Consistencia de memoria en C++

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

1 Modelo de memoria

2 Tipos atómicos

3 Relaciones de ordenación

4 Modelos de consistencia

5 Barreras

6 Conclusión

C++ y consistencia de memoria

- C++11 define un **modelo de concurrencia** propio como parte del propio lenguaje.

- **Objetivo**: Evitar la necesidad de escribir código en lenguajes de más bajo nivel (C, ensamblador, ...) para obtener mayores prestaciones.
  - Tipos atómicos.
  - Mecanismos de sincronización de bajo nivel.

- Permite la construcción de **estructuras de datos libres de cerrojos**.

Objetos y posiciones de memoria

- **Objeto**: Es una región de almacenamiento.
  - Una secuencia de uno o varios bytes.

- **Posición de memoria**: Es un objeto de un tipo escalar o una secuencia de campos de bits adyacentes.

- **Un objeto se almacena en una o varias posiciones de memoria**.

Ejemplo

- Estructura:

```
struct {
  int i;
  char c;
  int d: 10;
  int e: 16;
  double f;
};
```

- Posiciones de memoria:
  1. **i**.
  2. **c**.
  3. **d**, **e**.
  4. **f**.

Reglas

- Dos hilos pueden acceder a **posiciones de memoria distintas** de forma simultánea.
- Dos hilos pueden acceder a la **misma posición de memoria** de forma simultánea si ambos accesos son de **lectura**.
- Si dos hilos intentan acceder de forma simultánea a la **misma posición de memoria** y alguno de los accesos es de **escritura** existe una **condición de carrera potencial**.
  - Depende de si se obliga un **orden entre ambos accesos**.

Ordenamiento y condiciones de carrera

- **Solución clásica**: Uso de mecanismos de **sincronización**.
  - Permite garantizar la **exclusión mutua**.
  - Basado en SO $\rightarrow$ Puede ser costoso.
- **Alternativa**: Uso de **operaciones atómicas** para garantizar **ordenamiento**.
  - Si no se establece el **orden entre dos accesos** a una posición de memoria.
  - alguno de los accesos **no es atómico**,
  - y al menos uno de los accesos es una **escritura**,
  - estos constituyen una **carrera de datos** y el **comportamiento del programa no está definido**.

Orden de modificación

- **Orden de modificación**: Secuencia de escrituras sobre un objeto.
  - Si dos hilos ven distintos ordenes de modificación sobre un objeto hay una **carrera de datos**.
  - Las modificaciones no tienen por qué ser visibles en el mismo instante en todos los hilos.

- Una lectura posterior a una escritura en un mismo hilo observa el valor escrito o un valor posterior en su **orden de modificación**.

1 Modelo de memoria

2 Tipos atómicos

3 Relaciones de ordenación

4 Modelos de consistencia

5 Barreras

6 Conclusión

Operaciones atómicas

- Son **operaciones indivisibles**.
  - Si un hilo realiza una **lectura atómica** de una variable y otro una **escritura atómica** de la misma variable y no hay **más hilos accediendo**:
    - La lectura devuelve el **valor previo** a la escritura o el **valor escrito**.
  - Si alguna de las operaciones (lectura o escritura) es **no atómica** el **comportamiento no está definido**.
    - Se puede obtener un valor que no sea ni el anterior ni el posterior.

Tipos atómicos

- El tipo genérico **atomic<T>** permite definir variables atómicas para el tipo **T**, donde **T** es:
  - Un tipo integral.
  - Un tipo puntero.
  - El tipo **bool**.
  - No está definido para tipos reales (**float**, **double**).
  - También para tipos definidos por el usuario que cumplan con algunas restricciones.
- Todos los tipos atómicos tienen un miembro **is_lock_free()**.
  - Determina si su implementación es libre de cerrojos.
- Además existe un tipo **atomic_flag**:
  - El único que garantiza ser libre de cerrojos.

