TP3 et 4: Co-routines et threads coopératifs

Matthieu Lemerre Guillaume Girol Grégoire Menguy

IN201 - Cours de système d'exploitation

Contents

Le but de ce TP est de mettre en place un système de threads coopératifs, tels qu'on peut les trouver dans les systèmes embarqués, ou dans les runtimes de languages comme Javascript, Python, Go...

Important: pour ce TP, il faudra que vous linkiez les fichiers C et assembleur avec les options de compilateur `-static`, sinon votre code pourrait ne pas marcher. Par exemple, "gcc -static -Wall tp3.c tp3.s -o tp3".

1 Co-routines

Dans ce TP, nous utiliserons une taille de pile de 4096 octets.

```
#define STACK_SIZE 4096
```

Question 1. *Allouer statiquement 4 piles de taille* `STACK_SIZE`.

Une co-routine peut être représentée par un pointeur sur une pile:

```
typedef void * coroutine_t;
```

et manipulée par les trois fonctions suivantes:

```
/* Quitte le contexte courant et charge les registres et la pile de CR. */
void enter_coroutine(coroutine_t cr);

/* Sauvegarde le contexte courant dans p_from, et entre dans TO. */
void switch_coroutine(coroutine_t *p_from, coroutine_t to);

/* Initialise la pile et renvoie une coroutine telle que, lorsqu'on entrera dedans,
   elle commencera à s'exécuter à l'adresse initial_pc.  */
coroutine_t init_coroutine(void *stack_begin, unsigned int stack_size,
                           void (*initial_pc)(void));
```

Nous proposons d'utiliser le format de pile suivant pour les co-routines qui ne sont pas exécutées. Nota bene: en x86, la pile "pousse vers le bas", i.e. rajouter quelque chose à la pile fait décrémenter l'adresse du sommet de la pile.

```
| pile non-utilisee         |
|---------------------------|   <-- adresse de la coroutine
| valeur de r15             |
| valeur de r14             |
| valeur de r13             |
| valeur de r12             |
| valeur de rbx             |
| valeur de rbp             |
| valeur du program counter |
|---------------------------|   <--- adresse de la coroutine + 20
| pile utilisée             |
|                           |
```

En d'autres termes les registres de la co-routine sont stockés par dessus le reste de la pile dans l'ordre "pc, rbp, rbx, r12, r13,r14,r15". Les registres "caller-saved"/"volatile" (rax,rcx,rdx,r1 à r11, les flags) ne sont pas sauvés.

Question 2. *Complétez l'implémentation suivante de* `enter_coroutine`. *La fonction prend une co-routine non utilisée, doit charger les valeurs pour les registres rsp (stack pointer), rbx, rbp, r12,r13,r14 et r15 (program counter). L'argument de la fonction se trouve initialement dans rdi.*

```
.global enter_coroutine       // Makes enter_coroutine visible to the linker
enter_coroutine:
        mov %rdi,%rsp         /* RDI contains the argument to enter_coroutine. */
```

```
                                /* And is copied to RSP. */
        pop %r??
        pop %r??
        pop %r??
        pop %r??
        pop %r??
        pop %r??
        ret                     /* Pop the program counter */
```

Question 3. *Complétez l'implémentation suivante de* `init_coroutine`*. Cette fonction initialise une co-routine*

```
coroutine_t init_coroutine(void *stack_begin, unsigned int stack_size,
                           void (*initial_pc)(void))
{
  char *stack_end = ((char *)stack_begin) + stack_size;
  void **ptr = stack_end;
  ptr--;
  *ptr = initial_pc;
  ptr--;
  *ptr = ...
  ptr--;
  ...
  return ptr;
}
```

Question 4. *Testez votre code en créant une fonction ne prend ni ne renvoie aucun argument; qui boucle indéfiniment (en affichant un message et incrémentant un compteur); en initialisant une coroutine tel que initial_pc commence à cette fonction; puis en entrant dans cette co-routine.*

Question 5. *Complétez l'implémentation suivante de* `switch_coroutine`*. Cette fonction doit être exécutée depuis une co-routine; elle sauvegarde l'état de la co-routine courante et charge l'état d'une autre co-routine. Le premier argument de* `switch_coroutine` *se trouve dans le registre rdi, et le second dans rsi.*

```
.global switch_coroutine      // Makes switch_coroutine visible to the linker
switch_coroutine:
        push %r??
        push %r??
```

```
push %r??
push %r??
push %r??
push %r??
mov %rsp,(%rdi)       /* Store the stack pointer to *(first argument) */
mov %rsi,%rdi
jmp enter_coroutine   /* Call enter_coroutine with the second argument. */
```

