CSE 344 System Programming – Homework 3 Report

Ahmet Özdemir

1- Introduction

First of all, I will start by explaining the general flow I followed in the assignment. The program is written in 3 separate files as *system.c*, *systemHelper.c* and *systemHelper.h*.

Main function is located in *system.c* file, here firstly an object is created from an enum structure named *VehicleType* which is created to determine the vehicle types. This will be used to create attendant threads. Then an owner and two attendant threads (valet for pickup, valet for automobile) are defined. After this point, the semaphores given to us in the assignment document are initialised by calling the ***initialize()*** function.

```
void initialize()
{
    sem_init(&newPickup, 0, 0);
    sem_init(&inChargeForPickup, 0, 0);
    sem_init(&newAutomobile, 0, 0);
    sem_init(&inChargeForAutomobile, 0, 0);

    pthread_mutex_init(&lock, NULL);
}
```

After initialisation, the defined threads are started with the *pthread_create()* function and the main part of the program starts.

```
#include "systemHelper.h"

int main(int argc, char const *argv[])
{
    VehicleType types[2] = {AUTOMOBILE, PICKUP};

    pthread_t owner;
    pthread_t attendant[2];

    initialize(); /* Kaynakları başlat */

    pthread_create(&owner, NULL, carOwner, NULL); /* aynı anda sadece bir araç girişini sağlamak için sadece bir tane
    pthread_create(&attendant[0], NULL, carAttendant, &types[0]);
    pthread_create(&attendant[1], NULL, carAttendant, &types[1]);

```

In the code, a vehicle arrives in the temporary parking area and if there is space in the spots of its type, it is taken by the valet to be parked there, otherwise a message is printed on the screen as "There is no temporary space for ...". Since the end of the programme is not specified in the assignment document, I have created a simulation with a total of 30 vehicles. In other words, there will be a total of 30 parking requests that may or may not be parked. In addition, in order to make the parking simulation a little more realistic, each pickup type vehicle waits for 3 seconds in the car park, while automobile type vehicles wait for 5 seconds. After these waiting times, the vehicles are removed from the car park and new vehicles are taken by the valet to be parked in the empty spaces in line with other new requests. In addition, the process of a valet taking the vehicle to the parking place is simulated to be 0.5 seconds. These values will be understandable when looking at the codes. I mentioned these details in the comment lines.

2- 'running' variable

Controlling Program Flow: The running variable controls the running of the main loops of the program. This allows the program to stop or continue under a certain condition.

Thread Synchronization: The running variable ensures that created threads are stopped synchronously. This ensures that the program closes properly.

1- ***Initial Value:*** When the program starts, the running variable is set to TRUE (1), indicating that the program should run.
2- ***Use in Thread Loops:*** The running variable is checked in loops in the carOwner and carAttendant functions. These loops will continue to run as long as the running variable is TRUE.
3- ***Program Stop:*** The variable vehicleCount keeps track of the number of simulated vehicles. When this number reaches TOTAL_VEHICLES, the running variable is set to FALSE (0), indicating that the program should end.
4- ***Stopping Threads:*** In the main function, after the running variable is set to FALSE, waiting threads (carAttendant functions) are woken up via semaphores and these threads are properly terminated.

In summary, the running variable is used to control the running state of the program and threads and to terminate the program properly when a certain condition is met.

```
112
113  void * carAttendant(void * arg)
114  {
115      VehicleType type = *(VehicleType *)arg;
116
117      while (running)
118      {
```

```
39
40  void * carOwner(void * arg)
41  {
42      while (running)
43      {
44          int carType = randomGenerator(); /* 0: Otomobil, 1: Pickup */
45
```

Program termination:

```
vehicleCount++; /* Araç sayacını artır */

if (vehicleCount >= TOTAL_VEHICLES) /* TOTAL_VEHICLES kadar araç simüle ediyoruz */
{
    printf("* Terminating the programme...\n");
    running = FALSE;
    pthread_mutex_unlock(&lock);
    exit(EXIT_SUCCESS);
}
```

3- CarOwner thread

The *carOwner* thread is responsible for simulating vehicles arriving at the car park and placing them in the appropriate parking spaces. This thread repeats the process of arriving and parking vehicles until a certain number is reached. Its functionality can be explained in the following steps:

1. Determining the Vehicle Type:

The *carOwner* thread determines a random vehicle type (car or pickup) each cycle using the *randomGenerator* function. This function generates a random number that returns 0 or 1. 0 means car and 1 means pickup.

```
40  void * carOwner(void * arg)
41  {
42      while (running)
43      {
44          int carType = randomGenerator(); /* 0: Otomobil, 1: Pickup */
45
46          if (carType == 0)
47          { ...
66          }
67
68          else if (carType == 1)
69          { ...
88          }
89
```

2. Parking Space Control and Vehicle Parking:

- If the randomly selected vehicle type is a car and there are free car parking spaces available, a parking space is freed *mFree_automobile*-- and the car is parked. After this, the function *sem_post(&inChargeForAutomobile)* is called to signal a *carAttendant* thread.

- If the randomly selected vehicle type is pickup and there are free pickup parking spaces available, a parking space is freed *mFree_pickup*-- and the vehicle is parked. After this, the function *sem_post(&inChargeForPickup)* is called to send a signal to a *carAttendant* thread.

- If there is no parking space for the selected vehicle type, the relevant information message is printed on the screen.

```
if (carType == 0)
{
    /*-------- critical region start --------*/

    pthread_mutex_lock(&lock);

    if (mFree_automobile > 0)
    {
        mFree_automobile--;
        printf("+ Automobile parked. Available empty automobile spaces: %d pickup spaces: %d\n", mFree_automobile, mFree_pickup);
        sem_post(&inChargeForAutomobile);
    }

    else
    {
        printf("x There is no empty space for automobile.\n");
    }
    pthread_mutex_unlock(&lock);

    /*-------- critical region end --------*/
}

else if (carType == 1)
{
    /*-------- critical region start --------*/

    pthread_mutex_lock(&lock);

    if (mFree_pickup > 0)
    {
        mFree_pickup--;
        printf("+ Pickup parked. Available empty automobile spaces: %d pickup spaces: %d\n", mFree_automobile, mFree_pickup);
        sem_post(&inChargeForPickup);
    }

    else
    {
        printf("x There is no empty space for pickup.\n");
    }
    pthread_mutex_unlock(&lock);

    /*-------- critical region end --------*/
}
```

3. Updating the Vehicle Counter:

After the vehicle parking operation, the total number of parked vehicles *vehicleCount* is incremented by one. This counter is used to ensure that the programme terminates when it reaches a certain number.

```
vehicleCount++;
```

4. In and Out of Critical Zone:

- Vehicle parking is considered as a critical region because multiple threads accessing the parking spaces at the same time may cause data inconsistency. Therefore, mutex lock is taken with *pthread_mutex_lock(&lock)* and mutex lock is released with *pthread_mutex_unlock(&lock)*.

5. Vehicle Arrival Simulation:

The arrival of vehicles is simulated by applying a half-second waiting time usleep(500000) after each vehicle parking operation.

These steps describe the operation of the `carOwner` thread and emphasise why each step is important. This thread is designed in accordance with the principles of concurrent programming and resource management.

4 – carAttendant thread

The *carAttendant* thread is responsible for simulating the exit of vehicles from the car park and making the parking spaces available again. This thread works separately for different types of vehicles and manages the spaces in the car park. Its functionality can be explained in the following steps:

1. Determining the Vehicle Type:

The *carAttendant* thread works with the vehicle type (car or pickup) given to it when it was created. This type allows the thread to perform operations only for a specific vehicle type.

```
113  void * carAttendant(void * arg)
114  {
115      VehicleType type = *(VehicleType *)arg;
116
117      while (running)
118      {
119          if (type == PICKUP)
120          { ...
136          }
137
138          else if (type == AUTOMOBILE)
139          { ...
155          }
156      }
157      return NULL;
158  }
159
```

2. Waiting and Vehicle Exit:

- If the thread is created for a pickup vehicle, it waits for a pickup vehicle to exit with the function *sem_wait(&inChargeForPickup)*. When the pickup vehicle exits, it waits for 3 seconds and then empties the parking space.

- If the thread is created for a car vehicle, it waits for a car vehicle to exit with the function *sem_wait(&inChargeForAutomobile*. When the car exits, it waits for 3 seconds and then empties the parking space.

