Arquitectura de Sistemas

Práctica 9: Exclusión mutua

Gustavo Romero López

Updated: 14 de mayo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

Objetivos

- **Verificar** la existencia de una **condición de carrera** en un programa que accede a un recurso compartido, el terminal, para imprimir un cierto paralelo.
- **Implementar** las diferentes **soluciones** vista en teoría para resolver el problema, tanto las que **funcionan** como las que sabemos que **no**.
- Implementar **otras** soluciones de la literatura.
- **Comparar** el rendimiento de cada uno de ellos midiendo...
  - El número de **paralelos correctos** impresos.
  - El número de **paralelos totales** impresos.
  - La justicia, el número de **hebras diferentes** ejecutadas.
  - El **tiempo** empleado (real/usuario/sistema).

Soluciones propuestas

- Algoritmo de la panadería de Lamport.
- Cerrojo: versión incorrecta con condición de carrera.
- Cerrojo implementado mediante `test_and_set`.
- Cerrojo implementado mediante el builtin de gcc `__sync_lock_test_and_set()`[1].
- Cerrojo implementado mediante `test_and_test_and_set`.
- Semáforo binario de C++11: `std::mutex`.
- Envoltorios RAII: `std::lock_guard` y `std::scoped_lock`.
- Ticket lock.

- Marcha atrás exponencial (exponential backoff).

[1] `https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html`

Programa de partida: secuencial.cc I

```
22  void seccion_critica()
23  {
24      std::cout << "[" << std::this_thread::get_id() << "]: ";
25      for (size_t i = 0; i < 10; ++i) std::cout << i;
26      std::cout << '\n';
27  }
28
29  //----------------------------------------------------
30
31  void hebra()
32  {
33      latch.arrive_and_wait();
34      while (!stop) seccion_critica();
35  }
```

Paralelización con condición de carrera: `paralelo.cc`

- **Verifique** la existencia de una condición de carrera.

```
int main()
{
    std::jthread threads[N];

    for (auto &i: threads) i = std::jthread(hebra);

    std::this_thread::sleep_for(75ms);
    stop = true;
}
```

- **Compare** los resultados de `secuencial.cc` y `paralelo.cc`.

Algoritmo de la panadería de Lamport: lamport.cc

- **Modifique** `paralelo.cc` de forma que se utilice el algoritmo de la panadería para conseguir la exclusión mutua en el acceso a la sección crítica por parte de las hebras.
- **Compare** `paralelo.cc` con `lamport.cc` observando todos los parámetros señalados como objetivos.

Cerrojo con condición de carrera: cerrojo.cc

- **Descargue** `cerrojo.cc`.
- Esta versión vista en clase que era **incorrecta** por contener una condición de carrera.

```
class cerrojo
{
public:
    void adquirir()
    {
        while (cerrado);
        cerrado = true;
    }

    void liberar() { cerrado = false; }

private:
    volatile bool cerrado = false;
} c;
```

- **Compare** `paralelo.cc`, `lamport.cc` y `cerrojo.cc`.

Cerrojo correcto con test_and_set

- **Modifique** cerrojo.cc de forma que ahora funcione correctamente mediante el empleo de instrucciones atómicas.

```
22  template<typename T> T test_and_set(volatile T *spinlock)
23  {
24      T ret = true;
25      __asm__ __volatile__("xchg %0, %1" : "+r"(ret), "+m"(*spinlock));
26      return ret;
27  }
31  class cerrojo
32  {
33  public:
34      void adquirir() { while (test_and_set(&cerrado)); }
35      void liberar() { cerrado = false; }
36
37  private:
38      volatile bool cerrado = false;
39  } c;
```

- **Compare** paralelo.cc, lamport.cc, cerrojo.cc y tas.cc.

builtin de gcc `__sync_lock_test_and_set()`

- **Modifique** el ejemplo anterior de forma que utilice el **builtin** de gcc `__sync_lock_test_and_set()`[2] en lugar de nuestra versión casera del mismo.
- **Compare** `paralelo.cc`, `lamport.cc`, `cerrojo.cc`, `tas.cc` y `tasb.cc`.

