TD 1

Primitives de gestion de fichiers
Programmation Sys. en C

Objectifs du TD :

L'objectif de ce TD est, d'une part, de se familiariser avec la programmation en C et acquérir des connaissances en prog. C nécessaires pour réaliser facilement les TDs et les TPs à venir. D'autre part, ce TD permettra aux étudiants d'apprendre les primitives ou les fonctions de gestion de fichiers via l'écriture d'un programme en C.

Travail à effectuer :

Ce TD consiste à :

1) Apprendre avec l'aide du chargé de TD les primitives pour ouvrir/fermer un fichier (***open/close***), se positionner dans un fichier (***lseek***) et lire/écrire dans un fichier (***read/write***).

2) Ecrire un programme C nommé « ***init.c*** » qui initialise la valeur **100** dans un fichier nommé **NOMBRE**

3) Ecrire un autre programme C nommé « ***ajout.c*** » qui :
   - i) **Ouvre** le fichier **NOMBRE** ;
   - ii) effectue une **boucle de 10 tours** permettant de :
     - a) **lire** dans le **fichier NOMBRE** et sauvegarder la valeur lue dans un **compteur**,
     - b) augmenter de 1 à chaque tour la valeur du **compteur** et
     - c) écrire la valeur du **compteur** dans le fichier **NOMBRE**.
   - iii) Affiche la dernière valeur sauvegardée dans le fichier.

1) Les **primitives de gestion d'un fichier** sont décrites en détail dans ce qui suit :

a. Ouvrir un fichier :
   i. **Syntaxe** :
   #include <sys/types.h>
   #include <sys/stat.h>
   #include <fcntl.h>
   int **open**(const char *pathname, int flags)
   int **open**(const char *pathname, int flags, mode_t mode) ;
   flags : O_RDONLY, O_WRONLY et O_RDWR
   ii. **Exemple** : pour ouvrir le fichier NOMBRE en écriture seulement et en mode d'accès 0664, nous devons taper :
   int df = open ("NOMBRE",O_WRONLY|O_CREAT,0664) ;
   O_CREAT : cette option permet de créer le fichier si ce dernier n'existait pas au moment de l'appel de la fonction ***open***.
   iii. **Valeur de retour** : le descripteur du fichier (une valeur entière) si succès et -1 si erreur

b. Se positionner dans un fichier :
   i. Syntaxe :
   #include <sys/types.h>
   #include <unistd.h>
   off_t **lseek**(int fd, off_t offset, int whence) ➔ à *whence*, il faut ajouter l'offset, qui peut être positif ou négatif, pour obtenir la nouvelle position. Le premier paramètre ***fd*** est le descripteur du fichier
   ii. Exemples :
      1. Pour se positionner au début du fichier : lseek(df_n,(off_t)0,SEEK_SET)
      2. Pour se positionner à la fin du fichier : lseek(df_n,(off_t)0,SEEK_END)
      3. Pour se positionner à la position courante : lseek(df_n,(off_t)0,SEEK_CUR)
   iii. **Valeur de retour** : la position courante (nombre d'octets à partir du début du fichier) si succès et -1 si erreur

c. Lire dans un fichier :
   i. Syntaxe :
   #include <unistd.h>
   ssize_t **read**(int fd, void * buf, size_t count)
   ii. Exemple : nbo_lus=read(df_n,&nombre,sizeof(int))
   iii. Valeur de retour : le nombre d'octets lus dans le fichier si succès et -1 si erreur

d. Ecrire dans un fichier :
   i. Syntaxe :
   #include <unistd.h>
   ssize_t **write**(int fd, void * buf, size_t count)

ii. **Exemple :** nbo_ecrits=write(df_n,&nombre,sizeof(int))
iii. **Valeur de retour** : le nombre d'octets écrits dans le fichier si succès et -1 si erreur

e. Fermer un fichier :
   i. **Syntaxe :**
      #include <unistd.h>
      int close(int fd)
   ii. **Exemple : close**(df_n);
   iii. **Valeur de retour** : 0 si succès et -1 si erreur

2)

Ayant vu ensemble la syntaxe et des exemples d'utilisation des primitives de gestion de fichiers, nous allons maintenant répondre à la deuxième question : nous allons écrire le programme *init.c* qui permet d'initialiser une valeur entière (par exemple 100) dans un fichier nommé NOMBRE. Si le fichier NOMBRE n'existe pas, dans le programme nous allons forcer sa création (flag : O_CREAT).

```
/* Programme d'initialisation init.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <unistd.h>
#include <errno.h>
```

A compléter …

3) L'étudiant a une demie-heure à disposition pour répondre à cette question en tenant compte de tous les points spécifiés ci-dessus dans la question 3) et en écrivant un programme C, nommé *ajout.c*. Ensuite, la solution est présentée et expliquée par le chargé du TD.

```
/*Programme d'ajout ajout.c*/

#include <sys/types.h>
#include <errno.h>
#include <sys/file.h>
#include <unistd.h>

#define N_Boucles     10
```

A compléter …

A venir
Primitives de gestion des répertoires (opendir, readdir, mkdir, ...)

pos = lseek (df, offset, org)

s = stat (nom, &buf)

s = fstat (df, &buf) parcil

s = flock (df, options) : vérouillage

s = fcntl (df, cmd, ...) : vérouillage et autre opération

TD 2

Primitives de gestion de fichiers – partie 2
Programmation Sys. en C

Objectifs du TD :

L'objectif de ce TD est de permettre aux étudiants d'apprendre les ***primitives*** *ou les* ***fonctions*** de ***gestion*** *des* ***répertoires*** via l'écriture et l'exécution des programmes en C.

