OS: Processes (cont.)

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk	Ashwin Alaparthi	Lobi Zhao
Angie Cao	Austin Lin	Pearl Liu
Aniket Ghorpade	Hassan Rizwan	Perrie Quek

pollev.com/tqm

- How are you?

Administrivia

- Simplekv (HW03)
  - Due Friday (2/14)
  - Recommend taking a look sooner rather than later
    - Once you figure out what data members you need, consider talking to a TA or I about it
  - Is more work than previous assignments, not a lot though.

- Check-in 02
  - To be posted tomorrow

- retry_shell (HW04)
  - Posted Tomorrow or Friday
  - Due 2/21
  - Should have everything you need after this lecture

Lecture Outline

- **Processes & fork() (wrapup)**
- execvp()
  - C++ Interoperability
- wait(), waitpid() and exit status

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

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Hint 1: there are three processes

Hint 2: Each prints out twice "Hi" and "Bye"

Sequence 1:	Sequence 2:
Hi 1	**Hi 3**
Bye	**Hi 1**
Hi 2	**Hi 2**
Bye	**Bye**
Bye	**Bye**
Hi 3	**Bye**

A.	No	No
B.	No	Yes
C.	Yes	No
D.	Yes	Yes
E.	We're lost…	

Lecture Outline

- Processes & fork() (wrapup)
- **C++ Interoperability**
- execvp()
- wait(), waitpid() and exit status
- Documentation Reading

std::array

- Similar to vector, we have array
  - Both contain a sequence of data that we can index into

- Main differences: the size
  - Vector is resizable (grows to whatever length we need)
  - Array is a static size (size is determined at compile time)

- Main differences: the allocation
  - To support being resizable, vector uses a lot of dynamic allocation
  - **Array does not use any dynamic allocation**

array example

```
int main(int argc, char* argv[]) {
  array<int, 3> arr {6, 5, 4};
  // arr.push_back(3); push_back does not exist!

  cout << arr.size() << endl; // prints 3
  cout << arr.at(2) << endl;  // prints 4

  // iterates through all elements and prints them
  for (const auto& element : arr) {
     cout << element << endl;
  }

  return EXIT_SUCCESS;
}
```

C++ Arrays

- C arrays are considered dangerous, and not safe to use
  - Length is not attached to the array
  - There is no bounds checking
  - Arrays are not readable code
    Consider this CIS 5480 Example:
    What do you think “commands” represents?

```
// example from CIS 5480
struct parsed_command {
  int num_commands;
  char*** commands;
};
```

- In our code, we will use C++ Arrays instead, but we need to call C code that expects C arrays…

C++ Arrays -> C array

- Can use .data() and .size() to convert to a C array

```
int sumAll(int* a, int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}

int main(){
  array<int, 1024> arr{};
  sumAll(arr.data(), arr.size());
}
```

C++ Vectors -> C array

- Can use .data() and .size() to access the underlying C array

```
int sumAll(int* a, int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}

int main(){
  vector<int> vec{3, 4, 5};
  sumAll(vec.data(), vec.size());
}
```

C++ Vectors -> C array

- Can use .data() and .size() to access the underlying C array

```
int sumAll(int* a, int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}

int main(){
  vector<int> vec{3, 4, 5};
  sumAll(vec.data(), vec.size());
}
```

pollev.com/tqm

- Does this code correctly print 10?
  - Assume this code compiles

```
int sum_carr(int* arr, size_t len) {
  int sum = 0;
  for (size_t i = 0; i < len; i++) {
    sum += arr[i];
  }
  return sum;
}

int* vec_to_carr(vector<int> vec) {
  return vec.data();
}


int main() {
  vector<int> my_vals {1, 2, 3, 4};
  int* arr = vec_to_carr(my_vals);
  cout << sum_carr(arr, my_vals.size()) << endl;
}
```

C++ Strings -> C Strings

- C++ Strings can grant access to the underlying C-String through the function `.c_str()`

- This is useful for when interfacing with C code from C++:

```
#include <fcntl.h>     // for open()
#include <unistd.h>    // for close()
  ...
  string fname{"foo.txt"};
  const char* fname_cstr = fname.c_str();
  int fd = open(fname_cstr, O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...
  close(fd);
```

