Car Park Management System and Simulator

CAB403 Systems Programming
Semester 2, 2021

Due date: 2021-11-01 (Monday, Week 14)
Weight: 40%
Group size: Up to 4
Specification version: 1.00

Overview

Your task is to develop and submit three pieces of software relating to a **car park management system**:

- A car park management system, henceforth referred to as the ‘manager’ - software that takes care of the automated aspects of running a car park, interacting with license plate readers, boom gates and electronic displays to ensure smooth operation and accurate reporting of the car park.
- A car park simulator, henceforth referred to as the ‘simulator’ - software that simulates all of the aforementioned pieces of hardware such that the manager can be tested without physical access to the actual car park systems. This also includes simulating the movement of vehicles around the car park.
- A fire alarm system for the car park, henceforth known as the ‘fire alarm system’ - software that interacts with temperature sensors within the car park and follows required procedures if a fire is detected. This component is regarded as a **safety critical system**, and while a functioning one has already been implemented (firealarm.c, available via Blackboard), your task is to assess it, write a report on its suitability as a safety-critical software component, and if necessary, fix or rewrite it.

These three pieces of software are all separate programs. They must be written in standard C (no other programming languages are permitted) and be compilable and runnable on the provided MX Linux virtual machine as-is. You will need to make use of the POSIX threads library (pthreads), POSIX shared memory and other libraries that you have used in the CAB403 practical classes to complete this assignment.

To simulate the interactions between the manager and the physical hardware present in the car park, a shared memory segment is used by the three programs to communicate- for example, the manager will check the status of the license plate recognition (LPR) sensors and control boom gates and digital signs by accessing this segment. A detailed description of the structure of this shared memory segment is available in the section titled ‘Shared memory’.

What you need to submit

You are required to submit, via Blackboard, a .zip archive (you can create these in MX Linux either by selecting the files in the GUI, right-clicking and choosing 'Compress', or by using the command line program 'zip') consisting of the following:

- The C source code (may consist of .c and .h files) of your manager, simulator and fire alarm systems.
- A Makefile which will automatically build the above three programs upon typing 'make'.
- Other files needed by your system (e.g. plates.txt)
- A report (in .docx or .pdf format - MX Linux comes with LibreOffice Writer which can be used to create this) consisting of the following information:
    - The names and student numbers of everyone in your team
    - A statement of completeness describing how much of the assignment has been completed and any issues (e.g. known bugs)
    - A statement of contribution detailing what each member of your team contributed to the project
    - An assessment of the safety-critical fire alarm system component. This in turn will consist of:
        - Identification of safety-critical standards that the provided firealarm.c fails (e.g..from NASA's The Power of 10, ISO 26262-6:2018, MISRA C)
        - Description of the approach you have taken to fix the problems (e.g. editing firealarm.c, completely rewriting it etc.)
        - Potential safety-critical concerns and reservations of your new implementation (for example, some safety-critical software guidelines cannot be followed perfectly - list these and how you have mitigated the concerns they pose.)

You are also required to submit a **video demonstration**, which should be no longer than 5 minutes in length, demonstrating your system in action. You can either upload a video file as part of your submission to Blackboard or upload it to e.g. YouTube and submit a link.

Car park structure

The simulator and manager work with an abstract model of a multi-level car park with 1-5 entrances, 1-5 exits and 1-5 levels, each of which can contain some maximum number of vehicles. For the purposes of this exercise, we will assume a car park of 5 entrances, 5 exits and 5 levels, with a capacity of 20 vehicles per level; however, you are **required** to include constants / preprocessor #defines to allow these to be easily configured. This way the client will be able to easily recompile the software to work with different car park configurations.

Cars seeking to gain entry to the car park queue up at one of the entrances. Once a car appears at the front of the queue, its license plate will be read by an LPR. The manager will then look that car up in its internal list of vehicles with permission to enter the park, and if this is granted, it will display the number referring to the level of the car park that the car should park at on the digital sign above the entrance and raise the boom gate. Each level of the car park also has an LPR detecting cars when they enter and exit. This is used to determine

where cars are within the car park for the purpose of ensuring incoming cars are only directed towards levels where there is free space available.
Cars seeking to leave the car park will head to one of the exits. Once they are detected by an exit LPR, the boom gate at the exit will open, allowing the car to leave.

