Rapport de projet “Formule 1” Groupe 7

Rapport de seconde session

Melvin Campos Casares, Maxime De Cock,
Dominik Fiedorczuk, Hubert Van De Walle

FIA FORMULA ONE
WORLD CHAMPIONSHIP

26 aout 2019

Table des matières

Rapport du projet : F1-of-Linux

Groupe 7

Notre groupe est constitué de 4 personnes :

- Melvin Campos Casares
- Maxime De Cock
- Dominik Fiedorczuk
- Hubert Van De Walle

Introduction et présentation du projet

Ce projet consiste à présenter un week-end complet d’un grand prix de Formule 1, depuis les séances d’essais du vendredi jusqu’à la course du dimanche, en passant par les essais du samedi et la séance de qualifications.

Notre but consiste à reproduire cela dans un langage de programmation performant à l’exécution des méthodes implémentées, le langage C. Nous devons générer un affichage qui gèrera les séances d’essais libres, les qualifications ainsi que la course. De plus, certaines informations doivent être disponible : temps au tour, temps secteur, disqualification, arrêt aux stands, temps depuis le début de la course.

De plus, nous devons appliquer des concepts vus en cours en première année ainsi qu’en deuxième : processus père-fils (dont `fork` est la création d’un nouveau processus utilisateur), sémaphores (pour gérer la synchronisation des processus) et la mémoire partagée (allocation et utilisation par appel des mémoires partagées via leurs identificateurs).

Cahier des charges

Nous voulons organiser un grand prix contenant 20 voitures, à la manière de la Formule 1. Leurs numéros sont : 44, 77, 5, 7, 3, 33, 11, 31, 18, 35, 27, 55, 10, 28, 8, 20, 2, 14, 9, 16.

Un circuit de F1 est divisé en 3 secteurs (S1, S2, S3).

Le calendrier d’un week-end de Formule 1 est établi de la manière suivante :

- Vendredi matin, une séance d’essais libres d’une durée de 1h30 (P1)
- Vendredi après-midi, une deuxième séance d’essais libres d’une durée de 1h30 (P2)
- Samedi matin, une troisième et dernière séance d’essais libres d’une durée de 1h (P3)
- Samedi après-midi, une séance de qualifications est organisée et divisée en 3 parties :
  - Q1, d’une durée 18 minutes et visant à éliminer les 5 dernières voitures qui occuperont les places 16 à 20 sur la grille de départ de la course en fonction de leur meilleur temps au tour,
  - Q2, d’une durée 15 minutes, qui éliminera les 5 dernières voitures suivantes et qui occuperont les places 11 à 16 sur la grille de départ de la course en fonction de leur meilleur temps au tour,
  - Q3, d’une durée 12 minutes et permettant de classer les 10 voitures restantes pour établir les 10 premières places sur la grille de départ de la course en fonction de leur meilleur temps au tour.
- Dimanche après-midi, la course en elle-même visant à obtenir un podium ainsi qu’un classement général typique des courses de voitures.

Ce projet devra prendre en charge plusieurs particularités qui seront développées dans les points ci-dessous.

Première partie : gestion des séances d’essai, qualifications et course

Lors des séances d’essais (P1, P2, P3) : Il est nécessaire de relever les temps dans les différents secteurs (au nombre de 3) à chaque passage de chacune des voitures.

De plus, dans l’affichage des séances d’essais, il est important de connaître le meilleur temps à chaque secteur ainsi que d’autres informations pertinentes comme si la voiture est au stand (PIT) ou si elle abandonne la séance (OUT). Malgré que les voitures soient au stand où ait abandonné la séance, on conserve toujours le meilleur temps de la voiture ainsi que son classement.

Pendant la séance d’essais, le classement des voitures se fait en fonction de leur tour complet le plus rapide. À la fin des séances d’essais, on conserve le classement final.

Lors des qualifications (Q1, Q2, Q3) : Lors des qualifications,le temps des 3 secteurs à chaque passage pour chaque voiture est à relever.

De plus, dans l’affichage des qualifications, il est important de connaître le meilleur temps à chaque secteur ainsi que d’autres informations pertinentes comme si la voiture est au stand (PIT) ou si elle abandonne la séance (OUT). Malgré que les voitures soient au stand où ait abandonné la séance, on conserve toujours le meilleur temps de la voiture ainsi que son classement.

Le classement des voitures se fait en fonction de leur tour complet le plus rapide.

La particularité avec les qualifications est un temps réduit et l’importance de conserver le classement de chacune des séances afin d’en définir l’ordre de départ de la course.

- À la fin de la première qualification, 15 voitures resteront qualifiées pour la 2ème séance et les 5 dernières sont placées à la fin de la grille de départ (places 16 à 20),
- À la fin de la deuxième qualification, il reste 10 voitures qualifiées pour la 3ème séance et les 5 dernières sont placées dans les places 11 à 15 de la grille de départ,
- Le classement de la troisième qualification attribue les places 1 à 10 de la grille de départ.

