TEC | Tecnológico de Costa Rica

Operating Systems Principles

Project #1

Animated ThreadBridge

Authors:
Sebastián González Quesada (2013030999)
Daniel Moya Sánchez (2013103444)
Willberth Varela Guillén (20130537925)
Giovanni Villalobos Quirós (2013030976)

Professor:
Alejandra Bolaños

14 of October, 2017

1 Introduction

1.1 Threads

The threads can be seen as "lightweight" processes that can separate work from an original process for efficiency. Each one of the threads still need data structures like address space, file descriptors, among others but the difference between normal processes is that threads can share those resources so they can be doing simultaneous tasks on different parts of the program. [2]
Nevertheless threads need their own stack, local variables and program counter, but their are called lightweight because they have a smaller context than regular processes, which implies that context switches between threads are cheaper. The POXIS threads specification provides a priority thread model, with the possibility of preemptive policies, signal handling and primitives to provide mutual exclusion as well as synchronized waiting. [4]
The implementation of threads in the *pthreads* uses a number of UNIX standard library routines and UNIX kernel calls and is driven by some key objectives, such as allow preemptability (e.g. in round-robin), enable a fast content switch between threads, have the smallest critical section time spent and keep an unlimited stack growth of event interruptions. [4]

1.2 Scheduling

In general, schedulers lead the resources to an specific work for it to finish. They often try to keep all the computer resources busy (trying to be fair and load balancing). A process scheduler decides which process runs at a certain time and could take the processor away (preemptive scheduler). [7]
The scheduling of threads is not different from the process scheduler, it involves managing states and events, for example, a thread may be blocked waiting for some event, ready to execute, running or terminated. Once a thread is selected to go from ready to running, the dispatcher is called and if there is another running thread then a context switch has to be performed. [4]
There are different scheduling algorithms like First Come First Served (FCFS), Earliest Deadline First (EDF), Short Job First (SJF), priority scheduling, Round-robin scheduling, multilevel queue scheduling, among others. Each vary in their implementation, complexity as well as their advantages and disadvantages. In general, they work by moving certain threads in a general queue (the ready list) according to an attribute which is prioritized (like time to execute), some of these algorithms are implemented in this project, as mention is section 1.3.

1.3 Project Specification

The project requires the re-implementation of the Pthreads library with the name mypthreads. In this case re-implementation means a facade for each function, for example mythread_join() is a function that calls the original pthread function pthread_join(). Each of the threads represents a car that wants to go though a bridge (resource), or a control device, such as an officer or semaphore. The following functions were implemented in the project:

- mythread_create
- mythread_end
- mythread_yield
- mythread_join
- mythread_detach
- mymutex_init
- mymutex_destroy
- mymutex_lock
- mymutex_unlock
- mymutex_trylock
- mymutex_setsched (new method)
- FCFS Scheduler
- Round Robin Scheduler
- Priority Scheduler
- Shortest Job First

- Real time Scheduler

There will be four different bridges, which will be controlled by the following methods:

- Controlled by a transit officer
- Controlled by a semaphore
- Not controlled at all (jungle law)

The cars will be generated at a certain rate determined by an exponential distribution, a random speed based on an average speed, and a type based on a percentage. The type of cards available are: normal cars, ambulances (will use a soft real time algorithm) and radioactive cars (will use a hard real time algorithm). All the program is parameterized by a configuration file.
The bridge should be simulated in software (GUI and messages at console level) and in hardware (with different LEDs colors) that report the different events and status of each of the bridges (in hardware there is one less bridge).

2 Development environment

The following tools were used in the development of this project:

- Sublime Text: It is a text editor with special features such as the Command Palette, split editing, excellence responsiveness and cross-platform [3]. It also provides the auto-completion and markup text, which greatly helps the programmer but does not bound it like a fully complete programming IDE.
- GCC: The GNU compiler which includes front ends for popular languages like C, C++ and libraries for these languages. It is free software and provides a regular, high quality releases. [5]
- SDL2 Library: The Simple DirectMedia (SDL) library provides a cross-platform layer designed to supply low level access to audio, keyboard, mouse, joystick and graphics hardware via OpenGL and Direct3D. It is written in C and works natively with C++ and has bindings to another languages. [6]
- Arduino IDE: It is an open-source free software that enables physical computing based on a simple I/O board and a development environment with a processing and wiring language. The code developed can be connected to computer software or work on its own. [1]

3 Continuous learning attribute analysis

3.1 Knowledge integration

This project is based on the re-implementation of pthread library so the first investigation that had to be made was about pthreads. We learned a lot about this because all the members of our team have never been to involved in this topic. We learned that pthreads are implemented in the basic C library, so it was necessary to download it to build it again and make test to avoid damage to the system C library. Inside the C there are different modules oriented to different functionalities, the module oriented to thread manage is the nptl module. Inside nptl we found the pthread library where we had to mask the methods. All of this structure of the C library was always a gray area but with this project we learned how it was. To build the C library we learned that it used autotools, so we had to run it using the previous knowledge about that tool to build it.

3.2 Solutions to problems

To solve the problem of the re-implementation of the pthreads a linux command was used to discovered what files we had to modify in order to include new functions and don't mess with the others. The command grep allow us to search for words that represent functions already included in the library and this let us know what files we had to modify. The idea was to look for a existing method, such as pthread_create or pthread_detach into the glibc library files and see where the name appears. The command was, for example: " grep -rnw ./ -e 'pthread_detach' ", we could see the command use the flags -rnw, the r means to search recursively in all subdirectories of the actual directory, the n means to numerate the lines of the file where it matches the pattern, the w means to search for only whole words and the e flag let us give the pattern to search in this case we searched for the word 'pthread_detach'.
Another problem that involved the pthreads was at the time of compilation, it gived segmentation fault, something very strange so investigating a little it was found that it may be the version of the library, we had in our linux glibc-2.23 and we had downloaded the 2.26, so that was the problem that we solved using the 2.23.

