EPHEC

Rapport de projet “Formule 1”
Groupe 4

OS travaux pratiques

Daniel O., Martin M., Morgan V., Martin P.

07 dec 2019

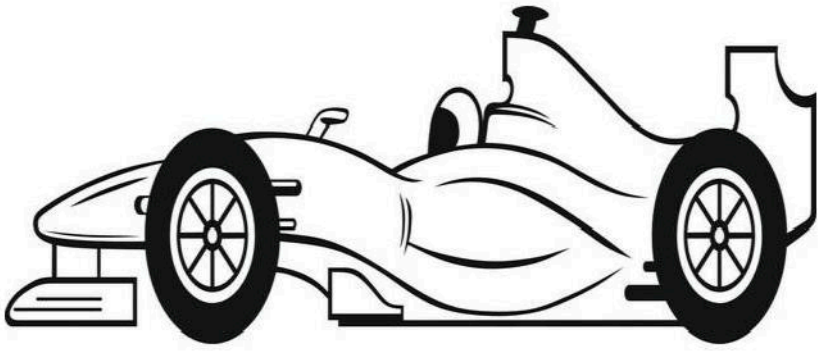

Table des matières

1 Rapport du projet : F1-of-Linux

Groupe 4

Notre groupe est constitué de 4 personnes :

- Daniel Olivier
- Martin Michotte
- Morgan Valentin
- Martin Perdaens

1.1 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).

1.2 Cahier des charges du projet

1.2.1 Projet OS Octobre 2019

Le but du projet est de gérer 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.

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 d’1h30 (P1)
- Vendredi après-midi, une séance d’essais libres d’1h30 (P2)
- Samedi matin, une séance d’essais libres d’1h (P3)
- Samedi après-midi, la séance de qualifications, divisée 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.

1.2.2 Première partie : gestion des séances d’essai, des qualifications et de la 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 voitures 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 inter-processus
- ☒ Utiliser les sémaphores pour synchroniser l’accès à la mémoire partagée

1.3 Explication des particularités du code

1.3.1 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.

1.3.2 Mémoire partagée et communication entre processus

La mémoire partagée est un moyen efficace de transférer des données entre processus indépendants (issus de programmes binaires séparés, de propriétaires différents). Il s'agit d'un ensemble d'adresses (perçu sous la forme d'un bloc d'octets) apparaissant dans l'espace d'adressage du processus qui le crée. Les autres processus pouvant alors « attacher » le même segment de mémoire partagée dans leur propre espace d'adressage (virtuel).

Si un processus écrit dans la mémoire partagée, la modification est immédiatement perçue par tout autre processus ayant accès à cette mémoire partagée.

La mémoire partagée ne dispose d'aucun dispositif de synchronisation. Rien ne permet de veiller automatiquement à ce qu'un processus ne puisse commencer à lire la mémoire alors qu'un autre processus n'y a pas terminé son écriture : c'est au programmeur de régler l'accès à cette ressource commune aux processus ayant accès à cette mémoire partagée.

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

```
1 typedef struct F1_Car {
2     int id;
3     double lap_time;
4     double s1;
5     double s2;
```

```
6       double s3;
7       int best_s1;
8       int best_s2;
9       int best_s3;
10      int stand;
11      int out;
12      int lap;
13      int best_lap_time;
14      int done;
15  } F1_Car;
```

Sous Linux, la mémoire partagée peut s’utiliser via les appels systèmes `shmget`, `shmat` et `shmdt`. L’appel système `shmget` permet de créer un segment de mémoire partagée. Le premier argument de `shmget` est une clé qui identifie le segment de mémoire partagée. Cette clé est en pratique encodée sous la forme d’un entier qui identifie le segment de mémoire partagée. Elle sert d’identifiant du segment de mémoire partagée dans le noyau. Un processus doit connaître la clé qui identifie un segment de mémoire partagée pour pouvoir y accéder. On utilise la clé IPC_PRIVATE pour la création de ce dernier. Le deuxième argument de `shmget` donne le nombre d’octets du segment. Enfin, le troisième argument est une combinaison (par OU bit à bit) de constantes (telles que IPC_CREAT pour la création) et de droits d’accès (comme 0666). Par exemple pour créer un segment on utilisera typiquement l’option IPC_CREAT|0666, et pour l’acquisition simplement 0666.

```
1  #include <sys/ipc.h>
2  #include <sys/shm.h>
3
4  int shmget(key_t key, size_t size, int shmflg);
```

L’appel système `shmget` retourne un entier qui identifie le segment de mémoire partagée à l’intérieur du processus si il réussit et -1 sinon. Il est important de noter que si l’appel à`shmget` réussit, cela indique que le processus dispose des permissions pour accéder au segment de mémoire partagée, mais à ce stade il n’est pas accessible depuis la table des pages du processus.

```
1  F1_Car *car;
2    int struct_shm_id = shmget(IPC_PRIVATE, sizeof(F1_Car) * circuit.
       number_of_cars, 0600 | IPC_CREAT);
3      if (struct_shm_id == -1) {
4          perror("shmget failed !");
5          exit(EXIT_FAILURE);
6      }
```