3. Parking Space Check and Update:

- When the pickup vehicle exits, the available free pickup parking spaces are increased by one *mFree_pickup*++ by entering the critical zone *pthread_mutex_lock(&lock)*. After this operation, the relevant information message is printed on the screen and the mutex lock is released *pthread_mutex_unlock(&lock)*.

- When the car vehicle exits, the available free car parking spaces are increased by one *mFree_automobile*++ by entering the critical zone *pthread_mutex_lock(&lock)*. After this operation, the related information message is printed on the screen and the mutex lock is released *pthread_mutex_unlock(&lock)*.

```
else if (type == AUTOMOBILE)
{
    sem_wait(&inChargeForAutomobile);
    sleep(3); /* Otomobil otoparkta 3 saniye bekler */

    /*-------- critical region start --------*/

    pthread_mutex_lock(&lock);

    if (mFree_automobile < NUM_AUTO)
    {
        mFree_automobile++;
        printf("- The automobile has been removed. Number of available free automobile parking spaces: %d\n", mFree_automobile);
    }
    pthread_mutex_unlock(&lock);

    /*-------- critical region end --------*/
}
```

```c
if (type == PICKUP)
{
    sem_wait(&inChargeForPickup);
    sleep(3); /* Pickup otoparkta 3 saniye bekler */

    /*-------- critical region start --------*/

    pthread_mutex_lock(&lock);

    if (mFree_pickup < NUM_PICK)
    {
        mFree_pickup++;
        printf("- The pickup has been removed. Number of available free pickup parking spaces: %d\n", mFree_pickup);
    }
    pthread_mutex_unlock(&lock);

    /*-------- critical region end --------*/
}
```

4. In and Out of Critical Zone:

- The vehicle exit process is considered as a critical zone, because multiple threads accessing the parking spaces at the same time may cause data inconsistency. For this reason, entry and exit is done using mutex lock.

The *carAttendant* thread ensures that vehicles in the car park exit in an orderly manner and that the parking spaces are made available again. This thread is designed in accordance with the principles of concurrent programming and resource management.

5- Header File

Here are the macros, global variables, function declarations, libraries and enum structure I use.

```c
#ifndef SYSTEM_HELPER_H
#define SYSTEM_HELPER_H

#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define TOTAL_VEHICLES 30
#define NUM_PICK 4
#define NUM_AUTO 8
#define FALSE 0
#define TRUE 1

extern sem_t newPickup;
extern sem_t inChargeForPickup;
extern sem_t newAutomobile;
extern sem_t inChargeForAutomobile;

extern int mFree_automobile;
extern int mFree_pickup;
extern int running;
extern int vehicleCount;

extern pthread_mutex_t lock;

typedef enum
{
    AUTOMOBILE,
    PICKUP
}
VehicleType;

int randomGenerator(void);
void initialize();
void clean();
void * carOwner(void *);
void * carAttendant(void *);

#endif /* SYSTEM_HELPER_H */

```

6 – Makefile and Usage

Structure and Control Mechanisms of Makefile

1. Variables:

- `CC`: `gcc` is used as compiler.

- `CFLAGS`: Compilation flags. These flags include warning messages (`-Wall`), strict standards (`-pedantic-errors`), the GNU99 standard (`-std=gnu99`), POSIX and GNU sources (`-D_POSIX_C_SOURCE=200809L -D_GNU_SOURCE`) and large file support (`-D_FILE_OFFSET_BITS=64`). Also, the pthread library has been added (`-pthread`).

- `LIBS`: Additional libraries. The maths library (`-lm`) is used in this project.

- `DEPS`: Dependency files. The `systemHelper.h` header file is specified.

- `OBJ`: Object files. `systemHelper.o` and `system.o` are specified.

2. General Rules:

- `%.o: %.c $(DEPS)`: This rule compiles all `.c` files into `.o` files. When header files specified as dependencies change, the corresponding `.o` files are rebuilt.

3. Targets:

- `all`: calls the `system` target and compiles the whole programme.

- `system`: Creates the `system` executable by linking object files (`$(OBJ)`).

- `valgrind`: Calls the `system` target and then checks for memory leaks with Valgrind. It provides a detailed memory leak report with `--leak-check=full --show-leak-kinds=all --track-origins=yes` flags.

- `helgrind`: Invokes the `system` target and then checks thread safety with Helgrind.

