Licence d'informatique
Module de Programmation des systèmes

Examen seconde session 2006

Philippe MARQUET

Septembre 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 orphelin ?
2. Pourquoi un processus orphelin ne peut-il devenir zombi ? □

Exercice 2

Un manuel de programmation système met en garde sur l'utilisation conjointe de verrous et de signaux.

1. Expliquez ce qui motive cette mise en garde.
2. Identifiez une situation dans laquelle il est commode d'utiliser verrous et signaux. □

2 `mpipe`: tube à écritures multiples

On va développer une commande

```
mpipe command_1 command_2 ... command_n
```

qui exécute simultanément et indépendamment les n commandes désignées. On notera que dans cette version préliminaire, les commandes ne prennent pas d'arguments. Les entrées/sorties standard de ces commandes sont redirigées comme suit :

- les $n-1$ premières commandes produisent leur résultat dans un unique tube ;
- la dernière commande prend ses entrées depuis ce tube.

L'exécution de la commande `mpipe` prend fin à la terminaison des n commandes. Cette terminaison est un succès si et seulement si l'ensemble des commandes ont terminé sur un succès.

Exercice 3

Identifiez les grandes étapes d'une implantation de la commande `mpipe`. En particulier explicitez bien les fermetures de descripteurs nécessaires à la bonne terminaison des n commandes. □

On considère le code suivant :

```
static int ncmd;                        /* n, number of commands */
static int rstatus = EXIT_SUCCESS;      /* mpipe status */

int
main(int argc, char *argv[])
{
    ncmd = argc-1;
```

```
    ...

    exit(rstatus);
}
```

Exercice 4
Complétez le code de la fonction `main()` pour mettre en œuvre votre proposition faite à l'exercice précédent. □

3 `cnthole` : comptage 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 sont alloués sur le disque sous la forme d'une suite de blocs de caractères. 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()`. Pour chaque bloc, l'adresse du premier octet est un multiple de `BLKSIZE`. Généralement, on évite d'allouer les bloc plein de zéros, c'est-à-dire les blocs qui contiennent `BLKSIZE` octets nuls. Dans la suite, on appellera « trou » un bloc plein de zéros non alloué.

On désire écrire une commande `cnthole` :

```
cnthole filename...
```

permettant de connaître combien de blocs seraient gagnés si les fichiers désignés par les arguments étaient constitués d'autant de trous que possible.

Dans ses grandes lignes, la commande `cnthole` consiste, pour chacun des fichiers de la ligne de commande, à

- récupérer la taille `BLKSIZE` qui lui est propre ;
- lire le fichier par blocs de `BLKSIZE` octets ;
- pour chaque bloc, si ce bloc est plein de zéros, compter un bloc gagné potentiel ;
- afficher sur la sortie standard le nombre de blocs gagnés potentiels.

Cette commande sera implantée par des appels successifs à la fonction

```
static int cnt_hole(int fd);
```

retournant le nombre de blocs pouvant être gagnés pour le fichier correspondant au descripteur `fd`. 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 5
Donnez les déclarations et le code du début de la fonction `cnt_hole()` consistant en l'initialisation d'une variable `BLKSIZE`. □

Exercice 6
Donnez le code de la boucle principale de la fonction `cnt_hole()` itérant sur les lectures du fichier source pour compter les blocs de zéros. □

Exercice 7
Donnez le code de la fonction `main()` de la commande `cnthole` itérant sur les paramètres de la ligne de commande et réalisant l'ouverture du fichier, l'appel à `cnt_hole()`, la fermeture du fichier et l'affichage du résultat. □

Exercice 8
La commande `cnthole` proposée ne calcule pas combien de blocs pourraient être gagnés pour un fichier, mais plus précisément combien de blocs seraient gagnés entre les deux implantations du fichier :

- sans aucun trou ; et
- avec autant de trous que possible.

Ce qui est différent, en particulier pour un fichier comportant déjà des trous.

Donnez les grandes lignes d'une commande qui fournirait effectivement le nombre de blocs pouvant être gagnés pour un fichier donné. □

4 `watch` : surveillance d'exécution

On désire écrire une commande `watch` qui exécute une commande donnée sous la surveillance de l'utilisateur : l'utilisateur doit saisir au clavier un caractère `<intr>` (c'est-à-dire `control-C`) à intervalle régulier. Si à l'échéance du délai le caractère n'a pas été saisi, l'exécution de la commande sera interrompue.

Sur la ligne de commande, on fournira donc la valeur de l'intervalle (en secondes), la commande à exécuter et ses paramètres :

```
watch duration command [args]...
```

Dans ses grandes lignes, la commande `watch` consiste donc à créer un processus fils qui exécutera la commande `command` ; à définir un traitant de signal qui met fin à ce processus fils et à associer ce traitant de signal au signal de temporisation `SIGALRM` ; à définir un traitant de signal pour `SIGINT` qui réinitialise le temporisateur. On utilisera la primitive

```
int alarm(unsigned seconds);
```

armant ou ré-armant le temporisateur.

On suppose que le programme définit et initialise les variables suivantes :

```
static pid_t pid;               /* the child process */
static int duration;
static char *command[];

int
main(int argc, char *argv[])
{
    assert(argc >= 3);

    duration = atol(argv[1]);
    command = argv+2;

    ...
```

Exercice 9
Soient les deux fonctions `sigint_hdlr()` et `sigalrm_hdlr()` devant respectivement être associées comme traitant de signal des signaux `SIGINT` et `SIGALRM`. Donnez la suite du code de la fonction `main()` installant ces traitants de signaux et armant le minuteur à `duration` secondes. □

Exercice 10
Donnez la suite de la fonction `main()` assurant la création du processus fils et l'exécution par celui-ci de la commande demandée. □

Exercice 11
Expliquez comment la fonction

```
static int sigalrm_hdlr();
```

va interrompre le processus fils et donnez le code de cette fonction. □

Exercice 12
Supposant que le processus père boucle sur un appel à la fonction `pause()`, expliquez comment terminer proprement le programme `watch` à la terminaison de son fils et donnez le code correspondant. □

Fin du sujet