Travail à effectuer :

Ce TD consiste à :

1) Apprendre avec l'aide du chargé de TD les primitives pour créer/supprimer un répertoire (***mkdir/rmdir***), ouvrir/fermer un répertoire (***opendir/closedir***) et parcourir ou lire un répertoire (***readdir***).
2) Il s'agit maintenant d'écrire un programme en C permettant d'ouvrir, de parcourir un répertoire, et d'obtenir de l'information sur les fichiers dans ce répertoire à la manière d'un ***ls*** (affiche le contenu d'un répertoire). Pour cela, nous allons utiliser les primitives de la bibliothèque standard (fichier d'en-tête ***dirent.h***), présentées dans (1).

1) Les **primitives de manipulation d'un répertoire** sont décrites en détail dans ce qui suit :

   a. Créer un nouveau répertoire :
      i. **Syntaxe :**

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

int **mkdir**(const char *dirname, mode_t mode) ; dans *mode* l'utilisateur peut spécifier les permissions d'accès. L'argument *mode* peut prendre une des valeurs suivantes :

S_ISUID	04000	bit set-UID
S_ISGID	02000	bit set-GID
S_ISVTX	01000	bit « sticky »
S_IRWXU	00700	lecture/écriture/exécution du propriétaire
S_IRUSR	00400	le propriétaire a le droit de lecture
S_IWUSR	00200	le propriétaire a le droit d'écriture
S_IXUSR	00100	le propriétaire a le droit d'exécution
S_IRWXG	00070	lecture/écriture/exécution du groupe
S_IRGRP	00040	le groupe a le droit de lecture
S_IWGRP	00020	le groupe a le droit d'écriture
S_IXGRP	00010	le groupe a le droit d'exécution
S_IRWXO	00007	lecture/écriture/exécution des autres
S_IROTH	00004	les autres ont le droit de lecture
S_IWOTH	00002	les autres ont le droit d'écriture
S_IXOTH	00001	les autres ont le droit d'exécution

ii. **Exemples** :
mkdir("rep1", 0664)
iii. **Valeur de retour** : 0 si succès et -1 si erreur

b. Supprimer un répertoire
i. **Syntaxe** :
#include <sys/stat.h>
int **rmdir**(const char *dirname); pour que l'utilisateur puisse supprimer le répertoire, ce dernier doit être vide !!
ii. **Valeur de retour** : 0 si succès et -1 si erreur

c. Ouvrir un répertoire :
i. **Syntaxe** :
#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char *dirname) ➔ ouvrir un *directory stream*.
ii. **Exemple** :
- DIR * dir1 ;
- dir1 = opendir("rep1") ;

iii. **Valeur de retour** : un pointeur vers un *directory stream* si succès et -1 si erreur

d. Lire dans un répertoire :

i. **Syntaxe :**

#include <sys/types.h>
#include <dirent.h>
struct dirent ***readdir**(DIR *dir) ; *readdir* lit dans le répertoire l'élément à la position courante et se positionne sur le prochain élément pour la prochaine lecture.

ii. **Exemple** :

```
DIR *dir1;
struct dirent *entry;
dir1 = opendir("rep1");
entry = readdir(dir1) ;
```

iii. **Valeur de retour** : un pointeur sur une structure de type ***dirent***, contenant des données, si succès et -1 si erreur. La structure ***dirent*** contient les données ci-dessous :

```
struct dirent {
    ino_t d_ino;                    /* inode number */
    off_t d_off;                    /* offset to the next dirent */
    unsigned short int d_reclen;    /* length of this record */
    unsigned char d_type;           /* type of file */
    char d_name[256];               /* We must not include limits.h! */
};
```

taille cellule
taille contenu

Le type du fichier peut être : inconnu, Fifo, dispositif à caractères (ex. /dev/sda), répertoire, dispositif à blocks, fichier normal, lien symbolique, socket, ...

DT_UNKNOWN 0
DT_FIFO 1
DT_CHR 2
DT_DIR 4
DT_BLK 6
DT_REG 8
DT_LNK 10
DT_SOCK 12
DT_WHT 14

e. Fermer un répertoire :

i. **Syntaxe :**

#include <sys/types.h>
#include <dirent.h>
int **closedir**(DIR * dir)

ii. **Exemple :** int d = closedir(dir1);

iii. **Valeur de retour** : 0 si succès et -1 si erreur

Pour obtenir plus de détails sur les primitives présentées ci-dessus utilisez ***man nom_primitive*** → man opendir, man closedir, man readdir …

