Apellidos: *Nombre:*

Concurrencia y Paralelismo

Bloque I: Concurrencia

Junio 2022

1. *Gestión de salida [2 puntos]* Un sistema gestiona la salida a disco a través de un único thread que recibe peticiones del resto de threads del sistema. Cuando un thread quiere realizar una operación de escritura, envía una petición a través de una cola. Para simplificar el problema, asumimos que la cola es segura frente a accesos concurrentes (no es necesario bloquear el acceso a ella).

```
struct request {
    request_info inf;
    ...
};

// Funciones ya implementadas en otros modulos del sistema
struct request *remove_request();
void insert_request(struct request *r);


void submit_request(struct request *r) {
    insert_request(r);
}

void wait_till_done(struct request *r) {
    ...
}

void *io_thread(void *ptr) {
  while(1) {
    struct request *r;

    r = remove_request();
    do_io(r->inf);
  }
}
```

El thread que gestiona las escrituras se ejecuta en `io_thread`. Los threads que quieren realizar una operación llaman a `submit_request`, e insertan una `struct request` en la cola. Esa estructura es retirada de la cola por el thread de escritura, y se comparte entre esos dos threads.

Implemente la función `wait_till_done`, y modifique `io_thread` y `submit_request` para que los threads que han enviado una petición puedan llamarla para esperar hasta que ésta se haya completado. Si la escritura ya se completó la llamada debería volver inmediatamente. Puede añadir más campos a la estructura `request` si lo necesita.

```
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

```
struct request {
    request_info inf;
    pthread_cond_t *request_not_done;
    pthread_mutex_t *request_m;
    int done;
};

struct request *remove_request();
void insert_request(struct request *r);


void submit_request(struct request *r) {
    r->done = 0;
    pthread_mutex_init(&r->request_m, NULL);
    pthread_cond_init(&r->request_not_done, NULL);
    insert_request(r);
}

void wait_till_done(struct request *r) {
    pthread_mutex_lock(&r->request_m);
    if(!r->done)
        pthread_cond_wait(&r->request_not_done, &r->request_m);
    pthread_mutex_unlock(&r->request_m);
}

void *io_thread(void *ptr) {
    while(1) {
        struct request *r;

        r = remove_request();
        do_io(r->inf);

        pthread_mutex_lock(&r->request_m);
        r->done = 1;
        pthread_cond_broadcast(&r->request_not_done);
        pthread_mutex_unlock(&r->request_m);
    }
}
```

2. *Threads que simulan corredores de una carrera por relevos [2 puntos]*

   Crear la función `runner()` que simula un corredor en una carrera de relevos.

   Condiciones a tener en cuenta:

   - No pueden usarse variables globales.
   - No puede hacerse espera activa.
   - No pueden usarse semáforos.
   - El primer (1) y el último (N) corredor son especiales.
   - El primero tiene que esperar a que se de la salida, y que ésta sea válida. Si no es válida hay que repetir la salida. La variable `valid_start` indica si la salida es válida. La variable de condición `start` se lanza cada vez que hay una salida.
   - Cada corredor indicará con un mensaje desde que hora está listo, a que hora recibe el testigo y a que hora termina su relevo. Usar la función `seconds_since_start()` para saber cuantos segundos han pasado desde el comienzo de la carrera.
   - El último corredor de cada equipo indicará en que posición ha llegado.
   - Los corredores de cada equipo tienen que correr en orden de número dentro del equipo (parámetro `id_runner`).
   - Cada corredor tiene que pasar el testigo al siguente corredor del mismo equipo. Pista, úsese un campo en la `struct team`.
   - El programa tiene que funcionar incluso si un corredor se despierta en un momento que no le toque correr.
   - Puede suponerse que la salida se efectua después de que todos los corredores estén listos para empezar.
   - Cada corredor tiene que ejecutar la función `run_100_meters()`, que es la que simula que ha corrido los metros que le corresponden.
   - Se pueden añadir los campos que sean necesarios a las `struct global` y a la `struct team`.
   - La `struct global` es la misma para todos los corredores.
   - La `struct team` es la misma para todos los miembros de un equipo.
   - Hay que minimizar el tiempo que pasa desde que un corredor puede correr hasta que empieza a correr.
   - Añadir un comentario a cada campo de las estructuras indicando qué significa y a que valor se inicializa.

