Introducción a Concurrencia II. Interbloqueo e Inanición

Juan Quintela. Javier París.
{quintela, javier.paris}@udc.es

Interbloqueo (Deadlock)

Interbloqueo es una situación en que dos o más procesos están esperando por recursos que tiene ocupado el otro.

Ejemplo con una función que devuelve la suma de dos variables protegidas:

```
int protected_add(int *v1, mutex *m1, int *v2, mutex *m2) {
    int x;
    lock(m1);
    lock(m2);
    x=v1+v2;
    unlock(m2);
    unlock(m1);
    return x;
}
```

Interbloqueo

- Desde dos threads distintos se llama a la función con las mismas variables pero en orden cambiado:

  int valor1, valor2; // Dos variables compartidas

  mutex m_valor1, m_valor2; // Mutex para proteger las variables valor1 y valor2 respectivamente

- thread1:

  protected_add(&valor1, &m_valor1, &valor2, &m_valor2);

- thread2:

  protected_add(&valor2, &m_valor2, &valor1, &m_valor1);

Interbloqueo

Thread 1:

```
int protected_add(
int *v1 (=valor1),
mutex *m1(=m_valor1),
int *v2(=valor2),
mutex *m2 (=m_valor2))
{
    int x;
    lock(m1);
    lock(m2);
    x=v1+v2;
    unlock(m2);
    unlock(m1);
    return x;
}
```

Thread 2:

```
int protected_add(
int *v1 (=valor2),
mutex *m1(=m_valor2),
int *v2(=valor1),
mutex *m2 (=m_valor1))
{
    int x;
    lock(m1);
    lock(m2);
    x=v1+v2;
    unlock(m2);
    unlock(m1);
    return x;
}
```

Interbloqueo

Cambiamos los nombres de las variables en el código para ver más clara la ejecución.

Thread 1:

```
int protected_add(
int *v1 (=valor1),
mutex *m1(=m_valor1),
int *v2(=valor2),
mutex *m2 (=m_valor2))
{
    int x;
    lock(m_valor1);
    lock(m_valor2);
    x=valor1+valor2;
    unlock(m_valor2);
    unlock(m_valor1);
    return x;
}
```

Thread 2:

```
int protected_add(
int *v1 (=valor2),
mutex *m1(=m_valor2),
int *v2(=valor1),
mutex *m2 (=m_valor1))
{
    int x;
    lock(m_valor2);
    lock(m_valor1);
    x=valor2+valor1;
    unlock(m_valor1);
    unlock(m_valor2);
    return x;
}
```

Interbloqueo

El thread 1 bloquea m_valor1, y el thread 2 bloquea m_valor2

Thread 1:

```
int protected_add(
int *v1 (=valor1),
mutex *m1(=m_valor1),
int *v2(=valor2),
mutex *m2 (=m_valor2))
{
    int x;
    lock(m_valor1);
    lock(m_valor2);
    x=valor1+valor2;
    unlock(m_valor2);
    unlock(m_valor1);
    return x;
}
```

Thread 2:

```
int protected_add(
int *v1 (=valor2),
mutex *m1(=m_valor2),
int *v2(=valor1),
mutex *m2 (=m_valor1))
{
    int x;
    lock(m_valor2);
    lock(m_valor1);
    x=valor2+valor1;
    unlock(m_valor1);
    unlock(m_valor2);
    return x;
}
```

Interbloqueo

El thread 2 intenta bloquear m_valor2, pero está bloqueado por el thread 2,
y el thread 2 intenta bloquear m_valor1, pero está bloqueado por el thread 1.
=> El thread 1 y el thread 2 están en interbloqueo, porque cada uno espera por el otro.

Thread 1:

```
int protected_add(
int *v1 (=valor1),
mutex *m1(=m_valor1),
int *v2(=valor2),
mutex *m2 (=m_valor2))
{
    int x;
    lock(m_valor1);
    lock(m_valor2); // Espera eterna
    x=valor1+valor2;
    unlock(m_valor2);
    unlock(m_valor1);
    return x;
}
```

Thread 2:

```
int protected_add(
int *v1 (=valor2),
mutex *m1(=m_valor2),
int *v2(=valor1),
mutex *m2 (=m_valor1))
{
    int x;
    lock(m_valor2);
    lock(m_valor1); // Espera eterna
    x=valor2+valor1;
    unlock(m_valor1);
    unlock(m_valor2);
    return x;
}
```

Interbloqueo

- Condiciones necesarias para que exista:
  - Exclusión mutua: existen recursos no compartibles.
  - Hold and wait: se permite que un proceso tenga recursos reservados mientras espera para reservar otros.
  - No apropiación: El sistema operativo no puede liberar recursos reservados.
  - Espera circular: Los procesos esperan por recursos reservados por procesos que esperan por los primeros.
- Negar cualquiera de estas condiciones lo evita.

Interbloqueo: Tratamiento.