Cette modification à la table des pages du processus se fait en utilisant `shmat`. Cet appel système permet d’attacher un segment de mémoire partagée à un processus. Il prend comme premier argument l’identifiant du segment de mémoire retourné par `shmget`. Le deuxième argument est un pointeur vers la zone mémoire via laquelle le segment doit être accessible dans l’espace d’adressage virtuel du processus. Généralement, c’est la valeur NULL qui est spécifiée comme second argument et le noyau choisit l’adresse à laquelle le segment de mémoire est attaché dans le processus. Il est aussi possible

de spécifier une adresse dans l’espace d’adressage du processus. Le troisième argument permet, en utilisant le drapeau SHM_RDONLY, d’attacher le segment en lecture seule ou 0 pour un segment en lecture/écriture. `shmat` retourne l’adresse à laquelle le segment a été attaché en cas de succès et (void *) -1 en cas d’erreur.

```c
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

```c
   car = shmat(struct_shm_id, NULL, 0);
    if (car == (void *) (-1)) {
        perror("shmat failed !");
        exit(EXIT_FAILURE);
    }
```

1.3.3 Sémaphores

La notion de sémaphore est implémentée dans la plupart des systèmes d’exploitation. Il s’agit d’un concept fondamental car il permet une solution à la plupart des problèmes d’exclusion.

Un sémaphore est un mécanisme empêchant deux processus ou plus d’accéder simultanément à une ressource partagée. Sur les voies ferrées, un sémaphore empêche deux trains d’entrer en collision sur un tronçon de voie commune. Sur les voies ferrées comme dans les ordinateurs, les sémaphores ne sont qu’indicatifs : si un machiniste ne voit pas le signal ou ne s’y conforme pas, le sémaphore ne pourra éviter la collision.

De même si un processus ne teste pas un sémaphore avant d’accéder à une ressource partagée, le chaos peut en résulter. Un sémaphore binaire n’a que deux états :

- 0 verrouillé (ou occupé).
- 1 déverrouillé (ou libre).

Quand il vaut zéro, un processus tentant de l’acquérir doit attendre qu’un autre processus ait augmenté sa valeur car le sémaphore ne peut jamais devenir négatif. Dans le cas de notre projet on utilise les sémaphores POSIX qui sont disponibles dans la librairie standard C (GNU). La glibc offre donc une implémentation des sémaphores.

Dans notre projet on utilise les fonctions suivantes de la librairie pour gérer un sémaphore de type `sem_t` :

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Pour pouvoir utiliser un sémaphore, il faut d’abord l’initialiser. Cela se fait en utilisant la fonction `sem_init` qui prend comme premier argument un pointeur vers le sémaphore à initialiser, deuxième argument pshared indique si ce sémaphore sera partagé entre les threads d’un processus ou entre processus. Si pshared vaut 0, le sémaphore est partagé entre les threads d’un processus si non c’est entre les processus. Enfin, le troisième argument value spécifie la valeur initiale du sémaphore.

Les deux principales fonctions de manipulation des sémaphores sont `sem_wait` et `sem_post`.

- `sem_wait()` décrémente (verrouille) le sémaphore pointé par sem. Si la valeur du sémaphore est plus grande que 0, la décrémentation s’effectue et la fonction revient immédiatement. Si le sémaphore vaut zéro, l’appel bloquera jusqu’à ce que soit il devienne disponible pour effectuer la décrémentation (c’est-à-dire la valeur du sémaphore n’est plus nulle), soit un gestionnaire de signaux interrompe l’appel.
- `sem_post()` incrémente (déverrouille) le sémaphore pointé par sem. Si, à la suite de cet incrément, la valeur du sémaphore devient supérieure à zéro, un autre processus ou thread bloqué dans un appel `sem_wait()` sera réveillé et procédera au verrouillage du sémaphore.

Ces deux opérations sont bien entendu des opérations qui ne peuvent s’exécuter simultanément. Leur implémentation réelle comprend des sections critiques qui doivent être construites avec soin. La section critique dans notre projet est lors de l’affichage.

```
1  #include <semaphore.h>
2
3  sem_t *sem;
4
5  sem_init(sem, 1, 1);
6
7  sem_wait(sem);
8  // section critique : affichage voir le code dans le fichier display.c
9  sem_post(sem);
10
11 sem_destroy(sem);
```

1.3.4 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 « détachement » de la mémoire partagée et ensuite ce dernier est supprimée.

L’appel système `shmdt` permet de détacher un segment de mémoire qui avait été attaché en utilisant `shmat`. L’argument passé à `shmdt` doit être l’adresse d’un segment de mémoire attaché préalablement par `shmat`. Lorsqu’un processus se termine, tous les segments auxquels il était attaché sont détachés lors de l’appel à `exit`.

Détacher la mémoire partagée ne la supprime pas. Détacher la mémoire partagée permet juste de casser la correspondance entre les pages de l’espace virtuel dédiées au segment de mémoire et les pages frames de la mémoire physique dédiées au segment de mémoire partagée. Pour réellement la mémoire partagée on fait appel à la fonction `shmctl`.

L’appel système `shmctl` prend trois arguments. Le premier est un identifiant de segment de mémoire partagée retourné par `shmget`. Le deuxième est une constante qui spécifie une commande. On utilise uniquement la commande IPC_RMID qui permet de retirer le segment de mémoire partagée dont l’identifiant est passé comme premier argument. Si il n’y a plus de processus attaché au segment de mémoire partagée, celui-ci est directement supprimé. Sinon, il est marqué de façon à ce que le noyau retire le segment dès que le dernier processus s’en détache. `shmctl` retourne 0 en cas de succès et -1 en cas d’échec.

