Programmation Concurrente

Principes et concepts
2024-2025

Chapitre6: Semaphores

Plan

- Définition sémaphore
- Implémentation
- Exemples
- Limites

Description

- Sémaphore = type abstrait
  - Structure de données (compteur + file d'attente d'entités)
  - Interface (opérations, dont initialisation, sur la structure de données)
- Un sémaphore est semblable à un entier, mais avec 4 différences :
  - Un sémaphore est initialisé à une valeur entière ≥ 0
  - Les seules opérations possibles sont l'incrémentation ou la décrémentation.
  - Il n'est, en général, pas possible de lire la valeur d'un sémaphore.
  - Lorsqu'un thread décrémente un sémaphore, si le résultat est négatif, le thread est alors suspendu jusqu'à ce qu'un autre thread incrémente le sémaphore.
  - Lorsqu'un thread incrémente un sémaphore, un des threads suspendus est réveillé (si au moins un thread est suspendu).

Implémentation(1)

```
void decrement(semaphore  S)
{
        if (S  > 0)
                S --;
        SINON (Attendre sur S)
}
void increment(semaphore S)
{
        S ++;
        if ( des threads sont en attente sur S )
                débloquer une des tâches de la file associée à S
}
```

Valeur d'un sémaphore

- Signification de la valeur d'un sémaphore :
  - Valeur > 0 : représente le nombre de threads pouvant décrémenter le sémaphore sans bloquer.
  - Valeur < 0 : représente le nombre de threads bloqués en attente.
  - Valeur == 0 : signifie qu'aucun thread n'est bloqué, mais que si un thread décrémente la valeur, alors il sera bloqué.

Terminologie

- La terminologie originale utilisée par Dijkstra était **V** pour l'incrémentation (***V****erhoog* signifiant augmenter) et **P** pour la décrémentation (***P****robeer te verlagen* signifiant « essayer de réduire »).
- Au fil du temps, plusieurs terminologies ont vu le jour afin de rendre les opérations effectuées plus explicites.
- Terminologies communément utilisées pour l'incrémentation :
  - **V, increment, signal, post**
- Terminologies pour la décrémentation :
  - **P, decrement, wait**
- Dans les pseudo-codes qui suivent nous utilisons la terminologie **signal**/**wait**, car elle nous paraît la plus parlante.

Comparaison mutex / sémaphore

- Mutex et sémaphore sont des mécanismes par attente passive possédant chacun une file d'attente de threads bloqués.
- Un mutex est un mécanisme d'exclusion mutuelle, alors qu'un sémaphore est un mécanisme de signalisation.
- Les sémaphores permettent de résoudre une plus grande classe de problèmes que les mutex.
- Contrairement à un mutex, un sémaphore ne possède pas de propriétaire !
- Similairement à un mutex, un sémaphore binaire (sémaphore initialisé à 1) peut-être utilisé pour verrouiller une section de code.
- ATTENTION: un sémaphore binaire n'est pas un mutex !

Sémaphore

- Type de donnée: **sem_t**
- Création: int **sem_init**(sem t *sem, int pshared, unsigned int valeur)

  pshared !=0 => sémaphore partagée entre plusieurs processus
- Destruction: int **sem_destroy**(sem t *sem) (personne ne doit être bloqué dessus)

- Interface d'usage
  - int **sem_wait**(sem t * sem) Réalise un P()
  - int **sem_post**(sem t * sem) Réalise un V()
  - int **sem_trywait**(sem t * sem) P() ou renvoie EAGAIN pour éviter un blocage
  - int **sem_getvalue**(sem t * sem) Retourne la valeur de la sémaphore

