System Security

The Rabbit Hole into How your Computer Runs Programs

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2023 / 2024

Part I

The Abstraction: The Process

What is a Process?

- The **Process** is a running program.
  - A program is just a bunch of instructions sitting on the disk.
- A typical system may be seemingly running tens or even hundreds of processes at the same time.
  - Although there are only a few physical CPUs available, **how can the OS provide the illusion of a nearly-endless supply of said CPUs?**

Time Sharing (and Space Sharing)

- Time Sharing is a basic technique used by an OS to share a resource.
  - By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question (e.g., the CPU, or a network link) can be shared by many.

- Space Sharing is when a resource is divided (in space) among those who wish to use it.
  - For example, disk space is naturally a space-shared resource; once a block is assigned to a file, it is normally not assigned to another file until the user deletes the original file.

Mechanisms and Policies

- To implement virtualization of the CPU, the OS will need both some low-level machinery and some high-level intelligence.

- We call the low-level machinery mechanisms;
  - **Mechanisms** are low-level methods or protocols that implement a needed piece of functionality.
  - Context switch is a mechanism.

- On top of these mechanisms resides some of the intelligence in the OS, in the form of policies.
  - **Policies** are algorithms for making some kind of decision within the OS.
  - The scheduling policy in the OS will make the decision about which program should run first by the OS.

Separate Policy and Mechanism

- In many operating systems, a common design paradigm is to separate high-level policies from their low-level mechanisms.

- You can think of the mechanism as providing the answer to a **how** question about a system;
  - for example, how does an operating system perform a context switch?

- The policy provides the answer to a which question;
  - for example, which process should the operating system run right now?

- Separating the two allows one easily to change policies without having to rethink the mechanism,
  - and is thus a form of **modularity**, a general software design principle.

Machine State

- **Machine State:** what a program can read or update when it is running.
  - At any given time, what parts of the machine are important to the execution of this program?

- **Address Space:** the memory that the process can address
  - Instructions lie in memory; the data that the running program reads and writes sits in memory as well.

- **Registers:**
  - many instructions explicitly read or update registers.
  - **Program Counter (PC)** (sometimes called the instruction pointer or IP) tells us which instruction of the program is currently being executed
  - **Stack Pointer** is used to manage the stack for function parameters, local variables, and return addresses.

- **I/O Information.**
  - May include a list of the files the process currently has open.

Process API 1/2

- **Create:**
  - When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.

- **Destroy (forcefully):**
  - Of course, many processes will run and just exit by themselves when complete; when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.

- **Wait:**
  - wait for a process to stop running.

Process API 2/2

- **Misc Control:**
  - most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).