```
1 #include <sys/ipc.h>
2 #include <sys/shm.h>
3
4 int shmdt(const void *shmaddr);
5 int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Deuxièmement, on fait presque la même chose avec les sémaphores mais avec des fonctions différentes. La fonction `sem_destroy` permet de libérer un sémaphore qui a été initialisé avec `sem_init`. Les sémaphores consomment des ressources qui peuvent être limitées dans certains environnements. Il est important de détruire proprement les sémaphores dès qu’ils ne sont plus nécessaires.

```
1 shmdt(car);
2 shmctl(struct_shm_id, IPC_RMID, NULL);
3 sem_destroy(sem);
```

1.3.5 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 la fonction `fork`, faisant partie des appels système POSIX. Elle permet de donner naissance à un nouveau processus qui est sa copie.

La création des processus fils est 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 des étapes de qualifications et de la course.

Rôle des processus fils

Dans le cadre de ce projet, les fils sont seulement chargés à courir. Càd exécuter les étapes à faire pour un week-end complet d’un grand prix de Formule 1. Pour y arriver on utilise une boucle `while()` avec comme condition si le temps de l’étape chosi n’a pas écoulé, alors les fils courent. Pour la course de dimanche les fils courent tant qu’ils n’ont pas fini les tours à faire. Les crash sont gérés à l’interieur du la boucle `while()`.

Le code du fils est présent dans le fichier de code source `child.c`.

Affichage

Pour pouvoir les données dans table, on utilise une librairie public `libfort` disponible sur github : https://github.com/seleznevae/libfort. Voilà un exemple du résultat lors de l’étape Q2.

Position	NAME	S1	S2	S3	OUT	PIT	LAP	LAP TIME	BEST LAP TIME
1	7	32” :03	39” :03	44” :77	0	0	13	1’ :02” :19	1’ :19” :42
2	35	42” :53	41” :27	39” :23	0	1	9	1’ :11” :71	1’ :22” :31
3	40	36” :13	30” :03	44” :12	0	0	16	1’ :03” :36	1’ :44” :28
4	40	40” :04	43” :03	43” :03	0	0	10	1’ :40” :11	1’ :51” :47
5	77	33” :11	34” :43	42” :59	0	1	11	1’ :17” :23	2’ :12” :73

Il y a également une deuxième table pour savoir qui a le meilleur temps dans chacun des secteurs.

SECTOR	NAME	TIME
S1	3	31” :03
S2	42	33” :27
S3	36	38” :44

Le code de la crétion de ces tables est présent dans le fichier de code source `display.c`. Les autres parties du code arrivent par après.

Le trie

Pour pouvoir classer les voitures en fonction de leur tour complet le plus rapide, ou en fonction de leur rapididté lors la course, on utilise la fonction de la librairie `qsort`.

```
1 void qsort(void *base, size_t nel, size_t width,
2            int (*compar)(const void *, const void *));
```

Le premier est un pointeur vers le début de la zone mémoire à trier. Le second est le nombre d’élé-

ments à trier. Le troisième contient la taille des éléments stockés dans le tableau. Le quatrième argument est un pointeur vers la fonction qui permet de comparer deux éléments du tableau. Cette fonction retourne un entier négatif si son premier argument est inférieur au second et positif ou nul sinon.

Les deux arguments de type `(const void *)` font appel à l'utilisation de pointeurs `(void *)` qui est nécessaire car la fonction doit être générique et pouvoir traiter n'importe quel type de pointeurs. `(void *)` est un pointeur vers une zone quelconque de mémoire qui peut être casté vers n'importe quel type de pointeur par la fonction de comparaison. Le qualificatif const indique que la fonction n'a pas le droit de modifier la donnée référencée par ce pointeur, même si elle reçoit un pointeur vers cette donnée. On retrouve régulièrement cette utilisation de const dans les signatures des fonctions de la librairie pour spécifier des contraintes sur les arguments passés à une fonction.

Un exemple de fonction de comparaison est la fonction `strcmp` de la librairie standard. Le pseudo-code repris ci-dessous est notre implémentation de la fonction `qsort`.

```
int compare(const void *left, const void *right) {
    const F1_Car *process_a = (F1_Car *) left;
    const F1_Car *process_b = (F1_Car *) right;

    if (strcmp(circuit.step_name, "RACE")) {
        if (process_a->best_lap_time < process_b->best_lap_time)
            return -1;
        else if (process_a->best_lap_time > process_b->best_lap_time)
            return 1;
        else
            return 0;
    } else {
        if (process_a->lap < process_b->lap)
            return 1;
        else if (process_a->lap > process_b->lap)
            return -1;
        else
            return 0;
    }
}

sem_wait(sem);
memcpy(car_array, data, sizeof(F1_Car) * circuit.number_of_cars);
sem_post(sem);
qsort(car_array, circuit.number_of_cars, sizeof(F1_Car), compare);
```

Avant de trier on fait une copie des données du struct partagée entre les processus par la fonction `memcpy`. Cette fonction permet de copier un bloc de mémoire spécifié par le paramètre source, et dont la taille est spécifiée via le paramètre size, dans un nouvel emplacement désigné par le paramètre destination. Il est bien entendu qu'il est de notre responsabilité d'allouer suffisamment de mémoire pour le bloc de destination afin qu'il puisse contenir toutes les données.

1.4 Difficultés rencontrées et solutions

Concernant les difficultés rencontrées, la mémoire partagée et le classement de départ pour la course étaient les plus gros challenges du projet.