It is assumed that, in this abstract model of a car park, none of the entrances or exits have any special affinity with each other or with the different levels of the car park, from a location / proximity perspective. They are simply used to allow more cars to enter/exit the car park at a time.

The roles of the **manager:**

- Monitor the status of the LPR sensors and keep track of where each car is in the car park
- Tell the boom gates when to open and when to close (the boom gates are a simple piece of hardware that can only be told to open or close, so the job of automatically closing the boom gates after they have been open for a little while is up to the manager)
- Control what is displayed on the information signs at each entrance
- As the manager knows where each car is, it is the manager's job to ensure that there is room in the car park before allowing new vehicles in (number of cars < number of levels * the number of cars per level). The manager also needs to keep track of how full the individual levels are and direct new cars to a level that is not fully occupied
- Keep track of how long each car has been in the parking lot and produce a bill once the car leaves.
- Display the current status of the parking lot on a frequently-updating screen, showing how full each level is, the current status of the boom gates, signs, temperature sensors and alarms, as well as how much revenue the car park has brought in so far.

The roles of the **fire alarm system:**

- Monitor the status of temperature sensors on each car park level
- When a fire is detected, activate alarms on every car park level, open all boom gates and display an evacuation message on the information signs

The roles of the **simulator:**

- Simulate cars:
    - A simulated car receives a random license plate (sometimes on the list, sometimes not) and queues up at a random entrance to the car park, triggering an LPR when it reaches the front of the queue.
    - After triggering the LPR, the simulated car will watch the digital sign. If the sign contains a number, it will keep note of that number (the level where the car has been instructed to park) and then wait for the boom gate to open. If the sign contains any other character, the simulated car will just leave the queue and drive off, disappearing from the simulation.
    - After the boom gate opens, the car will drive to the level it was instructed to drive to, triggering the level LPR in the process.
    - The car will then park for a random amount of time.
    - After the car has finished parking, it will leave, setting off the level LPR again. It will then drive towards a random exit. Upon reaching that exit, it will set off

the exit LPR and wait for the boom gate to open. Once the boom gate is open, it will leave the car park and disappear from the simulation.

- Simulate boom gates:
    - Boom gates take a certain amount of time to open and close. Once the manager has instructed a closed boom gate to open or an open boom gate to close, the simulator's job is to wait for a small amount of time before putting the boom gate into the open/closed state.
- Simulate temperature:
    - Each level of the car park has a temperature sensor, sending back the current temperature (in degrees celsius). The simulator will frequently update these values with reasonable random values. The simulator should also be able to simulate a fire by generating higher values, in order to test / demonstrate the fire alarm system.

Timings

Things take a long time to happen in a real car park - cars take a substantial amount of time to actually move around, gates take a while to open and so forth. For the purpose of this exercise, and to make running simulations less time consuming, the simulator and manager operate within an accelerated timescale, with delays measured in **milliseconds**. This means that it only takes a small amount of time for the car park to be occupied by many cars. You will need to use an appropriate high resolution sleep function (`usleep()` or `nanosleep()`) in order to invoke the appropriate delays. It is suggested that you create your own function that sleeps for a certain number of milliseconds. You can then, for example, apply a multiplier to the sleep duration calculated within that function in order to experiment with different timescales (for the purposes of debugging.) Your submitted code should use the timings indicated below (or similar timings), however.

Note that many of these delays will happen simultaneously - for example, the time that is spent by a simulated car moving around only delays the progress of that particular car, not other cars etc. For this reason it is highly recommended that pthreads is used, which will greatly simplify the work required to implement the simulator and manager.

Simulator timings

- Every 1-100ms*, a new car will be generated by the simulator with a random license plate, and will start moving towards a random entrance.

*Note that when a range is listed, this means the simulator needs to generate a random number in that range (e.g. between 1 and 100 inclusively) and then wait for that length of time. Note that we are not marking you based on the statistical correctness of your choice of random number generator - the stdlib.h rand() function is fine, for example (but keep in mind that you should protect calls to rand() with a mutex as rand() accesses a global variable containing the current random seed.).

