Overview

Learning Objectives

In this section we will learn how to create and control processes. This should help us answer the following questions:

- What is **Process API**?
- What calls are related to Process API?
- How do we create and control processes?
- What interfaces should an OS have for process creation and control?

Intro to Process API

Process API is a collection of controls offered by the OS for controlling processes.

The interfaces described here should be available in any modern operating system to control processes.

- **Create:** An operating system has to allow the creation of new processes. When you input a command into the shell or double-click an icon, the OS creates a new process to start the program.
- **Destroy:** Systems also have to be able to destroy processes. Processes often exit when finished, but when they don't, the user may want to terminate them, so providing an interface to end a runaway process is quite helpful.
- **Wait:** Waiting for a process to finish running is sometimes helpful; so, some form of waiting interface is typically provided.
- **Miscellaneous Control:** Other controls beyond killing or waiting for a process are occasionally possible. For example, most operating systems allow you to **pause** and **resume** a process.
- **Status:** This information might include how long a process has been running and its current state.

With UNIX, you can create a new process using a pair of system calls: `fork()` and `exec()`.

A process can use `wait()` if it wants to wait for a process it has created to complete. Let's explore some of these interfaces in more detail.

##

Fill in the blank to complete the statement.

fork( )

The `fork()` system call is used to create a new process. The code to the left shows a program that creates a process.

Examine this code, then run the program by copying the code below into the terminal and pressing `return`.

```
./p1
```

The output should look similar to the following:

```
hello world (pid:286)
hello, I am parent of 287 (pid:286)
hello, I am child (pid:287)
```

- The process prints a hello world message when it starts
- That message also includes its **process identifier**, or **PID**.

The process has a **PID** of **286** (or something similar; on UNIX systems, the **PID** is used to name the process if someone wants to do something with it, such as (for example) terminate it.

- The process calls the `fork()` system call to create a new process. This new process is a **child** process and is almost an exact copy of the **parent** process that created it.
  - For the OS, this means there are now two copies of the program `p1` running, and both are about to return from the `fork()` system call.
  - The child also doesn't start at `main()`, but it starts working as if it had called `fork()` on its own. Notice the "Hello, world!" message only appeared once.

The child isn't an exact copy. Although it has its own address space, registers, a PC, etc., it returns a different value to the process that called `fork()`. The parent receives the PID of the new child process, but the child receives a return code of zero.

`p1.c` also does not provide a deterministic output. When the child process is created, there will be two active processes in the system: the parent and the child.

If we are running on a single CPU (for simplicity), then either the child or parent could run. This means the output is **non-deterministic**, and the processes may not always run in the same order!

Create a function, `fork_or_end` that:

- Creates a new process and stores the result into a variable
- Asserts that the result is 0 or greater, and
- Returns the result of the `fork()` call

wait( )

In some situations, it is useful for a parent to **wait** for their child process to finish running. We can do this with the `wait()` system call (or its more specific partner `waitpid()`).

The parent this program, `p2.c` calls `wait()` to delay its execution until the child finishes running. When the child is done, `wait()` returns to the parent.

Examine this code, then run the program by copying the code below into the terminal and pressing `return`.

```
./p2
```

The output should look similar to the following:

```
hello world (pid:487)
hello, I am child (pid:488)
hello, I am parent of 488 (wc:488) (pid:487)
```

Adding a `wait()` call to the code makes the output **deterministic**. Meaning we can be sure of what the output will be.

In this code, we can be sure the child will print first. How do we know? If the parent runs first, it will call `wait()` immediately. The `wait()` call won't return until the child has run and exited.

So, even if the parent runs first, it will politely wait for the child to finish, then `wait()` returns, and then the parent print its message.

Create a function, `wait_or_end()` that:

- Stores the result of a `wait` call into a variable, and
- Asserts that the result is not equal to 0

exec( )

The `exec()` system call is an important part of the process creation API. `exec()` is useful when you want to run a program that is different from the program that is calling it.

Using `fork()` in `p2.c` is only useful if you want to keep multiple copies of the same program running. In this example, we use `exec()` to run a different program.

