CS202 (003): Operating Systems
Process II

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202 and Yang Tang's CS202

Last time…

How does process access system resources?

User-level Process ⇄ **System Calls!** ⇄ OS Kernel

System calls are the mechanism by which user-level programs ask the OS to do things for them

What is a System Call?

A system call looks like a function call in C

- Process control (e.g., fork, exit)
- File management (e.g., open, read, write)
- Device management (e.g., ioctl)
- Information maintenance (e.g., time, date)
- Communication (e.g., pipe, socket)

```
int fd = open(const char* path, int flags)
write(fd, const void *, size_t)
read(fd, void *, size_t)
```

You can always use the command

```
man 2 <syscall>
```

to get the documentation

System Call ≠ Function Call

Calling Convention

All registers (except %rax) are call-preserved.
Kernel must save and restore all registers (except %rax)

```
; Calling a function named 'print_hello'
call print_hello
```

Instruction Used

```
; Performing a 'write' system call
mov rax, 1          ; system call number for 'write'
……                  ; setting up the parameters
syscall             ; invokes OS to do the write
```

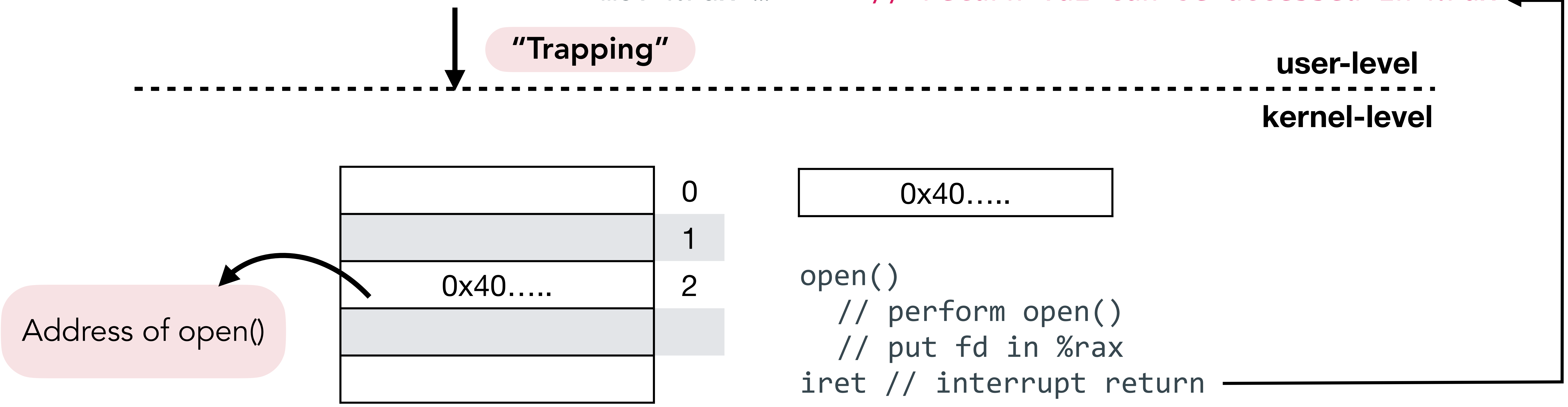
Switch to privilege mode!

Switching to Privilege Mode

open

```
mov $2, %rax     // System call number for open()
mov ..., %rdi    // First argument
mov ..., %rsi    // Second argument
syscall          // Software interrupt to switch to kernel mode

mov %rax …        // return val can be accessed in %rax
```




```
open()
  // perform open()
  // put fd in %rax
iret // interrupt return
```

Privileged Mode v.s. Unprivileged Mode

"Kernel Mode"	"User Mode"
Unrestricted access to system resources	No direct access to system resources
Can access both user programs and kernel programs	No direct access to kernel programs
Can refer to any memory block in the system and can also direct the CPU for the execution of an instructions	Can only refer to memory allocated for user mode

Hardware knows the difference between kernel and user modes and enforce it!

