Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Julio 2015

1. **Implementar una cola que pueda usarse desde threads** [1.75 puntos]

   Partimos de la implementación de la cola que aparece a continuación. Tareas a realizar:

   - Hacer que la cola pueda usarse por varios threads a la vez.
   - La interfaz no puede cambiar.
   - Los elementos de sincronización necesarios estarán ocultos en el interfaz.
   - Si un thread intenta quitar un elemento y no hay elementos disponibles esperará hasta que los haya.
   - Minimizar el número de veces que se despierta a los threads.
   - Si hay más de SIZE elementos en la cola, un thread que intente añadir un elemento tiene que esperar a que haya menos elementos.

```
#define SIZE 100
struct node {
  struct node *next;
  struct data *data;
};
struct queue {
    struct node *first;
        struct node *last;
};

int queue_add(struct queue *queue,   struct data *data)
{
    struct node *new;

        new = malloc(sizeof(*new));

        if (new == NULL)
            return -1;

        new->data = data;
        new->next = NULL;
        if (last != NULL)
            last->next = new;
        else
            first = new;
        last = new;
}

struct data *queue_remove(struct queue *queue)
{
    struct data *result;
        struct node *old;

    if (first == NULL)
            return NULL;
        old = first;
        result = old->data;
        if (last == first)
            last = NULL;
        first = old->next;
        free(old);
        return result;
}
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);
```

Solución

```
#define SIZE 100
struct node {
    struct node *next;
    struct data *data;
};
struct queue {
    struct node *first;
    struct node *last;
    pthread_mutex_t *lock;
    pthread_cond_t *empty;
    pthread_cond_t *full;
    int elements;
    int waiting_empty;
    int waiting_full;
};

int queue_add(struct queue *queue,  struct data *data)
{
    struct node *new;

    new = malloc(sizeof(*new));

    if (new == NULL)
        return -1;

    new->data = data;
    new->next = NULL;

    pthread_mutex_lock(queue->lock);
    while(queue->elements==SIZE) {
        queue->waiting_full++;
        pthread_cond_wait(queue->full, queue->lock);
    }
    if (queue->last != NULL)
        queue->last->next = new;
    else
        queue->first = new;
    queue->last = new;
    if(queue->waiting_empty) {
        queue->waiting_empty--;
        pthread_cond_signal(queue->empty);
    }
    queue->elements++;
    pthread_mutex_unlock(queue->lock);
}

struct data *queue_remove(struct queue *queue)
{
    struct data *result;
    struct node *old;

    pthread_mutex_lock(queue->lock);
    while(queue->first == NULL) {
        queue->waiting_empty++;
        pthread_cond_wait(queue->empty, queue->lock);
    }
    old = queue->first;
    result = old->data;
    if (queue->last == first)
        queue->last = NULL;
    queue->first = old->next;
    queue->elements--;
    if(queue->waiting_full) {
        queue->waiting_full--;
        pthread_cond_signal(queue->full);
    }
    pthread_mutex_unlock(queue->lock);

    free(old);
    return result;
}
```

2. **Cambio de Barberos** [1.75 puntos]

En una barbería deciden permitir que los clientes puedan escoger a su barbero favorito para que les corte el pelo. Para ello cada cliente tiene un `prefered_barber`, que tendrá el número del barbero que quiere que le corte el pelo, o -1 si le da igual. Para gestionar los clientes existe una cola global donde se insertan los clientes que no tienen preferencia(`customers`), y una cola por cada barbero donde se insertan los clientes que quieren que les corte el pelo un barbero específico(`barber_queue`)

