UNIVERSITE IBN ZOHR
FACULTE DES SCIENCES
AGADIR

جامعة ابن زهر
كليـــة العلـــوم
أكاديـر

Compte RenduTP1 : Programmation Temps Réel

Filières:
Ingénierie Informatique et Systèmes Embarqués

Nom :
NABLI Amine
OUARRACH Oualid

Encadré par :
M. OUKDACH Yassine

Année Universitaire 2023/2024

Exercice 1

1. Calculer pour chaque tâches le facteur d'utilisation du processeur :

$$U = C/P$$

T0:

U = 2/6 = 0.333

T1:

U = 3/8 = 0.375

T2:

U = 4/24 = 0.166

2. Calculer pour chaque tâches le facteur de charge du processeur

$$Ch = C/D$$

On a des taches périodiques à échéance sur requête, alors D = P, ce qui implique Ch = C/P = U

T0:

ch = 0.333

T1:

ch = 0.375

T2:

ch = 0.166

3. Calculer pour chaque tâches le temps de réponse.

$$TRi = fi - ri$$

Pour la tache 0 :

Période 1 : 2-0 =2
Période 2: 8-6 =2
Période 3 : 14-12=2
Période 4 : 20-18=2

Pour la tache 1 :

Période 1 : 5-0=5
Période 2 : 11-8=3
Période 3 : 21-16=5

Pour la tache 2 :

Période 1 : 16-0 =16

4. Calculer pour chaque tâches la laxité nominal

$$L = D - C$$

Pour T0 : 6 - 2 = 4
Pour T1 : 8 - 3 = 5
Pour T2 : 24 - 4 = 20

5. Calculer pour chaque tâches la gigue de release relative, la gigue de release absolue, la gigue de fin relative et la gigue de fin absolue.

La gigue de release relative est

$$RRJ_i = \max_j | (s_{i,j} - r_{i,j}) - (s_{i,j} - 1 - r_{i,j} - 1) |$$

Pour T0 :

RRJ0 = (6 – 6) – (0 - 0) = 0
RRJ0= (12 – 12) – (6 - 6) = 0
RRJ0= (18 – 18) – (12 - 12) = 0
Donc RRJ0= 0

Pour T1 :

RRJ1 =(8 – 8) – (2 - 0) = 2
RRJ1 =(16 – 16) – (8 - 8) = 0
Donc RRJ1= 2

Pour T2:

RRJ2 = 5 – 0 = 5
RRJ2= 5

La gigue de release absolue est

$$ARJ_i = \max_j (s_{i,j} - r_{i,j}) - \min_j (s_{i,j} - r_{i,j})$$

Pour T0 :

$s_{i,j} - r_{i,j} = 0 - 0 = 0$
$s_{i,j}+1 - r_{i,j}+1 = 6 - 6 = 0$
$s_{i,j}+2 - r_{i,j}+2 = 12 - 12 = 0$
$s_{i,j}+3 - r_{i,j}+3 = 18 - 18 = 0$
Donc ARJ0 = 0

Pour T1 :

$s_{i,j} - r_{i,j} = 2 - 0 = 2$
$s_{i,j}+1 - r_{i,j}+1 = 8 - 8 = 0$
$s_{i,j}+2 - r_{i,j}+2 = 16 - 16 = 0$
Donc ARJ1 = 2 – 0 = 2

Pour T2 :

$s_{i,j} - r_{i,j} = 5 - 0 = 5$
Donc ARJ2 = 5

La gigue de fin relative est

$$RFJ_i = \max_j | (f_{i,j} - r_{i,j}) - (f_{i,j} - 1 - r_{i,j} - 1) |$$

Pour T0 :

RFJ0 = | (8 - 5) – (2 – 0) | = 1
RFJ0 = | (14 - 12) – (8 – 5) | = 1
RFJ0 = | (20 - 18) – (14 – 12) | = 1
Donc RFJ0 = 1

Pour T1 :

RFJ1 = | (11 - 8) – (5 – 0) | = 2
4RFJ0 = | (21 - 16) – (11 – 8) | = 2
Donc RFJ1 = 2

Pour T2 :

RFJ2 = |16 – 0 | = 16
Donc RFJ2 = 2

La gigue de fin absolue est

$$AFJ_i = \max_j (f_{i,j} - r_{i,j}) - \min_j (f_{i,j} - r_{i,j})$$

Pour T0 :

$f_{i,j} - r_{i,j} = |(11 - 8) - (5 - 0)| = 2$

$f_{i,j+1} - r_{i,j+1} = 8 - 6 = 2$

$f_{i,j+2} - r_{i,j+2} = 14 - 12 = 2$

$f_{i,j+3} - r_{i,j+3} = 20 - 18 = 2$

Donc AFJ0 = 2 – 2 = 0

Pour T1 :