4 Program design

4.1 Software design

The design of the software side of the program is as follows:

- Pthread re-implementation: Following the scheme of the original pthreads, an extra file was added for each new method to be implemented; these new files mask the original functions of pthreads. The following files in pthreads library were modified so the new files would be recognized:
  - glibc-2.23/nptl/Makefile:
    In the line with the rule libpthread-routines = nptl-init vars events version pt-interp pthread_create pthread_exit pthread_detach **mythread_<name>**
  - glibc-2.23/conform/data/pthread.h-data:
    In the line 108 this line needed to be added **mythread_<name>(<arg1,arg2...>)**
  - glibc-2.23/posix/annexc.c:
    In the line 248 "pthread_condattr_setpshared", "pthread_create", "pthread_detach", **"mythread_<name>"**,...
  - glibc-2.23/nptl/Versions:
    In the line 45 the name of the new functions needs to be added in pthread_exit; pthread_detach; **mythread_<name>;**
  - glibc-2.23/sysdeps/unix/sysv/linux/sh/libpthread.abilist:
    In the line 119 it was need to add GLIBC_2.2 **mythread_<name> F**
  - glibc-2.23/manual/threads.texi:
    This is only for documentation purposes **@c mythread_<name>**
  - glibc-2.23/sysdeps/unix/sysv/linux/hppa/pthread.h:
    In the line 230 this needed to be added: **extern int mythread_<name>(<arg1,arg2...>)**
  - glibc-2.23/sysdeps/nptl/pthread.h:
    In the line 230 this needed to be added: **extern int mythread_<name>(<arg1,arg2...>)**

  The correspondingly files that were added for each new method are explained below, based on [2]

  - mymutex_destroy.c:
    This function eliminates the mutex object referenced by mutex
  - mymutex_init.c:
    Initialises the mutex referenced by mutex with attributes specified by attr
  - mymutex_lock.c:
    The mutex lock object referenced by mutex is blocked
  - mymutex_trylock.c:
    Almoust the same as lock but with the difference that if the locked object is locked then it returns immediately
  - mymutex_unlock.c:
    Function detach and release the mutex object that is specified in mutex;
  - mythread_create.c: This function works exactly as the pthread_create function,it starts a new thread in the calling process, but receives an extra argument that is the type of scheduler as was asked in the specification.
  - mythread_detach.c:
    This implementation allow the thread to end and that it resources are automatically release again to the system.
  - mythread_end.c: The main function terminates calling the thread and return a value that is specified in the retval attribute, is available to another thread in the same process that calls pthread_join.
  - mythread_join.c:
    Waits till the thread that this function is called for, terminate and if it already has then it returns immediately.
  - mythread_yield.c:
    This function makes that the thread leave the processor and this same thread is put at the end of the running queue.
  - mymutex_setsched
    This function changed the scheduler that was initialized the thread for the other specified in the parameter.

- Parser: The first method used is the *getDataConfig*(), this loads the configuration file. Once this is done, the right bridge is selected to assign the corresponding values. Because only a whole line is loaded at a time, the method *strSplit*() has to be used to separate the values from the attributes. With the function *assignedVarBridge*() the values are assigned in the attributes of each bridge structure. Lastly, the method to handle the values at the main function (*main.c*) is *getVarBridge*; it receives an specific number bridge and returns a structure with all the bridge attributes.
- Manager threads structure: The threads and all their data was stored and moved (accordingly to a specific schedule algorithm) in a list, with the following functions to access or store elements:
  - push: Adds an element to the list; first it validates if the list is empty, if so it adds the element as the first and last one, else it adds the element to the front of the list and the ex-former element will now be the second one.
  - append: Operates similar to the push method, but now the element is added at the end of the list.
  - pop: Get the first element of the list out (and returns it) and reassigns the new first element.
  - insert: takes an specific position, validates it, and if it exists, a new element is added at that position and the old element is pushed back in the list.
  - searchNodo: It receives a specific value which is compared to an specific parameter (which is also entered) of each of the nodes in the list, if it matches, the node is returned.
  - search[parameter]: The [parameter] options are: PositionId, PositionSpeed, PositionTimeLim and PositionPriority. The value entered is compared with the specific attribute of all nodes, and if it matches the proper index of that node is returned.
  - listaVacia: Returns 1 if the list is empty and 0 if not.
  - get_length: Returns the length of the list.
- Cars generation: Different attributes distinguish one specific thread from another one, these attributes are implemented as described below:
  - NextSpawnTime: Specifies what time has to pass until another car (thread) is created. This is calculated by the exponential equation:

$$\text{NextSpawnTime} = \lambda e^{-\lambda x} \tag{1}$$

    where λ is:

$$\lambda = \frac{\log(2)}{\text{median}}$$

    And *median* is a parameter obtained directly from the configuration file. The variable x in equation (1) represents the time starting when the first thread is created, which means that the more time passes, the faster the frequency with which the cars are generated.
  - Speed: Determines how fast a car is, which changes the time it needs to pass a whole bridge. The value is calculated from a central value as follows:

$$\text{Speed} = \text{average} \pm \text{oscillation} \tag{2}$$

    where *oscillation* represents an internal parameter to establish how much a certain speed can variate from the central value. In the tests done the value used was either 1 or 2.
  - Type: Describes whether the car is a normal car, an ambulance or a radioactive car. The parameters of percentage of ambulances and percentages of radioactive cars are used to determine the type of car. This is done by generating a random number between 0 and 100 and choosing the type of car depending on the range that the random number got in, where the ranges are calculated based on the percentages.
  - Id: Represents a sequential number starting from 1, which establish a creation order of the threads and also identifies each one of them.
- Scheduler and bridge control synchronization:

  In order to maintain the requirements of the project the abstraction of the scheduler problem was implemented in a way that the core of scheduler algorithm is abstracted in several functions.
  - runSched: This functions is implemented by a thread and it continuously checks, in a loop, if there area a bridge available to let the cars pass. It uses three flags: flag_bridge to check which direction the cars can pass, bridge_in_use that represents is some car is crossing the bridge right now and sincronizationOfficial a flag to check synchronization with transit officer.

- callSched: This is used to make an abstract representation of the times that a scheduler is called, receiving as parameters the characteristics of the new car created (if is the case in which a car created make a call with the flag NEW_READY) or if it is called when a car passed from READY_RUNNING (this means the call is made with the flag READY_RUNNIG as parameter) because other car terminate his execution. This function executes a call to the respective scheduler used.

For the scheduler activation functions already described, each bridge has two queues one at each side of it that represent the cars that are waiting to pass by. This queues are used for the schedulers to let the cars pass by the bridge depending on their individual algorithm. Each algorithm is separated in two main parts, depending on the way that is was called if it was by a new element arrival or a car terminate passing the bridge.

- New element arrival: This part of this algorithms is in charge of insert in the queue the new element, this can variate depending on the type of the algorithm and it will be explained later on.
- Bridge available to pass: This part of the algorithm creates a new thread with the first element of the queue representing that a new process can run, as the queue is already in order it just had to be created. This part of the algorithm is the same for all the methods.

