Processes

2025 Winter ECE353: Systems Software

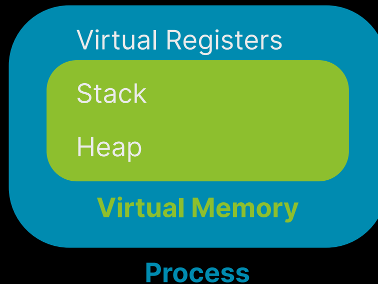
Lecture 3

Jon Eyolfson

2.0.1

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

Recall: A Process is an Instance of a Running Program



We Can Add More to a Process

Virtual Registers

Stack

Heap

Global Variables

Virtual Memory

Open File Descriptors

Process

A Process Control Block (PCB) Contains All Information

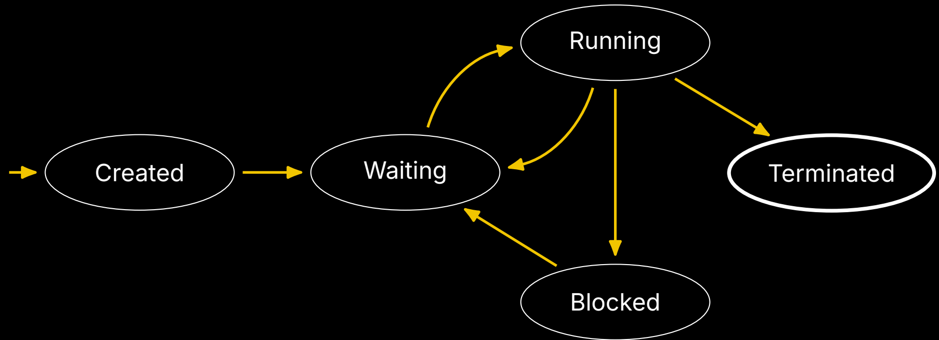
Specifically, in Linux, this is the `task_struct` you can browse on GitHub

It contains:

- Process state
- CPU registers
- Scheduling information
- Memory management information
- I/O status information
- Any other type of accounting information

Each process gets a unique process ID (pid) to keep track of it

Process State Diagram (You Could Rename Waiting to Ready)



You Can Read Process State Using the “proc” Filesystem

There’s a standard `/proc` directory (on Linux) that represents the kernel’s state

- These aren’t real files, they just look like it!

Every directory that’s a number (process ID) in `/proc` represents a process

There’s a file called `status` that contains the state (used for Lab 1)

We Could Create Processes from Scratch

We load the program into memory and create the process control block
(this is what Windows does)

Unix decomposes process creation into more flexible abstractions

Note: for programs with threads, this is a bad idea
(prefer `posix_spawn` instead)

Instead of Creating a New Process, We Could Clone It

- Pause the currently running process, and copy it's PCB into a new one
  - This will reuse all of the information from the process, including variables!
- Distinguish between the two processes with a parent and child relationship
  - They could both execute different parts of the program together
- The new processes can either continue or load a new program

`fork` Creates a New Process, A Copy of the Current One

`int fork(void)` as the following API:

- Returns the process ID of the newly created child process
  - -1: on failure
  - 0: in the child process
  - >0: in the parent process

There are now 2 processes running

- Note: they can access the same variables, but they're separate
  - Operating system does "copy on write" to maximize sharing

On POSIX Systems, You Can Find Documentation Using `man`

We'll be using the following APIs:

- `fork`
- `execve`
- `wait` (next lecture)

You can use `man <function>` to look up documentation,
or `man <number> <function>`

2: System calls
3: Library calls

fork-example.c Has One Process Execute Each Branch

```
int main(int argc, char *argv[]) {
  pid_t returned_pid = fork();
  if (retured_pid == -1) {
    int err = errno;
    perror("fork failed");
    return err;
  }
  if (returned_pid == 0) {
    printf("Child returned pid: %d\n", returned_pid);
    printf("Child pid: %d\n", getpid());
    printf("Child parent pid: %d\n", getppid());
  }
  else {
    printf("Parent returned pid: %d\n", returned_pid);
    printf("Parent pid: %d\n", getpid());
    printf("Parent parent pid: %d\n", getppid());
  }
  return 0;
}
```

`execve` Replaces the Process with Another Program, and Resets

`execve` has the following API:

- `pathname`: Full path of the program to load
- `argv`: Array of strings (array of characters), terminated by a null pointer
  Represents arguments to the process
- `envp`: Same as `argv`
  Represents the environment of the process
- Returns an error on failure, does not return if successful

`execve-example.c` Turns the Process into `ls`

```
int main(int argc, char *argv[]) {
  printf("I'm going to become another process\n");
  char *exec_argv[] = {"ls", NULL};
  char *exec_envp[] = {NULL};
  int exec_return = execve("/usr/bin/ls", exec_argv, exec_envp);
  if (exec_return == -1) {
    exec_return = errno;
    perror("execve failed");
    return exec_return;
  }
  printf("If execve worked, this will never print\n");
  return 0;
}
```

The Operating System Creates Processes

The operating system has to:

- Maintain process control blocks, including state
- Create new processes
- Load a program, and re-initialize a process with context

Linux Terminology Is Slightly Different

You can look at a process' state by reading `/proc/<PID>/status | grep State`
  Replace `<PID>` with the process ID (or `self`)

R: Running and runnable [Running and Waiting]
S: Interruptible sleep [Blocked]
D: Uninterruptible sleep [Blocked]
T: Stopped
Z: Zombie

The kernel lets you explicitly stop a process to prevent it from running
  You or another process must explicitly continue it

