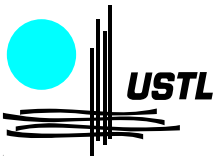

Licence d'informatique
Module de Programmation des systèmes

Examen première session 2007

Philippe MARQUET

Mai 2007

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

1 Questions de cours

Exercice 1
On désire éliminer un processus zombi. Pour chacune des actions suivantes
- indiquez si elle résout le problème ;
- expliquez pourquoi.

1. On le tue ; c'est-à-dire, on lui envoie le signal `SIGKILL`.
2. On tue son père.
3. On le fait continuer ; c'est-à-dire, on lui envoie le signal `SIGCONT`.
4. On tue tous ses fils. □

Exercice 2
Lors de la création d'un fichier par la primitive `open()`, un paramètre de type `mode_t` précise les droits d'accès qui seront associés au fichier. Par ailleurs, la création de fichiers utilise le masque de droits du processus défini par `umask`.

1. Expliquez comment ces deux définitions de droits se conjuguent.
2. Expliquez pourquoi ces deux définitions de droits coexistent. □

2 `intrus` : lister les intrus

Il s'agit de développer une commande `intrus` dont l'objet est de lister les entrées sous le répertoire courant qui n'appartiennent pas à l'utilisateur courant.

En sus des primitives vues en cours, on utilisera la fonction POSIX

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

uid_t getuid(void);
```

qui retourne le numéro d'utilisateur de l'utilisateur courant.

Exercice 3
Donnez le code d'une fonction

```
int intrus(struct stat *st);
```

qui, étant donnée une valeur de type `struct stat`, vérifie que l'entrée dans le système de fichier correspondant appartient bien à l'utilisateur courant. La fonction `intrus()` retourne un booléen : une valeur `TRUE` si l'entrée est un intrus, une valeur `FALSE` sinon. □

Exercice 4
Donnez le code d'une fonction

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

qui vérifie que l'entrée de nom `path` n'est pas un intrus. De plus, dans le cas d'un répertoire, cette fonction vérifie pour chacune de ses entrées, et ce récursivement, qu'il ne s'agit pas d'un intrus. Typiquement, la fonction `main()` de la commande `intrus` consisterait en un simple appel à `intrus_file(".")`. □

3 `syncrwe` : lecteurs rédacteurs à priorité égale

Nous avons vu en cours un algorithme de synchronisation des accès à un fichier par des lecteurs et rédacteurs. Il s'agit ici de modifier la priorité pour donner une priorité égale aux lecteurs et aux rédacteurs et de mettre en œuvre cet algorithme pour 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 5
Donnez la liste des fonctions d'une bibliothèque `syncrwe` (synchronisation de lecteurs/rédacteurs) qui serait en charge d'assurer cette cohérence. □

Exercice 6
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 7
Explicitez ce que signifie « priorité égale » entre les lecteurs et les rédacteurs, en opposition à une priorité aux lecteurs ou aux rédacteurs. □

Exercice 8
Identifiez les structures de donnés nécessaires et donnez les algorithmes des fonctions principales de la bibliothèques de synchronisation pour assurer cette priorité égale. □

Exercice 9
Fournissez une implémentation des fonctions principales de cette bibliothèque de synchronisation à priorité égale. □

4 `rptr`: répétons dans des tubes

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

Cette commande `rptr`
- 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 `rptr` peuvent accepter des paramètres comme illustré par l'exemple suivant :

```
% rptr "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 4.

```
#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 10
Expliquez la hiérarchie de processus qui doit être créée pour réaliser la commande `rptr`. Explicitez ce que doit faire chacun des processus. En particulier notez bien les créations de tubes et les fermetures de descripteurs inutiles. □

Nous allons utiliser le tableau `cmds` pour mémoriser les informations des `ncmd` commandes lancées par `rptr`, 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 11
Complétez les `[...]` du squelette suivant de la fonction `main()` de `rptr`

```
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 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 `rptr` 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 `rptr` se termine.

Exercice 12
Complétez la fonction `main()` pour réaliser cette recopie de l'entrée standard dans les tubes connectés aux commandes. □

Une commande lancée peut se terminer avant la fin de `rptr`, par exemple sur une erreur ou parce que la commande a trouvé ce qu'elle cherchait sur l'entrée standard. Si toutes les commandes lancées se terminent avant la fin de `rptr`, il n'y a plus lieu de continuer.

Exercice 13
Il s'agit donc d'identifier la terminaison de ces commandes lancées. Deux choix sont possibles :
- utilisation de `wait()` ou `waitpid()` pour attendre la fin d'un processus fils ;
- définition d'une fonction traitant de signal pour le signal de terminaison d'un processus fils.

Expliquez et motivez , dans le cas présent de la commande `rptr`, le choix le plus judicieux pour détecter la terminaison des processus fils. □

Exercice 14
Fournissez le complément à l'implantation actuelle de `rptr` pour mettre fin à l'exécution de `rptr` à la terminaison de toutes les commandes lancées.

On pourra utiliser une valeur nulle pour le champs `cmds_pid` de `struct cmd_s` pour indiquer que le processus est terminé. □

Il peut être objecté que plutôt que la terminaison des processus fils, c'est la fermeture de tous les tubes qui devrait entraîner la fin de la commande `rptr`.

Exercice 15
Donnez les avantages et inconvénients de cette autre détection de la terminaison sur celle déjà réalisée de la terminaison des fils. □

Fin du sujet