Operaciones sobre tipos atómicos

- Las operaciones sobre atómicos pueden especificar opcionalmente un ordenamiento de memoria.
  - Por defecto **memory_order_seq_cst**.
- Operaciones de almacenamiento:
  - **memory_order_relaxed**, **memory_order_release**, **memory_order_seq_cst**.
- Operaciones de lectura:
  - **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_seq_cst**
- Operaciones lectura-modificación-escritura:
  - **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel**, **memory_order_seq_cst**.

atomic_flag

- Tipo atómico **más simple posible**.
  - **Dos estados posibles**: **activado** o **desactivado**.
  - Siempre es libre de cerrojos.
  - Siempre hay que iniciarlos explícitamente a desactivado.

```
std::atomic_flag f1 = ATOMIC_FLAG_INIT;
```

  - **Operaciones**:
    - **Desactivar**:

```
f1.clear();
```

    - **Activar y comprobar** valor previo:

```
f1.test_and_set();
```

  - Pueden indicar el orden de memoria de la operación.

Ejemplo: Un *spin lock*

- Cerrojo que no hace uso de servicios del SO.
  - Útil si los bloqueos van a durar muy poco tiempo y se quiere evitar problemas de cambio de contexto.

spin lock mutex

```
class spinlock_mutex {
private:
  std::atomic_flag f;
public:
  spinlock_mutex() : f{ATOMIC_FLAG_INIT} {}

  void lock() {
    while (f.test_and_set()) {}
  }
  void unlock() {
    flag.clear();
  }
};
```

atomic_bool

- Más operaciones que **atomic_flag**.
- Se puede iniciar y asignar con **bool**s.
- No se puede copiar de otro **atomic<bool>**.
- Modificación: **a.store(orden)**
- Consulta: **a.exchange(b, orden)**
- Conversión automática a **bool** (consistencia sec.): **a.load(orden)**

Ejemplo

```
std::atomic<bool> a;
bool x = a.load(std::memory_order_acquire);
a.store(true);
x = a.exchange(false, std::memory_order_acq_rel);
```

Comparación e intercambio

- Compara el valor del atómico con un valor **esperado**.
  - Si son iguales almacena el valor **deseado** en el atómico.
  - Si no son iguales no modifica el atómico.
  - Siempre retorna indicación de éxito fracaso.

- Dos versiones:
  1. **a.compare_exchange_weak(e,d)**:
     - Permite fallos espúreos (cambio de contexto) en algunas arquitecturas.
     - Puede comportarse como si ***this!=e** incluso aunque sean iguales
  2. **a.compare_exchange_strong(e,d)**:
     - No permite fallos espúreos.

atomic_address

- Acceso atómico a una dirección de memoria.
- No se puede copiar.
- Se puede copiar un puntero (**void***).
- Interfaz similar a atomic<bool>:
  - **is_lock_free()**, **load()**, **store()**, **exchange()**, **compare_exchange_weak()**, **compare_exchange_strong()**.
- Operaciones adicionales.
  - **fetch_add()**, **fetch_sub()**.
    - Permiten especificar ordenamiento.
    - Devuelven valor previo al cambio.
- **+=**, **-=**.
  - Devuelven valor posterior al cambio.
  - Todas usan aritmética de byte.
- Otras aritméticas con **atomic<T*>**.

atomic<integral>

- Aplicable a todos los tipos integrales.
- Operaciones generales:
  - **is_lock_free()**, **load()**, **store()**, **exchange()**, **compare_exchange_weak()**, **compare_exchange_strong()**.
- Operaciones aritméticas:
  - **fetch_add()**, **fetch_sub()**, **fetch_and()**, **fetch_or()**, **fetch_xor()**.
  - **+=**, **-=**, **&=**, **|=**, **^=**.
  - **++x**, **x++**, **--x**, **x--**
  - No hay otras operaciones aritméticas (*****, **/**, **%**).

