Arquitectura de Sistemas

Práctica 12: El problema de la consistencia de memoria

Gustavo Romero López

Updated: 30 de mayo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

Objetivos

- Descubrir el problema de la consistencia de memoria.
- Aprender a resolverlo con ayuda de...
  - El **compilador**: evitar la reordenación de instrucciones.
  - El **procesador**: instrucciones de barrera de memoria.
  - El **sistema operativo**: uso del planificador.
  - **Lenguajes de programación** de alto nivel: escogen la solución más adecuada en cada caso.
- Se proporciona un programa que muestra de forma evidente el problema.
- Se pide al estudiante modificarlo de manera que se resuelva el problema empleando los diferentes mecanismos propuestos.

mensaje.cc I

mensaje.cc

```
int listo = 0;
int mensaje[N];

void f(size_t i)
{
    mensaje[i % N] = 42; // mensaje
    listo = 1;           // aviso
}
```

- Imagine que dos hebras se comunican de esta forma:
  - Una hebra escribe un mensaje y avisa mediante `listo = 1`.
  - La otra hebra espera `listo == 1` para recuperar el mensaje.

mensaje.cc II

- Compruebe cuál es la causa del problema mediante un desensamblador: `cutter`, `gdb`, `objdump`...
- Estudie por qué ocurre el problema en `mensaje.cc`.
- Estudie si añadir **`volatile`** lo arregla (`mensaje2.cc`).

```
volatile int listo = 0;
```

- ¿Añadir dos **`volatile`** lo arreglará mejor (`mensaje3.cc`)?

```
volatile int listo = 0;
volatile int mensaje[N];
```

ipc.cc |

```
void productor()
{
    for (size_t i = 0; i < N; ++i)
    {
        mensaje[i % N] = 0x1234;
        listo = 1;
        while (listo == 1);
    }
}

void consumidor()
{
    for (size_t i = 0; i < N; ++i)
    {
        while (listo == 0);
        std::cout << std::hex << mensaje[i] << '\n';
        listo = 0;
    }
}
```

ipc.cc II

- `ipc.cc` es una versión multihebra de `mensaje.cc`... ¿Funciona? ¿Por qué?
- `ipc-{0,1,2,3,g,s,z}` son versiones de `ipc.cc` compiladas con diferentes grados de optimización. Compare las implementaciones de los métodos `productor()` y `consumidor()` y podrá descubrir la respuesta.
- `makefile` contiene el objetivo `att` para ayudarle a desensamblar los binarios.
- Tras observar las diferentes versiones del código... ¿Quién ha metido la pata: yo, tú, el compilador, la cpu,...?
- ¿Sabe cómo arreglarlo?

reorder.cc I

```
int x = 0, y = 0;

void hebra1()
{
    ...
    x = 1;
    r1 = y;
    ...
}
```

```
void hebra2()
{
    ...
    y = 1;
    r2 = x;
    ...
}
```

- ¿Es posible que `r1 == 0 && r2 == 0`? $\Longrightarrow$ `reorder.cc`

reorder.cc II

Culpables del desastre: el compilador y/o el procesador.

- **Copie** el programa reorder.cc.
- Estudie el código y ejecútelo para comprobar que existe un problema de consistencia de memoria.

reorder.cc III

```
void t1()
{
    while(run)
    {
        m1.lock();
        while (rng() != 0);
        x = 1;
        r1 = y;
        m3.unlock();
    }
}
```

```
void t2()
{
    while(run)
    {
        m2.lock();
        while (rng() != 0);
        y = 1;
        r2 = x;
        m4.unlock();
    }
}
```

Primera no solución: `volatile` (la mala)

- **Copie** el programa `reorder.cc` y renómbrelo como `reorder-volatile.cc`
- ¿Cree que añadir **`volatile`** resolverá el problema?
  - cambie: `int x, y, r1, r2;`
  - por: `volatile int x, y, r1, r2;`
- Estudie el código ensamblador generado para averiguar qué es lo que hace el GCC diferente ahora para evitar el problema.
- Recuerde que en realidad **`volatile`** lo único que hace es evitar ciertas optimizaciones en el acceso a memoria.
- **`volatile`** se diseñó originalmente para acceder a dispositivos de E/S mapeados en memoria y cuyos valores no debían ser almacenados en caché por razones obvias.

Segunda no solución: el compilador (otra mala)

- **Copie** el programa `reorder.cc` y renómbrelo como `reorder-gcc.cc`
- La forma de decirle al compilador GCC que no reordene los accesos a memoria es:

```
asm volatile("":::"memory");
```

- ¿Se resuelve el problema?
- Estudie el código ensamblador generado para averiguar qué es lo que hace el GCC diferente ahora para evitar el problema.

Tercera solución: compilador y procesador

- **Copie** el programa `reorder.cc` y renómbrelo como `reorder-cpu.cc`
- Los procesadores de la familia 80x86 tienen diversas instrucciones que crean una barrera en la memoria y evitan que pueda reordenar los accesos a memoria. En este caso utilizaremos `mfence`:

  ```
  asm volatile("mfence":::"memory");
  ```

- ¿Se resuelve el problema?
- Estudie el código ensamblador generado para averiguar qué es lo que hace el GCC diferente ahora para evitar el problema.
- ¿Qué puede pasar si sustituye `mfence` por `lfence` o `sfence`?

Cuarta solución: el planificador del sistema operativo

- **Copie** el programa `reorder.cc` y renómbrelo como `reorder-sched.cc`
- Ejecutar todas las hebras sobre el mismo procesador evita que el procesador pueda reordenar los accesos a memoria.
- Esta solución solo funcionará si el compilador previamente no ha reordenado los accesos a memoria.
- Solución: `asm volatile ("":::"memory");`.

Como escoger la CPU 0 para ejecutar una hebra

```
#include <sched.h>
cpu_set_t cpus;
CPU_ZERO(&cpus);
CPU_SET(0, &cpus);
pthread_setaffinity_np(hebra, sizeof(cpu_set_t), &cpus);
```

Mejor solución: el lenguaje de programación I

- Como este problema es tan común lo mejor es dejar que el lenguaje se encargue de resolverlo utilizando el método más eficaz en cada caso.
- En C++11 podemos utilizar la clase `atomic`:

```
#include <atomic>
std::atomic<int> x, y;
```

- **Copie** el programa `reorder.cc` y renómbrelo como `reorder-atomic.cc` y modifique la declaración de las variables `x` e `y` para que sean de tipo `std::atomic<int>`.
- ¿Cómo ha cambiado el ensamblador?
- Incluso así podemos fastidiarla... vea `reorder-atomic-x.cc`

Mejor solución: el lenguaje de programación II

- Estudie como se ordenan los accesos a memoria: `https://en.cppreference.com/w/cpp/atomic/memory_order`.
- `std::memory_order`:
  - `memory_order_relaxed`: ninguna restricción de ordenación.
  - `memory_order_acquire`: impide que las instrucciones que siguen a la operación de lectura se ejecuten antes de la lectura.
  - `memory_order_release`: impide que las instrucciones que siguen a la operación de escritura se ejecuten antes de la escritura.
  - `memory_order_acq_rel`: combinación de `memory_order_acquire` y `memory_order_release`.
  - `memory_order_seq_cst`: equivalente a `memory_order_acq_rel` pero con una restricción adicional de ordenación global.