Processes (cont.)

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW05 posted after class
  - Should be a pretty short assignment, just a “hands on” for the git stuff so that you aren’t as scared when you see it again :)
  - Due tomorrow night at midnight
- Check-in due before class
  - Re-opens processed during class
- HW06 posted after class Wednesday
  - Not everything you need will be talked about till Wednesday
  - Will have extended deadline due to fall break

Lecture Outline

- **fork() & virtual memory**
- exec
- wait
- ordering

Creating New Processes

- `pid_t fork();`
  - Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
    - *almost everything
  - The new process has a separate virtual address space from the parent
  - Returns a `pid_t` which is an integer type.

Parent vs Child

- After a successful call to fork() there are two processes
  - The original calling process (the “Parent” in this relationship)
  - The newly created clone (the “child” in this relationship)

- The two processes have different return values from fork() and different process ID’s

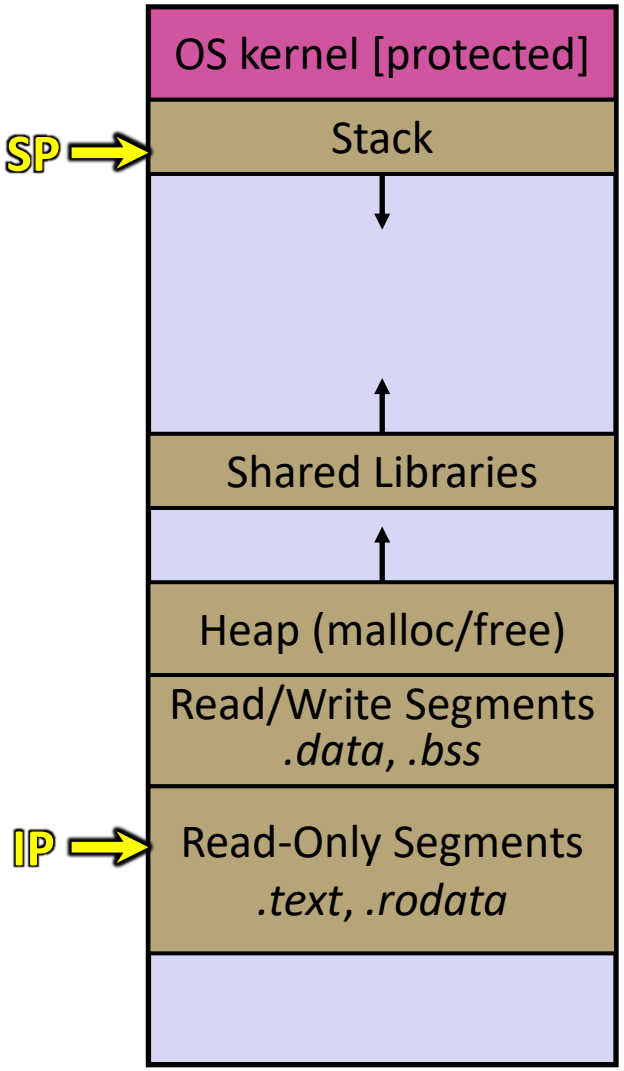
- A process can have any number of children, but only one child.

- Some operations done on a child can only be done by its parent.

Definition: Process

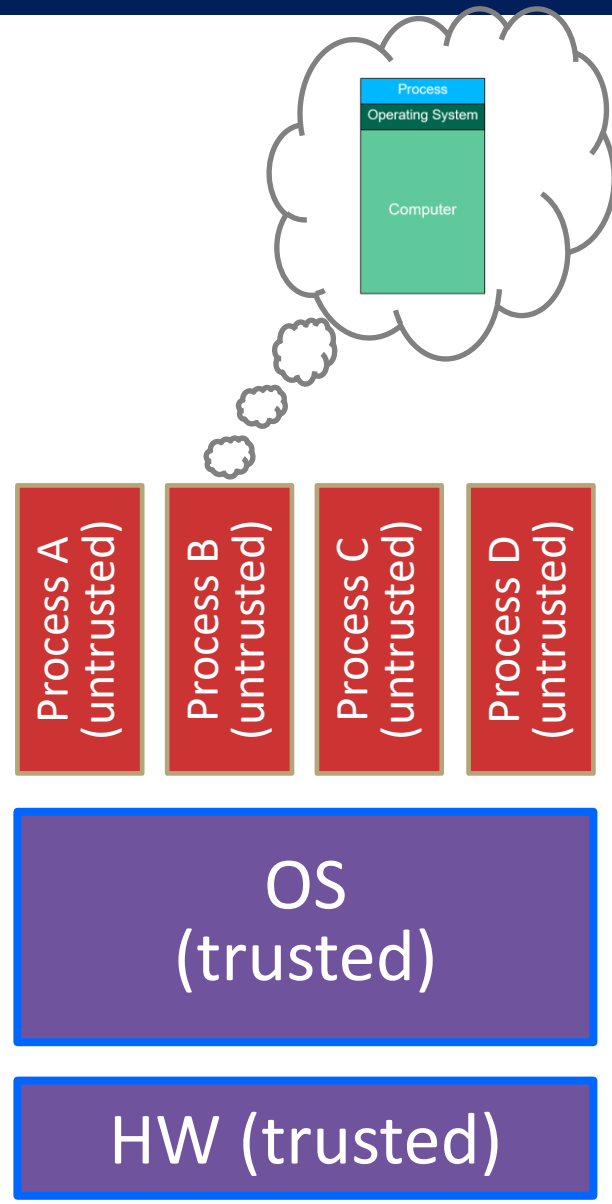
- Definition: An instance of a program that is being executed (or is ready for execution)

- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources



OS: Protection System

- OS isolates process from each other
  - Each process seems to have exclusive use of memory and the processor.
    - This is an **illusion**
    - More on Memory when we talk about virtual memory later in the course
  - OS permits controlled sharing between processes
    - E.g. through files, the network, etc.
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly



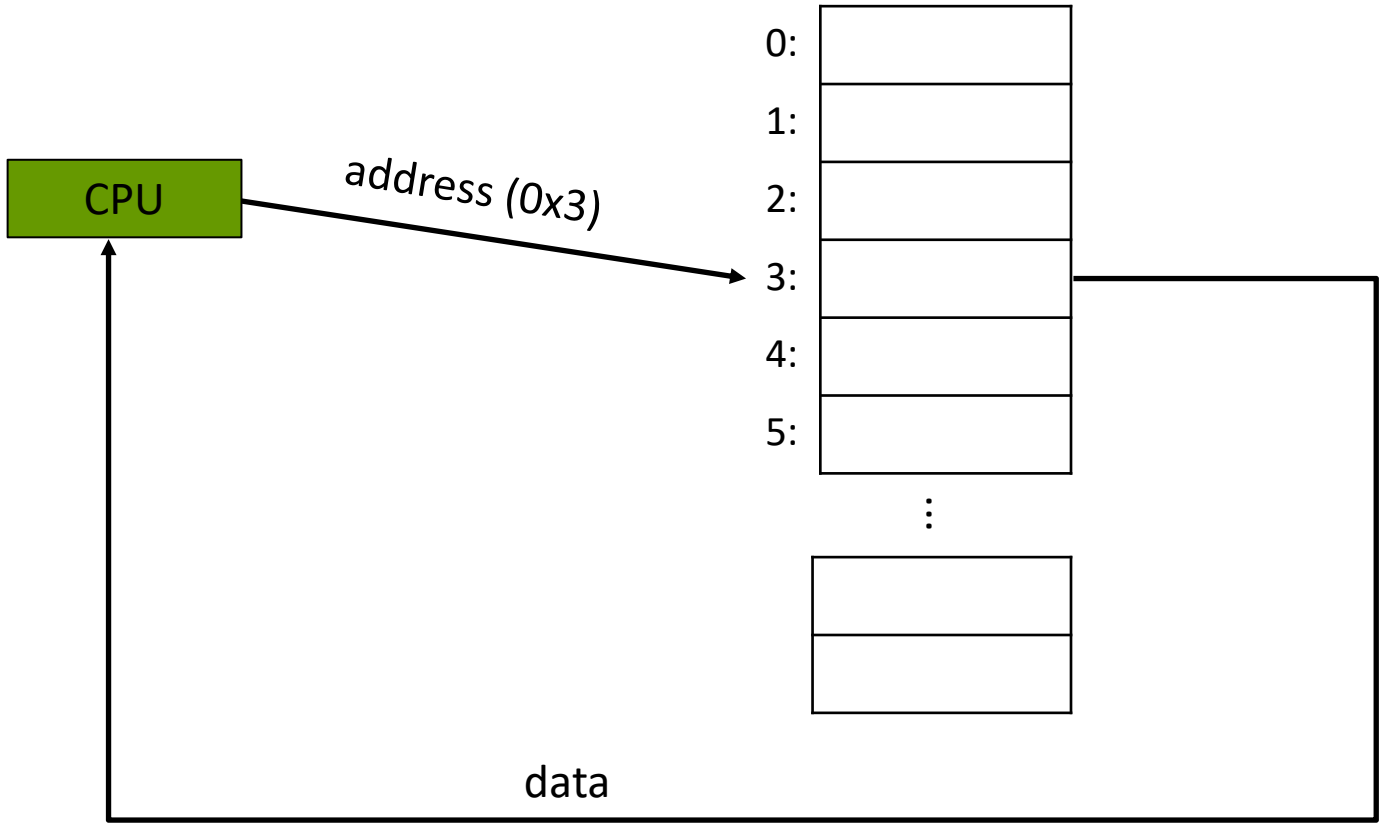
Demo: vmem.cpp

- If processes have separate memory? What happens to points when we fork()?

```
int main() {
  int* ptr = new int(0);
  pid_t pid = fork();

  if (pid == 0) {
    *ptr += 20;
    cout << "I am the child!\n";
    cout << "Address in ptr is: " << ptr << "\n";
    cout << "value at *ptr is: " << *ptr << endl;
    delete ptr;
  } else {
    *ptr += 15;
    cout << "I am the parent!\n";
    cout << "Address in ptr is: " << ptr << "\n";
    cout << "value at *ptr is: " << *ptr << endl;
    delete ptr;
  }
}
```