Lecture Outline

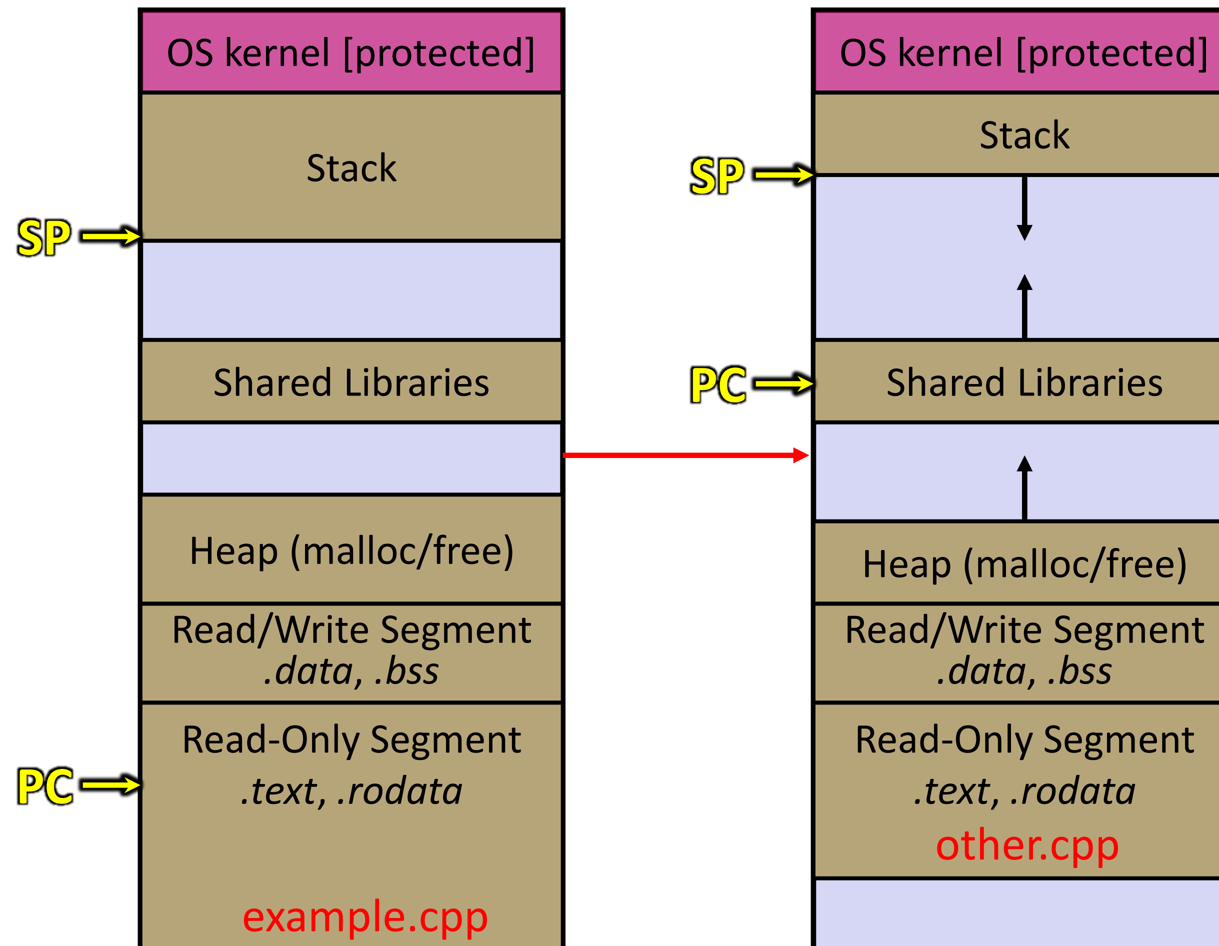
- Processes & fork() (wrapup)
- C++ Interoperability
- **execvp()**
- wait(), waitpid() and exit status
- Documentation Reading

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 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

Exec Demo

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

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.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens if we use fork() and exec() together?

pollev.com/tqm

- 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
    array<const char*, 3> argv = {
      "echo", "hello", nullptr
    };
    execvp(argv.at(0), const_cast<char**>(argv.data());
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

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

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

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

pollev.com/tqm

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

  // fork a process to exec clang
  pid_t clang_pid = fork();

  if (clang_pid == 0) {
    // we are the child
    array<const char*, 5> argv = {
      "clang-15", "-o", "hello","hello_world.c", nullptr
    };
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }

  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    array<const char*, 2> argv {"./hello", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

broken_autograder.cpp

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

Lecture Outline

- Processes & fork() (wrapup)
- C++ Interoperability
- execvp()
- **wait(), waitpid() and exit status**

From a previous poll:

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

  // fork a process to exec clang
  pid_t clang_pid = fork();

  if (clang_pid == 0) {
    // we are the child
    array<const char*, 5> argv = {
      "clang-15", "-o", "hello","hello_world.c", nullptr
    };
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }

  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    array<const char*, 2> argv {"./hello", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

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[]) {
  // fork a process to exec clang
  pid_t clang_pid = fork();