- Once a car reaches the front of the queue, it will wait 2ms before triggering the entrance LPR.

- Boom gates take 10ms to fully open and 10ms to fully close.
- After the boom gate is open, the car takes another 10ms to drive its parking space (triggering the level LPR for the first time).
- Once parked, the car will wait 100-10000ms before departing the level (and triggering the level LPR for the second time).
- It then takes the car a further 10ms to drive to a random exit and trigger the exit LPR.
- Every 1-5ms, the temperature on each level will change to a random value

Manager timings

- After a boom gate has been fully opened, it will start to close 20ms later. Cars entering the car park will just drive in if the boom gate is fully open after they have been directed to a level (however, if the car arrives just as the boom gate starts to close, it will have to wait for the boom gate to fully close, then fully open again.)
- Cars are billed based on how long they spend in the car park (see the Billing section for more information.)

Fire alarm timings

- The fire alarm system will collect temperature readings every 2ms for the purpose of determining if a fire has occurred
- Once the fire alarm system is active, the character ‘E’ will be displayed on every digital sign in the parking lot. 20ms later, they will all show ‘V’, then ‘A’, ‘C’, ‘U’, ‘A’, ‘T’, ‘E’, ‘ ‘, then looping back to the first E again.

Permitted vehicle identification

When the manager is run, it will read a file in the current directory called `plates.txt` - this file will contain the license plates of permitted vehicles, one per line, like this:

```
029MZH
088FSB
174JJD
376DDS
451HLR
481WPQ
549QHD
594QVK
688QHN
931KQD
```

(A longer example plates.txt file is available on Blackboard.)
Whenever a vehicle triggers an entrance LPR, its license plate should be checked against the contents of this file. For performance / scalability reasons, the license plates need to be read into a hash table, which will then be checked when new vehicles show up. Using the hash table exercise from Practical 3 as a base is recommended, although not required.

If a vehicle shows up with a license plate not in `plates.txt`, the digital sign will display the character 'X' and the boom gate will not open for that vehicle.

Billing

Cars are billed at a rate of 5 cents for every millisecond they spend in the car park (that is, the total amount of time between the car showing up at the entrance LPR and the exit LPR). This is tracked per car and the amount of time, shown in dollars and cents, is written next to the car's license plate into a file when the car leaves (note that cars that are turned away at the entrance attract no such fee - the fee is only for cars that are accepted into the car park.) The manager writes these, line at a time, to a file named `billing.txt`, each time a car leaves the car park. The billing.txt file will be created by the manager if it does not already exist, and must be opened in **append** mode, which means that future lines will be written to the end of the file if the file already exists (this will avoid the accidental overwriting of old billing records). Here is an example billing file:

```
029MZH $8.25
088FSB $20.80
174JJD $14.95
376DDS $32.50
451HLR $11.00
```

Fire detection

The fire alarm system utilises temperature sensors to determine if a fire has occurred. Each temperature sensor (there is one on each level) returns a signed 16-bit integer containing the current temperature it is picking up. Because of the potential for noise and incorrect values being generated by the temperature sensor, the fire alarm system will smooth the data in the following way:

- For each temperature sensor, the monitor will store the temperature value read from that sensor every 2 milliseconds. Out of the 5 most recent temperature readings, the median temperature will be recorded as the 'smoothed' reading for that sensor, e.g:

Raw temperature	32°	30°	30°	29°	30°	40°	36°	32°	31°	29°	28°
Smoothed temperature	N/A	N/A	N/A	N/A	30°	30°	30°	32°	32°	32°	31°

  (As the table indicates, the first four values read in do not have a smoothed temperature. Smoothed temperatures are only available from the 5th value on.)

- The 30 most recent smoothed temperatures are then analysed (before 30 smoothed temperatures have been read, the fire alarm system cannot use this sensor to detect the presence of a fire.

The fire alarm system then uses two approaches to determine the presence of a fire- **fixed temperature** and **rate of rise.** If either of these approaches detects a fire, the alarm is triggered.

Fixed temperature fire detection

Out of the 30 most recent smoothed temperatures produced by a sensor, if 90% of them are 58°C or higher, the temperature is considered high enough that there must be a fire.

