Examen final de PG204 - Programmation Système

Informatique 2^{eme} année 2013/2014

—*Mathieu Faverge* - mfaverge@enseirb-matmeca.fr—

Le présent examen peut être réalisé avec l'aide des documents de cours et de TDs et bien entendu doit être réalisé sans l'aide de son voisin ! La durée de l'examen est de 2h.

Nom:	Prénom:	Groupe:

▸**Exercice 1.** *Cours*

Répondez aux question suivantes de manière claire et concise *en respectant l'espace donné pour la réponse.*

1. *Après un* `fork()`*, les variables du processus père peuvent-elles être modifiées par le processus fils?*
2. *Dans le cas de zone mémoire partagée à l'aide de* `shm`*:*
   (a) *à quoi sert une clé?*
   (b) *quelle est la différence entre le détachement d'une zone mémoire (fonction* `shmdt`*) et la commande de destruction d'une zone mémoire (fonction* `shmctl(shmid,IPC_RMID,NULL)`*)?*
3. *A propos des processus:*
   (a) *A quoi sert la fonction* `wait`*?*
   (b) *A quoi sert la fonction* `pthread_join`*?*
   (c) *Hormis le fait d'opérer sur des objets différents, que peut faire la fonction* `pthread_join`*, et qui n'est pas faisable avec* `wait`*?*
4. *En quoi un thread est considéré comme un processus* léger *par rapport à un processus classique dit* lourd*?*

▸**Exercice 2.** *Signaux*

On considère le code suivant les questions.

1. *A quoi sert l'instruction* `signal(SIGUSR1, g)` *?*
2. *A quoi sert l'instruction* `kill(getpid(), SIGUSR1)` *?*
3. *Quels sont les affichages possibles de ce programme ? Justifiez chacun de ces affichages.*
4. *Que se passe-t-il si on inverse les instructions* `alarm(aleatoire())` *et* `sleep(aleatoire())` *?*

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

int aleatoire();
void f();
void g();

int main(int argc, char *argv[])
{
  srand(time(NULL)); // init aleatoire
  signal(SIGALRM, f);
  signal(SIGUSR1, g);
  alarm(aleatoire());
  sleep(aleatoire());
  kill(getpid(), SIGUSR1);
  exit(EXIT_SUCCESS);
}
```

```c
// entier aleatoire entre 0 et 2 (inclus)
int aleatoire()
{
  float max = 3;
  int nombre = (int)( max * rand() / RAND_MAX);
  return(nombre);
}

void f()
{
  printf("A\n");
  exit(EXIT_SUCCESS);
}

void g()
{
  printf("S\n");
  exit(EXIT_SUCCESS);
}
```

▸**Exercice 3.** *Fork, pthread et mémoire*

1. *Dans ce programme, certaines données sont transmises entre processus. Précisez dans quel sens sont transmises ces données : du père vers le fils, du fils vers le père, ou entre fils ?*

2. *Quel mécanisme est utilisé pour transmettre ces données ?*

3. *Que renvoie la fonction* `calculer()` *? Décrivez le fonctionnement de cette fonction.*

4. *Dans ce programme, précisez dans quelle zone de la mémoire virtuelle se trouve la variable* `alea`*. Est-ce une variable partagée et y a-t-il accès concurrent à cette variable* `alea` *? Si oui, précisez à quelle ligne, et proposez une modification du code pour y remédier. Si non, justifiez votre réponse.*

5. *Proposez une solution qui utilise des processus légers (*`threads` *à la place des processus lourd qui permettent d'obtenir le même résultat.*