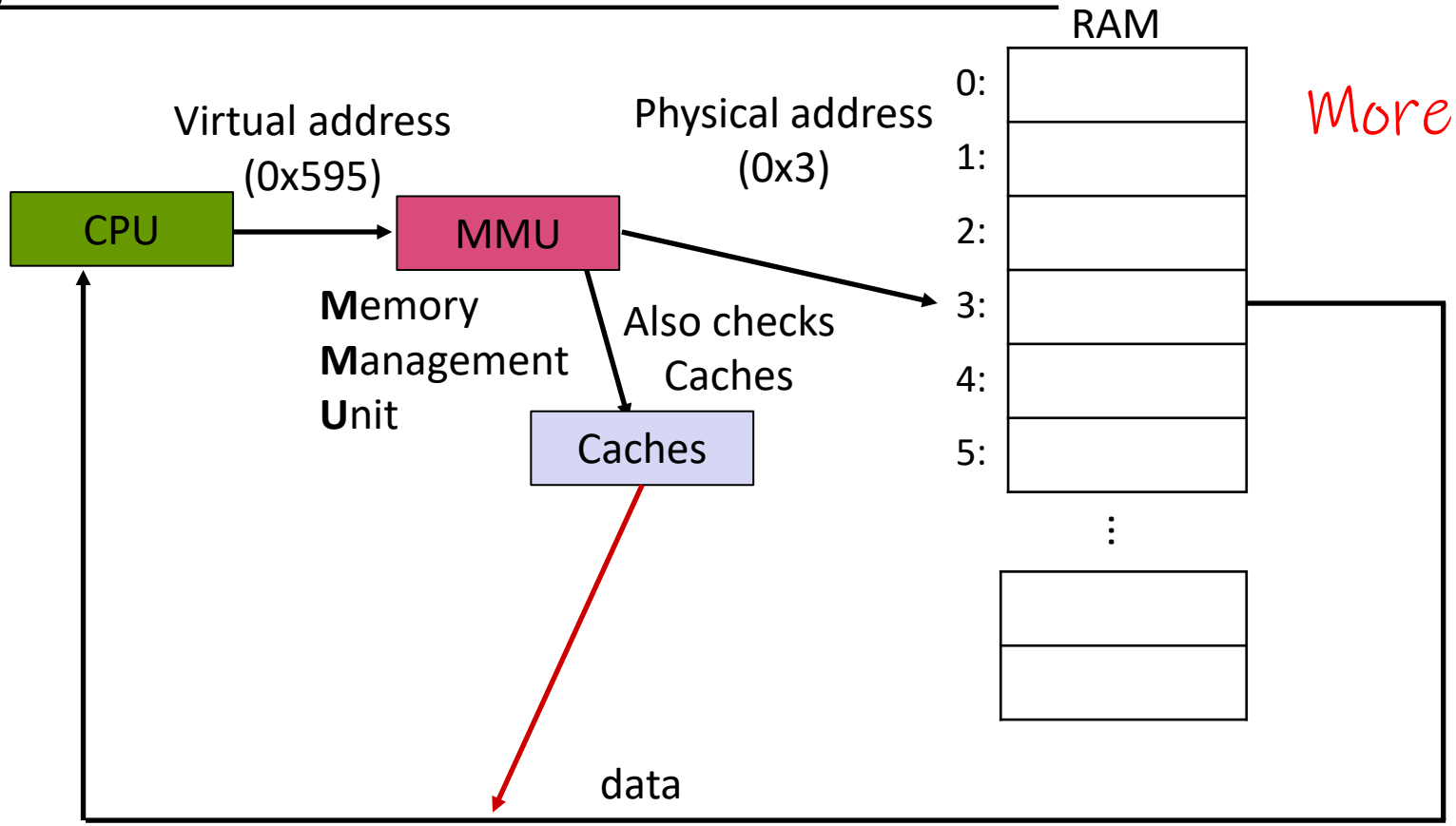
Memory (as we know it from CIS 2400)

- The CPU directly uses an address to access a location in memory



Virtual Address Translation

- Programs don't know about many of things going on under the hood with memory. They send an address to the MMU, and the MMU will help get the data. **Programs don't work with "real" addresses**.



More in 4480/5480

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 the parent and child immediately after fork() is their process id and the return value from fork() each process gets

Lecture Outline

- fork() & virtual memory
- **exec**
- wait
- ordering

execvp()

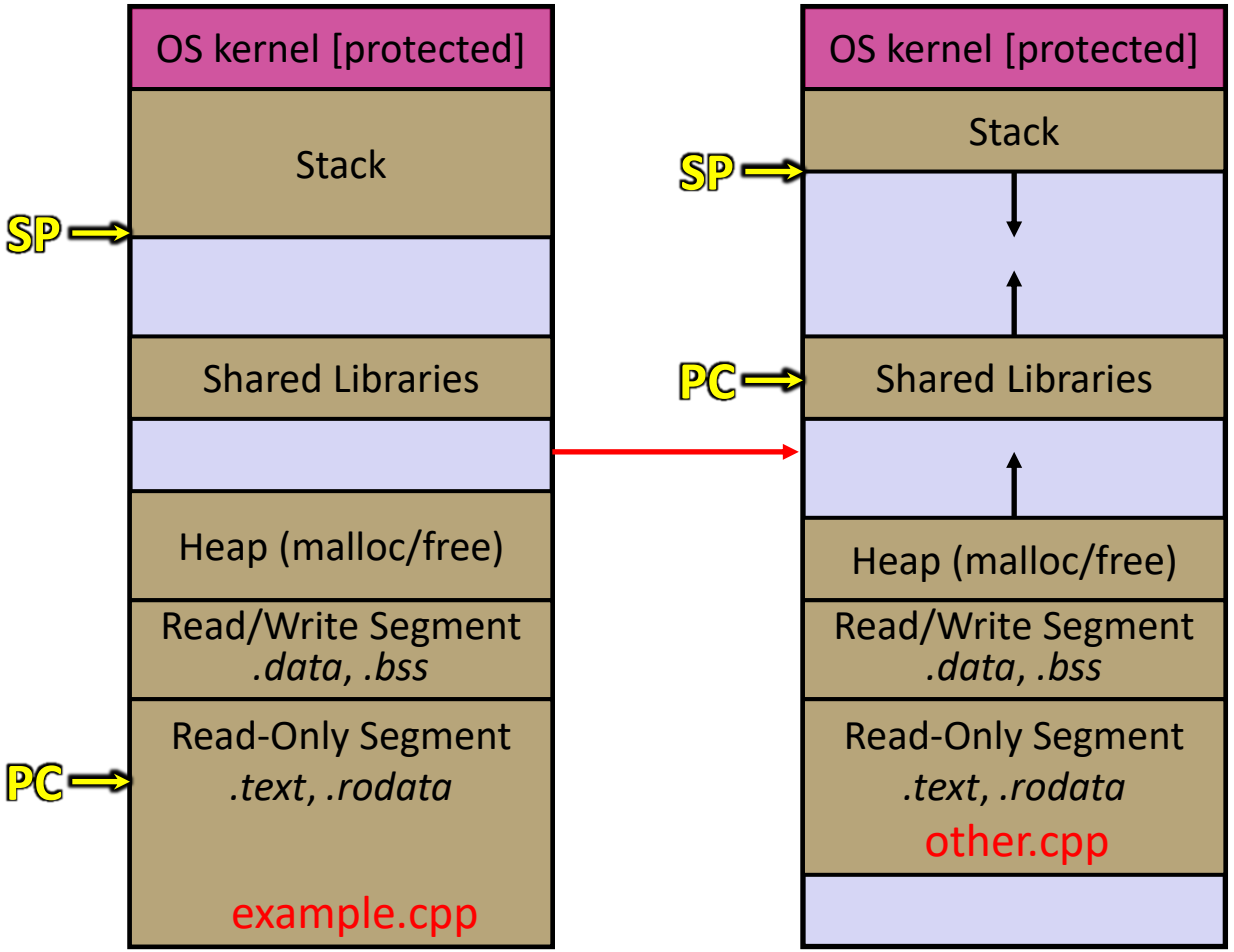
- execvp

```
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/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

const and execv*

- execvp (and the family of exec* functions) are written in C for POSIX
  - They take in C arrays
  - They take in a `char *const argv[]`.

- Often times we want to use a string() to form argv:
  - data() gets a pointer to underlying array/c-string?
  - Why doesn't this code work?

```
// str is a command to execute
void execute(const string& str) {
  vector<char*> argv;
  argv.push_back(str.data());
  argv.push_back(nullptr);

  execvp(argv.at(0), argv.data());
}
```

string::data returns a const char*

const_cast and execv*

- execvp (and the family of exec* functions) are written in C for POSIX
  - They take in C arrays
  - They take in a `char *const argv[]`.

- Const cast can be used to strip const. You should almost never do it. This is the only time I know offhand where const_cast should be used.

- Execvp promises to not change argv at all, but it is not marked const all the way since it would make correct C code give warnings when you compile it. "Backwards compatibility"

```
// str is a command to execute
void execute(const string& str) {
  vector<char*> argv;
  argv.push_back(const_cast<char*>(str.data()));
  argv.push_back(nullptr);

  execvp(argv.at(0), argv.data());
}
```

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens to allocated memory when we call exec?
    - Valgrind command if you want to check children:
      `valgrind --trace-children=yes`

- Show how this is the same as running the command directly in the shell

Ed Discussion

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

```
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);
  }

  cout << ":)" << endl;

  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;
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

Ed Discussion

- What are all the valid prints that this code could have?
- Assume functions don't fail

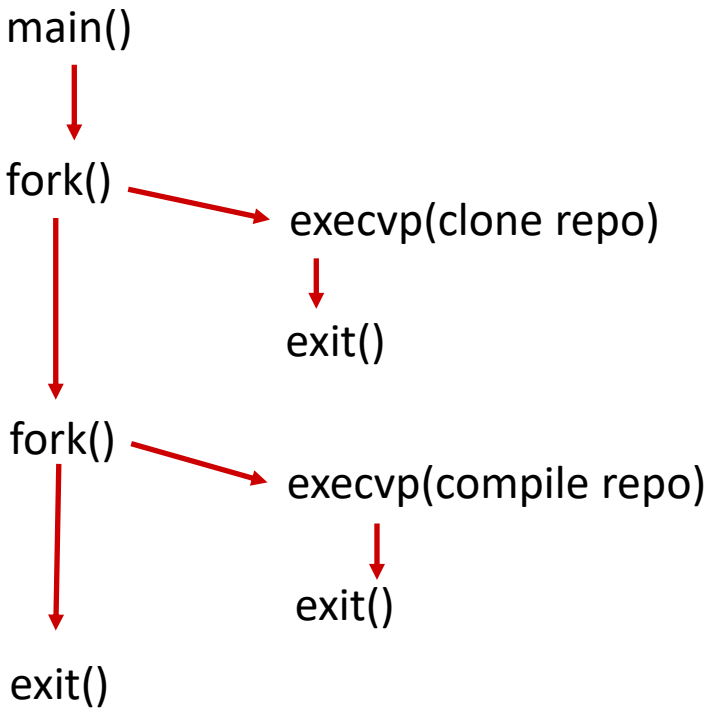
```
void weirdprint(const string& to_print) {
  vector<char*> argv;
  argv.push_backK(const_cast<char*>("echo"));
  argv.push_back(const_cast<char*>(to_print.data()));
  argv.push_back(nullptr);
  execvp(argv.at(0), argv.data());
}
```

```
int main() {
  pid_t cpid = fork();
  weirdprint("fork");


  if (cpid == 0) {
    cpid = fork();
    if (cpid == 0) {
      weirdprint("grandchild");
    } else {
      weirdprint("child");
    }
    weirdprint("exiting");
    exit(EXIT_SUCCESS);
  }
  weirdprint("parent");
}
```

Ed Discussion

- 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
    array<const char*, 4> argv[] = {"git", "clone", "repo_name.git", nullptr};
    execvp(argv[0], const_cast<char**>(argv.data()));
  }

