Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Mayo 2016

1. **Acceso de Grupos a un Museo** [2 puntos]

   Se quiere diseñar un sistema para controlar la entrada de grupos a un museo con capacidad limitada (`capacity`). Los grupos tienen que gestionar la entrada y salida de forma conjunta, es decir:

   - Todos los miembros tienen que entrar al mismo tiempo.
   - En caso de que no haya espacio, todos los miembros del grupo esperarán hasta que lo haya, aunque hubiera plazas para que una parte del grupo entrase.

   Cada miembro del grupo ejecuta la función `visitor`, y recibe como parámetro una estructura `group` que comparte con todos los demás miembros del mismo grupo.

   Añada el código necesario para gestionar la entrada y salida de grupos a la función `visitor`. Intente minimizar el número de veces que se despierta a los procesos en espera.

```
struct group {
    const int members; // total number of members in the group
    pthread_mutex_t *counter_m;
    int counter;  // members currently inside the museum
};

int capacity;
pthread_mutex_t lock;
pthread_cond_t no_space;

void *visitor(void *arg) {
    struct group *grp = arg;

    // Gestion de acceso

    visit();

    // Gestion de salida

}

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);
```

Solución

```
struct group {
    const int members; // Total Number of members in the group
    pthread_mutex_t *counter_m;
    int counter;  // Members active
    pthread_cond_t group_entry;
};

int capacity;  // Number of available places
pthread_mutex_t lock;
pthread_cond_t no_space;

void *visitor(void *arg) {
    struct group *grp = arg;

    pthread_mutex_lock(grp->counterm);
    if(counter==0) {
        pthread_mutex_lock(&lock);
        while(free_places<grp->members)
            pthread_cond_wait(&no_space, &lock);
        free_places -= grp->members;
        pthread_mutex_unlock(&lock);
    }
    grp->counter++;
    if(grp->counter==grp->members)
        pthread_cond_broadcast(grp->group_entry);
    else
        pthread_cond_wait(grp->group_entry, grp->counterm);
    pthread_mutex_unlock(grp->counterm);

    visit();

    pthread_mutex_lock(grp->counterm);
    grp->counter--;
    if(counter==0)
        pthread_mutex_lock(&lock);
        free_places += grp->members;
        pthread_cond_broadcast(&no_space);
        pthread_mutex_unlock(&lock);
    }
    pthread_mutex_unlock(grp->counterm);
}
```

2. **Colas de prioridad de ejecución** [2 puntos]

   Hay un sistema que tiene **NUM_COLAS** colas de ejecución, cada una indica una prioridad, siendo la cero la que tiene más prioridad.

   - Suponganse que `queue_add()`, `queue_remove()`, y `queue_get()`, están implementadas.
   - `queue_add()` añade un elemento por el final a una cola dada.
   - `queue_remove()` borra el primer elemento de una cola dada.
   - `queue_get()` devuelve el primer elemento de una cola dada.
   - `time_mark()` asigna al trabajo en cuestión la hora actual.
   - `time_wait_longer()` indica si el trabajo lleva en la cola más de un segundo.

```
struct job {
  ...
  time_t time;
  int prio;
};
void time_mark(struct job *);
bool time_wait_longer(struct job *);
struct queue {
  ...
};
struct queues {
  struct queue[NUMQUEUES];
} queues;
void queue_add(struct queue*, struct job*);
void queue_remove(struct queue*);
struct job* queue_get(struct queue*);

int queue_job(struct job *job, unsigned int prio)
{
  if (prio >= NUMQUEUES) {
    return -1;
  }
  time_mark(job);
  queue_add(queue[prio], job);
}

void run(void)
{
  int i;
  while(true) {
    for (i = 0; i < NUMQUEUES; i++) {
      struct job * job = queue_get(queues.queues[i]);
      if (job) {
        queue_remove(queues.queues[i]);
        execute(job);
        queue_add(queues.queues[i], job);
        break;
    }
  }
}
```

   Se pide:

   - Proteja el acceso a las colas. Permita la máxima concurrencia.
   - Crear un thread que duerme por un segundo, y cualquier trabajo que lleve más de 5 segundos esperando lo pasa a la cola de prioridad anterior a la que ocupa.
   - Una vez que se ejecuta el trabajo, vuelve a la cola de prioridad inicial (`prio`).

