TP2_PBT

Night

February 2023

1 Fonctionnement d'une bibliothèque de niveau utilisateur : GnuPth

Avantages

- Simple à implémenter car en espace utilisateur
- performant
- portable , facilité de développement

Inconvénients

- peu adapté au SMP et multicoeur
- mauvais avec les appels systèmes (blocages) car l'ensemble n'existe que sur un seul processus noyau

2 Fonctionnement d'une bibliothèque de niveau système : Pthread

Avantages

- adapté au SMP et multicoeur
- performant notamment avec les appels systèmes

Inconvénients

- très peu portable, difficile à implémenter
- coût plus élevé

3 Évaluation des bibliothèques Pthread et GnuPth sur SMP (processeur multi-coeur)

3.1 q3

On peut écrire un petit code effectuant des opérations simples sur un tableau et constater les performances par rapport au nombre de coeurs utilisés.

3.2 q4

Voici ici un exemple du calcul de la somme des éléments (passés comme arguments dans le struct) pour un tableau de taille N sur NB_THREADS*N threads avec GnuPTH (sans clock ici):

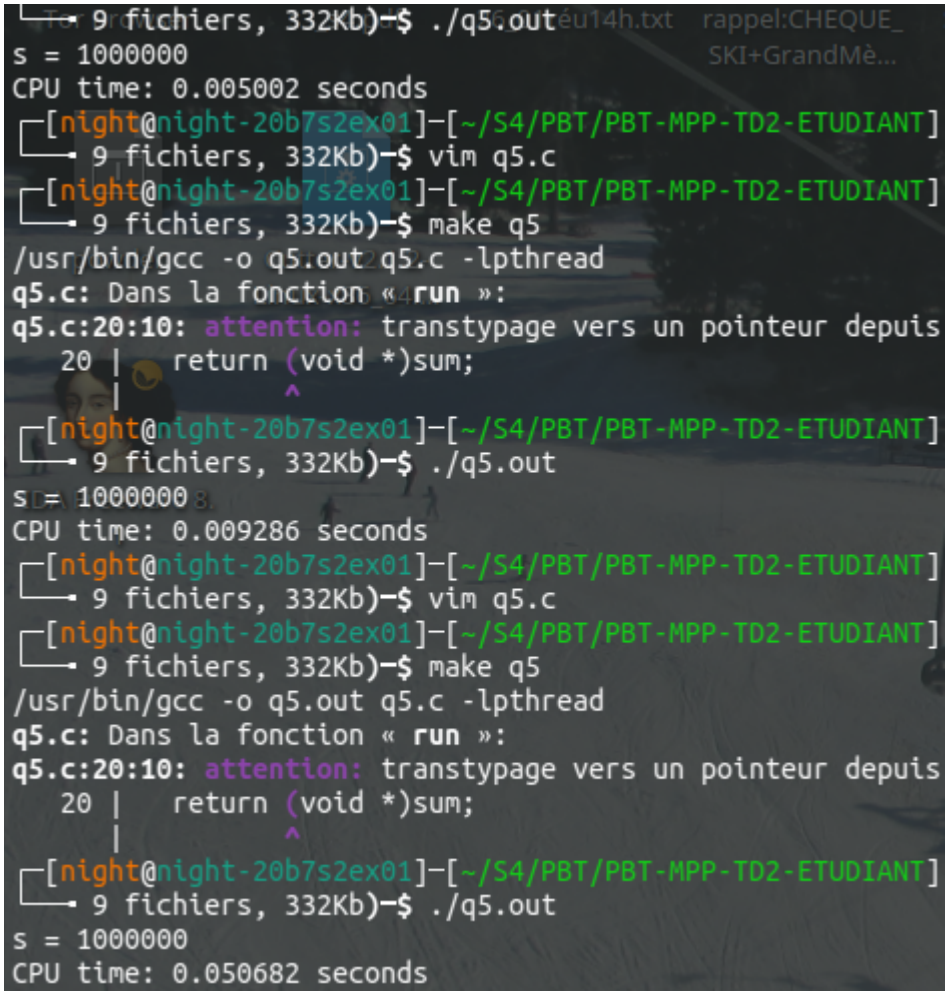
```c
#include <pth.h>
#include <stdio.h>

#define NB_THREADS 10
#define N NB_THREADS * 100000000
int tab[N];

struct args {
  int start;
  int end;
};

void * run(void * arg) {
  int sum = 0;
  struct args a = *(struct args *)arg;
  for (int i = a.start; i < a.end; i++)
    sum += tab[i];
  return (void *)sum;
}

int main(int argc, char ** argv) {

  if (!pth_init()) return 1;

  for (int i = 0; i < N; i++)
    tab[i] = 1;


  pth_t pids[NB_THREADS];
  struct args args[NB_THREADS];

  for (int i = 0; i < NB_THREADS; i++)
    args[i] = (struct args){ .start = i * N, .end = (i + 1) * N };

  for (int k = 0; k < NB_THREADS; k++)
    pids[k] = pth_spawn(PTH_ATTR_DEFAULT, run, &(args[k]));

  int res[NB_THREADS] = { 0 };
  for (int k = 0; k < NB_THREADS; k++)
    pth_join(pids[k], (void **)&res[k]);

  pth_kill();

  int s = 0;
  for (int i = 0; i < NB_THREADS; i++)
    s += res[i];

  printf("s = %d\n", s);

  return 0;
}
```