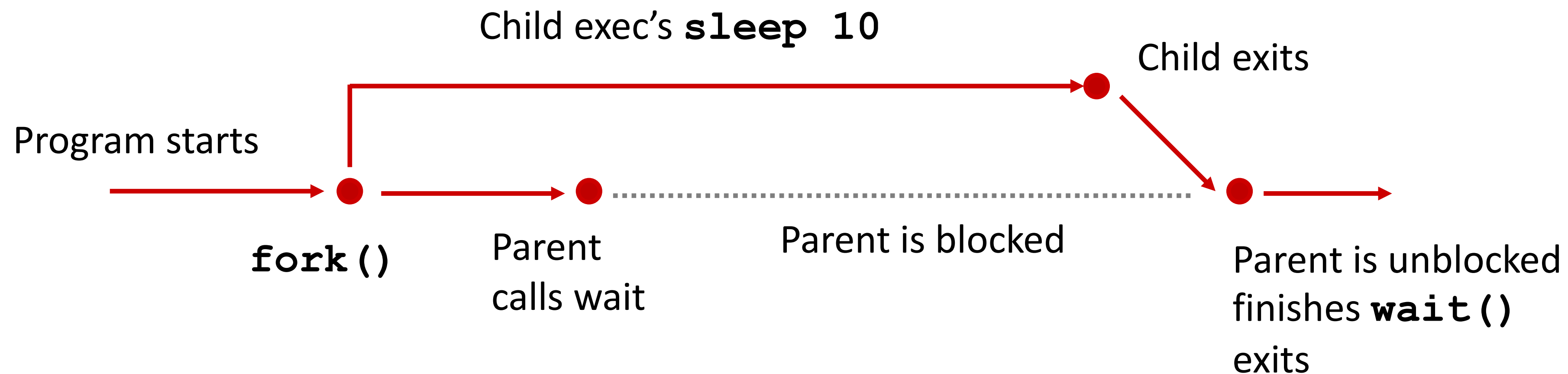
  if (clang_pid == 0) {
    // we are the child
    array<const char*, 5> argv = {
      "clang-15", "-o", "hello","hello_world.c", nullptr
    };
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  wait(NULL); // should error check, not enough slide space :(
  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    array<const char*, 2> argv {"./hello", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

autograder.c

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

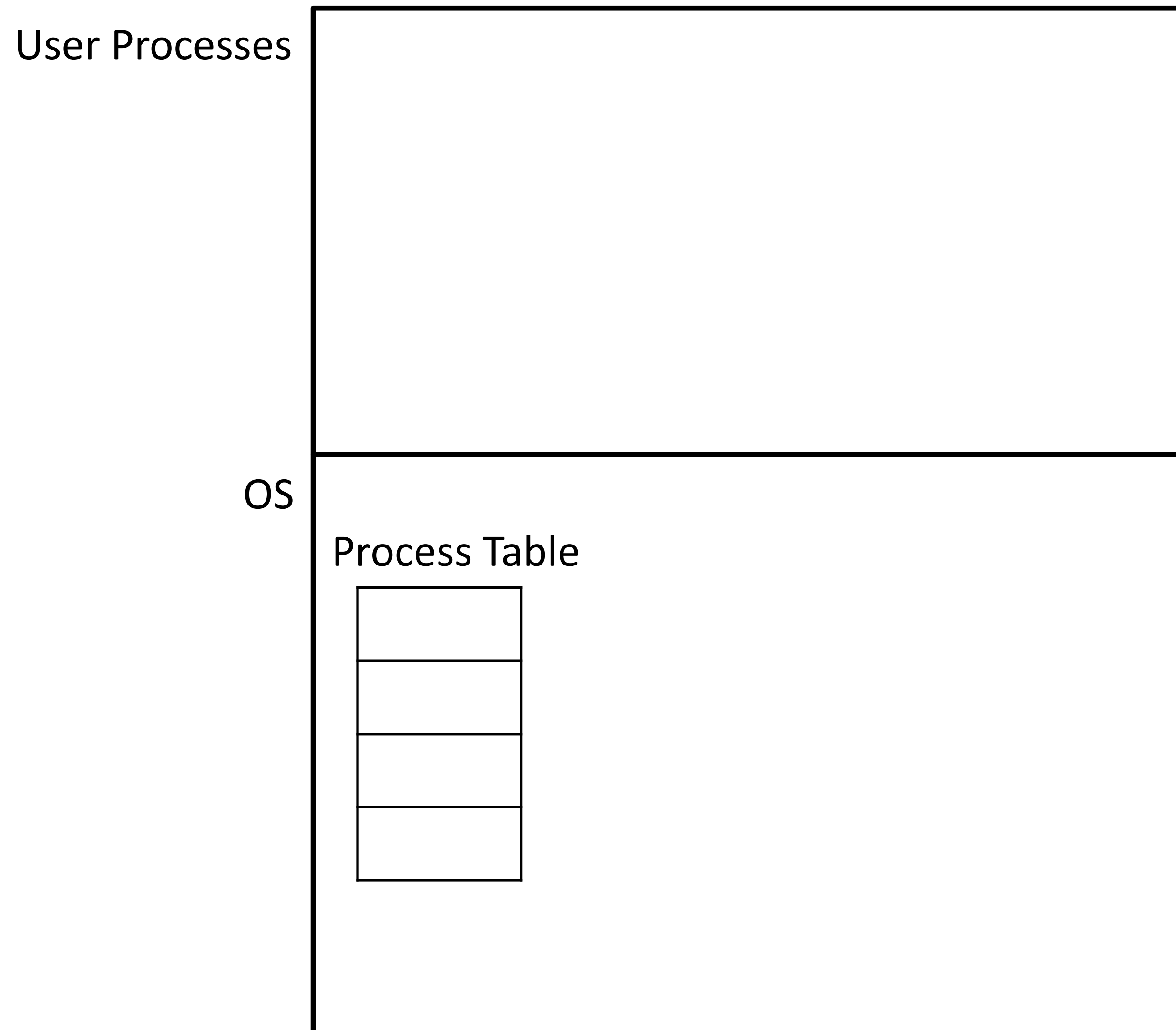
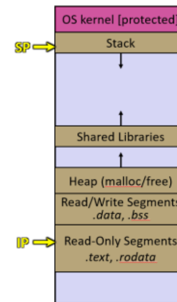


