Processes (cont.): exec, wait, signal

Computer Systems Programming, Fall 2025

Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

Eric Zou, Joseph Dattilo, Aniket Ghorpade, Shriya Sane

Zihao Zhou, Eric Lee, Shruti Agarwal, Yemisi Jones

Connor Cummings, Shreya Mukunthan, Alexander Mehta, Raymond Feng

Bo Sun, Steven Chang, Rania Souissi, Rashi Agrawal

Sana Manesh

Poll Everywhere

pollev.com/cis4480

- How is penn-vector going?
  - I haven't started
  - I have read the spec
  - I've setup the container
  - I've started writing code
  - I've started writing code and I am pretty sure I understand what is going on
  - I'm done!

Administrivia

- First Assignment (HW00 penn-vector)
  - Released already!
  - “Due” This Friday 09/05
  - **Extended** to be due the same time as HW01 (Friday the 12th)
  - Mostly a C refresher

- Pre semester Survey
  - Anonymous
  - Short!
  - Due Friday the 5th

- **OH Started Last week, Levine 612! Check out the Course Calendar!**

Administrivia

- Second Assignment (HW01 penn-shredder)
  - Releases after today's lecture sometime tonight
  - Due **Friday Next week** 09/12
  - Intro to system calls, processes, etc.
  - Short Q&A and demo at end of class ☺

- First Check-in
  - Was Due Today! (Extended until the 9th @ 1:45PM…)
  - Don't forget to do them!!!

Administrivia

- Recitation starts this week on Thursday!
  - Look out for an announcement on Ed from the TAs with more information sometime soon.

Lecture Outline

- **Processes & Fork Revisited**
- `exec`
- `wait` & process states
- Hardware interrupts
- Software signals
- Process States updated
- penn-shredder demo

Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent address space
  - Each process is scheduled by the OS
    - ***There is no way to guarantee the order processes are executed, without*** using some functions we have not talked about (yet),
  - 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

```c
int main(){

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

*almost everything

Parent vs Child: Separate Address Space

```c
int main(){

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

pid = 1160

Stack

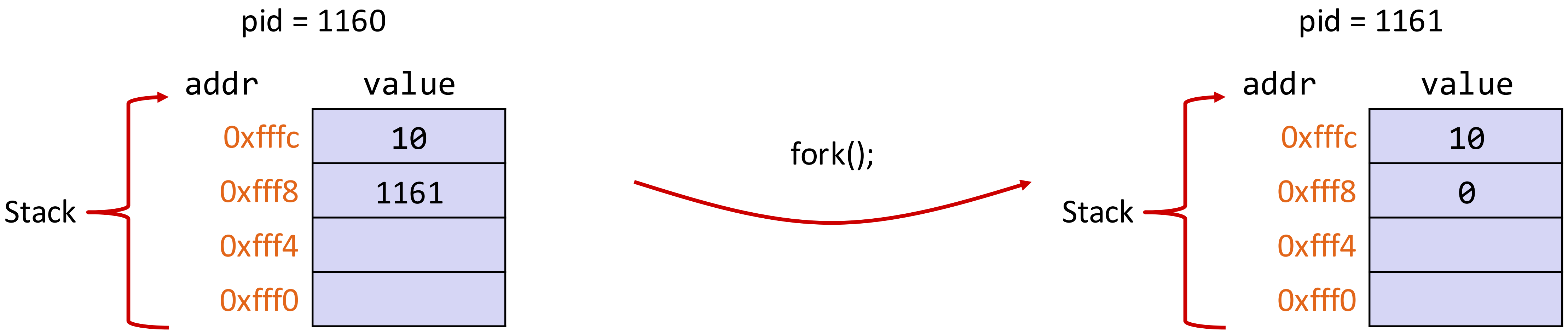
addr	value
0xfffc	10
0xfff8	
0xfff4	
0xfff0	

*almost everything

Parent vs Child: Separate Address Space

```
int main(){

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

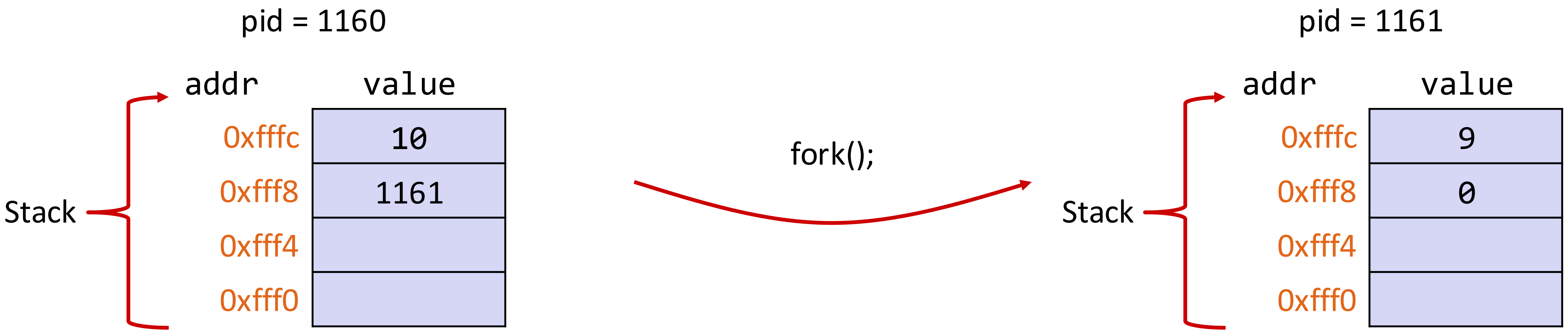


*almost everything

Parent vs Child: Separate Address Space

```
int main(){

        int x = 10;
        pid_t child = fork();
        if(child == 0) 10--;
  ───►  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

Poll Everywhere

pollev.com/cis4480

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 3; i++) {
    pid_t pid = fork();
    if(pid == 0) {
        printf(":)\n");
    }
  }

  return EXIT_SUCCESS;
}
```

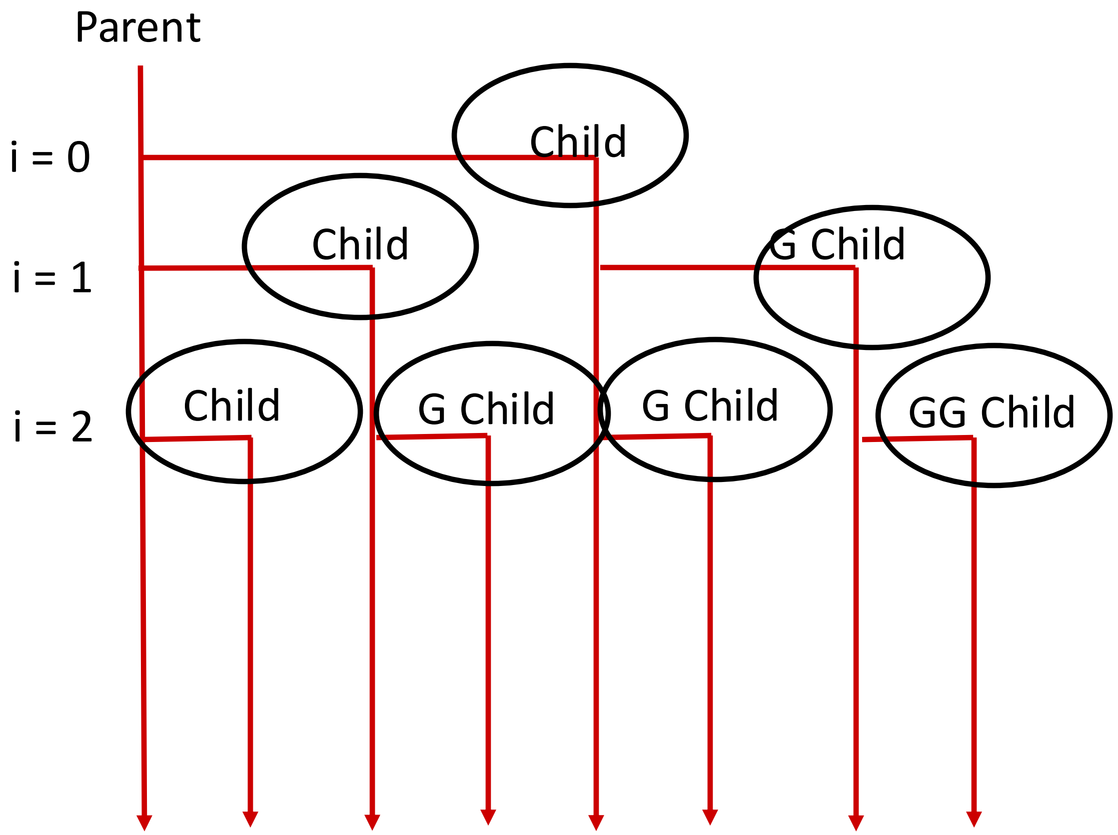
Poll Everywhere

pollev.com/cis4480

- How many times is ":)" printed?

