

Cours: Programmation Système

Filière: SMI Semestre S6

El Mostafa DAOUDI
daoudie@yahoo.com

*Département d'Informatique,
Faculté des Sciences
Université Mohammed Premier
Oujda*

Février 2020

El Mostafa DAOUDI- p.1

Plan du cours

Ch.I. Introduction aux processus

Ch.II. Threads Posix.

Ch.III. Gestion des signaux sous Unix

Ch.IV. Communication classique entre processus

- Synchronisation père /fils
- Tubes anonymes
- Tubes nommés

Ch.V. Compilation et Exécution d'un programme

El Mostafa DAOUDI- p.2

Références:

- Ressources Internet
- Livres: En particulier la référence suivante:
 - Titre: Programmation système en C sous Linux
Signaux, processus, threads, IPC et sockets
2e édition
 - Auteur: Christophe Blaess
 - Edition: Eyrolles

Remarques:

- Le chapitre II (les threads)
Le paragraphe 6. (les conditions) n'est pas inclus dans les examens.
- Le chapitre III (les signaux)
Le sous paragraphe 9.4. (La primitive « sigaction() »)
n'est pas inclus dans les examens.
- Le chapitre V n'est pas inclus dans les examens.

Chapitre I. Introduction aux processus

1. Introduction

- Le système Unix/Linux est multi-tâche: Plusieurs programmes peuvent s'exécuter simultanément sur le même processeur.
- Puisque le processeur ne peut exécuter qu'une seule instruction à la fois, le noyau va donc découper le temps en tranches de quelques centaines de millisecondes (quantum de temps) et attribuer chaque quantum à un programme
⇒ le processeur bascule entre les programmes.
- L'utilisateur voit ses programmes s'exécuter en même temps.
⇒ Pour l'utilisateur, tout se passe comme si on a une exécution réellement en parallèle.

2. Définitions

- Programme: c'est un fichier exécutable stocké sur une mémoire de masse. Pour son exécution, il est chargé en mémoire centrale.
- Processus (process en anglais), est un concept central dans tous systèmes d'exploitation:
 - C'est un programme en cours d'exécution; c'est-à-dire, un programme à l'état actif.
 - C'est l'image de l'état du processeur et de la mémoire pendant l'exécution du programme. C'est donc l'état de la machine à un instant donné.

3. Contexte d'un processus

- Le contexte d'un processus (Bloc de Contrôle de Processus : BCP) contient toutes les ressources nécessaires pour l'exécution d'un processus.
⇒ Il regroupe le code exécutable, sa zone de données, sa pile d'exécution, son compteur ordinal ainsi que toutes les informations nécessaires à l'exécution du processus.
- L'opération qui consiste à sauvegarder le contexte d'un processus et à copier le contexte d'un autre processus dans le processeur s'appelle changement (ou commutation) de contexte.
- La durée d'une commutation de contexte varie d'un constructeur à un autre, elle reste toutefois très faible.

Remarque:

- A chaque instant, le processeur ne traite qu'un seul processus.
⇒ Un processeur peut être partagé par plusieurs processus. L'ordonnanceur (un algorithme d'ordonnancement) permet de déterminer à quel moment arrêter de travailler sur un processus pour passer à un autre.
- Les processus permettent d'effectuer plusieurs activités en "même temps".

Exemple :

Compiler et en même temps imprimer un fichier.

4. Relations entre processus

□ Compétition

Situation dans laquelle plusieurs processus doivent utiliser simultanément une ressource à accès exclusif (ressource ne pouvant être utilisée que par un seul processus à la fois)

Exemples

- imprimante

□ Coopération

Situation dans laquelle plusieurs processus collaborent à une tâche commune et doivent se synchroniser pour réaliser cette tâche.

Exemples: soient p1 et p2 deux processus

- p1 produit un fichier, p2 imprime le fichier
- p1 met à jour un fichier, p2 consulte le fichier

- Synchronisation: La synchronisation se ramène au cas suivant : un processus doit attendre la terminaison d'un autre processus.

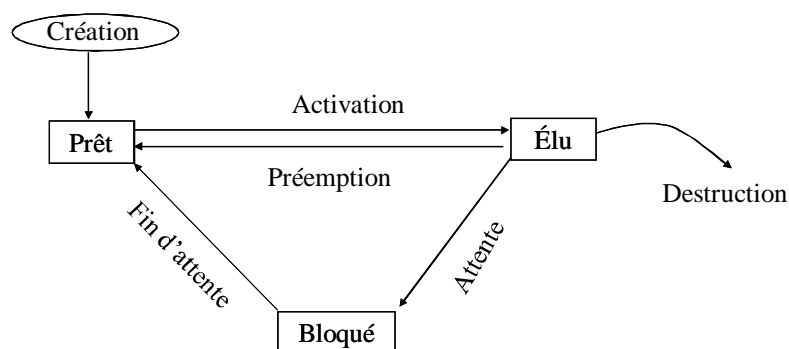
5. États d'un processus

A un instant donné, un processus peut être dans l'un des états suivants :

- Actif (Elu, en Exécution): Le processus en cours d'exécution sur un processeur (il n'y a donc qu'un seul processus actif en même temps sur une machine mono-processeur). On peut voir le processus en cours d'exécution en tapant la commande « *ps* » sur une machine Unix.
 - Un processus élu peut être arrêté, même s'il peut poursuivre son exécution.
 - Le passage de l'état actif à l'état prêt est déclenché par le noyau lorsque la tranche de temps attribué au processus est épuisée.

- **prêt** : Le processus est suspendu provisoirement pour permettre l'exécution d'un autre processus. Le processus peut devenir actif dès que le processeur lui sera attribué par le système (il ne lui manque que la ressource processeur pour s'exécuter).
- **bloqué (en attente)**: Le processus attend un événement extérieur (une ressource) pour pouvoir continuer, par exemple lire une donnée au clavier. Lorsque la ressource est disponible, il passe à l'état "prêt".
Un processus bloqué ne consomme pas de temps processeur; il peut y en avoir beaucoup sans pénaliser les performances du système.
- **Terminé** : Le processus a terminé son exécution.

La transition entre ces trois états est matérialisée par le schéma suivant:



6. Les identifiants d'un processus

- Chaque processus est identifié par un numéro unique, le PID (Processus IDentification) et il appartient à un propriétaire identifié par UID (User ID) et à un groupe identifié par GID (Group ID).
- Le PPID (Parent PID) d'un processus correspond au PID du processus qui l'a créé.

Les primitifs

- La fonction « int getpid() » renvoie le numéro (PID) du processus en cours.
- La fonction « int getppid() » renvoie le numéro du processus père (PPID). Chaque processus a un père, celui qui l'a créé.
- La fonction « int getuid() » permet d'obtenir le numéro d'utilisateur du processus en cours (UID) .
- La fonction « int getgid() » renvoie le numéro du groupe du processus en cours (GID).

7. Création d'un processus

- Un processus est créé au moyen d'un appel système (« fork() » sous UNIX/Linux). Cette création est réalisée par un autre processus (processus père).
- Au démarrage du système, le processus « init » est lancé et les autres processus descendent directement ou indirectement de lui .
⇒ notion d'arborescence de descendance des processus (père-fils). Cette arborescence admet un ancêtre unique et commun à tous: le processus « **init** ».
- Pour avoir des informations sur les processus on utilise les commandes shells: « **ps** » ou « **pstree** »

L'appel système de « fork() »

L'appel système « fork() » dans un processus permet de créer un nouveau processus. Elle est déclarée dans `<unistd.h>`.

⇒ Il faut inclure le fichier d'en-tête `<unistd.h>` (`# include <unistd.h>`) dans le programme qui appelle « fork() »

Syntaxe

```
pid_t fork(void);
```

- La fonction « fork() » renvoie un entier.
- « pid_t » est un nouveau type qui est identique à un entier. Il est déclaré dans « `/sys/types.h` ». Il est défini pour l'identificateur du processus. A la place de « pid_t » on peut utiliser « int ».

- Le processus qui a invoqué la fonction « fork() » est le processus père tandis que le processus créé est le processus fils.
- Les processus père et fils ne se distinguent que par la valeur de retour de «fork()».
 - Dans le processus père: la fonction « fork() » renvoie le numéro du processus fils (processus nouvellement créé).
 - Dans le processus fils: la fonction « fork() » renvoie 0.
 - En cas d'échec, le processus fils n'est pas créé et la fonction renvoie « -1 ».

Utilisation classique sans gestion des erreurs:

```
# include <unistd.h>
# include <stdio.h>
...
if (fork() != 0) {
    /*Exécution du code correspondant au processus père */
}
else { /* if (fork() == 0) */
    /*Exécution du code correspondant au processus fils */
}
```

Exemple 1: Le processus père crée un fils, ensuite chaque processus affiche son identifiant.

```
#include <stdio.h>
#include <unistd.h>
int main() {
    if(fork() !=0) {
        printf("Je suis le pere: ");
        printf(" Mon PID est %d\n",getpid());
    } else {
        printf("Je suis le fils:");
        printf(" Mon PID est %d\n",getpid());
    }
}
```

Exécution:

```
% ./testfork
Je suis le pere: Mon PID est 1320
Je suis le fils: Mon PID est 1321
%
```

Attention: l’affichage peut apparaître dans l’ordre inverse.

El Mostafa DAOUDI- p.19

Exemple 2: Le processus père crée un fils ensuite affiche son identifiant ainsi que celui de son fils. Le fils affiche son identifiant ainsi que celui de son père.

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid=fork(); // appel de la fonction fork()
    if (pid!=0) {
        printf("Je suis le pere:");
        printf(" mon PID est %d et le PID de mon fils est %d\n", getpid(), pid);
    } else {
        printf("Je suis le fils.");
        printf(" Mon pid est:%d et le PID de mon pere est %d\n ",getpid(), getppid());
    }
}
```

Exemple d’exécution:

```
% ./testfork
Je suis le pere: Mon PID est 1320 et le PID de mon fils est 1321
Je suis le fils: Mon PID est 1321 et le PID de mon pere est 1320
%.
```

El Mostafa DAOUDI- p.20

- Après l'appel de la fonction « fork() », le processus père continue l'exécution du même programme.
- Le nouveau processus (le processus fils) exécute le même code que le processus parent et démarre à partir de l'endroit où « fork() » a été appelé.

Exemple 3:

```
int i=8 ;
printf("i= %d \n", i); /* exécuté uniquement par le processus père */

int pid=fork(); /* A partir de ce point, le père et le fils exécutent le
                même programme*/
printf(" Au revoir \n" );
```

Exemple d'exécution

```
i=8
Au revoir
Au revoir
```

Remarque:

- i=8 est affiché par le processus père
- Au revoir est affiché par les processus père et fils.

Exemple 4:

```
int i=8 ;
printf("i= %d \n", i); /* exécuté uniquement par le processus père */
if (fork() != 0) { /*à partir de ce point, le fils commence son exécution*/
    printf("Je suis le père, mon PID est %d\n", getpid());
} else {
    printf("Je suis le fils, mon PID est %d\n", getpid());
}
printf(" Au revoir \n" ); /* exécuté par les deux processus père et fils*/
}
```

Exemple d'exécution

i=8

Je suis le fils, mon PID est 1271

Au revoir

Je suis le père, mon PID est 1270

Au revoir

El Mostafa DAOUDI- p.23

- Le processus créé (le processus fils) est un clone (copie conforme) du processus créateur (père). Il hérite de son père le code, les données la pile et tous les fichiers ouverts par le père. Mais attention: les variables ne sont pas partagées.
⇒ Un seul programme, deux processus, deux mémoires virtuelles, deux jeux de données

El Mostafa DAOUDI- p.24

Exemple 5:

```
int i=8 ;
if (fork() != 0) {
    printf("Je suis le père, mon PID est %d\n", getpid());
    i+= 2;
} else {
    printf("Je suis le fils, mon PID est %d\n", getpid());
    i+=5;
}
printf("Pour  PID = %d, i = %d\n", getpid(), i);  /* exécuté par
                                                  les deux processus */
}
```

Exemple d'exécution

Je suis le fils, mon PID est 1271
Je suis le père, mon PID est 1270
Pour PID=1270, i = 10
Pour PID=1271, i = 13

8. Appel système de « fork() » avec gestion des erreurs

8.1. Rappel sur la variable « errno »

Dans le cas où une fonction renvoie « -1 » c'est qu'il y a eu une erreur, le code de cette erreur est placé dans la variable globale « errno », déclarée dans le fichier « errno.h ». Pour utiliser cette variable, il faut inclure le fichier d'en-tête « errno.h » (#include <errno.h>).

8.2. Rappel sur la fonction « perror() »

La fonction « perror() » renvoie une chaîne de caractères décrivant l'erreur dont le code est stocké dans la variable « errno ».

Syntaxe de la fonction « perror() »

```
void perror(const char *s);
```

La chaîne « s » sera placée avant la description de l'erreur. La fonction affiche la chaîne « s » suivie de deux points, un espace (un blanc) et en fin la description de l'erreur et un retour à la ligne.

Attention: Il faut utiliser « perror() » directement après l'erreur, en effet si une autre fonction est exécutée mais ne réussit pas, alors c'est le code de la dernière erreur qui sera placée dans « errno » et le code d'erreur de la première fonction sera écrasé.

Utilisation classique de « fork() » avec gestion des erreurs:

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

main () {
    pid_t pid;
    if ((pid=fork()) == - 1) {
        /* code si échec \n */
        printf("Code de l'erreur pour la création: %d \n", errno);
        perror("Erreur de création");
    }
    else {
        if (pid != 0) {
            /* Exécution du code correspondant au processus père */
        }
        else { /* if (fork() == 0) */
            /* Exécution du code correspondant au processus fils */
        }
    }
}
```

9. Primitives de recouvrement: les primitives exec()

La famille de primitives « `exec()` » permettent le lancement de l'exécution d'un programme externe provenant d'un fichier binaire. Il n'y a pas création d'un nouveau processus, mais simplement changement de programme. Diverses variantes de la primitive « `exec()` » existent et diffèrent selon le type et le nombre de paramètres passés.

Cas où les arguments sont passés sous forme de liste

- La primitive «`execl()`»
`int execl(char *path, char *arg0, char *arg1, ..., char *argn, NULL)`
 - « `path` » indique le nom et le chemin absolu du programme à exécuter (chemin/programme).
Exemple: « `path` » pour exécuter la commande « `ls` » est: `"/bin/ls"`.
 - « `arg0` »: désigne le nom du programme à exécuter. Par exemple si le premier paramètre est `"/bin/ls"` alors « `arg0` » vaut : `"ls"`.
 - « `argi` » : désigne le i-ième argument à passer au programme à exécuter.

Exemple: lancer la commande « ls -l /tmp » à partir d'un programme.

```
#include <stdio.h>
#include <unistd.h>

int main(void){
    execl("/bin/ls","ls","-l","/tmp", NULL) ;
    perror("echec de execl \n");
}
```

- **La primitive «execlp() »**

```
int execlp(char *fiche, char *arg0,char *arg1,...,char *argn, NULL)
```

- « fiche » indique le nom du programme à exécuter. Si le programme à exécuter est situé dans un chemine de recherche de l'environnement alors il n'est pas nécessaire d'indiquer le chemin complet du programme

Exemple: pour exécuter la commande « ls », « fiche » vaut "ls".

- « arg0 »: est le nom du programme à exécuter. En général identique au premier si aucun chemin explicite n'a été donné. Par exemple si le premier paramètre est "ls", « arg0 » vaut aussi: "ls".

Exemple: lancer la commande « ls -l /tmp » à partir d'un programme.

```
#include <stdio.h>
#include <unistd.h>
int main(void){
    execlp("ls", "ls", "-l", "/tmp", NULL) ;
    perror("echec de execlp \n");
}
```

Cas où les arguments sont passés sous forme de tableau

- Primitive « execv() »
int execv(char *path, char *argv[])
- Primitive « execvp() »
int execvp(char *fiche, char *argv[])
 - « argv » : pointeur vers un tableau qui contient le nom et les arguments du programme à exécuter. La dernière case du tableau vaut « NULL ».
 - « path » et « fiche » ont la même signification que dans les primitives précédentes.

Remarque:

Il est important de noter que les caractères que le shell expande (comme ~) ne sont pas interprétés ici.

10. La primitive « sleep() »

Syntaxe

int *sleep(int s)*

Le processus qui appelle la fonction « sleep() », est bloqué pendant « s » secondes.

Exemple: Le programme suivant nommé «testfork.c » bloque le processus père de 10 secondes (appel de « sleep(10) ») et le processus fils de 5 secondes (appel de « sleep(5) ») .

```
int main() {
    if (fork() != 0) {
        printf(" je suis le père, mon PID est %d\n", getpid());
        sleep(10) /* le processus père est bloqué pendant 10
                  secondes */
    } else {
        printf(" je suis le fils, mon PID est %d\n", getpid());
        sleep(5) /* le processus fils est bloqué pendant 5
                 secondes */
    }
    printf(" PID %d : Terminé \n", getpid())
}
```

Compilation et exécution

```
% gcc -o testfork testfork.c
```

On lance « testfork » en background, ensuite on lance la commande « ps -f »

```
% ./testfork & ps -f
```

```
je suis le fils, mon PID est 2436
```

```
je suis le père, mon PID est 2434
```

```
[2] 2434
```

UID	PID	PPID	TTY	CMD
etudiant	1816	1814	pts/0	bash
etudiant	2434	1816	pts/0/testfork
etudiant	2435	1816	pts/0	ps -f
etudiant	2436	2434	pts/0/testfork

```
2436 : Terminé
```

```
2434 : Terminé
```

```
%
```

El Mostafa DAOUDI- p.37

11. La primitive «exit() »

La primitive «exit() » permet de terminer le processus qui l'appelle. Elle est déclarée dans le fichier « stdlib.h ».