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



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

Lecture Outline

- fork() & virtual memory
- exec
- **wait**
- ordering

From a previous poll:

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

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

  pid = fork();
  if (pid == 0) {
    // we are the child
    array<const char*, 4> argv = {"make", "-C",
                                  "repo_name", nullptr};
    execvp(argv[0], const_cast<char**>(argv.data()));
  }

  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

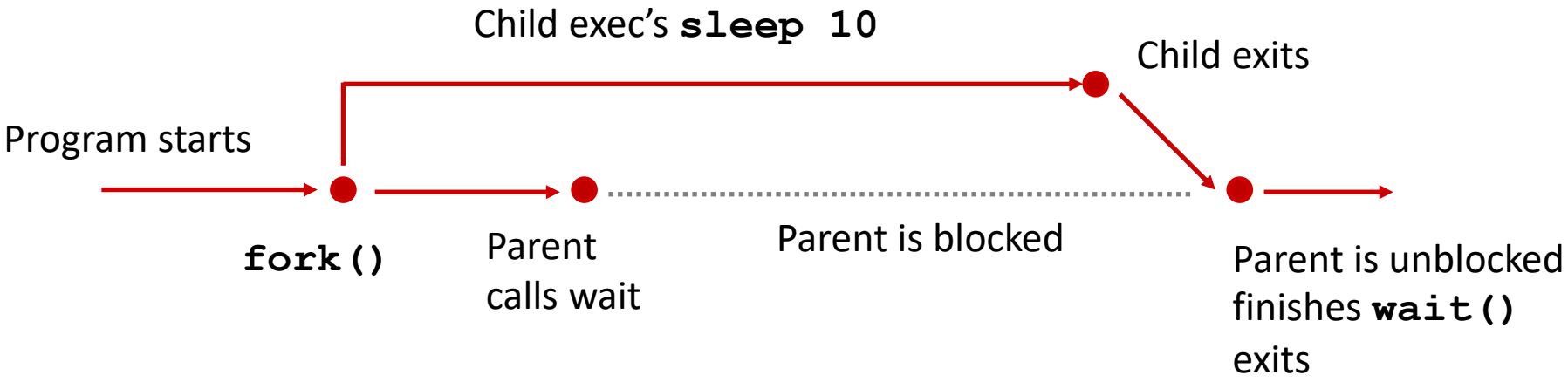
Fixed code from poll

```
int main(int argc, char* argv[]) {
  // fork a process to exec clang