- El interbloqueo se puede tratar mediante varias técnicas, algunas aplicadas por el sistema operativo, otras por el propio sistema concurrente.
- Algunos sistemas operativos (como Unix o Windows) ignoran el problema. Es tarea del programador de espacio de usuario evitar que se produzca.

Interbloqueo: Detección

- En detección se permite que ocurran interbloqueos, y se corrigen después.
- Como el sistema operativo conoce que recursos tiene reservados un proceso, y por que recursos espera puede saber cuando se produce interbloqueo.
- Una vez detectado puede
  - Matar a los procesos involucrados.
  - Apropiar recursos reservados y dárselos a otro proceso para resolver la situación.

Interbloqueo: Prevención

- En prevención se trata de evitar que una de las 4 condiciones necesarias se produzca:
  - Exclusión mutua, impidiendo que los procesos tengan acceso exclusivo a los recursos.
  - Evitar hold and wait:
    - Haciendo que los procesos reserven todos los recursos necesarios de una vez en vez de mantener reservados unos mientras esperan.
    - Haciendo que los procesos liberen los recursos reservados si no pueden reservar más.
  - Evitar la no apropiación. Es necesario implementar algún mecanismo de roll-back.
  - La espera circular se puede evitar con una reserva ordenada de recursos.

Interbloqueo: Prevención evitando hold and wait

Ejemplo: evitar mantener recursos reservados.

```
int protected_add(int *v1, mutex *m1, int *v2, mutex *m2) {
    int x;
    int success=0;
    do {
        lock(m1);
        if(try_lock(m2)) {  // Si no reserva m2 libera m1 y vuelve a probar
            x=v1+v2;
            unlock(m2);
            success=1;
        }
        unlock(m1);
    } while(!success);
    return x;
}
```

Interbloqueo: Prevención con Reserva Ordenada

- Ejemplo reserva ordenada:

```
int protected_add(int *v1, mutex *m1, int ordenv1,
int *v2, int ordenv2, mutex *m2)
{
    int x;
    if(ordenv1 < ordenv2) {
        lock(m1); lock(m2);
    } else {
        lock(m2); lock(m1);
    }
    x=v1+v2;
    unlock(m2);
    unlock(m1);
}
```

Introducción: Prevención con Reserva Ordenada

- Para llamarlo definimos un orden en los recursos:

  ```
  #define ORDENV1 1
  #define ORDENV2 2
  int valor1, valor2; // Dos variables compartidas
  mutex m_valor1, m_valor2; // Mutex para proteger valor1 y valor2
  ```

- thread1:

  ```
  protected_add(&valor1, &m_valor1, ORDENV1,
                &valor2, &m_valor2, ORDENV2);
  ```

- thread2:

  ```
  protected_add(&valor2, &m_valor2, ORDENV2,
                &valor1, &m_valor1, ORDENV1);
  ```

Interbloqueo: Evitación

- En evitación se intenta prevenir que el sistema entre en una situación de interbloqueo conociendo el estado del sistema y los recursos que los procesos pueden reservar en el futuro.
- Requiere conocer a priori el uso de recursos que va a hacer un proceso, lo que limita mucho su uso.

Inanición

- Un proceso puede estar esperando acceso a un recurso compartido sin conseguirlo. Esta situación se denomina inanición.
- Por ejemplo, si requerimos la reserva de todos los recursos simultáneamente, los procesos que requieran una gran cantidad de recursos pueden quedar en este estado si hay muchos procesos que requieren pocos compitiendo por ellos.

Inanición

Ejemplo: Un thread que necesita bloquear muchos recursos.

```
mutex *m1, *m2, *m3;
void *bloqueo_mucho(void *arg) {
    while(1) {
        lock(m1);
        if (try_lock(m2)) {
            if(try_lock(m3)) {
                // Hacer algo con los recursos protegidos por m1, m2 y m3
            }
        unlock(m2);
        }
        unlock(m1);
    }
}
```

Se usa trylock porque si no se consigue bloquear m2 o m3 queremos minimizar el tiempo que se mantienen mutex bloqueados => implica que el thread se tiene que encontrar los 3 mutex libres para tener éxito.

Inanición

Otros threads solo necesitan un mutex:

```
void *bloqueam1(void *arg) {
    lock(m1);
    // Hacer algo con el recurso protegido por m1
    unlock(m1);
}
void *bloqueam2(void *arg) {
    lock(m2);
    // Hacer algo con el recurso protegido por m2
    unlock(m2);
}
void bloqueam3(void *arg) {
    lock(m3);
    // Hacer algo con el recurso protegido por m3
    unlock(m3);
}
```

Inanición

- ¿Que ocurre si hay muchos threads ejecutando bloquem1, bloqueam2 y bloqueam3? => Es muy probable que en todo momento alguno de los mutex esté bloqueado.
- El thread que ejecuta bloqueamucho no es capaz de entrar en su sección crítica => inanición.