# Processes Continued: exec, wait, signals

## 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

Poll Everywhere

pollev.com/tqm

- Any Philly food/restaurant recommendations?

# Administrivia

- First Assignment (penn-vector)
  - Already posted
  - **"Due" this Friday, 9/04**
  - Mostly a C refresher

- Pre-Semester Survey
  - Anonymous
  - Short!
  - Already posted
  - **Due Wednesday, 9/09**

# Administrivia

- Second Assignment (penn-shell)
  - Releases **TONIGHT**
  - Partner assignment – start looking! Thread posted on Ed later to assist
  - Milestone 0 due Friday 9/11
  - Milestone 1 due Friday 9/18
  - Full assignment due Wednesday 9/23
  - Basic shell implementation
    - System calls, processes, I/O redirection, pipes, job control, and more!

- Check the course calendar for office hour times!

# Lecture Outline

- **Processes & Fork refresher**
- exec
- Wait & process states
- Hardware interrupts
- Software signals
- Process states updated

# Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent ***address space (more on that later!)***
  - Each process is scheduled by the OS
    - Without using some functions we have not talked about (yet), there is no way to guarantee the order processes are executed
  - Processes are created by `fork()` system call
    - Only difference between processes is their process id and the return value from `fork()` each process gets

# Revisting fork()

```
pid_t fork();
```

- Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
- Fork returns the ***pid of the child*** in the parent, but ***0 in the child.***
- The new process has a separate ***virtual address space*** from the parent

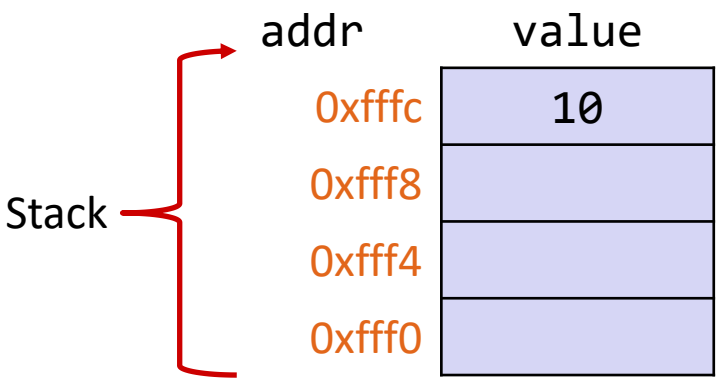
```
int main(){

    int x = 10;
    pid_t child = fork();
    if(child == 0) x--;
    printf("This is the pid: %d\n", child);
}
```

*almost everything

# Parent vs Child: Separate Address Space

```
int main(){

    int x = 10;
    pid_t child = fork();
    if(child == 0) x--;
    printf("This is the pid: %d\n", child);
}
```

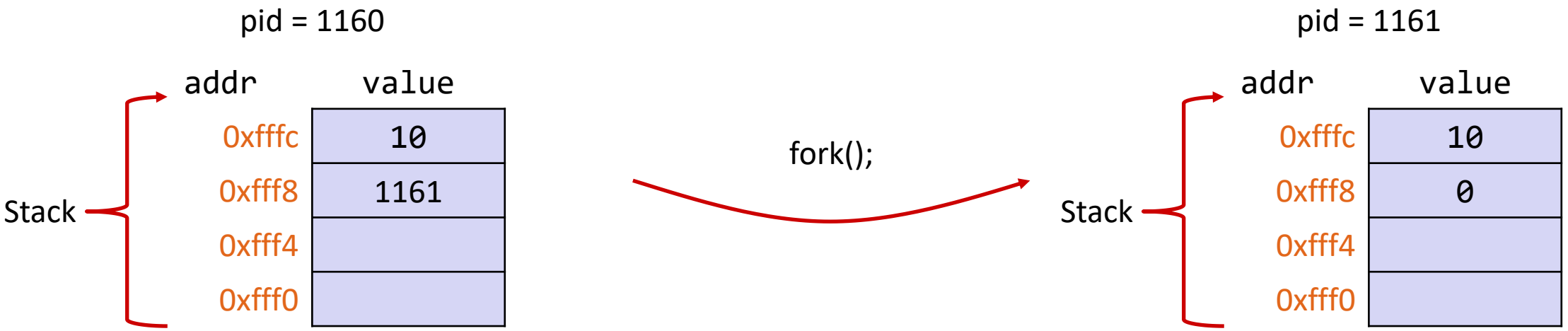
pid = 1160




*almost everything

# Parent vs Child: Separate Address Space

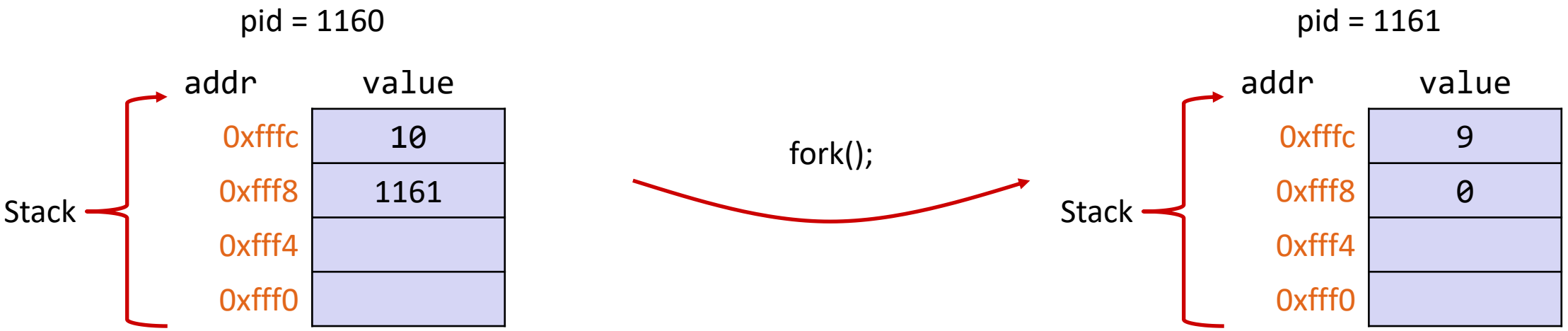
```
int main(){

        int x = 10;
        pid_t child = fork();
        if(child == 0) x--;
        printf("This is the pid: %d\n", child);
}
```




*almost everything

# Parent vs Child: Separate Address Space

```
int main(){

        int x = 10;
        pid_t child = fork();
        if(child == 0) x--;
        printf("This is the pid: %d\n", child);
}
```




```
root@eb681338c92b:~/workspace/codeplayground/lecture2# ./simple_fork
This is the pid: 1198
This is the pid: 0
```

*almost everything

# Exiting a Process

```
void exit(int status);
```

- Causes the current process to exit normally
- Automatically called by **`main()`** when it returns
- Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
  - This is the same int returned by **`main()`**
- The exit status is accessible by the parent process with **`wait()`** or **`waitpid()`**.

Poll Everywhere

pollev.com/tqm

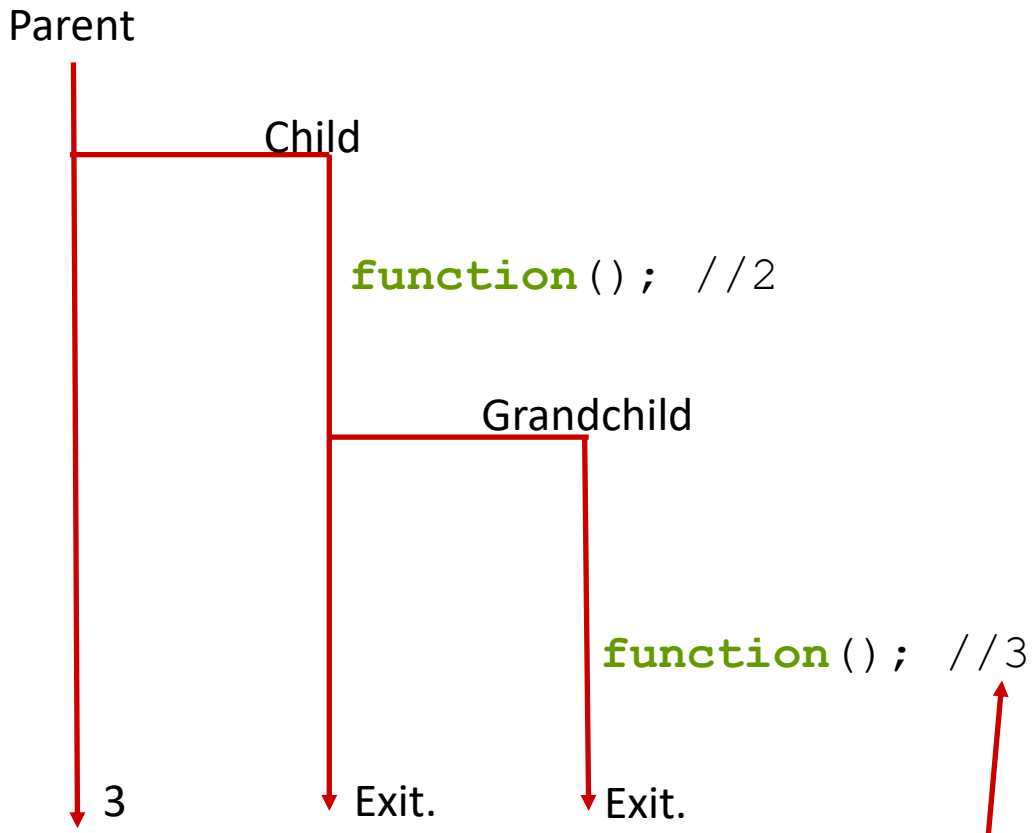
```c
int global_num = 1;

void function() {
  global_num++;
  printf("%d\n", global_num);
}
int main() {
  pid_t id = fork();

  if (id == 0) {
    function();
    id = fork();
    if (id == 0) {
      function();
    }
    return EXIT_SUCCESS;
  }

  global_num += 2;
  printf("%d\n", global_num);
  return EXIT_SUCCESS;
}
```

- How many numbers are printed?
- What number(s) get printed from each process?

Poll Everywhere

pollev.com/kyrdo

```c
int global_num = 1;

void function() {
  global_num++;
  printf("%d\n", global_num);
}
int main() {
  pid_t id = fork();

  if (id == 0) {
    function();
    id = fork();
    if (id == 0) {
      function();
    }
    return EXIT_SUCCESS;
  }

  global_num += 2;
  printf("%d\n", global_num);
  return EXIT_SUCCESS;
}
```




*Remember, the child starts with an *identical* copy of the it's parents memory.

Poll Everywhere

pollev.com/tqm

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 3; i++) {
    fork();
  }

  printf(":)\n");
  return EXIT_SUCCESS;
}
```

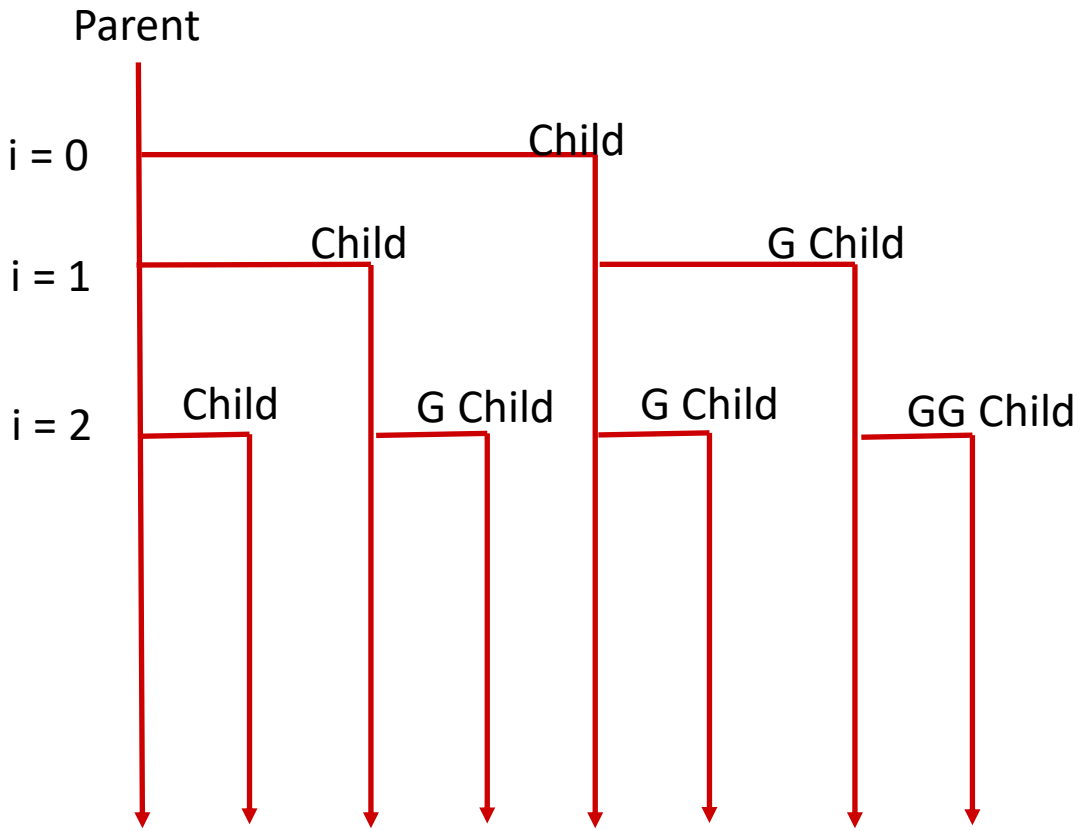
Poll Everywhere

pollev.com/tqm

- How many times is ":)" printed?

