Lecture 4: Programs and Processes

Definition, Creation, and Management

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Recap

- A program can be viewed as a **state machine**, and running code means state transitions.
- Our code itself can only change its own **internal state**.
- To affect anything **outside** the program, we must use **system calls** to go through the operating system.

Abstractions

This gives us a useful abstraction:

Program

Computation + System Calls

Operating System

System Calls + Objects

System Call Interface:

- The system-call interface links programs and the OS.
- In UNIX, **everything is a file**.
- All resources are accessed and controlled through **file descriptors (FDs)**.
- **Pipes** allow us to connect programs by wiring one FD's output to another FD's input.

Transition to Today

- When we discussed `FD`, `dup`, and shell, we were talking about **processes**.
- Even in the state-machine view, the key question is: *what is the state?*
- The answer is: the **process**.

We usually write a single program that starts from the main function, runs through, and ends at return. The operating system must provide us with the ability to create processes—otherwise, how could so many programs run inside the OS?

Today's topic: What a process is, and how to create and manage processes.

Porgrams and Processes

Virtualization

- One of the most fundamental abstractions that the OS provides to users: the **process**.
- Treat the physical computer as if it were a set of “virtual computers.”
  - Each program runs as if it has its own dedicated machine.

Program vs. Process

```c
#include <unistd.h>

int main() {
    while (1) {
        write(1, "Hello, World!\n", 13);
    }
}
```

- A **program** is a static description of a state machine.
  - It describes all possible states of the program.
- When a program (state) is running, it becomes a **process**.
  - It asks OS for system calls.
  - It asks OS to allocate resources such as memory, registers and CPU time.
  - It has its own state, which includes the program counter, registers, memory, and open file descriptors.
  - Multiple processes can run the same program.

Process State in the OS

- A program can directly manage its **internal state** (e.g., variables, control flow).
- Some parts of the process state are maintained **outside** the program, by the OS.
  - Example: the **Program Counter (PC) –** managed by the CPU.
  - Example: the **Process ID (PID) –** assigned and managed by the OS.
- To access such information, a program must use **system calls**.
  - e.g., `getpid()` returns the current process ID.

```
ps ax | grep 12345 # replace 12345 with your PID
```

- `ps ax`: list all processes, including those without a terminal.
- `| grep`: filter the lines to keep your PID.

/proc

- A kernel-provided **virtual filesystem** exposing process and system info.
- Each running process has a directory: `/proc/<pid>`.
- Common entries: `status`, `cmdline`, `cwd`, `exe`, `fd/`, `environ`, `task/`.
- Global info examples: `/proc/cpuinfo`, `/proc/meminfo`.
- Files under `/proc` are generated by the kernel. Regular files live on real filesystems.

