Licence Mention Informatique, 3ème année

Programmation des systèmes
Travaux dirigés – Système de fichiers et threads

L'équipe pédagogique

Introduction

Ce document[1] est le support de travaux dirigés pour le cours de Programmation de Système, au 3ème année de Licence Informatique à l'Université de Lille. Le document a été crée par Philippe Marquet : il a évolué au cours des années avec des contributions par toute les équipes pédagogiques successives[2].

Table des matières

1 TD 1 : Opérations de base sur un fichier **3**
1.1 La commande "cat" 3
1.2 Sauvegarder des données sur un fichier 3

2 TD 2 : Implantation d'une bibliothèque d'entrées/sorties tamporisées **7**

3 TD 3 : Retrouver une commande **8**

1. Énoncé placé sous licence Creative Commons by-nc-sa (`https://creativecommons.org/licenses/by-nc-sa/3.0/deed.fr`).

2. Philippe Marquet, Gilles Grimaud, Samuel Hym, Giuseppe Lipari, et beaucoup d'autres.

4 TD 4 : Clonage de processus **9**
4.1 Compréhension du clonage simple 9
4.2 Généalogie de processus 10
4.3 Quatre fils 11
4.4 Multi-fourche 11

5 TD 5 : les tubes **13**
5.1 Redirection 13
5.2 Calcul parallèle 13
5.3 Connexion de deux processus par un tube anonyme 13
5.4 Connexion de deux commandes par un tube anonyme 14
5.5 Attendre le lancement d'une commande 14

1 TD 1 : Opérations de base sur un fichier

1.1 La commande “cat”

On propose ici d'implémenter la commande Unix `cat` pour afficher le contenu d'un fichier. Notre commande `mcat` prendra en entrée sur la ligne de commande un nom de fichier, ouvrira le fichier, lira son contenu et l'affichera sur la sortie standard.

Question 1 *Utiliser les appels de la librairie standard.* Dans ce première exercice, on utilisera la fonction `fopen()` pour ouvrir un fichier, la fonction `fgetc()` pour lire un caractère, la fonction `printf()` pour l'afficher sur la sortie standard, et la fonction `fclose()` pour fermer le fichier.

Pour obtenir les paramètres de la ligne de commande, et donc le nom du fichier, on utilise les paramètres `argc` et `argv[]` de la fonction `main()`.

N'oubliez pas de vérifier que le fichier existe et qu'il est lisible. ■

Question 2 *Utilises les appels POSIX* Maintenant, on utilisera les appels système `open()`, `close()`, `read()` et `write()`.

L'appel système `read()` demande l'adresse d'un tampon mémoire et la taille de ce tampon. Nous allons utiliser un tampon de taille 16 octets :

```
#define TAILLE 16
char tampon[TAILLE];
```

La taille peut varier en fonction de nos besoin : plus grand est le tampon, plus d'octets on lit avec un seul appel et donc on nécessitera de moins d'appels.

On discutera de l'importance de la taille du tampon dans le prochain TD. ■

1.2 Sauvegarder des données sur un fichier

Quand on écrit des données sur un fichier, on a plusieurs choix : on peut écrire dans un format textuel ou dans un format binaire. Dans le premier cas, les fichier contient l'encodage des caractères d'un texte binaire. Dans le deuxième cas, le fichier contient la copie des octets qui représentent les donnée en mémoire. Par exemple, supposons de vouloir écrire sur un fichier la valeur de la variable entier suivante :

```
int data = 825391;
```

Sur la plupart des ordinateurs actuels, la variable entier `data` est représentée en mémoire avec quatre octets (32 bits), et sa valeur hexadécimale est `00 0C 98 2F`.

Pour écrire la valeur de la variable `data` sur un fichier, on peut utiliser une représentation binaire : dans ce cas, on utilisera l'appel système `write()` comme dans l'exemple suivant :

```
int data = 825391;

int f = open("myfile", O_WRONLY | O_CREAT, 0x777);
```

```
int r = write(f, &data, sizeof(int));
close(f);
```

Dans ce cas, le fichier `myfile` contera 4 octets, dont la valeur sera `00 0C 98 2F`. Pour vérifier, il est possible de lire le fichier et imprimer son contenu en hexadécimal avec la commande `hexdump`, comme dans l'exemple suivant :

```
\$ hexdump -x  myfile
0000000    982f    000c
0000004
```

Attention : les octect sont à lire dans l'ordre inverse : d'abord les 2 octets de poids faible (`982f`), après les octets de poids fort (`000c`). La première colonne est la position dans le fichier. Le fichier compte 4 octets, comme on peut vérifier avec la commande `stat myfile`.