$f_{i,j} - r_{i,j} = (5 - 0) = 5$

$f_{i,j+1} - r_{i,j+1} = 11 - 8 = 3$

$f_{i,j+2} - r_{i,j+2} = 21 - 16 = 5$

Donc AFJ1 = 5 – 3 = 2

Pour T2 :

$f_{i,j} - r_{i,j} = |(11 - 8) - (5 - 0)| = 2$

$f_{i,j+1} - r_{i,j+1} = 16 - 0 = 16$

Donc AFJ2 = 16

Exercice 2

Ex2

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *func(void *arg) {
  //On fait un cast comme char a l'argument et on l'affiche
  char *msg = (char *) arg;
  printf("Message = %s\n", msg);
  return NULL;
}

int main(int argc, char *argv[]) {
  //On cree une thread et on lui passe une chaine de caractere comme argument
  pthread_t t;
  char *msg = "Threads are awesome!";
  if (pthread_create(&t, NULL, func, msg) != 0) {
    perror("thread creation error");
    return EXIT_FAILURE;
  }
  //Le programme main attend la fin d'execution du thread
  if(pthread_join(t, NULL) !=0){
        perror("Error");
        EXIT_FAILURE;
    }
  return EXIT_SUCCESS;
}
```

```
Message = Threads are awesome!
```

Exercice 3

Ex3

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *Tachel (void *arg) {
  int i=0;
  while(i<5){
    printf("Execution de Tache 1\n");
    sleep (1);
    i++;
  }
  return NULL;
}


void *Tache2 (void *arg) {
  int j=0;
  while (j<3){
    printf("Execution de Tache 2\n");
    sleep (2);
    j++;
  }
  return NULL;
}
```

```c
//Test 1
int main(int argc, char *argv[]){
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, Tachel, NULL);
  pthread_create(&thread2, NULL, Tache2, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return EXIT_SUCCESS;
}
```

```
Execution de Tache 2
Execution de Tache 1
Execution de Tache 1
Execution de Tache 2
Execution de Tache 1
Execution de Tache 1
Execution de Tache 2
Execution de Tache 1
```

1 - On remarque que les deux taches sont exécutées parallèlement, et l'ordonnanceur bascule chaque fois entre les deux taches

```c
//Test 2
int main(int argc, char *argv[]){
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, Tachel, NULL);
  pthread_join(thread1, NULL);
  pthread_create(&thread2, NULL, Tache2, NULL);
  pthread_join(thread2, NULL);
  return EXIT_SUCCESS;
}
```

```
Execution de Tache 1
Execution de Tache 1
Execution de Tache 1
Execution de Tache 1
Execution de Tache 1
Execution de Tache 2
Execution de Tache 2
Execution de Tache 2
```

2 - On remarque que la tache 1 est exécuté entièrement puis on crée et lance la tache 2 après.

3 - Test 1, on crée le thread1 et le thread2 et on les exécute parallèlement, et a l'ordonnanceur de choisir comment les ordonnancer Test 2, on crée le thread1 et on le lance jusqu'à sa fin d'exécution, puis on crée le thread2 et on le lance jusqu'à sa fin d'exécution.

Exercice 4

Ex4

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *thread_func1(void *arg) {
  printf("Thread 1: Bonjour !\n");
  return NULL;
}

void *thread_func2(void *arg) {
  printf("Thread 2: Salut !\n");
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t t1;
  pthread_t t2;

  //creation et lancement du thread 1
  if (pthread_create(&t1, NULL, thread_func1, NULL) != 0) {
    perror("thread creation error");
    return EXIT_FAILURE;
  }

  //creation et lancement du thread 2
  if (pthread_create(&t2, NULL, thread_func2, NULL) != 0) {
    perror("thread creation error");
    return EXIT_FAILURE;
  }

  //attendre la fin d'execution du thread 1
  if(pthread_join(t1, NULL) !=0){
        perror("Error");
        EXIT_FAILURE;
    }

  //attendre la fin d'execution du thread 1
    if(pthread_join(t2, NULL) !=0){
        perror("Error");
        EXIT_FAILURE;
    }

  return EXIT_SUCCESS;
}
```

```
Thread 2: Salut !
Thread 1: Bonjour !
```

Ex4

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *thread_func1(void *arg) {
  printf("Thread 1: Bonjour !\n");
  return NULL;
}

void *thread_func2(void *arg) {
  printf("Thread 2: Salut !\n");
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t t1;
  pthread_t t2;

  //creation et lancement du thread 1
  if (pthread_create(&t1, NULL, thread_func1, NULL) != 0) {
    perror("thread creation error");
    return EXIT_FAILURE;
  }

  //creation et lancement du thread 2
  if (pthread_create(&t2, NULL, thread_func2, NULL) != 0) {
    perror("thread creation error");
    return EXIT_FAILURE;
  }

  //attendre la fin d'execution du thread 1
  if(pthread_join(t1, NULL) !=0){
        perror("Error");
        EXIT_FAILURE;
    }

  //attendre la fin d'execution du thread 1
    if(pthread_join(t2, NULL) !=0){
        perror("Error");
        EXIT_FAILURE;
    }

  return EXIT_SUCCESS;
}
```

```
Thread 2: Salut !
Thread 1: Bonjour !
```