Three Ways to Invoke the Kernel

1. System Calls

2. Interrupts

It is a hardware event

It allows a device to notify the kernel that it needs attention.

When interrupt happens…

1. Process stops running
2. CPU invokes interrupt handler
3. Kernel starts running
4. Kernel handles the interrupt
5. Kernel returns control

Process is not aware that interrupts happened

Hardware and kernel need to save **all** process state (when interrupt starts), and restore all of it (when interrupt finishes)

3. Exceptions

CPU cannot execute process instructions

(for this class), an exception happens means "**the process did something wrong**"

When exception happens…

1. CPU knows immediately
2. CPU invokes exception handler
3. Kernel handles the exception by either:
    1. kill the process (default, **segfault**)
    2. signal to the process (and **signal handler** handles the rest)
    3. silently handle the exception

What is a System Call?

A system call looks like a function call in C

- Process control (e.g., fork, exit)
- File management (e.g., open, read, write)
- Device management (e.g., ioctl)
- Information maintenance (e.g., time, date)
- Communication (e.g., pipe, socket)

```c
pid_t pid = fork();

    if (pid == 0) {
        getpid();  // Child process
    } else {
        getpid();  // Parent process
    }
```

How do we create a process?

A System Call!

Process creation

```
fork(); // Create a new process
```

System Calls for Process Control

Process creation

```
fork(); // Create a new process
```

Process identification

```
getpid();  // Calling process pid
getppid();  // Parent of the calling process pid
```

System Calls for Process Control

Process creation

Parent Process *running code.c*

```
17 int ret = fork(); // Create a new process
```