```c
int main(int argc, char* argv[]) {
  for (int i = 0; i < 3; i++) {
    pid_t pid = fork();
    if(pid == 0) {
       printf(":)\n");
    }
  }

  return EXIT_SUCCESS;
}
```



When will pid be equal to 0?
Each time a new process is created!

Lecture Outline

- Processes & Fork Refresher
- **exec**
- wait & process states
- Hardware interrupts
- Software signals
- Process States updated
- penn-shredder demo

exec*()

- A family of functions that load in a ***new program*** for execution.
  - Replaces the currently running program while using the ***same process***.

- Things that are newly initialized include
  - Stack
  - Heap
  - Data segments (i.e. globals)
  - Registers (Stack Pointer, Program Counter, Argument-Registers…)
  - Text Segment (This one should make sense!)
  - And more….

execve()

```
int execve(const char *file,
           char* const argv[],
           char* const envp[]);
```

executes the program referred to by **file**.

- `const char *file`
  - The file(path) of the executable to well, execute
- `char* const argv[]` (An array of char *s)
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL as the last entry of the array
  - `int main(int argc, char *argv[])`

execve()

```
int execve(const char *file,
           char* const argv[],
           char* const envp[]);
```

- `char* const envp[]`
  - list of environment vars that become the environment of the exec'd program.
  - Not important for this course, learn more by doing `man environ`
  - Use this: `char* const envp[] = {NULL};`

- Return Value
  - If successful, there is no return value!
  - On failure, execve returns -1

Aside: The Exit Status

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

- Initiates the 'tear-down' of a process. "Graceful" exit.
- Generally called by `main()` when control falls off main or when main returns
- `int status`
  - EXIT_SUCCESS or EXIT_FAILURE
  - The exit status of a child is accessible by its parent process via `wait()` or `waitpid()`…

Typically used as follows:

```
execve(…);
exit(…); //shouldn't have reached this…
```

don't confuse exit() with _exit()

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/cis4480

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

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

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

   printf(":) \n");

  return EXIT_SUCCESS;
}
```

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

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

   printf(":) \n");

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/cis4480

```c
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };
  // fork a process to exec clang
  pid_t clang_pid = fork();

  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
            "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv, envp);
    exit(EXIT_FAILURE);
  }

  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    char* hello_argv[] = {"./hello", NULL};
    execve(hello_argv[0], hello_argv, envp);
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

This code compiles, but doesn't do what we want. The program attempts to compile some code and then run it.

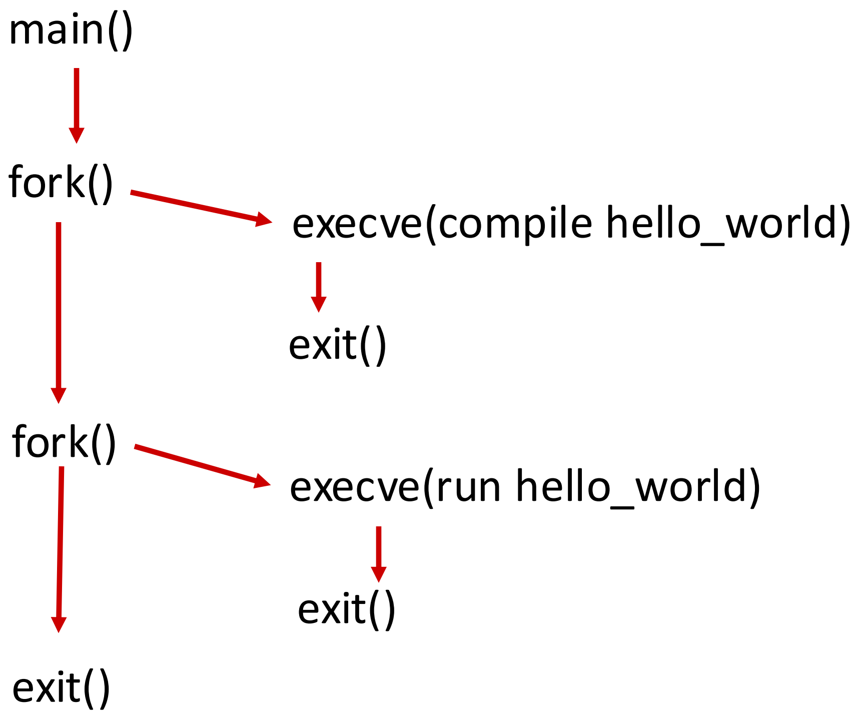
Why is this broken?

- Clang is a C compiler
- Assume exec'ing the compiler works (hello_world.c compiles)
- Assume we gave the correct args to exec in both cases

broken_autograder.c

Poll Everywhere

pollev.com/cis4480



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

- Clang is a C compiler
- Assume it compiles
- Assume I gave the correct args to exec

Lecture Outline

- Processes & Fork Refresher
- exec
- **wait & process states**
- Hardware interrupts
- Software signals
- Process States updated
- penn-shredder demo

From the previous example:

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

  // fork a process to exec clang
  pid_t clang_pid = fork();
  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
                "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv, envp);
    exit(EXIT_FAILURE);
  }

  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    char* hello_argv[] = {"./hello", NULL};
    execve(hello_argv[0], hello_argv, envp);
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

broken_autograder.c

What do we need to happen for this to work correctly?

Waiting for Processes to Finish

- `pid_t wait(int *wstatus);`
  - int *wstatus
    - Output parameter containing the status of the terminated child.
  - Returns process ID of child who changed states for or -1 on error (e.g. no children to wait for)

Calling process waits for any child process to change status to terminated.

```
int main(){
  pid_t pid = fork();
  if(pid == 0){
   //....
   return EXIT_SUCCESS;
  }
  int status;
  pid_t wpid = wait(&status);
  // do something with status
}
```

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 broken_autograder.c

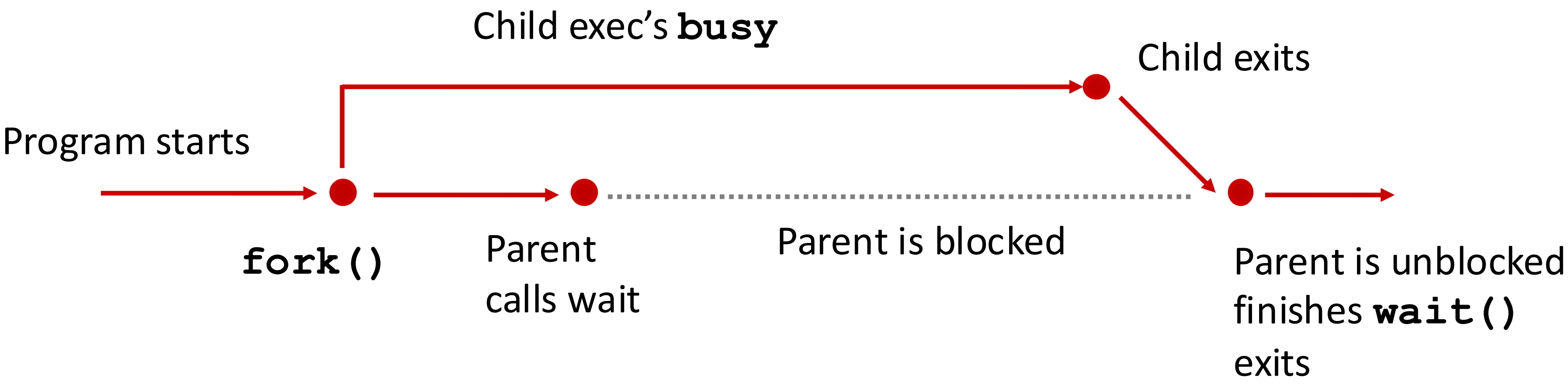
```
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };
  // fork a process to exec clang
  pid_t clang_pid = fork();
  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
                "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv, envp);
    exit(EXIT_FAILURE);
  }
  wait(NULL); // should error check, not enough slide space :(
  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    char* hello_argv[] = {"./hello", NULL};
    execve(hello_argv[0], hello_argv, envp);
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

autograder.c

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:



discuss

- Can a child finish before parent calls wait?

What if the child finishes first?

- In the timeline I drew, the parent called wait before the child executed.
  - In our example, it is extremely likely this happens if the child is calling **`sleep 10`**, **`./busy`**, **`etc.`**
  - 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 OS maintains a table of all processes that aren't "completely done"
- Each process in this table has a process control block (**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

