.UBAfiuba
FACULTAD DE INGENIERÍA

Taller de Programación I

Introducción a Concurrencia en Rust

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

.UBAfiuba FACULTAD DE INGENIERÍA

Índice

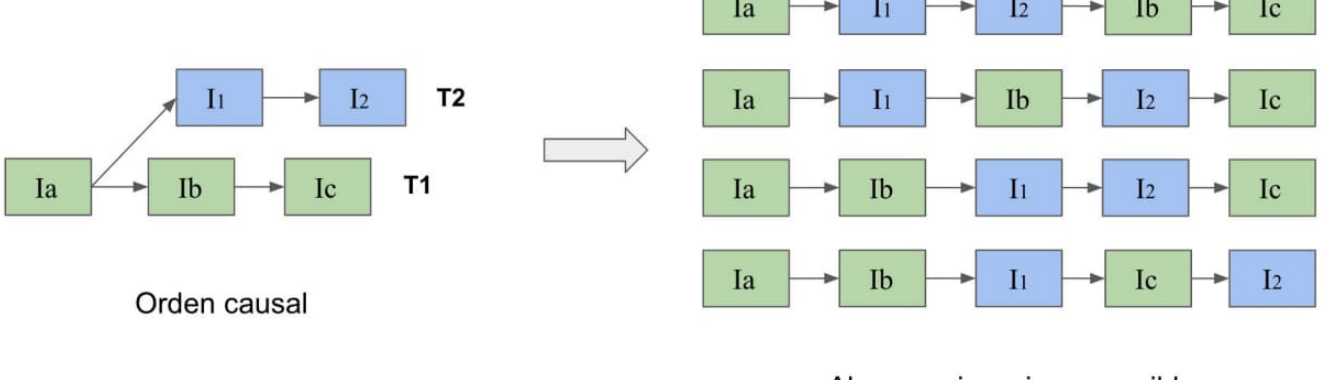
Qué es la concurrencia?

- **Programación Concurrente:** diferentes partes de un programa se ejecutan independientemente.
- **Programación Paralela:** diferentes partes de un programa se ejecutan al mismo tiempo.

Definiciones

- **Programa concurrente:** consiste en un conjunto finito de procesos secuenciales.
- **Procesos:** están compuestos por un conjunto finito de instrucciones atómicas.
- **Ejecución del programa concurrente:** resulta al ejecutar una secuencia de instrucciones atómicas que se obtiene de intercalar arbitrariamente las instrucciones atómicas de los procesos que lo componen.

Ejecución del programa concurrente



Desafíos de la concurrencia

Necesidad de sincronizar y comunicar procesos diferentes.

- **Sincronización:** coordinación temporal entre distintos procesos
- **Comunicación:** datos que necesitan compartir los procesos para cumplir la función del programa

Problema de la Programación Concurrente

- **Condiciones de Carrera:** donde hilos de ejecución acceden a datos o a recursos en un orden inconsistente.
- **Deadlocks:** donde dos hilos de ejecución están esperando el uno por el otro para avanzar, requiriendo de un recurso que el otro tiene, previniendo que ambos avancen. No hay *progreso productivo*.
- Aparición de bugs que se manifiestan un escenario particular y son difíciles de reproducir.

.UBAfiuba FACULTAD DE INGENIERÍA

Índice

Procesos en UNIX

Un proceso está formado por:

- Programa: instrucciones que conforman el programa a ejecutar
- Datos del usuario: espacio de memoria modificable por el usuario, por ejemplo: datos propios del programa, heap
- Pila del sistema: se utiliza para almacenar parámetros y direcciones de retorno durante el llamado a subrutinas
- Estructuras de datos del kernel:
  - Fila en la tabla de procesos: PCB (Process Control Block)

Procesos en UNIX (II)

Información de control que el SO almacena en el *PCB*:

