



UNIX

Programmation Système

```
int main(void){
    int pid1=0;
    int status;
    if ((pid1=fork()) == -1){
        perror("Creation fils ");
        exit(1);
    }
    if (pid1 == 0) {
        /* Recouvrement du fils */
        execlp("monProg", "monProg", NULL);
        perror("Lancement monProg");
        exit(2);
    }
}
```



PROGRAMMATION SYSTEME UNIX : PLAN



Programmation système

Rappels de C

L'API système UNIX



UNIX- Programmation Système

2

Index Général

Programmation Système	3
Rappels de C	7
L'API système Unix	19
Les accès fichiers	23
Les redirections	37
L'API Posix	43
Threads	44
Mutex	54
Sémaphores	59
Variables conditionnelles	63
Les processus	68
Communication interprocessus	89
Tubes	93
Signaux	100
IPC	118
Communication réseau	141

Programmation système

- Système : Unix, Linux
- Langage de programmation : C
- Compilateur : gcc



Les objectifs de ce cours sont :

- l'approfondissement des concepts théoriques de système d'exploitation par leur mise en pratique,
- la compréhension des concepts de programmation concurrente et de parallélisme, leur problématique et la connaissance des solutions existantes,
- la maîtrise des solutions techniques nécessaires à la mise en œuvre d'applications multitâches sous Linux,

Les prérequis de ce cours sont :

- la connaissance des principes généraux des systèmes d'exploitation,
- la maîtrise des commandes de base Unix/Linux
- des notions de programmation en langage C.

Les séances de travaux pratiques s'effectueront sur des plates-formes Linux en utilisant l'API système C. Le compilateur utilisé est GNU C Compiler (gcc)

QUOI ?

Faire appel dans les programmes utilisateurs aux services spécifiques d'un système d'exploitation

- création et coordination des tâches (programmation multitâche/concurrente)
- gestion spécifique des données
- accès aux périphériques
- communication
- autres ...

COMMENT ?

Chaque système d'exploitation offre une API
(Interface de Programmation)

- bibliothèque de fonctions (appels système)
- spécifique au système d'exploitation
- fonctions appelées dans les programmes
- le processus utilisateur exécute du code système (mode noyau)

LA NORMALISATION

UNIX est une "famille" de systèmes, conformes à des normes

- Single UNIX Specification version 3 (POSIX 2001)
- XPG4
- POSIX 1003.1
- ...



POSIX : Portable Operating System Interface uniX

- Spécification d'un ensemble de fonctions permettant de solliciter les services de base d'un système d'exploitation
- Ne constitue pas la définition d'un système d'exploitation
- Objectif : garantir le développement d'applications portables au niveau du code source, entre les systèmes d'exploitation conformes à la norme en masquant les spécificités du système
- Fournit une liste de points d'accès aux services du système
- Pour chaque fonction, le comportement attendu dans les différentes circonstances susceptibles de se produire est complètement défini

PROGRAMMATION SYSTEME UNIX : PLAN



Programmation système

Rappels de C

- La bibliothèque standard C
- Les arguments de `main()`
- Paramètres d'un exécutable
- Environnement shell d'un processus
- Gestion des erreurs

L'API système UNIX

LA BIBLIOTHÈQUE STANDARD C



- Fonctions déclarées dans les fichiers d'entête :

**assert.h ctype.h errno.h float.h limits.h
locale.h math.h setjmp.h signal.h stdarg.h
stddef.h stdio.h stdlib.h string.h time.h**

- Disponibles quelque soit le système (*Unix, Windows, ...*)
et l'environnement de développement (*gcc, Borland C, Visual Studio, ...*)

LES ARGUMENTS DE main()



- `main()` : fonction à nombre de paramètres variable
- Prototypes de la fonction `main()`

`int main (void);`

`int main (int argc , char **argv);`

`int main (int argc, char **argv, char **envp);`

RAPPELS DE C

PARAMÈTRES D'UN EXÉCUTABLE



Passer des paramètres au lancement d'un exécutable

- Arguments de la fonction main()

