Sémaphore local

Anthony Araye et Camille Schnell

21 février 2018

Sommaire

1 Introduction

En informatique, lorsque deux programmes exécutés en parallèle souhaitent accéder à la même donnée au même moment, cela peut créer des conflits, notamment si l'un des deux modifie la donnée.

LocalSemaphores permet l'implémentation d'un système de sémaphores inter processus hérités par les fils d'un processus afin de gérer des sections critiques.

Les sémaphores peuvent être initialisés avec un nombre de ressources donné (fonction *sem_initialize(int nb)*), et gèrent ensuite l'allocation des ressources et la file d'attente des processus bloqués grâce aux fonctions *sem_acquire(int sid)* et *sem_release(int sid)*. La fonction *sem_destroy(int sid)* permet de détruire un sémaphore et de désallouer la file d'attente associée à ce dernier.

Un sémaphore est détruit lorsque le processus qui l'a créé ainsi que tous ses fils meurent. Sa non-rémanence lui permet ainsi d'être plus efficace que les IPC (Inter Processus Communication) utilisés habituellement sous linux.

2 Manuel d'utilisation

2.1 Appels systèmes

2.1.1 initialize

Nom

#include <sem.h>
int sem_initialize(int *nb***)**

Description

sem_initialize() alloue et initialise un sémaphore avec *nb* comme valeur initiale.
nb représente le nombre de ressources disponibles. Cette valeur doit être supérieure ou égale à 0, elle vaut 1 par défaut.

Valeur renvoyée

sem_initialize() renvoie l'id du sémaphore dans le cas où celui-ci a bien été créé, -1 si une erreur a été levée.

Erreurs

ENOMEM : lorsque l'allocation ne s'est pas faite.
EFAULT : lorsque la valeur passée en argument est invalide (*nb* < 0 ou *nb* > MAX_INT).
ETOOMANY : lorsqu'on a atteint le nombre limite de sémaphores par processus.

2.1.2 acquire

Nom

#include <sem.h>
int sem_acquire(int *sid***)**

Description

sem_acquire décrémente la valeur du sémaphore d'id *sid* si le nombre de ressources disponibles est non nul. Dans le cas où cette valeur est inférieur ou égale à 0, alors il bloque le processus courant et le met dans la file d'attente associée au sémaphore.

Valeur renvoyée

sem_acquire renvoie 0 dans le cas où le processus s'est bien effectué, -1 si une erreur a été levée.

Erreurs

ENOMEM : lorsque l'allocation ne s'est pas faite.
EFAULT : lorsque le *sid* passé en argument ne correspond pas à un sémaphore existant.

2.1.3 release

Nom

#include <sem.h>
int sem_release(int *sid***)**

Description

sem_release incrémente le nombre de ressources disponibles du sémaphore d'id *sid*. Dans le cas où la file associée à *sid* est non vide, alors il réveille le processus en tête de file et saute la tête de la file.

sem_release ne peut être appelé que quand au moins une ressource est utilisée.

Valeur renvoyée

sem_release renvoie 0 dans le cas où le processus s'est bien effectué, -1 si une erreur a été levée.

Erreurs

ENOMEM : lorsque l'allocation ne s'est pas faite.

EFAULT : lorsque le *sid* passé en argument ne correspond pas à un sémaphore existant.

2.1.4 destroy

Nom

#include <sem.h>
int sem_destroy(int *sid***)**

Description

sem_destroy détruit le sémaphore d'id *sid* et désalloue la file d'attente.

Valeur renvoyée

sem_destroy renvoie 0 dans le cas où le sémaphore a bien été détruit, -1 si une erreur a été levée.

Erreurs

EFAULT : lorsque le *sid* passé en argument ne correspond pas à un sémaphore existant.

2.1.5 debug

Nom

#include <sem.h>
int sem_debug(int *sid***, int*** *v***, int*** *pids***, [int** *pidSize***], int** *compteurDeReference***)**

Description

sem_debug permet de connaître le nombre de processus dans la file d'attente du sémaphore d'id *sid*, utile lors d'un debug.

v représente le nombre de ressources, *pids* représente les processus dans la file d'attente, *pidSize* représente la taille du tableau *pids*.

Valeur renvoyée

sem_debug renvoie le nombre de processus dans la file d'attente du sémaphore d'id *sid*, -1 si une erreur a été levée.

Erreurs

EFAULT : lorsque le *sid* passé en argument ne correspond pas à un sémaphore existant.