```c
int main(int argc, char* argv[]) {
  for (int i = 0; i < 3; i++) {
    fork();
  }

  printf(":)\n");
  return EXIT_SUCCESS;
}
```



# Lecture Outline

- Processes & Fork refresher
- **exec**
- Wait & process states
- Hardware interrupts
- Software signals
- Process states updated

# exec*()

- Loads in a new program for execution

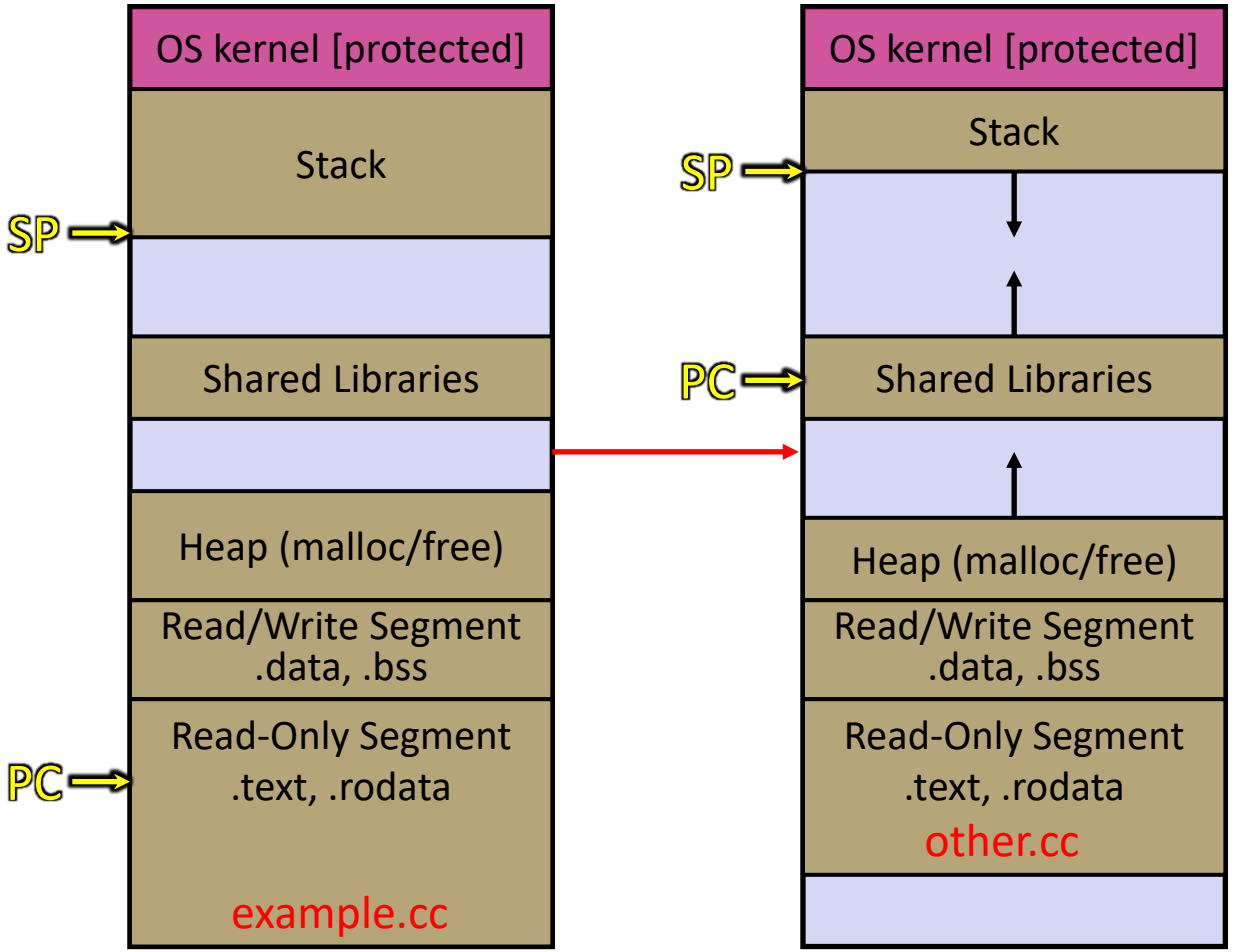
- Program Counter (rip in x86, pc in risc-v/arm) , SP, registers, and memory are all reset so that the specified program can run

# execvp()

- ```
  int execvp(const char *file,
             char* const argv[]);
  ```
- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL/nullptr as the last entry of the array
- Returns -1 on error. Does NOT return on success
  ```

# Exec Visualization

- Exec takes a process and discards or “resets” most of it




NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

# Aside: Exiting a Process

```
void exit(int status);
```

- Causes the current process to exit normally
- Automatically called by **`main()`** when main returns
- Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
  - This is the same int returned by **`main()`**
- The exit status is accessible by the parent process with **`wait()`** or **`waitpid()`**. (more on these functions next lecture)

# Exec Demo

- See `exec_example.c`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens to allocated memory when we call exec?

Poll Everywhere

pollev.com/tqm

- In each of these, how often is ":)" printed? Assume functions don't fail

```c
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[] = {"echo",
                    "hello",
                    nullptr};
    execvp(argv[0], argv);
  }

  printf(":)\n");

  return EXIT_SUCCESS;
}
```

```c
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    return EXIT_SUCCESS;
  }

  printf(":)\n");

  return EXIT_SUCCESS;
}
```

Poll Everywhere

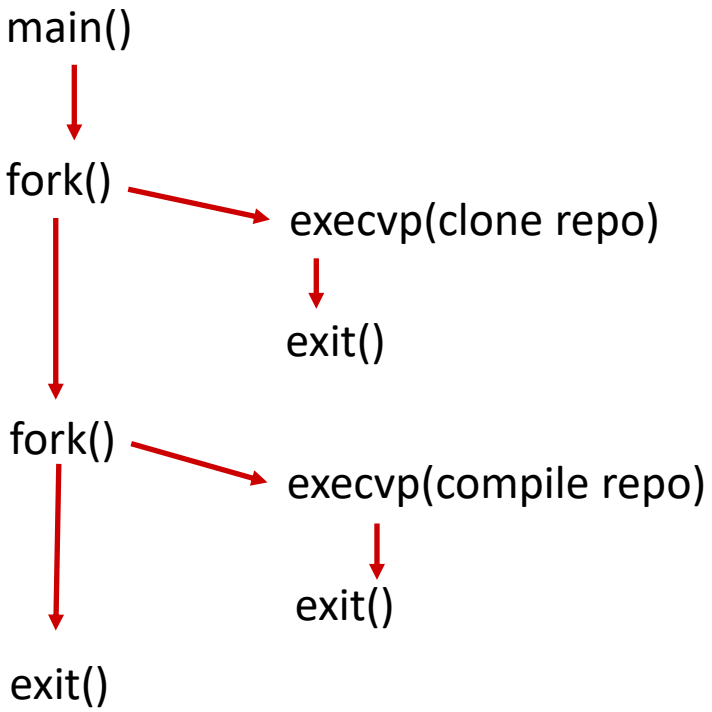
pollev.com/tqm

- What's wrong with this code? It tries to clone a repo and then compile the code in that repo. Assume the argv[]'s are set up correctly to do this.

```
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[4] = {"git", "clone", "repo_name.git", nullptr};
    execvp(argv[0], argv);
  }

  pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[4] = {"make", "-C", "repo_name", nullptr};
    execvp(argv[0], argv);
  }
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm




This code is broken. It compiles, but it doesn't do what we want. Why?

# Lecture Outline

- Processes & Fork refresher
- exec
- **Wait & process states**
- Hardware interrupts
- Software signals
- Process states updated

# From a previous poll:

```
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[4] = {"git", "clone", "repo_name.git", nullptr};
    execvp(argv[0], argv);
  }

  pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[4] = {"make", "-C", "repo_name", nullptr};
    execvp(argv[0], argv);
  }
  return EXIT_SUCCESS;
}
```

This code is broken. It compiles, but it doesn't **always** do what we want. Why?

We can't garuntee the ordering of the processes, is there any way we can synchronize things some?

# waitpid()

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

- Calling process waits for a child process (specified by **pid**) to exit
  - Also cleans up the child process
- Gets the exit status of child process through output parameter **wstatus**
- **options** are optional, pass in 0 for default options in *most* cases
- Returns process ID of child who was waited for or -1 on error

By default, only waits for in status change to "terminated"

# Execution Blocking

- When a process calls **`wait()`** and there is a process to wait on, the calling process **blocks**

- If a process **blocks** or is **blocking** it is not scheduled for execution.
  - It is not run until some condition “unblocks” it
  - For **`wait()`**, it unblocks once there is a status update in a child

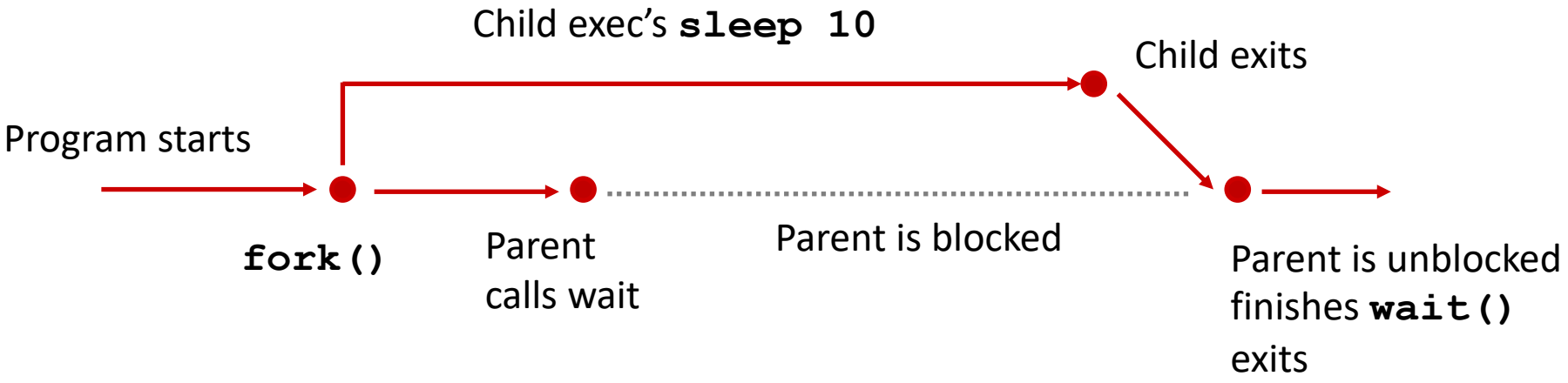
# Fixed code from poll

