Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Julio 2014

1. **Filósofos con más tenedores** [2 puntos]

   Un grupo de filósofos se sienta a cenar, y con sus tradicionales problemas para conseguir dos cubiertos por persona deciden seguir el siguiente algoritmo para poder comer:

```
#define N ... // Numero de filosofos
#define RIGHT(x) (((x)+1) %N)
#define LEFT (x) (((x)==0) (N-1):((x)-1)))
pthread_cond_t waiting[N];
pthread_mutex_t mutex;
int state[N];

void filosofo(int i) {
        while(1) {
                think();
                pickup(i);
                eat();
                put_down(i);
        }
}

void pickup(int i) {
        pthread_mutex_lock(&mutex);
        state[i] =HUNGRY;
        while (state[LEFT(i)] ==EATING || state[RIGHT(i) ==EATING)
                pthread_cond_wait(&waiting[i], &mutex);
        state[i] =EATING;
        pthread_mutex_unlock(&mutex);
}

void put_down(int i) {
        pthread_mutex_lock(&mutex);
        state[i]=THINKING;
        if(state[LEFT(i)] ==HUNGRY)
        pthread_cond_signal(&waiting[LEFT(i)]);
        if(state[RIGHT(i)] ==HUNGRY)
                pthread_cond_signal(&waiting[RIGHT(i)]);
        pthread_mutex_unlock(&mutex);
}
```

   La sociedad para la prevención de la inanición de los filósofos decide solucionar el problema de escasez de cubiertos y realiza una campaña de captación de fondos para comprar más tenedores. Una vez comprados se observa que no llegan para que cada filósofo tenga dos. Para optimizar el uso de los nuevos tenedores se decide mantener la distribución de un cubierto entre cada dos filosofos, y dejar los nuevos en el medio de la mesa para que los pueda usar cualquier filósofo que los necesite. Para controlar el acceso a estos nuevos cubiertos se establecen las siguientes reglas:

   a) Los filósofos deben intentar usar primero los cubiertos que comparten con sus filósofos vecinos.

   b) Si alguno de esos está ocupado, pueden intentar usar los que están en el medio de la mesa si están libres.

   c) Como buenos compañeros, si al terminar de comer dejan algún cubierto en el medio de la mesa deberían avisar a los filósofos que estén esperando para comer para que puedan usarlos.

   Modificar el algoritmo inicial para que contemple los cambios.

```
#define M ... // Numero de tenedores extra
```

2. **Implementar un buffer multithread** [2 puntos]

   Partimos de la implementación de un buffer que funciona para un solo thread. Realiza buffering de los bloques de disco que ya se han leido/escrito. El ejercicio consiste en cambiar el código para que funcione con multiples threads. Cosas a tener en cuenta:

   - Asumase que `lock` y `cond` están inicializados correctamente.
   - Se puede bloquear como máximo un `thread` a la vez.
   - No se puede llamar a `block_read()` ni `block_write()` con el `lock` cogido.
   - Usese `in_transit` para indicar que se está realizando una lectura/escritura en ese buffer.

```
struct buf {
        bool dirty;
        unsigned int num;
        char buf[BLOCK_SIZE];
        pthread_mutex_t lock;
        pthread_cond_t cond;
        bool in_transit;
}

struct buf buffer[BUFFER_SIZE];


int write_data(char *buf, unsigned int block_num)
{
        int i = hash(block_num);
        int res;

        if (buffer[i].dirty && buffer[i].num != block_num) {
                res = block_write(buffer[i].buf, buffer[i].num);
                if (res) {
                        return res;
                }
                buffer[i].num = block_num;
        }
        buffer[i].dirty = true;
        memcpy(buffer[i].buf, buf, BLOCK_SIZE);
        return 0;
}

int read_data(char *buf, unsigned int block_num)
{
        int i = hash(block_num)

        if (buffer[i].num != block_num) {
                if (buffer[i].dirty) {
                        res = block_write(buffer[i].buf, buffer[i].num);
                        if (res) {
                                return res;
                        }
                        buffer[i].dirty = false;
                }
                res = block_read(buffer[i].buf, block_num);
                if (res) {
                        buffer[i].num = BLOCK_ERROR;
                        return res;
                }
                buffer[i].num = block_num;
        }

        memcpy(buf, buffer[i].buf, BLOCK_SIZE);
        return 0;
}

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

3. **Servidor de Cifrado en Erlang** [1 punto]

Disponemos de un sistema que puede enviar información cifrada con un algoritmo de clave pública/clave privada a múltiples destinos. Como parte de este sistema hay un proceso servidor que se encarga de almacenar las claves públicas de los distintos destinos y cifrar los mensajes. La implementación de este servidor es la siguiente:

```
-module(crypt).

-export([start/0, init/0, add_key/2, encrypt/2]).

start() ->
        Pid = spawn(?MODULE, init, []),
        register(crypt, Pid).

init() ->
        loop([]).

add_key(Dst, Key) ->
        crypt ! {add_key, Dst, Key}.

encrypt(Dst, Msg) ->
        crypt ! {encrypt, Dst, Msg, self()},
        receive
                {encrypt, Reply} -> Reply
        end.

loop(Keys) ->
        receive
                {add_key, Dst, Key} ->
                        loop([{Dst, Key} | [{X,Y} || {X,Y} <- Keys, X /= Dst]]);
                {encrypt, Dst, Msg, From} ->
                        case lookup(Dst, Keys) of
                                {ok, Key} ->
                                        N_Msg = encoder:code(Key, Msg), %encoder:code cifra Msg usando la clave Key
                                        From ! {encrypt, {ok, N_Msg}};
                                {error, not_found} ->
                                        From ! {encrypt, no_key}
                        end,
                        loop(Keys)
        end.

lookup(Dst, []) -> {error, not_found};
lookup(Dst, [{Dst, Key} | _]) -> {ok, Key};
lookup(Dst, [_ | T]) -> lookup(Dst, T).
```

Este módulo usa un único proceso para codificar los mensajes, por lo que solo puede usar un procesador. Cuando el sistema tiene mucha carga este proceso se convierte en un cuello de botella. Para solucionarlo, se pide que el módulo arranque un proceso auxiliar para realizar el cifrado cada vez que se reciba una petición `encrypt`, es decir, la ejecución de `encoder:code/2` debe poder hacerse simultáneamente para varios mensajes. Implemente los cambios necesarios.