The following schedulers were to be implemented:

- Round Robin Scheduler:
- Priority Scheduler:

  This scheduler was implemented in order to check for the priority of the cars that arrived to make them go first into the bridge. When is called with the flag NEW_READY it means it has to insert a new element with the characteristics already received as a parameter, but as it is an algorithm that stands on priority it have to insert the element into the position depending on their priority. This is where the functions of the Manager Thread are used in order to get the position where it should be inserted depending on his priority and then inserting it in that position.
- Shortest Job First:

  This algorithms makes the faster cars go first into the bridge. Because of this when a car arrives with more speed than the cars in the queue it has to go before them, to achieve this a method in Manager Thread allow of to get the position of the queue in which the car has to go and returned it, then with other function we can insert the element in that position.
- FCFS Scheduler: This algorithm let the first element that arrived to go first, this means that the elements in the queue just has to be appended at the end, this is made using the append function of manager thread.
- Real time scheduler:

  This is a real time algorithm this means that there are certain priority times that has to be achieved in order to accomplish the process time. So when a car arrived with the flag of NEW_READY it means depending on if it hard or soft different politics should be implemented. For the case of the Ambulance that is consider a soft case them the measure to take is that it is pushed at the front of the queue, for the case of the Radioactive car this measure is also taken but besides, it has to change the bridge control method implemented to make sure the time is reached, this means that it have to alter the behavior especially of the transit officer because the numbers of car of the algorithm are going to be reset every time a interruption of this time gets there. And also at the beginning of the arrival of this type of car if the speed is not enough according to the time_limit, then the speed is changed using a formula that gives the car the minimum speed to make it to the bridge in time.

The following bridge control methods were implemented:

- Transit officer:

  This method is implemented by only one function that is in charge of keep count of the cars and synchronize them with its signals with the arrival of the cars and also with the signal bridge_in_use that indicates if the bridge is available or not. As this function run as a thread then it need to synchronize its use with the others cars because in K cars have passed by and the next thread that arrives is not the transit officer, another car will try to pass by. So when the quantity of cars K have passed then the bridge will block in order to wait till the thread of the officer arrived.
- Semaphore:

  This thread is managed by time, a loop will run inside with a delay in time that emulates the semaphore change rate.
- Jungle law:

  This algorithm was implemented in the way that the driver itself are the ones that allow the others drivers to pass. So in the function we implement that when a car want to pass first it have to check that there are not other cars circulating into the bridge. So checking the flag bridge_in_use of each bridge. The algorithm let or not pass the next car in the queue.

Figure 1: Diagram of the physical bridges

- GUI: the GUI was designed with a relative high frequency of refresh so that it could constantly monitor the values from the logical layer and updating the images on the window as well. As a parallel concept, the GUI communicates with the rest of the program through polling. The following images (and their horizontal mirrors) were loaded previous to the execution of the program and display when necessary:
  - Background: the background image (figure 2) consists basically of a blue vertical rectangle, which represents the "river" that is under the bridge and which cars have to pass. The surroundings are not appreciated in this document because they are white.

Figure 2: Background image.

  - Basic bridge: the bridge image (figure 3) is used for the 4 bridges and its length varies according to the value *largeBridge* in the configuration file.

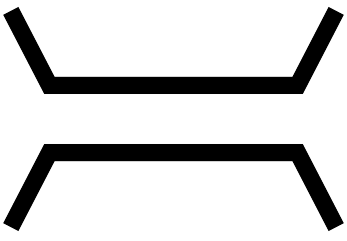

Figure 3: Bridge image.

– Normal car: the normal car image is showed in figure 4.

Figure 4: Normal car image.

– Ambulance: the ambulance image is showed in figure 5.

Figure 5: Ambulance image.

– Radioactive car: the radioactive car image is showed in figure 6. Because this was the largest image of all the 3 cars, in the final GUI the length was shortened a little.

Figure 6: Radioactive car image.

– Semaphore: the semaphore image with its red color is showed in figure 7, and with the green color in figure 8. There is a semaphore on each side of the four bridges and there is no "yellow" light.

Figure 7: Semaphore with red light image.

Figure 8: Semaphore with green light image.

– Transit Officer: the transit officer image who lets the cars pass is showed in figure 9, and the stopping the cars in figure 10. There is one officer at each side of the four bridges.

Figure 9: Transit Officer letting the cars pass.

Figure 10: Transit officer stopping the cars.

