Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université de Carthage

Ecole Nationale d'Ingénieurs de Carthage

Fascicule de TP

Module : Programmation Système et Réseaux

Niveau : 2ème Année Ingénieur Informatique

Réalisé par

Mme Inès ACHOUR YAZIDI

Module : Programmation système et réseaux
Nature : Travaux pratiques.
Volume horaire : 1 heure 30 minutes par semaine.
Public cible : 2ème année Ingénieur en Informatique.
Semestre : 1ère

Prérequis :

- Module Programmation.
- Module Système d'exploitation.
- Module Atelier Système d'exploitation.

Modalité d'évaluation :

- Mini-projet (50%).
- Examen de TP (50%).

Objectifs

Ces travaux pratiques permettent aux étudiants de mettre en pratique les connaissances acquises dans les séances de cours. A la fin, les étudiants seront en mesure programmer en C sous Linux. Ils leur permettent d'écrire des programmes en C manipulant les fichiers et assurant la communication entre les processus via les tubes anonymes (les pipes) et les tubes nommés (prises) tout en intégrant les threads et les techniques de synchronisation telles que les sémaphores et les mutex.

Références

[1] Joelle DELACROIX, « LINUX : Programmation système et réseaux », édition DUNOD, 2003.

[2] Jean-Paul ARMSPACH, Pierre COLIN, Frédérique OSTRÉ-WAERZEGGERS, « Linux : Initiation et utilisation », DUNOD, 1999.

[3] Bart LAMIROY, Laurent NAJMAN, Huges TALDOT, « Linux : préparation à la certification LPI 101 », Collection Synthex, Pearson Education, 2006.

TP n°1 : Gestion des fichiers

TP n°2 : Gestion des processus

TP n°3 : Les tubes

TP n°4 : Les Threads

TP N°1
Gestion des fichiers

Le langage C ne distingue pas les fichiers à accès séquentiel des fichiers à accès direct, certaines fonctions de la bibliothèque livrée avec le compilateur permettent l'accès direct. Les fonctions standards sont des fonctions d'accès séquentiel.

1 - Déclaration: **FILE *fichier; /* majuscules obligatoires pour FILE */**

On définit un pointeur.

La déclaration des fichiers doit figurer AVANT la déclaration des autres variables.

2 - Ouverture: **FILE *fopen(char *nom, char *mode);**

On passe donc 2 chaînes de caractères : le nom: celui figurant sur le disque, exemple: « a :\toto.dat » - et le **mode** (pour les fichiers TEXTES) :

- « r » lecture seule
- « w » écriture seule (destruction de l'ancienne version si elle existe)
- « w+ » lecture/écriture (destruction ancienne version si elle existe)
- « r+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version.
- « a+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

Mode (pour les fichiers BINAIRES) :

- « rb » lecture seule
- « wb » écriture seule (destruction de l'ancienne version si elle existe)
- « wb+ »lecture/écriture (destruction ancienne version si elle existe)
- « rb+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version.
- « ab+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

A l'ouverture, le pointeur est positionné au début du fichier (sauf « a+ » et « ab+ »)

Exemple : **FILE *fichier ;**

fichier = fopen(« a :\toto.dat », « rb ») ;

3 – Fermeture : **int fclose(FILE *fichier);**

Retourne 0 si la fermeture s'est bien passée, EOF en cas d'erreur.

Il faut toujours fermer un fichier à la fin d'une session :

Exemple : **FILE *fichier ;**

fichier = fopen(« a :\toto.dat », « rb ») ;

/* Ici instructions de traitement */

fclose(fichier) ;

4 - Destruction: **int remove(char *nom);**

Retourne 0 si la destruction s'est bien passée.

Exemple : **remove(« a :\toto.dat ») ;**

5 - Renommer: **int rename(char *oldname, char *newname);**

Retourne 0 si le renommage s'est bien passé.

6 - Positionnement du pointeur au début du fichier: **void rewind(FILE *fichier);**

7- Ecriture dans le fichier:

- int putc(char c, FILE *fichier);

Ecrit la valeur de c à la position courante du pointeur, le pointeur avance d'une case mémoire. Retourne EOF en cas d'erreur.

