Multithreading y recursos compartidos

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

Multithreading

```
1  int counter = 0;
2
3  void inc() {
4      ++counter;
5  }
6
7  int main(int argc, char* argv[]) {
8      std::thread t1 {inc};
9      std::thread t2 {inc};
10
11     t1.join(); t2.join();
12     return counter;
13 }
```


- En C++11 podemos ejecutar una función en su propio hilo con **`std::thread`**
- Luego, debemos esperar a que los hilos terminen sincronizando las ejecuciones con **`join`**

Instrucciones no atómicas

```
1  void inc() {
2      ++counter;
3  }
```

```
1  mov eax <-- counter
2  add eax <-- 1
3  mov counter <-- eax
```


- En general las instrucciones no son intrucciones atómicas. La intrucción más simple en C++ **`++counter`** requiere la ejecución de tres instrucciones del microprocesador.
- Por supuesto, los detalles de que instrucciones y cuantas son ejecutadas por el microprocesador dependen de la arquitectura de este.

Acceso concurrente: caso felíz

Thread 1

```
1  mov eax <-- counter
2  add eax <-- 1
3  mov counter <-- eax
   ---
   ---
   ---
```

Thread 2

```
   ---
   ---
   ---
4  mov eax <-- counter
5  add eax <-- 1
6  mov counter <-- eax
```

Registros del Thread 1
eax = 0 eax = 1

Registros del Thread 2
eax = eax = 1 eax = 2

Data segment (Threads 1 y 2)
counter = 0 counter = 1 counter = 2

- Los hilos comparten el heap y el data segment.
- Por cada hilo tiene su propio stack y su propio set de registros.
- En el caso felíz las instrucciones que mutan una variable compartida no se interfieren entre sí y el resultado final es el esperado: `counter == 2`

Acceso concurrente: race condition

Thread 1

```
1 mov eax <-- counter
2 add eax <-- 1
  ---
  ---
  ---
  mov counter <-- eax
```

Thread 2

```
  ---
  ---
3 mov eax <-- counter
4 add eax <-- 1
5 mov counter <-- eax
  ---
```

Registros del Thread 1
eax = 0 eax = 1

Registros del Thread 2
eax = eax = 0 eax = 1

Data segment (Threads 1 y 2)
counter = 0 counter = 1

- Pero el Sistema Operativo puede decidir detener un hilo arbitrariamente y comenzar a ejecutar otro.
- Cuando se accede a un recurso compartido, mutable, para operaciones de lectura y escritura, se abre la posibilidad de que la modificación se realize de forma incompleta.

Acceso concurrente: race condition

Thread 1

```
1 mov eax <-- counter
2 add eax <-- 1
  ---
  ---
  ---
6 mov counter <-- eax
```

Thread 2

```
  ---
  ---
3 mov eax <-- counter
4 add eax <-- 1
5 mov counter <-- eax
  ---
```

Registros del Thread 1
eax = 1

Registros del Thread 2
eax = 1

Data segment (Threads 1 y 2)
counter = 1

- Una race condition se debe al acceso no-atómico de lecto/escritura de un recurso compartido.
- Si el recurso compartido es inmutable o solamente se accede a él para operaciones de lectura, no existe la posibilidad de tal error.
- En el presente caso el hilo 1 no pudo completar su escritura dando resultados incorrectos. Esto se lo denomina race condition.
- Aunque es comun ver el término race condition en programacion multithreading, el problema pueda darse en otros contextos como en la programación multiprocess o en la programación orientada a eventos (donde los handlers no sean atómicos).

Sincronización: mutual exclusion

```
1 int counter = 0;
2 std::mutex m;
3
4 void inc() {
5     m.lock();
6     ++counter;
7     m.unlock();
8 }
```

- Para evitar la race condition debemos hacer que los hilos se coordinen entre sí para evitar que accedan al objeto compartido a la vez.
- Existen varias estrategias de sincronización: semáforos, colas, sockets (estos últimos también usados para comunicación)
- Un mutex es un objeto que nos permitirá forzar la ejecución de un código de forma exclusiva por un hilo a la vez. En C++ `std::mutex`

Acceso atómico

Thread 1

```
1  siz m
2  mov eax <-- counter
3  add eax <-- 1
   ---
   mov counter <-- eax
   zer m
   ---
```