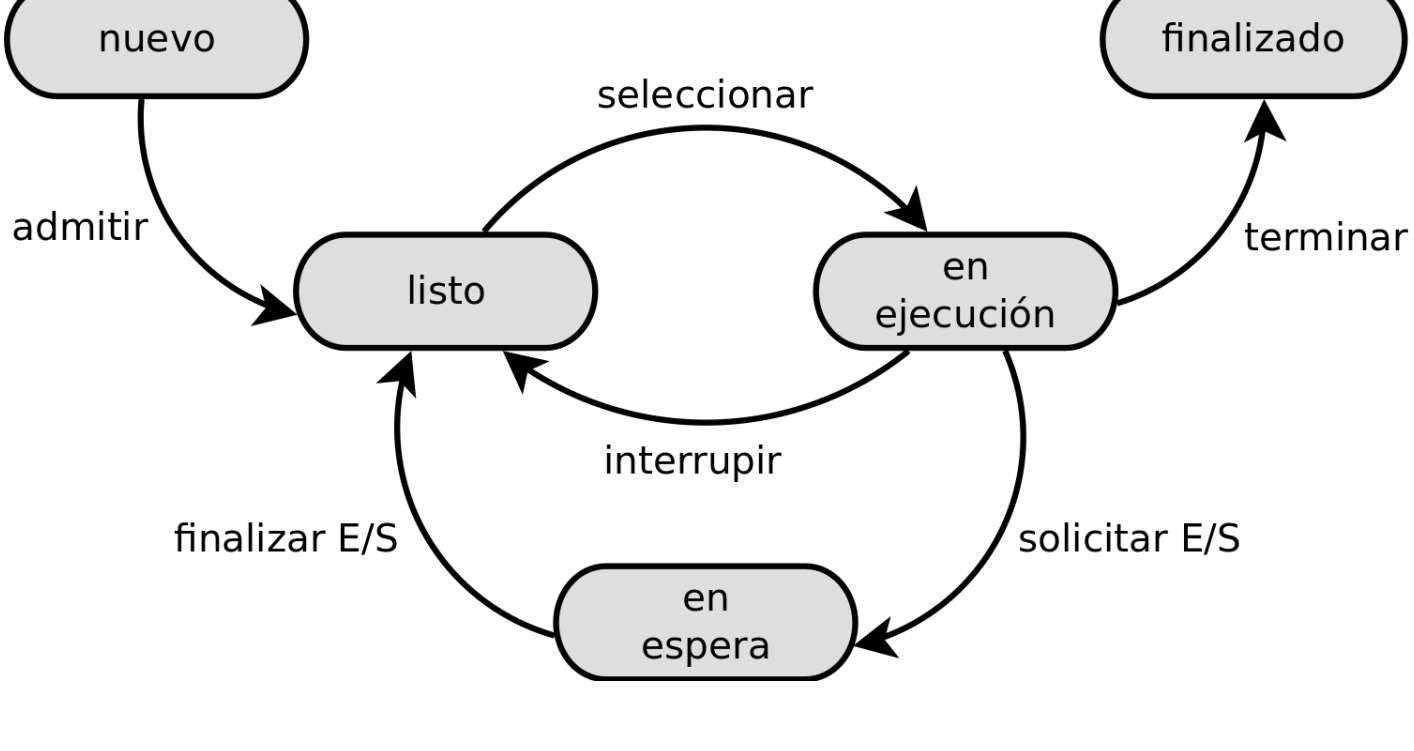
- Datos de identificación del proceso
  - Identificador del proceso
  - Identificador del proceso que lo creó
  - Identificador del usuario y del grupo
- Datos del estado del proceso: permite suspender y retomar el proceso
  - Registros de propósito general de la CPU
  - Stack pointer
  - Instruction pointer
- Datos de control del proceso
  - Estado del proceso y otra información de *scheduling* (p.ej. prioridad)
  - Estructura del proceso: identificadores de los hijos
  - Datos de IPC (señales, mensajes)
  - Contadores de tiempo de uso de CPU

Creación de un proceso

¿En qué momento se crea un proceso?