Rate-of-rise fire detection

Out of the 30 most recent smoothed temperatures produced by a sensor, if the most recent temperature is 8°C (or more) hotter than the 30th most recent temperature, the temperature is considered to be growing at a fast enough rate that there must be a fire.

For testing and demonstration purposes, your simulator should have the ability to generate both of these scenarios, to ensure that both successfully trigger the alarm.

Shared memory

The three processes communicate via a shared memory segment named PARKING (all caps). The PARKING segment is 2920 bytes in size. The simulator needs to create this segment when it is first started (the segment may already exist from a previous run, in which case the simulator creates it again, overwriting the old one. The manager and fire alarm system both open the existing PARKING segment when they are started (and will print an error message and exit if it is not present.) The segment contains space for 5 parking lot entrances, 5 exits and 5 levels.

Inter-process communication

Most of the values in the shared memory structure are accompanied by a **mutex** and a **condition variable**. These must be initialised by the simulator when the shared memory segment is created. Note that these mutexes and condition variables are accessed by different processes, which means they need to be **process shared**. By default, mutexes and condition variables are `PTHREAD_PROCESS_PRIVATE`, which means `pthread_mutexattr_setpshared()` and `pthread_condattr_setpshared()` need to be used to make these `PTHREAD_PROCESS_SHARED`, which will allow them to work correctly across multiple processes.

Mutexes are used to protect the integrity of shared data. The normal approach before accessing most values in the shared memory structure is to lock the associated mutex, read/write the value, then unlock the associated mutex.

Condition variables are used to allow threads to wait for a particular value to change in order to avoid busywaiting. **Busywaiting must be avoided in this assignment where possible** - for a description of busywaiting, see Appendix B. When changing a value, the associated

condition variable should be broadcast to, allowing threads that are waiting on that value to wake up and check it.

The process for waiting on a particular shared memory variable is usually the following:

- The thread that is waiting on a value (e.g. for a boom gate to switch to its 'open' state) first acquires the mutex associated with that value.
- The thread then waits on the associated condition variable, passing the mutex as the second parameter. This will also **unlock** the mutex while the thread is waiting.
- Once the value has been changed and the condition variable broadcast to, the thread will wake up, re-acquire the mutex and check the value again, before making a decision about whether to continue waiting or not.
- Finally, the thread will unlock the mutex and continue.

The temperature sensors and alarms are **not** protected by mutexes. Writes to them need to be atomic and the 'volatile' keyword in C should be used to ensure that reads and writes reflect the true contents of the shared memory segment.

Shared memory structure

Bytes 0-1439 contain space for 5 entrances, each 288 bytes large. Each entrance consists of the following:

- A license plate recognition sensor (96 bytes, at bytes 0-95)
- A boom gate (96 bytes, at bytes 96-191)
- An information sign (96 bytes, at bytes 192-287)

(Definitions for each of these appear later in this document, as some are reused in different areas.)

Bytes 1440-2399 contain space for 5 exits, each 192 bytes large. Each exit consists of the following:

- A license plate recognition sensor (96 bytes, at bytes 0-95)
- A boom gate (96 bytes, at bytes 96-191)

Bytes 2400-2919 contain space for 5 levels, each 104 bytes large. Each level consists of the following:

- A license plate recognition sensor (96 bytes, at bytes 0-95)
- A temperature sensor (2 bytes, at bytes 96-97)
- An alarm (1 byte, at byte 98)
- Padding (5 bytes, at bytes 99-103), not used for anything

Each license plate recognition (LPR) sensor is 96 bytes large and consists of the following:

- A pthread_mutex_t mutex lock (40 bytes, at bytes 0-39)
- A pthread_cond_t condition variable (48 bytes, at bytes 40-87)
- A license plate, 6 characters long (6 bytes, at bytes 88-93)
- Padding (2 bytes, at bytes 94-95), not used for anything

Vehicles pass in front of the various LPR sensors at different times depending on where the LPR is located:

- The LPR at the entrance will detect a vehicle approaching the boom gate
- The LPR at the exit will detect a vehicle approaching the exit boom gate

- The LPR on each level of the parking lot will detect any vehicle entering or exiting that level.

