Acciones atómicas

Programación concurrente

Repaso:

- Un programa concurrente es un conjunto de programas secuenciales (su ejecución produce una traza de un conjunto de trazas posibles).
- Los programas concurrentes pueden hacer un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Para un programa concurrente funcione correctamente, hay que garantizar:
  - **Safety:** Nada malo ocurre nunca
  - **Liveness:** Algo bueno eventualmente ocurre

Repaso:

Sección crítica

Llamamos sección crítica a la parte del programa que accede a memoria compartida y que deseamos que sea ejecutada atómicamente.

Exclusión mutua

Llamamos exclusión mutua (o mutex) al problema de asegurar que dos *threads* no ejecuten una sección crítica simultáneamente.

Asumiendo:

- No hay variables compartidas entre sección crítica y no crítica.
- La sección crítica siempre termina.
- El *scheduler* es débilmente *fair*.

Repaso:

Requerimientos de la exclusión mutua:

1. **Mutex:** En cualquier momento hay como máximo un thread en la región crítica.

2. **Garantía de entrada:** Un thread intentando entrar a su sección crítica tarde o temprano lo logrará.

Repaso:

Operación atómica

Una operación es llamada atómica si se ejecuta de forma indivisible, es decir, que el proceso ejecutándola no puede ser desplazado (por el scheduler) hasta que se completa su ejecución.

Las operaciones atómicas son la unidad más pequeña en la que se puede listar una traza, pues no hay *interleavings* posibles en tales operaciones.

Repaso:

- ¿Podemos garantizar exclusión mutua usando únicamente lectura y escritura a memoria?
- Vimos las siguientes soluciones al problema de exclusión mutua:
  - 2 threads: Peterson
  - N threads: Bakery (Lamport)

Pregunta

¿Cuál es la complejidad del algoritmo de Bakery?

```
global boolean[] sacandoTicket = new boolean[n];
global int[] ticket = new int[n];

thread {
  id = 0;
  // seccion no critica
  sacandoTicket[id] = true;
  ticket[id] = 1 + maximo(ticket);
  sacandoTicket[id] = false;
  for (j : range(0,n)) {
    while (sacandoTicket[j]);
    while (ticket[j] != 0 &&
           (ticket[j] < ticket[id] ||
           (ticket[j] == ticket[id] && j < id)));
  }

  // SECCION CRITICA

  ticket[id] = 0;
  // seccion no critica
}
```

Pregunta

En otras palabras, ¿Cuánto operaciones se deben ejecutar para que un thread acceda a la sección crítica?

- Si no es el thread que tiene el menor ticket, entonces la espera puede ser tan larga como queramos (no está acotada).
- Si es el thread que tiene el menor ticket, entonces tiene que ejecutar $O(n)$ operaciones, donde n es la cantidad de threads.

Más preguntas

- ¿Podrá hacerse más eficiente? Requiriendo menos de $O(n)$ operaciones para acceder a la sección crítica el thread que puede hacerlo.
- ¿Qué otras operaciones atómicas pueden idearse para resolver el problema de la exclusión mutua?

Recordando: Intento I

```
global boolean flag = false;

thread { //                    thread {
  // seccion no critica          // seccion no critica
  while (flag);                  while (flag);
  flag = true;                   flag = true;
  // SECCION CRITICA             // SECCION CRITICA
  flag = false;                  flag = false;
  // seccion no critica          // seccion no critica
}                              }
```

Recordando: Intento I

```
global boolean flag = false;

thread { //                    thread {
  // seccion no critica          // seccion no critica
  while (flag);                  while (flag);
  flag = true;                   flag = true;
  // SECCION CRITICA             // SECCION CRITICA
  flag = false;                  flag = false;
  // seccion no critica          // seccion no critica
}                              }
```

- ¿Cuál era el problema con esto?

Recordando: Intento I

```
global boolean flag = false;

thread { //                    thread {
  // seccion no critica          // seccion no critica
  while (flag);                  while (flag);
  flag = true;                   flag = true;
  // SECCION CRITICA             // SECCION CRITICA
  flag = false;                  flag = false;
  // seccion no critica          // seccion no critica
}                              }
```

- ¿Cuál era el problema con esto? **No cumple Mutex**

Recordando: Intento I

```
global boolean flag = false;

thread { //                    thread {
  // seccion no critica          // seccion no critica
  while (flag);                  while (flag);
  flag = true;                   flag = true;
  // SECCION CRITICA             // SECCION CRITICA
  flag = false;                  flag = false;
  // seccion no critica          // seccion no critica
}                              }
```

- ¿Cuál era el problema con esto? **No cumple Mutex**
- ¿Es menos complejo que Bakery?

