Rapport du projet de Formule 1 - Groupe 7

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

8 janvier 2018

Contents

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 année : processus père-fils (dont fork est la création d'un nouveau processus utilisateur), sémaphores (pour gérer la synchronisation de la mémoire ainsi que réduire l'accès aux routes partagées) et la mémoire partagée (allocation et utilisation par appel des mémoires partagées via leurs identificateurs).

Cahier des charges

Il y a 20 voitures engagées dans un grand prix. 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 F1 est établi comme suit :

- Vendredi matin, une séance d'essais libres de 1h30 (P1)
- Vendredi après-midi, une séance d'essais libres de 1h30 (P2)
- Samedi matin, une séance d'essais libres de 1h (P3)
- Samedi après-midi, la séance de qualifications, divisé en 3 parties :
- Q1, durée 18 minutes, qui élimine les 5 dernières voitures (qui occuperont les places 16 à 20 sur la grille de départ de la course)
- Q2, durée 15 minutes, qui élimine les 6 voitures suivantes (qui occuperont les places 11 à 16 sur la grille de départ de la course)
- Q3, durée 12 minutes, qui permet de classer les 10 voitures restantes pour établir les 10 premières places sur la grille de départ de la course
- Dimanche après-midi, la course en elle-même.

Votre projet devra prendre en charge les choses suivantes.

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

Lors des séances d'essais (P1, P2, P3) :

- Relever les temps dans les 3 secteurs à chaque passage pour chaque voiture
- Toujours savoir qui a le meilleur temps dans chacun des secteurs
- Classer les voitures en fonction de leur tour complet le plus rapide
- Savoir si une voiture est aux stands (P)
- Savoir si une voiture est out (abandon de la séance)
- Dans ces 2 derniers cas, on conserve toujours le meilleur temps de la voiture et celle-ci reste dans le classement
- Conserver le classement final à la fin de la séance

Lors des qualifications (Q1, Q2, Q3) :

- Relever les temps dans les 3 secteurs à chaque passage pour chaque voiture
- Toujours savoir qui a le meilleur temps dans chacun des secteurs
- Classer les voitures en fonction de leur tour complet le plus rapide
- Savoir si une voiture est aux stands (P)
- Savoir si une voiture est out (abandon de la séance)
- Dans ces 2 derniers cas, on conserve toujours le meilleur temps de la voiture et celle-ci reste dans le classement
- A la fin de Q1, il reste 15 voitures qualifiées pour Q2 et les 5 dernières sont placées à la fin de la grille de départ (places 16 à 20)
- A la fin de Q2, il reste 10 voitures qualifiées pour Q3 et les 5 dernières sont placées dans les places 11 à 15 de la grille de départ
- Le classement de Q3 attribue les places 1 à 10 de la grille de départ
- Conserver le classement final à la fin des 3 séances (ce sera l'ordre de départ pour la course)

Lors de la course :

- Le premier classement est l'ordre sur la grille de départ
- Le classement doit toujours être maintenu tout au long de la course (gérer les dépassements)
- Relever les temps dans les 3 secteurs à chaque passage pour chaque voiture
- Toujours savoir qui a le meilleur temps dans chacun des secteurs
- Toujours savoir qui a le tour le plus rapide
- Savoir si la voiture est out (abandon) ; dans ce cas, elle sera classée en fin de classement
- Savoir si la voiture est aux stands (PIT), gérer le temps aux stands et faire ressortir la voiture à sa place dans la course (généralement 2 ou 3 PIT par voitures)
- Conserver le classement final et le tour le plus rapide

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

De plus, il vous est demandé de paramétrer votre programme.

En effet, les circuits peuvent être de longueur très variable et dès lors 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

Planning

Lors de l'avancement du projet, nous avons travaillé minimum par deux. Nous nous concertions afin de nous réunir (et le cas échéant, nous faisions de la vidéoconférence).

Afin de faciliter l'édition du projet, nous avons utilisé un système de contrôle de versions, nous permettant d'effectuer des changements chacun de notre coté (git). Nous avons également créé un makefile afin de faciliter la compilation du projet.

Nous nous sommes réparti les tâches de la manière suivante :

- Melvin a mis au clair le cahier des charges afin que nous comprenions tous un peu mieux les attentes pour ce projet,
- Dominik et Maxime ont implémenté une fonction permettant la génération du random, utilisée dans la génération du temps des secteurs ainsi que pour certaines probabilités,
- Hubert a pris en charge le côté programmation lorsque les autres membres du groupe ne voyaient pas comment implémenter les demandes du cahier des charges,
- Melvin et Dominik ont debug le projet lors du problème SIGSEGV.