By calling `execvp()`, the child process runs program `wc`, a word count program.`wc` runs on source file `p3.c`, and reports how many lines, words, and bytes are in the file:

Examine this code, then run the program by copying the code below into the terminal and pressing `return`.

```
./p3
```

The output should look similar to the following:

```
hello world (pid:622)
hello, I am child (pid:623)
 32 123 966 p3.c
hello, I am parent of 623 (wc:623) (pid:622)
```

The `execvp()` call in this example:

- Take the name of an executable (`wc`) and some arguments (`p3.c`)
- Loads code and static data from that executable
- Overwrites its current code segment and current static data with it
- Reinitializes the heap, stack and other parts of the memory space of the program.
- It then runs that program, passing in any arguments as `argv` to that process.

Instead of creating a new process, it transforms existing program (formerly `p3`) into a new program (`wc`).

After the `exec()` in the child, it is almost as if `p3.c` never ran. A successful call to `exec()` never returns.

Why The API?

Why would we need this kind of interface for simply creating new processes?

Separating `fork()` and `exec()` is important in building a UNIX shell. Doing this lets the shell run code **after** calling `fork()` but **before** calling `exec()`.

This allows for setting up the environment for the program that's about to be run.

The shell is only a user-made program. It:
* Shows you a **prompt**
* Waits for you to type something into it.

You type in a command and, normally, the shell:

- Finds where the program lives
- Calls `fork()` to create a new process to run the command
- calls some form of `exec()` to run the command
- calls `wait()` to wait for the command to finish.

When the child is finished running, the shell:

- returns from `wait()`
- prints out the prompt again, waiting for your next command.

Redirection & Pipes

Separating `fork()` and `exec()` lets the shell do a host of useful things.

Redirection

Let’s explore an example. Copy and paste the following into the terminal and run the command.

```
wc p3.c > newfile.txt
```

The output of the program `wc` is **redirected** with the greater than sign (>), into the output file `newfile.txt`. This send any output from the soon-to-run program, `wc`,into the file instead of the screen.

We can see this in the code sample to the left, `p4.c`. Run the program to see the output.

```
./p4
```

It doesn’t look like much when run, but the program:
* Called `fork()` to create a new child process
* Then, ran `wc` using a call to `execvp()`

You don’t see output printed to the screen because it’s been redirected to the file `p4.output`.

When we `cat` the output file, we can see all the expected output from running `wc` printed to the terminal.

Pipes

UNIX pipes have a similar implementation, but uses the `pipe()` system call. When this is done, the output of one process is connected to an in-kernel **pipe**, and the input of another process is connected to that same pipe. So the output of one process can be used as the input to the next process.

We can use this to string together long and useful chains of commands.

For example, we can use pipes and the utilities `grep` and `wc` to look for a word in a file, and then count how many time it occurs. Type the following into the terminal and run to see the results.

```
grep -o foo footester.txt | wc -l
```

Process Control

Apart from `fork()`, `exec()`, and `wait()`, there are a several of other ways to interact with UNIX processes.

- The `kill()` system call can be used to send pause, die, and other useful command signals to a process.
- In most UNIX shells, certain keystroke combinations are configured to deliver a specific signal to the process that is currently running.
    - `control-c` sends a `SIGINT` (interrupt) to the process (normally terminating it)
    - `control-z` sends a `SIGTSTP` (stop) signal and pauses the process in mid-execution (you can resume it later with a command,like the `fg` built-in command found in many shells).

To deliver external events to processes, the signals subsystem provides an infrastructure for receiving and processing signals within processes, as well as sending signals to individual processes and **process groups**.

To communicate this way, a process must use the `signal()` system call to “catch” various signals. This makes sure that when a signal is received, the process stops normal execution and runs a specific piece of code.

##

Fill in the blanks to complete the statements below.

Users

Who can send a signal to a process and who can't?

Generally, systems we use can have multiple **users** using them at the same time. If a user can send terminating signals like `SIGINT` willy nilly to any process, then the system's security and usability will be compromised.

Modern systems are deeply concerned with the concept of a **user**. The **user** enters a password when logging in to establish credentials for gaining access to system resources. After that, the user may launch one or more programs and fully control them (pause, kill, etc.).

It is the job of the operating system to **allocate resources** (such as CPU, memory, and disk) to each user (and their processes) in order to meet overall system goals.

▼ **Superusers**

A system needs a user who can administer it without being restricted. A user should be allowed to kill a random process (e.g., if it is abusing the system) even if they did not initiate it. A capable user should be able to conduct commands like shutdown (which, surprise, shuts down the system).

In UNIX-based systems, the superuser has these powers (sometimes called root). The superuser can kill other users' processes. Becoming root gives you access to all of the destructive powers of the computing world, but it also increases security (and avoids costly mistakes).