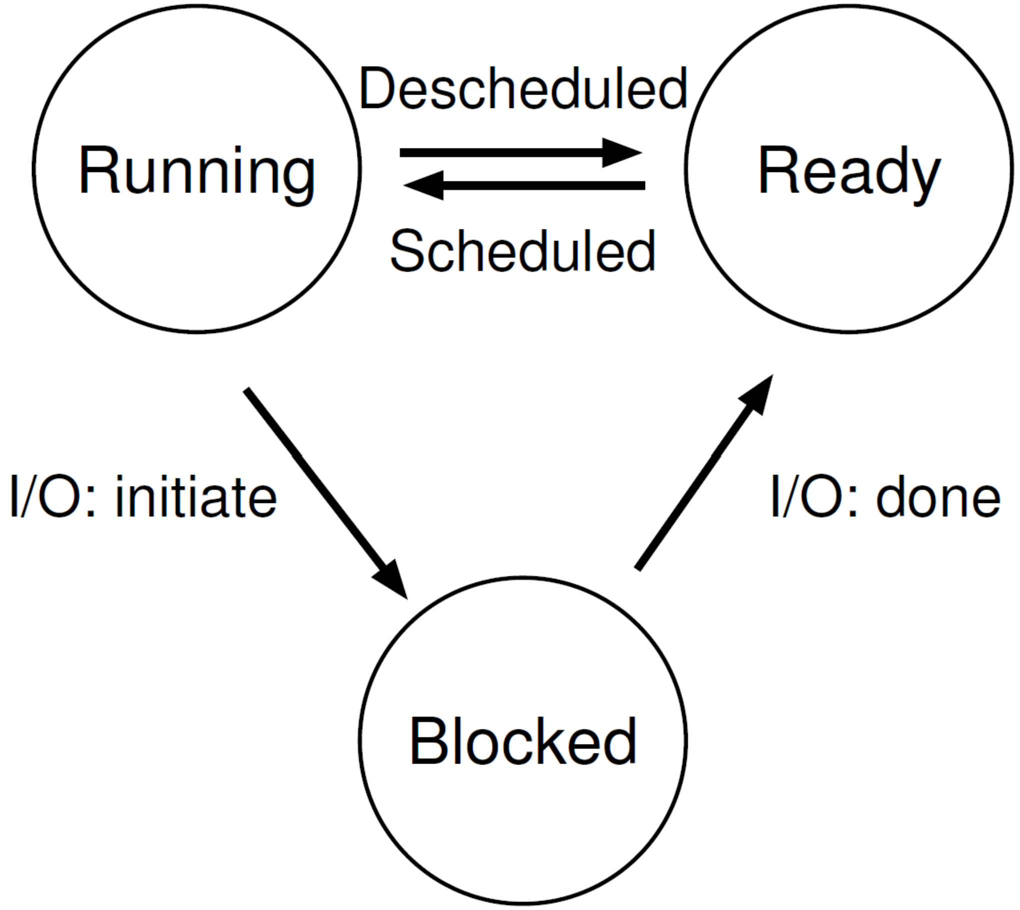
- **Status:**
  - status information about a process as well, such as how long it has run for, or what state it is in.

Process States

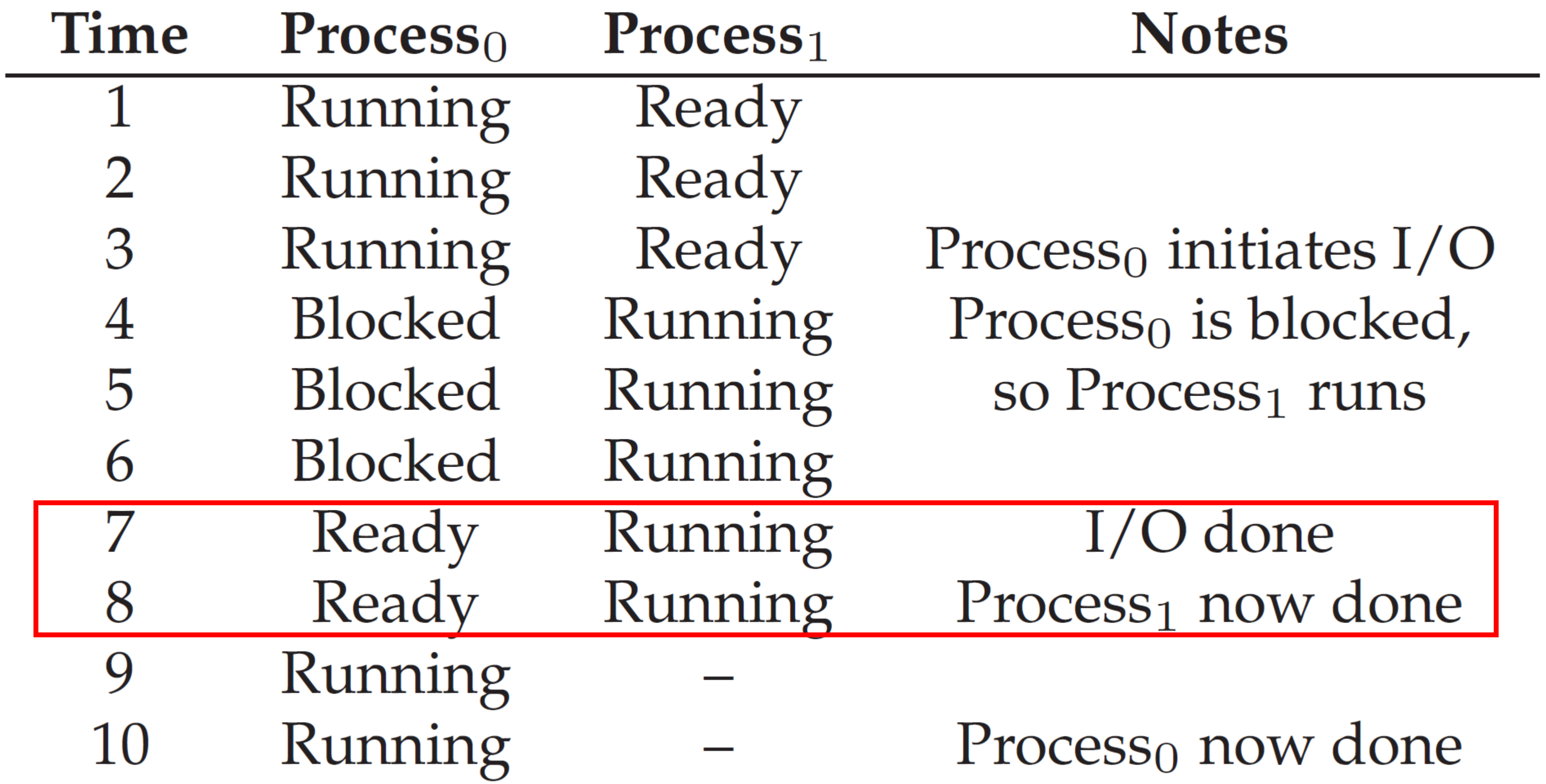
- **Running:**
  - A process is running on a processor. This means it is executing instructions.
- **Ready:**
  - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked:**
  - A process has performed some kind of operation that makes it not ready to run until some other event takes place.
  - A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition

- Being moved from ready to running means the process has been **scheduled**;
- Being moved from running to ready means the process has been **descheduled**.



Tracing Process States



Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Process Creation

- OS creates some internal data structures.
- OS allocates an address space
  - Loads code, data from disk
  - Creates runtime stack, heap
- OS opens basic files (STDIN, STDOUT, STDERR)
- OS initializes CPU registers

Data Structure

- **Process List (task list):** keep track of all the processes in the system.
  - Which process is currently running.
  - All processes that are ready.
  - All blocked processes.

- **Process Control Block (PCB) or process descriptor:**
  - The C structure that contains information about each process.

PCB

- PCB is a data structure maintained for every process it manages.
  - process state (e.g. running)
  - process identifier (PID)
  - program counter
  - stack pointer
  - registers
  - Address space
  - list of open files
  - …

- Fields
  - Certain fields are updated when the process state changes.
  - Certain fields change too frequently.

Proc File System 1/2

- Proc file system (procfs) is a virtual file system created on the fly when the system boots.
  - It contains useful information about the system.

Directory	Description
/proc/crypto	list of available cryptographic modules
/proc/filesystems	list of the file systems supported by the kernel
/proc/meminfo	summary of how the kernel is managing its memory.
/proc/version	containing the Linux kernel version, distribution number, gcc version number (used to build the kernel) and any other pertinent information relating to the version of the kernel currently running

Proc File System 2/2

- It contains useful information about the processes that are currently running.

Directory	Description
/proc/PID/cmdline	Command line arguments.
/proc/PID/environ	Values of environment variables.
/proc/PID/exe	Link to the executable of this process.
/proc/PID/maps	Memory maps to executables and library files.
/proc/PID/mem	Memory held by this process.
/proc/PID/statm	Process memory status information.
/proc/PID/status	Process status in human readable form.
/proc/PID/fd	Directory, which contains all file descriptors.

