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);
```

---

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;
```

```
}
```

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

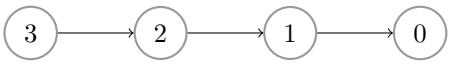
   El módulo chain implementa una cadena de procesos, donde cada elemento de la cadena conoce el PID del siguiente. Los procesos se numeran de N (primer proceso) a 0 (último).

   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...
```
