Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Julio 2016

1. **Puente Estrecho** [2 puntos]

   Un puente antiguo de ancho reducido solo permite el tráfico simultáneo de vehículos en un sentido. Se desea diseñar un sistema que controle el acceso al puente de forma que un vehículo espere a la entrada si hay vehículos cruzando en sentido opuesto.

   Cada vehículo está representado por un thread que ejecuta la función `car`, y recibe como parámetro la dirección en la que cruza (0 o 1).

```
void enter_bridge(int direction) {
    ...
}

void exit_bridge(int direction) {
    ...
}

void *car(void *arg) {
    int direction = *((int *) arg);

    enter_bridge(direction);

    // Cross the bridge

    exit_bridge(direction);
}

int pthread_mutex_lock(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);
```

   Implemente las funciones `enter_bridge` y `exit_bridge`.

   Solución

```
pthread_mutex_t bridge;
pthread_mutex_t dir_m[2];
int car_count[2];

void enter_bridge(int direction) {
    pthread_mutex_lock(dir_m[direction]);
    if(car_count[direction]==0)
        pthread_mutex_lock(bridge);
    car_count[direction]++;
    pthread_mutex_unlock(dir_m[direction]);
}

void exit_bridge(int direction) {
    pthread_mutex_lock(dir_m[direction]);
    car_count[direction]--;
    if(car_count[direction]==0)
        pthread_mutex_unlock(bridge);
    pthread_mutex_unlock(dir_m[direction]);
}
```

2. **Comunicar threads a través de dos canales de comunicación** [2 puntos]

   Partimos de dos colas de elementos: `queue1` y `queue2`. Tenemos dos tipos de threads: `thread1` y `thread2`. Los threads `thread1`, quitan elementos de la `queue1`, le aplican la función `f1` y meten el resultado en `queue2`. La función `thread2` hace el trabajo inverso, quita los elementos de `queue2`, les aplica la función `f2` y mete los resultados en `queue1`. Se da el código de ejemplo y los prototipos de las funciones que se pueden utilizar. El código de ejemplo no realiza ninguna protección de ninguna de las colas. Proteger las dos colas teniendo en cuenta que:

   - `mutex1` se usa para proteger `queue1`.
   - `mutex2` se usa para proteger `queue2`.
   - minimizar el tiempo que se tiene cogido cada `mutex`.
   - Hay que tratar el caso de que `queue1` y `queue2` estén llenas (esperar a que haya sitio). Existe la función `queue_is_full()` que nos indica si la cola está llena. Consejo: Usar variables de condición.
   - Hay que tratar el caso de que `queue1` y `queue2` estén vacías (esperar a que haya elementos). Existe la función `queue_is_empty()` que nos indica si la cola está vacía. Consejo: Usar variables de condición.
   - Crear un thread de control que cada 10 segundos comprueba en ambas colas qué elementos cumplen el predicado `element_is_too_old()` y los quita de la cola.

```
struct queue queue1;
struct queue queue2;
bool queue_is_full(&struct queue queue);
bool queue_is_empty(&struct queue queue);

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t queue1_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t queue1_empty = PTHREAD_COND_INITIALIZER;

pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t queue2_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t queue2_empty = PTHREAD_COND_INITIALIZER;

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
bool element_is_too_old(struct element *e);


void *thread1(void *ptr)
{
        while(1) {
                struct element *new;
                struct element *old = remove_element(&queue1);
                new = f1(old);
                insert_element(&queue2, &new);
        }
        return NULL;
}

void *thread2(void *ptr)
{
        while(1) {
                struct element *new;
                struct element *old = remove_element(&queue2);
                new = f1(old);
                insert_element(&queue1, &new);
        }
        return NULL;
}
```

Solución

```
void *thread1(void *ptr)
{
        while(1) {
                struct element *new;

                pthread_mutex_lock(&mutex1);
                while(queue_is_empty(&queue1))
                    pthread_cond_wait(&queue1_empty, &mutex1);
                if(queue_is_full(&queue1))
                    pthread_cond_broadcast(&queue1_full);
                struct element *old = remove_element(&queue1);
                pthread_mutex_unlock(&mutex1);

                new = f1(old);

                pthread_mutex_lock(&mutex2);
                while(queue_is_full(&queue2))
                    pthread_cond_wait(&queue2_full, &mutex2);
                if(queue_is_empty(&queue2))
                    pthread_cond_broadcast(&queue2_empty);
                insert_element(&queue2, &new);
                pthread_mutex_unlock(&mutex2);
        }
        return NULL;
}

// Thread2 es igual que thread1 cambiando 1 por 2 (mutex1 => mutex2, ...)

void check_queue(queue *q, pthread_mutex_t *q_lock) {
        queue tmp;

        pthread_mutex_lock(q_lock);
        while(!queue_is_empty(q)) {
            struct element *e;

            e=remove_element(q);
            if(!element_is_too_old(e)) insert_element(&tmp, e);
        }

        while(!queue_is_empty(&tmp))
            insert_element(q, remove_element(&tmp));
        pthread_mutex_unlock(q_lock);
}

void *control(void *ptr)
{
    while(1) {
        sleep(10);
        check_queue(&queue1, &mutex1);
        check_queue(&queue2, &mutex2);
    }
}
```

3. **Almacén de Productos** [1 punto]

   Se desea implementar un almacén donde se puedan guardar y recuperar productos. El API es el siguiente:

   - `store:start()`, que arranca un almacén y devuelve su pid.
   - `store:store(S, P)`, que guarda el producto P en el almacén con pid S.
   - `store:get(S, F)`, donde S es un almacén, y F una propiedad sobre productos. La función devuelve {`error, no_product`} si no hay ningún producto que cumpla F en el almacén, y {`ok, P`} (siendo P un producto que cumple `F(P)`) si lo hay. Al devolver un producto se elimina del almacén.

```
-module(store).

-export([start/0, store/2, get/2]).
-export([loop/0]).

%% API

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

store(S, P) -> ...
get(S, F) -> ...

%% Internal Functions

loop() ->
    ...
```

   Implemente las funciones `store/2`, `get/2` y `loop/0`. Puede añadir funciones auxiliares y parámetros a `loop` si lo necesita.

Solución

```
-module(store).

-export([start/0, store/2, get/2]).
-export([loop/1]).

%% API

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

store(S, P) -> S ! {store, P}.
get(S, F) ->
    S ! {get, F, self()},
    receive

    end.

%% Internal Functions

loop(Prods) ->
    receive
        {store, P} ->
            loop([P | Prods]);
        {get, F, From} ->
            case lookup(Prods, F) of
                {ok, P, N_Prods} ->
                    From ! {get_reply, {ok, P}},
                    loop(N_Prods);
                no_product ->
                    From ! {get_reply, {error, no_product}},
                    loop(Prods)
            end
    end.

lookup(Prods, F) ->
    lookup(Prods, [], F).

lookup([], _, _) ->
    no_product
lookup([P | Rest], Acc, F) ->
    case F(P) of
        true ->
            {ok, P, Acc++Rest};
        false ->
            lookup(Rest, [P | Acc], F)
    end.
```