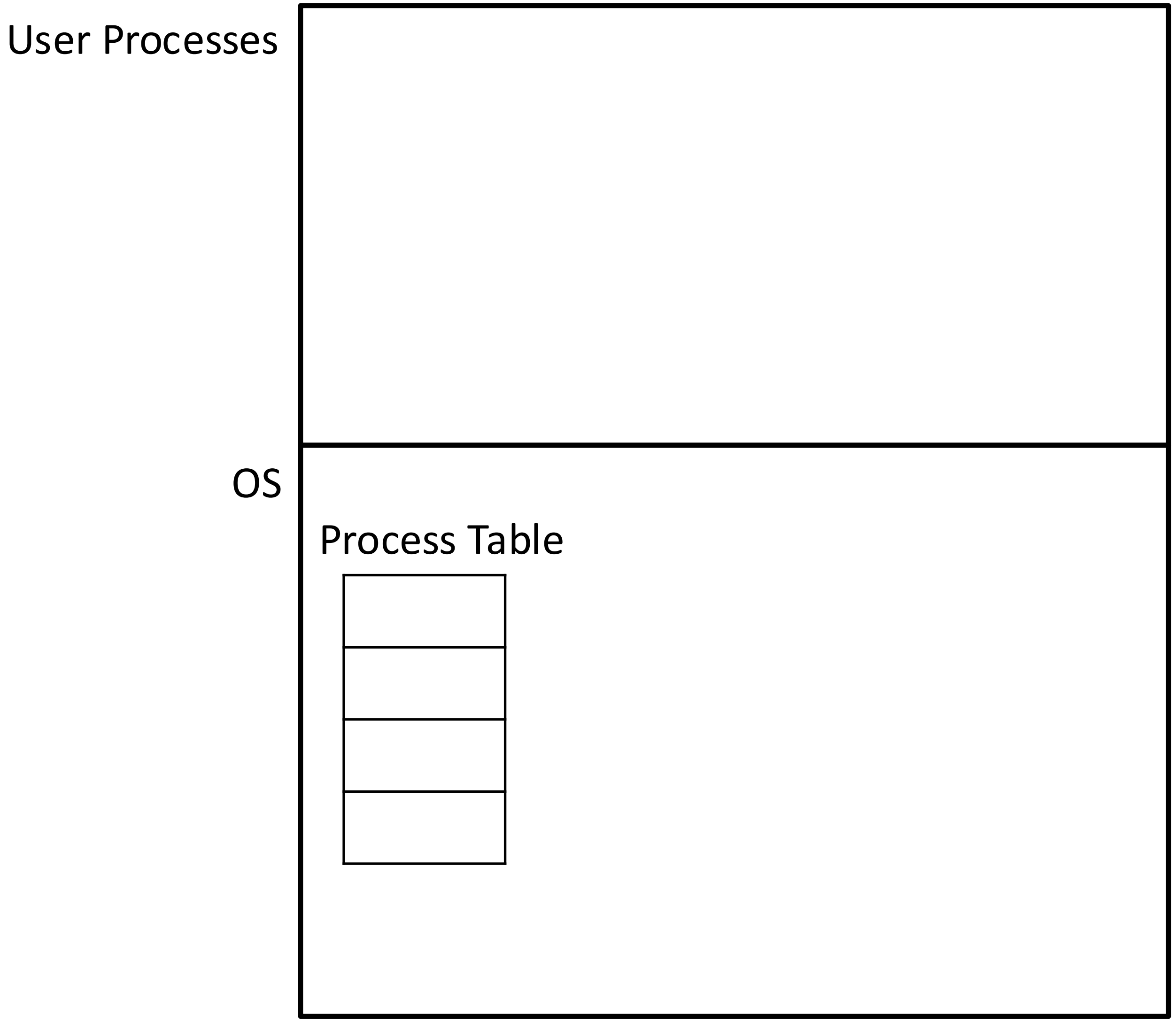


Diagram: wait_example.c

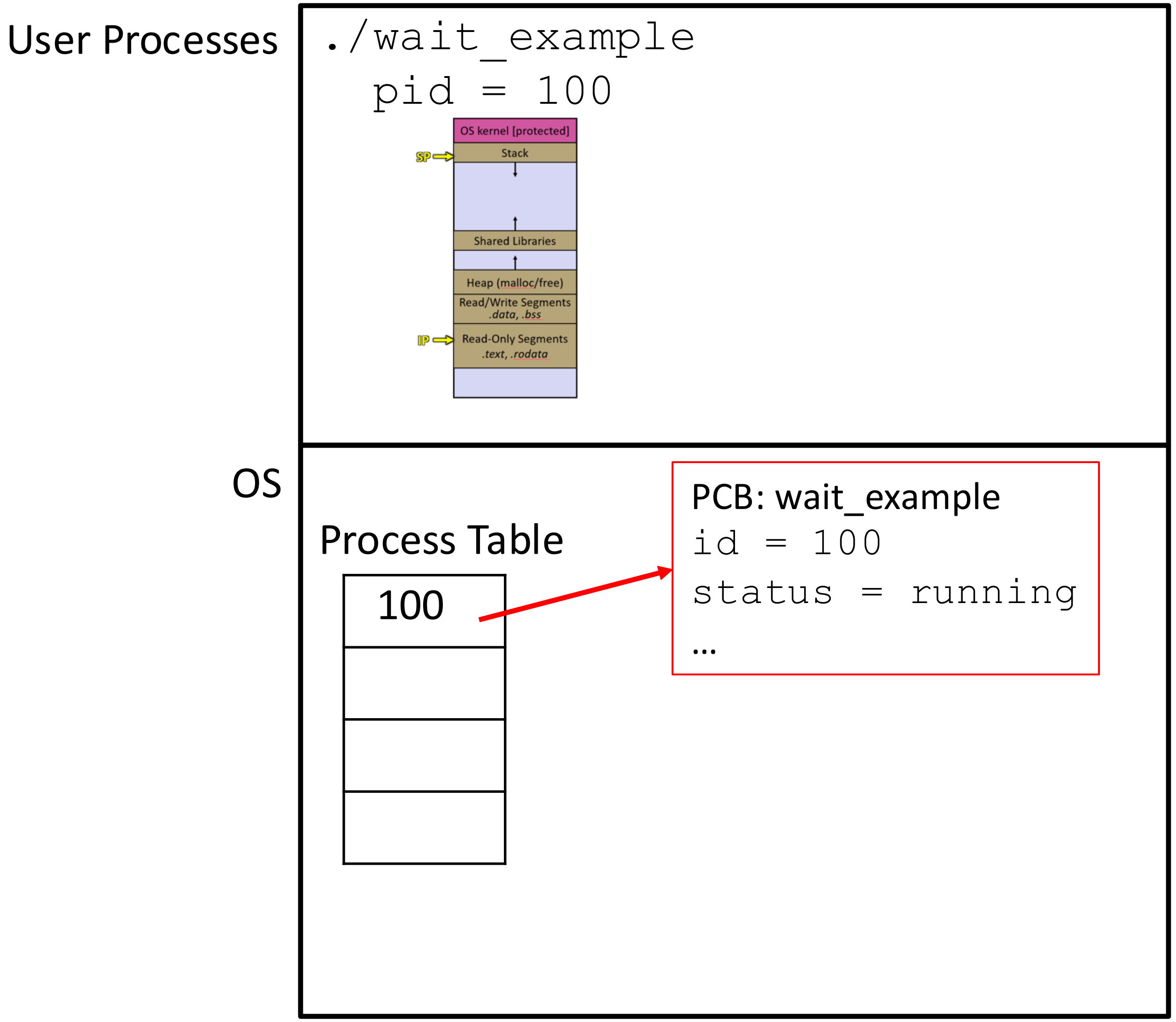


Diagram: wait_example.c

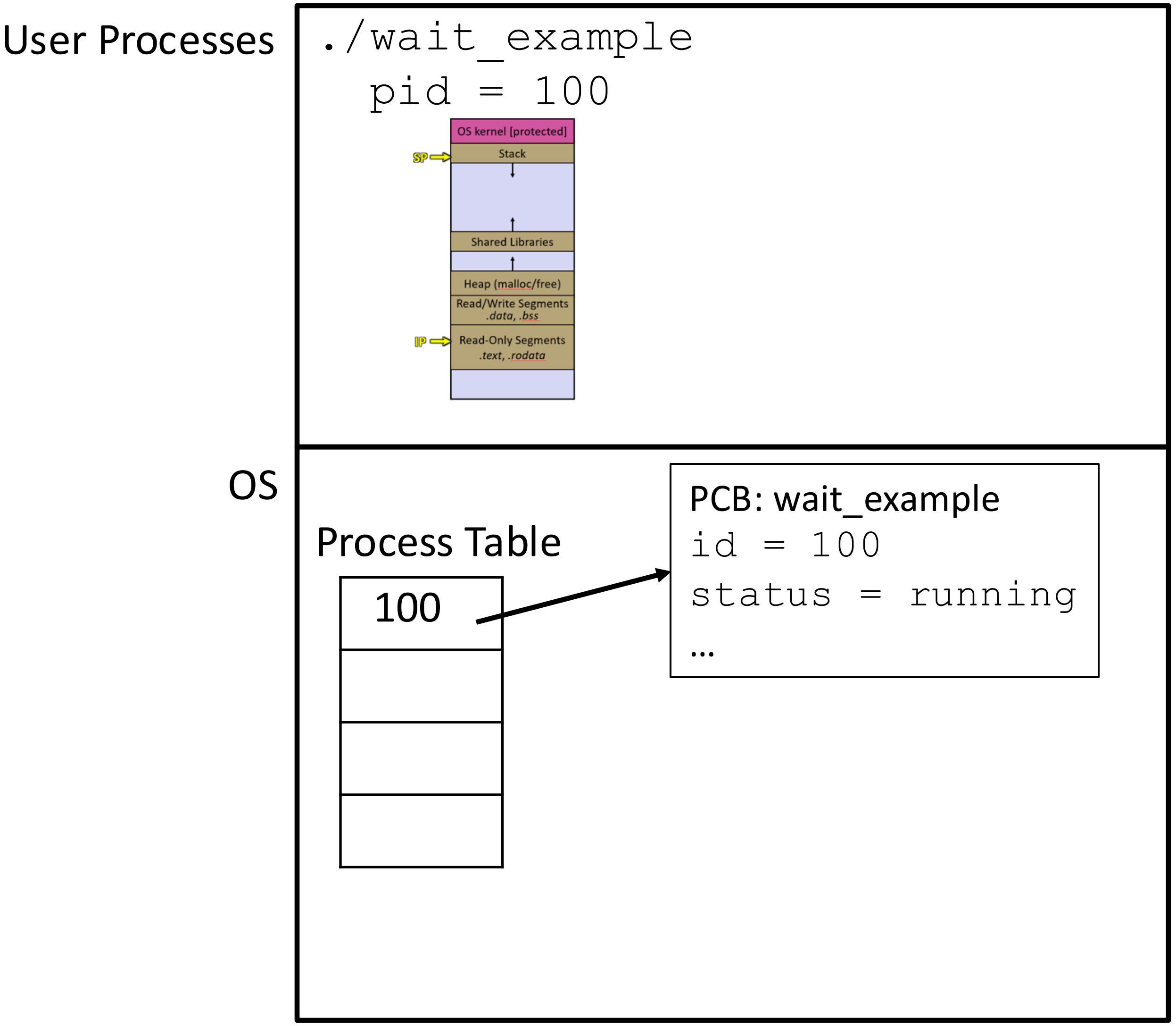
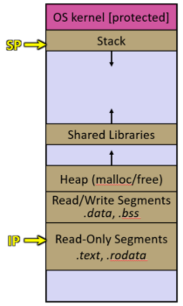


