Apellidos: **Nombre:**

Concurrencia y Paralelismo

Grado en Ingeniería Informática

Examen Mayo 2018

1. **Control de Elementos en una Cola** [2 puntos]

   Dada la implementación de una cola mediante un array circular como la utilizada en la práctica 2:

```
typedef struct _queue {
    int size;     // max queue size
    int used;     // current number of elements
    int first;    // index of the first element
    void **data; // element array
    pthread_mutex_t *lock;
} *queue;

int q_insert(queue q, void *elem) {
    pthread_mutex_lock(q->lock);
    if(q->size==q->used) {
        pthread_mutex_unlock(q->lock);
        return 0;
    }

    q->data[(q->first+q->used) %q->size] = elem;
    q->used++;

    pthread_mutex_unlock(q->lock);
    return 1;
}

void *q_remove(queue q) {
    void *res;

    pthread_mutex_lock(q->lock);
    if(q->used==0) {
        pthread_mutex_unlock(q->lock);
        return NULL;
    }

    res=q->data[q->first];

    q->first=(q->first+1) %q->size;
    q->used--;
    pthread_mutex_unlock(q->lock);

    return res;
}


void q_run_when_gt(queue q,void (*f)(queue), int size) {
    ...
}
```

Implemente una función `q_run_when_gt(queue q, void (*f) (queue), int size)`, que llame a la función `f` pasando la cola `q` como parámetro cuando la cola `q` tenga más de `size` elementos. Solo es necesario llamar a la función `f` una vez por cada llamada a `q_run_when_gt`. La función `f` se debe llamar con el mutex de la cola bloqueado. La implementación de la cola puede modificarse si es necesario.

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

Solución

La función `q_run_when_gt` necesita comparar el número de elementos de la cola (`q->used`) con el tamaño pasado por parámetro (`size`). En caso de que no haya un número suficiente de elementos, hay que parar al thread que llamó a `q_run_when_gt` hasta que los haya. Por tanto,

vamos a necesitar una condición para la espera. Como esa espera está ligada al número de elementos de la cola, la condición se guarda en la estructura `queue`.

```
typedef struct _queue {
    int size;     // max queue size
    int used;     // current number of elements
    int first;    // index of the first element
    void **data; // element array
    pthread_mutex_t *lock;
    pthread_cond_t *wait_for_size;
} *queue;
```

En `q_run_when_gt` hay comprobar si hay que esperar a que haya el número de elementos necesario (es decir, hay que esperar si `size <= q->used`). Otros threads van a modificar q-¿used, por tanto hay que proteger esa comprobación hasta que se haga la espera, o se llame a `f`. Por ello se bloquea el mutex que protege `q->used` (`q->lock`), y se suelta durante la espera para que otros threads puedan modificar la cola (`pthread_cond_wait(q->wait_for_size, q->lock)`).

```
void q_run_when_gt(queue q,void (*f)(queue), int size) {
    pthread_mutex_lock(q->lock);
    while(size > q->used)
        pthread_cond_wait(q->wait_for_size, q->lock);
    f(q);
    pthread_mutex_unlock(q->lock);
}
```

Como hemos puesto threads a esperar, hay que despertarlos en algún sitio. Como dependemos de que el tamaño suba hasta una cierta cantidad, lo lógico es despertar desde la función de insertar:

```
int q_insert(queue q, void *elem) {
    pthread_mutex_lock(q->lock);
    if(q->size==q->used) {
        pthread_mutex_unlock(q->lock);
        return 0;
    }

    q->data[(q->first+q->used) %q->size] = elem;
    q->used++;

    pthread_cond_broadcast(q->wait_for_size);
    pthread_mutex_unlock(q->lock);
    return 1;
}
```

Es necesario hacer un broadcast porque puede haber más de un thread llamando a `q_run_when_gt`. Y hay que hacerlo siempre porque no sabemos el tamaño por el que cada thread espera. Si quisieramos evitar despertar a los threads cada vez que se inserta habría que habilitar un mecanismo para registrar todos lo tamaños por los que hay algún thread esperando.

