Processes, wait(), signal() and more!

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Proj0 (penn-shredder) Due 09/13 @ 11:59 pm
  - This includes git & docker setup instructions. Do this part ASAP, it can take a while to debug issues with setup
  - **This assignment is done on your own**
- Check-in Quiz 0 Due before this lecture
  - Still open, to account for students joining the course a bit late
  - Don't expect this to be true with future quizzes
- Check-in Quiz 1 Due in ~1 week

Administrivia

- Optional Recitations!
  - We are going to try having optional recitations
  - First one is today after lecture from 3:30 to 4:30
  - On zoom and Moore 100C
  - This one is about C refresher including valgrind & GDB
  - Materials will be shared afterwards
  - Will it always be this time slot? idk, TBD
  - Will we have one every week? Probably not, TBD

Discuss

- Any questions, comments or concerns from last lecture or the check-in Quiz?

Discuss

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

```c
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;
}
```

```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;
}
```

Lecture Outline

- **`wait()`, blocking, Zombies & PCB**
- `kill()`, `signal()`, `alarm()`
- Process Groups
- Process Lifetime

From last time:

```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

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

- `pid_t wait(int *wstatus);`

Usual change in status is to "terminated"

  - 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 last lecture

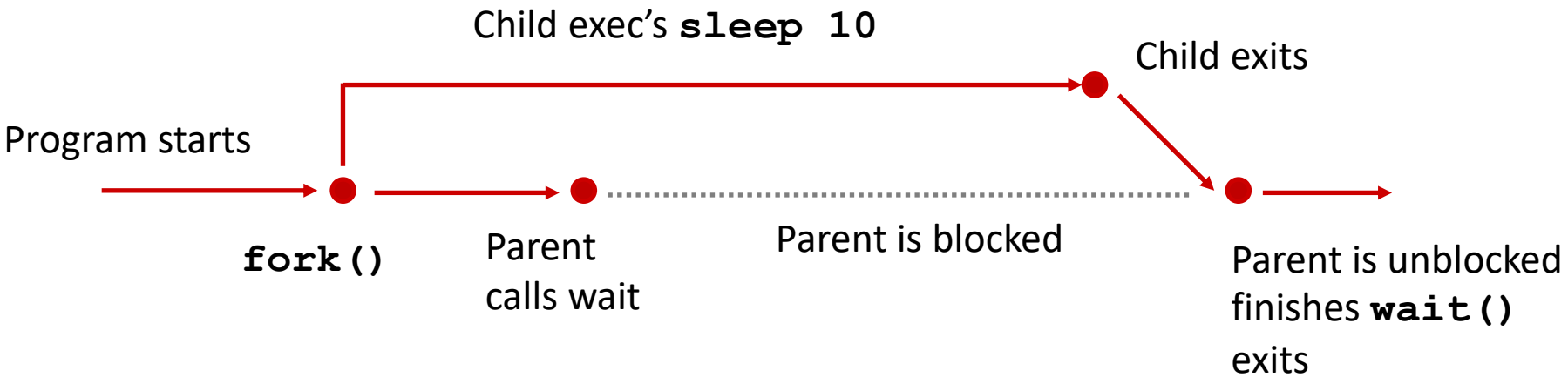
```c
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };
  // fork a process to exec clang
  pid_t clang_pid = fork();
  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
              "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv,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

