Apellidos: **Nombre:**

Concurrencia y Paralelismo

Grado en Ingeniería Informática

Examen Julio 2017

1. **Almacén Robotizado** [2 puntos] Se desea modelar un almacén robotizado en el que hay una serie de cajas (`boxes`) y un brazo robótico que recoge objetos en ellas y los mueve hasta una zona de recogida (`outbox`). Las cajas están dispuestas en linea, y el robot se mueve desde la zona de recogida hasta el final de la linea, donde da la vuelta y retorna hasta la zona de recogida. En cada viaje puede coger objetos de más de una caja, tanto a la ida como a la vuelta. En cada caja hay varias unidades (`in_box`) de un único objeto (descrito en `name`).

   Cuando un usuario desea un producto de una caja lo indica incrementando `requests` por la cantidad que desea. El robot cogerá los productos solicitados (restándolos de `in_box` y sumándolos en `in_robot`), y al llegar a la zona de recogida los depositará (moviendolos de `in_robot` a `in_outbox`), donde el usuario podrá recogerlos.

```
#define NUMBOXES ...

struct box {
    char *name;             // Name of the object
    int in_box;             // Number of objects in the box
    int in_robot;           // Number of objects from this box carried by the robot
    int in_outbox;          // Number of objects from this box in the delivery area
    int requests;           // Number of objects from this box requested by users
    pthread_mutex_t lock;
    pthread_cond_t wait;  // Users wait for the robot to pick up the requested objects.
} boxes[NUMBOXES];

struct order {
    int box;                // Box from which the user is requesting objects
    int quantity;           // Number of objects requested
};

void *robot(void *arg) {
    int direction=1;    // 1 going, -1 coming back
    int current_box=-1; // start at the outbox, boxes numbered from 0 to NUMBOXES-1
    int i;

    while(1) {
        if(current_box==-1) { // At the outbox. Empty robot and change direction
            for(i=0; i<NUMBOXES; i++) {
                pthread_mutex_lock(&boxes[i].lock);
                if(boxes[i].in_robot>0) {
                    boxes[i].in_outbox+=boxes[i].in_robot;
                    boxes[i].in_robot=0;
                    pthread_cond_broadcast(&boxes[i].wait); // Users may pick up objects
                }
                pthread_mutex_unlock(&boxes[i].lock);
            }
            direction=1;
        } else {
            pthread_mutex_lock(&boxes[current_box].lock);
            boxes[current_box].in_robot+=boxes[current_box].requests;
            boxes[current_box].in_box-=boxes[current_box].requests;
            boxes[current_box].requests=0;
            pthread_mutex_unlock(&boxes[current_box].lock);

            if(current_box==NUMBOXES-1) direction=-1; // At the last box. Reverse direction
        }
        current_box += direction;
    }
}

void *user(void *arg) {
    struct order *order=arg;
    ...
}
```

Implemente `user` de forma que:

- El usuario compruebe si hay suficientes objetos en el la caja para atender la petición. Si no los hay se marcha sin solicitar nada.

- Si los hay los solicita incrementando `requests` y esperando a que el robot los lleve hasta la zona de recogida.
- Por último los retira descóntandolos de `in_outbox`.
- Puede haber múltiples usuarios haciendo peticiones al mismo tiempo.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(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_unlock(pthread_mutex_t *mutex);
```

Solución

```
void *user(void *arg) {
    struct order *order=arg;

    pthread_mutex_lock(&boxes[order->box].lock);
    if(order->quantity + boxes[order->box].requests > boxes[order->box].in_box) {
        pthread_mutex_unlock(&boxes[order->box].lock);
        return NULL;
    }

    boxes[order->box].requests+=order->quantity;

    while(boxes[order->box].in_outbox<order->quantity)
        pthread_cond_wait(boxes[order->box].wait, lock);

    boxes[order->box].in_outbox-=order->quantity;

    pthread_mutex_unlock(&boxes[order->box].lock);
    return NULL;
}
```

---

2. **Consumidores con Especificaciones** [2 puntos] En el problema de productores/consumidores es posible que a los consumidores no les sirva cualquier producto del buffer, sino que necesiten productos que cumplan una serie de condiciones. Se pide modificar el problema para esta situación. Se puede asumir que las siguientes funciones ya están implementadas:

   - El tipo `specification` representa una especificación de las condiciones que debe cumplir un producto para que un consumidor pueda utilizarlo. El consumidor recibe un `specification` como parámetro dentro de `consumer_info`, y debe utilizar productos para los que la función `satisfies` se cumpla.
   - `insert(e)` inserta el elemento e en el buffer.
   - `buffer_is_full()` comprueba si el buffer está lleno.
   - El tipo `position` junto con las funciones `first`, `next` y `is_end` permite recorrer todos los elementos del buffer.
   - `get_element` devuelve el elemento que está en una posición (sin quitarlo del buffer), y `remove_element` lo elimina.