2.2 Exemple

```c
int main(int argc, char* argv) {
  int s1, s2, p1, p2, status, pid1, pid2;

  // init semaphores
  if(s1 = sem_initialize() == -1) {
    perror("sem_initialize");
    exit(1);
  }
  if(s2 = sem_initialize() == -1) {
    perror("sem_initialize");
    exit(1);
  }

  // acquire semaphores : les prochains sem_acquire seront bloques
  if(p1 = sem_acquire(s1) == -1) {
    perror("sem_acquire");
    exit(1);
  }
  if(p2 = sem_acquire(s2) == -1) {
    perror("sem_acquire");
    exit(1);
  }

  // fork pour creer fils 1 et fils 2
  if(pid1 = fork() == -1) {
    perror("fork");
    exit(1);
  } else {
    // fils 1
    sem_acquire(s1);
    write(1,"fils 1",6);
    sem_release(s1);
    exit(0);
  }
  if(pid2 = fork() == -1) {
    perror("fork");
    exit(1);
  } else {
    // fils 2
    sem_acquire(s2);
    write(1,"fils 2",6);
    sem_release(s2);
    exit(0);
  }
```

```
45
46      // pere
47      if(pid1 >= 1) {
48        sem_release(s1);
49        wait(&status); // on attend la fin du fils 1
50        sem_release(s2); // on reveille le fils 2
51        wait(&status);
52      }
53      else if(pid2 >= 1){
54        sem_release(s2);
55        wait(&status); // on attend la fin du fils 2
56        sem_release(s1); // on reveille le fils 1
57        wait(&status);
58      }
59      write(1,"\n",1);
60      return 0;
61    }
62
```

3 Implémentation

4 Recette

L'implémentation d'un sémaphore local sera testée par des programmes C. Nous allons écrire deux programmes afin de tester le sémaphore de différentes manières.

4.1 Premier programme de test

Objectif

Dans ce premier programme, les sémaphores vont agir comme des mutex (initialisés à 1). L'objectif est à partir de deux threads de naviguer entre deux sections critiques l'une après l'autre.

Implémentation

Deux threads *tPing* et *tPong* seront lancés. Il y aura deux sections critiques contenant chacune un affichage à l'écran indiquant la section critique dans laquelle le thread se situe ("ping" ou "pong").
Le sémaphore *semPong* sera initialisé à 0 pour commencer par l'affichage de "ping". Le sémaphore *semPing* sera ainsi initialisé à 1.

Résultat attendu

Le résultat attendu lors du lancement de ce programme C est un affichage sur la console d'une suite de "ping pong ping pong...".

4.2 Deuxième programme de test

Objectif

Dans le deuxième programme de test, les sémaphores seront utilisés pour permettre de résoudre le problème des lecteurs et rédacteurs énoncé en introduction.

Implémentation

Nous décidons ici de donner la priorité aux lecteurs. Un lecteur pourra donc accéder à la base de données s'il n'y a aucun rédacteur en cours d'écriture. Un rédacteur pourra y accéder s'il n'y a aucun autre rédacteur en cours d'écriture, aucun lecteur en cours de lecture et aucun lecteur dans la file d'attente.
Trois sémaphores sont nécessaires :

- Un sémaphore *semLecteurs* qui protégera l'accès à la variable *nbLecteurs* indiquant le nombre de lecteurs actuellement sur la base de données.
- Un sémaphore *semRedacteurs* empêchant l'accès à la base de données aux autres rédacteurs lorsqu'un rédacteur y accède.
- Un sémaphore *redacteur* qui permet au premier lecteur de bloquer l'accès aux rédacteurs et au rédacteur accédant à la ressource de bloquer l'accès aux lecteurs pendant la rédaction.

On aura ensuite 4 fonctions définissant les 4 actions possibles :

- *startRead()* permettra d'incrémenter la variable *nbLecteurs*. Si celle-ci est égale à 1 (premier lecteur), on lancera *redacteur.acquire()* pour empêcher aux rédacteur d'accéder à la ressource.
- *endRead()* décrémentera la variable *nbLecteurs*. Si celle-ci est égale à 0 (plus aucun lecteur), on lancera *redacteur.release()* pour permettre à un éventuel rédacteur de commencer la rédaction (on va le débloquer s'il était en file d'attente).
- *startWrite()* bloquera l'accès à la ressource aux autres rédacteurs ainsi qu'aux lecteurs en faisant *semRedacteurs.acquire()* et *redacteur.acquire()*.
- *endWrite()* autorisera à nouveau l'accès à la ressource, en priorisant les lecteurs car on fera en premier *redacteur.release()* (libération éventuelle d'un lecteur en attente) puis *semRedacteurs.release()*.

Résultat attendu

Le résultat attendu lors du lancement de ce programme C est :

- Lorsqu'un lecteur accède à la base de données, on l'affiche sur la console avec le nombre total actuel de lecteurs. De même lorsqu'un lecteur quitte la base de données.
- Lorsqu'un rédacteur accède à la base de données, on l'affiche sur la console. De même lorsque le rédacteur quitte la base de données.