2. **Algoritmo de los barberos** [1.75 puntos]

Partimos del ejemplo dado en clase de tener múltiples barberos. Existe una cola para que los clientes se atiendan en orden de llegada. Hay `N` barberos numerados del `1` al `N`.

El ejercicio consiste en cambiar el código para que un cliente pueda pedir que le atenda un barbero en partícular $t->barber\_num$. La cola `0` se reserva para los clientes a los que les da igual el barbero. Estos clientes tendrán $t->barber\_num == 0$

Cuando un barbero despierta, primero mira en su cola particular, y en segundo lugar en la cola cero.

```
pthread_mutex_t mutex[NUM_BARBERS+1];
pthread_cond_t cond[NUM_BARBERS+1];
struct queue queue[NUM_BARBERS+1];
struct customer {
  pthread_cond_t wait;
  int num;
}
bool queue_is_empty(&struct queue queue);
void insert_customer(&struct queue queue, struct customer *customer);
struct customer *retrieve_customer(&struct queue queue);
int waiting_customers;
pthread_mutex_lock_t mutex_waiting;

void *barber_function(void *ptr)
{
        struct barber_info *t =  ptr;

        while(true) {
                struct customer *customer;
                pthread_mutex_lock(&mutex[0]);

                while (queue_is_empty(&queue[0])) {
                        pthread_cond_wait(&cond[0], &mutex[0]);
                }

                customer = retrieve_customer(queue[0]);

                pthread_cond_signal(&customer->wait);
                waiting_customers--;

                pthread_mutex_unlock(&mutex[0]);
                cut_hair(t->barber_num, customer->num);
                free(customer);
                }
        }
}

void *customer_function(void *ptr)
{
        struct customer_info *t =  ptr;
        struct customer *customer = malloc(sizeof(struct customer));
        int barber = t->barber_num;

        pthread_mutex_lock(&mutex_waiting);
        if(waiting_customers == num_chairs)  {
                printf("waiting room full for customer %d\n", t->customer_num);
                pthread_mutex_unlock(&mutex_waiting);
                return NULL;
        }
        pthread_mutex_unlock(&mutex_waiting);

        pthread_cond_init(&customer->wait, NULL);
        customer->num = t->customer_num;

        pthread_mutex_lock(&mutex[0]);
        insert_customer(queue[0], customer);

        pthread_cond_broadcast(&cond[0]);
        waiting_customers++;
        pthread_cond_wait(&customer->wait, &mutex[0]);
        pthread_mutex_unlock(&mutex[0]);
        get_hair_cut(t->customer_num);
        return NULL;
}
```

