TP 4 : Ordonnanceur de *threads* coopératif

Utilisation de filaments *System V*

Systèmes et supports d'exécution — 2008–2009

Alexandre DENIS, Olivier AUMAGE
denis@labri.fr, aumage@labri.fr
http://dept-info.labri.fr/~denis/

L'objectif de ce TP est de concevoir un ordonnanceur de *threads* coopératif en utilisant les primitives `ucontext` des filaments *System V*. Nous testerons cet ordonnanceur avec le *quicksort* multi-thread écrit lors des deux TP précédents.

1. `ucontext` : les filaments *System V*

Sur la plupart des systèmes Unix, l'API `ucontext` permet de gérer des contextes pour créer des filaments. Pour l'utiliser, il faut bien entendu commencer par un :

```
#include <ucontext.h>
```

Cette API contient essentiellement les types et fonctions suivantes :

```
typedef struct
{
  void  *ss_sp;     /* Base address of stack */
  int    ss_flags;  /* Flags */
  size_t ss_size;   /* Number of bytes in stack */
} stack_t;

typedef struct ucontext
{
  struct ucontext *uc_link;
  sigset_t uc_sigmask;
  stack_t uc_stack;
  mcontext_t uc_mcontext;
  ...
} ucontext_t;

int  getcontext (ucontext_t *ucp);
int  setcontext (const ucontext_t *ucp);
int  swapcontext(ucontext_t *oucp, ucontext_t *ucp);
void makecontext(ucontext_t *ucp, void *func(), int argc, ...);
```

`setcontext` — saute vers un contexte, un peu à la manière de `longjmp`. C'est un appel sans retour possible (puisqu'on ne sauvegarde pas le contexte courant).

`getcontext` — sauvegarde le contexte courant pour pouvoir y revenir, un peu à la manière de `setjmp`.

`swapcontext` — changement de contexte. Cette fonction provoque un saut vers le contexte `ucp` mais sauvegarde auparavant le contexte courant dans `oucp` (pour un retour ultérieur).

`makecontext` — construit un nouveau contexte. En réalité, on procède toujours par modification d'un contexte existant préalablement obtenu par `getcontext`. En pratique, pour créer un nouveau contexte, on utilise la séquence d'instructions suivante :

```
ucontext_t ucp;           /* le nouveau contexte */
getcontext(&ucp);         /* on copie le contexte courant */
ucp.uc_stack.ss_size = 64*1024;
                          /* taille de la pile : on choisit 64ko */
ucp.uc_stack.ss_sp = malloc(ucp.uc_stack.ss_size);
                          /* on alloue la nouvelle pile */
ucp.uc_stack.ss_flags = 0;
                          /* réinitialisation des flags de signaux */
ucp.uc_link = NULL;       /* fonction où se brancher quand
                           * ma_fonction sera terminée ; NULL pour
                           * une destruction du contexte. */
makecontext(&ucp, &ma_fonction, 1, &mon_argument);
                          /* mise à jour du contexte sur la nouvelle
                           * pile et spécification du point d'entrée */
ucontext_t ici;           /* on va y sauvegarder le contexte courant */
swapcontext(&ici, &ucp); /* hop, le grand saut ! */
```

Pour de plus amples explications sur l'utilisation de ces fonctions, reportez-vous aux pages de manuel du système et à la section 23.4 du manuel de la GNU C library disponible en ligne ou par la commande `info libc`.

2. Construction d'un ordonnanceur coopératif

À l'aide des primitives `ucontext`, nous voulons construire un ordonnanceur de *threads* coopératif rudimentaire. Nous appelons notre bibliothèque de *threads* `mythread`. L'objectif de ce TP est de parvenir à une version opérationnelle des fonctions suivantes :

threads — `mythread_create`, `mythread_join`, `mythread_yield`;

mutex — `mythread_mutex_init`, `mythread_mutex_lock`, `mythread_mutex_unlock`;

conditions — `mythread_cond_init`, `mythread_cond_wait`, `mythread_cond_signal`.

Note : le lecteur attentif aura remarqué que puisqu'il s'agit d'un ordonnancement *coopératif* sans préemption, il n'y a pas besoin de *spinlock*, ni dans l'ordonnanceur, ni dans les *mutex*.