Pour utiliser cette primitive, il faut inclure le fichier d'en-tête « stdlib.h » (#include <stdlib.h>) .

Syntaxe:

```
void exit (int status)
```

- « status » est un code de fin qui permet d'indiquer au processus père comment le processus fils a terminé (par convention: status=0 indique une terminaison normale sinon indique une erreur).

El Mostafa DAOUDI- p.38

Exemple:

```
#include <stdio.h>
# include <stdlib.h>
#include <unistd.h>
main() {
    if (fork() == 0) { // fils
        printf("je suis le fils, mon PID est %d\n", getpid());
        exit(0);
        /* la partie qui suit est non exécutable par le fils */
    } else {
        printf("je suis le pere, mon PID est %d\n", getpid());
    }
}
```

Ch.II. Les threads Posix

I. Présentation générale

1. Introduction

- L'exécution d'un processus s'effectue dans son contexte (Bloc de Contrôle de Processus : BCP) qui contient toutes les ressources nécessaires pour l'exécution du processus. Il regroupe le code exécutable, sa zone de données, sa pile d'exécution, son compteur ordinal ainsi que toutes les informations nécessaires à l'exécution du processus.
- Quand un processus crée un processus fils ce dernier est un clone (copie conforme) du processus créateur (père). Il hérite de son père le code, les données la pile et tous les fichiers ouverts par le père. Il exécute le programme de son père mais dans son propre contexte.

- Quand il y a changement de processus courant (le processus est suspendu et le processeur est libéré pour exécuter un autre processus), il y a une commutation ou changement de contexte (on recharge toutes les informations nécessaires à l'exécution du nouveau processus), le noyau s'exécute alors dans le nouveau contexte.
- En raison de ce contexte, on parle de processus lourd (ou processus), en opposition aux processus légers (les threads), car ceux ci partagent une grande partie du contexte avec leurs pères).

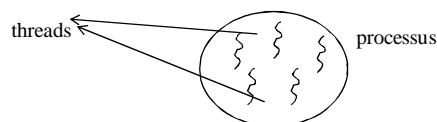
2. Définitions des threads

- Conceptuellement, lorsqu'on lance l'exécution d'un programme, un nouveau processus est créé. Dans ce processus, un thread est créé pour exécuter la fonction principale « main() ». Ce thread est appelé le thread principal (ou thread original). Ce thread peut créer d'autres threads au sein du même processus; tous ces threads exécutent alors le même programme, mais chaque thread est chargé d'exécuter une partie différente (une fonction) du programme à un instant donné.
⇒ Un (une) thread (fil d'exécution) est une partie du code d'un programme (une fonction), qui se déroule parallèlement à d'autres parties du programme.

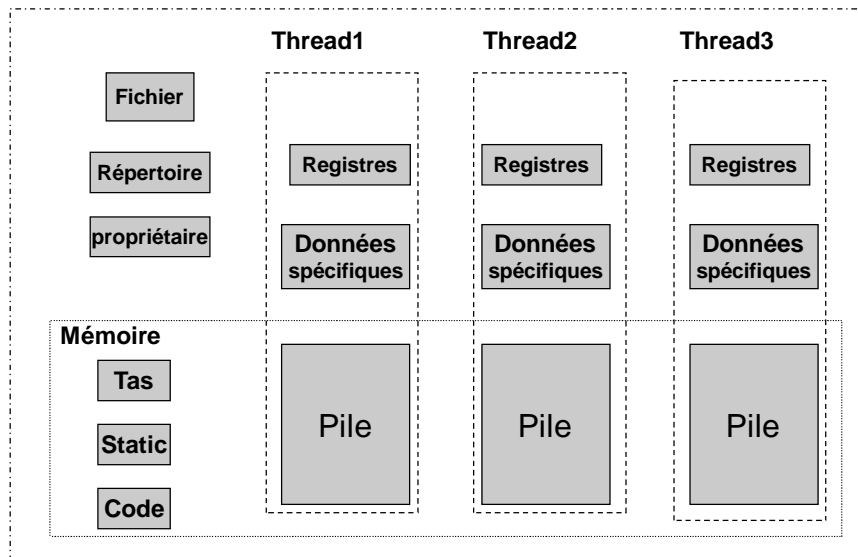
- Le concept multithread (concept multitâches) est un mécanisme (autre que le concept processus) qui permet à un programme d'effectuer plusieurs tâches en même temps.
- Les threads s'exécutent de manière alternative (en cas de mono processeur) ou en parallèle (en cas de multiprocesseur). Un thread, qui termine son quantum de temps, sera interrompu pour permettre l'exécution d'un autre thread et attend son tour pour être ré-exécuté. C'est le système qui s'occupe de l'ordonnancement des threads.

3. Caractéristiques

- Un thread ne peut exister qu'au sein d'un processus (processus lourd).
- Le thread créateur et le thread créé partagent tous les deux le même espace d'adressage (code et variables partagées), les mêmes descripteurs de fichiers et autres ressources.
 ⇒ Les threads demandent moins de ressources que les processus lourds (les processus créés avec l'appel « fork() »). Raison pour laquelle on les appelle aussi processus légers.
- Chaque thread possède sa propre pile d'exécution.
- Ils disposent d'un mécanisme de synchronisation



Thread et processus



El Mostafa DAOUDI- p.45

Remarques:

- Si un thread appelle une des fonctions « exec() », tous les autres threads se terminent.
- Si un thread modifie la valeur d'une variable partagée les autres threads verront la valeur modifiée.
- Si un thread ferme un descripteur de fichier, les autres threads ne peuvent plus lire ou écrire dans ce fichier.

El Mostafa DAOUDI- p.46

4. Avantages et inconvénients des threads

Avantages

- Utilisation d'un système multi-processeurs.
- Utilisation de la concurrence naturelle d'un programme. Par exemple, une activité peut faire des calculs alors qu'une autre attend le résultat d'une E/S
- Bonne structuration du programme, en explicitant la synchronisation nécessaire.
- Moins coûteuses en terme temps et espace mémoire.

Inconvénients

- Surcoût de développement: nécessité d'expliciter la synchronisation, vérifier la réentrance des bibliothèques, danger des variables partagées.
- Surcoût de mise-au-point: débogage souvent délicat (pas de flot séquentiel à suivre).

II. Les Threads de POSIX (Portable Operating System Interface): Interface « pthread »

1. Introduction

- GNU/Linux implémente l'API de threading standard POSIX (appelée aussi « pthread »). C'est un standard de librairie supporté par de nombreuses implantations (SUN/Solaris 2.5, Linux, FreeBSD, Digital UNIX4.0, AIX 4.3, HP-UX 11.0, IRIX 6.2, openVMS 7.0 ...)
- Cette librairie contient les fonctions nécessaires pour:
 - La manipulation des threads (création, terminaison ...)
 - La synchronisation(verrous, variables condition)
 - Les primitives annexes: données spécifiques à chaque thread, politique d'ordonnancement ...
 -

2. Environnement de développement

□ Compilation

- Les fichiers C utilisant les threads devront comporter la directive « include<pthread.h> »
- Les fonctions de « pthread » se trouvent dans la librairie « libpthread », pour la compilation d'un fichier source C utilisant les threads, il faut rajouter « -lpthread » sur la ligne de commande lors de l'édition de liens.

□ Erreurs

- Les procédures de la librairie « pthread » renvoient 0 en cas de succès ou un code (>0) en cas d'erreur.

Exemple: Pour le programme « test_thread.c » on compile comme suit:

```
% gcc test_thread.c -o test_thread -lpthread
```

3. Création des threads

Chaque thread d'un processus possède un identifiant de thread de type « pthread_t » (correspond généralement au type entier).

Chaque thread exécute une fonction de thread. C'est une fonction qui contient le code que doit exécuter le thread. La terminaison de la fonction entraîne la terminaison du thread.

- Sous GNU/Linux, les fonctions de thread ne prennent qu'un seul paramètre de type « void* », ce paramètre est l'argument de thread. Le programme peut utiliser ce paramètre pour passer des données à un nouveau thread.
- La fonction a un type de retour « void* » qui peut être utilisé pour que le thread renvoie des données à son créateur lorsqu'il se termine.

La création d'un thread est réalisée grâce à la fonction « pthread_create() » déclarée dans <pthread.h>

Syntaxe:

```
# include<pthread.h>

int pthread_create ( pthread_t *idthread,
                    const pthread_attr_t *attr,
                    void * (*fonc) (void *),
                    void *arg );
```

La fonction « pthread_create() » crée un nouveau thread identifié par « idthread », à laquelle sont attribués les attributs « attr », pour exécuter la fonction « **fonc** » qui est appelée avec l'argument « **arg** ».

Remarque: Le retour de la fonction « **fonc** » correspond à la terminaison du thread.

- « idthread »: adresse de la zone mémoire (pointeur) dans laquelle l'identifiant du nouveau thread sera stocké en cas de succès.
- « attr »: un pointeur vers un objet d'attribut de thread. Les attributs sont utilisés pour définir la priorité et la politique d'ordonnancement. Si on passe «pthread_attr_default» ou 0 ou NULL, le thread est créé avec les attributs par défaut.
- « fonc »: un pointeur vers la fonction à exécuter dans le thread. Il contient le code à exécuter par le thread. La fonction doit avoir la forme « void *nom_fonction(void* arg) ».
- «arg»: adresse mémoire de(s) paramètre(s) à passer à la fonction. Cet argument est transmis à la fonction de thread lorsque celui-ci commence à s'exécuter. Le type de l'argument étant « void * », on pourra le transformer en n'importe quel type de pointeur pour passer un argument au thread. On pourra même employer une conversion explicite en « int » pour transmettre une valeur.

Remarque importante:

L'appel de la fonction « pthread_create() » se termine immédiatement et le thread créateur (qui a appelé « pthread_create () ») continue son exécution (exécution de l'instruction qui suit l'appel). Une fois le nouveau thread est prêt à être exécuter, Linux ordonnance les deux threads de manière asynchrone (concurrente).

Exemple 1: Dans le programme « test_create.c » suivant le thread principal (fonction « main() ») crée deux threads. Les threads créés affichent respectivement « n » et « m » fois les lettres 'a' et 'b' où n et m sont passés en paramètres aux fonctions threads.

```

#include <pthread.h>
#include <stdio.h>
void *affiche_a (void * arg);
void *affiche_b (void * arg);
int main () {
    int n=100, m=120;
    pthread_t thread_id1, thread_id2 ;
    /* créer le premier thread pour exécuter la fonction «affiche_a» */
    pthread_create (&thread_id1, NULL, affiche_a, (void *)&n );
    /*créer le deuxième thread pour exécuter la fonction «affiche_b»*/
    pthread_create (&thread_id2, NULL, affiche_b, (void *)&m);
    sleep(15);
    return 0;
}

```

El Mostafa DAOUDI- p.55

```

void* affiche_a (void * v) {
    int *cp=(int *)v; // conversion de *v en int (n=*v)
    int i;
    for (i=1;i<=*cp;i++) // affiche n fois 'a'
        fputc ('a', stderr );
}

void* affiche_b (void *v) {
    int *cp=(int *)v; // conversion de *v en int (m=*v)
    int i;
    for (i=1;i<=*cp; i++) // affiche m fois 'b'
        fputc ('b', stderr );
}

```

El Mostafa DAOUDI- p.56

Remarque:

Puisque les paramètres à passer aux fonctions threads sont des entiers de type « int » (les entiers « n » et « m »), les appels des fonctions de création des threads peuvent s'écrire tout simplement:

```
/* Le programme principal . */
#include <pthread.h>
#include <stdio.h>
void *affiche_a (void * arg);
void *affiche_b (void * arg);
int main () {
    int n=100, m=120;
    pthread_t thread_id1, thread_id2 ;
    /* créer le premier thread pour exécuter la fonction «affiche_a» */
    pthread_create (&thread_id1, NULL, affiche_a, (void *)n );
    /*créer le deuxième thread pour exécuter la fonction «affiche_b»*/
    pthread_create (& thread_id2, NULL, affiche_b, (void *)m);
    sleep(15);
    return 0;
}
```

Compilation:

```
% gcc -o test_create test_create.c -lpthread
```

Résultats d'exécution

```
% ./test_create
```

a

a

b

a

b

a

b

b

....

Le programme affiche 100 fois la lettre 'a' et 120 fois la lettre 'b'.
L'affichage des lettres est entrelacé mais imprévisible.

4. Variables locales et variables globales

- Les variables globales sont accessibles par tous les threads.
- Une variable locale à un thread est implanté sur sa propre pile, par conséquent elle n'est pas accessible par les autres threads.

Exemple 1: Variables locales

```
#include <pthread.h>
#include <stdio.h>
#define N 3
void * fonc_loc(void *arg){
    int n=8, m=10;
    n+=(int )arg;
    m+=n;
    printf("Dans thread numero : %d , n=%d , m=%d .\n ",(int) arg+1,n,m);
}
main(){
    int i, n=1, m=2;
    pthread_t  thread[N];
    for (i=0; i<N; i++)
        pthread_create(&thread[i], NULL, fonc_loc, (void *)i);
    printf("Dans main:  n = %d m = %d \n", n, m);
    sleep(10);
}
```

El Mostafa DAOUDI- p.61

Résultats d'exécution:

Dans main: n = 1 m = 2

Dans thread numero : 1 , n=8 , m=18 .

Dans thread numero : 3 , n=10 , m=20 .

Dans thread numero : 2 , n=9 , m=19 .

El Mostafa DAOUDI- p.62

Exemple 2: Variables globales

```
#include <pthread.h>
#include <stdio.h>
#define N 3
int n=10, m=8; // variables globales
void * fonc_glob(void *arg){
    n+=(int )arg;
    m+=n;
    printf("Dans thread numero : %d , n=%d , m=%d .\n ",(int) arg+1,n,m);
}
main(){
    int i;
    pthread_t thread[N];
    for (i=0; i<N; i++)
        pthread_create(&thread[i], NULL, fonc_glob, (void *)i);
    printf("Dans main:  n = %d  m = %d \n", n, m);
    sleep(10);
}
```

El Mostafa DAOUDI- p.63

Résultats d'exécution (imprévisible):

Dans main: n =10 m = 8

Dans thread numero : 1 , n=10 , m=18 .

Dans thread numero : 2 , n=11 , m=29 .

Dans thread numero : 3 , n=13 , m=42 .

El Mostafa DAOUDI- p.64

Exemple 3: Variables globales. On modifie l'exemple précédent en rajoutant un sleep(1) dans le thread 1.

```
#include <pthread.h>
#include <stdio.h>
#define N 3
int n=10, m=8;
void * fonction(void *arg);
main(){
    int i;
    pthread_t thread[N];
    for (i=0; i<N; i++){
        pthread_create(&thread[i], NULL, fonc_glob, (void *)i);
        printf("Dans main:  n = %d  m = %d  \n",n,m);
        sleep(10);
    }
```

El Mostafa DAOUDI- p.65

```
void * fonc_glob(void *arg){
    if( ((int) arg)==0)
        sleep(1);
    n+=(int )arg;
    m+=n;
    printf("Dans thread numero : %d , n=%d , m=%d  \n ",(int) arg+1,n,m);
}
```

Résultats d'exécution (imprévisible):

Dans main: n = 10 m = 8

Dans thread numero : 2 , n=11 , m=19 .

Dans thread numero : 3 , n=13 , m=32 .

Dans thread numero : 1 , n=13 , m=45 .

El Mostafa DAOUDI- p.66

5. Terminaison des threads

5.1. Terminaison de l'activité principal

La terminaison de l'activité initiale d'un processus (fonction «main()») entraîne la terminaison du processus et par conséquent la terminaison des autres threads encore actifs dans le cadre du processus.

5.2. Terminaison par exit

L'appel de « exit() » par un thread quelconque (principal ou fils) entraîne la fin du processus et par conséquent la terminaison de tous les threads du processus.

N.B. Seul le thread principal qui doit appeler « exit() » en s'assurant qu'il n'y ait plus de threads actifs.

5.3. La terminaison par la fonction «pthread_exit()»

La fonction «pthread_exit() » permet de terminer le thread qui l'appelle.

Syntaxe:

```
#include <pthread.h>
void pthread_exit (void *p_status);
```

- «*p_status »: pointeur sur le résultat éventuellement retourné par la thread. Il peut être accessible par les autres thread, du même processus, par l'intermédiaire de la fonction « pthread_join() ».

- Notez que « pthread_exit(NULL) » (pthread_exit(), pthread_exit(0)) est automatiquement exécuté en cas de terminaison du thread sans appel de «pthread_exit()».

6. Synchroniser sur terminaison des threads

6.1. La fonction « pthread_join() »

Un thread peut suspendre son exécution jusqu'à la terminaison d'un autre thread en appelant la fonction « pthread_join() ».

Syntaxe:

```
int pthread_join (pthread_t thrd, void **code_retour);
```

- « thrd » : désigne l'identifiant du thread qu'on attend sa terminaison (le thread qui l'appelle attend la fin du thread de TID « thrd »)
- « code_retour »: un pointeur vers une variable « void* » qui recevra la valeur de retour du thread qui s'est terminé et qui est renvoyé par «pthread_exit()».

El Mostafa DAOUDI- p.69

Exemple d'utilisation: le thread principal (la fonction main()) doit attendre la terminaison de tous les threads qu'il a créé avant de se terminer.

Remarque:

- Si le thread attendu se termine anormalement, la valeur retournée dans « code_retour » est -1.
- Si on n'a pas besoin de la valeur de retour du thread, on passe « NULL » au second argument.
- Au plus un thread peut attendre la fin d'un thread donné.

