Projet Programmation Système

COUVY Julien, LEYX Sebastien

4 janvier 2017

1 Introduction

L'objectif de ce projet était de mettre en réseau un jeu de morpion aveugle avec une architecture client-serveur similaire à celle vue dans le TP sur le serveur de chat. Nous avons donc implementé une architecture TCP écrite en Python avec la méthode select().

2 Sauvegarde et chargement des cartes

2.1 Choix

2.2 Résultats

3 Gestion des temporisateurs

3.1 Choix

Nous avons dabord créé une structure 'Event' représentant une nouvelle occurrence d'un timer (bombe ou mine). La structure comprend les champs suivants :

```
typedef struct event_s {
    unsigned long daytime; // Heure au moment où la structure est allouée
    struct itimerval delay; // Structure de temps spécifiant le délai avant l'action
    void* event_param; // Sauvegarde du paramètre passé par SDL
    struct event_s *prev; // Chainage double (élément précédent)
    struct event_s *next; // Chainage double (élément suivant)
} Event;
```

Les évènements sont ajoutés dans une liste double chaînée dès leur création (placement d'une bombe ou d'une mine par l'utilisateur) puis sont ensuites triés dans l'ordre de déclenchement. Pour se faire, on vérifie pour chaque élément de la liste si l'horaire stocké dans la structure de l'évènement ajouté au délais avant l'explosion est inférieur à ce même calcul pour les autres Events. Il est important de noter que le délai doit être converti en millisecondes si l'on se réfère au prototype de la fonction timer_set.

Fonctions associées :

```
unsigned long delay_of_event(Event *e);
bool compare_delay(Event* a, Event* b);
// Applique un tri bulles sur une liste dont on donne l'adresse du premier élément.
void sort_events(Event** head);
// Ajoute l'élément pointé par new_event dans une liste DC
void add_event(Event** head, Event** new_event);
```

Pour gérer la réception des signaux nous créons un thread démon chargé de boucler indéfiniment en attendant les SIGALRM envoyés par le système lorsque le délai de chacun des Events atteint 0 (grâce à la fonction **setitimer** appelé dans **timer_set** dont le comportement est analogue à alarm() en plus de la gestion des structures de temps **itimerval**).

Fonctions associées :

```
void* daemon_handler(void* argp);
void signal_handler(int signo);
```

À la réception d'un SIGALRM, le thread démon se charge de l'appel à la fonction **sdl_push_event** tout en lui passant en argument le paramètre de l'évènement déclenché. De plus il doit se charger, d'une part de libérer l'espace mémoire dans la liste de l'évènement terminé et de mettre à jour la tête de file, d'autre part de la gestion des évènements à venir. On distingue deux possiblités où :

— Un ou plusieurs évènements seraient très rapprochés dans le temps de celui venant de se déclencher. On considère le temps minimum espacant deux évènements de façon arbitraire. Le démon doit alors répéter les étapes précédentes en négligeant la différence de temps.

```
do {
// gestion de l'Event
} while  ((curr_timer + 100000UL > next_timer) && head != NULL);
// 100000UL = 100ms, le délai minimum entre deux évènements.
```

— Les évènements suivants se déclenche avec plus de 100ms d'intervalle. Dans ce cas, il est nécessaire de mettre en place un nouveau signal d'alarme avec le délai mis à jour correspondant au prochain évènement. Cela se traduit par un nouvel appel à la fonction **setitimer** avec pour paramètre une structure **itimerval** dont le délai correspond à la différence entre le délai stocké dans la structure du prochain évènement et l'horaire au moment de l'appel.

Pour protéger les accès espaces de données partagées nous avons mis en place un mutex que l'on initialise dès l'appel à la fonction **timer_init**. Les protections sont ajoutées dès qu'une manipulation de la liste DC à lieu notamment :

À la création d'un nouveau timer

```
void timer_set (Uint32 delay, void *param)
{
//...
    pthread_mutex_lock(&mutex);
    add_event(&head, &e);
    sort_events(&head);
//...
    pthread_mutex_unlock(&mutex);
}
```

Au sein du thread démon lors de la mise à jour de la liste

```
void* daemon_handler(void* argp)
{
//...
 while (1)
    {
        //...
        do {
            pthread_mutex_lock(&mutex);
        // SDL + mise à jour liste DC
            pthread_mutex_unlock(&mutex);
        } while ((curr_timer + 100000UL > next_timer) && head != NULL);
}
```

3.2 Résultats

— Gestion des structures de données
  - ☑ Liste DC fonctionnelle
  - ☑ Structure évènement
  - ☑ Tri de la liste
  - ☒ Rapport sans faute de valgrind (SDL ?)

— Implémentation simple
  - ☑ Placement d'une bombe (timing, sprite : ok)
  - ☑ Placement d'une mine (timing, armement, sprite : ok)
  - ☑ Explosions respectant l'ordre chronologique

— Implémentation complète
  - ☑ Placement de plusieurs bombes
  - ☑ Placement de plusieurs mines
  - ☑ Explosions respectant l'ordre chronologique
  - ☒ Explosions simultannées (peut glitcher par moments)