Programming Assignment 2

CSE 3320.002

Due: October 26, 2016 11:59PM

Description

After years of construction on Cooper St., Arlington has decided to replace all vehicle traffic on Cooper with a rail system. The *Mobile Autonomous Vehicle* (MAV) system will replace automobile traffic around the UTA area. You will be writing code for a monitor to manage MAV flow through a four-way intersection.

Each MAV is a small electric train that travels on separate tracks back and forth between stations. At peak capacity the City of Arlington plans to run 255 MAVs simultaneously.

Due to MAV train space, power, and passenger constraints intersections require special handling:

1. At most one MAV can cross an intersection at a time.
2. MAVs arriving from the same direction line up behind the first MAV already at the crossing and cross in that order.
3. For safety of passengers and pedestrians each MAV takes 10 seconds to cross the intersection
4. Traffic at the crossing on the right has the right of way, e.g. with a northbound MAV and a westbound MAV waiting at the intersection the westbound MAV has precedence over the northbound MAV.
5. Simultaneously arriving MAVs, approaching from opposite directions can be deadlocked at a crossing and will need to be mediated by the MAV monitor
6. MAVs arriving from the right always have the right of way (unless the waiting MAV receives a signal to go);
7. Deadlocks have to be prevented
8. Starvation has to be prevented

In our system each MAV shall controlled by a separate thread. It is your task to modify the provided MAV Monitor and MAV code, to control traffic flow and prevents deadlock at a crossing. Part of the solution is to use one mutex lock for the crossing, so that at most one MAV at a time can pass. The mutex lock will act as a monitor for the MAV operations at the crossing. MAV trains that occupy the intersection simultaneously will collide causing passenger injury, and worse, a failing grade.

Besides the mutex lock, you will also need a set of condition variables, and their associated mutex, to control the synchronization of the MAVs. You need a condition variable per direction to queue and release MAVs arriving from each direction (Northbound Queue, Eastbound Queue, Southbound Queue, Westbound Queue).

All MAVs on the right have precedence. However, using this rule can cause starvation if too many MAVs are queued in one direction. To prevent starvation, when a MAV is waiting to cross but MAVs have continuously been arriving and dispatched from the right have had the right of way for five consecutive crossings, you will let a MAV that just passed the intersection to signal a waiting MAV on its left.

When deadlock occurs the MAV Manager must signal one of the MAVs to proceed, e.g. in the case of a north/south deadlock the northbound MAV. In the case of the east/west deadlock the eastbound MAV.

You will need a count for each direction to keep track of the number of MAVs waiting in line and potentially need more condition variables to control right-of-way.

Since the MAV trains take 10 seconds to traverse the intersection there is a limit of 8640 crossings per day. Your code will be tested with 8640 events.

Data File and Testing

The program will take, as input, a data file, indicating a sequence of arrivals of MAVs from the four directions.

The data files used to test your program consists of up to 8640 intersection arrivals events consisting of three integers in the following format:

```
[TIME IN SECONDS SINCE MIDNIGHT] [MAV TRAIN ID] [DIRECTION OF TRAVEL]
0 0 N
0 1 E
10 3 E
11 4 W
```

File Format

- Time in seconds since midnight - A positive integer specifying the arrival time of the train, in seconds since midnight. Valid range is 0-8640.
- MAV Train ID - A positive integer specifying the ID number of the arriving train. Valid range 1-255.
- Direction of travel - The direction of travel of the train represented by a character:

  N - northbound

  E - eastbound

  S - southbound

  W - westbound

  This data field is case insensitive. ’N’ and ’n’ are equivalent.

The input file handling code will be provided to you as well as a few sample MAV train schedules. To fully test your application you will need to create additional data files.

The code you are provided already outputs the proper text. **Important note**: Your application test cases will be graded autonomously comparing your output to my output. You must not change the provided output. The only output your should add is the DEADLOCK statement below.

Example output:

```
$ ./mavmon schedule.txt 1
Current time: 0 MAV 1 from North arrived at the intersection
Current time: 0 MAV 1 from North entering the intersection
Current time: 9 MAV 1 from North leaving the intersection
Current time: 11 MAV 8 from East arrived at the intersection
Current time: 11 MAV 8 from East entering the intersection
Current time: 22 MAV 9 from East arrived at the intersection
Current time: 22 MAV 2 from West arrived at the intersection
Current time: 22 DEADLOCK detected. Signaling train 2 to enter the intersection
```

Nonfunctional Requirements

Your source file shall be named msh.c.