  pid_t clone_ pid = fork();

  if (clone_pid == 0) {
    // we are the child
    array<const char*, 4> argv[] = {"git", "clone", "repo_name.git", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  waitpid(clone_pid, nullptr, 0); // should error check, not enough slide space :(

  // fork to run the compiled program
  pid_t make_pid = fork();
  if (make_pid == 0) {
    // the process created by fork
    array<const char*, 4> argv = {"make", "-C", "repo_name", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

Demo: `wait_example`

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

- Execution timeline:



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.cpp

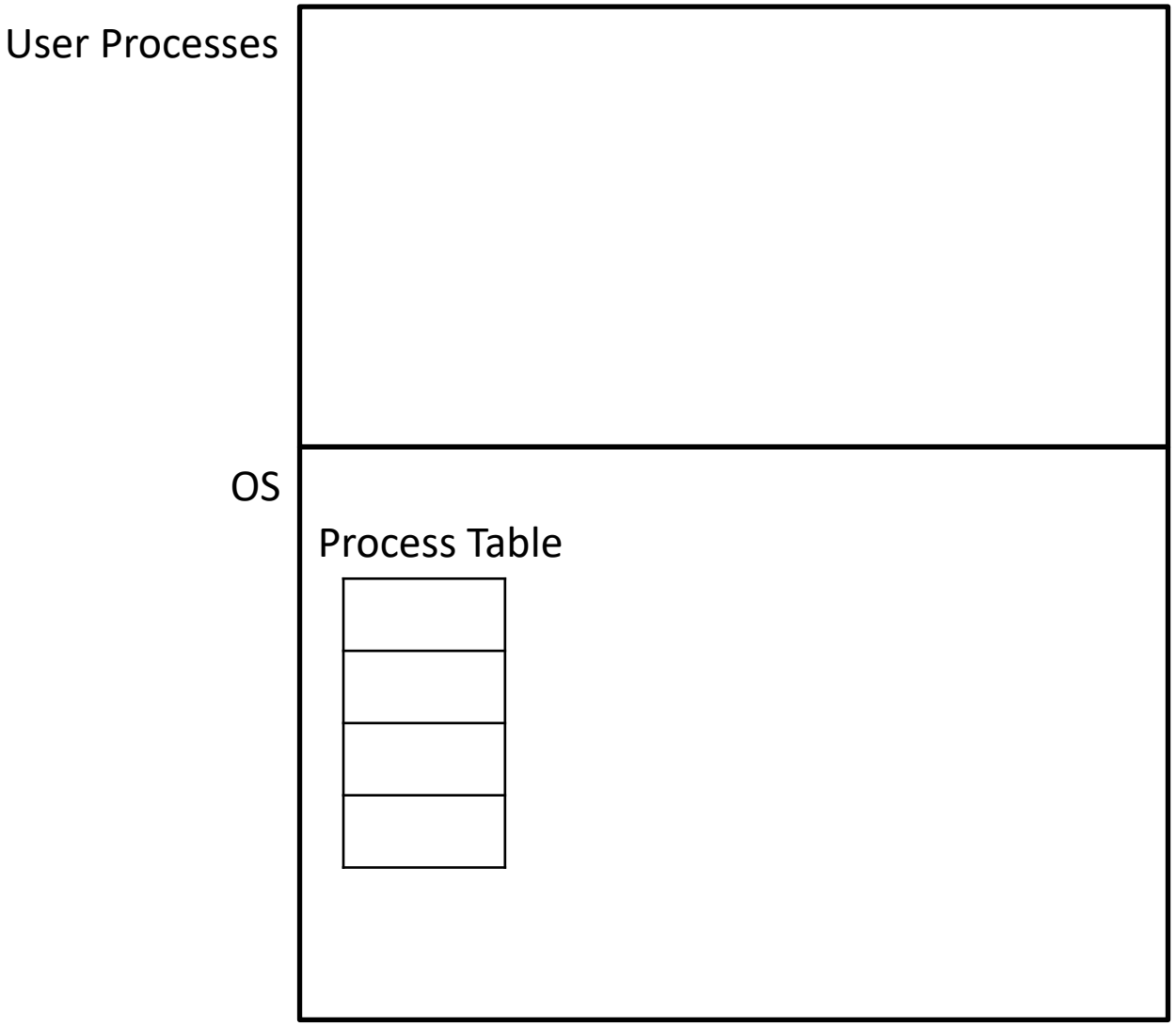