```
typedef ... specification;
bool satisfies(element e, specification sp);

struct consumer_info {
    specification sp;
};

void insert(element e);
bool buffer_is_full();

element get_element(position p);
void remove_element(position p);

position first(); // Get the position of the first element in the buffer
void next(position p); // Advance p
bool is_end(position p); // check if p is at the end of the buffer.

pthread_mutex_t lock;
pthread_cond_t buffer_full, no_valid_element;

void *producer (void *arg) {
    while(1) {
        element e=create_element();

        pthread_mutex_lock(&lock);
        while(buffer_is_full(b))
            pthread_cond_wait(&buffer_full, &lock);

        insert_element(e);
        pthread_cond_broadcast(no_valid_element);

        pthread_mutex_unlock(&lock);
    }
}

void *consumer (void *arg) {
    struct consumer_info *info=arg;
    position p;
    element e;

    while(1) {
        pthread_mutex_lock(&lock);
        ...
        pthread_mutex_unlock(&lock);
        process(e);
    }
}
```

Implemente el consumidor de forma que espere hasta que haya algún elemento que cumpla su especificación y lo procese.

Solución

```
void *consumer (void *arg) {
    struct consumer_info *info=arg;
    position p;
    element e;

    while(1) {
        pthread_mutex_lock(&lock);

        while(1) {
            for(p=first(); !is_end(p); next(p)) {
                if(satisfies(get_element(p), info->sp)) break;
            }
            if(!is_end(p)) break;
            else pthread_cond_wait(no_valid_element, lock);
        }

        if(buffer_is_full())
            pthread_cond_broadcast(&buffer_full);

        e=remove_element(p);

        pthread_mutex_unlock(&lock);

        process(e);
    }
}
```

---

3. **Gestor de Alarmas** [1 punto] Una alarma es un mensaje que un proceso solicita recibir pasado un cierto tiempo. Una forma sencilla de implementar una alarma en Erlang es crear un proceso para que espere el tiempo necesario y haga el envío.

```
1> alarm:set_alarm(Alarm_Manager, wake_up, 100). %Receive the msg "wake_up" after 100 ms
```

Se pide implementar un sistema de gestión de alarmas que proporcione la siguiente interfaz:

- `start`, que arranca el gestor de alarmas y devuelve su PID.
- `set_alarm/3`, que crea una alarma para envíar un mensaje tras un cierto número de milisegundos. Debe devolver un identificador para la alarma. La implementación del identificador se puede hacer de varias formas (con un PID, un número, ...)
- `cancel_alarm/2`, que recibe un identificador de una alarma, y la cancela.

El gestor debe controlar qué alarmas están configuradas. También debería cancelar automáticamente las alarmas de los procesos que mueran antes de que las alarmas salten.

Para crear y cancelar las alarmas se proporcionan dos funciones:

- `send_after`, que crea un proceso para enviar un mensaje a un destino al cabo de un cierto tiempo. Devuelve el PID del proceso creado.
- `stop_timer`, que mata un proceso dado su PID.

```
-module(alarm).

-export([start/..., init/..., set_alarm/3, cancel_alarm/2, timer/3]).

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

set_alarm(Alarm_Manager, Msg, Time) ->
    ...

cancel_alarm(Alarm_Manager, Alarm) ->
    ...

init(...) ->
    process_flag(trap_exit, true),
    loop(...).

loop(...) ->
    receive
        {'EXIT', Pid, Reason} ->
            ...
    end.

stop_timer(Pid) ->
    exit(Pid, cancel_timer).

send_after(Msg, Dst, Time) ->
    spawn(?MODULE, timer, [Msg, Dst, Time]).

timer(Msg, Dst, Time) ->
    receive
    after Time ->
        Dst ! Msg
    end.
```

Solución

```
-module(alarm).

-export([start/0, init/0, set_alarm/3, cancel_alarm/2, timer/3]).

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

set_alarm(Alarm_Manager, Msg, Time) ->
    Alarm_Manager ! {set_alarm, Msg, Time, self()},
```

```
    receive
        {alarm, Id} -> Id
    end.

cancel_alarm(Alarm_Manager, Alarm) ->
    Manager ! {cancel_alarm, Alarm}.

init() ->
    process_flag(trap_exit, true),
    loop([]).

loop(Alarms) ->
    receive
        {'EXIT', Pid, Reason} ->
            cancel(Pid, Alarms),
            loop([{Owner_Pid, Alarm_Pid} || {Owner_Pid, Alarm_Pid} <- Alarms, Pid/=Owner_Pid]);
        {set_alarm, Msg, Time, Pid} ->
            link(Pid),
            Alarm_Pid = send_after(Msg, Pid, Time),
            From ! {alarm, Alarm_Pid},
            loop([{Pid, Alarm_Pid}| Alarms]);
        {cancel_alarm, Alarm} ->
            stop_timer(Alarm),
            loop([{Pid, Alarm_Pid} || {Pid, Alarm_Pid} <- Alarms, Alarm_Pid /= Alarm])
    end.

cancel(Pid, Alarms) -> lists:foreach(fun ({P,A}) -> if P == Pid -> stop_timer(A); _ -> ok end, Alarms).

cancel(_, []) -> ok;
cancel(Pid, [{Pid, Alarm} | T])  -> stop_timer(Alarm), cancel(T);
cancel(Pid, [_|T]) -> cancel(T).
```