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.

Le problème des lecteurs et des rédacteurs énoncé par Dijkstra illustre bien la situation :
Soit une base de données pouvant être accédée par des lecteurs qui vont simplement consulter les données, et des rédacteurs qui vont les modifier.
Les contraintes sont les suivantes :

- *Plusieurs lecteurs peuvent accéder en même temps à la base de données.*
- *Lorsqu'un rédacteur modifie la base de donnée, personne d'autre ne peut y accéder (ni les lecteurs, ni un autre rédacteur).*

L'implémentation d'un système de sémaphores afin de gérer des sections critiques va permettre de répondre au problème ci-dessus. En effet, le sémaphore va agir comme un verrou et mettre en attente des programmes ou threads s'ils souhaitent accéder à une section critique en cours d'utilisation par un ou plusieurs autres programmes ou threads.

2 Manuel d’utilisation

2.1 Tutoriel d’utilisation

2.2 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.

2.3 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.

2.4 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_releaserenvoie 0 dans le cas où le processus s'est bien effectué, -1 si une erreur a été levé.

Erreurs

?

2.5 destroy

Nom

sem_destroy - Détruit le sémaphore

Synopsis

#include <sem.h>

int sem_destroy(sem **sem*)

Description

sem_destroy() détruit le sémaphore *sem* 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 levé.

Erreurs

?

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.