Processes (cont.): exec, wait, signal

Computer Systems Programming, Spring 2025

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/tqm

- 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” Friday 01/24
  - Extended to be due the same time as HW01 (Friday the 31st)
  - Mostly a C refresher

- Pre semester Survey
  - Anonymous
  - Short!
  - Due Wednesday the 28th

- Some OH later today 3:30 – 7pm (Levine 307)

Administrivia

- Second Assignment (HW01 penn-shredder)
  - Releases after today's lecture
  - Due Friday next week 01/31
  - Intro to system calls, processes, etc.
  - Short Q&A and demo at end of class ☺

- Github repo setup instructions
  - Posted after lecture today

- First Check-in
  - Releases tomorrow
  - Due before lecture on the 30th (please do before 28th)

Lecture Outline

Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent address space
  - 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

Poll Everywhere

pollev.com/tqm

```
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/tqm

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }

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

Lecture Outline

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

exec*()

- Loads in a new program for execution

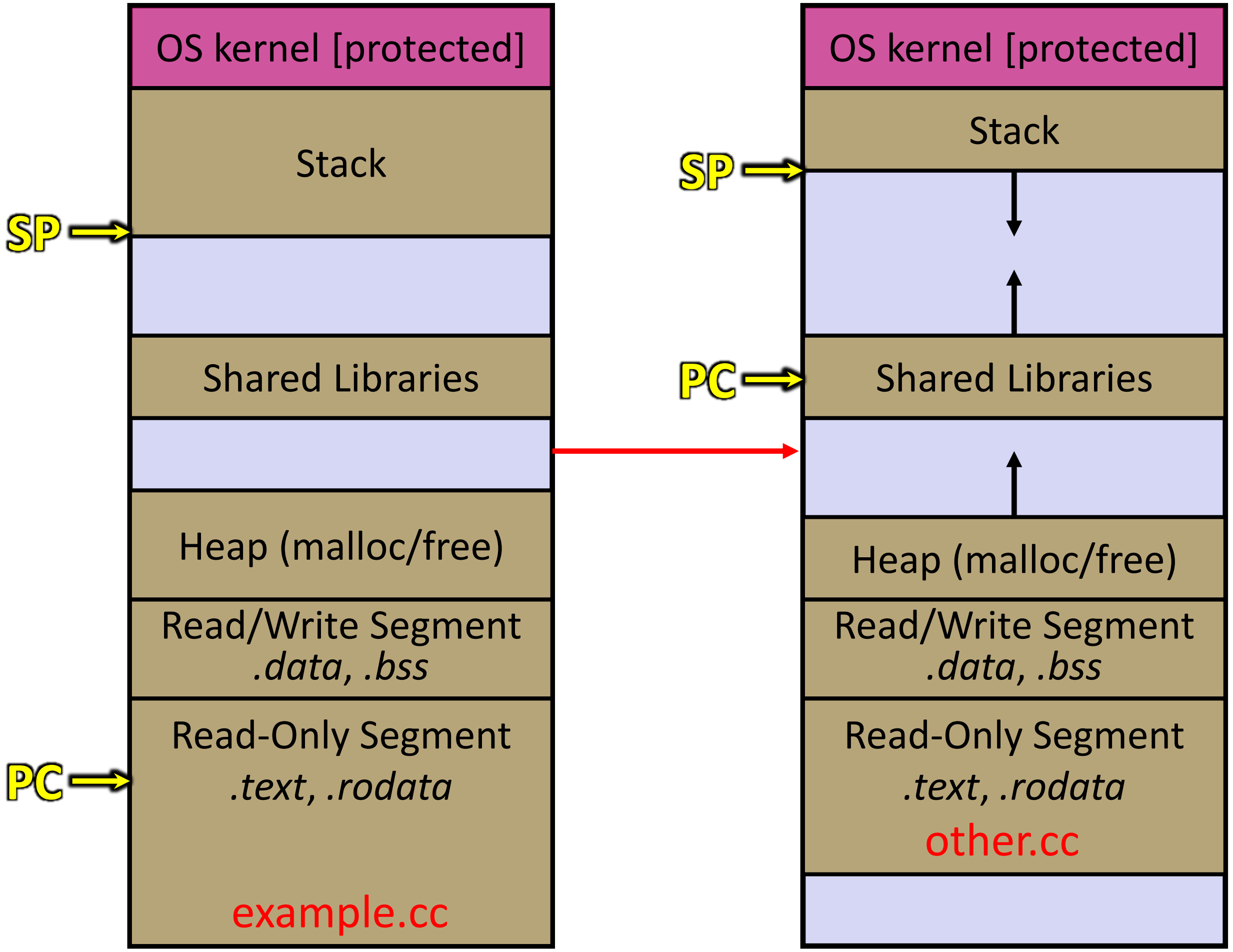
- PC, SP, registers, and memory are all reset so that the specified program can run

execve()

- ```
  int execve(const char *file,
             char* const argv[]
             char* const envp[]);
  ```
- 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 as the last entry of the array
- Just pass in an array of `{ NULL };` as envp
- 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

- ```c
  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

```
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 is broken. It compiles, but it doesn't do what we want. It is trying to compile some code and then run it.

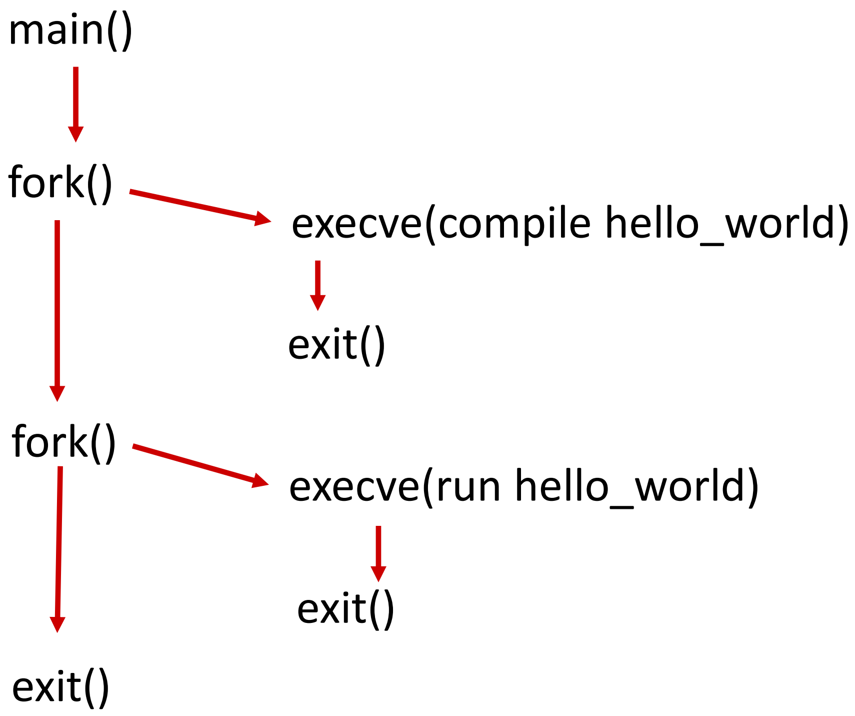
Why is this broken?

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

utograder.c

Poll Everywhere

pollev.com/tqm



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

Poll Everywhere

pollev.com/tqm

- 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[]) {
  char* envp[] = { NULL };

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

   printf(":) \n");

  return EXIT_SUCCESS;
}
```

Lecture Outline

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

From a previous poll:

```
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

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

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

"waiting" for updates on a Process

Usual change in status is to "terminated"

- ```
  pid_t wait(int *wstatus);
  ```
  - Calling process waits for any child process to change status
    - Also cleans up the child process if it was a zombie/terminated
  - Gets the exit status of child process through output parameter **`wstatus`**
  - Returns process ID of child who was waited for or `-1` on error
  ```