Diagram: wait_example.cpp

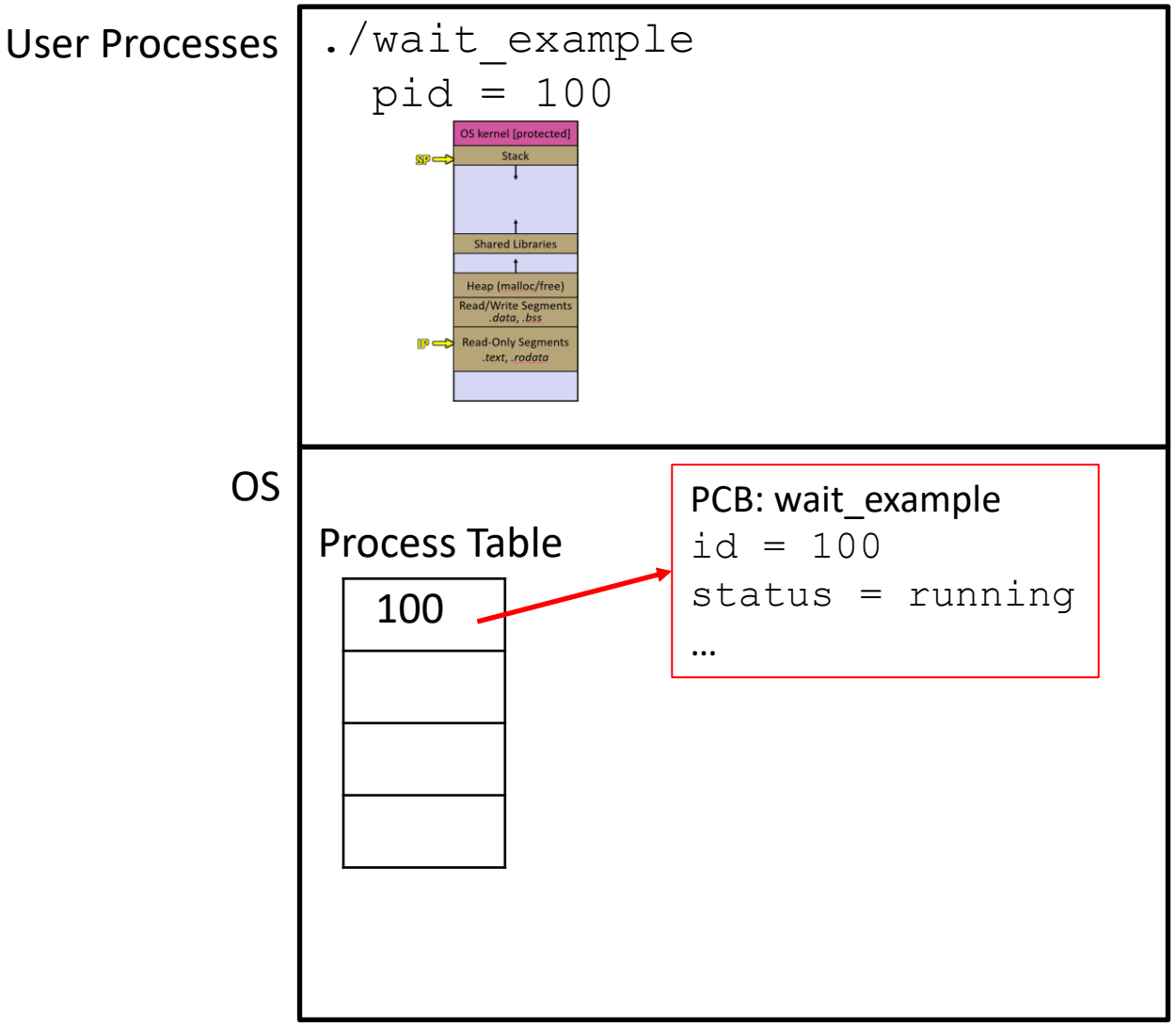


Diagram: wait_example.cpp

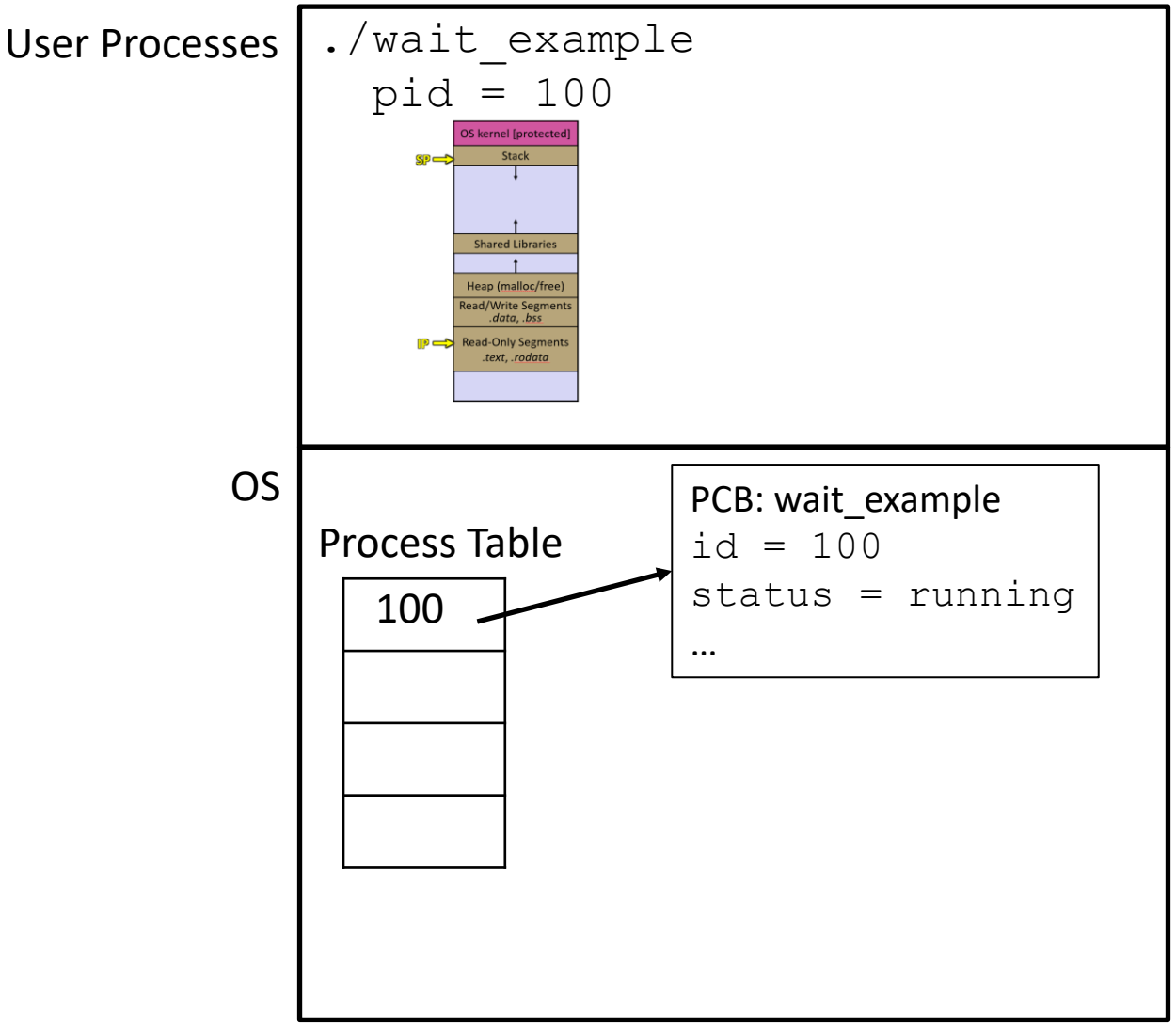


Diagram: wait_example.cpp

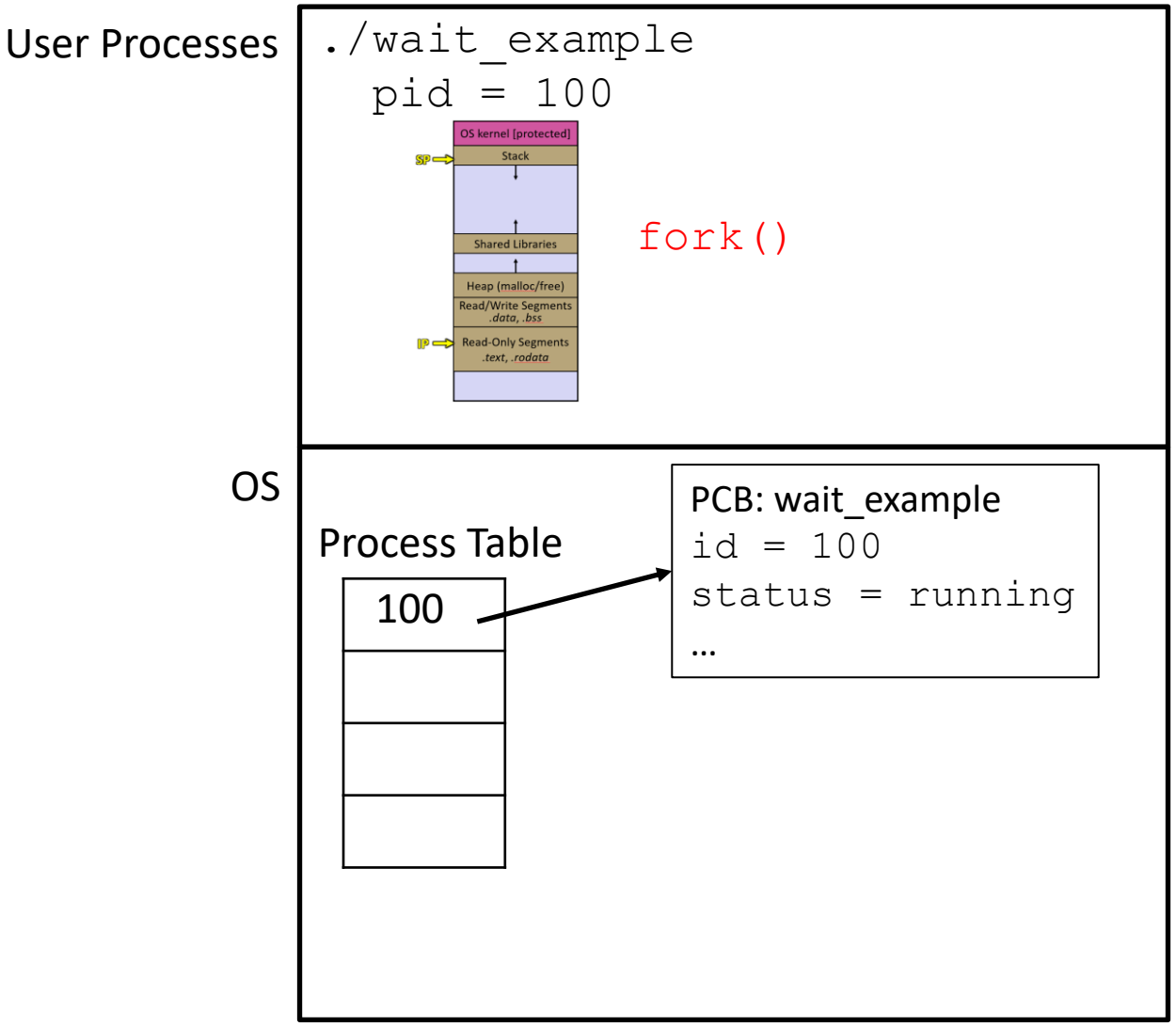


Diagram: wait_example.cpp

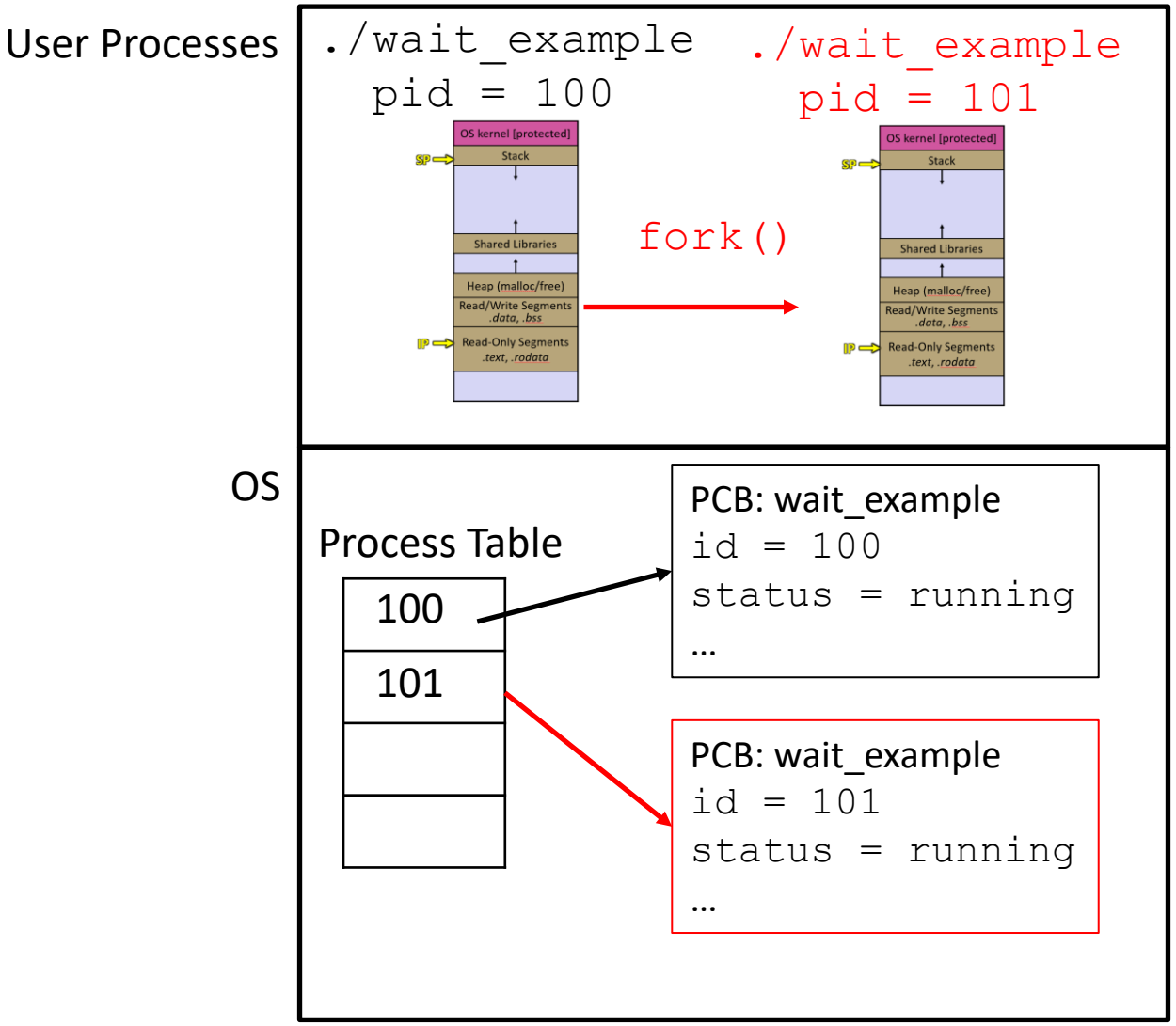


Diagram: wait_example.cpp

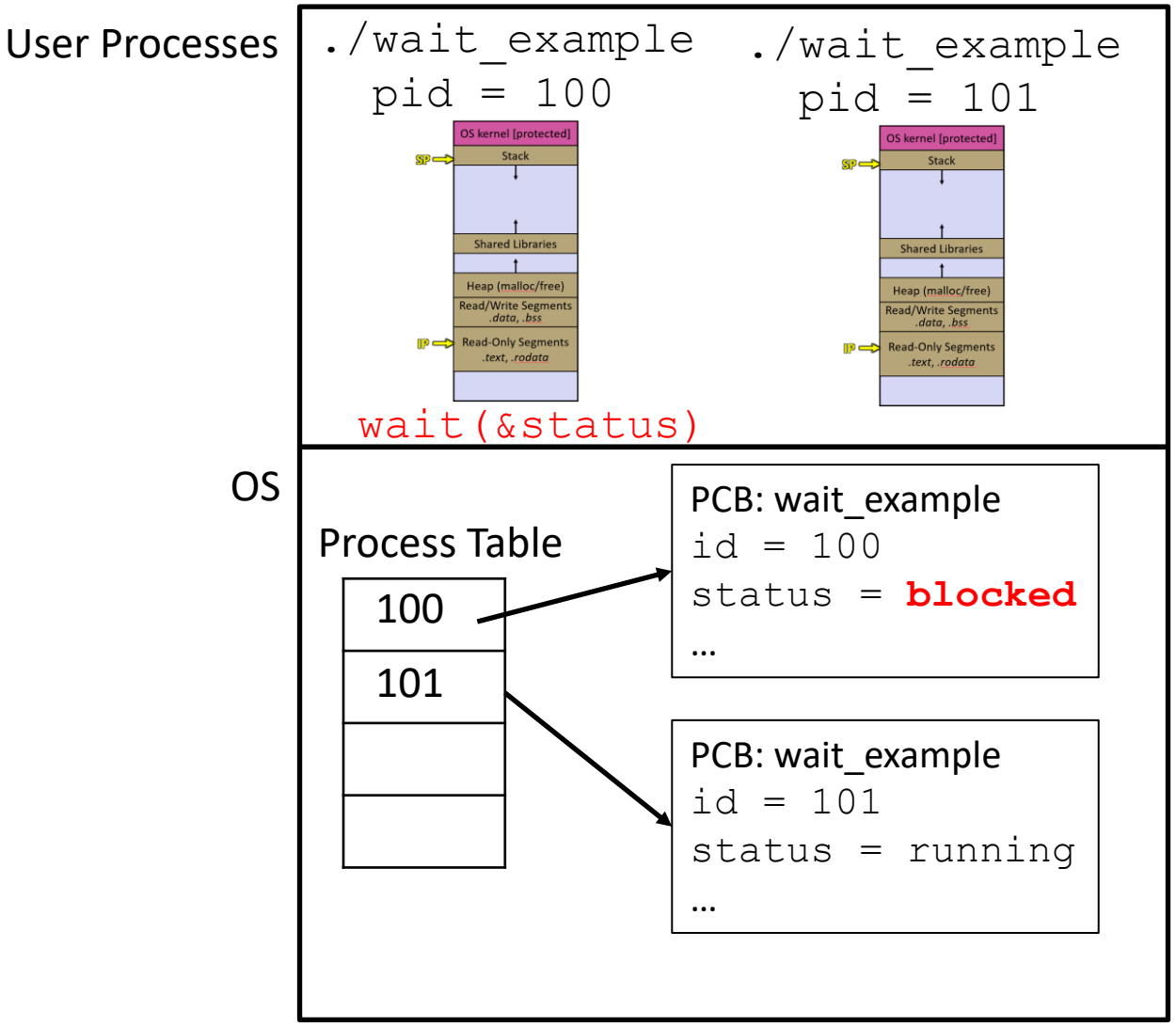


Diagram: wait_example.cpp

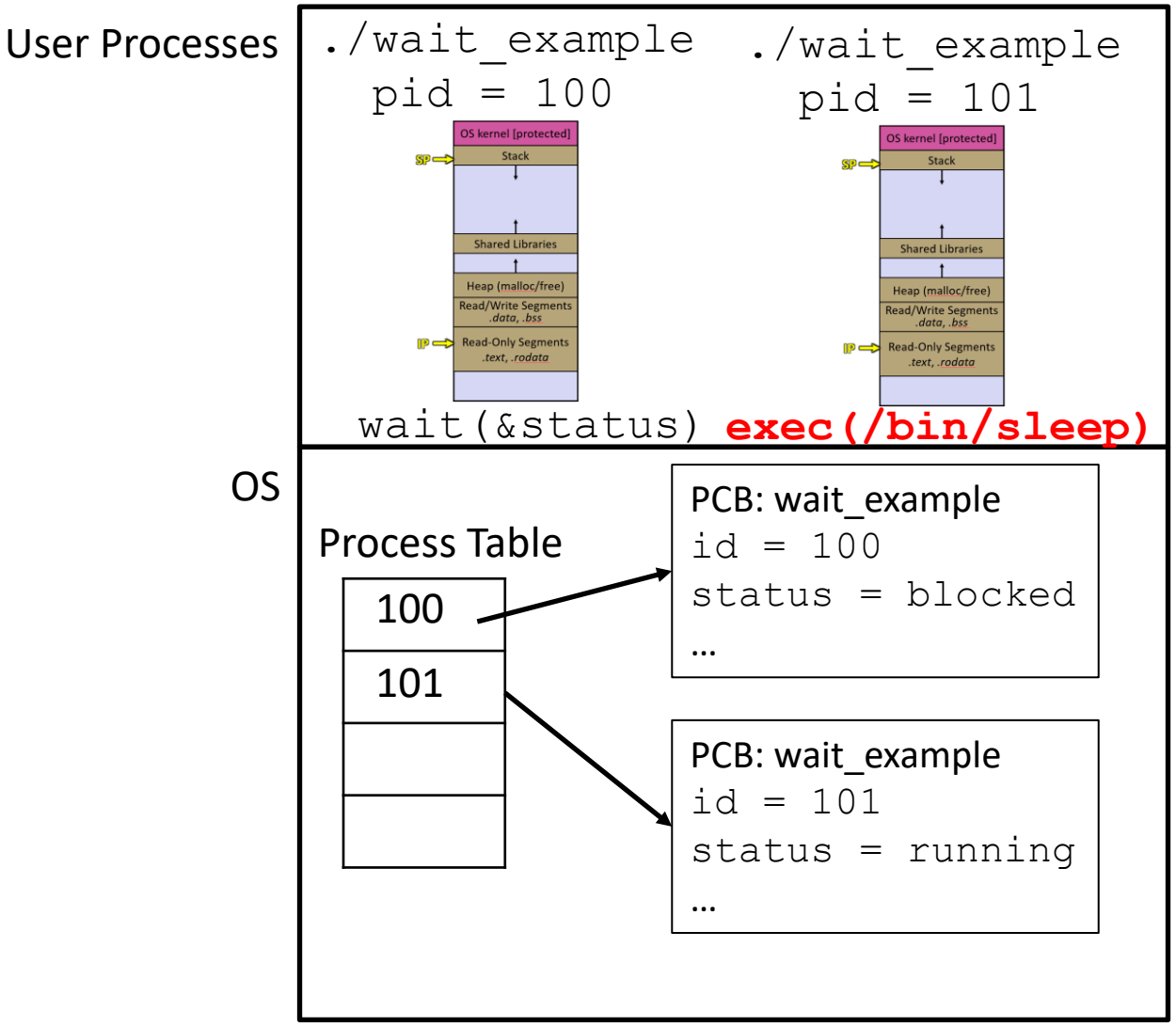


Diagram: wait_example.cpp

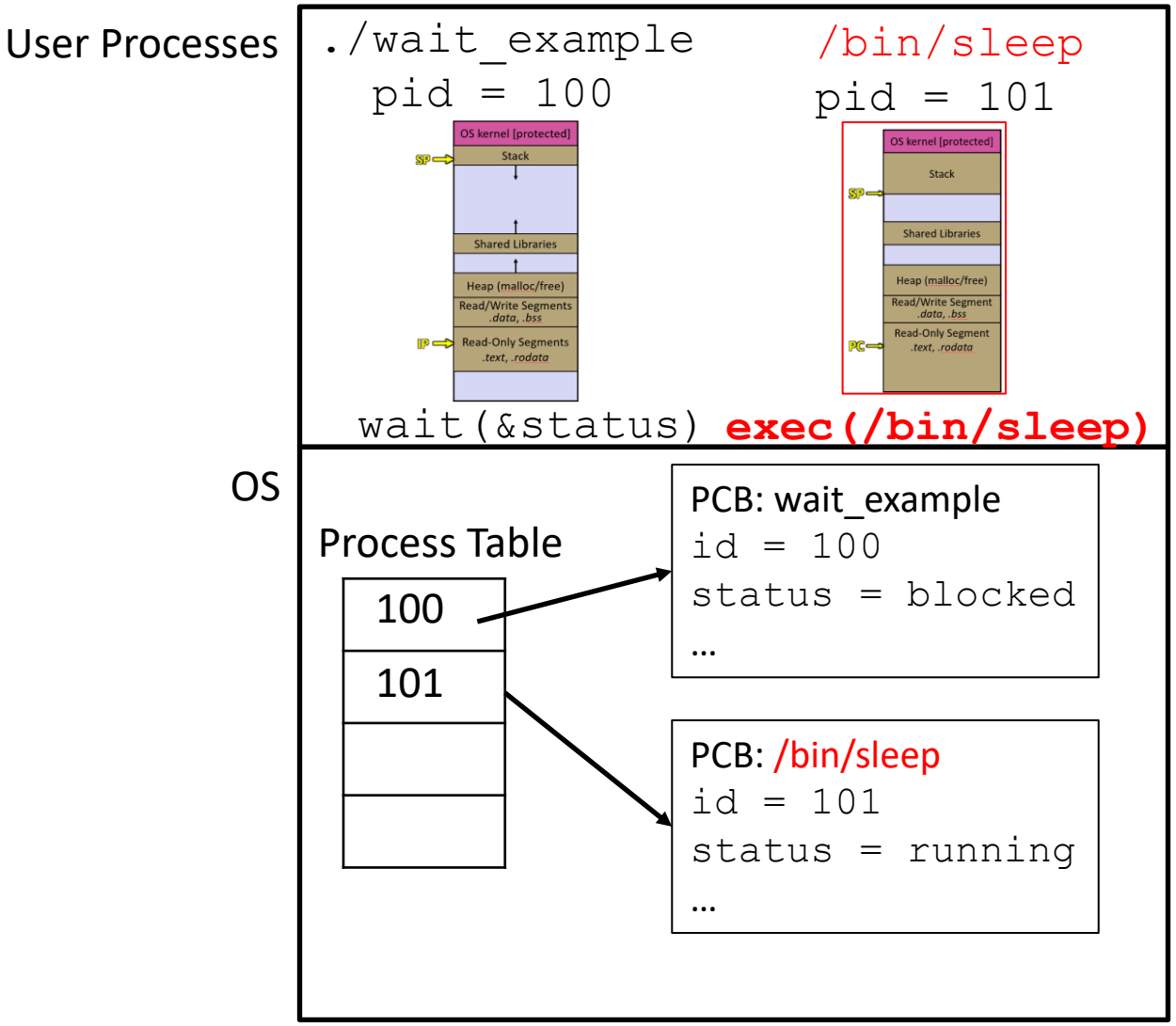


Diagram: wait_example.cpp

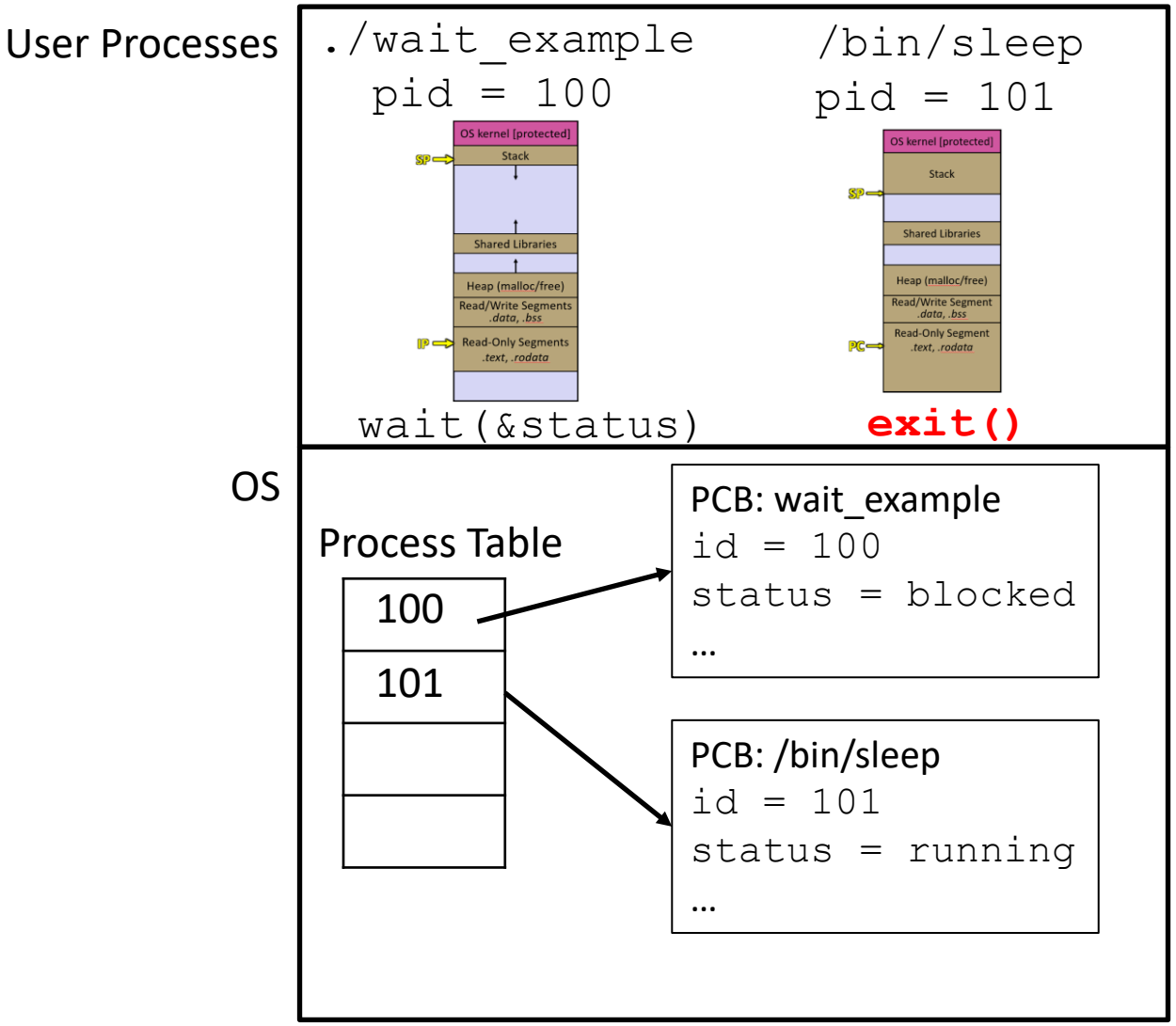


Diagram: wait_example.cpp

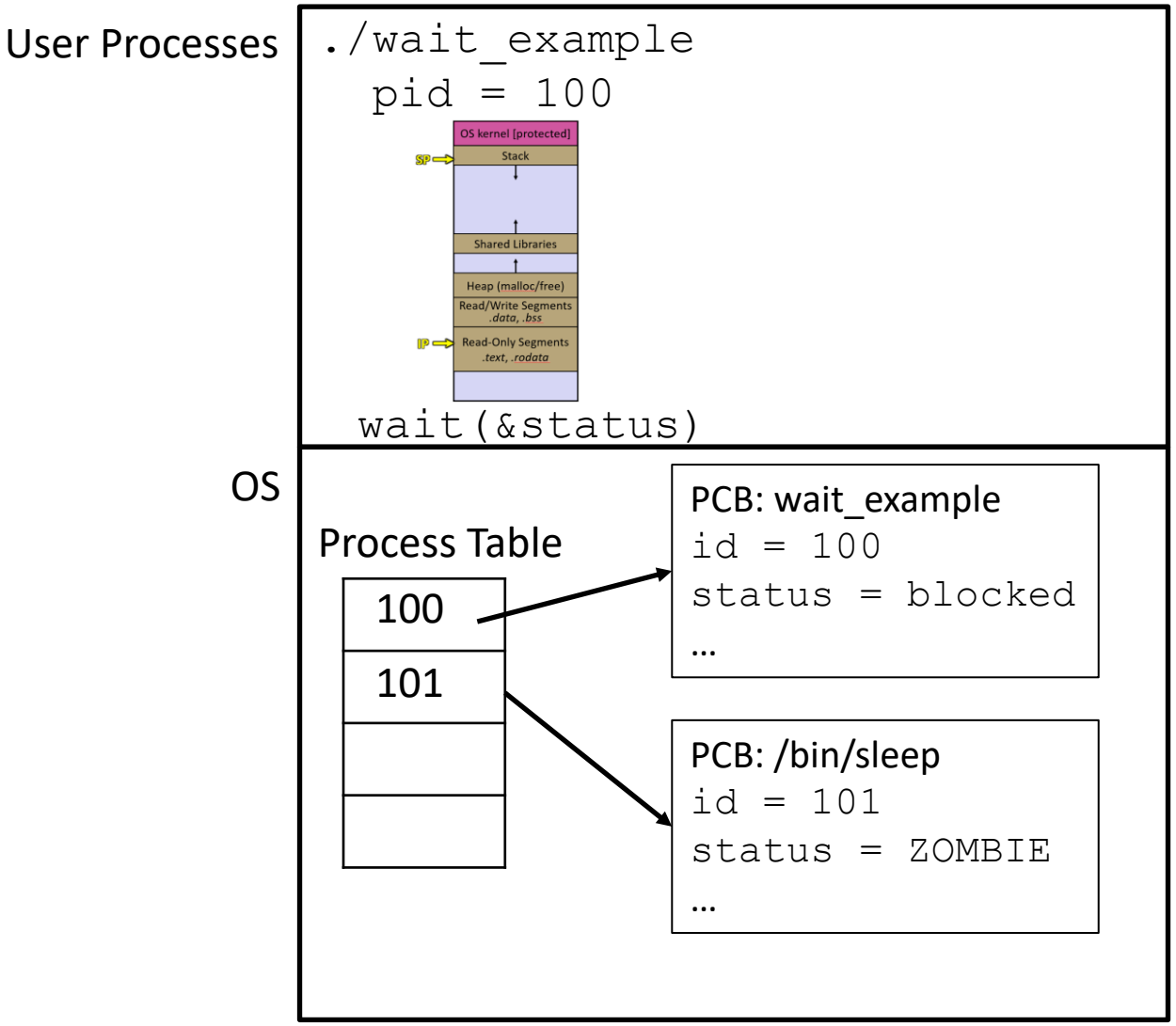


Diagram: wait_example.cpp

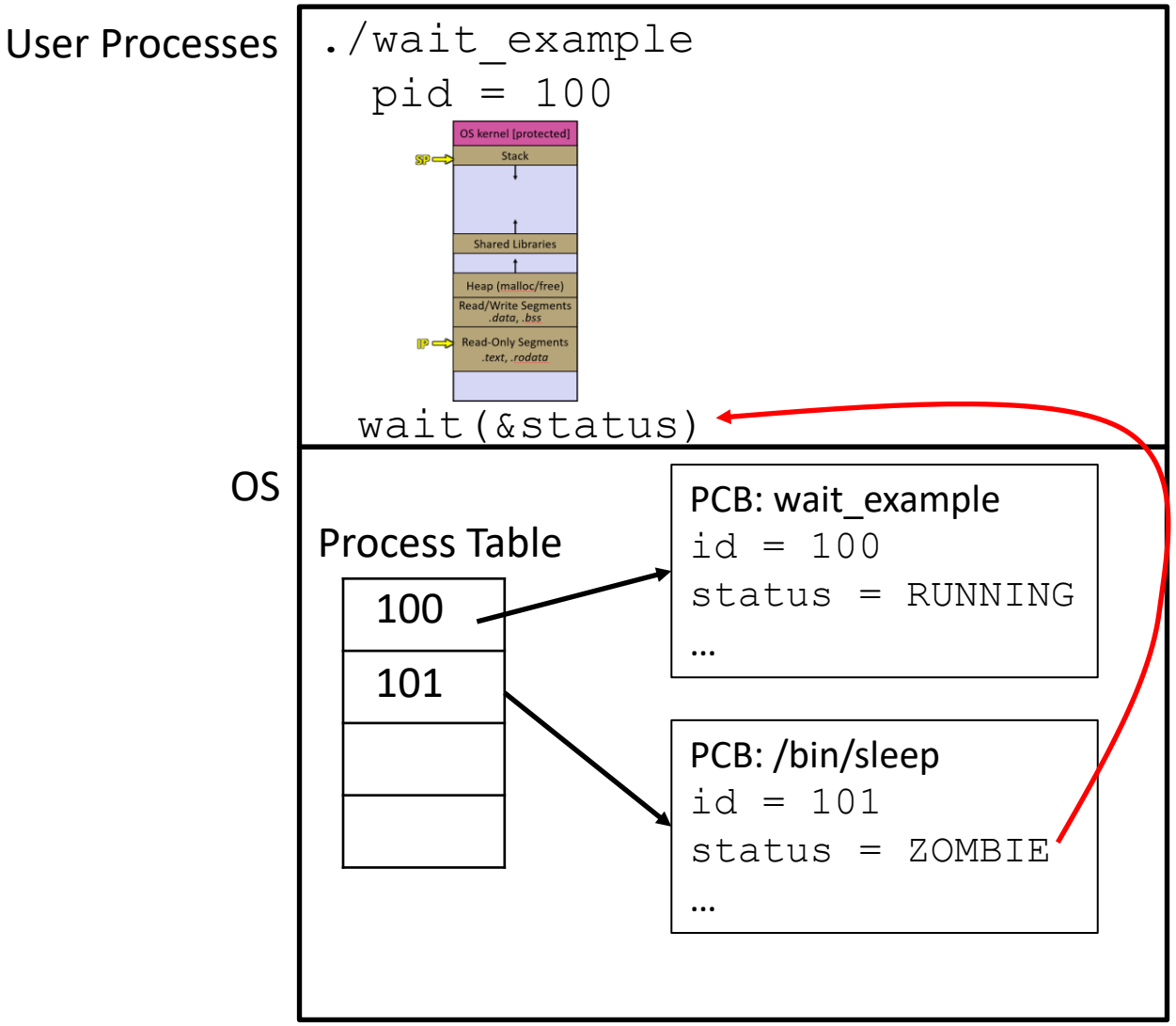


Diagram: wait_example.cpp

