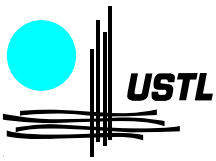

Licence d'informatique
Module de Programmation des systèmes

Examen première session 2006

Philippe MARQUET

Juin 2006

Durée : 3 heures.
Documents de cours et TD autorisés.
Les réponses seront concises et concrètes.

1 Questions de cours et TP

Exercice 1

1. Qu'est-ce qu'un processus zombi ?
2. Pourquoi toute tentative d'utilisation de l'appel système `kill()` sur un processus zombi est-elle vouée à l'échec ? □

Exercice 2

1. Quelle propriété distingue un verrou récursif d'un verrou traditionnel ?
2. Explicitez une situation nécessitant l'emploi de verrous récursifs. □

2 `psem` : des sémaphores avec des tubes

Un sémaphore est l'association d'un compteur et d'une file d'attente d'activités en attente. Un programmeur (débutant ?) se propose de construire une bibliothèque de manipulation de sémaphores à l'aide de tubes anonymes :

- une opération de notification sur le sémaphore est réalisée par une simple écriture d'un caractère dans le tube ;
- une opération d'attente sur le sémaphore est réalisée par une lecture d'un caractère depuis le tube.

Le principe de base d'une telle implantation est de considérer que le tube contient autant de caractères que la valeur du compteur du sémaphore.

L'interface de cette bibliothèque est donnée par le fichier `psem.h` suivant :

```
/* ------------------------------
   Basic semaphore interface
   ------------------------------------------------------------*/

#ifndef _PSEM_H_
#define _PSEM_H_

typedef struct psem_s psem_t;

/* allocation + initialisation ; deletion */
psem_t *psem_new(int);
void psem_free(psem_t *);

/* P & V */
```

```
int psem_p(psem_t *);
int psem_v(psem_t *);

#endif
```

Les quelques exercices suivants consistent à fournir une implémentation de cette interface et à discuter de la pertinence de l'approche.

Exercice 3
En ayant remarqué qu'un sémaphore était implémenté par un simple tube, donnez le code C d'une définition du type `structpsem_s`. □

Exercice 4
L'initialisation d'un tube consiste à allouer dynamiquement une structure `psem_t`, à créer un tube et à écrire dans ce tube d'un nombre de caractères égal au nombre initial de jetons associés au sémaphore. Donnez une implémentation de la fonction `psem_new()`. □

Exercice 5
La primitive `psem_p()` consiste simplement à lire un caractère depuis le tube. Expliquez pourquoi cette lecture ne retournera pas 0 mais sera bloquante si le tube est vide. □

Exercice 6
On désire que la destruction d'un sémaphore suite à un appel à la fonction `psem_free()` fasse en sorte que d'éventuels processus bloqués en attente sur le sémaphore (donc en lecture sur le tube) puisse être débloqués et que la primitive `psem_p()` se termine alors sur une erreur.

On réalise donc une implémentation de `psem_free()` qui comporte un appel à `close()` sur le tube faisant en sorte que la lecture sur le tube retourne zéro.

On remarque que le comportement attendu de terminaison de la primitive `read()` sur une erreur ne se produit pas.

Expliquez pourquoi. □

Exercice 7
Une des motivations de cette implantation est la possibilité de partager des sémaphores entre des processus. Expliquez ce en quoi l'utilisation de cette bibliothèque pour partager des sémaphores entre des processus légers POSIX (*threads)* peut être problématique. □

3 `cphole` : copie de fichiers creux

L'utilisation des fichiers à accès direct permet le positionnement du curseur (position courante) dans le fichier au delà de la fin du fichier. Le « trou » laissé au delà de l'ancienne fin de fichier par ce déplacement contenant exclusivement des zéros. L'implémentation habituelle est de ne pas écrire ces zéros sur le disque, on parle de « fichiers creux ».

Plus précisément les fichiers étant alloués sur le disque sous la forme d'une suite de blocs de caractères, on évite d'allouer les blocs pleins de zéros. La taille `BLKSIZE` de ces blocs est donnée par le champ `st_blksize` de la structure `structstat` retournée par les appels `stat()`, `lstat()`, et `fstat()`. Un « bloc plein de zéros » est donc un bloc de `BLKSIZE` octets nuls dont l'adresse du premier octet est un multiple de `BLKSIZE`.

On désire écrire une commande `cphole`, variante de la commande `cp` :

```
cphole filein fileout
```

réalisant la copie du fichier `filein` en un fichier `fileout`, ce fichier `fileout` étant constitué d'autant de trous que possible.

Dans ses grandes lignes, la commande `cphole` consiste à
- créer le fichier destination et récupérer la taille `BLKSIZE` qui lui est propre ;
- lire le fichier source par blocs de `BLKSIZE` octets. Pour chaque bloc :

  - si ce bloc est plein de zéros, déplacer la position courante dans le fichier destination,
  - sinon, écrire ce bloc dans le fichier destination ;
- fermer les accès aux fichiers.