It can be assumed that, in a normal case, a vehicle will appear on LPRs 4 times- once at an entrance, twice on the floor the vehicle parks on, and once when exiting the parking lot. When a vehicle passes an LPR, the vehicle's license plate is written to the 6 character license plate field in that LPR and the LPR's condition variable is signalled. The mutex is used to protect the license plate data against concurrent access.

Each boom gate is 96 bytes large and consists of the following:

- A pthread_mutex_t mutex lock (40 bytes, at bytes 0-39)
- A pthread_cond_t condition variable (48 bytes, at bytes 40-87)
- The boom gate's status, 1 character long (1 byte, at byte 88)
- Padding (7 bytes, at bytes 89-95), not used for anything

There are 4 acceptable values for the boom gate's status:

- 'C' - Closed. This is the default value every gate should start in.
- 'O' - Open. Vehicles can only pass through the gate while it is open.
- 'R' - Raising. Boom gate is currently in the process of being raised. To open the boom gate, the manager sets the gate's status character to 'R' and signals the condition variable. The simulator will then set the status character to 'O' after 10 milliseconds.
- 'L' - Lowering. Boom gate is currently in the process of being lowered. To close the boom gate, the manager sets the gate's status character to 'L' and signals the condition variable. The simulator will then set the status character to 'C' after 10 milliseconds.

Note that the only acceptable status changes are as follows: C -> R, R -> O, O -> L, L -> C. Only the manager can change the boom gate's status from C to R or O to L and only the simulator can change the boom gate's status from R to O or L to C. Any other status changes are in error.

An information sign is 96 bytes large and consists of the following:

- A pthread_mutex_t mutex lock (40 bytes, at bytes 0-39)
- A pthread_cond_t condition variable (48 bytes, at bytes 40-87)
- The information sign's display, 1 character long (1 byte, at byte 88)
- Padding (7 bytes, at bytes 89-95), not used for anything

The information sign is very basic and only has room to display a single character. It is used to show information to drivers at various points:

- When the driver pulls up in front of the entrance boom gate and triggers the LPR, the sign will show a character between '1' and '5' to indicate which floor the driver should park on.
- If the driver is unable to access the car park due to not being in the access file, the sign will show the character 'X'.
- If the driver is unable to access the car park due to it being full, the sign will show the character 'F'.
- In the case of a fire, the information sign will cycle through the characters 'E' 'V' 'A' 'C' 'U' 'A' 'T' 'E' ' ', spending 20ms on each character and then looping back to the first 'E' after displaying the space character..

To show a new message on the sign, the display character is set and the condition variable broadcast to (to inform any processes waiting for the sign to change.)

The per-level temperature sensor is 2 bytes long and consists of a **signed** 16-bit integer representing the current temperature (in degrees celsius) detected on that floor. Note that the temperature sensors are not expected to be perfect and may occasionally contain incorrect values. This will be written to by the simulator and read by the fire alarm system (note that there is no mutex protecting this value).

The per-level alarm is 1 byte long and simply contains a value of 0 or 1 depending on whether the alarm is on or not (default is 0). Again, no mutex protects this value.

Some additional notes about the shared memory segment's layout:

- The size of the mutex (40 bytes) and condition variable (48 bytes) **are platform-dependent.** The values that have been given are for the MX Linux virtual machine where we will be testing your code. You are free to develop the code on the platform of your choice, but just make sure to keep in mind where your code will be run.
- You can directly address the variables via their memory address (e.g. `(pthread_mutex_t *)(ptr + 96)` to get a pointer to the mutex for the boom gate for Entrance #1), but it may be better to create a struct with the same fields to assist with layout.
- Appendix A contains the locations of every variable in the shared memory segment, which you might find useful as a reference

Unusual behaviour of vehicles

When a vehicle shows up at the entrance and is permitted entry, it is shown a level number and it is expected that the vehicle will go and park on that level. However, in reality, once the vehicle is inside the car park it can drive wherever it wants to go, including to a different level, or directly to the exit. A vehicle can even move between levels. For this reason, LPRs on each level are present, detecting a vehicle accessing other levels. Potential unusual activity of vehicles does need to be taken into account by the manager when determining how to assign vehicles to levels. This needs to happen in the following way:

- The car park keeps track of both the number of vehicles assigned to each level and the level that each vehicle is currently assigned to
- When a vehicle is directed by the digital sign to a level, that vehicle will be assigned to that level. Hence, if two vehicles arrive, one after another, they will not both be directed to a level that only has space for one more vehicle.
- However, if the vehicle is detected moving into another level by one of the per-level LPRs, it will then be counted as being assigned to that level **if there is room on that level**. That is, if a vehicle was directed towards level 1, it would initially count towards level 1's capacity. However, if that vehicle is then spotted by level 2's LPR, if level 2 has room for it, it will be subtracted from level 1's capacity and then added to level 2's.

Status display

When the manager program is run, the terminal it is run in should display a (text mode) screen showing the current status of the car park. The exact layout and formatting is up to you, but the following information is expected to be presented:

- Current state of each LPR, boom gate and digital sign
- Current state of each temperature sensor
- Number of vehicles and maximum capacity on each level
- Total billing revenue recorded by the manager thus far

The way the information is presented also has some requirements:

- The display should update frequently (e.g. every 50ms)
- To prevent visual fatigue from trying to process rapidly scrolling text, all the above information should be able to fit into a single screen. In addition, before printing out the current status, the screen should be cleared (e.g. with `system("clear")`), again, to prevent scrolling.

Video demonstration

You need to produce a short video demonstration showing your software in action, with most of the focus being on the manager program, given that its status display should be able to show you what is going on. You will want to demonstrate ordinary operation, as well as exceptional situations such as a spike in temperatures indicating the presence of a fire (and the response of the fire alarm system.)

Two examples programs that can be used to record a video fairly easily are Zoom and OBS Studio. With Zoom you can start up a meeting by yourself (or with other members of your team), share your screen, start recording the meeting and then demonstrate it that way.

The video is your way of demonstrating how your software is supposed to work, as well as showing that you've implemented the functionality you are required to. As a general rule, if you want to receive marks for a certain component in your software, you need to demonstrate it in the video.

Appendix A - Shared memory layout

- 0-39 pthread_mutex_t for LPR for Entrance #1 (40 bytes)
- 40-87 pthread_cond_t for LPR for Entrance #1 (48 bytes)
- 88-93 license plate reading for LPR for Entrance #1 (6 bytes)
- 94-95 padding (2 bytes)
- 96-135 pthread_mutex_t for boom gate for Entrance #1 (40 bytes)
- 136-183 pthread_cond_t for boom gate for Entrance #1 (48 bytes)
- 184-184 status character for boom gate for Entrance #1 (1 byte)
- 185-191 padding (7 bytes)
- 192-231 pthread_mutex_t for information sign for Entrance #1 (40 bytes)
- 232-279 pthread_cond_t for information sign for Entrance #1 (48 bytes)