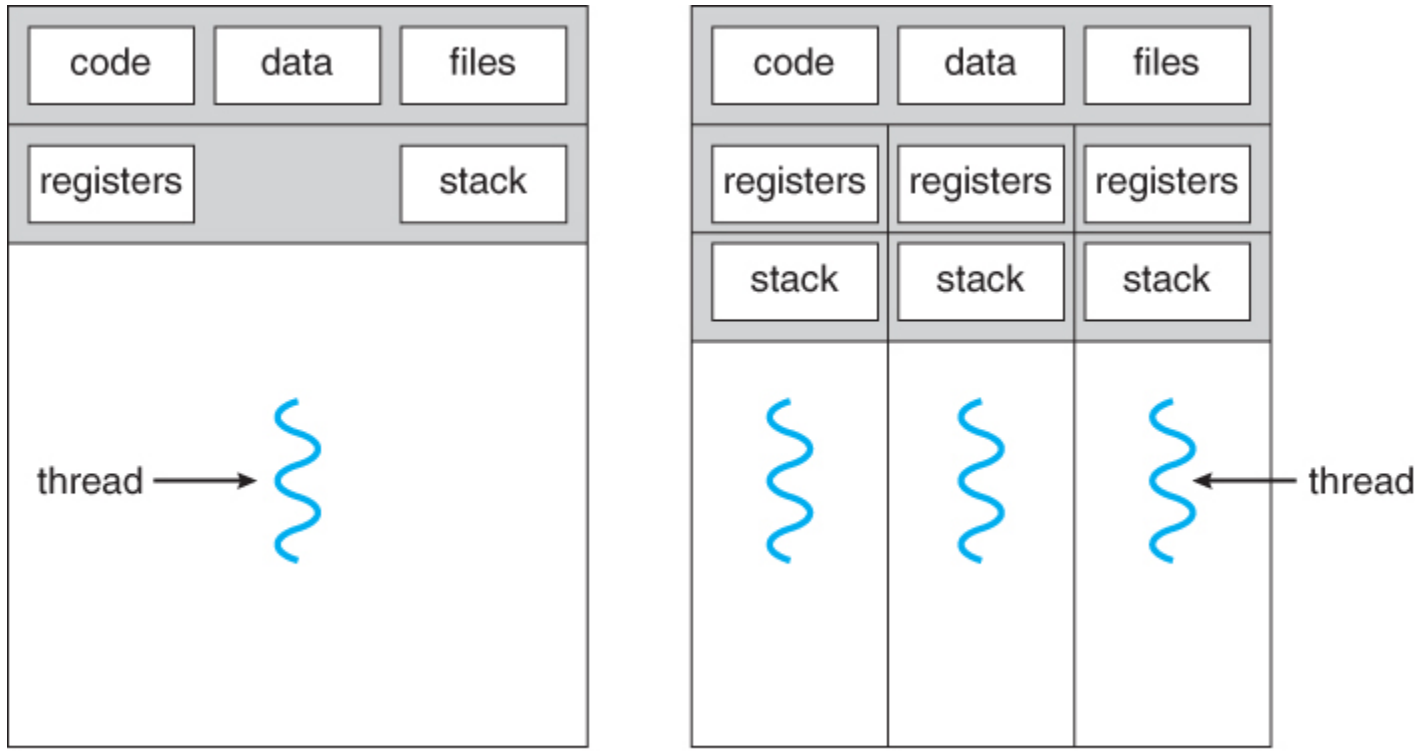
- Inicialización del sistema
- Ejecución de una llamada al sistema por parte de otro proceso que está en ejecución (en breve veremos cuál es)
- Por pedido del usuario
- Inicio de un *batch job*

Estados de ejecución de un proceso (teórico)



Threads Introducción

Los Threads comparten los recursos del proceso, entre ellos, el espacio de memoria. Cada thread mantiene su propia información de estado (stack, PC, registros).



Índice

Concurrencia en Rust

- El mecanismo de ownership y el sistema de tipos de Rust previene los problemas de la Concurrencia en tiempo de compilación.
- Los errores se manifiestan en tiempo de compilación -> *fearless concurrency*

Existen dos modelos de mapear threads: *green threads* y usar *threads del sistema operativo* (el modelo de Rust).

Threads en Rust (I)

Rust permite crear Threads del SO, utilizando la función `thread::spawn()`.
Recibe como parámetro un *moving closure*.
Retorna el handle del thread (*JoinHandle<T>*).

```
let join_handle: thread::JoinHandle<_> =
thread::spawn(|| {
        // codigo del thread hijo
});
```

Los threads son programados por el shceduler del SO.

Threads en Rust (II)

El thread padre puede esperar a que un thread hijo creado finalice utilizando la función join().

```
pub fn join(self) -> Result<T>
```

Se invoca sobre el handle del thread obtenido con spawn:

```
child.join();
```

Threads en Rust (III) - Ejemplo

```rust
use std::thread;
static NTHREADS: i32 = 10;

// main thread
fn main() {
  // Vector para almacenar handles de los threads
  let mut children = vec![];

  for i in 0..NTHREADS {
    // Crear otro thread
    children.push(thread::spawn(move || {
      println!("this is thread number {}", i);
        }));
  }
  ...
```

Threads en Rust (III) - Ejemplo

```
...
for child in children {
  // Esperar que terminen los threads
  let _ = child.join();
  }
}
```

Threads en Rust (IV) - Otras funciones

- `thread::sleep();`
  Suspende la ejecución del thread durante el tiempo dado. El tiempo se especifica con funciones de std::time::Duration, por ejemplo:

  ```
  thread::sleep(Duration::from_millis(1));
  ```

- `thread::yield_now();`
  Cede el timeslice al scheduler del SO.

