Introduction

The assignment in the lab was to implement a bus using synchronization primitives such that it satisfies certain requirements when tasks are batch-scheduled onto it. The tasks are transmitted over a half-duplex bus that has constraints on the maximum number of concurrent tasks, and must allow high-priority tasks to be transmitted before normal-priority ones. The tasks also have a variable transmission latency.

Data Structures

We have instantiated synchronisation primitives along with pointers to them, as well as a few supporting counters and flags. We have also `typedef`'d locks and condition variables:

```
typedef struct lock Lock;
typedef struct condition Condition;

Lock lock_mutex;                        // Universal lock for both condition variables.
Lock * lock_mutex_p = &lock_mutex;

Condition waitingToGoNP[2];             // Condition Variable for Normal-Priority tasks.
Condition * waitingToGoNP_p = waitingToGoNP;

Condition waitingToGoHP[2];             // Condition Variable for High-Priority tasks.
Condition * waitingToGoHP_p = waitingToGoHP;

static int bus_usage = 0;  // Current number of tasks on bus.
static int bus_enable = 1; // Flag determines if bus will accept new tasks.
static int waitersNP[2];   // Number of Normal-Priority tasks waiting in each dir.
static int waitersHP[2];   // Number of High-Priority tasks waiting in each dir.
static int currentDirection = SENDER; // Current direction of flow.
```

Algorithms

The problem in this assignment maps almost completely onto the *narrow bridge* problem, with the exception of priority levels.

For priority levels, we make sure that high-priority tasks going in the current direction are sent first. If there are no waiting high-priority tasks in the current direction, but high-priority tasks waiting to go in the opposite direction, the bus will be emptied and the direction reversed. The exact mechanism will be described further down.

getSlot Simply, each thread tries to get a slot on the bus in `getSlot()` by attempting to acquire the *lock*. If acquired, a condition must be met, or else the thread will block, releasing the lock so that another thread may enter and attempt to satisfy the requirement. The condition for getting a slot is:

- the bus is not full
- if the bus is not full but in use, the flow must be in the same direction as the thread task
- if the thread task is of normal priority, it may not proceed if thread tasks of high priority are waiting.
- the bus has not been disabled.

If the condition is *not* met, the thread task will proceed to wait (suspend) on a condition variable unique to its priority level.

On successfully satisfying the condition and acquiring the bus, the global bus direction is set to the direction of the task.

transferData Once on the bus, a thread task simply sleeps for a random number of ticks, using `timer_sleep()`. This is done to simulate various data transfer latencies.

leaveSlot When leaving the bus, in `leaveSlot()`, the thread again attempts to acquire the lock and enter the critical section. Once it has acquired it, it will attempt to *signal* tasks waiting to go in the same direction as itself. If high-priority tasks are waiting to go in the current direction, it will signal them. If, however, there are high-prioriy tasks waiting to go in the opposite direction, the `bus_enable` flag will be toggled off and no new tasks will be signalled. This will cause the bus to empty over time.

If no high-priority tasks are waiting on either side, it will try to signal normal-priority tasks waiting to go in the current direction. If no tasks are waiting to go in the current direction, then it will not signal at all.

If the bus is empty, it will re-enable the `bus_enable` flag (if necessary), then it will *signal* to a maximum of three tasks waiting to go in the opposite direction in order to fill the bus. The bus direction will also be reversed. Since separate queues exist for high- and normal-priority tasks, they will be signalled to in order of priority.

Synchronisation

Two *Condition Variables* (similar to Monitors) are used in conjunction with a *lock* that ensures mutual exclusion in critical sections.

In this case the critical sections are those that grant access to, and exit from the bus. The is to say, the entire `getSlot()` and `leaveSlot()` functions, as they manipulate variables critical for gaining access to the bus.

As explained in the *rationale*, condition variables easily allow threads to block until a particular condition is reached.

Rationale

Since we want to execute certain operations dependent on the state of the bus at a given time, mutual exclusion is not sufficient. The task could have been solved with either semaphores or the slightly higher-level condition variables (which use semaphores internally). The latter was chosen as it can *broadcast*, waking all threads blocked on that particular condition variable, which is convenient. They also allow threads acquire a lock, and then release it and block from within a critical section, if a certain condition is not met. They reacquire the lock upon being signalled (or broadcasted to).

This solution outlined in the algorithm section is more efficient than the more naive approached of broadcasting to normal and high-priority waiters alike each time a task leaves the bus and then letting them compete, as it reduces the chance of waking threads unnecessarily.

Addendum

The testfile `batch-scheduler.c` had to be modified with a `timer_sleep()` statement at the end, or it would simply exit and terminate the emulation prematurely, before any threads had time to traverse the bus.