Ma librairie de gestion de threads

Objectifs du TP :

1. Écrire un ordonnanceur de threads coopératifs en utilisant les primitives ucontext des threads *Systems V* ;
2. Tester cet ordonnanceur avec les programmes des TP précédents.

1 Les contextes *System V*.

Sur la plupart des systèmes UNIX, l'API `ucontext` permet de gérer des contextes pour créer des threads. 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, car on ne sauvegarde pas le contexte courant.

getcontext - sauvgarde 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 par copie d'un contexte existant et modification de cette copie. 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 */
                                /* on alloue la nouvelle pile */
ucp.uc_stack.ss_sp = malloc(ucp.uc_stack.ss_size);
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 ! */
```

Question 1.1. *Complétez le programme /home/mgallet/enseignement/ASR2/TP5/changement.c pour vous habituer aux changements de contexte.*

Pour de plus amples explications sur l'utilisation de ces fonctions, reportez-vous aux pages de manuel du système.

2 Construction d'un ordonnanceur coopératif

À l'aide des primitives ucontext, nous voulons construire un ordonnanceur de threads coopératif rudimentaire et non préemptif. 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`.

2.1 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 (pas connu de notre bibliothèque de *threads*), 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.

2.2 Ordonnanceur simple

Dans un premier temps, nous construisons un ordonnanceur doté uniquement des fonctions `create`, `join`, `self` et `yield`.

Question 2.1. *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).*

Question 2.2. *Comment gère-t-on le* `mythread_self` *? Indication : à tout moment, un seul thread à la fois est actif...*

Question 2.3. *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.*

Question 2.4. *Testez avec vos programme des TPs précédents !*

2.3 Primitives de synchronisation

Nous ajoutons maintenant la gestion des mutex et conditions.

Question 2.5. *Définir les types* `mythread_mutex_t` *et* `mythread_cond_t`*. Écrire les fonctions correspondantes pour les mutex et conditions. Est-il possible que*

Question 2.6. *tous les threads soient en attente en même temps ? À quelle situation cela correspond-il ?*

Question 2.7. *Mesurez les performances de vos threads. Comparez avec les performances de* `pthread`*. Alors ?*

3 Aller plus loin

3.1 Préemption

Question 3.1. *La bibliothèque repose sur une gestion coopérative. Expliquez ce que cela signifie.*

Question 3.2. *Quel est l'intérêt d'une gestion préemptive ? Quels mécanismes peut-on envisager pour la mettre en place. Implémentez un tel mécanisme.*

3.2 Ordonnancement

Question 3.3. *Définissez et implémentez une politique d'ordonnancement autre que* `FIFO`*. Vous pourrez vous inspirer de la page de* `man` *de* `sched.h`*.*

3.3 Comparaison avec la pthread

Question 3.4. *Quelle est la différence fondamentale entre votre bibliothèque de threads et la pthread ? Donnez des illustrations contrètes de possibles inconvénients. Existe-t-il un moyen de les contourner au moins partiellement ? Implémentez, expérimentez . . .*

3.4 Documentation

Documentez votre bibliothèque avec une/des page(s) de `man`.