Solución

Nos piden que hagamos que cada barbero tenga una cola privada. Vamos a dejar para más adelante la cola compartida. Lo primero que se nos ocurre es cambiar todas las apariciones de `[0]` por `[t->num]` en el barbero y `[barber]` en el cliente. Pero al hacer estos vemos que entonces `waiting_customers` pasa a estar protegido para cada barbero/cliente por un `mutex` distinto. Pensamos que hacer, y nos damos cuenta que en el enunciado aparece un `mutex_waiting` que casi no esta usado más que para ... proteger `waiting_customers`. Hacemos primero este segundo cambio. Nótese como hemos cambiado un poco el código para que no tengamos nunca dos `mutex` a la vez y no pueda haber interbloqueo.

```
void *barber_function(void *ptr)
{
        struct barber_info *t =  ptr;

        while(true) {
                struct customer *customer;
                pthread_mutex_lock(&mutex[0]);

                while (queue_is_empty(&queue[0])) {
                        pthread_cond_wait(&cond[0], &mutex[0]);
                }

                customer = retrieve_customer(queue[0]);

                pthread_cond_signal(&customer->wait);
                pthread_mutex_unlock(&mutex[0]);
                pthread_mutex_lock(&mutex_waiting);
                waiting_customers--;
                pthread_mutex_unlock(&mutex_waiting);
                cut_hair(t->barber_num, customer->num);
                free(customer);
                }
        }
}

void *customer_function(void *ptr)
{
        struct customer_info *t =  ptr;
        struct customer *customer = malloc(sizeof(struct customer));
        int barber = t->barber_num;

        pthread_mutex_lock(&mutex_waiting);
        if(waiting_customers == num_chairs)  {
                printf("waiting room full for customer %d\n", t->customer_num);
                pthread_mutex_unlock(&mutex_waiting);
                return NULL;
        }
        waiting_customers++;
        pthread_mutex_unlock(&mutex_waiting);

        pthread_cond_init(&customer->wait, NULL);
        customer->num = t->customer_num;

        pthread_mutex_lock(&mutex[0]);
        insert_customer(queue[0], customer);

        pthread_cond_broadcast(&cond[0]);
        pthread_cond_wait(&customer->wait, &mutex[0]);
        pthread_mutex_unlock(&mutex[0]);
        get_hair_cut(t->customer_num);
        return NULL;
}
```

Una vez hecho este cambio, ya podemos hacer que funcionen las colas para barberos individuales. Por ahora en la cola cero no hacemos nada.

```
void *barber_function(void *ptr)
{
        struct barber_info *t =  ptr;

        while(true) {
                struct customer *customer;
                pthread_mutex_lock(&mutex[t->barber_num]);
```

```
                while (queue_is_empty(&queue[t->barber_num])) {
                        pthread_cond_wait(&cond[t->barber_num], &mutex[t->barber_num]);
                }

                customer = retrieve_customer(queue[t->barber_num);

                pthread_cond_signal(&customer->wait);
                pthread_mutex_unlock(&mutex[t->barber_num]);
                pthread_mutex_lock(&mutex_waiting);
                waiting_customers--;
                pthread_mutex_unlock(&mutex_waiting);
                cut_hair(t->barber_num, customer->num);
                free(customer);
                }
        }
}

void *customer_function(void *ptr)
{
        struct customer_info *t =  ptr;
        struct customer *customer = malloc(sizeof(struct customer));
        int barber = t->barber_num;

        pthread_mutex_lock(&mutex_waiting);
        if(waiting_customers == num_chairs)  {
                printf("waiting room full for customer %d\n", t->customer_num);
                pthread_mutex_unlock(&mutex_waiting);
                return NULL;
        }
        waiting_customers++;
        pthread_mutex_unlock(&mutex_waiting);

        pthread_cond_init(&customer->wait, NULL);
        customer->num = t->customer_num;

        pthread_mutex_lock(&mutex[barber]);
        insert_customer(queue[barber], customer);

        pthread_cond_broadcast(&cond[barber]);
        pthread_cond_wait(&customer->wait, &mutex[barber]);
        pthread_mutex_unlock(&mutex[barber]);
        get_hair_cut(t->customer_num);
        return NULL;
}
```

Ahora vamos a comenzar a hacer la parte donde implementamos el hecho de que hay clientes que les vale cualquier barbero. Estos usarán el barbero cero para indicar este hecho. Cuando un barbero no tiene trabajo, ahora va a tener que mirar en dos colas: la propia y si en esta no hay nada, la cero. Antes de implementar esta parte, vamos a cambiar el bucle while en un bucle `true` con la condición dentro. Como vamos a tener que quitar de varias colas, tambien metemos dentro de la condición quitar de la cola correspondiente.

```
void *barber_function(void *ptr)
{
        struct barber_info *t =  ptr;

        while(true) {
                struct customer *customer;
                pthread_mutex_lock(&mutex[t->barber_num]);

                while (true) {
                    if (!queue_is_empty(&queue[t->barber_num])) {
                            customer = retrieve_customer(queue[t->barber_num);
                            break;
                        }
                    pthread_cond_wait(&cond[t->barber_num], &mutex[t->barber_num]);
                }

                pthread_cond_signal(&customer->wait);
                pthread_mutex_unlock(&mutex[t->barber_num]);
                pthread_mutex_lock(&mutex_waiting);
                waiting_customers--;
                pthread_mutex_unlock(&mutex_waiting);
                cut_hair(t->barber_num, customer->num);
                free(customer);
    }
```

```
}
```