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

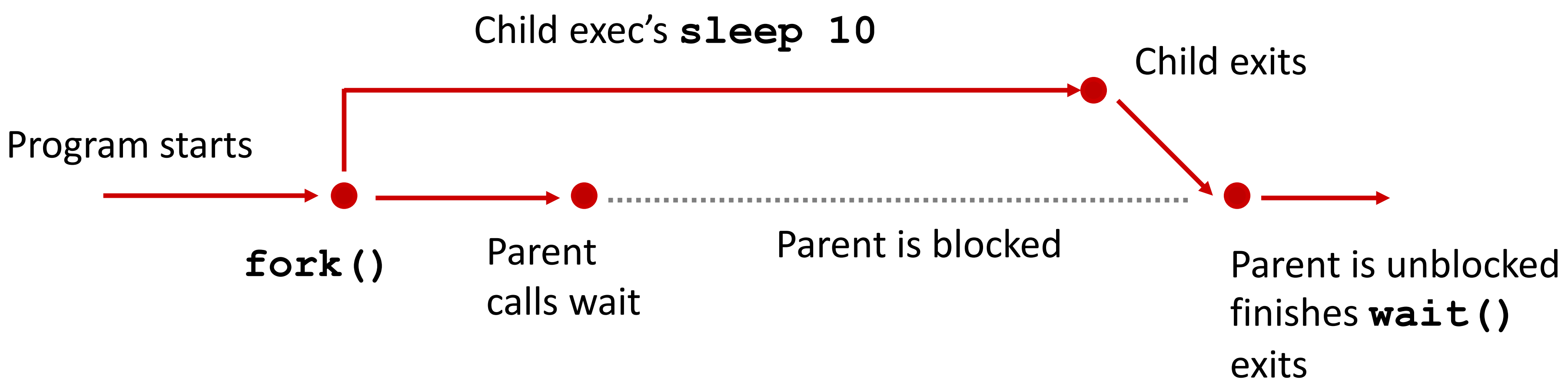
```
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };
  // fork a process to exec clang
  pid_t clang_pid = fork();
  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
                "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv,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

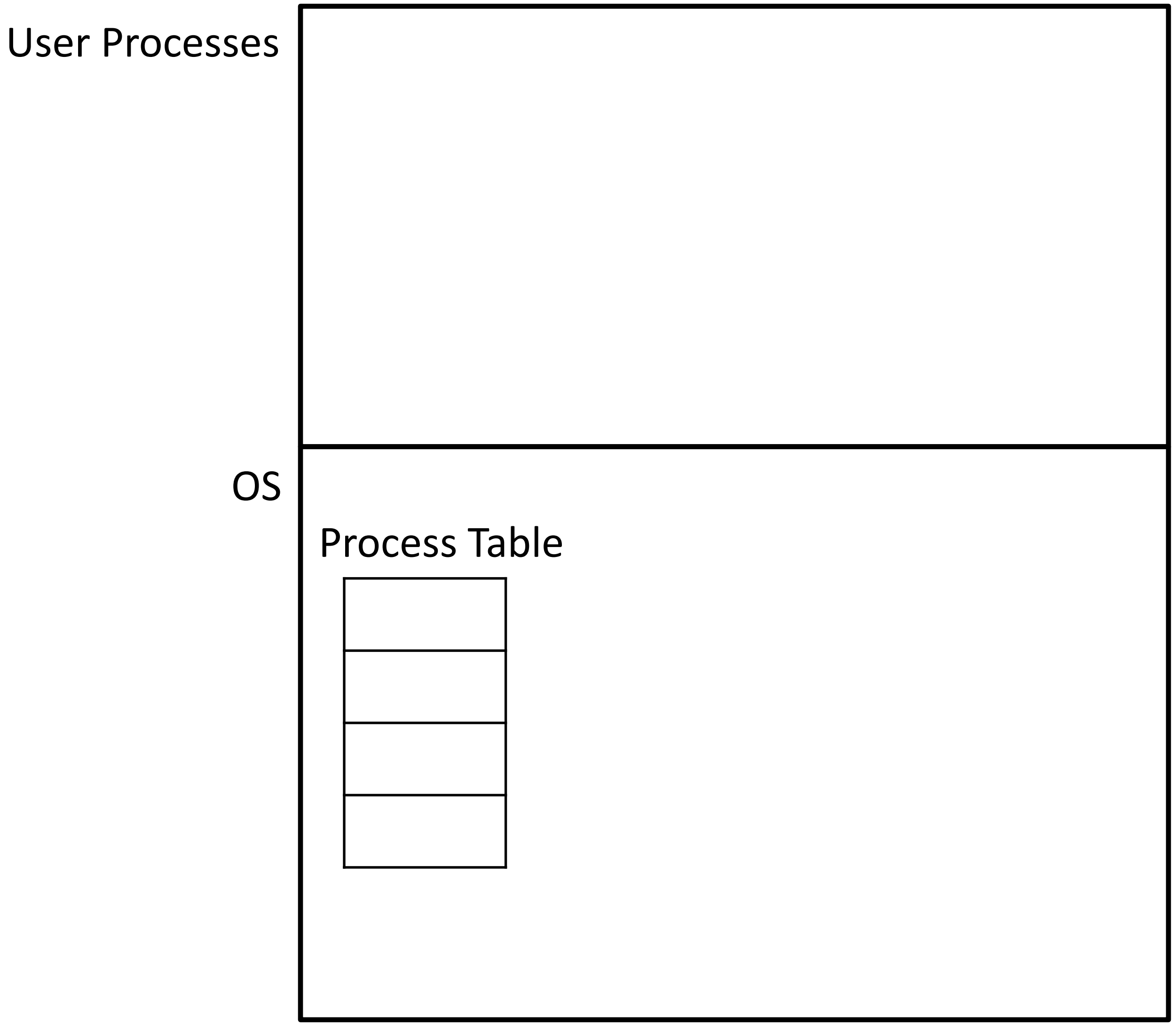