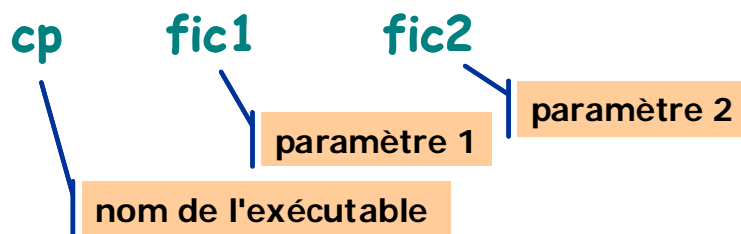
```
int main ( int argc , char **argv )
```

argc : nombre de chaînes de caractères de la ligne de commande

argv : la ligne de commande, terminée par NULL



Exemple



```
argc ---> 3  
argv[0] ---> cp\0  
argv[1] ---> fic1\0  
argv[2] ---> fic2\0  
argv[3] ---> NULL
```

PARAMÈTRES D'UN EXÉCUTABLE



Affichage d'un fichier dont le nom est passé en argument

```
int main ( int argc, char** argv ) {  
    int fd, nb;  
    char buf[512];  
    if ( argc != 2 ) {  
        write( 2, "Nb arguments incorrect\n", 25); exit(1); }  
    if ( ( fd = open( argv[1], O_RDONLY ) ) == -1 ) {  
        perror("Ouverture"); exit(1); }  
    while ( ( nb = read( fd, buf, sizeof(buf) ) ) > 0 ) {  
        write( 1, buf, nb ); }  
    close(fd); return(0);  
}
```

ENVIRONNEMENT SHELL D'UN PROCESSUS



Accès à l'environnement shell d'un processus

➤ Arguments de la fonction `main()`

`int main (int argc , char **argv , char **envp)`

`argc` : nombre de chaînes de caractères de la ligne de commande

`argv` : la ligne de commande, terminée par `NULL`

`envp` : liste des variables d'environnement, terminée par `NULL`

RAPPELS DE C

ENVIRONNEMENT SHELL D'UN PROCESSUS



Exemple : affichage des variables d'environnement

```
int main ( int argc, char** argv, char** envp ) {  
    int i=0;  
    for (i=0; envp[i] != NULL; i++ )  
        printf( "%s\n", envp[i] );  
    printf("\n%d variables d'environnement initialisées\n", i );  
    return(0);  
}
```



Extrait de l'affichage issu de l'exécution du programme ci-dessus

```
LC_TIME=en_US.ISO8859-1  
SUN_SUNRAY_TOKEN=Payflex.500940e200130100  
PATH=/usr/bin:/opt/java/bin:/usr/local/bin:/usr/sfw/  
bin:/opt/sfw/bin:/usr/ccs/bin:/opt/staroffice6.0/pro  
gram:/opt/Acrobat5/bin:/opt/Netscape:/opt/mozilla:/o  
pt/eclipse/bin:/opt/sudo.1.6.7/bin:/usr/local/teTeX/  
bin/sparc-sun-  
solaris2.9:/opt/SUNWspro/bin:/opt/gnat3.15p/bin:/opt  
/fuzz2000:/opt/flex-254/bin:/opt/gawk-  
311/bin:/usr/dt/bin:/usr/openwin/bin:/bin:/usr/bin:/  
usr/ucb  
OSTYPE=solaris  
PWD=/home/people/profs/luter/PROG_SYST  
  
49 variables d'environnement initialisées  
.
```

RAPPELS DE C

ENVIRONNEMENT SHELL D'UN PROCESSUS



➤ Accès au contenu d'une variable d'environnement

```
char *getenv ( const char *name );
```

paramètre : nom de la variable

valeur de retour : contenu de la variable ou NULL (erreur)

➤ Créer ou modifier une variable d'environnement

```
int putenv ( const char *cmde );
```

paramètre : commande d'initialisation

valeur de retour : 0 ou -1 (erreur)



Accès au contenu d'une variable d'environnement : exemple

```
#include <stdlib.h>
int main (void) {
    printf("Le shell courant est %s\n", getenv("SHELL"));
    return 0;
}
```

Création d'une variable d'environnement : exemple

```
#include <stdlib.h>
int main (void) {
    putenv ( "MAVAR = 123");
    printf("MAVAR contient: %s\n", getenv("MAVAR"));
    return 0;
}
```

GESTION DES ERREURS



- Tout accès à une ressource système (fichier, mémoire, ...) est susceptible d'échouer
- Il est **indispensable** de tester le code retour de chaque appel système et d'effectuer le traitement approprié pour éviter un comportement erratique des programmes
- La bibliothèque standard C offre des fonctions qui permettent de récupérer les codes erreur du système



La bibliothèque standard C permet de récupérer la cause des erreurs système, soit par le numéro de l'erreur, soit par le message associé.

