Auxiliar 8: SpinLock

José Astorga

SpinLock

Los spin-lock son una herramienta de sincronización primitiva: no dependen del sistema operativo o un scheduler de procesos.

Usos:

- Exclusión mutua (Mutex)
- Esperar (Condición)

Creación:

int lk = OPEN;

Inicio sección crítica:

spinLock(&lk);

Fin sección crítica:

spinUnlock(&lk);

Spin-locks: Para garantizar exclusión mutua (mutex)

```
int sl = OPEN; //variable global, inicialmente abierto

//Algún core:

...

spinLock(&sl); //entrada a sección crítica

...

spinUnlock(&sl); //salida de sección crítica
```

Spin-locks: Para esperar (condición)

```
int *m; //variable global

//Core 1:

int w = CLOSED; //crea spinLock cerrado

m = &w;

spinLock(&w); //espera, tratando de cerrar un spinLock cerrado (wait)

...

//Core 2:

spinUnlock(m);//abre spinLock para que Core 1 se desbloquee (signal)
```

SpinLock: Implementación

```
void spinLock(volatile int *psl) {
  do {
    while (*psl==CLOSED)
      ;
  } while (swap(psl, CLOSED)!=OPEN);
}
```

P1. Lector/escritor usando spin-lock

La siguiente implementación es incorrecta. Haga un diagrama de threads que muestre que un lector puede entrar junto con un escritor.

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}

void enterWrite() {
    spinLock(&write_lck);
}
```

```
void exitRead() {
    spinLock(&mutex_lck);
    readers--;
    spinUnlock(&mutex_lck);

    if (readers == 0)
        spinUnlock(&write_lck);
}

void exitWrite() {
    spinUnlock(&write_lck);
}
```

T1

T2

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}
```

T1

```
enterRead
    spinLock(&write_lck) //readers == 0
    spinLock(&mutex_lck)
    readers++;
    spinUnlock(&mutex_lck)
```

T2

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}
```

T1

```
enterRead
    spinLock(&write_lck) //readers == 0
    spinLock(&mutex_lck)
    readers++;
    spinUnlock(&mutex_lck)
```

T2

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}
```

```
enterRead
    // readers != 0
```

T1

```
enterRead
    spinLock(&write_lck) //readers == 0
    spinLock(&mutex_lck)
    readers++;
    spinUnlock(&mutex_lck)

exitRead
    spinLock(&mutex_lck)
    readers--;
    spinUnlock(&mutex_lck)
    spinUnlock(&write_lck) //readers == 0
```

T2

```
enterRead
    // readers != 0
```

```
void exitRead() {
    spinLock(&mutex_lck);
    readers--;
    spinUnlock(&mutex_lck);

    if (readers == 0)
        spinUnlock(&write_lck);
}
```

T1

```
enterRead
    spinLock(&write_lck) //readers == 0
    spinLock(&mutex_lck)
    readers++;
    spinUnlock(&mutex_lck)

exitRead
    spinLock(&mutex_lck)
    readers--;
    spinUnlock(&mutex_lck)
    spinUnlock(&write_lck) //readers == 0
```

T2

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}
```

```
enterRead
    // readers != 0
    spinLock(&mutex_lck);
    .
    .
    readers++;
    spinUnlock(&mutex_lck);
```

T1

```
enterRead
    spinLock(&write_lck) //readers == 0
    spinLock(&mutex_lck)
    readers++;
    spinUnlock(&mutex_lck)

exitRead
    spinLock(&mutex_lck)
    readers--;
    spinUnlock(&mutex_lck)
    spinUnlock(&write_lck) //readers == 0

enterWriter
    spinLock(&write_lck);
```

T2

```
void enterWrite() {
    spinLock(&write_lck);
}
```

```
enterRead
    // readers != 0
    spinLock(&mutex_lck);
    .
    .
    readers++;
    spinUnlock(&mutex_lck);
```

P1. Lector/escritor usando spin-lock

Modifique la solución anterior para que funcione correctamente.

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}

void enterWrite() {
    spinLock(&write_lck);
}
```

```
void exitRead() {
    spinLock(&mutex_lck);
    readers--;
    spinUnlock(&mutex_lck);

    if (readers == 0)
        spinUnlock(&write_lck);
}

void exitWrite() {
    spinUnlock(&write_lck);
}
```

P1. Lector/escritor usando spin-lock

Modifique la solución anterior para que funcione correctamente.

```
void enterRead() {
    if (readers == 0)
        spinLock(&write_lck);

    spinLock(&mutex_lck);
    readers++;
    spinUnlock(&mutex_lck);
}

void enterWrite() {
    spinLock(&write_lck);
}
```

```
void exitRead() {
    spinLock(&mutex_lck);
    readers--;
    spinUnlock(&mutex_lck);

    if (readers == 0)
        spinUnlock(&write_lck);
}

void exitWrite() {
    spinUnlock(&write_lck);
}
```

P2. Función team

Considere una máquina multicore en la que no existe un núcleo de sistema operativo y por lo tanto no hay un scheduler de procesos.