Parent Process

Starts execution from **L18** of code.c

```
ret = getpid(Child Process);
```

Child Process

Starts execution from **L18** of code.c

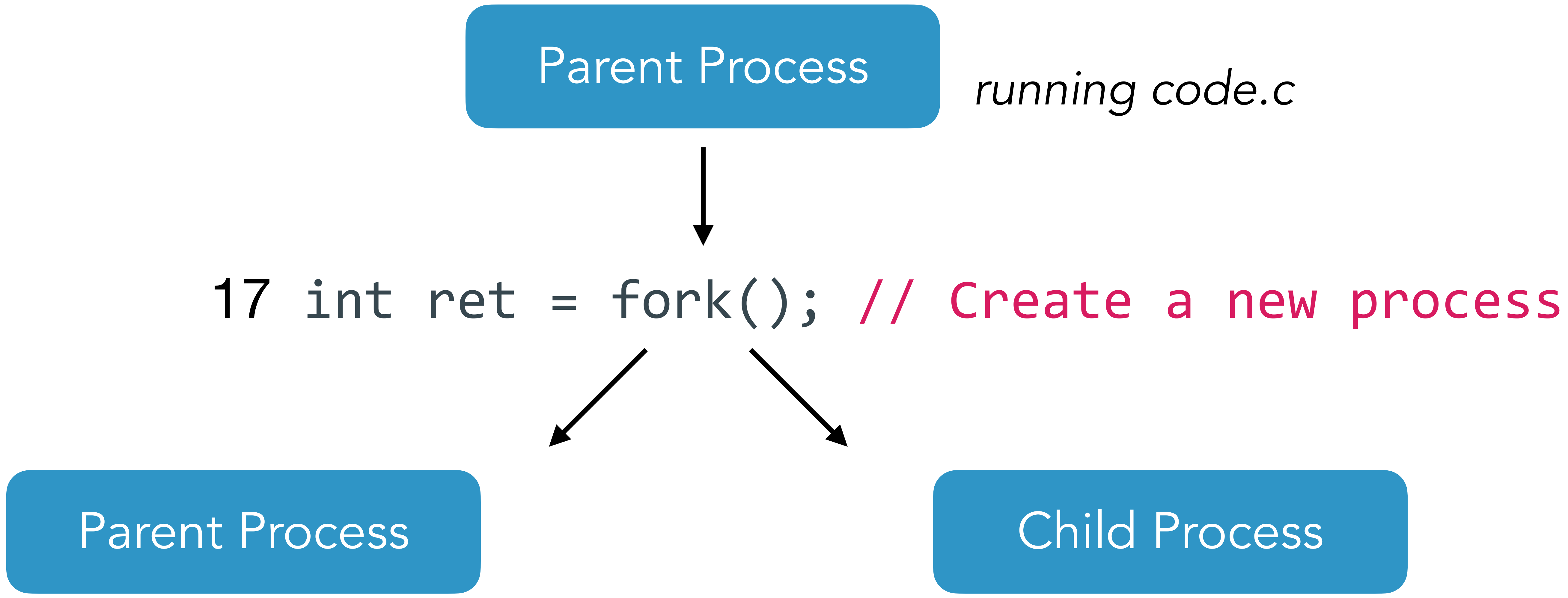
```
ret = 0;
```

Child Process inherits program code, program counter, memory, opened files from Parent Process

Child Process has different **ret** value, **pid**, parent, running time, file locks from Parent Process

System Calls for Process Control

Process creation

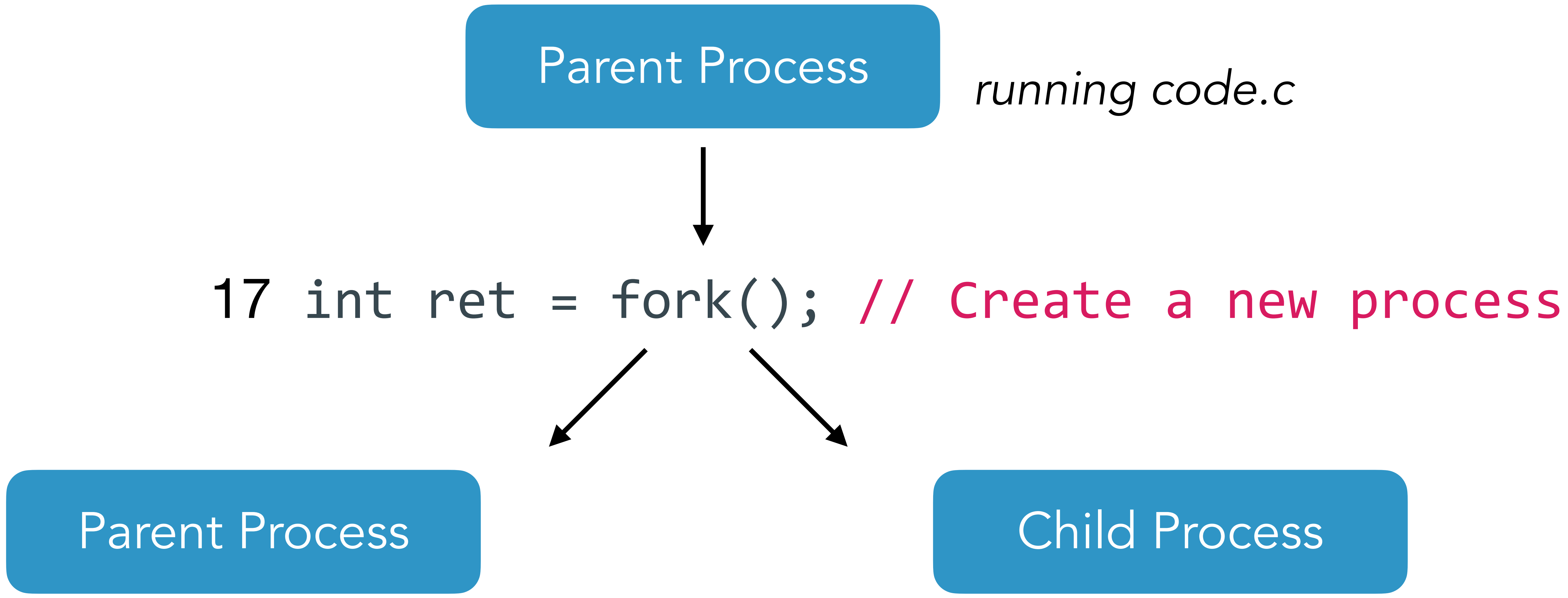


Who runs first?