El Mostafa DAOUDI- p.70

Exemple: Dans le programme « test_create.c » le thread principal (fonction « main() ») crée deux threads. Les threads créés affichent respectivement « n » et « m » fois les lettres 'a' et 'b' où n et m sont passés en paramètres aux fonctions threads.

A la place de « sleep(10) » dans les exemples précédents, on appelle la fonction « pthread_join () ».

```
#include <pthread.h>
#include <stdio.h>
void* affiche_a (void *v) {
    int *cp=(int *)v;
    int i;
    for (i=1;i<=*cp;i++)
        fputc ('a', stderr );
}

void* affiche_b (void *v) {
    int *cp=(int *)v;
    int i;
    for (i=1; i<=*cp; i++)
        fputc ('b', stderr );
}
```

```

/* Le programme principal . */
int main () {
    int n=100, m=120;
    pthread_t thread_id1, thread_id2 ;
    /* créer le premier thread pour exécuter la fonction «affiche_a» */
    pthread_create (&thread_id1, NULL, affiche_a, (void *)n );
    /*créer le deuxième thread pour exécuter la fonction «affiche_a»*/
    pthread_create (& thread_id2, NULL, affiche_b, (void *)m);

    /* S'assurer que le premier thread « thread_id1 » est terminé . */
    pthread_join ( thread_id1 , NULL );

    /* S'assurer que le second thread « thread_id2 » est terminé . */
    pthread_join ( thread_id2 , NULL );
    return 0;
}

```

El Mostafa DAOUDI- p.73

6.2. Valeurs de retour des threads

Si le second argument de « pthread_join() » n'est pas NULL, la valeur de retour du thread sera stockée à l'emplacement pointé par cet argument.

Exemple:

Un thread lit une valeur au clavier puis la transmet au thread principal.

```

include <pthread.h>
#include <stdio.h>
# define N 3
void *fonc_thread(void *v) {
    int k;
    scanf("%d",&k);
    pthread_exit((void *)k);
}

```

El Mostafa DAOUDI- p.74

```

#include <pthread.h>
#include <stdio.h>
void *fonc_thread(void *v) {
    int k;
    scanf("%d",&k);
    pthread_exit((void *)k);
}
int main(){
    int donnee;
    void * res;
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, fonc_thread, NULL);
    pthread_join(thread_id, &res );
    donnee=(int) res;
    printf(" donnee = %d \n", donnee);
    return 0;
}

```

Remarque:

- « pthread_join() »: Suspend l'exécution du thread appelant jusqu'à la terminaison du thread indiqué en argument.
- « pthread_join() »: Rempli le pointeur passé en second argument avec la valeur de retour du thread.

III. Synchronisation

1. Introduction

- Les threads au sein du même processus sont concurrents et peuvent avoir un accès simultané à des ressources partagées (variables globales, descripteurs de fichiers, etc.). Si leur accès n'est pas contrôlé, le résultat de l'exécution du programme pourra dépendre de l'ordre d'entrelacement de l'exécution des instructions.
⇒ leur synchronisation est indispensable afin d'assurer une utilisation cohérente de ces données.
- Le problème de synchronisation se pose essentiellement lorsque:
 - Deux threads concurrents veulent accéder en écriture à une variable globale.
 - Un thread modifie une variable globale tandis qu'un autre thread essaye de la lire.

Des mécanismes sont fournis pour permettre la synchronisation des différents threads au sein de la même tâche:

- La primitive « `pthread_join()` » (synchronisation sur terminaison c'est déjà vu): permet à un thread d'attendre la terminaison d'un autre.
- Synchronisation avec une attente active (solution algorithmique).
- Les « mutex » (sémaphores d'exclusion mutuelle): C'est un mécanisme qui permet de résoudre le problème de l'exclusion mutuelle des threads.
- Les conditions (événements).

2. Exemples introductifs

Exemple 1

Soient «thread1» et «thread2» deux threads concurrents. Supposons que le thread1 exécute la séquence d'instructions «A1» et le thread2 exécute la séquence d'instructions «A2».

Les deux séquences d'instructions consistent à décrémenter une variable globale « V » initialisé à « v0 ».

thread 1

A1: x variable locale

- | |
|-----------|
| 1. x=V-1; |
| 2. V=x; |

thread2

A2: x variable locale

- | |
|-----------|
| 1. x=V-1; |
| 2. V=x; |

Les deux threads partagent la même variable globale «V». Le résultat de l'exécution concurrente de «A1» et «A2» dépend de l'ordre de leur entrelacement (ordre des commutations d'exécution des deux threads).

Premier scénario: Supposons que c'est le « thread1 » qui commence l'exécution et que l'ordonnanceur ne commute les threads et alloue le processeur au « thread2 » qu'après la fin d'exécution de « A1 ».

Le premier thread (exécution de « A1 »)

- lit la valeur initiale « v0 » dans un registre du processeur.
- décrémente la valeur de « V » d'une unité (exécute l'instruction « x=V-1 »).
- écrit la nouvelle valeur dans la variable « V » (instruction « V=x »). La nouvelle valeur de « V » est « v0-1 »

Ensuite l'ordonnanceur commute les tâches et alloue le processeur au deuxième thread.

Le deuxième thread (exécution de « A2 »)

- lit la valeur de « V » qui est « v0-1 ».
- décrémente la valeur de « V » d'une unité (exécute l'instruction « x=V-1 »).
- écrit la nouvelle valeur dans la variable « V » (instruction « V=x »). La nouvelle valeur de « V » est « v0-2 »;

Donc après la fin d'exécution des deux threads, la valeur finale de « V » est égale à « v0-2 » ce qui est attendu.

Deuxième scénario: On suppose que l'ordonnanceur commute les deux threads et alloue le processeur entre les deux threads avant la fin de leur d'exécution. Supposons que c'est le « thread1 » qui commence l'exécution.

- Le « thread1 » lit la valeur initiale « v0 » puis effectue l'opération « $x = V - 1$ ».
- Avant que le « thread1 » écrit la nouvelle valeur dans la variable « V » (instruction « $V = x$ »), l'ordonnanceur commute les threads et alloue le processeur au « thread2 ».
- Le « thread2 » lit la valeur initiale de « V » (soit v0) et effectue les opérations « $x = V - 1$ » et « $V = x$ ». La nouvelle valeur de « V » devienne « v0-1 ».
- L'ordonnanceur réactive le premier thread qui continue son exécution au point où il était arrêté, c'est-à-dire effectue l'opération « $V = x$ », avec la valeur de x qui est « v0-1 ».

Les opérations dans les threads sont effectuées dans l'ordre suivant:

[thread1.1](#); [thread2.1](#); [thread2.2](#); [thread1.2](#).

Donc, après l'exécution des instruction dans cet ordre, la valeur finale de « V » est égale à « v0-1 » au lieu de « v0-2 » (valeur attendue).

Exemple 2: Soit « tab » un tableau globale d'entiers. Soient « thread1 » et « thread2 » deux threads concurrents: l'un remplit le tableau « tab » (modifie les éléments du tableau) et l'autre affiche le contenu du tableau après modification.

[thread1](#)

pour (i=0; i<N; i++)
 tab[i]=2*i;

[thread2](#)

pour (i=0; i<N; i++)
 afficher (tab[i]);

Puisque l'ordonnanceur commute entre les deux threads: il est possible que le « thread2 » commence son exécution avant la fin du remplissage du tableau (résultats erronés)

- ⇒ Pour avoir un affichage cohérent, on doit appliquer un mécanisme de synchronisation.
- ⇒ S'assurer que l'affichage ne peut avoir lieu qu'après la fin de remplissage du tableau.

3. Section Critique

Une section critique est une suite d'instructions dans un programme dans laquelle se font des accès concurrents à une ressource partagée, et qui peuvent produire des résultats non cohérents et imprévisibles lorsqu'elles sont exécutées simultanément par des threads différents.

- ⇒ l'accès à cette section critique doit se faire en exclusion mutuelle, c'est-à-dire deux threads ne peuvent s'y trouver au même instant (un seul thread au plus peut être dans la section critique à un instant donné).
- ⇒ l'exécution de cette partie du code est indivisible ou atomique. C'est-à-dire si un thread commence l'exécution, aucun autre thread ne peut l'exécuter avant la fin d'exécution du premier thread.

Exemple: Reprenons l'exemple 1.

- Puisque les deux threads accèdent de manière concurrente à la même variable partagée « V », alors la valeur finale dépend de la façon d'exécution des deux séquences d'instructions « A1 » et « A2 » par les deux threads.
 - ⇒ Les séquences d'instructions « A1 » et « A2 » sont des sections critiques. Elle opèrent sur une variable partagée qui est «V».
- Pour éviter le problème de concurrence, on doit synchroniser les deux threads, c'est-à-dire s'assurer que l'ensemble des opérations sur la variable partagée (accès + mise à jour) est exécuté de manière atomique (indivisible).
 - ⇒ Si les exécutions de « A1 » et « A2 » sont atomiques, alors le résultat de l'exécution de « A1 » et « A2 » ne peut être que celui de « A1 » suivie de « A2 » ou de « A2 » suivie de « A1 ».

4. Mécanisme de synchronisation sur attente active

Reprenons l'exemple2: modification et affichage des éléments du tableau.

La synchronisation sur attente active est une solution algorithmique qui permet, par exemple, de bloquer l'exécution des instructions d'affichage du tableau (dans « thread2 ») jusqu'à la fin d'exécution des instructions des mises à jour du tableau (dans « thread1 »).

De manière générale, le prototype de l'attente active est le suivant (ressource est globale):

<u>Thread 1</u>	<u>Thread 2</u>
ressource occupée = true; utiliser ressource; /* Section critique : exécution de la partie qui nécessite la synchronisation: */ ressource occupée = false;	while (ressource occupée) { }; /* Rester boucler jusqu'à ce que ressource occupée devienne false */ ressource occupée = true;

- Méthode très peu économique: « thread2 » occupe le processeur pour une boucle vide.

Solution avec une attente active:

```
include <pthread.h>
#include <stdio.h>
# define N 300
pthread_t thread1,thread2;

int ressource=0; /* variable globale */

void *fonc_thread1(void *n);
void *fonc_thread2(void *n);
main(){
    pthread_create(&thread1, NULL, fonc_thread1, NULL );
    pthread_create(&thread2, NULL, fonc_thread2, NULL);
    pthread_join(thread1,NULL);
    pthread_join(thread2, NULL);
}
```

El Mostafa DAOUDI- p.87

```
void *fonc_thread1(void *n) {
    int i;
    for(i=0;i<N;i++)
        tab[i]=2*i;
    ressource=1;
    /* modification de la variable « ressource » après avoir mis à jour
    le tableau, c'est-à-dire après avoir été sortie de la section critique*/
}
void *fonc_thread2(void *n) {
    int i;
    while (ressource==0); /* attend que la valeur de « ressource »
                           soit ≠ de 0 (boucle vide) avant d'entrer
                           dans la section critique */
    for(i=0;i<N;i++)
        printf("%d",tab[i]);
}
```

El Mostafa DAOUDI- p.88

5. Les Mutexs (sémaphore d'exclusion mutuelle)

Les mutex, raccourcis de MUTual EXclusion locks (verrous d'exclusion mutuelle), sont des objets de type «pthread_mutex_t » qui permettent de mettre en œuvre un mécanisme de synchronisation qui résout le problème de l'exclusion mutuelle.

⇒ éviter que des ressources partagées d'un système ne soient utilisées en même temps par plus qu'un thread.

Il existe deux états pour un mutex (disponible ou verrouillé), et essentiellement deux fonctions de manipulation des mutex (une fonction de verrouillage et une fonction de libération).

- Lorsqu'un mutex est verrouillé par un thread, on dit que ce thread tient le mutex. Tant que le thread tient le mutex, aucun autre thread ne peut accéder à la ressource critique.
⇒ Un mutex ne peut être tenu que par un seul thread à la fois.
- Lorsqu'un thread demande à verrouiller un mutex déjà maintenu par un autre thread, le thread demandeur est bloqué jusqu'à ce que le mutex soit libéré.

Le principe des mutex est basé sur l'algorithme de Dijkstra

- Opération P (verrouiller l'accès).
- Accès à la ressource critique (la variable globale).
- Opération V (libérer l'accès).

Les fonctions de manipulations des mutexs:

pthread_mutex_init (): permet de créer le mutex (le verrou) et le mettre à l'état "unlock" (ouvert ou disponible).

pthread_mutex_destroy(): permet de détruire le mutex.

pthread_mutex_lock (): tentative d'avoir le mutex. S'il est déjà pris, le thread est bloqué

pthread_mutex_trylock(): appel non bloquant. Si le mutex est déjà pris, le thread n'est pas bloqué

pthread_mutex_unlock() : rend le mutex et libère un thread

5.1. Création de mutex

La création de mutex (verrou) consiste à définir un objet de type « pthread_mutex_t » et de l'initialiser de deux manières.

- Initialisation statique à l'aide de la constante « PTHREAD_MUTEX_INITIALIZER »
- Initialisation par appel de la fonction « pthread_mutex_init() » déclarée dans <pthread.h>, qui permet de créer le verrou(le mutex) et le mettre en état "unlock".

Syntaxe:

```
int pthread_mutex_init(pthread_mutex_t *mutex_pt,  
pthread_mutexattr_t *attr);
```

- « mutex_pt » : pointe sur une zone réservée pour contenir le mutex créé.
- attr : ensemble d'attributs à affecter au mutex. On le met à NULL

Code retour de la fonction:

0 : en cas de succès
!=0 : en cas d'échec

5.2. L'opération P pour un mutex.

Appel bloquant:

La fonction « pthread_mutex_lock() » permet, à un thread de réaliser de façon atomique, une opération P (verrouillé le mutex) par un thread. Si le mutex est déjà verrouillé (tenu par un autre thread), alors le thread qui appelle la fonction « pthread_lock() » reste bloqué jusqu'à la réalisation de l'opération V (libérer le mutex) par un autre thread.

Syntaxe:

```
int pthread_mutex_lock(pthread_mutex_t *mutex_pt);
```

- mutex_pt : pointe sur le mutex à réserver (à verrouiller)

Code retour de la fonction :

0 : en cas de succès

!=0 : en cas d'erreur

Appel non bloquant

La fonction « pthread_mutex_trylock » permet, de façon atomique, de réserver un mutex ou de renvoyer une valeur particulière si le mutex est réservé par un autre thread, le thread n'est pas bloqué.

Syntaxe:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex_pt);
```

- « mutex_pt » : pointe sur le mutex à réserver

Code retour de la fonction:

1: en cas de réservation

0 : en cas d'échec de la réservation

!=0 : en cas d'erreur

5.3. L'opération V pour un mutex

L'appel de la fonction « pthread_mutex_unlock() » permet de libérer un mutex et de débloquer les threads en attente sur ce mutex.

Syntaxe:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex_pt);
```

- mutex_pt : pointe sur le mutex à libérer

Code retour de la fonction :

0 : en cas de succès

!=0 : en cas d'erreur

Exemple:

Reprendre l'exemple2 sur les tableaux. Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher son contenu en utilisant les sémaphores d'exclusion mutuelle (les mutex)

Solution avec les mutex

```
#include <stdio.h>
#include <pthread.h>
#define N 1000
pthread_t      th1, th2;
pthread_mutex_t mutex;
int            tab[N];
void *écriture_tab (void * arg);
void *lecture_tab (void * arg);
main ( ) {
    pthread_mutex_init (&mutex, NULL);
    pthread_mutex_lock (&mutex);
    pthread_create (&th1, NULL, écriture_tab, NULL) ;
    pthread_create (&th2, NULL, lecture_tab, NULL);
    pthread_join (th1, NULL);
    // pthread_mutex_unlock (&mutex);
    pthread_join(th2,NULL);
}
```

El Mostafa DAOUDI- p.97

```
void *écriture_tab (void * arg) {
    int i;
    for (i = 0 ; i <N ; i++) {
        tab[i] = 2 * i;
        printf ("écriture, tab[%d] vaut %d\n", i, tab[i]);
    }
    pthread_mutex_unlock (&mutex);
    pthread_exit (NULL);
}

void *lecture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    for (i = 0 ; i < N ; i++)
        printf ("lecture, tab[%d] vaut %d\n", i, tab[i]);
    pthread_mutex_unlock (&mutex);
}
```

El Mostafa DAOUDI- p.98

5.4. La fonction de destruction d'un mutex

On peut détruire le sémaphore par un appel à la fonction «pthread_mutex_destroy() ».