Attention ! **La valeur du code erreur après un appel système n'est valable que tant qu'aucun autre appel n'a été effectué.**

Numéro d'erreur

Il est stocké dans la variable globale `errno`, accessible après l'inclusion du fichier d'entête `errno.h`.

Le `man` de chaque fonction précise les mnémoniques correspondant aux numéros d'erreur susceptibles d'être positionnés par cette fonction.

On notera que les fonctions de la bibliothèque standard peuvent aussi positionner `errno`, dans la mesure où elles encapsulent des appels système. Par exemple, la fonction standard `fopen()` positionne `errno` à tous les codes erreurs pouvant être retournés par l'appel système `open()`.

La liste des mnémoniques disponibles et des messages associés est consultable dans le `man` de `errno`

Message d'erreur

Les fonctions `perror()` et `strerror()` permettent d'accéder au message associé au numéro d'erreur.

GESTION DES ERREURS



- Affichage d'un message utilisateur et du dernier message d'erreur système sur la console

```
#include <stdio.h>

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

paramètre : le message utilisateur

- Exemple

```
if ( ( fd = open( "fic", O_RDONLY ) ) == -1 ) {
    perror("ouverture fic");
    exit(1);
}
```



Si le fichier n'existe pas, affichage du message :

```
ouverture fic : no such file or directory
```


GESTION DES ERREURS



- Accès au numéro d'erreur système : variable globale **int errno** déclarée dans **errno.h**

```
if ( ( fd = open( "fic", O_RDONLY ) ) == -1 ) {  
    switch(errno) {  
        case ENOENT: /* fichier inexistant */  
            fd = open( "fic", O_RDWR | O_CREAT, 0640);  
            break;  
        default: /* autre erreur */  
            perror("ouverture fic");  
            exit(1);  
    }  
}
```



Extrait du man de la fonction `open()` :

- « **ENOENT** `O_CREAT` is not set and the named file does not exist. Or, a directory component in *pathname* does not exist or is a dangling symbolic link. »

RAPPELS DE C

GESTION DES ERREURS



➤ Récupération du dernier message d'erreur système

```
#include <string.h>

char * strerror ( int errnum );
```

paramètre : le numéro d'erreur (errno)
valeur de retour : le message d'erreur

➤ Exemple

```
if ( ( fd = open( "fic", O_RDONLY ) ) == -1 ) {
    printf("%s\n", strerror(errno));
    exit(1);
}
```



UNIX- Programmation Système

18

Si le fichier n'existe pas, affichage du message :

no such file or directory

Gestion des codes erreur système et multithreading

Dans un contexte multithread, on utilisera la version réentrante (extension GNU) :