Exercice 5

```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>

#define NUM_TASKS 4

typedef struct{
    int id;
    int duree;
}PeriodicTask;

void *taskFunction(void *arg){
    PeriodicTask* task = (PeriodicTask*)arg;

    //Configurer le thread pour qu'il soit annulabe
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

    while (1){
        sleep(task->duree);
        printf("Execution de la tache %d\n",task->id);
 /*
   Point d'annulation, dans notre exemple, a chaque fois que le thread arrive a ce point
   il verifie si il a recu une demande d'annulation de la part du programme main
 */
        pthread_testcancel();
    }
    return NULL;
}
int main(){

    int periods[NUM_TASKS] = {1,2,3,4};
    PeriodicTask tasks[NUM_TASKS];
    pthread_t threads[NUM_TASKS];
    int i;

/*
  Creation des threads, en passant comme argument une structure PeriodicTask contenant l'id
  du thread et la duree de suspension du thread (sleep)
*/
    for(i = 0 ;  i < NUM_TASKS ; i++){
        tasks[i].id = i+1;
        tasks[i].duree = periods[i];
        pthread_create(&threads[i], NULL, taskFunction, (void *)&tasks[i]);
    }

    //Envoyer une demande d'annulation et attendre la fin de l'execution a tous les threads,
    for(i = 0 ; i < NUM_TASKS ; i++){
        pthread_cancel(threads[i]);
        pthread_join(threads[i],NULL);
    }
    return 0;
}
```

```
Execution de la tache 1
Execution de la tache 2
Execution de la tache 3
Execution de la tache 4
```

Exercice 6

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define TAILLE_TAB 19
#define NUM_THREADS 5
int sum = 0;
pthread_mutex_t lock;

typedef struct{
    int *debut;
    int *fin;
} PartialSumArgs;

void *sum_partial(void *args){

    PartialSumArgs *partialArgs = (PartialSumArgs *)args;
    int partialSum = 0;
    int *p;

    //on calcule la somme partiel de notre section du tableau definit par les pointeurs debut et fin.
    for (p = partialArgs->debut; p < partialArgs->fin; p++){
        partialSum += *p;
    }

/*
 mutex primitive de synchronization permet a un seul thread d'acceder a une ressource critique
 a la fois, la fonction pthread_mutex_lock permet d'acquerir le mutex qui nous permet
 d'ajouter notre somme partiel a la somme total, si le mutex est encore d'utilisation par un
 autre thread on attend jusqu'a sa liberation puis on le saisie.
*/
    pthread_mutex_lock(&lock);
    sum += partialSum;
    pthread_mutex_unlock(&lock);
}
```

Ex5

```c
int main(){
    //creation de la table des nombres
    int nombres[TAILLE_TAB];
    int i;
    for (i = 0; i < TAILLE_TAB; ++i){
        nombres[i] = i + 1;
    }

/*
 creation de les tables de threads et structue(PartialSumArgs) contenant deux pointeurs sur
le
 debut et la fin d'une section du tableau que le thread va en calculer la somme
 int range : le nombre des nombres du tableau affecter a chaque thread (sauf la derniere
 thread dans des cas speciales)
 cas speciale : (10 nombres dans le tableau et 3 thread), le thread 1 et 2 vont calculer la
 somme de trois nombre du tableau et le thread 3 va calculer la somme des quatres dernier
 elements
*/
    pthread_t threads[NUM_THREADS];
    PartialSumArgs partialSumArgs[NUM_THREADS];
    int range = TAILLE_TAB / NUM_THREADS;

    pthread_mutex_init(&lock, NULL);

    for (i = 0; i < NUM_THREADS; i++){
        partialSumArgs[i].debut = nombres + i * range;
        partialSumArgs[i].fin = nombres + ((i == NUM_THREADS - 1) ? TAILLE_TAB : (i+1)
*range);
        if (pthread_create(&threads[i], NULL, sum_partial, (void *)&partialSumArgs[i]) != 0){
            perror("Erreur");
            return EXIT_FAILURE;
        }
    }

    //attendre la fin d'execution de tous les threads
    for (i = 0; i < NUM_THREADS; ++i){
        pthread_join(threads[i], NULL);
    }

    printf("Somme totale : %d\n", sum);// Somme totale : 190
    pthread_mutex_destroy(&lock);

    return 0;
}
```