Note 2 : puisqu'il n'y a pas de préemption, nous faisons un ordonnancement *par nécessité*, c'est-à-dire que les seuls appels qui provoqueront éventuellement un changement de contexte sont `join` (attente d'un *thread* pas prêt), `mutex_lock` (dans le cas où le verrou est pris), et `cond_wait`.

Méthode d'approche

La structuration de base de notre ordonnanceur de *threads* s'articulera autour des entités suivantes :

types — un *thread* est représenté par une structure de descripteur de *thread*, qui contient son `ucontext`, son état (prêt, en attente sur un `join`, en attente sur un mutex, etc. — faire un type énuméré ou des `#define`) et d'autres informations à déterminer. Le type `mythread_t` peut par exemple être un pointeur vers une telle structure.

listes de *threads* — à tout moment, un *thread* est dans une et *une seule liste* de *threads*. Ces listes sont :
- une liste globale de *threads* prêts, c'est-à-dire les *threads* qui ne sont pas en attente. Nous la gérons en FIFO : quand on veut donner la main à un *thread*, nous le prenons en tête de liste ; quand un *thread* passe en état « prêt », nous l'insérons en queue de liste.
- une liste de *threads* en attente sur un *join*, soit globale, soit localisée dans chaque structure de descripteur de *thread* (i.e. chaque *thread* « sait » qui attend sa terminaison).
- des listes de *threads* en attente sur le *mutex*, dans chaque *mutex* (comme vu en cours).
- idem pour les attentes sur conditions.

gestion du démarrage — le *thread* principal n'étant pas comme les autres (il n'est pas connu de notre bibliothèque de *threads* car il démarre avant que notre bibliothèque ne s'initialise), nous allons nous en passer. Pour plus de simplicité, l'initialisation de notre bibliothèque de *threads* créera un nouveau *thread* et ne rendra jamais la main au `main`.

Ordonnanceur simple

Dans un premier temps, nous construisons un ordonnanceur doté uniquement des fonctions `create`, `join`, `self` et `yield`.
- Mettre en place les structures de base de l'ordonnanceur de *threads* : définition du type `mythread_t`, liste des *threads* prêts, liste des *threads* en attente (seul le `join` peut provoquer une attente, pour l'instant).
- Comment gère-t-on le `mythread_self` ? Indication : à tout moment, un seul *thread* à la fois est actif...
- Comment gère-t-on le `mythread_join` ? Indication : la fonction que vous passez à `makecontext` n'est pas forcémment celle qui a été passée à `mythread_create` par l'utilisateur.
- Testez avec votre *quicksort multi-thread* du TP 2 ! Note : il est normal de ne pas avoir d'accélération puisque nos *threads* n'introduisent pas de concurrence (seulement un entrelacement).

Primitives de synchronisation

Nous ajoutons maintenant la gestion des mutex et conditions.
- Définir les types `mythread_mutex_t` et `mythread_cond_t`.
- Écrire les fonctions correspondantes pour les mutex et conditions. Est-il possible que tous les *threads* soient en attente en même temps ? À quelle situation celà correspond-il ?
- Testez vos *threads* avec les programmes 2 et 3 du TP 1. Testez avec votre *quicksort* à tâches du TP 3.
- Mesurez les performances de vos *threads* avec le programme 4 du TP 1. Comparez avec les performances de `pthread`. Alors ?

3. Pour les plus téméraires...

Que faut-il changer pour transformer votre ordonnanceur coopératif en ordonnanceur préemptif ?

Pour construire un spinlock, on pourra utiliser le `testandset` en assembleur pour processeur x86 :

```
unsigned int testandset(volatile unsigned int *val)
{
  unsigned ret;
  __asm__ __volatile__(
       "lock xchgl %0, %1"
       : "=q"(ret), "=m"(*val)
       : "0"(1), "m"(*val)
       : "memory");
  return ret;
}
```

4. Question subsidiaire

Que faudrait-il ajouter et/ou modifier dans votre ordonnanceur *coopératif* pour implémenter un mécanisme similaire aux *générateurs* de Python (cf `http://docs.python.org/tut/node11.html`, section 9.10), lui même basé sur la notion de co-routine (cf `http://en.wikipedia.org/wiki/Coroutines` sur le sujet, par exemple) ?