On Unix, the Kernel Launches A Single User Process

After the kernel initializes, it creates a single process from a program

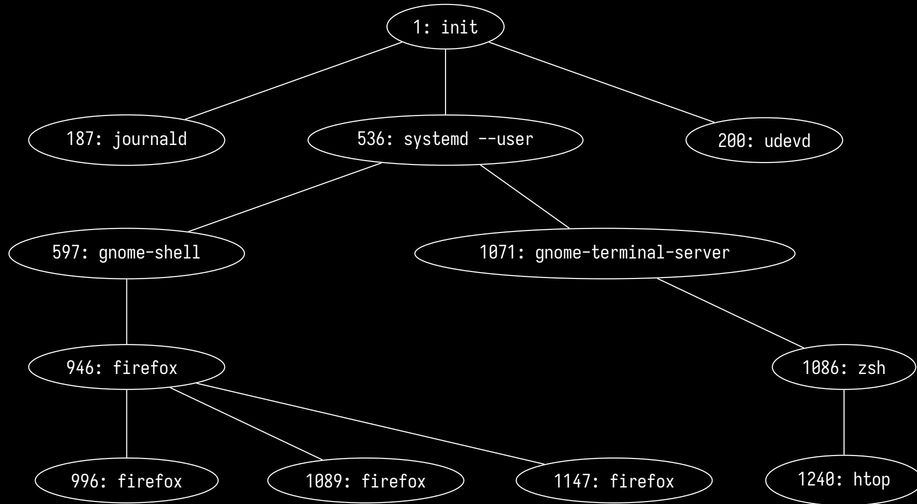
This process is called `init`, and it looks for it in `/sbin/init`

- Responsible for executing every other process on the machine
- Must always be active, if it exits the kernel thinks you're shutting down

For Linux, `init` will probably be `systemd` but there's other options

Aside: some operating systems create an "idle" process that the scheduler can run

A Typical Process Tree on the Virtual Machine



How You Can See Your Process Tree

Use `htop`

You can press `F5` to switch between tree and list view

Processes Are Assigned a Process ID (pid) On Creation and Does Not Change

The process ID is just a number, and is unique for every **active** process

On most Linux systems the maximum pid 32768, and 0 is reserved (invalid)

Eventually the kernel will recycle a pid, after the process dies, for a new process

Remember: each process has its own *address space* (independent view of memory)

Maintaining the Parent/Child Relationship

Previously, we made sure that our parent exited last (by using `sleep`)

What happens if the parent process exits first, and no longer exists?

The Parent Process is Responsible for Its Child

The operating system sets the exit status when a process terminates (the process terminates by calling `exit`)

It can't remove its PCB yet

The minimum acknowledgment the parent has to do is read the child's exit status

There's two situations:

1. The child exits first (zombie process)
2. The parent exits first (orphan process)

You Need to Call `wait` on Child Processes

`wait` as the following API:

- `status`: Address to store the wait status of the process
- Returns the process ID of child process
    - -1: on failure
    - 0: for non blocking calls with no child changes
    - >0: the child with a change

The wait status contains a bunch of information, including the exit code

- Use `man wait` to find all the macros to query wait status
    - You can use `waitpid` to wait on a specific child process

`wait-example.c` Blocks Until The Child Process Exits, and Cleans Up

```c
int main(int argc, char *argv[]) {
  pid_t pid = fork();
  if (pid == -1) { return errno; }
  if (pid == 0) {
    sleep(2);
  }
  else {
    printf("Calling wait\n");
    int wstatus;
    pid_t wait_pid = wait(&wstatus);
    if (WIFEXITED(wstatus)) {
      printf("Wait returned for an exited process! pid: %d, status: %d\n",
             wait_pid, WEXITSTATUS(wstatus));
    }
  }
  return 0;
}
```

A Zombie Process Waits for Its Parent to Read Its Exit Status

The process is terminated, but it hasn’t been acknowledged

A process may have an error in it, where it never reads the child’s exit status

The operating system can interrupt the parent process to acknowledge the child

- It is just a suggestion and the parent is free to ignore it
  - This is a basic form of IPC called a signal

The operating system has to keep a zombie process until it’s acknowledged

- If the parent ignores it, the zombie process needs to wait to be re-parented

An Orphan Process Needs a New Parent

- The child process lost its parent process
  - The child still needs a process to acknowledge its exit
- The operating system re-parents the child process to `init`
  - The `init` process is now responsible to acknowledge the child

`orphan-example.c` The Parent Exits Before the Child, `init` Cleans Up

```c
int main(int argc, char *argv[]) {
  pid_t pid = fork();
  if (pid == -1) {
    int err = errno;
    perror("fork failed");
    return err;
  }
  if (pid == 0) {
    printf("Child parent pid: %d\n", getppid());
    sleep(2);
    printf("Child parent pid (after sleep): %d\n", getppid());
  }
  else {
    sleep(1);
  }
  return 0;
}
```

zombie-example.c The Parent Monitors the Child To Check Its State

```c
pid_t pid = fork();
// Error checking
if (pid == 0) {
  sleep(2);
}
else {
  // Parent process
  int ret;
  sleep(1);
  printf("Child process state: ");
  ret = print_state(pid);
  if (ret < 0) { return errno; }
  sleep(2);
  printf("Child process state: ");
  ret = print_state(pid);
  if (ret < 0) { return errno; }
}
```

You’re Responsible for Managing Processes

The operating system maintains a strict parent/child relationship

You should be able to identify (and prevent) the following:

- Zombie processes
- Orphan processes