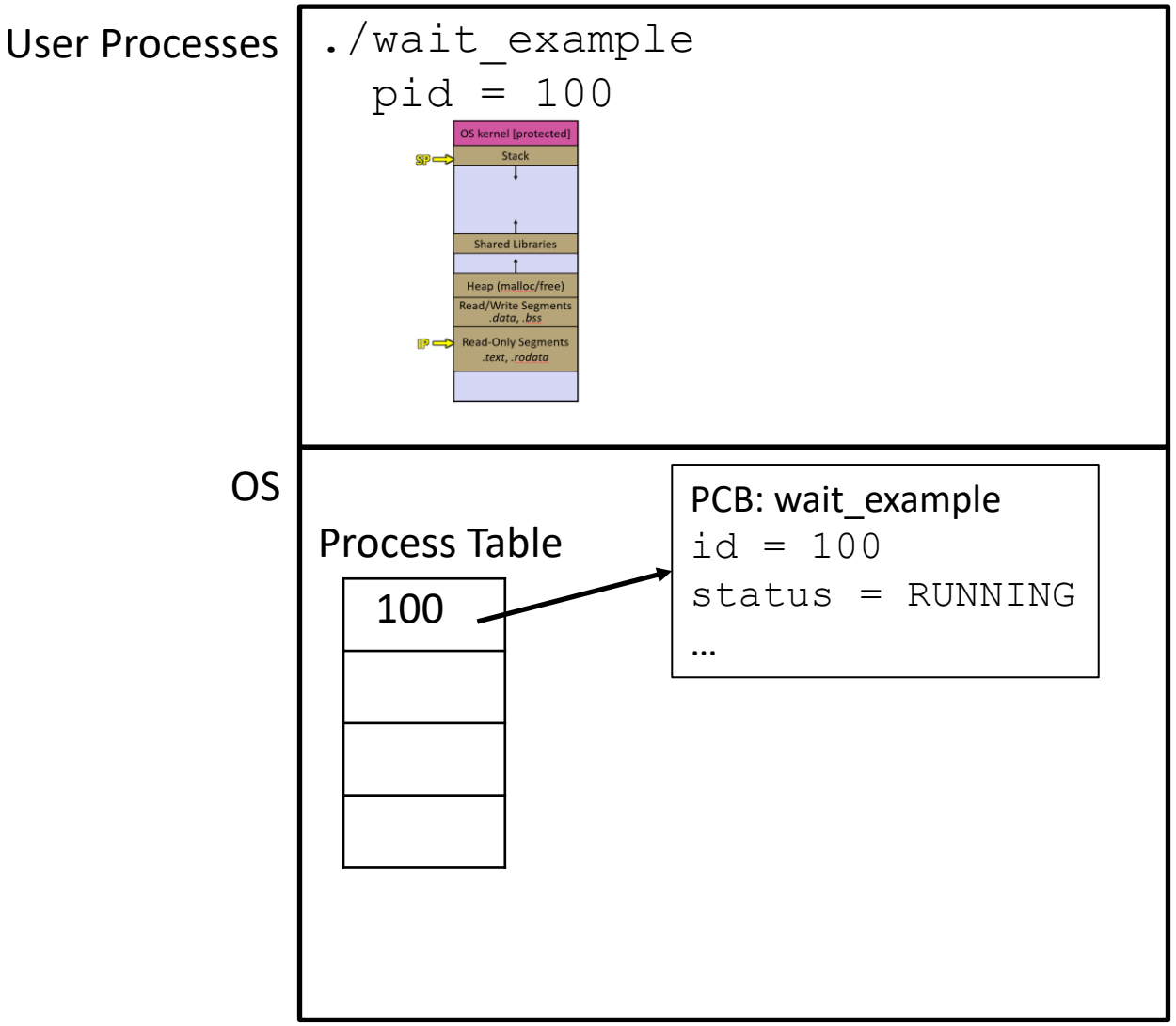


Diagram: wait_example.cpp

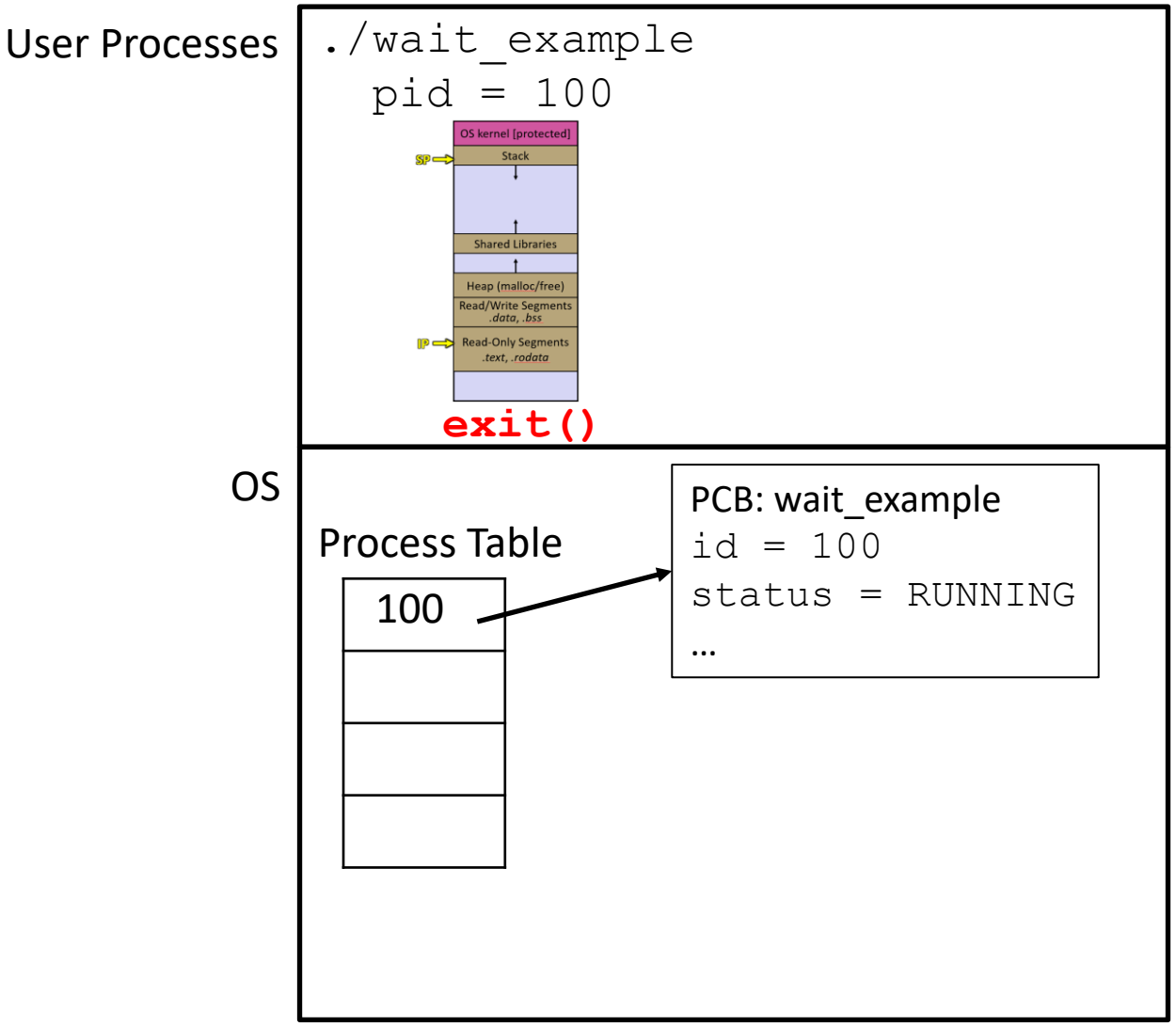
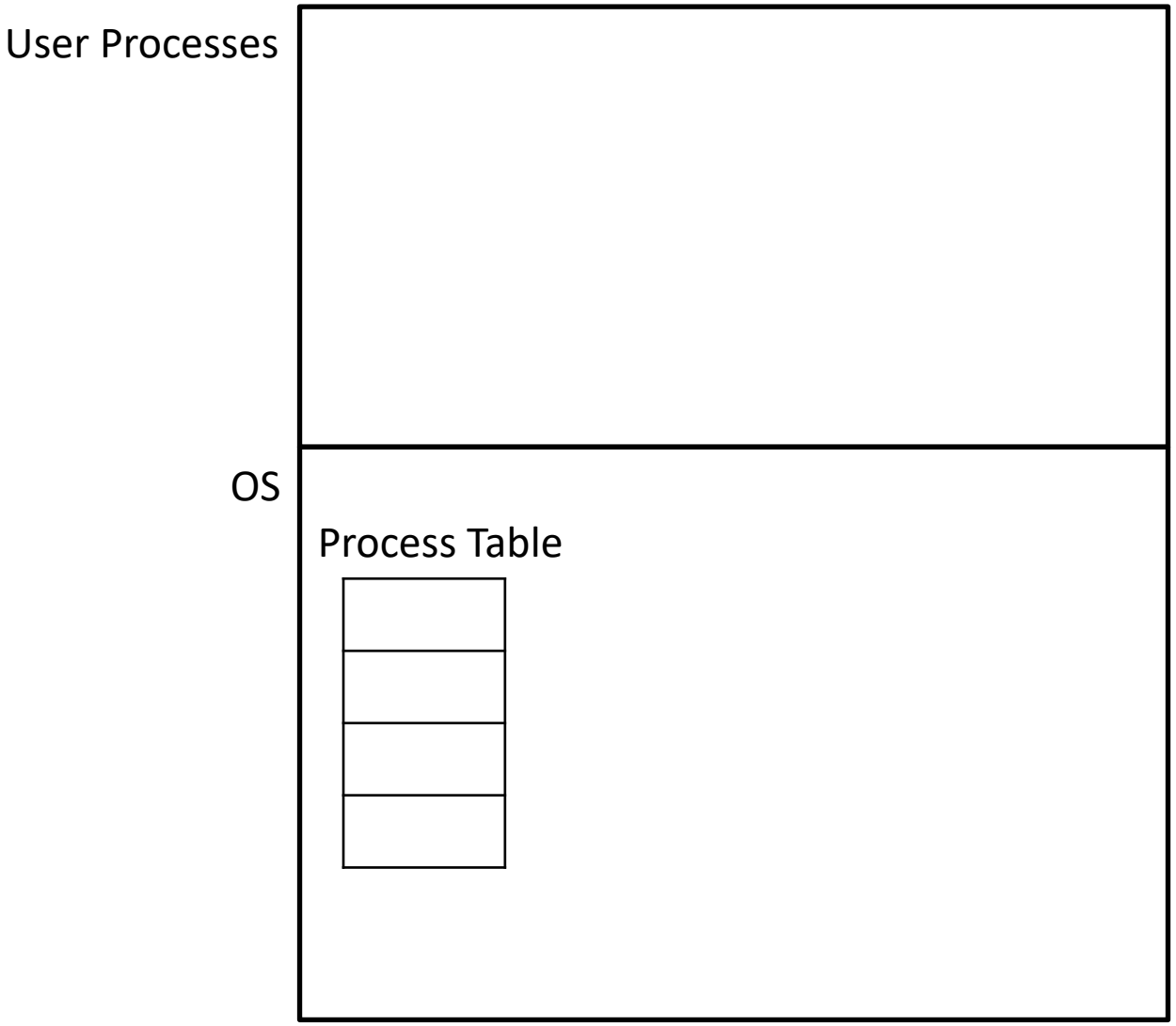


Diagram: wait_example.cpp



`./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.cpp`

Lecture Outline

- fork() & virtual memory
- exec
- wait
- **ordering**

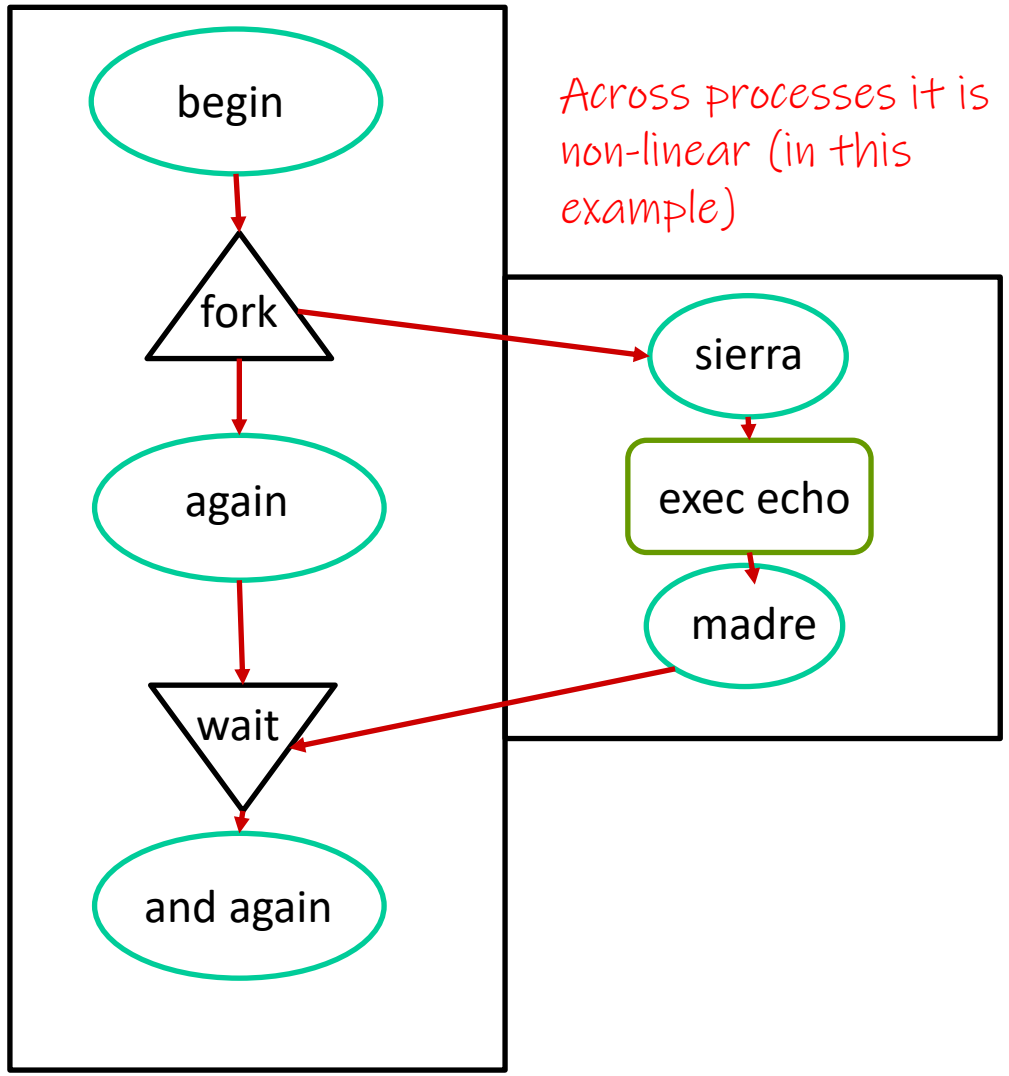
Linearizability

- We can think of the actions done within a process as being “linear”
- A set of operations are linearizable if: The actions performed can be represented as a sequential history
- Actions done across processes cannot always be ordered

Linearizability

- We can represent this with a DAG
- Arrows show dependencies on what must come before/after a certain operation.

