Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Junio 2015

1. **Proteger un programa que comprime páginas con varios threads** [1.75 puntos]

   Tenemos un programa que realiza el siguiente código:

```
while (true){
    char *buffer = compress(page);
    sent = qemu_put_buffer(f, buffer);
}
```

   El problema con este código es que es muy lento debido a la compresión. Una posible solución es que la compresión la realicen varios threads a la vez, cada uno una página distinta. Para ello vamos a usar el código de ejemplo que aparece a continuación.

   El código no tiene ninguna protección para concurrencia, y hay comentarios que indican donde se debe esperar y donde se deben señalar que una operación ya ha terminado. El ejercicio consiste en proteger las estructuras de datos para acceso multithread.

   - `quit_comp_thread`: significa que ya hemos terminado, el thread debe salir. Es una variable global que se pone a `true` en el thread principal.
   - `start`: se usa para que el thread principal le indique al thread de compresión que tiene trabajo que hacer.
   - `done`: lo usa el thread de compresión para decirle al thread principal que ha terminado.
   - `buffer`: El buffer comprimido de la petición previa.
   - `page`: la página que hay que comprimir.
   - `compress_thread`: Código que ejecuta cada uno de los threads compresores.
   - `start_compression`: función que usa el thread principal para indicarle a un thread compresor concreto que página tiene que comprimir.
   - `compress_page`: La función que llama el bucle principal para comprimir una página. Devuelve la página anterior que estaba comprimiendo el thread que se elige.
   - Cuando llega una página nueva para comprimir, primero buscamos un thread que este libre. Si no hay ninguno libre, esperamos a que se libere uno. Cuando tenemos uno libre, copiamos la página anterior que ha comprimido y comenzamos la compresión de la página actual.

```
struct CompressParam {
    bool start;
    bool done;
    char *buffer;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int page;
} copm_param[THREADS];

bool quit_comp_thread;
pthread_mutex mutex;
pthread_cond cond;

static void *compress_thread(void *opaque)
{
    struct CompressParam *param = opaque;

    while (!quit_comp_thread) {
        while (!param->start && !quit_comp_thread) {
                // Wait for work to do
        }
        if (!quit_comp_thread) {
            param->buffer = compress(param->page);
        }
        param->start = false;
        param->done = true;

        // tell compress_page() that we have finished
    }

    return NULL;
}

static inline void start_compression(struct CompressParam *param)
{
    param->done = false;
    param->start = true;
    // Tell compress thread that there are work to do
}

static int compress_page(QEMUFile *f, int page)
{
    int i;
    bool sent = false;

    while (true) {
        for (i = 0; i < THREADS; i++) {
            if (comp_param[i].done) {
                if (comp_param[i].buffer) {
                    sent = qemu_put_buffer(f, comp_param[i].buffer);
                    comp_param[i].buffer = NULL;
                }
                param->page = page;
                start_compression(&comp_param[i]);
                break;
            }
        }
        if (sent) {
            break;
        } else {
                // Wait for a compress_thread to finish
        }
    }
    return sent;
}
```

2. **Intercambiador** [1.75 puntos]

En un sistema existe una zona compartida con varias zonas de información que se leen y se escriben concurrentemente. Los procesos leen y escriben en una zona que se determina de forma aleatoria cada vez que se produce un acceso. El código del sistema es el siguiente:

```
#define ZONAS ...
pthread_mutex_t lock[ZONAS];
pthread_cond_t espera[ZONAS];

int lectores[ZONAS];
int en_uso[ZONAS];

typedef datos ...
datos leer(int zona);
void escribir(datos d, int zona);

void leer() {
        int zona;
        datos d;
        zona=rand() % ZONAS;

        pthread_mutex_lock(&lock[zona]);
        while(en_uso[zona] && lectores[zona]==0) pthread_cond_wait(&espera[zona], &lock[zona]);
        lectores[zona]++;
        if(lectores[zona]==1) en_uso=1;
        pthread_mutex_unlock(&lock[zona]);

        d = leer(zona);

        pthread_mutex_lock(&lock[zona]);
        lectores[zona]--;
        if(lectores[zona]==0) {
                en_uso[zona]=0;
                pthread_cond_broadcast(&espera[zona]);
        }
        pthread_mutex_unlock(&lock[zona]);
}

void escritor(datos d) {
        int zona;
        zona = rand() % ZONAS;

        pthread_mutex_lock(&lock[zona]);
        while(en_uso[zona]) pthread_cond_wait(&espera[zona], &lock[zona]);
        en_uso[zona]=1;
        pthread_mutex_unlock(&lock[zona]);

        escribir(d, zona);

        pthread_mutex_lock(&lock[zona]);
        en_uso[zona]=0;
        pthread_cond_broadcast(&espera[zona], &lock[zona]);
        pthread_mutex_unlock(&lock[zona]);
}
```

