Conservatoire National des Arts et Métiers
FOAD Ile De France

Communication & Synchronisation

Exercices et Travaux Pratiques
20 septembre 2020

Sommaire

Exercice 1 : Signaux

Pour envoyer un signal il faut utiliser la commande **kill**, par défaut **kill PID** envoie le signal **SIGTERM** autrement il faut préciser le numéro ou mnémonique du signal, par exemple **kill -9 PID** pour envoyer le signal **SIGKILL** au processus **PID**.

Pour masquer ou récupérer un signal (se protéger contre l'action par défaut) utilisez la commande **trap**, le signal **9 SIGKILL** ne peut pas être masqué ou récupéré, c'est l'arme absolue pour tuer un processus.

Ci dessous un processus **boucle** qui est une boucle sans fin qui affiche une lettre A toutes les secondes.

Script "boucle" :

```
echo $$      #afficher PID
while true   #boucle sans fin
do
       echo A
       sleep 1
done
```

Script "boucle" avec masquage du signal 15 **SIGTERM** :

```
trap "" 15
echo $$
while true
do
       echo A
       sleep 1
done
```

Script "boucle" avec récupération du signal 15 **SIGTERM** :

```
int15() {    #traitement du signal 15
       echo "arrêt refusé"
}
trap "int15" 15 #début du programme
echo $$
while true
do
       echo A
       sleep 1
done
```

- Lancez le processus boucle dans un terminal et ensuite testez les signaux à partir d'un autre terminal :

```
$kill PID          # SIGTERM ARRET
$kill -9 PID       # SIGKILL ARRET NON MASQUABLE
$kill -17 PID      # SIGTSTOP SUSPENSION ABSOLUE
$kill -18 PID      # SIGTSTP SUSPENSION
$kill -19 PID      # SIGCONT REPRISE
```

- Vous pouvez aussi arrêter votre processus **boucle** à partir de son propre terminal avec **CTRL/C** (équivalent **SIGINT**) ou en fermant son terminal (équivalent **SIGHUP**)

Exercice 2 : Solution Mutex aux Problème des Réservations

Ci-dessous une solution au problème de réservation en utilisant un Mutex dans le cas d'une application multithread : 2 threads de réservation sont lancés :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static pthread_mutex_t my_mutex;    // Mutex
static int compteur;                // Ressource a protéger

void *reservation (void * arg) {    // Thread de réservation
  pthread_t pthr = pthread_self();
  while (1) {
    pthread_mutex_lock (&my_mutex); // Prise du mutex, si pris attente
    if (compteur > 0) {
      compteur--;                       // décrémenter compteur
      pthread_mutex_unlock (&my_mutex); // libération du mutex
      printf("Place réservée thread %u\n",(unsigned int)pthr);
    }
    else {
      pthread_mutex_unlock (&my_mutex); // libération du mutex
      printf("Plus de places thread %u\n",(unsigned int)pthr);
      break;
    }
  }
}

int main (int ac, char **av) {
  pthread_t th1, th2;
  void *ret;
  compteur = 10;              // 10 places à réserver
  pthread_mutex_init (&my_mutex, NULL);

  /* Threads de réservation */
  if (pthread_create (&th1, NULL, reservation, NULL) < 0) {
    printf("Faute création thread 1\n");
    exit(-1);
  }
  if (pthread_create (&th2, NULL, reservation, NULL) < 0) {
    printf("Faute création thread 2\n");
    exit(-1);
  }

  /* Retour */
  (void)pthread_join (th1, &ret); //attente terminaison thread th1
  (void)pthread_join (th2, &ret); //attente terminaison thread th2
  exit (0);
}
```

** N'oubliez pas de compiler votre programme avec l'option* ***`-pthread`***

Exercice 3 : Le Problème des Producteurs / Consommateurs

Il existe une implémentation Unix système V et une implémentation Posix des Sémaphores, nous présentons dans ce document l'implémentation Posix plus concise et plus intuitive :

`sem_init()` : Création d'un sémaphore avec initialisation du nombre de jetons
`sem_destroy()` : Suppression d'un sémaphore
`sem_wait()` : Fonction "P" (décrémenter jetons avec attente si pas de jetons disponibles)
`sem_post()` : Fonction "V" (incrémenter jetons)
`sem_open()` : Création d'un sémaphore nommé avec initialisation des jetons
`sem_close()` : Suppression d'un sémaphore nommé

Rappel

Les deux processus doivent se synchroniser entre eux de façon à respecter certaines contraintes de bon fonctionnement :
1) Le producteur ne peut déposer un message dans le tampon s'il n'y a plus de place libre.
2) Le consommateur ne peut retirer un message depuis le tampon s'il est vide.
3) Le consommateur ne doit pas retirer un message que le producteur est en train de déposer (message incomplet).

Etude de la solution

Le nombre de messages disponibles dans le tampon peut être symbolisé par un nombre correspondant de jetons disponibles :
- le producteur incrémente le nombre de jetons disponibles à chaque message déposé.
- le consommateur décrémente le nombre de jetons disponibles à chaque message retiré.

Cependant un seul sémaphore ne suffit pas car un sémaphore "jetons messages disponibles" mettra en attente le consommateur lorsque le tampon sera vide mais ne mettra pas en attente le producteur lorsque le tampon sera plein.

Il va donc falloir utiliser 2 sémaphores un dit `"plein"` dont les jetons représentent les messages disponibles dans le tampon et un dit `"vide"` dont les jetons symbolisent les places vides dans le tampon.

Ensuite pour être certains que le consommateur n'est pas en train de récupérer un message en cours de production, nous ferons en sorte qu'il récupère systématiquement le plus ancien des messages du buffer tampon (en gérant un index circulaire dans le buffer).

