L3 INFORMATIQUE

Eléments de correction du contrôle de Programmation Unix – 14 janvier 2021

Michel SOTO

AUCUN DOCUMENT AUTORISÉ

Durée: 1 H 30

Le barème est indicatif - Nombre de pages: 3

La concision de vos réponses, leur clarté et le soin apporté à leur mise en forme seront pris en compte.

PARTIE I : CONNAISSANCE DU COURS

Question 1 (4 points – 5 mn)

Répondez aux affirmations suivantes uniquement par "VRAI", ou "FAUX" ou "NE SAIS PAS".

Barème : **réponse exacte : +1 point, réponse fausse : -0,5 point sur la copie, "ne sais pas" : 0 point**

a) Le flag `SA_RESTART` de la structure `sigaction` permet de redémarrer l'exécution d'un gestionnaire de signal interrompue par un autre signal. **FAUX**

b) L'ouverture d'un tube nommé est bloquante par défaut. **VRAI**

c) Les compteurs de `file table` ne s'incrémentent que lors d'un `fork()`. **FAUX**

d) En mode non connecté, le nommage de la socket du client n'est pas obligatoire. **VRAI**

Question 2 (4 points – 20 mn)

```
#define MAX_THREADS 5
void *Bonjour(void *thread_id){
 long tid;

 tid = *(long *)thread_id;
 printf("Bonjour! Je suis le thread
        %ld!\n", tid);
 pthread_exit(NULL);
}// Bonjour

int main(int argc, char *argv[]){
 pthread_t threads[MAX_THREADS];
 int r;
 long t;

for(t = 0; t < MAX_THREADS; t++){

  r = pthread_create(&threads[t], NULL,
                     Bonjour, (void *)&t);
  if (r){printf("ERREUR; pthread_create()
                 a retourné %d\n", r);
         exit(EXIT_FAILURE);
  } // if
} // for

 pthread_exit(NULL);

} // main
```

Le programme ci-dessus compile parfaitement (les `include` ont été supprimés pour gagner de la place). Les threads créés par ce programme sont numérotés de 1 à 5. Chaque thread affiche sont n° puis se termine.

a) Dans les séquences d'exécution ci-dessous, certains numéros de thread n'apparaissent pas du tout alors que d'autres numéros apparaissent plusieurs fois. Expliquez pourquoi.

C'est l'adresse de la variable `t` qui est passée en paramètre de chaque thread `Bonjour`. La valeur de `t` est modifiée à chaque itération du `for` dans le thread `main`. Lorsqu'un thread `Bonjour` obtient le processeur, la valeur de `t` a été modifiée, entre temps, dans le thread `main` et ne correspond plus à la valeur que possédait `t` au moment de la création du thread `Bonjour`.

Afin que chaque thread `Bonjour` affiche correctement son numéro, il faut passer lui passer la valeur de `t` et non l'adresse de `t`.

b) **Conformément à votre explication en a), réécrivez uniquement les instructions à l'origine de cette situation** afin que chaque thread affiche correctement son numéro.

Dans `Bonjour` l'instruction : `tid = *(long *)thread_id;`
devient : `tid = (long)thread_id;`

Dans `main` l'instruction: `r = pthread_create(&threads[t], NULL, Bonjour, (void *)&t);`
devient : `r = pthread_create(&threads[t], NULL, Bonjour, (void *)t);`

```
./Bonjour
Bonjour! Je suis le thread 2!
Bonjour! Je suis le thread 3!
Bonjour! Je suis le thread 4!
Bonjour! Je suis le thread 4!
Bonjour! Je suis le thread 5!
```

```
./Bonjour
Bonjour! Je suis le thread 2!
Bonjour! Je suis le thread 2!
Bonjour! Je suis le thread 3!
Bonjour! Je suis le thread 5!
Bonjour! Je suis le thread 5!
```

PARTIE II : APPLICATION DU COURS

NE PERDEZ PAS DE TEMPS

!! Vous êtes dispensés de la vérification des valeurs de retour des primitives système et des `includes`. !!

Dans les questions suivantes, **le code devra être rédigé selon les règles de l'art** : vérification du nombre de paramètres éventuels, **indentation**, **commentaires** et **propreté**.

Question 3 (6 points – 30 mn)

Le sémaphore inventé par Edsger Dijkstra autorise N (capacité du sémaphore) processus ou threads à entrer en même temps en section critique. Le sémaphore `mutex` de la librairie `pthread` permet à un seul et seul thread d'entrer en section critique. Le sémaphore `mutex` est donc un cas particulier du sémaphore de E. Dijkstra pour lequel N vaut 1. Vous devez implémenter le sémaphore de Dijkstra en utilisant les sémaphores `mutex`, les variables condition de la librairie `pthread` et un *compteur interne* pour la capacité du sémaphore

a) Complétez le contenu du type `t_semaphore`

```
// ========================================================
// Définition du type implémentant un semaphore de Dijkstra
// ========================================================
   typedef struct{
       int compteur;
       pthread_mutex_t mutex ;
       pthread_cond_t cond ;
   } t_semaphore;
```

b) Écrivez le code de la fonction `pthread_semaphore_init`

```
// ========================================================
    t_semaphore *pthread_semaphore_init(int val_compteur) {
// ========================================================
// Crée un sémaphore et initialise son compteur à val_compteur
// RETOURNE l'adresse du sémaphore créé
// ========================================================
       t_semaphore *s = malloc(sizeof(t_semaphore)) ;
       pthread_mutex_init (&s->mutex, NULL);
       pthread_cond_init (&s->cond, NULL);
       s->compteur = val_compteur ;
       return s ;
   }
```

