Systèmes d'exploitation

Test n°2

Partie A : Questions conceptuelles

1. Définir les termes suivants :

- **a. Fil :** Un fil (thread) est une unité d'exécution dans un programme, partageant la mémoire et les ressources avec d'autres fils dans le même processus.
- **b. Sémaphore :** Un sémaphore est un mécanisme de synchronisation utilisé pour limiter l'accès à des ressources partagées dans un environnement multithread.
- **c. Conditions de course :** Une condition de course survient lorsque plusieurs threads accèdent à des données partagées simultanément, entraînant des comportements imprévisibles ou incorrects.

2. Différences entre mutexes et sémaphores.

- **Mutex :** Utilisé pour protéger une seule ressource. Seul un thread peut le posséder à un moment donné. Exemple : Protéger l'accès à une section critique.
- **Sémaphore :** Permet à plusieurs threads d'accéder à un nombre limité de ressources. Exemple : Limiter l'accès à un pool de connexions réseau.

3. Blocages dans un programme multithread.

Les blocages peuvent survenir lorsque deux ou plusieurs threads attendent indéfiniment des ressources détenues par d'autres threads. Techniques pour éviter les blocages :

- Utiliser un ordonnancement des ressources (hiérarchie d'acquisition).
- Implémenter des timeout ou des mécanismes de reprise après détection de blocage.

4. But de la synchronisation des threads.

La synchronisation des threads garantit que les ressources partagées sont utilisées de manière cohérente et sûre. Elle est essentielle pour éviter les conditions de course et garantir la stabilité et la fiabilité du programme.

5. Section critique et utilisation des sémaphores.

Une section critique est une partie du code où une ressource partagée est modifiée. Les sémaphores peuvent être utilisés pour contrôler l'accès à la section critique, garantissant qu'un seul thread y accède à la fois.

Partie B : Problèmes de programmation

1. Création et synchronisation de threads

Code en C :

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

sem_t sem;

void* print_numbers(void* arg) {
    int thread_id = *(int*)arg;

    for (int i = 1; i <= 5; i++) {
        sem_wait(&sem); // Acquire the semaphore
        printf("Thread %d: %d\n", thread_id, i);
        sem_post(&sem); // Release the semaphore
    }

    return NULL;
}

int main() {
    pthread_t threads[3];
    int ids[3] = {1, 2, 3};

    sem_init(&sem, 0, 1); // Binary semaphore for synchronization

    for (int i = 0; i < 3; i++) {
```

```
        pthread_create(&threads[i], NULL, print_numbers, &ids[i]);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&sem);
    return 0;
}
```

2. Utilisation du sémaphore

Code en C :

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

#define MAX_COMPUTERS 3

sem_t computers;

void* use_computer(void* arg) {
    int student_id = *(int*)arg;

    printf("Student %d is waiting for a computer.\n", student_id);
    sem_wait(&computers); // Acquire a computer
    printf("Student %d is using a computer.\n", student_id);
    sleep(2); // Simulate time using the computer
    printf("Student %d is done using the computer.\n", student_id);
    sem_post(&computers); // Release the computer

    return NULL;
}

int main() {
    pthread_t students[10];
    int ids[10];

    sem_init(&computers, 0, MAX_COMPUTERS);

    for (int i = 0; i < 10; i++) {
```

```
30          ids[i] = i + 1;
31          pthread_create(&students[i], NULL, use_computer, &ids[i]);
32      }
33
34      for (int i = 0; i < 10; i++) {
35          pthread_join(students[i], NULL);
36      }
37
38      sem_destroy(&computers);
39      return 0;
40  }
```