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.

Intéraction entre les processus

Le processus père s'occupe de la gestion de la course, permettant de prévenir les processus enfants quand il s'agit de passer à l'étape suivante. Il choisit également qui est autorisé à passer à l'étape suivante en fonction de son classement. De plus celui-ci gère l'affichage, en utilisant les infos fournies par les processus voitures via une structure en mémoire partagée.

Difficultés rencontrées et solutions

Compréhension du cahier des charges

Au vu du cahier des charges reçu, nous avons eu des difficultés à comprendre plus concrètement comment mettre en oeuvre certaines implémentations demandées. À force de recherche et également de quelques questions posées dans le cadre du temps consacré spécifiquement au projet en cours, nous avons su comment avancer dans le projet de façon plus claire et concise malgré l'impact non négligeable sur la gestion du temps.

Random

Lors de l'implémentation du random, nous avons rencontré des difficultés concernant la génération de plusieurs valeurs différentes et dans un interval précis.

```
1 srand(getpid());
```

Nous avons réglé ce problème en seedant le PRNG *(Pseudo Random Number Generator)* avec le PID, étant unique pour chaque processus.

Conversion du temps

Nous avons également rencontré quelques problèmes lors de l'implémentation de *"timeUnit"*.

"timeUnit" est une structure permettant de passer de millisecondes en toute autre valeur de temps réel, et ce, dans les deux sens.

En effet, nos conversions n'étaient pas correctement écrites au départ, faisant en sorte que nos conversions ne nous rendaient des valeurs peu probables et arrondies; la conversion entre millisecondes et toute autre valeur de temps réel n'étaient pas correctement implémentée.

Après avoir reparcouru le code et avoir corrigé cela, la conversion fonctionne correctement.

Sort

Nous avions rencontré un problème de segmentation et de viol d'intégrité de mémoire partagée suite à l'utilisation du malloc. En effet, malloc était trop petit, impliquant de fait le problème rencontré avec la mémoire partagée.

Synchronisation

Nous avons rencontré un problème de synchronisation entre les sémaphores utilisé dans le projet. En effet, nous n'utilisions pas correctement les fonctions wait et signal des sémaphores, impliquant par conséquent la nécessité de déboguer le projet afin de détecter précisément les zones posant problème et de pouvoir les corriger.

Deadlock

Le problème de deadlock rencontré provenait des processus fils, plus précisément lors de la synchronisation de ces derniers. En déboguant le projet, nous l'avons fixé.

Allocation mémoire dynamique

En utilisant une longueur variable pour le circuit, le nombre de tours varie et il faut donc allouer les tableaux contenant la durée des différents secteurs de manière dynamique. Par manque de temps, nous l'avons pas implémenté.

Évolutions futures

Un meilleur affichage

L'affichage actuellement présent dans le projet n'est pas exactement celui demandé d'après le cahier des charges. Une réimplémentation de l'affichage pour contenir toutes les informations demandées avec, éventuellement, un jeu de couleurs pourrait s'avérer intéressante.

Par exemple, nous pourrions implémenter un affichage ressemblant à ceci :

Afficher en Temps tour si P (P se trouve en S3) ou OUT (si abandon ou crash). Dans le cas de OUT, la voiture est dernière du classement.

Les codes d'affichage pour les tableaux sont :

Code	S	P	OUT
Description	Secteur	Stand (= PIT)	Abandon

Pour les essais libres :

Tri temps meilleur tour

Numéro de la voiture	S1	S2	S3	Temps meilleur tour
2	35	39	38	1min 52s
55	40	36	39	1min 55s
14	39	40	37	1min 56s
28	37	40	39	1min 56s

Pour les qualifications :

Tri temps meilleur tour

Numéro de la voiture	S1	S2	S3	Temps tour	Temps meilleur tour
14	39	40	37	1min 56s	1min 32s
2	35	39	38	1min 52s	1min 47s
55	40	36	39	1min 55s	1min 55s

Numéro de la voiture	S1	S2	S3	Temps tour	Temps meilleur tour
28	39	40	37	1min 56s	1min 39s

Pour la course :

Tri position dans la course

Numéro de la voiture	S1	S2	S3	Temps tour	Lap
2	35	39	38	1min 52s	14
55	40	36	39	1min 55s	14
14	39	40	37	1min 56s	14
28	39	40	37	1min 56s	14

Conclusion

Ce projet nous a 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 (notamment le classique *"print"*) pour déboguer notre méthode et avancer dans le projet. Nous avons pu appliquer plusieurs concepts vu en cours théorique au courant du quadrimestre et comprendre plus concrètement ce que ces concepts permettent de faire (allocation d'une zone mémoire, appel d'une zone mémoire, sémaphores, algorithmes, fork, etc.).