Exemple : **putc('A', fichier) ;**

- int putw(int n, FILE *fichier);

Idem, n de type int, le pointeur avance du nombre de cases correspondant à la taille d'un entier (4 cases en C standard). Retourne n si l'écriture s'est bien passée.

- int fputs(char *chaîne, FILE *fichier); idem avec une chaîne de caractères, le pointeur avance de la longueur de la chaine ('\0' n'est pas rangé dans le fichier). Retourne EOF en cas d'erreur.

Exemple : **fputs(« BONJOUR ! », fichier) ;**

-int fwrite(void *p,int taille_bloc,int nb_bloc,FILE *fichier); p de type pointeur, écrit à partir de la position courante du pointeur fichier nb_bloc X taille_bloc octets lus à partir de l'adresse p. Le pointeur fichier avance d'autant. Le pointeur p est vu comme une adresse, son type est sans importance. Retourne le nombre de blocs écrits.

Exemple: taille_bloc=4 (taille d'un entier en C), nb_bloc=3, écriture de 3 octets.

int tab[10] ;

fwrite(tab,4,3,fichier) ;

- int fprintf(FILE *fichier, char *format, liste d'expressions); réservée plutôt aux fichiers ASCII. Retourne EOF en cas d'erreur.

Exemples: **fprintf(fichier,"%s","il fait beau");**

fprintf(fichier, "%d",n);

fprintf(fichier,"%s%d","il fait beau",n); // Le pointeur avance d'autant.

8 - Lecture du fichier:

- int getc(FILE *fichier); lit 1 caractère mais retourne un entier n; retourne EOF si erreur ou fin de fichier; le pointeur avance d'une case.

Exemple: **char c ;**

c = (char)getc(fichier) ;

int getw(FILE *fichier); idem avec un entier; le pointeur avance de la taille d'un entier.

Exemple: **int n ;**

n = getw(fichier) ;

- char *fgets(char *chaine,int n,FILE *fichier); lit n-1 caractères à partir de la position du pointeur et les range dans chaine en ajoutant '\0'.

- int fread(void *p,int taille_bloc,int nb_bloc,FILE *fichier); analogue à fwrite en lecture. Retourne le nombre de blocs lus, et 0 à la fin du fichier.

- int fscanf(FILE *fichier, char *format, liste d'adresses); analogue à fprintf en lecture.

9 - Gestion des erreurs:

fopen retourne le pointeur NULL si erreur (Exemple: impossibilité d'ouvrir le fichier).

fgets retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.

La fonction **int feof(FILE *fichier)** retourne 0 tant que la fin du fichier n'est pas atteinte.

La fonction **int ferror(FILE *fichier)** retourne 1 si une erreur est apparue lors d'une manipulation de fichier, 0 dans le cas contraire.

10 - Fonction particulière aux fichiers à acces direct:

int fseek(FILE *fichier,int offset,int direction) déplace le pointeur de offset cases à partir de direction.

Valeurs possibles pour direction :

0 -> à partir du début du fichier.

1 -> à partir de la position courante du pointeur.

2 -> en arrière, à partir de la fin du fichier.

Retourne 0 si le pointeur a pu être déplacé.

EXERCICES :

Exercice 1 : Copier un fichier texte dans un autre : créer un fichier "essai.dat" sous l'éditeur. Tester le programme en vérifiant après exécution la présence du fichier copié dans le directory.

Exercice 2 : Calculer et afficher le nombre de caractères d'un fichier ASCII (Utiliser n'importe quel fichier du répertoire).

Exercice 3 : Créer et relire un fichier binaire de 10 entiers.