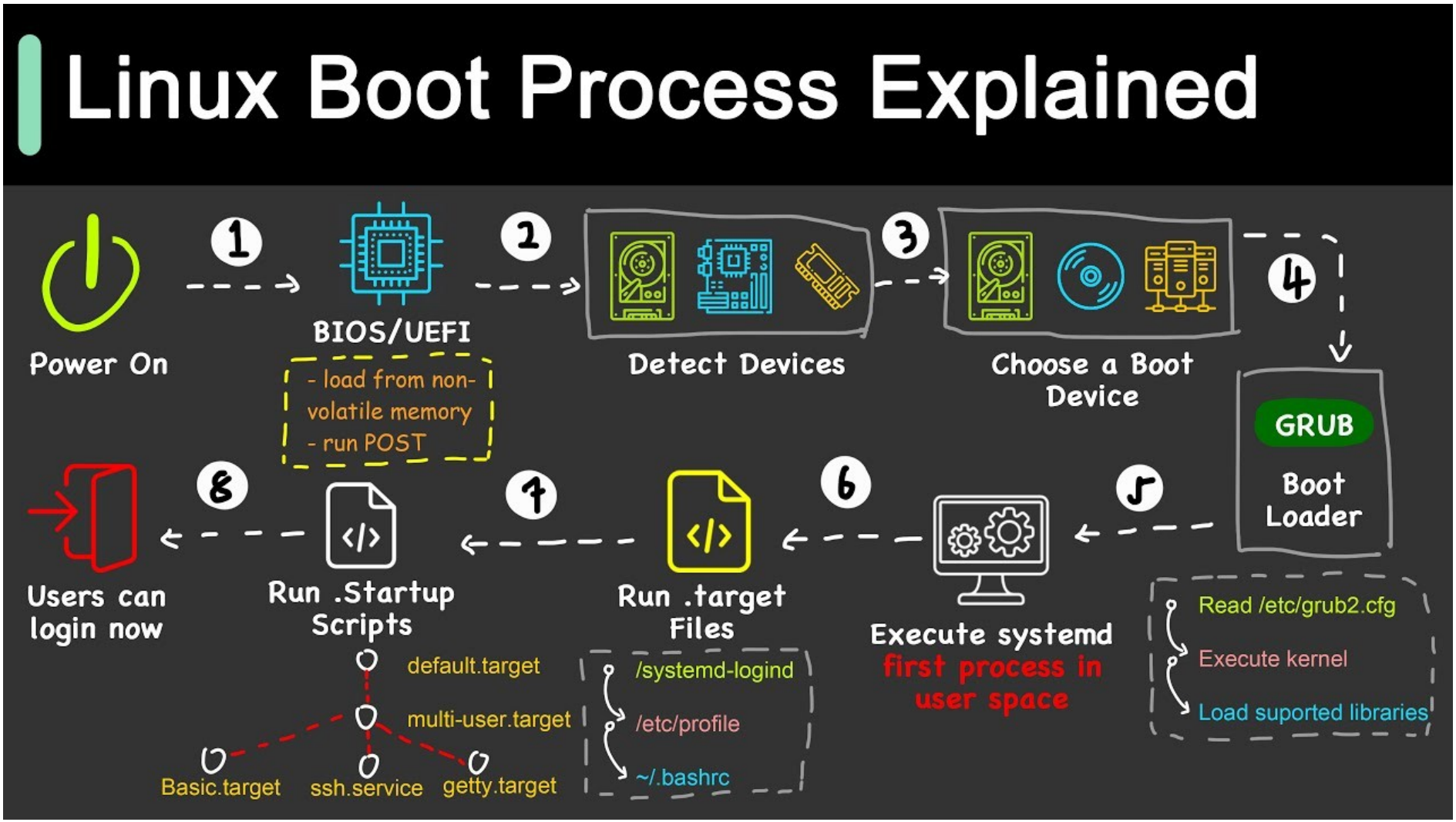
Try it: proc

In-Class Quiz: PID Limits and Wraparound

You can continue to ask ChatGPT:

Prompt: I see process IDs increase when new processes start. Are PIDs limited? If yes, do they wrap around and get reused? What is the actual limit on Linux and how can I check it?

Creation


Linux Boot Process Explained
1
Power On
BIOS/UEFI
- load from non-volatile memory
- run POST
2
Detect Devices
3
Choose a Boot Device
4
GRUB
Boot Loader
5
Read /etc/grub2.cfg
Execute kernel
Load suported libraries
Execute systemd
first process in user space
6
Run .target Files
/systemd-logind
/etc/profile
~/.bashrc
7
Run .Startup Scripts
default.target
multi-user.target
Basic.target
ssh.service
getty.target
8
Users can login now

How the First Process Is Born (Overview)

- 1) Power on $\rightarrow$ Firmware (BIOS/UEFI)
- 2) Firmware detects hardware devices
- 3) Firmware chooses a boot device
- 4) Boot loader (e.g., GRUB) starts
- 5) Boot loader loads the Kernel and initramfs
- 6) Kernel takes control and initializes hardware
- 7) Kernel executes **PID 1** (`/sbin/init`, usually `systemd`)
- 8) `systemd` starts services, and the login prompt is ready

Step 1: Firmware

- A low-level software embedded in a hardware chip on the motherboard. It's the very first code the CPU executes after power-on.
- **BIOS vs. UEFI (Two Main Firmware Interfaces)**
  - **BIOS (Basic Input/Output System)**: The legacy firmware interface. It has limited features, runs in 16-bit mode, and has limitations on disk size (¡2TB) and partitions.
  - **UEFI (Unified Extensible Firmware Interface)**: The modern successor. It is powerful, featuring a graphical interface, networking capabilities, Secure Boot, and overcomes the limitations of BIOS.
- **Core Tasks of the Firmware**
  - **POST (Power-On Self-Test)**: It checks that critical hardware like the CPU, RAM, and graphics card are functioning correctly.
  - **Initialize Hardware**: It prepares the most basic hardware environment for the next boot stage.
  - **Select Boot Device**: It finds a bootable device according to a preset order (e.g., HDD, USB, Network) and loads the boot loader from it.

