Multiprocessing pitfalls

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2023-12-05

Introduction

Previously, on SSP-RS

- Parallelism: performing multiple tasks simultaneously
- Why parallelism: performances, energy consumption
- Two ways of managing multitasking in Rust: threads (lecture 8), async (lecture 9)
- How Rust threads are safer than in legacy system programming
- How Rust async is different than in other programming languages

- There is another (still) popular way of implementing multitasking on UNIX systems: `fork()`-based multiprocessing with IPC[1]-based communication between processes
- This lecture: pitfalls in traditional multiprocessing and how to avoid them in Rust

[1] Inter Process Communication

Fork-based multiprocessing — redux

Process creation:

- `fork()`: “creates a new process by duplicating the calling process.”
  - Called once, return twice: parent process + child process.
  - The entire memory of the parent process is duplicated into the child, with copy on write (CoW) optimization.
- `exec{l,v}{p,e,}()`: “replaces the current process image with a new process image.”
  - Used to execute an external program, usually after fork.
  - **Shelling out:** UNIX programming pattern where a parent process `fork`+`exec` a separate program to delegate a task to it.
  - The separation between `fork` and `exec` allows to manipulate file descriptors destinations in between, enabling input/output **redirections** between parent, child, and the filesystem.

Inter-process communication (IPC) — redux

Q: how do processes communicate?

- Communication vs Synchronization

IPC mechanisms (1):

- `wait()`: "suspends execution of the calling thread until one of its children terminates."
  - Basic synchronization ("wait until termination")
  - Limited support for communication: integer sent from child to parent (upon termination)
- `pipe()`: byte streams shared by processes belonging to the same family (i.e., a parent…child path exists between them)
  - Limited read/write atomicity guarantees (up to `PIPE_BUF` bytes)
- `mkfifo()`: like pipes, but also among unrelated processes (via the filesystem)
- `signal()` & `kill()`: notification of out-of-bound events
- File locking: synchronization only, filesystem-based
  - Can support safe (albeit slow) communication via the filesystem
  - Implementations: `flock()`, `fcntl()`, `lockf()`

Inter-process communication (IPC) — redux (cont.)

IPC mechanisms (2):

- Unix domain sockets: like FIFOs, but with more advanced networking and session-control features
  - Intuition: single-host networking
- Message queues
  - Implementations: POSIX message queues
- Shared memory: memory regions accessible via raw pointers by multiple processes
  - Synchronization not included!
  - Implementations: `mmap`, POSIX shared memory
- Semaphores
  - Operations: P, V
  - Implementations: POSIX semaphores

And much more if you leave the POSIX boundaries…

Don’t call `fork()`

Fork — what could possibly go wrong?

```c
int main(int argc, char **argv) {
    while (1) {
        pid_t pid = fork();
        if (pid == 0) {
            do_stuff();
        }
    }
}
```

1. If `do_stuff()` doesn't terminate, the child process will re-enter the loop, `fork()`-ing again → potential fork bomb depending on what `do_stuff()` does.
2. Zombie apocalypse machine! → parent process does not wait for children,[2] potentially creating a lot of zombie processes.

Note how in both cases our reasoning on the potential problems is not local to the code we are auditing.

[2] at least based on what we can see in this snippet; can the child reaping logic be elsewhere?

Fork — what could possibly go wrong? (cont.)

```
1  int main(int argc, char **argv) {
2      pid_t pid1 = fork();
3      pid_t pid2 = fork();
4      if (pid1 == 0) {
5          do_stuff1();
6          return 0;
7      }
8      if (pid2 == 0) {
9          do_stuff2();
10         return 0;
11     }
12     waitpid(pid1, NULL, 0);
13     waitpid(pid2, NULL, 0);
14 }
```

- If both `fork()` succeed 4 processes, rather than 3, come out of line 3.
- `do_stuff1` is executed twice, not once.

(Imagine this broken logic in a busy loop… → `fork()` bomb.)

Fork — what could possibly go wrong? (cont.)

```c
int createChildAndSayHello() {
    pid_t pid = fork();
    if (pid == 0) {
        sayHello();
        return 0;
    }
    waitpid(pid, NULL, 0);
}
```

- Child process `return`-s (to parent function) instead of `exit`-ing (= guaranteed to terminate).
- The call stack is inherited by the child process at `fork` time.
- Child process will return to the parent's function who called `createChildAndSayHello`, executing code probably intended for the parent process only.

