Apellidos: *Nombre:*

Concurrencia y Paralelismo

Grado en Ingeniería Informática

Julio 2019

1. *Productores/Consumidores [2 puntos]*

   Dado el siguiente código de productores/consumidores:

```
typedef struct {
        Queue *q;
        pthread_mutext_t *lock;
        pthread_condition_t *empty;
        pthread_condition_t *full;
} Buffer;

void producer(Buffer *b) {
        while(1) {
                Element e = new_element();
                pthread_mutex_lock(b->lock);
                while (queue_full(b->q)) {
                        pthread_cond_wait(b->full, b->lock);
                }
                queue_insert(b->q, e);
                pthread_cond_broadcast(b->empty);
                pthread_mutex_unlock(b->lock);
        }
}

void consumer(Buffer *b) {
        Element e;

        while(1) {
                pthread_mutex_lock(b->lock);
                while (queue_empty(b->q) {
                        pthread_cond_wait(b->empty, b->lock);
                }
                e = queue_remove(b->q);
                pthread_cond_broadcast(b->full);
                pthread_mutex_unlock(b->lock);
                // Do something with e
        }
}
```

Transfórmese de tal forma que cumpla:

- Use `pthread_cond_signal()` en vez de `pthread_cond_broadcast()`.
- Despierte como máximo a un thread.
- Solo se despertará a un thread en el caso de que haya alguno esperando.
- No se pueden usar variables globales.

Solución

Notese que en el caso de que haya un consumidor esperando, si llega más de un productor, todos despiertan al consumidor hasta que éste se despierte. Esto es necesario porque no hay garantías de que el consumidor que estaba esperando sea el primero que recibe el elemento. Puede aparecer uno nuevo que reciba el elemento.

```
typedef struct {
Queue *q;
        pthread_mutext_t *lock;
        pthread_condition_t *empty;
        pthread_condition_t *full;
        // number of consumers waiting
        int consumers;
        // number of producers waiting
        int producers;
} Buffer;

void producer(Buffer *b) {
while(1) {
         Element e = new_element();
                pthread_mutex_lock(b->lock);
                while (queue_full(b->q)) {
                        b->producers ++;
                 pthread_cond_wait(b->full, b->lock);
                        b->producers --;
                }
                queue_insert(b->q, e);
                if (b->consumers) {
                 pthread_cond_signal(b->empty);
                }
                pthread_mutex_unlock(b->lock);
}
}

void consumer(Buffer *b) {
Element e;

while(1) {
         pthread_mutex_lock(b->lock);
         while (queue_empty(b->q) {
                  b->consumers ++;
               pthread_cond_wait(b->empty, b->lock);
                  b->consumers --;
         }
                e = queue_remove(b->q);
                if (b->producers) {
                 pthread_cond_signal(b->full);
                }
                pthread_mutex_unlock(b->lock);
                // Do something with e
}
}
```

---

2. *Mutex recursivos [1.75 puntos]*

   El siguiente código intercambia dos variables en un array `buffer` protegido por un mutex por posición (`b_lock`):

```
pthread_mutex_t *b_lock[buffer_size];
int             buffer[buffer_size];

void *swap(void *ptr) {
    int tmp;

    int i=rand() % buffer_size;
    int j=rand() % buffer_size;

    pthread_mutex_lock(b_lock[MIN(i,j)]);
    pthread_mutex_lock(b_lock[MAX(i,j)]);

    tmp       = buffer[i];
    buffer[i] = buffer[j];
    buffer[j] = tmp;

    pthread_mutex_unlock(b_lock[i]);
    pthread_mutex_unlock(b_lock[j]);
}
```

   Si en una iteración un thread genera el mismo valor para i y j intentará bloquear el mismo mutex 2 veces, y se interbloqueará a sí mismo.

   En un mutex recursivo un mesmo proceso puede bloquear un mutex múltiples veces. El mutex no se desbloquará hasta que se llamé a unlock el mismo número de veces que se llamó a lock. En el ejemplo anterior, si los mutex de `b_lock` fuesen recursivos entonces generar el mismo valor para i y j no bloquearía al thread, que llamaría a lock 2 veces y después a unlock 2 veces sobre el mismo mutex.