Maintenant, on veut écrire la même variable sur un fichier en format textuel, par exemple en décimal. Pour faire ça, on traduit sa valeur dans une chaîne de caractères, chaque caractère est une chiffre de sa représentation décimale. On utilise d'abord la fonction `sprintf()` pour faire cette traduction :

```
char tampon[16];
int data = 825391;

int n = snprintf(tampon, 16, "%s", data);
int f = open("textfile", O_WRONLY | O_CREAT, 0x777);
int r = write(f, tampon, n);
close(f);
```

Dans ce cas, le fichier aura taille égale a 6 octets. Vous pouvez visualiser le contenu du fichier avec `cat textfile` et lire la valeur décimale directement. En utilisant `hexdump`, vous pouvez voir les deux représentations :

```
\$ hexdump -C textfile
00000000  38 32 35 33 39 31           |825391|
00000006
```

Dans la deuxième colonne vous voyez les valeurs des octets en hexadécimale. Chaque octet est la valeur ASCII du chiffre correspondant : par exemple, la valeur hexa 38 correspond à la valeur décimale 56 et au caractère '`8`'; la valeur hexa 32 correspond à 50 en décimale et au caractère '`2`', et ainsi de suite. Dans la troisième colonne, vous trouvez le caractères ASCII correspondants au valeurs dans la deuxième colonne.

Pour faire court, il est possible d'utiliser directement la fonction `fprintf()`, sans passer par le tampon.

```
int data = 825391;
FILE * f = fopen("textfile", "w");
fprintf(f, "%d", data);
fclose(f);
```

Si vous utilisez le code `%h` au lieu de `%d`, vous aurez la représentation en hexadécimale dans la chaîne de caractères. Je vous invite à essayer par vous mêmes.

Question 3 *Écrire un tableau d'entiers en binaire* Étant donnée un tableau d'entiers de taille `n`, écrire ce tableau sur un fichier en format binaire.

```
#define TAILLE 256
int tableau[TAILLE];
int n;

// remplissage tableau ...
// n contient le nombre d'entiers présents dans le tableau
// n <= TAILLE

int f = open("tableau", O_WRONLY | O_CREAT, 0x777);
// TODO: sauvegarder le tableau sur le fichier ...
close(f);
```

■

Question 4 *Lire un tableau d'entiers en binaire* Écrire maintenant le code pour lire un fichier qui contient un certain nombre d'entiers `n` dans un tableau d'entiers. C'est l'inverse de l'exercice précédent. ■

Question 5 *Écrire un tableau d'entiers en textuel (décimal)* Étant donnée un tableau d'entiers de taille `n`, écrire ce tableau sur un fichier en format textuel. Chaque entier est suivi par les caractère '\n' (retour à la ligne) qui fonctionne de séparateur. ■

Question 6 *Lire un tableau d'entiers en textuel (décimal)* Écrire maintenant le code pour lire le fichier textuel produit à l'exercice précédent. ■

Question 7 *Écrire une structure de donnés sur un fichier* Considerez la structure des données suivante :

```
struct personne {
  char nom[128];
  char prenom[128];
  int age;
  int taille_cm;
};

struct personne bd[256]; // liste de personnes
int n_personnes;
```

Le tableau `bd` contient un ensemble de données personnels. La variable `n_personnes` contient le nombre de personnes valides dans le tableau (inférieur à 256).

Écrire le contenu du tableau sur un fichier en format binaire. ■

Question 8 *Lire des structure de donnés d'un fichier* Écrire un programme qui fait l'opération inverse de l'exercice précédent, c'est à dire il lit un fichier binaire contenant les données des structures `struct personne`. ■

2 TD 2 : Implantation d'une bibliothèque d'entrées/sorties tamporisées

On propose ici une implantation d'une version très rudimentaire d'une bibliothèque d'entrées/sorties tamporisées. Seuls les équivalents des primitives `fopen()`, `fclose()`, `fgetc()`, et `fputc()` seront implantés. Pour se distinguer des versions standard, on utilisera le préfixe « k ».

La gestion d'un tampon utilisateur nécessite de mémoriser un certain nombre d'informations. On pourra penser notamment à la position courante dans le tampon, l'index du tampon dans le fichier, le nombre de caractères valides dans le tampon, un indicateur signalant que le contenu du tampon a été modifié, et une référence au fichier sous la forme du descripteur associé aux appels système. On définira donc une structure de données, `KFILE`, pour enregistrer toutes les données associées à un fichier ouvert.