4.2 UML

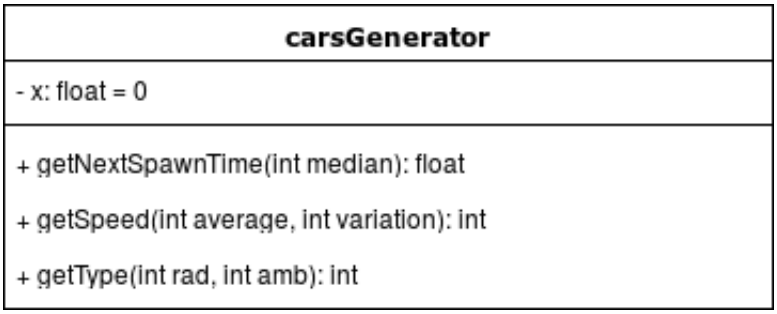

Figure 11: UML of carsGenerator.c file

Figure 12: UML of parser.c file

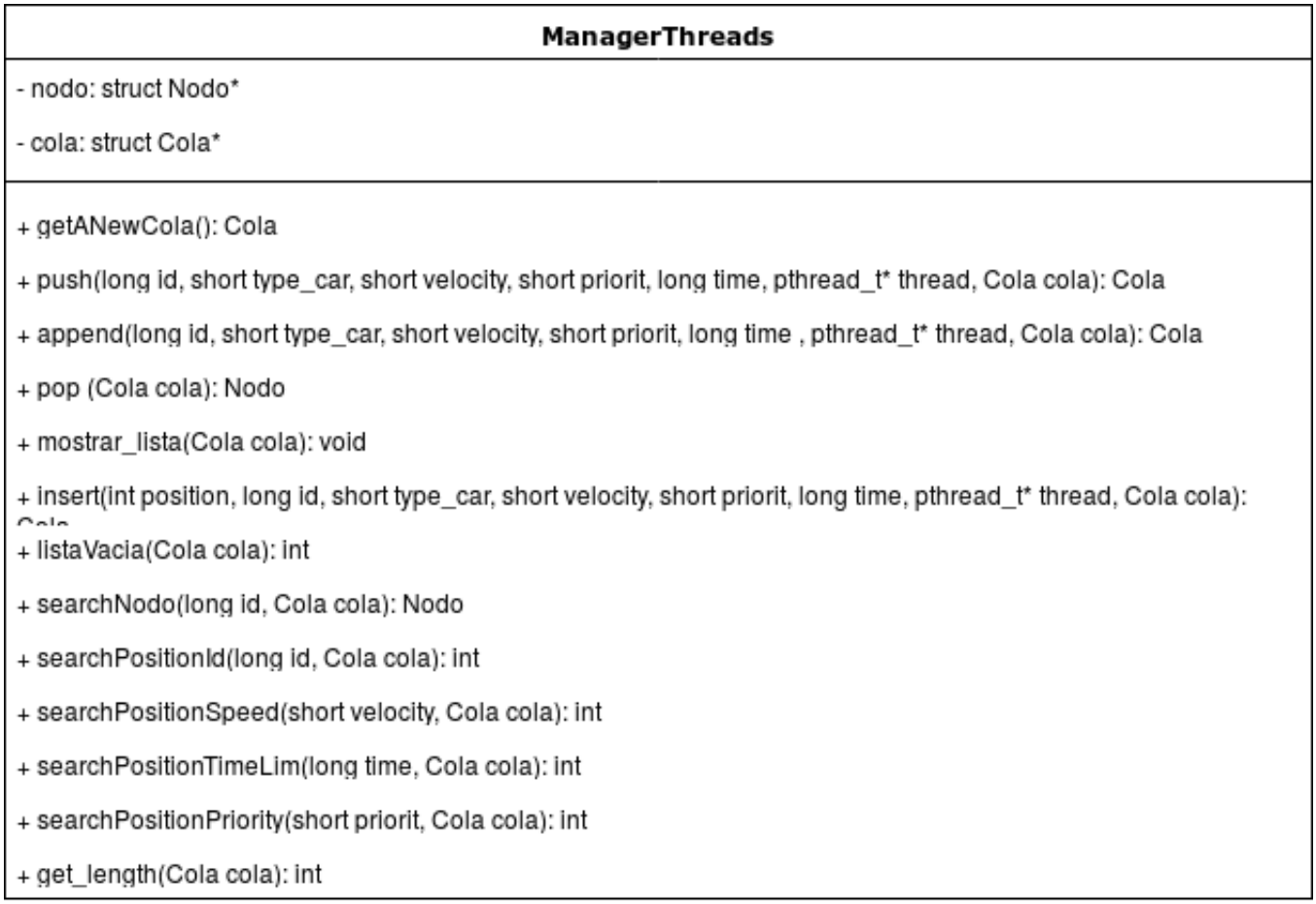

Figure 13: UML of ManagerThreads.c file

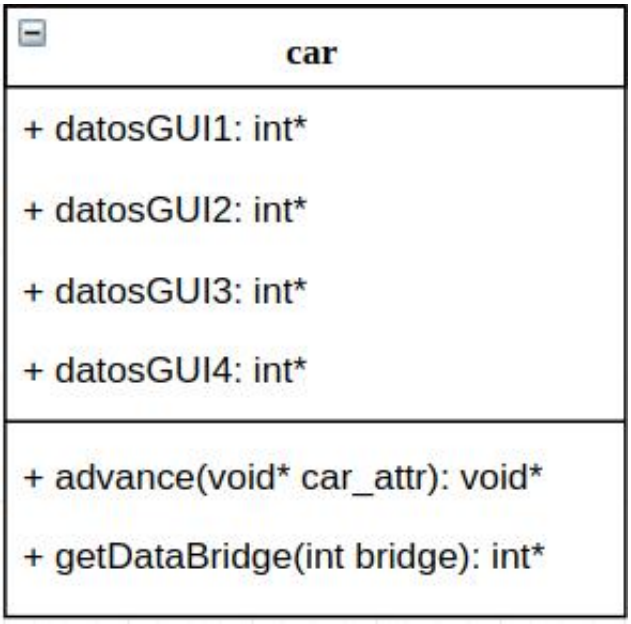

Figure 14: UML of car.c file

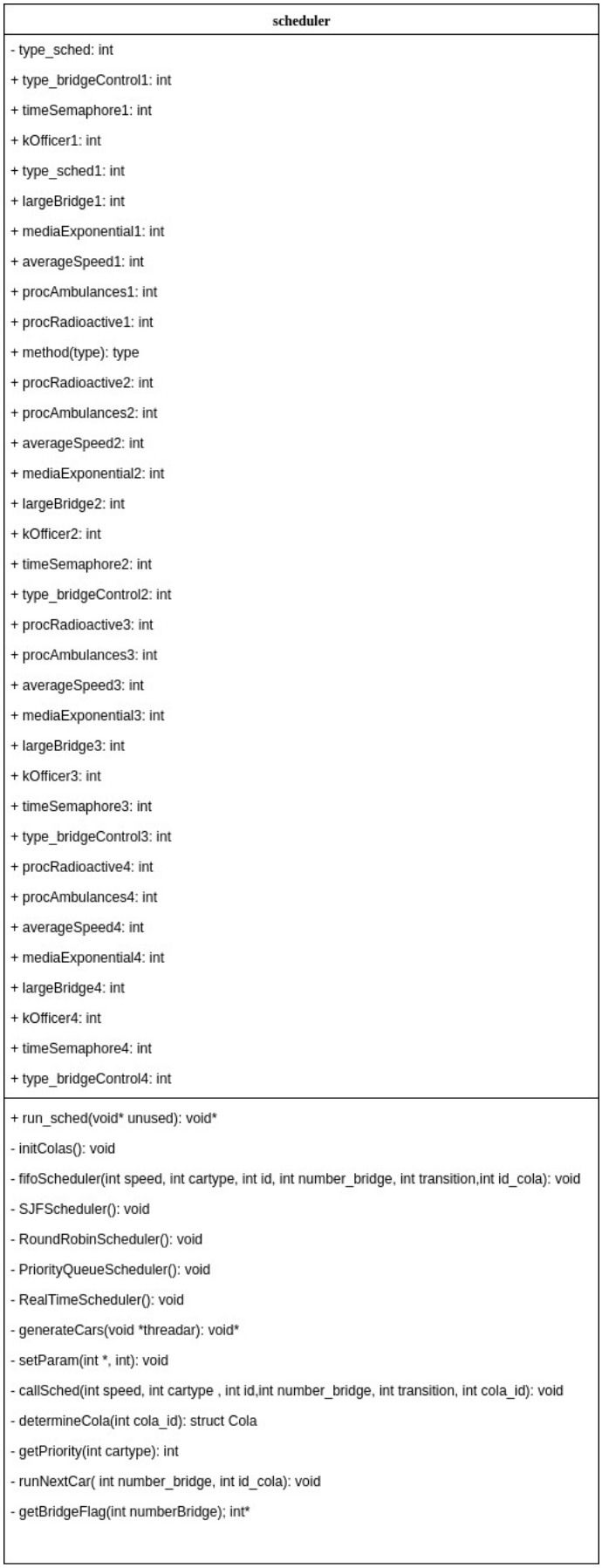

Figure 15: UML of scheduler.c file

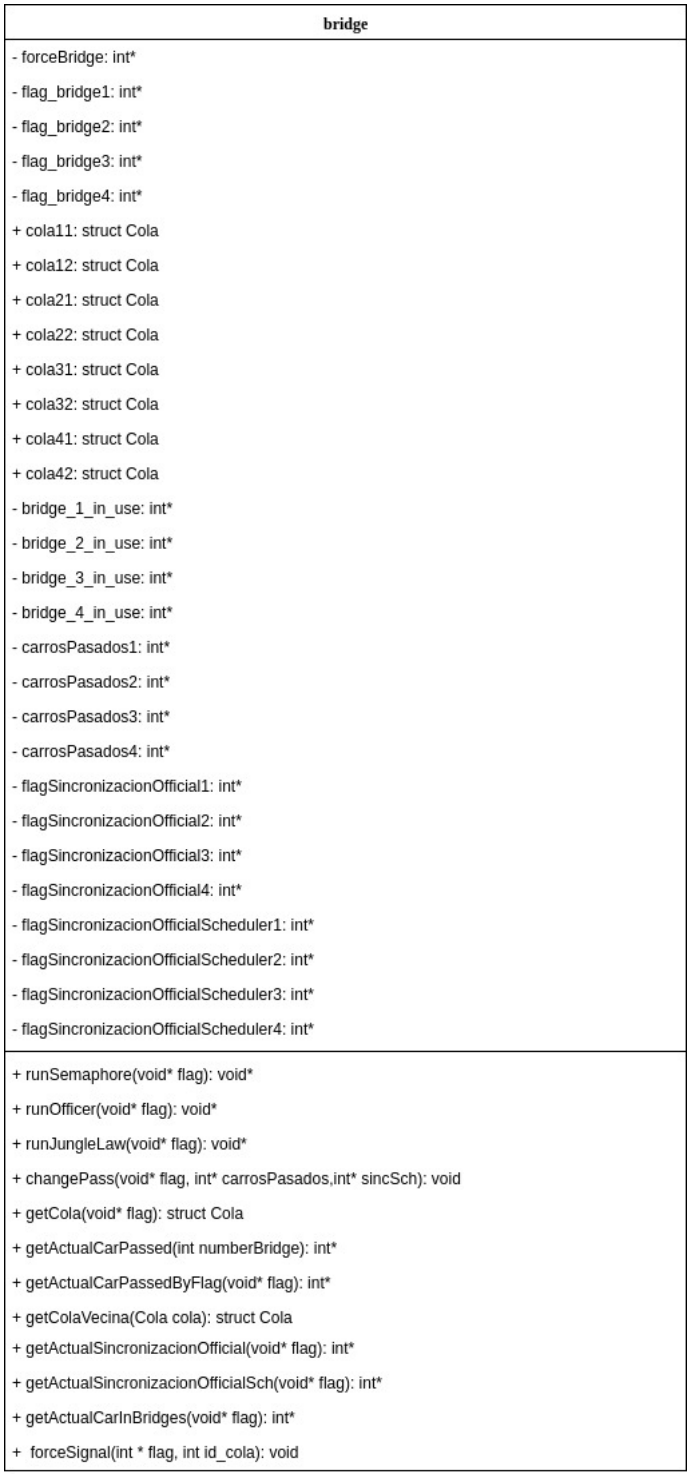

Figure 16: UML of bridge.c file

4.3 Physical Simulation

Arduino board was used to control the physical structure, in addition to this for the implementation of the bridge a wooden base is used along with cardboard to hold all necessary components in the hardware. To represent the bridge physically, pallets were used to realize the small structure of the bridge. To show how cars pass by the bridge of pallets a strip of LEDs which functionally serially was used, in which different cars are represented with different LEDs colors as showed in figure 17.

