Apellidos: **Nombre:**

Concurrencia e Paralelismo

Grado en Ingeniería Informática

Exame Julio 2018

1. **Librería de semáforos usando mútex** [2 puntos]

   Implemente una librería de semáforos utilizando los mutex y condiciones de la librería pthread. Las operaciones a implementar son:

   - `sem_init(sem s, int value)`, que inicializa un semáforo con valor `value`.
   - `sem_p(sem s)`, que decrementa el valor del semáforo si es mayor que 0, y espera en caso contrario.
   - `sem_v(sem s)`, que despierta a un thread si el semáforo vale 0 y hay threads esperando, o incrementa el valor del semáforo en caso contrario.

   No utilice esperas activas. Intente minimizar el número de threads que se despiertan.

```
typedef struct {
    ...
} *sem;

void sem_init(sem s, int value) {
    ...
}

void sem_p(sem s) {
    ...
}

void sem_v(sem s) {
    ...
}
```

```
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_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Solución

Vamos a necesitar un contador para el valor del semáforo, y un mutex y una condición para proteger ese contador, y esperar en la operación `sem_p` cuando el contador valga 0.

Queremos también minimizar las veces que despertamos a threads, así que vamos a utilizar `pthread_cond_signal`, y un contador para el número de threads que están esperando en `sem_p`.

```
typedef struct {
    int value;
    int waiting_threads;
    pthread_mutex_t lock;
    pthread_cont_t wait;
} *sem;
```

La inicialización consiste en fijar el valor del semáforo, poner el número de threads esperando a 0, e inicializar el mutex y las condiciones.

```
void sem_init(sem s, int value) {
    s->value=value;
    s->waiting_threads=0;
    pthread_mutex_init(&s->lock, NULL);
    pthread_cond_init(&s->wait, NULL);
}
```

En `sem_p` bloqueamos el mutex para comprobar si el valor del semáforo es mayor que 0. Si lo es simplemente se resta 1 y se desbloquea. Si es igual a 0, hay que esperar, y se incrementa el contador de threads en espera antes de llamar a `pthread_cond_wait`. Una vez despierto, hemos optado por no decrementar el contador. Esto hay que tenerlo en cuenta al escribir `sem_v`.

En todo momento se ha mantenido el mutex bloqueado desde la comprobación de `s->value` hasta que el thread duerme.

```
void sem_p(sem s) {
    pthread_mutex_lock(&s->lock);
    if(s->value<=0) {
        s->waiting_threads++;
        pthread_cond_wait(&s->wait, &s->lock);
    } else
        s->value--;
    pthread_mutex_unlock(&s->lock);
}
```

La función `sem_v` tiene que seguir lo que ha hecho `sem_p`. Tenemos un contador que marca si hay threads esperando, por lo tanto lo primero es comprobar si hay alguno. Si lo hay, simplemente se decrementa ese contador y se despierta a uno de ellos llamando a `pthread_cond_signal`. Si no hay threads esperando simplemente se incrementa el valor del semáforo.

```
void sem_v(sem s) {
    pthread_mutex_lock(&s->lock);
    if(s->waiting_threads) {
        s->waiting_threads--;
        pthread_cond_signal(&s->wait);
    } else
        s->value++;
    pthread_mutex_unlock(&s->lock);
}
```

La solución completa queda:

```
typedef struct {
    int value;
    int waiting_threads;
    pthread_mutex_t lock;
    pthread_cont_t wait;
} *sem;

void sem_init(sem s, int value) {
    s->value=value;
    s->waiting_threads=0;
    pthread_mutex_init(&s->lock, NULL);
    pthread_cond_init(&s->wait, NULL);
}

void sem_p(sem s) {
    pthread_mutex_lock(&s->lock);
    if(s->value<=0) {
        s->waiting_threads++;
        pthread_cond_wait(&s->wait, &s->lock);
    } else
        s->value--;
    pthread_mutex_unlock(&s->lock);
}

