Architecture et système d'exploitation — Bibliothèque de *threads* partie 2

Pascal Garcia

23 mars 2022

1 Création de *threads*

Nous avons utiliser dans le *TP* précédent la fonction `create_thread`, qui permet à l'utilisateur de créer des *threads*. Nous allons maintenant implémenter cette fonction, mais nous allons tout d'abord regarder comment le compilateur *gcc* gère l'appel d'une procédure, et comment utiliser une liste variable d'arguments en *C*.

1.1 Gestion des appels de procédures par le compilateur *gcc*

Lors de l'appel à une fonction, au tout début de l'entrée dans la fonction, le haut de la pile contient l'adresse de retour. Les 6 premiers arguments de la fonction se trouvent respectivement dans les registres de 64 bits : `rdi`, `rsi`, `rdx`, `rcx`, `r8` et `r9` et les autres arguments se trouvent dans la pile. Pour simplifier, nous n'utiliserons pas plus de 6 arguments pour un `thread` et donc nous ne nous soucierons pas de la gestion des arguments dans la pile. Nous rappelons que la pile croît des adresses les plus hautes vers les adresses les plus basses.

En fin de fonction, avant l'instruction `ret`, le pointeur de pile (`rsp`) pointe sur l'adresse de retour et l'instruction `ret` aura pour effet de dépiler cette valeur dans le compteur ordinal (`rip`).

1.2 Utilisation d'une liste variable d'arguments en *C*

Nous allons illustrer l'utilisation des listes variables d'arguments grâce au programme suivant :

```c
#include <stdarg.h>
#include <stdio.h>

void affiche(int nb_args, ...)
{
  va_list varg;
  va_start(varg, nb_args);
  for (int j = 0; j < nb_args; j++)
    {
      printf("paramètre %d : %d\n", j+1, va_arg(varg, int));
    }
  va_end(varg);
}

int main() {
  affiche(5, 1, 2, 3, 4, 5);
  return 0;
}
```

Après avoir déclaré une variable `varg` de type `va_list`, on l'initialise grâce à la procédure `va_start` en donnant en paramètre le nombre d'arguments. On peut ensuite accéder aux différents arguments en précisant leurs types grâce à la fonction `va_arg`. Enfin, on appel la procédure `va_end` (à chaque invocation de `va_start` doit correspondre une invocation de `va_end` dans la même fonction).

1.3 Implémentation

Nous allons maintenant implémenter la fonction :

```
int create_thread(long initial_address, int priority, char* name, int nb_args, ...)
```

qui va nous permettre de créer des *threads*. Pour ce faire, nous allons avoir besoin :

- de la procédure `enable_interrupt` qui permet de démasquer les interruptions (fichier `thread.c`) dont le code est le suivant :

```
void enable_interrupt()
{
  thread* threadptr = &thread_table[current_thread];
  enable();
  long rdi, rsi, rdx, rcx, r8, r9;
  rdi = threadptr->registers[REGISTER_TO_INDEX(RDI)];
  rsi = threadptr->registers[REGISTER_TO_INDEX(RSI)];
  rdx = threadptr->registers[REGISTER_TO_INDEX(RDX)];
  rcx = threadptr->registers[REGISTER_TO_INDEX(RCX)];
  r8 = threadptr->registers[REGISTER_TO_INDEX(R8)];
  r9 = threadptr->registers[REGISTER_TO_INDEX(R9)];
  __asm__ volatile("movq %0, %%rdi\n\t"
      "movq %1, %%rsi\n\t"
      "movq %2, %%rdx\n\t"
      "movq %3, %%rcx\n\t"
      "movq %4, %%r8\n\t"
      "movq %5, %%r9\n\t"
      ::"g"(rdi), "g"(rsi), "g"(rdx), "g"(rcx), "g"(r8), "g"(r9)
      : "rdi", "rsi", "rdx", "rcx", "r8", "r9");
}
```

  Cette procédure permet de démasquer les interruptions lorsque le *thread* va prendre la main pour la première fois. Cette procédure fait appel à la procédure `enable` permettant de démasquer les interruptions, et restaurent ensuite les registres contenant les paramètres éventuels du *thread* qui sont écrasés par l'appel à `enable`.

- de la procédure `userret` qui détruit le *thread* courant et dont le code est simplement :

```
void userret()
{
  destroy_thread(get_thread_id());
}
```

  Lorsqu'un *thread* sera terminé, cette procédure va permettre de libérer ses ressources et de passer la main à un autre *thread*.