Cette commande sera implantée par une fonction

```
static int copy_file_hole(const char *src, const char *dst);
```

les chaînes de caractères `src` et `dst` identifiant les noms des fichiers source et destination de la copie. Cette fonction retourne -1 en cas d'erreur.

On fournit la fonction utilitaire suivante :

```
int bnull(const void *b, size_t len);
```

qui retourne vrai si et seulement si les `len` octets à partir de l'adresse `b` sont nuls.

Exercice 8
Donnez les déclarations et le code du début de la fonction `copy_file_hole()` consistant en l'ouverture des deux fichiers et l'initialisation d'une variable `BLKSIZE`. On vérifiera aussi que les opérations de déplacement du curseur sont légales dans le fichier destination. □

Exercice 9
Donnez le code de la boucle principale de la fonction `copy_file_hole()` itérant sur les lectures du fichier source. □

Exercice 10
Supposant que le code réalisant la fermeture des deux fichiers consiste en de simples appels à `close()`, on remarque que parfois la taille apparente du fichier destination est différente de celle du fichier source. Expliquez comment cela est possible. □

Exercice 11
Expliquez comment éviter le phénomène observé à l'exercice 10 et donnez le code de finalisation de la fonction `copy_file_hole()` évitant ce problème. □

Exercice 12
Donnez les grandes lignes d'un scénario permettant de tester à minima votre commande `cphole`. Identifiez une suite de commandes Unix permettant de mettre en œuvre ce test. □

4 Trace de signaux

Soit le programme `sigxam06.c` suivant :

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

#define MAXWAIT    8192

static int count = 0;
static pid_t child_pid;         /* 0, FALSE for the child process */

static void
pr_nsig(int nsig, char *adjective)
{
    printf("%s (pid %d) %d signaux %s\n",
```

```
            child_pid ? "pere" : "fils",
            getpid(),
            nsig,
            adjective);
}

static void
countin(int sig)
{
    int i;

    count++;

    if(! child_pid)
        for(i=0; i<MAXWAIT; i++)
            ;
}

static void
pr_stat(int sig)
{
    pr_nsig(count, "recus");

    if(child_pid)
        wait((int *)0);

    exit(EXIT_SUCCESS);
}

int
main (int argc, char *argv[])
{
    struct sigaction sa;
    int nsig, i;
    pid_t self = getpid();

    nsig = atoi(argv[1]);

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = countin;
    sigaction(SIGUSR1, &sa, (struct sigaction *) 0);

    sa.sa_handler = pr_stat;
    sigaction(SIGUSR2, &sa, (struct sigaction *) 0);

    switch((child_pid = fork())) {
        case 0:
            for(;;)
                pause();
            break;
        default :
            for(i=0; i<nsig; i++) {
                kill(child_pid, SIGUSR1);
```

```
                kill(self, SIGUSR1);
            }

            kill(child_pid, SIGUSR2);

            pr_nsig(nsig, "envoyes");

            kill(self, SIGUSR2);
        }
        exit(EXIT_FAILURE);
    }
```

Que l'on compile par la commande

```
% make sigxam06
gcc -Wall -Werror -ansi -pedantic -D_XOPEN_SOURCE=500 -g sigxam06.c -o sigxam06
```

Exercice 13
Expliquez à quel moment les fonctions `countin()` et `pr_stat()` sont appelées. □

Exercice 14
Une exécution produit le résultat suivant

```
% ./sigxam06 1000
fils (pid 8390) 776 signaux recus
pere (pid 8389) 1000 signaux envoyes
pere (pid 8389) 1000 signaux recus
```

Expliquez pourquoi il apparaît que le processus fils n'a reçu que 776 signaux. □

Exercice 15
Expliquez comment se termine ce programme :

1. Comment se termine le processus fils ?
2. Comment se termine le processus père ?
3. Quel est le processus qui termine en premier ? Pourquoi ?
4. Dans quel état passe ce processus terminant en premier ?
5. Que se passe-t-il à la terminaison du second processus ? □

On ajoute au code de `sigxam06.c` la fonction `halt()` suivante :

```
static void
halt(int sig)
{
    pr_nsig(count, "reçus");
    printf("%s (pid %d) termine sur interruption\n",
           child_pid ? "pere" : "fils",
           getpid());
    exit(EXIT_FAILURE);
}
```

Exercice 16
On désire qu'à la saisie au clavier d'un caractère `<intr>` (`control-C`), le processus père et le processus fils se terminent tous deux par un appel à cette fonction `halt()`.

Donnez le code C devant être écrit pour ce faire. □

Fin du sujet