Después de toda la preparación, ya podemos realizar el cambio. En el barbero, vamos a comprobar si hay algo en la cola cero antes de ir a dormir. Y en el cliente, vamos a comprobar si es el barbero cero, y en ese caso tenemos que despertar a todos los barberos. Para no complicar el código y dejarlo más claro, repetimos todo el código para el caso de elegir o no barbero.

```
void *barber_function(void *ptr)
{
        struct barber_info *t =  ptr;

        while(true) {
                struct customer *customer;
                pthread_mutex_lock(&mutex[t->barber_num]);

                while (true) {
                 if (!queue_is_empty(&queue[t->barber_num])) {
                 customer = retrieve_customer(queue[t->barber_num);
                                break;
                        }
                        pthread_mutex_lock(&mutex[0]);
                 if (!queue_is_empty(&queue[0])) {
                 customer = retrieve_customer(queue[0);
                        pthread_mutex_unlock(&mutex[0]);
                                break;
                        }
                        pthread_mutex_unlock(&mutex[0]);
        pthread_cond_wait(&cond[t->barber_num], &mutex[t->barber_num]);
                }

                pthread_cond_signal(&customer->wait);
                pthread_mutex_unlock(&mutex[t->barber_num]);
                pthread_mutex_lock(&mutex_waiting);
                waiting_customers--;
                pthread_mutex_unlock(&mutex_waiting);
                cut_hair(t->barber_num, customer->num);
                free(customer);
                }
        }
}

void *customer_function(void *ptr)
{
        struct customer_info *t =  ptr;
        struct customer *customer = malloc(sizeof(struct customer));
        int barber = t->barber_num;

        pthread_mutex_lock(&mutex_waiting);
        if(waiting_customers == num_chairs)  {
                printf("waiting room full for customer %d\n", t->customer_num);
                pthread_mutex_unlock(&mutex_waiting);
                return NULL;
        }
        waiting_customers++;
        pthread_mutex_unlock(&mutex_waiting);

        pthread_cond_init(&customer->wait, NULL);
        customer->num = t->customer_num;

        pthread_mutex_lock(&mutex[barber]);
        insert_customer(queue[barber], customer);

        if (barber == 0) {
         int i;

                for (i = 0; i <= NUM_BARBERS; i++) {
                 pthread_cond_signal(&cond[i]);
                }
        } else {
        pthread_cond_signal(&cond[barber]);
        }
        pthread_cond_wait(&customer->wait, &mutex[barber]);
        pthread_mutex_unlock(&mutex[barber]);
        get_hair_cut(t->customer_num);
        return NULL;
}
```

