

CS 4414/5416: Systems Programming

Processes

David Bindel

2026-10-06

Which of these are elements of a page table entry structure that might be stored in a TLB?

- The virtual memory address
- The physical frame number
- Write and execute permission flags
- The accessed and dirty bits
- Device and block number in swap space

Logistics

- Prelim and other marked things are up
- P3 final represents *only* the autograded work
 - Will have additional entries for rest

- See: How to check in
- We now have a PollEv login from Canvas
- Freebies on PollEv days so far
- Will try to make it harder to skip registering!

- P3 (minishell) due Monday, 10/19
- Lab 7 is Friday: building pipelines
- NB: labs/projects are fair game for exams!
- No class 10/13 - Fall Break!

- You *are* allowed to refactor
 - Just maintain the tested interfaces
- Pipes from built-ins are not required
 - But are interesting should you wish
- URL input should accommodate large files
 - Don't just assume it will fit in memory!
- Should still clean up after failed pipelines

- No lecture 10/13 - enjoy fall break!
- I will be away for 10/15 lecture
- Prof Adrian Sampson will introduce TCP sockets

Recap

This module: Unix abstractions

- Basic abstractions (files, pipes, virtual memory)
- Processes and inter-process communication
- Introduction to TCP sockets

(Yes, some overlap with 3410/4410.)

- Today: processes and IPC
 - Review of processes from 3410
 - Managing children
 - Scheduling
 - Signal handling
 - Inter-process comm mechanisms
- Thu: pipes, Unix sockets, and shared memory
- Fri: lab on Unix pipelines

Process Basics

What is a Process?

An instance of a program in execution

- Contains full execution state of a program
 - CPU state, memory map, file descriptors, ...
- Is the main unit of execution in multi-tasking

Kernel-maintained PCB data structure tracks

- Process ID
- Process status (running, sleeping, etc)
- Program counter and register state
- File descriptor table
- Page table

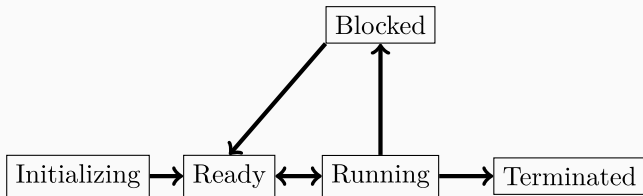
and more!

Processes run in user mode; kernel takes control when

- A hardware interrupt happens
 - Basis for *pre-emptive multitasking*
- An instruction *traps* into kernel
 - Could be a deliberate syscall
 - Or via something like a TLB miss

Crossing kernel boundary is not free!

Life Cycle of a Process



Thread < process!

- Per thread: stack, PC, CPU state
- Threads share: memory address, files, etc
- We'll address threads later
 - OS mostly deals with processes

Ask Me a Question!

Managing Children

Two phases to launch a new program (C interface):

- Call **fork** for new process
 - In parent: returns with child PID
 - In child: returns 0
- Child calls **exec** to start new program

Rust does a combination.

Subprocesses in Rust

```
use std::process::Command;

let mut child = Command::new("ls") // the program to run
    .arg("-la")                    // add one argument (or .args(...))
    .spawn()?;                     // fork + exec, returns Child handle
```

- `std::process::Command` is a *builder*
 - Set things up via methods on `Command`
- Launch immediately with `spawn`
- Launch and wait with `output` or `status`

- Command line arguments via `arg` or `args`
- Also pass information through *environment variables*
 - These are just key/value string pairs
 - Default is to inherit parent process env
 - Edit with `env`, `envs`, `env_clear`, `env_remove`

- **Command** methods set **stdin**, **stdout**, **stderr**
- Can take **Stdio** object built from
 - **Stdio::inherit()** - inherit from parent
 - **Stdio::piped()** - set up a pipe
 - **Stdio::from(file)** - connect to a file
- A **Child** has **stdin**, **stdout**, **stderr** fields
 - Use **take()** to connect to something else

Caveat Lector!

```
let mut child = Command::new("ls") // the program to run
    .arg("-la")                      // add one argument (or .args([...]))
    .stdout(Stdio::piped())         // Prep for standard output
    .spawn()?;                      // fork + exec, returns a Child handle

// --
// Try to set up another subprocess to read, and it fails to launch...
// --

let mut sink = std::io::sink();
let mut stdout = child.stdout.take();
std::io::copy(&mut stdout, &mut sink); // IMPORTANT!

let status = child.wait()?;         // block until it finishes
```

Why is that line important?