Thread 2

```
   ---
   ---
   ---
4  siz m
   ---
   ---
   siz m
```

Registros del Thread 1
eax = eax = 0 eax = 1

Registros del Thread 2
eax =

Data segment (Threads 1 y 2)
counter = 0 mutex = 1

- El uso del mutex puede verse como una variable booleana. La instrucción `siz` setea a 1 si la variable es 0 de forma atómica mientras que `zer` setea la variable a 0 de forma atómica.
- Cuando el hilo 2 trata de tomar or adquirir el mutex (en C++ `m.lock()`), la instrucción `siz` falla (el variable no es 0) y el hilo deja de ejecutarse.

Acceso atómico

Thread 1

```
1  siz m
2  mov eax <-- counter
3  add eax <-- 1
   ---
5  mov counter <-- eax
6  zer m
   ---
```

Thread 2

```
   ---
   ---
   ---
4  siz m
   ---
   ---
   siz m
```

Registros del Thread 1
eax = 1

Registros del Thread 2
eax =

Data segment (Threads 1 y 2)
counter = 1 mutex = 0

- Efectivamente el uso de un mutex hace que un solo un hilo a la vez puedan acceder al objeto compartido y modificarlo.
- Por supuesto, una vez accedido al recurso, el hilo debe liberar el mutex, de otro modo el recurso queda inaccesible para el resto de los hilos, potencialmente llevando a una situación de dead lock.

Protección de los recursos: monitor

```
1  class ProtectedCounter {
2    int counter;
3    std::mutex m;
4
5    public:
6    void inc() {
7        m.lock();
8        ++counter;
9        m.unlock;
10   }
11 };
```

```
1  ProtectedCounter counter;
2
3  void inc() {
4      counter.inc();
5  }
```

- El recurso compartido y el mutex se usan en combinación.
- Una buena práctica es la de encapsular tanto el recurso como su mutex en un objeto que sabe como protegerse a si mismo.
- Este patrón es llamado Monitor y es otra forma de sincronización que algunos lenguajes ofrecen directamente.

Proteger es más que usar mutexs

```
1  class ProtectedList {
2    std::list<int> list;
3    std::mutex m;
4
5    public:
6    bool has(int x) {
7        m.lock();
8        bool b = list.has(x);
9        m.unlock;
10       return b;
11   }
12
13   void add(int x) {
14       m.lock();
15       list.add(x);
16       m.unlock;
17   }
18 };
```

```
1  ProtectedList list;
2
3  void add_uniq(int x) {
4      if (not list.has(x)) {
5          list.add(x);
6      }
7  }
```


- Pero usar mutex/monitor no es garantía: no es solo una cuestión de proteger cada método del recuso con un mutex.
- En este caso hay una race condition: un hilo puede preguntar si un elemento está en la list y si no agregarlo. Pero justo antes de llamar a `list.add` otro hilo ejecuta el mismo fragmento de código, preguntando y agregando el mismo valor. Luego, el hilo original termina agregando un valor que ya está en la lista.
- Esto se debe a que las operaciones `list.has` y `list.add` son dependientes y deben ejecutarse de forma atómica. Esto es lo que se conoce como una critical section.

Métodos de un monitor: critical sections

```
1  class ProtectedList {
2    std::list<int> list;
3    std::mutex m;
4
5    public:
6    void add_if_hasnt(int x) {
7        m.lock();
8        if (not list.has(x))
9            list.add(x);
10       m.unlock;
11   }
12 };
```

```
1  ProtectedList list;
2
3  void add_uniq(int x) {
4      list.add_if_hasnt(x);
5  }
```


- Una buena implementación de un monitor no solo encapsula el recurso y el mutex sino que además ofrece un método por cada critical section.
- La intefaz pública de un monitor esta moldeada por las critical sections.

Appendix

Referencias

Referencias I

- Bjarne Stroustrup.
  The C++ Programming Language.
  Addison Wesley, Fourth Edition.
- Andrew S. Tanenbaum.
  Modern Operating System.
  Prentice Hall, Second Edition.
- Mordechai Ben-Ari.
  Principles of Concurrent and Distributed Programming
  Addison Wesley, Second Edition.