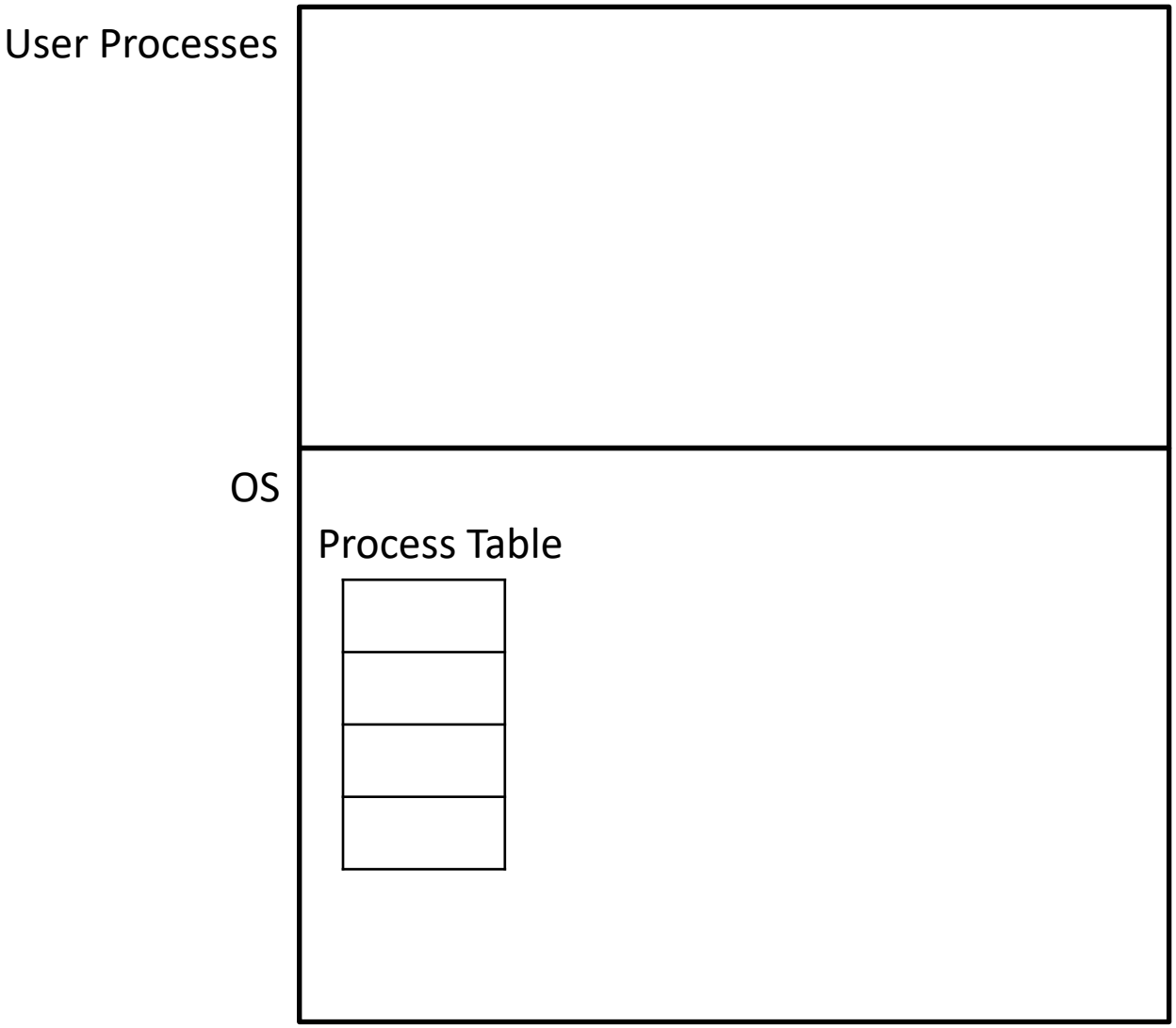


Diagram: wait_example.c

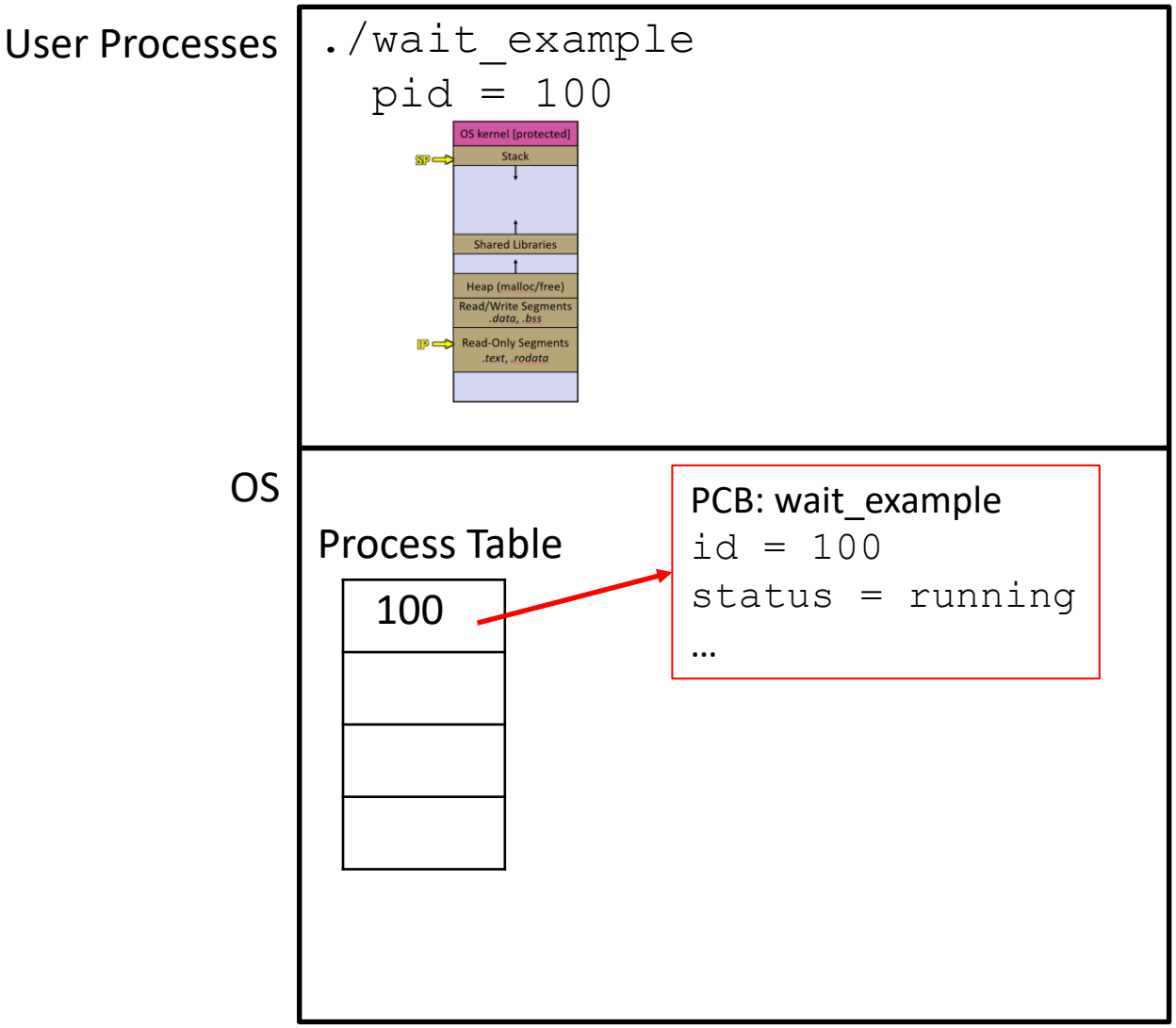


Diagram: wait_example.c

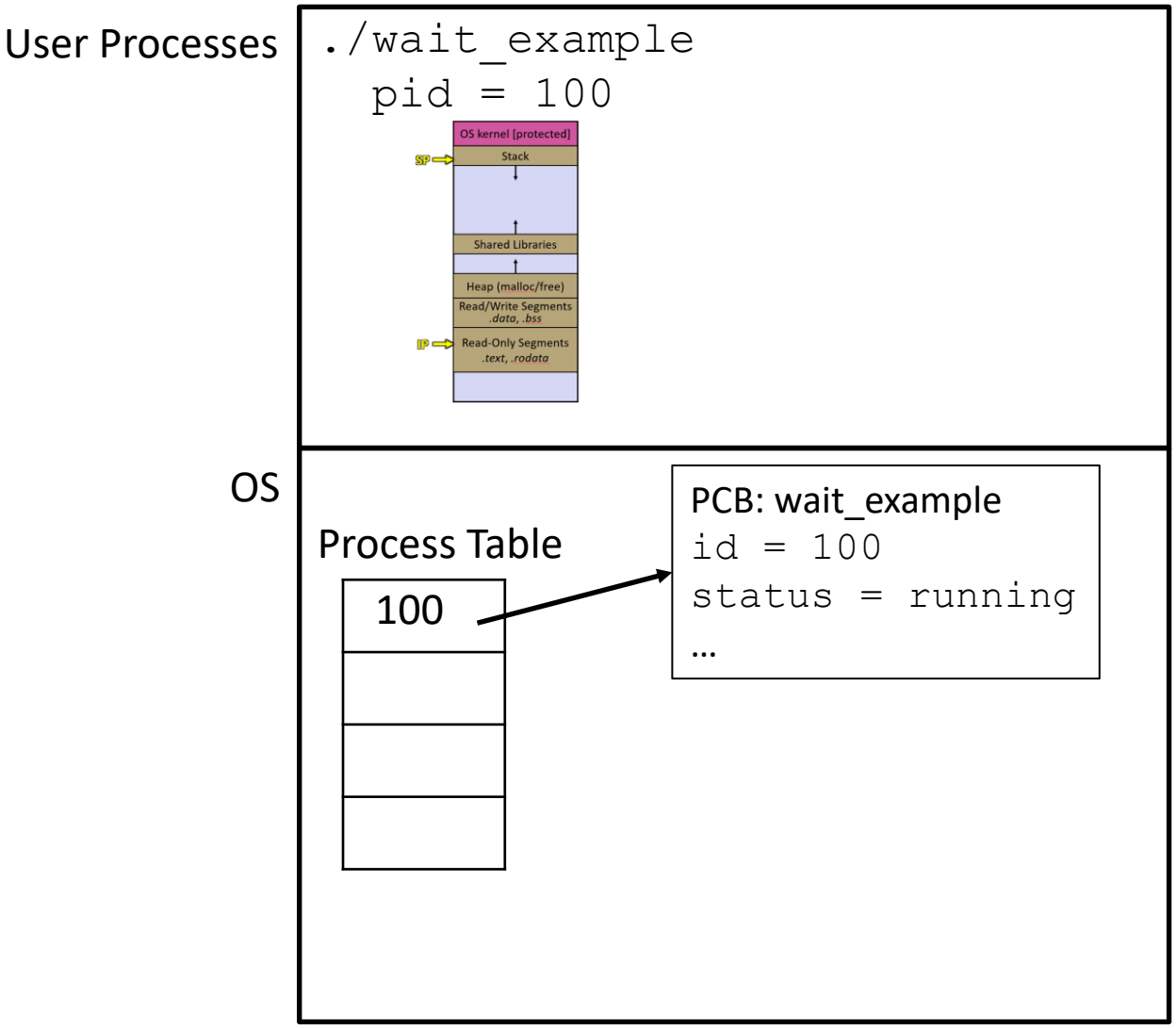


Diagram: wait_example.c

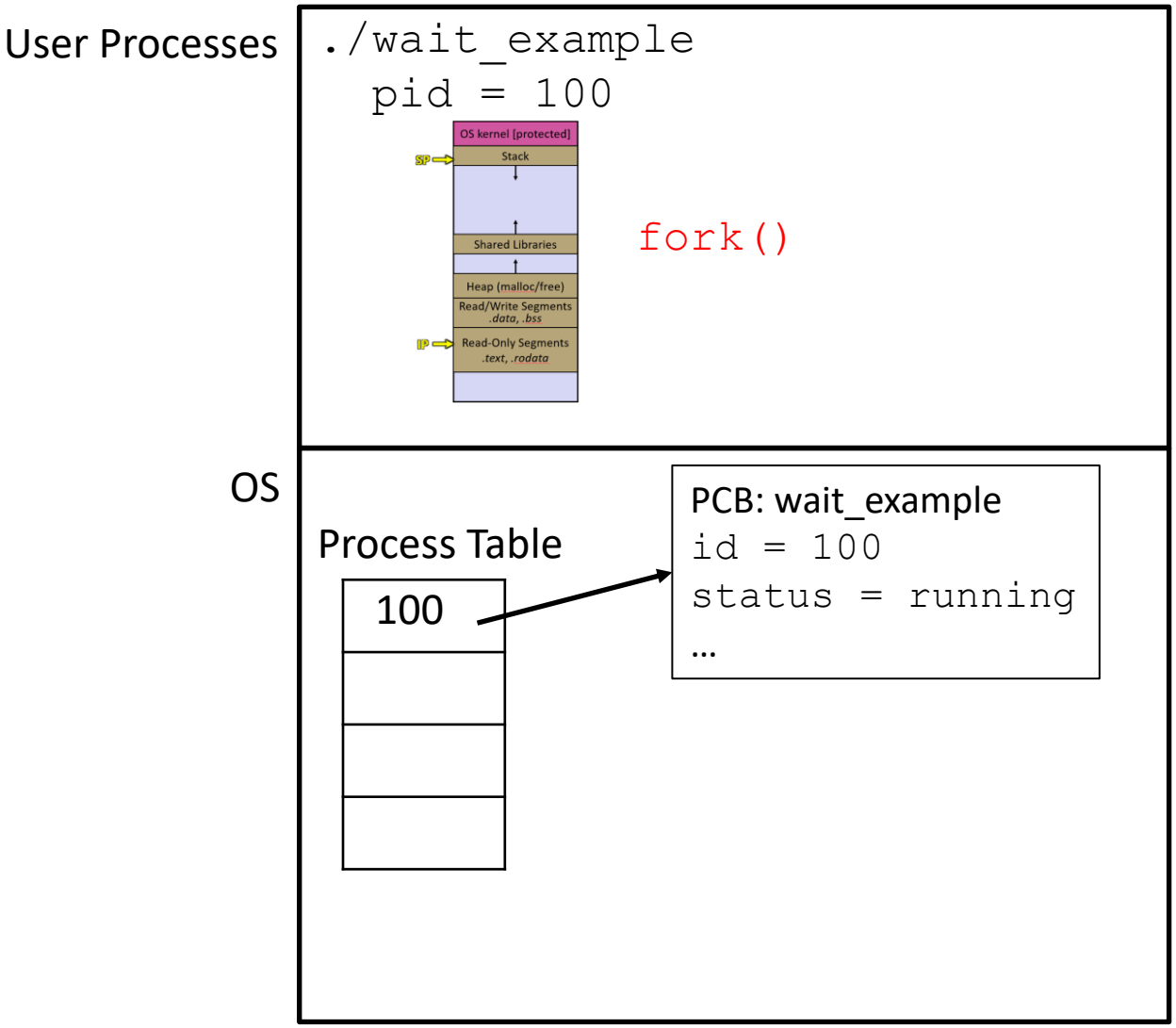


Diagram: wait_example.c

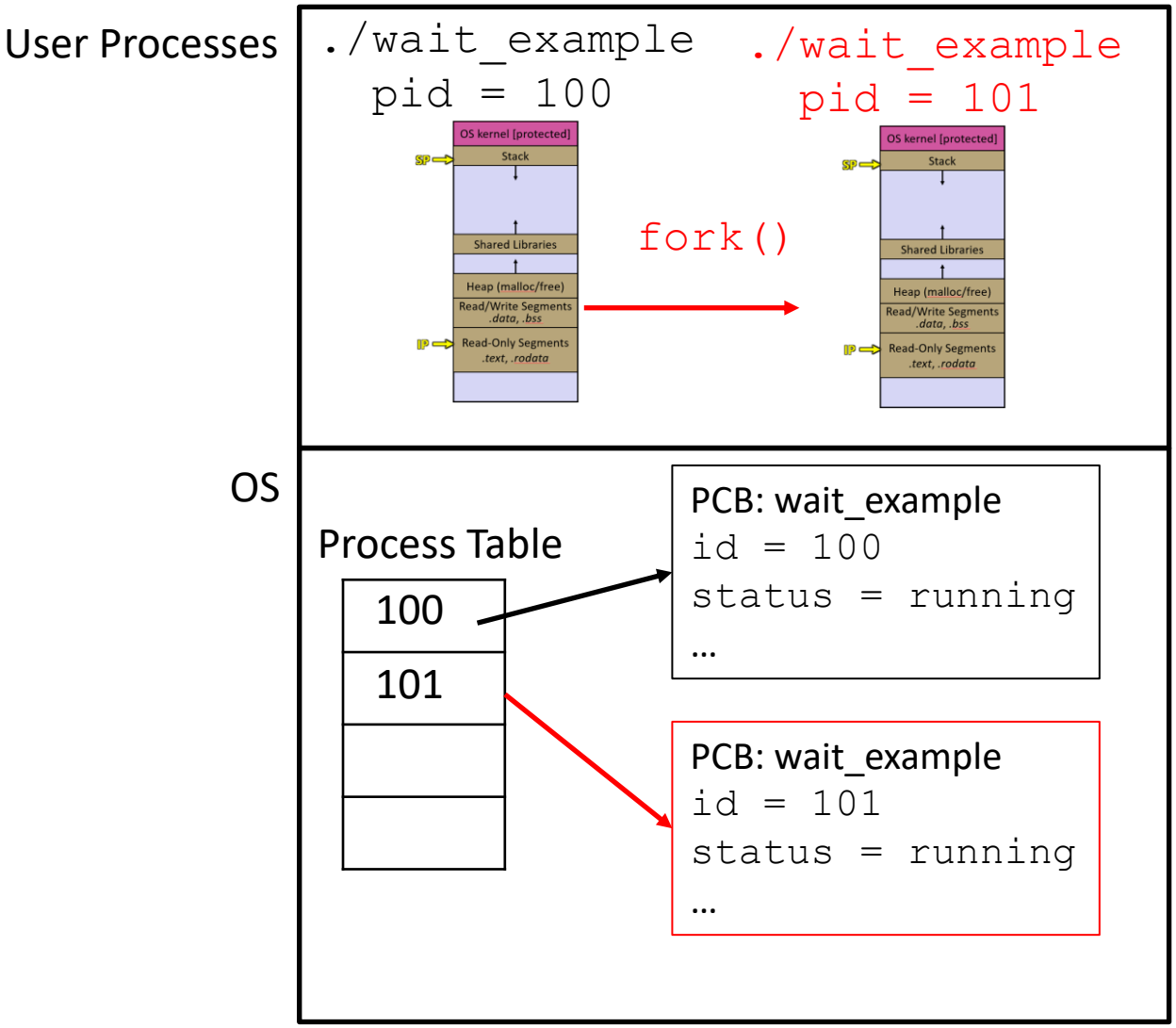


Diagram: wait_example.c

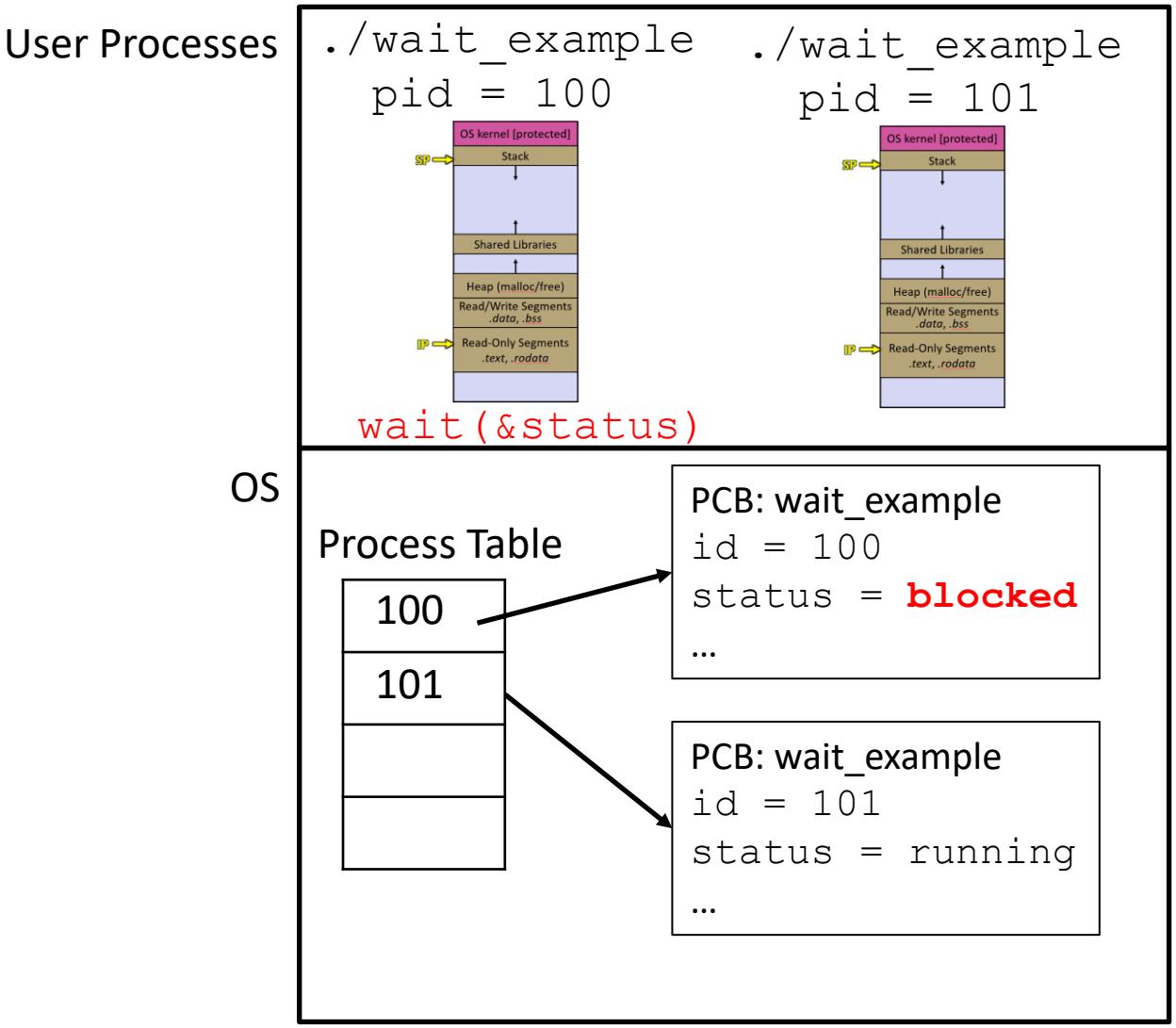


Diagram: wait_example.c

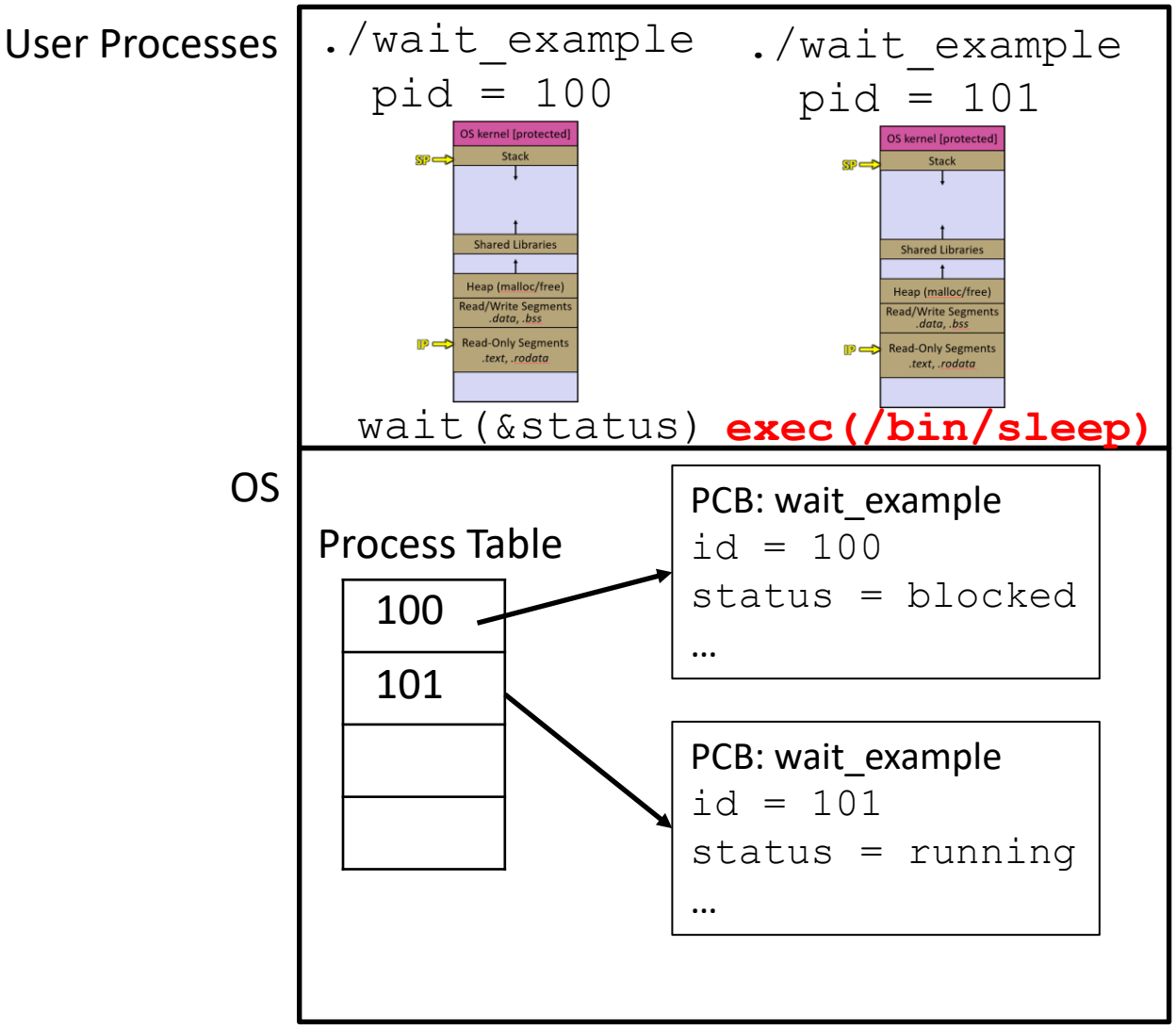


Diagram: wait_example.c

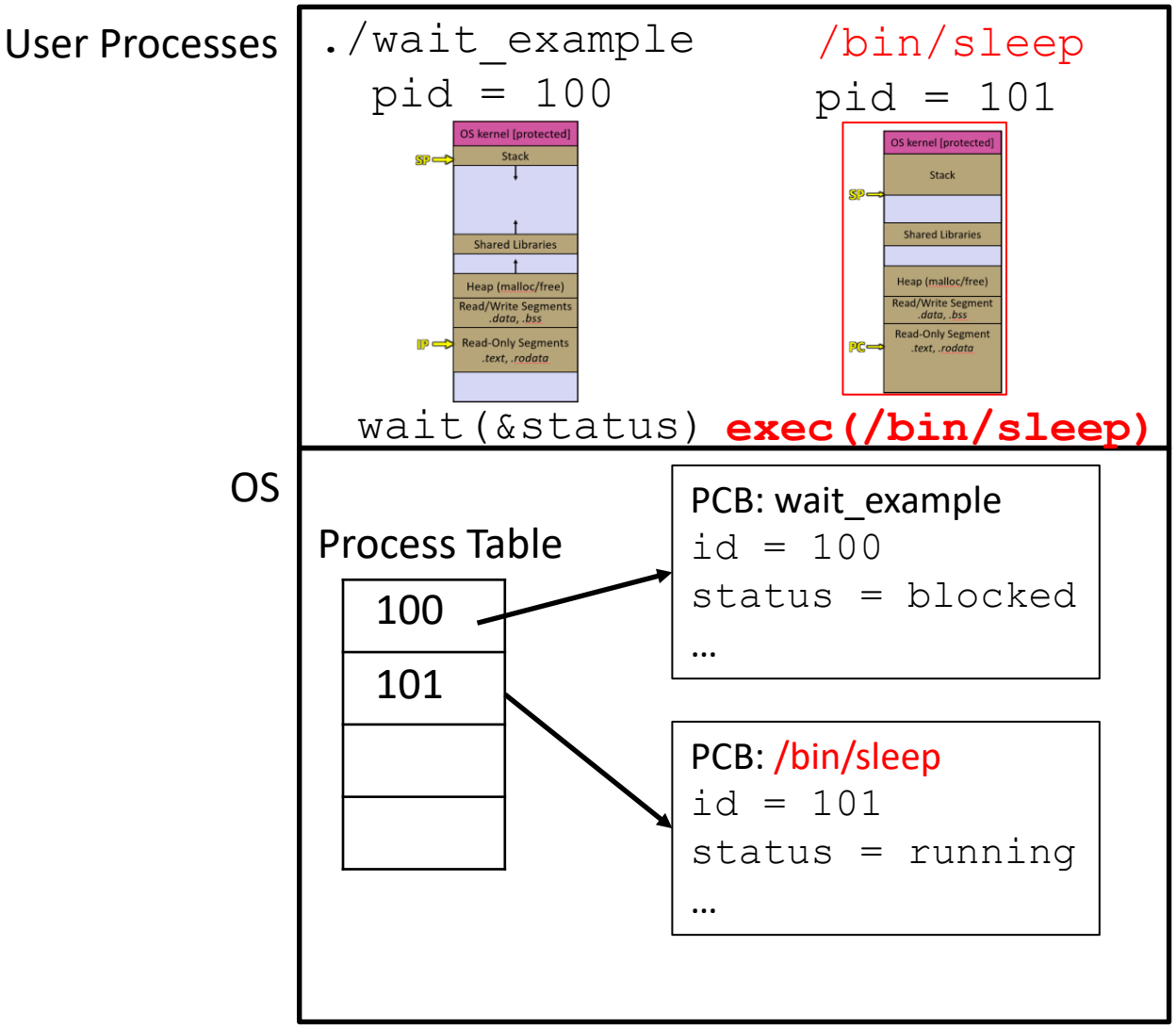


Diagram: wait_example.c

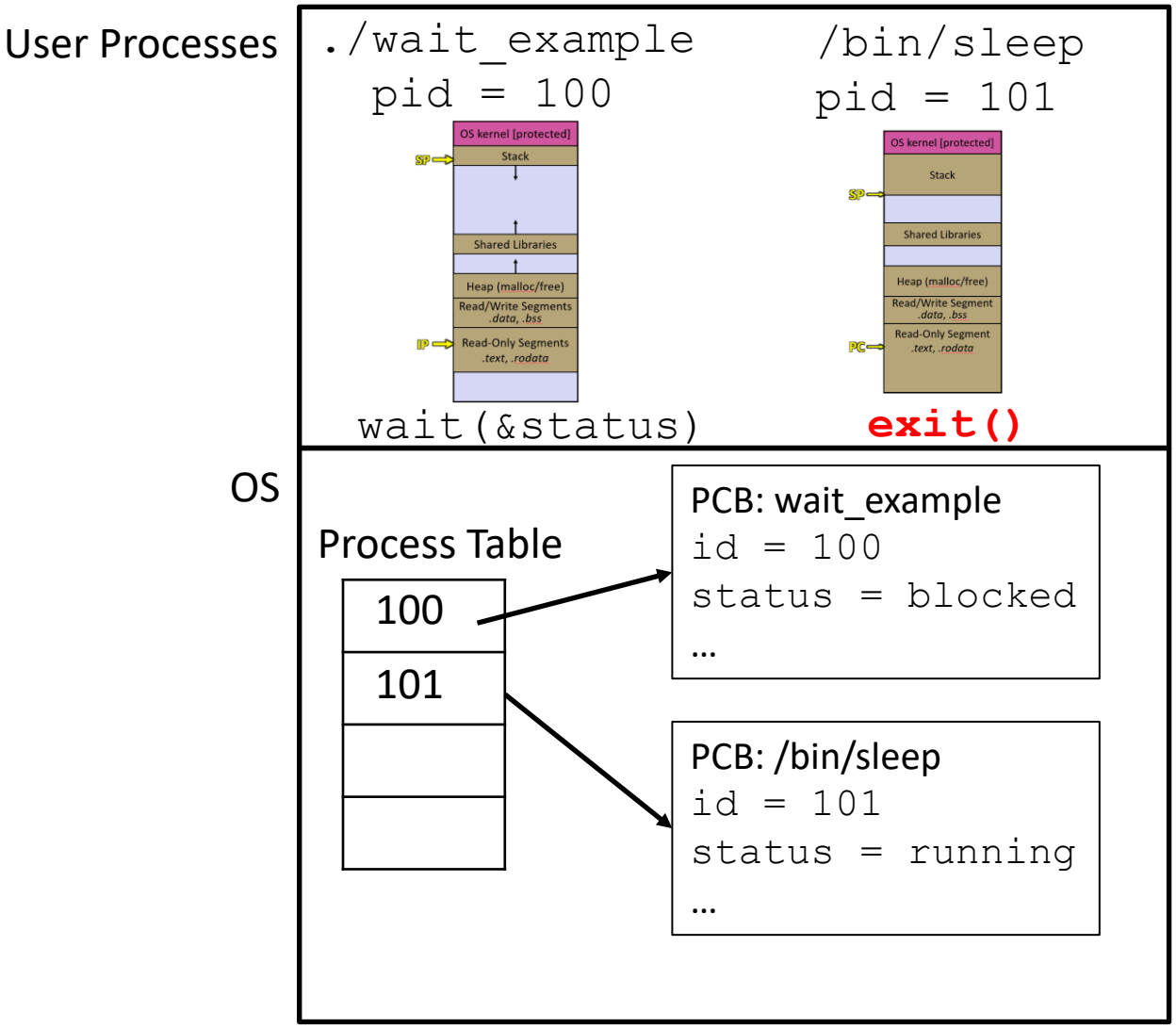


Diagram: wait_example.c

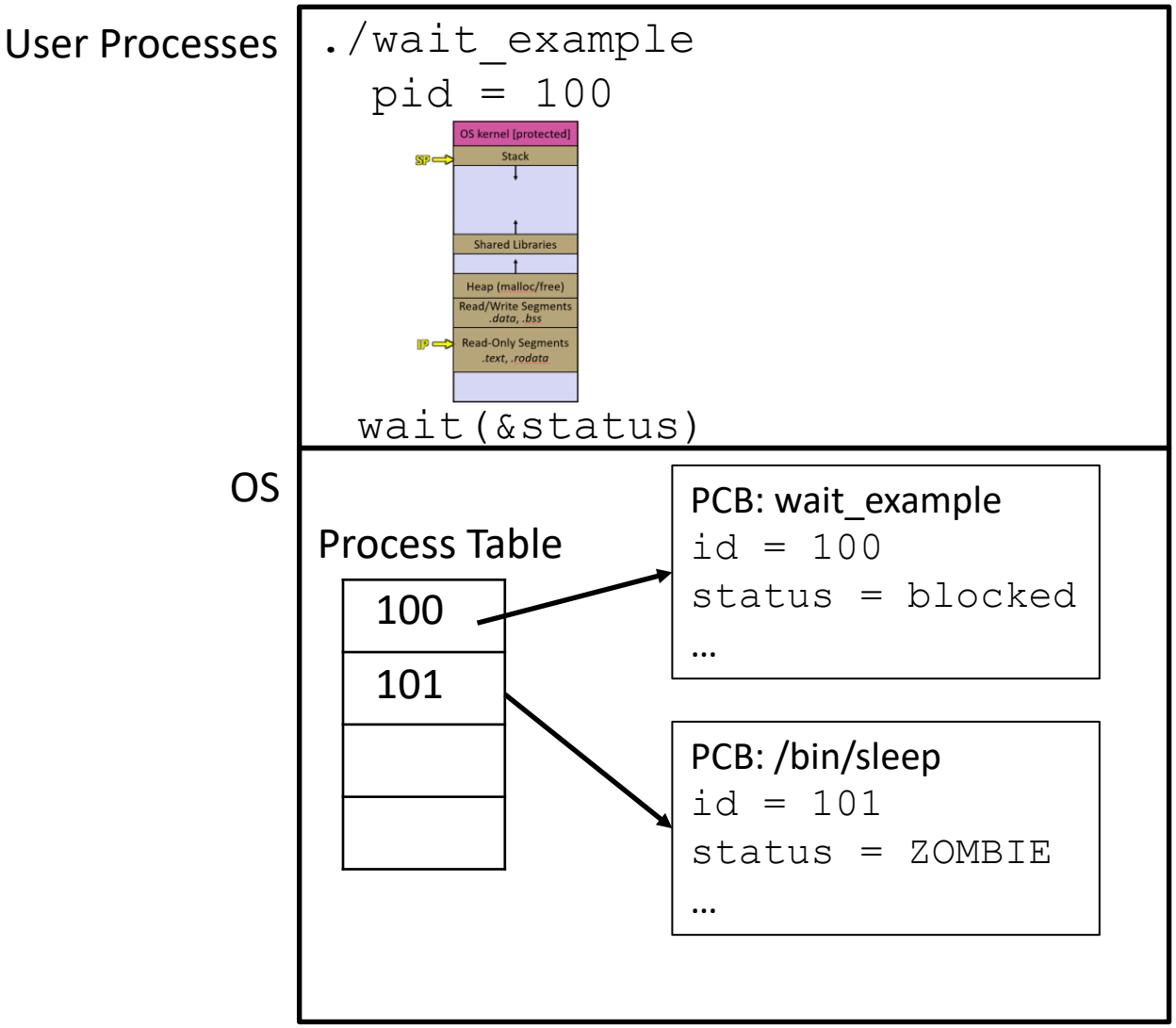


