Rapport du projet de programmation système

Aza, Willem, Tremor Sullyvan, Delar Emmanoe

10 décembre 2017

Table des matières

1 Introduction

En groupe de 3 étudiants, nous avons réalisé un projet qui faisait appel au cours de programation système que nous avons suivi tout au long du semestre. Le but de ce projet était de nous introduire au système d'exploitation. Tout au long du projet nous avons du manipuler les fonctions primitives (appels systèmes) fournies par le système d'exploitation. Ces fonctions nous ont permis de développer quelques mécanismes de base destinés à servir dans le développement d'un petit jeu de plateforme 2D.
Le projet s'est déroulé en plusieurs parties distinctes. La première partie portait sur la mise en place d'un mécanisme de gestion des fichiers, nous avons eu à développer les fonctions de sauvegarde (map_save) et de chargement (map_load) des cartes du jeu. La deuxième partie portait sur la gestion des temporisateurs (signaux) permettant de planifier les différents événements du jeu.

2 Gestion des fichiers

2.1 map_save

Cette fonction est basée sur les appels systèmes et consiste à mémoriser les informations du jeu à un instant t, cette mémorisation est faite pendant la partie, quand le joueur appuie sur la touche 's' de son clavier.
Avant de lire ou d'écrire dans un fichier, on notifie son accès par la fonction **open()**. Si cette fonction retourne sans erreur, on obtient alors un pointeur sur un flot de donnée (stream) et non pas le fichier maps/saved.map ouvert sur le disque. Ce flot sera utilisé lors de l'écriture ou la lecture.
Une fois que nous avons accès au fichier en mode écriture, nous enregistrons les données. A l'aide des primitives fournis, notre algorithme se charge de récuperer les détails de la carte. Ces données sont ecrit avec l'appel systeme **write()** sur le flot. On procède à l'ajout de données dans un ordre bien précis. Cette ordre est important car nous aurons à lire ces données plus tard. Parmis les données enregistrées, on retouve des éléments essentiels tels que la taille et la largeur de la fenêtre, le nombre d'objets du jeu... A partir de ces données on mémorise plus facilement la position de chacun des éléments existants, leur nom et toutes autres informations concernant cet objet, afin de les restituer convenablement à l'aide de boucles.
Après les traitements, on rompt la liaison entre le fichier et le flot de donnée grâce à l'appel systeme **close()**

2.2 map-load

Aprés avoir réalisé l'ecriture sur le descripteur de fichier avec l'appel systeme **write()**, nous avons du utiliser la fonction **read()** qui nous permet cette fois ci de lire un nombre d'octets depuis ce descripteur de ficher. Cette fonction nous a permis d'implémenter la partie chargement du jeu.
Cette fonction consiste à charger une partie qui aurait été sauvergardée au

préable par le joueur. Le chargement a pour but de récupérer toutes les informations stockées dans le fichier par défaut : maps/saved.map ou tout autre fichier à condition que celui-ci respecte le format de sauvegarde des différents informations.
Avant de lire les donnés, on notifie l'accès au fichier, toujours avec la fonction **open()**. A la différence de la fonction map_load, nous sollicitons un accés en lecture uniquement en ajoutant l'argument **O_RDONLY**. Nous verifions que le retour d'appel des fonctions s'effectue sans erreur grâce à notre fonction **verification()**.

On procède à la lecture de données selon l'ordre d'écriture fait dans le fichier par map_save. Parmis les données, on retrouve des éléments essentiels tels que les dimensions de la fenêtre, du nombre d'objets possibles de cette partie ... A partir de ce relevé d'informations nous ferons le stockage dans des variables prévues à cet effet. Une fois ces variables instanciées nous procédons à la création d'une partie grâce aux fonctions prédéfinies telles que **map allocate()**, **map_set**...
Pour charger une partie sauvegardée, il faudra lancer le jeu avec les arguments suivant : « ./game -l maps/saved.map ».

2.3 Utilitaire de manipulation de carte

Le programme **maputil()** est capable de modifier un fichier contenant une carte. Il permet également d'avoir des informations sur ce fichier et aussi de modifier les données enregistrées. Nous pouvons modifier les dimensions en largeur et en hauteur de la carte par exemple. L'une des difficultées était de faire attention à ne pas perdre les objets déja disposés sur la carte après un changement de dimensions.
Nous avons alors choisi de sauvegarder avec notre fonction map_save uniquement les objets ajoutés par l'utilisateur sur la carte, pour ensuite les restituer à leur place avec la fontion map_load. Ces objets sont récupéré grâce à la fonction **map_get** qui retourne une valeur différente de « MAP_OBJECT_NONE ».Si un de ces objets est de coordonnée supérieur à celui de la nouvelle carte, il sera alors ignoré et ensuite supprimé.
Pour réaliser cet exécutable nous faisons des appels systèmes plutôt qu'utiliser les fonctions existantes en faisant attention que le retour d'appel s'effectue correctement grâce à notre fonction **verification()**. En fonction du nombre d'arguments en ligne de commande nous pouvons déterminer s'il sagit d'une modification ou d'un relevé d'informations.

Les relevés d'informations se font grâce à **get()** qui lit notre fichier et qui instanciera une variable, puis la renverra afin de restituer l'information voulue par l'utilisateur selon l'option (- -getwidth, ...) choisie. **get()** fait appel à **lseek()** pour se deplacer à la bonne position, nous utilisons un pointeur pour l'instanciation de la valeur voulue, l'instanciation est realisée grâce à **verif_ES()** qui dans le cas d'un relevé d'informations fera appel à **read()**. Une fois que l'opération voulue est faite, **get()** renvoie la position courante (utile selon l'opération que l'on souhaite faire).

Les modifications des données ont presque le même procédé sauf que **ve-**

rif_ES() fera appel à **write()**. Pour l'option - -pruneobjects qui efface des données inutilisées, on effectue une recherche d'objets inutilisés, on classifie les objets selon leur existance ou non dans la partie, ce classement est réalisé par **exchange()**, les objets inutilisés finissent en fin de liste par échange, on utilise **ftruncate()** pour nettoyer la fin du fichier et une mise à jour est faite.

3 Gestion des temporisateurs

Dans la version du jeu que nous proposons, nous avons la possibilité d'armer plusieurs temporisateurs afin d'effectuer des transitions entre les animations (explosion de bombes, mines, personnage qui clignote lorsqu'il se cogne). Pour cela, nous avons utilisé la fonction unix « setitimer »qui permet de generer un signal SIGALRM, après l'expiration d'une minuterie, au processus courant.
Toutefois, cette fonction a ces limites, elle ne permet d'armer qu'un seul temporisateur à la fois. La figure ci-dessous illustre le nouveau comportement attendu que nous avons implémenté.

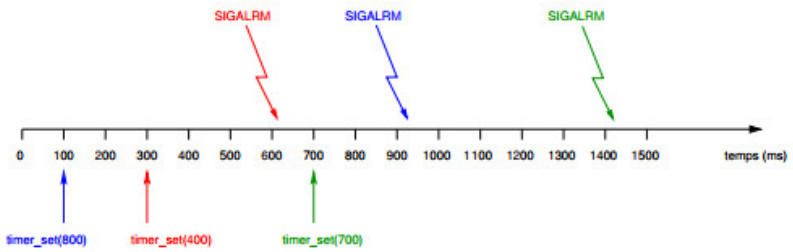

FIGURE 1 – Possibilité d'armer plusieurs temporisateurs

3.1 implémentation du gestionnaire de temporisateurs

Pour pouvoir armer plusieurs temporisateurs, nous avons implementer une liste chainée qui stocke tous les appellent à la fonction setitimer.
Dans cette structure, nous mémorisons les informations comme le délai de déclenchement de l'évènement, la date d'ajout de cet évènement, et une variable qui stocke la différence entre la date d'ajout de l'evenement et la date de celui qui le suit chronologiquement dans le temps.
Nos primitives permettent de gerer ces informations et de trier notre collection d'élements dans l'ordre chronologique croissant. Ainsi, le delai de temporisation de l'élement se trouvant à la tête de la liste est le plus petit et celui du dernier élement est le plus grand.

3.2 Installation du traitant, timer-init

Une fois que nous pouvons generer plusieurs signaux, nous devons maitenant les traiter.
L'appelle systeme « sigaction »examine les signaux recu par le processus courant et nous permet alors de changer le comportement prevu grace a une fonction traitante.
Nous avons implementer la fonction « timer-init »qui crée des threads. Ce thread s'execute en parallele du processus courant et a pour tâche d'attendre indefiniment un évènement, représenté par le signal SIGALRM, grace à une boucle while. La fonction **sigprocmask** nous a permit au préalable de masquer les signaux SIGALRM au processus principal. Lorsque arrive le signal, il est alors traité par le thread et ses informations sont corectement rangés dans notre structure de donnée.
Comme cette structure est une ressource partagée par les différentes fonctions de notre programme, ces données sont manipulées selon le principe de l'exclusion mutuelle. Nous utilisons au début et à la fin du code critique l'objet mutex « pthread-mutex-lock »et « pthread-mutex-unlock »qui permet de protéger les données de modifications concurrentes.

4 Difficultés rencontrées.

4.1 tempo.c

Tous ne s'est pas passé comme prevu dans la partie gestion des temporisateurs. Lors de l'ajout d'une première bombe, il nous était impossible d'en avoir une autre qui explose aprés cette première. C'est à dire que si on pose autant de bombe qu'on veut, entre le temps du dépot de la première bombe et son explosion, alors elles exploseront toutes en même temps que la premiere.

FIGURE 2 – Représentation du probème

Après plusieurs jours de recherches, le problème à été réglé. Il sagissait d'un erreur de notre part. Les nouveaux élements avec un délai de déclenchement

supérieur étaient ajoutés au début de notre liste chainée au lieu d'être mis à la fin.