Syntaxe:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex_pt);
```

- mutex_pt : pointe sur le mutex à détruire

Code retour de la fonction:

0 : en cas de succès

!=0 : en cas d'erreur

6. Les conditions (événements)

Une condition est un mécanisme permettant de synchroniser plusieurs threads à l'intérieur d'une section critique.

C'est une autre technique de synchronisation qui utilise les variables de « conditions » représentées par le type « pthread_cond_t ».

6.1. Principe d'utilisation des conditions:

- Le principe consiste à bloquer un thread (une activité) sur une attente d'évènement (le thread se met en attente d'une condition). Lorsque la condition est réalisée par un autre thread, ce dernier l'en avertit directement. Pour cela on utilise essentiellement deux fonctions de manipulation des conditions:
 - l'une est l'attente de la condition, le thread appelant reste bloqué jusqu'à ce qu'elle soit réalisée;
 - et l'autre sert à signaler que la condition est remplie.
- La variable condition est considérée comme une variable booléenne un peut spéciale par la bibliothèque «*pthread*». Elle est toujours associée à un « *mutex* », afin d'éviter les problèmes de concurrences.
 - La variable de condition sert à la transmission des changements d'état.
 - Le mutex assure un accès protégé à la variable.

Le thread qui doit attendre une condition

- On initialise la variable condition et le *mutex* qui lui est associé.
- Le thread bloque le *mutex*. Ensuite, il invoque une routine d'attente (par exemple la routine « *pthread_cond_wait()* ») qui attend que la condition soit réalisée.
- Le thread libère le mutex.

Le thread qui réalise la condition

- Le thread travaille jusqu'à avoir réalisé la condition attendue
- Il bloque le mutex associé à la condition
- Le thread appelle la fonction « *pthread_cond_signal()* » pour montrer que la condition est remplie.
- Le thread débloquent le mutex.

6.3. Création (Initialisation) des variables de condition

Une condition est de type « pthread_cond_t » peut être initialisée:

- de manière statique:
pthread_cond_t condition=PTHREAD_COND_INITIALIZER;
- par appel de la fonction « pthread_cond_init() » qui permet de créer une nouvelle condition.

Syntaxe:

```
int pthread_cond_init(pthread_cond_t * cond_pt, pthread_condattr_t *attr);
```

- « cond_pt » : pointeur sur la zone réservée pour recevoir la condition (pointe sur la nouvelle condition).
- « attr » : attributs à donner à la condition lors de sa création. Il est mis à NULL.

Code retour de la fonction:

0 : en cas de succès
!=0 : en cas d'erreur.

6.4. fonctions d'attente sur une condition

La fonction « pthread_cond_wait() » permet l'attente d'une condition envoyée par exemple par la fonction « pthread_cond_signal() ».

Syntaxe:

```
int pthread_cond_wait(pthread_cond_t*cond_pt, pthread_mutex_t*mutex_pt);
```

- « cond_pt » : pointeur sur la condition à attendre
- « mutex_pt » : pointeur sur le mutex à libérer pendant l'attente.

Code retour de la fonction:

0 : en cas de succès
!=0 : en cas d'erreur.

Principe de fonctionnement:

Le thread appelant doit avoir verrouiller au préalable le mutex «mutex_pt». L'appel a pour effet:

- libérer le mutex « mutex_pt »
- attendre un signal sur la condition désignée et de reprendre son exécution, en verrouillant à nouveau le mutex.

6.5. Réveil d'un thread: fonction de signal d'une condition

La fonction « pthread_cond_signal() » permet de réveiller un des threads attendant la condition désignée.

- Si aucun thread ne l'attendait, le signal est perdu.
- Si plusieurs threads attendent sur la même condition, un seul d'entre eux est réveillé mais on ne peut pas prédire lequel.

Syntaxe:

```
int pthread_cond_signal(pthread_cond_t *cond_pt);
```

- « cond_pt » : pointeur sur la condition à signaler

Code retour de la fonction:

0 : en cas de succès.

!=0 : en cas d'erreur.

6.6. Réveil de tous les threads

La fonction « pthread_cond_broadcast() » permet de réveiller tous les threads en attente sur une condition.

Syntaxe

```
#include <pthread.h>
```

```
int pthread_cond_broadcast(pthread_cond_t *cond_pt);
```

- cond_pt : pointeur sur la condition à signaler

Code retour de la fonction:

0 : en cas de succès

!=0 : en cas d'erreur

6.7. La fonction de destruction d'une condition

Une condition non utilisée peut être libérée par appel de la fonction « `pthread_cond_destroy()` ». Aucun autre thread ne doit être en attente sur la condition, sinon la libération échoue sur l'erreur EBUSY.

Syntaxe:

```
int pthread_cond_destroy(pthread_cond_t *cond_pt);
```

Permet de détruire les ressources associées à une condition

- `cond_pt` : pointeur sur la condition à détruire.

Code retour de la fonction :

0 : en cas de succès.

!=0 : en cas d'erreur.

Schéma de mise en œuvre des conditions

Thread attendant la condition	Thread signalant la condition
Appel de <code>pthread_mutex_lock()</code> : blocage du mutex associé à la condition	
Appel de <code>pthread_cond_wait()</code> : déblocage du mutex	
.... Attente	
	Appel de <code>pthread_mutex_lock()</code> sur le mutex
	Appel de <code>pthread_cond_signal()</code> , qui réveille l'autre thread
Dans <code>pthread_cond_wait()</code> , tentative de récupérer le mutex. Blocage	
	Appel de <code>pthread_mutex_unlock()</code> . Le mutex étant libéré, l'autre thread se débloque
Fin de <code>pthread_cond_wait()</code>	
Appel de <code>pthread_mutex_unlock()</code> pour revenir à l'état initial.	

Exemple

Refaire l'exemple2 sur la lecture et l'écriture d'un tableau global en utilisant les variables de conditions. Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher son contenu.

```
#include <stdio.h>
#include <pthread.h>
#define N 100
int tab[N];
pthread_t thread1, thread2;
pthread_mutex_t mutex;
pthread_cond_t condition;
void *ecriture_tab (void * arg);
void *lecture_tab (void * arg);
main () {
    pthread_mutex_init (&mutex, NULL);
    pthread_cond_init (&condition, NULL);
    pthread_create (&th1, NULL, ecriture_tab, NULL);
    pthread_create (&th2, NULL, lecture_tab, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
}
```

```

void *ecriture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    for (i = 0 ; i < N ; i++)
        tab[i] = 2 * i;
    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
}

void *lecture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    pthread_cond_wait(&condition,&mutex);
    for (i = 0 ; i < N ; i++)
        printf ("lecture, tab[%d] vaut %d\n", i, tab[i]);
    pthread_mutex_unlock (&mutex);
}

```

El Mostafa DAOUDI- p.111

Problème, on risque de perdre le signal. Au moment du réveil l'autre thread n'est pas encore dans l'état d'attente.

Une solution exacte peut être élaborée comme suit:

El Mostafa DAOUDI- p.112


```

#include <stdio.h>
#include <pthread.h>
#define N 100
int tab[N];
pthread_t thread1, thread2;
pthread_mutex_t mutex;
pthread_cond_t condition;
int test=0;
void *ecriture_tab (void * arg);
void *lecture_tab (void * arg);
main ( ) {
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&condition, NULL);
    pthread_create (&th1, NULL, ecriture_tab, NULL) ;
    pthread_create (&th2, NULL, lecture_tab, NULL);
    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
}

```

El Mostafa DAOUDI- p.113

```

void *ecriture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    for (i = 0 ; i <N ; i++)
        tab[i] = 2 * i;
    test=1;
    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
}
void *lecture_tab (void * arg) {
    int i;
    pthread_mutex_lock (&mutex);
    while (test==0)
        pthread_cond_wait(&condition,&mutex);
    for (i = 0 ; i < N ; i++)
        printf ("lecture, tab[%d] vaut %d\n", i, tab[i]);
    pthread_mutex_unlock (&mutex);
}

```

El Mostafa DAOUDI- p.114

Ch.III. Gestion des signaux sous Unix

1. Introduction

- Un signal (ou interruption logicielle) est un message très court qui permet d'interrompre un processus pour exécuter une autre tâche.
- Il permet d'avertir (informer) un processus qu'un événement (particulier, exceptionnel, important, ...) c'est produit.
- Le processus récepteur du signal peut réagir à cet événement sans être obligé de le tester. Il peut:
 - Soit ignorer le signal.
 - Soit laisser le système traiter le signal avec un comportement par défaut.
 - Soit le capturer, c'est-à-dire dérouter (changer) provisoirement son exécution vers une routine particulière qu'on nomme gestionnaire de signaux (signal handler).

2. Emetteurs d'un signal

Un signal peut être envoyé par:

- Le système d'exploitation (déroutement : événement intérieur au processus généré par le hard (le matériel)) pour informer les processus utilisateurs d'un événement anormal (une erreur) qui vient de se produire durant l'exécution d'un programme (erreur virgule flottante (division par 0), violation mémoire,).
- Un processus utilisateur: pour se coordonner avec les autres, par exemple pour gérer une exécution multi-processus plus ou moins complexe (par exemple du branch-and-bound).
- La frappe d'une touche par l'utilisateur du terminal (par exemple CTRL-C (interruption), CTRL \ génèrent les signaux SIGINT et SIGQUIT dont l'action par défaut est de terminer la processus).

3. Caractéristiques des signaux

Il existe un nombre « NSIG » (ou « _NSIG») de signaux différents. Ces constantes sont définies dans « signal.h ».

- Chaque signal est identifié par:
 - un nom défini sous forme de constante symbolique commençant par « SIG ». Les noms sont définis dans le fichier d'en-tête <signal.h>
 - un numéro, allant de 1 à « NSIG ».
- Attention:** la numérotation peut différer selon le système.
- La commande
% kill -l
permet d'afficher la liste de tous les signaux définis dans le système.
- On ne connaît pas l'émetteur du signal.
- Le signal « 0 » a un rôle particulier. On l'utilise pour vérifier l'existence d'un processus.

- Si le processus exécute un programme utilisateur : traitement immédiat du signal reçu,
- S'il se trouve dans une fonction du système (ou system call) : le traitement du signal est différé jusqu'à ce qu'il revienne en mode utilisateur, c'est à dire lorsqu'il sort de cette fonction.

4. Etats des signaux

- Un signal pendant (pending) est un signal en attente d'être pris en compte.
- Un signal est délivré lorsqu'il est pris en compte par le processus qui le reçoit.
- La prise en compte d'un signal entraîne l'exécution d'une fonction spécifique appelée « handler »: c'est l'action associée au signal. Elle peut être soit:
 - La fonction prédéfinie dans le système (action par défaut).
 - Une fonction définie par l'utilisateur pour personnaliser le traitement du signal.

Liste incomplète des signaux

1. SIGHUP : rupture de connexion
2. SIGINT : interruption terminal
3. SIGQUIT : signal quitter du terminal
4. SIGILL : instruction illégale
5. SIGTRAP : point d'arrêt exécution pas à pas
6. SIGABRT : terminaison anormale du processus.
7. SIGBUS : erreur accès bus
8. SIGFPE : erreur arithmétique
9. SIGKILL : terminaison impérative. Ne peut être ignoré ou intercepter
10. SIGUSR1 : signal utilisateur 1
11. SIGSEGV : accès mémoire invalide
12. SIGUSR2 : signal utilisateur 2
13. SIGPIPE : écriture dans un conduit sans lecteur disponible
14. SIGALRM : alarme horloge: expiration de timer
15. SIGTERM : signal 'terminer' du terminal
17. SIGCHLD : processus fils stoppé ou terminé
18. SIGCONT : continuer une exécution interrompue
19. SIGSTOP : interrompre l'exécution. Ne peut être ignoré ou intercepter
20. SIGTSTP : signal d'arrêt d'exécution généré par le terminal
21. SIGTTIN : processus en arrière plan essayant de lire le terminal
22. SIGTTOU : processus en arrière plan essayant d'écrire sur le terminal
23. SIGURG : donnée disponible à un socket avec bande passante élevée
24. SIGXCPU : quota de temps CPU dépassé
25. SIGXFSZ : taille maximale de fichier dépassée
26. SIGVTALRM : échéancier virtuel expiré
27. SIGPROF : expiration de l'échéancier de profilage
31. SIGSYS : appel système invalide

5. Les différents traitements par défaut des signaux

A chaque type de signal est associé un gestionnaire du signal (« handler ») par défaut appelé « SIG_DFL ».

- Les 5 traitements par défaut disponibles sont :
 - terminaison normal du processus.
 - terminaison anormal du processus (par exemple avec fichier core).
 - signal ignoré (signal sans effet).
 - stoppe le processus (le processus est suspendu).
 - continuation d'un processus stoppé.
- Un processus peut ignorer un signal en lui associant le handler «SIG_IGN ».
- « SIG_DFL » et « SIG_IGN » sont les deux seules macros prédéfinies.

6. Signaux particuliers

Pour tous les signaux, l'utilisateur peut remplacer le « handler » par défaut par un « handler » personnalisé qui sera défini dans son programme, à l'exception de certains signaux qui ont des statuts particuliers et qui ne peuvent pas être interceptés, bloqués ou ignorés :

- « SIGKILL » permet de tuer un processus.
- « SIGSTOP » permet de stopper un processus (stopper pour reprendre plus tard, pas arrêter).
- « SIGCONT » permet de faire reprendre l'exécution d'un processus stoppé (après un « SIGSTOP »).

7. Manipulation des signaux

- Un processus peut envoyer un signal à un autre processus ou à un groupe de processus.
- Un processus qui reçoit le signal agit en fonction de l'action (du « handler ») associé au signal.
- On peut mettre, à la place des « handlers » définis par défaut, des « handlers » particuliers permettant un traitement personnalisé.
- La manipulation des signaux peut se faire:
 - Par la ligne de commande (frappe au clavier). Par exemple des combinaisons des touches Ctrl-C, Ctrl-Z, Ctrl-Q.
 - Dans un programme utilisateur, essentiellement par les deux primitives principales :
 - ✓ la fonction « kill() » pour envoyer des signaux.
 - ✓ les fonctions « signal() » ou « sigaction() » pour mettre en place une action personnalisée à un signal donné.

El Mostafa DAOUDI- p.123

7.1. Envoi d'un signal par un processus

Les processus ayant le même propriétaire peuvent communiquer entre eux en utilisant les signaux. La primitive « kill() » permet d'envoyer un signal « sig » vers un ou plusieurs processus.

Syntaxe

int kill(pid_t pid, int sig) ;

- « sig » désigne le signal à envoyer (on donne le nom ou le numéro du signal).
- Quand « sig » est égal à 0 aucun signal n'est envoyé, mais la valeur de retour de la primitive « kill() » permet de tester l'existence ou non du processus « pid » (si kill(pid,0) retourne 0 (pas d'erreur) le processus de numéro « pid » existe).

El Mostafa DAOUDI- p.124

- « pid » désigne le ou les processus destinataires (récepteurs du signal).
 - Si $\text{pid} > 0$, le signal est envoyé au processus d'identité « pid ».
 - Si $\text{pid}=0$, le signal est envoyé à tous les processus qui sont dans le même groupe que le processus émetteur (processus qui a appelé la primitive « kill() »).
- Remarque:** Cette possibilité peut être utilisée avec la commande shell « % kill -9 0 » pour tuer tous les processus en arrière-plan sans indiquer leurs identificateurs de processus).
- Si $\text{pid}=-1$, le signal est envoyé à tous les processus (non-conforme POSIX).
- Si $\text{pid} < -1$, le signal est envoyé à tous les processus du groupe du processus de numéro |pid|.
- La primitive « kill() » renvoie 0 si le signal est envoyé et -1 en cas d'échec.

Remarque:

- « kill » signifie tuer en anglais, mais son rôle n'est pas seulement d'envoyer un signal pour tuer un processus.
- Les processus endormis (états bloqués) sont réveillés par l'arrivée d'un signal et passent à l'état prêt.
- Les signaux sont sans effet sur les processus zombies (Tout processus qui se termine passe dans l'état « *zombi* » tant que son père n'a pas pris connaissance de sa terminaison).

Exemple1: Considérons le programme « test1.c » suivant:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void main(void) {
    pid_t p;
    if ((p=fork()) == 0) { /* processus fils qui boucle */
        while (1);
    }
    else { /* processus père */
        sleep(10);
        printf("Fin du processus père %d : \n", getpid());
    }
}
```

El Mostafa DAOUDI- p.127

Résultats d'exécution:

```
% ./test1 & ps -f
UID      PID      PPID      ...      CMD
etudiant  2763      2761      ...      bash
etudiant  3780      2763      ....     ./test1
etudiant  3782      3780      ...      ./test1
% Fin du processus père: 3780
```

Après quelques secondes, le programme affiche le message prévu dans «printf()» et se termine.

```
% ps -f
UID      PID      PPID      ...      CMD
etudiant  2763      2761      ...      bash
etudiant  3782      1        ...      ./test1
```

Remarque:

Le processus père a terminé son exécution mais le fils continue son exécution et devient orphelin (devient le fils du processus «init»).

El Mostafa DAOUDI- p.128

Exemple2: On modifie le programme précédent en envoyant un signal (le signal « SIGUSR1 ») au processus fils. Ce signal provoquera la terminaison du processus fils.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void main(void) {
    pid_t p;
    if ((p=fork()) == 0) { /* processus fils qui boucle */
        while (1);
    }
    else { /* processus père */
        sleep(10);
        printf("Envoi de SIGUSR1 au fils %d\n", p);
        kill(p, SIGUSR1);
        printf("Fin du processus père %d\n", getpid());
    }
}
```

El Mostafa DAOUDI- p.129

Résultats d'exécution:

```
% ./test1 & ps -f
UID      PID      PPID      ...      CMD
etudiant  2763      2761      ...      bash
etudiant  4031      2763      ....     ./test1
etudiant  4033      4031      ...      ./test1
```

```
% Envoi de SIGUSR1 au fils 4033
```

```
Fin du processus père 4031
```

Après quelques secondes, le programme affiche les message prévus dans «printf()» et se termine.

```
% ps -f
UID      PID      PPID      ...      CMD
etudiant  2763      2761      ...      bash
```

Remarque:

Le processus père a interrompu l'exécution de son fils par l'envoi du signal «SIGUSR1».

El Mostafa DAOUDI- p.130

Exemple3: Envoie d'un signal (le signal « SIGUSR1 ») à tous les processus

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void main(void) {
    pid_t p1, p2;
    if ((p1=fork()) != 0) { /* processus père */
        if ((p2=fork()) == 0) { /* processus fils 2 qui boucle */
            while (1);
        }
        sleep(10);
        printf(« Envoi de SIGUSR1 aux fils %d et %d \n", p1,p2);
        kill(0, SIGUSR1);
        printf("Fin du processus père %d\n", getpid());
    }
    else { /* processus fils 1 qui boucle */
        while (1);
    }
}
```

El Mostafa DAOUDI- p.131

Résultats d'exécution:

% ./test1 & ps -f

UID	PID	PPID	...	CMD
etudiant	1635	1633	...	bash
etudiant	2007	1635/test1
etudiant	2008	2007/test1
etudiant	2009	2007/test1

% Envoi de SIGUSR1 aux fils 2008 et 2009

Après quelques secondes, le programme affiche les message prévu dans « printf() » et se termine.