Lors de la course : L’affichage de la course présente le classement de l’ordre sur la grille de départ. Le classement doit toujours être maintenu durant la course, même s’il y a des dépassements. Il est important de savoir qui a le meilleur temps dans chacun des secteurs et également qui a le tour le plus rapide.

Comme pour les essais libres et les qualifications, il est nécessaire de relever les temps dans les différents secteurs à chaque passage de chacune des voitures.

- Si une voiture est en abandon de course (out), elle sera classée en fin de classement.
- Si la voiture est aux stands (PIT), le temps au stand est comptabilisé dans son temps et elle ressort à sa place dans la course. Par ailleurs, pour ce point, il y a généralement 2 ou 3 PIT par voiture par course.

À la fin de la course, on conserve le classement final et le tour le plus rapide.

***Remarque :* les stands se trouvent toujours dans le secteur 3.**

De plus, il est demandé de paramétrer le programme. En effet, les circuits peuvent être de longueur variable et le nombre de tours pour la course varie également (on essaie que le nombre total de kilomètres soit toujours plus ou moins le même pour chacune des courses du calendrier).

On vous demande de :

- Réaliser le programme en C sous Linux ;
- Utiliser la mémoire partagée comme moyen de communication interprocessus ;
- Utiliser les sémaphores pour synchroniser l’accès à la mémoire partagée.

Analyse du travail

Afin de reprendre au mieux ce projet, nous avons retenu les analyses faites avec la professeur suite au premier rendez-vous du 2 avril. Suite à ce meeting, nous avons décidé de commencer par décortiquer les demandes et en faire un tableau ainsi qu’un flowchart afin de mieux visualiser le projet :

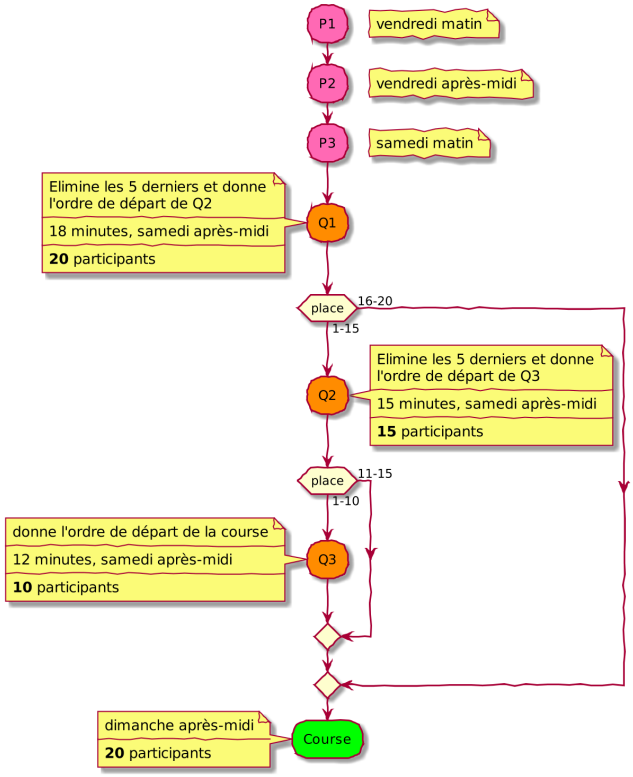

Fig. 1 : Flowchart

Explication des particularités du code

Fonctionnalités du code

Le programme prend en tant qu’arguments le nom d’une étape du week-end de Formule 1 ainsi que la longueur d’un tour en kilomètres. Si ce dernier n’est pas fourni, une valeur par défaut est attribuée.

On lance la phase sélectionnée pour chacune des voitures participantes. Lors de la simulation, les voitures participantes vont générer des temps aléatoires à chaque secteur.

Un tableau de valeurs reprenant des informations diverses est ensuite affiché afin de pouvoir suivre l’évolution de l’étape choisie. Les informations représentée dans ce dernier dépendent de l’étape concernée. Ce tableau est également trié en fonction du meilleur temps de tour par pilote ou, dans le cadre de la course, trié en fonction de leur position.

Au départ de la course, chaque participant démarre dans l’ordre précédemment déterminé par les séances de qualifications et avec une pénalité relative à leur position de départ.

Lorsque la simulation d’une étape est terminée, les positions des pilotes est sauvegardée dans un fichier. Ce fichier sera chargé lors de l’étape suivante afin de déterminer les participants ainsi que leurs positions.