2) Il s'agit maintenant d'écrire un programme en C permettant d'ouvrir, de parcourir un répertoire, et d'obtenir de l'information sur les fichiers dans ce répertoire à la manière d'un ls (ls : affiche le contenu d'un répertoire). Pour cela, nous allons utiliser les primitives de la bibliothèque standard (fichier d'en-tête dirent.h), présentées dans 1).

a. Ecrire une fonction qui liste les noms des fichiers et des sous-répertoires dans un répertoire donné. Pour rappel, les fonctions *stat* et *fstat* vues dans la partie 1 du cours fournissent le statut du fichier (ses caractéristiques). Vous trouvez la syntaxe de ces 2 fonctions et la structure ***stat*** dans le support du cours (pages 20-23).

b. Améliorez la fonction précédente pour pouvoir indiquer aussi le type du fichier. Nous ne testerons que les fichiers ordinaires, les répertoires et les liens symboliques.

c. Modifiez le programme précédent pour afficher en plus la taille de chaque entrée du répertoire étudié.

d. Affichez aussi les droits associés à chaque fichier.

chdir (chemin) : modif rep courant
link (fichier, lien) : crée un lien vers un fichier
unlink (fichier) : supprimer lien

TD ISE

```
int main(){
/**
 * @file init.c
 * @author : Arsène
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(){
    int nb=100;
    int df=open("NOMBRE", O_WRONLY|O_CREAT,0666);
    lseek(df,(off_t)0,SEEK_SET);
    write(df,&nb,sizeof(int));
    close(df);
    return 0;
}
```

3)

```
/**
 * @file: ajout.c
 * @author: Arsène
 */
#include <sys/types.h>
#include <errno.h>
#include <sys/file.h>
#include <unistd.h>
int main(){
    int df=open("NOMBRE",O_RDWR);
    for(int i=0;i<100;i++){
    lseek(df,(off_t)0,SEEK_SET);
```

```
int compteur = 0;
int *pt_compteur = &compt;
for (int i = 0; i < 10; ++i) {
    lseek(fd, 0, SEEK_SET);
    read(fd, &compteur, sizeof(int));
    compteur++; lseek(fd, (off_t)0, SEEK_SET);
    write(fd, &compteur, sizeof(int));
}
close(fd);
printf("%d", cpt);
return 0;
}
```

Sem 3 PSE TD 1

1)/

2) a)

```
void afficherDossier (char *nomDossier){
    DIR *fichier = opendir(nomFichier);
    struct dirent *entry;

    while(entry=readdir(fichier) != null){
        printf("%s \n",
            entry->d_name);
    }
    closedir(fichier);
}
```

b)

```
void afficherDossier(char *nomFichier){
    DIR*fichier = opendir(nomFichier);
    struct dirent *entry;

    while(entry=readdir(fichier
        != null){
        printf("%s\t%c,
        entry->d_name,
        entry->d_type)

    closedir(fichier);
}
```

c) d)

```
void afficherDossier(char *nomDossier)
    DIR *dossier = opendir(nomDossier);
    struct dirent* entry;
    struct stat* tampon;
    while(entry = readdir(dossier)){
        stat(entry->d_name, tampon
        printf("%s\t%c\t%o\n",
            entry->d_name,
            entry->d_type,
            tampon->st_size);
        printf("\t%o", getMode(tampon
        ->st_mode);
    }
    closedir(dossier);
}
```

TD 3

Gestion des Processus

Objectifs du TD :

L'objectif principal de ce TD est de comprendre et maitriser l'utilisation des fonctions ***fork()***, ***wait()***, ***waitpid()***, ***exit()*** ...
Pour rappel :
fork() crée un nouveau processus, fils du processus appelant cette fonction.
wait() et ***waitpid()***, avec ***exit()*** permettent de synchroniser les processus père et fils. Ces fonctions permettent au programmeur de mettre en attente un processus père jusqu'à la terminaison d'un processus fils.

On considère le programme C suivant :

```
1.  int main ( ) {
2.     int a, b=-2, statut=-2, idpro=-1;
3.     for (a = 1; a <=2; a++) {
4.          idpro = fork ( );
5.          if (idpro == 0) {
6.               b=6 ;
7.               if (a < 2) {
8.                    idpro = fork ( );
9.                    if (idpro == 0) {
10.                        exit (b - a);
11.                   } /* end if */
12.                   wait (&statut);
13.              } /* end if */
14.              exit (b - a);
15.         } /* end if */
16.         if (a < 2) wait (&statut);
17.    } /* end for */
18.    idpro = wait (&statut);
19. } /*end main */
```

pid_t fork() : création, renvoie 0 au fils et le pid du fils au père

pid_t getpid() et pid_t getppid()
pid appelant pid père

wait : bloque appelant jusqu'à ce qu'un fils se termine
pid_t wait(int* ptr_status)
retourne le pid du fils qui se termine
ptr status : paramètre de exit() du fils dans les bits significatifs
pid_t waitpid(pid_t pid, int* status, int options)
option : blocage ou non
pid : pid du fils à terminer

EXEC

int execl(char *path, char *arg1, ...)
- crée pas de nv processus
- remplace pile, données et code du proc courant
- complémentaire à fork (pour garder le proc actuel on exec).