- 280-280 character display for information sign for Entrance #1 (1 byte)
- 281-287 padding (7 bytes)
- 288-327 pthread_mutex_t for LPR for Entrance #2 (40 bytes)
- 328-375 pthread_cond_t for LPR for Entrance #2 (48 bytes)
- 376-381 license plate reading for LPR for Entrance #2 (6 bytes)
- 382-383 padding (2 bytes)
- 384-423 pthread_mutex_t for boom gate for Entrance #2 (40 bytes)
- 424-471 pthread_cond_t for boom gate for Entrance #2 (48 bytes)
- 472-472 status character for boom gate for Entrance #2 (1 byte)
- 473-479 padding (7 bytes)
- 480-519 pthread_mutex_t for information sign for Entrance #2 (40 bytes)
- 520-567 pthread_cond_t for information sign for Entrance #2 (48 bytes)
- 568-568 character display for information sign for Entrance #2 (1 byte)
- 569-575 padding (7 bytes)
- 576-615 pthread_mutex_t for LPR for Entrance #3 (40 bytes)
- 616-663 pthread_cond_t for LPR for Entrance #3 (48 bytes)
- 664-669 license plate reading for LPR for Entrance #3 (6 bytes)
- 670-671 padding (2 bytes)
- 672-711 pthread_mutex_t for boom gate for Entrance #3 (40 bytes)
- 712-759 pthread_cond_t for boom gate for Entrance #3 (48 bytes)
- 760-760 status character for boom gate for Entrance #3 (1 byte)
- 761-767 padding (7 bytes)
- 768-807 pthread_mutex_t for information sign for Entrance #3 (40 bytes)
- 808-855 pthread_cond_t for information sign for Entrance #3 (48 bytes)
- 856-856 character display for information sign for Entrance #3 (1 byte)
- 857-863 padding (7 bytes)
- 864-903 pthread_mutex_t for LPR for Entrance #4 (40 bytes)
- 904-951 pthread_cond_t for LPR for Entrance #4 (48 bytes)
- 952-957 license plate reading for LPR for Entrance #4 (6 bytes)
- 958-959 padding (2 bytes)
- 960-999 pthread_mutex_t for boom gate for Entrance #4 (40 bytes)
- 1000-1047 pthread_cond_t for boom gate for Entrance #4 (48 bytes)
- 1048-1048 status character for boom gate for Entrance #4 (1 byte)
- 1049-1055 padding (7 bytes)
- 1056-1095 pthread_mutex_t for information sign for Entrance #4 (40 bytes)
- 1096-1143 pthread_cond_t for information sign for Entrance #4 (48 bytes)
- 1144-1144 character display for information sign for Entrance #4 (1 byte)
- 1145-1151 padding (7 bytes)
- 1152-1191 pthread_mutex_t for LPR for Entrance #5 (40 bytes)
- 1192-1239 pthread_cond_t for LPR for Entrance #5 (48 bytes)
- 1240-1245 license plate reading for LPR for Entrance #5 (6 bytes)
- 1246-1247 padding (2 bytes)
- 1248-1287 pthread_mutex_t for boom gate for Entrance #5 (40 bytes)
- 1288-1335 pthread_cond_t for boom gate for Entrance #5 (48 bytes)
- 1336-1336 status character for boom gate for Entrance #5 (1 byte)
- 1337-1343 padding (7 bytes)
- 1344-1383 pthread_mutex_t for information sign for Entrance #5 (40 bytes)
- 1384-1431 pthread_cond_t for information sign for Entrance #5 (48 bytes)

- 1432-1432 character display for information sign for Entrance #5 (1 byte)
- 1433-1439 padding (7 bytes)
- 1440-1479 pthread_mutex_t for LPR for Exit #1 (40 bytes)
- 1480-1527 pthread_cond_t for LPR for Exit #1 (48 bytes)
- 1528-1533 license plate reading for LPR for Exit #1 (6 bytes)
- 1534-1535 padding (2 bytes)
- 1536-1575 pthread_mutex_t for boom gate for Exit #1 (40 bytes)
- 1576-1623 pthread_cond_t for boom gate for Exit #1 (48 bytes)
- 1624-1624 status character for boom gate for Exit #1 (1 byte)
- 1625-1631 padding (7 bytes)
- 1632-1671 pthread_mutex_t for LPR for Exit #2 (40 bytes)
- 1672-1719 pthread_cond_t for LPR for Exit #2 (48 bytes)
- 1720-1725 license plate reading for LPR for Exit #2 (6 bytes)
- 1726-1727 padding (2 bytes)
- 1728-1767 pthread_mutex_t for boom gate for Exit #2 (40 bytes)
- 1768-1815 pthread_cond_t for boom gate for Exit #2 (48 bytes)
- 1816-1816 status character for boom gate for Exit #2 (1 byte)
- 1817-1823 padding (7 bytes)
- 1824-1863 pthread_mutex_t for LPR for Exit #3 (40 bytes)
- 1864-1911 pthread_cond_t for LPR for Exit #3 (48 bytes)
- 1912-1917 license plate reading for LPR for Exit #3 (6 bytes)
- 1918-1919 padding (2 bytes)
- 1920-1959 pthread_mutex_t for boom gate for Exit #3 (40 bytes)
- 1960-2007 pthread_cond_t for boom gate for Exit #3 (48 bytes)
- 2008-2008 status character for boom gate for Exit #3 (1 byte)
- 2009-2015 padding (7 bytes)
- 2016-2055 pthread_mutex_t for LPR for Exit #4 (40 bytes)
- 2056-2103 pthread_cond_t for LPR for Exit #4 (48 bytes)
- 2104-2109 license plate reading for LPR for Exit #4 (6 bytes)
- 2110-2111 padding (2 bytes)
- 2112-2151 pthread_mutex_t for boom gate for Exit #4 (40 bytes)
- 2152-2199 pthread_cond_t for boom gate for Exit #4 (48 bytes)
- 2200-2200 status character for boom gate for Exit #4 (1 byte)
- 2201-2207 padding (7 bytes)
- 2208-2247 pthread_mutex_t for LPR for Exit #5 (40 bytes)
- 2248-2295 pthread_cond_t for LPR for Exit #5 (48 bytes)
- 2296-2301 license plate reading for LPR for Exit #5 (6 bytes)
- 2302-2303 padding (2 bytes)
- 2304-2343 pthread_mutex_t for boom gate for Exit #5 (40 bytes)
- 2344-2391 pthread_cond_t for boom gate for Exit #5 (48 bytes)
- 2392-2392 status character for boom gate for Exit #5 (1 byte)
- 2393-2399 padding (7 bytes)
- 2400-2439 pthread_mutex_t for LPR for Level #1 (40 bytes)
- 2440-2487 pthread_cond_t for LPR for Level #1 (48 bytes)
- 2488-2493 license plate reading for LPR for Level #1 (6 bytes)
- 2494-2495 padding (2 bytes)
- 2496-2497 temperature sensor for Level #1 (2 bytes)
- 2498-2498 fire alarm for Level #1 (1 byte)

