IN405 – Système d'exploitation – Projet LV21

Soussan
Jimmy
21510907

Hammad
Amir
21605073

Présentation de notre projet :

Tout d'abord, on entre un fichier.bjack qui correspond au format demandé. Le fichier est lu avec les fonctions de banque.c et toutes les données vont se répartir entre deux structures :

```c
struct init{

    int nbJoueurs; // nombre de joueurs
    int nbDecks;
    int nbMains;

};
typedef struct init INIT;
struct la_banque {


    int nbJetons;
    char typeMise[5];
    int valStop;
    int objJetons;

};
typedef struct la_banque BANQUE;
```

La structure INIT permet de lire la première ligne donc le nombre de joueur etc … et en fonction de ces paramètres, nous allons initialiser dans le main un tableau de BANQUE contenant chacune une ligne de paramètres du fichier (donc pour chaque joueur son nombre de jetons de départ par exemples).

```c
BANQUE * B = malloc(sizeof(BANQUE) * I.nbJoueurs);

for (int i = 0; i<I.nbJoueurs;i++){  // lecture de cha

    B[i] =lire_banque(argv[1],i);

}
```

C'est dans la fonction lire_banque de banque.c qu'on lit le fichier en prenant en compte que le type de mise ne se lit pas comme un entier mais comme une chaîne de caractères. Ensuite on va initialiser un tableau de thread lui aussi d'une taille égale au nombre de joueurs, car chaque thread s'occupe d'un joueur particulier.

Dans la structure joueur ci-dessous, il y a toutes les données de base que l'on va écrire dans les fichiers mais aussi des données pratiques comme lastMise qui permet de connaître la

dernière mise et ainsi adapter la suivante en fonction du comportement du joueur, mais aussi d'autres permettant de savoir quand s'arrêter dans le fichier, ou encore quel fichier ouvrir et à quelle ligne écrire.

```c
struct joueur {

    char cartes[20];
    int total_Joueur;
    char banque[20];
    int total_Banque;
    int mise;
    int gain;
    int nbJetons;

    int lastMise; // permet de prendre en compte la derniere mise pour les 200- et 10+
    int lastGain; // même utilité

    int i; // pour ouvrir joueur(i) (joueur1, joueur2 ...)
    int t; // permet de connaitre le tour pour savoir écrire à quelle ligne du fichier joueur on écrit
    int stop; // si le joueur a atteint 0 ou son objectif
};
typedef struct joueur JOUEUR;
```

Pour simplifier l'écriture et les fonctions, on utilise une dernière structure contenant les autres, le deck utilisé et d'autres paramètres : c'est la structure game.

```c
struct game{ // pour utiliser les phtreads

    JOUEUR * J;
    INIT I;
    BANQUE *B;


    int tour;
    int joueur;

    deck_t * Deck;


};typedef struct game GAME;
```

On lance donc une fonction void * à qui on envoie un paramètre GAME. Cette fonction va traiter un joueur en particulier, écrire dans son fichier mais aussi afficher dans le terminal ses cartes, sa mise, ses gains. La banque est traitée dans le main car elle n'est pas dans un thread. Ce qui donne un résultat comme celui-ci lors de l'exécution :

```
Tour 1 :

- La banque a pioché 7Q, total : 17

- Le joueur 3 a pioché 642Q, total : 22
- Sa mise était de 10, son gain est de 0 il est à : 1490

- Le joueur 2 a pioché QQ, total : 20
- Sa mise était de 200, son gain est de 400 il est à : 5220

- Le joueur 1 a pioché 8346, total : 21
- Sa mise était de 50, son gain est de 100 il est à : 250
```

Nous avons essayé de traiter aussi tous les problèmes, le terminal affiche un message d'erreur lorsqu'on met un fichier qui n'est pas aux normes.

Difficultés rencontrées :

Nous avons rencontré un certain nombre de difficultés dans notre projet.
Tout d'abord, avec les fonctions write et read, nous avions appris à lire des entiers avec les autres fonctions de lecture de fichier mais pas celles-ci. Après quelques recherches, la solution est apparue pour convertir des entiers en chaîne de caractère, mais pour ce qui est de l'inverse nous avons dû écrire la fonction nous même (convString).

De même, pour accéder à une certaine ligne de fichier, nous avons fait nos propres fonctions goLine et Nextline (l'une pour la lecture, l'autre pour l'écriture)

Les threads nous ont causé plusieurs problème. Tout d'abord, le programme se lançait dans une boucle infinie lorsqu'on utilisait trop de joueurs ou trop de tours. Ensuite, tout se lançait sans problème de compilation ou d'exécution, mais au lieu d'écrire dans 3 joueurs différents par tour par exemple, le programme écrivait 3 fois dans le même joueur. Il était impossible de faire jouer les autres.
Nous avons donc essayé de comprendre le fonctionnement des mutex ; et après quelques test on a pu trouver une solution assez simple.

```
pthread_mutex_lock (&mutex); // on verouille le mutex pour le numéro du joueur
pthread_cond_signal (&condition);
i = G->joueur;
G->joueur--;
pthread_mutex_unlock (&mutex);
```

G->joueur correspond au numéro du joueur, et lorsqu'on ne demandait pas au thread de faire un lock, ce nombre pouvait être pris à n'importe quel moment par n'importe quel joueur. On a donc décidé de bloquer au moment de prendre le numéro du joueur, puis de passer au suivant (en décroissant).

Après avoir le rendu le projet au chargé de TD, on a reçu un retour qui nous explique que notre programme ne fonctionne pas et affiche : segmentation fault.
On ne s'en était pas rendu compte, car l'erreur ne s'affichait pas.
Afin de comprendre d'où venait l'erreur de segmentation, nous avons utilisé Gdb et augmenté considérablement le nombre de tours (446). Nous n'avons pas réussi à résoudre le problème par manque de temps après sa découverte, mais nous avons supposé qu'il était lié à une mauvaise utilisation des mutex.