Part II

Process API

The CRUX

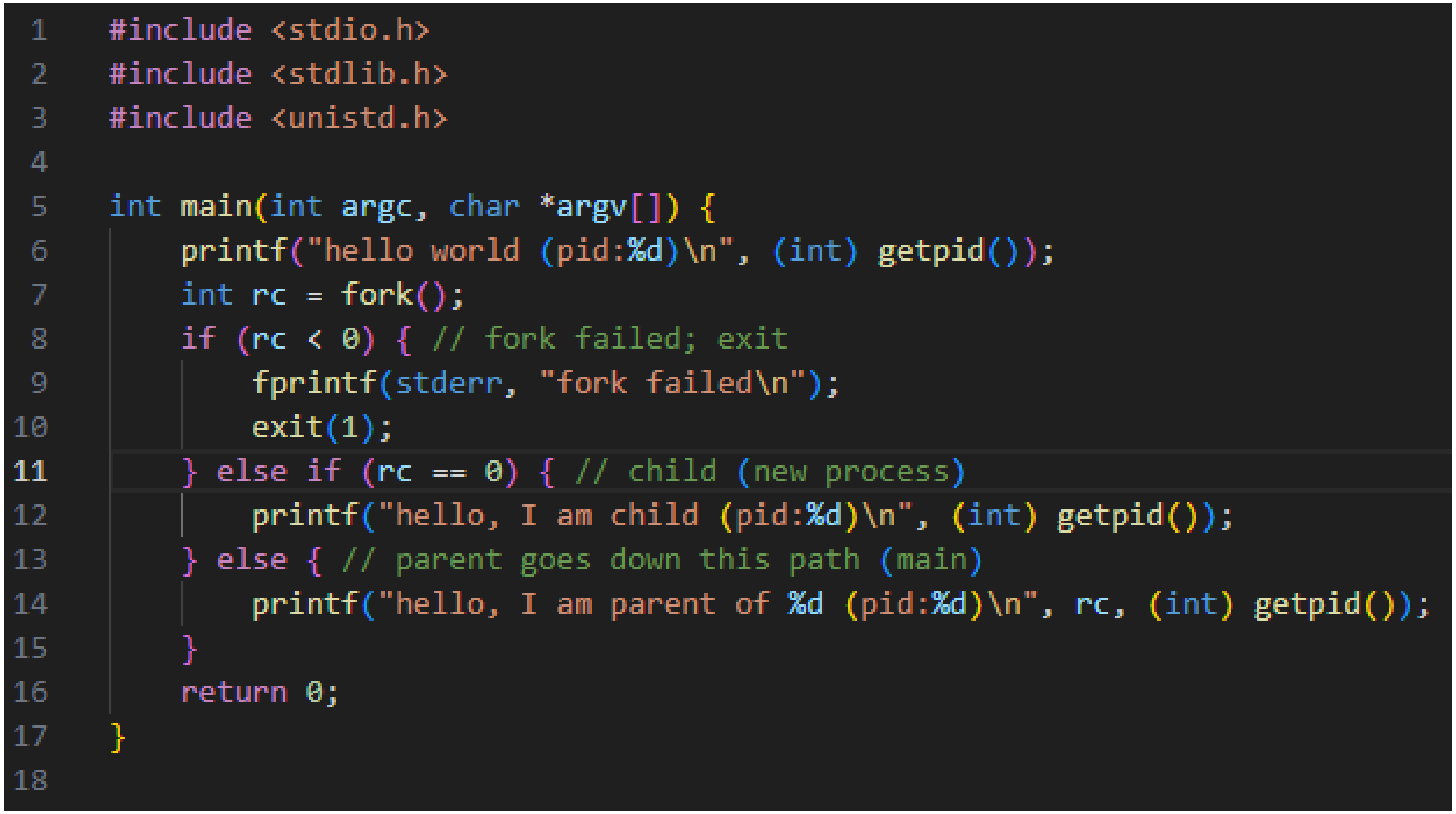
- **What interfaces should the OS present for process creation and control?**

The fork() System Call

- **The fork()** system call is used to create a new process.

- The odd part: the process that is created is an (almost) exact copy of the calling process.
  - It has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of fork() is different.

Example of Fork



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

Question

- **Who is executed first: the parent or the child?**

The wait() System Call

- **The wait()**, and sibling **waitpid()**, system call makes the parent wait for a child process to finish what it has been doing.

