Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Mayo 2014

1. **Anillo de Threads** [2 puntos]

   Tenemos un sistema formado por N threads, numerados de 0 a N-1, que se comunican usando una estructura en forma de anillo. Cada thread tiene una cola asociada donde recibe elementos del thread anterior, los procesa, y los inserta en la cola del siguiente. Si alguno de los threads no tiene elementos para procesar duerme. El sistema implementado es el siguiente:

```
#define N ... // Numero de threads en el anillo
#define PREV(i) (i == 0) ? (N-1): (i-1))
#define NEXT(i) ((i+1)% N)

queue *q[N];                  // Cola de cada thread
pthread_mutex_t lock[N];  // mutex para proteger la cola de cada thread
pthread_cond_t  empty[N];

element remove(queue *q);           // Quitar elemento de cola
void insert(element e, queue *q);  // Insertar elemento en cola
int size(queue *q);                 // Devuelve el numero de elementos en la cola

int nodo(int num) // Num es el numero de thread dentro del anillo
{
        element *old, *new;
        int next =NEXT(num); // Siguiente en el anillo

        while(1) {
                pthread_mutex_lock(&lock[num]);
                if(size(q[num])==0)
                        pthread_cond_wait(&empty[num], &lock[num]);
                old = remove(q[num]);
                pthread_mutex_unlock(&lock[num]);

                new = process(old);

                pthread_mutex_lock(&lock[next]);
                insert(new, q[next]);
                if(size(q[next])==1)
                        pthread_cond_signal(&empty[next]);
                pthread_mutex_unlock(&lock[next]);
        }
}

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

   Se desea que el sistema sea capaz de procesar información en las dos direcciones, es decir, que el nodo **i** pueda:

   - Recibir elementos del thread **PREV(i)**, procesarlos, y encolarlos en el thread **NEXT(i)**, como hasta ahora.
   - Recibir elementos del thread **NEXT(i)**, procesarlos, y encolarlos en el thread **PREV(i)**.

   Modifique la implementación para añadir el comportamiento deseado.

   (Sugerencia: Añadir una cola más para cada thread para los elementos que van en la nueva dirección).

2. **Productores/Consumidores** [2 puntos]

Dadas las funciones de productores y consumidores de la práctica:

```
int producer_num = 0;
int consumer_num = 0;
pthread_mutex_t producer_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t consumer_mutex = PTHREAD_MUTEX_INITIALIZER;
void *producer_function(void *ptr)
{
        struct thread_info *t = ptr;
        unsigned int seed = t->thread_num;

        while(1) {
                pthread_mutex_lock(&product_count_mutex);
                if (product_count >= iterations) {
                        pthread_mutex_unlock(&product_count_mutex);
                        return NULL;
                }
                pthread_mutex_unlock(&product_count_mutex);
                struct element *e = produce_element();
                pthread_mutex_lock(&buffer_mutex);
                while(count == buffer_size) {
                        pthread_cond_wait(&buffer_full, &buffer_mutex);
                }
                insert_element(e);
                if(count==1)
                        pthread_cond_broadcast(&buffer_empty);
                pthread_mutex_unlock(&buffer_mutex);
        }
        return NULL;
}

void *consumer_function(void *ptr)
{
        struct thread_info *t = ptr;
        while(1) {
                pthread_mutex_lock(&consum_count_mutex);
                if (consum_count >= iterations) {
                        pthread_mutex_unlock(&consum_count_mutex);
                        return NULL;
                }
                consum_count ++;
                pthread_mutex_unlock(&consum_count_mutex);
                pthread_mutex_lock(&buffer_mutex);
                while(count==0){
                        pthread_cond_wait(&buffer_empty, &buffer_mutex);
                }
                struct element *e = remove_element();
                if(count == buffer_size -1)
                        pthread_cond_broadcast(&buffer_full);
                pthread_mutex_unlock(&buffer_mutex);

                consume_element(e);
                free(e);
        }
        return NULL;
}
int clock_gettime(clockid_t clk_id, struct timespec *tp);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex,
       const struct timespec *restrict abstime);
struct timespec {

        time_t   tv_sec;
        long     tv_nsec;
```

Se desea que los productores/consumidores esperen un máximo de 10ms. Si esperan más de ese tiempo, deben de cambiar al otro tipo (i.e. de consumidor a productor y de productor a consumidor). Hay que tener en cuenta que nunca podemos quedarnos con menos de un consumidor o productor. Para ello llevese una cuenta exacta del número de consumidores/productores en todo momento. Estos dos número tienen que reflejar la realidad en todo momento. Usar un mutex distinto para cada contador.

3. **Servidor Erlang que guarda un número** [1 puntos]

Disponemos de un servidor en Erlang que guarda un número, y permite incrementarlo y decrementarlo por una cantidad que se le pasa como parámetro, o reiniciarlo a 0.

```
-module(num_server).

-export([start/0, init/0, add/1, sub/1, reset/0]).

start() ->
        Pid = spawn(num_server, init, []),
        register(num_server, Pid).

add(N) ->
        num_server ! {add, N}.

sub(N) ->
        num_server ! {sub, N}.

reset() ->
        num_server ! reset.

init() ->
        loop(0).

loop(N) ->
        receive
                {add, X} ->
                        loop(N + X);
                {sub, X} ->
                        loop(N - X);
                reset ->
                        loop(0)
        end.
```

Añada al servidor:

- El que ha implementado el servidor se ha olvidado de proporcionar una forma de consultar el contador, por lo que no es muy útil. Haga que las funciones **add/1** y **sub/1** devuelvan el nuevo valor del contador.
- También se quiere poder conocer cuantas operaciones ha realizado el servidor. Modifique el servidor para que cuente cuantas operaciones **add** y **sub** se han realizado, y añada una función **get_ops/0** para consultarlo. El contador debería reiniciarse cuando se hace un **reset**.