1.5 Évolutions futures

1.5.1 Intégration de codes couleurs dans l’affichage : DONE !

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).

1.5.2 Affichage cliquable : TODO !

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.

1.5.3 Options lié à la pression d’une touche de clavier : TODO !

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

1.5.4 Phase d’essai entièrement libre : TODO !

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.

2 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…).

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. Malheureusement, la programmation présentée ne correspondant et ne remplissant pas toutes les demandes, cela nous a entraînés dans une seconde tentative pour 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.

3 Exemplaire du code

3.1 child.c

```c
#include "child.h"

int time_passed = 0;
int current_lap = 0;
F1_Car *vehicle;
Circuit circuit;

void car_crash() {
    if (car_crashed(10000000))
        vehicle->out = 1;
    else
        vehicle->out = 0;
}

int finished_running() {
    if (!strcmp(circuit.step_name, "RACE")) {
        return current_lap == circuit.number_of_laps;
    } else {
        return time_passed >= circuit.step_total_time;
    }
}

int msleep(unsigned int tms) {
    return usleep(tms * 1000);
}

void child(sem_t *sem, F1_Car *car, int *car_names) {

    random_seed(getpid());
    vehicle = car;
    vehicle->id = *car_names;

    while (!finished_running()) {

        //(!strcmp(circuit.step_name, "RACE")) ? sleep(10) : 0;

        sem_wait(sem);
        vehicle->s1 = sector_range(30, 45, 10000000);
        if (vehicle->best_s1 == 0 || vehicle->best_s1 > vehicle->s1) {
            vehicle->best_s1 = vehicle->s1;
        }
        car_crash();
        sem_post(sem);

        sem_wait(sem);
```

```
46          vehicle->s2 = sector_range(30, 45, 10000000);
47          if (vehicle->best_s2 == 0 || vehicle->best_s2 > vehicle->s2) {
48              vehicle->best_s2 = vehicle->s2;
49          }
50          car_crash();
51          sem_post(sem);
52
53          sem_wait(sem);
54          vehicle->s3 = sector_range(30, 45, 10000000);
55
56          int i = 1;
57          vehicle->stand = 0;
58          while (stand_probability(10)) {
59
60              vehicle->s3 += stand_duration(1, 100);
61              i++;
62              vehicle->stand = 1;
63          }
64          if (vehicle->best_s3 == 0 || vehicle->best_s3 > vehicle->s3) {
65              vehicle->best_s3 = vehicle->s3;
66          }
67          car_crash();
68          msleep(80);
69
70          vehicle->lap_time = vehicle->s1 + vehicle->s2 + vehicle->s3;
71          time_passed += vehicle->lap_time;
72
73          if (vehicle->best_lap_time == 0 ||
74              vehicle->best_lap_time > vehicle->lap_time)
75              vehicle->best_lap_time = vehicle->lap_time;
76          vehicle->lap++;
77          current_lap = vehicle->lap;
78          (time_passed >= circuit.step_total_time || current_lap ==
               circuit.number_of_laps) ? vehicle->done = 1 : 0;
79          sem_post(sem);
80          sleep(1);
81      }
82  }
```

3.2 child.h

```
1  //
2  // Created by danny on 2/10/19.
3  //
4  #pragma once
5
6  #include "time.h"
7  #include "prng.h"
8  #include <semaphore.h>
```

```c
#include <time.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <stdbool.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>


#define NUMBER_OF_CARS 20

typedef struct Circuit {
    char *step_name;
    int step_total_time;
    int number_of_laps;
    int lap_km;
    int number_of_cars;
    int race_km;
} Circuit;


typedef struct F1_Car {
    int id;
    double lap_time;
    double s1;
    double s2;
    double s3;
    int best_s1;
    int best_s2;
    int best_s3;
    int stand;
    int out;
    int lap;
    int best_lap_time;
    int done;
} F1_Car;

void child(sem_t *sem, F1_Car *car, int *car_names);

void car_crash();

int finished_running();

int msleep(unsigned int tms);
```

3.3 display.c

```c
//
// Created by danny on 5/10/19.
//

#include "display.h"


Circuit circuit;
F1_Car car_array[20];


int compare(const void *left, const void *right) {
    const F1_Car *process_a = (F1_Car *) left;
    const F1_Car *process_b = (F1_Car *) right;

    if (strcmp(circuit.step_name, "RACE")) {
        if (process_a->best_lap_time < process_b->best_lap_time)
            return -1;
        else if (process_a->best_lap_time > process_b->best_lap_time)
            return 1;
        else
            return 0;
    } else {
        if (process_a->lap < process_b->lap)
            return 1;
        else if (process_a->lap > process_b->lap)
            return -1;
        else
            return 0;
    }
}

void print_table() {

    ft_table_t *table = ft_create_table();

    ft_set_border_style(table, FT_DOUBLE2_STYLE);

    ft_set_cell_prop(table, 0, FT_ANY_COLUMN, FT_CPROP_ROW_TYPE,
       FT_ROW_HEADER);
    ft_set_cell_prop(table, 0, FT_ANY_COLUMN, FT_CPROP_CELL_TEXT_STYLE,
        FT_TSTYLE_BOLD);
    ft_set_cell_prop(table, 0, FT_ANY_COLUMN, FT_CPROP_CONT_FG_COLOR,
       FT_COLOR_CYAN);

    ft_write_ln(table, "POSITION", "NAME", "S1", "S2", "S3", "OUT", "
       PIT", "LAP", "LAP TIME", "BEST LAP TIME");