VEROUILLAGE FICHIER

int flock(int fd, int operation)
operation : LOCK_SH (partagé)
LOCK_EX (exclusif)
LOCK_UN (unlock)
LOCK_NB : ne pas bloquer la fonction lors de la demande de verrouillage

Pour les processus père et fils dans le code ci-dessus, complétez les 4 tableaux en bas.
Le but est d'indiquer les valeurs des variables aux instants spécifiés dans la première colonne du tableau. On suppose que :
(1) la fonction ***fork*** s'exécute avec succès,
(2) l'identificateur du processus père est **100**, et
(3) le système d'exploitation affecte aux processus fils des identificateurs consécutifs (101, 102, ...) selon leur ordre de <u>création</u>.
La phrase "1°/2° exec. inst. X" signifie la première/deuxième exécution de l'instruction numéro X.

<u>Attention</u>, il faut indiquer :
- **NE**, quand la variable n'existe pas (dans le cas où le processus lui-même n'existe pas). C'est-à-dire le processus n'est pas encore créé ou le processus s'est terminé.
- **U**, quand on ne peut pas dire avec certitude si la variable existe ou quelle est sa valeur.

Processus **100**	Variable a	Variable b	Variable idpro	Variable statut
Avant la 2° exec. inst. 5	2	-2	103	1280
Avant la 2° exec. inst. 10	2	-2	103	1280
Après l'instruction 18	3	-2	103	4x256=1024

↳ renvoie b - a = 6 - 1 = 5 au proc père

Processus **101**	Variable a	Variable b	Variable idpro	Variable statut
Avant la 1° exec. inst. 5	1	-2	0	-2
Avant la 1° exec. inst. 10	1	6	102	-2
Après l'instruction 18	NE	NE	NE	NE

↳ renvoie b - a = 6 - 1 = 5 au proc père

Processus **102**	Variable a	Variable b	Variable idpro	Variable statut
Avant la 1° exec. inst. 5	NE	NE	NE	NE
Avant la 1° exec. inst. 10	1	6	0	-2
Après l'instruction 18	NE	NE	NE	NE

↳ renvoie b - a = 6 - 2 = 4 au proc père

Processus **103**	Variable a	Variable b	Variable idpro	Variable statut
Avant la 2° exec. inst. 5	2	-2	0	1280
Avant la 2° exec. inst. 10	2	6	0	1280
Après l'instruction 18	NE	NE	NE	NE

int lockf(int fd, int cmd, off_t len)
→ partie d'un fichier

COM INTER PROCESSUS

-> Fichiers
-> Tubes
-> files de msg
-> mémoire partagée (semaphores)
-> signaux

Tubes

-> fichier virtuel ouvert en lecture & ecriture
int pipe(int fd[2]
fd[0] ouvert en lecture et fd[1] en ecriture
-> le père et le fils partagent les df associés aux tubes
-> quand l'un écrit à une extremité l'autre peut lire

Semaphore :

-> struct avec :
- un entier
- une file
- 2 méthodes : P -> decrementation de la valeur et blocage éventuel de l'appelant
V -> incrémentation et reveil d'un proc (action atomique

- Constructeur

P : cpt--
si cpt < 0
bloquer appelant
insérer dans file
le desc du processus
fsi

V : cpt ++
si cpt > 0
extraire dans la file
éveiller le proc qui
corresp
fsi

TD 4

Communications entre Processus

Tubes classiques

Objectifs du TD :

L'objectif principal de ce TD est de permettre à deux ou plusieurs processus (*de la même famille*) de communiquer entre eux, avec l'utilisation des **tubes classiques**. Pour réaliser cet objectif, et donc pour mettre en œuvre la communication interprocessus, dans ce TD, nous allons apprendre et maitriser l'utilisation des fonctions ***pipe(), fork()***, ***wait()***, ***exit()***, etc.
Dans ce TD, nous nous focalisons sur les tubes classiques et nous supposons que les processus font partie de la même famille (un processus et ses propres fils).

Pour rappel :

- ***pipe()*** crée un tube de communication classique pour le processus en exécution.
- ***fork()*** crée un nouveau processus, fils du processus appelant la fonction *fork*.
- ***wait()*** avec ***exit()*** permettent de synchroniser les processus père et fils. Ces fonctions permettent, par exemple, au programmeur de mettre en attente un processus père jusqu'à la terminaison d'un processus fils.