Diagram: wait_example.c

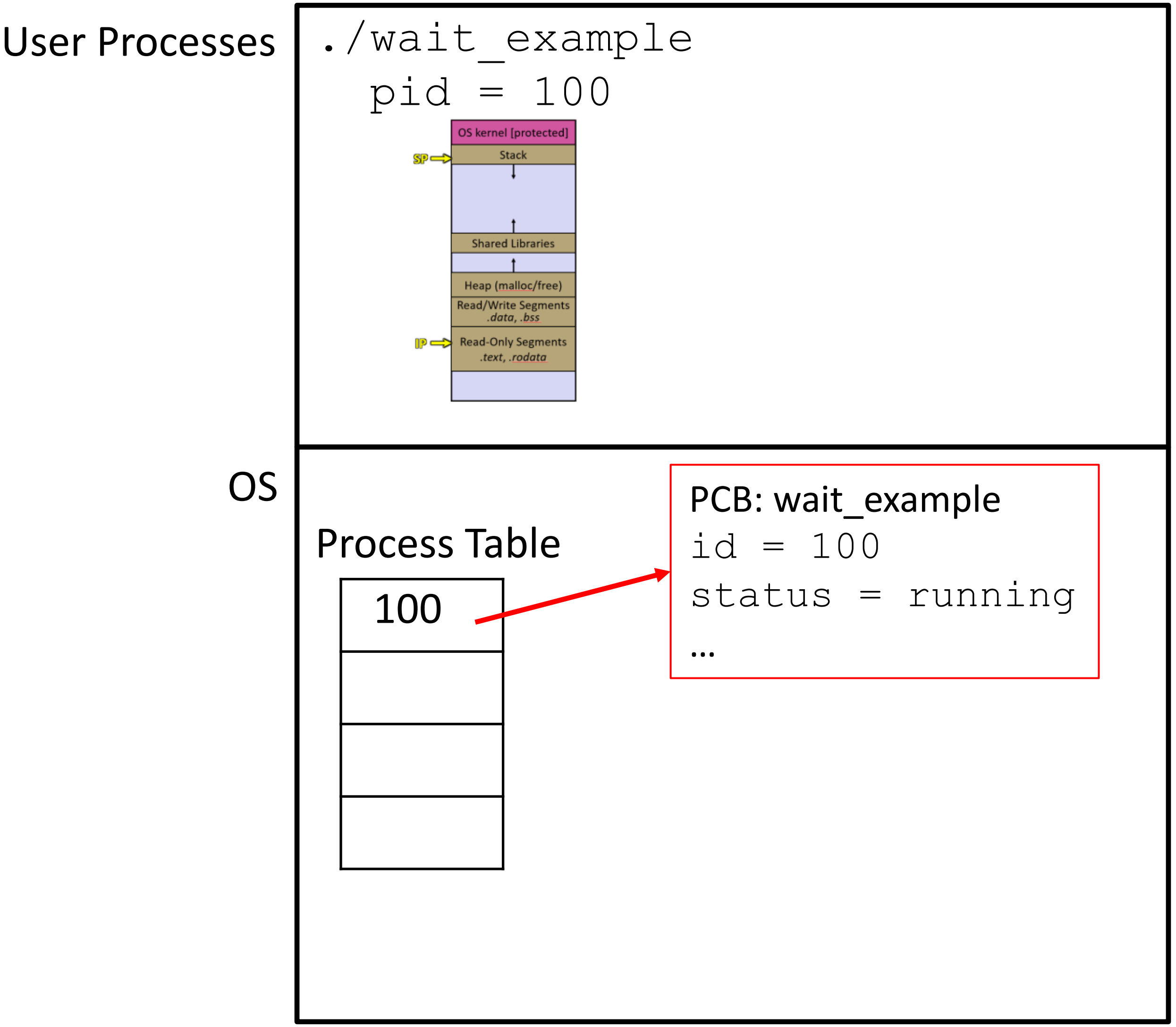


Diagram: wait_example.c

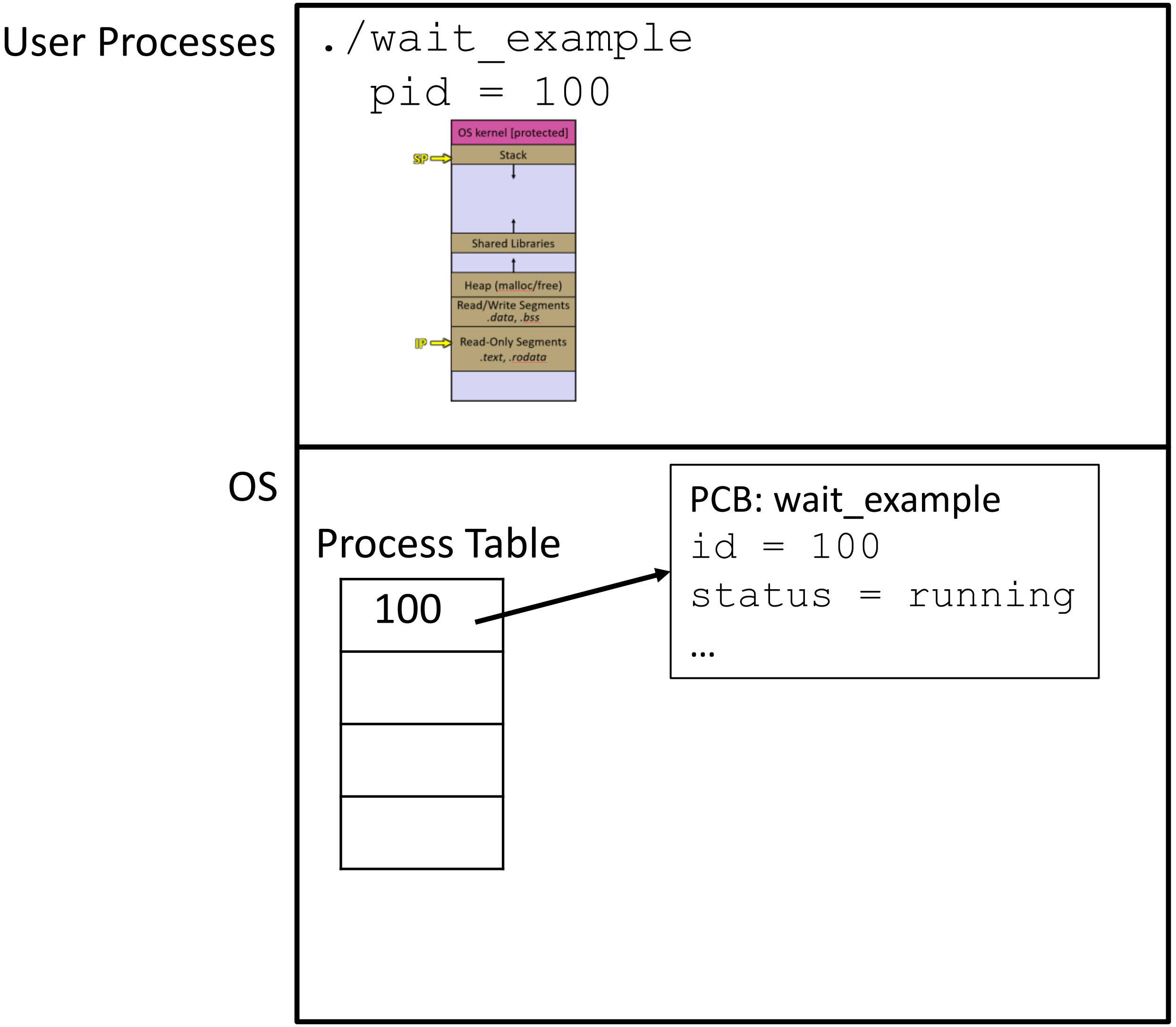


Diagram: wait_example.c

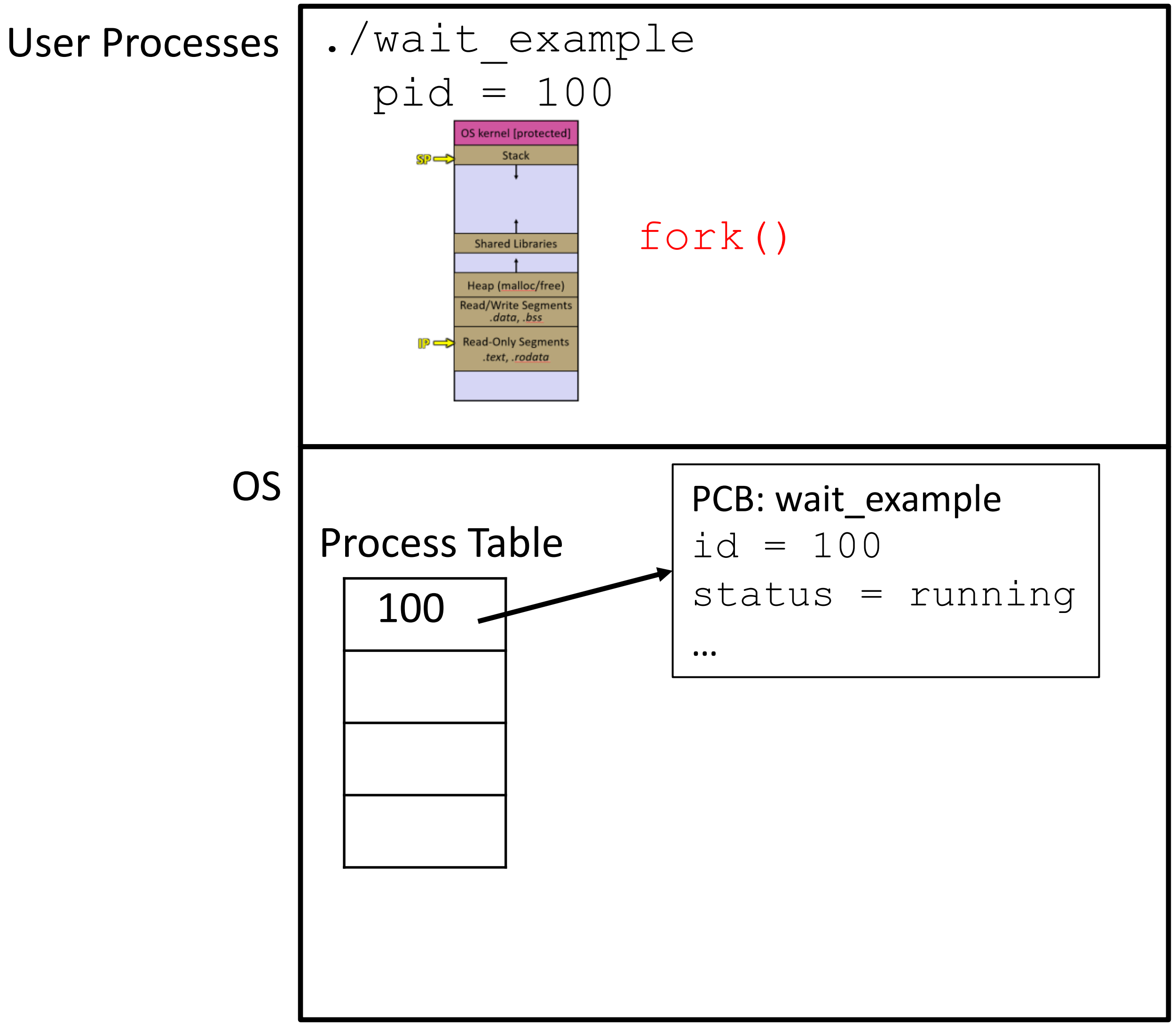


Diagram: wait_example.c

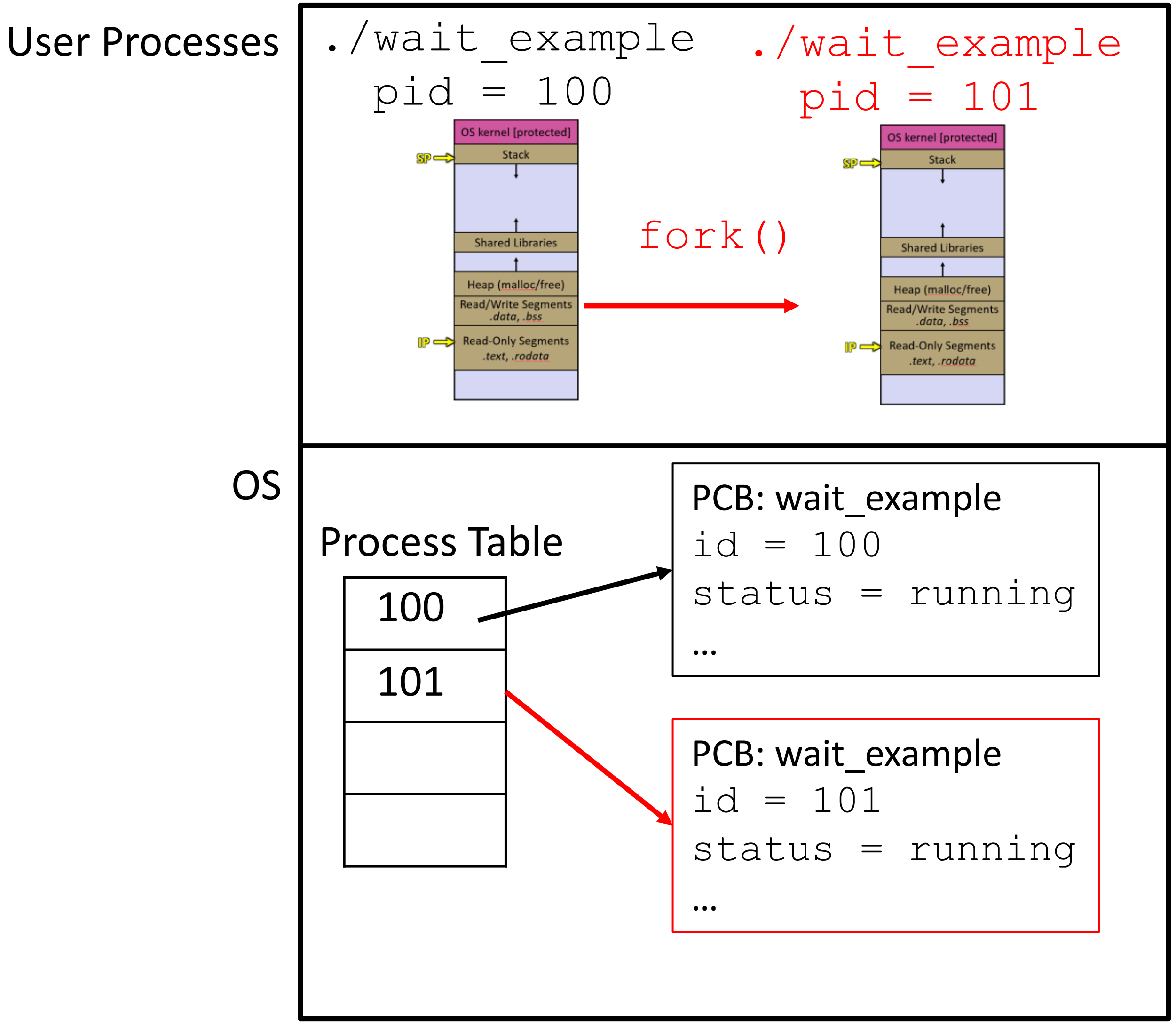


Diagram: wait_example.c

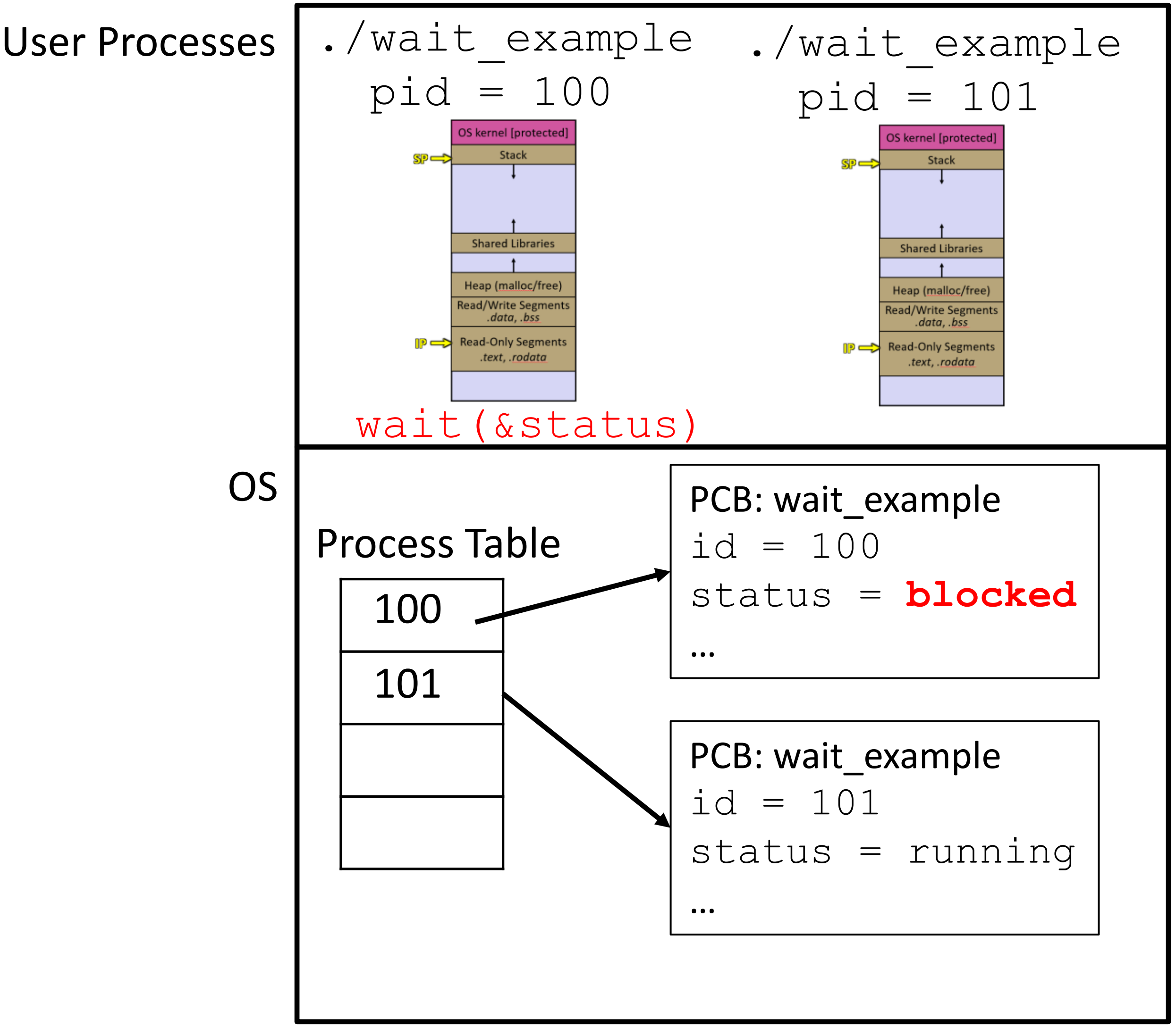


Diagram: wait_example.c

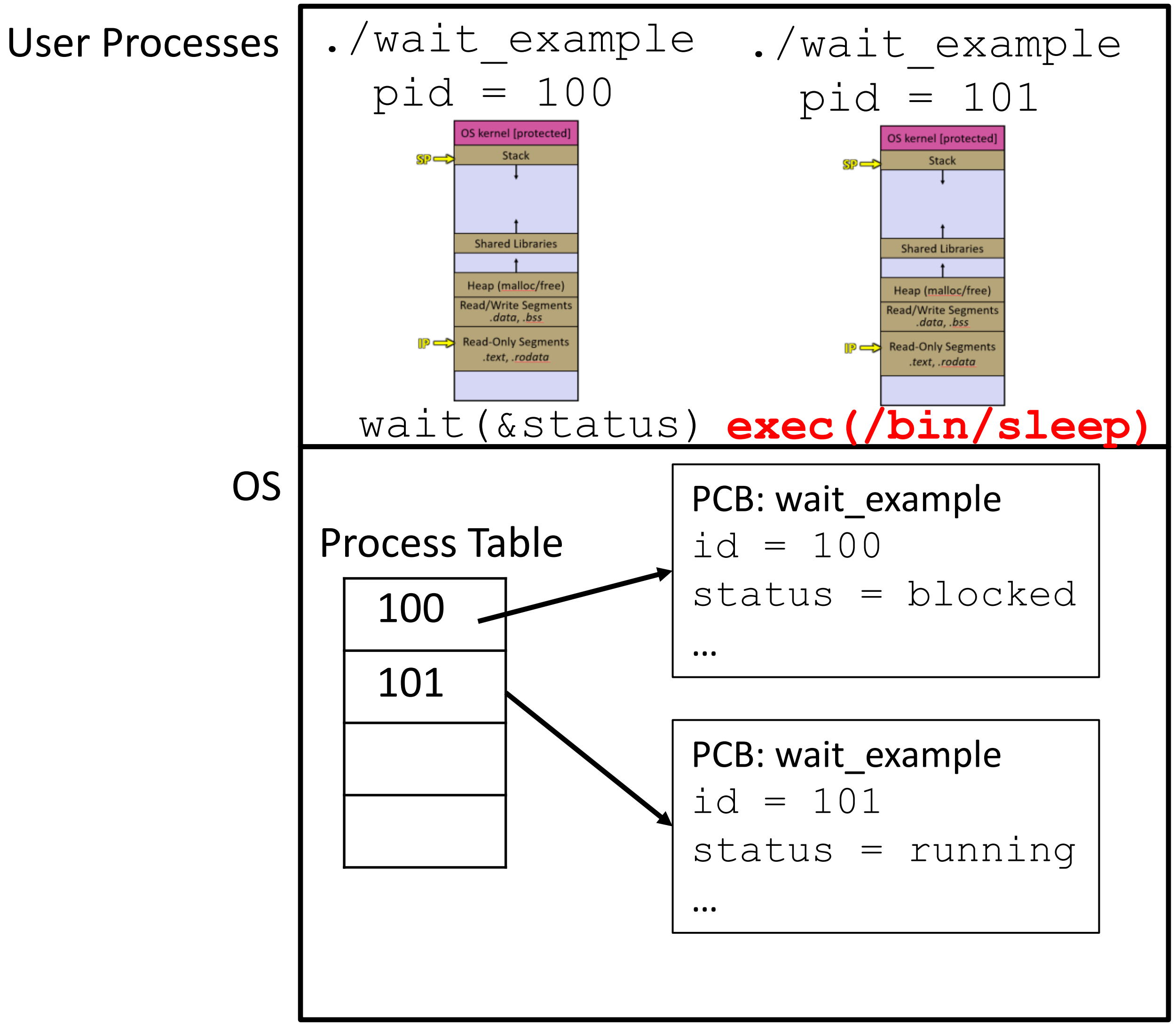


Diagram: wait_example.c

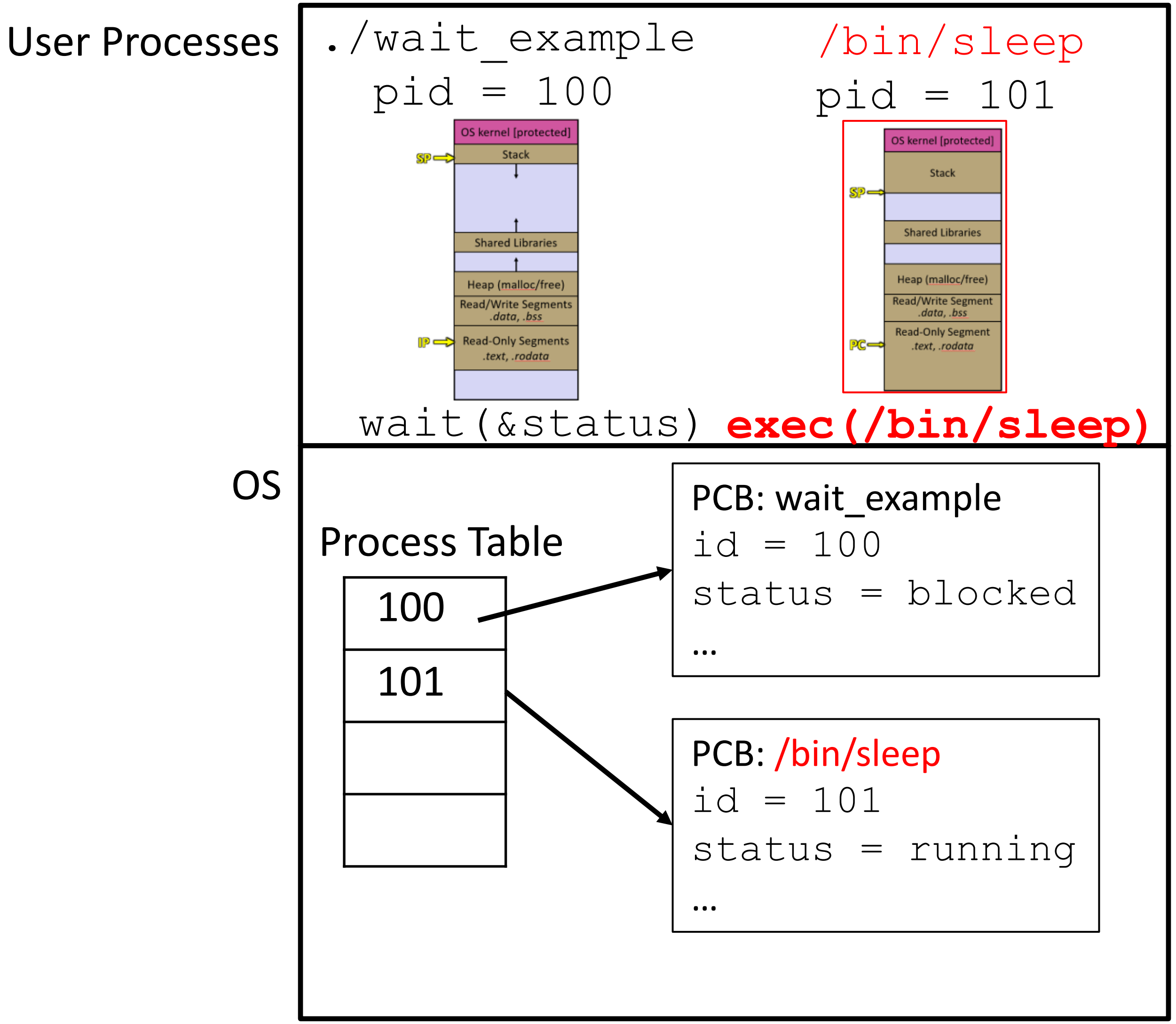


