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 hace 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 proceso en la región crítica.
2. **Ausencia de deadlocks y livelocks:** Si varios procesos intentan entrar a la sección crítica alguno lo logrará.
3. **Garantía de entrada:** Un proceso intentando entrar a su sección crítica tarde o temprano lo logrará.

Repaso:

- ¿Podemos garantizar exclusión mutua usando únicamente lectura y escritura a memoria?
- Vimos las siguientes soluciones al problema de exclusión mutua:
  - 2 threads: Dekker y 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

Es decir, ¿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 (es decir, 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? Es decir, tardar menos de $O(n)$ operaciones para acceder a la sección crítica el thread que puede hacerlo?
- ¿Qué otras acciones atómicas pueden idearse para resolver el problema de la exclusión mutua?

Definición

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.

Definición

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.

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 más sencillo de comprender 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 más sencillo de comprender que Bakery? **Es más sencillo**

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 más sencillo de comprender que Bakery? **Es más sencillo**
- ¿Podemos idear una instrucción atómica que nos permita corregirlo? ¿Qué es lo que tendría que hacer?

Test and Set

```
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

```
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();
shared.value = 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
}
```

Test and Set

```
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();
shared.value = 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
}
```

- **Mutex:** Sí
- **Ausencia livelock/deadlocks:**
- **Garantía de entrada:**

Test and Set

```
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();
shared.value = 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
}
```

- **Mutex:** Sí
- **Ausencia livelock/deadlocks:** Sí
- **Garantía de entrada:**

Test and Set

```
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();
shared.value = 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
}
```

- **Mutex:** Sí
- **Ausencia livelock/deadlocks:** Sí
- **Garantía de entrada:** Sí

TestAndSet

```
global Ref shared = new Ref();
shared.value = 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
}
```

- 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();
shared.value = 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
}
```

- Si el thread es el que debe acceder a la sección crítica, ¿cuánto tiempo tarda en hacerlo?
  - Como tiene que únicamente ejecutar TestAndSet una vez, y esta devuelve `false`, tarda $O(1)$.
- ¿Para cuántos threads es generalizable esta solución?

TestAndSet

```
global Ref shared = new Ref();
shared.value = 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
}
```

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

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp       = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}
```

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp        = sref.value;
    sref.value  = lref.value;
    lref.value  = temp;
}
```

Esquema:

1. Uso un "token" para indicar que quiero entrar a la sección crítica.
2. Intercambio mi "token" con uno global (de forma atómica).
3. Si el "token" que agarré dice que la sección crítica está libre puedo entrar, sino espero (repitiendo el intercambio).

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp        = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}

global Ref shared = new Ref();
shared.value = false;
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp        = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}

global Ref shared = new Ref();
shared.value = false;
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:**
- **Ausencia livelock/deadlocks:**
- **Garantía de entrada:**

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp        = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}

global Ref shared = new Ref();
shared.value = false;
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:** Sí
- **Ausencia livelock/deadlocks:**
- **Garantía de entrada:**

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp        = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}

global Ref shared = new Ref();
shared.value = false;
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:** Sí
- **Ausencia livelock/deadlocks:** Sí
- **Garantía de entrada:**

Exchange

```
atomic void Exchange(Ref sref, Ref lref) {
    temp        = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}

global Ref shared = new Ref();
shared.value = false;
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

- **Mutex:** Sí
- **Ausencia livelock/deadlocks:** Sí
- **Garantía de entrada:** Sí

Exchange

```
global Ref shared = new Ref();
shared.value = false;
thread {
  IntRef local = new IntRef();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // 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?

Exchange

```
global Ref shared = new Ref();
shared.value = false;
thread {
  IntRef local = new IntRef();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // 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?
  - Como tiene que únicamente ejecutar Exchange una vez, y esta devuelve 0, tarda $O(1)$.
- ¿Para cuántos threads es generalizable esta solución?

Exchange

```
global Ref shared = new Ref();
shared.value = false;
thread {
  IntRef local = new IntRef();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  Ref local = new Ref();
  local.value = true;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value);
  // 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?
  - Como tiene que únicamente ejecutar Exchange una vez, y esta devuelve 0, tarda $O(1)$.
- ¿Para cuántos threads es generalizable esta solución?
  - Puede generalizarse a N threads sin problemas.

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 acciones atómicas nos permiten solucionar el problema de acceso a la sección crítica.
- Por ejemplo:
  - TestAndSet()
  - Exchange()
- Todas las soluciones que vimos usando estas acciones atómicas acceden a la sección crítica en $O(1)$.

Busy waiting

Todas las soluciones vistas en esta clase son ineficientes dado que consumen tiempo de procesador en las esperas.

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

Próxima clase: **Semáforos**