La fonction `create_thread` doit réaliser les opérations suivantes :

- chercher un numéro de *thread* libre (index correspondant à une entrée `FREE` du tableau `thread_table`) ;
- ajouter l'identifiant du *thread* dans la liste `alive_list` (fichier `thread.h`) qui permet de connaître les *threads* présents dans le système (la liste `ready_list` ne suffit pas) ;

- mettre à jour la pile du *thread* (l'adresse la plus basse de la pile correspond à l'adresse de `stack[0]` de la structure `thread` (fichier `thread.h`)) afin qu'elle soit comme celle de la figure 1. Nous allons maintenant expliquer pourquoi la pile initiale du *thread* doit avoir cette forme. Nous verrons dans la section suivante que la commutation entre deux *threads* se fera dans une procédure écrite en assembleur qui aura la forme suivante :

  ```
  sauvegarde des registres du thread qui perd la main
  restauration des registres du thread qui va prendre la main
  ret
  ```

  L'instruction assembleur `ret` (retour d'un appel de fonction) a pour effet de mettre dans le compteur ordinal la valeur en sommet de pile et d'incrémenter le pointeur de pile. Donc, lorsque l'on commutera pour la première fois sur un *thread*, l'instruction `ret` aura pour effet de mettre dans le compteur ordinal l'adresse de la procédure `enable_interrupt` (qui permet de démasquer les interruptions). Quand cette procédure se terminera, son instruction `ret` aura pour effet de mettre dans le compteur ordinal l'adresse de la fonction qui correspond au code du *thread*. La pile aura alors la forme indiquée dans la figure 2. On voit alors que l'adresse de retour de la fonction est le début du code de la fonction `userret` qui permettra de libérer les ressources allouées au *thread* et de passer la main à un autre *thread*.
  Pour résumer la fonction `create_thread` doit ;
- mettre à jour les registres de la structure `thread` correspondants aux paramètres du *thread* ;
- mettre à jour le registre `RSP` de la structure `thread` qui indique la valeur sauvegardée (ici la valeur initiale) du pointeur de pile ;
- mettre à jour l'état du *thread* (initialement dans l'état `SUSPENDED`), sa priorité, son nom ;
- incrémenter le nombre de *threads* existant dans le système (variable globale `nb_thread`).

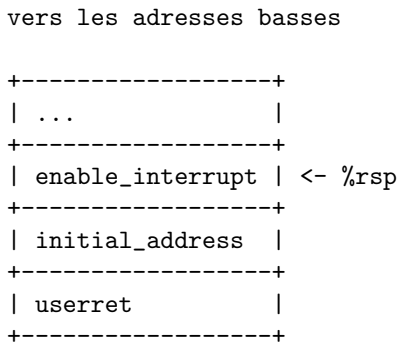

FIGURE 1 – État initial de la pile d'un *thread*

Question 1 : Implémenter la fonction `create_thread` (fichier `thread.c`).

2 Changement de contexte

Nous allons maintenant réaliser la commutation de contexte entre deux *threads*. Pour ce faire, nous aurons besoin d'une procédure écrite en assembleur (fichier `ctxsw.h`) :

```
void ctxsw(long* old_registers, long* new_registers)
```

Le paramètre `old_registers` est l'adresse du tableau `long registers[NB_REGISTERS]` (voir la structure `thread` du fichier `thread.h`) du *thread* courant qui va perdre la main. Le paramètre `new_registers` est l'adresse du tableau `long registers[NB_REGISTERS]` du *thread* qui va prendre la main. Cette procédure va permettre de sauvegarder les registres du *thread* courant dans le tableau dont l'adresse est

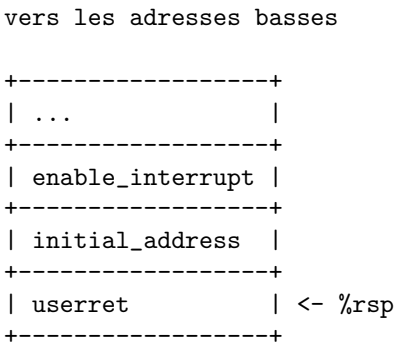

FIGURE 2 – État de la pile lorsque le code de la fonction correspondante au *thread* va être exécuté

`old_registers` et de mettre à jour les registres de la machine grâce aux valeurs contenues dans le tableau situé à l'adresse `new_registers`. Si `thread1` est le *thread* courant et `thread2` le *thread* vers lequel on veut commuter, on effectuera l'appel suivant :

```
ctxsw(thread_table[thread1].registers, thread_table[thread2].registers)
```

Question 2 : Implémenter la fonction `bool reschedule()` (fichier `reschedule.c`) qui :
- choisit un des *threads* de la `ready_list` de priorité supérieure ou égale à la priorité du *thread* courant et,
- effectue le changement de contexte entre le *thread* courant et le *thread* qui va prendre la main.

