ECE 353: Systems Software
Lecture 7

Process Practice

1.1.0

Jon Eyolfson
January 23, 2023

Trying to Get a Better Midterm Room

Changing to Feburary 28th (Tuesday)

Asked for better rooms without arm tables

Let me know any major issues or if we need to coordinate with other courses

On a RISC-V CPU, There's 3 Terms for "Interrupts"

Interrupt

Triggered by external hardware,
handled by the kernel (needs to respond quickly)

Exception

Triggered by an instruction (divide by zero, illegal memory access),
default handler is the kernel (calling process suspended),
the process can optionally handle some of these themselves

Trap

Transfer of control to a trap handler caused by either
an exception or an interrupt (code that runs)

A system call would be a *requested trap*

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?

We Explored Basic IPC in an Operating System

Some basic IPC includes:

- `read` and `write` through file descriptors (could be a regular file)
- Redirecting file descriptors for communcation
- Signals

Signals are like interrupts for user processes

The kernel has to handle all 3 kinds of “interrupts”

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.

```
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`

```
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;
}
```