1 Modelo de memoria

2 Tipos atómicos

3 Relaciones de ordenación

4 Modelos de consistencia

5 Barreras

6 Conclusión

Relación **sincroniza-con**

- **Relación** entre operaciones sobre **tipos atómicos**.
- Una **escritura** sobre un valor atómico **sincroniza-con** una **lectura** sobre ese valor atómico **que lee el valor**:
  1. Almacenado por **esa escritura**.
  2. Almacenado por una **escritura subsiguiente** del mismo hilo que realizó la escritura.
  3. Almacenado por una **secuencia** de operaciones **read-modify-write** sobre el valor de cualquier hilo en la que la primera operación leyó el valor almacenado por la escritura.

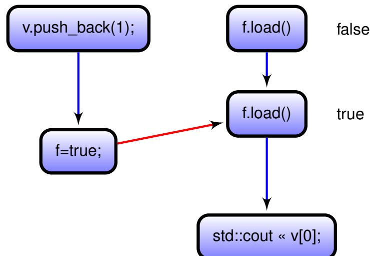
Relación **ocurre-antes**

- Especifica qué operación **ve los efectos** de otra operación.
- Dentro de un hilo, una operación **ocurre-antes** que otra si aparece en una **sentencia anterior**.
  - **No hay orden entre dos operaciones que aparecen en la misma sentencia**.
- Entre dos hilos, una operación en un hilo **ocurre-antes** que otra operación de otro hilo si:
  1. Existe una relación **sincroniza-con** entre ambas.
  2. Existe una cadena de relaciones **ocurre-antes** y **sincroniza-con** entre ellas.

Ordenación: Consistencia secuencial

Ejemplo

```
std::vector<int> v;
std::atomic_bool f(false);

void escritor() {
  v.push_back(1); // #1
  f = true; // #2
}

void lector() {
  while(!f.load()) { // #3
    std::this_thread::sleep(
      std::milliseconds(1));
  }
  std::cout << v[0] << std::endl; // #4
}
```



- Único resultado posible: **v[0] == 1**.

1 Modelo de memoria

2 Tipos atómicos

3 Relaciones de ordenación

4 Modelos de consistencia

5 Barreras

6 Conclusión

Consistencia secuencial

- **memory_order_seq_cst**.
- El programa es consistente con una **vista secuencial**.
- Si todas las operaciones sobre atómicos son **secuencialmente consistentes**, el comportamiento del programa multihilo es como si todas las operaciones se realizasen en algún orden particular en un único hilo.
- No puede haber reordenaciones.
- Es el modelo más simple de razonar.
- Es el modelo más costoso en rendimiento.

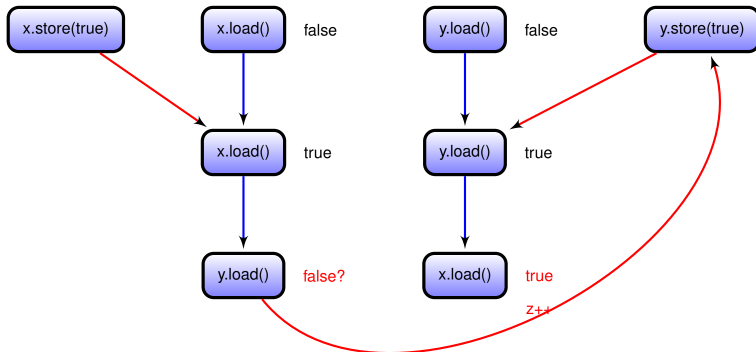
Acceso

```
td::atomic<bool> x, y;
std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_seq_cst);
}

void g() {
  y.store(true, std::memory_order_seq_cst);
}

void h() {
  while (!x.load(std::memory_order_seq_cst)) {}
  if (y.load(std::memory_order_seq_cst)) ++z;
}

void i() {
  while (!y.load(std::memory_order_seq_cst)) {}
  if (x.load(std::memory_order_seq_cst)) ++z;
}
```