Diagram: wait_example.c

User Processes

./wait_example

pid = 100



fork()

OS

Process Table

100

PCB: wait_example

id = 100

status = running

…

Diagram: wait_example.c

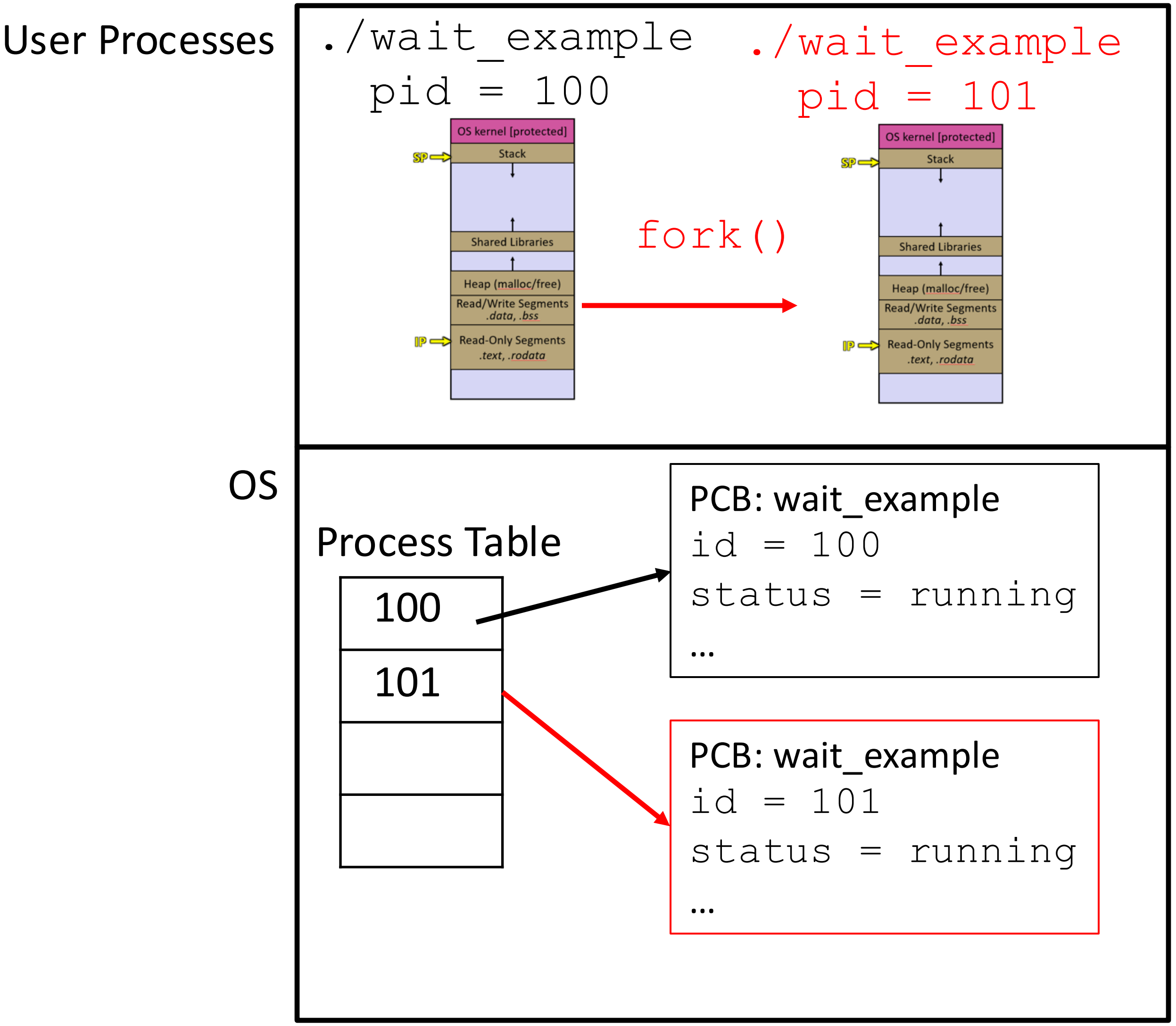


Diagram: wait_example.c