```

```
45      for (int i = 0; i < circuit.number_of_cars; i++) {
46          F1_Car current = car_array[i];
47
48          char sector1_time[10], sector2_time[10], sector3_time[10],
               lap_time[10], best_lap_time[10];
49
50          to_string(current.s1, sector1_time);
51          to_string(current.s2, sector2_time);
52          to_string(current.s3, sector3_time);
53          to_string(current.lap_time, lap_time);
54          to_string(current.best_lap_time, best_lap_time);
55
56          ft_printf_ln(table, "%d|%d|%.6s|%.6s|%.6s|%d|%d|%d|%.7s|%.7s",
               i + 1,
57                       current.id, sector1_time, sector2_time,
                            sector3_time, current.out,
58                       current.stand, current.lap, lap_time,
                            best_lap_time);
59
60          (current.stand)
61          ? ft_set_cell_prop(table, i + 1, FT_ANY_COLUMN,
               FT_CPROP_CONT_FG_COLOR, FT_COLOR_DARK_GRAY)
62          : ft_set_cell_prop(table, i + 1, 6, FT_CPROP_CONT_FG_COLOR,
               FT_COLOR_LIGHT_GRAY);
63      }
64
65      ft_set_cell_prop(table, 1, FT_ANY_COLUMN, FT_CPROP_CONT_FG_COLOR,
           FT_COLOR_LIGHT_GREEN);
66      ft_set_cell_prop(table, 2, FT_ANY_COLUMN, FT_CPROP_CONT_FG_COLOR,
           FT_COLOR_LIGHT_BLUE);
67      ft_set_cell_prop(table, 3, FT_ANY_COLUMN, FT_CPROP_CONT_FG_COLOR,
           FT_COLOR_LIGHT_YELLOW);
68
69      ft_table_t *second_table = ft_create_table();
70      ft_write_ln(second_table, "SECTORS", "NAME", "TIME");
71      ft_set_border_style(second_table, FT_DOUBLE2_STYLE);
72
73      ft_set_cell_prop(second_table, 0, FT_ANY_COLUMN, FT_CPROP_ROW_TYPE,
            FT_ROW_HEADER);
74      ft_set_cell_prop(second_table, 0, FT_ANY_COLUMN,
           FT_CPROP_CELL_TEXT_STYLE, FT_TSTYLE_BOLD);
75      ft_set_cell_prop(second_table, 0, FT_ANY_COLUMN,
           FT_CPROP_CONT_FG_COLOR, FT_COLOR_CYAN);
76
77      char s1_time[10], s2_time[10], s3_time[10], winner[10];
78
79      to_string(car_array[best_sector("S1")].best_s1, s1_time);
80      to_string(car_array[best_sector("S2")].best_s2, s2_time);
81      to_string(car_array[best_sector("S3")].best_s3, s3_time);
82      to_string(car_array[best_lap_time()].best_lap_time, winner);
83
```

```
84      ft_printf_ln(second_table, "%s|%d|%s", "S1", car_array[best_sector(
           "S1")].id, s1_time);
85      ft_printf_ln(second_table, "%s|%d|%s", "S2", car_array[best_sector(
           "S2")].id, s2_time);
86      ft_printf_ln(second_table, "%s|%d|%s", "S3", car_array[best_sector(
           "S3")].id, s3_time);
87      (!strcmp(circuit.step_name, "RACE")) ?
88      ft_printf_ln(second_table, "%s|%d|%.7s", "Winner", car_array[
           best_lap_time()].id, winner) : 0;
89
90      clear();
91      printf("%s", ft_to_string(table));
92      printf("%s", ft_to_string(second_table));
93      ft_destroy_table(table);
94      ft_destroy_table(second_table);
95  }
96
97  int best_sector(char sector[]) {
98      int sector_number = 0, id = 0;
99      for (int i = 0; i < circuit.number_of_cars; i++) {
100
101         if (!strcmp(sector, "S1")) {
102             if (sector_number == 0 || car_array[i].best_s1 <
                   sector_number) {
103                 sector_number = car_array[i].best_s1;
104                 id = i;
105             }
106         } else if (!strcmp(sector, "S2")) {
107             if (sector_number == 0 || car_array[i].best_s2 <
                   sector_number) {
108                 sector_number = car_array[i].best_s2;
109                 id = i;
110             }
111         } else if (!strcmp(sector, "S3")) {
112             if (sector_number == 0 || car_array[i].best_s3 <
                   sector_number) {
113                 sector_number = car_array[i].best_s3;
114                 id = i;
115             }
116         }
117     }
118     return id;
119 }
120
121 int best_lap_time() {
122     int winner = 0, id = 0;
123     for (int i = 0; i < circuit.number_of_cars; i++) {
124         if (winner == 0 || car_array[i].best_lap_time < winner) {
125             winner = car_array[i].best_lap_time;
126             id = i;
127         }
```

```
128     }
129     return id;
130 }
131
132 int finished() {
133     for (int i = 0; i < circuit.number_of_cars; ++i) {
134         if (car_array[i].out) {
135             return 1;
136         }
137     }
138     return 0;
139 }
140
141 void display(sem_t *sem, F1_Car *data) {
142
143     init_window();
144
145     while (1) {
146         sem_wait(sem);
147         memcpy(car_array, data, sizeof(F1_Car) * circuit.number_of_cars
            );
148         sem_post(sem);
149         qsort(car_array, circuit.number_of_cars, sizeof(F1_Car),
            compare);
150         if (finished() || car_array[9].done) {
151             break;
152         }
153         print_table();
154         sleep(1);
155     }
156     sleep(1);
157     save_ranking();
158     terminate_window();
159 }
```

