CAR PARK MANAGEMENT SYSTEM & SIMULATOR REPORT

Solo Team - Group 111

Johnny Madigan

October 2021

Contents

1 Demonstration Video

2 Statement of Contribution

Johnny Madigan	Invented, designed, documented, & demonstrated

3 Statement of Completeness

3.1 General requirements

Feature/Requirement	Status
Configurable no. of entrances/exits/levels 1-5	Implemented
Configurable capacity per level	Implemented
Configurable min/max temperatures	Implemented
Configurable chance of spawning authorised cars	Implemented
Configurable duration for running all 3 programs	Implemented
Configurable slow motion	Implemented

3.2 Simulator Requirements

Feature/Requirement	Status
Simulates movement of vehicles around the car park	Implemented
Simulates hardware (LPRs, boom gates, signs, alarms, temperature sensors)	Implemented
Checks bounds of *config.h* as these are user inputs prone to human-error	Implemented
Creates a dynamic shared memory object	Implemented
Shared memory's structure complies with specification	Implemented
Shared memory is setup for inter-process communication	Implemented
Avoids busy-waiting where possible	Implemented
Strict synchronisation using mutex locks/condition variables to avoid race-conditions/deadlocks	Implemented
Spawn cars every 1-100ms with random license plate	Implemented
Random license plate has a chance of being authorised	Implemented
Cars queue outside random entrances	Implemented
Cars trigger entrance LPR after 2ms	Implemented
Entrance threads read license plates into LPR and broadcasts Manager for decision-making	Implemented
Cars read the digital sign and act accordingly	Implemented
Car leaves simulation if sign displays not authorised (X)	Implemented

Car leaves simulation if sign displays car park is full (F)	Implemented
Car leaves simulation if sign displays "EVACUATE"	Implemented
Car enters if assigned a level no.	Implemented
Sets gates to opened if being raised (10ms)	Implemented
Sets gates to closed if being lowered (10ms)	Implemented
Authorised cars that enter are given their own thread so they can move/park independently	Implemented
Cars take 10ms to drive to their assigned level from the entrance	Implemented
Cars take 10ms to drive to a random exit after parking	Implemented
Cars trigger the level LPR when entering/exiting	Implemented
Cars queue at their random exit	Implemented
Exit threads read license plates into LPR and broadcasts Manager for billing	Implemented
Simulates temperature (1-5ms) per level using an algorithm to be as realistic as possible	Implemented
Ability to simulate a fire after configuring the minimum/-maximum temperatures	Implemented
Fires (rise or spike in temperature) trigger the Fire-Alarm System	Implemented
Cars park for a random amount of time (100-10000ms)	Implemented
Unusual behaviour	Absent

3.3   Manager requirements

| Feature/Requirement	Status
Automates aspects of running a car park	Implemented
Checks bounds of *config.h* as these are user inputs prone to human-error	Implemented
Reads in authorised license plates from *plates.txt* into hash-table	Implemented
Validates license plate format and length before adding to hash-table	Implemented
Locates and maps shared memory object to its data space	Implemented
Communicates with Simulator and Fire-Alarm System via inter-process communication	Implemented
Monitor status of all LPR sensors	Implemented
Keeps track of which car is assigned to which level	Implemented
Keeps track of each level's current capacity to direct new cars to avaliable spaces	Implemented
Keeps track of car park's total capacity (full or not)	Implemented
Keeps track of total customers for the simulation	Implemented
Ability to raise gate if closed	Implemented
Ability to lower gate if opened	Implemented
Keeps gates opened for 20ms before lowering	Implemented
Controls sign display	Implemented
Updates sign display to 'F' if car park full	Implemented