- Linux also lets you change
 - Process user/group/permissions
 - Virtual FS root
- System-dependent! Not part of `std::process::Command`
- Use `std::os::unix::process::CommandExt` methods

Handling Return Values

```
use std::process::Command;

let mut child = Command::new("ls") // the program to run
    .arg("-la")                     // add one argument (or .args([...]))
    .spawn()?;                     // fork + exec, returns a Child handle
let status = child.wait()?;        // block until it finishes
let code: i32 = status.code().unwrap_or(0);
```

- Child in terminated state just holds onto exit code
- **wait** on child to complete and get code
 - Otherwise, “zombie” process continues on!

What if parent exits without `wait` on children?

- Parent is responsible for cleaning up!
- Zombie children are adopted by init process
- Init process has a “reaper” to finish the zombies
 - Connect to `SIGCHLD` handler

Why should crates like zerocopy disallow references in mapped data structures?

- (a) References do not implement the Clone trait
- (b) References will be invalid when moved to another process
- (c) References will be invalid when moved even in the same process

Signals

What's a Signal?

A *signal* notifies the process of a system event

- Ex: Ctrl-C, terminal hangup, child stopped
- Frequently sent on system events
- Can send explicitly with `kill`

- Each signal has a default handler
 - Options: Core dump, terminate, ignore
- Can sometimes write your own – carefully!
 - Communicate through volatile variables
 - Very few functions are safe to call
- In Rust, use signal-hook crate

Scheduling

- Everyone ready should get a fair share
- Important tasks should get more CPU time
- Best to re-schedule repeatedly to same core

Of course, this matters for benchmarking...

Nice

```
# Run date at niceness of shell + 5
nice -n 5 date

# Demand better priority (e.g. for timing runs)
su nice -n -10 runuser -u dbindel date
```

- “Niceness” is a property used by scheduler
 - There’s an internal priority that overrides
- More nice means that it waits longer
- Can use negative niceness as superuser
- **renice** adjusts niceness of running process

- Can also tell OS what core we prefer
 - Helps with cache re-use!
 - Think P2 timing...
- Relevant command is **taskset**
- Can even reserve CPUs at boot time!
 - Use **isolcpus** to bootloader

Ask Me a Question!

Inter-Process Communication

Why Separate Processes?

- Composition and separate concerns
 - Pipelines of “small sharp tools”
 - Front-end / back-end
- Mixed-language programming
- Programming services

- Shellouts, wrappers, intermediate files
- Pipes and FIFOs
- Shared memory
- Unix domain sockets
- Network sockets

- Message queues
- Remote procedure calls
- Promises

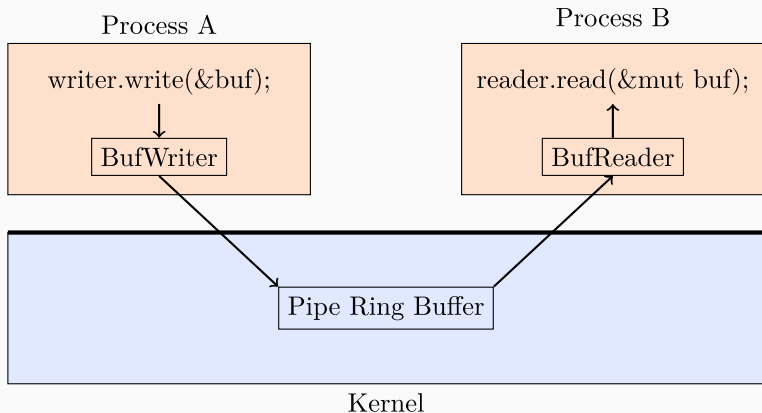
- How do we maintain consistent distributed state?
- How do we track software version inconsistencies?
- How do we handle partial failure?
- How do we debug issues?
- What's with all the buffers?
 - Zero copy is a magical thing...

Need to handle many possible errors:

- Crash of child (or parent)
- Connection errors
- Weird I/O errors!

Broad problem: *partial* failure

Buffering



- Many buffers (at reader, writer, in kernel)
- But they aren't infinite
- Don't forget to flush!

Writers block when out of buffer

- Blocked writer won't terminate!
- Easy way to get things to hang...

How do you feel about today's lecture?

- (a) Replicates 3410, skip or shorten in F27
- (b) Mostly review, nice to have a reminder
- (c) Lots was new to me, but I mostly followed
- (d) This is way too fast, I was repeatedly lost

Wrap-Up

- Process: an instance of a program in execution
 - System code involves many processes
 - Often parents launching children
 - OS scheduling is process-by-process
- Outside files, processes are *mostly* isolated
- Can send in a little info via signals
- More general inter-process communication is key!

- Thu lecture: pipes, sockets, shared memory IPC
- Lab 7 is Friday: building pipelines
- No class 10/13 - Fall Break!
- P3 (minishell) due Monday, 10/19

On Ed: If you are having trouble with PollEv after this lecture, please post on today's Ed thread (and try to help us figure it out before Thu).