Exclusión mutua

Programación concurrente

Hasta ahora vimos:

Programas secuenciales:

Programas concurrentes:

Hasta ahora vimos:

Programas secuenciales:

- Existe una **única** traza de ejecución.

Programas concurrentes:

- Existe un **conjunto** de trazas de ejecución.

Hasta ahora vimos:

Programas secuenciales:

- Existe una **única** traza de ejecución.
- Existe un **único** estado final.

Programas concurrentes:

- Existe un **conjunto** de trazas de ejecución.
- Existe un **conjunto** de estados finales.

Hasta ahora vimos:

Programas secuenciales:

- Existe una **única** traza de ejecución.
- Existe un **único** estado final.
- Criterio de correctitud sobre el estado final.

Programas concurrentes:

- Existe un **conjunto** de trazas de ejecución.
- Existe un **conjunto** de estados finales.
- Nos interesan propiedades de **safety** y **liveness**.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.

Hasta ahora vimos:

- 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.

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo de traza indeseable:

```
T1 -> counter + 1 ----> { counter = 0 }
```

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo de traza indeseable:

```
T1 -> counter + 1 ----> { counter = 0 }
T2 -> counter + 1 ----> { counter = 0 }
```

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo de traza indeseable:

```
T1 -> counter + 1 ----> { counter = 0 }
T2 -> counter + 1 ----> { counter = 0 }
T1 -> counter = 1 ----> { counter = 1 }
```

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo de traza indeseable:

```
T1 -> counter + 1 ----> { counter = 0 }
T2 -> counter + 1 ----> { counter = 0 }
T1 -> counter = 1 ----> { counter = 1 }
T1 -> print(counter) -> { counter = 1 }
```

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo de traza indeseable:

```
T1 -> counter + 1 ----> { counter = 0 }
T2 -> counter + 1 ----> { counter = 0 }
T1 -> counter = 1 ----> { counter = 1 }
T1 -> print(counter) -> { counter = 1 }
T2 -> counter = 1 ----> { counter = 1 }
```

Ejemplo

N threads suman 1 a un contador compartido e imprimen el valor:

```
global int counter = 0;

repeat (N)
  thread {
    counter = counter + 1;
    print(counter);
  }
```

Ejemplo de traza indeseable:

```
T1 -> counter + 1 ----> { counter = 0 }
T2 -> counter + 1 ----> { counter = 0 }
T1 -> counter = 1 ----> { counter = 1 }
T1 -> print(counter) -> { counter = 1 }
T2 -> counter = 1 ----> { counter = 1 }
T2 -> print(counter) -> { counter = 1 }  (Pérdida de sumas)
```

Definiciones

Sección crítica

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

Exclusión mutua

Llamamos exclusión mutua 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*.

Requerimientos de la exclusión mutua

Una solución al problema de exclusión mutua debe satisfacer los siguientes requerimientos:

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.

Esquema general

```
global variable compartida;

thread T:
    seccion no critica  // no usa la variable compartida
    entrada a la seccion critica
    SECCION CRITICA     // siempre termina
    salida de la seccion critica
    seccion no critica  // no usa la variable compartida
```

Pregunta

¿Podemos resolver el problema de la exclusión mutua para **dos threads** asumiendo que las únicas operaciones atómicas son la **lectura** y la **escritura** de variables?

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
}                              }
```

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
}                              }
```

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

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
}                              }
```

- **Mutex:** No (traza).
- **Garantía de entrada:**

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
}                              }
```

- **Mutex:** No (traza).
- **Garantía de entrada:** Sí, el flag empieza en false así que el primero siempre puede entrar. Al salir el flag se restablece en false, por lo que el otro va a poder entrar.

Intento II

```
global boolean[] flags = {false, false};

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  otro = (id + 1) % 2;              otro = (id + 1) % 2;
  flags[id] = true;                 flags[id] = true;
  while (flags[otro]);              while (flags[otro]);
  // SECCION CRITICA                // SECCION CRITICA
  flags[id] = false;                flags[id] = false;
  // seccion no critica             // seccion no critica
}                                 }
```

Intento II

```
global boolean[] flags = {false, false};

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  otro = (id + 1) % 2;              otro = (id + 1) % 2;
  flags[id] = true;                 flags[id] = true;
  while (flags[otro]);              while (flags[otro]);
  // SECCION CRITICA                // SECCION CRITICA
  flags[id] = false;                flags[id] = false;
  // seccion no critica             // seccion no critica
}                                 }
```

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

Intento II

```
global boolean[] flags = {false, false};

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  otro = (id + 1) % 2;              otro = (id + 1) % 2;
  flags[id] = true;                 flags[id] = true;
  while (flags[otro]);              while (flags[otro]);
  // SECCION CRITICA                // SECCION CRITICA
  flags[id] = false;                flags[id] = false;
  // seccion no critica             // seccion no critica
}                                 }
```

- **Mutex:** Sí, setea el flag y luego lee el flag del otro.
- **Garantía de entrada:**

Intento II

```
global boolean[] flags = {false, false};

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  otro = (id + 1) % 2;              otro = (id + 1) % 2;
  flags[id] = true;                 flags[id] = true;
  while (flags[otro]);              while (flags[otro]);
  // SECCION CRITICA                // SECCION CRITICA
  flags[id] = false;                flags[id] = false;
  // seccion no critica             // seccion no critica
}                                 }
```

- **Mutex:** Sí, setea el flag y luego lee el flag del otro.
- **Garantía de entrada:** No (traza con deadlock).