The first bridge works with the semaphore method, and it represents the semaphore with 4 LEDs, two at each side, red and green, to indicate the direction of the cars as showed in figure 18. The second bridge is controlled by a transit officer, which is represented by 2 LEDs, one at each side, blue. When one LEDs is turned on, that side of the bridge is allowed to pass, as showed in figure 18. Finally, the last bridge is ruled by jungle law, and because it does not use any intermediate, it does not have any extra LEDs. One important point to consider is that the frequency of the arduino is the lowest, and because

of this in some cases some steps are missed when printing the movement of a car.

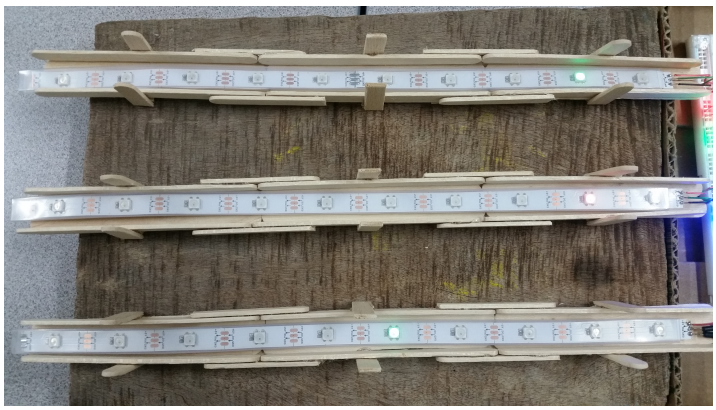

Figure 17: Physical representation of the bridges

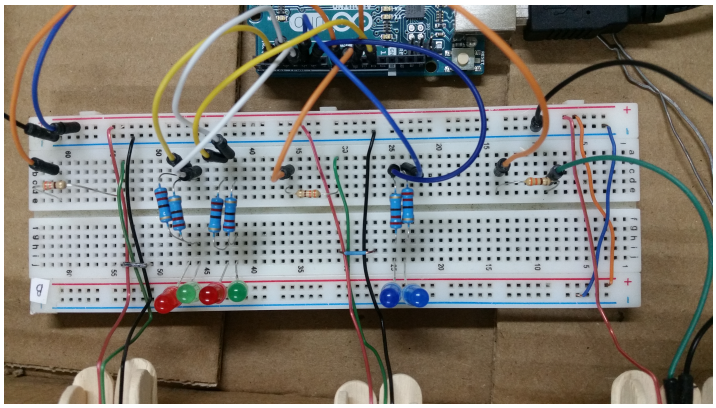

Figure 18: Representation of the semaphores and transit officer.

