Concurrencia y Paralelismo

Grado en Ingeniería Informática

Junio 2020

1. *Entrada de grupos en un supermercado [0.75 puntos]*

   Vamos a modelar un sistema de control de entrada a un supermercado durante el estado de alarma. Las reglas son las siguentes:

   - Dentro del supermercado puede haber como máximo 5 personas.
   - Como regla general, los clientes tienen que ir al supermercado individualmente. Las únicas excepciones son:
     - una persona dependiente puede ir acompañado de la persona que la cuida.
     - un progenitor puede ir con hasta dos hijos.
   - Cada uno de los casos del punto anterior se considera un grupo.
   - Los grupos tienen que esperar fuera del supermercado, guardando dos metros de distancia entre cada grupo. El máximo número de grupos que pueden esperar fuera de un supermercardo son cuatro. Los grupos van entrando según haya sitio, sin necesidad de respetar el orden de llegada. Si ya hay cuatro grupos esperando fuera los nuevos clientes se van a buscar otro supermercado.
   - No se pueden usar variable globales.

   Completar el código de los tres tipos de grupos para que se cumplan en todo momento las reglas.

```
struct estado {
     ....
};

// Operaciones adicionales permitidas
void irse_a_otro_supermercado(void);

void persona_sola(struct estado *s)
{
        hacer_la_compra(s, 1);
}

void persona_dependiente(struct estado *s)
{
        hacer_la_compra(s, 2);
}

void progenitor_con_hijos(struct estado *s, int hijos)
{
        assert(hijos >= 1 && hijos <= 2);
        hacer_la_compra(s, 1 + hijos);
}

void hacer_la_compra(struct estado *s, int personas)
{
        ....
        comprar_alimentos();
        ....
}
```

Solución

```
struct estado {
        pthread_mutex_t m;
        pthread_cond_t c;
        int personas_dentro;
        int grupos_fuera;
};

void hacer_la_compra(struct estado *s, int personas)
{
        pthread_mutex_lock(&s->m);
        if (s->personas_dentro == 5 && s->grupos_fuera == 4) {
                pthread_mutex_unlock(&s->m);
                irse_a_otro_supermercado();
                return;
        }
        s->grupos_fuera++;
        while (5 - s->personas_dentro < personas) {
              pthread_mutex_wait(&s->m, &s->c);
        }
        s->grupos_fuera--;
        s->personas_dentro += personas;
        pthread_mutex_unlock(&s->m);

        cormprar_alimentos();

        pthread_mutex_lock(&s->m);
        if (s->grupos_fuera)
                pthread_cond_broadcast(&s->c);
        s->personas_adentro -= personas;
        pthread_mutex_unlock(&s->m);
}
```

2. *Sistema de gestión de turnos [0.75 puntos]*

   El siguiente código intenta modelar un sistema de turnos como los que se utilizan en carnicerías, pescaderías o correos para gestionar la espera de los clientes. En estos sistemas cada cliente coge un número y espera hasta que se llama. Cada cliente va a correr en un thread separado, y además habrá un empleado en su propio thread que atiende el servicio.

```
struct turn{
    int current; // Current number being served (starts at 0)
    int last;    // Last number assigned (starts at -1)
    pthread_cond_t no_customers;
    pthread_cond_t customer_wait;
    ...
};

void *employee(void *ptr) {
    struct turn *t = ptr;

    while(1) {
        while(t->current > t->last) // No customers
            pthread_cond_wait(&t->no_customers, ...);

        t->current++;
        pthread_cond_broadcast(&t->customer_wait);

        serve();
    }

    return NULL;
}

void *customer(void *ptr) {
    struct turn *t = ptr;

    ...
    buy();

    return NULL;
}
```

   Complete las funciones employee y customer y añada los elementos de sincronización que necesite a la estructura turn.

   ¿Sería posible modificar el problema para que se despertase únicamente al cliente que tiene el turno en ese momento? Si crees que sí, explica como modificarías la implementación (no es necesario reescribir el código, solo explicar que añadirías/modificarías), y si crees que no, justifica por qué.

Solución

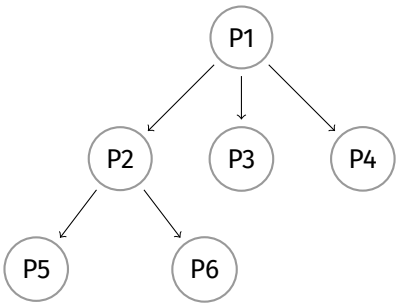
```
struct turn{
    int current;
    int last;
    pthread_cond_t no_customers;
    pthread_cond_t customer_wait;
    pthread_mutex_t lock;
};

void *employee(void *ptr) {
    struct turn *t = ptr;

    while(1) {
        pthread_mutex_lock(&t->lock);
        while(t->current > t->last)
            pthread_cond_wait(&t->no_customers, &t->lock);

        t->current++;
        pthread_cond_broadcast(&t->customer_wait);
        pthread_mutex_unlock(&t->lock);

        serve();
    }
}

void *customer(void *ptr) {
    struct turn *t = ptr;
    int myturn;

    pthread_mutex_lock(&t->lock);
    myturn=++t->last;
    if(myturn == t->current)
        pthread_cond_signal(&t->no_customers);
    while(myturn <= t->current)
        pthread_cond_wait(&t->customer_wait, t->lock);
    pthread_mutex_unlock(&t->lock);
    buy();

    return NULL;
}
```

3. *Árbol de Procesos [0.5 puntos]*

   Tenemos un sistema donde hay un árbol n-ario de procesos, donde cada nodo es un proceso que guarda una lista con los pids de sus hijos. Los nodos se crean llamando a `start_node/0`, y añadimos hijos a un nodo existente con `add_child/2`.

```
-module(tree).

-export([start_node/0, add_child/2, height/1]).

%% API

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

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

height(Tree) ->
    ...

%% Internal functions

init_node() ->
    node_loop([]).

node_loop(Children) ->
    receive
        {add_child, Child_Tree} ->
            node_loop(Children ++ [Child_Tree]);
        ...
    end.
```

   Por ejemplo, el árbol:



   Se crearía de la siguiente forma:

```
P1 = start_node().
P2 = start_node().
P3 = start_node().
P4 = start_node().
P5 = start_node().
P6 = start_node().
add_child(P2, P5).
add_child(P2, P6).
add_child(P1, P2).
add_child(P1, P3).
add_child(P1, P4).
```

   Implemente la función `height/1` que calcula la altura de un árbol de procesos.

```
-module(tree).

-export([start_node/0, add_child/2, height/1, init_node/0]).

%% API

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

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

height(Tree) ->
    Tree ! {get_height, self()},
    receive
        {height_reply, Height} ->
            Height
    end.

%% Internal functions

init_node() ->
    node_loop([]).

node_loop(Children) ->
    receive
        {add_child, Child_Tree} ->
            node_loop(Children ++ [Child_Tree]);
        {get_height, From} ->
            Heights = [height(Tree) || Tree <- Children],
            Height = 1 + lists:foldl(fun erlang:max/2, -1, Heights),
            From ! {height_reply, Height},
            node_loop(Children)
    end.
```