Mario Platformer

Camille AUSSIGNAC, Manon PHILIPPOT, Louis LEDUC, Université de Bordeaux

Première Partie : Sauvegarde et chargement

Sauvegarde

Nous sauvegardons les données des objets et les informations de la partie comme ceci :

DATA	**TYPE**
LARGEUR	INT
HAUTEUR	UNSIGNED
NB_OBJETS	UNSIGNED
<OBJETS>	UNSIGNED
NOMFICH_LEN	INT
NOMFICHIER	INT
NB_FRAME	INT
SOLIDITY	INT (0, 1 ou 2)
DESTRUCTIBLE	INT (0 ou 4)
COLLECTIBLE	INT (0 ou 8)
GENERATOR	INT (0 ou 16)

Nous les sauvegardons dans un fichier map_block.save dans le dossier maps. Nous avons choisis cette implémentation car il est plus facile de séparer ces informations en deux pour le reste du code, et cela facilitera l'ajout de nouveaux type d'objet sans prendre le risque de modifier objets.txt qui ne doit surtout pas être modifié. Les nouveaux types d'objets seront ajouté dans le fichier map_block.save. Chaque block sera codé par un int, sachant que les objets 0 à 9 seront les 10 objets de bases donné dans les sources. Les nouveaux objets seront nommé pareillement : 10, 11, 12, etc.

1 Fonction maputil

1.1 fonctions de base

La fonction maputil doit pouvoir effectuer plusieurs opérations de lecture/ecriture directement sur le fichier de sauvegarde, pour cela, nous devons utiliser les appels système de type open, write, read etc.

Voici par exemple la fonction permettant d'aller chercher la hauteur de la carte et de de la retourner

Listing 1: Maputil.

```
1  unsigned get_height(const char* filename){
2      int fd = open(filename, O_RDONLY);
3      unsigned height = 0;
4      int r = read(fd, &height, sizeof(unsigned));
5      return height;
6  }
```

1.2 agrandissement/rétrécissement

Apres ces fonctions de base, nous avons du coder une fonction un peu plus complexe qui a pour but d'agrandir ou de rétrécir la carte. Pour cela, re-penchons nous sur l'implémentions de notre fichier save : il dispose d'un premier unsigned qui est la hauteur, puis la longueur puis le nombre d'objets. Nous avons décidé de garder toutes ces informations pour réécrire totalement le fichier saved.map, en modifiant bien sur la hauteur ou la longueur suivant la demande. Ensuite nous devons rajouter ces nouvelles dimensions dans notre "grille" contenant les objets de notre carte, pour cela, nous rajoutons des cases vides a droite ou en haut de la carte.

Listing 2: modification de la hauteur.

```
1  void set_height(const char* filename, unsigned new_height){
2      int fd = open(filename, O_RDWR, 777);
3      unsigned original_height=0;
4      unsigned original_width=0;
5      unsigned nb_objects = 0;
6      read(fd, &original_height, sizeof(unsigned));
7      read(fd, &original_width, sizeof(unsigned));
8      read(fd, &nb_objects,sizeof(unsigned));
9      int descr[2];
10     pipe(descr);
11     //agrandissement
12     for(int j = 0; j < (int) (new_height - original_height); j++){
13         //ajouter case vide
14         int buff = -1;
```

```
15          for (int i =0; i < original_width; i++)
16          {
17              write(descr[1], &buff, sizeof(buff));
18          }
19      }
20      int kept_height;
21      if (original_height > new_height) {
22          kept_height = new_height;
23      } else {
24          kept_height = original_height;
25      }
26      lseek(fd, (original_height - kept_height) * original_width * ↩
            sizeof(int), SEEK_CUR);
27      for (int i = 0; i < kept_height; i++){
28          //taille d'une ligne du tableau buff;
29          //read(fd, buff, sizeof(buff)) ;
30          //write(decr[1], buff, sizeof(buff));
31          int buffer;
32          for (int i =0; i < original_width; i++)
33          {
34              read(fd, &buffer, sizeof(int));
35              write(descr[1], &buffer, sizeof(buffer));
36          }
37      }
38      lseek(fd,0, SEEK_SET);
39      write(fd,&new_height, sizeof(new_height));
40      write(fd,&original_width,sizeof(original_width));
41      write(fd,&nb_objects, sizeof(nb_objects));
42
43      int buffer2;
44      for (int i = 0; i < (int) (new_height * original_width); i++)
45      {
46          read(descr[0], &buffer2, sizeof(int));
47          write(fd, &buffer2, sizeof(int));
48      }
49      close(descr[0]);
50      close(descr[1]);
51      close(fd);
52  }
```

Le procédé est le suivant : nous mettons une ligne ou une colonne (suivant si l'ont veut modifier la hauteur ou la longueur) de la grille dans un pipe et nous rajoutons ou enlevons autant de cases que demandé. Nous mettons ensuite le tout dans le fichier save en prenant garde de tout écraser avant.
Plusieurs bugs sont survenus prenant beaucoup de temps a trouver car il s'agissait principalement de bugs sur un read qui mettait la valeur qu'il lisait dans un mauvais type, et ainsi quand on écrivait cette valeur, nous nous retrouvions avec une carte complètement faussée.

1.3 remplacement des objets d'une carte

C'est principalement en voyant le travail a fournir dans cette partie que nous avons opté pour la sauvegarde des objets appartenant a une map dans un autre fichier. Cela nous a facilité la tache car nous pouvions accéder directement à tout les objets sans avoir a manipuler les autres données contenues dans le fichier de sauvegarde principal. Pour remplacer les objets, on a d'abord stocké dans une variable la chaîne de caractère qui va caractériser l'objet (ex: flower.png) puis on va lire le fichier map_block et essayer de trouver une correspondance. Si il y en a une, on modifie les champs appartenant a cet objet en ajoutant ceux passé en paramètre avec un write. Sinon on crée un nouvel objet a la fin du fichier contenant les caractéristiques souhaitées.

1.4 suppression des objets inutilisés

Pour cette partie aussi, l'implémentation des objets dans un autre fichier va nous être utile. Il faut savoir qu'a un fichier save.map est associé un fichier map_block.save crée a la sauvegarde, celui ci est une liste de tout les objets que nous connaissons. Pour le délester, nous allons parcourir toute la grille et insérer dans un tableau un entier qui va faire correspondre un objet de la liste avec un objet de notre tableau puis nous supprimons les lignes des objets n'étant pas dans notre tableau. Pour finir, nous associons a chaque objet un nouveau entier qui correspondra a notre nouveau map_block. Et nous reparcourons notre grille en écrivant la nouvelle valeur de l'objet. Par exemple si nous n'avons que des fleurs et que le code des fleurs est 5, ce code devient 0 car nous avons supprimé tout les autres objets dans le map_block.

Deuxième partie : Gestion des temporisateurs

2 Démon récepteur de signaux

Listing 3: timer_init.

```
1  int timer_init (void)
2  {
3    sigset_t mask;
4    sigemptyset(&mask);
5    sigaddset(&mask, SIGALRM);
6    pthread_sigmask(SIG_BLOCK, &mask, NULL);
7
8    pthread_t thread;
9
10   if(pthread_create(&thread, NULL, daemon, NULL) == -1)
11   {
12     perror("pthread_create");
13     return EXIT_FAILURE;
14   }
15
16   return 1;
17 }
```

La fonction timer_init créé un thread exécutant la fonction daemon à l'aide de la fonction pthread_create. Le bon déroulement de la création du thread est vérifié par un test retournant EXIT_FAILURE en cas de problème. Afin que ce thread soit l'unique thread du processus qui puisse recevoir les signaux SIGALRM, un masque de blocage de signaux vide auquel a été ajouté uniquement le signal SIGALRM a été créé et appliqué à l'ensemble des autres threads créés par timer_init grâce à la fonction pthread_sigmask.

Listing 4: Thread daemon.

```
1  void *daemon(void *arg)
2  {
3    sigset_t mask;
4    sigfillset(&mask);
5    sigdelset(&mask, SIGALRM);
6
7    struct sigaction sa;
8    sa.sa_handler = handler;
9    sa.sa_flags = 0;
10   sigemptyset(&sa.sa_mask);
11
12   sigaction(SIGALRM, &sa, NULL);
13
14   while(1)
```

```
15    {
16      sigsuspend(&mask);
17    }
18  }
```

La fonction daemon déclare une structure sigaction pour la mise en place d'un gestionnaire du signal SIGALARM. Dorénavant, à chaque signal SIGALRM reçu, la fonction handler est appelée. Elle effectue ensuite une boucle infinie appelant la fonction sigsuspend ayant pour argument un masque bloquant tous les signaux sauf SIGALRM. Celle-ci remplace temporairement le masque de signaux du processus appelant avec le masque fourni et suspend le processus jusqu'à la livraison d'un signal SIGALRM.

Listing 5: Test de l'implémentation.

```
1   struct itimerval timer;
2   // configure le timer pour expirer apres 250msec...
3   timer.it_value.tv_sec = 0;
4   timer.it_value.tv_usec = 250000;
5   // ... et toutes les 250 msec
6   timer.it_interval.tv_sec = 0;
7   timer.it_interval.tv_usec = 250000;
8   // enclenche le timer
9   setitimer(ITIMER_REAL, &timer, NULL);
```

Pour tester cette implémentation, des signaux SIGALRM sont générés à intervalle de temps régulier dès le lancement du thread par l'intermédiaire d'un temporisateur.

Listing 6: handler.

```
1  void handler(int sig){
2    printf("L'identit  du thread courant est : %d\n", pthread_self()↩
       );
3  }
```

Chaque signal SIGALRM récéptionné par la fonction daemon entraîne l'exécution de handler qui imprime l'identité du thread courant.

3 Implémentation simple

_set.

```
1  void *param_event;
2  void timer_set (Uint32 delay, void *param)
3  {
4    // sauvegarde de param
5    param_event = param;
```

```
6
7   struct itimerval timer;
8   // configure le timer pour expirer apr s delay msec...
9   timer.it_value.tv_sec = 0;
10  timer.it_value.tv_usec = delay;
11  // ... et seulement 1 fois
12  timer.it_interval.tv_sec = 0;
13  timer.it_interval.tv_usec = 0;
14  // enclenche le timer
15  setitimer(ITIMER_REAL, &timer, NULL);
16 }
```

L'implémentation de timer_set pour armer un seul temporisateur est relativement simple. Il faut d'abord sauvegarder la valeur param dans une variable déclarée à l'extérieur de la fonction, afin qu'elle soit accessible par toutes les fonctions, notamment handler. Ensuite il faut déclarer un temporisateur dont on configure le délais avec l'argument delay et l'intervalle à 0. Ensuite on enclenche le temporisateur avec la fonction setitimer, qui prend en paramètre le temporisateur configuré.

```
1  void handler(int sig){
2    printf("sdl_push_event(%p) appel e au temps %ld\n", param_event,↩
         get_time ());
3  }
```

Une fois SIGALRM récupéré, le thread démon affiche un message avec handler.

4 Implémentation complète

```
1  typedef struct linked_list
2  {
3    struct itimerval timer;
4    void *param;
5    unsigned long time_signal;
6    struct linked_list *next;
7  } linked_list;
8
9  void insert(linked_list **ll, linked_list **event)
10 {
11   linked_list *tmp = NULL;
12   linked_list *cll = *ll;
13   while(cll && cll->time_signal < (*event)->time_signal)
14   {
15     tmp = cll;
16     cll = cll->next;
17   }
```

```
18    (*event)->next = cll;
19    if(tmp)
20      tmp->next = (*event);
21    else
22      *ll = (*event);
23  }
24
25  void pop(linked_list **ll)
26  {
27    linked_list *tmp = (*ll)->next;
28    free(*ll);
29    *ll = tmp;
30  }
31
32  linked_list *first_event = NULL;
```

Tout d'abord, une structure de liste simple chaînée est implémentée. Elle contient un temporisateur, le param, un unsigned long correspondant à la date d'émission du signal de l'évènement en question, et une référence vers l'objet suivant de la liste. Deux fonctions sont ensuite implémentées afin de de manipuler cette structure. La première, insert, insère l'élément passé en second argument dans la liste passé en premier argument, tout en le triant dans l'ordre chronologique de délivrance du signal. La seconde, pop, supprime le premier élément de la liste : le deuxième élément devient ainsi le premier élément. Enfin, une liste vide first_event est créée. Elle contiendra l'ensemble des évènements à déclencher pendant le jeu.

_set.

```
1   void timer_set (Uint32 delay, void *param)
2   {
3     linked_list *event = malloc(sizeof(struct linked_list));
4
5     // Protection des acc s aux structures de donn es partag es
6     pthread_mutex_lock(&mutex);
7
8     event->timer.it_value.tv_sec = delay/1000;
9     event->timer.it_value.tv_usec = (delay%1000)*1000;
10    event->timer.it_interval.tv_sec = 0;
11    event->timer.it_interval.tv_usec = 0;
12
13    event->param = param;
14
15    event->time_signal = get_time() + delay*1000;
16
17    event->next = NULL;
18
19    insert(&first_event, &event);
20
```

```
21    if(event == first_event)
22    {
23      if(setitimer(ITIMER_REAL, &(first_event->timer), NULL) == -1)
24      {
25        perror("setitimer timer_set");
26        exit(1);
27      }
28    }
29    // Fin de la protection des acc s aux structures de donn es ↩
          partag es
30    pthread_mutex_unlock(&mutex);
31  }
```

La fonction timer_set a été totalement réimplémentée. Elle créée d'abord un élément de liste chaînée correspondant à un évènement du jeu, initialise l'ensemble de ses données, l'insère chronologiquement dans la liste chaînée déclarée au préalable, et déclenche le temporisateur correspondant au premier évènement de la liste chaînée. Un test a été implémenté pour la fonction setitimer, et quitte le programme en cas de problème. Comme la liste chaînée peut être utilisée de manière concurrente par le programme principal (dans timer_set) et par le thread récepteur de signaux (dans le traitant), les accès aux structures de données partagées sont protégés par un mutex.

```
1   void handler(int sig)
2   {
3     // Protection des acc s aux structures de donn es partag es
4     pthread_mutex_lock(&mutex);
5
6     /* Si un   vnement   suit le premier   vnement   de la liste, on ↩
          d clenche
7        le premier   vnement   et on arme un temporisateur pour le ↩
             deuxi me */
8     if(first_event->next != NULL)
9     {
10      unsigned long current = first_event->time_signal;
11      unsigned long after = first_event->next->time_signal;
12      unsigned long diff = after - current;
13      sdl_push_event (first_event->param);
14      pop(&first_event);
15      if(first_event != NULL)
16      {
17        first_event->timer.it_value.tv_sec = diff/1000000;
18        first_event->timer.it_value.tv_usec = (diff%1000000);
19        first_event->timer.it_interval.tv_sec = 0;
20        first_event->timer.it_interval.tv_usec = 0;
21        if(setitimer(ITIMER_REAL, &(first_event->timer), NULL) == -1)
22        {
23          perror("setitimer handler");
```

```
24          exit(1);
25        }
26      }
27      else
28      {
29        return;
30      }
31    }
32    else
33    {
34        sdl_push_event (first_event->param);
35        pop(&first_event);
36    }
37
38    // Fin de la protection des acc s aux structures de donn es ↩
          partag es
39    pthread_mutex_unlock(&mutex);
40  }
```

La fonction handler déclenche les évènements. Si le premier élément de la liste contient un élément next, alors l'évènement du premier élément de la liste est déclenché et le temporisateur de next est à nouveau armé. Sinon, l'évènement du premier élément de la liste est déclenché. Comme la liste chaînée peut être utilisée de manière concurrente par le programme principal (dans timer_set) et par le thread récepteur de signaux (dans le traitant), les accès aux structures de données partagées sont protégés par un mutex.

5 Mise en service dans le jeu

En remplaçant le printf par un appel à sdl_push_event et en retournant 1 à la fin de timer_init, tout fonctionne correctement. Il est possible de déposer des bombes et poser des mines.