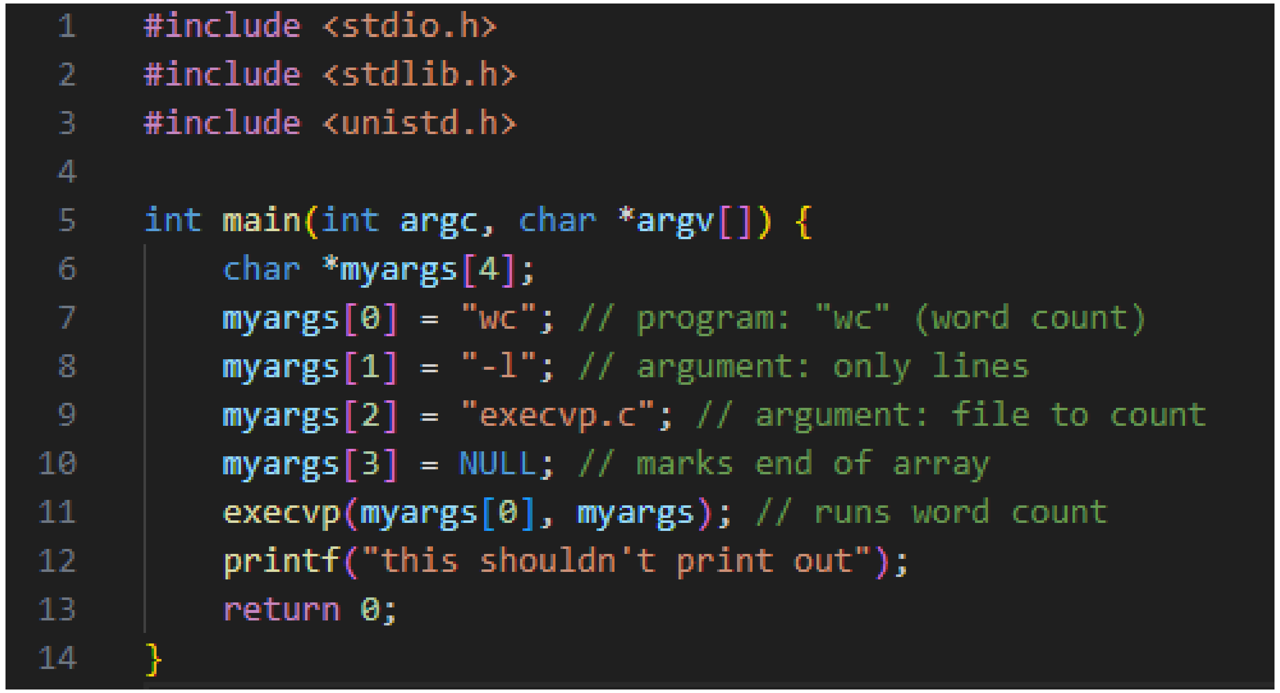
- Now we can make the child always run first.

The exec() System Call

- **The exec()** family, e.g. **execvp()**, system call runs a program that is different from the calling program.