```
char * strerror_r (int errnum, char *buf, size_t n);
```

« The *strerror_r* function works like *strerror* but instead of returning the error message in a statically allocated buffer shared by all threads in the process, it returns a private copy for the thread. »

Source : http://www.gnu.org/software/libc/manual/html_node/Error-Messages.html

errno et **perror()** sont protégés contre les accès concurrents.



L'API système UNIX

- Généralités
- Fichiers
- Threads Posix
 - Threads
 - Outils de synchronisation
- Processus
- Communication interprocessus
 - Tubes
 - Signaux
 - IPC
- Communication réseau

GENERALITES

UN PEU D'AIDE ???

man nom_de_la_fonction

man fork

man -s2 nom_de_la_fonction

man -s2 time --> aide sur la fonction time()

man -s2 intro --> informations générales sur les appels système



Les sections du man Linux

L'option **-s** de la commande **man** permet de préciser le numéro de section dans laquelle rechercher l'aide en cas d'homonymie. Par exemple **open** est à la fois une commande shell (section 1) et un appel système (section 2). Par défaut, la page d'aide appartenant à la section de plus petit numéro sera affichée.

Section 1 : commandes standard Unix et commandes internes au shell

Section 2 : Appels systèmes

Section 3 : Fonctions et bibliothèques des langages de programmation
En particulier, l'aide sur les fonctions C se trouve dans la section 3C.

Section 4 : Périphériques et fichiers spéciaux

Section 5 : Formats de fichiers

Section 6 : Jeux et démos

Section 7 : Divers (standards et normes, environnement, tables de caractères, ...)

Section 8 : Administration système

Section 9 : Drivers

QUELQUES FICHIERS D'ENTETE SPECIFIQUES

➤ **UNIX**

- **unistd.h** gestion des processus, gestion des fichiers
- **fcntl.h** en particulier **open()** et **creat()**
- **signal.h** gestion des signaux ANSI
- **sys/ipc.h** communication interprocessus

➤ **POSIX**

- **pthread.h** threads Posix
- **semaphore.h** sémaphores Posix



Portabilité

On se rappellera que l'usage de fonctions spécifiques à un système d'exploitation fait perdre sa portabilité à un programme source.



LES ACCES FICHIERS

LES FICHIERS

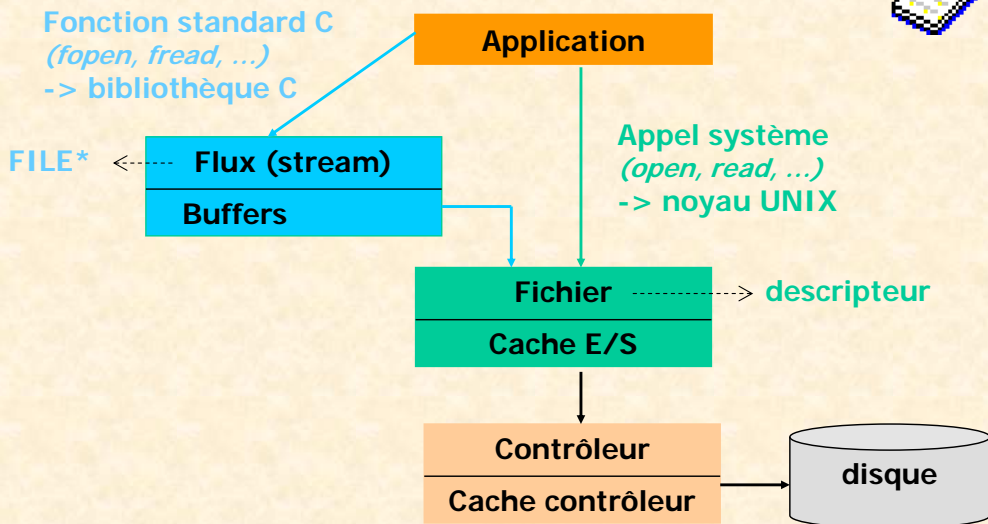
NIVEAUX DE GESTION



- Gestion standard bibliothèque C ("haut niveau")
 - normalisée (C ANSI) + + +
 - bufferisation + + +
 - entrées/sorties formatées + + +
 - gestion par flux (stream) = descripteur de fichier + buffer
- Gestion par appel système Unix ("bas niveau")
 - appel direct aux primitives spécifiques du noyau
 - pas de bufferisation - - -
 - entrées/sorties non formatées - - -
 - accès aux fichiers spéciaux + + +
 - accès à des fonctionnalités spécifiques + + +

LES FICHIERS

NIVEAUX DE GESTION



LES FICHIERS

GESTION UNIX



- Un fichier est manipulé par son **descripteur**
 - nombre entier alloué à l'ouverture du fichier = entrée dans la table des descripteurs
 - une table des descripteurs par processus (héritée) : correspondance avec le fichier physique
 - 3 descripteurs ouverts par défaut au lancement d'un processus interactif :
 - 0** : entrée standard, **1** : sortie standard
 - 2** : sortie d'erreur standard
 - L'entrée dans la table des descripteurs est supprimée à la fermeture du fichier



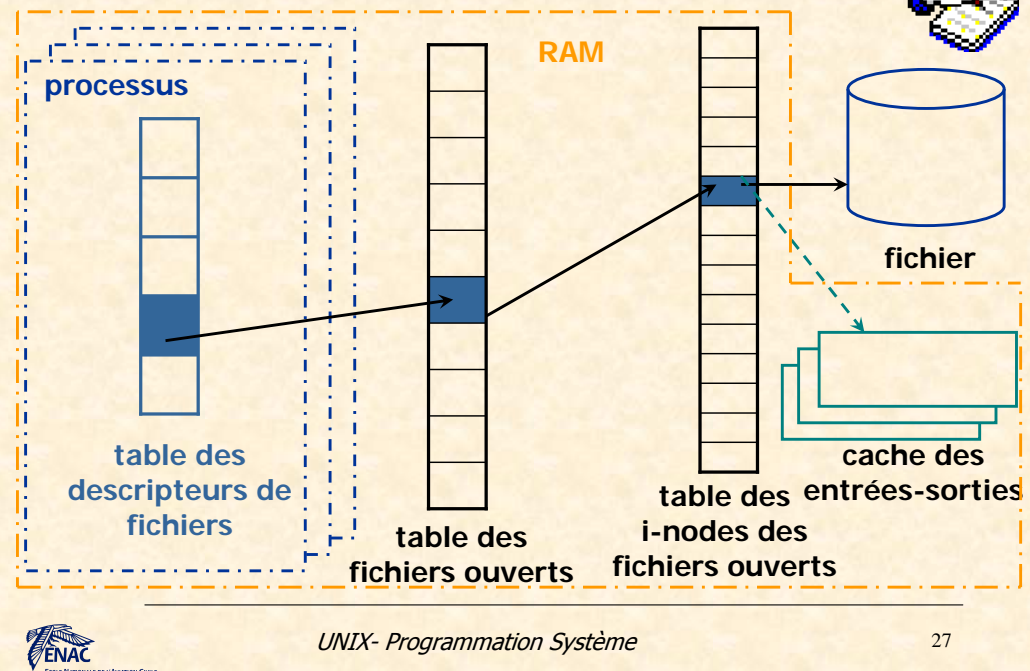
Un processus « hérite » de la table des descripteurs de fichiers de son père (processus qui l'a créé).