Mémoire partagée et communication entre processus

On crée une zone de mémoire partagée puis on y attache un tableau de structure.

```
SharedStruct *data;

int struct_shm_id =
    shmget(IPC_PRIVATE, sizeof(SharedStruct) *
    options.participant_count, 0600 | IPC_CREAT);

data = shmat(struct_shm_id, NULL, 0);
```

La mémoire partagée contient un tableau de structure comportant les informations de secteurs entre autres choses.

```
typedef struct SharedStruct {
    int id;
    int s1;
    int s2;
    int s3;
    int best_s1;
    int best_s2;
    int best_s3;
    int best_lap_time;
    int lap;
    int sector;
```

```
12      int out;
13      int pit;
14      int done;
15  } SharedStruct;
```

Dans notre cas, la mémoire partagée n’est accédée ou modifiée qu’avec un seul « écrivain » et un seul « lecteur » à la fois ; il n’y aura jamais plus d’une écriture et lecture en même temps. Ici, chaque processus fils est un écrivain alors que le lecteur est le processus père.

La sémaphore nous permettent de garantir l’accès exclusif à la mémoire partagée. Les opérations `sem_wait(sem_t *sem)` et `sem_post(sem_t *sem)` permettent respectivement de verrouiller et déverrouiller une sémaphore.

Libération des ressources de l’ordinateur

Afin de libérer les ressources de l’ordinateur, plusieurs étapes sont réalisées une fois que les processus enfants ont terminé leur fonction et que le programme est prêt à quitter.

Premièrement, il y a « destruction » de la sémaphore par le biais de l’opération `sem_destroy(sem_t *sem)`.

Ensuite, on se détache des zones de mémoire partage et ensuite on les supprime.

```
1  shmdt(data);
2  shmctl(struct_shm_id, IPC_RMID, NULL);
3
4  sem_destroy(sem);
5  shmdt(sem);
6  shmctl(sem_shm_id, IPC_RMID, NULL);
```

Création et gestion des processus

Chaque voiture correspond à un processus fils, tandis que le père s’occupe de la gestion des étapes et de l’affichage.

La création des processus se fait par le biais de la fonction `fork`, faisant partie des appels système POSIX. Elle permet de donner naissance à un nouveau processus qui est sa copie.

Nos `fork` sont présent dans le fichier de code source `main.c`.

Rôle du processus père

Dans notre cas, nous avons un processus père donnant naissance au nombre de processus fils nécessaire à l’étape choisie. Chaque processus fils représente une voiture.

Le processus père, quant à lui, va lire des informations provenant de la mémoire partagée. Il s’occupe également de l’affichage ainsi que du tri tout comme la sauvegarde des informations sur fichier.

Difficultés rencontrées et solutions

Concernant les difficultés rencontrées, suite à la reprise de ce projet pour la seconde session, il n’y a pas eu particulièrement de nouveaux problèmes rencontrés.

Comme indiqué dans le point traitant la compréhension du cahier des charges, grâce à la communication avec la professeur ainsi que du travail réalisé par chacun des membres pour la prise de note, la création d’un flowchart et d’un tableau détaillé reprenant les informations importantes de façon claire et concise, nous avons su éviter la plupart des difficultés possiblement rencontrées.

Compréhension du cahier des charges

Au vu du cahier des charges reçu, nous avons eu des difficultés à comprendre plus concrètement comment mettre en œuvre certaines implémentations demandées tant lors de la première session que lors de la seconde session. À force de recherche et également de questions posées dans le cadre du temps consacré spécifiquement au projet en cours lors de la première session, nous avons accumulé différentes notes nous permettant de mieux visualiser ce qui nous avait posé problème.

Deux rendez-vous ont été convenu avec la professeur (2 avril 2019 à 15h et 25 juin 2019 à 11h) afin de mieux visualiser les demandes, de comprendre ce qui n’avait pas été lors de la première session et également les détails qui n’avaient pas été correctement compris ou nous paraissant tout simplement trop flou.

Suite à ces rendez-vous avec la professeur, le flowchart et le tableau détaillé contenant les informations importantes des demandes dans le cahier des charges fourni, cette difficultée à été résolue.

Évolutions futures

Intégration de codes couleurs dans l’affichage

Il s’agit certes d’une implémentation de moindre importance, mais cela pourrait s’avérer pratique pour ressortir de manière plus rapide les informations les plus importantes. Par exemple, on pourrait réaliser un code couleur pour :

- Les 3 premières places dans le classement,
- Le temps le plus rapide au tour,
- La voiture ayant le temps le plus rapide au tour depuis le début de la course,
- La ou les voiture(s) ayant abandonné la course (OUT).

Affichage cliquable

Comme à la manière de `htop` dans Linux, la possibilité de cliquer sur un des en-têtes de colonne afin de trier automatiquement l’affichage en fonction de cette colonne pourrait s’avérer intéressante. En effet, si l’utilisateur souhaite prêter plus particulièrement son attention sur une catégorie d’information précise, cela pourrait lui être utile.

Options lié à la pression d’une touche de clavier

Une autre idée d’implémentation est de proposer des options en fonction d’un bouton appuyé lorsque le programme est en cours de fonctionnement.

