Architecture et système d'exploitation — Bibliothèque de *threads* partie 1

Pascal Garcia

23 mars 2022

1 Création de *threads*, ordonnancement et structures de données

Nous allons présenter les fonctions permettant de créer des *threads*, de les rendre prêts à s'exécuter et les fonctions permettant de gérer l'ordonnancement entre les *threads*. Nous présenterons ensuite les structures de données associées à la gestion des *threads*.

1.1 Fonctions de création et de lancement

- La fonction `int create_thread(long initial_address, int priority, char* name, int nb_args, ...);` (fichier `thread.h`) permet de créer un *thread*. On a :
  - ⋆ `initial_address` est l'adresse de la fonction qui sera le code du *thread* ;
  - ⋆ `priority` est la priorité du *thread*, plus cette valeur est grande et plus la priorité est importante ;
  - ⋆ `name` est le nom du *thread* ;
  - ⋆ `nb_args` est le nombre de paramètres de la fonction représentant le code du *thread*. On suppose que tous les arguments sont de type `int` ;
  - ⋆ la fonction retourne un entier représentant le numéro du *thread* ou -1 si le *thread* n'a pu être créé. Le *thread* créé est initialement dans l'état *suspendu* (état `SUSPENDED`). Pour le rendre prêt à être exécuté (état `READY`), il faut utiliser la fonction `resume` que nous décrivons ci-dessous.
- La fonction `bool resume(int thread_id)` (fichier `thread.h`) permet de passer le *thread* `thread_id` de l'état `SUSPENDED` à l'état `READY`.

1.2 Exemple

Nous donnons ci-dessous un exemple de programme que nous allons commenter :

```c
#include <stdio.h>
#include "thread.h"

void fct(long n)
{
  while(1) {
    printf("%ld\n", n);
  }
}

int main()
{
  int t1 = create_thread((long)fct, DEFAULT_PRIORITY, "thread1", 1, 1);
  int t2 = create_thread((long)fct, DEFAULT_PRIORITY, "thread2", 1, 2);
  int t3 = create_thread((long)fct, DEFAULT_PRIORITY, "thread3", 1, 3);
  resume(t1);
  resume(t2);
  resume(t3);
```

```
  return 0;
}
```

Ce programme va créer trois *threads*, dont le code est la fonction `fct`. La fonction `fct` pour le *thread* `t1` aura comme paramètre la valeur `1`, pour le *thread* `t2` ce sera la valeur `2` et pour le *thread* `t3` la valeur `3`. Les trois *threads* passent dans l'état `READY` grâce à la fonction `resume`. Notons que la fonction `main` est aussi un *thread* qui est créé par une des fonctions d'initialisation de la bibliothèque.
Lorsque l'on exécutera ce programme, on verra apparaître à l'écran une succession de `1`, `2` et `3`. Nous allons décrire dans la sous-section suivante comment fonctionne l'ordonnancement entre les *threads*.

1.3 Ordonnancement

Un *thread* qui est prêt à s'exécuter (état `READY`) sera choisi par l'ordonnanceur selon sa priorité. Afin que tous les *threads* (ayant la priorité la plus forte) puissent être servis de manière équitable, il est alloué un certain quantum de temps ($100ms$) à chaque *thread* avant que l'ordonnanceur ne soit appelé en vu de passer la main à un autre *thread* de même priorité. La variable globale `preempt` permet de connaître le temps d'exécution restant pour le *thread* en cours d'exécution.
Toutes les $10ms$, le traitant d'interruption `clock_handler` est appelé. Son rôle est de compter le nombre de pas de temps (par tranche de $10ms$) qui s'est écoulé depuis le début du programme (les interruptions liées au *timer* sont activées par une des fonctions d'initialisation de la bibliothèque) et d'appeler l'ordonnanceur quand le temps alloué au *thread* courant est terminé.