The arduino code uses a single method which initializes all the corresponding variables and begins the reading. Once the corresponding variables have been assigned, the validation of which LEDs to turn on and in what position, depending on the bridge and the direction from left to right or from right to left, starts in a cycle until the program ends. The communication was performed serially with a constant reading to update the variables as soon as possible in order to have equal representation in the GUI that in the physical part.
The main method of the program is constantly sendind data through the arduino-serial-lib.h library. For the configuration, first the port is defined by which the program is to communicate with the arduino, then it continues to initiate the communication with the port, if the connection is successful then it returns a zero, however in a given case that causes an error then the return is a -1. When the connection is already verified it is free to use the methods of the library in which it writes in the arduino or reads of the same, in this case only write was used.
Figure 19 shows the complete scheme of the project:

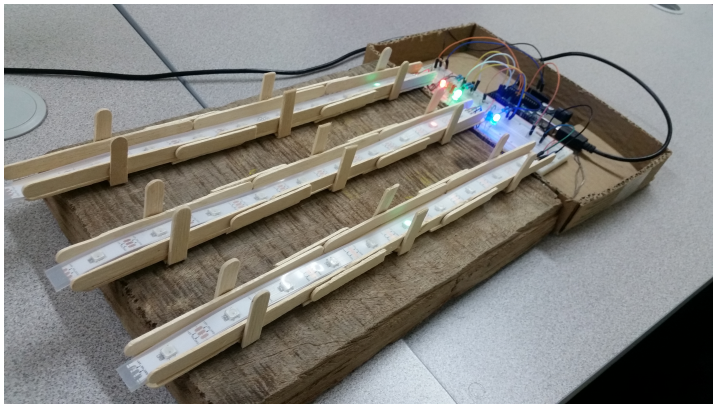

Figure 19: Model of the physical part of the project.

5 Instructions of how to use the program

1. SDL2 Library configuration:

   Run these commands to the install of the basic SDL2 library:

   ```
   sudo apt-get install libsdl2-dev
   sudo apt-get install libsdl2-image-dev
   ```

2. Pthreads set up: make sure you have the 2.23 version of glibc, you can verify this with the command:

   ```
   ldd --version
   ```

   Next download the glibc-2.23.tar.gz package from:

   `https://ftp.gnu.org/gnu/glibc/glibc-2.23.tar.gz`

   Extract it to your home directory and enter the *glibc-2.23* directory. From there, enter these commands:

   ```
   mkdir build
   cd build
   mkdir usr
   ../configure --prefix=/home/$(whoami)/glibc-2.23/build/usr  --enable-addons
   make -j$(nproc)
   make -j$(nproc) install
   ```

3. Set the parameters in the configuration file: For each of the four bridges, the following parameters need to be set correctly:

   - schBridge: Select between " Official " to have the access to the bridge controlled by a transit officer, " Semaphore " to have the access controlled by a semaphore of red and green lights and finally " Jungle " to have the access not controlled at all and follow a jungle law. It is important to leave a space between the quotation marks and the specific word.
   - timeSemaphore: Insert an integer bigger than 1, that represents the time in seconds the semaphore will last until it changes its state.
   - kOfficer: Insert an integer bigger than 1, that represents the number of cars that the transit officer will let pass from one side until it changes to the other side of the bridge.
   - schThreads: Select between " FIFO " to have a First Come First Served scheduler, " SJF " to have a Shortest Job First scheduler, " Round_Robin " to have a Round Robin scheduler, " Priority_Queue " to have a priority queue driven scheduler and finally " Real_Time " to have a real time scheduler. It is important to leave a space between the quotation marks and the specific word.
   - largeBridge: Insert an integer bigger than 1, that represents how large the bridge is. For the GUI and the physical simulation, we recommend a number between 5 and 10, so the passing cars can be appreciated.
   - mediaExponential: Insert an integer bigger than 1, that represents the media of the exponential function to determine how frequent the cars will be created. We recommend a number between 2 and 5.
   - averageSpeed: Insert an integer bigger than 2 (because there is a fixed variation of 1 and the speed needs to be bigger than 0), that represents the average speed of each car. We recommend a number between 2 and 5, to fully appreciate the cars going through the bridge. In case the physical mode is turned on, we recommend an average speed of 2 (the lowest) to fully appreciate the LEDs while moving, this is because the arduino is the slowest reading part of the program.
   - procAmbulances: Insert an integer between 0 and 100, that represents the percentage of ambulances that can be created.
   - procRadioactive: Insert an integer between 0 and 100 - procAmbulances, that represents the percentage of ambulances that can be created.
   - GUIon: Select between 1 to activate the GUI or 0 to not.
   - fisicPartON: Select between 1 to activate the physical simulation or 0 to now.

4. Compile the program: the project comes with a Makefile, so in the project directory just run:

   ```
   make
   ```

5. Run the program:

   ```
   ./main
   ```

6 Student activity log

This section present the activity log of each member of the group project.

