

Programmation Concurrente

Problèmes classiques

Exercice 1

Une tribu de cannibales a une manière de manger très conviviale: ils mangent tous du ragoût de missionnaires depuis la même énorme casserole.

- La casserole contient N portions à manger.
- Lorsqu'un cannibale veut manger, il se sert une portion à moins que la casserole soit vide.
- Si la casserole est vide, alors le cannibale réveille le chef et attend que celui-ci finisse de remplir la casserole.

Chaque cannibale exécute le pseudo-code suivant :

```
cannibal() {  
    while (true) {  
        get_serving_from_pot()  
        eat()  
    }  
}
```

Le chef exécute le pseudo-code suivant :

```
chief() {  
    while (true) {  
        put_servings_in_pot(N)  
    }  
}
```

Les contraintes de synchronisation sont les suivantes :

- On part du principe que `get_serving_from_pot` n'est pas thread-safe mais que `eat` et `put_servings_in_pot` le sont.
- Les cannibales ne peuvent pas appeler `get_serving_from_pot()` si la casserole est vide.
- Le chef doit seulement appeler `put_servings_in_pot(N)` lorsque la casserole est vide.

Modifiez le pseudo-code ci-dessus à l'aide de sémaphore(s) et mutex(es), afin de satisfaire les contraintes de synchronisation énoncées.

Exercice 2

Soit un pont situé sur une route à sens unique que des véhicules traversent toujours dans le même sens.

- Deux types de véhicules traversent le pont : des voitures et des camions.
- La masse d'un camion est exactement le double de celle d'une voiture.
- Le pont peut supporter une charge maximale correspondant à huit voitures (ou quatre camions, ou six voitures et un camion, etc.).
- Les voitures et camions font un tour en campagne pendant un certain temps (long) puis traversent le pont (rapide).
- Le nombre total de voitures et camions est arbitraire.

Voici le pseudo-code d'une solution modélisant ce système à l'aide de sémaphore :

```
bridge = semaphore(8)

car() {
    while (true) {
        drive_around(long_time)
        wait(bridge)
        cross_bridge(short_time)
        post(bridge)
    }
}

truck() {
    while (true) {
        drive_around(long_time)
        wait(bridge)
        wait(bridge)
        cross_bridge(short_time)
        post(bridge)
        post(bridge)
    }
}
```

Question 1

Montrez que la solution proposée ci-dessus souffre de deadlock.

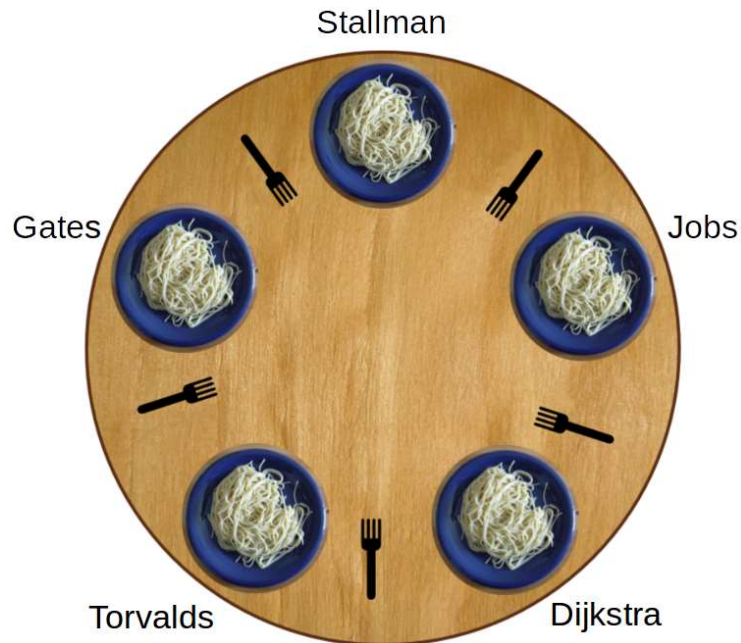
Question 2

Proposez une solution garantissant:

- Aucun deadlock.
- Respect de la charge maximale supportée par le pont.
- Les attentes doivent être passives

Exercice 3

Le problème du dîner des philosophes est un problème classique de concurrence car il démontre les difficultés rencontrées dans le cadre de l'allocation de ressources. Ce problème fût originellement formulé par Edsger Dijkstra en 1965 à l'époque où les dinosaures dominaient encore la faune terrestre.