Au début de ce projet, nous avions rencontré quelques difficultés de compréhension par rapport au cahier des charges, ce qui nous a pris un peu de temps au début et, par conséquent, un retard par rapport au planning que nous avons pu rattraper en courant de quadrimestre.

Exemplaire du code

src/main.c

```
1  #include "car.h"
2  #include "carstruct.h"
3  #include "display.h"
4  #include "options.h"
5  #include "sharedstruct.h"
6  #include "util.h"
7  #include <semaphore.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <sys/mman.h>
11 #include <sys/types.h>
12 #include <sys/wait.h>
13 #include <unistd.h>
14
15 int main() {
16     char *CAR_NAMES[NUMBER_OF_CARS] = {"44", "77", "5",  "7",  "3", "33
       ", "11", "31", "18", "35",
17                                        "27", "55", "10", "28", "8", "20
                                             ", "2",  "14", "9",  "16"};
18
19     pid_t pid = 0;
20     int car_index = 0;
21
22     size_t shared_struct_size = sizeof(SharedStruct);
23     SharedStruct *shared_struct = (SharedStruct *)create_shared_memory(
       shared_struct_size);
24
25     sem_t *sem = init_shared_sem(0);
26
27     shared_struct->sem = sem;
28     shared_struct->step = -1;
29
30     for (int i = 0; i < NUMBER_OF_CARS; i++) {
31         shared_struct->car_structs[i].name = CAR_NAMES[i];
32
33         for (int j = 0; j < 7; ++j) {
34             RaceStep race_step = {.lap = 0, .stand = 0, .out = 0, .done
                   = 0, .allowed = 0};
35             shared_struct->car_structs[i].race_steps[j] = race_step;
```

```
36          }
37      }
38
39      for (int i = 0; i < NUMBER_OF_CARS; i++) {
40          car_index = i;
41          pid = fork();
42          if (pid == 0)
43              break;
44      }
45
46      if (pid < 0) {
47          fprintf(stderr, "An error occurred while forking: %d\n", pid);
48          exit(1);
49      } else if (pid == 0) {
50          car(shared_struct, car_index);
51          exit(0);
52      } else {
53          display(shared_struct);
54          for (int i = 0; i < NUMBER_OF_CARS; ++i)
55              wait(NULL);
56          munmap(shared_struct, shared_struct_size);
57          munmap(sem, sizeof(sem_t));
58          exit(0);
59      }
60  }
```

