TME1 : Programmation Concurrente, généralités

P. Trebuchet

4 février 2019

Exercice 1 – Implantation de processus et threads

Question 1

En utilisant l'appel système `pid_t fork(void);`, créer deux processus affichant la valeur de retour de l'appel à `fork`.

Solution:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>

int main()
{
        int i=0;
        int pid=fork();
        printf("Ma variable pid est %d\n",pid);
}
```

Question 2

Ecrire maintenant un programme possédant une variable entière `i`, créant un processus fils et tel que le père affiche un message indiquant qu'il est le père et demande à l'utlisateur de saisir une valeur pour `i` au clavier et tel que le fils commence par dormir 4 secondes, puis affiche le contenu de `i`.

Solution:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>

int main()
{
        int i=0;
        int pid=fork();
        if(pid){
                printf("Je suis le pere et ma varible pid est %d\n",pid);
                printf("donne moi un nombre maintenant\n");
                scanf("%d",&i);
                printf("ma variable i vaut %d\n",i);
```

```
                }
        else
        {
                sleep(4);
                printf("\nje suis le fils et ma variable pid vaut %d\n",pid);
                printf("je suis le fils et ma variable i vaut %d\n",i);
        }


}
```

Question 3

En utilisant la fonction `pid_t getpid(void);` (`pid_t` est compatible avec `int`), écrire le même programme avec des `threads` POSIX sauf que le père attend cette fois ci la fin du fils avant de terminer. Que constatez vous quant au contenu de la variable `i` dans le fils ? Qu'en deduisez vous sur le fonctionnment des threads Posix ? Quelle précautions sont à prendre pour éviter au programme précédant de produire un résultat indéfini ?

Solution:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include <pthread.h>
#include<stdlib.h>

int i=0;
pthread_t th1;

void *fun_fils(void *toto)
{
  sleep(10);
  printf("c'est moi le fils et le contenu de pid vaut %d\n",getpid());
  printf("c'est moi le fils et le contenu de i vaut %d\n",i);
  return NULL;
}

int main()
{
  if (!pthread_create(&th1,NULL,fun_fils,NULL)) {
    printf("je suis le pere et ma variable pid vaut %d\n",getpid());
    printf("donne moi vite (moins de 10 secondes) un nombre \n");
    scanf("%d",&i);
    printf("je suis le pere et ma variable i vaut %d\n",i);
    pthread_join(th1,NULL);
  } else {
    printf("Creation de thread impossible.\n");
  }

}
```

On voit ici que la mémoire est partagéee il faut répondre avant les 4 secondes de sommeil pour le fils sinon on ne sait pas ce qui se passe.

Exercice 2 – Compteur partagé

Question 1 – Architecture de base

Ecrire un programme utilisant l'API des `pthread` effectuant la tâche suivante :

Le programme principal initialise deux variables entières `temp` et `SHARED_compteur` à 0 puis lance un nombre `NB_THREAD` de pthread éxecutant la routine suivante :

- lire la valeur de la variable partagée dans `temp`
- rendre la main à l'ordonnanceur (`usleep, sched_yield`)
- incrémenter la valeur de `temp`
- rendre la main à l'ordonnanceur
- incrémenter la variable `SHARED_compteur`

Une fois les `pthreads` lancés, le programme principal attend tant que la valeur de `SHARED_compteur` soit `NB_THREAD` puis il affiche `"TERMINE"`

Dans un premier temps, ne pas synchroniser le programme.

Solution:

```c
// Fichier compteur1.c
// Compteur partage en Posix Threads
// version NON-SYNCHRONISEE

// pour compiler (sous linux) :
// gcc -Wall --std=c99 -pedantic -pedantic-errors compteur1.c -o compteur1 -lpthread

#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sched.h>
#include<pthread.h>

#define NB_THREAD 10
int SHARED_compteur = 0;

void* THREAD_IncrementeCompteur(void *arg)
{
  int temp = SHARED_compteur;
  sched_yield();
  SHARED_compteur=temp+1;
  sched_yield();
  printf("Compteur = %d\n", SHARED_compteur);
  return NULL;
}

int main(void)
{
  int i;
  pthread_t incr_threads[NB_THREAD];

  for(i=0;i<NB_THREAD;i++)
    pthread_create(&incr_threads[i], NULL, THREAD_IncrementeCompteur, NULL);

  while(SHARED_compteur!= NB_THREAD)
```

```
    {
      // ne rien faire
    }
  printf("TERMINE\n");
}
```

Question 2 – Terminaison et jonction

Le programme principal utilise une boucle qui vérifie la valeur du compteur. Cela s'appelle une attente active qui occupe inutilement le processeur. Pour détecter la terminaison des threads, on peut utiliser le mécanismes de jonction. Ecrire une variante du programme avec jonction.

Solution:

```
// Fichier compteur2.c
// Compteur partage en Posix Threads
// version non-synchronisee avec jonction

// pour compiler (sous linux) :
// gcc -Wall --std=c99 -pedantic -pedantic-errors compteur2.c -o compteur2 -lpthread

#include<stdlib.h>
#include<unistd.h>
#include<sched.h>
#include<pthread.h>
#include<stdio.h>

#define NB_THREAD 10

int SHARED_compteur;

void* THREAD_IncrementeCompteur(void *arg)
{
  int temp = SHARED_compteur;
  sched_yield();
  SHARED_compteur=temp+1;
  sched_yield();
  printf("Compteur = %d\n", SHARED_compteur);
  return NULL;
}

int main(void)
{
  int i;
  pthread_t incr_threads[NB_THREAD];
  void *status;

  for(i=0;i<NB_THREAD;i++)
    pthread_create(&incr_threads[i], NULL, THREAD_IncrementeCompteur, NULL);
  for(i=0;i<NB_THREAD;i++)
    pthread_join(incr_threads[i],&status);

  if(SHARED_compteur== NB_THREAD)
```

```
    printf("TERMINE\n");

}
```

Remarque:
Les ressources allouées à un thread ne sont libérées que dans quelques cas :
- Le thread principale prend fin.
- On fait un `pthread_join`
- Le thread fini est dans un état `detached`, auquel cas son code de retour est inaccessible.

Le programme affiche-t-il tout le temps "TERMINE" ? Si non, donnez une exécution (avec `NB_THREAD=2`) qui n'affiche rien

Solution:
Thread 1 :
temp = 0,SHARED_compteur = 0 Thread 2 :
temp = 0, SHARED_compteur = 0 Thread 2 :
SHARED_compteur = 1 Thread 1 :
SHARED_compteur = 1
A la fin :
SHARED_compteur = 1 (et pas 2=NB_THREAD)

Exercice 3 – Producteurs et Consommateurs

Le patron Producteur/Consommateur est un des patrons les plus courants dès lors qu'on utilise des threads. On le retrouve dans de très nombreuses applications :
- Jeux : les routines d'IA modifient l'aire de jeu et la routine d'affichage affiche le nouvel univers
- Sommation : Des routines auxiliaires calculent de éléments de la somme totale, et une routine regroupe les différents éléments
- ...

Question 1 – Architecture de base

La patron s'articule autour de deux ensembles :
- une équipe de producteurs qui produit des données
- une équipe de consommateurs qui les consomme.

Les deux équipes communiquent entre elles par une zone de transfert. Dans cet exercice on utilisera un `FIFO` de taille fixe comme Buffer de communication entre les équipes. Une implantation simple de cette structure de donnée se fait en utilisant un tableau de taille fixe et deux pointeurs dans ce tableau : l'un pour désigner le prochain endroit où on pourra consommer une donnée et l'autre pour désigner le prochain endroit où mettre une donnée.

Dans notre exercice, on enregistre un nombre `BUF_SIZE` maximum de données de type `int`. On écrira une fonction `Buffer* CreerBuffer()` pour créer un buffer.

On écrira en outre les fonctions auxiliaires :
- **`void`** `EcrireBuffer(Buffer *buf,` **`int`** `val)` pour ajouter un élément (afficher "P" sur la sortie d'erreur si le buffer est plein).
- **`int`** `LireBuffer(Buffer *buf)` pour lire le plus ancien élément (si disponible, afficher "V" sur la sortie d'erreur si le buffer est vide)

Ecrire les fonctions mentionées ci-dessus

Question 2

Ecrire deux fonctions qui seront exécutées respectivement par les threads producteurs et les threads consommateurs :

— Une fonction de prototype : **void** * THREAD_Producteur(**void** * arg), implantantant la tache dévolue au producteur, i.e. produisant NB_TURN entiers à intervalle régulier (CF commande usleep pour la temporisation)
— Une fonctions de prototype : **void** * THREAD_Consommateur(**void** * arg), implantant la tâche dévolue au consommateur, i.e. consommant des entiers à intervalle régulier (CF commande usleep pour la temporisation) et affichant les entiers consommés.

Ecrire cette premiere version sans synchronisation **Solution**:

```
// Fichier prodcons1.c
// Producteurs/Consommateurs
// Premiere version : sans synchronisation
// compilation
// gcc -Wall --std=c99 prodcons1.c -o prodcons1 -lpthread

#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <pthread.h>
// Constantes
#define BUF_SIZE 10
#define NB_TURN 55
#define TRUE 1
#define FALSE 0

// Structure de donnee BUFFER
typedef struct _Buffer {   int nb_elem;   int elem[BUF_SIZE]; } Buffer;

Buffer * CreerBuffer()
{
  Buffer* buf = (Buffer *) malloc(sizeof(Buffer));
  if(buf==NULL)
    {
      fprintf(stderr,"Impossible d'allouer le buffer\n");
      exit(EXIT_FAILURE);
    }   buf->nb_elem=0;
  return buf;
}

void EcrireBuffer(Buffer *buf, int val)
{
  int cont = FALSE;

  while(!cont)
    {
      if(buf->nb_elem == BUF_SIZE)
        // BUFFER plein
          fprintf(stderr,"P");
      else
        // OK, on peut ecrire
        cont=TRUE;
    }
  buf->elem[buf->nb_elem] = val;
```

```
  sched_yield();   buf->nb_elem++;
}

int LireBuffer(Buffer *buf)
{
  int cont = FALSE;
  while(!cont)
    {
      if(buf->nb_elem == 0)
        {
          // BUFFER vide
          fprintf(stderr,"V");
        }
      else
        {
          // OK, on peut lire
          cont=TRUE;
        }
    }

  int val = buf->elem[0];
  int i;   sched_yield();
  for(i=1;i<buf->nb_elem;i++)
    buf->elem[i-1]=buf->elem[i];
  sched_yield();

  buf->nb_elem--;
  return val;
}

// Routine du producteur

void * THREAD_Producteur(void * arg)
{
  Buffer *buf = (Buffer *) arg;
  int i;
  for(i=0;i<NB_TURN;i++)
    {
      EcrireBuffer(buf,i);
    }
  return NULL;
}

// Routine du consommateur
void * THREAD_Consommateur(void * arg)
{
  Buffer *buf = (Buffer *) arg;

  int compteur =0;
  int val;
  while(compteur<NB_TURN)
    {
```

```
      val = LireBuffer(buf);
      printf("Valeur recue = %d\n",val);
      compteur++;
    }
  return NULL;
}

// Programme principal
int main(void)
{
  pthread_t thread_prod;
  pthread_t thread_cons;
  long stat_prod;
  long stat_cons;

  Buffer * buf = CreerBuffer();
  pthread_create(&thread_prod,NULL,THREAD_Producteur,(void *) buf);


  pthread_create(&thread_cons,NULL,THREAD_Consommateur,(void *) buf);
  pthread_join(thread_prod, (void**) &stat_prod);
  pthread_join(thread_cons, (void**) &stat_cons);

}
```

En redirigeant la sortie d'erreur vers `/dev/null`, donnez un exemple d'affichage du programme.

Solution:

```
Valeur recue = 0
Valeur recue = 1
Valeur recue = 1
Valeur recue = 2
Valeur recue = 2
Valeur recue = 4
Valeur recue = 4
Valeur recue = 6
Valeur recue = 6
Valeur recue = 8
Valeur recue = 8
Valeur recue = 10
Valeur recue = 10
Valeur recue = 12
Valeur recue = 12
Valeur recue = 14
Valeur recue = 14
Valeur recue = 14... 40 fois
```

Quels sont les problèmes posés par ce programme ?

Solution:

— les lecteurs peuvent lire plusieurs fois la meme valeur produite
— attentes actives

Rappel : API POSIX Thread (Extrait)

Quelles sont les fonctionalités de base de l'API POSIX PThread
cf. Rappels Posix Threads

Creation de thread

```
int pthread_create(pthread_t  *  thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);
```

Exemple :

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

void* THREAD_Routine(void *arg)
{    /* routine de thread*/
  return NULL;
}

int main(void)
{
  pthread_t thread_id;
  int ok;

  ok = pthread_create(&thread_id, NULL, THREAD_Routine, /* arg */ NULL);
  if(ok!=0)
    {
      fprintf(stderr,"Impossible de creer de le thread\n");
      exit(EXIT_FAILURE);
    }
}
```

Destruction de thread

```
void pthread_exit(void *retval);
// note: retval est disponible dans pthread_join
```

Identification

```
  pthread_t pthread_self(void);
  int pthread_equal(pthread_t thread1, pthread_t thread2);
```

Jonctions

```
int pthread_join(pthread_t th, void **thread_return);
```