We don’t know. That depends on the **process scheduling**.

System Calls for Process Control

Process creation



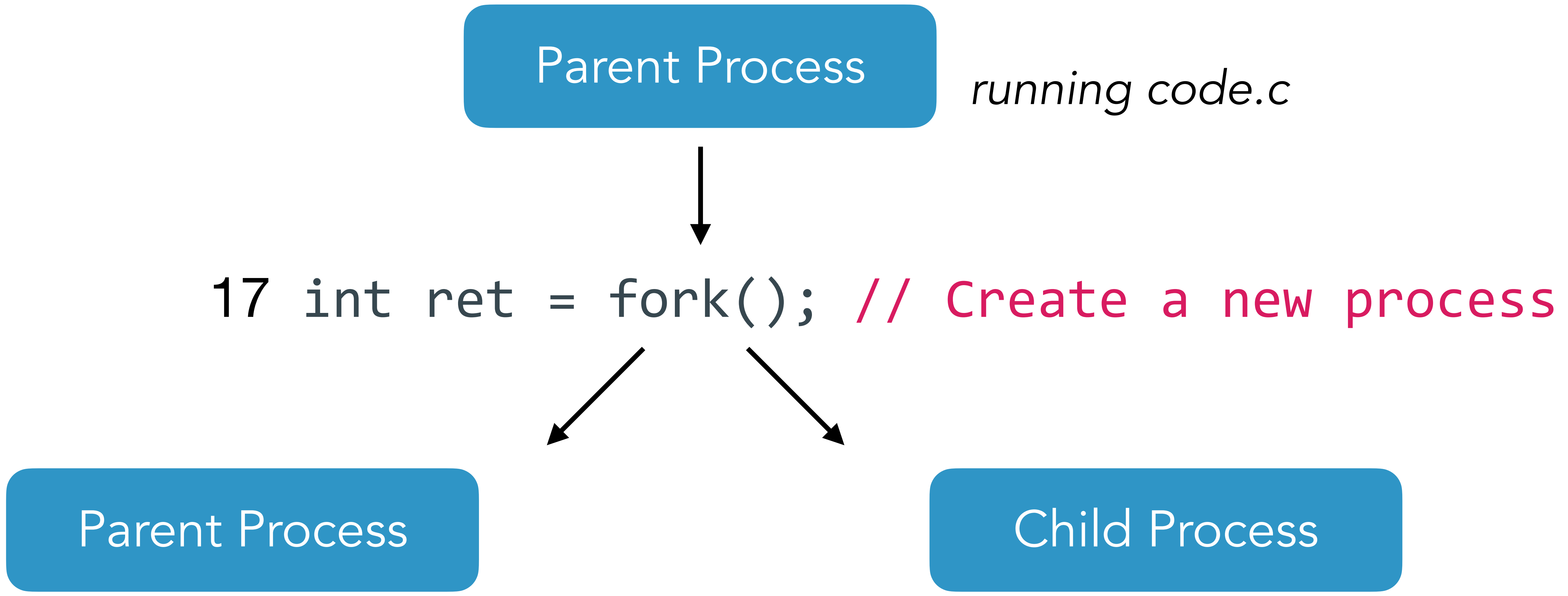
Is it possible to make sure child process finish first?

Yes, we can use `wait()` system call[1].
Parent process can call `wait()` to delay its execution until child finishes executing.
When the child is done, `wait()` returns to the parent.

[1]There are a few cases where wait() returns before the child exits; read the **man page** for more details.