Par exemple, un programme lancé à partir de la ligne de commande du shell (commande externe) hérite de la table des descripteurs de fichiers du processus shell.

On notera qu'il s'agit d'une copie de cette table. Elle pourra ensuite être modifiée de manière indépendante par les deux processus. En revanche, l'héritage d'un fichier ouvert implique le partage de la position courante dans le fichier.

LES FICHIERS

GESTION UNIX



Chaque processus possède sa propre **table des descripteurs de fichiers (TDF)**.

Chaque entrée dans cette table contient :

- les attributs du descripteur (actuellement sous Linux seul *close-on-exec* est défini),
- un pointeur sur l'entrée correspondante de la table des fichiers ouverts du système.

Chaque entrée dans la **table des fichiers ouverts (TFO)** contient :

- un compteur de références (nombre de descripteurs qui pointent vers cette entrée),
- le mode d'ouverture,
- la position courante dans le fichier,
- un pointeur sur l'entrée correspondante de la **table des inodes des fichiers ouverts**.

Lors de l'ouverture d'un fichier, les traitements suivants sont effectués :

- Allocation d'un descripteur dans la TDF,
- Allocation d'une entrée dans la TFO : mode fourni, offset en début de fichier (sauf si le mode est O_APPEND),
- Recherche de l'inode dans la table des inodes des fichiers ouverts. S'il n'y est pas, allocation d'une entrée libre de la table et chargement de l'inode.
- Initialisation du compteur de référence à 1,
- Mémorisation de l'adresse de l'entrée de l'inode dans la TFO,
- Retour du numéro du descripteur alloué.

LES FICHIERS

DESCRIPTEUR DE FICHIER



- Concerne tous les types de « fichiers »
 - fichiers ordinaires (programmes, données)
 - fichiers spéciaux
 - tubes de communication (pipelines)
 - sockets (communication réseau)
 - répertoires
 - périphériques



Les fichiers spéciaux

Pour harmoniser les accès à différents mécanismes par les mêmes appels système, la manipulation des pipelines (communication FIFO entre processus via la mémoire), sockets (communication entre processus via le réseau) et périphériques se fait via des descripteurs de fichiers.

On notera qu'il existe 2 types de fichiers spéciaux périphériques : périphériques « caractère » et périphériques « bloc ». On les trouvera généralement dans le répertoire `/dev` et ses sous répertoires.

Dans `/dev`, on trouve aussi les « pseudo périphériques » suivants :

`/dev/zero` utilisé pour générer des zéros binaires

`/dev/random`, `/dev/urandom` utilisés pour générer des nombres aléatoires

`/dev/null` « trou noir » du système, utilisé pour se débarrasser des fichiers et des affichages

`/dev/loopN` utilisés pour simuler un périphérique bloc par un fichier.

LES FICHIERS

DESCRIPTEUR DE FICHIER



- Opérations associées aux descripteurs
 - création, ouverture, fermeture
 - lecture, écriture
 - déplacement dans un fichier
 - duplication (--> redirections)
 - entrées/sorties asynchrones
 - opérations de contrôle sur les périphériques

La pertinence de ces opérations dépend du type de fichier auquel est lié le descripteur. Par exemple, il est impossible de se déplacer dans un pipeline qui fonctionne en FIFO.

Pour les périphériques, la possibilité d'effectuer une de ces opérations est conditionnée par la présence de la fonction correspondante dans le pilote du périphérique (sauf pour les redirections).

LES FICHIERS

DESCRIPTEUR DE FICHIER et FLUX



➤ dans /usr/include/stdio_impl.h:

```
struct __FILE_TAG { /* une définition du type FILE */
    ssize_t _cnt; /* number of available characters in buffer */
    unsigned char *_ptr; /*next character from/to here in buffer */
    unsigned char *_base; /* the buffer */
    unsigned char _flag; /* the state of the stream */
    unsigned char _file; /* UNIX System file descriptor */
    unsigned __orientation:2; /* the orientation of the stream */
    unsigned __ionolock:1; /* turn off implicit locking */
    unsigned __seekable:1; /* is file seekable? */
    unsigned __filler:4;
};
```



UNIX- Programmation Système

30

Un fichier est manipulé en C standard via une structure de type **FILE** dont l'adresse est retournée à l'ouverture par la fonction **fopen()**.

Exemple

```
FILE *f;
f = fopen( "/home/luter/toto", "w");
fputs("une chaine de caracteres", f);
fclose(f);
```

Ce fichier d'entête présent sous Unix illustre le fait que le type standard **FILE** dépend bien du système d'exploitation sous-jacent.

On notera en particulier le descripteur de fichier Unix codé sur un octet.

LES FICHIERS

PRIMITIVES D'ACCES AUX FICHIERS



➤ Ouverture / Création

#include <fcntl.h>

```
int open ( const char *path, int oflags, mode_t mode );
```

path : le chemin d'accès au fichier

oflags : flags d'ouverture (O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC, O_EXCL, O_NONBLOCK)

mode : droits d'accès au fichier (seulement si mode = O_CREAT)

valeur de retour : le descripteur du fichier ou -1 (erreur)



Ouverture d'un fichier en lecture seule

```
int fd;  
fd = open( "fic", O_RDONLY );
```

Création d'un fichier et ouverture en écriture seule

```
int fd;  
fd = open( "fic1", O_WRONLY|O_CREAT|O_TRUNC, 0644 );
```

Ouverture d'un périphérique en lecture seule

```
int fd;  
fd = open("/dev/ttyUSB0", O_RDONLY|O_NOCTTY);
```

LES FICHIERS

PRIMITIVES D'ACCES AUX FICHIERS



➤ Lecture de données

size_t read (int fd, char *buffer, size_t nb);

fd : le descripteur du fichier

buffer : adresse du buffer de réception des données lues

nb : nombre d'octets à lire

valeur de retour : le nombre d'octets lus ou -1 (erreur)

➤ Ecriture de données

size_t write (int fd, char *buffer, size_t nb);

fd : le descripteur du fichier

buffer : adresse du buffer des données à écrire

nb : nombre d'octets à écrire

valeur de retour : le nombre d'octets écrits ou -1 (erreur)



UNIX- Programmation Système

32

Lecture

```
int fd;
int nbLus;
char buffer[512];

fd = open( "fic", O_RDONLY );
...
nbLus = read(fd, buffer, sizeof(buffer));
```

Ecriture

```
int fd;
int nbEcrits;
char buf = "un autre texte" ;