```
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[4] = {"git", "clone", "repo_name.git", nullptr};
    execvp(argv[0], argv);
  }
  waitpid(clone_pid, nullptr, 0); // should error check, not enough slide space :(

  pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[4] = {"make", "-C", "repo_name", nullptr};
    execvp(argv[0], argv);
  }
  return EXIT_SUCCESS;
}
```

# Demo: `wait_example`

- See `wait_example.c`
  - Brief demo to see how a process blocks when it calls wait()
  - Makes use of **`fork()`**, **`execve()`**, and **`wait()`**

- Execution timeline:



**Briefly Discuss**

- What if the child finishes first?

- In the timeline I drew, the parent called wait before the child executed.
  - In the program, it is extremely likely this happens if the child is calling **`sleep 10`**
  - What happens if the child finishes before the parent calls wait?
  - Will the parent not see the child finish?

# Process Tables & Process Control Blocks

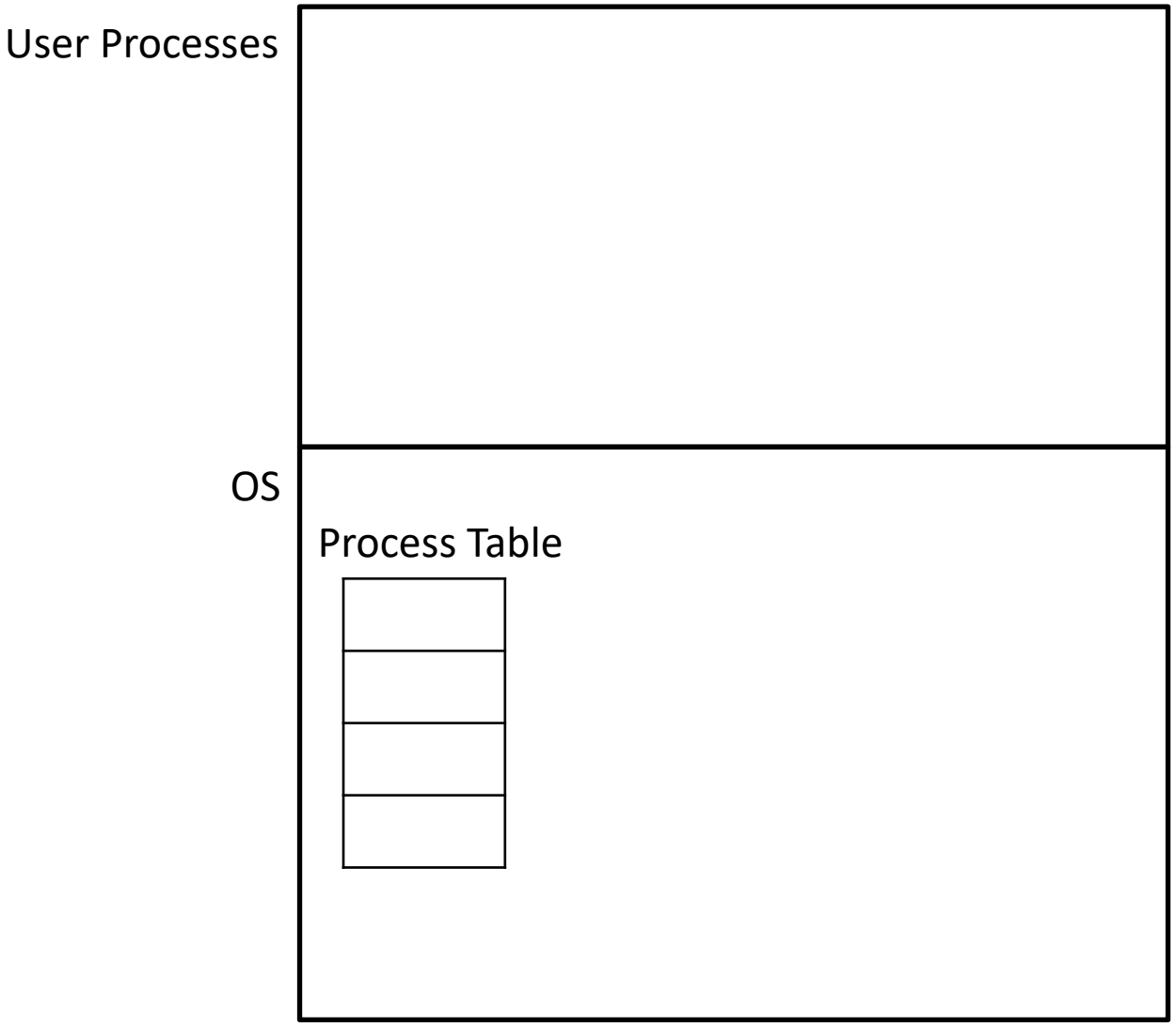
- The operating system maintains a table of all processes that aren't "completely done"

- Each process in this table has a **p**rocess **c**ontrol **b**lock (**PCB**) to hold information about it.
- A PCB can contain:
  - Process ID
  - Parent Process ID
  - Child process IDs
  - Process Group ID
  - Status (e.g. running/zombie/etc)
  - Other things (file descriptors, register values, etc)

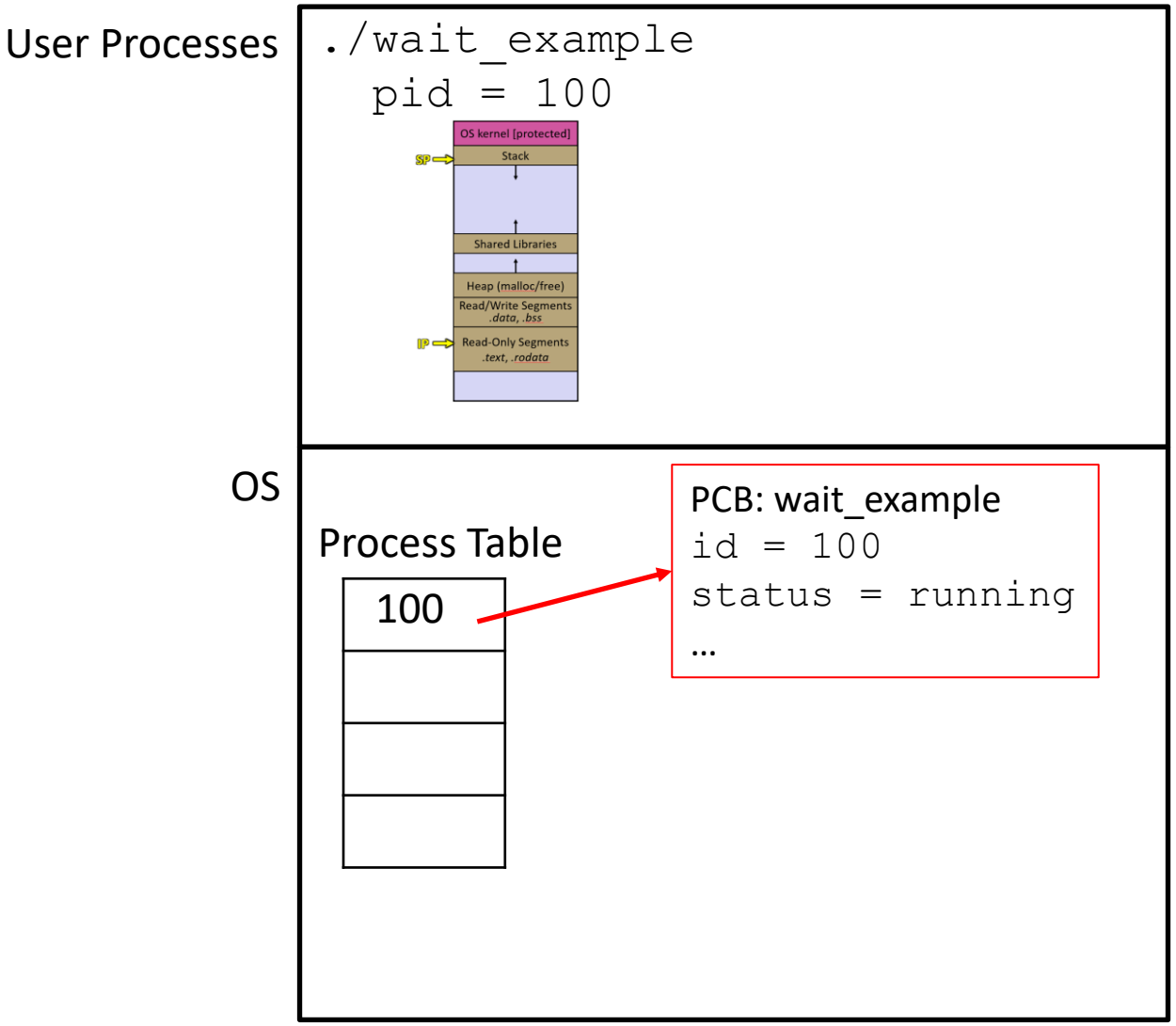
# Zombie Process

- Answer: processes that are terminated become “zombies”
  - Zombie processes deallocate their address space, don’t run anymore
  - still “exists”, has a PCB still, so that a parent can check its status one final time
  - If the parent call’s wait(), the zombie becomes “reaped” all information related to it has been freed (No more PCB entry)