Le **makefile** compile les questions 4, 5, 7 et 8, pour GnuPth il faut charger la lib dynamique que l'on a installée:

```
export LD_LIBRARY_PATH=pth-2.0.7/install/lib/
make q4
```

3.3 q5

`make q5`



- GnuPth est meilleur pour NB_THREADS petit
- **Pthread** est meilleur pour **NB_THREADS grand**

3.4 q6

Sachant que les différents threads s'exécuteront à la suite dans une bibliothèque utilisateur / en parallèle dans une bibliothèque systèmen, on peut utiliser **sleep** pour évaluer la gestion des appels bloquants.

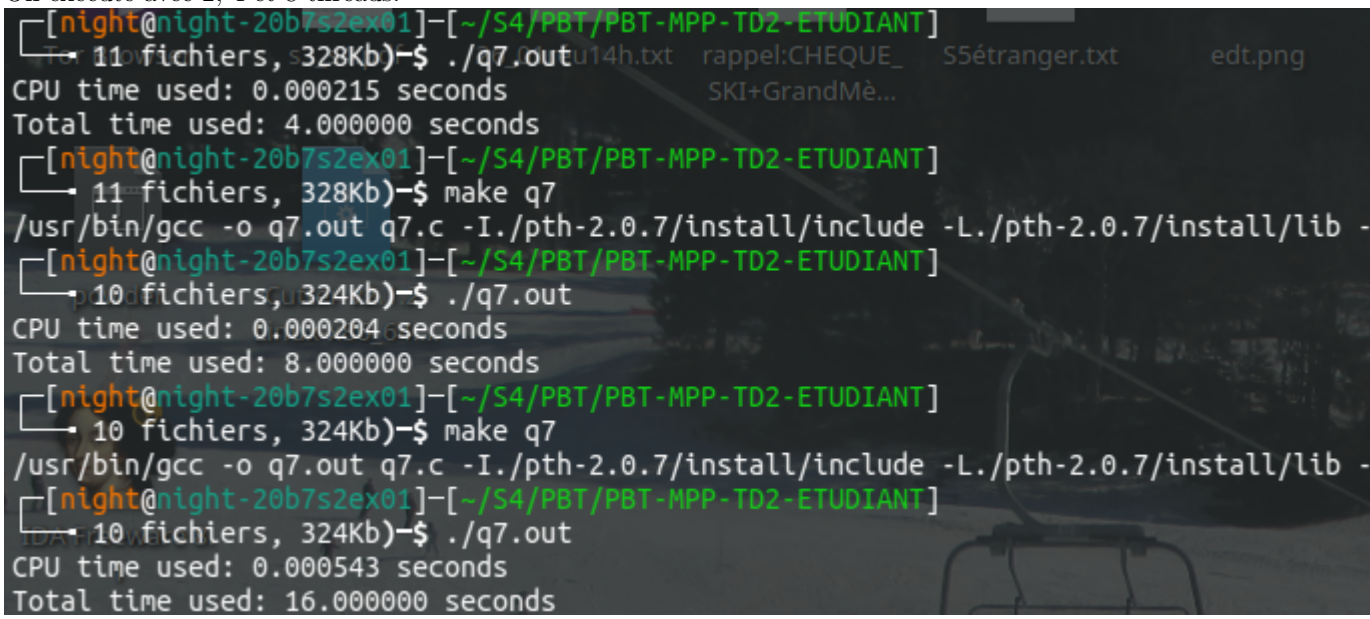
3.5 q7

On éxécuter maintenant sleep pour chaque thread:

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

#define NB_THREADS 2

void * run(void * arg) {
  sleep(2);
  return NULL;
}
```

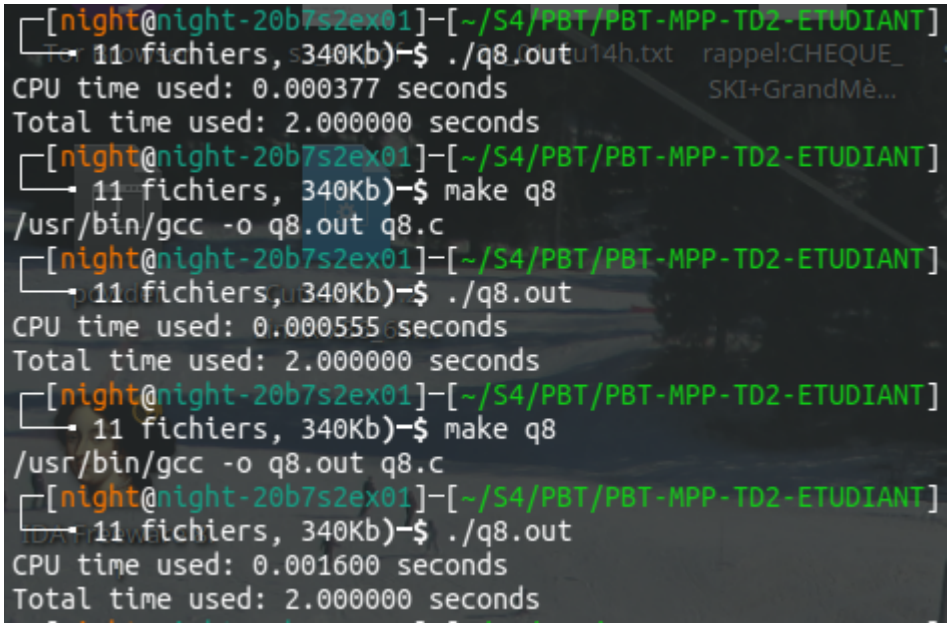
```c
int main(int argc, char ** argv) {

  if (!pth_init()) return 1;

  pth_t pids[NB_THREADS];
  for (int k = 0; k < NB_THREADS; k++)
    pids[k] = pth_spawn(PTH_ATTR_DEFAULT, run, NULL);

  for (int k = 0; k < NB_THREADS; k++)
    pth_join(pids[k], NULL);

  pth_kill();

  return 0;
}
```

On éxécute avec 2, 4 et 8 threads:



3.6 q8

On éxécute avec 2, 4 et 8 threads:



Le temps cpu est un peu plus faible mais le temps d'éxécution est doublé pour GnuPth alors que ce dernier reste constant à 2s pour Pth pour chaque éxécution; ce qui est normal car GnuPth n'utilise qu'un seul processus noyau (bibliothèque utilisateur).

4 Fonctionnement d'une bibliothèque mixte : mthread

4.1 q9

Avantages

- adapté au SMP et multicoeur
- performant

Inconvénients

- peu flexible, difficile à implémenter, nécessiste un support noyau et ordonnancement utilisateur
- parfois limité par la partie utilisateur sur les appels systèmes

4.2 1: Découverte du code de la bibliothèque mthread

4.2.1 q10

L'ordonnanceur se trouve dans le fichier **mthread.c lignes 148 à 205** . Il s'agit de la procédure:

```
void __mthread_yield(mthread_virtual_processor_t * vp) .
```

4.2.2 q11

L'ordonnanceur récupère le thread courant ainsi que le suivant avec mthread_remove_first sur la liste des threads prêts du processeur virtuel.

Si la macro TWO_LEVEL est définie et si la liste des threads prêts est vide, la fonction mthread_work_take est appelée pour récupérer un travail en attente à partir de la file d'attente des travaux de bas niveau.

Si le processeur virtuel dispose d'un thread prêt à exécuter qui a été préempté lors d'une exécution précédente, le thread est inséré en fin de liste des threads prêts en appelant la fonction mthread_insert_last.

Si le thread courant n'est pas idle, il est remis dans la file de l'ordonnanceur. Si le thread suivant est NULL , le prochain état devient idle.

Ensuite, si le thread suivant n'est pas NULL et différent du thread courant, ils sont inversés avec mthread_mctx_swap. Le processeur virtuel est ensuite récupéré et libéré si nécessaire.

4.2.3 q12

On trouve dans mthread.c les fonctions:

- void mthread_list_init(mthread_list_t *list) (**ligne 27**)
- void mthread_insert_first(struct mthread_s *item, mthread_list_t *list) (**ligne 40**)
- void mthread_insert_last(struct mthread_s *item, mthread_list_t *list) (**ligne 57**)
- mthread_s *mthread_remove_first(mthread_list_t *list) (**ligne 75**)

4.2.4 q13

Pour insérer un thread prêt de la liste, on appelle:

```
mthread_insert_last(/*nouveau thread*/, &(vp->ready_list)); .
```

4.2.5 q14

Pour bloquer un thread, on utilise le membre status de struct mthread_s:

```
current->status == BLOCKED
```

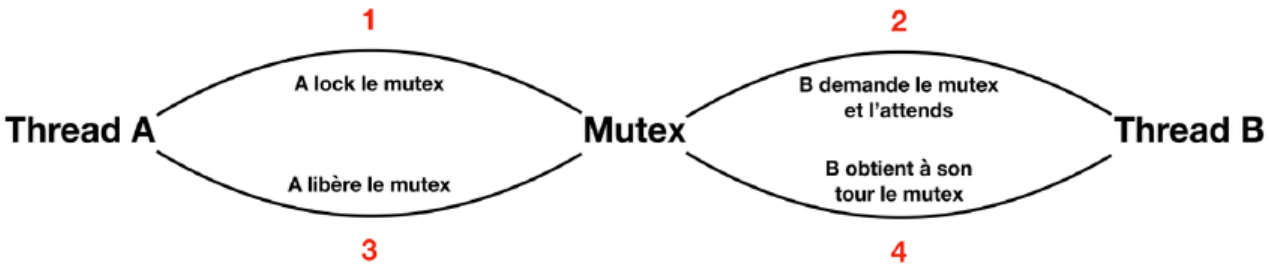
4.3 2: Mutex

4.3.1 q15

Le mutex permet aux threads l'accès à une ressource partagée de manière sécurisée et synchronisée (pas de race condition).

Lorsqu'un thread souhaite accéder à la ressource, il acquiert le mutex en appelant **pthread_mutex_lock**, ce qui lui permet d'accéder à la ressource en toute sécurité.

Lorsqu'il a fini d'utiliser la ressource, il libère le mutex en appelant **pthread_mutex_unlock**, ce qui permet à un autre thread d'acquérir le mutex et d'accéder à la ressource.



4.4 3 Mise en place des mutex dans mthread

Les fonctions ont déjà été implémentées dans mthread_mutex.c et retournent 0 en cas de bon fonctionnement.

4.4.1 q16

Voir ligne 32

mthread_mutex_init vérifie si les arguments sont valides, initialise le nombre de threads en attente et le verrou à 0, ce qui signifie que le mutex est déverrouillé.

Enfin, la fonction appelle une fonction __mthread_ensure_list_init pour initialiser une liste de threads en attente associée au mutex.

4.4.2 q17

Voir ligne 107

mthread_mutex_lock vérifie si les arguments sont valides, ensuite la fonction verrouille le spinlock associé au mutex, afin de s'assurer que l'opération de verrouillage est atomique et ne sera pas interrompue par une autre thread. Si nb_thread est égal à 0, cela signifie que le mutex n'est pas déjà verrouillé par une autre thread. Dans ce cas, la fonction met à jour nb_thread à 1, déverrouille le spinlock et retourne immédiatement.

Si nb_thread est différent de 0, cela signifie que le mutex est déjà verrouillé par une autre thread. Dans ce cas, la fonction récupère l'identifiant de la thread appelante (self), insère cette thread dans la liste associée au mutex, passe l'état de cette thread à "BLOCKED" (bloquée), et met à jour un pointeur vers le spinlock associé au mutex dans la structure de la virtual processor courante. Enfin, la fonction invoque mthread_yield pour passer la main à un autre thread.

Enfin, la fonction journalise le fait que le mutex a été acquis,

4.4.3 q18

Voir ligne 138

mthread_mutex_unlock permet de déverrouiller un mutex en libérant l'accès à une section critique. Elle commence par vérifier si le pointeur de mutex est valide, puis verrouille le mutex. Ensuite, si la liste des threads bloqués sur ce mutex n'est pas vide, elle retire le premier thread de la liste, le marque comme "en cours d'exécution", et l'insère dans la liste des threads prêts. Sinon, elle met le compteur de threads bloqués à 0. Enfin, elle déverrouille le mutex et retourne 0 pour indiquer que l'opération s'est déroulée avec succès.

4.4.4 q19

Voir ligne 68

La fonction mthread_mutex_destroy() est utilisée pour détruire un mutex qui n'est plus nécessaire. Elle commence par vérifier si le pointeur de mutex fourni n'est pas NULL. Ensuite, elle s'assure que la liste du mutex est correctement initialisée. Elle verrouille ensuite le mutex et vérifie s'il est occupé par un thread. Si c'est le cas, elle renvoie une erreur EBUSY. Sinon, elle libère la mémoire allouée pour la liste du mutex et déverrouille le mutex. Enfin, elle enregistre la destruction du mutex dans les logs et renvoie 0.

4.4.5 q20

Voir ligne 91

La fonction mthread_mutex_trylock essaie de verrouiller un mutex, mais retourne immédiatement avec une erreur EBUSY si le mutex est déjà verrouillé.

4.4.6 q21

4.4.7 q22