src/display.c

```
1  #include "display.h"
2  #include "options.h"
3  #include "step.h"
4  #include "timeunit.h"
5  #include "util.h"
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <unistd.h>
10
11 // contains index of qualified cars
12 int qualified[NUMBER_OF_CARS];
13 int number_of_cars_allowed;
```

```
14
15 int current_step;
16 char step_name[5];
17
18 // function called from the main method as the parent process
19 void display(SharedStruct *shared_struct) {
20     init_step(shared_struct, P1);
21     display_step(shared_struct, P1);
22
23     init_step(shared_struct, P2);
24     display_step(shared_struct, P2);
25
26     init_step(shared_struct, P3);
27     display_step(shared_struct, P3);
28
29     init_step(shared_struct, Q1);
30     display_step(shared_struct, Q1);
31
32     init_step(shared_struct, Q2);
33     display_step(shared_struct, Q2);
34
35     init_step(shared_struct, Q3);
36     display_step(shared_struct, Q3);
37
38     init_step(shared_struct, RACE);
39     display_step(shared_struct, RACE);
40 }
41
42 // set the current step name, allow the cars for the next step
       depending on their positions
43 // and the signal them with a semaphore
44 void init_step(SharedStruct *shared_struct, int step) {
45     current_step = step;
46
47     // they're all qualified for P1
48     if (step == P1) {
49         for (int i = 0; i < NUMBER_OF_CARS; ++i) {
50             qualified[i] = i;
51         }
52     }
53
54     if (step == P1)
55         strcpy(step_name, "P1");
```

```
56      else if (step == P2)
57          strcpy(step_name, "P2");
58      else if (step == P3)
59          strcpy(step_name, "P3");
60      else if (step == Q1)
61          strcpy(step_name, "Q1");
62      else if (step == Q2)
63          strcpy(step_name, "Q2");
64      else if (step == Q3)
65          strcpy(step_name, "Q3");
66      else if (step == RACE)
67          strcpy(step_name, "race");
68
69      if (step == Q2)
70          number_of_cars_allowed = 15;
71      else if (step == Q3 || step == RACE)
72          number_of_cars_allowed = 10;
73      else
74          number_of_cars_allowed = 20;
75
76      char *names[number_of_cars_allowed];
77
78      for (int i = 0; i < number_of_cars_allowed; ++i) {
79          int car_index = qualified[i];
80          names[i] = shared_struct->car_structs[car_index].name;
81          shared_struct->car_structs[car_index].race_steps[step].allowed
                = 1;
82      }
83
84      printf("\n\nThe followings cars are qualified for the coming step:\
            n");
85      print_car_names(names, number_of_cars_allowed);
86      printf("\n");
87
88      sleep(1);
89
90      shared_struct->step = step;
91
92      for (int i = 0; i < NUMBER_OF_CARS; ++i) {
93          sem_post(shared_struct->sem);
94      }
95  }
96
```

```
97  // function used to display a step
98  void display_step(SharedStruct *shared_struct, int step_index) {
99      printf("Starting %s--------------------------\n\n", step_name);
100
101     while (!done(shared_struct->car_structs)) {
102
103         sleep(1);
104
105         struct e sorted[NUMBER_OF_CARS];
106         if (current_step == RACE) {
107             sort_car_by_lap(sorted, shared_struct->car_structs,
                    current_step);
108         } else {
109             sort_car_by_time(sorted, shared_struct->car_structs,
                    current_step);
110         }
111
112         if (current_step == RACE) {
113             printf("\nname out current lap\n");
114             printf("---- --- -----------\n");
115         } else {
116             printf("\nname out current lap best lap time\n");
117             printf("---- --- ----------- ------------\n");
118         }
119
120         for (int i = 0; i < NUMBER_OF_CARS; ++i) {
121             Car *car = &shared_struct->car_structs[sorted[i].car_index
                    ];
122             RaceStep *race_step = &car->race_steps[step_index];
123             if (!race_step->allowed)
124                 continue;
125
126             char time[25];
127             to_string(sorted[i].value, time);
128
129             char out[4];
130             if (race_step->out)
131                 strcpy(out, "yes");
132             else
133                 strcpy(out, "no");
134
135             if (current_step == RACE) {
136                 printf("%2s   %-3s %-2d\n", car->name, out, race_step->
```

```
                    lap);
137             } else {
138                 printf("%2s   %-3s %-2d          %s\n", car->name, out,
                        race_step->lap, time);
139             }
140         }
141     }
142
143     printf("%s done------------------------------\n\n", step_name);
144     printf("Summary\n");
145
146     struct e sorted[NUMBER_OF_CARS];
147     if (current_step == RACE) {
148         sort_car_by_lap(sorted, shared_struct->car_structs,
            current_step);
149     } else {
150         sort_car_by_time(sorted, shared_struct->car_structs,
            current_step);
151     }
152
153     if (current_step == RACE) {
154         printf("\nname out current lap\n");
155         printf("---- --- -----------\n");
156     } else {
157         printf("\nname out current lap best lap time\n");
158         printf("---- --- ----------- ------------\n");
159     }
160
161     for (int i = 0; i < NUMBER_OF_CARS; ++i) {
162         Car *car = &shared_struct->car_structs[sorted[i].car_index];
163         RaceStep *race_step = &car->race_steps[step_index];
164         if (!race_step->allowed)
165             continue;
166
167         char time[25];
168         to_string(sorted[i].value, time);
169
170         char out[4];
171         if (race_step->out)
172             strcpy(out, "yes");
173         else
174             strcpy(out, "no");
175
```

```
176             if (current_step == RACE) {
177                 printf("%2s   %-3s %-2d\n", car->name, out, race_step->lap)
                        ;
178             } else {
179                 printf("%2s   %-3s %-2d          %s\n", car->name, out,
                        race_step->lap, time);
180             }
181         }
182 
183         if (current_step == RACE) {
184 
185             for (int i = 0; i < NUMBER_OF_CARS; ++i) {
186                 Car *car = &shared_struct->car_structs[sorted[i].car_index
                        ];
187                 RaceStep *race_step = &car->race_steps[step_index];
188 
189                 if (!race_step->allowed || race_step->out)
190                     continue;
191                 printf("\n\nThe winner of the race is %s !!!\n", car->name)
                        ;
192                 break;
193             }
194 
195         } else {
196             sleep(2);
197         }
198     }
199 
200     // function returning 1 if the current step is done
201     int done(Car *cars) {
202         int count = 0;
203         for (int i = 0; i < NUMBER_OF_CARS; ++i) {
204             Car *car = &cars[i];
205             if (!car->race_steps[current_step].allowed)
206                 continue;
207             if (!car->race_steps[current_step].done)
208                 ++count;
209         }
210         return count == 0;
211     }
```