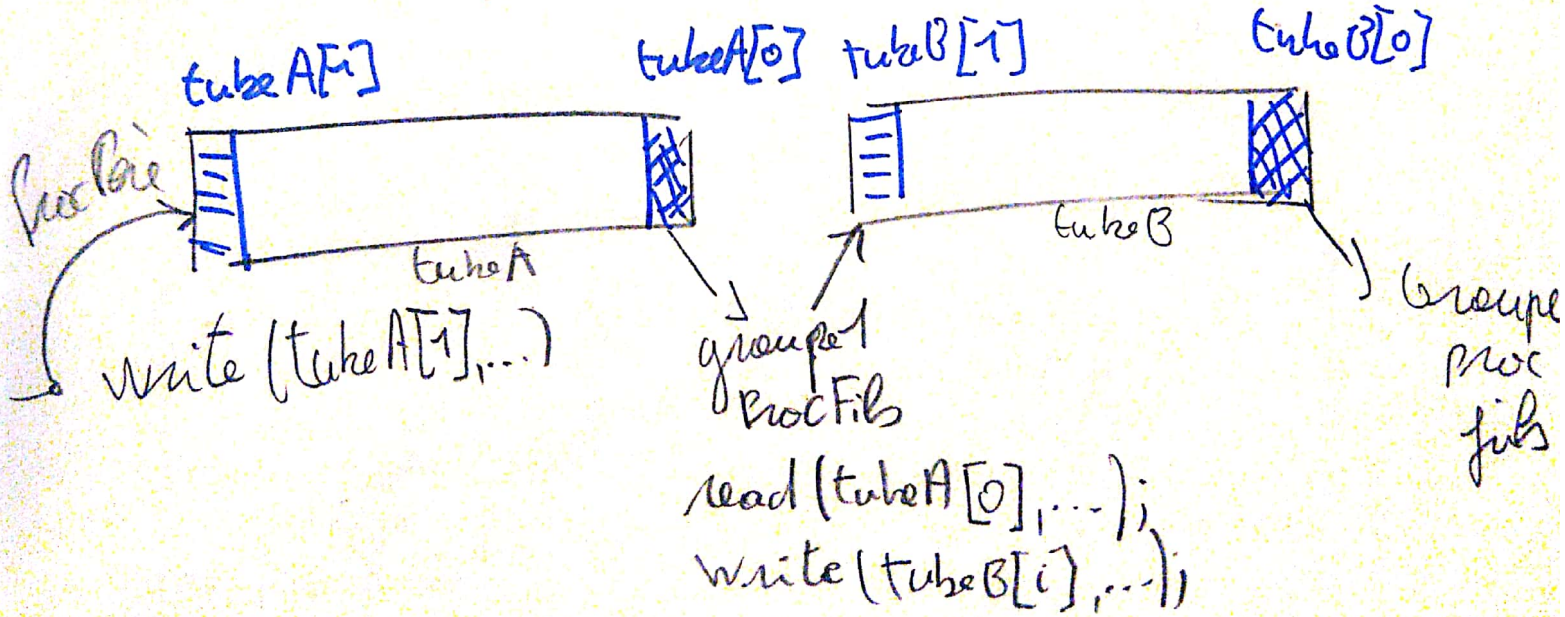
Dans l'Annexe ci-joint (pages 3-4), on considère un extrait d'un programme C. Ce programme est composé des étapes suivantes :

a) Création de tubes
b) Création de fils
c) Lecture d'une phrase (ou une chaine de caractères) sur le clavier
d) Transmission de cette phrase à un premier groupe de processus (***Groupe1***), en utilisant un premier tube
e) Transmission de cette phrase des processus du groupe *Groupe1* vers d'autres processus du second groupe « ***Groupe2*** », en utilisant un deuxième tube
f) Assemblage, par les processus du groupe Groupe2, des caractères et affichage sur l'écran.

Répondez aux questions suivantes :

1) Combien de processus fils y a-t-il dans le programme ?
2) Quel est la tâche (ou le rôle) du processus père ?
3) Quelle est la tâche de chaque processus fils ?
4) Est-ce que **tous** les processus fils sont identiques ? Justifiez votre réponse.
5) Quelle valeur prend etat[ordre], pour ordre =0, 1, 2, 3 ?
6) Compléter le tableau suivant tout en tenant compte de toutes les étapes (**a-f**) mentionnées ci-dessus.
7) Comment peut-on modifier le programme dans l'**annexe** pour qu'on puisse réaliser les tâches des fils en exécutant (chargeant à partir du disque) d'autres programmes ?

