Apellidos: **Nombre:**

Concurrencia y Paralelismo

Grado en Ingeniería Informática

Examen Mayo 2017

1. **Implementar Futures en C** [2 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 `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 pueden hacer varios `force` para cada `future`. Existe una operacion `free_future` para liberar 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`.

   Ejemplo de uso:

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

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

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

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

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

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

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

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

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

int pthread_create(pthread_t *thread, void *(*start_routine) (void *), void *arg);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Solución

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

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_cond_signal(&future->cond);
        pthread_mutex_unlock(&future->mutex);

        return NULL;
}

struct future *promise(void *(*f)(void *), void *arg)
{
        struct future *future = malloc(sizeof()struct future));
        if (!future)
         return NULL;
        future->f = f;
        future->arg = arg;
        future->done = false;
        pthread_create(&future->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. **Sincronización tren/pasajero** [2 puntos]

Se desea implementar un sistema que simule el proceso de subida de pasajeros a un tren en una serie de estaciones de una linea. Los pasajeros y el tren están representados por threads independientes.

El tren sigue el siguiente algoritmo:

a) Al llegar a una estación, el tren comprueba si hay pasajeros esperando en el andén (**waiting_count**). Si no hay pasajeros continúa hasta la siguiente estación.

b) Si hay pasajeros comienza el proceso de carga marcando el flag **loading** de la estación, y avisando a los pasajeros mediante la condición **station_wait**.

c) A continuación espera a que los pasajeros le avisen de que han terminado de subir al tren.

d) Se quita la marca del flag **loading** para avisar de que se ha terminado el proceso de carga, y se parte hacia la siguiente estación.

```
#define STATIONS ...

pthread_cond_t loading_wait;

struct station_info {
    pthread_mutex_t lock;
    pthread_cond_t station_wait;
    int waiting_count;
    int loading;
} station[STATIONS];


struct passenger_info {
    int station;
};


void *train(void *args) {
    int current_station = 0;

    while(1) {
        pthread_mutex_lock(&station[current_station].lock);

        while(station[current_station].waiting_count>0) {
            station[current_station].loading=1;
            pthread_cond_broadcast(&station[current_station].station_wait);
            pthread_cond_wait(&loading_wait, &station[current_station].lock);
            station[current_station].loading=0;
        }
        pthread_mutex_unlock(&station[current_station].lock);
        current_station=(current_station+1) %STATIONS;
    }
}


void *passenger(void *args) {
    struct passenger_info *info=args;

    ...

    return NULL;
}
```

Se pide implementar la función **passenger** teniendo en cuenta:

- El campo **station** de la estructura **passenger_info** indica en que estación espera el pasajero.
- Los pasajeros deberían añadirse a la cuenta de pasajeros en espera al llegar a la estación, y esperar hasta que el tren empiece el proceso de carga.
- Al finalizar el proceso de carga los pasajeros deberían avisar al tren para que salga de la estación. La solución debería avisar al tren una única vez por cada parada.

Solución

```
#define STATIONS ...

pthread_cond_t passenger_wait;

struct station_info {
    pthread_mutex_t lock;
    pthread_cond_t station_wait;
    int waiting_count;
    int loading;
} station[STATIONS];

struct passenger_info {
    int station;
};

void *train(void *args) {
    int cur_station = 0;

    while(1) {
        pthread_mutex_lock(&station[cur_station].lock);

        while(station[cur_station].waiting_count>0) {
            station[cur_station].loading=1;
            pthread_cond_broadcast(&station[cur_station].station_wait);
            pthread_cond_wait(&loading_wait, &station[cur_station].lock);
            station[cur_station.loading=0;
        }
        pthread_mutex_unlock(&station[cur_station].lock);
        cur_station=(cur_station+1) %STATIONS;
    }
}

void *passenger(void *args) {
    struct passenger_info *info=args;

    pthread_mutex_lock(&station[info->station].lock);
    station[info->station].waiting_count++;

    while(!station[info->station].loading)
        pthread_cond_wait(&station[info->station].station_wait, &station[info->station].lock);

    station[info->station].waiting_count--;
    if(station[info->station].waiting_count==0) {
        pthread_cond_signal(&loading_wait);
    }
    pthread_mutex_unlock(&station[info->station].lock);

    return NULL;
}
```

3. **Supervisor** [1 punto]

   Se desea monitorizar el progreso de un grupo de procesos que están realizando un trabajo de cálculo. Para ello se implementa un sistema de supervisión que permite iniciar un grupo de procesos. Cada proceso `i` de ese grupo aplica la función **F** al elemento número `i` de la lista **ArgList**.

   La implementación inicial de ese servicio es la siguiente:

```
-module(supervisor).

-export([start/3, get_progress/1, init/4]).

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

get_progress(Supervisor) ->
    ...

init(N,Mod,F, ArgList) ->
    process_flag(trap_exit, true),
    PidList = start_workers(N, Mod, F, ArgList, []),
    loop(PidList).

start_workers(0, _, _, _, PidList) -> PidList;
```

```
start_workers(N, Mod, F, [Arg|T], PidList) ->
    start_workers(N-1, Mod, F, T, [spawn_link(Mod, F, Arg) | PidList]).

loop(Subs) ->
    receive
        {'EXIT', Pid, _} ->
            ...
            loop(Subs)
    end.
```

Se pide completar la implementación para que la función **get_progress** devuelva la tupla **{Terminados, Porcentaje}**, donde **Terminados** es el número de procesos del grupo que han terminado, y **Porcentaje** es el porcentaje de procesos que han terminado sobre el total de procesos iniciados. Puede añadir parametros a **loop** si lo necesita.

Solución

```
-module(supervisor).

-export([start/3, get_progress/1, init/4]).

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

get_progress(Supervisor) ->
    Supervisor ! {get_progress, self()},
    receive
        {progress, T, P} -> {T, P}
    end.

init(N,Mod,F, ArgList) ->
    process_flag(trap_exit, true),
    PidList = start_workers(N, Mod, F, ArgList, []),
    loop(PidList, N, 0).

start_workers(0, _, _, _, PidList) -> PidList;
start_workers(N, Mod, F, [Arg|T], PidList) ->
    start_workers(N-1, Mod, F, T, [spawn_link(Mod, F, Arg) | PidList]).

loop(Subs, Total, Finished) ->
    receive
        {'EXIT', Pid, _} ->
            loop(Subs, Total, Finished+1);
        {get_progress, From} ->
            From ! {progress, Finished, Finished/Total*100},
            loop(Subs, Total, Finished)
    end.
```