3.4 display.h

```
1  //
2  // Created by danny on 5/10/19.
3  //
4
5  #pragma once
6
7  #include <semaphore.h>
8  #include "child.h"
9  #include "window.h"
10 #include "time.h"
11 #include "files.h"
12 #include "../lib/fort.h"
```

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void display(sem_t *_sem, F1_Car *data);
int compare(const void *left, const void *right);
int best_sector(char sector[]);
int best_lap_time();
```

3.5 files.c

```c
//
// Created by danny on 26/10/19.
//

#include "files.h"


Circuit circuit;
F1_Car car_array[20];

void save_ranking() {

    FILE *file = fopen(circuit.step_name, "w");

    if (file == NULL) perror("fopen failed !"), exit(EXIT_FAILURE);;

    for (int i = 0; i < circuit.number_of_cars; i++) {
        char best_lap_str[10];
        to_string(car_array[i].best_lap_time, best_lap_str);
        fprintf(file, "%d --> %s\n", car_array[i].id, best_lap_str);
    }

    if (fclose(file) != 0) perror("fclose failed !"), exit(EXIT_FAILURE
       );;
}

void
read_files(int qualified_cars[], int race_ranking[], int
   last_cars_of_Q1[], int last_cars_of_Q2[], char file_to_read[],
           int lines_to_read) {

    int path_size = findSize(file_to_read);

    char path_name[path_size];
    getcwd(path_name, path_size);
    char file_path_name[path_size];
    sprintf(file_path_name, "%s/%s", path_name, file_to_read);

```

```
37      FILE *cmd;
38      char result[NUMBER_OF_CARS];
39      char command_file_path_name[path_size];
40      sprintf(command_file_path_name, "egrep -o '^[0-9]{1,2}' '%s'",
           file_path_name);
41
42      cmd = popen(command_file_path_name, "r");
43      if (cmd == NULL) perror("popen failed !"), exit(EXIT_FAILURE);;
44
45      int i = 0, j = 0, k = 0;
46      while (fgets(result, sizeof(result), cmd)) {
47
48          if (i < lines_to_read) {
49              qualified_cars[i] = atoi(result);
50              if (strcmp(file_to_read, "Q3") == 0) {
51                  race_ranking[i] = atoi(result);
52              }
53              i++;
54          } else {
55              if (strcmp(file_to_read, "Q1") == 0) {
56                  last_cars_of_Q1[j] = atoi(result);
57                  j++;
58              } else if (strcmp(file_to_read, "Q2") == 0) {
59                  last_cars_of_Q2[k] = atoi(result);
60                  k++;
61              }
62          }
63      }
64
65      if (pclose(cmd) != 0) perror("pclose failed !"), exit(EXIT_FAILURE)
           ;;
66  }
67
68  int findSize(char file_name[]) {
69
70      FILE *file = fopen(file_name, "r");
71
72      if (file == NULL) {
73          printf("%s : %s", circuit.step_name, "file not Found!\n");
74          return -1;
75      }
76
77      fseek(file, 0L, SEEK_END);
78
79      int res = ftell(file);
80      if (fclose(file) != 0) perror("fclose failed !"), exit(EXIT_FAILURE
           );;
81      return res;
82  }
83
84  void save_eliminated_cars(char file_to_save[], int array[]) {
```

```
85
86      FILE *file = fopen(file_to_save, "w");
87
88      if (file == NULL)
89          perror("fopen failed !"), exit(EXIT_FAILURE);;
90
91      for (int i = 0; i < 5; i++) {
92          fprintf(file, "%d\n", array[i]);
93      }
94
95      if (fclose(file) != 0)
96          perror("fclose failed !"), exit(EXIT_FAILURE);;
97  }
98
99  void read_eliminated_cars(char file_to_read[], int array[]) {
100
101     char results[5];
102
103     FILE *file = fopen(file_to_read, "r");
104
105     if (file == NULL)perror("fopen failed !"), exit(EXIT_FAILURE);;
106
107     int i = 15, j = 10;
108     while (fgets(results, sizeof(results), file)) {
109
110         if (strcmp(file_to_read, "lastQ1") == 0) {
111             array[i] = atoi(results);
112             i++;
113         }
114
115         if (strcmp(file_to_read, "lastQ2") == 0) {
116             array[j] = atoi(results);
117             j++;
118         }
119     }
120
121     if (fclose(file) != 0)
122         perror("fclose failed !"), exit(EXIT_FAILURE);;
123 }
```

3.6 files.h

```
1  #pragma once
2
3
4  #include "child.h"
5  #include "window.h"
6  #include "time.h"
7  #include "../lib/fort.h"
```

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <semaphore.h>


void save_ranking();

int findSize(char file_name[]);

void read_files(int qualified_cars[], int race_ranking[], int last_cars_of_Q1[], int last_cars_of_Q2[], char file[],
                int lines_to_read);

void save_eliminated_cars(char file_to_save[], int array[]);

void read_eliminated_cars(char file_to_read[], int array[]);
```