Imaginons par exemple les options suivantes :

- F1 : Help
- F2 : Mettre en pause / Reprendre
- F3 : Afficher / Retirer les codes couleurs
- F4 : Tri en fonction du meilleur temps au tour
- F5 : Tri en fonction du meilleur temps au tour total
- F10 : Quitter

Phase d’essai entièrement libre

Par souci de facilité (et pour se concentrer sur d’autres parties nécessitant plus de temps et de travail), nous avons décidé que les voitures présentes lors d’une séance d’essai libre démarrent toutes comme s’il s’agissent d’une étape classique (une qualification ou une course).

Il serait possible, sans nécessairement y consacrer un temps considérable, de permettre aux différents pilotes de commencer et arrêter leurs séances d’essais libres lorsqu’ils le souhaitent voire même s’ils

rouleront lors de la séance. La question concrète serait : *Est-ce que lors de la limite du temps imparti d’une séance d’essais libres, un pilote souhaite prendre le volant ou non et si oui, pour combien de tours ou combien de temps ?*

Cela correspondrait bien plus à une course de Formule 1 en condition réelle.

Conclusion

L’avantage de ce projet est l’application de concepts multiples vue en cours théorique au courant du premier quadrimestre. Cela nous a permis de comprendre plus concrètement ce que ces concepts permettent de faire (allocation d’une zone mémoire, appel d’une zone mémoire, sémaphores, algorithmes, fork, etc.).

Lors de la première session, ce projet nous avait permis d’apprendre à programmer de façon plus assidue. Lors de l’écriture d’une nouvelle méthode, nous testions systématiquement le projet et en cas de problème, nous prenions le temps de relire le code (et si nécessaire, nous testions différentes méthodes pour déboguer et avancer dans le projet). Nous avions rencontré plusieurs difficultés de compréhension par rapport au cahier des charges ainsi que d’autres difficultés rencontrées, nous avions accumulé un retard par rapport au planning que nous avions fixé au départ, mais l’avions rattrapé en courant de quadrimestre. Malheureusement, la programmation présentée ne correspondant et ne remplissant pas toutes les demandes, cela nous a entraînés dans une seconde tentative pour ce projet.

En cette seconde session, nous avons changé de méthodologie et avons porté une importance quasi capitale sur le fait de réaliser les tests de méthodes. Nous avons appris de nos erreurs et avons eu des moments constructifs d’échange avec la professeur afin de réussir au mieux ce projet. Nous avons également découvert l’utilité de l’utilisation de quelques librairies, ainsi que d’une documentation disponible en ligne, nous permettant de mieux comprendre certaines implémentations nécessaires.

Exemplaire du code

child.c

```c
#include "child.h"
#include "random.h"
#include <stdio.h>
#include <time.h>
#include <unistd.h>

#define divider 100

int time_passed = 0;
int next_pit_lap = 0;
int current_lap = 1;
Options options;
SharedStruct *data;
int min_sec;
int max_sec;
int chance_arg;

int sector() { return next_random(min_sec, max_sec); }

int pit_duration() { return next_random(20000, 25000); }

int pit_lap_rand() { return next_random(15, 25); }

int out_rand() {
    if (data->lap == 1)
        return 0;
    int out = chance(chance_arg);
    if (out) {
        data->out = data->done = 1;
    }

    return out;
}

void set_next_pit() { next_pit_lap = pit_lap_rand() + data->lap; }

// function to sleep during x ms
void sleep_ms(int ms) {
    struct timespec sleep_time = {.tv_sec = 0, .tv_nsec = ms *
       1000000};
    nanosleep(&sleep_time, NULL);
}

int step_done() {
    if (options.choosen_step == RACE) {
        return current_lap == options.number_of_laps;
```

```
46      } else {
47          return time_passed >= options.total_time;
48      }
49  }
50
51  void child(sem_t *sem, SharedStruct *d, Options o, int position) {
52      options = o;
53      data = d;
54
55      int middle = 5500 * options.lap_length;
56      min_sec = middle - 0.125 * middle;
57      max_sec = middle + 0.125 * middle;
58
59      chance_arg = 10000 / options.lap_length;
60
61      init_random((unsigned int)getpid());
62
63      if (options.choosen_step == RACE) {
64          sleep_ms(position * 100);
65      }
66
67      set_next_pit();
68      while (!step_done()) {
69          int sleep_time;
70          sem_wait(sem);
71          data->s1 = sector();
72          data->sector = 1;
73          if (data->best_s1 == 0 || data->best_s1 > data->s1) {
74              data->best_s1 = data->s1;
75          }
76
77          if (out_rand()) {
78              sem_post(sem);
79              exit(0);
80          }
81
82          sleep_time = data->s1 / 100;
83
84          sem_post(sem);
85          sleep_ms(sleep_time);
86
87          sem_wait(sem);
88          data->s2 = sector();
89          data->sector = 2;
90          if (data->best_s2 == 0 || data->best_s2 > data->s2) {
91              data->best_s2 = data->s2;
92          }
93
94          if (out_rand()) {
95              sem_post(sem);
96              exit(0);
```

```
97          }
98
99          sleep_time = data->s2 / 100;
100         sem_post(sem);
101         sleep_ms(sleep_time);
102
103         sem_wait(sem);
104         data->s3 = sector();
105         // if we should make a pit stop now
106         if (data->lap == next_pit_lap) {
107             data->s3 += pit_duration();
108             data->pit++;
109             set_next_pit();
110         }
111         data->sector = 3;
112
113         if (out_rand()) {
114             sem_post(sem);
115             exit(0);
116         }
117
118         sleep_time = data->s3 / 100;
119         if (data->best_s3 == 0 || data->best_s3 > data->s3) {
120             data->best_s3 = data->s3;
121         }
122         int lap_time = data->s1 + data->s2 + data->s3;
123         time_passed += lap_time;
124         if (data->best_lap_time == 0 || data->best_lap_time > lap_time)
                {
125             data->best_lap_time = lap_time;
126         }
127
128         data->lap++;
129         current_lap = data->lap;
130         sem_post(sem);
131         sleep_ms(sleep_time);
132     }
133
134     sem_wait(sem);
135     data->done = 1;
136     sem_post(sem);
137 }
```