```
int main() {
  cout << "begin\n";
  pid_t pid = fork();
  if (pid == 0) {
    cout << "sierra\n";
    execvp("echo", {"echo", "madre", nullptr});
  }
  cout << "again\n";
  waitpid(pid, nullptr, 0);
  cout << "and again\n";
}
```



Notice how it is linear within a single process

Across processes it is non-linear (in this example)

Polls

- See Ed ☺

Ed Discussion

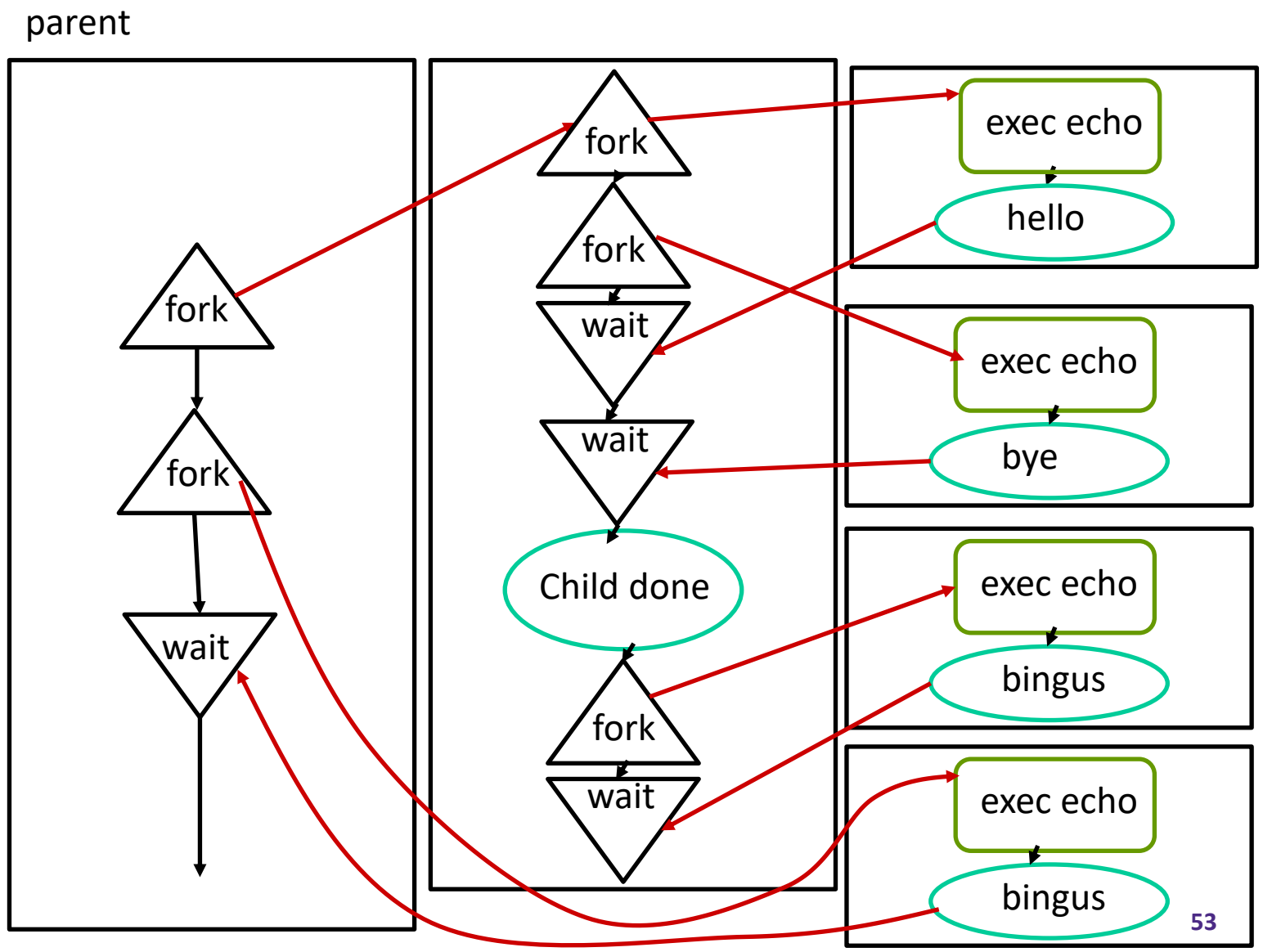
- How many prints can this program have?