Le numéro du *thread* courant est dans la variable globale `current_thread`. La liste globale `ready_list` contient les numéros des *threads* prêts à s'exécuter triés par ordre de priorité. Le *thread* courant n'est pas dans la `ready_list`. La fonction `bool reschedule()` permet de changer le *thread* courant en passant la main à un des *threads* dont le numéro est dans la `ready_list`. Cette fonction renvoie `true` si on a effectivement changé de *thread*. Elle renvoie `false` si aucun *thread* de même priorité (ou supérieure) n'était présent dans la `ready_list`. Notons que c'est cette fonction qui, entre autres, réinitialise la variable `preempt` pour que le *thread* qui va prendre la main puisse avoir son quantum de temps d'exécution.

La fonction `bool ready(int thread_id, bool resched)` est une fonction utilitaire qui permet de mettre dans la liste des prêts (`ready_list`) le *thread* `thread_id` et d'appeler ou non la fonction `reschedule` si le paramètre `resched` est à la valeur `true`. Cette fonction rend `true` si le numéro de *thread* est correct et renvoie `false` sinon.

Question 1 : compléter le traitant d'interruptions `clock_handler` (fichier `time.c`) afin de passer la main à un autre *thread* lorsque le quantum de temps du *thread* en cours est dépassé. Compiler et exécuter le programme décrit plus haut afin de le tester.

1.4 Structures de données

Le fichier `thread.h` décrit les structures de données liées aux *threads*. Nous allons les passer en revue.

```
#define DEFAULT_PRIORITY 10
```

Cette constante est utilisable par l'utilisateur lorsqu'il crée un *thread* pour indiquer une priorité par défaut.

```
#define MAX_NB_THREAD 16
```

Cette constante permet de fixer le nombre maximum de *threads* que l'utilisateur pourra créer.

```
#define STACK_SIZE 65536
```

Cette constante définie la taille de la pile allouée à un *thread*.

```
#define NULL_THREAD 0
```

Cette constante est le numéro d'un *thread* spécial qui est créé à l'initialisation du système. Ce *thread* prend la main au lancement du système et lorsqu'aucun *thread* n'est prêt à s'exécuter.

```
#define is_bad_thread_id(x) (x <= 0 || x >= MAX_NB_THREAD)
```

La *macro* `is_bad_thread_id` permet de savoir si le numéro de *thread* est un numéro valide.

```
#define NB_REGISTERS 8
```

Cette constante définie le nombre de registres à sauvegarder lors d'un changement de contexte.

```
enum {CURRENT, READY, WAITING, JOIN, ASLEEP, SUSPENDED, RECEIVE, FREE};
```

Cette énumération représente les différents états dans lesquels peut se trouver un *thread* :

- `CURRENT`, le *thread* est en cours d'exécution (il n'y a qu'un seul *thread* dans cet état). Le numéro du *thread* est alors dans la variable `current_thread` ;
- `READY`, le *thread* est prêt à s'exécuter. Le numéro du *thread* est alors dans la `ready_list` ;
- `WAITING`, le *thread* est bloqué dans la file d'attente d'un sémaphore (voir la section suivante) ;
- `JOIN`, le *thread* est en attente de la destruction d'un autre *thread* ;
- `ASLEEP`, le *thread* est endormi et sera réveillé par le `clock_handler` lorsque le temps d'attente sera écoulé ;
- `SUSPENDED`, le *thread* est suspendu. Pour qu'il redevienne prêt à s'exécuter, il faut utiliser la méthode `resume`. C'est l'état initial d'un *thread* après sa création ;
- `RECEIVE`, le *thread* est dans l'attente d'un message ;
- `FREE`, le *thread* n'est pas alloué.

```
typedef struct {
  int   state;
  int   priority;
  int   semaphore;
  char* name;
  long  registers[NB_REGISTERS];
  long  stack[STACK_SIZE];
  list  join_list;
  int   join_thread;
  int   message;
  bool  received;
} thread;
```

Cette structure représente le contexte d'un *thread* :

