CS 111: Operating System Principles
Lecture 5

Process API
2.0.0

Jon Eyolfson
July 6, 2021

Linux Terminology Is Slightly Different

You can look at a process' state by reading `/proc/<pid>/state`

Replace `<pid>` with the process ID

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 POSIX Systems, You Can Find Documentation Using `man`

We'll be using the following APIs:

- `execve`
- `fork`
- `wait`

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

2: System calls
3: Library calls

`execve` Loads Another Program, and Replaces Process with A New One

`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;
}
```

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

`fork` 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

fork-example.c Has One Process Execute Each Branch

```
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 returned pid: %d\n", pid);
    printf("Child pid: %d\n", getpid());
    printf("Child parent pid: %d\n", getppid());
  }
  else {
    printf("Parent returned pid: %d\n", pid);
    printf("Parent pid: %d\n", getpid());
    printf("Parent parent pid: %d\n", getppid());
  }
  return 0;
}
```

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

```
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 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 no 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 Exists, 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;
}
```

We Used System Calls to Create Processes

You should be comfortable with:

- `execve`
- `fork`
- `wait`

This includes understanding processes and their relationships