Apellidos: *Nombre:*

Concurrencia y Paralelismo

Grado en Ingeniería Informática

Julio 2021

1. *Clientes en múltiples colas [2.5 puntos]*

   Queremos implementar un sistema donde una serie de clientes esperan a ser atendidos por un servidor. Los clientes esperan en una cola indicada por `c→queue_number` de entre N posibles, numeradas de 0 a N-1. Dentro de cada cola los clientes deben ser atendidos en orden de llegada, y una vez son avisados por el servidor llaman a `get_service()`.

   El servidor atiende a un cliente de cada vez, rotando entre las colas, es decir, atendiendo un cliente de la cola 0, después de la 1, y así sucesivamente. Si una cola está vacía, el servidor debe pasar a la siguiente. No es necesario controlar el caso de que todas las colas estén vacías. Para simular el proceso de atender al cliente el servidor llama a `serve()`.

   Las llamadas a `serve()` y `get_service()` deben hacerse sin ningún mutex bloqueado.

```
#define N ...

void  insert(queue *q, void *);
void *remove(queue *q);
int   elements(queue *q);

struct customer {
    int queue_number;
};

queue *q[N];

void *customer(void *ptr) {
    struct customer *c = ptr;

    ...

    get_service();
}

void *server(void *ptr) {
    while(1) {

        ...
        serve();
    }
}
```

   Complete la implementación de `customer` y `server` para que simulen el comportamiento descrito.

2. *Threads que simulan corredores de una carrera por relevos [2.5 puntos]*

   a) Crear la función `runner()` que simula un corredor en una carrera de relevos. Cosas a tener en cuenta:

   - No pueden usarse variables globales.
   - No puede hacerse espera activa.
   - El primer (1) y el último (N) corredor son especiales.
   - El primero tiene que esperar a que se dé la salida, y que ésta sea válida. Si no es válida hay que repetir la carrera. La variable `valid_start` indica si la salida es válida. La variable de condición `start` se lanza cada vez que hay una salida.
   - El último corredor tiene que imprimir un mensaje cuando llega a meta.
   - Cada corredor imprimirá 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.
   - Los corredores tienen que correr en orden de número (parámetro `id_runner`).
   - Cada corredor tiene que pasar el testigo al siguiente corredor. Pista, úsese un campo en la estructura `token`.
   - El programa tiene que funcionar incluso si un corredor se despierta en un momento en que no le toque correr.
   - Cada corredor tiene que ejecutar la función `run_100_meters()`, que es la que simula que ha corrido los metros que le corresponde.

```
struct token {
  int total_num_runners;
  pthread_cond_t start;
  bool valid_start;
};

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

   b) Modificar la implementación anterior para que se pueda usar `signal()` en vez de `broadcast()`, es decir, cada corredor despierta solo al siguiente. Tiene que seguir funcionando en el caso de que a un corredor se le despierte cuando no le toca correr.

```
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);
```