Se quiere que el sistema pueda intercambiar la información de dos de las zonas de forma atómica, es decir, que cualquier thread que acceda a las zonas compartidas vea o bien las dos zonas tal y como estaban antes de cambiar, o tal y como quedan después del cambio.

Implemente la función `void swap(int i, int j)` que intercambie los contenidos de las zonas i y j. Para ello téngase en cuenta:

- El cambio debe ser atómico.
- Si alguna de las zonas está ocupada se debe esperar a que esa zona se libere, pero sin bloquear el acceso a ninguna de las dos mientras se espera.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(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_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

3. **Diccionario Concurrente** [1.5 puntos]

Un diccionario es un servicio que permite almacenar y recuperar pares clave/valor. En un sistema donde existe un diccionario se observa que hay un número alto de consultas simultaneas y supone un cuello de botella. Para mejorar el rendimiento se decide implementar un diccionario concurrente que mantenga una copia completa del diccionario en varios procesos. De esa forma se puede realizar consultas contra cualquiera de ellos y se puede repartir la carga.

Cuando se realizan actualizaciones es necesario actualizar todas las copias del diccionario, por lo que cada uno de los procesos guarda una lista con todos los procesos que forman parte del diccionario.

```
-module(dict).

-export([start/1, init_store/0, get/2, put/3, sync_put/3]).

%%% Arranque del sistema
start(N) -> % arranca N diccionarios
        L = start_stores(N), % arrancar diccionarios
        send_store_list(L, L), % enviar copia de la lista de diccionarios
        loop(L).

start_stores(0) -> [];
start_stores(N) -> [spawn(?MODULE, init_store, []) | start_stores(N-1)].

send_store_list([], _) -> ok;
send_store_list([S|T], L) -> S ! {store_list, L}, send_store_list(T,L).

init_store() ->
        receive
                {store_list, L} -> loop_store([], L)
        end.

%%% API
get(K, S) ->
        S ! {get, K, self()},
        receive
                {get_reply, R} -> R
        end.

put(K, V, S) -> ...

sync_put(K, V, S) -> ...

%%% Loop y auxiliares
loop_store(Dic, L) ->
        receive
                {get, K, From} ->
                        R = lookup(K, Dic),
                        From ! {get_reply, R},
                        loop(Dic, L)
        end.

lookup(_, []) -> not_found;
lookup(K, [{K, V} | _]) -> {ok, V};
lookup(K, [_|T]) -> lookup(K, T).

store(K, V, []) -> [{K,V}];
store(K, V, [{K,_} | T]) -> [{K,V} | T];
store(K, V, [{K2, V2}|T]) -> [{K2, V2} | store(K, V, T)].
```

Se pide implementar:

(a) La función `put(K,V,S)`, que dada una clave `K`, un valor `V` y un proceso `S` almacena el par `{K,V}` en el sistema. El proceso S debe encargarse de que los otros procesos almacenen el par `{K, V}`.

(b) La función `sync_put(K,V,S)`, que deberá hacer lo mismo, pero además garantizando que cuando se vuelva de la llamada a `sync_put` todos los procesos del diccionario tienen almacenado el par `{K,V}`