SY40 TP8 /yann derré

1) prg1.c

Voici la description complétée des différentes étapes d'exécution pthread.

```
  printf("thread 2 : iGlobal = %d\n",iGlobal);
  printf("thread 2 : terminaison\n");
  return (void*) iGlobal;
  }

int main(int argc,char* argv[],char* arge[])
  {
  pthread_t thread2;       //declaration d'un thread
  int param[]={0,1,MarqueurFinArg};
  int codeRetour;
  if (pthread_create(&thread2,NULL,f,param)==-1) { //creation du thread
      printf("pb pthread_create\n"); exit(1);
  }
  printf("thread 1 : thread 2 created,    iGlobal = %d\n",iGlobal);

  sleep(1);

  printf("thread 1 : thread2 end, iGlobal = %d\n",iGlobal);
  pthread_join(thread2,(void**) &codeRetour);
  printf("thread 1 : thread2 joined, iGlobal = %d   codeRetour = %d\n",iG
  }
```

```
(base) yannderre@MacBook-Pro-2 [12:55:41] [~/Downloads/TP8/sources]
-> % ./prg1
thread 1 : thread 2 created,    iGlobal = 1
thread 2 : iGlobal = 1  arg[0] = 0  arg[1] = 1
thread 1 : thread2 end, iGlobal = 2
thread 2 : iGlobal = 2
thread 2 : terminaison
thread 1 : thread2 joined, iGlobal = 2   codeRetour = 2
```

2) prg2.c

Le programme 2 est modifié pour gérer à la fois l'incrémentation multithreadée et gère l'exclusion mutuelle via l'ajout d'un mutex.

```
int iGlobal = 1;
pthread_mutex_t mutex;
```

La fonction f d'incrémentation a été modifiée pour verrouiller le mutex avant l'incrémentation et le déverrouiller lorsqu'elle se termine :

```
void *f(void *arg)
{
    int threadnum = *(int *)arg;
    printf("thread %d : iGlobal = %d\n", threadnum, iGlobal);
    pthread_mutex_lock(&mutex);
    iGlobal++;
    sleep(2);
    pthread_mutex_unlock(&mutex);
    printf("thread %d : iGlobal++ = %d\n", threadnum, iGlobal);
    return NULL;
}
```

La fonction main() se charge d'initialiser le nombre de threads dynamiquement, d'initialiser le mutex utilisé, et de créer chaque thread via pthread exécutant la fonction f d'incrémentation. On passe également le numéro du thread en argument à la fonction f pour l'affichage dans le printf().

Chaque thread se termine via pthread_join, on affiche le iGlobal final et on detruit le mutex et le tableau thread alloué.

```c
int main(int argc, char *argv[], char *arge[])
{
    if (argc < 2)
    {
        printf("Usage: %s <num_threads>\n", argv[0]);
        return 1;
    }
    int num_threads = atoi(argv[1]);
    pthread_t *threads = malloc(num_threads * sizeof(pthread_t));
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < num_threads; i++)
    {
        int *threadnum = (int *)malloc(sizeof(int));
        *threadnum = i+1;
        if (pthread_create(&threads[i], NULL, f, threadnum) == -1)
        { // creation du thread
            printf("pb pthread_create\n");
            exit(1);
        }
    }
    for(int i = 0; i < num_threads; i++)
    {
        if(pthread_join(threads[i], NULL) == -1)
        {
            printf("pb pthread_join\n");
            exit(1);
        }
    }

    printf("final iGlobal = %d\n", iGlobal);

    pthread_mutex_destroy(&mutex);
    free(threads);
    return 0;
}
```

La fonction main chargée d'initialiser les mutex, threads, et affiche le résultat final.

3) prg4.c

Le programme initialise une variable *INT_QQUE*, correspondant à la valeur maximale que l'int peut prendre sur le système (définit par les macros de l'implémentation POSIX).

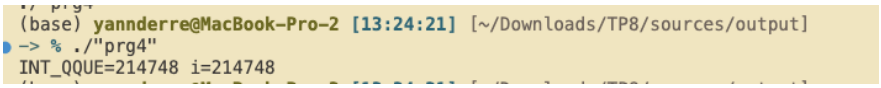
INT_QQUE vaut donc (2147483647/10000) soit 214748 lorsque arrondi. Le but du programme est de synchroniser 2 threads pour incrémenter la variable i, et arrêter le programme losque i est égal à *INT_QQUE*. Pour éviter une race condition entre l'incrémentation et la vérification de la variable, il est nécessaire d'utiliser les mutex + cond_wait pour synchroniser les threads.
Pour cela, le programme possède 2 threads, le main thread et le second thread. Le premier se chargera d'initialiser le thread2 via pthread_init qui executera codeThread2(), puis exécute codeThread1.

Le main thread initialise également les pthread_cond et mutex, qui sont nécessaire pour contrôler la synchronisation de la variable i.

codeThread1 appelle ***pack1_attendre***, qui verouille le mutex, vérifie si *i* vaut *INT_QQUE*, attend tant que ça n'est pas le cas et débloque le mutex.

codeThread2, lui, va appeler ***pack1_signalerUn***, qui verouille le mutex, incrémente *i* et vérifie si *i* vaut *INT_QQUE*, débloque le mutex, et recommence tant que la condition n'est pas remplie. Le signal de fin de l'itération s'effectue par ***pthread_cond_signal***, qui va envoyer le signal à l'attente de ***pack1_attendre***.