Fork — what could possibly go wrong? (cont.)

```
1  pid_t createChild(char *argv, int readFd, int writeFd) {
2      pid_t pid = fork();
3      if (pid == -1) { throw Exception("fork failed"); }
4      if (pid == 0) { // child
5          if (dup2(readFd, STDIN_FILENO) < 0) {
6              throw Exception("dup2 on stdin failed");
7          }
8          if (dup2(writeFd, STDOUT_FILENO) < 0) {
9              throw Exception("dup2 on stdout failed");
10         }
11         execvp(argv[0], argv);
12         throw Exception("exec failed");
13     }
14     return pid;  // parent
15 }
```

- Same footgun as before, C++ style!
- Exceptions propagate up the call stack, so by *only looking at this code* we have no idea how the various thrown exceptions will be handled.
- They are thrown in the child (in between `fork` and `exec`), but could end up being handled by code only mean to be executed by the parent.

Fork — what could possibly go wrong? (cont.)

```c
// A process running multiple threads arrives here.
pid_t pid = fork();
if (pid == 0) {
    const char **args = malloc(sizeof(char *) * num_args);
    execvp(args[0], args);  // note: upon execvp() all memory will be freed
    free(args);  // if we are here, execvp() failed
    exit(1);
}
```

- Threads share a common virtual address space.
- `malloc()` is thread-safe: it uses a lock to avoid memory corruption during allocation.
- When you `fork()` from a multi-threaded process, a single-threaded child process is created.
- The child gets a copy of parent's memory, shared by all parent threads, *including locks*.
- What if: thread 1 calls `fork()` while tread 2 holds the `malloc()` lock?
- Child process will probably never free the lock (because it continues execution from *thread 1* `fork()` point); when it will try to `malloc()` later it will **deadlock**.

→ Arguably the biggest danger with `fork()`.

Mixing threads and processes

- Mixing multithreading and multiprocessing is generally a bad idea.
- It is also hard to avoid *with 100% certainty*.
- You can be sure that *your* code is not multi-threaded, but what about third-party code?
  - Are you sure library code is not running a background housekeeping thread when you `fork()`?

(Yet another instance of the *non-local reasoning* problem when auditing code that we observed before.)

Rule of thumb

- Don't mix multithreading and multiprocessing.
- (Yes, it's hard to make sure you aren't; that's [system programming] life.)
- If you really need to use multiprocessing: move children logic to a separate executable, `exec` it, and make sure you `exit` after `exec`, just in case it fails.

Fork's pitfalls — recap

1. Accidentally nesting `fork()`-s when spawning child processes
2. Runaway children
3. Failure to clean up zombie processes
4. Thread-based deadlocks after `fork()`

fork+exec — flexibility or footgun?

- `fork`+`exec` parallelism dates back to the 70's. It is an excellent example of **early UNIX design:**
  - *"Do only one thing and do it well"*
  - *"Everything is a file descriptor"*
  - Flexibility via chaining of operations. In between `fork` and `exec` you can:
    - Rewire file descriptors → redirections
    - Change signal masks, environment variables, CPU and memory pinning, etc.
- More modern OS have more complex syscalls that do everything at once: `CreateProcess*()` (Windows) and `clone*` (Linux, non-POSIX extension). From `man 2 clone`:

  By contrast with fork(2), these system calls provide more precise control over what pieces of execution context are shared between the calling process and the child process. For example, using these system calls, the caller can control whether or not the two processes share the virtual address space, the table of file descriptors, and the table of signal handlers.

- Legacy UNIX fork/exec: more flexibility → more things that programmers can get wrong!
- Solution: **safe higher-level abstractions**, implemented correctly once and for all.
- *Less flexible* than `fork`+`exec`, but they remain around if you really need them.

Rust at your `Command`

`Command`

- Part of the Rust stdlib, `std::process::Command` provides a **safe high-level abstraction** to **shell out**/delegate tasks to external programs via **controlled** `fork`+`exec` multiprocessing.
- Based on the **builder pattern**:
  - you first provide a description of the command you want to execute → by concatenating invocations of *build methods*,
  - then you run the command → by invoking a *terminal method*.
- Provides **limited flexibility** to customize the environment in which the external program will run:
  - working directory,
  - environment variables,
  - stdio/stdout/stderr redirection.

Similar abstractions elsewhere: `subprocess` (Python), `Boost.Process` (C++).

`Command` — command building

```rust
let command = if cfg!(target_os = "windows") {
    Command::new("cmd")
            .args(["/C", "echo hello"])
} else {
    Command::new("sh")
            .arg("-c")
            .arg("echo hello")
};
```

- Note: the command hasn't been *run* yet, we are still building it using the DSL (Domain-Specific Language) of `Command` methods chaining.
- Arguments are well separated, each of them corresponding to an `arvg` element.
  - No implicit interpretation of magic shell characters.
  - Mitigates shell injection risks (but still…).