% ps -f

UID	PID	PPID	...	CMD
etudiant	1635	1633	...	bash

Remarque:

Le processus père a interrompu l'exécution de ses deux fils (envoi du signal «SIGUSR1»).

El Mostafa DAOUDI- p.132

Exemple4: Tester si un processus spécifié existe

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
void main(void) {
    .....
    sleep(2);
    if ( kill(p, 0)==0)
        printf(" le processus %d  existe d\n",p);
    else
        printf(" le processus %d  n'existe pas d\n",p );
}
```

7.2. Traitement personnalisé de signal (Mise en place d'un handler)

Les signaux, autres que « SIGKILL », «SIGCONT» et «SIGSTOP», peuvent avoir un « handler » spécifique installé par un processus.

- La primitive « signal() » peut être utilisée pour installer des handlers personnalisés pour le traitement des signaux. Cette primitive fait partie du standard de C.

Syntaxe:

```
#include<signal.h>
signal (int sig, new_handler);
```

- Elle met en place le handler spécifié par « new_handler() » pour le signal « sig ».
- La fonction « new_handler » est exécutée par le processus à la délivrance (la réception) du signal. Elle reçoit le numéro du signal.
- A la fin de l'exécution de cette fonction, l'exécution du processus reprend au point où elle a été suspendue.

Exemple1: test.c

```
#include <stdio.h>
#include <signal.h>
```

```
int main(void) {
    for (;;) { }
    return 0;
}
```

Résultat d'exécution:

%. /test

Le programme boucle.

Si on appuie sur CTRL-C le programme s'arrête.

Exemple 2: On met en place un handler dans le programme « test.c » qui permet de terminer le processus seulement lorsqu'on appuie deux fois sur «Ctrl-C» (signal «SIGINT»).

Programme 1:

```
#include <stdio.h>
#include <signal.h>
void hand(int signum) {
    printf(" Numéro du signal est %d  \n", signum);
    printf("Il faut appuyer sur Ctrl-C une 2ème fois pour terminer \n");
}
int main(void) {
    signal(SIGINT, hand); /* installation du nouveau handler */
    for (;;) { } /* boucle infinie */
    return 0;
}
```

Résultat d'exécution:

%. /test

Le programme boucle.

- Si on appuie sur CTRL-C, le programme affiche

Numéro du signal est 2

Il faut appuyer sur Ctrl-C une 2ème fois pour terminer

- Si on appuie maintenant sur CTRL-C, le programme affiche

Numéro du signal est 2

Il faut appuyer sur Ctrl-C une 2ème fois pour terminer

- Le programme ne s'arrête jamais avec Ctrl-C

El Mostafa DAOUDI- p.137

Exemple 2 suite: On modifie le programme précédent pour que le programme s'arrête lorsqu'on appuie une deuxième fois sur «Ctrl-C» (signal «SIGINT»). Pour cela il faut ré-installer le comportement par défaut du signal « SIGINT ».

Programme 2:

```
#include <stdio.h>
#include <signal.h>
void hand(int signum) {
    printf(" Dan hand : Numéro du signal est %d \n", signum);
    printf("Il faut appuyer sur Ctrl-C une 2ème fois pour terminer\n");
    /* rétablir le handler par défaut du signal « SIGINT » en utilisant la
    macro « SIG_DFL » */
    signal(SIGINT, SIG_DFL);
}
int main(void) {
    signal(SIGINT, hand); /* installation du nouveau handler */
    for (;;) { } /* boucle infinie */
    return 0;
}
```

El Mostafa DAOUDI- p.138

Résultat d'exécution:

%. /test

Le programme boucle.

- Si on appuie sur CTRL-C, le programme affiche

Dan hand : Numéro du signal est 2

Il faut appuyer sur Ctrl-C une 2ème fois pour terminer

- Si on appuie maintenant sur CTRL-C, le programme s'arrête.

Exemple3: On reprend l'Exemple2 du paragraphe 7.1. et on modifie le handler par défaut du signal « SIGUSR1 » pour que le fils l'ignore.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void main(void) {
    pid_t p;
    if ((p=fork()) == 0) { /* processus fils qui boucle */
        signal(SIGUSR1, SIG_IGN); /* Installe le handler « SIG_IGN »
                                   pour le signal « SIGUSR1 » */
        while (1);
    }
    else { /* processus père */
        sleep(10);
        printf("Envoi de SIGUSR1 au fils %d\n", p);
        kill(p, SIGUSR1);
        printf("Fin du processus père %d\n", getpid());
    }
}
```

Résultat d'exécution

- Après quelques secondes, le programme affiche le message prévu dans «printf()» et se termine.
- Le processus fils ne se termine pas (boucle infini) car le signal « SIGUSR1 » a été ignoré. Puisque le processus père est terminé alors le PPID du processus fils devient le processus « init » (PID=1).

Exemple 4 (déroutement): Le système détecte une erreur de calcul

```
#include <signal.h>
#include <stdio.h>
main() {
    int a, b, resultat;
    printf("Taper a : "); scanf("%d", &a);
    printf("Taper b : "); scanf("%d", &b);
    resultat = a/b;
    printf("La division de a par b = %d\n", resultat);
}
```

Résultat d'exécution

%. /test

Taper a: 12

Taper b: 0

Exception en point flottant

et le programme s'arrête.

Exemple 5 (déroutement): Mise en place d'un handler pour le signal « SIGFPE ».

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
void hand_sigfpe() {
    printf("\n Erreur: division par 0 !\n");
    exit(1);
}
main() {
    int a, b, resultat;
    signal(SIGFPE, hand_sigfpe);
    printf("Taper a : "); scanf("%d", &a);
    printf("Taper b : "); scanf("%d", &b);
    resultat = a/b;
    printf("La division de a par b = %d\n", resultat);
}
```

Résultat d'exécution

```
% ./test
Taper a: 12
Taper b: 0
Erreur: division par 0!
```

7.3. Attente d'un signal

La primitive « pause() » bloque (endorme: attente passive) le processus appelant jusqu'à l'arrivée d'un signal puis action en fonction du comportement associé au signal

7.3. Attente d'un signal

La primitive « pause() » bloque (endorme: attente passive) le processus appelant jusqu'à l'arrivée d'un signal quelconque puis action en fonction du comportement associé au signal. Elle est déclarée dans <unistd.h>.

Syntaxe:

```
#include<unistd.h>
int pause (void);
```

A la prise en compte d'un signal, le processus peut :

- se terminer si le « handler » associé est « SIG_DFL » ;
- exécuter le « handler » correspondant au signal intercepté.

Remarque: « pause() » ne permet pas d'attendre un signal de type donné ni de connaître le nom du signal qui l'a réveillé.

Exemple d'attente

```
int nb_req=0;
int pid_fils, pid_pere;
void Hand_Pere( int sig ){
    nb_req++;
    printf("\t le pere traite la requête numero %d du fils \n", nb_req);
}
void main() {
    if ((pid_fils=fork()) == 0) { /* FILS */
        pid_pere = getppid();
        sleep(2); /* laisser le temps au père de se mettre en pause */
        for (i=0; i<10; i++) {
            printf("le fils envoie un signal au pere\n");
            kill (pid_pere, SIGUSR1); /* demande de service */
            sleep(2); /* laisser le temps au père de se mettre en pause */
        }
    }
    else { /* PERE */
        signal(SIGUSR1, Hand_Pere);
        while(1) {
            pause(); /* attend une demande de service */
            sleep(5); /* réalise le service */
        }
    }
}
```

8. Opérations sur les ensembles de signaux

8.1. Libellés des signaux

Il existe des outils pour afficher les libellés des signaux

La table « sys_siglist [] »

C'est une table globale de chaînes de caractères contenant les libellés des signaux :

char * **sys_siglist** [numero_signal].

L'exemple suivant permet d'afficher ces libellés:

```
#include <stdio.h>
#include <signal.h>
main () {
    int i;
    printf("Liste des libellés : \n");
    for (i=1; i < NSIG; i++)
        printf ("signal  %d :  %s\n ", i, sys_siglist[i]);
}
```

El Mostafa DAOUDI- p.147

Résultat d'exécution

Liste des libellés
signal 1 : Hangup
signal 2 : Interrupt
signal 3 : Quit
signal 4 : Illegal instruction
signal 5 : Trace/breakpoint trap
signal 6 : Aborted
signal 7 : Bus error
signal 8 : Floating point exception
signal 9 : Killed
signal 10 : User defined signal 1
signal 11 : Segmentation fault
signal 12 : User defined signal 2
signal 13 : Broken pipe
signal 14 : Alarm clock
signal 15 : Terminated
signal 16 : Stack fault
signal 17 : Child exited
signal 18 : Continued
....

El Mostafa DAOUDI- p.148

La fonction « strsignal() »

Elle permet d'afficher le libellé d'un signal. Il faut inclure le fichier d'entête « #include <string.h> »

L'exemple suivant permet d'afficher les libellés de tous les signaux.

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

main () {
    int i,sig;

    for (sig=1; sig<NSIG; sig++)
        printf("%s \n",strsignal(sig));
}
```

8.2. Configuration des ensembles de signaux

- Type ensemble de signaux:
Le type « sigset_t » désigne un ensemble de signaux.
Il est défini dans <signal.h>
- Fonctions de manipulation des ensembles de signaux:
On peut manipuler les ensemble de signaux à l'aide des fonctions suivantes:
 - int sigemptyset (sigset_t * ens);
Initialise l'ensemble des signaux «ens» à l'ensemble vide.
 - int sigfillset (sigset_t *ens);
Initialise (remplit) l'ensemble «ens» avec tous les signaux.

- `int sigaddset (sigset_t * ens, int sig);`
Ajoute le signal « sig » à l'ensemble « ens ».
- `int sigdelset (sigset_t * ens, int sig);`
Supprime (retire) le signal « sig » de l'ensemble « ens ».
- `int sigismember (const sigset_t * ens, int sig);`
Teste l'appartenance du signal «sig» à l'ensemble «ens». Elle retourne 1 (vrai) si le signal «sig» appartient à l'ensemble «ens» sinon elle retourne 0 (false). Elle peut également renvoyer -1 en cas d'erreur.

Exemple:

L'exemple suivant permet de créer un ensemble qui contient tous les signaux ensuite les afficher:

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
sigset_t ens;

void main () {
    int i;
    sigfillset (&ens); /* remplit l'ensemble « ens » par tous les
                        signaux */

    for(sig=1; sig<NSIG; sig++) /* afficher le numéro du
                                signal avec son libellé */
        if(sigismember(&ens, sig))
            printf("%d : %s : \n",sig, strsignal(sig));
}
```

9. Masquage (blocage) des signaux

- Un processus peut masquer (ou bloquer) les signaux qu'il souhaite. Le masque d'un processus est l'ensemble des signaux bloqués:
 - Il peut être hérité du processus père.
 - Le traitement (La délivrance ou la prise en compte) des signaux bloqués est différée (retardée) jusqu'à ce qu'ils soient supprimés du masque (débloqués).
- Pour installer un masque : 2 étapes.
 1. Construction de l'ensemble des signaux.
 2. Installation du masque par un appel adapté:
 - sigprocmask()
 - sigsuspend()
 - sigaction() (installe un handler et un masque)

El Mostafa DAOUDI- p.153

9.1. La primitive « sigprocmask() »

La primitive « sigprocmask() » permet de masquer ou de démasquer un ensemble de signaux. Elle retourne 0 en cas de succès et -1 en cas d'erreur.

Syntaxe:

```
#include <signal.h>
int sigprocmask(int op, const sigset_t *ens, sigset_t *ens_ancien);
```

- « ens » pointeur sur le nouveau ensemble des signaux à masquer.
- « ens_ancien »: pointeur sur le masque antérieur. Il est récupéré au retour de la fonction.
- « op » précise ce que on fait avec ces ensembles. Les valeurs de «op» sont des constantes symboliques définies dans « signal.h ».

Si le troisième argument « ens_ancien » est un pointeur non NULL, alors la fonction installe un masque à partir des ensembles pointé par « ens » et « ens_ancien ».

El Mostafa DAOUDI- p.154

Valeur du paramètre « op » :

Le paramètre « op » détermine le nouvel ensemble

- Si « op=SIG_SETMASK »

Nouveau masque = « ens ».

Les seuls signaux masqués (bloqués) seront ceux de l'ensemble «ens».

- Si « op=SIG_BLOCK »:

Nouveau masque= « ens » U « ens_ancien »

Les signaux masqués (bloqués) seront ceux des ensembles « ens » et «ens_ancien

- Si « op=SIG_UNBLOCK »

Nouveau masque = « ens_ancien » – « ens »

Démasquage des signaux de l'ensemble « ens ».

Exemple:

L'exemple suivant permet de bloquer les signaux SIGINT et SIGQUIT pendant 10 secondes avant d'être pris en compte.

```
#include <stdio.h>
#include <signal.h>
sigset_t ens;
int sig;

main() {
    //construction de l'ensemble ens={SIGINT, SIGQUIT}
    sigemptyset(&ens);
    sigaddset(&ens,SIGINT); // ajoute le signal SIGINT à ens
    sigaddset(&ens,SIGQUIT); // ajoute le signal SIGQUIT à ens
```

```

/* Installation du masque ens. Si on appui sur ^C ou ^Q :
   leur traitement sera retardé. */

sigprocmask(SIG_SETMASK, &ens, NULL);
printf("Masquage mis en place.\n");
sleep(10);

/* Après 10 secondes. Les signaux masqués seront traités */
sigemptyset(&ens);
printf("Deblocage des signaux masques\n");
sigprocmask(SIG_SETMASK, &ens, NULL);
/* le nouveau masque est vide */
while(1);
}

```

El Mostafa DAOUDI- p.157

9.2 La primitive « sigsuspend() »

Elle permet de manière atomique :

- de faire passer le processus appelant en attente (en sommeil), c'est bloquer le processus appelant.
- de masquer (bloquer) les signaux spécifiés (installe le masque spécifié)

Elle est équivalente à:

```

/* début opération atomique
   sigprocmask (SIG_SETMASK, ens, 0);
   pause(0);
/* fin d'opération atomique

```

El Mostafa DAOUDI- p.158

Syntaxe:

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *ens) ;
```

- « ens » pointeur sur l'ensemble de signaux à masquer

➤ Si un signal non masqué arrive alors:

- traitement attendu
- restauration du masque avant (restitution du masque original)

➤ Sinon (si un signal masqué arrive) alors:

- ignore le signal émis.

9.3. Structure « sigaction »

La primitive « sigaction() » peut être utilisée pour installer un handler personnalisé pour le traitement d'un signal. Elle prend comme arguments des pointeurs vers des structures « sigaction » définies comme suit:

Syntaxe:

```
struct sigaction {  
    void (*sa_handler)() ;  
    sigset_t sa_mask ;  
    int sa_flags ;  
}
```

- Le champ « sa_handler » pointeur vers une fonction (un handler) qui sera chargé de gérer le signal. La fonction peut être:

- « SIG_DFL » : correspond à l'action par défaut pour le signal.
- « SIG_IGN » : indique que le signal doit être ignoré.
- une fonction de type « void » qui sera exécutée par le processus à la délivrance (la réception) du signal. Elle reçoit le numéro du signal.

- Le champ « sa_mask » correspond à l'ensemble de signaux supplémentaires, à masquer (bloquer) pendant l'exécution du handler, en plus des signaux déjà masqués.
- Le champ « sa_flags »: indique des options liées à la gestion du signal

Remarque:

Le champ «sa_handler » est obligatoire.

9.4. La primitive « sigaction() »

La primitive « sigaction() » est légèrement plus complexe que la primitive « signal() ». En plus d'installer un handler personnalisé pour le traitement d'un signal, elle permet aussi d'installer un masque.

Syntaxe:

```
#include <signal.h>
int sigaction(int sig, struct sigaction *action,
              struct sigaction *action_ancien );
```

- « sig »: désigne le numéro du signal pour lequel le nouveau handler est installé. Si « sig » est inférieur ou égal à 0, supérieur ou égal à NSIG, ou égal à SIGKILL ou SIGSTOP, « sigaction() » échoue.

- « action »: désigne un pointeur qui pointe sur la structure « sigaction » à utiliser. La prise en compte du signal entraîne l'exécution du handler spécifié dans le champ « sa_handler » de la structure « action ».

Si la fonction n'est ni « SIG_DFL » ni « SIG_IGN », le signal « sig » ainsi que ceux contenus dans la liste « sa_mask » de la structure « action » seront masqués pendant le traitement selon la fonction.

- « action_ancien » contient l'ancien comportement du signal, on peut le positionner NULL.

Remarque:

Lorsqu'un handler est installé, il restera valide jusqu'à l'installation d'un nouveau handler.