```
Initialisation
sémaphore "plein" = 0
sémaphore "vide" = n
```

```
Producteur
Fonction "P" sémaphore "vide"
dépose le message suivant
Fonction "V" sémaphore "plein"
```

```
Consommateur
Fonction "P" sémaphore "plein
Prélève le plus ancien message
Fonction "V" sémaphore "vide"
```

Mise en Application

On traite le problème du producteur / consommateur dans le cas multithread : Le producteur est le fils qui produit les articles et le consommateur est le père qui les consomme, dans cet exemple le buffer tampon ne contient que 3 emplacements.

```
#include <semaphore.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define MAX_PROD 10           // Max produits à produire au total
#define MAX_BUF 3             // Taille max buffer tampon
typedef struct semaphore {    // structure 2 sémaphores vide et plain
 sem_t sem_vide;
 sem_t sem_plein;
} sema;
static int buf[MAX_BUF];      // buffer produits

/* Thread Producteur */
void *producteur(void *arg) {
  int ip = 0;                   // index circulaire  dans buffer
  int nbprod = 0;               // nombre de produits produits
  int produit = 1001;           // identifiant produit produit
  sema *args = (sema *) arg;
  while(nbprod < MAX_PROD) {
  sem_wait(&args->sem_vide);    // Fonction "P" sémaphore vide
  buf[ip] = produit;
  sem_post(&args->sem_plein);  // Fonction "V" sémaphore plein
  printf("produteur: buf[%d]=%d\n", ip, produit);
  produit++;
  nbprod++;
  ip = (ip + 1) % MAX_BUF;    // maj index circulaire
  }
  return NULL;
}

/* Thread Consommateur */
void *consommateur(void *arg) {
  int ic = 0;                  // index circulaire dans buffer
  int nbcons = 0;              // nombre de produits consommés
  int produit;                 // identifiant produit consommé
  sema *args = (sema *) arg;
  while(nbcons < MAX_PROD) {
  sleep(1);
  sem_wait(&args->sem_plein); // Fonction "P" sémaphore plein
  produit = buf[ic];
  sem_post(&args->sem_vide);  // Fonction "V" sémaphore vide
  printf("consommateur: buf[%d]=%d\n", ic, produit);
  nbcons++;
  ic = (ic + 1) % MAX_BUF;      // maj index circulaire
  }
  return NULL;
}