3.7 main.c

```c
#include "child.h"
#include "display.h"
#include <getopt.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

Circuit circuit;
F1_Car *car;


int car_names[NUMBER_OF_CARS] = {44, 77, 5, 7, 3, 33, 11, 31, 18, 35,
                                 27, 55, 10, 28, 8, 20, 2, 14, 9, 16};

int qualified_cars[15], race_ranking[20], last_cars_of_Q1[15], last_cars_of_Q2[10];

void print_usage() {
    printf("%s", "Usage: ./prog --day [dayName] --step [stepName]\n");
    printf("%s", "Usage: For race you can specify the lap length, by default it's 7km !\n");
    printf("%s", "Usage: ./prog --day [dayName] --step [stepName] --length [number]\n");
    printf("%s", "Use the --help command for more information. \n");
    exit(EXIT_FAILURE);
}

```

```
27 void help() {
28     printf("\n%s\n\n", "These are some commands used to run this
          program.");
29     printf("%s\n", "For P sessions : \t There are run on fridays & P3
          on sat. Use the --day command.");
30     printf("%s\n", "\t\t\t followed by a day name and which step needs
          to be runned for the P sessions.");
31     printf("%s\n\n", "\t\t\t --day fri --step P2 for instance.");
32     printf("%s\n", "For Q sessions : \t There are run on saturdays, use
           the --day command.");
33     printf("%s\n", "\t\t\t followed by a day name and which step needs
          to be runned for the Q sessions.");
34     printf("%s\n\n", "\t\t\t --day sat --step Q3 for instance.");
35     printf("%s\n", "For the RACE session : \t It's run on sundays, use
          the --day command.");
36     printf("%s\n", "\t\t\t followed by a day name. Here you can specify
           the race's lap length.");
37     printf("%s\n", "\t\t\t by default it's 7km, the --length command is
           optional.");
38     printf("%s\n\n", "\t\t\t --day sun --step RACE --length 10 for
          instance. ");
39     exit(EXIT_SUCCESS);
40 }
41
42 int main(int argc, char **argv) {
43
44     signal(SIGINT, return_cursor);
45
46     circuit.lap_km = 7;
47     circuit.race_km = 305;
48
49     int user_km = 0;
50     char day_name[5], step_name[5];
51
52     static struct option long_options[] = {{"day",    required_argument
          , NULL, 'd'},
53                                            {"step",   required_argument
                                                , NULL, 's'},
54                                            {"length", required_argument
                                                , NULL, 'l'},
55                                            {"help",   no_argument, 0,
                                                 'h'},
56                                            {NULL, 0,
                                                                    NULL,
                                                0}};
57
58     char opt;
59     while ((opt = getopt_long(argc, argv, "hd:s:l:", long_options, NULL
          )) != EOF) {
60         switch (opt) {
61             case 'h':
```

```
62                  help();
63                  break;
64              case 'd':
65                  strcpy(day_name, optarg);
66                  break;
67              case 's':
68                  strcpy(step_name, optarg);
69                  break;
70              case 'l':
71                  user_km = atoi(optarg);
72                  break;
73              default:
74                  print_usage();
75          }
76      }
77
78      if (!strcmp(day_name, "fri")) {
79          if (!strcmp(step_name, "P1")) {
80              circuit = (Circuit) {.number_of_cars = 20, .step_name = "P1
                    ", .step_total_time = minutes_to_ms(90)};
81          } else if (!strcmp(step_name, "P2")) {
82              circuit = (Circuit) {.number_of_cars = 20, .step_name = "P2
                    ", .step_total_time = minutes_to_ms(90)};
83          } else {
84              print_usage();
85          }
86      } else if (!strcmp(day_name, "sat")) {
87          if (!strcmp(step_name, "P3")) {
88              circuit = (Circuit) {.number_of_cars = 20, .step_name = "P3
                    ", .step_total_time = minutes_to_ms(60)};
89          } else if (!strcmp(step_name, "Q1")) {
90              circuit = (Circuit) {.number_of_cars = 20, .step_name = "Q1
                    ", .step_total_time = minutes_to_ms(18)};
91          } else if (!strcmp(step_name, "Q2")) {
92              circuit = (Circuit) {.number_of_cars = 15, .step_name = "Q2
                    ", .step_total_time = minutes_to_ms(15)};
93              read_files(qualified_cars, race_ranking, last_cars_of_Q1,
                    last_cars_of_Q2, "Q1", 15);
94          } else if (!strcmp(step_name, "Q3")) {
95              circuit = (Circuit) {.number_of_cars = 10, .step_name = "Q3
                    ", .step_total_time = minutes_to_ms(12)};
96              read_files(qualified_cars, race_ranking, last_cars_of_Q1,
                    last_cars_of_Q2, "Q2", 10);
97          } else {
98              print_usage();
99          }
100     } else if (!strcmp(day_name, "sun")) {
101         if (!strcmp(step_name, "RACE")) {
102             circuit.number_of_cars = 20;
103             circuit.step_name = "RACE";
104             circuit.step_total_time = minutes_to_ms(120);
```

```
105             read_files(qualified_cars, race_ranking, last_cars_of_Q1,
                    last_cars_of_Q2, "Q3", 10);
106             read_eliminated_cars("lastQ2", race_ranking);
107             read_eliminated_cars("lastQ1", race_ranking);
108             read_eliminated_cars("Q3", race_ranking);
109 
110             if (user_km == 0) {
111                 circuit.number_of_laps = circuit.race_km / circuit.
                        lap_km;
112             } else if (user_km > 0) {
113                 circuit.number_of_laps = circuit.race_km / user_km;
114             } else {
115                 print_usage();
116             }
117         } else {
118             print_usage();
119         }
120     } else {
121         print_usage();
122     }
123 
124     !strcmp(circuit.step_name, "Q2") ? save_eliminated_cars("lastQ1",
            last_cars_of_Q1)
125                                      : !strcmp(circuit.step_name, "Q3")
                                             ? save_eliminated_cars("lastQ2
                                            ",
126 
127                                                                        :
                                                                           NULL
                                                                           ;
128 
129     int struct_shm_id = shmget(IPC_PRIVATE, sizeof(F1_Car) * circuit.
            number_of_cars, 0600 | IPC_CREAT);
130     if (struct_shm_id == -1) {
131         perror("shmget failed !");
132         exit(EXIT_FAILURE);
133     }
134 
135     car = shmat(struct_shm_id, NULL, 0);
136     if (car == (void *) (-1)) {
137         perror("shmat failed !");
138         exit(EXIT_FAILURE);
139     }
140 
141     int sem_shm_id = shmget(IPC_PRIVATE, sizeof(sem_t), 0600 |
            IPC_CREAT);
142     if (sem_shm_id == -1) {
```

```
143             perror("shmget failed !");
144             exit(EXIT_FAILURE);
145         }
146         sem_t *sem = shmat(sem_shm_id, NULL, 0);
147         if (sem == (void *) (-1)) {
148             perror("shmat failed !");
149             exit(EXIT_FAILURE);
150         }
151 
152         sem_init(sem, 1, 1);
153 
154         int i;
155         pid_t pid = 0;
156         for (i = 0; i < circuit.number_of_cars; i++) {
157             pid = fork();
158             if (pid == 0)
159                 break;
160         }
161 
162         switch (pid) {
163             case -1:
164                 fprintf(stderr, "fork failed !");
165                 exit(EXIT_FAILURE);
166             case 0:
167                 (!strcmp(circuit.step_name, "Q2") || !strcmp(circuit.
                       step_name, "Q3")) ?
168                 child(sem, &car[i], &qualified_cars[i]) : !strcmp(circuit.
                       step_name, "RACE") ?
169                                                         child(sem, &car[i
                                                            ], &
                                                            race_ranking[i
                                                            ]) :
170                                                         child(sem, &car[i
                                                            ], &car_names[
                                                            i]);
171 
172                 exit(0);
173             default:
174                 display(sem, car);
175                 for (int j = 0; j < circuit.number_of_cars; j++) {
176                     wait(NULL);
177                 }
178         }
179         shmdt(car);
180         shmctl(struct_shm_id, IPC_RMID, NULL);
181 
182         sem_destroy(sem);
183         shmdt(sem);
184         shmctl(sem_shm_id, IPC_RMID, NULL);
185         exit(EXIT_SUCCESS);
186 }
```