src/car.c

```c
#include "car.h"
#include "random.h"
#include "step.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

//#define DEBUG

#ifdef DEBUG
#define DIVIDER 100000
#endif // DEBUG

#ifndef DEBUG
#define DIVIDER 1000
#endif // DEBUG

// the average time of a sector
int average_time;

int variance;

// 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);
}

// function called after the fork, once for each car
void car(SharedStruct *shared_struct, int index) {
    init_rand((unsigned int)getpid());

    average_time = 40000;
    variance = average_time * 10 / 100;

    step(shared_struct, index, P1, minutes(90), 0);
    step(shared_struct, index, P2, minutes(90), 0);
    step(shared_struct, index, P3, minutes(60), 0);
```

```
40
41      step(shared_struct, index, Q1, minutes(18), 0);
42      step(shared_struct, index, Q2, minutes(15), 0);
43      step(shared_struct, index, Q3, minutes(12), 0);
44
45      // TODO
46      int lap_number = 15;
47      step(shared_struct, index, RACE, minutes(90), lap_number);
48      exit(0);
49  }
50
51  // function used to generate random times and sleep depending on the
        value
52  // the values are also assigned inside the race_step struct
53  void generate_lap(RaceStep *race_step, int lap) {
54      race_step->stand = 0;
55      for (int i = 0; i < 3; ++i) {
56          int rand = bounded_rand(average_time - variance, average_time +
                variance);
57          sleep_ms(rand / DIVIDER);
58
59          if (i == 2 && proba(1, 100)) {
60              int time_at_stand = bounded_rand(19000, 21000);
61              sleep_ms(time_at_stand / DIVIDER);
62              rand += time_at_stand;
63              race_step->stand = 1;
64          }
65
66          if (lap != 0 && proba(1, 600)) {
67              race_step->out = 1;
68              break;
69          }
70
71          race_step->time[lap][i] = rand;
72      }
73  }
74
75  // function called once for each step of the formula 1 weekend
76  void step(SharedStruct *shared_struct, int car_index, int step_index,
        TimeUnit min, int lap_number) {
77      while (shared_struct->step != step_index) {
78      }
79
```

```
80      sem_wait(shared_struct->sem);
81
82      Car *car = &shared_struct->car_structs[car_index];
83      RaceStep *race_step = &car->race_steps[step_index];
84
85      if (!race_step->allowed) {
86          race_step->done = 1;
87          return;
88      }
89
90      int total_time = to_ms(min);
91
92      int current_time = 0;
93      int lap = 0;
94
95      while (1) {
96          generate_lap(race_step, lap);
97
98          if (race_step->out)
99              break;
100
101         int sum = 0;
102         sum += race_step->time[lap][0];
103         sum += race_step->time[lap][1];
104         sum += race_step->time[lap][2];
105
106         if ((step_index != RACE && current_time + sum > total_time) ||
                (step_index == RACE && lap == lap_number)) {
107             break;
108         } else {
109             current_time += sum;
110             race_step->lap = lap++;
111         }
112     }
113
114     race_step->done = 1;
115 }
```