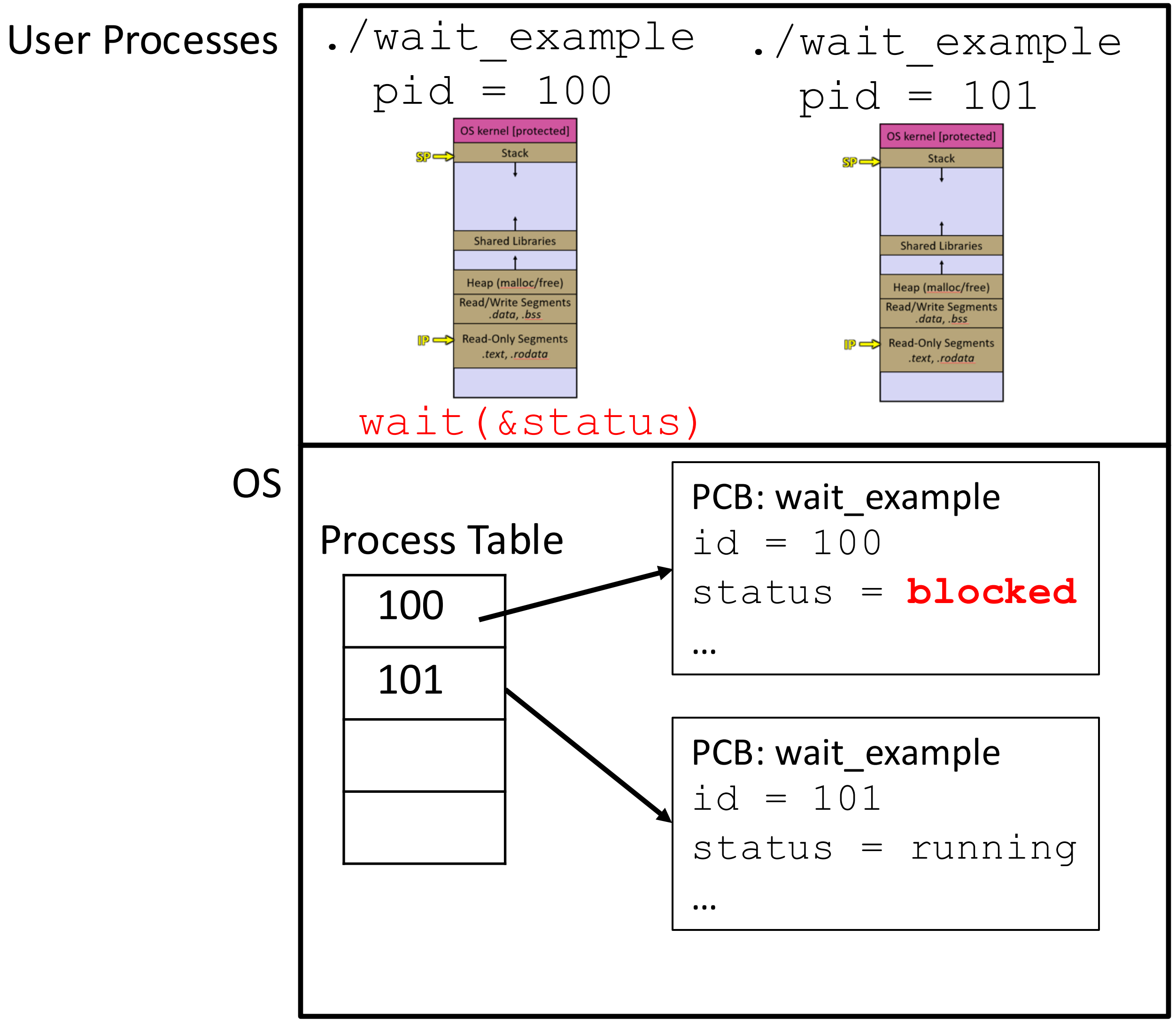


Diagram: wait_example.c

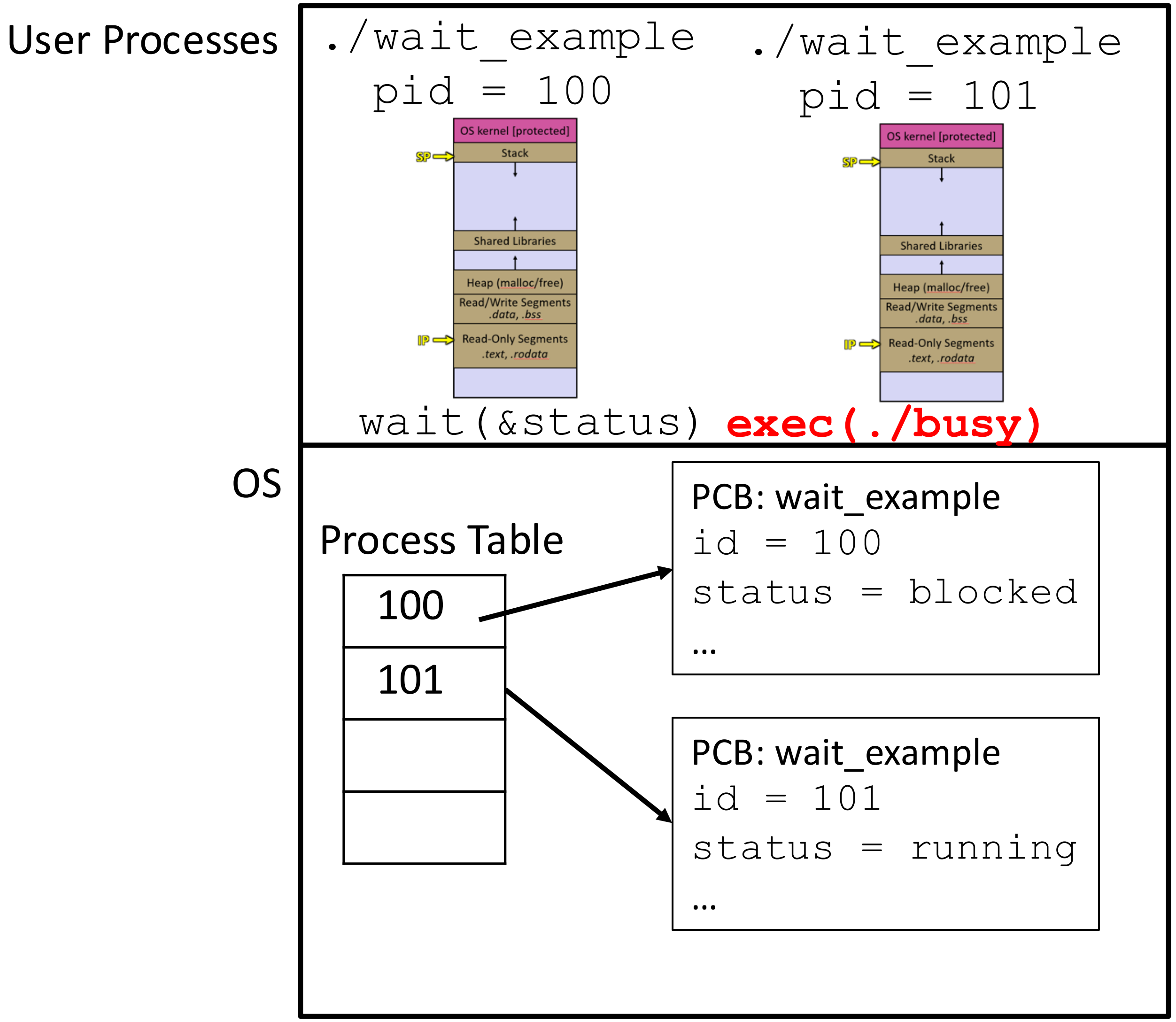


Diagram: wait_example.c

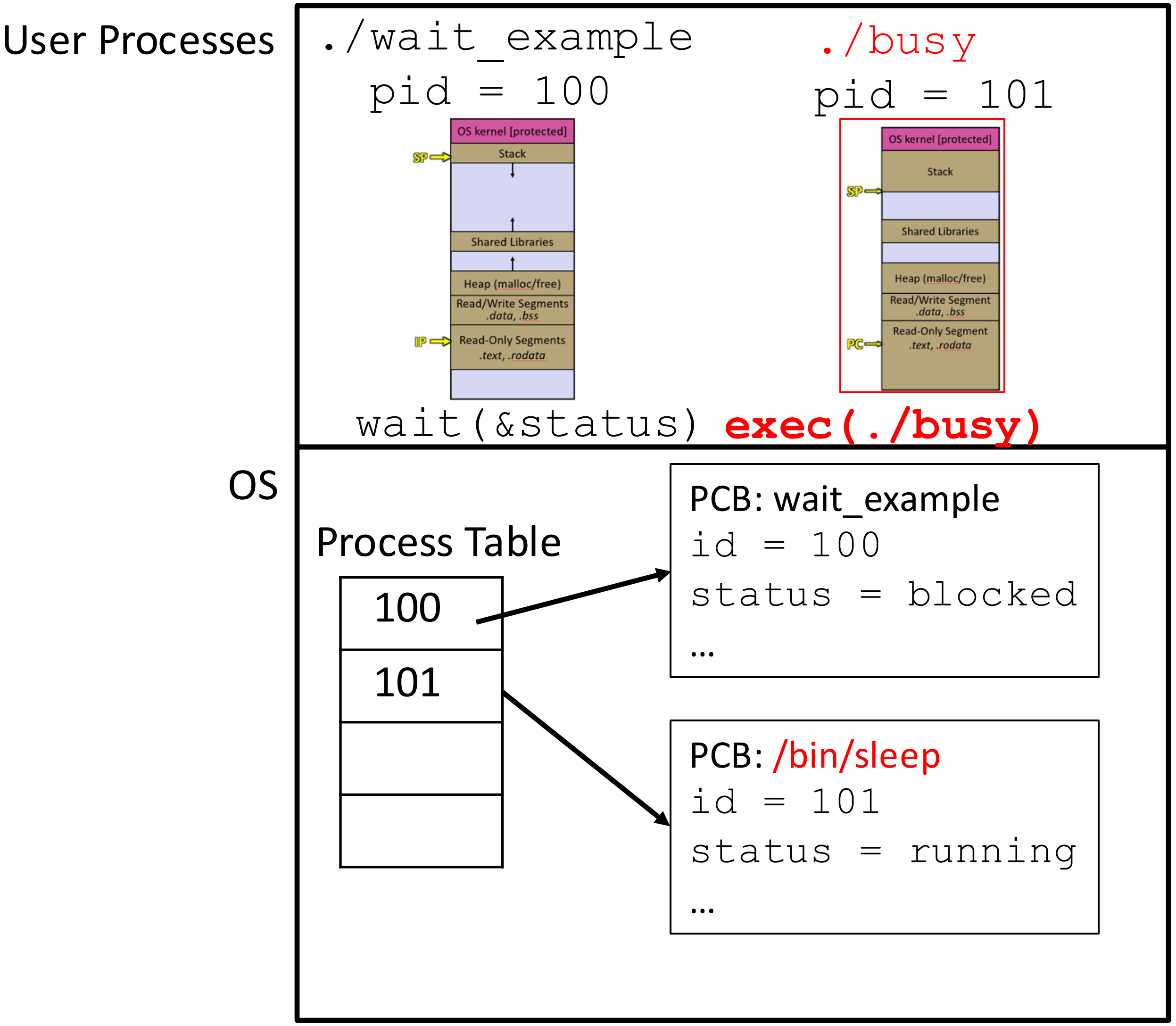


Diagram: wait_example.c

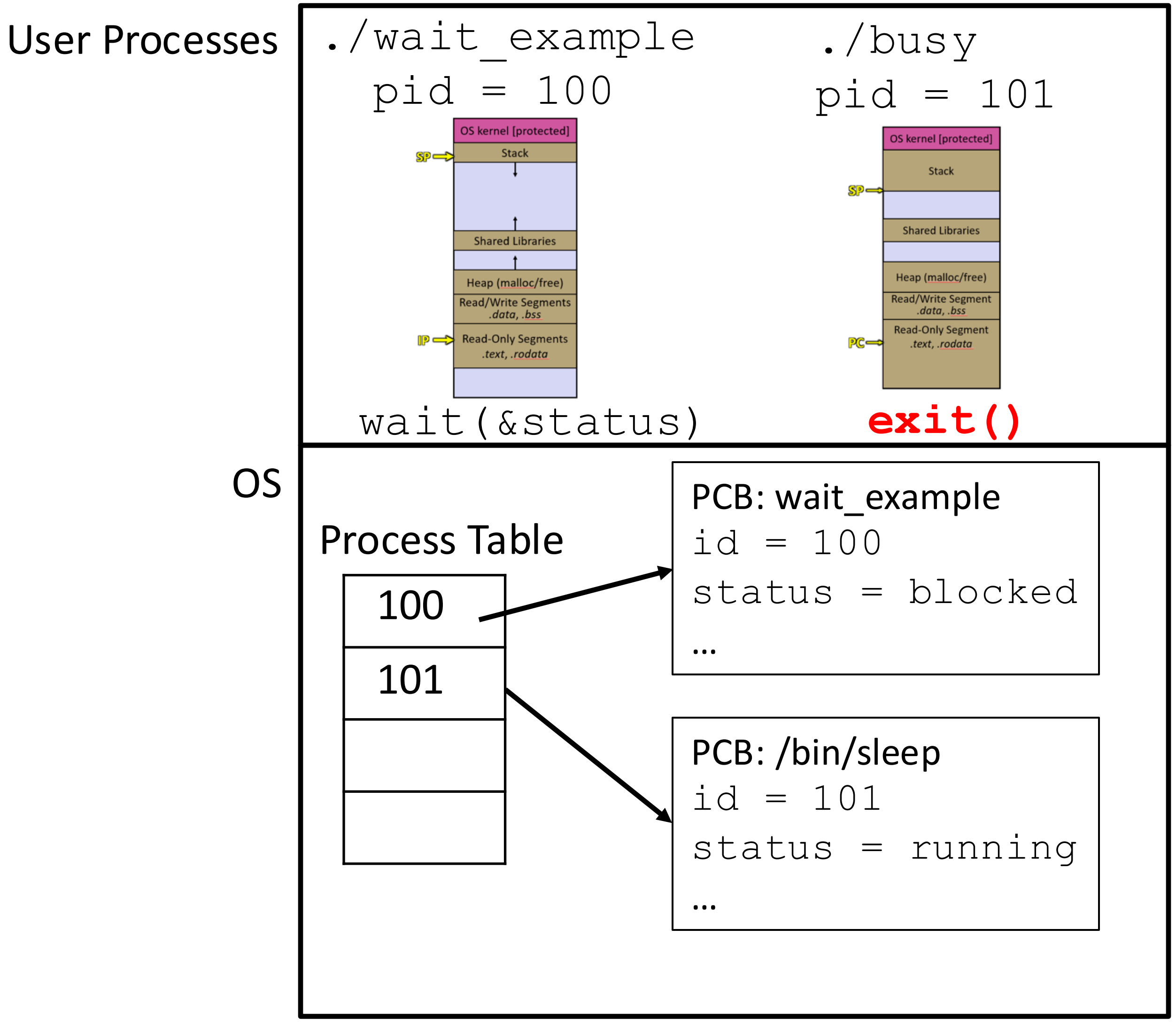
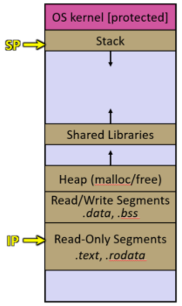