Diagram: wait_example.c

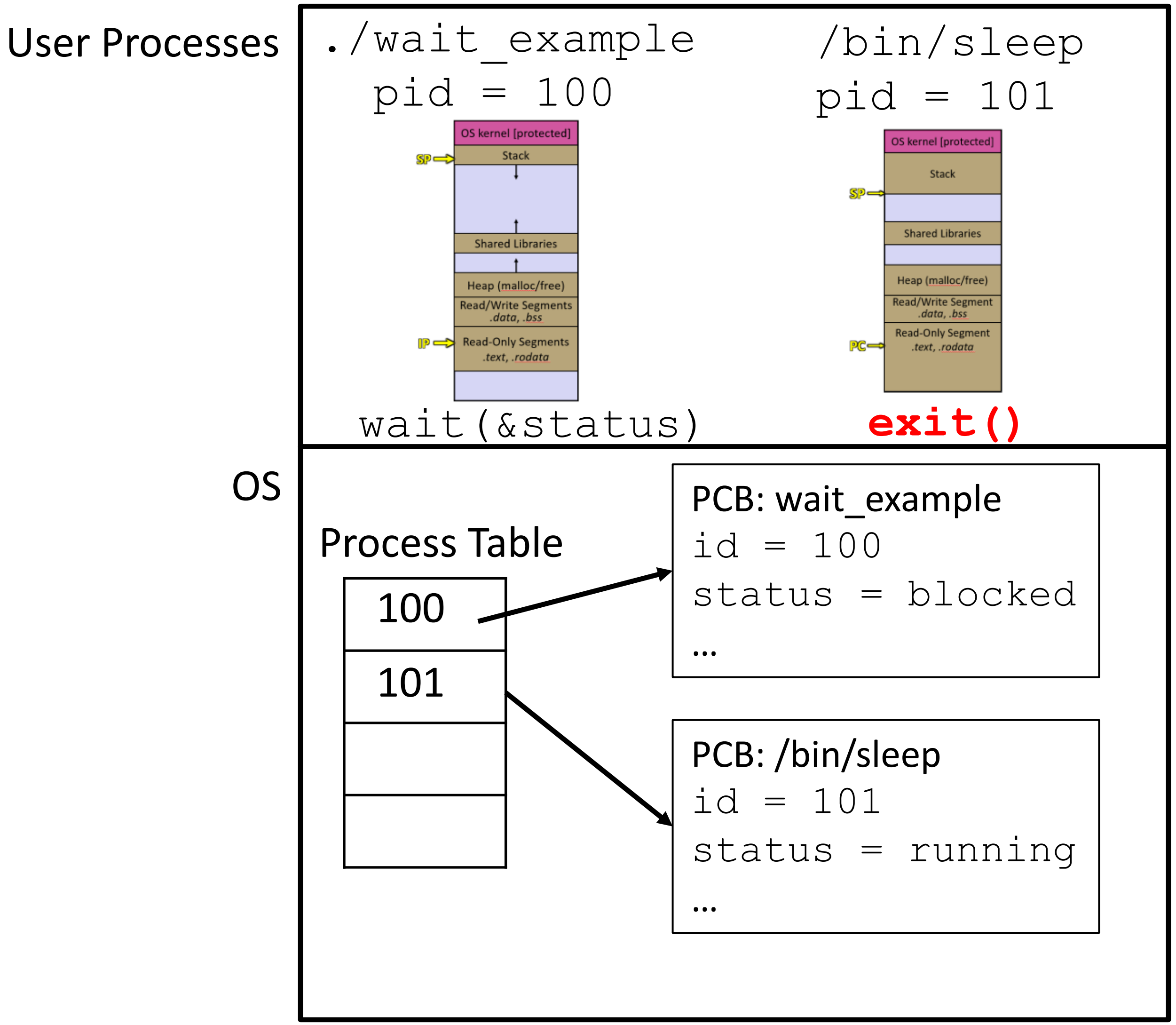


Diagram: wait_example.c

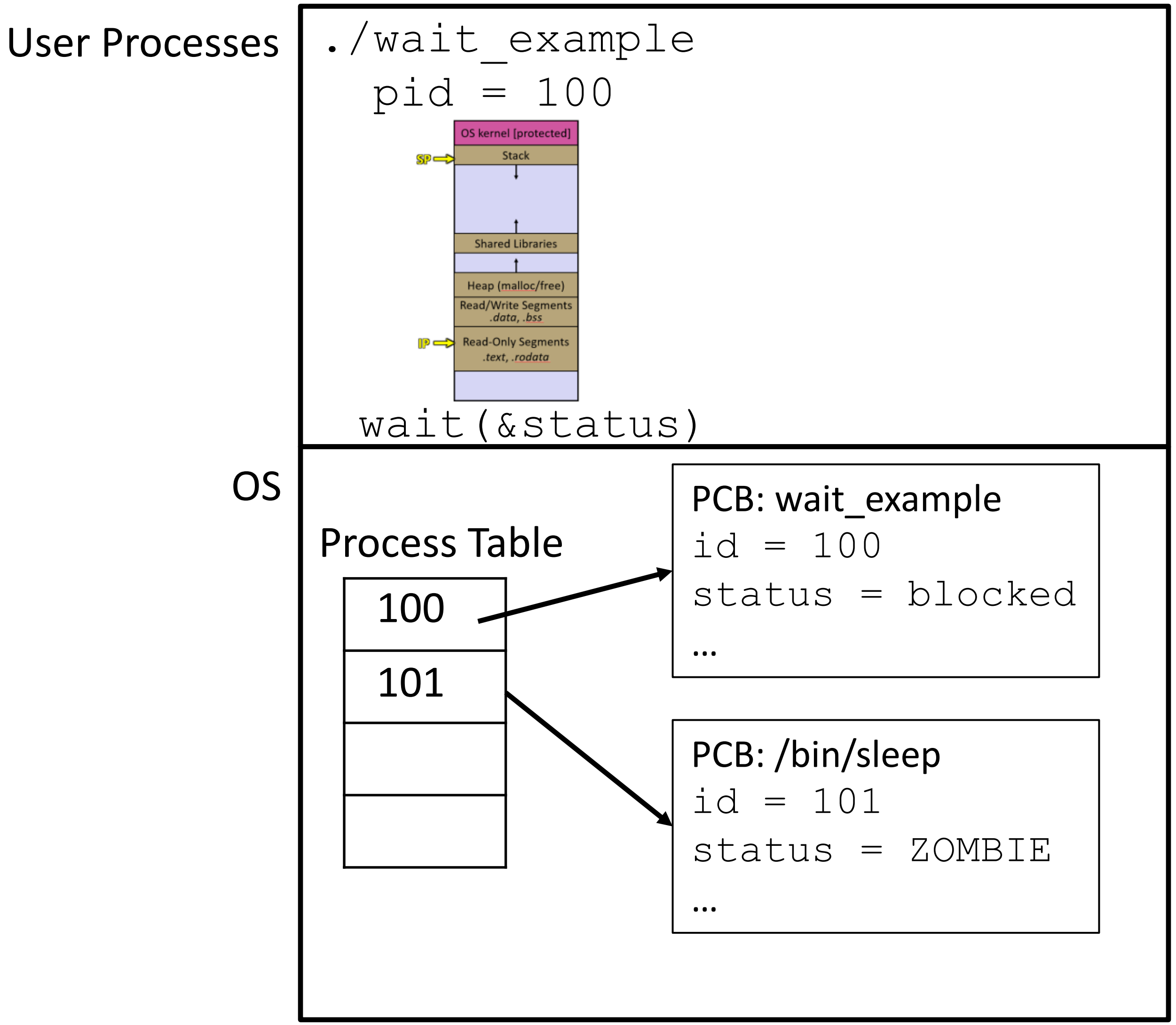


Diagram: wait_example.c

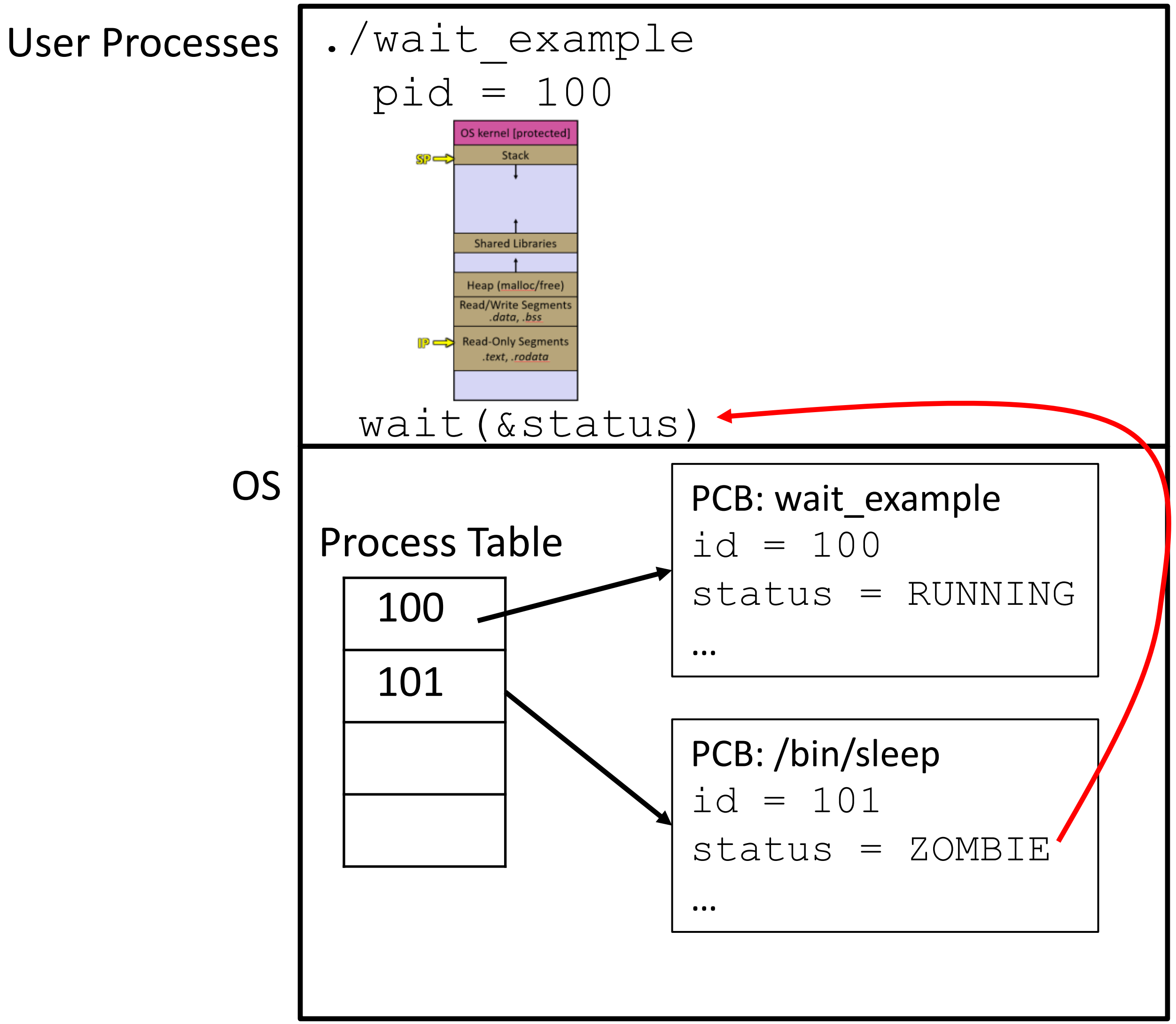


Diagram: wait_example.c

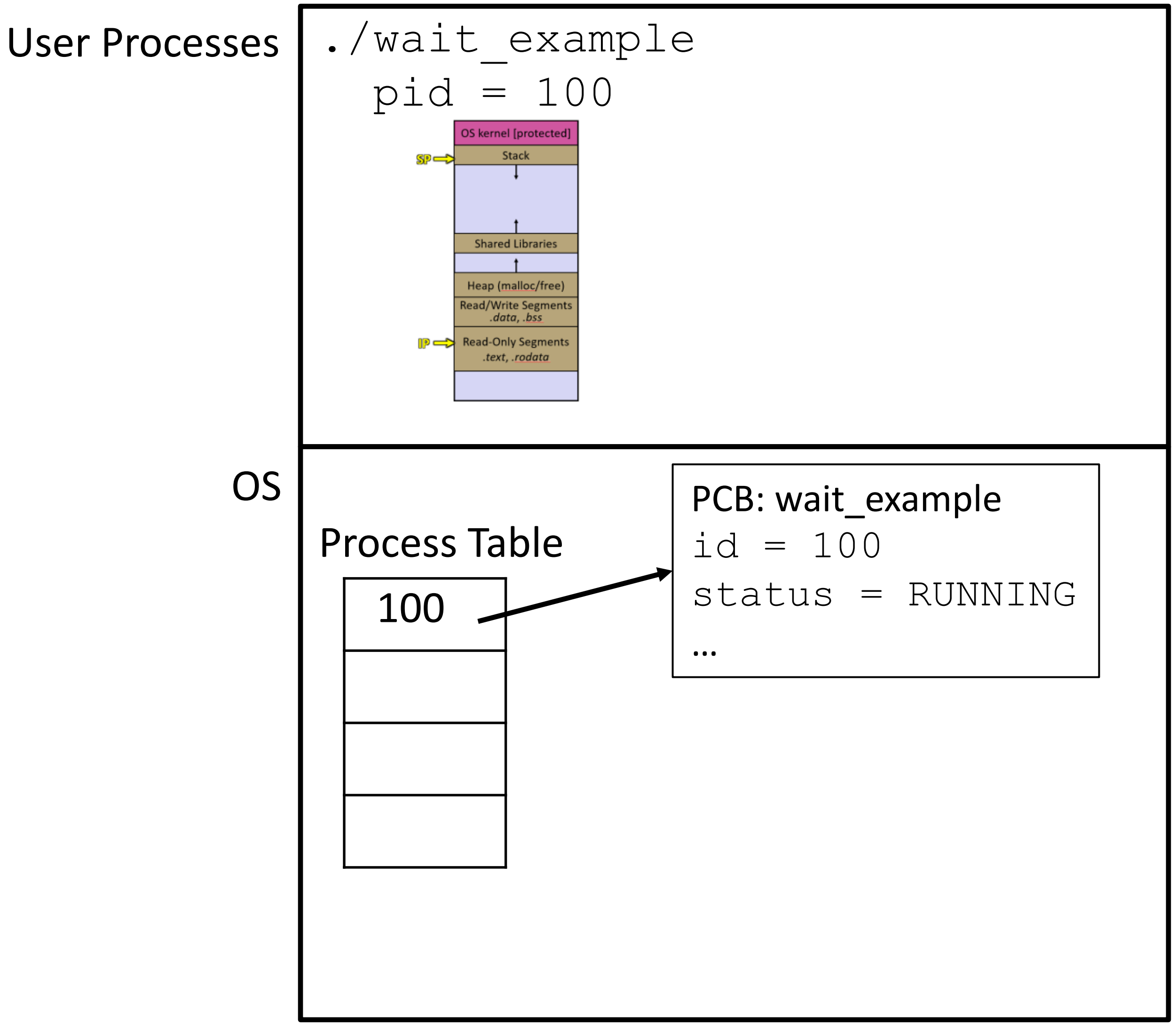


Diagram: wait_example.c

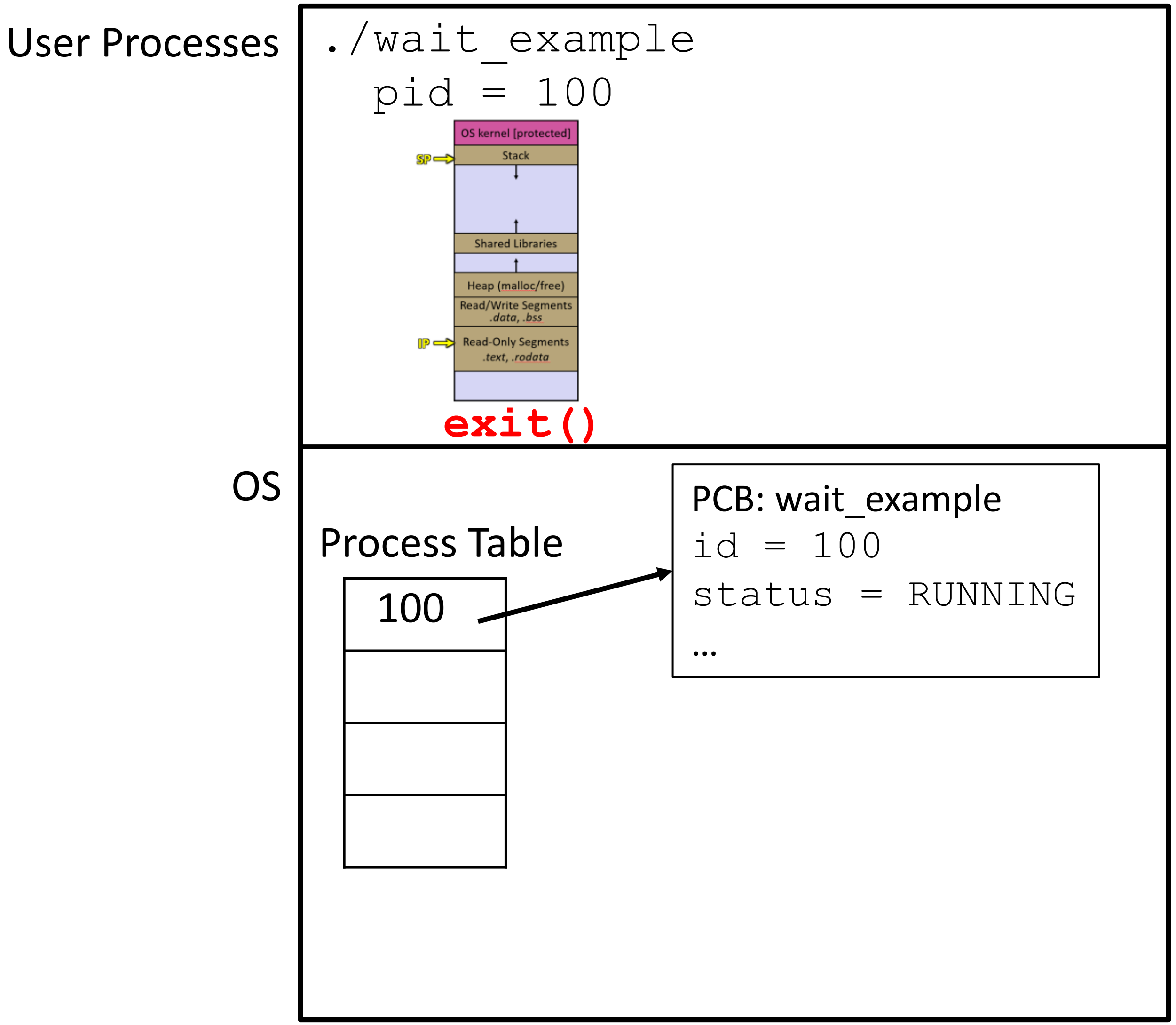
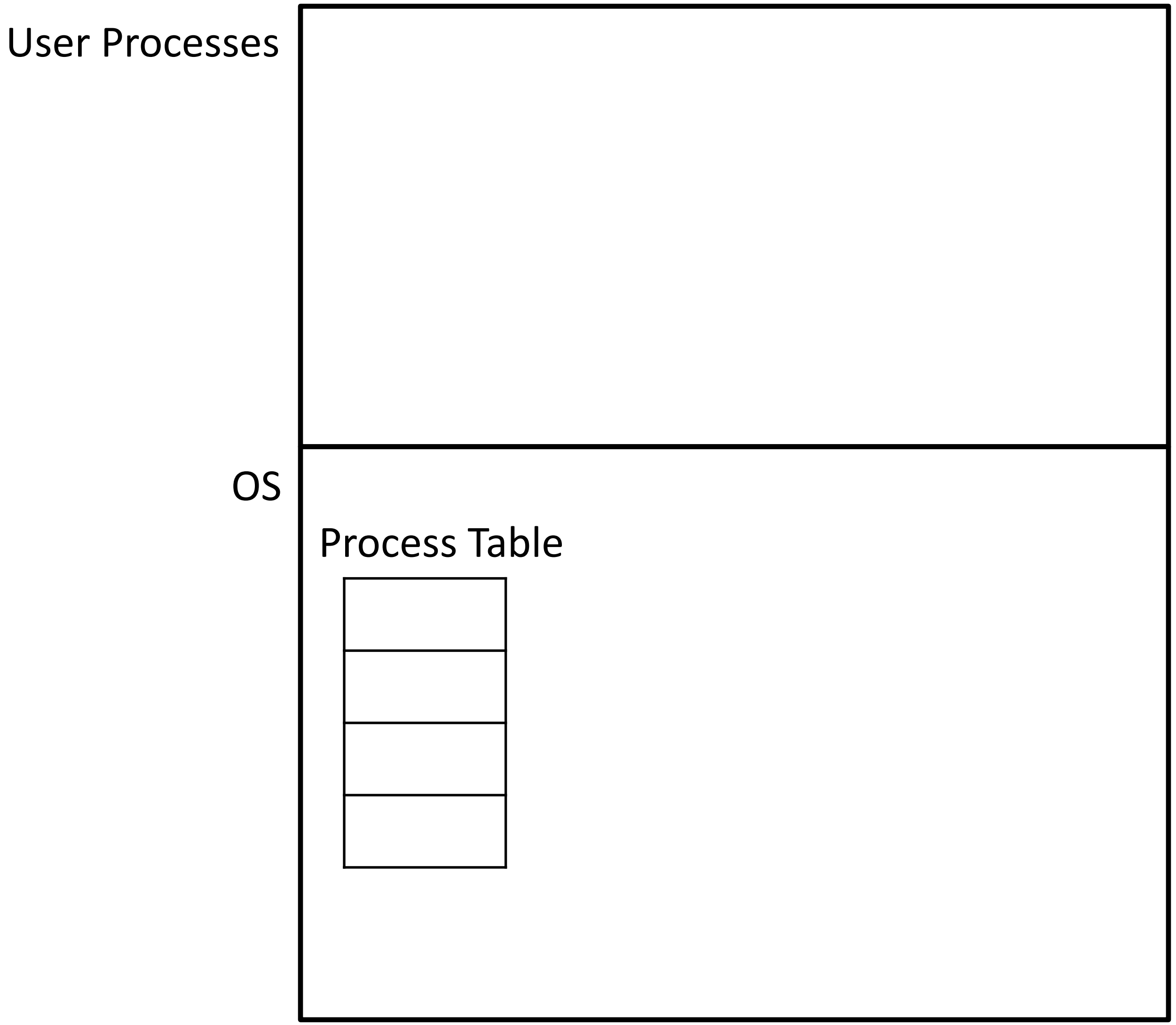


Diagram: wait_example.c



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