child.h

```
1  #pragma once
2
3  #include <semaphore.h>
4  #include "struct.h"
```

```
5  #include "var.h"
6
7  void child(sem_t *sem, SharedStruct *data, Options options, int
       position);
```

curses.c

```
1  #include "curses.h"
2
3  void init_window() {
4      initscr();
5      start_color();
6      init_pair(1, COLOR_WHITE, COLOR_BLACK);
7      init_pair(2, COLOR_BLACK, COLOR_GREEN);
8      refresh();
9  }
10
11 void terminate_window() { endwin(); }
```

curses.h

```
1  #pragma once
2
3  #include <ncurses.h>
4
5  void init_window();
6
7  void terminate_window();
```

display.c

```
1  #include "display.h"
2  #include "../lib/fort.h"
3  #include "time.h"
4  #include "var.h"
5  #include <semaphore.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <unistd.h>
10
11 WINDOW *status_win;
12 Options options;
13 int done = 0;
14 SharedStruct copy[20];
```

```
15
16 int comparator(const void *a, const void *b) {
17     const SharedStruct *pa = (SharedStruct *)a;
18     const SharedStruct *pb = (SharedStruct *)b;
19     if (options.choosen_step != RACE) {
20         if (pa->best_lap_time < pb->best_lap_time)
21             return -1;
22         else if (pa->best_lap_time > pb->best_lap_time)
23             return 1;
24         else
25             return 0;
26     } else {
27         if (pa->out == pb->out) {
28             if (pa->lap < pb->lap)
29                 return 1;
30             else if (pa->lap > pb->lap)
31                 return -1;
32             else {
33                 if (pa->sector > pb->sector)
34                     return 1;
35                 else if (pa->sector < pb->sector)
36                     return -1;
37                 else
38                     return 0;
39             }
40         } else {
41             return pa->out - pb->out;
42         }
43     }
44 }
45
46 void print_table() {
47     ft_table_t *table = ft_create_table();
48
49     ft_set_cell_prop(table, 0, FT_ANY_COLUMN, FT_CPROP_ROW_TYPE,
       FT_ROW_HEADER);
50     ft_write_ln(table, "POSITION", "NAME", "S1", "S2", "S3", "LAP", "
       OUT",
51                 "PIT", "DONE", "BEST LAP TIME");
52
53     for (int i = 0; i < options.participant_count; i++) {
54         SharedStruct current = copy[i];
55         char s1_str[25];
56         to_string(current.s1, s1_str);
57         char s2_str[25];
58
59         if (current.sector == 1)
60             strcpy(s2_str, "");
61         else
62             to_string(current.s2, s2_str);
63
```

```
64          char s3_str[25];
65          if (current.sector < 3)
66              strcpy(s3_str, "");
67          else
68              to_string(current.s3, s3_str);
69
70          char lap_str[25];
71          to_string(current.best_lap_time, lap_str);
72
73          ft_printf_ln(table, "%d|%2s|%8s|%8s|%8s|%d|%d|%d|%d|%10s", i +
               1,
74                       CAR_NAMES[current.id], s1_str, s2_str, s3_str,
75                       current.lap, current.out, current.pit, current.
                            done,
76                       lap_str);
77      }
78
79      const char *table_string = ft_to_string(table);
80      mvprintw(0, 0, "%s\n", table_string);
81      ft_destroy_table(table);
82      refresh();
83  }
84
85  // return car index
86  int best_s1() {
87      int best = 0;
88      int index = 0;
89      for (int i = 0; i < options.participant_count; i++) {
90          if (best == 0 || copy[i].best_s1 < best) {
91              best = copy[i].best_s1;
92              index = i;
93          }
94      }
95      return index;
96  }
97
98  // return car index
99  int best_s2() {
100     int best = 0;
101     int index = 0;
102     for (int i = 0; i < options.participant_count; i++) {
103         if (best == 0 || copy[i].best_s2 < best) {
104             best = copy[i].best_s2;
105             index = i;
106         }
107     }