- 2499-2503 padding (5 bytes)
- 2504-2543 pthread_mutex_t for LPR for Level #2 (40 bytes)
- 2544-2591 pthread_cond_t for LPR for Level #2 (48 bytes)
- 2592-2597 license plate reading for LPR for Level #2 (6 bytes)
- 2598-2599 padding (2 bytes)
- 2600-2601 temperature sensor for Level #2 (2 bytes)
- 2602-2602 fire alarm for Level #2 (1 byte)
- 2603-2607 padding (5 bytes)
- 2608-2647 pthread_mutex_t for LPR for Level #3 (40 bytes)
- 2648-2695 pthread_cond_t for LPR for Level #3 (48 bytes)
- 2696-2701 license plate reading for LPR for Level #3 (6 bytes)
- 2702-2703 padding (2 bytes)
- 2704-2705 temperature sensor for Level #3 (2 bytes)
- 2706-2706 fire alarm for Level #3 (1 byte)
- 2707-2711 padding (5 bytes)
- 2712-2751 pthread_mutex_t for LPR for Level #4 (40 bytes)
- 2752-2799 pthread_cond_t for LPR for Level #4 (48 bytes)
- 2800-2805 license plate reading for LPR for Level #4 (6 bytes)
- 2806-2807 padding (2 bytes)
- 2808-2809 temperature sensor for Level #4 (2 bytes)
- 2810-2810 fire alarm for Level #4 (1 byte)
- 2811-2815 padding (5 bytes)
- 2816-2855 pthread_mutex_t for LPR for Level #5 (40 bytes)
- 2856-2903 pthread_cond_t for LPR for Level #5 (48 bytes)
- 2904-2909 license plate reading for LPR for Level #5 (6 bytes)
- 2910-2911 padding (2 bytes)
- 2912-2913 temperature sensor for Level #5 (2 bytes)
- 2914-2914 fire alarm for Level #5 (1 byte)
- 2915-2919 padding (5 bytes)

Appendix B - Busy loops / busy waiting

Busy loops / busy waiting are something you need to avoid. A busy loop is when you have a loop that is waiting for something (e.g. for a value to change) , and what it does is continuously inspect the value waiting for it to change, using up large amounts of CPU time in the process. This is bad for battery life and bad for other processes trying to compete with your wasteful process on the same CPU.

The correct approach is usually to wait for something, e.g. by using a condition variable. When the variable you are waiting for is changed by some other thread, have it signal the condition variable, waking your waiting thread up such that it only checks the value when there is a reason to believe that value might have changed.

If waiting on a condition variable is not possible, at least have your thread sleep for a short amount of time between checks, rather than using all the CPU.