Acciones atómicas

Programación concurrente

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.

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.
- Definimos sección crítica a la parte del programa que accede a memoria compartida.

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.
- Definimos sección crítica a la parte del programa que accede a memoria compartida.
- Vimos algoritmos que garantizan exclusión mutua usando únicamente lectura y escritura a memoria.

Pregunta

¿Cuál es la complejidad del algoritmo de Bakery?

```
global boolean[] entrando = replicate(n,false);
global int[] numero] = replicate(n,0);

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

  // SECCION CRITICA

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

Más preguntas

- ¿Podrá hacerse más eficiente?
- ¿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? ¿Qué complejidad tiene?

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? ¿Qué complejidad tiene?
- ¿Podemos idear una instrucción atómica que nos permita corregirlo? ¿Qué es lo que tendría que hacer?

Test and Set

```
atomic boolean TestAndSet(ref) {
  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) {
  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 Object shared = record();
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
}
```

Exchange

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

Exchange

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

global Object shared = record();
shared.value = 0;

thread {
  local = record();
  local.value = 1;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value == 1);
  // seccion critica
  shared.value = 0;
  // seccion no critica
}
```

```
thread {
  local = record();
  local.value = 1;
  // seccion no critica
  do Exchange(shared, local);
  while (local.value == 1);
  // seccion critica
  shared.value = 0;
  // seccion no critica
}
```

Problema

- Las soluciones anteriores garantizan no garantizan la atención en orden de llegada.

Problema

- Las soluciones anteriores garantizan no garantizan la atención en orden de llegada.
- ¿Podemos idear una instrucción atómica que nos permita garantizar el orden?

Fetch and Add

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

Fetch and Add

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

gobal Object ticket = record();
gobal Object turn   = record();
ticket.value = 0;
turn.value   = 0;

thread {
  int myTurn;
  // seccion no critica
  myTurn = FetchAndAdd(ticket, 1);
  while (turn.value != myTurn.value);
  // SECCION CRITICA
  FetchAndAdd(turn, 1);
  // seccion no critica
}
```

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.