Diagram: wait_example.c

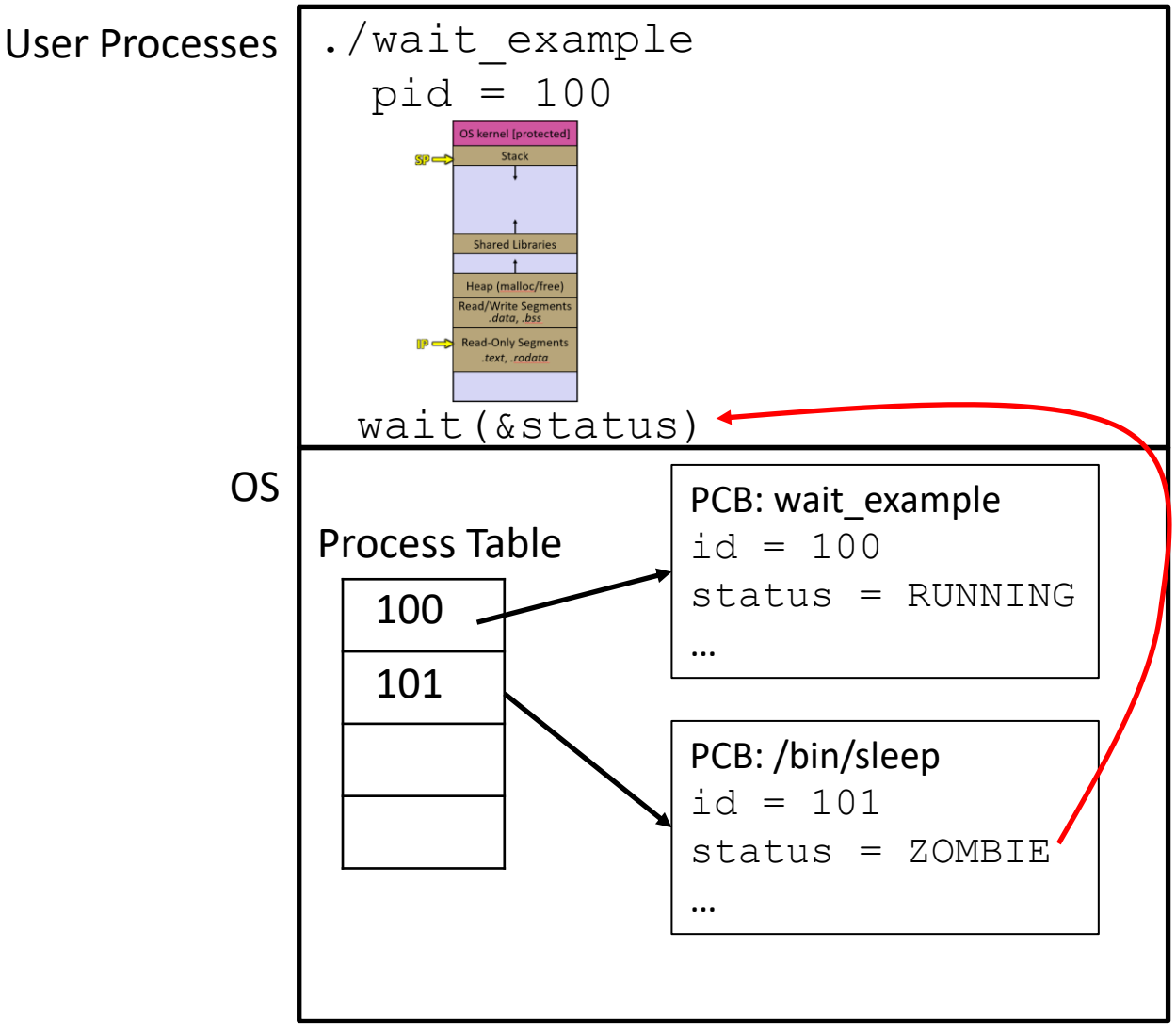


Diagram: wait_example.c

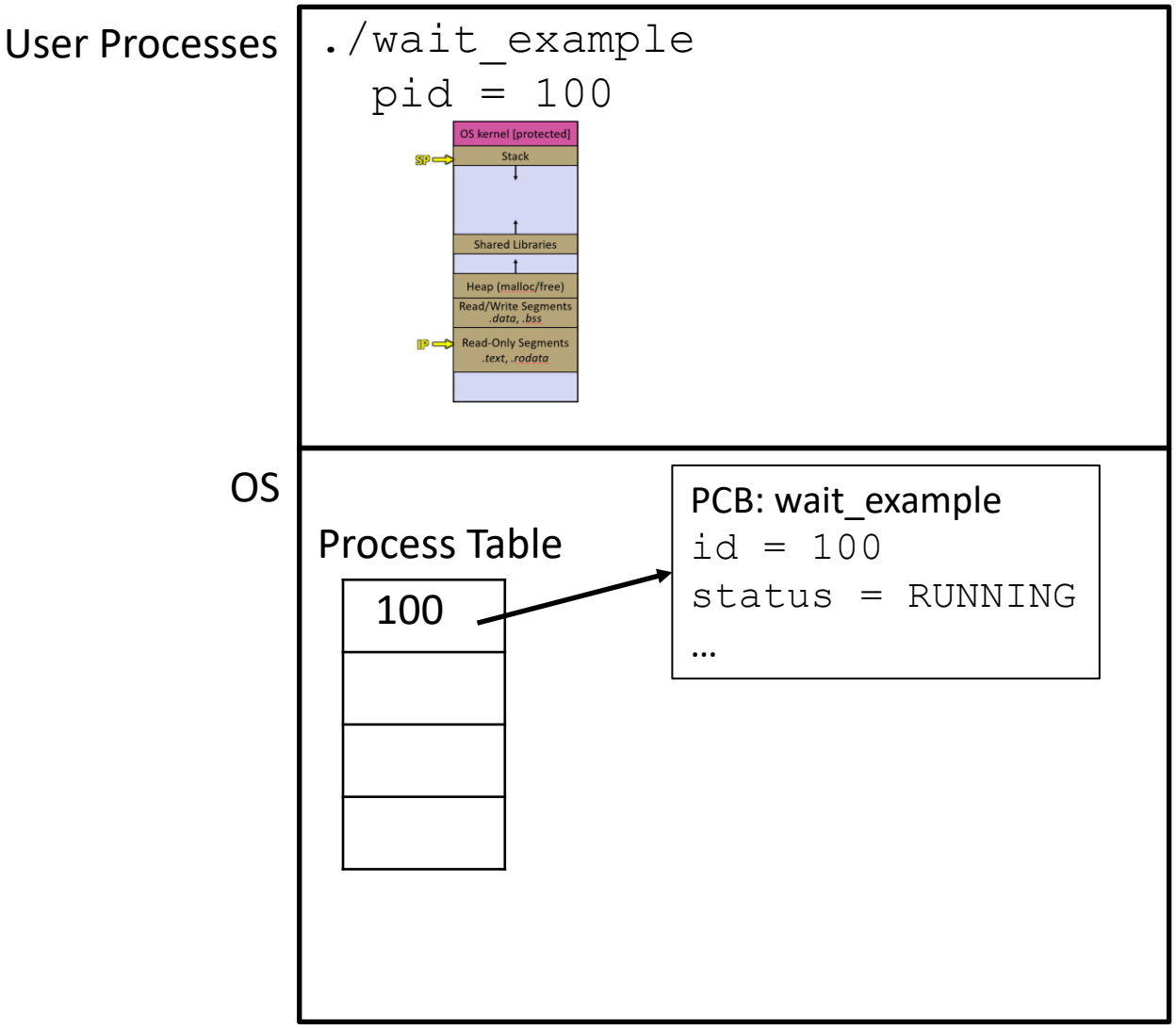


Diagram: wait_example.c

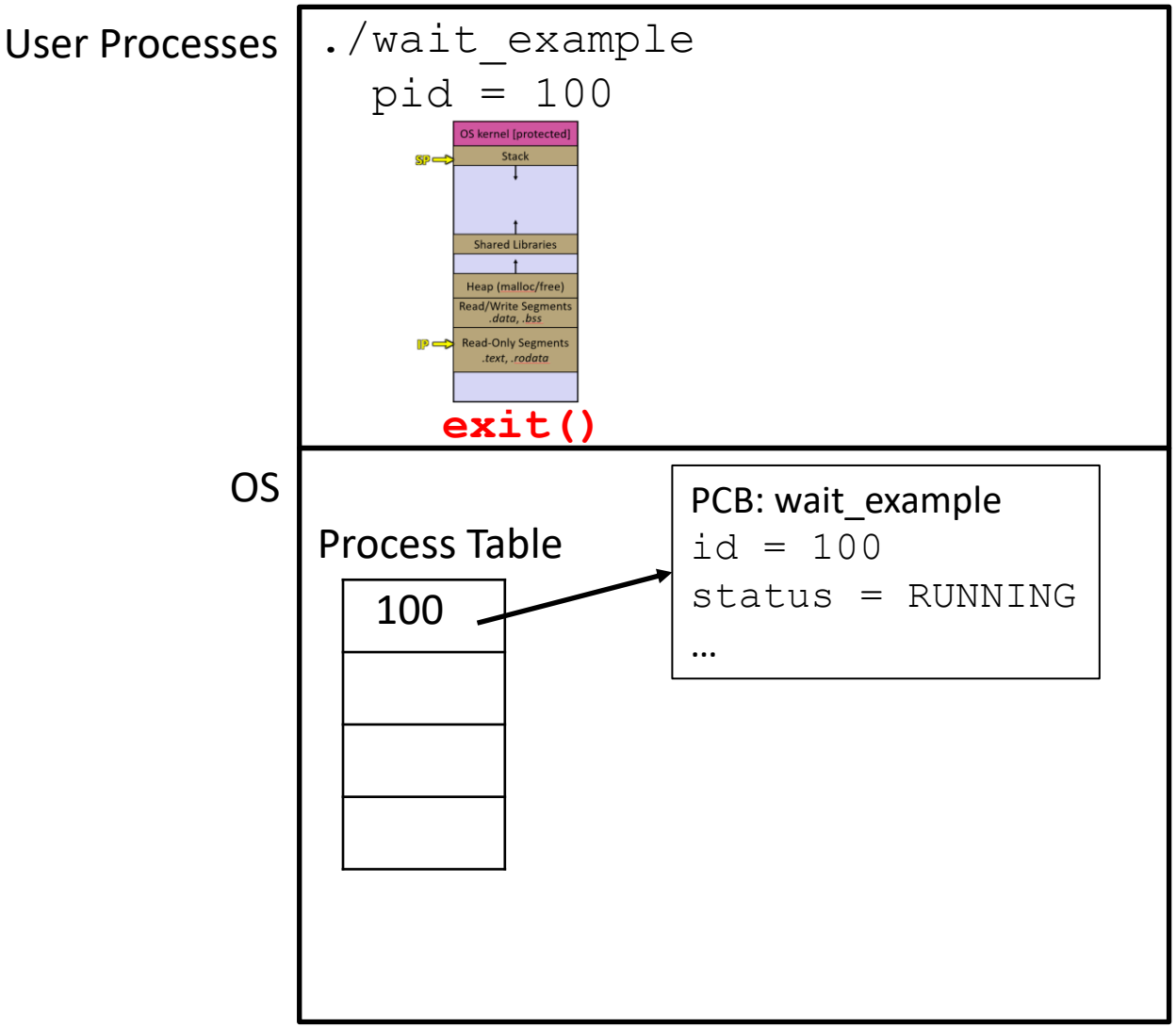
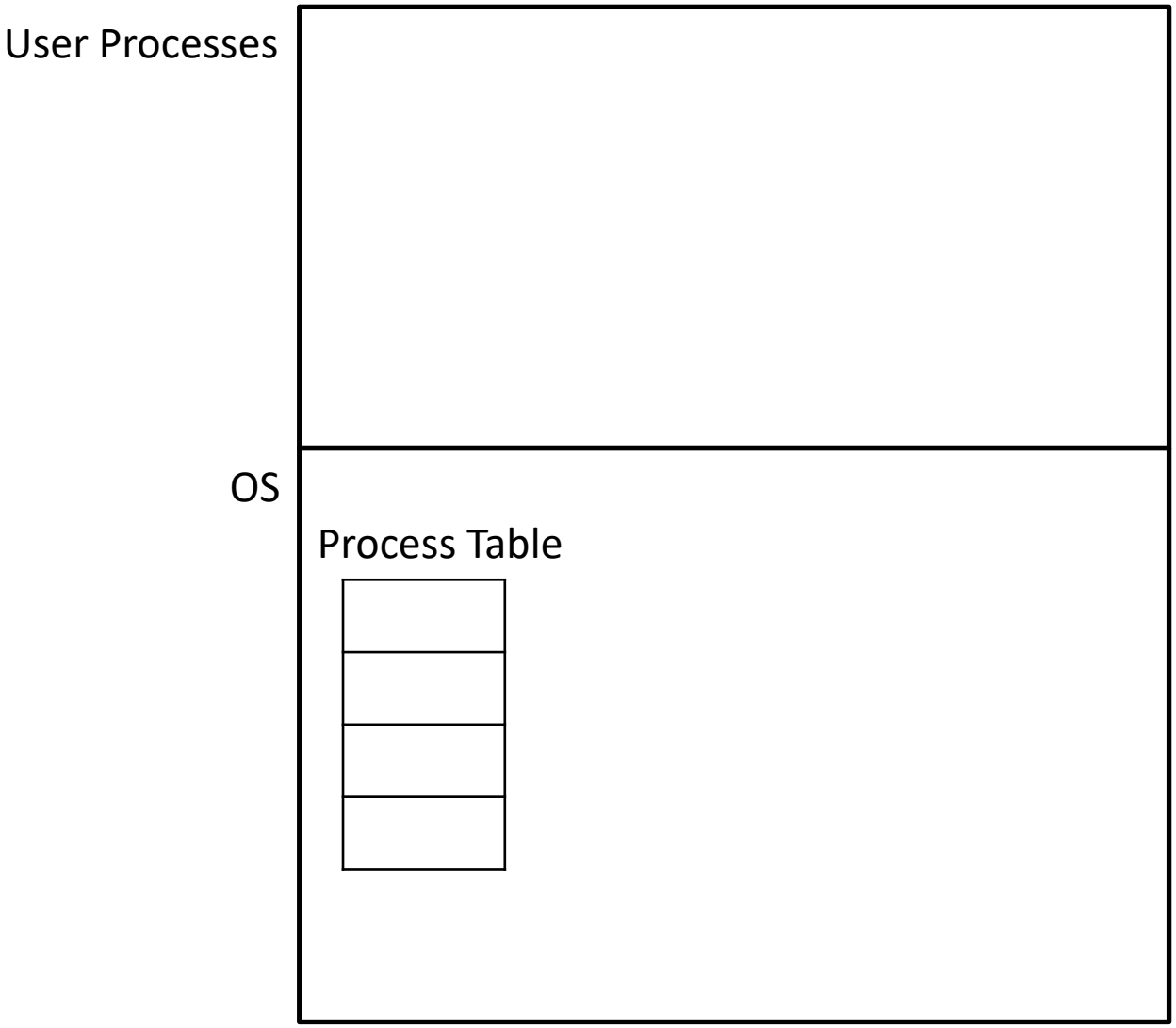


Diagram: wait_example.c



`./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

- `wait()`, blocking, Zombies & PCB
- **`kill()`, `signal()`, `alarm()`**
- Process Groups
- Process Lifetime

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` → Default: terminate the process
  - `SIGALRM`
  - `SIGSEGV` → Default: terminate & core dump

signal()

- You can change how a certain signal is handled