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

Example of execvp



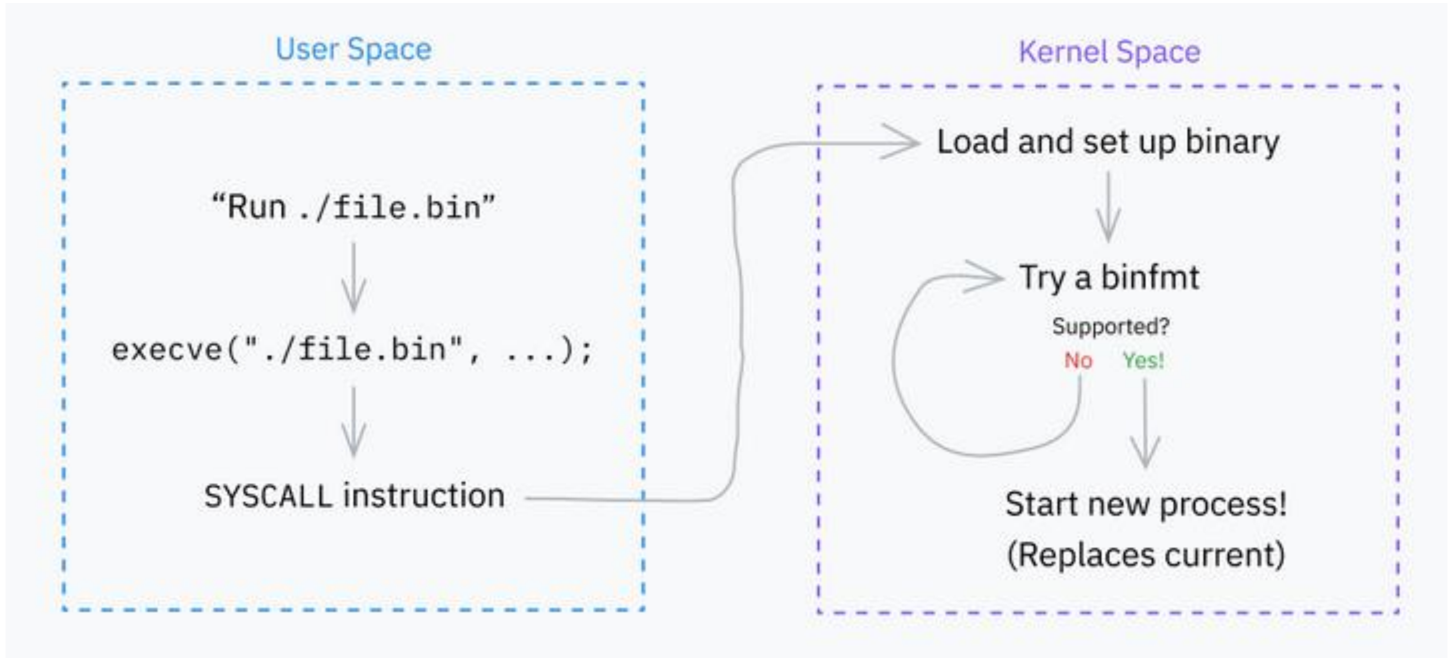
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    char *myargs[4];
    myargs[0] = "wc"; // program: "wc" (word count)
    myargs[1] = "-l"; // argument: only lines
    myargs[2] = "execvp.c"; // argument: file to count
    myargs[3] = NULL; // marks end of array
    execvp(myargs[0], myargs); // runs word count
    printf("this shouldn't print out");
    return 0;
}
```

More About exec()

- **The exec()** family
  - loads code (and static data) from that executable
  - overwrites its current code segment (and current static data) with it;
  - the heap and stack and other parts of the memory space of the program are re-initialized.

- The exec() does not create a new process;
  - it transforms the currently running program into a different running program.
  - A successful call to exec() never returns.

Basic Behavior of Exec Syscalls



Process Control and Users

- **The kill()** system call sends signals to a process.
  - E.g., directives to pause, die, etc.
  - Keystroke combinations are configured to deliver a specific signal to the currently running process: control-c (SIGNINT) and control-z (SIGNSTP)

- This naturally raises the question: who can send a signal to a process, and who cannot?

Part III

Format Highlight

Binfmts

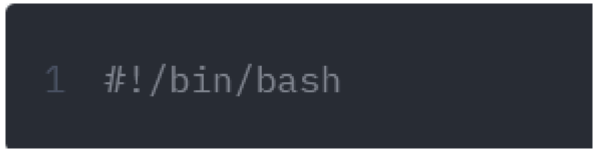
- A **binary format** is basically a data structure responsible for executing program files--the ones marked with execute permission.

- One of the kernel major job is to understand the program format.
- This involves
  - Looking for magic number in the first 256 bytes of the program.
  - Decode the start of the program (again the 256 bytes).
  - Checking the file extension.

- If no binary format is able to run the executable file, the system call returns the ENOEXEC error code (``Exec format error'').

Example: Scripts

- **Shebang**: the line at the start of some scripts that specifies the path to the interpreter.