Exemple 1: Le programme suivant (« test_sigaction.c ») masque les signaux « SIGINT » et « SIGQUIT » pendant 30 secondes et ensuite rétablit les traitements par défaut.

```
#include <stdio.h>
#include <signal.h>
struct sigaction action;
int main(void) {
    action.sa_handler=SIG_IGN
    sigaction(SIGINT,&action, NULL);
    sigaction(SIGQUIT,&action, NULL);
    printf("les signaux SIGINT et SIGQUIT sont ignorés \n");
    sleep(30);
    printf("Rétablissement des signaux \n");
    action.sa_handler=SIG_DFL
    sigaction(SIGINT,&action, NULL);
    sigaction(SIGQUIT,&action, NULL);
    while(1);
}
```

El Mostafa DAOUDI- p.165

Résultat d'exécution

% ./test_sigaction

les signaux SIGINT et SIGQUIT sont ignorés

//On appuie sur CTRL-C out CTRL -Q, il ne se passe rien. Après 30 secondes on a l'affichage suivant:

Rétablissement des signaux

**// Maintenant, si on appuie sur CTRL-C ou CTRL-Q, le
// programme s'arrête.**

El Mostafa DAOUDI- p.166

Exemple 1: Le programme suivant (« test_sigaction.c ») masque le signal « SIGQUIT » pendant l'exécution du handler installé pour le signal « SIGINT » et ensuite rétablit les traitements par défaut.

```
#include <stdio.h>
#include <signal.h>
struct sigaction action;
void hand( int sig ){
    sleep(10);
    printf("Rétablissement des signaux \n");
}
```

El Mostafa DAOUDI- p.167

```
int main(void) {
    sigset_t ens;
    action.sa_handler=hand;
    sigemptyset(&ens);
    sigaddset(&ens,SIGQUIT); // ajoute le signal SIGQUIT à ens
    // Installation du masque ens. Le signal ^Q sera maské pendant
    // l'exécution du handler.

    action.sa_mask=ens;
    sigaction(SIGINT,&action, NULL);

    sleep(5);

    action.sa_handler=SIG_DFL;
    sigaction(SIGINT,&action, NULL);

    while(1);
}
```

El Mostafa DAOUDI- p.168

Résultats d'exécution

Test1:

```
% ./test_sigaction
```

```
// On appuie sur CTRL-Q, le programme s'arrête.
```

Test2:

```
% ./test_sigaction
```

```
// On appuie sur CTRL-C, le handler est exécuté. Sleep(10).
```

```
// Ensuite on appuie sur CTRL-Q, il ne se passe rien.
```

```
// En effet le signal « SIGQUIT » est retardé (bloqué)
```

```
// pendant l'exécution du handler.
```

Rétablissement des signaux

Le programme s'arrête.

El Mostafa DAOUDI- p.169

10. Liste des signaux pendant

Grâce à la fonction « sigpending() » on peut voir la liste des signaux, en attente d'être pris en compte.

Le code retour de la fonction est 0 si elle est bien déroulée ou -1 sinon.

Syntaxe:

```
#include <signal.h>
int sigpending(sigset_t *ens) ;
```

Retourne dans « ens » la liste des signaux bloqués (en attente d'être pris en compte).

Exemple: Le programme suivant (**test_pending.c**) suivant montre le masquage et le démasquage d'un ensemble de signaux et comment connaître les signaux pendants.

El Mostafa DAOUDI- p.170

```

#include <stdio.h>
#include <signal.h>
#include<stdlib.h>
sigset_t ens1, ens2;
int sig;
main() {
    //construction de l'ensemble ens1={SIGINT, SIGQUIT, SIGUSR1}
    sigemptyset(&ens1);
    sigaddset(&ens1,SIGINT); // ajoute le signal SIGINT à ens1
    sigaddset(&ens1,SIGQUIT); // ajoute le signal SIGQUIT à ens1
    sigaddset(&ens1,SIGUSR1); // ajoute le signal SIGUSR1 à ens1
    printf(" Numéros des signaux %d %d %d\n",SIGINT,SIGUSR1,SIGQUIT);
    // Installation du masque ens1
    sigprocmask(SIG_SETMASK, &ens1, NULL);
    printf("Masquage mis en place.\n");

```

El Mostafa DAOUDI- p.171

```

sleep(15);
/* affichage des signaux pendants: signaux envoyés mais non
délivrés car masqués*/
sigpending(&ens2);
printf("Signaux pendants:\n");
for(sig=1; sig<NSIG; sig++)
    if(sigismember(&ens2, sig))
        printf("%d \n",sig);
sleep(15);
/* Suppression du masquage des signaux */
sigemptyset(&ens1);
printf("Déblocage des signaux.\n");
sigprocmask(SIG_SETMASK, &ens1, (sigset_t *)0); NULL
sleep(15);
printf("Fin normale du processus \n");
exit(0);
}

```

El Mostafa DAOUDI- p.172

Résultat d'exécution

1^{er} test: on lance l'exécution et on attend la fin du programme

```
% ./test_pending  
Numéro des signaux  2  10  3  
Masquage mis en place  
Signaux pendant:  
Déblocage des signaux  
Fin normal du processus  
%
```

2^{ème} test: on lance l'exécution et ensuite on tappe Ctrl-C

```
% ./test_pending  
Numéro des signaux  2  10  3  
Masquage mis en place  
^C Signaux pendant:  
2  
Déblocage des signaux  
%
```

3^{ème} test: on lance l'exécution et ensuite on tappe Ctrl-C et Ctrl-Quit

```
%. /test_pending
Numéro des signaux  2  10  3
Masquage mis en place
^C ^\ Signaux pendant:
2
3
Déblocage des signaux
%
```

11. La fonction « raise () »

Syntaxe:

int raise (int sig)

La fonction « int raise (int sig) » envoie le signal de numéro « sig » au processus courant (au processus qui appelle « raise() »).

raise(sig) est équivalent à kill(getpid(),sig).

12. La fonction « alarm() »

L'appel système « alarm() » permet à un processus de gérer des temporisation(timeout) avant la routine concernée. Le processus est averti de la fin d'une temporisation par un signal « SIGALRM ».

Syntaxe:

```
#include<signal.h>  
unsigned int alarm(unsigned int sec)
```

- Pour annuler une temporisation avant qu'elle n'arrive à son terme, on fait « alarm(0) ».
- Le traitement par défaut du signal est la terminaison du processus. Pour modifier le comportement on installe un nouveau handler.

Exemple: le programme « test_alarm.c » suivant lit deux entiers. Ensuite demande de lire le résultat qui la compare avec le produit. Si le temps de réponse est plus grand que 3 secondes le programme se termine avec un message d'alarme, sinon il se termine normalement.

```

#include<stdio.h>
#include<signal.h>
main() {
    int a, b, resultat;
    printf("Entrer deux entiers ");
    scanf("%d%d",&a,&b);
    printf(" Donner le résultat de a*b : ");
    alarm(3);
    scanf("%d",&resultat);
    alarm(0); // annule la temporisation
    if(resultat == (a*b))
        printf(" Résultats juste \n ");
    else
        printf(" Résultat faut \n ");

    printf(" Fin normale du programme ");
}

```

El Mostafa DAOUDI- p.179

Résultat d'exécution 1 (on saisit le résultat de a*b avant le délais)

```

%.test_alarm
Entrer deux entier 12 12
Donner le résultat de a*b : 123
Résultat faut
Fin normale du programme

```

Résultat d'exécution 2 (on met plus de temps pour saisir le résultat)

Si on ne tape rien, au bout de 3 seconde on a le message « Minuterie d'alerte » sera affiché.

```

%.test_alarm
Entrer deux entier 12 12
Donner le résultat de a*b
Minuterie d'alerte
%

```

El Mostafa DAOUDI- p.180

Exemple: on installe un handler

```
#include<stdio.h>
#include<signal.h>
void hand(int signum) {
    printf(" délais dépassé. Num SIGALRM = %d \n" , signum);
    exit(1);
}
main() {
    int a, b, resultat;
    signal(SIGALRM,hand);
    printf("Entrer deux entiers ");
    scanf("%d%d",&a,&b);
    printf(" Donner le résultat de a*b : ");
    alarm(3);
    scanf("%d",&resultat);
    alarm(0); // annule la temporisation
    if(resultat == (a*b))
        printf(" Résultats juste \n ");
    else
        printf(" Résultat faut \n ");
    printf(" Fin normale du programme ");
}
```

Résultat d'exécution 2 (on met plus de temps pour saisir le résultat)

%.test_alarm

Entrer deux entier 12 12

Donner le résultat de a*b

délais dépassé. Num SIGALRM = 14

%

13. Signal SIGHUP

SIGHUP = Signal Hang UP (déconnexion de l'utilisateur): ce signal est envoyé automatiquement au processus si on ferme le terminal auquel il est attaché.

Chapitre IV. Communication classique entre processus

I. Synchronisation

1. Introduction

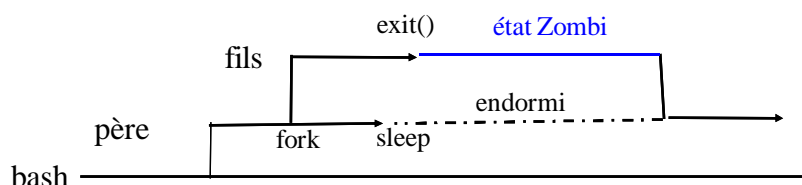
- Après création d'un processus par l'appel système « fork() », le processus créé ainsi que son père s'exécutent de façon concurrente.
- Lorsqu'un processus se termine, il envoie le signal « SIGCHLD » (SIGnal CHiLD) à son père et passe dans l'état zombi tant que son père n'a pas pris connaissance de sa terminaison, c'est à dire que la mémoire est libérée (le fils n'a plus son code ni ses données en mémoire) mais que le processus existe toujours dans la table des processus. Une fois le père lit le code retour du fils, à ce moment, le fils se termine et disparaît complètement du système.

- Dans le cas où le père meurt avant la terminaison de son fils, alors le processus fils sera rattaché au processus « init » (l'ID du processus « init » est 1).
- Afin de synchroniser le père avec ses fils (le processus père attende la terminaison de ses fils avant de se terminer), on utilise les fonctions de synchronisation « wait() » et « waitpid() » qui permettent au processus père de rester bloqué en attente de la réception d'un signal « SIGCHLD ».

2. Les processus zombi

- Quand un processus se termine (soit en sortant du « main() » soit par un appel à « exit() »), il délivre un code retour. Par exemple « exit(1) » donne un code retour égal à 1.
- Tout processus qui se termine passe dans l'état « **zombi** » tant que son père n'a pas pris connaissance de sa terminaison.
- Une fois le père ait connaissance de l'état de son fils, à ce moment le processus fils se termine et disparaît complètement du système

Exemple:



- Le temps pendant lequel le processus fils est en état zombi, le fils n'a plus son code ni ses données en mémoire; seules les informations utiles pour le père sont conservées.

Exemple: Soit le fichier « test_zombi.c »

```
#include <stdio.h>
# include <stdlib.h>
#include <unistd.h>
main() {
    if (fork() == 0) { /* fils */
        printf("je suis le fils, mon PID est %d\n", getpid());
        sleep(1);
    } else { /* père */
        printf("je suis le père, mon PID est %d\n", getpid());
        sleep(30);
    }
}
```

Exemple d'exécution:

```
%./tes-zombi & ps g
```

```
je suis le fils, mon PID est 3748
```

```
je suis le père, mon PID est 3746
```

PID	TTY	STAT	CMD
1716	pts/0	Ss	bash
3746	pts/0	S/test-zombi
3747	pts/0	R+/ps g
3748	pts/0	S/test-zombi

```
% ps g
```

PID	TTY	STAT	CMD
1716	pts/0	Ss	bash
3746	pts/0	S/test-zombi
3748	pts/0	Z	[test_zombi] <defunct>
3749	pts/0	R+	ps g

Dans ce cas le fils se termine et attend que le père soit réveillé car le père est dans un état bloqué (sleep(30) pour avoir connaissance de son état.

Pour que le fils se termine et disparait du système, il faut que le processus père lise l'état (le statut) de son fils.

3. Les processus orphelins

La terminaison d'un processus parent ne termine pas ses processus fils. Dans ce cas les processus fils deviennent orphelins et sont adoptés par le processus initial (PID 1), c'est-à-dire que « init » devient leur père.

Exemple: Soit le fichier « test_orp.c »

```
#include <stdio.h>
# include <stdlib.h>
#include <unistd.h>
main() {
    if (fork() == 0) { /* fils */
        printf("je suis le fils, mon PID est %d\n", getpid());
        sleep(30);
        exit(0);
    } else { /* père */
        printf("je suis le père, mon PID est %d\n",
getpid());
        exit(0);
    }
}
```


Après compilation on lance « test_orp » en background,
ensuite en lance la commande « ps -f »

```
% ./test_orp & ps -f
```

Je suis le fils, mon PID est 3790

Je suis le père, mon PID est 3788

```
[2] 3788
```

UID	PID	PPID	TTY	CMD
etudiant		1969	1750	pts/0 bash
etudiant		3789	1969	pts/0 ps -f
etudiant		3790	1	pts/0/test_orp

```
%
```

4. Synchronisation père et fils: La primitive « wait() »

La fonction wait() est déclarée dans <sys/wait.h>.

Syntaxe:

```
# include<sys/wait.h>
pid_t wait (int * status);
```

L'appel de la fonction « wait() » bloque le processus qui l'appelle en attente de la terminaison de l'un de ses processus fils.

- Si le processus qui appelle « wait() » n'a pas de fils, alors « wait() » retourne -1.
- Sinon « wait() » retourne le PID du processus fils qui vient de se terminer.
- Si le processus appelant admet au moins un fils en attente à l'état zombie, alors « wait() » revient immédiatement et renvoie le PID de l'un de ces fils zombis.

-« status » est un pointeur sur un « int ». Si le pointeur « status » est non NULL, il contient des information sur la terminaison du processus fils (code du signal qui a tué le fils, les circonstances de la terminaison du fils).

Pour la lecture et l'analyse des code retour, on a les macros suivantes:

- « WEXITSTATUS(status) »: fournit le code retour du processus s'il s'est terminé normalement.
- « WIFEXITED(status) »: vrai si le processus fils s'est terminé normalement (en sortant du « main() » ou par un appel à « exit() »).
- « WIFSIGNALED(status) »: vrai si le fils s'est terminé à cause d'un signal.
- « WTERMSIG(status) »: fournit le numéro du signal ayant tué le processus.
- « WIFSTOPPED(status) »: indique que le processus fils est stoppé temporairement.
- « WSTOPSIG(status) »: fournit le numéro du signal ayant stoppé le processus.

Pour attendre la terminaison de tous les fils il faut faire une boucle sur le nombre de fils. Supposons que le processus qui appelle « wait() » a « nb_fils » alors il appelle la fonction « wait() » « nb_fils » fois .

```
for(i=1; i<=nb_fils; i++)  
    wait(0);          /* ou wait(NULL); */
```

ou même

```
while (wait(0)!=-1);    /* while (wait(NULL)!=-1); */
```

5. Synchronisation père et fils: La fonction « waitpid() »

La fonction « waitpid() » permet d'attendre un processus fils particulier ou appartenant à un groupe.

Syntaxe:

- ```
#include<sys/wait.h>
pid_t waitpid (pid_t pid, int * status, int options);
```
- « pid »: désigne le pid du processus fils qu'on attend sa terminaison.
- Si pid >0: le processus père attend le processus fils identifié par «pid».
  - Si pid=0, le processus appelant attend la terminaison de n'importe quel processus fils appartenant au même groupe que le processus appelant.
  - Si pid =-1: le processus père attend la terminaison de n'importe quel processus fils, comme avec la fonction « wait() ».
  - Si pid<-1: le processus père attend la terminaison de n'importe quel processus fils appartenant au groupe de processus dont le numéro est |pid|.

- « status »: a exactement le même rôle que la fonction «wait()».
- « options »: permet de préciser le comportement de «waitpid()», les constantes suivantes :
  - WNOHANG: le processus appelant n'est pas bloqué si le processus spécifié n'est pas terminé. Dans ce cas, waitpid() renverra 0.
  - WUNTRACED: si le processus spécifié est stoppé, on peut accéder aux informations concernant les processus fils temporairement stoppés en utilisant les macros WIFSTOPPED(status) et WSTOPSIG(status) .

L'appel le plus simple est:

```
pid_t waitpid (pid_t pid, int * status, 0);
```

En cas de succès, elle retourne le « pid » du processus fils attendu.

## II Communication par les tubes (pipes)

### 1. Introduction

Un tube de communication est un canal ou tuyau (en anglais pipe) dans lequel un processus peut écrire des données (producteur, écrivain ou rédacteur) et un autre processus peut les lire (consommateur).

- C'est un moyen de communication unidirectionnel inter-processus. C'est le moyen de communication le plus simple entre deux processus.
- Pour avoir une communication bidirectionnelle entre deux processus, il faut créer deux tubes et les employer dans des sens opposés.



Il existe 2 types de tubes:

- Les tubes ordinaires (anonymes): ne permettent que la communication entre processus issus de la même application (entre père et fils, ou entre frères). En effet pour que deux processus puissent communiquer entre eux via un tube, il faut qu'ils disposent tous les deux du tube, donc les deux processus doivent descendre d'un même père (ou ancêtre commun) qui crée le tube.
- Les tubes nommés: permettent la communication entre processus qui sont issus des applications différentes. (communication entre processus créés dans des applications différentes.

**Remarque:** Nous nous limitons au cas où les processus résident dans le même système.

## 2. Caractéristiques

- Un tube possède deux extrémités, une est utilisée pour la lecture et l'autre pour l'écriture.
- La gestion des tubes se fait en mode FIFO (First In First Out: premier entré premier sorti). Les données sont lues dans l'ordre dans lequel elles ont été écrites dans le tube.
- La lecture dans un tube est destructive. C'est-à-dire que les données écrites dans le tube sont destinées à un seul processus et ne sont donc lues qu'une seule fois. Une fois une donnée est lue elle sera supprimée.
- Plusieurs processus peuvent lire dans le même tube mais ils ne liront pas les mêmes données, car la lecture est destructive.
- Un tube a une capacité finie.
- Il y a une synchronisation de type producteur/consommateur: si le tube est vide le lecteur (consommateur) attend qu'on écrit dans le tube avant de lire. Si le tube est plein, un écrivain (producteur) attend qu'il y a de la place pour pouvoir écrire.