Step 2: The Boot Loader

- A small program that runs after the firmware and before the operating system kernel. Its core mission is to find the OS kernel, load it into memory, and then hand over control to it.
- **GRUB (GRand Unified Bootloader)**
  - The most common boot loader for Linux systems. It is very powerful, supports multi-booting, and provides a menu for the user to select an OS or kernel version.
  - **Where is it located?**
    - For legacy **BIOS** systems, the initial boot code resides in the **Master Boot Record (MBR)**, the first 512-byte sector of the disk. The MBR's job is to load the next stage of GRUB.
    - For modern **UEFI** systems, which use the **GUID Partition Table (GPT)**, GRUB is loaded as an EFI application from a special partition called the EFI System Partition (ESP).
- **Key Operations of GRUB**
  - **Loads the Kernel and initramfs**:
    - **initramfs (Initial RAM Filesystem)**: A temporary, in-memory root filesystem. It contains essential drivers (e.g., for SATA or NVMe controllers) that the kernel needs to access the actual hard disk. Without it, the kernel might not be able to find and mount the real root filesystem.

Step 3: Kernel Startup

- **Initialize Core Subsystems**: It sets up memory management (e.g., paging), the process scheduler, device driver frameworks, etc.
- **Mounts the Root Filesystem**:
  - First, it uses the drivers in the `initramfs` (loaded by GRUB) to detect the hard disk.
  - Then, it mounts the real root filesystem (`/`) from the disk. From this point on, the system can access all the files on the hard drive.
- **Creates Kernel Threads**: It starts background processes that run only in kernel space to handle system tasks, such as scheduling (`kthreadd`), interrupt handling, etc.
- **Prepares to Start the First User-Space Process**: Once all low-level setup is complete, the kernel's final step is to create the very first "normal" process in the system.

Step 4: The Birth of `PID 1`

- **PID 1: The "Ancestor" of User Space**
  - The first user-space process created by the kernel. Its Process ID is always 1.
  - It is the ancestor of all other user processes. If PID 1 terminates unexpectedly, the entire system will experience a Kernel Panic.
- **`/sbin/init` and `systemd`**
  - The program the kernel executes is hardcoded to the path `/sbin/init`.
  - In modern Linux distributions, `/sbin/init` is usually a symbolic link to `systemd`.
  - `systemd` is a modern init system and service manager, responsible for starting and managing all system services.
- **PID 1's Special Duty: Adopting Orphans**
  - When a parent process exits before its child, the child becomes an "orphan process". PID 1 automatically "adopts" these orphans and cleans up their resources after they terminate, which prevents them from becoming "zombie processes".

Step 5: Userspace Is Ready

- **The Job of `systemd`**
  - **Starts Services**: `systemd` reads configuration files and starts required system services in parallel according to their dependencies, such as networking (`networkd`), SSH (`sshd`), etc.
- **Prepares for User Login**
  - `systemd` starts `getty` processes. **getty** (get teletype) monitors a terminal line (tty), displays a login prompt, and then executes the `login` program to verify credentials.
- **User Login and the Shell**
  - After successful authentication, the `login` program starts a shell (e.g., `bash`) for the user.
  - The shell executes startup scripts (like `/etc/profile`, `~/.bashrc`) to configure the user's environment (e.g., the `PATH` variable).
  - From now on, every command you type (like `ls` or `docker run`) creates a new process, which is a child of your shell process.

Booting Sequence: From Reset to PID 1

1. Power on; CPU jumps to firmware in ROM (BIOS or UEFI).
2. Firmware tests hardware and picks a boot device.
3. Boot loader (e.g., GRUB) is loaded from the device.
4. Boot loader loads the Linux kernel and initramfs into memory, then jumps to the kernel.
5. Kernel initializes memory and drivers, mounts initramfs, then mounts the real root filesystem.
6. Kernel starts the first user-space process: **PID 1** (`/sbin/init`, usually `systemd`).
7. PID 1 starts services and login; your shell runs.

Can We Really See Every Booting Instruction?

- *"Talk is cheap. Show me the code."* — Linus Torvalds
- **Computer system axiom**
  - If you can imagine it, someone has already done it.
- Simulation option: QEMU
  - Created by Fabrice Bellard
  - A fast and portable dynamic translator (USENIX ATC'05)
  - Powers Android emulators and many tools
- Real machine option: JTAG debugger
  - Hardware debug registers and pins
  - Integrates with `gdb`

A Side Story: The CIH "Hardware" Virus (1998)

- We normally think of viruses as a software problem. You can always format the disk and reinstall the OS to fix it. But what if a virus could attack the computer's firmware (BIOS)?
- **The Hidden Flaw**: Normally, the BIOS chip is read-only. But on motherboards of that era, manufacturers left a "secret sequence" to unlock it for updates. This secret was poorly guarded and documented.
- **The Attack**: The CIH virus, running as a normal program, used this sequence to unlock the BIOS. Then, it completely erased the firmware, filling it with garbage.