```c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 /* genere un entier aleatoirement
7    entre 0 et 9 (inclus) */
8 int aleatoire() {
9   srand(time(NULL)+getpid());
10  float max=10;
11  int nombre = (int)( max * rand() / RAND_MAX);
12  return(nombre);
13 }
14
15 int main(int argc, char *argv[]) {
16   printf("resultat = %d\n", calculer());
17   exit(EXIT_SUCCESS);
18 }
19
20 int calculer() {
21   int alea = aleatoire();
22   int i;
23   if (alea<5) {
24     int m[2];
25     pipe(m);
26     for (i=0; i<2; i++) {
27       int r;
28       if ((r = fork()) < 0) {
29         fprintf(stderr,"Erreur fatale : fork()\n");
30         exit(EXIT_FAILURE);
31       }
32       if (r == 0) {
33         close(m[0]);
34         int alea2 = calculer();
35         write(m[1], &alea2, sizeof(int));
36         close(m[1]);
37         exit(EXIT_SUCCESS);
38       }
39     }
40     close(m[1]);
41     for (i=0; i<2; i++) {
42       int lu;
43       read(m[0], &lu, sizeof(int));
44       alea += lu;
45     }
46     close(m[0]);
47     for (i=0; i<2; i++) {
48       int status;
49       wait(&status);
50     }
51   }
52   return alea;
53 }
```

▸**Exercice 4.** *Mémoire partagée*

On utilise un protocole producteur/consommateur vu en TD pour traiter un ensemble de données dont le code des deux programmes principaux est donné ci-dessous. Il s'agira de deux processus distinct: le premier exécute la fonction `produire()` *qui permet de générer les objets à traiter/stocker, le second les consomme une fois produits. Chacun d'eux est multi-threadé pour une meilleure efficacité.*

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

#define bufferSz 1024
int buffer[bufferSz];

pthread_mutex_t *mutexIn;
pthread_mutex_t *mutexOut;

sem_t *semNonPlein;
sem_t *semNonVide;

void *produce(void* arg) {
    while( 1 ) {
        sem_wait( semNonPlein );
        pthread_mutex_lock( &mutexIn );
        buffer[in] = produire_1_element();
        in = (in+1) % bufferSz;
        pthread_mutex_unlock( &mutexIn );
        sem_post( semNonVide );
    }
}

void *consume(void* arg) {
    int data;
    while( 1 ) {
        sem_wait( semNonVide );
        pthread_mutex_lock( &mutexOut );
        data = buffer[out];
        out = (out+1) % bufferSz;
        pthread_mutex_unlock( &mutexOut );
        sem_post( semNonPlein );
        consommer( data );
    }
}
```

```
int main(int argc, char *argv[]) {
    pthread_t tid[2];
    int producer = atoi(argv[1]) == 0;
    int pid;

    mutexIn  = malloc(sizeof(pthread_mutex_t));
    mutexOut = malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(&mutexIn);
    pthread_mutex_init(&mutexOut);

    semNonPlein = sem_open("semNonPlein",
                           O_CREAT, 0666,
                           bufferSz );
    semNonVide  = sem_open("semNonVide",
                           O_CREAT, 0666, 0);

    if (producer) {
        for(int i=0; i<2; i++) {
            pthread_create(&tid[i], NULL,
                           produce, NULL);
        }
    } else {
        for(int i=0; i<2; i++) {
            pthread_create(&tid[i], NULL,
                           consume, NULL);
        }
    }

    for(int i=0; i<2; i++) {
        pthread_join( &tid[i], NULL );
    }

    pthread_mutex_destroy( mutexIn );
    pthread_mutex_destroy( mutexOut );
}
```

1. **En ne se préoccupant que des fonctions produce and consume**, *expliquez l'utilité de chacun des deux sémaphores et des deux mutexes dans cette solution. A quoi correspond la valeur de chacun d'eux pendant l'exécution du programme ?*

2. *Peut-il y avoir inter-blocage entre les différents threads du processus producteur (de même pour le consommateur) ? Répondez par par oui ou par non, et si oui, donnez un exemple.*

3. *En pratique, on lance un processus pour le producteur et un pour le consommateur. Est-ce que le buffer, les mutexes et les sémaphores sont partagés entre les* **threads d'un même processus***? Justifiez votre réponse pour chacune de ces trois données.*

4. *En pratique, on lance un processus pour le producteur et un pour le consommateur. Est-ce que le buffer, les mutexes et les sémaphores sont partagés entre les* **différents processus***? Justifiez votre réponse pour chacune de ces trois données.*

5. *Si parmi ces données certaines ne sont pas partagées entre les processus, donnez deux solutions qui permettent de corriger cette lacune. Donnez l'implémentation de l'une d'entre elle. Vous ne recopierez pas tout le code, mais donnerez seulement les parties modifiées.*

6. *un filtre est souvent appliqué entre le producteur et le consommateur. On supposera qu'un seul processus mono-thread a le rôle de filtre. Modifiez les routines* `produce()` *et* `consume()` *ci-dessus, et donnez la*

routine `filtre()` *pour mette en œuvre les 3 types de processus (producteur, consommateur, filtre) tels que :*

- *les* **producteurs** *jouent le même rôle qu'avant : ils écrivent dans le buffer dès que possible*
- *le* **filtre** *teste les objets produits par les producteurs :*
  - *si le test est ok, alors l'objet peut être consommé par les consommateurs,*
  - *sinon, l'objet est remplacé par un nouveau via la fonction* `produire_1_element()`*, jusqu'à obtenir un objet qui réussisse le test*

  On dispose pour cela d'une fonction `test(obj)` *prenant en compte un objet, et retournant* 1 *si le test réussit, et* 0 *sinon.*
- *les* **consommateurs** *ne consomment que les objets ayant passé avec succès le test du filtre.*

Note: Vous repartirez du code de l'énoncé qu'il ait été modifié ou non par la question 3, et en considérant que le buffer, les mutexes et les sémaphores sont partagés entre les processus.

7. *Que faut-il modifier dans votre solution précédente pour que plusieurs* **processus** *filtres (effectuant le même test via la fonction* `test`*) puissent opérer en parallèle ?*