Cette fonction rend `false` si on n'a pas trouvé de *thread* de priorité supérieure ou égale et `true` sinon.

Nous allons maintenant écrire la procédure `ctxsw`, mais pour ce faire nous allons décrire la partie de la syntaxe assembleur qui nous sera utile :
- `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14` et `%r15` sont les registres (64 bits) qu'il faudra sauvegarder dans le champ `register` de la structure `thread`. Le registre `%rsp` est le pointeur de pile. Notons qu'on ne peut pas manipuler directement le compteur ordinal. Mais celui-ci sera placé sur la pile lors de l'appel à la procédure `ctxsw` (nous allons détailler ce point après).
- Soit `v` le contenu du registre `reg1`, alors l'instruction `movq n(reg1), reg2` (par exemple `movq 8(%rsp),%rax`) mettra le contenu de l'adresse `(v + n)` dans le registre `reg2` ;
- Soit `v` le contenu du registre `reg1`, alors l'instruction `movq reg2, n(reg1)` (par exemple `movq %rcx,8(%rax)`) mettra le contenu du registre `reg2` à l'adresse `(v + n)`.

La procédure `ctxsw` est à compléter dans le fichier `ctxsw.S`. Le squelette de cette procédure est le suivant :

```
        .globl  ctxsw
ctxsw:
        ...
        ret
```

Lors de l'appel à cette procédure, la pile du *thread* courant sera dans l'état représenté dans la figure 3. On peut voir que l'adresse de retour est déjà dans la pile du *thread* courant et n'a donc pas besoin d'être sauvegardée autre part. L'adresse du tableau `old_registers` est dans le registre `%rdi` et celle du tableau `new_registers` se trouve dans le registre `%rsi`. Il reste donc à sauvegarder les registres dans le tableau `old_registers` et mettre à jour les registres grâce au contenu du tableau `new_registers`.

Question 3 : Implémenter la procédure `void ctxsw(long* old_registers, long* new_registers)` (fichier `ctxsw.S`).

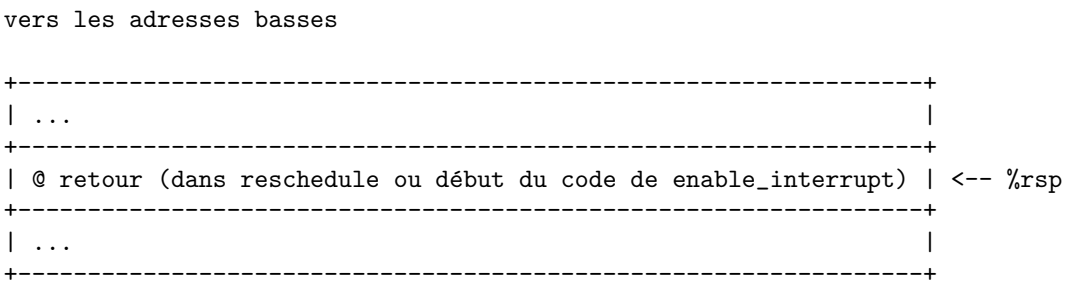

FIGURE 3 – État de la pile du *thread* courant lors de l'appel à *ctxsw*

3 Initialisation du système

Il nous reste à initialiser le système et à créer un *thread* démon qui prendra la main lorsqu'aucun *thread* ne sera prêt à s'exécuter (ce qui peut arriver si tous les *threads* sont endormis par exemple). La procédure `void sysinit()` (fichier `initialize.c`) est déjà écrite. Il vous reste à écrire la procédure `void nulluser()` (fichier `initialize.c`) qui est le point d'entrée de la bibliothèque (le `main` étant le point d'entrée du programme utilisateur). Une ébauche de la procédure `void nulluser()` est donnée ci-dessous :

```
void nulluser() {
  sysinit();
  //Créer le thread main et le lancer

  //Si on arrive ici c'est que plus aucun autre thread n'est prêt à s'exécuter (en effet,
  //le thread nulluser a la plus faible priorité). Si la liste 'alive_list' est vide, on
  //peut quitter le programme sinon, il faut effectuer une attente active.

  printf("no more thread to run... bye\n");
  _exit(0);
}
```

Question 4 : Implémenter la procédure `void nulluser()` (fichier `initialize.c`).