Universidade Federal do Rio de Janeiro

Universidade Federal
do Rio de Janeiro
Escola Politécnica

COS470 Sistemas Distribuídos

Trabalho Prático 1

Alunos	João Lacerda
	Igor Rocha
Professor	Daniel Ratton
Horário	Ter-Qui - 10:00-12:00

Rio de Janeiro, 04 de maio de 2022

Conteúdo

1 Introdução

O objetivo deste primeiro trabalho é explorar mecanismos de **IPC**: sinais, pipes e sockets. A linguagem escolhida para desenvolver o trabalho foi Rust, uma linguagem focada para sistemas que permite controle sobre operações de baixo nível. Todo o código do trabalho encontra-se neste repositório: https://github.com/jpedrodelacerda/cos470-sd, na pasta *trabalho1*.

2 Sinais

Para esta etapa, dois programas foram criados:

- `src/bin/signal/signal_consumer.rs`
- `src/bin/signal/signal_producer.rs`

O processo consumidor trata 3 sinais e imprime uma mensagem diferente para cada um deles enquanto o processo produtor envia sinais para um processo dado seu PID, no caso o processo consumidor.

2.1 Consumidor

No arquivo consumidor, são criadas as *signal handlers* e então registradas no processo. Além disso, são definidas as formas de espera do processo: *blocking wait* e *busy wait* que é passado através de um parâmetro.

2.1.1 Sintaxe

O consumidor deve ser executado da seguinte forma:

```
cargo run --bin signal_consumer [WAIT]
```

Onde o parâmetro `[WAIT]` deve ser `BLOCKING` ou `BUSY`.

2.1.2 *Blocking* vs *Busy wait*

O processo do consumidor pode operar de duas formas: *busy wait* ou *blocking wait*. No *busy wait*, o processo fica ocupando a CPU enquanto aguarda o sinal, enquanto o *blocking wait* coloca o processo em *Sleeping* até que o Sistema Operacional seja notificado que algum sinal chegou para ele. Para identificar o status de um processo com `pid` 1234, devemos executar o seguinte comando:

```
cat /proc/1234/status | grep State
```

A função de *busy wait* é apenas um loop infinito.

```
fn busy_wait() {
    println!("Running BUSY WAIT");
    loop {}
}
```

Na função do *blocking wait*, utilizamos a função `pause` para sinalizar para o SO que aquele processo pode ir para o estado Sleeping.

```
fn blocking_wait() {
    println!("Running BLOCKING WAIT");
    loop {
        unistd::pause();
    }
}
```

2.1.3 *Signal handlers*

As *signal handlers* foram implementadas utilizando a capacidade de interoperabilidade entre `C` e `Rust`.

```
extern "C" fn sigterm_handler(sig: libc::c_int) {
    println!("I received SIGTERM ({}). Bye bye!", sig);
    std::process::exit(0)
```

Depois de criadas, as *signal handlers* foram instaladas utilizando a função `nix::sys::signal::sigaction`.

2.2 Produtor

O produtor checa num primeiro momento se o `PID` existe e, em caso positivo, utiliza a função `nix::sys::signal::kill` para enviar os sinais para o `PID` fornecido.

2.2.1 Sintaxe

O produtor deve ser executado da seguinte forma:

```
cargo run --bin signal_producer [PID] [SIGNAL_CODE]
```

3 Estruturas

Como a lógica dos produtores e consumidores para as etapas de *pipes* e *sockets* são similares, foi criada uma única estrutura para o produtor e uma única estrutura para o consumidor.

3.1 Produtor

O produtor possui a seguinte estrutura:

```
pub struct Producer {
    writer: Option<File>,       // Escrever em file descriptor
    stream: Option<TcpStream>,  // Ler e escrever em socket
}
```

A estrutura é criada utilizando a função `sd::Producer::fron_fd`, consumindo um file descriptor, ou a função `sd::Producer::from_socket`, consumindo um socket. O produtor possui uma única função de escrita, `sd::Producer::write` que se adapta aos dois casos: *file descriptor* ou *socket*.

3.2 Consumidor

O consumidor possui a seguinte estrutura:

```
pub struct Consumer {
    reader: Option<Box<dyn Read>>, // Ler file descriptor
    listener: Option<TcpListener>, // Ler e escrever em socket
}
```

A estrutura do consumidor é similar a do produtor: possui as funções `sd::Consumer::from_fd` e `sd::Consumer::from_socket` com uma única função de leitura (que escreve na TcpStream quando é um `Consumer` criado via `sd::Consumer::from_socket`).

4 Pipe

Com o *pipe*, o programa criado, `src/bin/pipe/main.rs` recebe um número N e o produtor envia, através do pipe, $N + 2$ números para o consumidor através do pipe ($\mathrm{N}_0 = 1, N_n = 0$).

4.1 Criação do *pipe*

A criação do *pipe* é feita através da função `nix::unistd::pipe`, que no caso de sucesso, retorna um par de `std::os::unix::io::RawFd`, *file descriptors*.

4.2 *Fork*

Para a criação do produtor e do consumidor, foi utilizada a função `nix::unistd::fork`. Após o *fork*, são criadas as estruturas do produtor e consumidor utilizando os métodos `sd::Producer::from_fd` e `sd::Consumer::from_fd` e passando para cada um deles, um `std::os::unix::io::RawFd` para que pudessem escrever e ler no pipe.

4.3 Execução

- `cargo run --bin pipe [N]`

5 *Socket*

Para esta etapa, froam criados dois programas diferentes: um produtor e um consumidor no diretório `src/bin/socket/`.

O produtor é instanciado utilizando a função `sd::Producer::from_socket` passando o endereço do *socket* e depois envia os números executando a função `sd::Producer::produce_random_ints`.

```
let mut producer =
    Producer::from_socket("127.0.0.1:31337".to_string())
    .expect("failed to create producer");
producer.produce_random_ints(number_count).unwrap();
```

Após enviar o número para o consumidor, o produtor aguarda uma resposta de forma bloqueante. A mudança nesse comportamento pode ser feita utilizando o método `std::net::TcpStream::set_nonblocking`.

O consumidor é criado utilizando a função `sd::Consumer::from` passando o endereço do *socket*.

```
let mut consumer =
    Consumer::from_socket("127.0.0.1:31337".to_string())
    .expect("failed to create consumer");
let _ = consumer.read().unwrap();
```

5.1 Execução

- Consumidor: `cargo run --bin socket_consumer`
- Produtor: `cargo run --bin socket_producer [N]`