- ```
  sighandler_t signal(int signum,
                      sighandler_t handler);
  ```
- Uses the **sighandler_t** type: a function pointer
  - ```
    typedef void (*sighandler_t)(int);
    ```
- Returns previous handler for that signal
  - **SIG_ERR** when there is an error
- Pass in **SIG_IGN** to ignore the signal
- Pass in **SIG_DFL** for default behaviour
- Some signals like **SIG_KILL** and **SIG_STOP** can't be handled differently

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 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
  ```

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

discuss

```c
// assume this works
void list_push(list* this, float to_push) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) exit(EXIT_FAILURE);
  node->value = to_push;
  node->next = NULL;
  list->tail->next = node;
  list->tail = node;
}

void handler(int signo) {
  list_push(list, NaN);
}

int main(int argc, char* argv[]) {
  signal(SIGINT, handler);
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
}
```

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

- Assume we have implemented a linked list, and it works
- Assume **`list`** is an initialized global linked list

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 happens if the child process calls alarm? … and default handles it?

discuss

- 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

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

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

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 proj1: **`penn-shell`**

Lecture Outline

- `wait()`, blocking, Zombies & PCB
- `kill()`, `signal()`, `alarm()`
- **Process Groups**
- Process Lifetime

Process Groups

- Processes are associated together into Process Groups.
  - A process always is in a process group
- Allows for convenient process & signal management:
  - If ctrl + c (SIGINT) is sent to a process via the keyboard, it is also sent to all processes within its group
- When we create a process with fork(), the child belongs to the same process group as the parent
- Relevant for proj1: **`penn-shell`**

Process Group ID

- The process group ID is equal to a process ID
  - The process ID of the first process to exist in the group
  - If a process group "leader" terminates, can it's process ID be reused by another process? Even if the old group is still going?
  - Answer: no, that process ID will be reserved until the group is done

- ```
  int setpgid(pid_t pid, pid_t pgid);
  ```
- Sets page group id of the specified process to the new value
  - Only works if pgid specifies an existing process group
  - Or if pgid == pid, thus creating a new process group of that id
  ```

Process Groups: utility

- Can pass in **`-PGID`** (negative PGID) to **`kill()`** and **`waitpid()`**
- Doing so for **`kill()`** will send the signal to all processes in the group
- Doing so for **`waitpid()`** will wait for any process in the group
- You may find this useful for proj1: **`penn-shell`**

Lecture Outline

- `wait()`, blocking, Zombies & PCB
- `kill()`, `signal()`, `alarm()`
- Asynch `wait`
- Process Groups
- **Process Lifetime**

Process State Lifetime