src/random.c

```
1  #include "random.h"
```

```c
#include <stdlib.h>
#include <time.h>

void init_rand(unsigned seed) { srand(seed); }

/* example:
 * proba(2, 100) returns true 2% of the time;
 */
int proba(int n, int m) { return (rand() % m) < n; }

// min included, max excluded
int bounded_rand(int min, int max) { return rand() % (max + 2 - min) + min; }
```

src/timeunit.c

```c
#include "timeunit.h"
#include <stdio.h>
#include <stdlib.h>

TimeUnit to_time_unit(int ms) {
    struct TimeUnit timeUnit;
    div_t output;

    output = div(ms, 60000);
    timeUnit.m = output.quot;
    ms = output.rem;

    output = div(ms, 1000);
    timeUnit.s = output.quot;
    ms = output.rem;

    timeUnit.ms = ms;

    return timeUnit;
}

TimeUnit minutes(int minutes) { return (TimeUnit){.m = minutes, .s = 0, .ms = 0}; }

int to_ms(TimeUnit timeUnit) {
```

```
25      int ms = 0;
26      ms += timeUnit.ms;
27      ms += timeUnit.s * 1000;
28      ms += timeUnit.m * 60 * 1000;
29      return ms;
30  }
31
32  void to_string(int ms, char *str) {
33      struct TimeUnit time = to_time_unit(ms);
34      sprintf(str, "%d:%d'%d", time.m, time.s, time.ms);
35  }
```

src/util.c

```
1   #include "util.h"
2   #include "options.h"
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6   #include <sys/mman.h>
7   #include <unistd.h>
8
9   // print car names assuming a name length is at max 2
10  void print_car_names(char **names, int length) {
11      int size = 4 * length;
12      char str[size];
13      strcpy(str, "");
14      for (int i = 0; i < length; ++i) {
15          strcat(str, names[i]);
16          if (i != length - 1)
17              strcat(str, ", ");
18      }
19      printf("%s\n", str);
20  }
21
22  int comp(const void *a, const void *b) {
23      int first = ((struct e *)a)->value;
24      int second = ((struct e *)b)->value;
25
26      if (first == second)
27          return 0;
```

```
28      else if (first == -1)
29          return -1;
30      else if (second == -1)
31          return 1;
32      else
33          return first - second;
34  }
35
36  void sort_car_by_time(struct e *result, Car *car, int step) {
37      for (int i = 0; i < NUMBER_OF_CARS; ++i) {
38          Car *c = &car[i];
39          RaceStep *race_step = &c->race_steps[step];
40          if (!race_step->allowed) {
41              result[i] = (struct e){.car_index = i, .value = -1};
42              continue;
43          }
44
45          int lap_count = race_step->lap;
46          int laps[lap_count];
47
48          for (int j = 0; j < lap_count; ++j) {
49              laps[j] = race_step->time[lap_count][0];
50              laps[j] += race_step->time[lap_count][1];
51              laps[j] += race_step->time[lap_count][2];
52          }
53
54          result[i] = (struct e){.car_index = i, .value = min_from_array(
                laps, lap_count)};
55      }
56
57      qsort(result, NUMBER_OF_CARS, sizeof(struct e), comp);
58  }
59
60  void sort_car_by_lap(struct e *result, Car *car, int step) {
61      for (int i = 0; i < NUMBER_OF_CARS; ++i) {
62          Car *c = &car[i];
63          RaceStep *race_step = &c->race_steps[step];
64          if (!race_step->allowed) {
65              result[i] = (struct e){.car_index = i, .value = -1};
66              continue;
67          }
68
69          int lap_count = race_step->lap;
```

```
70
71          result[i] = (struct e){.car_index = i, .value = lap_count};
72      }
73
74      qsort(result, NUMBER_OF_CARS, sizeof(struct e), comp);
75  }
76
77  int min_from_array(const int *array, int size) {
78      int min_value = -1;
79      for (int i = 0; i < size; ++i) {
80          if (array[i] < min_value || min_value == -1) {
81              min_value = array[i];
82          }
83      }
84      return min_value;
85  }
86
87  // utility function returning a pointer from shared memory
88  void *create_shared_memory(size_t size) {
89      int prot = PROT_READ | PROT_WRITE;
90      int flags = MAP_ANONYMOUS | MAP_SHARED;
91      void *ptr = mmap(NULL, size, prot, flags, 0, 0);
92      if (ptr == MAP_FAILED) {
93          fprintf(stderr, "Error mmap\n");
94          exit(1);
95      }
96      return ptr;
97  }
98
99  // utility function returning a pointer to a semaphore in shared memory
100 sem_t *init_shared_sem(unsigned int init_value) {
101     sem_t *sem = (sem_t *)create_shared_memory(sizeof(sem_t));
102     sem_init(sem, 1, init_value);
103     return sem;
104 }
```