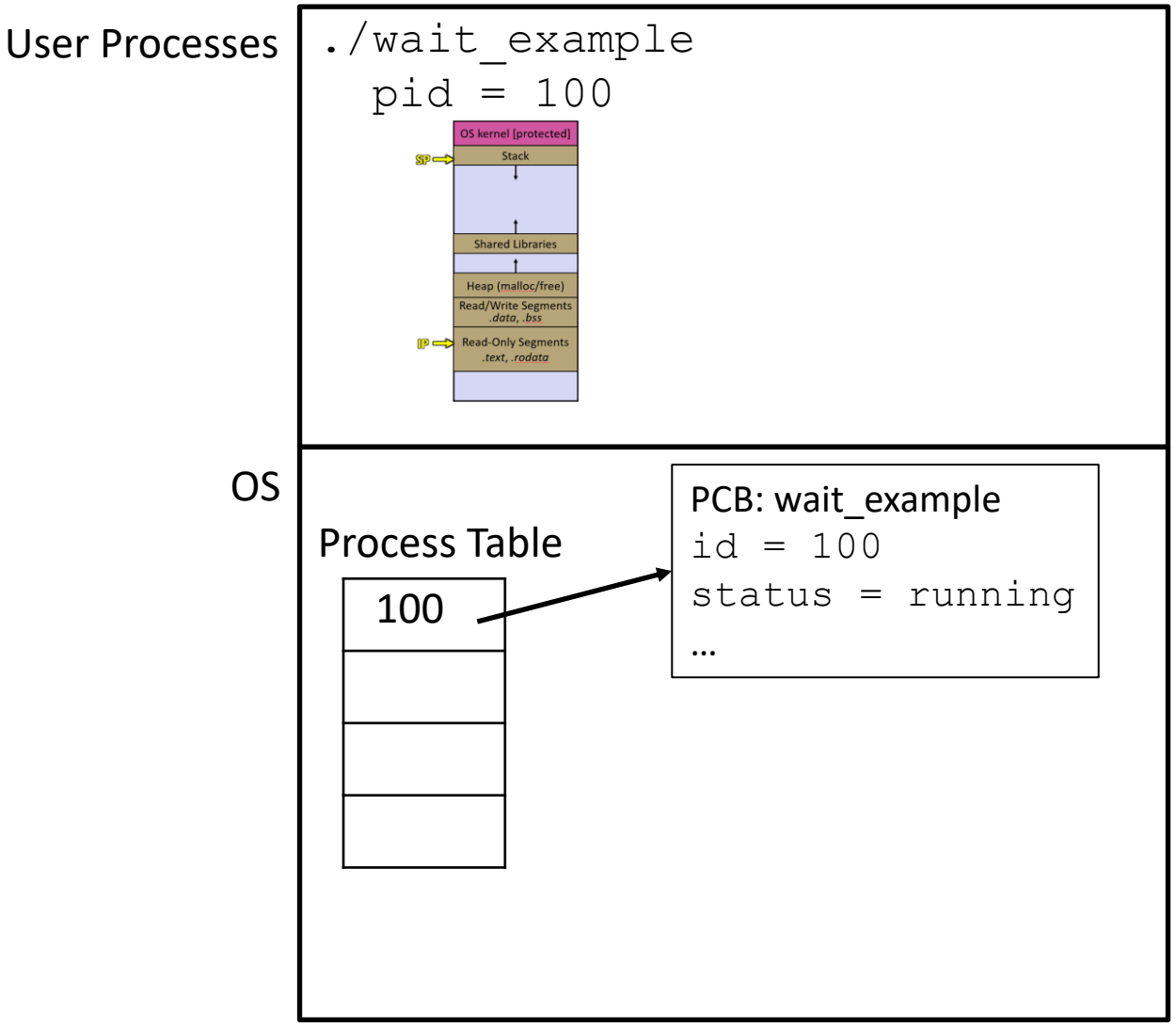
# Diagram: wait_example.c



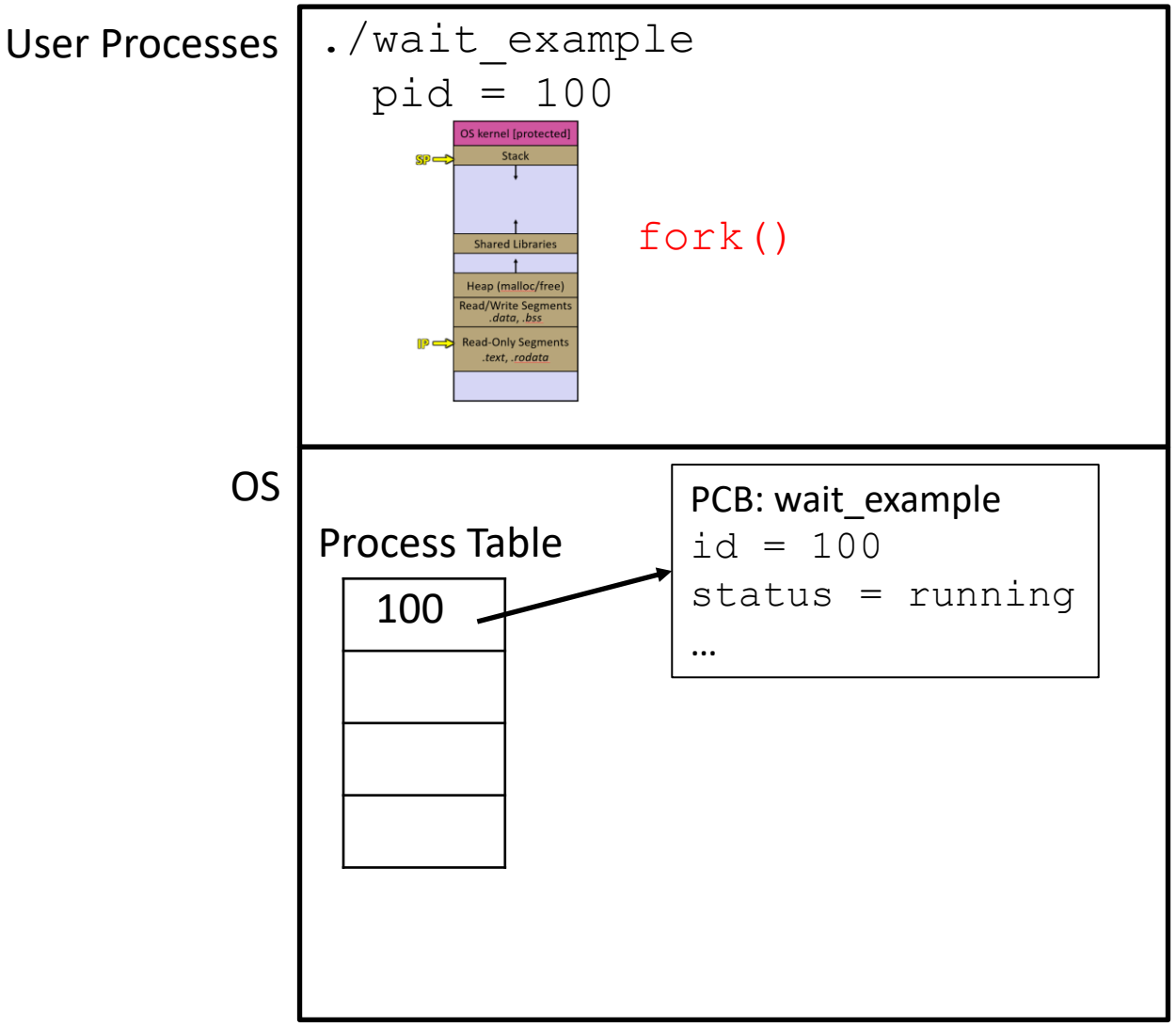
# Diagram: wait_example.c



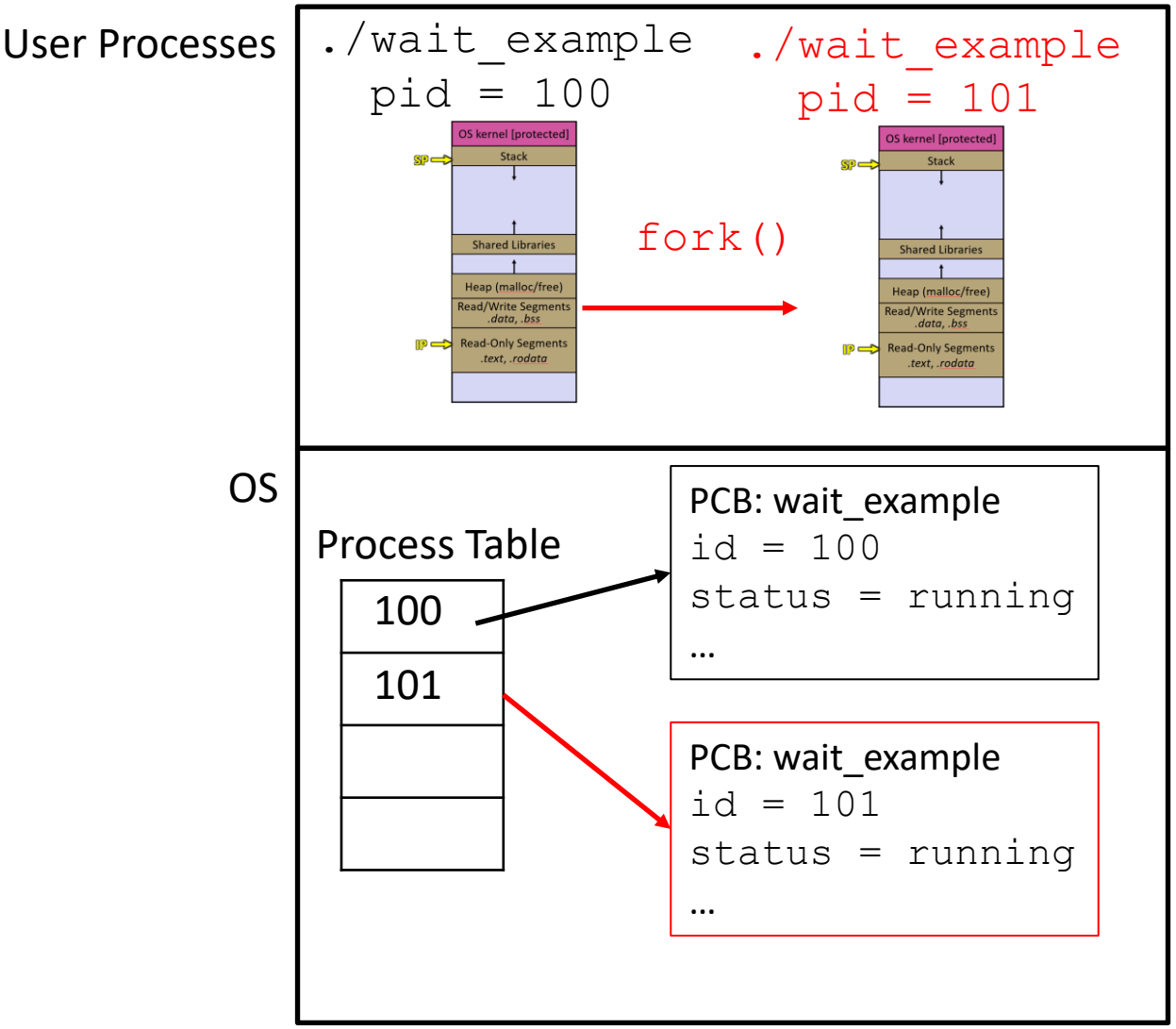
# Diagram: wait_example.c



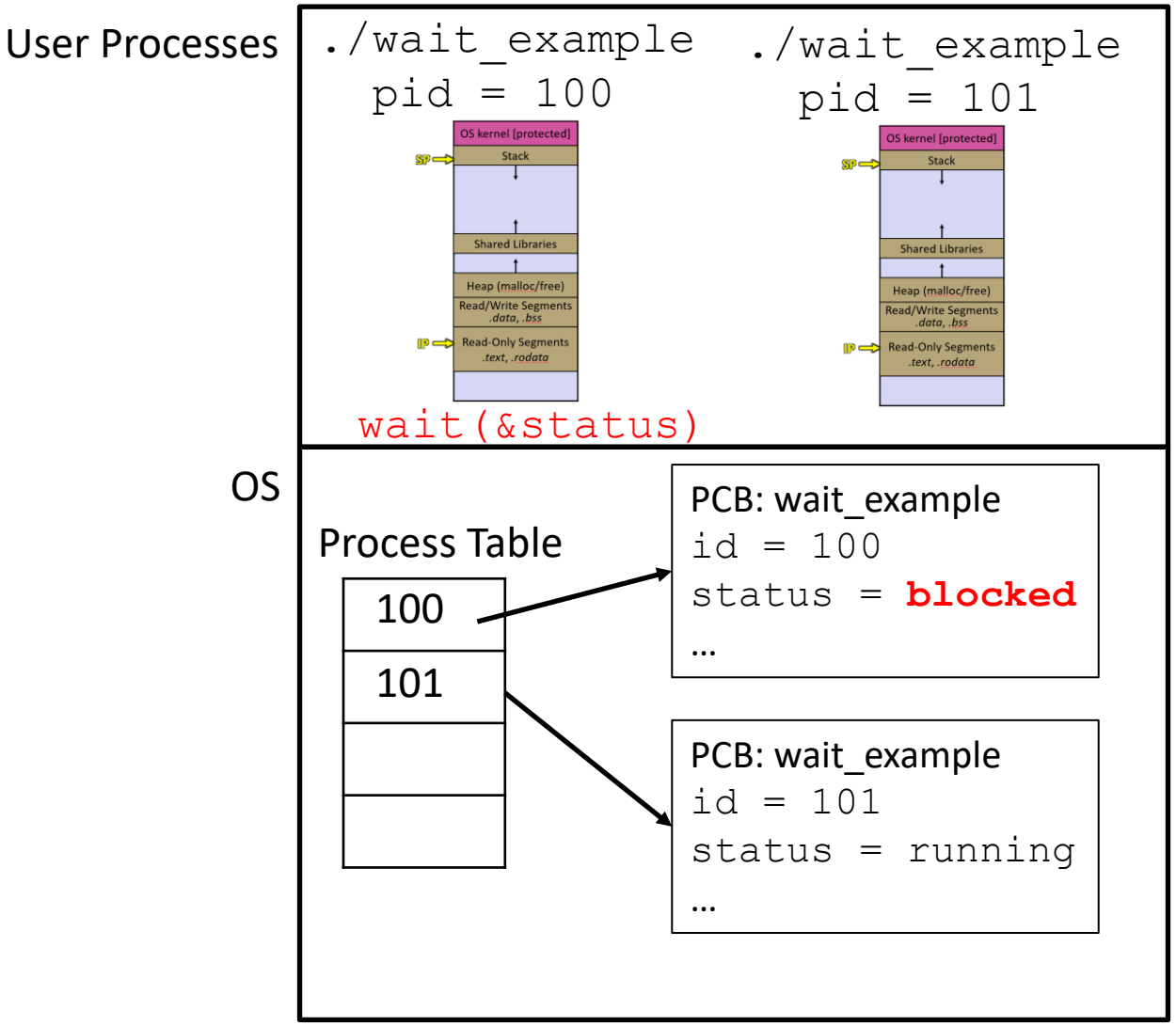
# Diagram: wait_example.c



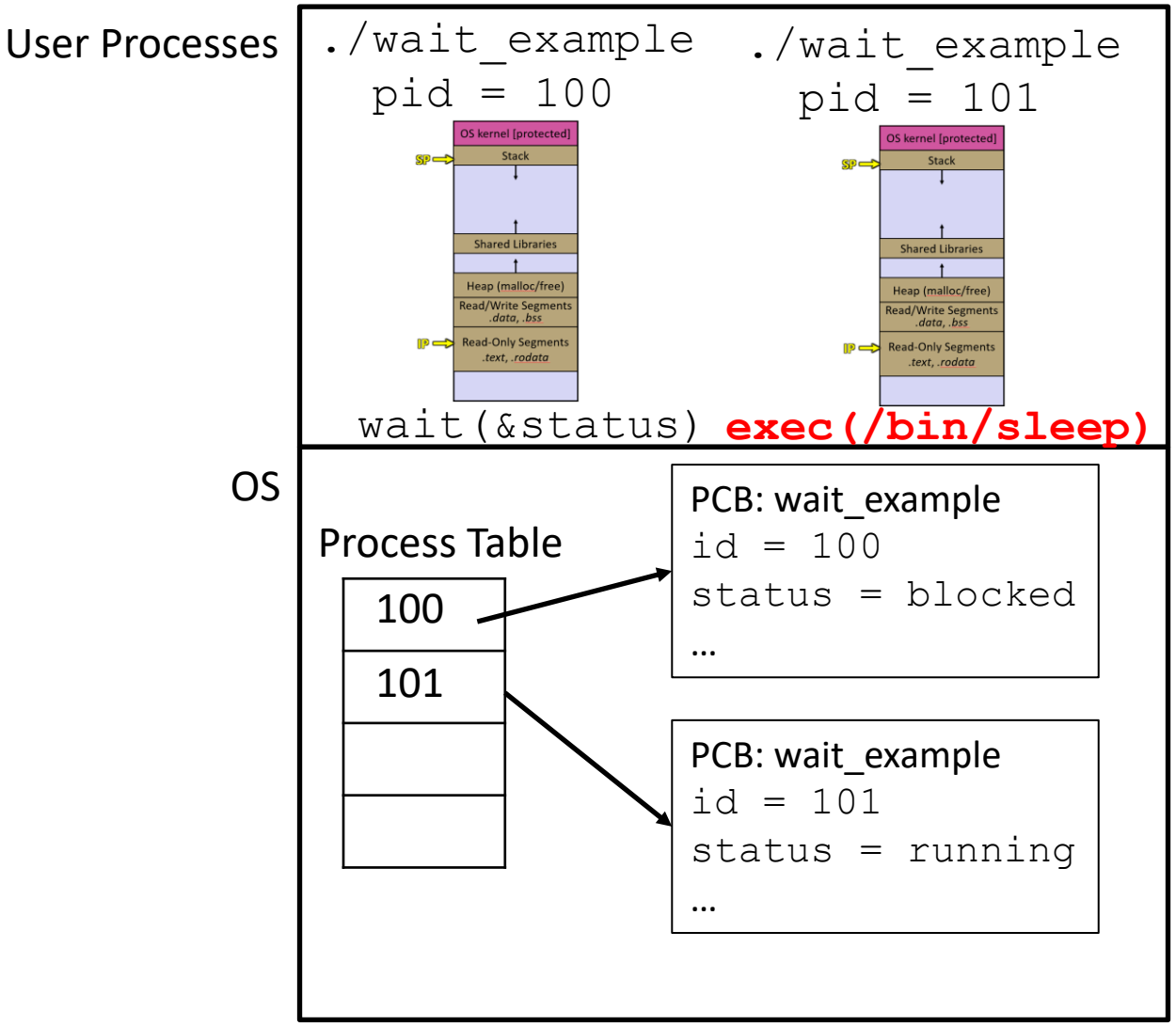
# Diagram: wait_example.c



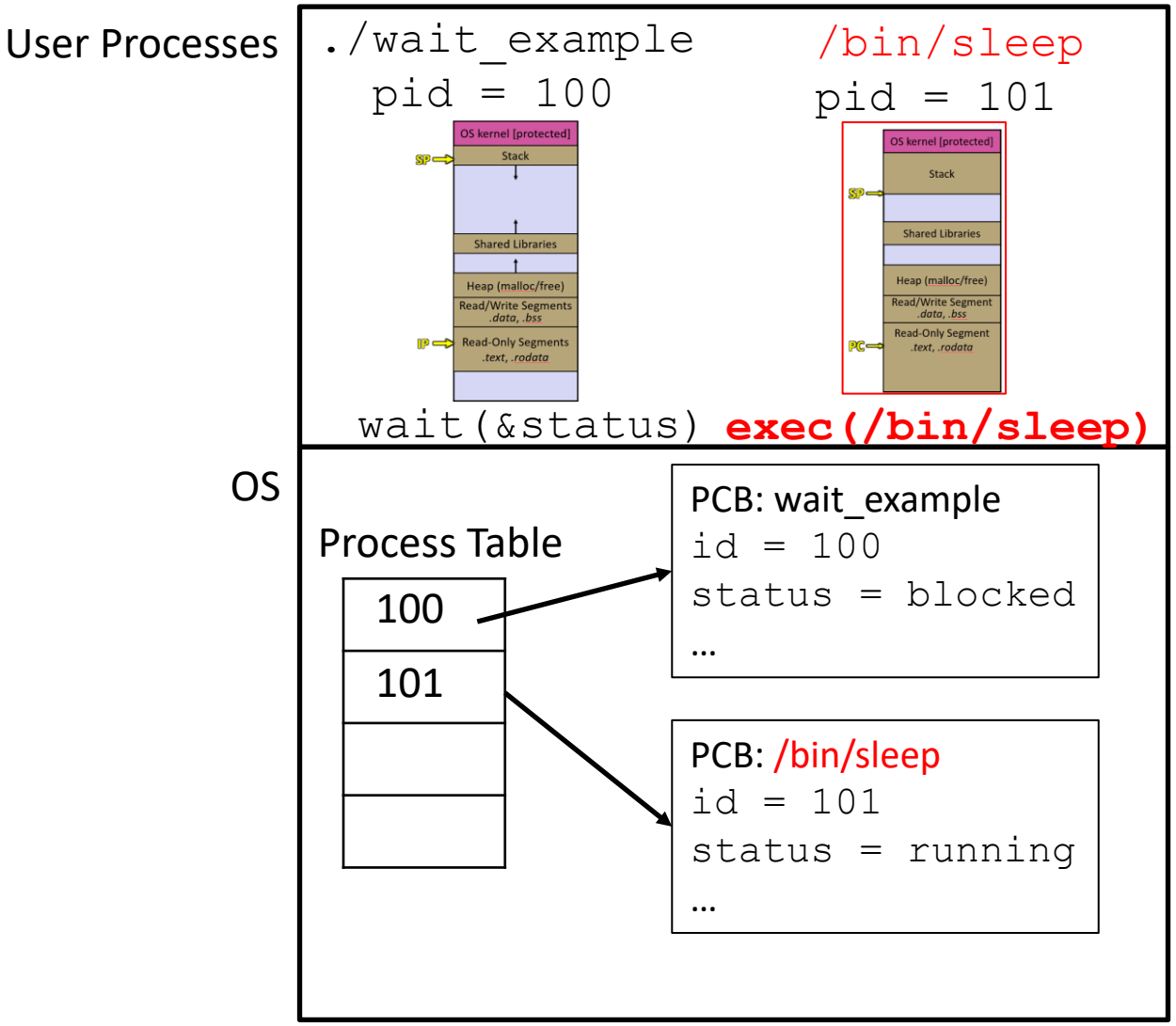
# Diagram: wait_example.c



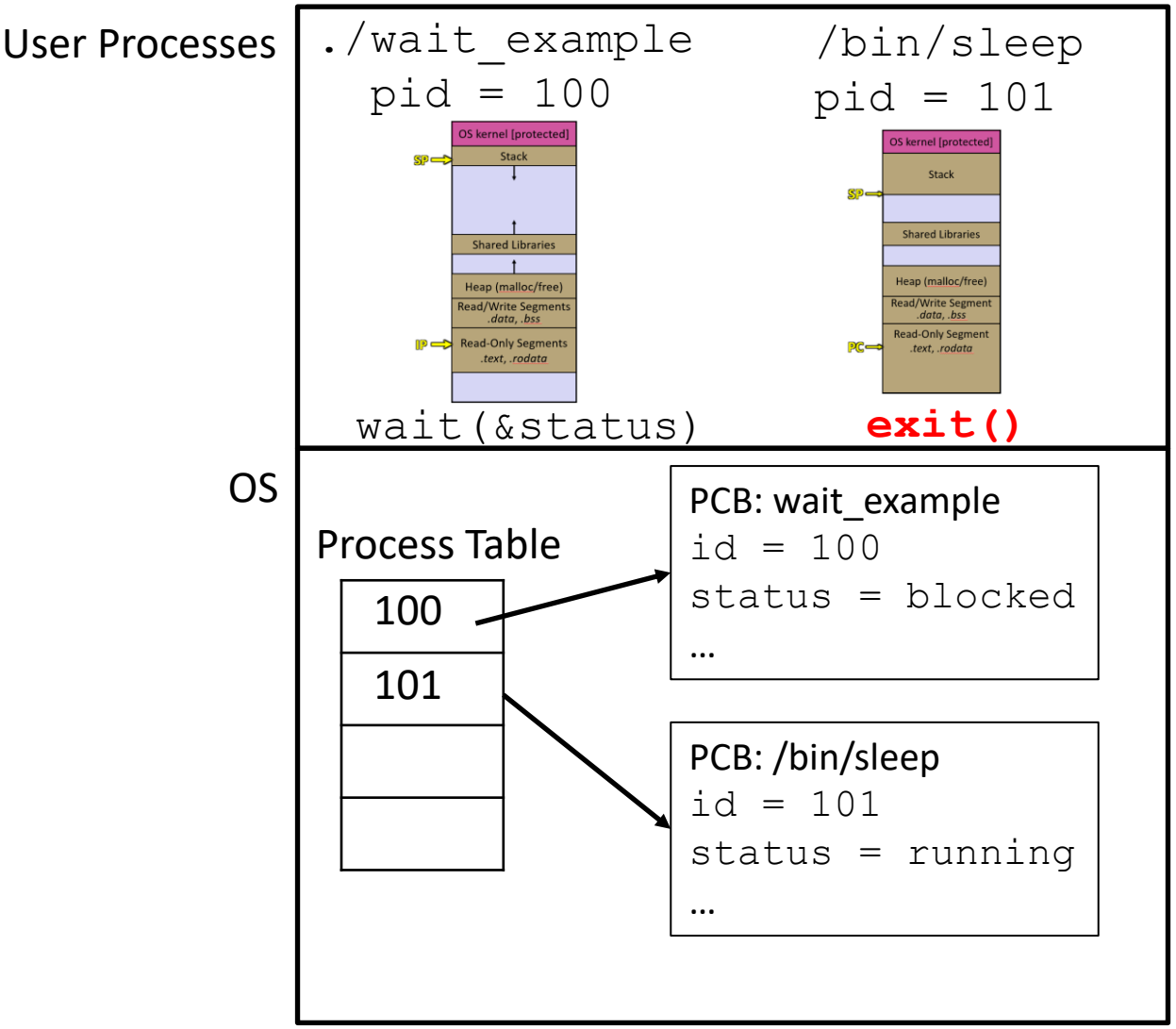
# Diagram: wait_example.c



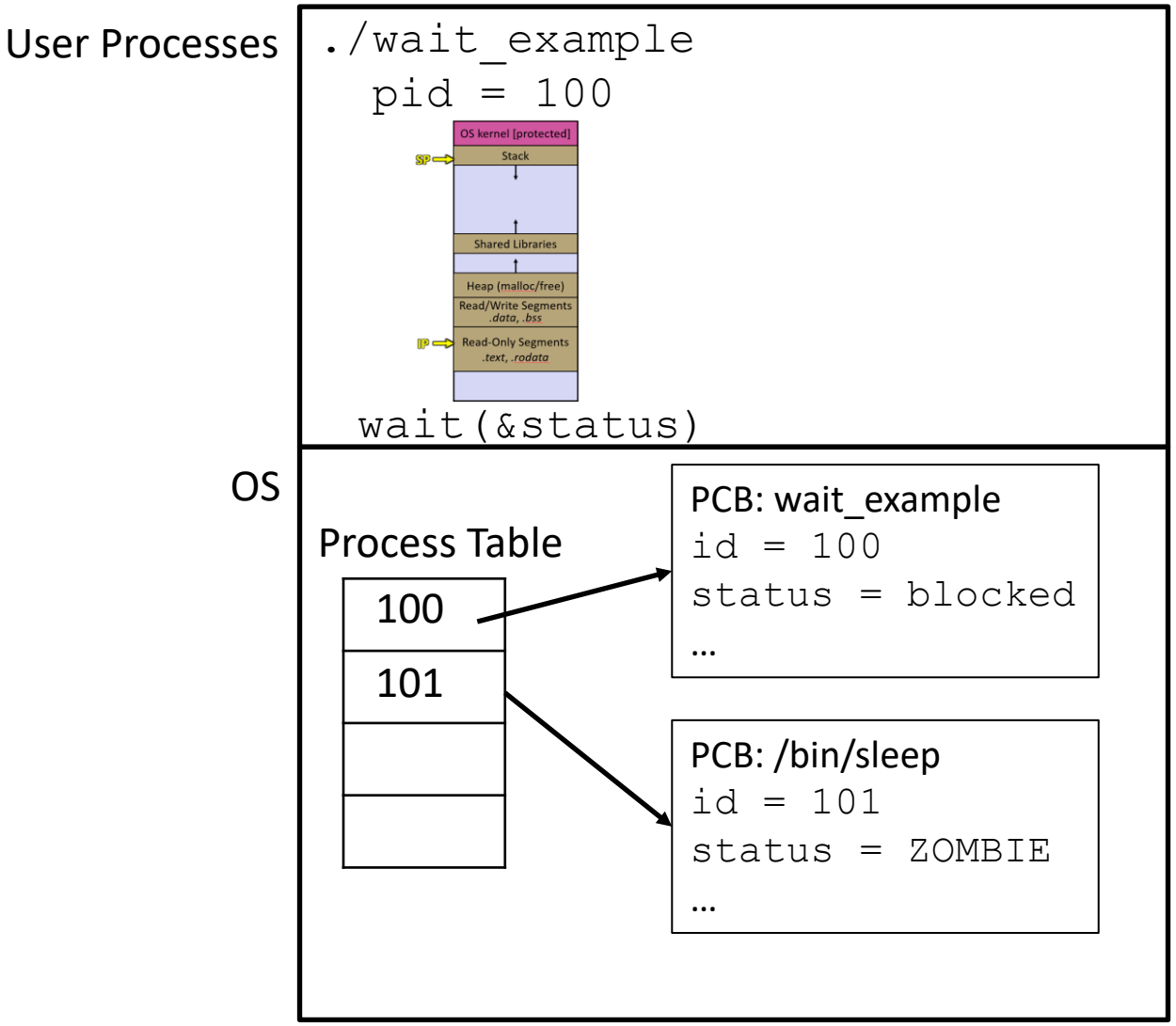
# Diagram: wait_example.c



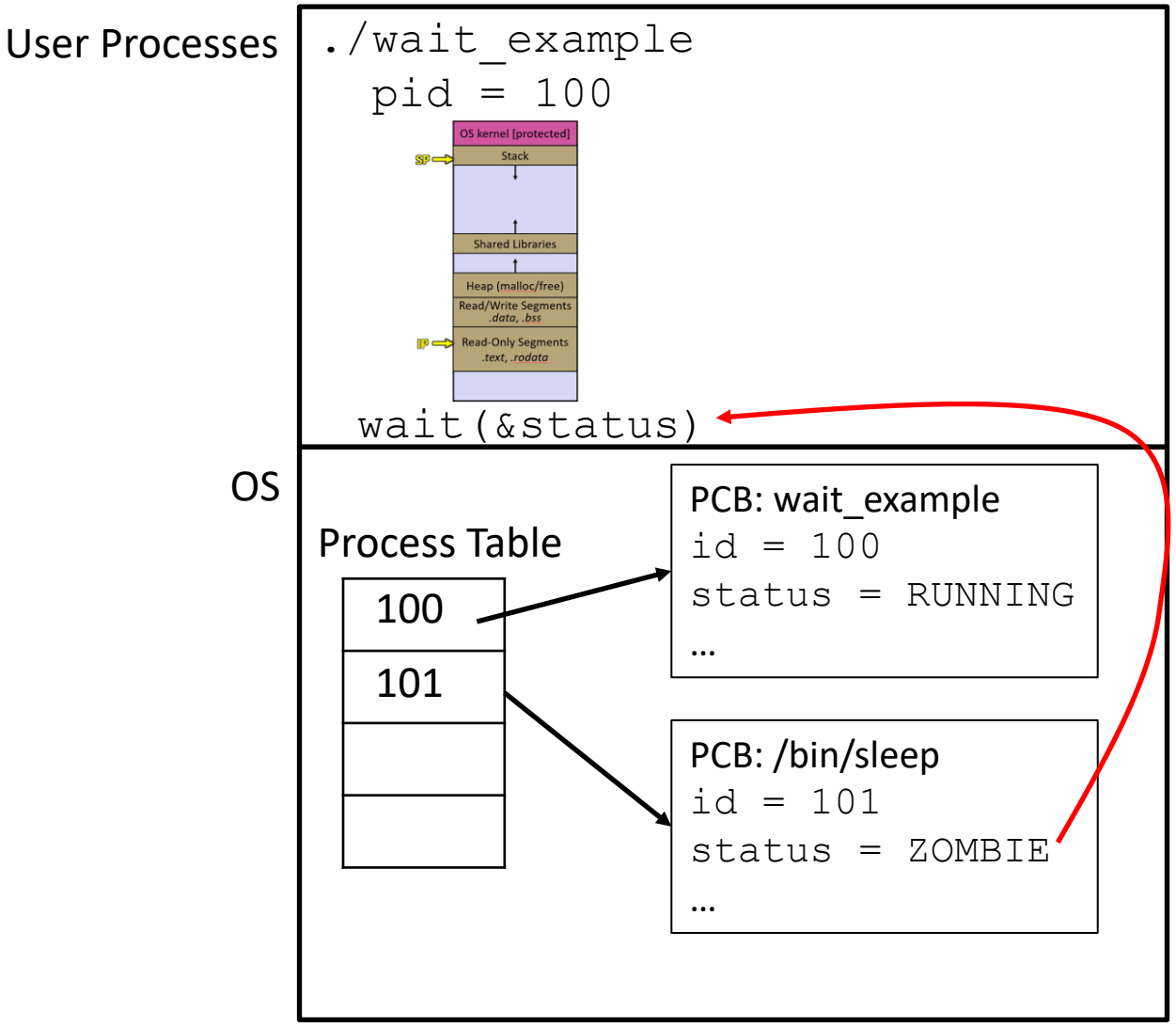
# Diagram: wait_example.c



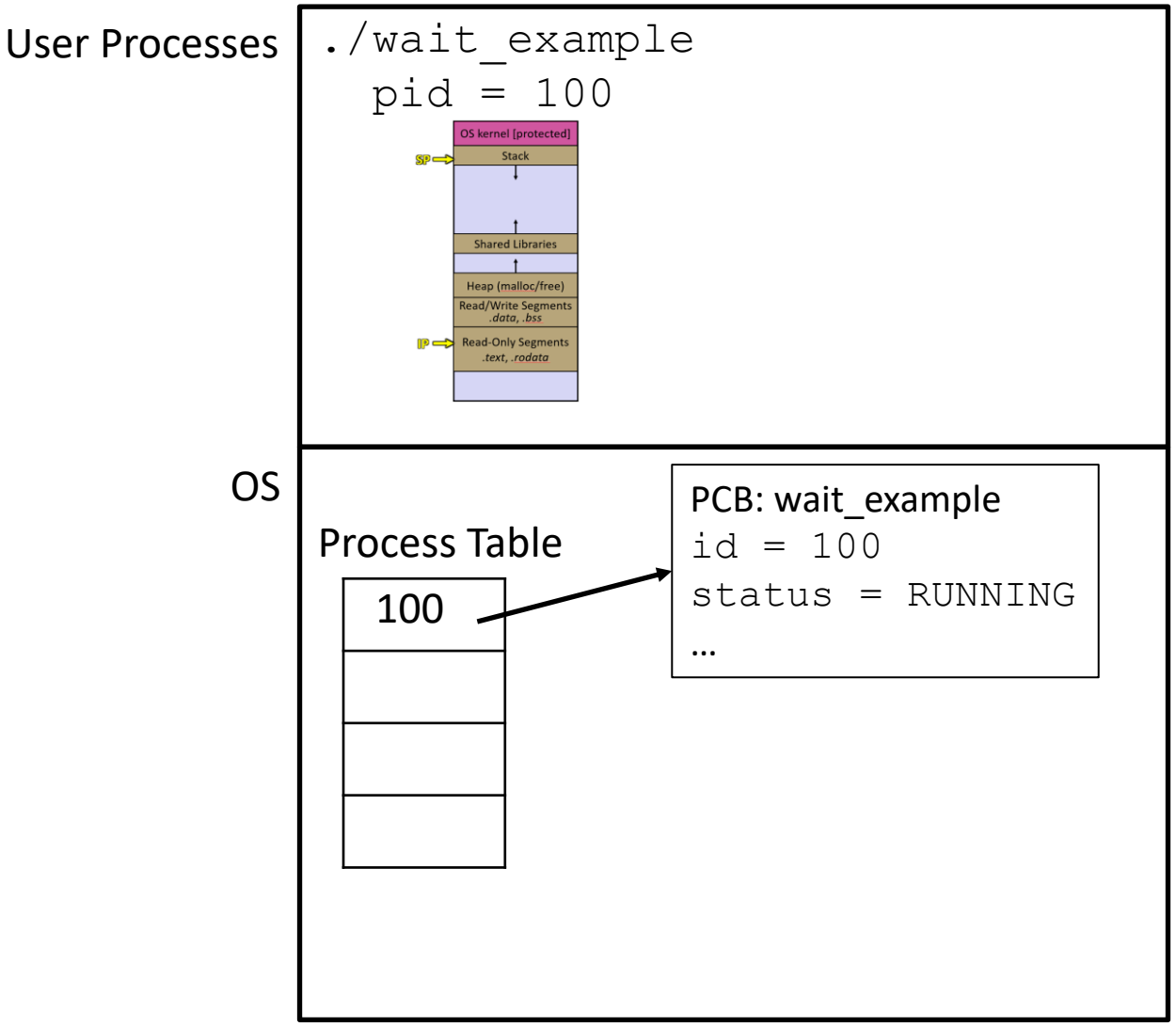
# Diagram: wait_example.c



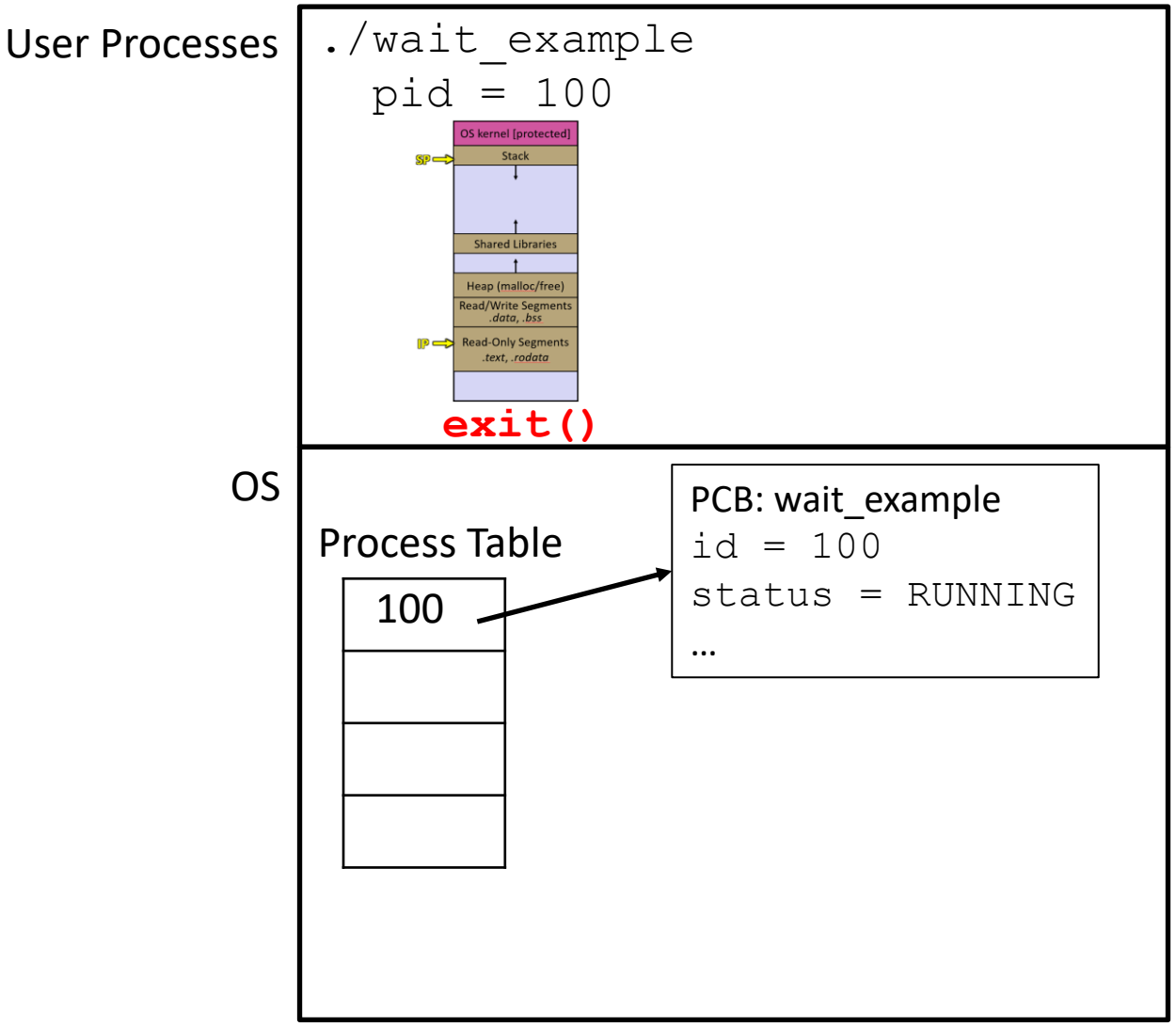
# Diagram: wait_example.c



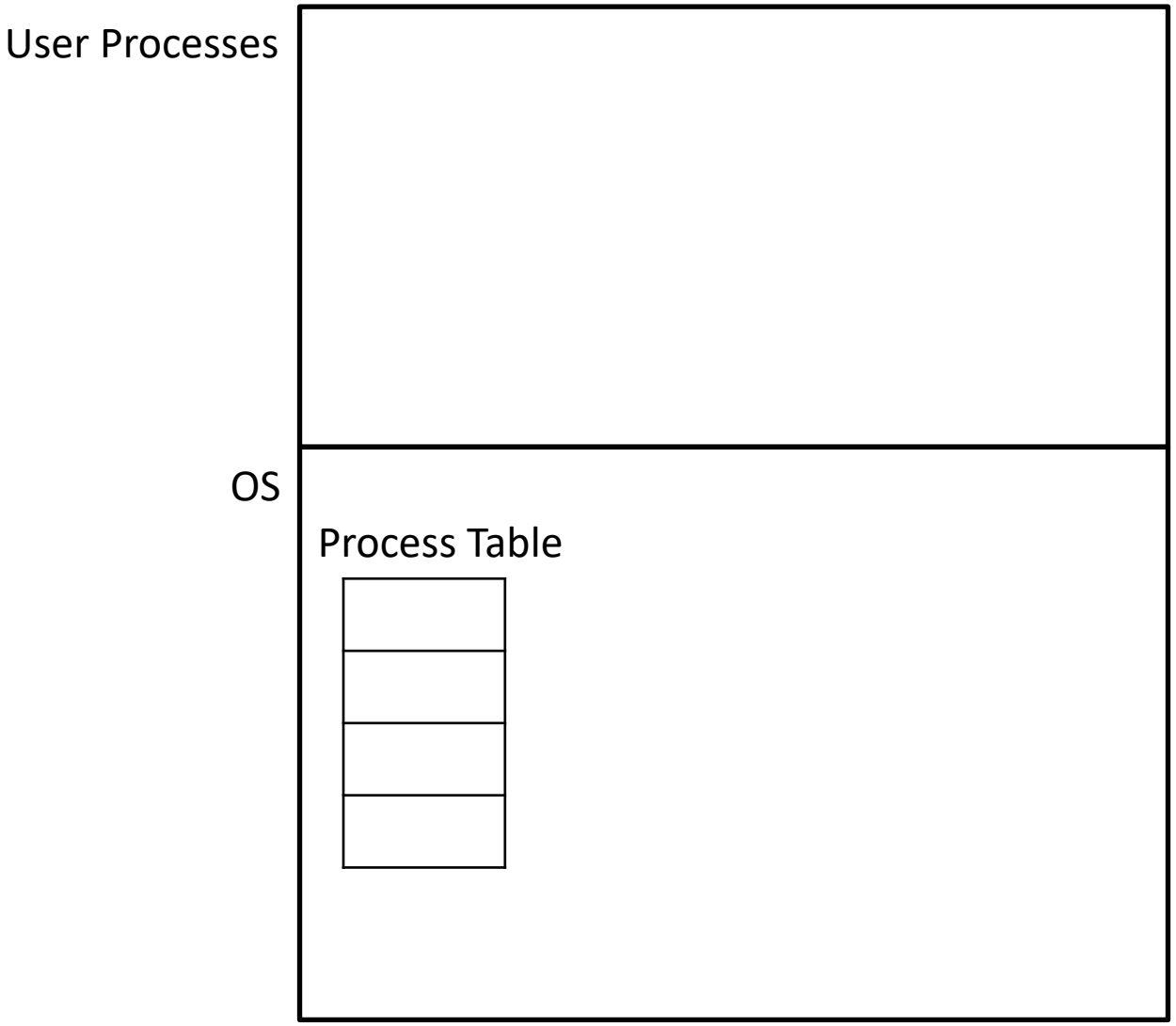
# Diagram: wait_example.c



# Diagram: wait_example.c



# Diagram: wait_example.c




`./wait_example`