Diagram: wait_example.c

User Processes

```
./wait_example
  pid = 100
```




```
wait(&status)
```

OS

Process Table

100
101

PCB: wait_example
`id = 100`
`status = blocked`
…

PCB: /bin/sleep
`id = 101`
`status = ZOMBIE`
…

Diagram: wait_example.c

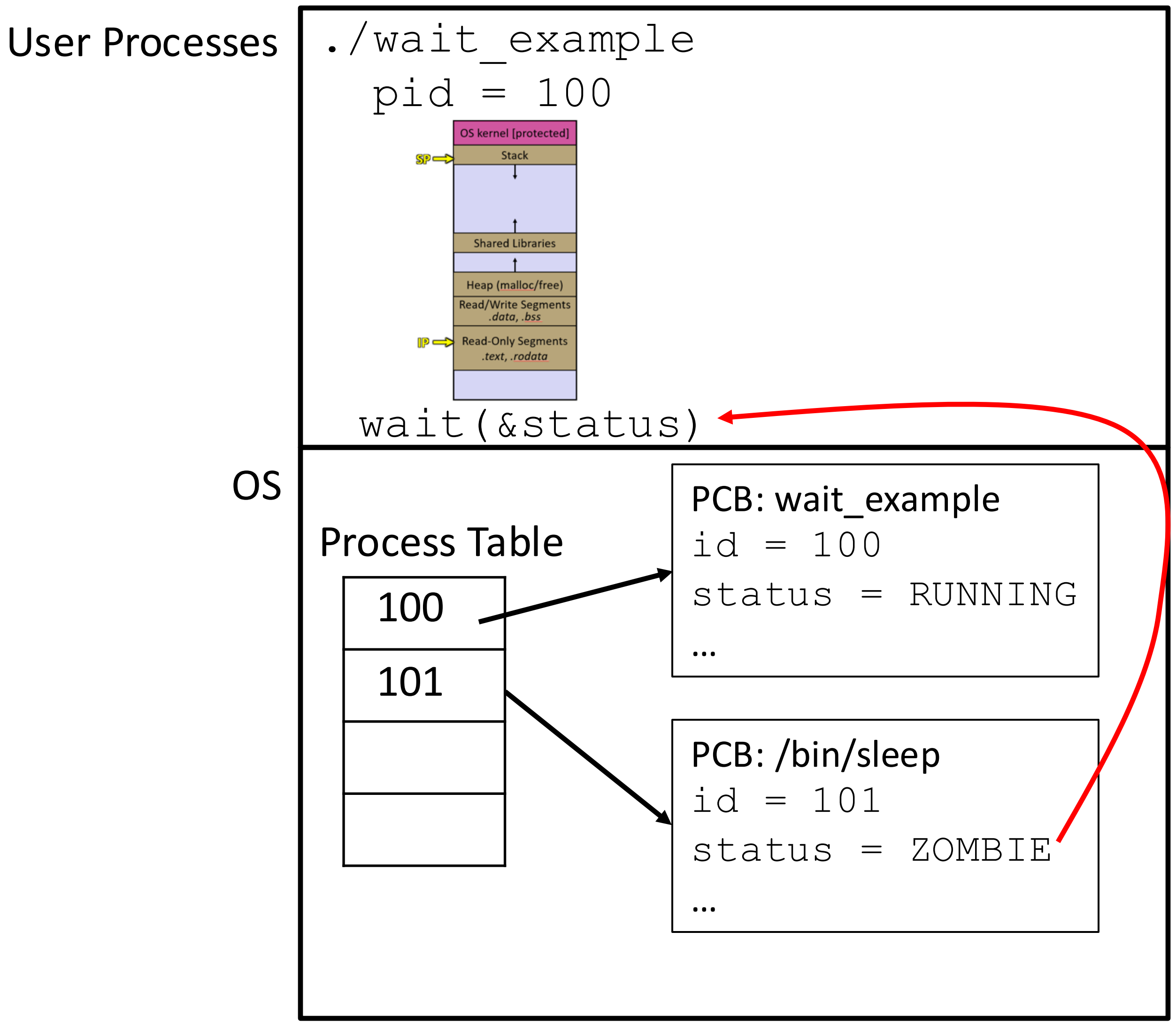


Diagram: wait_example.c

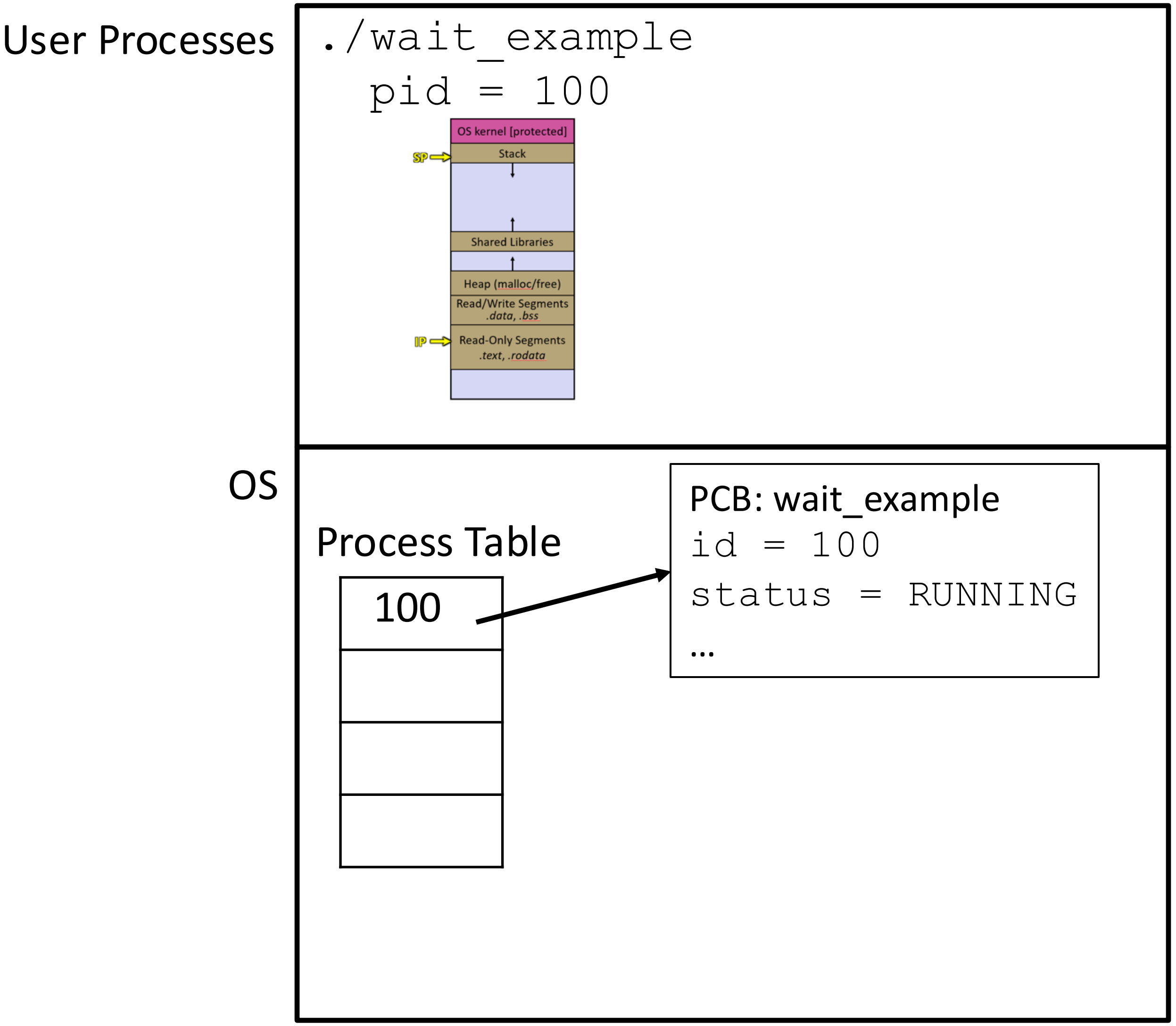


Diagram: wait_example.c

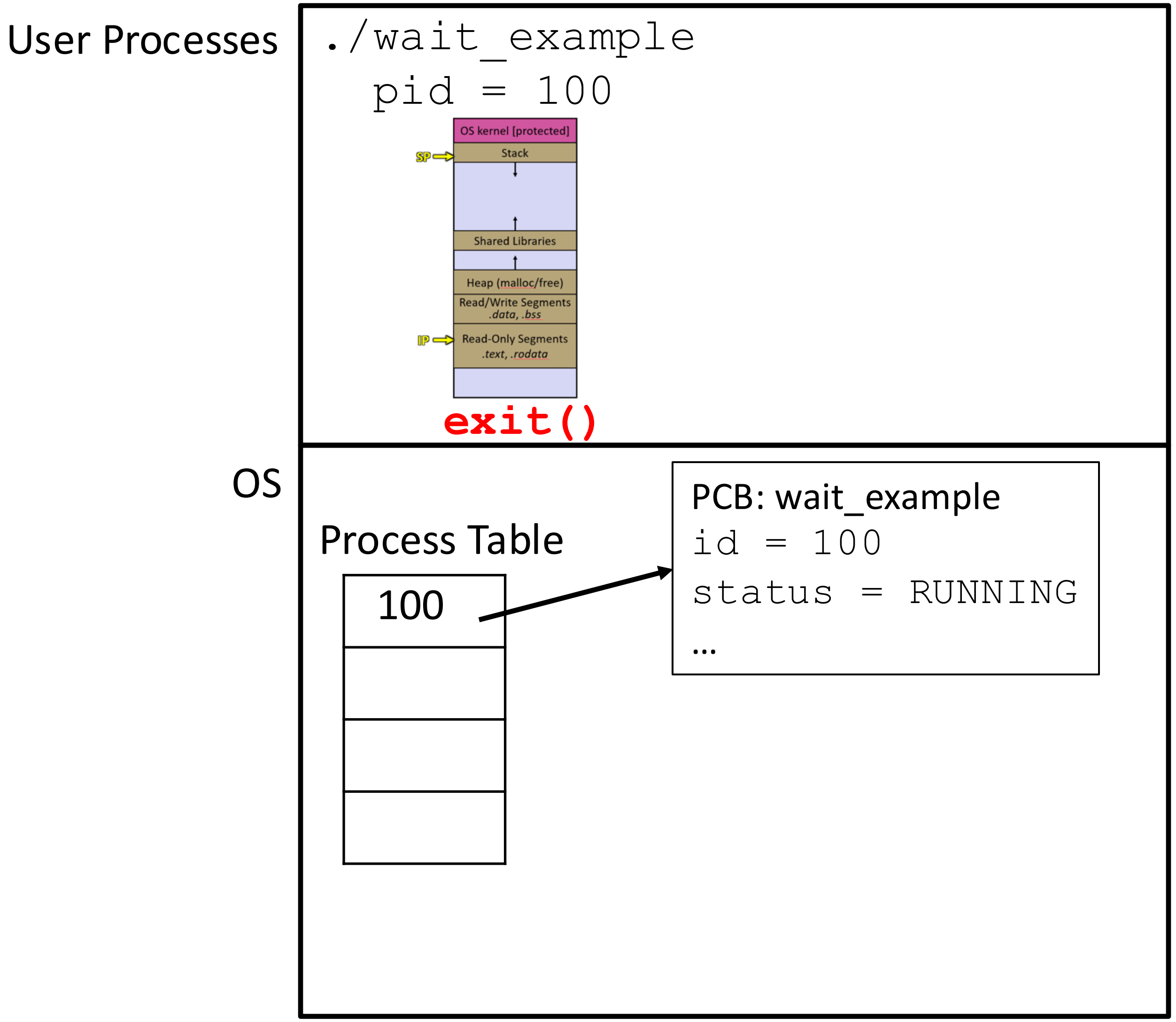
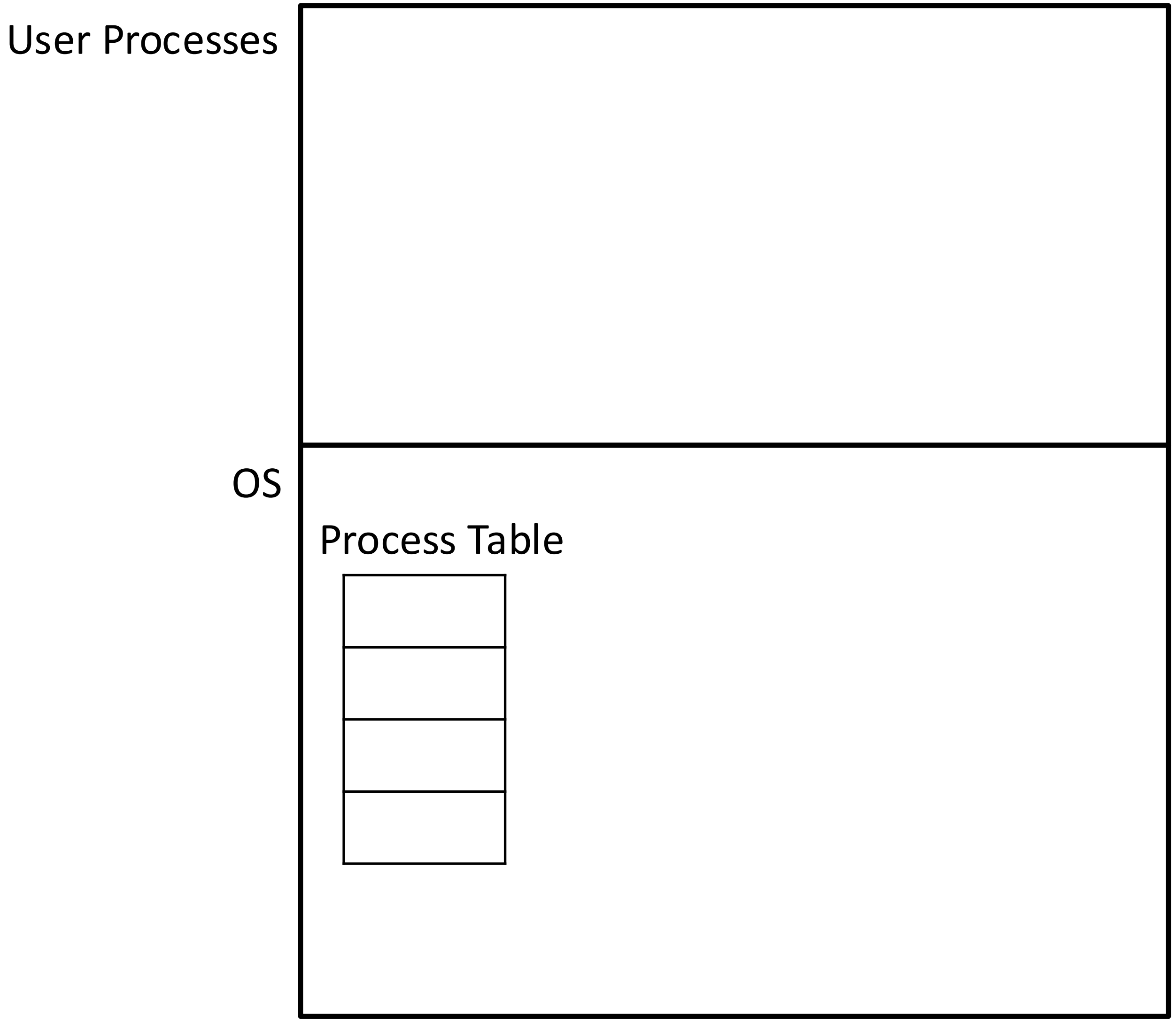


Diagram: wait_example.c



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

Demo: `state_example`

- See `state_example.c`
  - Brief code demo to see the various states of a process
    - Running
    - Zombie
    - Terminated
  - Makes use of `sleep()`, `wait()` and `exit()`!
  - Aside: `sleep()` takes in an integer number of seconds and blocks till those seconds have passed

But wait(), it gets better.

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

- **`pid_t pid`**
  - The pid of the child we are waiting for.
  - If pid is -1, then we wait for *any child process!*
- **`int *wstatus`**
  - Same as `wait()`
- **`int options`**
  - A set of bitwise-or'd flags to indicate behavior of waitpid!
  - Setting options to 0 makes waitpid return when a child has terminated.
- Returns process ID of child who was waited for or `-1` on error

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

Lecture Outline

- exec
- wait & process states
- **Hardware interrupts**
- Software signals
- Process States updated
- penn-shredder demo

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*)