Updates sign display to 'X' if car is not authorised	Implemented
Updates sign display to a level no. assigned to authorised cars	Implemented
Keeps track of how long each car's journey is (time entered-time leaving)	Implemented
Bills cars at 5 cents per millisecond	Implemented
Appends billing information to *billing.txt*	Implemented
Monitors status each level's alarm and acts accordingly	Implemented
Displays status of all hardware (LPRs, boom gates, signs, alarms, and temperature sensors)	Implemented
Displays status of each level's current capacity and total capacity	Implemented
Displays total customers so far	Implemented
Displays total revenue earned so far($$$)	Implemented
Updates this display frequently (every 50ms)	Implemented

3.4 Fire-Alarm System requirements

Feature/Requirement	Status
Detects potential fires using 2 algorithms (rise/spike in temperature)	Implemented
Complies with safety-critical guidelines where possible (*MISRA C/The Power of 10*)	Implemented
Checks bounds of *config.h* as these are user inputs prone to human-error	Implemented
Locates and maps shared memory object to its data space	Implemented
Communicates with Simulator and Manager via inter-process communication	Implemented
Monitors and collects each level's current temperature every 2ms	Implemented
Keeps track of each level's 5 most recent temperatures	Implemented
When 5 temps are collected, the median is the next smoothed temp	Implemented
Keeps track of each level's 30 most recently smoothed temps	Implemented
Uses an algorithm for detecting if 90% or more smoothed temps are 58+ degrees	Implemented
Uses a 2nd algorithm for detecting if the most recent smoothed temp is 8+ degrees hotter than the 30th most recent smoothed temp	Implemented
Activates all alarms in the event of a fire	Implemented
Starts raising all boom gates in the event of a fire	Implemented
Cycles through each letter of "EVACUATE" and displays it on all signs (every 20ms)	Implemented

4   Usage

The project structure includes 3 source folders, one for each program. In the main directory, *plates.txt* contains the list of authorised license plates (which you can update), and a configuration file called *config.h*. *plates.txt* MUST stay in the same directory as the executables, as the Simulator and Manager read this file to get the authorised license plates.
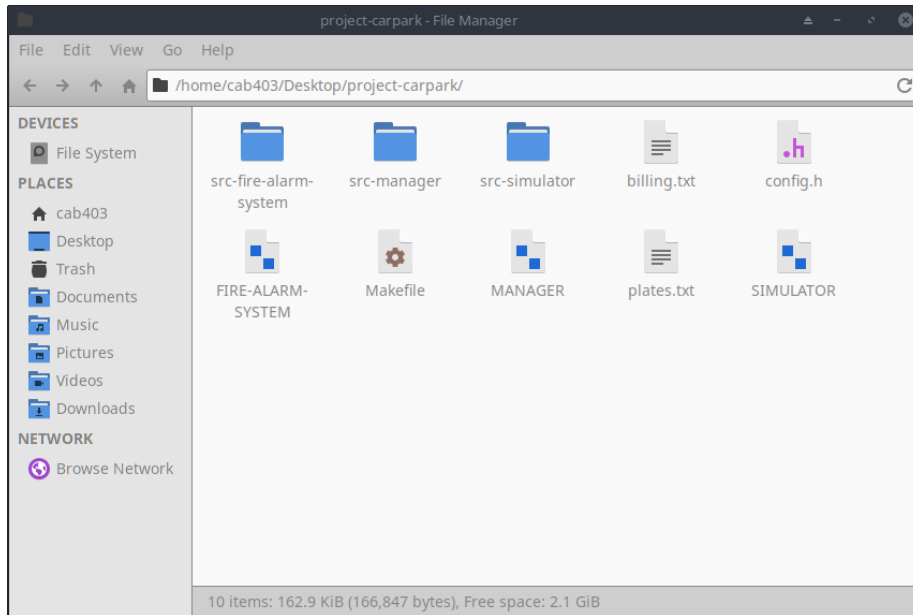
config.h allows you to customise the simulation in the following ways:

- Number of entrances, exits, and levels

- Parking capacity for each level

- Chance of spawning an authorised car

- Duration all 3 programs will run for

- Minimum and maximum temperatures

- Slow motion

Each source folder has its own Makefile, so you can clean and rebuild each program individually. There is a central Makefile in the main directory, which can clean and rebuild all 3 Makefiles at once. After building, 3 executable will be created in the main directory. When running the programs, make sure to run the Simulator first, as it's responsible for creating the shared memory object that the Simulator, Manager, and Fire-Alarm System will use to communicate between each-other. Please open 3 terminal windows and run each program in its own window. Please navigate into the main directory to run the following:

To clean:

```
$ make clean
```

To build:

```
$ make
```

To run the Simulator:

```
$ ./SIMULATOR
```

To run the Manager:

```
$ ./MANAGER
```

To run the Fire-Alarm System:

```
$ ./FIRE-ALARM-SYSTEM
```

After the programs finish, there will be a new file called *billing.txt* showing how much each customer was billed during the simulation.

5 Assessment of the supplied Fire Alarm system

5.1 NASA's *The Power of 10* Violations

RULE 1 VIOLATION
"Avoid complex flow constructs, such as goto and recursion"

The system does not avoid complex flow constructs such as *goto* and recursion. A *goto* statement is found in Main and the *deletenodes* function is recursive.

RULE 2 VIOLATION
"All loops must have fixed bounds. This prevents runaway code"

The system contains loops with no fixed bounds, a.k.a. runaway code. These infinite loops are found in Main (causing unreachable code), the *tempmonitor* function (thread will never return), and the *openboomgate* function (thread will never return).
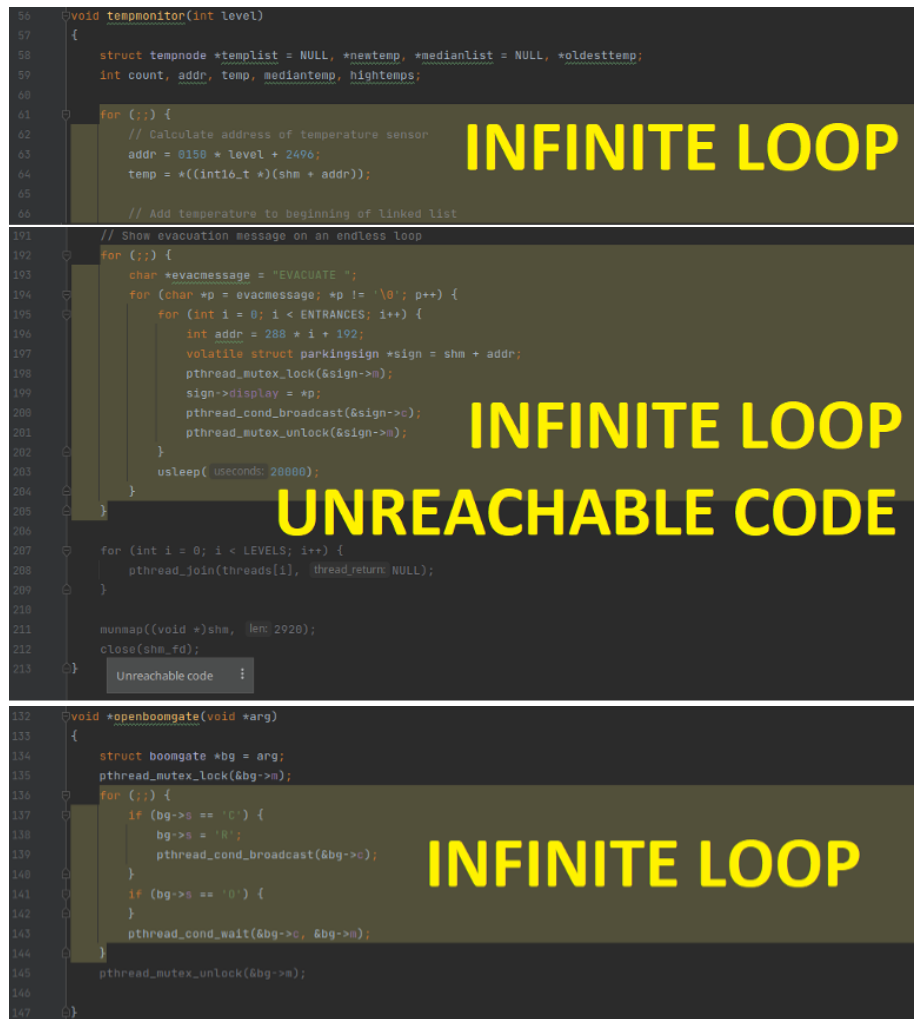
RULE 3 VIOLATION
"Avoid heap memory allocation"

