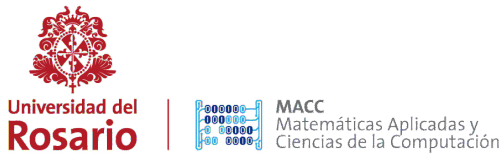
Sincronización

Carlos E. Alvarez[1].

[1]Dep. de Matemáticas aplicadas y Ciencias de la Computación, Universidad del Rosario

2020-I



Algoritmo determinista

Algoritmo determinista: Dada una entrada particular, siempre arroja el mismo resultado.

Algoritmo no determinista: Puede arrojar diferentes resultados con la misma entrada.

Condiciones de carrera

Sucede cuando dos o mas instrucciones paralelas intentan acceder el mismo espacio de memoria al mismo tiempo.

Se evidencia por la presencia de resultados aparentemente aleatorios en un algoritmo determinista.

Contando 3s

Función que cuenta el número de 3s que hay en un vector de enteros aleatorios de longitud n.

```
1  int count3s(const vector<int> &vec) {
2    int count = 0;
3    for(int i = 0; i < vec.size(); ++i){
4      if(vec[i] == 3)
5        count++;
6    }
7    return count;
8  }
```

Ej:

- El prof. proveerá la versión serial del código para generar y calcular el No. de 3s en un vector
- Implemente una versión paralela de `count3s`, haciendo que cada hilo realice la búsqueda en una sección diferente del vector y valla sumando al contador
- Verifique el tiempo de ejecución de la versión paralela y la correctitud del resultado

Resultados:

- Tiempo de ejecución disminuye
- Resultado es incorrecto

Debido a que varios hilos intentan sobre-escribir al mismo la variable `count`, se produce un resultado incorrecto. Esto se denomina *condición de carrera*.

Regiones mutuamente exclusivas (MUTEX)

Para evitar que se genere una condición de carrera sobre la variable `count`, podemos decirle al compilador que cada vez que un hilo esté escribiendo la variable, el espacio de memoria que esta ocupa debe bloquearse para que ningún otro hilo pueda hacerlo hasta que el anterior termine.

Esto lo logramos declarando un objeto `mutex` que nos permita hacer el bloqueo:

```
1   #include <mutex>
2   .
3   .
4   mutex m;
5   .
6     /*critical region*/
7     m.lock();
8     count++
9     m.unlock();
10  .
11  .
```

Ej:

- Cree en un objeto `mutex` en la función `count3s`
- Páselo por referencia a la función que ejecutan los hilos
- Dentro de la función que ejecutan los hilos, utilice el objeto `mutex` para crear una región crítica al rededor del incremento `count++`

¿Es el resultado correcto ahora? ¿Qué ha pasado con el tiempo de ejecución?

La región crítica genera un bloqueo, en el que todos los hilos deben esperar su turno. Mientras más veces se bloquee, mas lento será el código.

Disminuyendo la cantidad de bloqueos.

Ej:

- Cree la variable `priv_count` dentro de la función que ejecutan los hilos. Esto hace que la variable sea privada (no compartida)
- Acumule en esta variable el conteo que hace el hilo dentro del ciclo
- Genere una región crítica (fuera del ciclo) , en donde se acumulen los valores de `priv_count` y se guarden en `count`

¿Que sucede con el tiempo de ejecución?

Operaciones atómicas

Cuando se hace una operación básica (+,-,*,/,comparación) sobre una variable, el compilador puede asegurarse que solo un hilo realice la operación sobre la variable.

A diferencia de un mutex, las operaciones atómicas no bloquean la ejecución de los demás hilos.

Esto se logra declarando el tipo de variable como *atómico*.

```
1  #include <atomic>
2  .
3  .
4    atomic<type> identifier;
5  .
6  .
```

El prof. proveerá un archivo con un ejemplo.

Ej:

- Implemente una versión del código en la que reemplaza el uso del bloqueo `mutex` sobre `count` por el uso de una variable `count` de tipo `atomic<int>`