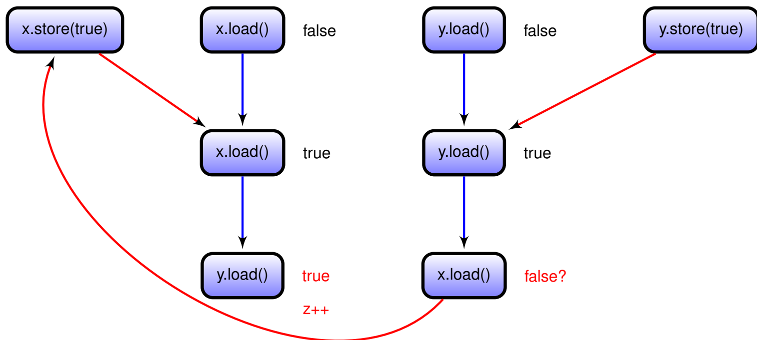
Lanzamiento de hilos

```
int main() {
  x = false;
  y = false;
  z = 0;

  std::thread t1{f};
  std::thread t2{g};
  std::thread t3{h};
  std::thread t4{i};

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  assert(z.load() !=0);

  return 0;
}
```

Consistencia sequencial: Análisis



Consistencia sequencial: Análisis



Ordenes secuencialmente no consistentes

- Deja de haber un **orden global de los eventos**.
  - Cada hilo puede tener **una vista diferente**.
    - Los hilos pueden no estar de acuerdo en el orden de los eventos.
  - Pero, . . .
    - **Todos los hilos deben estar de acuerdo en el orden de modificación de cada variable**.
- **Alternativas**:
  - Ordenamiento **relajado**.
  - Ordenamiento **adquisición liberación**.

Ordenamiento relajado

- **memory_order_relaxed**
- Operaciones relajadas sobre atómicos **no participan** en la relación **sincroniza-con**.

- Operaciones sobre misma variable en mismo hilo **si cumplen** relación **ocurre-antes**.

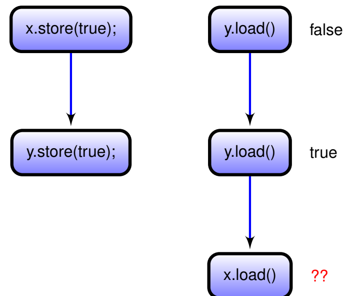
  - **No se pueden reordenar** accesos a una variable atómica dentro de un mismo hilo.

  - Una vez que un hilo ha visto un valor de una variable **no pude ver** un valor más antiguo de esa variable.

Ejemplo

Acceso a datos

```
std::atomic<bool> x, y; std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_relaxed);
  y.store(true, std::memory_order_relaxed);
}
void g() {
  while (!y.load(std::memory_order_relaxed)) {}
  if (x.load(std::memory_order_relaxed)) { ++z; }
}

int main() {
  x=false; y=false; z=0;
  std::thread t1{f}; std::thread t2{g};
  t1.join(); t2.join();
  return 0;
}
```



Orden de adquisición/liberación

- **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel**.
- Nivel **intermedio** de sincronización.
- Una operación de **liberación** que **escribe un valor** se **sincroniza-con** una operación de **adquisición** que **lee dicho valor**.
- **Impacto**:
  - **Distintos hilos pueden ver distintos órdenes**.
  - **No todos los órdenes son posibles**.

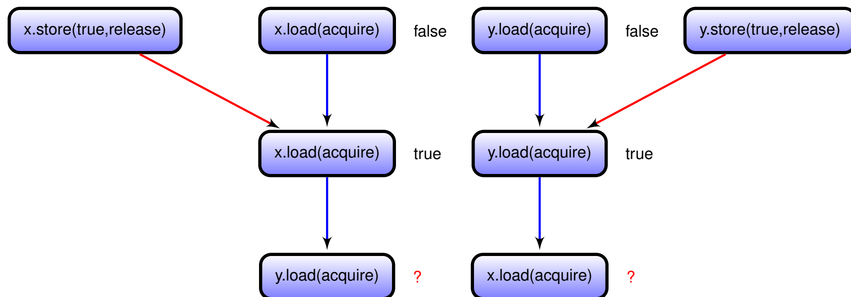
Acceso

```
std::atomic<bool> x, y;
std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_release);
}

void g() {
  y.store(true, std::memory_order_release);
}

void h() {
  while (!x.load(std::memory_order_acquire)) {}
  if (y.load(std::memory_order_acquire)) ++z;
}

void i() {
  while (!y.load(std::memory_order_acquire)) {}
  if (x.load(std::memory_order_acquire)) ++z;
}
```