(This and the following examples are adapted from the Command's stdlib doc.)

`Command` — running the command

Exit status only:

```rust
// pub fn status(&mut self) -> Result<ExitStatus>
let status = Command::new("cat")
                     .arg("file.txt")
                     .status()  // block parent, waiting for child to terminate
                     .expect("failed to execute process");

println!("process finished with: {status}");
```

`status()` = `fork` + `exec` + `waitpid`

… minus the footgun!

`Command` — running the command (cont.)

Output (stdout+stderr) capture too:

```rust
// pub fn output(&mut self) -> Result<Output>
let output = Command::new("cat")
                     .arg("file.txt")
                     .output()  // block parent, waiting for child to terminate
                     .expect("failed to execute process");

println!("status: {}", output.status);
io::stdout().write_all(&output.stdout).unwrap();
io::stderr().write_all(&output.stderr).unwrap();
```

`output()` = `fork` + `pipe` + `dup2` + `exec` + `waitpid`

… minus the footgun!

`Command` — running the command (cont.)

Spawning and joining later:

```
1  // pub fn spawn(&mut self) -> Result<Child>
2  let mut child = Command::new("cat")
3                           .arg("file.txt")
4                           .spawn()
5                           .expect("failed to execute child");
6
7  let ecode = child.wait()
8                   .expect("failed to wait on child");
9
10 assert!(ecode.success());
```

`Command` — environment fiddling

Environment variables:

```
Command::new("ls")
        .env("PATH", "/bin")  // add or replace entry

Command::new("ls")
        .env_remove("PATH")  // remove entry

Command::new("ls")
        .env_clear()  // remove all entries (e.g., for sandboxing)
```

Working directory:

```
Command::new("ls")
        .current_dir("/bin")
```

`Command` — environment fiddling (cont.)

Redirections:

```rust
let output = Command::new("echo")
    .arg("Hello, world!")
    .stdout(Stdio::null())
    .output()
    .expect("Failed to execute command");

assert_eq!(String::from_utf8_lossy(&output.stdout), "");
```

```rust
let output = Command::new("rev")
    .stdin(Stdio::null())
    .stdout(Stdio::piped())  // default value when using the output() terminator
    .output()
    .expect("Failed to execute command");

assert_eq!(String::from_utf8_lossy(&output.stdout), "");
```

`Command` — environment fiddling (cont.)

Pre-exec function:

```rust
// unsafe fn pre_exec<F>(&mut self, f: F) -> &mut Command
// where
//     F: FnMut() -> Result<()> + Send + Sync + 'static,
let cmd = Command::new("ls");
unsafe {
    cmd.pre_exec(function_to_run);
}
let child = cmd.spawn();
```

From `pre_exec` doc:

> *This closure will be run in the context of the child process after a fork. This primarily means that any modifications made to memory on behalf of this closure will not be visible to the parent process. This is often a very constrained environment where normal operations like malloc, accessing environment variables through std::env or acquiring a mutex are not guaranteed to work (due to other threads perhaps still running when the fork was run).*

The `unsafe` block requirement acts as a reminder for these safety considerations.

Fork's pitfalls vs `Command`

- ~~Accidentally nesting `fork()`s when spawning child processes~~
- ~~Runaway children~~
- Failure to clean up zombie processes
  - Still a risk if you use `spawn` and forget to `wait`
  - Mitigation: custom datatype that relies on `Drop` to reap zombies
- ~~Thread-based deadlocks after `fork()`~~

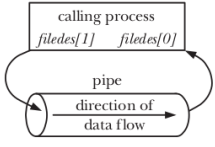
Don’t call `pipe()`

Pipe — redux

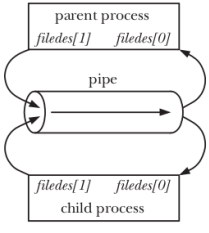
```c
#include <unistd.h>

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

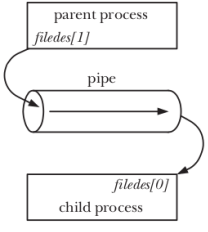
(1) After calling `pipe()`



(2) After calling `fork()`



(3) After closing unused FDs



(Images from TLPI.)

Pipe — what could possibly go wrong?

Q: what do you think could go wrong with pipes?

- Leaked file descriptors (common problem with all FD-based interfaces).
- Calling `close()` on the wrong values (e.g., the wrong end of the pipe).
- Use-before-pipe (i.e., performing I/O on uninitialized integer values… that could be valid file descriptors!).
- Use-after-close.

Solution — simple cases

- *Most* uses of `pipe` are related to redirections. We have already seen how `Command` support those without having to manipulate pipes directly, e.g.:

```rust
let output = Command::new("echo")
    .arg("Hello, world!")
    .stdout(Stdio::null())
    .output()
    .expect("Failed to execute command");
```

- To keep talking to a running process (a slightly more complex scenario) we can still manage with `Command`, using `Stdio::piped` like this:

```rust
let mut child = Command::new("cat")
    .stdin(Stdio::piped())
    .stdout(Stdio::piped())
    .spawn()?;

child.stdin.as_mut().unwrap().write_all(b"Hello, world!\n")?;
let output = child.wait_with_output()?;
drop(child_stdin);  // close stdin
```

Pipe — what could possibly go wrong? (cont.)

Deadlocks!

Scenario #1

- **Reading from a pipe is blocking** as long as there is at least one open writer to the same pipe.
- (After the last writer closes the pipe, reads immediately return an EOF.)
- It is enough for a single writer to forgot to close a pipe when done to deadlock all readers. :-(

Scenario #2

- Pipes are backed by a kernel buffer that can *fill up*. **Writing to a full pipe is blocking**.
- Assume two processes (A, B) are communicating via two pipes (A2B for A→B communication and B2A for B→A).
- Process A fills up A2B; meanwhile process B fills up B2A.
- Both blocks on their next write and cannot read to unblock their peer. → Deadlock :-(

Pipe — going further with other higher level abstractions

In the same spirit of `Command` for `fork`+`exec`, there exist crates that offer safe abstractions for playing with pipes (if you really need to!).

- `os_pipe` crate: abstracts other integer file descriptors for pipes

```rust
let (mut reader, mut writer) = os_pipe::pipe()?;
// XXX: If this write blocks, we'll never get to the read.
writer.write_all(b"x")?;
let mut output = [0];
reader.read_exact(&mut output)?;
assert_eq!(b"x", &output);
```

- `duct` crate: shell-like pipelines and redirections

```rust
let stdout = cmd!("echo", "hi").pipe(cmd!("sed", "s/i/o/")).read()?;
assert_eq!(stdout, "ho");
```

Don’t call `signal()`

Signals — redux

- UNIX mechanisms to handle out-of-bound events.
- Examples: segfault, division by zero, alarms, child process termination, Ctrl-C in the terminal, etc.
- Emission: automatic (kernel and/or hardware), `kill` syscall and CLI.
- Legacy handling: `signal` syscall.

From `man 7 signal`:

```
Signal      Standard   Action   Comment
--------------------------------------
SIGABRT      P1990      Core    Abort signal from abort(3)
SIGALRM      P1990      Term    Timer signal from alarm(2)
SIGBUS       P2001      Core    Bus error (bad memory access)
SIGCHLD      P1990      Ign     Child stopped or terminated
SIGCONT      P1990      Cont    Continue if stopped
SIGFPE       P1990      Core    Floating-point exception
SIGHUP       P1990      Term    Hangup detected on controlling terminal [...]
SIGILL       P1990      Core    Illegal Instruction
SIGINT       P1990      Term    Interrupt from keyboard
SIGKILL      P1990      Term    Kill signal
SIGPIPE      P1990      Term    Broken pipe: write to pipe with no readers [...]
[...]
```

man 2 signal — warning

```
NAME
  signal - ANSI C signal handling

SYNOPSIS
  #include <signal.h>

  typedef void (*sighandler_t)(int);
  sighandler_t signal(int signum, sighandler_t handler);

DESCRIPTION
  WARNING: the behavior of signal() varies across UNIX versions, and has also varied
  historically across different versions of Linux. Avoid its use: use sigaction(2) instead.
  See Portability below.

  [...]

  Portability
    The only portable use of signal() is to set a signal's disposition to SIG_DFL or SIG_IGN.
    The semantics when using signal() to establish a signal handler vary across systems
    (and POSIX.1 explicitly permits this variation); do not use it for this purpose.
```

(Sounds familiar?)

Signal — what could possibly go wrong?

OK, but that seems to be “just” a portability issue. What if we don’t care about that?