```
#!/bin/bash
```

- If the file does start with a shebang, the binfmt handler then reads the interpreter path and any space-separated arguments after the path. It stops when it hits either a newline or the end of the buffer.

- Shebangs
  - are handled by the kernel.
  - pull from only the first 256 bytes instead of loading the whole file.

Miscellaneous Interpreters

- You can register interpreters for various binary formats
  - based on a magic number or their file extension,
  - cause the appropriate interpreter to be invoked whenever a matching file is executed.

- There’s no way to specify interpreter arguments, so a wrapper script is needed if those are desired.

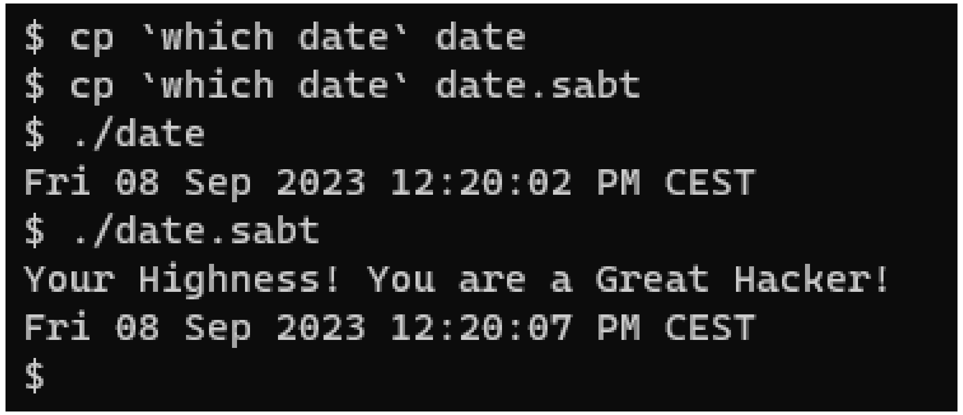
💿 Example for the .sabt extension

💿 Write a wrapper.

💿 Register the sabt format with its wrapper

- **update-binfmts** in Debian.

💿 Output