fd = open( "fic1", O_WRONLY|O_CREAT|O_TRUNC, 0644 );
...
nbEcrits = write(fd, "le texte a ecrire", 17);
...
nbEcrits = write(fd, buf, strlen(buf));
```


LES FICHIERS

PRIMITIVES D'ACCES AUX FICHIERS



- Déplacement du pointeur (position courante)

`off_t lseek (int fd, off_t offset, int methode);`

fd : le descripteur du fichier

offset : le déplacement (nombre d'octets)

methode : SEEK_SET, SEEK_CUR, SEEK_END

valeur de retour : l'offset ou -1 (erreur)

- Fermeture de fichier

`int close (int fd);`

fd : le descripteur du fichier

valeur de retour : -1 (erreur)



Remarques concernant la fermeture d'un fichier

Les ressources associées à un descripteur sont libérées à sa fermeture.

Si la suppression du fichier avait précédemment été demandée, elle devient effective à la fermeture du dernier descripteur resté ouvert.

Le moment de la synchronisation des caches dépend du système de fichiers. Il n'est pas garanti qu'elle soit effectuée à la fermeture.

LES FICHIERS

QUELQUES AUTRES FONCTIONS ...



chmod(), fchmod() Modification des droits d'accès

umask() Modification du masque

fcntl() Opérations sur le descripteur de fichier

fsync() Synchronisation des caches

fstat(), stat() Récupération d'informations sur le fichier

...



fcntl()

fcntl() permet d'effectuer des opérations sur un fichier ouvert :
modification des attributs (mode d'ouverture), redirection, pose de verrous
sur des portions de fichiers, gestion des signaux d'entrée-sortie liés à ce
fichier, mise en place de notifications de modification.

fstat(), stat()

stat() et **fstat()** permettent de récupérer des informations sur un
fichier (à partir de son chemin d'accès pour **stat()** et de son descripteur
pour **fstat()**). Ils renseignent une structure **stat** contenant les
informations suivantes (extrait du man) :

```
struct stat {  
    dev_t st_dev; /* ID of device containing file */  
    ino_t st_ino; /* inode number */  
    mode_t st_mode; /* protection */  
    nlink_t st_nlink; /* number of hard links */  
    uid_t st_uid; /* user ID of owner */  
    gid_t st_gid; /* group ID of owner */  
    dev_t st_rdev; /* device ID (if special file) */  
    off_t st_size; /* total size, in bytes */  
    blksize_t st_blksize; /* blocksize for file system I/O */  
    blkcnt_t st_blocks; /* number of 512B blocks allocated */  
    time_t st_atime; /* time of last access */  
    time_t st_mtime; /* time of last modification */  
    time_t st_ctime; /* time of last status change */  
};
```

LES FICHIERS

CONFIGURATION DES PERIPHERIQUES



ioctl() Fonction générique permettant de configurer le comportement des périphériques

mknod() Création d'un fichier spécial périphérique

tcgetattr(), tcsetattr() Configuration des terminaux



Remarques concernant les accès aux périphériques

- Seuls les processus disposant des droits d'accès administrateur (**root**) peuvent directement aux fichiers spéciaux périphériques.
- L'accès (en particulier en écriture) aux périphériques disposant d'un système de fichiers risque de mettre en péril la cohérence de celui-ci. Ces accès sont donc fortement déconseillés !



➡ Exemple 1



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

int main(int argc, char** argv){
    int fd, nb;
    char buf[512];
    if (argc!=2) {
        write(2, "Nb arguments incorrect\n",30);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDONLY)) == -1) {
        perror("Ouverture");
        exit(1);
    }
    while((nb = read(fd, buf, sizeof(buf))) > 0) {
        write(1,buf,nb);
    }
    close(fd)
    return(0);
}
```

LES REDIRECTIONS

LES REDIRECTIONS

MECANISME



- Modification de la table des descripteurs de fichiers
- Changement du fichier physique associé à un descripteur
- Mécanisme :
 - 1- fermeture d'un descripteur --> libère son entrée dans la table des descripteurs
 - 2- duplication du descripteur d'un fichier ouvert : affecte au fichier un descripteur disponible dans la table

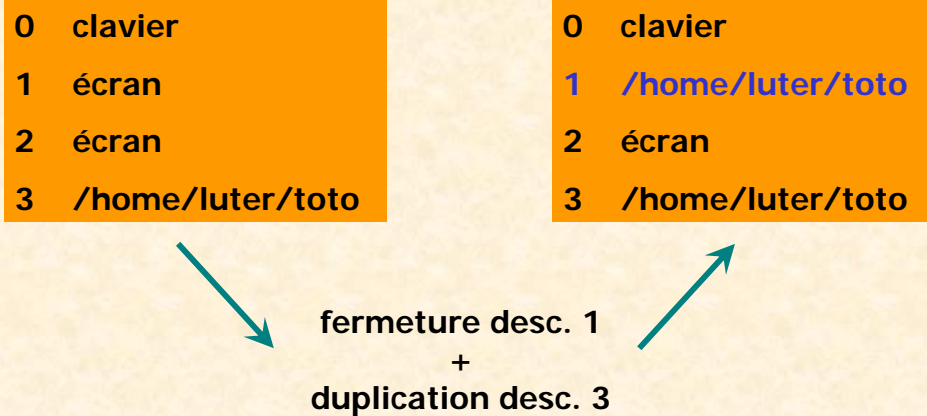
Les deux descripteurs pointent alors sur la même entrée de la table des fichiers ouverts.