System Calls for Process Control

Process creation



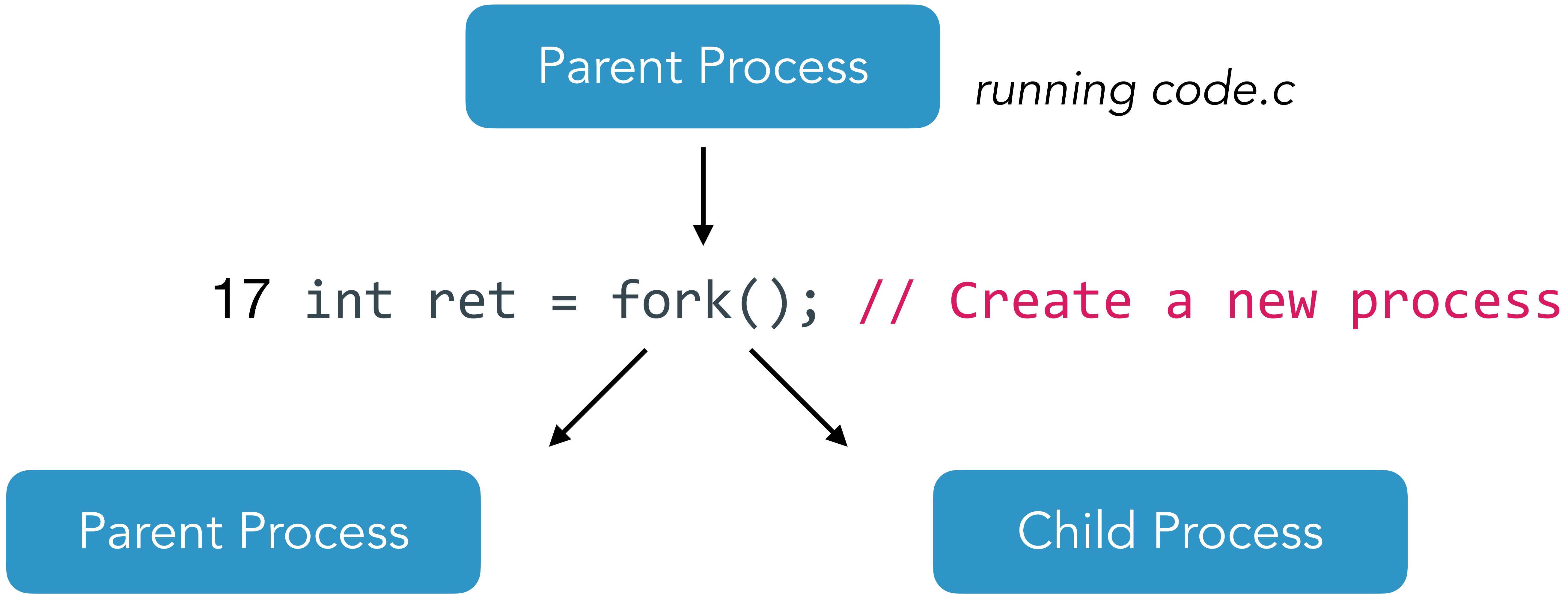
Suppose we have two users, what happens if one of them runs the following code?

```
for (i = 0; i < 10; i++) {
    fork();
}
while (1) {}
```

Whoever runs this code will gets a lot more of the CPU than the other

System Calls for Process Control

Process creation



Wait, are we never going to execute other program?

System Calls for Process Control

Process execution

Parent Process *running code.c*

↓

```
exec[2]("code2", args, env); // Replace current process
```

↓

Parent Process *running code2.c*

Starts execution of code2

It **never returns to** code.c

Parent Process *running code2.c* discards memory, registers of Parent Process *running code.c*

Parent Process *running code2.c* preserves pid, process relationship, running time of Parent Process *running code.c*

[2]On Linux, there are six variants of exec(): execl(), execlp(), execle(), execv(), execvp(), and execvpe(). Read the man pages to learn more.

Do you ever wonder what the shell is?

It is a **program** that creates **processes**

Human's **interface** to the computer

```
while (1) {
    write(1, ”$ ", 2);
    read_command(command, args); // parse input
    if ((pid = fork()) == 0)     // child?
        execve(command, args, 0);
    else if (pid > 0)            // parent?
        wait (0);                // wait for child
    else
        perror("failed to fork()");
}
```

Please view this and the subsequent code as pseudo-code. A lot of the details are omitted.