Is reaped by its parent. In our example, that is the terminal shell

# wait() status

- `status` output from `wait()` can be passed to a macro to see what changed
- `WIFEXITED()` true iff the child exited nomrally
- `WIFSIGNALED()` true iff the child was signaled to exit
- `WIFSTOPPED()` true iff the child stopped
- `WIFCONTINUED()` true iff child continued

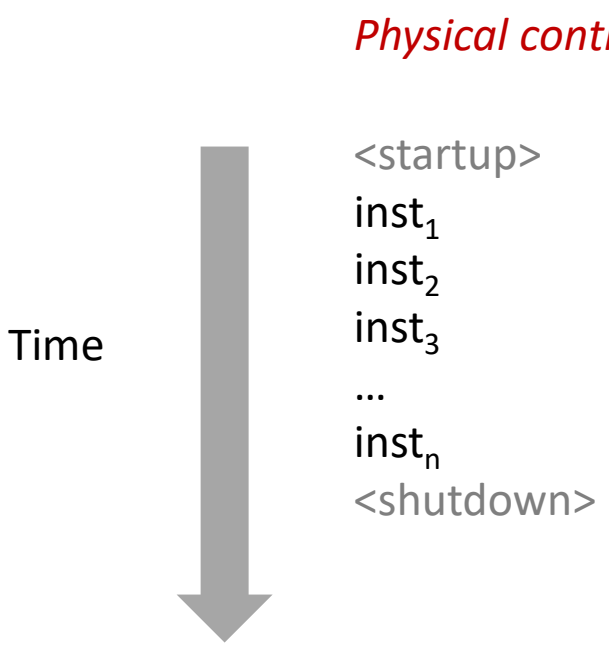
- Demo: see example in `exit_status.c`

# Lecture Outline

- Processes & Fork refresher
- exec
- Wait & process states
- **Hardware interrupts**
- Software signals
- Process states updated

# Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)



**Discuss then raise hands**

The bge instruction is being executed for the first time, which instruction is executed next?

- **A. bge**
- **B. add**
- **C. sub**
- **D. j**
- **E. I'm not sure**