Los barberos comprueban primero si hay algún cliente que les prefiere a ellos, y si no hay miran si hay clientes sin preferencia. Mientras le cortan el pelo a un cliente guardan la preferencia de ese cliente en su propia estructura como `current_customer_prefered`.

```
queue *customers;
queue *barber_queue[num_barbers];
int barber_state[num_barbers];
pthread_mutex_t mutex;
pthread_cond_t no_customers;

struct customer_info {
    int customer_num;
    pthread_cond_t cond;
    int time;
    int prefered_barber;
};
struct barber_info {
    int barber_num;
    int current_customer_prefered;
    pthread_cond_t cutting_hair;
} barbers[num_barbers];

void *barber_function(void *ptr)
{
    struct barber_info *t = ptr;
    while(1) {
        pthread_mutex_lock(&mutex);
        while(queue_is_empty(customers) && queue_is_empty(barber_queue[t->barber_num]) {
            barber_state[t->barber_num]=SLEEPING;
            printf("barber_%d_goes_to_sleep\n", t->barber_num);
            pthread_cond_wait(&no_customers, &mutex);
            barber_state[t->barber_num]=WORKING;
        }

        if(!queue_is_empty(barber_queue[t->barber_num]) c = queue_remove(t->barber_num);
        else c = queue_remove(customers);

        t->current_customer_prefered = c->prefered_barber;
        pthread_cond_signal(c->cond);

        pthread_mutex_unlock(&mutex);
        usleep(c->time); // cut_hair
    }
}

void *customer_function(void *ptr)
{
    struct customer_info *t = ptr;

    pthread_mutex_lock(&mutex);
    if(waiting_customers == num_chairs) {
        printf("waiting_room_full_for_customer_%d\n", t->customer_num);
        pthread_mutex_unlock(&mutex);
        return NULL;
    }

    if(t->prefered_barber == -1) insert_queue(customers, t);
    else insert_queue(barber_queue[t->prefered_barber], t);

    pthread_cond_broadcast(&no_customers);
    pthread_cond_wait(&t->cond, &mutex);

    pthread_mutex_unlock(&mutex);
    get_hair_cut(t->customer_num);
    return NULL;
}
```

Ahora mismo es posible que llegue un cliente nuevo cuando hay barberos libres, pero su barbero le está cortando el pelo a un cliente al que le daba igual quien le cortaba el pelo. Modifique el código para que si se da esta situación el cliente pueda avisar a su barbero preferido para que pare de cortar el pelo al cliente actual, lo vuelva a poner en la cola para que otro barbero le corte el pelo, y pase a cortarle el pelo al cliente nuevo.

```
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_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Solución

```
void *barber_function(void *ptr)
{
    struct timespec tout;
    struct barber_info *t =  ptr;
    int ret;

    while(1) {
        pthread_mutex_lock(&mutex);
        while(queue_is_empty(customers) && queue_is_empty(barber_queue[t->barber_num]) {
            barber_state[t->barber_num]=SLEEPING;
            printf("barber_%d_goes_to_sleep\n", t->barber_num);
            pthread_cond_wait(&no_customers, &mutex);
            barber_state[t->barber_num]=WORKING;
        }

        if(!queue_is_empty(barber_queue[t->barber_num]) c = queue_remove(t->barber_num);
        else c = queue_remove(customers);

        t->current_customer_prefered = c->prefered_barber;

        clock_gettime(CLOCK_REALTIME, &tout);
        tout.tv_sec+=(tout.tv_nsec+c->time*1000) / 1000000000;
        tout.tv_nsec=(tout.tv_nsec+c->time*1000) % 1000000000;

        ret = pthread_cond_timedwait(t->cutting_hair, &mutex, &tout); // cut_hair

        if(ret==ETIMEDOUT) { // finished cutting hair
            pthread_cond_signal(c->cond);
        } else { // signaled by a priority customer
            insert_queue(customers, c);
            pthread_cond_broadcast(&no_customers);
        }
        pthread_mutex_unlock(&mutex);
    }
}

void *customer_function(void *ptr)
{
    struct customer_info *t =  ptr;

    pthread_mutex_lock(&mutex);
    if(waiting_customers == num_chairs)  {
        printf("waiting_room_full_for_customer_%d\n", t->customer_num);
        pthread_mutex_unlock(&mutex);
        return NULL;
    }

    if(t->prefered_barber == -1) insert_queue(customers, t);
    else {
        if(barbers[t->prefered_barber].current_customer_prefered==-1)
            pthread_cond_signal(barbers[t->prefered_barber].prefered_client);
        insert_queue(barber_queue[t->prefered_barber], t);
    }

    pthread_cond_broadcast(&no_customers);
    pthread_cond_wait(&t->cond, &mutex);

    pthread_mutex_unlock(&mutex);
    get_hair_cut(t->customer_num);
    return NULL;
}
```

3. **Diccionario Concurrente** [1.5 puntos]

Un diccionario es un servicio que permite almacenar y recuperar pares clave/valor. En un sistema donde existe un diccionario se observa que hay un número alto de lecturas y escrituras simultaneas y supone un cuello de botella. Para mejorar el rendimiento se decide implementar un diccionario concurrente que reparta el diccionario entre varios procesos. Para realizar el reparto cada proceso tiene un número asignado de forma aleatoria al arrancar el programa entre 0 y N. Disponemos también de una función de hash que se aplica a las claves, y que nos dan un número entre 0 y N. Cada proceso del sistema almacenará los pares clave/valor para los que su número de proceso sea el más cercano al hash de la clave (por ejemplo, si tenemos procesos con números 1, 5 y 9, un par clave/valor cuyo hash sea 2 estaría guardado en el proceso número 1).

El sistema tiene dos tipos de procesos:

- Stores, que son los que guardarán las claves. Cada store tiene un número asignado.
- Un Frontend, registrado con el nombre dict, que será el proceso con el que se comunican los clientes del diccionario. Este proceso se encargará de enviar las peticiones al store que corresponda. Este proceso puede usar la función `hash:from_key(Key)` para obtener el hash asociado a una clave.

