Apellidos: *Nombre:*

Concurrencia y Paralelismo

Bloque I: Concurrencia

Julio 2022

1. *Thread activo [2 puntos]*

   Un sistema tiene NTHREADS threads, de los que únicamente hay uno activo en cada momento. Ese thread le pasa el turno a otro thread del grupo, e imprime un mensaje con su número y el del thread que continúa el ciclo.

```
#define NTHREADS ...

struct tag_group {
    int current_thread;  // active thread
    ...
};

struct thr_args {
    int my_number;
    struct tag_group *t; // Shared
};

void thread(void *ptr) {
    struct thr_args *p = ptr;
    int next;

    while(1) {
        ...
        next = rand() % NTHREADS;
        printf("thread %d tagging thread %d\n", p->my_number, next);
        ...
    }
}
```

Implemente la función thread de tal forma que el comportamiento sea el descrito. Cada thread debe despertar únicamente al siguiente, no a todos.

```
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_unlock(pthread_mutex_t *mutex);
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*fun) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Solución

```
#define NTHREADS ...

struct tag_group {
    int current_thread;  // active thread
    pthread_mutex_t m;
    pthread_cond_t cond[NTHREADS];
};

struct thr_args {
    int my_number;
    struct tag_group *t; // Shared
};

void thread(void *ptr) {
    struct thr_args *args = ptr;
    int next;

    while(1) {
        pthread_mutex_lock(&args->t->m);
        while(args->t->current_thread != args->my_number)
            pthread_cond_wait(&args->t->cond[args->my_number], &args->t->m);
        pthread_mutex_unlock(&args->t->m);

        next = rand() % NTHREADS;
        printf("thread %d tagging thread %d\n", args->my_number, next);

        pthread_mutex_lock(&args->t->m);
        args->t->current_thread = next;
        pthread_cond_signal(args->t->cond[next]);
        pthread_mutex_unlokc(&args->t->m);
    }
}
```

2. *Sistema de gestión de memoria [2 puntos]*

   Implemente un sistema de gestión de memoria. El API tiene dos operaciones, `malloc()` y `free()`. Existen varios `buckets` de tamaños de memoria, todos potencias de 2: `8, 16, 32, 64, 128, 256, 512`. El sistema funciona de la siguiente manera:

   - Existe un array `buckets` que contiene una entrada para cada uno de los tamaños.
   - Cada entrada de este array tiene una lista de bloques libres del tamaño correspondiente.
   - Cada elemento de la lista está compuesto por un campo de la estructura elem, que contiene un puntero a la zona de memoria libre `pointer`.
   - La función `elem_create()` crea un nuevo elemento.
   - La función `elem_delete(elem)` borra el elemento `elem`.
   - La función `list_insert(list, elem)` inserta el elemento `elem` en la lista `list`.
   - La función `list_next()` devuelve el primer elemento de la lista, si la lista esta vacía devuelve `NULL`. Este elemento se borra de la lista.
   - Las funciones anteriores no hay que implementarlas.

   Hay que implementar las siguientes funciones:

   - `void *malloc(size_t size)`. Esta función busca un bloque de memoria en el primer bucket en el que cabe este trozo. Es decir `malloc(7)` iría en el bucket de 8, y `malloc(59)` en el de 64.
   - Si no hay buckets disponibles para el tamaño correspondiente, se busca un bucket en el tamaño inmediatamente superior, se parte en dos, uno se mete en la lista y el otro se devuelve.
   - `void free(void *pointer, size_t size)`. Esta función libera el bloque de memoria que se le pasa. Nos indica el tamaño del bloque que se ha asignado para hacer todo más sencillo.
   - El sistema tiene que funcionar de forma multithread. Es decir, puede haber varios `malloc()` y `free()` ejecutándose al mismo tiempo.
   - Hay que minimizar el tiempo que hay un `mutex` bloqueado. Pista: Úsese un mutex por bucket.
   - El array de buckets está indexado por la potencia de 2 que contiene, teniendo en cuenta que empezamos en tamaño 8. La posición 0 contiene el tamaño 8, la posición 2 contiene el tamaño 16, etc. Es decir, el tamaño n (donde $n = 2^c$) se almacena en la posición $c - 3$.