Examples of using the shell

List current directory	`$ ls .`
Background process	`$ sleep 10` vs. `$ sleep 10 &`
Redirections	`$ ./first3 abcd efgh > foo`
Pipe	`$ ps xc	grep …`

How are these implemented?

```
$ sleep 10 &
```

```
while (1) {
    write(1, "$", 2);
    read_command(command, args); // parse input
    if ((pid = fork()) == 0) {
        execve(command, args, 0);
    }
    else if (pid > 0) {           // parent?
        if (foreground_process) {
         wait(0);                 //wait for child
        }
     }
    else
        perror("failed to fork()");
}
```

How are these implemented?

```
$ ./first3 abcd efgh > foo
```

```
while (1) {
    write(1, "$", 2);
    read_command(command, args); // parse input
    if ((pid = fork()) == 0) {
        if (output_redirected) {
          close(1);
          open("/tmp/foo", O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
         execve(command, args, 0);
    }
    else if (pid > 0)                // parent?
        wait (0);                    // wait for child
    else
        perror("failed to fork()");
}
```

Why does this implementation work?

```
$ ./first3 abcd efgh > foo
```

Redirection is fundamentally about manipulating file descriptors.

Every process starts with three file descriptors (fd):
0 (stdin): Input to the process
1 (stdout): Output from the process
2 (stderr): Error output from the process

Why does this implementation work?

```
$ ./first3 abcd efgh > foo
```

```
while (1) {
    write(1, "$", 2);
    read_command(command, args); // parse input
    if ((pid = fork()) == 0) {
        if (output_redirected) {
            close(1);
            open("/tmp/foo", O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
         execve(command, args, 0);
    }
    else if (pid > 0)                // parent?
        wait (0);                    // wait for child
    else
        perror("failed to fork()");
}
```

when command runs, fd 1 will refer to the redirected file

Why does this implementation work?

```
$ ./first3 abcd efgh > foo
```

```
while (1) {
    write(1, "$", 2);
    read_command(command, args); // parse input
    if ((pid = fork()) == 0) {
        if (output_redirected) {
            close(1);
            open("/tmp/foo", O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
        execve(command, args, 0);
    }
    else if (pid > 0)               // parent?
        wait (0);                   // wait for child
    else
        perror("failed to fork()");
}
```

We did not change ./first3! Only the environment changed.

How are these implemented?

Pipe

$ ps xc | grep …

```
void handle_pipeline(l_command, r_command) {
    int fdarray[2];
    pipe(fdarray);

    if ((pid = fork()) == 0) {  // child (left end of pipe)
        dup2(fdarray[1], 1); // make fd 1 the same as fdarray[1]
                             // which is the write end of the pipe
        close(fdarray[0]);
        close(fdarray[1]);
        parse(command1, args1, l_command);
        exec (command1, args1, 0);
    } else if (pid > 0) {   // parent (right end of pipe)
        dup2(fdarray[0], 0); // make fd 0 the same as fdarray[0]
                             // which is the read end of the pipe
        close(fdarray[0]);
        close(fdarray[1]);
        parse(command2, args2, r_command);
        exec(command2, args2, 0);
    }......
}
```

The power of fork() + exec()

fork() is simple: it takes no arguments

before exec(): we can manipulate environment, file descriptors, …

exec(): new process that may run in a different environment

This is a fundamental innovation in Unix, in contrasts with Windows…

CreateProcessA function (processthreadsapi.h)

Article • 02/08/2023 Feedback

In this article

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUserA function or CreateProcessWithLogonW function.

Syntax

C++ Copy

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa?redirectedfrom=MSDN

Takeaway: what is a good abstraction?

Simple but powerful

stdin (0), stdout (1), stderr (2)

file descriptors

fork/exec() separation

Very few mechanisms lead to a lot of possible functionality

To understand process…

How process see an abstract machine?

How OS implement the process abstraction?

What information of a process does OS keep track of?