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()`, `waitpid()` and `exit()`!

  - Aside: `sleep()` takes in an integer number of seconds and blocks till those seconds have passed

More: `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

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

- See example in `state_check.c`

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/tqm

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

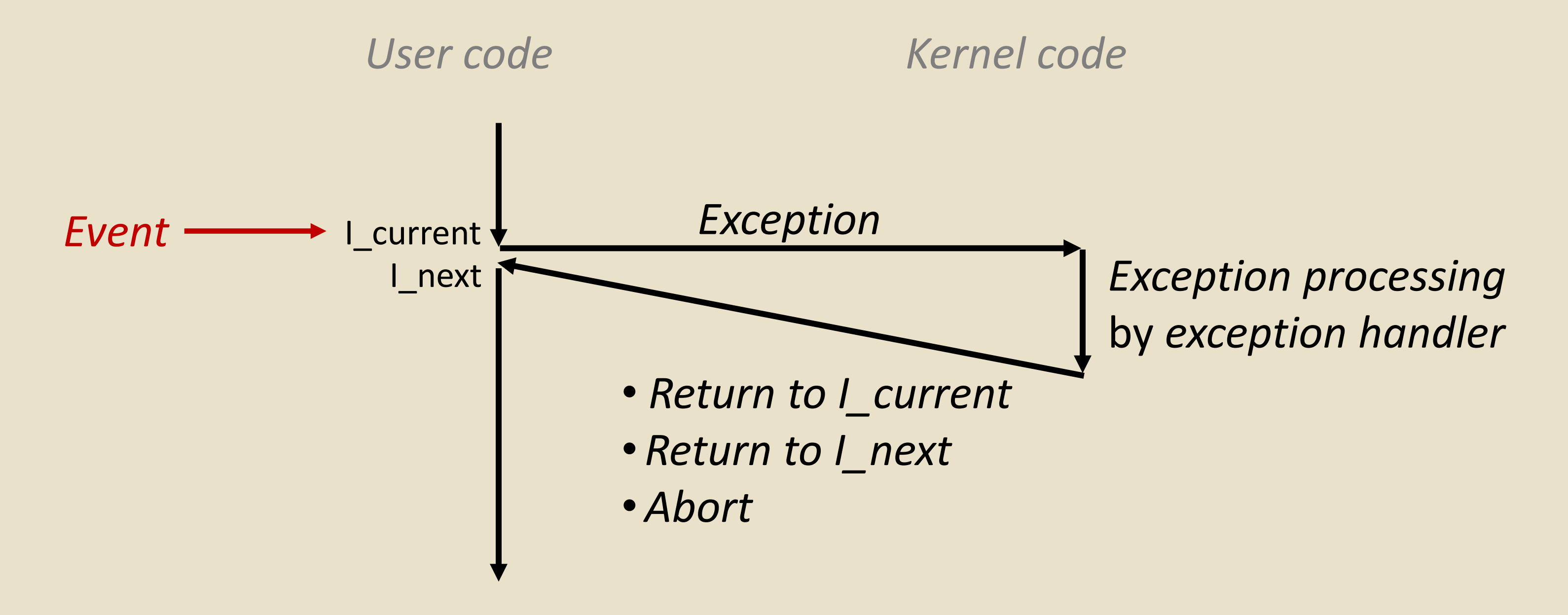
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Hardware Interrupts**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software

  what we will be looking at today

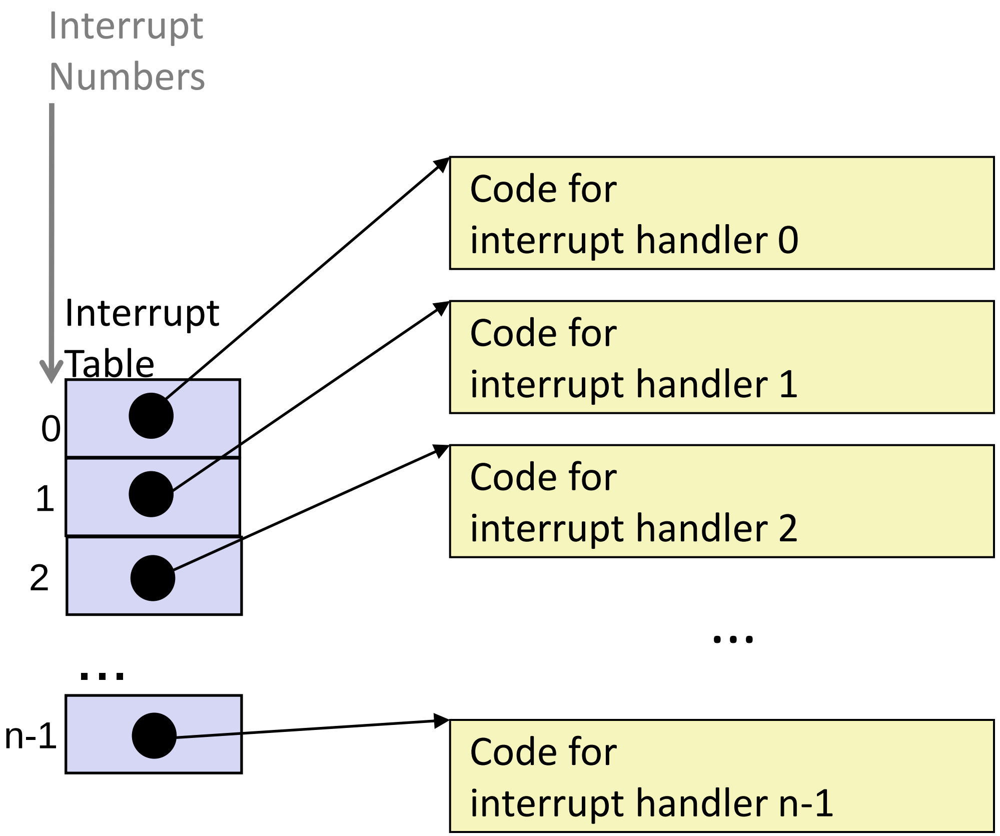
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software

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:
  - ***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

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

sigaction()

- You can change how a certain signal is handled
- ```
  int sigaction(int signum,   struct sigaction* act,
                struct sigaction* old);
  ```
- Signum specifies a 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 & behaviour for that signal through the **old** output parameter
- Some signals like **SIG_KILL** and **SIG_STOP** can't be handled differently
  ```