```
-module(dict).
-export([start/1, get/1, put/2, init_frontend/1, init_store/1]).

%%%%%% API %%%%%%
start(N) ->
    Stores = start_stores(N,N),
    spawn(?MODULE, init_frontend, [Stores]).

get(K) -> ...

put(K, V) -> ...

%%%%% Inicializacion %%%%%%
start_stores(0, _) -> [];
start_stores(S, N) -> [spawn(?MODULE, init_store, [random:uniform(N)]) | start_stores(S-1)].

init_frontend(Stores) ->
    register(dict, self()),
    loop_frontend(Stores).

init_store(N) ->
    loop_store(N, []).

%%%%% Loops y funciones auxiliares del diccionario
loop_frontend(Stores) ->
    receive
        ...
    end.

loop_store(N, D) ->
    receive
            ...
    end.

lookup(K, []) -> not_found;
lookup(K, [{K, V}|_]) -> {ok, V};
lookup(K, [_|T]) -> lookup(K, T).

store(K, V, []) -> [{K,V}];
store(K, V, [{K, _} | T]) -> [{K,V}|T];
store(K, V, [{K2, V2} | T]) -> [{K2,V2}| store(K,V,T)].
```

Se pide:

(a) Implementar la función `get(K)`, que dada una clave devuelve su valor. Modifique lo que sea necesario en `loop_frontend` y `loop_store`.

(b) Implementar la función `put(K, V)`, que dada una clave y un valor la guarda en el diccionario. Modifique lo que sea necesario en `loop_frontend` y `loop_store`.

Solución

```
-module(dict).
-export([start/1, get/1, put/2, init_frontend/1, init_store/1]).

%%%%%% API %%%%%%
start(N) ->
    Stores = start_stores(N,N),
    spawn(?MODULE, init_frontend, [Stores]).

get(K) ->
    dict ! {get, K, self()},
    receive
        {get_reply, Reply} -> Reply
    end.

put(K, V) ->
    dict ! {put, K, V}.

%%%%% Inicializacion %%%%%%
start_stores(0, _) -> [];
start_stores(S, N) ->
    Id = random:uniform(N),
    [{Id, spawn(?MODULE, init_store, [Id])} | start_stores(S-1)].

init_frontend(Stores) ->
    register(dict, self()),
    loop_frontend(Stores).

init_store(N) ->
    loop_store(N, []).

%%%%% Loops y funciones auxiliares del diccionario
loop_frontend(Stores) ->
    receive
        {get, K, From} ->
            Pid = find_closest_store(K, Stores),
            Pid ! {get, K, From},
            loop_frontend(Stores);
        {put, K, V} ->
            Pid = find_closest_store(K, Stores),
            Pid ! {put, K, V},
            loop_frontend(Stores)
    end.

find_closest_store(K, Stores) ->
    Hash = hash:from_key(K),
    Distances = [{abs(Hash-Id), Pid} || {Id, Pid} <- Stores],
    minimum(Distances).

minimum({_, Pid}, []) -> Pid;
minimum({Dist, Pid}, [{Dist2, Pid2} | T]) ->
    if
        Dist < Dist2 ->
            minimum({Dist, Pid}, T);
        true ->
            minimum({Dist2, Pid2}, T)
    end.

minimum([]) -> error;
minimun([H|T]) -> minimum(H,T).


loop_store(N, D) ->
    receive
        {get, K, From} ->
            From ! {get_reply, lookup(K, D)},
            loop_store(N, D);
        {put, K, V} ->
            loop_store(N, [{K, V} | [{K2, V2} || {K2,V2} <- D, K/= K2]])
    end.

lookup(K, []) -> not_found;
lookup(K, [{K, V}|_]) -> {ok, V};
lookup(K, [_|T]) -> lookup(K, T).

store(K, V, []) -> [{K,V}];
store(K, V, [{K, _} | T]) -> [{K,V}|T];
store(K, V, [{K2, V2} | T]) -> [{K2,V2}| store(K,V,T)].
```