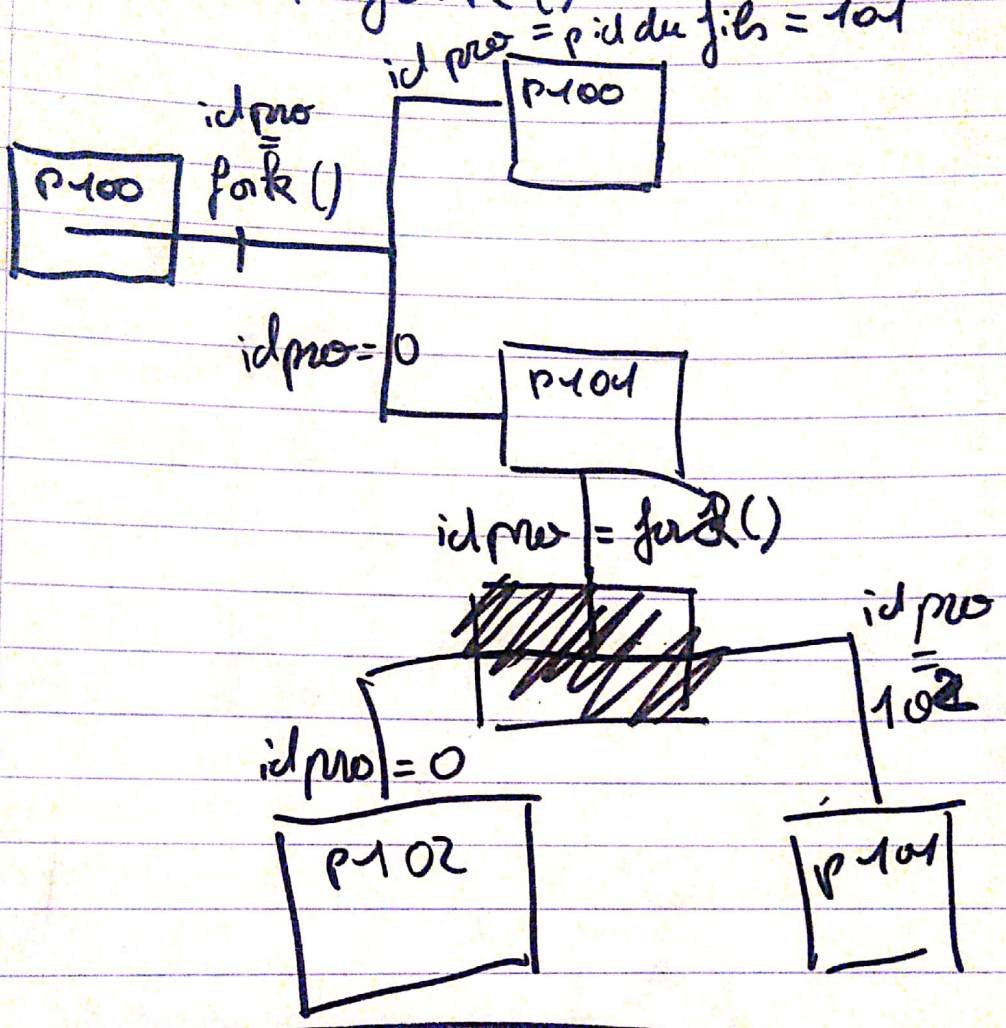
1	tubeA	**7**	read(tubeA[0])	**13**	tubeB[0]
2	tubeB	**8**	close(tubeA[0])	**14**	tubeB[0]
3	close(tubeB[0])	**9**	close(tubeB[1])	**15**	tubeB[0]
4	close(tubeA[1])	**10**	tubeA[0]	**16**	tubeA[0]
5	read(tubeA[0])	**11**	tubeA[1]	**17**	tubeB[0]
6	write(tubeB[1])	**12**	tubeB[1]	**18**	tubeB[1]



Annexe A

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


#define NbOfChild 2
#define       MOD    5


int main (int nbarg, char *tbarg[]) {
        int idpro, tubeA[2],tubeB[2], nbcarlus, attente, etat[2*NbOfChild], ordre, cptcar=0;
        char phrase[256],carlu;
        ssize_t taille;
        size_t tmax=255;

        srand(getpid());

        pipe(-1-); tubeA
        pipe(-2-); tubeB

        for (ordre=1; ordre<=NbOfChild; ordre++) {
                if ((idpro=fork ()) < 0) {
                        perror("creation de processus");
                        exit(errno);
                }

                if (idpro == 0) {//Groupe1
                        cptcar=0;
                        close(-3-); tubeB[0]
                        close(-4-); tubeA[1]
                        nbcarlus= read(-5-,&carlu,1); tubeA[0]
                        while (nbcarlus > 0) {
                                int nbecr;
                                cptcar ++;
                                attente = rand()%MOD;
                                sleep(attente);
                                nbecr=write(-6-,&carlu, 1); tubeB[1]
                                nbcarlus= read(-7-,&carlu,1); tubeA[0]
                        }
                        close(-8-); tubeA[0]
                        close(-9-); tubeB[1]
                        printf ("\t Je suis le fils %d, nbcarlus : %d et je me termine
\n", ordre, cptcar);
                        exit(0);
                }
        }
```

Fonction fork() :

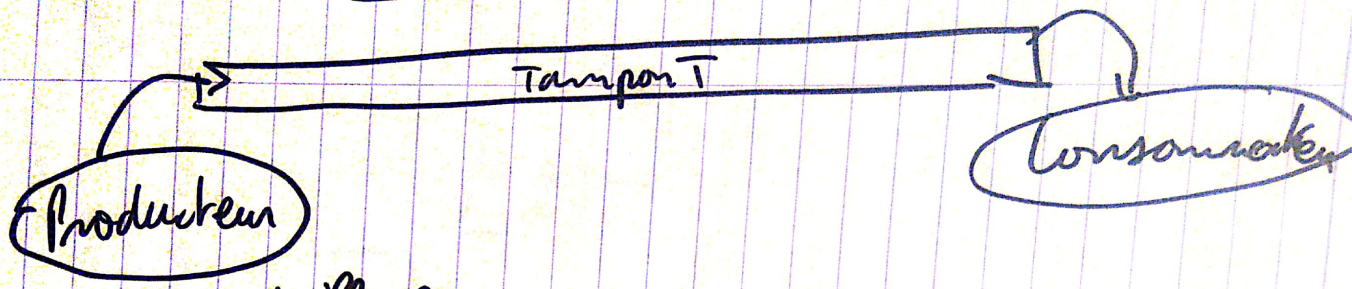


Semaphors (suite) :

Sémaphore binaire : (mutex).

Contrôle l'accès à la section critique
Sa valeur peut être que 1 ou 0.
Initialisé à 1

Producteur - Conso :



→ T borné de taille n
→ T rempli par P vidé par C
→ P peut produire quand C consomme
→ P et C synchro : C consomme pas si y'a rien
P insère pas si c'est plein

→ Compteur : nb d'éléments dans C
→ Quand P voit que T est vide en arrivant, il réveille C
→ Quand C voit que T est plein en repartant il ~~endort~~ réveille P

P : while en_vie
Produire(R)
insérer(R, tampon)
fwhile

C : while en_vie
R = extraire(tampon)
consommer R
fwhile

PB : conflit sur compteur : C teste compteur et trouve 0, on bascule sur P avant que C ne sleep, P inc cpt de 0 à 1 (c'était vide donc il réveil C mais il pas endormi..

→ Sémaphore ! mais cpt critique → sémaphore bin

Producteur - Consommateur : Solution :