Question 1 *Écrire un caractère* Proposez une fonction `int kputc(int ch, KFILE* kfile)` prenant en argument un caractère et un pointeur vers une structure `KFILE` et écrivant ce caractère dans le flot correspondant. Cette fonction retournant le caractère écrit dans le flot si tout s'est bien passé et `EOF` sinon. L'idée essentielle de cette fonction est de ne provoquer un appel système `write` que si le tampon mémoire du `KFILE` est rempli.

Vous définirez en même temps la structure `KFILE` en fonction des besoins pour écrire la fonction `kputc()`. ■

Question 2 *Ouvrir et fermer un flot* Proposez des fonctions :

— `KFILE* kopen(const char *path)` qui ouvre un `KFILE`,

— `int kclose(KFILE* kfile)` qui ferme un `KFILE`.

Vous pourrez profiter de l'écriture de ces fonctions pour définir des fonctions auxiliaires pertinentes. ■

Question 3 *Lire un caractère* Proposez une fonction `int kgetc(KFILE* kfile)` qui lit un caractère dans le flot et le retourne, ou retourne `EOF` si la fin du fichier est atteinte. Vous pourrez étendre la structure `KFILE` et modifier les fonctions précédentes si nécessaire. ■

3 TD 3 : Retrouver une commande

La commande Unix `which` recherche les commandes dans les chemins de recherche de l'utilisateur.

La variable d'environnement `$PATH` contient une liste de répertoires séparés par des « : ». Lors de l'invocation d'une commande depuis le shell, un fichier exécutable du nom de la commande est recherchée dans l'ordre dans ces répertoires. C'est ce fichier qui est invoqué pour l'exécution de la commande. Si aucun fichier ne peut être trouvé, le shell le signale par un message.

La commande `which` affiche le chemin du fichier qui serait trouvé par le shell lors de l'invocation de la commande :

```
$ echo $PATH
/bin:/usr/local/bin:/usr/X11R6/bin:/home/alice/bin
$ which ls
/bin/ls
$ echo $?
0
$ which foo
foo: Command not found.
$ echo $?
1
```

Question 1 *Liste des répertoires de recherche* Dans un premier temps, proposez une fonction `filldirs()` qui remplit un tableau `dirs` contenant la liste des noms des répertoires de `$PATH`. Ce tableau sera terminé par un pointeur `NULL`. ■

Question 2 *Une fonction `which()`* Écrivez maintenant une fonction `which()` qui affiche le chemin absolu correspondant à la commande dont le nom est passé en paramètre ou un message d'erreur si la commande ne peut être trouvée dans les répertoires de recherche de `$PATH`. Cette fonction retourne une valeur booléenne indiquant un succès ou un échec de la recherche.

On pourra utiliser l'appel système

```
#include <unistd.h>
int access(const char *path, int mode);
```

qui vérifie les permissions fournies sous la forme d'un *ou* des valeurs `R_OK` (*read*, lecture), `W_OK` (*write*, écriture), `X_OK` (*execution*, exécution), `F_OK` (existence). ■

Question 3 *La commande `which`* Terminez par écrire votre propre version de la commande `which` qui se termine par un succès si et seulement si toutes les commandes données en paramètre ont été trouvées dans les répertoires de recherche désignés par `$PATH`. ■

4 TD 4 : Clonage de processus

La commande `ps` affiche des informations sur les processus actifs. Sans options, elle affiche les processus du même terminal que le processus `ps`. La sortie contient uniquement le `pid` du processus, le nom du terminal, le temps cumulé d'exécution et le nom de la commande.

Les options les plus courantes sont :

`-a` liste tous les processus sauf ceux non associés à un terminal ;

`-f` donne plus d'information pour chaque processus ;

`-u uidlist` liste les processus des utilisateurs donnés dans `uidlist`.

Les commandes `top` et `htop` affichent les mêmes informations que `ps` en rafraîchissant périodiquement l'affichage.

La commande `kill` permet d'envoyer un signal à un processus. Par exemple `kill -s nomdesignal pid...` envoie le signal dont `nomdesignal` aux processus dont on donne le `pid`. `kill -l` donne la liste des noms des signaux existants. Nous utiliserons ici le signal `KILL` qui permet de tuer un processus.

4.1 Compréhension du clonage simple

Observez le programme suivant.

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

int main() {
    pid_t pid;

    printf("A: ce message s'affiche combien de fois ?\n");
    fflush(stdout);

    pid = fork();
    if (pid == 0) {
        /* qui suis-je, le pere ou le fils ? */
        printf("je suis le ...\n");
    } else {
        /* qui suis-je, le pere ou le fils ? */
        printf("je suis le ...\n");
    }

    printf("B: ce message s'affiche combien de fois ?\n");
    fflush(stdout);

    exit(EXIT_SUCCESS);
}
```

Question 1 *Répondre au questions* Répondez aux questions posées dans le code. ■

Question 2 Pouvez-vous déterminer l'ordre d'apparition des messages ? Si oui, donnez-le, sinon pourquoi ? Pouvez-vous alors déterminer un ordre partiel d'apparition des messages ? ■

Question 3 On ajoute dans le `else`, avant le `printf` l'instruction `wait(NULL)`. Cela change-t-il quelque chose ? ■

Question 4 Pour garantir un fonctionnement plus sûr de ce code, que manque-t-il ? ■

Question 5 À quoi les instructions `fflush(stdout)` servent-elles ? ■

4.2 Généalogie de processus

On exécute le programme suivant.

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

int main() {
    int i;
    pid_t pid;

    for (i = 0; i < 3; i++) {
        pid = fork();
        if (pid == -1) {        /* erreur */
            perror("erreur fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {  /* fils */
            fprintf(stderr, "fils : %d mon pid : %d mon père : %d\n", i, getpid(),
    getppid());
        } else {                /* pere */
            fprintf(stderr, "pere : %d mon pid : %d mon père : %d\n", i, getpid(),
    getppid());
        }
    }
    exit(EXIT_SUCCESS);
}
```

Question 6 Déroulez son exécution et indiquez ce qui est affiché. ■

Question 7 Combien de processus sont lancés en tout (il pourra être utile de représenter sous forme arborescente les processus) ? ■

Question 8 Quels appels systèmes peut-on utiliser pour changer l'affichage et permettre un meilleur suivi des processus (savoir qui est fils de qui) ? ■

4.3 Quatre fils

Question 9 Donnez un programme qui affiche les entiers de 0 à 3 par 4 processus différents. L'exécution de ce programme se termine après l'affichage des 4 entiers. ■

Question 10 Assurez que les processus fils affichent les entiers dans l'ordre croissant. ■

4.4 Multi-fourche

Considerons la fonction suivante :

```
void trif (void(*f1)(void), void(*f2)(void), void(*f3)(void));
```

Le processus exécutant `trif(f1, f2, f3)` engendre des processus exécutant respectivement les fonctions `f1()`, `f2()` et `f3()`, et attend la fin des processus engendrés pour terminer la fonction.

Cette fonction est par exemple utilisée dans le contexte suivant :

```
static void f(int seconds, const char *fname) {
  sleep(seconds);
  fprintf(stderr, "Fonction %s() exécutée par le processus "
  "de pid %d\n", fname, getpid());
}

static void fa() { f(4, "fa"); }
static void fb() { f(2, "fb"); }
static void fc() { f(3, "fc"); }

int main() {
  trif(fa, fb, fc);
```

```
    fprintf(stderr, "terminaison de main()\n");

    exit(EXIT_SUCCESS);
}
```

Question 11 Écrivez le code de la fonction `trif()`. ■

La fonction `multif()` généralise la fonction `trif()` précédente.

```
typedef int (*func_t) (char *);
int multif (func_t f[], char *args[], int n);
```

Le type `func_t` est défini comme « pointeur sur une fonction à un paramètre chaîne de caractères et retournant un entier ».

Les arguments de `multif()` sont un tableau de telles fonctions, un tableau des arguments à passer à ces fonctions, et la taille `n` de ces tableaux. Chacune des fonctions est exécutée par un processus différent. La fonction `f[i]()` sera appelée avec `args[i]` comme argument. Ce processus se termine en retournant comme statut la valeur de la fonction.

La fonction `multif()` se termine elle-même en retournant la conjonction des valeurs retournées par les processus fils : elle ne retourne `EXIT_SUCCESS` que si chacun des processus fils a retourné `EXIT_SUCCESS`.

Pour tester en pratique cette `multif()`, on définira une fonction `testfunc` telle que :

— si son argument est la chaîne `"true"`, elle retourne `EXIT_SUCCESS`,

— si son argument est la chaîne `"false"`, elle retourne `EXIT_FAILURE`.

On définira aussi d'autres comportements sur d'autres valeurs de l'argument (fonction qui prend du temps pour se terminer, par exemple en utilisant `sleep`, pour vérifier que les fonctions ne sont pas exécutées les unes après les autres mais de façon concurrente, etc.).

Enfin la *commande* `multif` prendra en argument les valeurs des arguments pour `testfunc` et se terminera avec la valeur de retour de `multif()` (`./multif true false` devra donc déclencher `multif()` avec `n = 2`, le tableau `f` contiendra 2 fois `testfunc`, le tableau `args` contiendra les chaînes `"true"` et `"false"` et se terminera en retournant `EXIT_FAILURE`).

