the background
- We may need to reap multiple dead children
- To avoid zombie process leaks

waitpid arguments

- `pid`: pid of child to wait on, -1 to wait on any
- `wstatus`: information on exit status of process
- `options`: ???

waitpid arguments

- One option is important for our shell:
  - `WNOHANG`
- This option means don’t wait for the next child to die if there isn’t already a dead one to reap
- We want to use this option when waiting for **background** processes because we never want to wait for them to die

Zombies

- Zombies aren’t necessarily bugs
- Zombies are part of the natural lifecycle of a process
- All processes become zombies for some amount of time after they die
- What you *should* fix:
  - Children that are zombies for a very long time
  - The number of zombies accruing over time

Signal exercises

ID	*Name*	*Signal Description*	*Shell Shortcut*	*Default Action*	*Override?*
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	**No**
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	**No**
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	**No**
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes

Thursday: hack on Lab 5

- Office hours are canceled Thursday due to a faculty candidate job talk at that same time
- The job talk is in this classroom immediately following class!
- Friday: we will learn where mallocs come from

TODO

- Assignment 5 assigned due November 20