- `state`, l'état du *thread* ;
- `priority`, la priorité du *thread* (plus ce nombre est élevé, plus la priorité est forte) ;
- `semaphore`, le numéro de sémaphore dans la file duquel le *thread* est bloqué ou -1 si le *thread* n'est dans la file d'attente d'aucun sémaphore. Ce numéro permet de retirer le *thread* de la file d'attente du sémaphore lors de la destruction du *thread* ;

- `name`, le nom du *thread* ;
- `registers`, la valeur des registres pour sauvegarder l'état de la machine lors de la commutation de contexte et pour remettre correctement à jour les registres lorsque le *thread* redevient actif ;
- `stack`, la pile du *thread*. En effet, les variables locales, les paramètres des fonctions et les adresses de retour de celles-ci doivent être sauvegardées lors de la commutation de contexte, et remises à jour correctement lorsque le *thread* reprend la main ;
- `join_list`, la liste des *threads* qui sont en attente de la destruction de ce *thread*. Lors de la destruction du *thread*, les *thread* contenus dans cette liste vont être mis dans l'état `READY` et pourront donc de nouveau être exécutés ;
- `join_thread`, le *thread* dont on attend la destruction. Ceci nous permet de retirer le *thread* de la liste `join_list` du *thread* `join_thread` lors de la destruction du *thread* (sinon lorsque le *thread* `join_thread` sera détruit il voudra remettre dans l'état `READY` un *thread* qui n'existe plus ou qui n'est pas le bon) ;
- `message`, le message envoyé à ce *thread* (dans cette implémentation, le message n'est qu'un entier) ;
- `received`, indique si un message a été envoyé à ce *thread*.

```
extern thread thread_table[];
```

Ce tableau contient les informations sur les *threads*. Notons que le numéro d'un *thread* correspond à son indice dans ce tableau.

```
extern int nb_thread;
```

Cette variable contient le nombre actuel de *threads* dans le système.

```
extern int current_thread;
```

Cette variable contient le numéro du *thread* courant (en cours d'exécution).

2 Atomicité

Dans notre gestionnaire de *threads*, il est important d'assurer l'atomicité de certaines opérations. Par exemple, lors de la création d'un *thread*, on va rechercher dans la table des *threads* `thread_table` un *thread* dont l'état est `FREE`. Le code pourrait analyser les entrées de la table `thread_table` et se rendre compte que l'entrée 2, par exemple, est libre. S'il y a commutation de contexte à ce moment là, le *thread* qui va prendre la main pourrait à son tour créer un autre *thread* et voir que l'emplacement 2 est libre dans la table `thread_table`. La seule façon de perdre la main pour un *thread* non volontairement est d'être interrompu par une interruption. On dispose de trois fonctions permettant de masquer ou non les interruptions :

- `void enable()`, autorise les interruptions ;
- `status disable()`, désactive les interruptions et renvoie l'état actuel des interruptions (masquées ou non) ;
- `void restore(status old)`, replace les interruptions dans l'état `old`.

3 Synchronisation des *threads*

Nous allons présenter les fonctions permettant de synchroniser les *threads*.

3.1 Fonctions *yield* et *join*

La procédure `void yield()` permet au *thread* courant de passer la main à un autre *thread* de priorité supérieure ou égale (s'il y en a un). La fonction `bool join(int thread_id)` permet de mettre en attente le *thread* courant jusqu'à la mort du *thread* `thread_id`. Cette fonction renvoie `false` si le *thread* `thread_id` n'existe pas. Elle renvoie `true` sinon.

Question 2 : compléter le programme de la figure 1 (fichier `main.c`) afin que l'affichage résultant de l'exécution de ce programme soit :

```
fct2
fct1
fin du main
```

```c
#include <stdio.h>
#include "thread.h"

void fct1(long t)
{
  printf("fct1\n");
}

void fct2()
{
  printf("fct2\n");
}

int main()
{
  int t2 = create_thread((long)fct2, DEFAULT_PRIORITY, "thread2", 0);
  int t1 = create_thread((long)fct1, DEFAULT_PRIORITY, "thread1", 1, t2);
  resume(t1);
  resume(t2);
  printf("fin du main\n");
  return 0;
}
```

FIGURE 1 – Programme à compléter pour obtenir l'affichage décrit dans le texte

Question 3 : compléter la procédure `yield` et la fonction `join` (fichier `yield.c` et `join.c`).