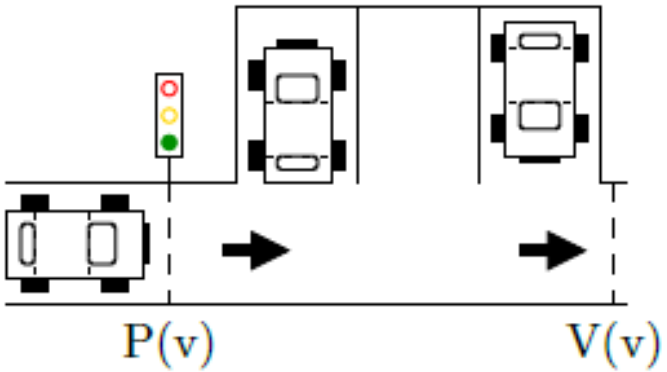
Exemple

Initialisation

```
sem = semaphore(3)   /* nombre de places */
```

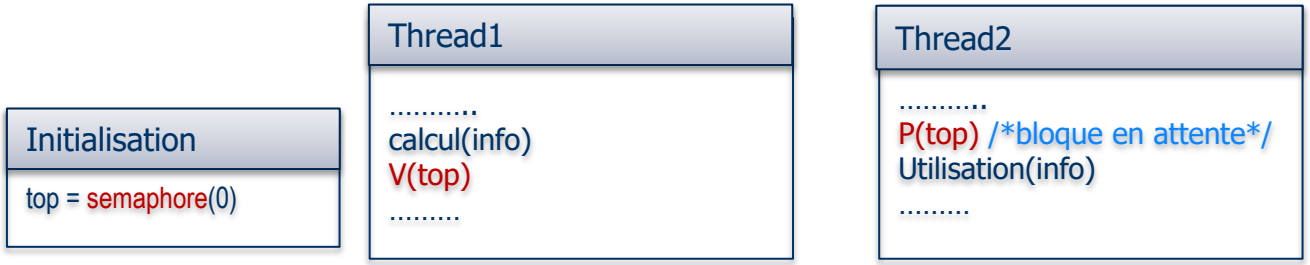
Initialisation

```
P(sem)
   Poser sa voiture au parking
   Aller faire les courses
   Reprendre la voiture
V(sem)
Partir
```

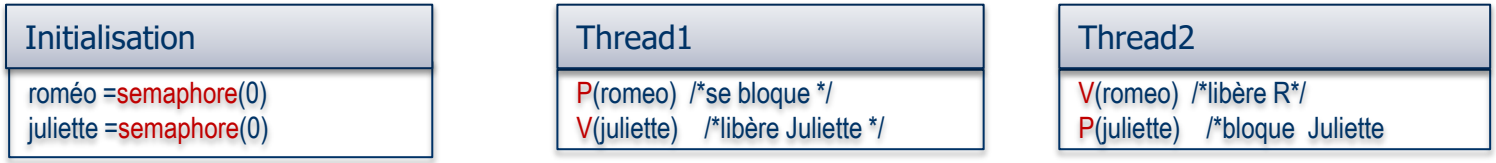


Rendez-vous

- Envoi de signal
  - Un thread indique quelque chose à un autre (disponibilité donnée)



  - Top = sémaphore privé (initialisé à 0) utilisé pour synchro avec quelqu'un, pas pour une ressource
- Rendez-vous entre deux threads
  - Les threads s'attendent mutuellement



- Rendez-vous entre trois processus et plus
  - On parle de barrière de synchronisation

Exemple_01

```c
sem_t semaphore;

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    // Attendre le sémaphore
    sem_wait(&semaphore);
    // Section critique
    printf("Thread %d: Debut de la section critique\n", thread_id);
    // Simuler un traitement
    sleep(2);
    printf("Thread %d: Fin de la section critique\n", thread_id);
    // Libère le sémaphore
    sem_post(&semaphore);
    return NULL;
}
```

```c
int main() {
  // Initialiser le sémaphore avec une valeur initiale de 1
    sem_init(&semaphore, 0, 1);
    pthread_t thread1, thread2;
    int id1 = 1, id2 = 2;
    pthread_create(&thread1, NULL, thread_function, &id1);
    pthread_create(&thread2, NULL, thread_function, &id2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    // Detruire la semaphore
    sem_destroy(&semaphore);
    return 0;
}
```

Limites et Résumé

- Exclusion mutuelle : Ok par définition de P et de INIT(Mutex,1)
- Absence de blocage : Ok par indivisibilité de P
- Condition de progression : Ok par définition de V
- Absence de famine : dépend de l'ordre de retrait de la file du sémaphore. Si l'ordre est FIFO = Ok
- Code identique : Ok

  Et c'est efficace ! Pendant l'attente des processus, ceux qui ne sont pas bloqués peuvent utiliser le CPU