```
$ cp `which date` date
$ cp `which date` date.sabt
$ ./date
Fri 08 Sep 2023 12:20:02 PM CEST
$ ./date.sabt
Your Highness! You are a Great Hacker!
Fri 08 Sep 2023 12:20:07 PM CEST
$
```

One Last Wonky Thing

- An exec syscall will always end up in one of two paths:
  - It will eventually reach an executable binary format that it understands, perhaps after several layers of script interpreters, and run that code,
  - Or, it will exhaust all its options and return an error code to the calling program.

- You might've noticed that shell scripts run from a terminal still execute if they don't have a shebang line or .sh extension.
  - The kernel's format handlers should have no clear way of detecting shell scripts without any discernible label!
  - Why does the shell script run as a shell script?

Shell Script with no shebang

- When you execute a file using a shell and the exec syscall fails, most shells will retry executing the file as a shell script by executing a shell with the filename as the first argument.

- This behavior is so common because it’s specified in POSIX.

https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/utilities/V3_chap02.html#tag_18_09_01_01

Part IV

Run a Program

Shell Demystified 1/2

- The shell is just a user program:
  - It shows you a prompt, while waiting for you to type something into it,
  - You then type a command into it,
  - It calls fork() to create a new child process to run the command,
  - It calls some variant of exec() to run the command,
  - It waits for the command to complete by calling wait().
  - When the child completes, the shell returns from wait() and prints out a prompt again, ready for your next command.

Shell Demystified 2/2

- The kernel parses to find information on how to load the program and where to place its code and data within the new virtual memory mapping.

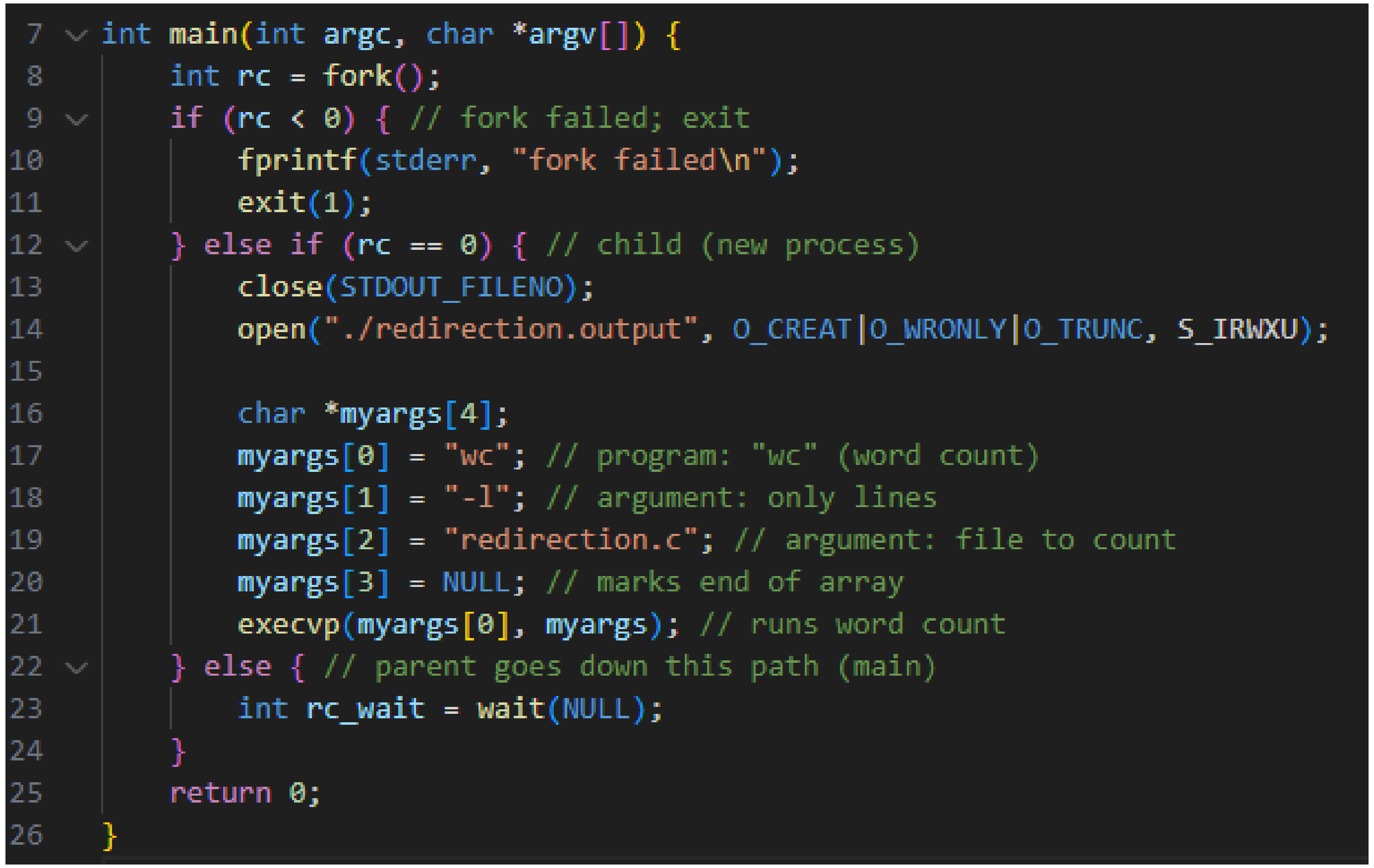
- The kernel can then load the program’s virtual memory mapping and return to userland with the program running,
  - which really means setting the CPU’s instruction pointer to the start of the new program code in virtual memory.

Fork and exec Separated

- This lets the shell run code after the call to fork() but before the call to exec();
  - this code can alter the environment of the about-to-be-run program,
  - and thus enables a variety of interesting features to be readily built.

- Example, output redirection
  - when the child is created, before calling exec(), the shell closes standard output and opens <file>.
  - By doing so, any output from the soon-to-be-running program are sent to the file instead of the screen.

Naive Redirection



```c
int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        close(STDOUT_FILENO);
        open("./redirection.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        char *myargs[4];
        myargs[0] = "wc"; // program: "wc" (word count)
        myargs[1] = "-l"; // argument: only lines
        myargs[2] = "redirection.c"; // argument: file to count
        myargs[3] = NULL; // marks end of array
        execvp(myargs[0], myargs); // runs word count
    } else { // parent goes down this path (main)
        int rc_wait = wait(NULL);
    }
    return 0;
}
```

What About Pipes?

- 📀 See Lab Session ☺