Diagram: wait_example.cpp

User Processes

./wait_example
pid = 100



OS

Process Table

100

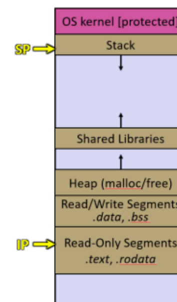
PCB: wait_example
id = 100
status = running
…

Diagram: wait_example.cpp

User Processes

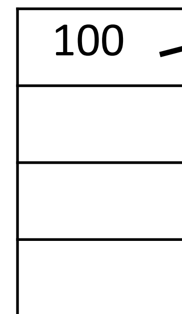
`./wait_example`

`pid = 100`



OS

Process Table



PCB: wait_example

`id = 100`

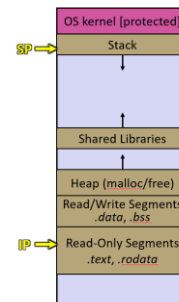
`status = running`

…

Diagram: wait_example.cpp

User Processes

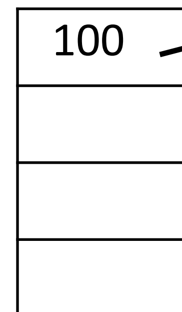
`./wait_example`
`pid = 100`



`fork()`

OS

Process Table



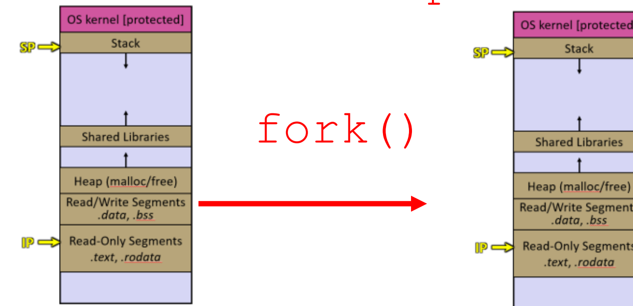
PCB: wait_example
`id = 100`
`status = running`
…

Diagram: wait_example.cpp

User Processes

./wait_example
pid = 100

./wait_example
pid = 101



OS

Process Table

100
101

PCB: wait_example
id = 100
status = running
...

PCB: wait_example
id = 101
status = running
...