108     return index;
109 }
110
111 // return car index
112 int best_s3() {
```

```
113     int best = 0;
114     int index = 0;
115     for (int i = 0; i < options.participant_count; i++) {
116         if (best == 0 || copy[i].best_s3 < best) {
117             best = copy[i].best_s3;
118             index = i;
119         }
120     }
121     return index;
122 }
123 
124 void write_status() {
125     mvwprintw(status_win, 0, 0, "%s circuit :%.3fkm\n", options.
            step_name,
126               options.lap_length);
127     if (done) {
128         mvwprintw(status_win, 1, 0, "%s\n", "DONE");
129     } else if (options.choosen_step == RACE) {
130         mvwprintw(status_win, 1, 0, "%d %s\n",
131                   options.number_of_laps - copy[0].lap, "laps remaining
                        ");
132     }
133 
134     int s1 = best_s1();
135     char s1_value[25];
136     to_string(copy[s1].best_s1, s1_value);
137 
138     int s2 = best_s2();
139     char s2_value[25];
140     to_string(copy[s2].best_s2, s2_value);
141 
142     int s3 = best_s3();
143     char s3_value[25];
144     to_string(copy[s3].best_s3, s3_value);
145 
146     mvwprintw(status_win, 3, 0, "Best sectors\n");
147     ft_table_t *table = ft_create_table();
148 
149     ft_set_cell_prop(table, 0, FT_ANY_COLUMN, FT_CPROP_ROW_TYPE,
            FT_ROW_HEADER);
150     ft_write_ln(table, "SECTOR", "NAME", "VALUE");
151     ft_printf_ln(table, "%s|%s|%s", "S1", CAR_NAMES[s1], s1_value);
152     ft_printf_ln(table, "%s|%s|%s", "S2", CAR_NAMES[s2], s2_value);
153     ft_printf_ln(table, "%s|%s|%s", "S3", CAR_NAMES[s3], s3_value);
154 
155     const char *table_string = ft_to_string(table);
156     mvwprintw(status_win, 3, 0, "%s\n", table_string);
157     ft_destroy_table(table);
158     wrefresh(status_win);
159     refresh();
160 }
```

```
161
162 int finished() {
163     for (int i = 0; i < options.participant_count; ++i) {
164         if (!copy[i].done) {
165             return 0;
166         }
167     }
168     return 1;
169 }
170
171 void save_ranking() {
172     FILE *f = fopen(options.step_name, "w");
173     if (f == NULL) {
174         perror("open");
175         exit(1);
176     }
177     for (int i = 0; i < options.participant_count; i++) {
178         fprintf(f, "%d\n", copy[i].id);
179     }
180     if (fclose(f) != 0) {
181         perror("fclose");
182         exit(1);
183     }
184 }
185
186 void display(sem_t *sem, SharedStruct *data, Options o) {
187     options = o;
188     init_window();
189     int y = options.participant_count + 4;
190     status_win = newwin(12, COLS, y, 0);
191
192     while (1) {
193         sem_wait(sem);
194         memcpy(copy, data, sizeof(SharedStruct) * options.
               participant_count);
195         sem_post(sem);
196         qsort(copy, options.participant_count, sizeof(SharedStruct),
197               comparator);
198         if (finished()) {
199             break;
200         }
201         print_table();
202         write_status();
203         sleep(1);
204     }
205     done = 1;
206
207     // still display for 5 seconds
208     for (int i = 0; i < 5; i++) {
209         sem_wait(sem);
```

```
210             memcpy(copy, data, options.participant_count * sizeof(
                    SharedStruct));
211             sem_post(sem);
212             qsort(copy, options.participant_count, sizeof(SharedStruct),
213                   comparator);
214             print_table(copy);
215             write_status();
216             sleep(1);
217         }
218 
219         sleep(6);
220         save_ranking();
221 
222         terminate_window();
223     }
```

display.h

```
1  #pragma once
2  
3  #include <semaphore.h>
4  #include "struct.h"
5  #include "curses.h"
6  #include "var.h"
7  
8  void display(sem_t *sem, SharedStruct *data, Options options);
```