```
        li    t0, 5   # load immediate 5 into t0
        li    t1, 2   # load immediate 2 into t1
        li    t2, 0   # load immediate 0 into t2

.LOOP
        add t2, t2, 1      # t2 = t2 + 1
        sub t0, t0, t1     # t0 = t0 - t1
        bge t0, x0, .LOOP # GOTO .loop if t0 > 0

.END
        j .END             # GOTO .END
                           # (infinite loop)
```

# Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for “exceptional control flow”

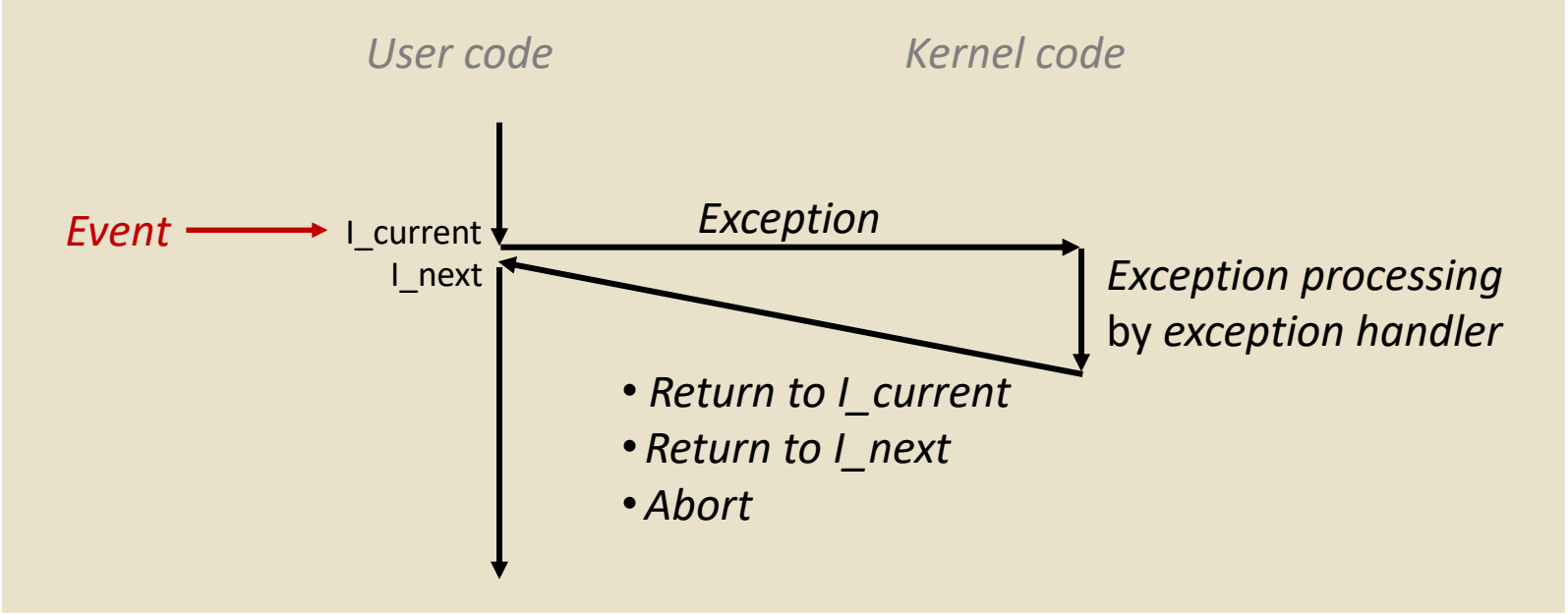
# Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Hardware Interrupts**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software

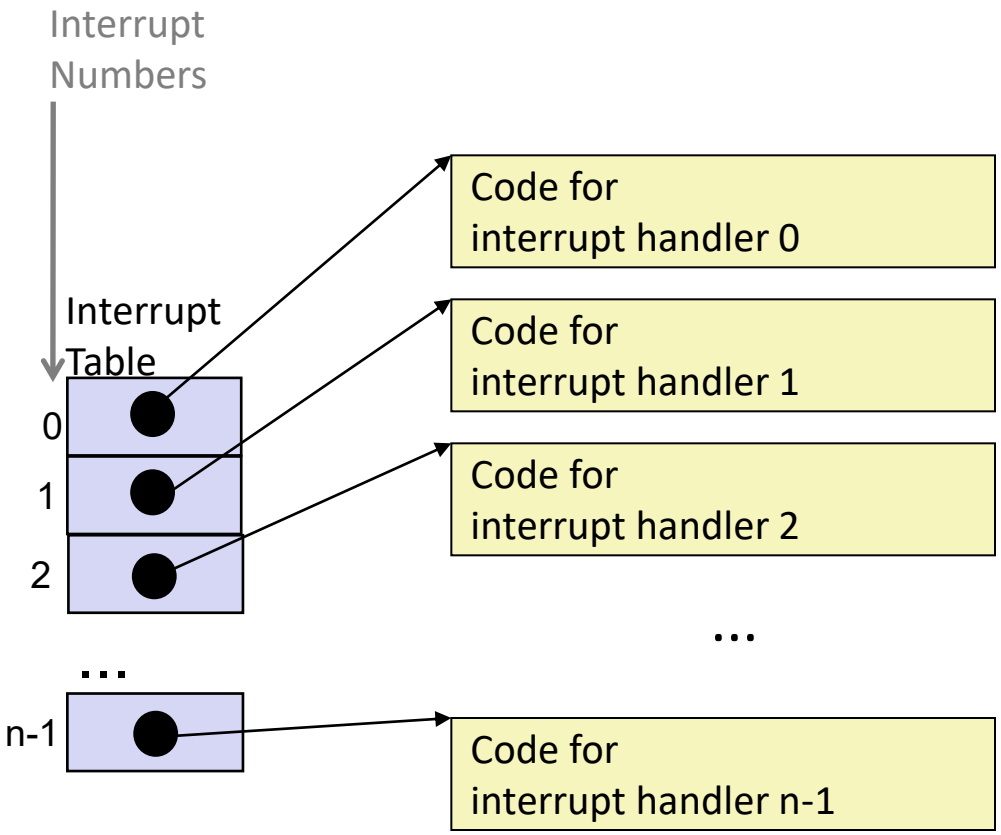
what we will be looking at today

# Interrupts

- An *Interrupt* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



# Interrupt Tables




- Each type of event has a unique number k
- k = index into table (a.k.a. interrupt vector)
- Handler k is called each time interrupt k occurs

# Asynchronous Interrupts

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

# Synchronous Interrupts

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but theoretically recoverable
    - Examples: page faults (recoverable), protection faults (recoverable sometimes), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

FUN FACT: the terminology and definitions aren't fully agreed upon. Many people may use these interchangeably

# Lecture Outline

- Processes & Fork refresher
- exec
- Wait & process states
- Hardware interrupts
- **Software signals**
- Process states updated

# Signals

- A Process can be interrupted with various types of signals
  - This interruption can occur in the middle of most code
- Each signal type has a different meaning, number associated with it, and a way it is handled
- These are different from an interrupt, but similar idea
  - signals are “higher level” and apply to a process. The kernel / some process will deliver the signal.
  - Interrupts are lower level mechanisms that cause the hardware to poke the kernel and respond
  - Some interrupts lead to a signal being sent (CTRL + C on keyboard -> SIGINT)

# Signals

- A Process can be interrupted with various types of signals
  - This interruption can occur in the middle of most code
- Each signal type has a different meaning, number associated with it, and a way it is handled (disposition)

- Examples:
  - `SIGCHLD` → Default Disposition: ignore
  - `SIGINT`
  - `SIGKILL`
  - `SIGALRM` → Default Disposition: terminate the process
  - `SIGSEGV` → Default Disposition : terminate & core dump

# sigaction()

- You can change how a certain signal is handled
- ```
  int sigaction(int signum, struct sigaction* act,
                            struct sigaction* old);
  ```
- `int signum -> is the signal`
- Uses the **`struct sigaction`** type to specify which signal handler to run and other options for how the signal should be handled

- Returns previous handler & behavior for that signal through the **`old`** output parameter

- You can not change the disposition of **`SIG_KILL`** and **`SIG_STOP`**.
  ```