void sem_v(sem s) {
    pthread_mutex_lock(&s->lock);
    if(s->waiting_threads) {
        s->waiting_threads--;
        pthread_cond_signal(&s->wait);
    } else
        s->value++;
    pthread_mutex_unlock(&s->lock);
}
```

---

2. **Lista doblemente enlazada** [1.75 puntos]

   Tenemos una lista circular doblemente enlazada. Podemos suponer:

   - La lista siempre tiene al menos un elemento.
   - Que tanto `pos` como *elem* son distintos de `NULL`
   - La operación de asignar punteros es atómica.

   Se pide modificar la implementación de `remove()` e `insert()` para que se cumpla:

   - La lista tiene que funcionar con varios threads recorriendola a la vez.
   - Un thread solo recorre la lista en uno de los sentidos. Si esta sucediendo una inserción puede ver (o no) el nuevo elemento, pero tiene que encontrarse en todo momento la lista circular (i.e. no puede encontrarse un `NULL`). Lo mismo con un borrado, puede ver (o no) el elemento borrado.
   - Para conseguir esto hay que escribir los punteros en el orden adecuado para que la lista este siempre conexa. Utilice un mútex por elemento de la lista. Puede suponerse que nunca se va a borrar dos veces el mismo elemento.
   - Pista: Pensar que sucede cuando se asigna el puntero NULL.

```
struct list {
  struct list *prev;
  struct list *next;
  ...
};

void remove(list *elem)
{
    if (elem->next) {
      elem->next->prev = elem->prev;
    }
    if (elem->prev) {
      elem->prev->next = elem->next;
    }
    elem->prev = NULL;
    elem->next = NULL;
  }

void insert(list *pos, list *elem)
{
    if (pos->next) {
      pos->next->prev = elem;
    }
    pos->next = elem;
    elem->prev = pos;
    elem->next = pos->next;
}
```

Solución

Necesitamos usar un `mutex` por elemento. Todos los elementos que vayamos a utilizar en un borrado o inserción necesitamos bloquearlos primero. Para que no pueda haber interbloqueos tenemos que elegir un orden. En este caso tenemos dos triviales, seguir `next` o `prev`. Elegimos `next`.

```
struct list {
  struct list *prev;
  struct list *next;
  pthread_mutex_t m;
  ...
};

void remove(list *elem)
{
    if (elem->prev) {
      pthread_mutex_lock(&elem->prev->m);
    }
```

```
    pthread_mutex_lock(&elem->m);
    }
    if (elem->next) {
      pthread_mutex_lock(&elem->next->m);
    }
    if (elem->next) {
      elem->next->prev = elem->prev;
    }
    if (elem->prev) {
      elem->prev->next = elem->next;
    }
    elem->prev = NULL;
    elem->next = NULL;
    if (elem->prev) {
      pthread_mutex_unlock(&elem->prev->m);
    }
    pthread_mutex_unlock(&elem->m);
    }
    if (elem->next) {
      pthread_mutex_unlock(&elem->next->m);
    }
  }

void insert(list *pos, list *elem)
{
    pthread_mutex_lock(&pos->m);
    if (pos->next) {
      pthread_mutex_lock(&pos->m);
    }
    if (pos->next) {
      pos->next->prev = elem;
    }
    pos->next = elem;
    elem->prev = pos;
    elem->next = pos->next;
    pthread_mutex_unlock(&pos->m);
    if (pos->next) {
      pthread_mutex_unlock(&pos->m);
    }
}
```

Ahora que tenemos los bloqueos correctamente, vamos a mirar que sucede cuando hacemos un borrado de un elemento. Necesitamos que cualquier thread que estuviese en ese momento en el elemento no se encuentre que desparece de la cadena. La solución es borrar las líneas que ponen `prev` y `next` a `NULL`. De esta forma siempre será capaz de continuar.

En el caso de insertar un elemento, lo que tenemos que conseguir es que el elemento nuevo este conectado es asignar los punteros del nuevo elemento antes de cambiar los punteros de los elementos ya existentes.

```
struct list {
  struct list *prev;
  struct list *next;
  pthread_mutex_t m;
  ...
};

void remove(list *elem)
{
    if (elem->prev) {
      pthread_mutex_lock(&elem->prev->m);
    }
    pthread_mutex_lock(&elem->m);
    }
    if (elem->next) {
      pthread_mutex_lock(&elem->next->m);
    }
    if (elem->next) {
      elem->next->prev = elem->prev;
    }
    if (elem->prev) {
      elem->prev->next = elem->next;
    }
    if (elem->prev) {
      pthread_mutex_unlock(&elem->prev->m);
    }
```

```
    pthread_mutex_unlock(&elem->m);
    }
    if (elem->next) {
      pthread_mutex_unlock(&elem->next->m);
    }
  }

void insert(list *pos, list *elem)
{
    pthread_mutex_lock(&pos->m);
    if (pos->next) {
      pthread_mutex_lock(&pos->m);
    }
    elem->prev = pos;
    elem->next = pos->next;
    if (pos->next) {
      pos->next->prev = elem;
    }
    pos->next = elem;
    pthread_mutex_unlock(&pos->m);
    if (pos->next) {
      pthread_mutex_unlock(&pos->m);
    }
}
```

---

3. **Anillo 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).

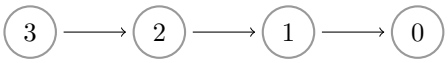
   La cadena se crea llamando a `start(N)`, donde `N` es el número del primer proceso. Esta función devuelve el PID del primer proceso de la cadena. La función `send(Chain, Msg)` envía un mensaje desde el primer elemento al último.

   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: 0 steps to go...
```