Le problème est le suivant :

- Cinq philosophes sont assis à une table.
- Chacun a devant lui une assiette de spaghetti.
- Entre chaque assiette se trouve une fourchette.

Un philosophe a trois états possibles :

- Penser pendant un temps aléatoire.
- Etre affamé (pendant un temps déterminé et fini, sinon il y a famine).
- Manger pendant un temps déterminé et fini.

Les philosophes sont aussi soumis aux contraintes suivantes :

- Quand un philosophe a faim, il attend que chacune des fourchettes à sa gauche et à sa droite soient disponibles.
- Chaque philosophe mange de manière un petit peu particulière : il utilise la fourchette à sa gauche et la fourchette à sa droite.
- Si un philosophe ne peut s'emparer des deux fourchettes, il reste affamé (bloqué) pendant un temps déterminé jusqu'à ce qu'une des fourchettes devienne libre.
- Plus d'un philosophe doit pouvoir manger en même temps.

Les philosophes mangent et pensent durant des temps aléatoires différents. De fait, leur changements d'état se produisent de manière asynchrone.

Le but de cet exercice est d'implémenter un programme modélisant notre table de cinq philosophes à l'aide de sémaphores afin qu'ils puissent tous manger, si possible au moins deux en même temps, sans qu'il n'y ait de problèmes concurrence (sans deadlock).

Indications

- Chaque philosophe est modélisé par un thread.
- Les fourchettes représentent les ressources que les threads doivent s'approprier (de manière exclusive) afin de progresser.
- Les philosophes sont numérotés de 0 à 4.
- Les fourchettes sont numérotées de 0 à 4.
- Le philosophe *i* dispose de la fourchette *i* sur sa droite et *i*+1 sur sa gauche.
- Vous pouvez baser votre implémentation sur le code incomplet donné ci-dessous.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

#define PHILOSOPHER_COUNT 5

#define RIGHT(x) ((x) % PHILOSOPHER_COUNT)
#define LEFT(x)  ((x+1) % PHILOSOPHER_COUNT)

typedef unsigned int uint;

static uint delay_ms;

// sleep time in milli-seconds
static void sleep_ms(uint time) {
    struct timespec unused, t = { time/1000, (time % 1000)*1000000 };
    nanosleep(&t, &unused);
}

static void delay(uint *seed) {
    sleep_ms(rand_r(seed) % delay_ms);
}

static void think(int id, uint *seed) {
    printf("%d thinks\n", id);
    delay(seed);
}

static void eat(int id, uint *seed) {
    printf("%d eats\n", id);
    delay(seed);
}

static void get_forks(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    printf("%d attempting to get forks %d & %d\n", id, left, right);
    printf("%d has forks %d & %d\n", id, left, right);
}

static void release_forks(int id) {
    int left = LEFT(id);
    int right = RIGHT(id);
    printf("%d releases forks %d & %d\n", id, left, right);
}

static void *philosopher(void *p) {
    int id = *((int *) p);
    uint seed = getpid() * pthread_self();

    while (true) {
```

```

        think(id, &seed);
        get_forks(id);
        eat(id, &seed);
        release_forks(id);
    }
    return NULL;
}

static void parse_args(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Syntax: philo <delay>\n");
        fprintf(stderr, "Example: philo 200\n");
        fprintf(stderr, "Note: delay is in milli-seconds\n");
        exit(EXIT_FAILURE);
    }

    delay_ms = atoi(argv[1]);
}

int main(int argc, char *argv[]) {
    parse_args(argc, argv);

    srand(time(NULL));

    setbuf(stdout, NULL); // Disable standard output buffering

    pthread_t philo_th[PHILOSOPHER_COUNT];
    int id[PHILOSOPHER_COUNT];

    for (int i = 0; i < PHILOSOPHER_COUNT; i++) {
        id[i] = i;
        pthread_create(&philo_th[i], NULL, philosopher, &id[i]);
    }

    for (int i = 0; i < PHILOSOPHER_COUNT; i++)
        pthread_join(philo_th[i], NULL);

    return EXIT_SUCCESS;
}

```