Recordando: Intento I

```
global boolean flag = false;

thread { //                    thread {
  // seccion no critica          // seccion no critica
  while (flag);                  while (flag);
  flag = true;                   flag = true;
  // SECCION CRITICA             // SECCION CRITICA
  flag = false;                  flag = false;
  // seccion no critica          // seccion no critica
}                              }
```

- ¿Cuál era el problema con esto? **No cumple Mutex**
- ¿Es menos complejo que Bakery? **Sí**

Recordando: Intento I

```
global boolean flag = false;

thread { //                    thread {
  // seccion no critica          // seccion no critica
  while (flag);                  while (flag);
  flag = true;                   flag = true;
  // SECCION CRITICA             // SECCION CRITICA
  flag = false;                  flag = false;
  // seccion no critica          // seccion no critica
}                              }
```

- ¿Cuál era el problema con esto? **No cumple Mutex**
- ¿Es menos complejo que Bakery? **Sí**
- ¿Podemos idear una instrucción atómica que nos permita corregirlo? ¿Qué es lo que tendría que hacer?

Test and Set

Definimos

```
class Ref{
  boolean value;
  public Ref(boolean v){this.value=v;}
}

atomic boolean TestAndSet(Ref ref) {
  boolean result = ref.value; // lee el valor antes de cambiarlo
  ref.value = true;    // cambia el valor por true
  return result;       // retorna el valor leido anterior
}
```

Test and Set

Definimos

```
class Ref{
  boolean value;
  public Ref(boolean v){this.value=v;}
}

atomic boolean TestAndSet(Ref ref) {
  boolean result = ref.value; // lee el valor antes de cambiarlo
  ref.value = true;    // cambia el valor por true
  return result;       // retorna el valor leido anterior
}

 global Ref shared = new Ref(false);

 thread {
   // seccion no critica
   while( TestAndSet(shared) );
   // SECCION CRITICA
   shared.value = false;
   // seccion no critica
 }
```

```
thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

TestAndSet

Veamos si cumple las propiedades

```
global Ref shared = new Ref(false);

thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:**
- **Garantía de entrada:**

TestAndSet

Veamos si cumple las propiedades

```
global Ref shared = new Ref(false);

thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:** Sí, sólo el 1ro que ejecuta TestAndSet toma el flag.
- **Garantía de entrada:**

TestAndSet

Veamos si cumple las propiedades

```
global Ref shared = new Ref(false);

thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:** Sí, sólo el 1ro que ejecuta TestAndSet toma el flag.
- **Garantía de entrada:** Casi, el flag comienza y se restablece a false, por lo que el 1er thread que puede tomar el flag lo hace y entra. Pero puede haber *starvation*.

TestAndSet

```
global Ref shared = new Ref(false);

thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

- Si el thread es el que debe acceder a la sección crítica, ¿cuánto tiempo tarda en hacerlo?

- ¿Para cuántos threads es generalizable esta solución?

TestAndSet

```
global Ref shared = new Ref(false);

thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

- Si el thread es el que debe acceder a la sección crítica, ¿cuánto tiempo tarda en hacerlo?
  - $O(1)$, ya que sólo tiene que ejecutar TestAndSet.
- ¿Para cuántos threads es generalizable esta solución?

TestAndSet

```
global Ref shared = new Ref(false);

thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

- Si el thread es el que debe acceder a la sección crítica, ¿cuánto tiempo tarda en hacerlo?
  - $O(1)$, ya que sólo tiene que ejecutar TestAndSet.
- ¿Para cuántos threads es generalizable esta solución?
  - Esta solución sirve para N threads (pero si N es no acotado hay chance de starvation).

Pregunta

¿Podemos idear una instrucción atómica que resuelva el problema de exclusión mutua para N threads, eficientemente, y sin posibilidad de *starvation*?

Fetch and Add

```
atomic int FetchAndAdd(IntRef ref, int x) {
  int local = ref.value;
  ref.value = ref.value + x;
  return local;
}
```

Fetch and Add

