Arquitectura de Sistemas

Práctica 12: Consistencia de memoria

Gustavo Romero López

Actualizado: 30 de mayo de 2018

Arquitectura y Tecnología de Computadores

Objetivos

- Descubrir el problema de la consistencia de memoria.
- Aprender a resolverlo con ayuda de...
  - El **compilador**: formas de evitar la reordenación.
  - 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 enunciados.

mensaje.cc I

mensaje.cc

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

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

- 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.
- El optimizador puede reordenar ambas escrituras para solapar el acceso a `listo` con el cálculo de `(i % N)`.

mensaje.cc II

- Utilice un desensamblador para comprobarlo: `objdump`, gdb...
- 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 I

ipc.cc

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

ipc.cc

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

- ipc.cc es una versión multihebra de mensaje.cc... ¿funciona?

ipc.cc II

- `ipc-{0,1,2,3,g,s}` son versiones de `ipc.cc` compiladas con diferentes grados de optimización. Compare los métodos `productor()` y `consumidor()` y seguramente descubra tanto la respuesta como el porque.
- `makefile` contiene el objetivo `att` para ayudarle a desensamblar los binarios.
- Tras observar las diferentes versiones del código... ¿Quién ha metido la pata?

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 solución: el compilador

- **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 una variable.
- `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 solución: el compilador

- **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: el 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 la ejecución de instrucciones. 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`
- Otra solución es hacer que todas las hebras se ejecuten sobre el mismo procesador y como efecto colateral los accesos a memoria no podrán reordenarse mal.

```
#include <sched.h>

cpu_set_t cpus;
CPU_ZERO(&cpus);
CPU_SET(0, &cpus);
pthread_setaffinity_np(hebra, sizeof(cpu_set_t), &cpus);
```

- ¿Cómo ha cambiado el código ensamblador para evitar el problema?
- Esta solución sólo funciona si el compilador no reordena los accesos a memoria.

Mejor solución: *...deja al maestro aunque sea un burro...*

- 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 código ensamblador para resolver el problema?