Diagram: wait_example.cpp

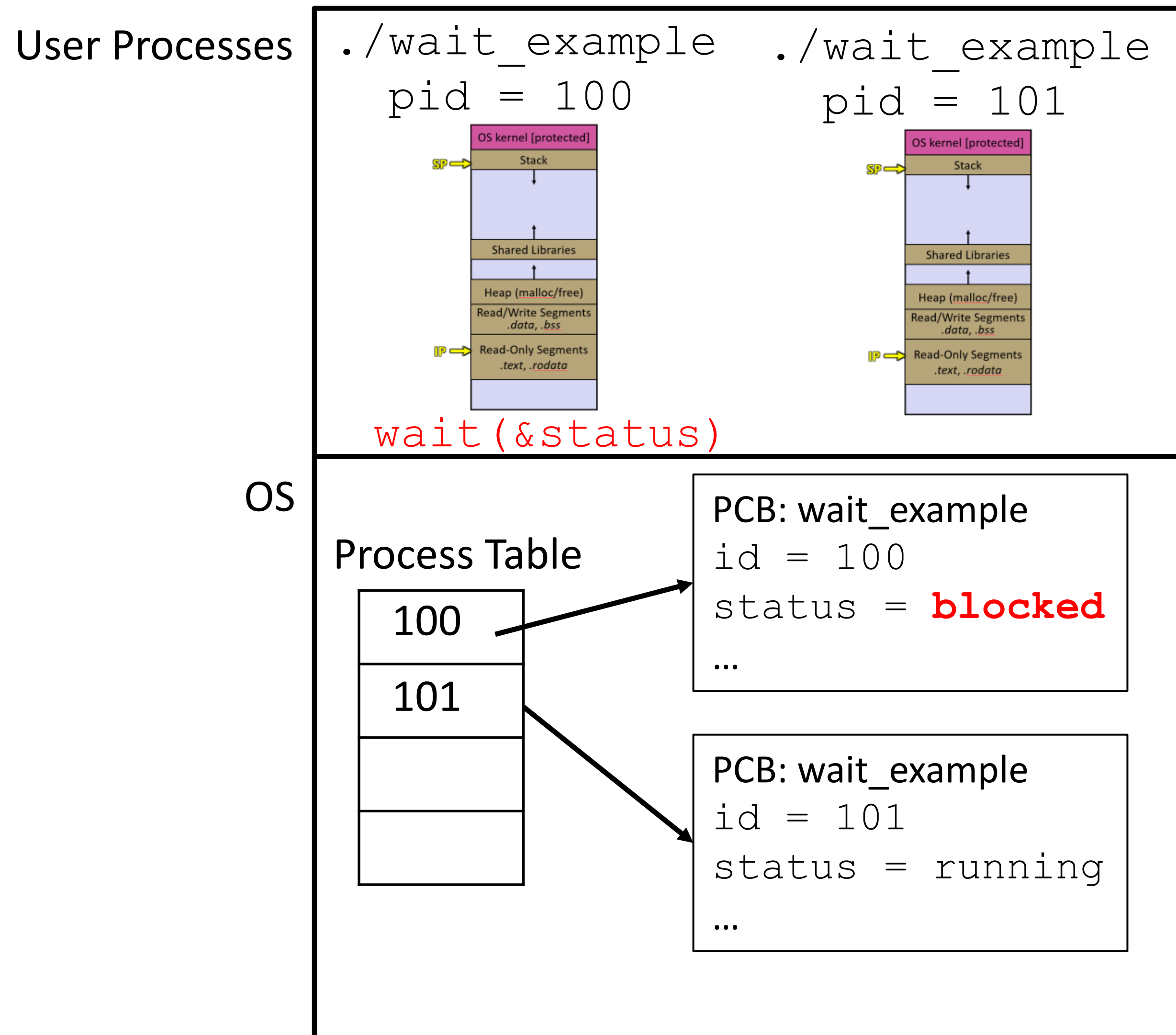


Diagram: wait_example.cpp

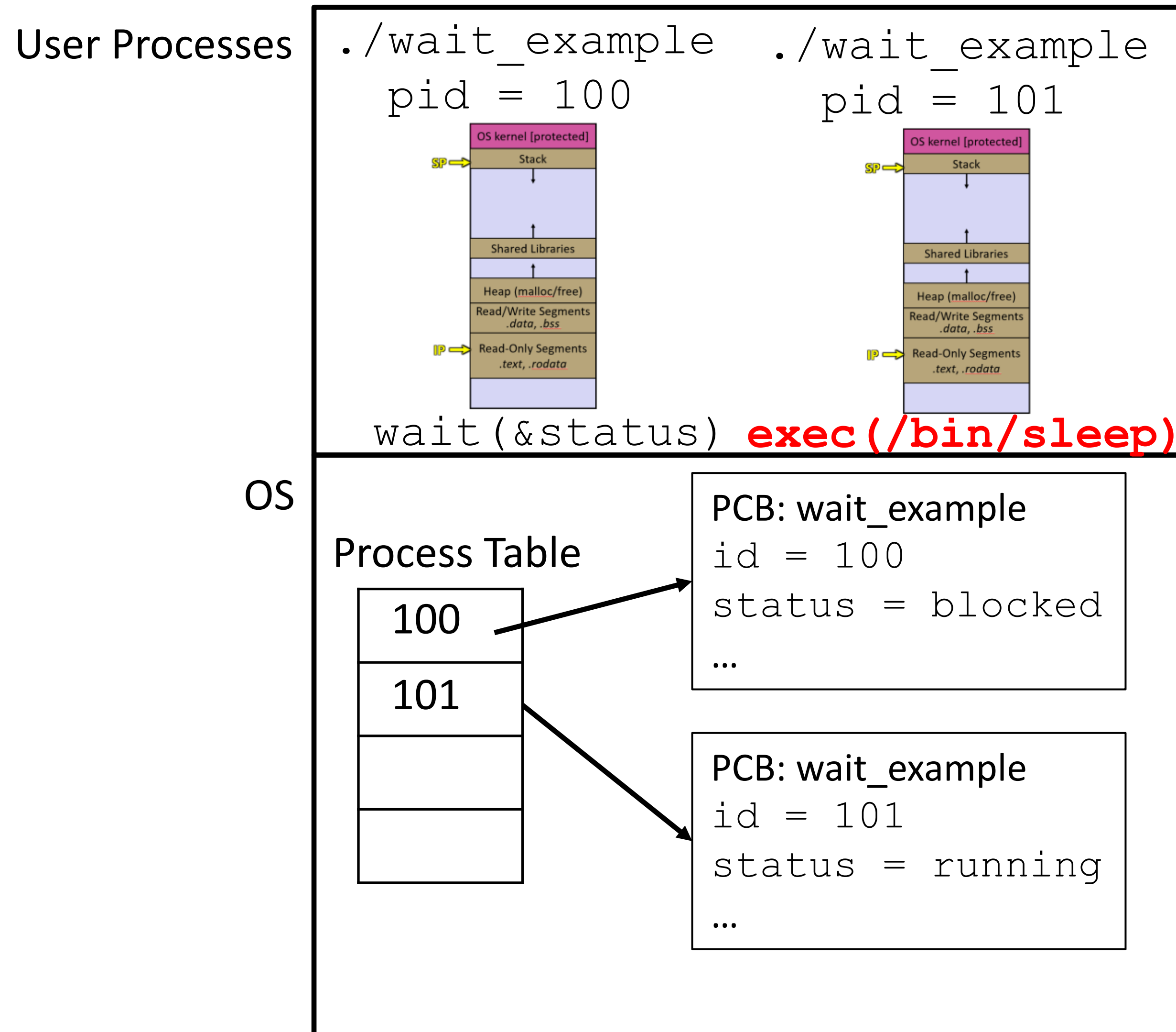
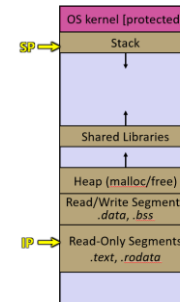


