Concurrencia y Paralelismo

Grado en Ingeniería Informática

Julio 2020

1. *Futures [0.75 puntos]*

   Una future es una abstracción para retrasar computaciones. Imaginemos que queremos calcular `c = f(a)` pero que `c` no lo usamos hasta mucho más adelante en el programa. Podemos transformar esto como `future = promise(f, a)` y en el uso de `c` como `result = force(future)`. Se da una implementación de referencia de `promise` y `force`. Puede asumirse que:

   - Las funciones que vamos a usar toman un argumento `void *` y devuelven un valor del mismo tipo.
   - Se puede hacer más de un un `force` para cada `future`. La operacion `free_future` libera la `future`.

   Implementar `future` para que cree un `thread` para realizar la operación y `force` para que si el thread ya ha terminado devuelva el valor calculado y en caso contrario espere a que el thread termine. La función `free_future` libera la memoria usada por la future. Pueden añadirse todos los campos que se consideren necesarios a la `struct future`. Notese que `free_future` tiene que liberar también la memoria del resultado.

   Ejemplo de uso:

```
struct arg {
        int i;
        int result;
}

void *add_one(void *v)
{
        struct arg *arg = v;
        arg->result = arg->i + 1;
        return arg->result;
}

int main(void)
{
        struct arg a;
        struct future *future;
        int *result;

        a.i = 5;
        future = promise(add_one, &a);
        result = force(future);
        printf("The result is %d\n", *result);
        free_future(future);
        return 0;
}
```

```
struct future {
        void *(*f)(void*);
        void *arg;
        void * result;
};

struct future *promise(void *(*f)(void *), void *arg)
{
        struct future *future = malloc(sizeof()struct future));
        assert(!future)
                return NULL;
```

```
        future->f = f;
        future->arg = arg;
        return future;
}


void *force(struct future *future)
{
        assert(future);
        future->result = future->f(future->arg);
        return future->result;
}

void free_future(struct future *future)
{
        assert(future);
        free(result);
        free(future);
}
```

Solución

```
struct future {
        void *(*f)(void*);
        void *arg;
        bool done;
        pthread_cond_t cond;
        pthread_mutex_t mutex;
};


void *calculate(void *arg)
{
        struct future *future = arg;
        void *result;
        result = future->f(future->arg);
        pthread_mutex_lock(&future->mutex);
        future->result = result;
        future->done = true;
        pthread_mutex_unlock(&future->mutex);
        pthread_cond_signal(&future->cond);

        return NULL;
}


struct future *promise(void *(*f)(void *), void *arg)
{
        struct future *future = malloc(sizeof()struct future));
        pthread_t thr;
        if (!future)
                return NULL;
        future->f = f;
        future->arg = arg;
        future->done = false;
        pthread_create(&thr, NULL, calculate, future);
        return future;
}


void force(struct future *future)
{
        void *result;
        assert(future);

        pthread_mutex_lock(&future->mutex);
        while(!future->done)
                pthread_cond_wait(&future->cond, &future->mutex);
        pthread_mutex_unlock(&future->mutex);
}

void free_future(struct future *future)
{
        void *result;
        assert(future);

        pthread_mutex_lock(&future->mutex);
        while(!future->done)
                pthread_cond_wait(&future->cond, &future->mutex);
        pthread_mutex_unlock(&future->mutex);
        free(future);
}
```

2. *Paso de peatones [0.75 puntos]*

   Vamos a modelar un paso de peatones, donde cada peatón y cada coche está representado por un thread. El cruce no tiene semáforo, por lo que los peatones tienen prioridad. Varios peatones pueden cruzar el paso simultaneamente, pero la carretera tiene un único carril y los coches tienen que pasar de uno a uno en el orden en que llegan al cruce.

```
struct crossing {
    pthread_mutex_t *lock;
    int pedestrians;                // Number of pedestrians (waiting or crossing)
    int cars_crossing;              // Number of cars crossing
    pthread_cond_t pedestrian_wait;
    queue *car_queue;
};

int             queue_size(queue *);
pthread_cond_t *queue_remove(queue *);
void            queue_insert(queue *, pthread_cond_t *);

void *pedestrian(void *ptr) {
    struct crossing *cross = ptr;

    pthread_mutex_lock(cross->lock);
    cross->pedestrians++;
    while(cross->cars_crossing > 0)
        pthread_cond_wait(cross->pedestrian_wait, cross->lock);
    pthread_mutex_unlock(cross->lock);

    cross();

    pthread_mutex_lock(cross->lock);
    cross->pedestrians--;
    if(cross->pedestrians==0 && queue_size(cross->car_queue) >0) {
        cross->cars_crossing++;
        pthread_cond_t *c = queue_remove(cross->car_queue);
        pthread_cond_signal(c);
    }
    pthread_mutex_unlock(cross->lock);

    return NULL;
}

void *car(void *ptr) {
    struct crossing *cross = ptr;
    ...
    cross();
    ...
}
```

Implemente la función `car` con el comportamiento del coche. Puede añadir más campos a la estructura crossing si lo considera necesario.