Signal handlers

- ```c
  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 behaviour 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
  ```

struct sigaction

- Has 5 different fields to specify the behaviour of how a signal should be handled. For our case, we only care about **sa_handler** and **sa_flags**
  - (for now)

```
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 ignore the signal
  - Set **sa_handler** to **SIG_DFL** for default behaviour

- In this class: set **sa_flags** to **SA_RESTART**
  - This makes it so that system calls are automatically restart/continue if they are interrupted by a signal.
  ```

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()

- Alarm

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

- Delivers the **SIGALRM** signal to the calling process after the specified number of seconds

- Default **SIGALRM** behaviour: terminate the process

- How to cancel alarms?
  - I leave this as an exercise for you: try reading the man pages

- HINT FOR EXTRA CREDIT: What is the default behaviour of SIGALRM? Can you take advantadge of the default behaviour?

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

kill()

- Can send specific signals to a specific process manually
- ```
  int kill(pid_t pid, int sig);
  ```
- pid: specifies the process
- sig: specifies the signal

- ```
  kill(child, SIGKILL);
  ```

- 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()**

```
#define _POSIX_C_SOURCE 1
```

Non blocking wait w/ `waitpid()`

- ```
  pid_t waitpid(pid_t pid, int *wstatus,
                           int options);
  ```
  - Can pass in `WNOHANG` for **`options`** to make **`waitpid()`** not block or "hang".
  - Returns process ID of child who was waited for or -1 on error or 0 if there are no updates in children processes and WNOHANG was passed in
  ```

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

  - LOOKS SIMILAR TO WHAT YOU ARE DIONG IN penn-shredder. DO NOT COPY THIS
    - **`waitpid()`** IS NOT ALLOWED
    - USING **`sleep()`** AND **`alarm()`** TOGETHER CAN CAUSE ISSUES

SIGCHLD handler

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

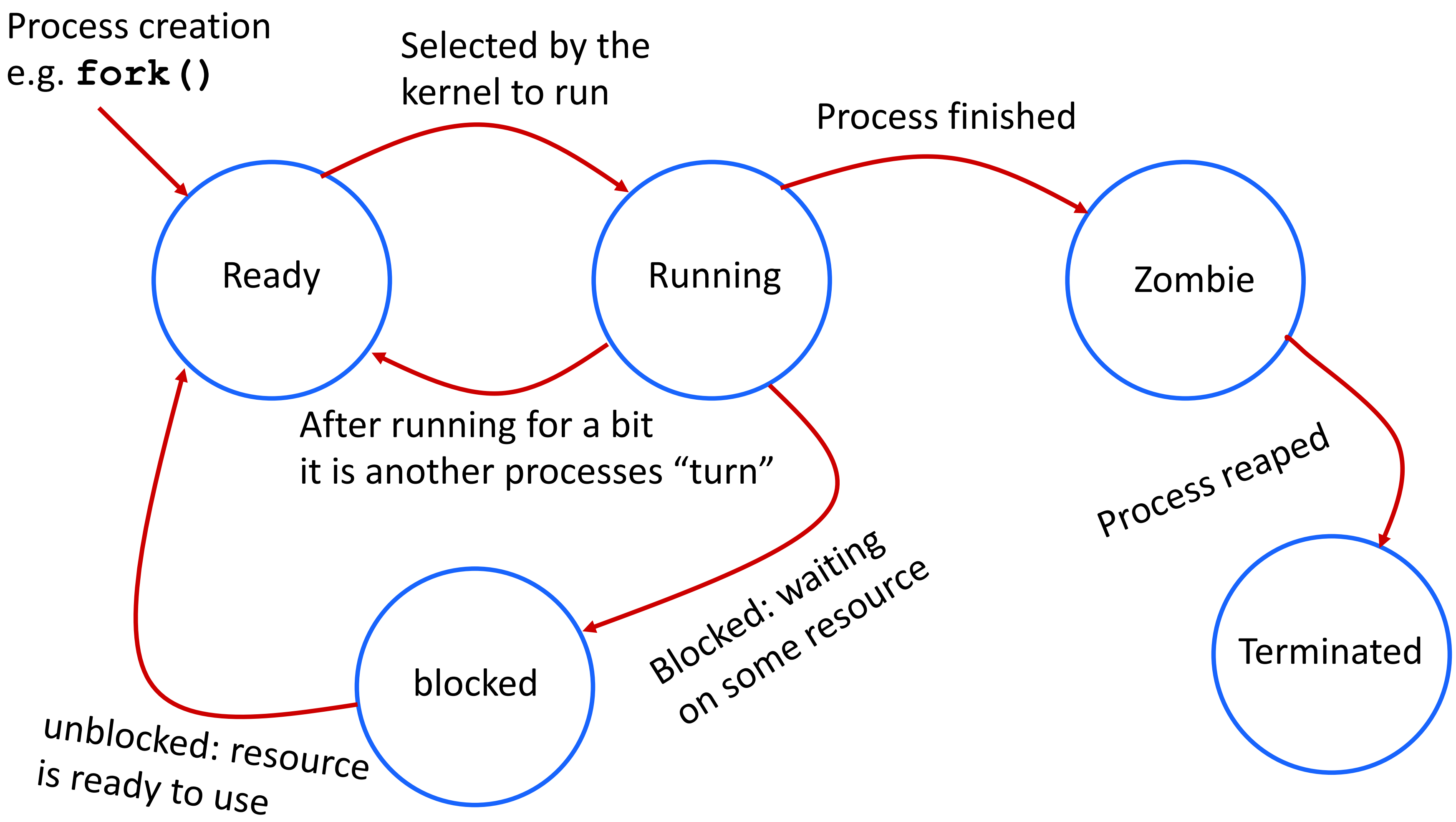
- You can write a signal handler for **`SIGCHLD`**, and use that to help handle children update statuses: allowing the parent process to do other things instead of calling **`wait()`** or **`waitpid()`**

- Relevant for proj2: **`penn-shell`**

Lecture Outline

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

Process State Lifetime



Lecture Outline

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