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Poll Everywhere

pollev.com/cis4480

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

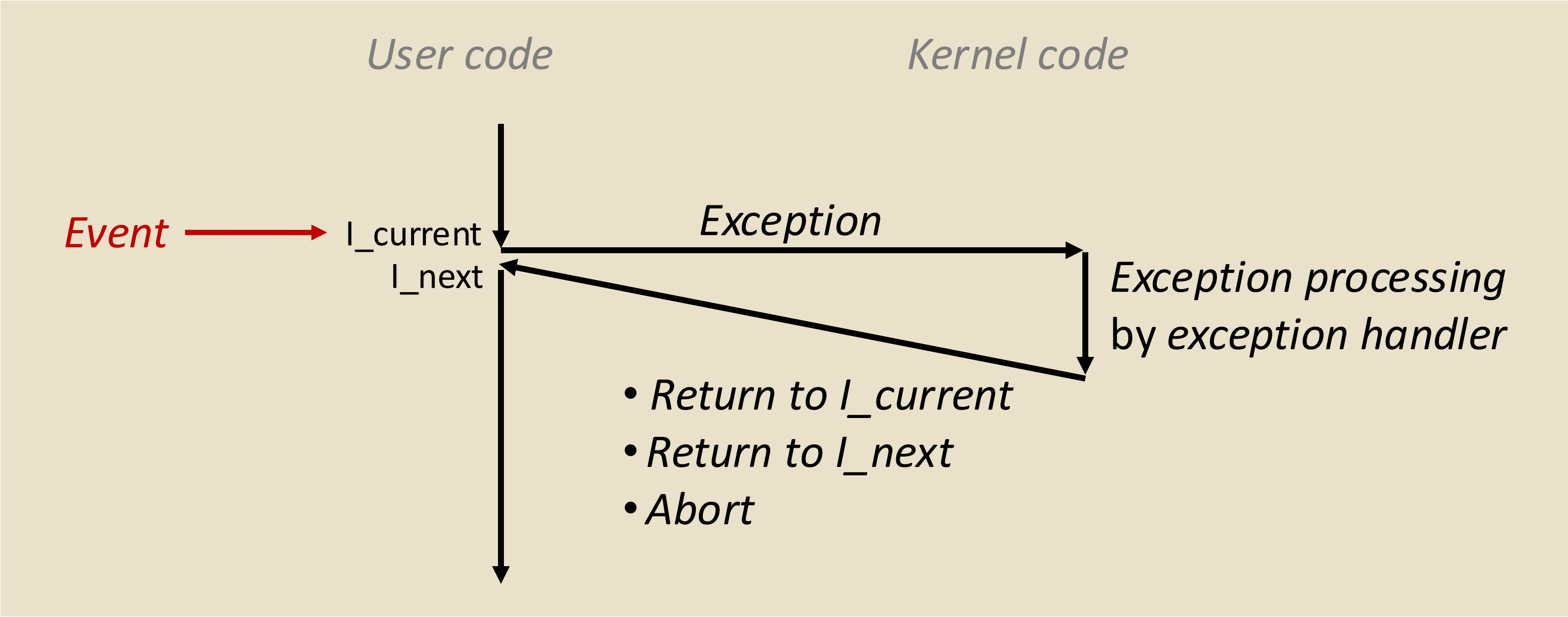
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Hardware Interrupts**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software

what we will be looking at today

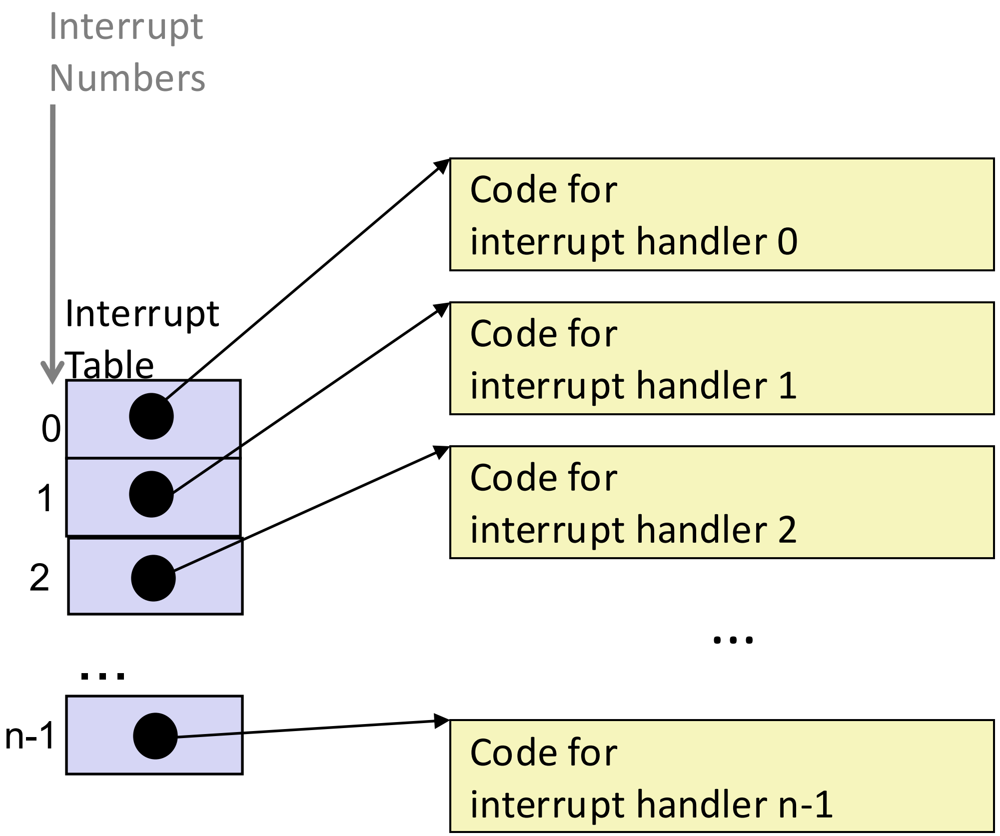
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software

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:

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

  - ***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

Lecture Outline

- Processes & Fork Refresher
- exec
- wait & process states
- Hardware interrupts
- **Software signals**
- Process States updated
- penn-shredder demo

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`
  - `SIGINT`
  - `SIGKILL`
  - `SIGALRM`
  - `SIGSEGV`

Default Disposition: ignore

Default Disposition: terminate the process

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

- HINT FOR OPTIONAL CHALLENGE: What is the default behavior of SIGALRM? Can you take advantage of the default behavior?

discuss

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

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

  alarm(15U);

  return EXIT_SUCCESS;
}
```

- Currently: program calls alarm then immediately exits

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

```c
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 penn-shredder.c file (before #includes) to use `kill()`

```c
#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

  - In penn-shredder **`waitpid()`** IS NOT ALLOWED so don't copy this. ☺
  - Plus, 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.

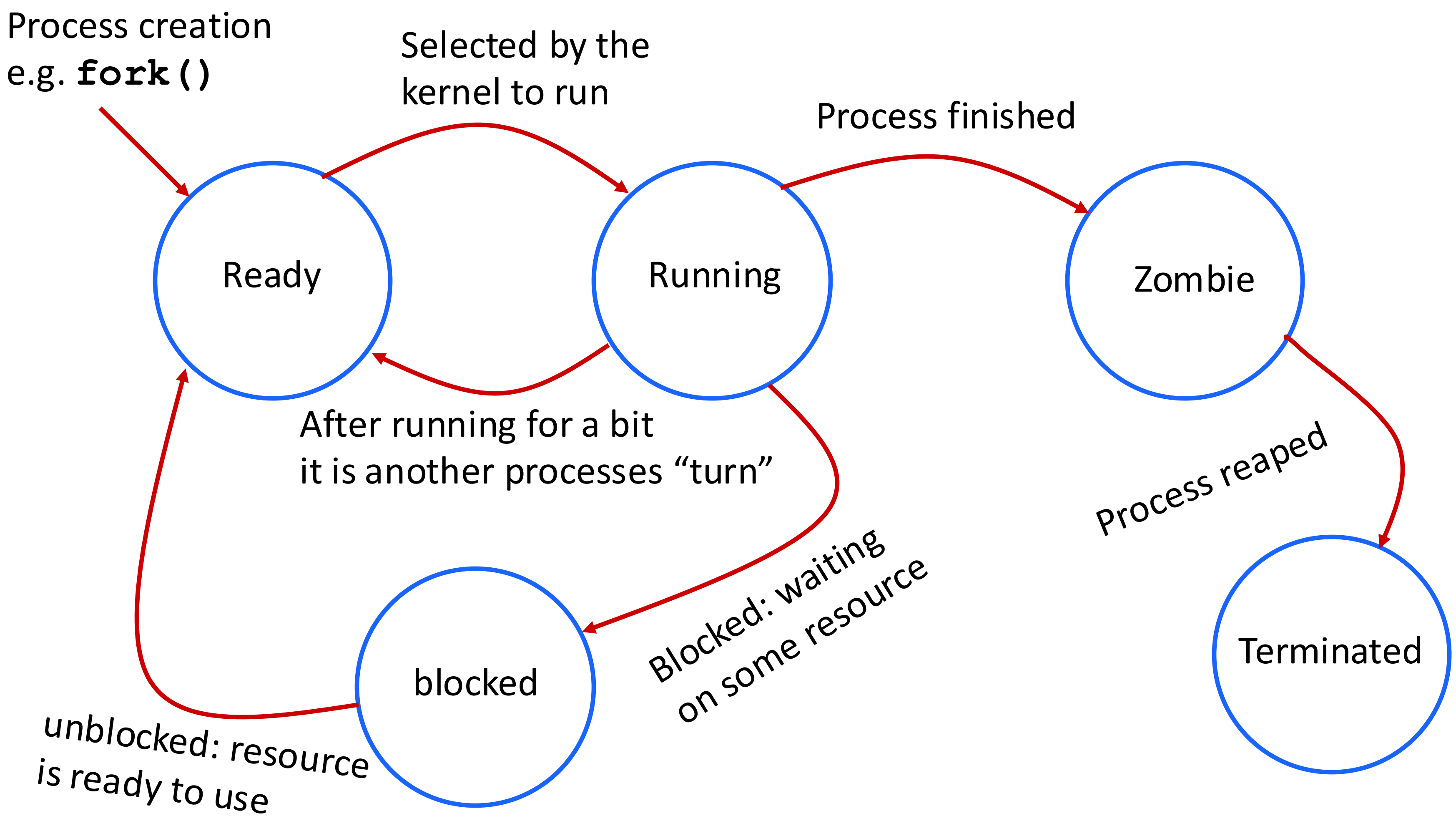
- You can write a signal handler for `SIGCHLD`, and use that to help handle children update statuses: allowing the parent process to do other things instead of calling `wait()` or `waitpid()`

- Relevant for proj2: `penn-shell`

Lecture Outline

- Processes & Fork Refresher
- exec
- wait & process states
- Hardware interrupts
- Software signals
- **Process States updated**
- penn-shredder demo

Process State Lifetime



Lecture Outline

- Processes & Fork Refresher
- exec
- wait & process states
- Hardware interrupts
- Software signals
- Process States updated
- **penn-shredder demo**