Lanzamiento de hilos

```
int main() {
  x = false;
  y = false;
  z = 0;

  std::thread t1{f};
  std::thread t2{g};
  std::thread t3{h};
  std::thread t4{i};

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  assert(z.load() !=0);
  return 0;
}
```

Análisis



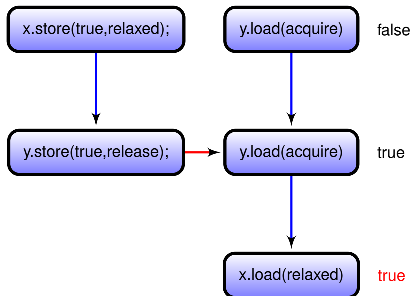
- Múltiples ordenes son posibles porque no hay relaciones *acquire* $\rightarrow$ *release*.

Combinación de órdenes

- Se puede obtener un efecto **equivalente** a **consistencia secuencial** con **menos coste**.

Acceso

```
std::atomic<bool> x, y; std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_relaxed);
  y.store(true, std::memory_order_release);
}
void g() {
  while (!y.load(std::memory_order_acquire)) {}
  if (x.load(std::memory_order_relaxed)) ++z;
}
int main() {
  x = false; y = false; z = 0;
  std::thread t1{f}; std::thread t2{g};
  t1.join(); t2.join();
  assert(z.load() !=0);

  return 0;
}
```



1 Modelo de memoria

2 Tipos atómicos

3 Relaciones de ordenación

4 Modelos de consistencia

5 Barreras

6 Conclusión

Barreras

- **Fuerzan ordenación** sin modificar datos.

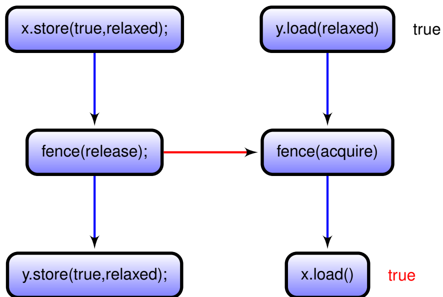
Ejemplo

```
std::atomic<bool> x, y;
std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_relaxed);
  std::atomic_thread_fence(std::memory_order_release);
  y.store(true, std::memory_order_relaxed);
}

void g() {
  while (!y.load(std::memory_order_relaxed)) {}
  std::atomic_thread_fence(std::memory_order_acquire);
  if (x.load(std::memory_order_relaxed)) ++z;
}
```

Hilos

```
int main() {
  x = false;
  y = false;
  z = 0;

  std::thread t1(f);
  std::thread t2(g);

  t1.join();
  t2.join();

  assert(z.load() !=0);
  return 0;
}
```

Barreras: Análisis



1 Modelo de memoria

2 Tipos atómicos

3 Relaciones de ordenación

4 Modelos de consistencia

5 Barreras

6 Conclusión

ARCOS

Resumen

- El modelo de memoria de C++ define las reglas de acceso a memoria de un programa correcto.
  - Permite programación portable de estructuras de datos libres de cerrojos.
- Los tipos atómicos permite realizar operaciones de memoria especificando un ordenamiento.
  - El ordenamiento por defecto es consistencia secuencial.
- Las relaciones *sincroniza-con* y *ocurre-antes* definen restricciones sobre los ordenamientos de operaciones.
- Las barreras permiten forzar ordenamientos sin modificar datos.

ARCOS

Referencias

- *C++ Concurrency in Action. Practical multithreading.*
  Anthony Williams.
  Capítulo 5.

Consistencia de memoria en C++

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid