Licence d'informatique
Module de Programmation des systèmes

Examen seconde session 2007

Philippe MARQUET

Juin 2007

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

1 Supprimer un zombi

Exercice 1
Un utilisateur désire supprimer un processus zombi de PID 666 qui exécute la commande `./zz`.
Pour chacune des actions suivantes,

- expliquez ce que l'utilisateur a voulu faire ;
- indiquez si cette action permet de supprimer le processus zombi ;
- expliquez pourquoi.

1. Première action, `kill -KILL` :

```
% kill -KILL 666
```

2. Seconde action, `kill 396` :

```
% ps -j
USER   PID  PPID STAT  TT       TIME COMMAND
phm    396   393 Ss    p1    0:00.54 -tcsh
phm    666   396 Z     p1    0:00.02 ./zz
[...]
% kill -KILL 396
```

3. Troisième action, `kill -CONT` :

```
% kill -CONT 666
```

4. Quatrième action, `kill 668 669` :

```
% ps -j | grep 666
USER   PID  PPID STAT  TT       TIME COMMAND
phm    666   396 Z     p1    0:00.02 ./zz
phm    668   666 R     p1    0:00.01 zz
phm    669   666 S     p1    0:00.01 zz
phm    681   396 S     p1    0:00.01 grep 666
% kill -KILL 668 669
```

2 `bulkiest` : trouver le plus encombrant

Il s'agit de développer une commande `bulkiest` dont l'objet est d'afficher l'entrée du répertoire courant et de ses sous-répertoires occupant le plus de place sur le disque.

Le nom et la taille de cette entrée recherchée seront mémorisées dans les variables suivantes au fur et à mesure du parcours des entrées du répertoire et de ses sous-répertoires :

```
/* l'entrée de taille maximale */
static char bulkiest_path[PATH_MAX+1];
static int  bulkiest_size = 0;
```

Exercice 2
Donnez le code d'une fonction

```
void check_if_bulkier (const char *path, struct stat *st)
```

qui, étant donnés un nom d'entrée et une valeur de type `struct stat` correspondante, met à jour les variables `bulkiest_path` et `bulkiest_size` si l'entrée désignée par les paramètres est plus encombrante que l'entrée la plus encombrante trouvée jusqu'à maintenant. □

Exercice 3
Donnez le code d'une fonction

```
void find_bulkiest_file(const char *path);
```

qui vérifie si l'entrée `path` est la plus encombrante. De plus, dans le cas d'un répertoire, cette fonction vérifie pour chacune des entrées de ce répertoire, et ce récursivement, si elle est la plus encombrante. □

On donne ici le `main()` de la commande `bulkiest` qui consiste en un simple appel à cette dernière fonction suivi d'un affichage du résultat :

```
int
main (int argc, char *argv[])
{
    /* on démarre de . */
    find_bulkiest_file(".");

    /* affichage terminal */
    printf("%s de taille %d\n", bulkiest_path, bulkiest_size);

    exit(EXIT_SUCCESS);
}
```

3 `syncrw` : lecteurs rédacteurs

Il s'agit de mettre en œuvre l'algorithme de synchronisation entre lecteurs et rédacteurs vu en cours pour des activités concurrentes implémentées par des processus légers. Soit le squelette de programme de la figure 1. Les processus légers lecteurs et écrivains se partagent les accès aux fichiers `files[]`. On désire assurer la cohérence de ce partage en assurant aux processus légers écrivains des accès exclusifs à chacun des fichiers.

Exercice 4
Donnez la liste des fonctions d'une bibliothèque `syncrw` (synchronisation de lecteurs/rédacteurs) qui serait en charge d'assurer cette cohérence. □

Exercice 5
Modifiez le squelette de programme pour insérer les nécessaires appels à cette bibliothèque de synchronisation. (Il n'est pas demandé de compléter les `...` du squelette de code.) □

Exercice 6
Identifiez les structures de données nécessaires à une implémentation de cette bibliothèque de synchronisation. □

```
#define NFILES          128

/* les données lues/écrites dans les fichiers */
struct data_s {
    ...
};

/* les NFILES fichiers */
struct file_s {
    ...
};
static struct file_s files[NFILES];

/* un lecteur */
void *
reader(void *arg)
{
    int i = (int) arg;
    struct data_s data;
    int status;

    do {
        status = read_data(files+i, &data);
        do_things(&data);
    } while (status);
}

/* un rédacteur */
void *
writer(void *arg)
{
    int i = (int) arg;
    struct data_s data;
    int status;

    do {
        data = ...;
        status = write_data(files+i, &data);
    } while (status);
}

int
main(int argc, char *argv[])
{
    int nreader = ...;
    int nwriter = ...;
    pthread_t *tids = malloc(...);
    int i;

    /* création des threads */
    for (i=0; i<nreader; i++)
        pthread_create(tids+..., NULL, reader, (void *)(i%NFILES));
    for (i=0; i<nwriter; i++)
        pthread_create(tids+..., NULL, writer, (void *)(i%NFILES));

    /* attente de la terminaison des threads */
    ...
}
```

FIG. 1 – Squelette de lecteurs/rédacteurs

Arguments de la ligne de commande

La bibliothèque `makeargv` fournit deux fonctions de création et destruction d'arguments :

```
extern int makeargv(const char *s, const char *delimiters, char ***argvp);
extern void freeargv(char **argv);
```

Vous vous inspirerez de l'exemple suivant `makeargv-main.c` pour utiliser cette bibliothèque :

```
#include "makeargv.h"

int
main (int argc, char *argv[])
{
    int i, status;

    for (i=1; i<argc; i++) { /* traiter argv[i] */
        char **cmdargv;
        char **arg;

        /* cr^^c3^^a9ation du argv de l'argument i */
        status = makeargv(argv[i], " \t", &cmdargv);
        assert(status>0);

        /* test: affichage */
        fprintf(stderr, "[%s]\t%% ", cmdargv[0]);
        for (arg=cmdargv; *arg; arg++)
            fprintf(stderr, "%s ", *arg);
        fprintf(stderr, "\n");

        /* lib^^c3^^a9ration m^^c3^^a9moire */
        freeargv(cmdargv);
    }

    exit(EXIT_SUCCESS);
}
```

Une exécution de ce programme d'exemple est la suivante :

```
% ./makeargv-main "cat -n" "grep -v foo" wc
[cat]   % cat -n
[grep]  % grep -v foo
[wc]    % wc
```

Exercice 7
Fournissez une implémentation des fonctions principales de cette bibliothèque de synchronisation. □

Exercice 8
L'algorithme de synchronisation proposé dans le cours donne la priorité aux lecteurs. Des alternatives telles priorités aux rédacteurs ou priorités égales sont mentionnées. Explicitez les différences entre ces différentes variantes. Illustrez par un exemple concret. □

4 `rptr2`: bis repetita dans des tubes

Il s'agit de développer une commande `rptr2` qui va répéter son entrée standard sur les entrées standard d'un ensemble de n commandes.

Cette commande `rptr2`

– crée n tubes ;

– lance n commandes qui vont chacune lire depuis un de ces tubes ;
– répète dans chacun de ces tubes ce qu'elle lit depuis son entrée standard.

Les n commandes lancées par `rptr2` peuvent accepter des paramètres comme illustré par l'exemple suivant :

```
% rptr2 "cat -n" "grep -v foo" wc
```

On utilisera la bibliothèque `makeargv` vue en TP pour gérer les paramètres de la ligne de commande. Un exemple est donné dans l'encart page précédente.

Exercice 9
Expliquez la hiérarchie de processus qui doit être créée pour réaliser la commande `rptr2`. Explicitez quel processus doit se charger de la création des tubes. Précisez bien quels descripteurs inutiles doivent être fermer par chacun des processus. □

Nous allons utiliser le tableau `cmds` pour mémoriser les informations des `ncmd` commandes lancées par `rptr2`, à savoir :
– le numéro du processus en charge de la commande ;
– le descripteur d'écriture dans le tube sur lequel lit la commande.

```
struct cmd_s {
    pid_t cmd_pid;
    int cmd_pipew;
};
static struct cmd_s *cmds;
static int ncmd;
```

Exercice 10
Complétez les `[...]` du squelette suivant de la fonction `main()` de `rptr2`

```
int
main (int argc, char *argv[])
{
    /* allocation du tableau de cmd_s */
    ncmd = argc-1;
    cmds = malloc(ncmd * sizeof(struct cmd_s));
    assert(cmds);

    /* création des tubes et fils */
    [...]
}
```

pour
– créer les tubes nécessaires ;
– créer les processus en charge de l'exécution des commandes ;
– assurer que l'entrée standard de ces processus se fasse depuis leur tube ;
– réaliser la mutation de ces processus pour l'exécution de la commande correspondante. □

La commande `rptr2` se doit ensuite de répéter ce qu'elle lit sur son entrée standard dans chacun des tubes la connectant aux commandes. À la rencontre de la fin de fichier sur son entrée standard, la commande `rptr2` se termine.

Exercice 11
Complétez la fonction `main()` pour réaliser cette recopie de l'entrée standard dans les tubes connectés aux commandes. □

Nous savons que l'écriture dans un tube pour lequel le nombre de lecteurs est nul provoque l'envoi du signal `SIGPIPE`. Ainsi il va être possible à la commande `rptr2` d'identifier qu'un

processus en charge de l'exécution d'une commande a terminé (ou a fermé le tube, ce qui est équivalent du point de vue de `rptr2`).

Si toutes les commandes lancées se terminent avant la fin de `rptr2`, il n'y a plus lieu de continuer.

Exercice 12
Il s'agit donc d'identifier la terminaison des commandes lancées. Fournissez le complément à l'implantation actuelle de `rptr2` pour mettre fin à l'exécution de `rptr2` à la fermeture de tous les tubes dans lesquels `rptr2` écrit. □

Fin du sujet