The CIH ”Hardware” Virus (Cont.)

- **The Result**: A hard-bricked PC. Without its basic instructions in the BIOS, the computer couldn’t even start.
- This is why it became known as a **hardware-level virus**. The only fix was not software, but a physical one: using a special programmer to re-flash the chip, or in many cases, replacing the motherboard entirely.
- Chen Ing-Hau, the author of CIH, was arrested but not convicted.

Process (State Machine) Management API

Process-Management System Calls

Operating system = manager of state machines

- Process management = state machine management

1. Process Creation: Fork()

Intuitive Idea (like Windows / POSIX spawn)

- Create a new state machine: `spawn(path, argv)`
- Terminate a state machine: `_exit()`
  - The name of `exit()` is used by `libc`

UNIX Answer

- Copy an existing state machine: `fork()`
- Replace its program image: `execve(path, argv)`
- Terminate the state machine: `_exit()`

Why UNIX Uses `fork+exec`

UNIX Philosophy: keep system calls simple, combine flexibly

- `fork()` creates an identical child process
- Child process can modify file descriptors, environment, etc.
- Then call `execve()` to load a new program
- Shell fits naturally:
  - Shell `fork()`s a child
  - Child `execve()`s user command

Comparison

- Intuition: `spawn = create + run (one step)`
- UNIX: `fork + exec` (two steps, more flexible)

Fork(): Create a State Machine

```
pid_t fork(void);
```

We now have a "state machine".

- We only need an API to create it.
- UNIX answer: `fork`
  - Make a full copy of the state machine (memory and register context).

Process Tree: `pstree`

Process creation forms a process tree

- $A \to B \to C$: if B exits, what is $ppid(C)$?
- It seems simple to "go up" one level.
- In fact it is more complex:
  - A child notifies its parent when it terminates via `SIGCHLD`.
  - The parent can catch this signal.
  - The naive "go up" rule can be wrong.

How can we verify this?

- Write a small program to observe the behavior. Try it: pstree

Orphan Processes and Reparenting

Definition

- If parent *B* exits while child *C* still runs, *C* becomes an orphan.
- The kernel immediately reparents *C* to an adopter.

Who adopts?

- The nearest *subreaper* if present (`prctl(PR_SET_CHILD_SUBREAPER)`).
- Otherwise PID 1 in the same PID namespace (init or systemd).

Effects

- `ppid(C)` changes to the adopter's PID.
- *C* keeps its own PID, open files, and memory state.

Orphan vs. Zombie

- *Zombie*: the child has exited but no one has called `wait`.
- *Orphan*: the child runs but its parent has exited.
- The adopter will call `wait*` and reap the child at exit.

Note on containers

- In a PID namespace the adopter is that namespace's PID 1.

Behavior of `fork()`

Immediate copy of the state machine

- Copies all state as a snapshot
  - CPU registers and every byte of memory
- Caveat: the process also has state in the OS such as `ppid`, open files, and signals
- Be careful with how these states are copied
- On failure it returns `-1`
  - `errno` explains the reason (see `man fork`)

How to tell the two processes apart?

- The new process receives return value `0`
- The caller receives the child PID, forming the parent-child relation

Understanding Behavior of `fork()`

Try it: forkHello.c

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    for (int i = 0; i < 2; i++) {
        fork();
        printf("Hello\n");
    }
}
```

Execution: Run the above program using the following commands:

1. `gcc 9_forkHello.c`
2. `./a.out`
3. `./a.out | cat`

Understanding Behavior of `fork()` (Cont.)

Behavior Analysis:

- Running `./a.out` directly produces a different number of lines compared to `./a.out | cat`.
- Following the principle that *"the machine is always right"*, we analyze the cause:
  - Hypothesis: libc buffering effect.
  - Verification: Compare system call sequences using `strace`.

Buffering Control:

- Use `setbuf(3)` or `stdbuf(1)` to manage standard input/output buffering.