[2] `https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html`

`test and test_and_set`: menos tráfico en el bus

ttas

```
void adquirir()
{
    while (ocupado == true || test_and_set(ocupado));
}
```

- Menor tráfico en el bus de memoria frente a `tas()`.
- Verifica si el cerrojo está cerrado accediendo a una variable en la caché local de cada procesador sin tener que acceder al cerrojo en memoria y así evita tráfico en el bus del sistema.
- **Modifique** el ejemplo anterior para hacer uso del doble test.
- **Compare** `paralelo.cc`, `lamport.cc`, `cerrojo.cc`, `tas.cc`, `tasb.cc` y `ttas.cc`.

`std::atomic` de C++11: operaciones atómicas

- Las operaciones atómicas no deberían ser tan difíciles de usar y por eso han aparecido clases para poder utilizarlas fácilmente.
- Lo que hemos hecho hasta ahora puede conseguirse de manera más práctica empleando `std::atomic<bool>` o su especialización `std::atomic_flag`.
- `std::atomic_flag` aporta dos ventajas:
  1. implentación libre de bloqueos (lockfree).
  2. permite bloquear/desbloquear hebras para evitar la espera ocupada, ahora también en `std::atomic<bool>`.
- **Modifique** el ejemplo anterior para hacer uso de alguna de las clases mencionadas.
- **Compare** `paralelo.cc`, `lamport.cc`, `cerrojo.cc`, `tas.cc`, `tasb.cc`, `ttas.cc`, `atomic.cc` y `flag.cc`.

- Encontrarse el trabajo ya hecho es lo mejor... :)
- **Modifique** alguno de los programas anteriores para que haga uso de los semáforos binarios de C++11, `std::mutex`[3].
- **Compare** `paralelo.cc`, `lamport.cc`, `cerrojo.cc`, `tas.cc`, `tasb.cc`, `ttas.cc`, `atomic.cc`, `flag.cc` y `mutex.cc`.

[3] `http://en.cppreference.com/w/cpp/thread/mutex`

lock_guard/scoped_lock/unique_lock

- ¿Le suena **RAII**: "Resource Acquisition Is Initialization"[4]?
- Los tres son envoltorios RAII para `std::mutex`:
  - `std::lock_guard`: un `std::mutex` y ámbito completo.
  - `std::scoped_lock`: al menos un `std::mutex` y ámbito completo.
  - `std::unique_lock`: necesita desbloquear el `std::mutex` antes del final de ámbito y uso con `std::condition_variable`.
- **Modifique** alguno de los programas anteriores para que haga uso de estos envoltorios..
- **Compare** `paralelo.cc`, `lamport.cc`, `cerrojo.cc`, `tas.cc`, `tasb.cc`, `ttas.cc`, `atomic.cc`, `flag.cc`, `mutex.cc`, `lock_guard.cc`, `scoped_lock.cc` y `unique_lock.cc`.

[4] https://es.wikipedia.org/wiki/RAII

- Investigue por su cuenta los cerrojos basados en turnos o *ticket locks*[5].
- **Modifique** alguna de las soluciones anteriores para que consiga la exclusión mutua mediante el uso de "ticket locks".
- **Compare** `paralelo.cc`, `lamport.cc`, `cerrojo.cc`, `tas.cc`, `tasb.cc`, `ttas.cc`, `atomic.cc`, `flag.cc`, `mutex.cc`, `lock_guard.cc`, `scoped_lock.cc`, `unique_lock.cc` y `ticketlock.cc`.

[5] `https://www.cs.rice.edu/~johnmc/scalable_synch/tocs91.pdf`

Marcha atrás exponencial: más eficiente todavía

- La mayoría de los métodos vistos hasta ahora pueden mejorarse aun más si aplicamos la técnica conocida como marcha atrás exponencial (*"exponential backoff"*):
  - **Dejar libre el procesador** en caso de encontrar el cerrojo ocupado.
  - La cantidad de **tiempo va aumentando** a medida que crece el número de reintentos.
- **Modifique** las soluciones vistas hasta ahora para incorporar esta mejora.
- **Compare** ambos tipos de soluciones.
- La mejora de rendimiento tiene peaje: la justicia.

¿Qué es más importante, el algoritmo o la implementación?

- **¿Sabes de alguna técnica/truco/implementación capaz de mejorar lo visto hasta ahora?**
- **¡Pruébalo y cuéntanoslo!**