Useful Tools

There are many command-line tools that are useful.

- `ps` command allows you to see what processes are running; you can read the **man pages** for useful flags to pass to ps.
- `top` displays which processes are running on the system and how much CPU and other resources they consume.
- `kill` and `killall` can send unpredictable signals to processes. Use these carefully. If you kill your window manager accidentally, you may find your computer quite difficult to operate.

You can also use many different kinds of CPU meters to quickly see the load on your system.

Summary

In this section, we explored various APIs related to UNIX process creation, including `fork()`, `exec()`, and `wait()`.

- In most systems, each process is identified by a number called a **Process ID (PID)**.
- UNIX systems use the `fork()` system call to start a new process.
- Created processes are called **child** processes; the creator is called the **parent**. The child process is a near-identical copy of the parent.
- Using `wait()`, a parent can wait for its child to finish its execution.
- `exec()` Lets the child process gain independence from the parent and execute a new program.
- **Signals** control processes by stopping, continuing, or terminating them.
- The operating system allows multiple **users** on the system and makes sure users can only control their own processes.
- A **superuser** can control all processes, but should be used with caution for security reasons.

Lab Intro

`generator.py` is a program that creates randomly generated C programs that use `fork()` in different ways.

Several flags control the way the code is generated.

- `-s`: **SEED** - different seeds create different programs
- `-n`: **NUM_ACTIONS** - how many actions (fork, wait) a program should include
- `-f`: **FORK_CHANCE** - - the chance, from 1-99 percent, that a `fork()` will be added
- `-w`: **WAIT_CHANCE** - the chance, from 1-99 percent, that a wait() will be added.
    - There has to be an outstanding fork for this to be called
- `-e` **EXIT_CHANCE** - same, but the chances the process will exit
- `-S` **MAX_SLEEP_TIME** - the maximum sleep time that is chosen when adding sleeps into the code

There are also flags that control which C files get created for the code:

- `-r`: **READABLE** - The file shown to you and optimized for readability
- `-R`: **RUNNABLE** - The file that will be compiled and run; it is identical to the above but adds print statements and such

The `-A` flag lets you specify the program exactly:

```
./generator.py -A "fork b,1 {} wait"
```

This command creates the default process (“a”), which then creates “b” that sleeps for 1 second, and `a` waits for `b` to finish.

Run it in the terminal to see the resulting C program.

We’ll use this to explore how `fork()` works.

Lab 1

Start by using the following command in the terminal to run the program.

```
./generator.py -n 1 -s 0
```

The result is a randomly generated C program similar to this:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>


void wait_or_die() {     int rc = wait(NULL);     assert(rc > 0); }
```

Let’s explore this code:

- `wait_or_die` and `fork_or_die` are just wrappers holding the `wait()` and `fork()` system calls.
    - These either:
        - Succeed, or
        - Find an error by checking the return code, `rc`, and exit using the `assert()` call

▼ assert()

This is a function that checks if an expression is true. If it is true, `assert()`

We can find the main process, process `a` in `main()`. This process:

- Calls `fork_or_die()` to create a child process.
- Then, waits for the child process to complete by calling `wait_or_die()`

The child process, `b`:
* Sleeps for 6 seconds
* Exits

Can you predict what the output will look like when the program runs?

Lab 2

Let’s see by running the program with the `-c` flag:

```
./generator.py -n 1 -s 0 -c
```

The output should looks similar to the following:

```
0 a+
0 a->b
6      b+
6      b-
6 a<-b
```

The first column shows is the time when events take place. In this example, two things happen at time 0:

- Process `a` starts running: `a+`
- Process `a` forks and creates process `b`: `a->b`

Then, `b` starts running and immediately sleeps for 6 seconds (`sleep(6)`). When it is finished sleeping, at time 6:

- `b` Prints that it was created:`b+`
- Exits less than one second later: `b-`

Once `b` exits:

- The `wait_or_die()` call in the parent process, `a`, returns and prints that the return has happened: `a<-b`

Explore

More elaborate examples can be created using this generator. For example:

`-A "fork b,1 {} fork c,2 {} wait wait"` - Creates a C program where process “a” creates two processes, “b” and “c”, and then waits for both.

To create the program **and** run it, use the `-c` flag at the end: `-A "fork b,1 {} fork c,2 {} wait wait" -c`

Explore this generator with different flags and arguments to broaden your understanding of creating new processes.