Diagram: wait_example.cpp

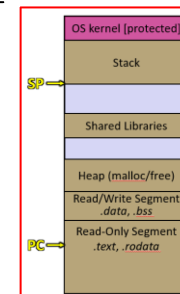
User Processes

./wait_example
pid = 100



wait(&status)

/bin/sleep
pid = 101



exec(/bin/sleep)

OS

Process Table

100
101

PCB: wait_example
id = 100
status = blocked
…

PCB: /bin/sleep
id = 101
status = running
…

Diagram: wait_example.cpp

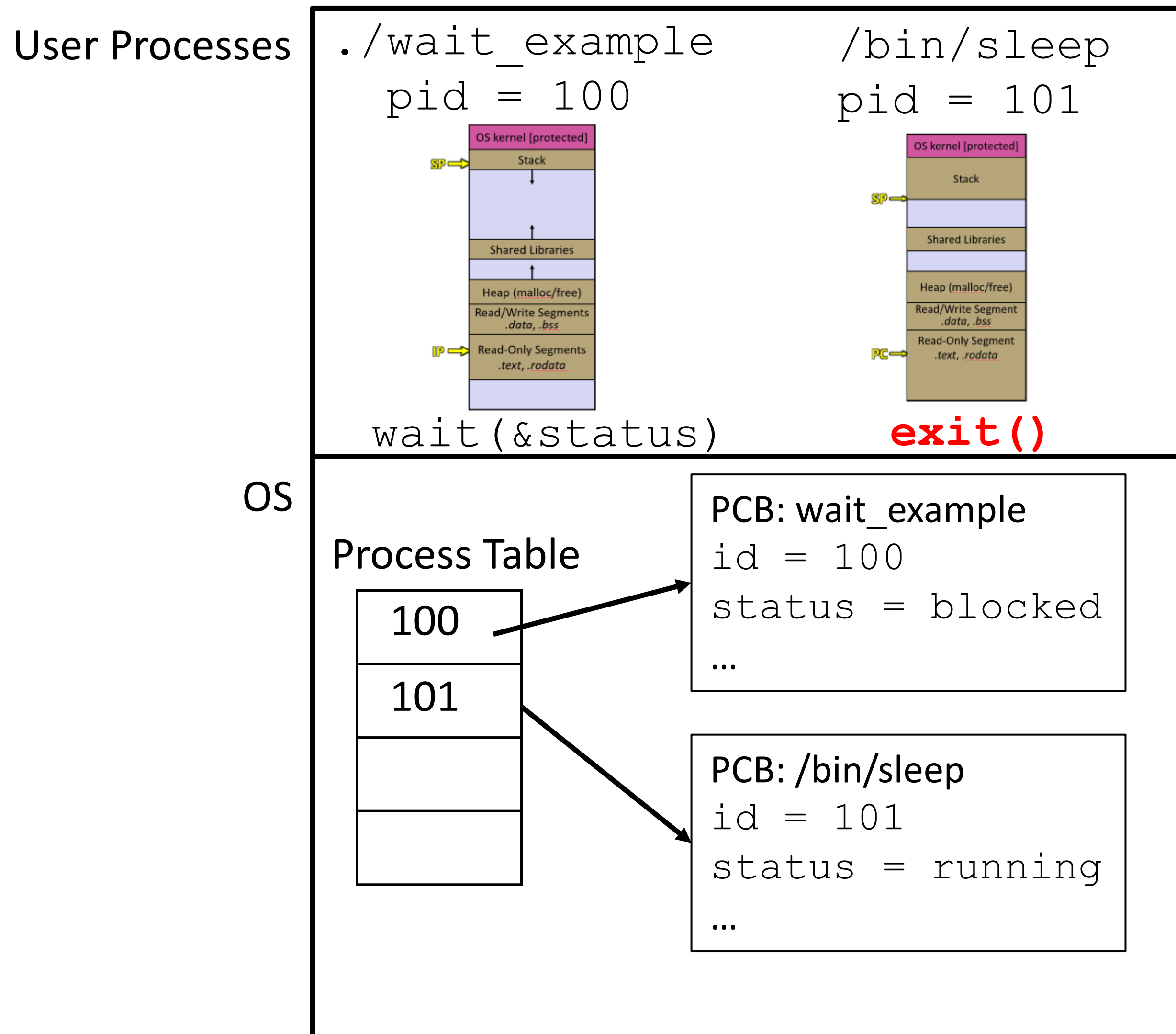
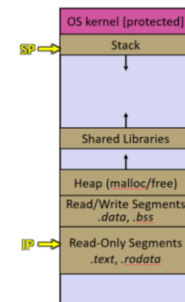


Diagram: wait_example.cpp

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.cpp

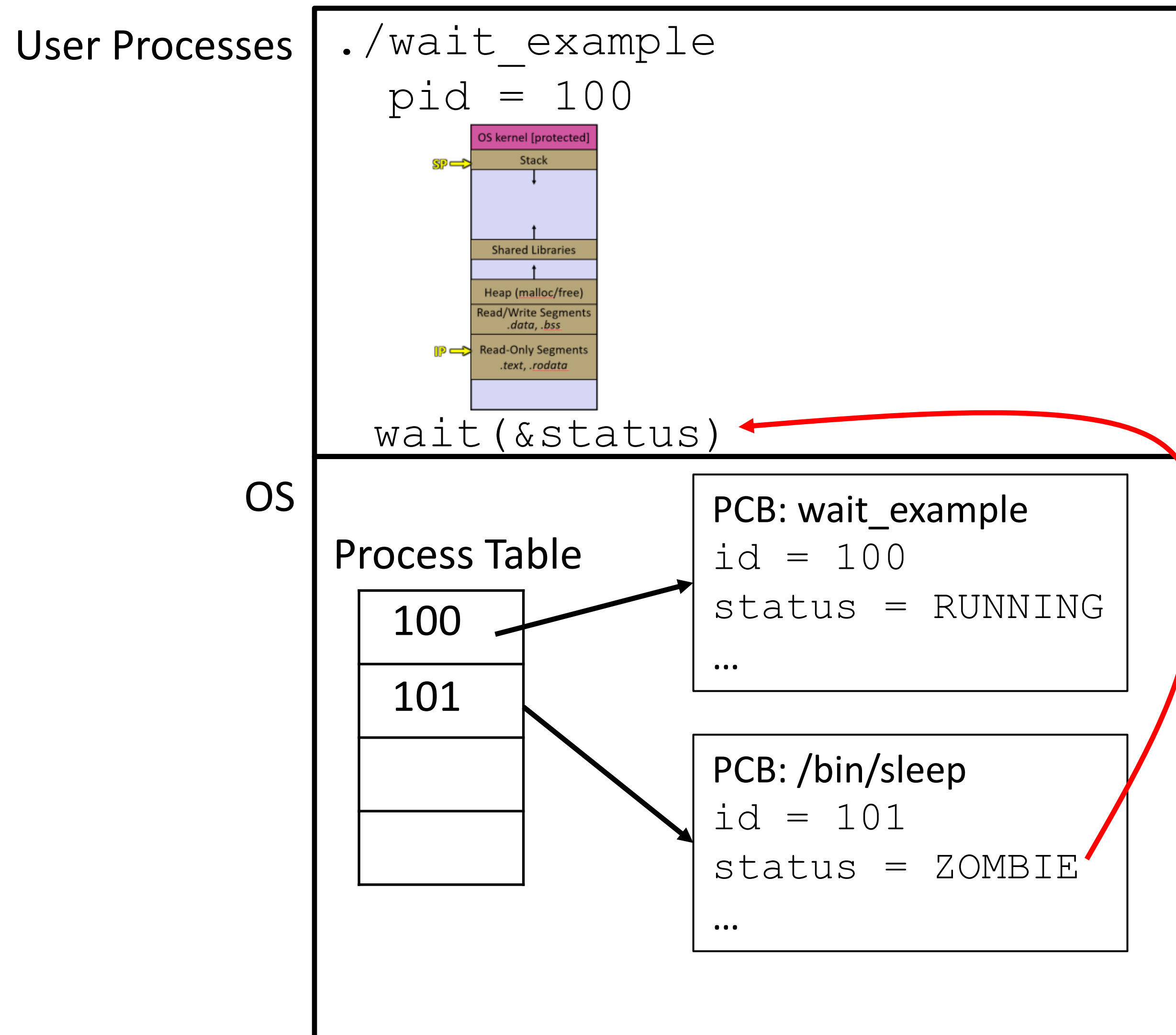
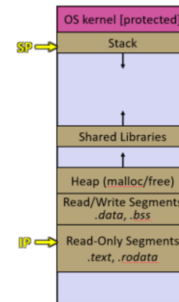


