Apellidos: *Nombre:*

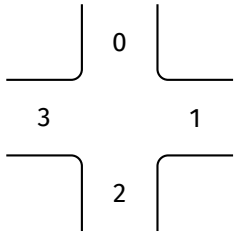
Concurrencia y Paralelismo

Grado en Ingeniería Informática

Mayo 2019

1. *Control de tráfico en una intersección [2 puntos]*

   Vamos a simular una intersección en una carretera donde hay un semáforo que deja pasar a los coches que llegan desde cada una de las direcciones consecutivamente.



   Tanto el semáforo como cada coche que llega a la intersección tiene un thread propio. La intersección se representa con un valor de tipo `intersection`, donde hay un campo `direction` que indica la dirección que puede cruzar ahora mismo, y una constante `total_directions` que indica el número total de direcciones en la intersección.

   El código siguiente implementa el cambio periódico de dirección:

```
typedef struct _intersection {
    int direction;               // Current direction
    int total_directions;        // Total number of directions
    pthread_mutex_lock *i_lock;
    ...
} *intersection;

int semaphore(intersection i) {

    while(1) {
        pthread_mutex_lock(i->i_lock);
        i->direction = (i->direction + 1) % i->total_directions;
        pthread_mutex_unlock(i->i_lock);

        sleep(10);
    }
}

int car(int direction, intersection i) {
    ...
}
```

   Implemente el comportamiento del coche para que espere hasta que el semáforo deje pasar a los coches de su dirección. Puede añadir campos a la estructura `intersection` y cambiar la implementación de la función `semaphore` 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

```
typedef struct _intersection {
    int direction;               // Current direction
    int total_directions;        // Total number of directions
    pthread_mutex_t *i_lock;
    pthread_cond_t *i_wait; // malloc(sizeof(pthread_cond_t) * total_directions);
                            //    (equivalent to pthread_cond_t i_wait[total_directions])
                            // initialized using pthread_cond_init(i_wait[i], NULL)
                            //    in a loop
} *intersection;

int semaphore(intersection i) {

    while(1) {
        pthread_mutex_lock(i->i_lock);
        i->direction = (i->direction + 1) % i->total_directions;
        pthread_cond_broadcast(i->i_wait[i->direction]); // Wake up the cars waiting for
                                                         //   green after changing
        pthread_mutex_unlock(i->i_lock);

        sleep(10);
    }
}

int car(int direction, intersection i) {
    pthread_mutex_lock(i->i_lock);
    while(direction != i->direction)                      // Check if the semaphore is
                                                          //   green for us while holding
                                                          //   i->lock

        pthread_cond_wait(i->i_wait[direction], i->i_lock);  // If it isn't, wait.
                                                             // Release i_lock while waiting,
                                                             // which will let the semaphore
                                                             // change i->direction


    pthread_mutex_unlock(i->lock);
}
```

2. *Comunicación entre barberos y clientes [1.75 puntos]*

   Dado el algorítmo de barberos dado en clase expuesto a continuación, conseguir que el barbero sepa a que cliente le está cortando el pelo y viceversa para que puedan imprimirse los mensajes con los valores correctos. Téngase en cuenta que debe funcionar tanto si el cliente espera en la sala de espera como si no hay espera. Y que el algorítmo tiene que funcionar cuando hay más de un barbero.

```
void barber(int barber)
{
    int customer = -1;
    while(1) {
        pthread_mutex_lock(&mutex);
        if (!waiting_customers) {
            free_barbers++;
            pthread_cond_wait(&no_customers, &mutex);
        }
        pthread_cond_signal(&waiting_room);
        waiting_customers--;
    }
    pthread_mutex_unlock(&mutex);
    printf("barber %d cut hair of customer %d\n", barber, customer);
}

void customer(int customer)
{
    int barber = -1;
    pthread_mutex_lock(&mutex);
    if(waiting_customers == MAX_CUSTOMERS)
        pthread_mutex_unlock(&mutex);
    else {
        if(barberos_libres>0) {
          pthread_cond_signal(&no_customers);
          free_barbers--;
        }
        waiting_customers++;
        pthread_cond_wait(&waiting_room, &mutex);

        pthread_mutex_unlock(&mutex);
        printf("customer %d got hair cut from barber %d\n", customer, barber);
    }
}
```