   Implemente una librería de mutex recursivos a partir de los mutex estándar de la librería pthread. Para almacenar el thread propietario del mutex puede usar la función `pthread_self()`, que devuelve el id del thread que llamó a la función, y la función `pthread_equal(t1, t2)` para saber si dos identificadores `pthread_t` son iguales.

```
typedef struct _r_mutex {
    int locked;               // number of locks (0 if free)
    pthread_t owner;          // thread that locked the mutex
    pthread_mutex_t *lock;
    pthread_cond_t waiting;   // sleep on this cond_t until the mutex is free
} *r_mutex;

int r_lock(r_mutex r) {
    ...
}

int r_unlock(r_mutex r) {
    ...
}

pthread_t pthread_self(); // returns the id of the calling thread
int       pthread_equal(pthread_t t1, pthread_t t2);  // returns 1 if t1==t2, 0 otherwise
```

Solución

```
typedef struct _r_mutex {
    int locked;               // number of locks (0 if free)
    pthread_t owner;          // thread that locked the mutex
    pthread_mutex_t *lock;
    pthread_cond_t waiting;   // sleep on this cond_t until the mutex is free
} *r_mutex;

int r_lock(r_mutex r) {
    pthread_t self=pthread_self();

    pthread_mutex_lock(r->lock);

    while(locked>0 && !pthread_equal(self, r->owner)) // Esperar si está bloqueado por otro thread
        pthread_cond_wait(r->waiting, r->lock);

    if(r->locked==0) r->owner = self; // Si no está bloqueado, somos el nuevo propietario
    r->locked++;

    pthread_mutex_unlock(r->lock);
    return 0;
}

int r_unlock(r_mutex r) {
    pthread_mutex_lock(r->lock);

    r->locked--;
    if(r->locked == 0)    // Si queda libre despertamos uno de los threads que estén esperando en lock
        pthread_cond_signal(r->waiting);

    pthread_mutex_unlock(r->lock);
}
```

3. *Barrera con salida periódica [1.25 puntos]*

   Implemente, usando paso de mensajes en Erlang, un sistema que dado un tiempo T, proporcione una función `barrier` de tal forma que si N procesos $P_0, \ldots, P_{n-1}$ llaman a la función `barrier` salga un P_i de la función cada T ms. No es necesario respetar el orden de llegada de los procesos.

   Por ejemplo, dados 4 procesos P_0, P_1, P_2, P_3 que llaman respectivamente a `barrier` en $t = 0, 1, 3, 5$, en una ejecución correcta para un periodo $T = 2$ P_0 saldría de `barrier` en $t = 0$, P_1 en $t = 2$, P_2 en $t = 4$, P_3 en $t = 6$.

   El módulo tiene el siguiente API:

   - `start(T)`, donde T son los ms entre procesos. Devuelve el PID del proceso que gestionará la barrera.
   - `barrier(B)`, donde B es el proceso gestor de una barrera.

   Partiendo del siguiente esqueleto, implemente las funciones `loop` y `barrier`. Se proporciona una función `sleep(T)` que espera T ms.

```
-module(barrier).

-export([start/1, barrier/1]).

start(T) ->
    spawn(?MODULE, loop, [T]).

barrier(B) ->
    ...

loop(T) ->
    ...

sleep(T) -> % sleep T ms
    receive
    after T -> ok
    end.
```

```
-module(barrier).

-export([start/1, barrier/1]).

start(T) ->
    spawn(?MODULE, loop, [T]).

%   Los procesos P_i envían un mensaje a la barrera, y esperan a la respuesta. El servidor
% contentará a un proceso cada T ms.
barrier(B) ->
    B ! {barrier, self()},
    receive
        go -> ok
    end.

loop(T) ->
    receive
        {barrier, From} ->  % Cuando se recibe un mensaje de un proceso
            From ! go,      %  se contesta,
            sleep(T),       %  se esperan T ms para separar las salidas de los procesos de barrier
            loop(T)         %  y se pasa a atender la siguiente petición
    end.

sleep(T) -> % sleep T ms
    receive
    after T -> ok
    end.
```