Se necesita formar múltiples equipos de 5 cores cada uno. Para ello, los cores invocan la función `team` indicando su nombre como argumento. Esta función espera hasta que 5 cores hayan invocado `team`, retornando un arreglo de 5 strings con los nombres del equipo completo.

Este es un ejemplo del uso de la función team:

```
int player(char* name) {
    for(;;) {
        char** mTeam = team(name);
        play(mTeam);
        sleep();
    }
}
```

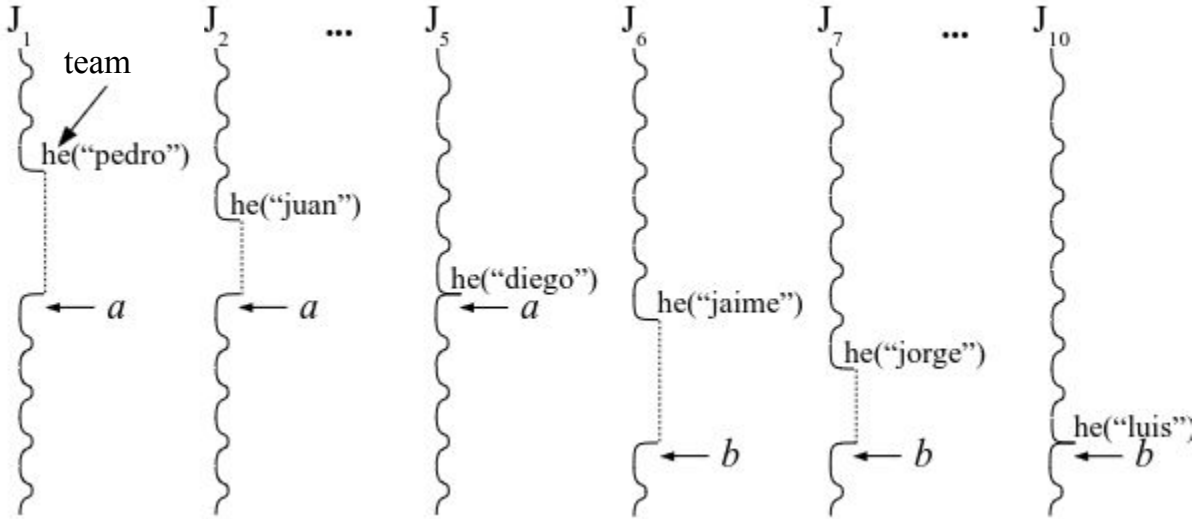
P2. Función team

Se debe programar la función `team` cuyo encabezado es: `char** team(char *name)`

Se dispone de `spin-locks` y la función `coreId()`. Necesitará usar variables globales y `malloc`.

Restricción: Dado que no hay un núcleo de sistema operativo, la única forma válida de esperar a que se forme el equipo es utilizando un spin-lock. Otras formas de busy-waiting no están permitidas. No hay fifoqueues (¡pero sí malloc!).

P2. Función Team



Extra: Control 2 - Otoño 2017

Considere una máquina con 8 cores físicos que comparten la memoria, sin un núcleo de sistema operativo y por lo tanto no hay scheduler de procesos. A cada core se le asignan inicialmente 100 euros. Un core puede robar cantidad euros del core número "*desde*" invocando la función:

```
void robar(int desde, int cantidad);
```

En tal caso se le resta "*cantidad*" euros a la tarea "*desde*" y se le suman al core que invocó robar. El parámetro "*cantidad*" es siempre mayor que cero.

Un primer invariante es que un core no puede almacenar una cantidad negativa de euros. Si el core *desde* no posee suficiente dinero para robarle *cantidad* entonces robar debe esperar hasta que el core *desde* sí posea al menos la cantidad requerida. El segundo invariante es que en un instante dado el core T no puede estar esperando robarle c euros al core U si U tiene al menos c euros.

Observe que cuando el core T lograr robar dinero, varios otros cores pueden estar esperando poder robarle dinero a T. No está especificado en qué orden deben robarle el dinero a T.

Ayuda: Use una matriz m de 8 por 8 punteros a spin-locks. Si m[i][j] no es nulo quiere decir que m[i][j] es la dirección de un spin-lock en el que el core i espera para robarle al core j.

Extra: Control 2 - Otoño 2017