6.1 Daniel Moya Sánchez

Date	Description	Time (hours)
19/09/2017	First meeting, initial small tasks were discussed and divided and had a general understatement of the project. The first doubts were analyzed with the professor after the class.	2
21/09/2017	Requirements extraction and group meeting for corresponding verification and individual assignments. Initial setup with the documentation template.	4.5
23/09/2017	Several examples and tutorials of the library SDL2 were done. The installation and make commands were documented correspondingly.	2
24/09/2017	Group meeting to clearly define the threads structure and discuss its methods. Code analysis of the small tasks divided, how to put it together and discussion of general format in code and project structure. SDL2 library investigation.	7
25/09/2017	The methods that return the attributes of speed, spawn of the next car and type of car were developed and properly tested (unit testing). Also the basic Makefile was created.	3
26/09/2017	Group meeting to define the whole structure of the project by distributing the functions needed in several source files. Some doubts arose and were later discussed in class with the professor.	3
27/09/2017	Group meeting to focus on the FIFO structure and the move of the different parameters needed. Initial difficulties were presented because there isn't the concept of objects in C language.	3.5
28/09/2017	Group meeting to continue analyzing how to properly pass the parameters between the scheduler, main, bridge and generator files. Doubts were clarified with a diagram for the professor to approve. Reorganization and putting together methods in main.c so a logical data path was followed.	5
29/09/2017	Group meeting to properly add the threads to the manager threads structure and unit testing about that part. Logical data path verified and analyzed.	2.5
30/09/2017	Group meeting to fully complete the FIFO method with the fully path from the configuration file data gathering through the set up and manipulation of the threads. Documentation worked and checked with the current status.	8
01/10/2017	Group meeting to divide the key remaining work. Coordination of how to extract the needed variables of threads into the GUI and how to properly visualize them.	4
02/10/2017	Group meeting to continue coordinating communication between logic and interface. Discussed the proper compile command and how to join all the current work. Worked on all the main methods to display GUI.	6
03/10/2017	Group meeting to test the GUI and logic union. Fixed several issues and agreed on how to extend the GUI methods.	4
04/10/2017	Added support to the GUI so that is shows the semaphore and inspector images in the right places and when the logical layer specifies it.	3
05/10/2017	Review of the documentation, added all the info that was possible at the time, like UML and GUI specifications.	3
09/10/2017	Reviewed the main method and the needed changes for the physical implementation to work with the GUI. Reviewed the documentation and added where it was possible.	3
12/10/2017	Added the remaining documentation, worked with the gpp, sha and time stamp requirements.	4
13/10/2017	Fixed minor bugs on the program, reviewed and updated the documentation file,also compared the requirements with the objectives achieved.	2

Table 1: Student activity log of Daniel Moya Sánchez.

6.2 Giovanni Villalobos Quirós

Date	Description	Time (hours)
19/09/2017	We had the first meeting of the project, it was basically understand the project and get some doubts about it, then we make like small jobs to be divided from the big ones and set the idea of the implementation. Later in class we asked the professor. Later that night I started investigating about the c library and pthreads, I founded it was inside the glibc library that was get from https://launchpad.net/ubuntu/+source/glibc/2.23-0ubuntu9 I build it and installed but it did not work.	5.5
20/09/2017	I found out that it was necessary to download other version using the command ldd –version and the version that my computer had was 2.23 and I was working with 2.26. I downloaded, build it and it worked fine. I made a small test of adding a printf inside a function and it worked.	3.0
21/09/2017	Group meeting to check the progress of everyone, continue testing the library.	3.5
22/09/2017	I started the job of adding new functions to the library but got stuck doing that because simply adding it didn't worked. I tried adding new source files but that failed too.	4.0
23/09/2017	In the morning the problem got solved by using the command grep and adding the function everywhere the pthread_create was. To follow the structure of pthread we added a file for each function. Then implemented the functions of mythread_create and mythread_end and made unit testing for the two.	4.0
24/09/2017	In the morning I added other functions that were mythread_yield mythread_join mythread_detach made unit testing for them, mythread_yield function was not recognized neither pthread_yield. After lunch we get together to set up more things about the project in general, we defined the structure of the thread structures, then end the test of the compilation of the pthread library.	6.0
25/09/2017	We had talked about the use of multicolor led and I had one at home that I had never used so I investigate about it and found out that it only needs VDD, GND and one pin for data input (that means one led per pin that allows multicolor RGB each). I found a library for arduino that help with the communication with the led and made a little program for testing the it and worked fine, but the price are almost as double as the normal multicolor led.	1.0
26/09/2017	We had a group meeting where we discussed some doubts we had of the project so we could ask the professor in the class, we agreed in the same doubts and made the basic implementation of the scheduler.c.	3.0
28/09/2017	We had a group meeting of 3 hours in the afternoon where we arranged to the way methods were passed to the scheduler.c and made some more doubts to ask the professor in the class. Later that night I tested the strip of lights we bought and it worked fine, allowing one pin per strip of lights making that easier for us and cheaper to use only one arduino. After class our doubts were cleared then in the night I implemented the modifications to the managerThreads.c . Now every method in this class has new parameter, a cola struct (also new), so we can use this class with any quantity of colas.	5.0
28/09/2017	We get together to continue making the FIFO algorithm, we successfully inserted cars in the managerThread list.	5.0
30/09/2017	We get together to continue with the FIFO and manage to complete it but we had a problem with the semaphore signal. And started making the arduino to C connection.	7.0
01/10/2017	Group meeting where we decided to divide some work to improve the progress of the proyect, I will keep making the scheduler algorithms.	4.0
02/10/2017	Group meeting. Started making the Traffic officer, but got synchronization problems that could not be resolved.	4.0
03/10/2017	Working at home in the morning, managed to end the Traffic officer.And group meeting in the afternoon	6.0
9/10/2017	Reunion at the library where we started implementing the Real_Time Scheduler	6
12/10/2017	Reunion at the library where we ended the implementation the Real_Time Scheduler	6

Table 2: Student activity log of Giovanni Villalobos Quirós.