### 3. Les Tubes anonymes

#### 3.1. Création d'un tube

Le tube est créé par appel de la primitive « pipe() », déclaré dans « unistd.h ». La création d'un tube correspond à celle de deux descripteurs de fichiers,

- l'un permet de **lire** dans le tube par l'opération classique de lecture dans un fichier (primitive «read()»).
- l'autre permet d'**écrire** dans le tube par l'opération classique d'écriture dans un fichier (primitive «write()»).

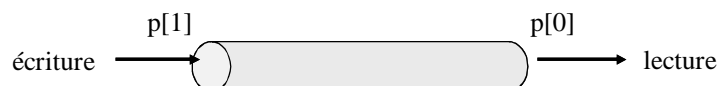
#### Syntaxe:

```
#include <unistd.h>
```

```
int pipe (int p[2]);
```

En cas de succès, le tableau «p» est rempli par les descripteurs des 2 extrémités du tube qui seront utilisés pour accéder (en lecture/écriture) au tube. Par définition:

- Le descripteur d'indice 0 (p[0]) désigne la sortie du tube, il est ouvert en lecture seule.
- Le descripteur d'indice 1 (p[1]) désigne l'entrée du tube, il est ouvert en écriture seule.



### Code retour

La valeur retournée par « pipe() »:

- 0 en cas de succès.
- -1 sinon (en cas d'échec).

### Exemple:

```
main () {
 int p[2];
 pipe(p); /* création du tube p*/

}
```

### 3.2. Fermeture d'un descripteur

Une fois le tube est créé, il est directement utilisable. On n'a pas besoin de l'ouvrir. Par contre, on doit fermer les descripteurs dont n'a pas besoin avec la primitive «close()».

### Syntaxe

close (int fd)

### Exemple:

```
close (p [1]); // fermeture du descripteur en écriture.
close (p [0]); // fermeture du descripteur en lecture.
```

### 3.3. Écriture dans un tube

L'écriture dans un tube se fait grâce à l'appel de la primitive «write()» en utilisant le descripteur p[1].

#### Syntaxe :

```
int write (int p[1], void *zone, int nb_car);
```

- « p[1] » : désigne le descripteur du flux.
- « zone » : désigne un pointeur sur la zone mémoire contenant les données à écrire dans le tube.
- « nb\_car »: désigne le nombre d'octets (caractères) que l'on souhaite écrire dans le tube.

#### Code retour:

- En cas de succès, elle retourne le nombre d'octets effectivement écrits.
- En cas d'erreur, elle retourne -1.

- Un producteur dans le tube est un processus qui détient le descripteur de lecture associé à ce tube.
- Le signal « SIGPIPE » arrête l'exécution du processus.
- Un tube peut avoir plusieurs producteurs. Les paquets des producteurs en accès simultanés peuvent être entrelacés.
- Pour une taille inférieure à PIPE\_BUF, les octets de chaque producteur sont écrits de façon atomique c'est-à-dire de manière consécutive dans le tube avant de passer à une autre écriture: pas d'interférence avec d'autres écritures).
- PIPE\_BUF est une constante définie dans le fichier <limits.h> (/usr/include/linux/limits.h) et vaut 4Ko=4096 octets sous Linux.
- Si le tube est plein l'appelant est bloqué jusqu'à ce qu'il y ait suffisamment de place pour pouvoir écrire dans le tube. La place se libère par la lecture dans le tube



**Exemple :** Processus écrit dans un tube une chaîne de N caractères.

```
#include <stdio.h>
#include <stdlib.h>
#define N 6
main () {
 char *c;
 int p[2];
 int nb_ecrits;
 c=(char *)malloc(N*sizeof(char));
 c="ABCDEF";
 pipe(p); /* création de tube */
 if ((nb_ecrits=write(p[1],c,6))!=-1) {
 printf("Erreur d'écriture dans le tube \n");
 exit(0);
 }
 printf("nb_ecrits = %d \n ", nb_ecrits);
}
```

---

El Mostafa DAOUDI- p.209

### Ecriture dans un tube fermé en lecture

Lorsqu'un processus tente d'écrire dans un tube sans lecteur (descripteur en lecture est fermé), alors il reçoit le signal SIGPIPE et il sera interrompu (si on ne traite pas ce signal).

#### Exemple:

```
main () {
 int p[2];
 pipe (p); // création de tube
 close (p [0]); // descripteur en lecture est fermé
 printf(" Debut d'écriture dans le tube ");
 if (write(p[1],"abc",3)==-1)
 printf(" erreur ecriture dans le tube");
 else
 printf("Pas d erreur");
 printf(" processus termine ");
}
```

**Résultat d'exécution:** Ce programme affiche le message « Debut d'écriture dans le tube » et s'arrête.

---

El Mostafa DAOUDI- p.210

### **3.4. Lecture dans un tube**

La lecture dans un tube s'effectue en appelant la primitive de lecture de fichier (« read() » ). La lecture est faite en utilisant le descripteur p[0].

#### **Syntaxe:**

```
int read (int p[0], void *zone, int nb_car);
```

- « P[0] » : désigne le descripteur du flux
- « zone » : désigne un pointeur sur une zone mémoire dans laquelle elles seront écrites les données après lecture.
- « nb\_car »: désigne le nombre d'octets (caractères) que l'on souhaite lire à partir du tube.

#### **Code retour:**

- En cas de succès, elle retourne le nombre d'octets effectivement lus.
- En cas d'erreur cette fonction retourne -1

### **Remarques:**

Si le tube contient moins de « nb\_car » octets, l'appelant lit les octets disponibles dans le tube.

- Si un lecteur tente de lire dans un tube vide alors:
  - Si le tube n'a pas de rédacteur (le descripteur en écriture est fermé) alors la fonction « read() » retourne 0 caractères lus.
  - Si le tube a un rédacteur, alors il reste bloqué jusqu'à ce que le tube ne soit pas vide.

### Règles à respecter

Le processus ferme systématiquement les descripteurs dont il n'a pas besoin:

- Un processus producteur  
⇒ écrit dans le tube. Il ferme le descripteur p[0]  
(close(p[0])).
- Un processus consommateur  
⇒ lit dans le tube. Il ferme le descripteur p[1]  
(close(p[1])).

**Exemple1 :** Un processus qui lit et écrit dans le même canal de communication :

```
#include <stdio.h>
#include <stdlib.h>
#define N 6
main () {
 char *c, *s;
 int p[2];
 int nb_lus, nb_ecrits;
 c=(char *)malloc(N*sizeof(char));
 s=(char *)malloc(N*sizeof(char));
 c="ABCDEF";
 pipe(p); /* création de tube */
 if ((nb_ecrits=write(p[1],c,N))!=-1) {
 printf("Erreur d'écriture dans le tube \n");
 exit(0);
 }
 printf("nb_ecrits = %d \n ", nb_ecrits);
```

```

if ((nb_lus=read(p[0],s,N))== -1) {
 printf("Erreur de lecture dans le tube \n");
 exit(0);
}
else if (nb_lus==0) {
 printf("pas de caractère lu \n");
 exit(0);
}
printf(" la chaîne lue est : %s \n", s);
}

```

### Lecture dans un tube vide avec rédacteur

**Exemple2:** Lecture dans un tube vide mais qui a un rédacteur.

L'exemple suivant illustre le cas de la lecture dans un tube vide avec producteur.

```

void main () {
 char c;
 int p[2];
 int nb_lus;
 pipe (p); // création de tube
 if ((nb_lus=read(p[0],&c,1))== -1)
 printf("erreur lecture dans le tube");
 else if (nb_lus==0)
 printf("Pas de caractère lu \n");
 printf("processus termine \n");
}

```

**Résultat d'exécution:** Ce programme reste bloqué.

### Lecture dans un tube vide sans rédacteur

**Exemple3:** Lecture dans un tube vide mais qui n'a pas de redacteur.

```
void main () {
 char c;
 int p[2];
 int nb_lus;
 pipe (p); // création de tube
 close(p[1]);
 if ((nb_lus=read(p[0],&c,1))==-1)
 printf("erreur lecture dans le tube");
 else if (nb_lus==0)
 printf("Pas de caractère lu \n");
 printf("processus termine \n");
}
```

**Résultat d'exécution:** Ce programme affiche:

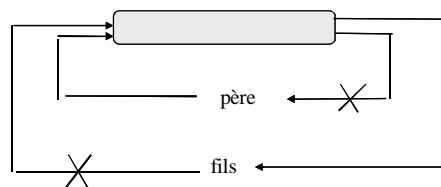
Pas de caractère lu  
processus termine

En effet, puisqu'il n'y a pas de producteur (close(p[1]));, la fonction «read()» retourne 0.

### Communication entre 2 processus:

Les exemples précédents ont traités le cas où c'est le même processus qui lit et écrit dans le tube.

- Utilisation de l'appel système fork()
- Si la communication va du père vers le fils
  - Le père ferme son descripteur de sortie de tube
  - Le fils ferme son descripteur d'entrée de tube



## 4. Les tubes nommés (named pipe)

### 4.1. Introduction

Les tubes nommés permettent la communication, en mode fifo, entre processus quelconques (sans lien de parenté particulier), contrairement aux tubes nommés où il est nécessaire que le processus qui a créé le tube est un ancêtre commun aux processus qui utilisent le tube pour communiquer. Ces tubes sont appelés *fifo*.

Un tube nommé possède un nom dans le système de fichiers. Il dispose d'un nom dans le système de fichier, et apparaisse comme les fichiers permanents. Pour l'utiliser, il suffit qu'un processus l'appelle par son nom.

C'est un fichier d'un type particulier. La commande « ls -l », affiche un tube avec la lettre « p », en position « type de fichier. Exemple si « fiche » est un tube nommé, avec la commande « ls -l » on obtient:

```
pwr-x--x-r- fiche // p pour pipe
```

### 4.2. Caractéristiques des tubes nommés

- Un tube nommé conserve toutes les propriétés d'un tube:
  - Unidirectionnel
  - taille limitée,
  - lecture destructive,
  - lecture/écriture réalisées en mode FIFO.
- Un tube nommé permet à des processus sans lien de parenté dans le système de communiquer.
- Tout processus possédant les autorisations appropriées peut l'ouvrir en lecture ou en écriture.
- S'il n'est pas détruit, **il persiste dans le système après la fin du processus** qui l'a créé.
- Un tube nommé est utilisé comme un fichier normal: on peut utiliser les primitives : « open() », « close() », « read() », « write() ».

### 4.3. Création d'un tube nommés dans un programme

La création d'un tube nommé se fait grâce à l'appel de la primitive « `mkfifo()` » déclarée dans `<sys/stat.h>`.

Code retour: `mkfifo()` retourne

- 0 en cas de succès (tube créé)
- -1 en cas d'échec (par exemple fichier de même nom existe)

#### Syntaxe:

```
#include <sys/stat.h>
int mkfifo (char *ref, mode_t droits)
```

- « `ref` » : désigne le nom du tube à créer (on rajoute généralement l'extension « `.fifo` » au nom du tube nommé).
- « `droits` » : désigne les droits d'accès au tube (fichier créé).

➤ Les droits peuvent être exprimés sous forme numérique dans une représentation octale qui doit être préfixée par un '0' suivi de 3 chiffres qui représentent respectivement les droits pour le propriétaire, le groupe et les autres utilisateurs.

- Le premier chiffre correspond aux droits d'accès pour le propriétaire.
- Le deuxième chiffre correspond aux droits d'accès pour le groupe.
- Le troisième chiffre correspond aux droits d'accès pour les autres utilisateurs.

L'écriture binaire (en base 2) de chaque chiffre définit les droits d'accès pour chaque utilisateur.

| octal | binaire | droits |
|-------|---------|--------|
| 0     | 000     | ---    |
| 1     | 001     | --x    |
| 2     | 010     | -w-    |
| 3     | 011     | -wx    |
| 4     | 100     | r--    |
| 5     | 101     | r-x    |
| 6     | 110     | rw-    |
| 7     | 111     | rwX    |

**Exemple:**

```
mkfifo("test_tube.fifo", 0644);
```

créé le tube nommé « test\_tube.fifo » et attribue les permissions lecture+écriture pour le propriétaire, la lecture pour le groupe et les autres utilisateurs.

➤ Les droits peuvent aussi être exprimés à l'aide des constantes ci-dessous, cumulées, par l'intermédiaire d'un OU binaire (|):

- ❑ S\_IRUSR : Autorisation de lecture pour le propriétaire du fichier.
- ❑ S\_IWUSR : Autorisation d'écriture pour le propriétaire du fichier.
- ❑ S\_IXUSR : Autorisation d'exécution pour le propriétaire du fichier.
- ❑ S\_IRWXU: Lecture + Écriture + Exécution pour le propriétaire du fichier.
- ❑ S\_IRGRP: Autorisation de lecture pour le groupe du fichier.
- ❑ S\_IWGRP: Autorisation d'écriture pour le groupe du fichier.
- ❑ S\_IXGRP: Autorisation d'exécution pour le groupe du fichier.
- ❑ S\_IRWXG: Lecture + Écriture + Exécution pour le groupe du fichier.
- ❑ S\_IROTH : Autorisation de lecture pour les autres utilisateurs.
- ❑ S\_IWOTH : Autorisation d'écriture pour les autres utilisateurs.
- ❑ S\_IXOTH : Autorisation d'exécution pour les autres utilisateurs.
- ❑ S\_IRWXO: Lecture + Écriture + Exécution pour les autres utilisateurs.

**Exemple:**

```
mkfifo("test_tube.fifo", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

créé le tube nommé « test\_tube.fifo » et attribue les permissions lecture+écriture pour le propriétaire, la lecture pour le groupe et les autres utilisateurs.



### Remarque:

Lorsqu'un processus crée un tube nommé, les permissions d'accès sont filtrées par un masque particulier. Pour modifier ce masque on utilise l'appel-système `umask()`, déclaré dans `<sys/stat.h>` :

```
int umask (int masque);
```

**Exemple:** créer un tube nommé avec les permissions « rw-rw-r-- »

```
#include <stdio.h>
#include <sys/stat.h>
main() {
 int masque=umask(0); // modifier le masque par défaut
 mkfifo("tube1.fifo", 0664);
 // ou
 mkfifo("tube1.fifo", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
}
```

### 4.4. Ouverture d'un tube nommés.

Une fois le tube créé, il peut être utilisé pour réaliser la communication entre deux processus. On ouvre l'entrée / la sortie du tube avec la primitive (fonction) « `open()` » définie comme suit:

```
int open(char *ref, int flags, 0)
```

```
int open(char *ref, int options, 0)
```

- « ref » : désigne le nom (préciser le chemin) du tube nommé à ouvrir (on donne le chemin).
- « flags (options) »: mode d'ouverture. Il indique si c'est l'entrée ou la sortie du tube. Il existe des constantes pour cela, déclarées dans « `fcntl.h` » :
  - `O_RDONLY` : pour l'entrée (ouvre uniquement en lecture)
  - `O_WRONLY` : pour la sortie (ouvre uniquement en écriture)
- En cas de succès, cette fonction retourne un descripteur du flux ouvert avec le mode spécifié. Ensuite, on peut écrire ou lire avec les primitives « `write()` » et « `read()` » comme les tubes classiques (anonymes).
- En cas d'erreur, cette fonction retourne -1

- Synchronisation

- ouverture en lecture: le processus est bloqué jusqu'à ce qu'un processus ait ouvert le tube en écriture
- ouverture en écriture: le processus est bloqué jusqu'à ce qu'un processus ait ouvert le tube en lecture

**Exemple:** On suppose que le tube nommé existe au préalable dans le système de fichiers, sinon il faut le créer. Les processus "Producteur" et "consommateur" communiquent via le tube nommé "toto.fifo".