- `clean`: Deletes all object files created during compilation and the `system` executable.

4. Phony Targets:

- `.PHONY: all clean valgrind helgrind`: These targets are specified to avoid conflicts with file names. This way, these targets will work correctly even if the target names have the same file names.

Instructions for usage

- Compiling the Programme:

    make all

or just

    Make

- Running the programme:

    ./system

- Checking for Memory Leaks:

    make valgrind

- Checking Thread Security:

    make helgrind

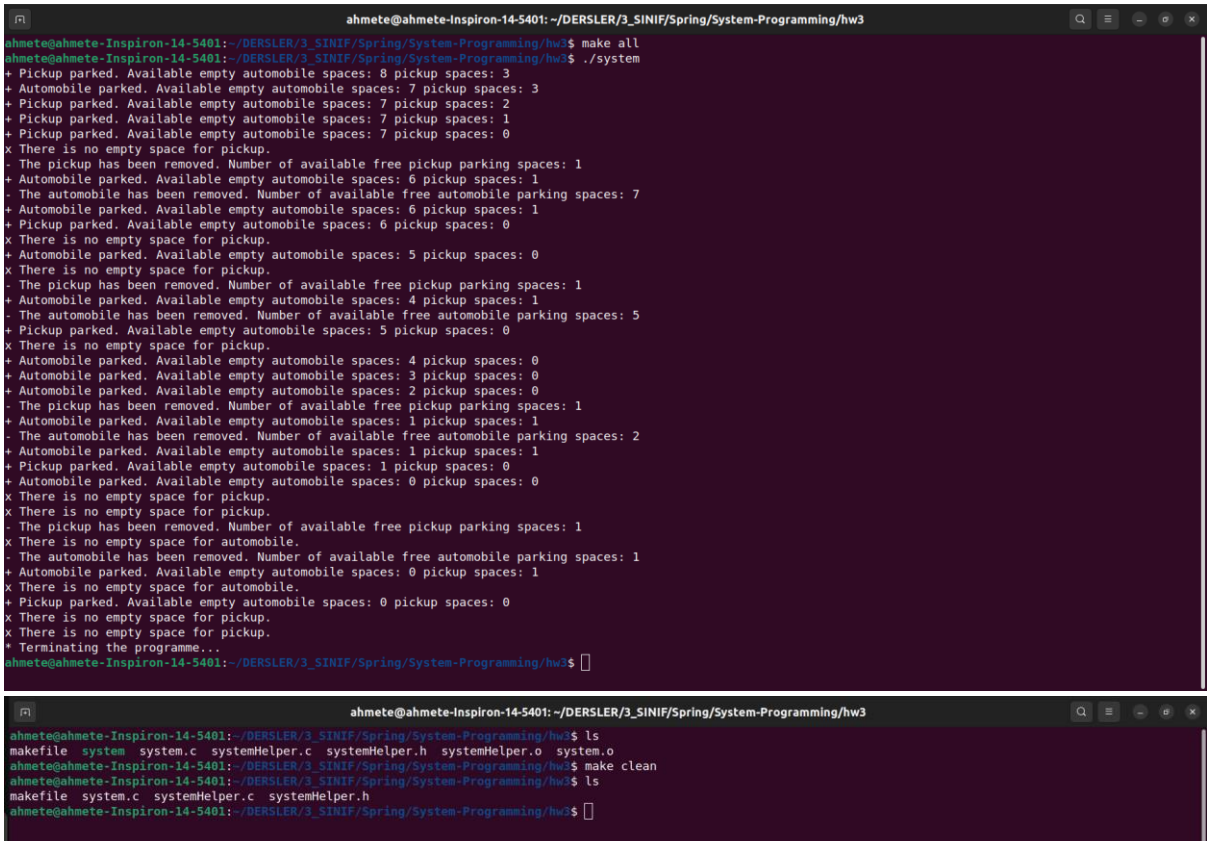
- Clearing Temporary Files:

    make clean

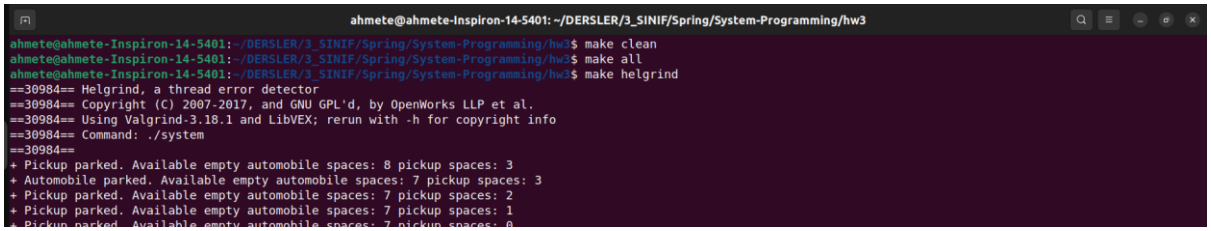
These instructions and explanations will help you understand how to use Makefile and the control mechanisms it contains.

```
.SILENT:

CC = gcc
CFLAGS = -Wall -pedantic-errors -std=gnu99 -pthread
CFLAGS += -D_POSIX_C_SOURCE=200809L -D_GNU_SOURCE
CFLAGS += -D_FILE_OFFSET_BITS=64
LIBS = -lm
DEPS = systemHelper.h
OBJ = systemHelper.o system.o

# general rules
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

# TARGETS:

all: system

# main target
system: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

# mem-check targets
valgrind: system
    valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./system

# thread safe control
helgrind: system
    valgrind --tool=helgrind ./system

# clean rule
clean:
    rm -f *.o system

.PHONY: all clean valgrind helgrind

```

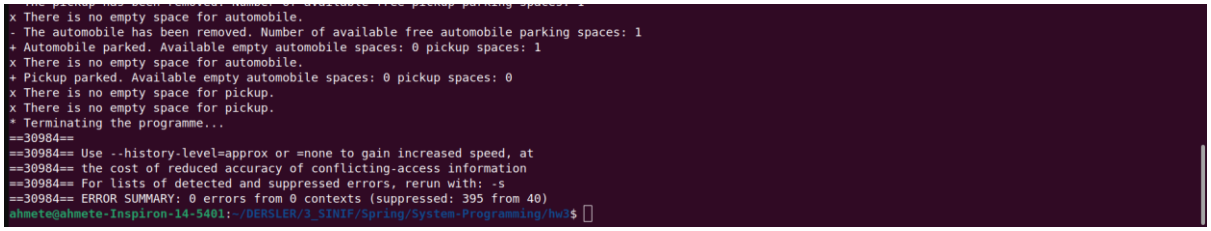
7- Outputs



For Helgrind (thread-safe control for race conditions)


ahmete@ahmete-Inspiron-14-5401: ~/DERSLER/3_SINIF/Spring/System-Programming/hw3
ahmete@ahmete-Inspiron-14-5401:~/DERSLER/3_SINIF/Spring/System-Programming/hw3$ make clean
ahmete@ahmete-Inspiron-14-5401:~/DERSLER/3_SINIF/Spring/System-Programming/hw3$ make all
ahmete@ahmete-Inspiron-14-5401:~/DERSLER/3_SINIF/Spring/System-Programming/hw3$ make helgrind
==30984== Helgrind, a thread error detector
==30984== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==30984== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==30984== Command: ./system
==30984==
+ Pickup parked. Available empty automobile spaces: 8 pickup spaces: 3
+ Automobile parked. Available empty automobile spaces: 7 pickup spaces: 3
+ Pickup parked. Available empty automobile spaces: 7 pickup spaces: 2
+ Pickup parked. Available empty automobile spaces: 7 pickup spaces: 1

...


x There is no empty space for automobile.
- The automobile has been removed. Number of available free automobile parking spaces: 1
+ Automobile parked. Available empty automobile spaces: 0 pickup spaces: 1
x There is no empty space for automobile.
+ Pickup parked. Available empty automobile spaces: 0 pickup spaces: 0
x There is no empty space for pickup.
x There is no empty space for pickup.
* Terminating the programme...
==30984==
==30984== Use --history-level=approx or =none to gain increased speed, at
==30984== the cost of reduced accuracy of conflicting-access information
==30984== For lists of detected and suppressed errors, rerun with: -s
==30984== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 395 from 40)
ahmete@ahmete-Inspiron-14-5401:~/DERSLER/3_SINIF/Spring/System-Programming/hw3$