Diagram: wait_example.cpp

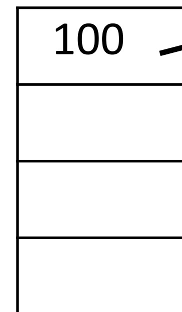
User Processes

`./wait_example`
`pid = 100`



OS

Process Table

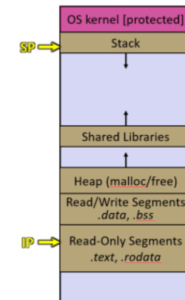


PCB: wait_example
`id = 100`
`status = RUNNING`
…

Diagram: wait_example.cpp

User Processes

./wait_example
pid = 100



exit()

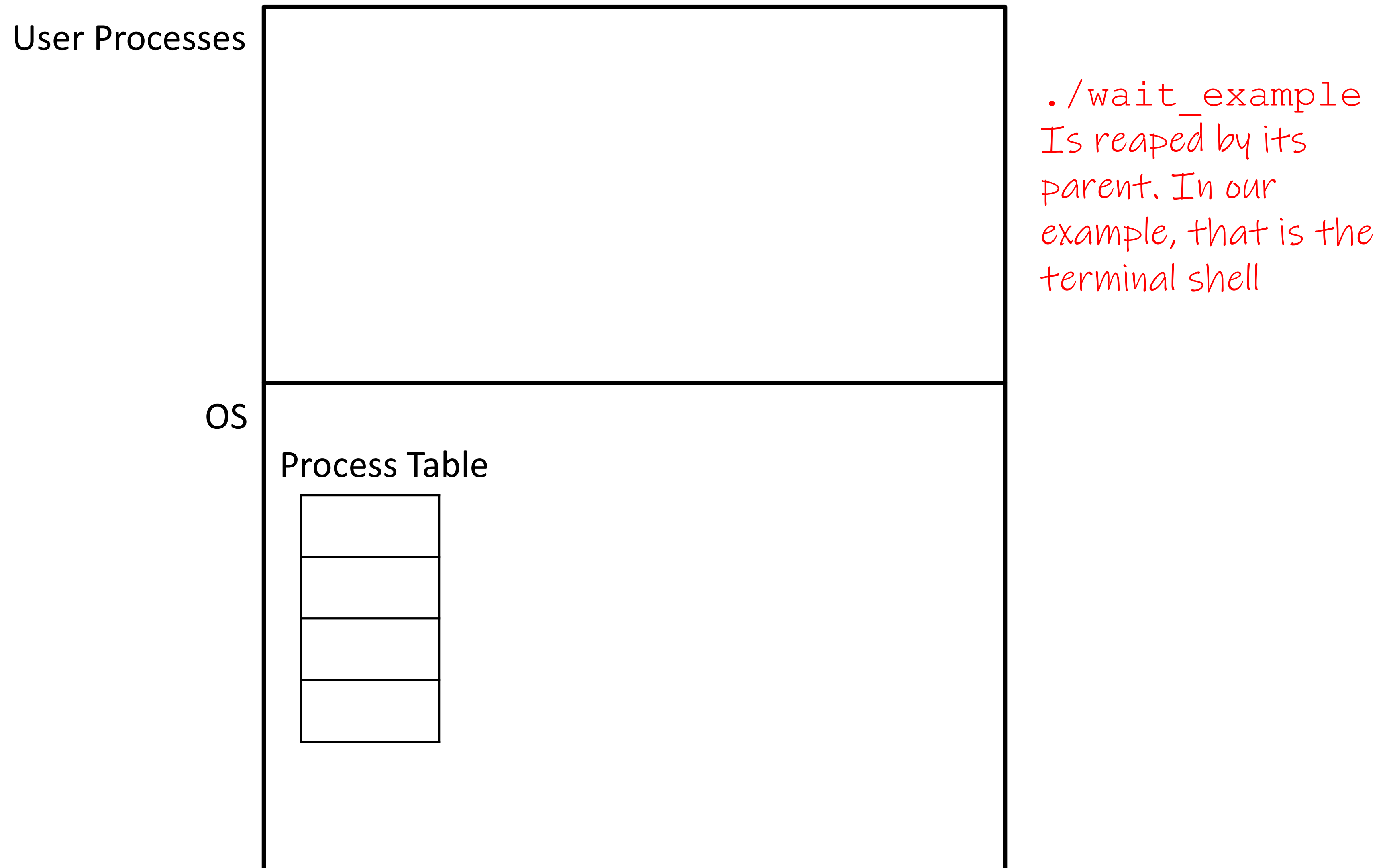
OS

Process Table

100

PCB: wait_example
id = 100
status = RUNNING
…

Diagram: wait_example.cpp



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

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

- Demo: see example in `exit_status.cpp`

pollev.com/tqm

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

  if (clang_pid == 0) {
    // we are the child
    array<const char*, 5> argv = {
      "clang-15", "-o", "hello","hello_world.c", nullptr
    };
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    array<const char*, 2> argv {"./hello", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  wait(NULL);  // previously before second fork()
  wait(NULL);
  return EXIT_SUCCESS;
}
```

We take our previous code that we fixed and modify it. Now we call wait twice at the end of the program.

What happens?

Does our code still always work?

That's it for now!

- More next lecture ☺