```
struct list_t;

struct elem {
  pointer *pointer; /* pointer to memory */
};

struct bucket {
  size_t size;
  list_t *blocks;
  ...
};
struct bucket buckets[NUM];

struct elem *elem_create(void);
vod elem_delete(struct elem *elem);
void list_insert(struct list_t list, struct list_t *elem);
struct elem *list_next(struct list_t *list,)
```

Solución

```
struct list_t;

struct elem {
        pointer *pointer; /* pointer to memory */
};

struct bucket {
        size_t size;
        list_t *blocks;
        pthread_mutex_t *m;
};
struct bucket buckets[NUM];

struct elem *elem_create(void);
vod elem_delete(struct elem *elem);
void list_insert(struct list_t list, struct list_t *elem);
struct elem *list_next(struct list_t *list)

// Power of the sizes
#define MIN_SIZE 3
#define MAX_SIZE 9

struct elem *buddy_malloc(int i)
{
        if (i >= MAX_SIZE)
                return NULL;
        pthread_mutex_lock(buckets[i - MIN].m);
        struct elem *e = list_next(buckets[i - MIN].blocks);
        pthread_mutex_unlock(buckets[i - MIN].m);
        if (e != NULL)
                return e;
        e = buddy_malloc(i + 1);
        if (e == NULL)
                return NULL;
        struct elem *buddy = elem_create();
        pthread_mutex_lock(buckets[i - MIN].m);
        list_insert(buckets[i - MIN].blocks, buddy);
        pthread_mutex_unlock(buckets[i - MIN].m);
        return e;
}

void *malloc(size_t size)
{
        for (int i = MIN_SIZE; i <= MAX_SIZE; i++) {
                if (size < 2 ^ i) {
                        pthread_mutex_lock(buckets[i - MIN].m);
                        struct elem *e = list_next(buckets[i - MIN].blocks);
                        pthread_mutex_unlock(buckets[i - MIN].m);
                        if (e == NULL) {
                                return NULL;
                        }
                        void *p = e->pointer;
                        free(e);
                        return pointer;
                }
        }
        printf("Size too big %d\n", size);
}

void free(void *pointer, size_t size)
{
        for (int i = MIN_SIZE; i <= MAX_SIZE; i++) {
                if (size < 2 ^ i) {
                        struct elem *e = elem_create();
                        e->pointer = pointer;
                        pthread_mutex_lock(buckets[i - MIN].m);
                        list_insert(buckets[i - MIN].blocks, e);
                        pthread_mutex_unlock(buckets[i - MIN].m);
                        return;
                }
        }
        printf("Invalid size %d\n", size);
}
```

3. *Sistema de log [1 punto]*

   En un sistema existe un servicio de log que recibe mensajes de otros procesos con un nivel de gravedad asociado, y los imprime por pantalla o no según el nivel mínimo de gravedad actual.

   Para eso, el servidor proporciona un API con dos funciones:

   - `set_log_level/2`, que permite cambiar el nivel de gravedad del servidor de log. Los mensajes solo se imprimirán si su nivel de gravedad es igual o superior al valor establecido.
   - `log/3`, que envía un mensaje con su nivel de gravedad al servidor.

   Implemente las funciones `set_log_level/2`, `log/3` y `loop` para que el sistema tenga el comportamiento descrito. Puede modificar el número de parámetros de loop si lo necesita.

```
-module(log).

-export([start/0, init/0, set_log_level/2, log/3]).

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

set_log_level(Log_Server, Level) ->
    ...

log(Log_Server, Msg, Level) ->
    ...

init() ->
    loop(0).

loop(Current_Level) ->
    ...

log_message(Msg, Level) ->
    io:format("[~w]: ~w~n", [Level, Msg]).
```

Solución

```
-module(log).

-export([start/0, init/0, set_log_level/2, log/3]).

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

set_log_level(Log_Server, Level) ->
    Log_Server ! {set_log_level, Level}.

log(Log_Server, Msg, Level) ->
    Log_Server ! {log, Msg, Level}.

init() ->
    loop(0).

loop(Current_Level) ->
    receive
        {set_log_level, Level} ->
            loop(Level);
        {log, Msg, Level} ->
            if
                Level >= Current_Level ->
                    log_message(Msg, Level);
                true ->
                    ok
            end,
            loop(Current_Level)
    end.

log_message(Msg, Level) ->
    io:format("[~w]: ~w~n", [Level, Msg]).
```