```
/* programme Producteur (fichier product.c) */
#include <stdio.h>
#include <sys/fcntl.h>
main() {
 int fd;
 fd = open("tube.fifo", O_WRONLY);
 write(fd, "bonjour \n", 7);
 printf("%d]> fin ecriture \n", getpid());
}

/* programme Consommateur (fichier consom.c) */
#include <stdio.h>
#include <sys/fcntl.h>
main() {
 int fd; char buf[8];
 fd = open("tube.fifo", O_RDONLY);
 read(fd, buf, sizeof(buf));
 printf("%d]> recu', %s \n", getpid(), buf);
}
```

```
$ /bin/mknod toto.fifo p
$./product &
[1] 2190
$./consom &
[2] 2191
2190] > fin ecriture
2191] > recu: bonjour
```

#### 4.5. Création d'un tube nommés en mode ligne de commande

On peut aussi utiliser, en ligne de commande, les commandes « mkfifo » (/usr/bin/mkfifo) et « mknod » (/bin/mknod) du shell pour créer un tube nommé :

##### Exemple:

```
% mknod tube1 p // crée le tube nommé "tube1"
 // p spécifie "pipe"
% mkfifo tube2 // crée le tube nommé "tube2"
% ls -l
prw-r--r-- 1 tube1
prw-r--r-- 1 tube2
```

#### **4.6.Fermeture et suppression d'un tube nommés.**

- La fermeture se fait grâce à la primitive `close()`  
`int close(int fd)`
  - fd : descripteur du flux
  - Retourne 0 en cas de succès et -1 sinon
- La suppression d'un tube nommé se fait grâce à la primitive `unlink()`.  
`int unlink(const char *pathname)`

### **Ch.V. Compilation et Exécution d'un programme**

#### **I. Introduction au “gcc”: compilateur C**

Le compilateur « gcc » (pour GNU C Compiler) permet de produire un fichier exécutable à partir d'un programme écrit en langage C. Il effectue les tâches suivantes à partir d'un fichier « .c »:

- Le preprocessing (le préprocesseur): interprète quelques commandes élémentaires pour compléter et mettre en forme le fichier source ;
- Le compiling (le compilateur): Il traduit le fichier source en code assembleur, un langage très proche du langage machine.
- L'assembling (l'assembleur): qui traduit le code assembleur en code (langage) machine. On obtient alors le fichier objet.
- Le linking (édition des liens ): qui interface les différents fichiers objets entre eux, et aussi avec les bibliothèques.

Pratiquement, par exemple à partir du fichier « test.c », la compilation consiste à effectuer les étapes suivantes:

- passer le préprocesseur: générer le fichier « test.i ».
- générer un fichier assembleur « test.s ».
- faire l'assemblage de ce fichier pour obtenir « test.o ».
- faire l'édition de liens avec les bibliothèques utiles.

## 2. Produire un exécutable

Pour compiler et produire un fichier exécutable on tape la commande « gcc »

% gcc nom\_fich.c

- nom\_fich.c : le nom du fichier « .c » à compiler

Par défaut, la compilation produit en sortie un fichier exécutable nommé « a.out ».

Pour exécuter le programme:

% ./a.out

« ./ » désigne que la recherche du fichier « a.out » se fait dans le répertoire courant. Si « . » est dans le « path » alors il suffit d'écrire:

% a.out

### Attention:

Si on compile plusieurs programmes dans le même répertoire:  
⇒ « a.out » est l'exécutable du dernier programme compilé.  
⇒ On perd les autres exécutables.

Pour donner un nom explicite au fichier exécutable, on utilise l'option « -o » (out) de « gcc ».

### Syntaxe

```
gcc nom_source.c -o nom_sortie
```

- nom\_source : le nom du fichier source (le fichier à compiler)
- nom\_sortie: le nom du fichier exécutable.

### Exemple:

```
% gcc test.c -o test
```

---

El Mostafa DAOUDI- p.235

## 3. Les fichiers objet

L'option « -c » de la commande « gcc » effectue les trois premières étapes de compilation mais ne fait pas l'édition de lien.

Le résultat: génération de fichiers objets (fichier « .o »). Ils contiennent des informations codées en binaire (format presque exécutable) qui représentent le programme contenu dans le fichier source.

### Exemple:

```
% gcc -c exemple.c
```

```
% gcc -c exemple.c -o exemple.o
```

génère un fichier « exemple.o » qui est un fichier objet.

---

El Mostafa DAOUDI- p.236

## 4. Fichiers en-tête

### Introduction

- Pour que plusieurs fichiers soient utilisables dans un programme principal, il faut donner les prototypes des fonctions définies dans ces fichiers et qui sont utilisées dans le programme.  
⇒ ceci permet de savoir le type de retour, le nombre et les types des arguments des fonctions.
- Les prototypes sont écrits dans des fichiers en-tête qui sont caractérisés par l'extension « .h »

### Inclusion d'un fichier en-tête

Pour que les vérifications des paramètres soient effectuées, les fichiers en-tête (les fichiers « .h ») doivent être inclus dans le programme. L'inclusion est réalisée par la commande

```
#include "nom_fichier"
```

ou

```
#include <nom_fiche >
```

```
#include "..."
```

⇒ la recherche s'effectue dans le répertoire courant

```
#include <...>
```

- ⇒ la recherche s'effectue dans les répertoires d'inclusion « /usr/include »

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

⇒ possibilité de sous-répertoires:

### Exemple

```
#include <stdio.h>
```

```
#include "fichiers.h"
```

```
#include "commande.h"
```

## 5. La compilation séparée

- Un gros programme peut être découpé en plusieurs fichiers indépendants.  
⇒ Séparer le code est un moyen d'avoir des parties du code réutilisables, i.e. des fonctions qui peuvent servir dans d'autres programmes.
- Lorsqu'on compile un programme qui utilise plusieurs fichiers, il faut donner les noms de tous les fichiers à utiliser. Un des fichiers doit contenir une fonction « main() ».

**Exemple:** Si le programme contient plusieurs fichiers sources : prog1.c, prog2.c, ... prog.c où « prog.c » est le programme principal (qui contient la méthode « main() »),

Pour compiler et générer l'exécutable:

```
% gcc prog1.c prog2.c ... prog.c -o prog
```

On peut aussi, tout d'abord générer le fichiers objets ensuite les compiler

```
% gcc -c prog1.c -o prog1.o
```

```
% gcc -c prog2.c -o prog2.o
```

```
% ...
```

```
% gcc -c prog.c -o prog.o
```

```
% gcc prog1.o prog2.o ... prog.o -o prog
```



Au lieu de garder les fichiers sources de ces fonctions qui peuvent servir pour d'autres programmes, on peut les rassembler dans des bibliothèques (librairies).

## **6. Les librairies (ou bibliothèques)**

Les librairies contiennent la définition des fonctions qui peuvent être utilisées par plusieurs programmes. Il existe deux types de librairies:

- les librairies statiques
- les librairies dynamiques.
- Le nom des librairies statiques est de la forme : « lib\*\*\*.a ». Dans le répertoire « /usr/lib », on trouve:
  - « libc.a »: librairie standard C (fonctions « malloc », « exit », etc.) ;
  - « libm.a »: librairie mathématique (fonctions « sqrt », « cos », etc.) ;
  - etc.
- Le nom des librairies dynamiques est de la forme: « lib\*\*\*.so »  
Dans « /usr/lib », on trouve les versions dynamiques des librairies données ci-dessus.

- Utiliser des bibliothèques statiques, revient à inclure la définition des fonctions de la bibliothèque dans le fichier exécutable, pendant l'étape de l'édition de liens (donc pendant la compilation et avant le lancement du programme).
- Utiliser des bibliothèques dynamiques, revient à indiquer au programme l'emplacement d'où il pourra charger en mémoire ces définitions après le lancement du programme.

L'avantage des bibliothèques statiques est que le fichier exécutable qui en résulte contient, avant l'exécution, tout ce qui lui est nécessaire pour fonctionner.

Alors que, si une bibliothèque dynamique a disparu, ou a été modifiée, un programme exécutable qui s'exécutait en utilisant cette bibliothèque peut ne pas fonctionner ou donne des résultats erronés.

Par contre, un programme obtenu par compilation avec une bibliothèque statique a un fichier exécutable beaucoup plus volumineux que le même programme obtenu par compilation avec une bibliothèque dynamique, puisque la définition des fonctions de la bibliothèque ne se trouve pas dans le fichier exécutable.

Enfin, si une bibliothèque statique est mise à jour alors, tout programme l'utilisant devra être recompilé pour qu'il puisse prendre en compte la modification. Dans le cas d'une bibliothèque dynamique, cette mise à jour n'a pas besoin de recompilation.

### Utilisation

Soit la librairie « libXXX.a » (ou « libXXX.so ») se trouvant dans un répertoire dont le chemin absolu est « chemin ».

Pour compiler un fichier source « prog.c » qui fait appel à des fonctions d'une librairie:

```
% gcc prog.c -Lchemin -lXXX -o prog
```

Si le programme contient plusieurs fichiers: prog1.c, prog2.c, ... , prog.c, alors, pour compiler :

```
% gcc prog1.c prog2.c ... prog.c -Lchemin -lXXX -o prog
```

**Remarque:** On peut générer les fichiers objets et ensuite compiler en utilisant les fichiers « .o » (voir avant)

Après l'option « -l », il faut mettre le nom de la librairie sans l'extension (donc sans rajouter « .a » , « .so ») et sans le préfixe « lib ».

Par exemple, pour la librairie mathématiques « libm.a », l'option de compilation est « -lm » ;

```
% gcc prog.c -Lchemin -lm -o prog
```

### Exemple:

Soit le programme suivant qui calcule et affiche la racine carrée du nombre qu'on lui fournit en entrée. On utilise pour cela, la fonction « sqrt() » qui est définie dans le fichier « libm.a », mais aussi dans le fichier « libm.so » qui se trouvent tous deux dans le répertoire « /usr/lib ». Voici le fichier « prog.c » :

```
#include <math.h>
int main() {
 double in;
 scanf("%f",&in);
 printf("%f\n", sqrt(in));
 return 0;
}
```

Pour compiler ce programme, il faut exécuter la commande :

```
% gcc prog.c -L/usr/lib -lm -o prog
```

### Création de bibliothèques

Soient les fichiers « prog1.c », « prog2.c », « prog3.c », ..., « progn.c » contenant des fonctions (autres que « main() »). On peut mettre ces fonctions dans une bibliothèque pour que d'autres programmes puissent les utiliser.

Dans un premier temps, il est nécessaire de compiler ces fichiers pour obtenir des fichiers objet.

```
gcc -c prog1.c -o prog1.o
gcc -c prog2.c -o prog2.o
gcc -c prog3.c -o prog3.o
...
gcc -c progn.c -o progn.o
```

### Création une librairie statique

- Pour créer une librairie statique à partir des fichiers objet, il faut utiliser la commande « ar » qui archive ces fichiers dans un seul fichier.
- L'option « -r » permet d'insérer les nouveaux fichiers dans l'archive.
- L'option « -v » (verbose) permet d'afficher à l'écran le nom des fichiers insérés.
- `ar -rv libtest.a prog1.o prog2.o prog3.o ... progn.o`  
La librairie libtest.a est prête à être utilisée dans une compilation.

### Création une librairie dynamique

Pour créer une librairie dynamique à partir des fichiers objet, on peut utiliser gcc avec l'option -shared.

```
% gcc -o libtest.so -shared prog1.o prog2.o prog3.o .. progn.o
```

La librairie « libtest.so » est prête à être utilisée dans une compilation.

### L'emplacement d'une librairie

En général, on place une librairie (ou un lien vers cette librairie) à un emplacement visible par tous les programmes qui sont susceptibles de l'utiliser.

Typiquement dans :

- « /usr/local/lib » si la librairie est susceptible d'être utilisée par plusieurs utilisateurs ;
- « ~/lib » si la librairie est susceptible d'être utilisée par un seul utilisateur.

## 7. Quelques options de gcc

- o nom : donne le nom du fichier de sortie.
- c : s'arrêter au fichier .o
- lx: ajouter la librairie « x » lors de l'édition de liens. Ceci fait référence au fichier:
  - /usr/lib/libx.a en cas de compilation statique
  - /usr/lib/libx.so en cas de compilation dynamique.
- L chemin\_repertoire\_bib : Recherche des librairies dans le chemin puis dans le chemin «standard» /lib.
- static: Force l'édition de lien à utiliser des librairies statiques (par défaut, les librairies utilisées sont dynamiques)

- E : appelle le préprocesseur. N'effectue pas la compilation.
- S: appelle le préprocesseur et effectue la compilation.  
N'effectue pas l'assemblage ni l'édition de lien. Seuls les fichiers assembleurs (« .s ») sont générés.
- ansi : Demande la compilation du code ANSI.
- w: Supprime tous les warnings.
- W: Active les warnings supplémentaires.
- Wall: Active tous les warnings possibles.
- I chemin\_repertoire\_include: permet d'indiquer qu'il faut chercher les fichiers « .h » dans le répertoire dont le chemin est précisé (en plus des répertoires qui sont parcourus habituellement).

### Remarques:

- gcc prend directement en ligne de commande les fichiers à traiter. L'ordre des fichiers n'a pas d'importance, sauf pour les fichiers de bibliothèques.
- Les bibliothèques doivent être passées à la fin, parce qu'elles ne sont utilisées que si l'éditeur de liens y trouve des symboles non résolus au moment de l'utilisation de la bibliothèque.

## II. Introduction au Makefile

### 1. La commande « make »

La commande « make »:

- située dans le répertoire « /usr/bin »
- utilisée pour exécuter un ensemble d'actions, en particulier pour la compilation l'édition de liens des programmes (pas nécessairement des programmes en langage C).
- très utile lorsque les codes sources sont répartis sur plusieurs fichiers.

Les actions à interpréter par la commande « make » sont regroupées dans un fichier qui contient les cibles, les dépendances, les commandes.

- généralement appelé « makefile » ou « Makefile »  
%make
- Il est possible de spécifier un autre nom de fichier à l'aide de l'option « -f ».

### Exemple

```
% make -f Makefile2
```

### Remarques:

- Si un argument est passé à « make », cet argument est considéré comme la cible principale,
- Sinon la première cible trouvée dans le fichier « makefile » ou « Makefile » est considéré comme cible principale.

Un autre avantage de « make » c'est qu'il ne reconstruit la cible que si elle n'est pas à jour. Par exemple, avant de reconstruire une cible principale, il regardera quelles sont les sous-cibles qui ne se sont pas à jour pour les reconstruire et ensuite reconstruit la cible principale.



### Exemple:

Supposons qu'on a un programme réparti sur deux fichiers plus le programme principal: « fiche1.c », « fiche2.c » et « calc.c » (programme principal).

- Pour la compilation et la génération de l'exécutable on lance la commande suivante:  
`%gcc -Wall fiche1.c fiche2.c calc.c -o calc`
- Si un fichier est modifié cette commande recompile tous les fichiers.

Par contre si on utilise « make », seuls les fichiers qui ont été modifiés qui seront recompilés.

## 2. Présentation de makefile

- Le makefile est un ensemble de règles qui décrivent les relations entre les fichiers sources et les commandes nécessaires à la compilation.
- Il contient aussi des règles permettant d'exécuter certaines actions utiles comme par exemple nettoyer le répertoire.
- Les commentaires, dans un « makefile » s'écrivent sur une ligne et commencent par « # »

Les règles décrites dans un « makefile » sont de la forme :

nom\_cible : liste\_dependances

<TAB> action

<TAB> action

...

**Attention:** Chaque action est précédée d'une tabulation (représentée par <TAB> ).

- La cible (target): désigne, en général, le nom du fichier à générer (à créer). La cible peut être aussi le nom d'une action à exécuter (par exemple nettoyer un répertoire).
- Une dépendance (dependency): désigne un fichier utilisé pour générer la cible correspondante.
- Action: c'est une ligne de commande Unix qui sera exécutée pour générer la cible à partir des dépendances. La commande sera exécutée lorsque au moins une des dépendances de la cible a été modifiée depuis le dernier appel de « make ».

### **Etapes de l'évaluation**

L'appel de « make »

- Sans argument: exécute la première règle rencontrée
- Avec argument: exécute la règle dont le nom est spécifiée en argument .

L'évaluation d'une règle s'effectue en plusieurs étapes :

- Les dépendances sont analysées, si une dépendance dépend d'une autre règle du « makefile », cette règle sera à son tour exécutée.
- Lorsque l'ensemble des dépendances a été analysé, et si les dépendances sont plus récentes que la cible, les commandes correspondant à la règle sont exécutées.
- Si le fichier cible n'existe pas, il considère la cible comme non à jour.
- Si « make » ne trouve pas le fichier correspondant à une dépendance et qu'il n'arrive pas à créer ce fichier, il génère une erreur.

### 3. Exemple de « makefile »

```
test: liste.o main.o
 gcc -o test main.o liste.o
liste.o: list.c
 gcc -c list.c -Wall -O
main.o: main.c exemple.h
 gcc -o main.o -c main.c -Wall -O
```

Pour créer l'exécutable « test », on tape:

```
%make
```

Pour créer « liste.o », on spécifie la cible « liste.o »:

```
%make liste.o
```

### Interprétation

- Par défaut, « make » commence par la première cible, «test» dans notre exemple.
- Avant d'exécuter la commande associée, il doit mettre à jour les fichiers qui apparaissent dans ses dépendances (ici « main.o » et « liste.o »). Puisque « main.o » et « liste.o » sont elles aussi des cibles, alors elles doivent être évaluées. Par conséquent, il évalue successivement
  - La règle associée à « liste.o », à condition que le fichier « liste.c » a été modifié depuis le dernier appel à « make ».
  - La règle associée à « main.o », à condition qu'au moins l'un des fichiers « main.c » ou « exemple.h » a été modifié depuis le dernier appel à « make »
- En fin il exécute les actions associées à la règle « test », à condition qu'au moins « main.o » ou « liste.o » a été modifié (mis à jour).

#### 4. Makefile: complément de règles

- Le Makefile de l'exemple précédent ne permet pas de générer plusieurs exécutables distincts.
- Les fichiers intermédiaires (par exemple les fichiers objets « .o ») restent sur le disque dur.
- Il ne permet pas de forcer la régénération intégrale du projet.

Pour cela, on peut introduire les règles complémentaires suivantes:

- all : en général, c'est la première ligne du fichier « makefile ». Elle permet de regrouper dans ses dépendances l'ensemble des exécutables à produire (séparés par espace).
- clean : elle permet de supprimer tout les fichiers intermédiaires.
- mrproper : elle permet de supprimer tout ce qui peut être régénéré et permet une reconstruction complète du projet.

Exemple: Considérons le « makefile » suivant:

```
all: test
test: liste.o main.o
 gcc -o test main.o liste.o
liste.o: liste.c
 gcc -c liste.c -Wall -O
main.o: main.c exemple.h
 gcc -o main.o -c main.c -Wall -O
clean:
 rm -rf *.o # ou rm test main.o liste.o
mrproper: clean
 rm -rf test
```

- Pour créer l'exécutable:  
    %make
- Pour supprimer les fichiers objets et exécutables du répertoire:  
    %make clean
- Pour régénérer le projet.  
    %make mrproper