```
struct global {
  pthread_cond_t start; /* broadcast when race begins */
  pthread_mutex_t m;    /* protect everything on this struct */
  bool valid_start;     /* indicates if the start was good. Initialized to false */
};

struct team {
  int num_runners_team;
};

void runner(struct global *g, struct team *t, int id_runner)
{
  ...
  run_100_meters();
  ...
}
int seconds_since_start(void);
```

Solución

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

struct global {
    pthread_cond_t start; /* broadcast when race begins */
    pthread_mutex_t m;    /* protect everything on this struct */
    bool valid_start;     /* indicates if the start was good. Initialized to false */
    int position;         /* current position at end */
};

struct team {
    int num_runners_team;
    char *team_name;
    pthread_cond_t wait_token[]; /* index is id_runner - 1 */
    pthread_mutex_t m;
    int next;
};

void runner(struct global *g, struct team *t, int id_runner)
{
        printf("Runner %d is ready at %d\n", id_runner, seconds_since_start());
        if (id_runner == 1) {
                pthread_mutex_lock(&g->m);
                while (!g->valid_start)
                        pthread_cond_wait(&g->start, &g->m);
                pthread_mutex_unlock(&g->m);
                pthread_mutex_lock(&m);
                t->next = 2;
        } else {
                pthread_mutex_lock(&t->m);
                while (t->next != id_runner) {
                        pthread_cond_wait(&t->wait_token[id_runner - 2]);
                }
                t->next++;
                pthread_mutex_unlock(&t->m);
        }

        printf("Runner %d starts running at %d\n", id_runner, seconds_since_start());
        run_100_metters();
        printf("Runner %d ends running at %d\n", id_runner, seconds_since_start());

        if (id_runner == t->num_runner_team) {
                pthread_mutex_lock(&g->m);
                int pos = g->position;
                pthread_mutex_lock(&g->m);
                printf("Team %s finish in  position %d\n", pos);
        } else {
                pthread_cond_signal(&t->wait_token[id_runner - 1]);
        }
}

int seconds_since_start(void)
{
        return time();
}
```

3. *Procesos en anillo [1 punto]*

   Tenemos un sistema con un grupo de procesos distribuidos en una estructura de anillo, donde cada proceso conoce únicamente el PID del siguiente.

   La implementación del sistema es la siguiente:

```
-module(ring).

-export([start/1, init/0, ring_size/1]).

start(N) ->
    Pids = start_processes(N),
    send_next(Pids),
    Pids.

ring_size(Pid) ->
    ...

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

send_next([], []) -> ok;
send_next([Pid | Pids], [Next_pid | Next_pids]) ->
    Pid ! {next, Next_pid},
    send_next(Pids, Next_pids).

send_next([Pid | Pids]) ->
    send_next([Pid | Pids], Pids ++ [Pid]).

init() ->
    receive
        {next, Next_pid} ->
            loop(Next_pid)
    end.

loop(Next_pid) ->
    ...
    loop(Next_pid).
```

   Implemente la función `ring_size/1`, que dado el pid de uno de los procesos del anillo, devuelve el número total de procesos que forman parte del mismo. Modifique únicamente las Funciones `ring_size` y `loop`

Solución

```
-module(ring).

-export([start/1, init/0, ring_size/1]).

start(N) ->
    Pids = start_processes(N),
    send_next(Pids),
    Pids.

ring_size(Pid) ->
    Pid ! {get_size, self(), Pid},
    receive
        {get_size_reply, Size} ->
            Size
    end.

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

send_next([], []) -> ok;
send_next([Pid | Pids], [Next_pid | Next_pids]) ->
    Pid ! {next, Next_pid},
    send_next(Pids, Next_pids).

send_next([Pid | Pids]) ->
    send_next([Pid | Pids], Pids ++ [Pid]).

init() ->
    receive
        {next, Next_pid} ->
            loop(Next_pid)
    end.

loop(Next_pid) ->
    receive
        {get_size, From, First} ->
            Next_pid ! {go_around, From, First, 1};
        {go_around, From, self(), N} ->
            From ! {get_size_reply, N};
        {go_around, From, First, N} ->
            Next_pid ! {go_around, From, First, N+1}
    end,
    loop(Next_pid).
```