/* Programme principal */
int main( ) {
  int p, i;
  pthread_t th1, th2;
```

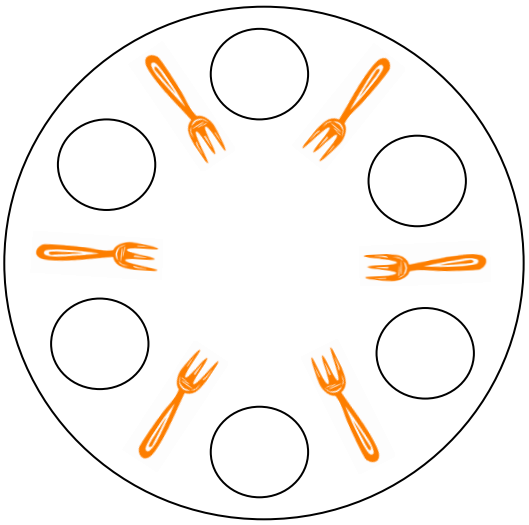
```
  sema args;
  if (sem_init(&args.sem_vide, 0, MAX_BUF) == -1) perror ("1");
  if (sem_init(&args.sem_plein, 0, 0) == -1) perror("2");
  if (pthread_create(&th1,NULL,producteur,&args) != 0) perror("3");
  if (pthread_create(&th2,NULL,consommateur,&args) != 0) perror("4");
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
}
```

** N'oubliez pas de compiler votre programme avec l'option* `-pthread`

Compléments de cours : Exemple du diner des Philosophes

L'interblocage n'est pas réservé exclusivement au comportement du logiciel, on peut avoir interblocage dans différents cas de figure, par exemple le problème du diner des philosophe :

6 philosophes se trouvent autour d'une table, chacun des philosophes a devant lui un plat de spaghettis, à gauche de chaque plat de spaghettis se trouve une fourchette.
Pour manger les spaghettis, le philosophe a en fait besoin de deux fourchettes : celle qui se trouve à gauche de son assiette, et celle qui se trouve à droite de son assiette, c'est à dire celle qui se trouve à la gauche de son voisin de droite.



Si un philosophe n'arrive pas à s'emparer des 2 fourchettes, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Interblocage

Si les actions des philosophes ne sont pas coordonnées, il peut y avoir interblocage total, c'est à dire que plus aucun philosophe ne peut manger :
Par exemple les philosophes s'emparent tous au même moment de leur fourchette gauche, si aucun d'entre eux ne la repose ils vont rester affamés éternellement.

Solution

Cet interblocage peut être évité par une coordination simple du temps à la table des philosophes :
Par exemple on numérote successivement les philosophes de 1 à 6 et on autorise les philosophes pairs à manger aux heures paires et les philosophes impairs à manger aux heures impaires, durant les heures non autorisées ils doivent reposer leurs fourchettes.

Compléments de cours : Exemple d'interblocage Oracle

Une transaction atomique Oracle peut impliquer plusieurs tables et les programmeurs verrouillent ces tables (ou des parties de ces tables) au début de la transaction et les libèrent à la fin de la transaction pour s'assurer de la cohérence des consultations et mises à jour. S'il n'a pas été prévu une stratégie de verrouillage des tables, des interblocages peuvent se produire. Dans l'exemple ci-dessous :

Le processus P1 verrouille les tables 1 puis 2

```
DECLARE
  l_deadlock_1_id  deadlock_1.id%TYPE;
  l_deadlock_2_id  deadlock_2.id%TYPE;
BEGIN
  -- Lock row in first table.
  SELECT id
  INTO   l_deadlock_1_id
  FROM   deadlock_1
  WHERE  id = 1
  FOR UPDATE;
   -- Pause.
  DBMS_LOCK.sleep(30);
  -- Lock row in second table.
  SELECT id
  INTO   l_deadlock_2_id
  FROM   deadlock_2
  WHERE  id = 1
  FOR UPDATE;
  -- Release locks.
  ROLLBACK;
END;
```

Le processus P2 lui verrouille les tables 2 puis 1

```
DECLARE
  l_deadlock_1_id  deadlock_1.id%TYPE;
  l_deadlock_2_id  deadlock_2.id%TYPE;
BEGIN
  -- Lock row in second table.
  SELECT id
  INTO   l_deadlock_2_id
  FROM   deadlock_2
  WHERE  id = 1
  FOR UPDATE;
   -- Pause.
  DBMS_LOCK.sleep(30);
  -- Lock row in first table.
  SELECT id
  INTO   l_deadlock_1_id
  FROM   deadlock_1
  WHERE  id = 1
  FOR UPDATE;
  -- Release locks.
  ROLLBACK;
END;
```

L'interblocage peut se produire, voici le message d'erreur Oracle :

```
*** 2006-09-13 09:11:40.646
*** ACTION NAME:() 2006-09-13 09:11:40.615
*** MODULE NAME:(SQL*Plus) 2006-09-13 09:11:40.615
*** SERVICE NAME:(SYS$USERS) 2006-09-13 09:11:40.615
```

```
*** SESSION ID:(137.7008) 2006-09-13 09:11:40.615

DEADLOCK DETECTED

[Transaction Deadlock]
Current SQL statement for this session:
SELECT ID FROM DEADLOCK_2 WHERE ID = 1 FOR UPDATE
----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
1AFBE484        16  anonymous block

The following deadlock is not an ORACLE error. It is a
deadlock due to user error in the design of an application
or from issuing incorrect ad-hoc SQL. The following
information may aid in determining the deadlock:
Deadlock graph:
…
…
…
```

- Le message insiste sur le fait qu'il ne s'agit pas d'une faute de fonctionnement de la base Oracle, mais d'une faute de conception des programmes, et c'est bien le cas.