Threads en Rust (V) - Otras características

- Los threads pueden tener un nombre: Sirve para identificar un thread que ejecuta `panic!`. El nombre puede obtenerse y usarse.
  Se crea el Thread con `thread::Builder`.

```
let builder = thread::Builder::new()
.name("mi_hilo".into());

let handler = builder.spawn(|| {
  assert_eq!(thread::current().name(),
  Some("hilo"))
}).unwrap();
```

Threads en Rust (VI) - Otras características

- También se puede configurar el tamaño del stack del thread (`stack_size()`).

Introducción: Traits Send y Sync

- El *trait marker* **Send** indica que el ownership del tipo que lo implementa puede ser transferido entre threads.
- Casi todos los tipos de Rust son **Send**. Hay excepciones: ej `Rc<T>`.
- Los tipos compuesto que están formados por tipos **Send** automáticamente son **Send**. Casi todos los tipos primitivos son **Send**, excepto *raw pointers*.

Introducción: Traits Send y Sync

- El *trait marker* **Sync** indica que es seguro para el tipo que implementa Sync ser referenciado desde múltiples threads.
- Esto es: T es **Sync** si **&T** (una referencia a T) es **Send**.
- Los tipos primitivos son **Sync** y los tipos compuesto que están formados por tipos **Sync** automáticamente son **Sync**.

Pasaje de mensajes

"Do not communicate by sharing memory; instead, share memory by communicating."

Canales

- Un canal tiene dos extremos: un emisor y un receptor.
- Una parte del código invoca métodos sobre el transmisor, con los datos que se quiere enviar.
- Otra parte chequea el extremo de recepción por la existencia de mensajes.
- Múltiples productores, un consumidor.
- **Transfieren el ownership del elemento enviado.**
- Para crear múltiples productores, se clona el extremo de envío.

Canales

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
  let (tx, rx) = mpsc::channel();

  thread::spawn(move || {
    let val = String::from("Hola");
      tx.send(val).unwrap();
  });

  let received = rx.recv().unwrap();
  println!("Recibido: {}", received);
}
```

Locks (teórico)

- Sirven para realizar exclusión mutua entre procesos
- Se implementan mediante variables de tipo *lock*, que contienen el estado del mismo
- Se utilizan mediante los métodos *lock()* y *unlock()*
  - Método *lock()*: el proceso se bloquea hasta poder obtener el lock.
  - Método *unlock()*: el proceso libera el lock que tomó previamente con lock.
- Para la implementación se necesita soporte tanto del hardware como del sistema operativo.

Locks en Rust

Rust provee locks compartidos (de lectura) y locks exclusivos (de escritura) en el módulo: `std::sync::RwLock`.
No se provee una política específica, sino que es dependiente del sistema operativo.
Se requiere que **T** sea **Send** para ser compartido entre threads y **Sync** para permitir acceso concurrente entre lectores.

```
use std::sync::RwLock;

let lock = RwLock::new(5);
```

Obtener Locks (I)

Obtener un lock de lectura

`fn read(&self) -> LockResult<RwLockReadGuard<T>>`

Bloquea al thread hasta que se pueda obtener el lock con acceso compartido. Puede haber otros threads con el lock compartido.

Obtener un lock de escritura

`fn write(&self) -> LockResult<RwLockWriteGuard<T>>`

Bloquea al thread hasta que se pueda obtener el lock con acceso exclusivo.

Retornan una protección que libera el lock con RAII.

Una vez obtenido el lock, se puede acceder al valor protegido.

Obtener Locks (II)

Ejemplo de lock de lectura

```
use std::sync::RwLock;

fn main() {
  let lock = RwLock::new(1);

  let n = lock.read().unwrap();

  println!("El valor encontrado es: {}", *n);

  assert!(lock.try_write().is_err());
}
```

Locks Envenenados

Un lock queda en estado *envenenado* cuando un thread lo toma de forma exclusiva (write lock) y mientras tiene tomado el lock, ejecuta `panic!`.

Las llamadas posteriores a `read()` y `write()` sobre el mismo lock, devolverán Error.

Bibliografía

- **The Rust Programming Language**, *https://doc.rust-lang.org/book/*
  - Chapter 15 16. Fearless Concurrency
- **Programming Rust: Fast, Safe Systems Development**, 1st Edition, Jim Blandy, Jason Orendorff. 2017.
  - Chapter 19 Concurrencia
- **Operating System Concepts**, Ninth Edition, Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, Cap. 4.
- **The Design of the Unix Operating System**, Maurice Bach
- **Modern Operating Systems**, Andrew S. Tanenbaum, Cuarta ed.