6.3 Sebastián González Quesada

Date	Description	Time (hours)
19/09/2017	We meet at Library from 16:00 to 18:00 and talked about work metodology The meeting could be summarized as a deal of how workloads will be divided	2
21/09/2017	Gruop meeting at Library from 16:00 to 19:00 Individual homeworks were stablished for each member of the group, such as unit tests of.	3
22/09/2017	We tried to compile myprhtread_create() with the declaration inside pthread_create.c. Also modifying the files listed: glibc-2.23/sysdeps/unix/sysv/linux/hppa/pthread.h glibc-2.23/sysdeps/nptl/pthread.h However we had no success.We made a research about the following functions: mythread_create, mythread_end, mythread_yield, mythread_join, mythread_detach, mymutex_init, mymutex_destroy, mymutex_lock, mymutex_unlock, mymutex_trylock.	3.5
23/09/2017	We could compile the library pthread for a new helloworld function: Using: $grep -rnw ./ -e <pthread function> I discover where it is necessary to include the new functions and that we have to create a new src file for each one. In pthread each function has a src C file, thereby we have had to create new C files. The library files specified in section 4.1 were modified	7
24/09/2017	Group meeting. We added the folders gclib/nptl gclib/conform gclib/sysdeps and gclib/posix to the repository.	8
26/09/2017	Group meeting at library, from 16:00 to 19:00. We defined the global project structure seen by a function distribution on source files.	3
27/09/2017	Group meeting at LabH. From 19:00 to 22:30. We were creating the FIFO schedul- er structure and generating cars from two generators that ran in separate threads. We had problems creating two queues because C is not an OOP language.	3.5
28/09/2017	We had a reunion at library of 3 hours. All we do was to reorganize the logic of functions and source files in an easy way to comprehend. We defined that the scheduler file will see the cars generators and queues of each bridges. Also that main.c is the file that will create every instantiation of threads. At the end of the day we have made some questions to the professor.	3
30/09/2017	Group meeting at Library since 10:30 until 6:00pm We had take Daniel's modification and structure for FIFO implementation Today we construct the scheduling structure of all the functions, but some ones as SJF, Priority, Round Robin and Real time are empty. Only FIFO is implemented in 80%	3.5
1/10/2017	We made a brief reunion to define who will get intro hardware implementation Also made some code explanation for the partners that didn't know the software I resolved the FIFO and semaphore problem removing a delay in the scheduler thread.	3
2/10/2017	Reunion at library from 2:00 until 20:00 I reimplemented the the last mypthread required method: mythread_setsched We talked about the communications between scheduler, gui and hardware For each bridge we defined a global variable that defines position of the car in the bridge, direction and kind of car.	6
3/10/2017	Reunion at library from 2:00 until 20:00 We first solved another segmentation fault that appeared when we added the GUI interface. Then we solved some details of the bridge customization, such as the different kind of cars and the lenght of the bridge. The Gui is now working.	6
9/10/2017	Reunion at the library where we started implementing the Real_Time Scheduler I added the glibc functions corrected to the repository.	6
14/10/2017	I have conclude some documentation parts and have done some final tests for verifying functionality.	6

Table 3: Student activity log of Sebastián González Quesada.

6.4 Willberth Varela Guillén

Date	Description	Time (hours)
19/09/2017	Group meeting in which a possible design of the hardware was planned defining materials, also the project was defined and the realization of the same. Certain objectives were defined that had to be asked to the teacher to clarify and to be able to realize the project.	3
20/09/2017	List of all the variables needed for the scheduling algorithms was obtained.	2
21/09/2017	Group meeting to overview progress and defined how to perform scheduling algorithms. Individual tasks were also allocated.	3
22/09/2017	I made the parser and the structure of the configuration file, also I made an investigation for the structure of the handling of the threads.	3.5
23/09/2017	I made the basic structure of threads with the necessary methods for handling the threads	3.5
24/09/2017	Group meeting in which the structure was defined, the thread manager will have as well as validated that the methods are correct. I corrected the methods for the structure where you enter all the data that the structure occupies.	6
26/09/2017	Group meeting in which a basic implementation of scheduler.c was done, in addition to the use of the same with the bridges to present this design to the teacher and to validate that this well the design.	3
27/09/2017	Group meeting where a basic test was carried out in which the carts are generated, according to a thread and can be added to the thread structure, also the structure was redesigned to present to the teacher.	3
01/10/2017	Group meeting where you raised what is missing from the project and divide the tasks to work. I add the description of the methods of the thread manager in the documentation.	4
02/10/2017	I did split the data, complete the implementation of a bridge, define the hardware structure correctly, wrote the functions of the thread manager in the documentation.	8
03/10/2017	Perform the split validations with respect to the corresponding variables of each bridge. I changed the code for the elaboration of the three bridges	4
09/10/2017	Group meeting, to see progress and what is missing, i changed the main in order to add the physical part to the program, and see what variables was occupied	3.5
11/10/2017	Finish defining the main with the necessary attributes for the submission to the physical part, partition to the program the physical part and the GUI part. Add to the documentation of the physical part and photographs of the final model.	4.5
12/10/2017	Group meeting, modify some methods and variables in the Arduino code. Add remaining UMLS to the documentation. Refine details with documentation.	6

Table 4: Student activity log of Willberth Varela Guillén.

7 Project final status

Everything but the Round Robin scheduler was implemented accordingly to the project specification.

8 Conclusions

At the end of this project, the following conclusions can be given:

- The Pthreads Library is extensive and its manipulation has to be done carefully. The "my_thread" methods helped to understand the nature of the threads and a lot of their implementation implications.
- It is necessary to define the process states for invoking the scheduler when the transitions occur between states.
- Synchronization is critical condition when working with threads.

- The data structure to handle the threads was key in the implementation of all the scheduler methods. The proper control of this structure facilitates its manipulation and proper reading.
- The structure to handle the threads was implemented with a simple queue because it is easier to manipulate their methods, for example, append, insert and search and adapt them according to different schedulers.
- The current physical implementation allowed to use only one Arduino PLC, since each bridge only supports one type of control and can be coordinated with the configuration file. This saves a lot of hardware and costs for the team.

9 Suggestions and recommendations

The following recommendations can be mentioned:

- This project is forced to be implemented in C code, but an object language implementation could be more code-efficient. The re-use of certain logic could had been avoided with the use of instantiation, for example.
- Other data structures could be used to handle the threads, for more efficient parameter search in the nodes, for example a double linked list.
- The use of another embedded system than the arduino, could had improved the frequency at witch the data is read from the main program, eliminating any missing data from the LEDs printing function.

References

[1] Arduino. *Arduino - Software*. 2017. URL: `https://www.arduino.cc/en/Main/Software` (visited on 09/30/2017).

[2] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[3] Sublime HQ Pty Ltd. *Sublime Text A sophisticated text editor for code, markup and prose*. URL: `https://www.sublimetext.com/` (visited on 09/29/2017).

[4] Frank Mueller et al. "A Library Implementation of POSIX Threads under UNIX." In: *USENIX Winter*. 1993, pp. 29–42.

[5] GCC team. *GCC the GNU Compiler Collection*. 2017. URL: `https://gcc.gnu.org/` (visited on 09/29/2017).

[6] SDL Wiki. *Simple DirectMedia Layer*. URL: `https://wiki.libsdl.org/` (visited on 09/30/2017).

[7] Wikipedia. *Scheduling (computing)*. 2017. URL: `https://en.wikipedia.org/wiki/Scheduling_(computing)#Process_scheduler` (visited on 09/30/2017).