3.2 Sémaphores

3.2.1 Principe

Les *sémaphores* vont permettre de synchroniser les *threads* de maniére plus fine. Un sémaphore possède un compteur et quatre opérations :

- l'opération de création, qui permet de créer un sémaphore en initialisant son compteur à la valeur désirée ;
- l'opération de destruction, qui permet de détruire le sémaphore ;
- l'opération P décrémente le compteur et :
  - ⋆ si le compteur est négatif, le *thread* courant est mis dans l'état `WAITING`, il est placé dans la file d'attente associée au sémaphore et un autre *thread* prend la main ;
  - ⋆ si le compteur est positif ou nul, le *thread* courant garde la main.

- l'opération V incrémente le compteur et :
  - ⋆ si le compteur est négatif ou nul, un *thread* de la file d'attente du sémaphore est retiré de celle-ci et placé dans la `ready_list` ;
  - ⋆ si le compteur est strictement positif, aucune autre action est nécessaire.

Dans notre gestionnaire de *threads*, l'opération de création d'un *semaphore* est la fonction `int create_semaphore(int count)`, elle prend en paramètre la valeur initiale du compteur et retourne l'identifiant du sémaphore ou -1 s'il n'y a plus de sémaphore disponible.

L'opération de destruction d'un sémaphore est la fonction `bool destroy_semaphore(int sem)`. Elle prend en paramètre l'identifiant d'un *sémaphore* et le détruit. Cette fonction retourne `true` si l'identifiant est valide et `false` sinon.

L'opération P est la fonction `bool p(int sem)`. Elle prend en paramètre l'identifiant d'un sémaphore, elle renvoie `false` si le numéro de *sémaphore* passé en paramètre n'est pas valide et renvoie `true` sinon.

L'opération V est la fonction `bool v(int sem)`. Elle prend en paramètre l'identifiant d'un sémaphore, elle renvoie `false` si le numéro de *sémaphore* passé en paramètre n'est pas valide et renvoie `true` sinon.

```
#include <stdio.h>
#include "thread.h"
#include "semaphore.h"

const unsigned int N = 1000000000;
volatile int n = 0;

void fct1()
{
  for (unsigned int i = 0; i < N; i++) n++;
}

void fct2()
{
  for (unsigned int i = 0; i < N; i++) n--;
}

int main()
{
  int t1 = create_thread((long)fct1, DEFAULT_PRIORITY, "thread1", 0);
  int t2 = create_thread((long)fct2, DEFAULT_PRIORITY, "thread2", 0);
  resume(t1);
  resume(t2);
  join(t1);
  join(t2);
  printf("%d\n",n);
  return 0;
}
```

FIGURE 2 – Programme à compléter pour protéger l'accès à la variable globale

Question 4 : soit le programme de la figure 2 (fichier `main.c`). Si on exécute ce programme, la valeur de n affichée dans le `main` pourra être différente de 0. Pourquoi ?
On voudrait protéger l'accès à la variable globale `n`. Ajouter un sémaphore permettant de résoudre ce problème.

```c
#include <stdio.h>
#include "thread.h"
#include "semaphore.h"

void fct1()
{
  printf("avant synchronisation\n");
  printf("après synchronisation\n");
}

void fct2()
{
  printf("avant synchronisation\n");
  printf("après synchronisation\n");
}

void fct3()
{
  printf("avant synchronisation\n");
  printf("après synchronisation\n");
}

int main()
{
  resume(create_thread((long)fct1, DEFAULT_PRIORITY, "thread1", 0));
  resume(create_thread((long)fct2, DEFAULT_PRIORITY, "thread2", 0));
  resume(create_thread((long)fct3, DEFAULT_PRIORITY, "thread3", 0));
  return 0;
}
```

FIGURE 3 – Programme à compléter pour obtenir un rendez-vous

Question 5 : soit le programme de la figure 3 (fichier `main.c`). On voudrait faire en sorte que les trois messages `avant synchronisation` s'affichent avant les trois messages `après synchronisation`. Réaliser ceci grâce à des sémaphores.