The system fails to avoid heap memory allocation (5 times) using the *malloc* function.

RULE 4 VIOLATION
"Restrict functions to a single printed page"

The system fails to restrict functions to a single printed page (avoid extremely long/multi-purposed/complex functions). The *tempmonitor* function spans 73 lines.

RULE 5 VIOLATION
"Use a minimum of two runtime assertions per function"

The system fails to use at least 2 runtime assertions. For example, the system should have checked if *shm_open* and *mmap* worked (not NULL).

RULE 6 VIOLATION
"Restrict the scope of data to the smallest possible"

The system fails to restrict scope of data to the smallest possible as there are global variables that do not need to be global. Integer variable *shm_fd* is only used in *Main* but is declared globally, and volatile void pointer *shm* is also declared globally and defined in main but used only once in another function (*tempmonitor*) and therefore should be passed as an argument to minimise scope.

RULE 10 VIOLATION
"Compile with all possible warnings active; all warnings should then be addressed before release of the software"

The system has left a large number of warnings unaddressed when compiled with flags *-Wall -Wextra -pedantic -g*.

5.2 *MISRA C:2012* Violations

Directive 1.1

Although the system contains comments indicating some sections of code, the system fails to document the behaviour of the file itself and functions within (brief, parameters, return values, etc.).

Directive 1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood	Required	Undecidable	No	This directive is not statically verifiable.

Directive 2.1

Likewise in *The Power of 10* evaluation, the supplied system left a large number of warnings unaddressed when compiling with flags *-Wall -Wextra -pedantic -g*.

Directive 2.1	All source files shall compile without any compilation errors	Required	Undecidable	No	No checker, but a successful analysis run confirms compliance.

Directive 4.12

Likewise in *The Power of 10* evaluation, the system uses the *malloc* function to allocate dynamic memory to the heap 5 times. Furthermore, there was no code following these *malloc* calls to handle failure (when *malloc* returns NULL) nor was the *free* function used to deallocate the heap memory.

Directive 4.12	Dynamic memory allocation shall not be used	Required	Undecidable	Yes	

Rule 2.1

An infinite loop near the end of *Main* prevents the system from cleaning up (*munmap* and *shm_close*) and returning an exit value (although no return value currently implemented).

Rule 2.1	A project shall not contain *unreachable code*	Required	Undecidable	Yes	

Rule 15.1

The system's *Main* function uses a *goto* statement to break out of a loop.

Rule 15.1	The *goto* statement should not be used	Advisory	Decidable	Yes	

Rule 15.5

The system's *deletenodes* function has 2 points of exit.

Rule 15.5	A function should have a single point of exit at the end	Advisory	Decidable	Yes	

Rule 17.2

The system's *deletenodes* function is recursive.

Rule 17.2	Functions shall not call themselves, either directly or indirectly	Required	Undecidable	Yes	

Rule 17.8

The system's *deletenodes* function modifies a linked-list of temperatures (as it deletes members in the given list).

Rule 17.8	A function parameter should not be modified	Advisory	Undecidable	Yes	

Rule 18.4

The system accesses segments of the shared memory object using pointer arithmetic ($shm + addr$).

Rule 18.4	The +, -, += and -= operators should not be applied to an expression of pointer type	Advisory	Decidable	Yes	

Rule 21.3

Likewise in *The Power of 10* evaluation, the system uses the *malloc* function to allocate dynamic memory to the heap 5 times. Furthermore, there was no code following these *malloc* calls to handle failure (when *malloc* returns *NULL*) nor was the free function used to de-allocate the heap memory.