Question 6. *Testez votre code en écrivant deux co-routines dans deux fonctions qui bouclent indéfiniment en incrémentant une variable locale, affichent un message différent, et passe la main à l'autre co-routine.*

2 Threads

Le problème des co-routines définies précédemment est que pour changer de co-routine, il est nécessaire de savoir vers qui "rendre la main". Or, on a souvent envie dans un programme multi-thread d'écrire son code indépendamment des autres threads qui peuvent être exécutés.

Dans cette section, nous allons implémenter le mécanisme de thread plus général au dessus du mécanisme de co-routine. L'idée est simplement d'avoir une co-routine principale (l'ordonnanceur) qui va donner la main à toutes les autres co-routine à tour de rôle. Les autres co-routines, quand elles veulent rendre la main, rendront la main à l'ordonnanceur.

Nous définissons un thread comme un pointeur vers le contexte d'une coroutine, et un statut pouvant avoir comme valeur *prêt* ou *bloqué*.

Question 7. *Définir un type de donnée correspondant à un thread.*

Le système de thread est implémenté de la manière suivante:

- Une coroutine est dédiée à l'*ordonnancement* (*scheduling*), i.e. la co-routine va choisir un thread prêt, et passer la main à la co-routine correspondante. Pour commencer nous conseillons d'implanter l'algorithme du tourniquet (i.e. exécuter chaque thread tour à tour).
- Une fonction `void yield(void)` doit être appellée par les threads périodiquement, afin que ceux-ci puissent permettre l'exécution des autres threads. Son implémentation consiste à passer la main à la co-routine d'ordonnancement (pour l'implémenter, il est conseillé de stocker un pointeur vers le thread courant dans une variable globale).

- Une fonction `thread_create`, qui initialise un thread et sa co-routine correspondante avec une fonction passée en paramètre. Le thread est initialement prêt à être exécuté.

Question 8. *Implémentez la co-routine d'ordonnancement, la fonction yield, et tester sur un programme qui exécute 3 threads incrémentant un compteur et affichant un message en boucle.*

3 Non-blocking IO

Le comportement par défaut de la fonction `getchar()`, qui renvoie un caractère tapé au clavier, est de bloquer le processus quand rien n'a été écris.

Question 9. *En quoi cela pose problème dans le cadre de notre système de thread coopératif?*

On peut rendre la fonction `getchar()` non bloquante en appellant, au début de la fonction main, la fonction `fcntl` ainsi:

```
fcntl(0, F_SETFL, fcntl(0, F_GETFL) | O_NONBLOCK);
```

Cela demande d'avoir auparavant inclu les headers suivants:

```
#include <unistd.h>
#include <fcntl.h>
```

Question 10. *Modifiez l'un de vos threads pour qu'il remette son compteur à 0 lorsqu'une entrée est tapée au clavier. Notes: la fonction* `getchar()` *renvoie -1 lorsque rien n'est écrit au clavier; l'entrée clavier n'est envoyé au programme que lorsque la touche entrée est appuyée.*

4 Application, synchronisation et communication

Nous allons implémenter le système suivant, permettant de distribuer un travail à faire à plusieurs threads.

- Il y a trois threads, un producteur et 2 consommateurs;
- Le producteur va lire une entrée sur le clavier et la placer en mémoire

- Un des consommateurs, A ou B, va lire cette entrée et écrire (fonction `putchar()`) autant de caractères A (ou B) que la touche clavier correspondante. Le consommateur doit appeller `yield()` après chaque caractère écrit.

- La communication se fera par l'intermédiaire de deux variables globales partagées:

  - L'une contiendra la valeur de la touche écrite par producteur
  - L'autre servira à la synchronisation, et pourra prendre deux valeurs:

    FULL (1) Lorsque le producteur a écris la valeur. Le producteur ne peut alors plus écrire de nouvelle valeur, mais un des consommateur peut la lire.

    EMPTY (0) Lorsque l'un des consommateurs a lu la valeur. Aucun consommateur ne peut plus alors lire la valeur, mais le producteur peut écrire une nouvelle valeur.