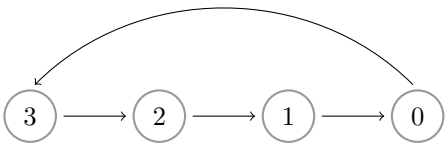
---

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: 0 steps to go...~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 la estructura creada sea un anillo:



Cambie la operación send a `send(Chain, Msg, Nodes)`, que envía el mensaje alrededor del anillo cruzando `Nodes` nodos. Un ejemplo de ejecución tras los cambios es:

```
1> C = chain:start(3).
<0.62.0>
2> chain:send(C, hola, 5).
hola: at node 3 with 5 steps to go...
hola: at node 2 with 4 steps to go...
hola: at node 1 with 3 steps to go...
```

```
hola: at node 0 with 2 steps to go...
hola: at node 3 with 1 steps to go...
hola: at node 2 with 0 steps to go...
```

Solución

Con el cambio de estructura a un anillo todos los nodos van a tener el mismo comportamiento: tienen un nodo siguiente, y cuando reciban un mensaje tienen que imprimirlo, y si el número de nodos que el mensaje tiene que recorrer es mayor que 0, lo reenvían al nodo siguiente. Por tanto un único loop va a ser suficiente y no hay que diferenciar al último como hasta ahora.

La inicialización es más compleja. Necesitamos que el último proceso creado (el 0) tenga como siguiente al primero. Para eso podemos ir pasando el PID del primer proceso en las funciones de inicialización, para que al llegar al último éste lo pueda usar como siguiente. El primero por tanto tiene que pasar su PID en la inicialización, y el resto simplemente lo van a reenviar. La función init del primer proceso tiene que ser distinta, y `start/1` tiene que crear el proceso ahí:

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

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

Si el número del último proceso es 0, solo hay que crear 1, por lo que `init_first` simplemente llama a `proc_loop` pasandose a sí mismo como siguiente. Tendríamos por tanto un anillo con un proceso, donde ese proceso se reenvía los mensajes a sí mismo.

Si hay más, el primer proceso crea al siguiente pasándole su PID como parámetro. Ese PID se va a arrastrar hasta el último proceso para usar como siguiente del último.

Para el resto simplemente se reenvía el PID del primero si el número de proceso el mayor que 0, y se utiliza el primero como siguiente cuando es 0:

```
init(First, 0) ->
    proc_loop(First, 0);
init(First, N) ->
    Next = spawn(?MODULE, init, [First, N-1]),
    proc_loop(Next, N).
```

Con eso las funciones init crean una estructura de anillo, donde todos los procesos corren en `proc_loop`, y guardan su número y el PID del siguiente proceso. Ahora hay que hacer que los mensajes se reenvíen un cierto número de nodos. Lo primero es modificar la función `send` para que reciba el número de nodos que el mensaje tiene que recorrer, y lo pase al anillo:

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

Y ahora cambiar el procesamiento de {`send, Msg, Nodes`} en `proc_loop` para que imprima el nuevo mensaje y reenvíe al siguiente si `Nodes` es mayor que 0:

```
proc_loop(Next, N) ->
    receive
        {send, Msg, Nodes} ->
            io:format("~w: at node ~w with ~w steps to go...~n", [Msg, N, Nodes]),
            if
                Nodes > 0 -> Next ! {send, Msg, Nodes-1};
                true -> ok
            end
    end,
    proc_loop(Next, N).
```

El módulo completo quedaría así:

```
-module(chain).

-export([start/1, init/2, init_first/1, send/3]).

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

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

init_first(0) ->
    proc_loop(self(), 0);
init_first(N) ->
    Next = spawn(?MODULE, init, [self(), N-1]),
    proc_loop(Next, N).

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

proc_loop(Next, N) ->
    receive
        {send, Msg, Nodes} ->
            io:format("~w: at node ~w with ~w steps to go...~n", [Msg, N, Nodes]),
            if
                Nodes > 0 -> Next ! {send, Msg, Nodes-1};
                true -> ok
            end
    end,
    proc_loop(Next, N).
```