src/car.h

```
1  #ifndef CAR_H
2  #define CAR_H
3
```

```
4  #include "carstruct.h"
5  #include "timeunit.h"
6  #include "sharedstruct.h"
7
8
9  void car(SharedStruct *shared_struct, int index);
10
11 void step(SharedStruct *shared_struct, int car_index, int step_index,
       TimeUnit min, int lap_number);
12
13 #endif //CAR_H
```

src/carstruct.h

```
1  #ifndef CARSTRUCT_H
2  #define CARSTRUCT_H
3
4  #include "racestep.h"
5
6  // struct containing a car name and a race step struct for each step
7  typedef struct Car {
8      char *name;
9
10     RaceStep race_steps[7];
11 } Car;
12
13 #endif //CARSTRUCT_H
```

src/display.h

```
1  #ifndef DISPLAY_H
2  #define DISPLAY_H
3
4  #include <semaphore.h>
5  #include "carstruct.h"
6  #include "sharedstruct.h"
7
8  void display(SharedStruct *shared_struct);
9
10 void init_step(SharedStruct *shared_struct, int step);
```

```
11
12 void display_step(SharedStruct *shared_struct, int step_index);
13
14 int done(Car *cars);
15
16 #endif //DISPLAY_H
```

src/options.h

```
1 #ifndef OPTIONS_H
2 #define OPTIONS_H
3
4 #define NUMBER_OF_CARS 20
5
6 #endif //OPTIONS_H
```

src/racestep.h

```
1  #ifndef RACESTEP_H
2  #define RACESTEP_H
3
4  // struct containing informations about a formula 1 step
5  typedef struct RaceStep {
6      int allowed;
7
8      int lap;
9
10     int stand;
11     int out;
12
13     int time[48][3];
14
15     int done;
16 } RaceStep;
17
18
19 #endif //RACESTEP_H
```

src/random.h

```h
#ifndef RANDOM_H
#define RANDOM_H

void init_rand(unsigned seed);

int proba(int n, int m);

int bounded_rand(int min, int max);

#endif //RANDOM_H
```

src/sharedstruct.h

```h
#ifndef STRUCT_H
#define STRUCT_H

#include "carstruct.h"
#include "options.h"
#include <semaphore.h>

// struct containing all the informations shared beetween cars and the
    main process
typedef struct SharedStruct{
    Car car_structs[NUMBER_OF_CARS];

    sem_t *sem;

    int step;
} SharedStruct;

#endif //STRUCT_H
```

src/step.h

```h
#ifndef STEP_H
#define STEP_H

#define P1 0
```

```
5  #define P2 1
6  #define P3 2
7
8  #define Q1 3
9  #define Q2 4
10 #define Q3 5
11
12 #define RACE 6
13
14 #endif //STEP_H
```

src/timeunit.h

```
1  #ifndef TIMEUNIT_H
2  #define TIMEUNIT_H
3
4  typedef struct TimeUnit {
5      int m;
6      int s;
7      int ms;
8  } TimeUnit;
9
10 TimeUnit to_time_unit(int ms);
11
12 TimeUnit minutes(int minutes);
13
14 int to_ms(TimeUnit timeUnit);
15
16 void to_string(int ms, char *str);
17
18
19 #endif //TIMEUNIT_H
```

src/util.h

```
1  #ifndef UTIL_H
2  #define UTIL_H
3
4  #include "carstruct.h"
5  #include <sys/types.h>
```

```
6  #include <semaphore.h>
7
8  // TODO rename
9  struct e {
10     int car_index;
11     int value;
12 };
13
14 void print_car_names(char **names, int length);
15
16 int number_of_car_allowed(Car *cars, int step);
17
18 void sort_car_by_time(struct e *result, Car *car, int step);
19
20 void sort_car_by_lap(struct e *result, Car *car, int step);
21
22 int min_from_array(const int *array, int size);
23
24 void *create_shared_memory(size_t size);
25
26 sem_t *init_shared_sem(unsigned int init_value);
27
28 #endif //UTIL_H
```