Práctica 3

Desarrollo extra

3.1. Descripción

Los alumnos dotarán al simulador de un **puente estrecho de una dirección** con capacidad más realista. Recordamos que el simulador posee una longitud de puente del tamaño `LENGTH=3` lo que significa que pueden están simultaneamente hasta tres vehículos **de tamaño normal** en el puente y que se tarda en cruzarlo 3 ciclos de reloj. El efecto de realismo se llevará a cabo mediante dos modificaciones del simulador:

- Se permite el cruce a otro tipo de vehículos como autobuses con un tamaño **mayor**. El autobús solamente podrá cruzar cuando haya más espacio libre en el puente, es decir cuando haya uno o ningún vehículo cruzando al mismo tiempo.
- El simulador tendrá **memoria**, lo que significa que recuerda el orden de llegada de cada vehículo, y cruzarán en el orden en el que llegaron a cada extremo del puente.

Con el fin de ilustrar el nuevo comportamiento se ilustra en el figura 3.1 las dos posibilidades que se pueden contemplar en el simulador con el ejemplo de entrada `example_extra.txt`.

La invocación del nuevo planificador se efectuará de manera análoga a la práctica obteniéndose el esquema de planificación (ver figura 3.1):

```
$ ./sim examples/example_extra.txt example1.log
```

3.2. Modificaciones de las estructuras de datos

La estructura de datos para el coche `tcar` incorpora el campo nuevo `my_turn` correspondiente al turno de llegada con el que llegó el vehículo. Además el campo `type` permite ahora la inclusión del tipo de vehículo autobús.

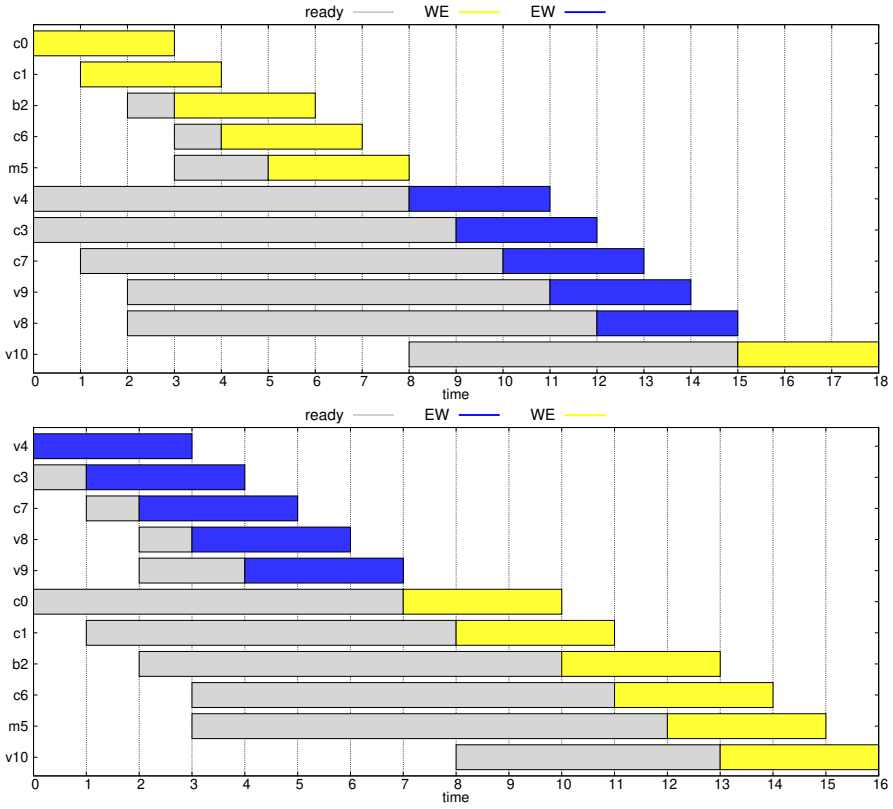

Figura 3.1: Simulación realista para el ejemplo `example_extra.txt`

```
typedef struct {
int id;
char type; // C=car, M=minivan, V=van, B=bus
int my_turn; // NEW
int my_direction; // 0, 1
int t_arrival;
int t_cross_in;
int t_cross_out;
} tcar;
tcar dcar;
```

La estructura de datos para el puente `tbridge` incorpora tres campos nuevos: (1) el cerrojo `mtx_turn` para asignar el turno de llegada al vehículo, (2) la variable `arrival_turn` que corresponde al turno de llegada que se incrementa cuando llega un vehículo nuevo (hay dos correspondientes cada uno de los extremos del puente), y (3) la variable `crossing_turn` que corresponde al turno que hay en ese momento para cruzar el puente:

```
typedef struct {
pthread_mutex_t mtx;
pthread_cond_t VCs[2];
```

```
int cars_on_bridge;
int cur_direction;
int cars_waiting[2];
pthread_mutex_t mtx_turn; //NEW
int arrival_turn[2]; // NEW
int crossing_turn[2]; //NEW
} tbridge;
tbridge dbridge;
```

3.3. Desarrollo de la parte opcional

A continuación se describirán las tareas a realizar para llevar a cabo con éxito codificación de las modificaciones en el simulador:

1. **[4 puntos]** Modificación para dotar al simulador del puente estrecho de orden de llegada. Se modificará la función `bridge_in` para asignar el turno de llegada del vehículo: almacenar en `my_turn` el valor del turno de llegada (`arrival_turn`), posteriormente se incrementará la variable `arrival_turn` protegiéndola adecuadamente en la sección crítica. El vehículo deberá esperar en el puente hasta que sea su turno para cruzar (`my_turn=crossing_turn`). Se recomienda tener en cuenta que a la salida del puente (función `bridge_out`), se deben de despertar a los vehículos en espera en esa dirección para que puedan evaluar si es su turno de cruce.

2. **[2 puntos]** Emular el comportamiento autobuses con un tamaño **mayor** como autobuses. Se modificará la función `bridge_in` para permitir que en caso de que vaya a cruzar una autobús se cumpla la condición de que haya uno o ningún vehículo cruzando al mismo tiempo.

3. **[3 puntos]** Implementar la barrera mediante semáforos en el fichero **barrier.c**. El código del simulador alojado en el campus virtual ya incluye las funciones relativas a los semáforos `#include <semaphore.h>` y el contenido de la estructura `sys_barrier_t` del fichero **barrier.h**:

   ```
   /* Synchronization barrier */
   typedef struct {
   sem_t mutex; /* Barrier lock */
   sem_t cond_slave; /* Variable where threads remain blocked */
   sem_t cond_master; /* Handshake master-slave */
   int nr_threads_arrived; /* Number of threads that reached the barrier */
   int max_threads; /* Number of threads that rely on the syncronization barrier */
   } sys_barrier_t;
   ```

   La función `sys_barrier_wait` debería implementar la barrera de forma que (1) hasta que no llegue el último hilo (denominado hilo maestro) se queden esperando en el semáforo **cond_slave**, (2) una vez que llega el último hilo debe incrementar el semáforo **cond_slave** (tantas veces como hilos esclavos están esperando en dicho semáforo) y (3) este último hilo ha de esperar a que todos los hilos se hayan despertado. Para implementar esta confirmación se utiliza el semáforo **cond_master**, que es decrementado por el maestro tantas veces como hilos estaban esperando en el **cond_slave**, y que a su vez los esclavos van incrementando a medida que se van despertando.