# struct sigaction

- Has 5 different fields to specify the behaviour of how a signal should be handled. For today, we only care about **sa_handler** and **sa_flags**

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t   sa_mask;
  int        sa_flags;
  void     (*sa_restorer)(void);
};
```

# struct sigaction

```
struct sigaction {
  void     (*sa_handler)(int);
  int        sa_flags;
  ...
};
```

- Set sa_handler equal to the signal handler we want to use
  - Set **`sa_handler`** to **`SIG_IGN`** to set disposition to IGNORE
  - Set **`sa_handler`** to **`SIG_DFL`** for default disposition

- In this class: set **`sa_flags`** to **`SA_RESTART`**
  - This makes it so that *certain* system calls are automatically restart/continue if they are interrupted by a signal. (wanna see the list? `man 7 signal` ☺)

# Signal handlers

- ```
  typedef void (*sighandler_t)(int);
  ```
- A function that takes in as parameter, the signal number that raised this handler. Return type is void
- Is **automatically** called when your process is interrupted by a signal
- Can manipulate global state
- If you change signal behavior within the handler, it will be undone when you return
- Signal handlers set by a process will be retained in any children that are created (think about why?)
  ```

# Demo ctrlc.c

- See `ctrlc.c`
  - Brief code demo to see how to use a signal handler
  - Blocks the ctrl + c signal: SIGINT
  - Note: will have to terminate the process with the `kill` command in the terminal, use `ps -u` to fine the process id

# alarm()

```
unsigned int alarm(unsigned int seconds);
```

- Delivers the **SIGALRM** signal to the calling process after the specified number of seconds
- Default **SIGALRM** disposition: ***terminate the process***
- How to cancel alarms?
  - I leave this as an exercise for you: try reading the man pages

**Discuss, then open notepad, vim, or vscode and actually try coding it out.**
**The expected program isn't very long.**

- Finish this program
- After 15 seconds, print a message and then exit
- Can't use the **`sleep()`** function, must use **`alarm()`**

```c
int main(int argc, char* argv[]) {

  alarm(15U);

  return EXIT_SUCCESS;
}
```

- Currently: program calls alarm then immediately exits

- Hint: take a look at ctrl.c demo! :)

# Demo no_sleep.c

- See `no_sleep.c`
  - "Sleeps" for 10 seconds without sleeping, using alarm
  - Brief code demo to see how to use a signal handler & alarm
  - Signal handler manipulates global state

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

- System call that sends a signal to a process (has a somewhat dramatic name).
- `pid_t pid`
  - Specifies the process to send the signal to
- `int sig`
  - The signal to forward!

- If for some reason **kill()** is not recognized and you #include everything you need: Put this at the top of your .c file (before #includes) to use `kill()`

```
#define _POSIX_C_SOURCE 1
```

# Non blocking wait w/ `waitpid()`

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

- Our first option! WNOHANG
- WNOHANG makes `waitpid` immediately return
  - Either: there is no update in a child's state or there was.
- You must check the return value of `waitpid`
- **WITH THIS OPTION WAITPID** Returns process ID of child who was waited for or `-1` on error ***or `0` if there are no updates in children processes***

# Demo impatient.c

- See `impatient.c`
  - Parent forks a child, checks if it finishes every second for 5 seconds, if child doesn't finish send SIGKILL

  - Using **`sleep()`** AND **`alarm()`** together can cause issues because on some systems, sleep uses alarm, go figure.

# SIGCHLD handler

- Whenever a child process updates, a **`SIGCHLD`** signal is received, and by default ignored.

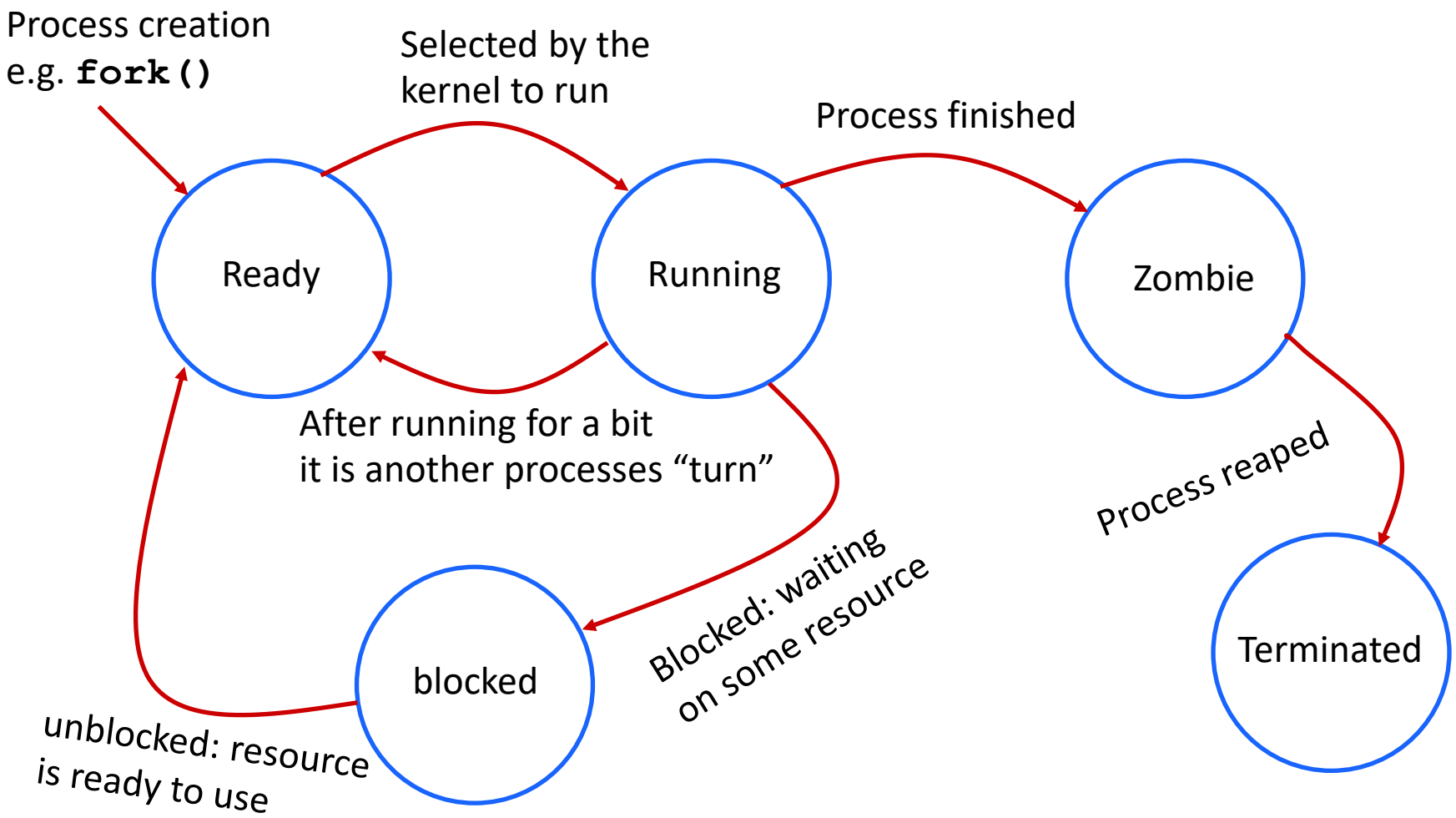
- You can write a signal handler for **`SIGCHLD`**, and use that to help handle children update statuses: allowing the parent process to do other things instead of calling **`wait()`** or **`waitpid()`**

- Relevant for proj1: **`penn-shell`**

# Lecture Outline

- Processes & Fork refresher
- exec
- Wait & process states
- Hardware interrupts
- Software signals
- **Process states updated**

# Process State Lifetime



# That's all for now!

- Next time:
  - More Processes and signals! ☺
- Hopefully you are doing well ☺
- Sign up for a penn-shell partner!