Process Practice

2024 Fall ECE 344: Operating Systems
Jon Eyolfson

Lecture 7
2.0.0

A Teaching Operating System

https://github.com/mit-pdos/xv6-riscv

Used in MIT graduate OS courses, it is a full OS you can run on the VM
You'll run it as a VM using QEmu (yes, a VM in a VM)

It's a re-implementation of Unix version 6 for RISC-V in C

Uniprogramming is for Old Batch Processing OSs

Uniprogramming: only one process running at a time
  Two processes are not parallel and not concurrent, no matter what

Multiprogramming: allow multiple processes
  Two processes can run in parallel or concurrently

Modern operating systems try to run everything in parallel and concurrently

The Scheduler Decides When To Switch

To create a process, the operating system has to at least load it into memory

When it's waiting, the scheduler (coming later) decides when it's running

We're going to first focus on the mechanics of switching processes

The Core Scheduling Loop Changes Running Processes

1. Pause the currently running process
2. Save its state, so you can restore it later
3. Get the next process to run from the scheduler
4. Load the next process' state and let that run

We Can Let Processes Themselves, or the Operating System Pause

Cooperative multitasking
The processes use a system call to tell the operating system to pause it

True multitasking
The operating system retains control and pauses processes

For true multitasking the operating system can:

- Give processes set time slices
- Wake up periodically using interrupts to do scheduling

Swapping Processes is called Context Switching

We've said that at minimum we'd have to save all the current registers
    We have to save all the values, using the same CPU as we're trying to save

There's hardware support for saving state, however you may not want to save everything

Context switching is pure overhead, we want it to be as fast as possible

Usually there's a combination of hardware and software to save as little as possible

A New API — `pipe`

```
int pipe(int pipefd[2]);
```

Returns `0` on success, and `-1` on failure (and sets `errno`)

`pipe` forms a one-way communication channel using two file descriptors
- `pipefd[0]` is the `read` end of the pipe
- `pipefd[1]` is the `write` end of the pipe

You can think of it as a kernel managed buffer
- Any data written to one end can be read on the other end

Aside: Using & in Your Shell

If you use & at the end of your command, your shell will start that process and return

e.g. `sleep 1 &`

It outputs the `pid` and lets you know when it's finished

The `|` character creates a pipe between two processes

The sneaky Bash fork bomb is: `:(){ :|:& };:`

Do not run this command

Let's See the Example

See: `07-process-practice/pipes.c`

If we remove the call to `write` in the parent, the child never exits

What happens to the child?

Final 2022 Question 1

For each program shown below, state whether it will produce the **same** output each time it is run, or whether it may produce **different** outputs when run multiple times. Explain why the program behaves like this.

```c
int main() {
  int i = 4;
  while (i != 0) {
    int pid = fork();
    if (pid == 0) {
      i--;
    }
    else {
      printf("%d\n", i);
      exit(0);
    }
  }
  return 0;
}
```

Final 2022 Question 2

Same as the previous question, except now there's a `waitpid`

```c
int main() {
  int i = 4;
  while (i != 0) {
    int pid = fork();
    if (pid == 0) {
      i--;
    }
    else {
      waitpid(pid, NULL, 0);
      printf("%d\n", i);
      exit(0);
    }
  }
  return 0;
}
```