Esta solución tiene una **race condition** en el caso de que el barbero se vaya a dormir después de comprobar la cola cero y que lo despierten porque hay un cliente en la cola cero. Una solución es bloquear la suelta de los mutex por el barbero y la espera con un mutex extra, que el cliente bloquea para despertar.

```
void *barber_function(void *ptr)
{
        struct barber_info *t =  ptr;

        while(true) {
                struct customer *customer;

                while (true) {
                    pthread_mutex_lock(&mutex[t->barber_num]);
                 if (!queue_is_empty(&queue[t->barber_num])) {
                 customer = retrieve_customer(queue[t->barber_num);
                            break;
                        }
                        pthread_mutex_lock(&mutex[0]);
                 if (!queue_is_empty(&queue[0])) {
                        customer = retrieve_customer(queue[0);
                        pthread_mutex_unlock(&mutex[0]);
                            break;
                        }
                    pthread_mutex_lock(&extra);
                    pthread_mutex_unlock(&mutex[0]);
                    pthread_mutex_unlock(&mutex[t->barber_num];
                    pthread_cond_wait(&cond[t->barber_num], &extra);
                    pthread_mutex_unlock(&extra);
                }

                pthread_cond_signal(&customer->wait);
                pthread_mutex_unlock(&mutex[t->barber_num]);
                pthread_mutex_lock(&mutex_waiting);
                waiting_customers--;
                pthread_mutex_unlock(&mutex_waiting);
                cut_hair(t->barber_num, customer->num);
                free(customer);
                }
        }
}

void *customer_function(void *ptr)
{
        struct customer_info *t =  ptr;
        struct customer *customer = malloc(sizeof(struct customer));
        int barber = t->barber_num;

        pthread_mutex_lock(&mutex_waiting);
        if(waiting_customers == num_chairs)  {
                printf("waiting room full for customer %d\n", t->customer_num);
                pthread_mutex_unlock(&mutex_waiting);
                return NULL;
        }
        waiting_customers++;
        pthread_mutex_unlock(&mutex_waiting);

        pthread_cond_init(&customer->wait, NULL);
        customer->num = t->customer_num;

        pthread_mutex_lock(&mutex[barber]);
        insert_customer(queue[barber], customer);

        pthread_mutex_lock(&extra);

        if (barber == 0) {
            int i;

            for (i = 0; i <= NUM_BARBERS; i++) {
                pthread_cond_signal(&cond[i]);
            }
        } else {
            pthread_cond_signal(&cond[barber]);
        }
        pthread_mutex_unlock(&extra);
```

```
        pthread_cond_wait(&customer->wait, &mutex[barber]);
        pthread_mutex_unlock(&mutex[barber]);
        get_hair_cut(t->customer_num);
        return NULL;
}
```

---

3. **Cadena de Procesos** [1.25 puntos]

   El módulo chain implementa una cadena de procesos, donde cada elemento de la cadena conoce el PID del siguiente. Los procesos se numeran de N (primer proceso) a 0 (último).

   La cadena se crea llamando a `start(N)`, donde `N` es el número del primer proceso. Esta función devuelve el PID del primer proceso de la cadena. La función `send(Chain, Msg)` envía un mensaje desde el primer elemento al último.

   Un ejemplo de ejecución es:

```
1> C = chain:start(3).
<0.62.0>
2> chain:send(C, hola).
hola: 3 steps to go...
hola: 2 steps to go...
hola: 1 steps to go...
hola: end of chain
```

---

donde la cadena `C` tendría la siguiente estructura:


```
-module(chain).

-export([start/1, init/1, send/2]).

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

send(Chain, Msg) ->
    Chain ! {send, Msg}.

init(0) ->
    final_proc_loop();
init(N) ->
    Pid = spawn(?MODULE, init, [N-1]),
    proc_loop(Pid, N).

final_proc_loop() ->
    receive
        {send, Msg} ->
            io:format("~w: end of chain~n", [Msg])
    end,
    final_proc_loop().

proc_loop(Next, N) ->
    receive
        {send, Msg} ->
            io:format("~w: ~w steps to go...~n", [Msg, N]),
            Next ! {send, Msg}
    end,
    proc_loop(Next, N).
```

Cambie el código proporcionado para que los mensajes vayan hasta el final de la cadena, y después vuelvan hasta el principio. La salida del ejemplo anterior tras las modificaciones debería ser:

```
1> C = chain:start(3).
<0.62.0>
2> chain:send(C, hola).
hola: 3 steps to go...
hola: 2 steps to go...
hola: 1 steps to go...
hola: end of chain, going back
hola: 1 steps back...
hola: 2 steps back...
hola: 3 steps back...
```