Rule 21.3	The memory allocation and deallocation functions of `<stdlib.h>` shall not be used	Required	Decidable	Yes	This rule was updated in Amendment 2.

Rule 21.6

The *fprintf* function is used in *Main* to print *"*** ALARM ACTIVE ***"* in standard error message format.

Rule 21.6	The Standard Library input/output functions shall not be used	Required	Decidable	Yes	

Rule 21.9

The *qsort* function is used to sort temperatures in ascending order to get the median (in the *tempmonitor* function).

Rule 21.9	The library functions *bsearch* and *qsort* of `<stdlib.h>` shall not be used	Required	Decidable	Yes	Adheres to Technical Corrigendum 1.

Rule 21.10

Although unused, the system includes the standard *time.h* library.

Rule 21.10	The Standard Library time and date functions shall not be used	Required	Decidable	Yes	This rule was updated in Amendment 2.

5.3 Additional Problems

The system's *Main* function does not have a return value while promising a return value of type integer.

The system does not meet the requirement of waiting 2 milliseconds before collecting temperatures.

The system has partially hard-coded the address for accessing exit's boom gates. *Main* assumes all exit's boom gates will always be at an offset of 1536 bytes, which is 5 entrances. As there could be anywhere between 1 to 5 entrances inclusive, the system will break immediately.

The system declares 2 unused global variables, a *pthread* mutex lock and *pthread* condition variable for the global alarm indicator.

5.4 Solution method

The approach I used to address the issues highlighted above, was to rewrite the whole program from scratch. This not only allowed me evaluate every decision I was making from a safety-critical POV, but also improve the logic and flow of data.

I wanted this new implementation to be uniform with the *Simulator* and *Manager*, so it would be easy to read and understand how all 3 programs work together. This meant updating how the system would find and map the shared memory object to its data-space. How the system would locate segments of the shared memory (3 formulas for locating entrances/exits/levels then accessing their hardware via arrow pointer notation). How thread cycles flow (no infinite loops, all threads return when the *end_simulation* flag is set to true). How thread safety is ensured with mutex locks, condition variables, and the *volatile* and *_Atomic* keywords. How the program is split into multiple source and header files for modularity/simplicity. And finally, how all bounds are checked/handled, especially *config.h* bounds as these are prone to human error.

The 2 key parts of the system that needed to be compliant with safety-critical standards was fixing the recursive linked-list function and replacing the *qsort* function. Rather than deleting linked-list (temperature) nodes recursively, the new implementation simply traverses the entire linked-list and after a certain count, the rest of the nodes will be deleted. The *qsort* function has been replaced with a bubble-sort function that sorts an array in ascending order before returning the median.

5.5 Current safety-critical concerns & mitigations

When rewriting the fire-alarm system, 3 *MISRA C* guidelines, unfortunately, could not be followed completely. First was no dynamic memory allocation (directive 4.12), second was the use of the standard *time.h* library (rule 21.10), and thirdly was the use of pointer arithmetic (rule 18.4). These violations were mitigated where possible.

Regarding dynamic memory, the need to constantly update to hold only the 5 most recent raw temperatures/30 most recently smoothed temperatures, is extremely tedious without the use of dynamic memory. Using a linked-list and its associated operations, greatly reduces code-complexity and overhead. However, as nodes are struct pointers, they must be malloced to have sufficient memory to hold the temperature and a pointer to the next node. This violation has been minimised by limiting the use of the *malloc* function and ensuring that all dynamic memory is released using *free*.

The *Simulator* and *Manager* use a custom function for precision when sleeping for milliseconds. However, this requires the *time.h* library for the *timespec* type and *nanosleep* function. Due to sensitive timing being a crucial part of the project, the custom function is still needed here, but isolated in one source/-header file.

Given the address of the first byte of the shared memory object, pointer arithmetic is necessary to locate items within.