3.8 prng.c

```c

#include "prng.h"

void random_seed(unsigned int seed) { srand(seed); }

int sector_range(int min, int max, int crashing_probability) {
  car_crashed(crashing_probability);
  return rand() % (max * 1000 + 1 - min * 1000) + min * 1000;
}

int stand_duration(int min, int max) {
  return rand() % (max * 1000 + 1 - min * 1000) + min * 1000;
}

int stand_probability(int seed) { return rand() % seed == 0; }

// runs in a certain probability, like 1/seed
int car_crashed(unsigned int seed) { return rand() % seed == 0; }
```

3.9 prng.h

```c
//
// Created by danny on 4/10/19.
//

#pragma once

#include <stdlib.h>

void random_seed(unsigned int seed);

int sector_range(int min, int max, int crashing_probability);

int stand_duration(int min, int max);

int stand_probability(int seed);

int car_crashed(unsigned int seed);
```

3.10 time.c

```c
//
// Created by danny on 19/10/19.
//
```

```
4
5  #include "time.h"
6
7  Time time_to_ms(int msec) {
8      Time formated_time;
9      div_t result;
10
11     result = div(msec, 60000);
12     formated_time.min = result.quot;
13     msec = result.rem;
14
15     result = div(msec, 1000);
16     formated_time.sec = result.quot;
17     msec = result.rem;
18
19     formated_time.msec = msec;
20     return formated_time;
21 }
22
23 int minutes_to_ms(int minutes) { return minutes * 60000; }
24
25 void to_string(int msec, char *str) {
26     Time time = time_to_ms(msec);
27     (time.min) ? sprintf(str, "%d':%d\"%d", time.min, time.sec, time.
          msec)
28                : sprintf(str, "%d\":%d", time.sec, time.msec);
29 }
```

3.11 time.h

```
1  #pragma once
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  typedef struct Time {
7      int min;
8      int sec;
9      int msec;
10 } Time;
11
12 Time time_to_ms(int msec);
13
14 int minutes_to_ms(int minutes);
15
16 void to_string(int msec, char *str);
```

3.12 window.c

```c

#include "window.h"

void init_window() { printf("\e[?1049h\e[?7l\e[?25l\e[2J\e[1;52r"); }

void clear() { printf("\e[55H\e[9999C\e[1J\e[1;55r"); }

void terminate_window() { printf("\e[?7h\e[?25h\e[2J\e[;r\e[?1049l"); }

void return_cursor() {
  clear();
  terminate_window();
  exit(EXIT_SUCCESS);
}
```

3.13 window.h

```c
#pragma once

#include <stdio.h>
#include <stdlib.h>

void init_window();

void terminate_window();

void clear();

void return_cursor();
```