```
sema mutex, full, empty
sema_init(&mutex, 1); sema_init(&full, 0);
sema_init(&empty, BUFFER_SIZE);
void producteur() {
    int item;
    while(1) {
        item = produire();
        down(&empty);
        down(&mutex);
        deposer(int, buffer);
        up(&mutex);
        up(&full);
    }
}

void consommateur() {
    int item;
    while(1) {
        down(&full);
        down(&mutex);
        item = retirer();
        up(&mutex)
        up(&empty)
        consommer(item);
    }
}
```

TD 5

Gestion de la mémoire

Description du problème

Par quelle magie, l'identificateur que j'ai écrit dans mon programme se transforme-t-il en une zone mémoire ? Plus important encore : qui fait quoi ?
Les acteurs de ce processus sont : le compilateur, l'éditeur de liens et le système d'exploitation. Chacun faisant sa part de travail en fonction des informations dont il dispose :

- le compilateur ne connaît que le fichier qu'on lui donne à compiler,
- l'éditeur de liens connaît l'ensemble du projet (et donc, en particulier, l'ensemble des besoins en mémoire),
- le système d'exploitation contrôle l'attribution des ressources au programme pendant tout le temps de son exécution en s'aidant des informations fournies par le compilateur et l'éditeur de liens.

Les contraintes suivantes doivent être respectées :

- Un processus ne doit pas pouvoir accéder n'importe où en mémoire et doit rester confiné dans les zones qui lui sont attribuées (contrainte de *sécurité*).
- N'importe quelle zone de la mémoire doit pouvoir être attribuée à un processus (contrainte de *libre position*).

La dernière contrainte signifie que d'une exécution à l'autre, la zone mémoire associée à l'identificateur, doit pouvoir changer.

1) Version minimale :

Dans cette version, nous faisons l'hypothèse qu'une grande zone contigüe de mémoire (un *segment*) est allouée à chaque processus. En particulier, notre identificateur fait référence à une partie de cette zone.

Exercice 1 - Mécanisme d'adressage et contrôle d'accès

1a) Comment le système va-t-il stocker les informations relatives à chaque segment (pour pouvoir les gérer) ?
1b) Comment l'éditeur de liens décrit-il l'identificateur dans l'exécutable de façon à respecter la contrainte de libre position (appelons ***d*** la valeur qu'il utilise) ?
1c) Comment respecter la contrainte de sécurité ?
1d) Comment faire pour que tout cela soit le plus efficace possible ?
1e) Cette solution n'est pas complètement satisfaisante. Pourquoi ?

2) Version paginée :

Nous gardons de la version précédente les mécanismes d'adressage et de contrôle. Pour pallier à la difficulté rencontrée dans la section précédente (segments de tailles différentes) nous divisons la mémoire en blocs de tailles égales ou ***pages*** (une page fait typiquement 512 ou 1024 octets). Chaque page est identifiée par un numéro (0 pour celle en début de mémoire, puis 1 ...).
Le segment est lui aussi divisé en pages. Un segment est donc maintenant un ensemble de pages pouvant être n'importe où en mémoire. Une ***table des pages*** est associée à chaque processus pour pouvoir situer chacune des pages : elle fait correspondre à chaque numéro de page du segment un numéro de page en mémoire.

Exercice 2 – Pagination

2a) Que devient la valeur ***d*** décrivant notre identificateur dans l'exécutable ?
2b) Décrivez l'algorithme permettant de déterminer l'adresse mémoire correspondant à ***d***.
2c) Que devient le contrôle d'accès dans ce contexte ?
2d) Supposons une taille de page de 1024 octets, une valeur de ***d*** de 5 734. Calculez l'adresse mémoire en utilisant la table des pages suivante :

N° Page Segment	0	1	2	3	4	5	6
N° Page Mémoire	34	2	67	41	114	50	890

2e) L'hypothèse du segment unique associé à chaque processus, n'est pas très réaliste. Expliquez pourquoi.

3) Segmentation et pagination :

Un processus possède donc un certain nombre de segments, identifié par un numéro 0, 1,... Il faut donc ajouter à la solution précédente une ***table des segments*** : chaque entrée de cette table contient une table des pages décrivant comment le segment est stocké en mémoire.

Exercice 3 - Segmentation

3a) Comment l'identificateur de mon code est-il désigné dans l'exécutable à présent ?
3b) Imaginons que nous disposions de 64 bits pour désigner mon identificateur. On appelle ***v*** la valeur en question. Comment mettre en œuvre la réponse à la question précédente (on suppose un maximum de 6 segments) ?
3c) Décrivez l'algorithme permettant de déterminer l'adresse mémoire correspondant à ***v***.

4) Mémoire virtuelle :

Le système de pagination permet de retirer de la mémoire les pages des processus qui ne sont pas utilisées pour les stocker, temporairement, sur le disque dur. Elles sont rapatriées dès que le processus en a besoin. Ce mécanisme, qui permet d'étendre la mémoire en utilisant le disque dur, est appelé ***mémoire virtuelle***.

Exercice 4 - Mémoire virtuelle

Comment modifier la solution précédente pour permettre l'utilisation de la mémoire virtuelle ?

Exercice 1

1) a) Il va stocker l'adresse de départ et la taille de chaque segment, respectivement la base et la limite

b) Etat donné un segment particulier, d est l'adresse relative par rapport au début

adresse = début + d

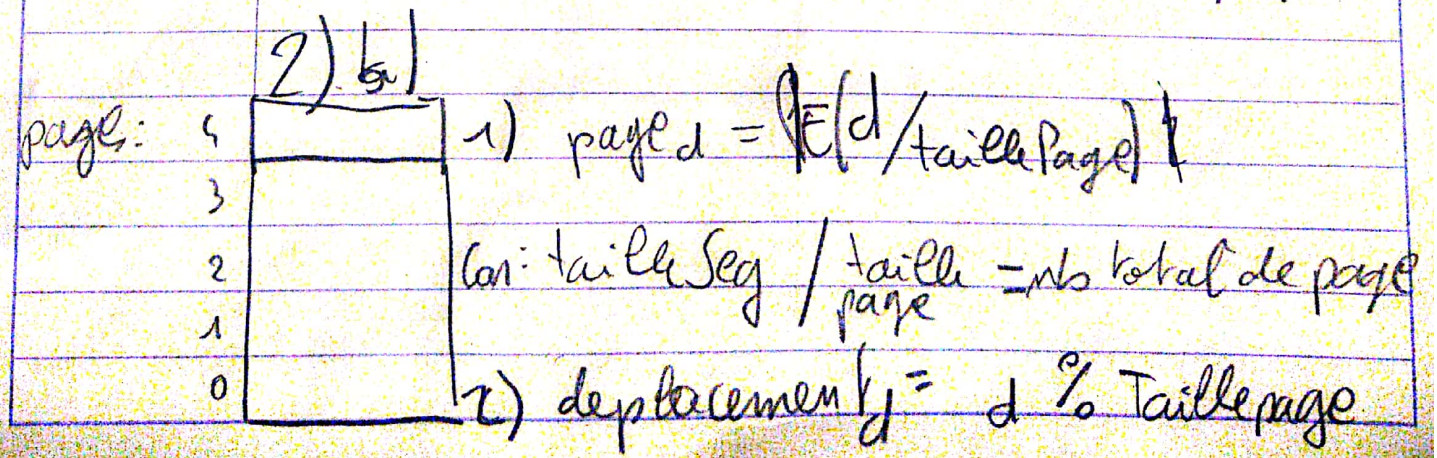
c) d doit être inférieure à la limite du segment.

d) Avoir un registre de base et de limite dans l'UC pour stocker les adresses. Le calcul des adresses est microcodé dans les construction du CPU

e) À cause de la fragmentation externe -> On arrive par à charger un programme même si l'espace total dispo le permet.

Exercice 2 a) Une adresse dans une page

2) b)



@ = NPageMem x TaillePage + Déplacement d

c) d < étendu

d) $page_d = \left| \frac{5734}{1024} \right|$ déplacement d = 5734 % 1024
= 5 = 614

@ = 5 x 1024 + 614
= 5814 → adresse physique

e) Chaque processus a plusieurs segments de taille différente car plein de données différente

Sem 7 PSE

Exercice 3

a) Il faut ajouter le numéro du segment s au déplacement d

b)

64 pour l'identificateur V
6 segment au plus => 3 bits

000
001
010
100
101
110
111

Les 3 premiers bits sont utilisés pour le numéro du segment. (poids fort) -> s

Il reste 61 bits de poids faible pour d.

$s = PE(v/2^{61})$

$d = v \% [2^{61}]$

c)

1. Avec $s = PE(v/2^{61})$ On connait l'adresse de la table de page pour le segment s.

$d = v \% [2^{61}] \Rightarrow$ page $d = PE(d / \text{taillepage})$

deplacement $d = d \% \text{taille page}$

On regarde dans la table de page dans quelle page mémoire est mis page d.

Enfin :

@ = N°PageMem x taille Page + Deplacem

Exercice 4 :

Mémoire Virtuelle → Mémoire physique

↳ On ajoute un drapeau dans la table de page pour savoir si elle est en mémoire ou pas.