3.2.2 Structures de données

Les structures de données associées à un *sémaphore* sont les suivantes (fichier `semaphore.h`) :

```
typedef struct {
  int state;
  int count;
  list waiting_list;
} semaphore;
```

- `state` indique si le sémaphore est utilisé ou non (`SFREE` ou `SUSED`) ;
- `count` est le compteur associé au sémaphore ;
- `waiting_list` est la liste des *threads* en attente sur ce sémaphore.

Le tableau `extern semaphore semaphore_table[];` contient les informations sur les *semaphores*. Le numéro d'un *semaphore* correspond à son indice dans ce tableau.

Question 6 : implémenter la fonction `bool p(int sem)` (fichier `semaphore.c`) qui réalise l'opération P décrite ci-dessus.

Question 7 : implémenter la fonction `bool v(int sem)` (fichier `semaphore.c`) qui réalise l'opération V décrite ci-dessus.

4 Communication entre *threads*

Nous allons proposer un mécanisme simple de communication entre *threads* grâce aux fonctions `send` et `receive` dont le fonctionnement est décrit ci-dessous :

- `int receive()`. Lorsque le *thread* courant fait cet appel, deux cas de figures peuvent se passer :
  - ⋆ un message lui a déjà été envoyé par un autre *thread* et cet appel retourne immédiatement le message reçu après avoir mis à `false` le champ `received` du contexte du *thread* courant ;
  - ⋆ aucun message ne lui a encore été envoyé, et le *thread* courant est alors mis en attente dans l'état `RECEIVE`.

- `bool send(int thread_id, int message)` permet d'envoyer au *thread* `thread_id` l'entier `message`. L'entier `message` est copié dans le champ `message` du contexte associé au *thread* `thread_id` et le champ `received` de cette structure est mis à `true`. Si le *thread* `thread_id` était en attente de recevoir un message, il est alors remis dans la `ready_list`. La fonction retourne alors la valeur `true`. Notons que si un message n'avait pas encore été lu par le *thread* `thread_id`, ou si l'identifiant `thread_id` n'existe pas, la fonction ne fait que renvoyer `false`.

Question 8 : soit le programme de la figure 4 (fichier `main.c`). On veut que le *thread1* envoie 10 messages (les entiers de 0 à 9) au *thread2* qui les affichera. Réaliser ceci grâce aux primitives `send` et `receive`.

Question 9 : implémenter les fonctions `send` et `receive` (fichier `message.c`).

5 Endormir un *thread*

La fonction `bool sleep(int n)` (fichier `sleep.h`) permet d'endormir le *thread* courant pendant $(n \times 10)ms$. Cette fonction retourne `true` si $n > 0$ et `false` sinon. Un *thread* mis en sommeil sera mis dans

```c
#include <stdio.h>
#include "thread.h"
#include "message.h"

int t1, t2;

void fct1()
{
}

void fct2()
{
}

int main()
{
  t1 = create_thread((long)fct1, DEFAULT_PRIORITY, "thread1", 0);
  t2 = create_thread((long)fct2, DEFAULT_PRIORITY, "thread2", 0);
  resume(t1);
  resume(t2);
  return 0;
}
```

FIGURE 4 – Programme à compléter afin d'envoyer des messages entre deux *threads*

l'état `ASLEEP`.

La procédure `void wake_up()` (fichier `sleep.h`) permet de réveiller le ou les *threads* dont le temps de mise en sommeil est terminé (cette procédure est appelée dans le traitant d'interruption `clock_handler` à chaque appel de celui-ci).

Question 10 : implémenter puis tester les fonctions `sleep` et `wake_up` (fichier `sleep.c`).