TP N°2

Gestion Des Processus

Exercice_1 :
Soit le programme suivant :

```
#include <unistd.h>
#include <stdio.h>
void main( )
{ int p;
int i;
p = fork();
switch (p)
{  case -1:
        printf("Erreur: echec du fork()\n");
        break;
        case 0: /* processus fils */
        printf("Processus fils: \n mon PID est %d, celui de
        mon père est %d \n", getpid(), getppid() );
        break;
        default: /* processus père */

        printf("Ici le père:\n mon pid est %d, mon fils est
   %d\n",getpid(),p);
        printf("Fin du père.\n");
}
}
```

1. Testez l'exécution de ce programme et discutez les différents scénarios de fin.

2. Rajoutez l'appel de la primitive wait pour que le fils se termine avant le père.

3. Peut-on modifier le programme de telle sorte que le père se termine avant le fils ?

Exercice_2:
Reprendre le programme ci-dessus et compléter le, pour qu'il affiche l'**uid** et le **gid** de chaque processus, en plus ce programme doit afficher le contenu d'une variable x initialisée à 2 (avant le fork) et modifié selon x+3 par le fils et selon x*5 par le père.

Exercice_3:
Ecrire un programme qui ouvre un fichier nommé « toto », en lecture et écriture, dont le contenu est la suite 123456789. Le programme fork ensuite; le fils écrit ab dans le fichier ensuite il s'endort et il lit 2 caractères; le père s'endort, lit 2 caractères et écrit AB.

Exercice_4:
Reprendre l'exercice 1, avec l'exécution de la commande ls –l par le fils.

TP N°3
Les tubes

Exercice 1:

```
#include <stdio.h>
# include <unistd.h>
int main()
{
int tube[2];
char buf[20];
pipe(tube);
if (fork())
{ /* père */
close (tube[0]);
write (tube[1], "bonjour", 8);
}
else
{ /* fils */
close (tube[1]);
read (tube[0], buf, 8);
printf ("%s bien recu \n", buf);
}
return 0 ;
}
```

Tester l'exécution de ce programme et expliquer son fonctionnement.

Exercice 2:

Écrire un programme C qui :

- Crée un fils. Le père et le fils communiquent via un tube anonyme (pipe).
- Le père attend des caractères depuis le clavier et les écrits dans le tube.
- Le fils (le consommateur) se contente d'afficher ce qu'il a lu dans le tube.

Exercice 3:

Écrire un programme C qui fait communiquer le père et le fils selon le schéma suivant :

- le processus père envoie deux entiers à son fils ;
- le fils lit les deux entier, fait l'addition et renvoie le résultat ;
- le père lit cette réponse et l'affiche.

Pour que la communication fonctionne dans les deux sens, on utilisera deux tubes : un pour le sens Père ➔Fils et un pour le sens Fils➔Père.

Exercice 4:

Ecrire un programme C contenant deux processus et qui correspond à l'exécution de la commande ls –l | wc –l . Nous choisissons de faire exécuter la commande ls –l par le processus fils et la commande wc –l par le processus père.

Exercice 5:

Écrire un programme C formé de trois processus et opérant comme suit :

- le processus père récupère un message à partir du clavier et l'envoie via un premier pipe à son premier processus fils ;
- ce fils récupère le message le crypte et ce en remplaçant chaque 'a' par un '*' et envoie ensuite le message crypté via un deuxième pipe à l'autre processus fils ;
- le deuxième fils récupère le message et le décrypte et le renvoie à son père via un troisième pipe ;
- le père récupère le message et l'affiche.

Exercice 6:

Ecrire un programme C composé de deux processus qui communiquent via un tube nommé (prise).

Le premier processus écrit la chaîne « 0123456789 » dans le tube et le deuxième processus la lit et l'affiche.

Remarque : vérifier l'existence du tube après l'exécution.

Exercice 7:

Résoudre l'exercice 4 par un tube nommé (prise).

TP N°4

Les Threads

1. Les Threads :

Un même processus peut se décomposer en plusieurs parties, qui vont s'exécuter simultanément en partageant les mêmes données en mémoire. Ces parties se nomment threads.
Du point de vue de l'utilisateur, les threads semblent se dérouler en parallèle. Lorsqu'une fonction bloque par exemple un programme (comme la fonction *recv*), si celui-ci dispose d'une interface graphique, il sera inactif tant que la fonction le bloquera. Les threads nous permettront de régler ce problème. Pour pouvoir travailler avec des threads, il faut choisir la bibliothèque Pthread.
Le terme Pthread est une abréviation de "POSIX Threads".
POSIX est lui un acronyme de "Portable Operating System Interface for UniX".
Déclarer un thread :
pthread_t th1;

Une fois que notre thread est déclaré, il va falloir le lier à une fonction de notre choix, la fonction désignée se déroulera ensuite en parallèle avec le reste de l'application. Pour réaliser cela nous allons utiliser la fonction *pthread_create* dont le prototype est donné ci-dessous.

int pthread_create(pthread_t th1,pthread_attr_t* attr, void*(start_routine)(void *), void* arg);*

- En cas de succès la fonction renvoie 0. En cas d'erreur, la fonction renvoie un code d'erreur non nul.
- L'argument **th1** correspond au thread qui va exécuter la fonction.
- L'argument **attr** indique les attributs du thread (l'adresse de départ, la taille de la pile, la politique d'ordonnancement qui lui est associé, la priorité qui lui est associé, etc.). Ce paramètre ne nous intéresse pas, nous mettrons donc celui-ci à NULL pour que les attributs par défaut soient utilisés.
- L'argument **start_routine** correspond à la fonction à exécuter.
- L'argument **arg** est un pointeur sur void qui sera passé à la fonction à exécuter. Si vous n'avez aucun paramètre à passer, mettez ce paramètre à NULL.

Une fois que notre thread est exécuté, il se peut que nous ayons besoin de savoir quand il se termine. La fonction *pthread_join* va permettre d'attendre la fin du thread c'est à dire la fin de l'exécution de la fonction exécutée par celui-ci. Voici le prototype:

*int pthread_join(pthread_t th, void** thread_return);*

- En cas de succès, la fonction renvoie 0. En cas d'erreur, la fonction renvoie un code d'erreur non nul.
- L'argument **thread** correspond au thread à attendre.
- L'argument **thread_return** est un pointeur sur la valeur de retour du thread.

int pthread_exist(void ret);*

- Met fin au thread qui l'exécute et retourne le paramètre **ret** qui peut être récupéré par un autre thread effectuant pour sa part un appel à la primitive *phtread_join*.

Exercice 1 :

Soit le programme suivant :

```
#include <stdio.h>
#include <pthread.h>
void *task1 (void *arg )
{ printf (" Thread 1 \n" );
pthread_exit (0);
return NULL;
}
void *task2 (void *arg )
{ printf (" Thread 2 \n");
pthread_exit (0);
return NULL;
}
int main ( )
{
pthread_t t1, t2;
printf ("main Init\n");
pthread_create (&t1, NULL, task1, NULL);
pthread_create (&t2, NULL, task2, NULL);
printf ("main Fin\n");
return 0;
}
```

1. Tester ce programme, l'ordre d'affichage est il déterministe ?
2. Modifier ce programme pour avoir toujours l'affichage suivant :

main Init
thread1
thread2
main Fin

3. Modifier le dernier programme et utiliser une seule fonction task au lieu de task1 et task2. L'output doit être le même que celui de la question 2.

Exercice 2 :

Soit le programme suivant :

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
/*1*/
int data=0;
void *fonctionplus (void *arg)
{ int i ;
for (i=0;i<4;++i)
{
/*2*/
data++;
```

```
printf("Thread plus %d\n", data);
/*3*/
}
pthread_exit(0);
return NULL;
}
void *fonctionmoins (void *arg)
{ int i;
for(i=0;i<4;++i)
{
/*4*/
data--;
printf("Thread moins %d\n", data);
/*5*/
}
pthread_exit(0);
return NULL;
}
int main (void)
{ pthread_t tid1, tid2 ;
void *status ;
int i ;
printf("main Init\n");
/*6*/
pthread_create(&tid1,NULL,fonctionplus,NULL) ;
pthread_create(&tid2,NULL,fonctionmoins,NULL);
for (i=0;i<4;++i)
{
/*7*/
printf ("\n Variable partage = %d\n",data) ;
/*8*/
}
printf ("main Fin \n") ;
pthread_join(tid1,&status) ;
pthread_join(tid2,&status);
/*9*/
return 0 ;
}
```

1. tester ce programme
2. Terminer les passages manquants marqués par les numéros en utilisant les mutex.
3. Balancer l'exécution entre le thread principal, le thread d'incrémentation et le thread de décrémentation en utilisant sleep.
4. Reprendre le code précédent en utilisant les sémaphores.