# Process Management

2024 Fall ECE 344: Operating Systems
Jon Eyolfson

Lecture 5
2.0.1

## 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

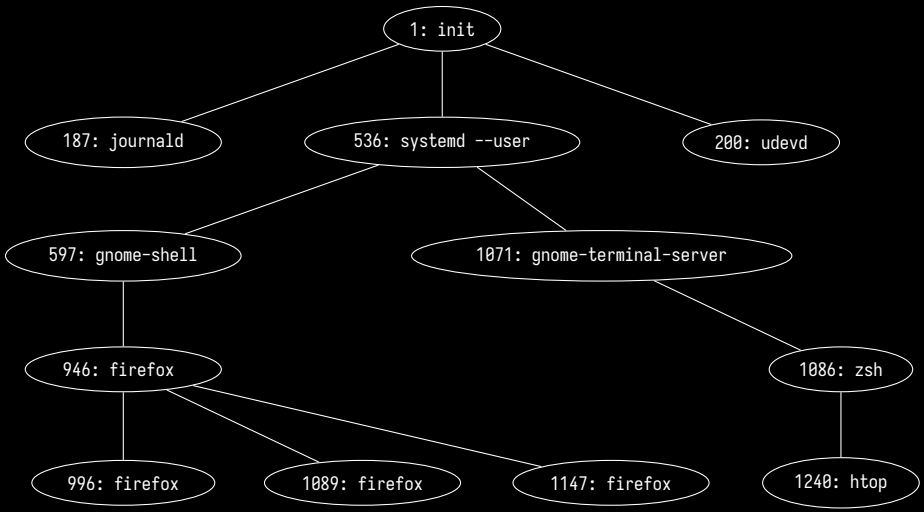
After the kernel initializes, it creates a single process from a program

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

```
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

```
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