```
void execute(const string& cmd, const string& arg) {
  vector<char*> argv;
  argv.push_back(const_cast<char*>(cmd.data()));
  argv.push_back(const_cast<char*>(arg.data()));
  argv.push_back(nullptr);

  execvp(argv.at(0), argv.data());
}
```

```
int main() {
  pid_t pid = fork();
  if (pid == 0) {
    pid = fork();
    if (pid == 0) {
      execute("echo", "hello");
    }
    pit_t pid2 = fork();
    if (pid2 == 0) {
      execute("echo", "bye");
    }
    waitpid(pid, nullptr, 0);
    waitpid(pid2, nullptr, 0);
    cout << "child done\n";
  }
  pid = fork();
  if (pid == 0) {
    exexute("echo", "bingus");
  }
  waitpid(pid, nullptr, 0);
}
```

Ed Discussion

```
int main() {
  pid_t pid = fork();
  if (pid == 0) {
    pid = fork();
    if (pid == 0) {
      execute("echo", "hello");
    }
    pit_t pid2 = fork();
    if (pid2 == 0) {
      execute("echo", "bye");
    }
    waitpid(pid, nullptr, 0);
    waitpid(pid2, nullptr, 0);
    cout << "child done\n";
  }
  pid = fork();
  if (pid == 0) {
    exexute("echo", "bingus");
  }
  waitpid(pid, nullptr, 0);
}
```



That's all for now!

- Next time:
  - More Processes ☺
  - The shell!

- Hopefully you are doing well ☺