Solución

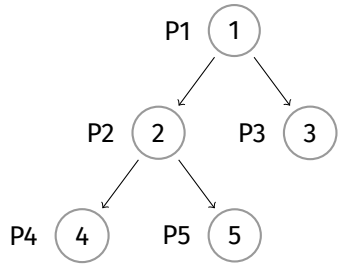
```
struct crossing {
    pthread_mutex_t *lock;
    int pedestrians;
    int cars_crossing;
    pthread_cond_t pedestrian_wait;
    queue *car_queue;
};

int queue_size(queue *);
pthread_cond_t *queue_remove(queue *);
void queue_insert(queue *, pthread_cond_t *);

void *pedestrian(void *ptr) {
    struct crossing *cross = ptr;

    pthread_mutex_lock(cross->lock);
    cross->pedestrians++;
    while(cross->cars_crossing > 0)
        pthread_cond_wait(cross->pedestrian_wait, cross->lock);
    pthread_mutex_unlock(cross->lock);

    cross();

    pthread_mutex_lock(cross->lock);
    cross->pedestrians--;
    if(cross->pedestrians==0 && queue_size(cross->car_queue) >0) {
        cross->cars_crossing++;
        pthread_cond_t *c = queue_remove(cross->car_queue);
        pthread_cond_signal(c);
    }
    pthread_mutex_unlock(cross->lock);
}

void *car(void *ptr) {
    struct crossing *cross = ptr;

    pthread_mutex_lock(cross->lock);
    if(cross->pedestrians > 0 || cross->cars_crossing > 0) {
        pthread_cond_t c;
        pthread_cond_init(&c, NULL);
        queue_insert(cross->q, &c);
        pthread_cond_wait(&c, cross->lock);
    } else cross->cars_crossing++;
    pthread_mutex_unlock(cross->lock);

    cross();

    pthread_mutex_lock(cross->lock);
    cross->cars_crossing--;
    if(cross->pedestrians > 0)
        pthread_cond_broadcast(cross->pedestrian_wait);
    else if(queue_size(cross->car_queue)>0) {
        cars_crossing++;
        pthread_cond_t *c = queue_remove(cross->car_queue);
        pthread_cond_signal(c);
    }
    pthread_mutex_unlock(cross->lock);

}
```

3. *Árbol de Procesos [0.5 puntos]*

   Tenemos un sistema donde hay un árbol binario de procesos, donde cada nodo es un proceso que guarda un valor y dos pids correspondientes a los subarboles izquierdo y derecho. Los nodos se crean llamando a `start_node/1`, y añadimos hijos a un nodo existente con `add_left_child/2` y `add_right_child/2`.

```
-module(tree).

-export([start_node/1, add_left_child/2, add_right_child/2, height/1, init_node/1]).

%% API

start_node(V) ->
    spawn(?MODULE, init_node, [V]).

add_left_child(Tree, Child_Tree) ->
    Tree ! {add_left_child, Child_Tree}.

add_right_child(Tree, Child_Tree) ->
    Tree ! {add_right_child, Child_Tree}.

get_value(Tree) ->
    ...

exists(Tree, V) ->
    ...

%% Internal functions

init_node(V) ->
    node_loop(V, none, none).

node_loop(V, Left, Right) ->
    receive
        {add_left_child, Child_Tree} ->
            node_loop(V, Child_Tree, Right);
        {add_right_child, Child_Tree} ->
            node_loop(V, Left, Child_Tree);
        ...
    end.
```

   Por ejemplo, el árbol:



   Se crearía de la siguiente forma:

```
P1 = start_node(1).
P2 = start_node(2).
P3 = start_node(3).
P4 = start_node(4).
P5 = start_node(5).
add_left_child(P2, P4).
add_right_child(P2, P5).
add_left_child(P1, P2).
add_right_child(P1, P3).
```

   Implemente la función `get_value/1` que devuelve el valor de la raíz de un árbol, y la función `exists/2`, que devuelve `true` si un valor existe en el árbol, y `false` si no existe.

```
-module(tree).

-export([start_node/1, add_left_child/2, add_right_child/2, height/1, init_node/1]).

%% API

start_node(V) ->
    spawn(?MODULE, init_node, [V]).

add_left_child(Tree, Child_Tree) ->
    Tree ! {add_left_child, Child_Tree}.

add_right_child(Tree, Child_Tree) ->
    Tree ! {add_right_child, Child_Tree}.

get_value(Tree) ->
    Tree ! {get_value, self()},
    receive
        {get_value_reply, V} ->
            V
    end.

exists(Tree, V) ->
    Tree ! {exists, V, self()},
    receive
        {exists_reply, R} ->
            R
    end.

%% Internal functions

init_node(V) ->
    node_loop(V, none, none).

node_loop(V, Left, Right) ->
    receive
        {add_left_child, Child_Tree} ->
            node_loop(V, Child_Tree, Right);
        {add_right_child, Child_Tree} ->
            node_loop(V, Left, Child_Tree);
        {get_value, From} ->
            From ! {get_value_reply, V},
            node_loop(V, Left, Right);
        {exists, V, From} ->
            From ! {exists_reply, true},
            node_loop(V, Left, Right);
        {exists, Val, From} ->
            From ! {exists_reply, check_exists(Left, Val) or check_exists(Right, Val)},
            node_loop(V, Left, Right)
    end.

check_exists(none, _) -> false;
check_exists(Tree, V) -> exists(Tree, V).
```