Puede asumir que dispone de la siguiente estructura de datos:

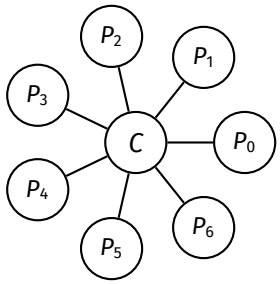
```
typedef struct _queue *queue;     // a FIFO structure

void  insert  (queue q, void *e); // Inserts e into q
void *remove  (queue q);          // removes an element from q
int   elements(queue q);          // returns the number of elements in q
```

Solución

```
struct c_info {
        int customer;
        int barber;
        pthread_cond_t wait;
};

queue q;

void barber(int barber)
{
        int customer = -1;
        struct c_info *element;

        while(1) {
                pthread_mutex_lock(&mutex);

                if (!waiting_customers) {
                        free_barbers++;
                        pthread_cond_wait(&no_customers, &mutex);
                }
                element = remove(q);
                q->barber = barber;
                customer = q->customer;
                pthread_cond_signal(&element->wait);
                waiting_customers--;
                pthread_mutex_unlock(&mutex);
                printf("barber %d cut hair of customer %d\n", barber, customer);
        }
}

void customer(int customer)
{
        int barber = -1;
        struct c_info element;

        pthread_mutex_lock(&mutex);
        if(waiting_customers == MAX_CUSTOMERS)
                pthread_mutex_unlock(&mutex);
        else {
                if(free_barbers > 0) {
                        pthread_cond_signal(&no_customers);
                        free_barbers--;
                }
                waiting_customers++;
                element.customer = customer;
                insert(q, &element);
                pthread_cond_wait(&element.wait, &mutex);
                pthread_mutex_unlock(&mutex);
                printf("barber %d cut hair of customer %d\n", element.barber, customer);
        }
}
```

3. *Comunicación en estrella [1.25 puntos]*

   En un sistema de comunicación en estrella hay un proceso central que se encarga de propagar los mensajes de los nodos situados en el exterior de la estrella.



   El nodo *C* recibe mensajes de los nodos P_i y los copia al resto de nodos de la estrella (es decir, a todos menos al que originó el mensaje). Por ejemplo, si en diagrama de ejemplo el nodo P_0 envía un mensaje debe ser recibido por P_1,...,P_6.

   El siguiente módulo es un esqueleto del código para el nodo central *C*, con dos funciones interfaz:

   a) `start(Pids)`, donde *Pids* es una lista con los Pids de los procesos que forman parte de la estrella (en el ejemplo $[P_0, ..., P_6]$)

   b) `send(Center, Msg)`, donde `Center` es el Pid de *C*, y `Msg` es el mensaje que se desea enviar a los otros procesos de la estrella. Esta función será llamada por los P_i.

```
-module(star).

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

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

send(Center, Msg) ->
    ...

loop(Pids) ->
    receive
        ...
    end.
```

Implemente las funciones `loop` y `send` para que los mensajes enviados por los procesos a través de `send` lleguen a todos los demás procesos de la estrella.

Solución

```
-module(star).

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

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

send(Center, Msg) ->
    Center ! {send, Msg, self()}.          %% Send the message to C. C has to
                                           %%   filter the sender process,
                                           %%   so we include self() in the message

loop(Pids) ->
    receive
        {send, Msg, From} ->               %% C receives a message Msg from From, so
                                           %%   it has to resend it to everyone in Pids
                                           %%   excluding From

            send_to_all(Pids, Msg, From),  %% We do that in send_to_all
            loop(Pids)                     %% and loop to process further requests
    end.

send_to_all([], _, _) ->                   %% Empty List, no more processes to send to
    ok;

send_to_all([From | T], Msg, From) ->      %% Head of the List is the process that sent Msg
    send_to_all(T, Msg, From);             %%    send to the rest of the processes

send_to_all([P | T], Msg, From) ->         %% P is not equal to From,
    P ! Msg,                               %% so send Msg to it,
    send_to_all(T, Msg, From).             %% and send to the rest of the processes
```