c) Écrivez le code de la fonction `pthread_semaphore_destroy`

```
// ============================================================
    void pthread_semaphore_destroy(t_semaphore *s) {
// ============================================================
// Libére les resources utilisées et détruit le sémaphore s
// ============================================================
        pthread_mutex_destroy(&s->mutex) ;
        pthread_cond_destroy(&s->cond) ;
        free (s) ;
   }
```

d) Écrivez le code de la fonction `pthread_semaphore_lock`

```
// ============================================================
    void pthread_semaphore_lock(t_semaphore *s) {
// ============================================================
//  Suspend le thread appelant si le compteur de s est <= 0
//  décrémente le compeur de s sinon
// ============================================================
// Le compteur du sémaphore doit être accédé en section critique
        pthread_mutex_lock(&s->mutex) ;
        while (s->compteur <= 0) {
              pthread_cond_wait (&s->cond,&s->mutex) ;
        }
        s->compteur-- ;
        pthread_mutex_unlock(&s->mutex) ;
   }
```

e) Écrivez le code de la fonction `pthread_semaphore_unlock`

```
// ============================================================
    void pthread_semaphore_unlock(t_semaphore *s) {
// ============================================================
// Incremente le compteur de s et libère les éventuels
// threads suspendus sur s
// ============================================================
// Le compteur du sémaphore doit être accédé en section critique
        pthread_mutex_lock(&s->mutex) ;
        s->compteur++ ;
        pthread_cond_broadcast (&s->cond) ;
        pthread_mutex_unlock (&s->mutex) ;
   }
```

Question 4 (6 points – 30 mn)

La commande `pcp n FS FD` crée plusieurs processus afin de **paralléliser** la copie du fichier `FS` dans le fichier `FD`.

Le paramètre `n` représente le degré de parallélisme souhaité par l'utilisateur : `n` = min (`n`, *taille `FS` en octet*) avec `n` > 0.

Cahier des charges :

- Chaque processus copie dans `FD` une partie de `FS` correspondant à *taille `FS`*/`n` octets.
- Le fichier `FD` ne doit pas exister au préalable.
- Aucune synchronisation ne sera utilisée entre les processus.
- Aucune optimisation de la taille du buffer d'entrée/sortie ne doit être effectuée.

Ecrivez en C le code de la commande `pcp`. **Rappel** : si plusieurs processus héritent d'un fichier déjà ouvert alors ils partagent le même curseur de lecture/écriture sur ce fichier.


```
/*============================================================*/
  void main(int argc, char *argv[]){
/*============================================================*/
    int fds, fdd, r, w, i, j, n, pid, offset, nb_octet, nb_processus, nb_octet_processus;
    char c;

    if (argc != 4){
        fprintf (stderr, "usage %s n fichier-source fichier_destination\n);
        exit(EXIT_FAILURE)
    }
    else {
        fds=open(argv[2], O_RDONLY);
// Récupération de la taille en octet du fichier à copier (FS)
        nb_octet=lseek(fds, 0, SEEK_END);

// Création du fichier destination (FD) et vérification qu'il n'existe pas
        fdd=open(argv[3], O_CREAT|O_EXCL|O_WRONLY, 0600);
        close (fds);
        close (fdd);

// Calcul du nombre de processus et du nombre d'octets que chaque
// processus doit écrire
        n = atoi(argv[1]); // Degré de parallélisme souhaité par l'utilisateur
        if (nb_octet <= n) { nb_processus = nb_octet;
                             nb_octet_processus = 1;
        }
        else { nb_processus = n;
               nb_octet_processus = (nb_octet/nb_processus) + 1;
        }

// le 1er processus commence à écrire à partir de la position 0 (offset)
// du fichier destination (FD)
        offset = 0;

        for (i=0 ; i<nb_processus; i++ ){
            pid=fork ();
            if (pid == 0) {// Fils
// Les fichiers FS et FD doivent être ouverts par chaque processus fils afin qu'ils
// ne partagent pas le même curseur de lecture/écriture sur chacun de
// ces fichiers
                fds=open(argv[2], O_RDONLY);
                lseek(fds, offset, SEEK_SET);
                fdd=open(argv[3], O_WRONLY);
                lseek(fdd, offset, SEEK_SET);

                r=read (fds, &c, 1);
                for (j = 0; (j < nb_octet_processus) && (r>0); j++){
                    write (fdd, &c, r);
                    r=read (fds, &c, 1);
                }
                exit(EXIT_SUCCESS);
            } // if (pid == 0)
// Calcul de la position (offset) de début d'écriture dans fichier destination (FD)
// pour le processus suivant
            offset+=nb_octet_processus;
        }// for

    }// if
} /* main() */
```

ANNEXE

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex-attr_t *mutexattr);
mutexattr :
        PTHREAD_MUTEX_INITIALIZER
        PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
        PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP
        NULL

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
     -  cond_attr : NULL

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);


off_t lseek(int fildes, off_t offset, int whence);
          -  whence : SEEK_SET par rapport au début du fichier
                      SEEK_CUR par rapport à la position courante
                      SEEK_END par rapport à la fin du fichier
          -  offset : position par rapport à whence
Renvoie le nouvel emplacement du curseur, mesuré en octet, depuis le début du fichier
```