main.c

```
1  #include "child.h"
2  #include "display.h"
3  #include "struct.h"
4  #include "time.h"
5  #include "var.h"
6  #include <errno.h>
7  #include <getopt.h>
8  #include <semaphore.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <sys/ipc.h>
13 #include <sys/shm.h>
14 #include <sys/types.h>
15 #include <sys/wait.h>
16 #include <unistd.h>
17 
18 int main(int argc, char **argv) {
```

```
19      int pid;
20      int i;
21      SharedStruct *data;
22      int valid = 0;
23      Options options;
24      // default value
25      options.lap_length = 7.0;
26
27      static struct option long_options[] = {
28          {"length", required_argument, NULL, 'l'},
29          {"step", required_argument, NULL, 's'},
30          {NULL, 0, NULL, 0}};
31
32      char ch;
33      float lap_length_maybe;
34      while ((ch = getopt_long(argc, argv, "l :s :", long_options, NULL))
            != -1) {
35          switch (ch) {
36          case 'l' :
37              lap_length_maybe = atof(optarg);
38              if (lap_length_maybe < 3 || lap_length_maybe > 10) {
39                  printf("Invalid lap length\n");
40                  exit(1);
41              }
42              options.lap_length = lap_length_maybe;
43              break;
44          case 's' :
45              valid = 1;
46              options.participant_count = 20;
47              if (strcasecmp(optarg, "P1") == 0) {
48                  options.choosen_step = P1;
49                  options.step_name = "P1";
50                  options.total_time = minute_to_ms(90);
51              } else if (strcasecmp(optarg, "P2") == 0) {
52                  options.choosen_step = P2;
53                  options.step_name = "P2";
54                  options.total_time = minute_to_ms(90);
55              } else if (strcasecmp(optarg, "P3") == 0) {
56                  options.choosen_step = P3;
57                  options.step_name = "P3";
58                  options.total_time = minute_to_ms(60);
59              } else if (strcasecmp(optarg, "Q1") == 0) {
60                  options.choosen_step = Q1;
61                  options.step_name = "Q1";
62                  options.total_time = minute_to_ms(18);
63              } else if (strcasecmp(optarg, "Q2") == 0) {
64                  options.choosen_step = Q2;
65                  options.step_name = "Q2";
66                  options.total_time = minute_to_ms(15);
67                  options.participant_count = 15;
68              } else if (strcasecmp(optarg, "Q3") == 0) {
```

```
69                  options.choosen_step = Q3;
70                  options.step_name = "Q3";
71                  options.total_time = minute_to_ms(12);
72                  options.participant_count = 10;
73              } else if (strcasecmp(optarg, "race") == 0) {
74                  options.choosen_step = RACE;
75                  options.step_name = "RACE";
76                  options.total_time = minute_to_ms(120);
77              } else {
78                  printf("invalid step, must be : P1|P2|P3|Q1|Q2|Q3|RACE\n
                        ");
79                  exit(1);
80              }
81
82              break;
83          }
84      }
85
86      options.number_of_laps = 300 / options.lap_length;
87
88      int custom = 0;
89      int ids[options.participant_count];
90      // verify if we need to load previous ranking
91      if (options.choosen_step > 3) {
92          custom = 1;
93          // previous step
94          char *file_to_read;
95          if (options.choosen_step == Q2)
96              file_to_read = "Q1";
97          else if (options.choosen_step == Q3)
98              file_to_read = "Q2";
99          else if (options.choosen_step == RACE)
100             file_to_read = "Q3";
101
102         FILE *f = fopen(file_to_read, "r");
103         if (f == NULL) {
104             printf("Previous rankings missing. please run the step
                        before this "
105                    "one\n");
106             exit(1);
107         }
108         char *line = NULL;
109         size_t len = 0;
110         int count = 0;
111         while (getline(&line, &len, f) != 1) {
112             if (count == options.participant_count ||
113                 (options.choosen_step == RACE && count == 10))
114                 break;
115             int id = atoi(line);
116             ids[count] = id;
117             count++;
```

```
118         }
119
120         fclose(f);
121
122         // we need to get the last 5 from Q2
123         if (options.choosen_step == RACE) {
124             f = fopen("Q2", "r");
125             if (f == NULL) {
126                 printf("A file was deleted ?\n");
127                 exit(1);
128             }
129             line = NULL;
130             len = 0;
131
132             count = 0;
133             while (getline(&line, &len, f) != 1) {
134                 if (count == 15)
135                     break;
136                 if (count < 10) {
137                     count++;
138                     continue;
139                 }
140                 int id = atoi(line);
141                 ids[count] = id;
142                 count++;
143             }
144             fclose(f);
145         }
146
147         // we need to get the last 5 from Q1
148         if (options.choosen_step == RACE) {
149             f = fopen("Q1", "r");
150             if (f == NULL) {
151                 printf("A file was deleted ?\n");
152                 exit(1);
153             }
154             line = NULL;
155             len = 0;
156
157             count = 0;
158             while (getline(&line, &len, f) != 1) {
159                 if (count == 20)
160                     break;
161                 if (count < 15) {
162                     count++;
163                     continue;
164                 }
165                 int id = atoi(line);
166                 ids[count] = id;
167                 printf("%d\n", id);
168                 count++;
```

```
169                 }
170                 fclose(f);
171             }
172         }
173
174         if (!valid) {
175             printf("You need to choose a race step with --step\n");
176             exit(1);
177         }
178
179         int struct_shm_id =
180             shmget(IPC_PRIVATE, sizeof(SharedStruct) * options.
                participant_count,
181                    0600 | IPC_CREAT);
182         if (struct_shm_id == -1) {
183             perror("shmget");
184             exit(1);
185         }
186
187         data = shmat(struct_shm_id, NULL, 0);
188         if (data == (void *)(-1)) {
189             perror("shmat");
190             exit(1);
191         }
192
193         SharedStruct blank = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0};
194         for (int i = 0; i < options.participant_count; i++) {
195             data[i] = blank;
196             if (custom)
197                 data[i].id = ids[i];
198             else
199                 data[i].id = i;
200         }
201
202         int sem_shm_id = shmget(IPC_PRIVATE, sizeof(sem_t), 0600 |
                IPC_CREAT);
203         if (sem_shm_id == -1) {
204             perror("shmget");
205             exit(1);
206         }
207         sem_t *sem = shmat(sem_shm_id, NULL, 0);
208         if (sem == (void *)(-1)) {
209             perror("shmat");
210             exit(1);
211         }
212
213         // init semaphore with 1 as initial value
214         sem_init(sem, 1, 1);
215
216         for (i = 0; i < options.participant_count; i++) {
217             pid = fork();
```

```
218             if (pid == 0)
219                 break;
220         }
221 
222         switch (pid) {
223         case -1 :
224             fprintf(stderr, "fork failed ?!");
225             exit(-1);
226         case 0 :
227             child(sem, &data[i], options, i);
228             exit(0);
229         default :
230             display(sem, data, options);
231 
232             // wait for children process to exit
233             for (int i = 0; i < options.participant_count; i++) {
234                 wait(NULL);
235             }
236 
237             shmdt(data);
238             shmctl(struct_shm_id, IPC_RMID, NULL);
239 
240             sem_destroy(sem);
241             shmdt(sem);
242             shmctl(sem_shm_id, IPC_RMID, NULL);
243 
244             exit(0);
245         }
246 }
```