```
atomic int FetchAndAdd(IntRef ref, int x) {
  int local = ref.value;
  ref.value = ref.value + x;
  return local;
}
global IntRef ticket = new IntRef(0);
global int turn = 0;
thread {
  // seccion no critica
  int myTicket = FetchAndAdd(ticket, 1);
  while (myTicket.value != turn);
  // SECCION CRITICA
  turn++;
  // seccion no critica
}
```

Fetch and Add

```
atomic int FetchAndAdd(IntRef ref, int x) {
  int local = ref.value;
  ref.value = ref.value + x;
  return local;
}
global IntRef ticket = new IntRef(0);
global int turn = 0;
thread {
  // seccion no critica
  int myTicket = FetchAndAdd(ticket, 1);
  while (myTicket.value != turn);
  // SECCION CRITICA
  turn++;
  // seccion no critica
}
```

- **Mutex:**
- **Garantía de entrada:**

Fetch and Add

```
atomic int FetchAndAdd(IntRef ref, int x) {
  int local = ref.value;
  ref.value = ref.value + x;
  return local;
}
global IntRef ticket = new IntRef(0);
global int turn = 0;
thread {
  // seccion no critica
  int myTicket = FetchAndAdd(ticket, 1);
  while (myTicket.value != turn);
  // SECCION CRITICA
  turn++;
  // seccion no critica
}
```

- **Mutex:** Sí, sólo entra el de `myTicket` igual a `turno`, y los tickets son únicos e incrementales ya que FaA es atómica.
- **Garantía de entrada:**

Fetch and Add

```
atomic int FetchAndAdd(IntRef ref, int x) {
  int local = ref.value;
  ref.value = ref.value + x;
  return local;
}
global IntRef ticket = new IntRef(0);
global int turn = 0;
thread {
  // seccion no critica
  int myTicket = FetchAndAdd(ticket, 1);
  while (myTicket.value != turn);
  // SECCION CRITICA
  turn++;
  // seccion no critica
}
```

- **Mutex:** Sí, sólo entra el de `myTicket` igual a `turno`, y los tickets son únicos e incrementales ya que FaA es atómica.
- **Garantía de entrada:** Sí, todos los threads que llegan entran respetando el orden de llegada.

Fetch and Add

```
global IntRef ticket = new IntRef(0);
global int turn = 0;
thread {
  // seccion no critica
  int myTicket = FetchAndAdd(ticket, 1);
  while (myTicket.value != turn);
  // SECCION CRITICA
  turn++;
  // seccion no critica
}
```

- Si el thread es el que debe acceder a la sección crítica, ¿cuanto tiempo tarda en hacerlo?

- ¿Para cuántos threads es generalizable esta solución?

Fetch and Add

```
global IntRef ticket = new IntRef(0);
global int turn = 0;
thread {
  // seccion no critica
  int myTicket = FetchAndAdd(ticket, 1);
  while (myTicket.value != turn);
  // SECCION CRITICA
  turn++;
  // seccion no critica
}
```

- Si el thread es el que debe acceder a la sección crítica, ¿cuanto tiempo tarda en hacerlo?
  - $O(1)$, ya que únicamente debe ejecutar `FetchAndAdd` una vez.
- ¿Para cuántos threads es generalizable esta solución?
  - Puede generalizarse a N threads sin problemas.

Java Atomics

El paquete `java.util.concurrent.atomic` de la librería estándar de Java provee clases con el comportamiento de las operaciones clásicas vistas en esta clase.

Por ejemplo:

- `AtomicBoolean` posee un método `compareAndSet`.
- `AtomicInteger` posee un método `getAndAdd`.

En la materia optamos por implementarlas nosotros mismos para comprender su funcionamiento.

Resumen

- Suponiendo únicamente como operaciones atómicas la lectura y la escritura, sincronizar n threads podemos usar el algoritmo de Bakery.
  - El problema es que el algoritmo de Bakery tiene una complejidad para el caso en que el thread debe acceder a la sección crítica de $O(n)$.
- Las operaciones atómicas nos permiten solucionar el problema de acceso a la sección crítica.
- Todas las soluciones que vimos usando estas operaciones atómicas acceden a la sección crítica en $O(1)$.

Busy waiting

Todas las soluciones vistas hasta ahora son ineficientes dado que utilizan "spinlocks", que consumen tiempo de procesador ciclando hasta que una condición cambie (*busy-waiting*).

Normalmente es deseable suspender la ejecución de un proceso que intenta acceder a la sección crítica hasta tanto sea posible.

Próxima clase: **Semáforos**