Solución

```
struct job {
  ...
  time_t time;
  int prio;
};
void time_mark(struct job *);
bool time_wait_longer(struct job *);
struct queue {
  ...
  pthread_mutex_t lock;
};
struct queues {
  struct queue[NUMQUEUES];
} queues;


int queue_job(struct job *job, unsigned int prio)
{
  if (prio >= NUMQUEUES) {
    return -1;
  }
  time_mark(job);
  pthread_mutex_lock(&queue[prio].lock);
  queue_add(queue[prio], job);
  pthread_mutex_unlock(&queue[prio].lock);
}

void run(void)
{
  int i;
  while(true) {
    for (i = 0; i < NUMQUEUES; i++) {
      pthread_mutex_lock(&queues.queues[i].lock);
      struct job * job = queue_get(queues.queues[i]);
      if (job) {
        queue_remove(queues.queues[i]);
        pthread_mutex_unlock(&queues.queues[i].lock);

        execute(job);

        pthread_mutex_lock(&queues.queues[job->prio].lock);
        queue_add(queues.queues[job->prio], job);
        pthread_mutex_unlock(&queues.queues[job->prio].lock);
        break;
      } else pthread_mutex_unlock(&queues.queues[i].lock);
  }
}

void *five_seconds(void *arg)
{
  int i;
  struct job *job;

  while(true) {
    sleep(1);
    for(i=1; i < NUMQUEUES; i++) {
        while(1) {
            pthread_mutex_lock(&queues.queues[i].lock);
            job = queue_get(&queues.queues[i]);
            if(job && time_wait_longer(job)) {
                queue_remove(&queues.queues[i]);
                pthread_mutex_unlock(&queues.queues[i].lock);
                time_mark(job);
                pthread_mutex_lock(&queues.queues[i-1].lock);
                queue_add(&queues.queues[i-1], job);
                pthread_mutex_unlock(&queues.queues[i-1].lock);
            } else {
                pthread_mutex_unlock(&queues.queues[i].lock);
                break;
            }
        }
    }
  }
}
```

3. **Notificador** [1 punto]

   Implemente en Erlang un servicio de propagación de mensajes. El API de este servicio tiene tres funciones:

   - `notifier:start()`, que arranca un servicio de propagación y devuelve su `PID`.
   - `notifier:register(N)`, que registra al proceso que llama a la función en el notificador `N`
   - `notifier:send(N, Msg)`, que envía el mensaje `Msg` a todos los procesos registrados en el notificador `N`.

```
-module(notifier).

-export([start/0, register/1, send/2]).
-export([loop/...]).

%% API

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

register(N) -> ...
send(N, Msg) -> ...

%% Internal Functions

loop() ->
    ...
```

   Implemente las funciones `send/2`, `register/1` y `loop`. Puede añadir parámetros a loop si lo necesita.

   Solución

```
-module(notifier).

-export([start/0, register/1, send/2]).
-export([loop/1]).

%% API

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

register(N) -> N ! {register, self()}.
send(N, Msg) -> N ! {send, Msg}.

%% Internal Functions

loop(Procs) ->
    receive
        {register, P} ->
            loop([P | [X || X <- Procs, X /= P]]);
        {send, Msg} ->
            send_msg(Procs, Msg),
            loop(Procs)
    end.

send_msg([], _) -> ok;
send_msg([P|T], Msg) ->
    P ! Msg,
    send_msg(T, Msg).

% send_msg con lists:foreach
% send_msg(Procs, Msg) ->
%     lists:foreach(fun(P) -> P ! Msg end, Procs)
```