Your source files must be ASCII text files. No binary submissions will be accepted.

Tabs or spaces shall be used to indent the code. Your code must use one or the other. All indentation must be consistent.

No line of code shall exceed 100 characters.

Each source code file shall have the following header filled out:

All code must be well commented. This means descriptive comments that tell the intent of the code, not just what the code is executing.

When in doubt over comment your code.

Keep your curly brace placement consistent. If you place curly braces on a new line , always place curly braces on a new end. Don't mix end line brace placement with new line brace placement.

Each function should have a header that describes its name, any parameters expected, any return values, as well as a description of what the function does. For example :

Remove all extraneous debug output before submission. The only output shall be the output of the commands entered or the shell prompt.

Getting the source

To get the code for assignment 1:

1. Change to your `cse3320-projects` directory

2. Enter the command:

   ```
   git pull /usr/local/share/CSE3320/cse3320-projects/
   ```

   Do not enter git clone.

3. The source for the assignment will be in the Assignment2 directory.

Building the source

1. Change to your `cse3320-projects/Assignment2` directory.

2. Type: make

Running the executable

1. Type: ./mavmon [data file] [time warp] and press <return>. [data file] is the data file holding your train schedule. [time warp] is an integer from 1 to 1000000. A value of 1 means 1 second in real world time equals 1 second in the simulation. A value of 1000000 means 1 millisecond in real world time equals 1 second in the simulation.

Grading

The assignment will be graded out of 100 points. Code that does not compile will earn 0.

How to submit homework

1. From your cse3320-projects/Assignment2/ directory type:

make submit assignment=2

Grading rubric

Requirement	Points Possible
No deadlock	15
No race conditions	10
Right of way	15
Starvation handled	15
Correct sequences for test cases	15
No train collisions	15
Code formatted correctly	5
No extra/debug output	5
Program submitted correctly	5
Total Points	**100**

Academic Integrity

This assignment must be 100% your own work. No code may be copied from friends, previous students, books, web pages, etc. All code submitted is checked against a database of previous semester's graded assignments, current student's code and common web sources. Code that is copied from an external source will result in a 0 for the assignment and referral to the Office of Student Conduct.

Hints and Suggestions

Consult the manual pages on cse3320 to understand the following library functions on mutex locks (pthread_mutex_t) and condition variables (pthread_cond_t):

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `int pthread_cond_destroy(pthread_cond_t *cond);`

Design

Your code will be contained in the following six functions:

```
void init( )
```

The `init( )` function is called upon application startup and should contain any initialization code your code needs.

```
void mediate( )
```

The `mediate()` function is called once a second. This should contain the logic of train routing. In this function your code will command trains to enter the intersection or wait.

```
void trainLeaves( uint32_t train_id, enum TRAIN_DIRECTION
train_direction )
```

The `trainLeaves()` function is called automatically once a train crosses the intersection. Any code needing to be run when a train leaves should be added here

```
void trainCross( uint32_t train_id, enum TRAIN_DIRECTION
train_direction )
```

The `trainCross()` function should be called when a train is free to leave the queue and enter the intersection.

```
void trainArrives( uint32_t train_id, enum TRAIN_DIRECTION
train_direction )
```

The `trainArrives()` function is called automatically when a new train arrival event occurs. All thread initialization should be placed here.

```
void * trainLogic( void * )
```

The `trainLogic()` function is the main function all threads should use when calling `pthread_create()`. Your train logic should call trainCross when the train is ready to enter the intersection.

For readability the following enumeration and char* array are provided in train.h

```
enum TRAIN_DIRECTION
{
  UNKNOWN = 0,
  NORTH   = 1,
  EAST    = 2,
  SOUTH   = 3,
  WEST    = 4,
  NUM_DIRECTIONS = 5
};

char * directionAsString[ NUM_DIRECTIONS ] =
{
  "Unknown",
  "North",
  "East",
  "South",
  "West"
};
```

You can print the value of the TRAIN_DIRECTION enumeration with the directionAsString[] array as follows:

```
printf("The enumeration NORTH can be printed %s\n,
          directionAsString[NORTH] );
```