Solución

Para que el mensaje pueda volver hacia atrás necesitamos que cada nodo conozca al anterior en la cadena. Para esto necesitamos modificar la inicialización. Cada vez que se cree un proceso nuevo en `init`, pasamos el pid del proceso que lo está creando. Por tanto, cada proceso recibe como parámetro el pid del proceso anterior:

```
init(0, Prev) ->
    final_proc_loop(Prev);
init(N, Prev) ->
    Pid = spawn(?MODULE, init, [N-1, self()]),
    proc_loop(Pid, N, Prev).
```

En el código actual se estaba diferenciando al último, porque tiene un comportamiento distinto a los demás. Ahora tenemos que separar también al primero, porque es el único que no tiene anterior en la cadena. Vamos a añadir un caso especial en la inicialización, y le crearemos después un loop distinto. En `start` creamos la cadena lanzando el primer proceso en `init_first`. La inicialización completa queda así:

```
start(N) ->
    spawn(?MODULE, init_first, [N]).

init_first(0) ->
    final_proc_loop();
init_first(N) ->
    Pid = spawn(?MODULE, init, [N-1, self()]),
    first_proc_loop(Pid, N).

init(0, Prev) ->
    final_proc_loop(Prev);
init(N, Prev) ->
    Pid = spawn(?MODULE, init, [N-1, self()]),
    proc_loop(Pid, N, Prev).
```

Ahora hay que codificar los `loops`. Tenemos que hacer tres, los del primero, el último, y los demás. Además hay que distinguir los mensajes que van hacia el final de la cadena de los que vuelven hacia el principio. Vamos a partir del mismo loop que teníamos, y añadir un nuevo mensaje para los mensajes que vuelven:

```
proc_loop(Next, N, Prev) ->
    receive
        {send, Msg} ->
            io:format("~w: ~w steps to go...~n", [Msg, N]),
            Next ! {send, Msg};
        {send_back, Msg} ->
            io:format("~w: ~w steps back...~n", [Msg, N]),
            Prev ! {send_back, Msg}
    end,
    proc_loop(Next, N, Prev).
```

Los del primero y el último son similares, y la solución completa quedaría así:

```
-module(chain).

-export([start/1, init_first/1, init/2, send/2]).

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

send(Chain, Msg) ->
    Chain ! {send, Msg}.

init_first(0) -> %%If N is 0 when starting the chain, there will be only one process
    final_proc_loop();
init_first(N) ->
    Pid = spawn(?MODULE, init, [N-1, self()]),
    first_proc_loop(Pid, N).

init(0, Prev) ->
    final_proc_loop(Prev);
```

```
init(N, Prev) ->
    Pid = spawn(?MODULE, init, [N-1, self()]),
    proc_loop(Pid, N, Prev).

final_proc_loop() -> %%Only one process
    receive
        {send, Msg} ->
            io:format("~w: end of chain, going back~n", [Msg])
    end,
    final_proc_loop().

final_proc_loop(Prev) -> %%Final process in a chain with more than one
    receive
        {send, Msg} ->
            io:format("~w: end of chain, going back~n", [Msg]),
            Prev ! {send_back, Msg}
    end,
    final_proc_loop(Prev).

proc_loop(Next, N, Prev) -> %%Intermediate process in a chain with more than two
    receive
        {send, Msg} ->
            io:format("~w: ~w steps to go...~n", [Msg, N]),
            Next ! {send, Msg};
        {send_back, Msg} ->
            io:format("~w: ~w steps back...~n", [Msg, N]),
            Prev ! {send_back, Msg}
    end,
    proc_loop(Next, N, Prev).

first_proc_loop(Next, N) -> %%First process in a chain with more than one
    receive
        {send, Msg} ->
            io:format("~w: ~w steps to go...~n", [Msg, N]),
            Next ! {send, Msg};
        {send_back, Msg} ->
            io:format("~w: ~w steps back...~n", [Msg, N]),
    end,
    first_proc_loop(Next, N).
```