Intento III

```
global int turno = 0;

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  while (turno != id);              while (turno != id);
  // SECCION CRITICA                // SECCION CRITICA
  turno = (id + 1) % 2;             turno = (id + 1) % 2;
  // seccion no critica             // seccion no critica
}                                 }
```

Intento III

```
global int turno = 0;

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  while (turno != id);              while (turno != id);
  // SECCION CRITICA                // SECCION CRITICA
  turno = (id + 1) % 2;             turno = (id + 1) % 2;
  // seccion no critica             // seccion no critica
}                                 }
```

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

Intento III

```
global int turno = 0;

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  while (turno != id);              while (turno != id);
  // SECCION CRITICA                // SECCION CRITICA
  turno = (id + 1) % 2;             turno = (id + 1) % 2;
  // seccion no critica             // seccion no critica
}                                 }
```

- **Mutex:** Sí, sólo el de id igual a turno accede.
- **Garantía de entrada:**

Intento III

```
global int turno = 0;

thread {                          thread {
  id = 0;                           id = 1;
  // seccion no critica             // seccion no critica
  while (turno != id);              while (turno != id);
  // SECCION CRITICA                // SECCION CRITICA
  turno = (id + 1) % 2;             turno = (id + 1) % 2;
  // seccion no critica             // seccion no critica
}                                 }
```

- **Mutex:** Sí, sólo el de id igual a turno accede.
- **Garantía de entrada:** No (si el thread 0 no llega, eg. falla).

Algoritmo de Peterson (II + III)

```
global int turno = 0;
global boolean[] flags = {false, false};

thread {
  id = 0:
  // seccion no critica
  otro = (id + 1) % 2;
  flags[id] = true;
  turno = otro;
  while (flags[otro] && turno == otro);
  // SECCION CRITICA
  flags[id] = false;
  // seccion no critica
}
```

Algoritmo de Peterson (II + III)

```
global int turno = 0;
global boolean[] flags = {false, false};

thread {
  id = 0:
  // seccion no critica
  otro = (id + 1) % 2;
  flags[id] = true;
  turno = otro;
  while (flags[otro] && turno == otro);
  // SECCION CRITICA
  flags[id] = false;
  // seccion no critica
}
```

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

Algoritmo de Peterson (II + III)

```
global int turno = 0;
global boolean[] flags = {false, false};

thread {
  id = 0:
  // seccion no critica
  otro = (id + 1) % 2;
  flags[id] = true;
  turno = otro;
  while (flags[otro] && turno == otro);
  // SECCION CRITICA
  flags[id] = false;
  // seccion no critica
}
```

- **Mutex:** Sí, sólo el de id igual a turno accede o el otro flag está en false.
- **Garantía de entrada:**

Algoritmo de Peterson (II + III)

```
global int turno = 0;
global boolean[] flags = {false, false};

thread {
  id = 0:
  // seccion no critica
  otro = (id + 1) % 2;
  flags[id] = true;
  turno = otro;
  while (flags[otro] && turno == otro);
  // SECCION CRITICA
  flags[id] = false;
  // seccion no critica
}
```

- **Mutex:** Sí, sólo el de id igual a turno accede o el otro flag está en false.
- **Garantía de entrada:** Sí, el flag del otro empieza (y se restablece) en false, por lo que el primero que llega puede entrar. Salvo cuando llegan los dos juntos, en cuyo caso entra el de id igual a turno.

Pregunta (ext.)

¿Podemos resolver el problema de la exclusión mutua para **n threads** asumiendo que las únicas operaciones atómicas son la **lectura** y la **escritura** en de variables?

Pregunta (ext.)

¿Podemos resolver el problema de la exclusión mutua para **n** **threads** asumiendo que las únicas operaciones atómicas son la **lectura** y la **escritura** en de variables?
Donde **n** es arbitrariamente grande, pero fijo.

Algoritmo de Bakery

```
global boolean[] pidiendoTicket = new boolean[n]; // {false,..,}
global int[] ticket = new int[n]; // {0,0,...0}

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

  // SECCION CRITICA

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

Bakery - Análisis

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

Bakery - Análisis

- **Mutex:** Sí, sólo el de ticket y id mínimo accede.
- **Garantía de entrada:**

Bakery - Análisis

- **Mutex:** Sí, sólo el de ticket y id mínimo accede.
- **Garantía de entrada:** Sí, pues se accede en orden (ticket, id), por lo que siempre al menos un thread puede entrar. Además, los tickets empiezan y se restablecen en 0, mientras que los flags empiezan y se restablecen en false, por lo que al salir siempre quedará un thread como el "siguiente".

Bakery - Análisis

- **Mutex:** Sí, sólo el de ticket y id mínimo accede.
- **Garantía de entrada:** Sí, pues se accede en orden (ticket, id), por lo que siempre al menos un thread puede entrar. Además, los tickets empiezan y se restablecen en 0, mientras que los flags empiezan y se restablecen en false, por lo que al salir siempre quedará un thread como el "siguiente".

La demostración formal excede el alcance de este curso.