```c
void exit_on_ctrl_c(int sig) {
    _exit(1);
}

int main() {
    signal(SIGINT, exit_on_ctrl_c);
    while (1) {
        sleep(1);
    }
    return 0;
}
```

Is this code safe?

Yes, it is, but for *very specific* reasons; it is not *trivially* safe.

- It calls the `_exit` syscall instead of the `exit` stdlib function.
- The signal handler will fire at most once.

Signal — what could possibly go wrong? (cont.)

```c
#define NUM_PROCESSES 10
static int sigchld_count = 0;

void count_sigchld(int sig) {
    sigchld_count++;
}

int main() {
    signal(SIGCHLD, count_sigchld);
    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (fork() == 0) { // child
            sleep(1);
            exit(0);  // will send SIGCHLD to parent
        }
    }
    printf("Created %d child processes\n", NUM_PROCESSES);
    for (int i = 0; i < NUM_PROCESSES; i++)  wait(NULL);
    printf("All %d children exited, received %d SIGCHLDs\n", NUM_PROCESSES, sigchld_count);
    return 0;
}
```

Q: is this code correct?

Signal — what could possibly go wrong? (cont.)

```
$ ./sigchld-count
Created 10 child processes
All 10 children exited, received 7 SIGCHLDs
```

(could be anything, up to 10 [rarely])

Can be made reliable using the `sigaction` syscall.

- Back in 2011, from a slide of yours truly for a system programming course: *"sigaction, not signal, should be used in all new code that deals with signals."*
- More than a decade later, `signal()`-based code still abounds!

Signal — what could possibly go wrong? (cont.)

```c
void print_on_ctrl_c(int sig) {
    printf("Received Ctrl-C, ignoring it :-P");
}

int main() {
    signal(SIGINT, print_on_ctrl_c);
    while (1) {
        sleep(1);
    }
    return 0;
}
```

Q: is this code safe?

No, it can **deadlock**.

Signal — what could possibly go wrong? (cont.)

- Stdlib I/O functions work on buffered streams that are eventually flushed to destination.
- Streams use file locks to work well out of the box in multithreaded contexts.
- Your code might receive SIGINT while executing `printf` code that has just taken a lock.
- The signal handler will try to execute `printf`, which will attempt to take the lock again, …
  → Deadlock `:-(`

Similar issues might arise when calling from a signal handler any other function that fiddles with a global lock somewhere. (It is a situation similar to the thread/process mixing scenario, but without threads!)

- So what can you safely do from within a signal handler?
- Not much, the only functions you can call are **async-signal-safe functions**, a predefined (small) set described in `man 7 signal-safety`.
- Note how memory allocation functions like `malloc`/`free` are *not* async-signal-safe functions.

~~Proper~~ Decent signal handling (in general)

- Signal handling is a bad, inherently unsafe API. You should avoid using it as much as possible.
- For those cases in which you cannot avoid it, use the **self-pipe trick**, invented by D. J. Bernstein in the early 90s:
  - General idea: minimize what the signal handler does: just note down that *something has to be done* and let something else handle it later. In practice:
  - To await a signal, block reading from a pipe; idea: you will read something from the pipe upon signal reception.
  - In the signal handler: write a single byte to the pipe and return.
- Variants: main code periodically checks the pipe with polling, async IO, or a dedicated thread.
- The Linux-specific `signalfd` syscall (2007) is a kernel-supported version of the self-pipe trick.

Decent signal handling (in Rust)

- CrtlC crate: handles `SIGINT` only, using the self-pipe trick.
  - Spawns a dedicated thread that:

```
loop {
  // read from pipe ;
  // call previously registered handler function ;
}
```

  - The usual Rust guarantees about multithreaded code apply!

- Signal-hook crate: *"Library for safe and correct Unix signal handling in Rust."*
  - `iterator` module to iterate over pending signals synchronously, possibly in a dedicated thread; asynchronous variants exist as well.
  - `flag` module to react to specific signals by just setting a flag (safely) upon arrival.
  - `low_level` module to register arbitrary actions, with chaining.

Takeaways

- Many of the common patterns/API for handling parallel tasks in legacy system programming are inherently unsafe.
- We have reviewed some of the DONTS associated to the (still) popular legacy syscalls: `fork`, `pipe`, and `signal`.
- The general approach to solve these issues is working with higher-level safe abstractions, implemented once and for all.
  - `fork` → `Command`
  - `pipe` → `Command`, possibly `os_pipe`
  - `signal` → self-pipe trick, `ctrlc`, `signal-hook`

Credits

- These slides contain material and ideas reused with permission from lecture 10 of Stanford's course CS 110L (2021, 2022) by Ryan Eberhardt, Armin Namavari, Will Crichton, Julio Ballista, and Thea Rossman.