.UBAfiuba
FACULTAD DE INGENIERÍA

Técnicas de Programación Concurrente I

Corrección / Locks

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

Facultad de Ingeniería
Universidad de Buenos Aires

Índice

Corrección

- En procesos (programas secuenciales) es suficiente con debuggear para encontrar errores, ya que ante una misma entrada se obtiene siempre la misma salida.
- En programas concurrentes, la salida puede depender del escenario que resultó en la ejecución.

Propiedades de Corrección

- *Safety:* debe ser verdadera siempre
- *Liveness:* debe volverse verdadera eventualmente

Propiedades tipo Safety

- *Exclusión mutua:* dos procesos no deben intercalar ciertas (sub)secuencias de instrucciones. Ejemplo: incremento de variable global.
- *Ausencia de deadlock:* un sistema que aún no finalizó debe poder continuar realizando su tarea, es decir, avanzar productivamente.

Propiedades tipo Liveness

- *Ausencia de starvation:* todo proceso que esté listo para utilizar un recurso debe recibir dicho recurso eventualmente
- *Fairness:* (equidad o justicia)

Un escenario es (débilmente) fair, si en algún estado en el escenario, una instrucción que está continuamente habilitada, eventualmente aparece en el escenario.

Índice

Definición del Problema

- Cada proceso se ejecuta en un loop infinito cuyo código puede dividirse en parte crítica y parte no-crítica
- Especificaciones de corrección:
  - Exclusión mutua: no deben intercalarse instrucciones de la sección crítica
  - Ausencia de deadlock: si dos procesos están tratando de entrar a la sección crítica, eventualmente alguno de ellos debe tener éxito
  - Ausencia de starvation: si un proceso trata de entrar a la sección crítica, eventualmente debe tener éxito

Sección Crítica (II)

- La sección crítica debe progresar (finalizar eventualmente)
- La sección no-crítica no requiere progreso (el proceso puede terminar o entrar en un loop infinito)

.UBAfiuba FACULTAD DE INGENIERÍA

Índice

Locks

- 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.

Índice

Introducción (I)

- Mecanismo de sincronismo de acceso a un archivo
- Se pueden utilizar también para sincronizar el acceso a cualquier otro recurso
- En Unix son *advisory*, es decir, los procesos pueden ignorarlos
- Dos tipos de locks:
  - Locks de lectura o *shared locks*: más de un proceso a la vez puede tener el lock
  - Locks de escritura o *exclusive locks*: sólo un proceso a la vez puede tener cualquier tipo de lock

Introducción (II)

Condiciones para poder tomar un lock

- Para poder tomar un *shared (read) lock*, el proceso debe esperar hasta que sean liberados todos los *exclusive locks*
- Para poder tomar un *exclusive (write) lock*, el proceso debe esperar hasta que sean liberados todos los locks (de ambos tipos)

Establecimiento de un lock

Pasos para establecer un lock:

1. Abrir el archivo a lockear
2. Aplicar el lock
   - 2.1 Mediante *fcntl()*
     - Completar los campos de la estructura `struct flock`
     - Utilizar *fcntl()*
   - 2.2 Mediante *flock()*
   - 2.3 Función *lockf()*: interface construida sobre *fcntl()*

Índice

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** if **&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**.

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

- **Principles of Concurrent and Distributed Programming**, M. Ben-Ari, Segunda edición (capítulos 1 y 2)
- Bibliografía de Rust.
- *Unix Network Programming, Interprocess Communications*, W. Richard Stevens, segunda edición