```
man setbuf
```

Understanding Behavior of `fork()` (Cont.)

When does the OS use line buffering?

- The operating system uses **line buffering** when writing to a terminal, meaning output is sent immediately when a newline character (`\n`) is encountered.

- If output is redirected (e.g., through a pipe), the standard output switches to **full buffering**, meaning data is only written when the buffer is full or when the program terminates.

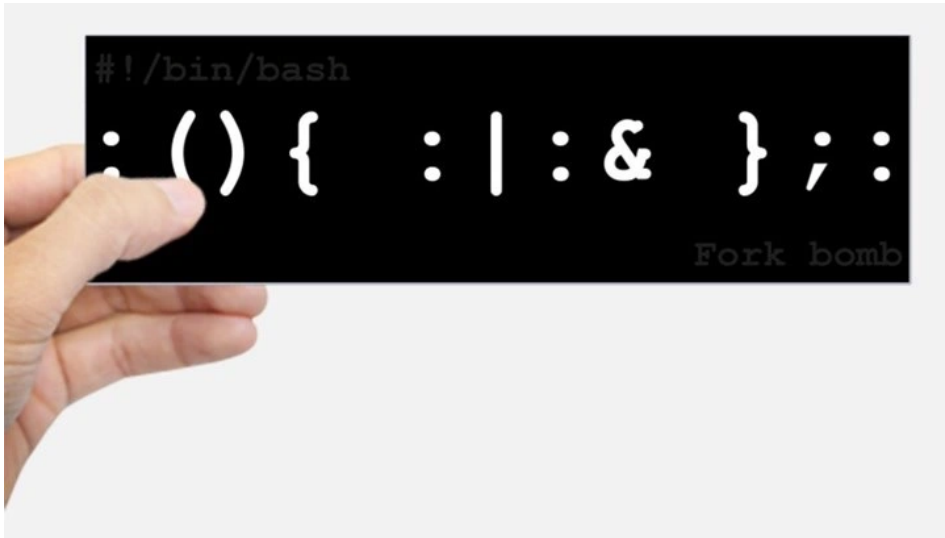
`fork()` creates an exact copy of the calling process, replicating every bit of its state, including the contents of buffers:

- The child process receives `0` as the return value.
- The parent process receives the child process ID.
- Other than the return value, the parent and child processes are identical and execute **in parallel** in the operating system.

Fork Bomb

Creating new state machines requires resources

- Continuously creating processes will eventually crash the system.
- In the past a bad program could freeze the lab. Modern Linux has OOM protection, so the kernel kills the process instead.
- **Do not run this on our campus computer systems!**
  - A quick story: my PhD student Jackie did this as an undergraduate. He did not realize it was a bomb-like script. The department servers crashed at midnight. The chair had to go to the lab and power off the machines. There was no punishment. He was told not to do it again.



Code Analysis: Fork Bomb

```
:(){ # define a function named ":" (bash allows symbol names)
  : | : & # call ":" twice via a pipeline; each side runs in a
      child shell
          # "&" backgrounds the pipeline so this function
              returns immediately
}; : # call ":" once -> spawns more -> exponential growth
```

```
f(){ # define function f
  f | f &
}
f # single trigger causes 1,2,4,8,... processes
```

Analogy to Nuclear Fission:

- A heavy atomic nucleus (U-235/Pu-239) is hit by a **neutron**, splitting into two lighter nuclei, releasing more **neutrons**.
- This results in **self-replication**.

2. Reset a State Machine

```
int execve(const char *filename, char * const argv[],
    char * const envp[]);
```

UNIX provides one API to reset the state machine

- Replace the current process image with the initial state described by an executable file.
- OS-kept state does not change: PID, working directory, open files.
  - Use `O_CLOEXEC` to close a file on `exec`.

`execve` is the only system call that executes a program

- It is the first system call you see in `strace`.

`execve()` sets the initial process state

`argc` & `argv`: command-line arguments

- The `main` parameters are provided by `execve`.

`envp`: environment variables

- View with the `env` command: `PATH`, `PWD`, `HOME`, `DISPLAY`, `PS1`, ...
- Use `export` to set variables for child processes.

```
export TK_VERBOSE=1
```

Program is correctly loaded into memory

- Code and data are mapped, the PC is at the entry point.

Example: `PATH` controls search order

- **PATH** = a list of directories searched left-to-right.
- **Who uses it**: the shell and `execvp()` when only a name is given.
- **gcc** calls the assembler `as`; it is found via `PATH`.

```
[pid] execve("/usr/local/sbin/as", ...) = -1
[pid] execve("/usr/local/bin/as", ...) = -1
[pid] execve("/usr/sbin/as", ...) = -1
[pid] execve("/usr/bin/as", ...) = 0 # found here
```

```
PATH="" /usr/bin/gcc a.c # no search path -> fails to
    find 'as'
PATH="/usr/bin" gcc a.c # add dir with 'as' -> works
```

3. Destroy the State Machine

```
void _exit(int status);
```

No debate here

- Immediately destroy the state machine and pass one status value.
  - The parent process can retrieve this value.

Takeaways

- Linux builds the entire application program world from an initial process (state machine).
- Through `fork`, `execve`, and `_exit`, we can create many child processes and then execute them **concurrently**.