Question 12 Écrivez le code de la fonction `multif()`. ■

5 TD 5 : les tubes

5.1 Redirection

Il s'agit d'implanter une commande :

```
to file cmd...
```

qui exécute les commandes `cmd...` les unes après les autres en redirigeant leur sortie standard vers le fichier `file` qui est éventuellement tronqué au début de l'exécution de `to`.

Question 1 Implanter la commande `to`. ■

5.2 Calcul parallèle

Pour renvoyer des données vers le processus père, on peut utiliser des tubes anonymes.

Question 2 Écrire un programme multi-processus dans lequel un processus père crée `N` processus fils. L'i-ème processus fils calcule tous les nombres premiers entre iM et $(i+1)M$, (ou M est le nombre d'éléments traités par un processus) et les renvoie sur un tube, puis se termine. Le processus père reçoit toutes les données sur le tube et les affiches à l'écran. ■

Maintenant, on veut implanter une communication bidirectionale. Chaque processus fils calcule les nombre premiers dans un intervalle `start, stop` defini par le père. Chaque fils donc fait la chose suivante :

— Il lit les deux nombres `start` et `stop` sur un tube de communication père-fils ;
— Il calcule tous les premiers entre `start` et `stop` et renvoie les resultats sur un deuxième tube de communication fils-père ;
— Il attends la prochaine couple `start-stop`.

Les fils terminent quand le père ferme le descripteur d'écriture du tube de communication père-fils.

Le père prends en argument le nombre maximale `Max` d'entiers à calculer. Il crée les tubes père-fils et fils-père. Il decoupe l'intervalle $[0, Max-1]$ en morceaux de taille M. Il envoie toutes les intervalles sur le tube père-fils, et il attends les résultats sur le tube fils-père.

Question 3 Implanter le programme. ■

5.3 Connexion de deux processus par un tube anonyme

Il s'agit d'implanter une fonction `compte_octets()` qui :

— crée un tube anonyme,
— crée un processus fils dans lequel on exécute une fonction `void parle(int fd)` en lui donnant le descripteur en écriture du tube ; le processus fils se termine quand cette fonction se termine,
— lit toutes les données dans le tube, les affiche et compte le nombre total d'octets lus,

— se termine quand toutes les données ont été lues et retourne le nombre d'octets lus.

Question 4 Implanter la fonction `compte_octets()`. ■

5.4 Connexion de deux commandes par un tube anonyme

Il s'agit d'implanter une commande

```
pipe command1 to command2
```

qui exécute les commandes `command1` et `command2` en redirigeant la sortie standard de la première commande vers un tube anonyme depuis lequel sera redirigée l'entrée standard de la seconde commande.

Question 5 Implanter la commande `pipe`. ■

5.5 Attendre le lancement d'une commande

On veut écrire un outil `pps <str>` (periodic ps) qui tous les deux seconds cherche à savoir si une commande contenant le mot `str` est en exécution. Pour faire ça, tous les deux seconds :

1. on lance la commande `ps axu` dans un processus fils ;
2. on lit la sortie standard du processus : si on trouve la chaîne contenue dans le premier argument (`argv[1]`), alors on l'affiche sur la sortie standard et on termine ; sinon on affiche *not yet* et on continue.

Pour chercher dans la sortie de `ps` il est possible de lire par lignes avec la fonction de bibliothèque `getline()` :

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

ou `lineptr` est un pointer vers un pointer vers un tampon qui contiendra la ligne lue du fichier ; `n` est un pointer vers un entier qui contiendra le nombre de caractères lus.

Si on donne `*lineptr=NULL` et `*n=0`, la `getline()` alloue dynamiquement (avec `malloc()`) un tampon de la bonne taille pour contenir la ligne et il renvoie le nombre de caractères lus dans `n`.

Si `*lineptr!=NULL`, la `getline()` fait un appel à `realloc()` pour adapter la taille du tampon et affecte la nouvelle valeur de `n`. Quand on aura terminé d'utiliser `getline()`, c'est à l'utilisateur de libérer la mémoire avec `free()`. Attention : `getline()` retourne -1 quand il n'y a plus rien à lire.

En fin, pour chercher l'occurrence de la sous-chaîne `s2` dans une chaîne `s1`, on peut utiliser la fonction de bibliothèque suivante :

```
int strncmp(const char *s1, const char *s2, size_t n);
```

qui renvoie `0` si la chaîne `s2` est égale au premiers `n` caractères de la chaîne `s1`.

Question 6 Implanter la commande `pps`. S'assurer que les fils se terminent quand le père se termine. ■