random.c

```
1 #include "random.h"
2 
3 void init_random(unsigned int seed) { srand(seed); }
4 
5 // produce a random number in the inclusive range [min, max]
6 int next_random(int min, int max) { return rand() % (max + 1 - min) +
      min; }
7 
8 int chance(int arg) { return (rand() % arg) < 1; }
```

random.h

```
1 #pragma once
2 
3 #include <stdlib.h>
```

```
4
5  void init_random(unsigned int seed);
6
7  int next_random(int min, int max);
8
9  int chance(int chance);
```

struct.h

```
1  #pragma once
2
3  typedef struct SharedStruct {
4      int id;
5      int s1;
6      int s2;
7      int s3;
8      int best_s1;
9      int best_s2;
10     int best_s3;
11     int best_lap_time;
12     int lap;
13     int sector;
14     int out;
15     int pit;
16     int done;
17 } SharedStruct;
```

time.c

```
1  #include "time.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  Time time_from_ms(int ms) {
6      Time time;
7      div_t output;
8
9      output = div(ms, 60000);
10     time.m = output.quot;
11     ms = output.rem;
12
13     output = div(ms, 1000);
14     time.s = output.quot;
15     ms = output.rem;
16
17     time.ms = ms;
18
```

```
19      return time;
20  }
21
22  Time minutes(int minutes) { return (Time){.m = minutes, .s = 0, .ms =
        0}; }
23
24  int to_ms(Time time) {
25      int ms = 0;
26      ms += time.ms;
27      ms += time.s * 1000;
28      ms += time.m * 60 * 1000;
29      return ms;
30  }
31
32  int minute_to_ms(int minutes) { return minutes * 60000; }
33
34  void to_string(int ms, char *str) {
35      struct Time time = time_from_ms(ms);
36      sprintf(str, "%d :%d'%d", time.m, time.s, time.ms);
37  }
```

time.h

```
1   #pragma once
2
3   typedef struct Time {
4       int m;
5       int s;
6       int ms;
7   } Time;
8
9   Time time_from_ms(int ms);
10
11  Time minutes(int minutes);
12
13  int to_ms(Time time);
14
15  int minute_to_ms(int minutes);
16
17  void to_string(int ms, char *str);
```

var.c

```
1   #include "var.h"
2
3   char const *const CAR_NAMES[CAR_COUNT] = {
4       "44", "77", "5",  "7",  "3", "33", "11", "31", "18", "35",
```

```
5       "27", "55", "10", "28", "8", "20", "2",  "14", "9",  "16"};
```

var.h

```
1  #pragma once
2
3  #define CAR_COUNT 20
4  extern char const *const CAR_NAMES[];
5
6  enum step {
7      P1, P2, P3, Q1, Q2, Q3, RACE
8  };
9
10 typedef struct Options {
11     enum step choosen_step;
12     char *step_name;
13     int total_time;
14     int number_of_laps;
15     float lap_length;
16     int participant_count;
17 } Options;
```