Une fois que ***pack1_attendre*** est dévérouillé, ***codeThread1*** peut finir de s'exécuter et imprime la valeur de *i* et *INT_QQUE*.



4) Dining philosopher’s problem

```c
int main()
{
    srand(time(NULL));

    pthread_t threads[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
    {
        philosopher_ids[i] = i;

        pthread_mutex_init(&philosophers[i].fork_mutex, NULL);
        pthread_cond_init(&philosophers[i].forks_condition, NULL);
        philosophers[i].forks_available = 1;
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
    {
        if (pthread_create(&threads[i], NULL, philosopher_process, &philosopher_ids[i]) != 0)
        {
            printf("Error creating thread for philosopher %d\n", i);
            return 1;
        }
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
    {
        pthread_join(threads[i], NULL);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
    {
        pthread_mutex_destroy(&philosophers[i].fork_mutex);
        pthread_cond_destroy(&philosophers[i].forks_condition);
    }

    return 0;
}
```

Pour chaque philosopher, j’initialise mon mutex et condition, et je crée un thread. En fin d’éxecution, je retourne au main thread via pthread_join, et je détruis les mutex.
Chaque pthread créé exécute cette fonction (ici tourne à l’infini et retire l’intérêt de nettoyer nos fuites de mémoire en fin d’exec, mais bonne pratique de le faire quand même)

```c
void *philosopher_process(void *arg)
{
    int philosopher_id = *(int *)arg;

    while (1)
    {
        think(philosopher_id);

        take_forks(philosopher_id);

        eat(philosopher_id);

        put_forks(philosopher_id);
    }

    return NULL;
}
```

La logique de répartiiton des couverts gauches et droit va se faire entre ***take_fork() et put_fork().***

```c
void take_forks(int philosopher_id) {
    int left_philosopher_id = philosopher_id;
    int right_philosopher_id = (philosopher_id + 1) % NUM_PHILOSOPHERS;

    if (philosopher_id % 2 == 0) {
        left_philosopher_id = (philosopher_id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS;
        right_philosopher_id = philosopher_id;
    }

    pthread_mutex_lock(&philosophers[left_philosopher_id].fork_mutex);
    while (philosophers[left_philosopher_id].forks_available == 0) {
        pthread_cond_wait(&philosophers[left_philosopher_id].forks_condition, &philosophers[left_philosopher_id].fork_mutex);
    }

    pthread_mutex_lock(&philosophers[right_philosopher_id].fork_mutex);
    while (philosophers[right_philosopher_id].forks_available == 0) {
        pthread_cond_wait(&philosophers[right_philosopher_id].forks_condition, &philosophers[right_philosopher_id].fork_mutex);
    }

    philosophers[left_philosopher_id].forks_available = 0;
    philosophers[right_philosopher_id].forks_available = 0;
    printf("Philosopher %d takes the forks.\n", philosopher_id);
}
```

Ici, chaque philosophe va verrouiller son mutex et le mutex du philosophe immédiat à droite (gauche si nombre paire de philosophes), philosophe 2 va verrouiller philosophe 3, philosophe 4 va verrouiller philosophe 0 (à l'aide du modulo).

Si les couverts ne sont pas disponibles, le thread attend. Cette fonctionnalité est ajoutée pour éviter une attente active présente si on utilise uniquement lock et unlock pour contrôler l'exécution.

```c
void put_forks(int philosopher_id) {
    int left_philosopher_id = philosopher_id;
    int right_philosopher_id = (philosopher_id + 1) % NUM_PHILOSOPHERS;

    if (philosopher_id % 2 == 0) {
        left_philosopher_id = (philosopher_id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS;
        right_philosopher_id = philosopher_id;
    }

    philosophers[left_philosopher_id].forks_available = 1;
    philosophers[right_philosopher_id].forks_available = 1;
    printf("Philosopher %d puts the forks back.\n", philosopher_id);

    pthread_mutex_unlock(&philosophers[left_philosopher_id].fork_mutex);
    pthread_mutex_unlock(&philosophers[right_philosopher_id].fork_mutex);

    pthread_cond_signal(&philosophers[left_philosopher_id].forks_condition);
    pthread_cond_signal(&philosophers[right_philosopher_id].forks_condition);

    sleep(rand() % 2);
}
```

put_forks() fonctionne de manière analogue, il déverouille les mutex, et signale aux threads en attente de continuer.

```
Philosopher 2 is eating.
Philosopher 2 puts the forks back.
Philosopher 2 is thinking.
Philosopher 2 takes the forks.
Philosopher 2 is eating.
Philosopher 2 puts the forks back.
Philosopher 1 is thinking.
Philosopher 1 takes the forks.
Philosopher 1 is eating.
Philosopher 2 is thinking.
Philosopher 0 puts the forks back.
Philosopher 0 is thinking.
Philosopher 4 takes the forks.
Philosopher 4 is eating.
^C
```

Voici l'output obtenu, on observe que philosopher 2 ne mange pas avec 1 ou 3. (0 et 2 puis, 1 et 4 mangent)