LES REDIRECTIONS

EXEMPLE



- Redirection de la sortie standard



Remarque : après cette redirection, l'entrée de la table des fichiers ouverts correspondant au fichier `/home/luter/toto` a son compteur de références égal à 2.

LES REDIRECTIONS

UTILISATION



- Réutilisation de code existant avec de nouvelles entrées/sorties

Exemple : `ls > toto`

- Mécanisme shell du pipeline : inclut deux redirections

Exemple : `ls | more`



`ls > toto`

- le code de `ls` écrit par défaut sur la sortie standard : `write (1, ...)`
- Avant le lancement du code de `ls`, le shell modifie la table des descripteurs de fichier du processus fils créé pour que le descripteur 1 pointe sur le fichier `toto` et non sur le terminal
- le shell lance ensuite l'exécution du code de `ls` dans le processus fils
- `ls` écrit maintenant dans le fichier `toto`
- quand l'exécution de `ls` se termine, la table des descripteurs de fichiers est détruite en même temps que le processus. Il n'y a plus trace de la redirection.

`ls | more`

- Un pipeline est une FIFO en mémoire où les informations écrites par la commande de gauche sont lues par la commande de droite
- Un pipeline apparaît dans la table des descripteurs de fichiers
- La sortie standard du processus exécutant la commande de gauche est redirigée vers l'entrée du pipeline (descripteur en écriture)
- L'entrée standard du processus exécutant la commande de droite est redirigée vers la sortie du pipeline (descripteur en lecture)
- `ls` écrit maintenant dans le pipeline et `more` lit maintenant dans le pipeline

LES REDIRECTIONS

PRIMITIVES



`int dup (int fd);`

fd : le descripteur du fichier à rediriger

valeur de retour : le nouveau descripteur (le plus petit disponible) ou -1 (erreur)

`int dup2 (int fd, int newfd);`

fd : le descripteur du fichier à rediriger

newfd : le nouveau descripteur (attention, s'il est déjà attribué, il est préalablement fermé)

valeur de retour : le nouveau descripteur ou -1 (erreur)

Remarque : **dup2(fd1, fd2);** Non interruptible, est équivalent à
close(fd2);
fd2=dup(fd1);



dup()

```
int fd;
char *buffer = "Le texte a ecrire dans le fichier";

fd = open("myfile.txt", O_CREAT|O_APPEND|O_WRONLY, 0644);
close(1);
dup(fd);
close(fd);
write(1, buffer, sizeof(buffer));
```

dup2()

```
int fd;
char *buffer = "Le texte a ecrire dans le fichier";

fd = open("myfile.txt", O_CREAT|O_APPEND|O_WRONLY, 0644);
dup2(fd, 1);
close(fd);
write(1, buffer, sizeof(buffer));
```

Dans ces deux exemples, la sortie standard (descripteur 1) est redirigée. Le texte sera donc écrit dans le fichier.



Exemple 2



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

int main(void){
    int fd;
    fd = open("monFichier", O_RDWR|O_CREAT|O_TRUNC, 0644);

    close(1);
    dup(fd);
    close(fd);
    // Redirection effectuee :
    // le descripteur numero 1 pointe sur le fichier toto
    execlp("ls", "ls", NULL);
    perror("Lancement ls");
    return(1);
}
```

LES THREADS POSIX

Les threads Posix (ou pthreads) constituent un standard (IEEE POSIX 1003.1c) d'API pour le développement multithread. Leur mise en œuvre est disponible sur plusieurs systèmes d'exploitation.

NOTION DE THREAD

= "fil" d'exécution = "processus léger"

- Unité d'ordonnancement
- même processus = même espace d'adressage
- Les threads possèdent leurs propres :
 - masque des signaux
 - priorité
 - pile
 - registres
- Gestion plus "légère", commutation plus rapide

LES THREADS POSIX

LA BIBLIOTHEQUE PTHREAD

- Implémentation des threads POSIX
- Fonctions **pthread_XXX()** déclarées dans **pthread.h**
- Valeur retournée
0 (OK) ou numéro de l'erreur
- Edition de liens avec gcc
option **-lpthread**
- **#define _REENTRANT** en 1ère ligne du code source



Les pthreads sont des threads créés dans l'espace utilisateur.

Sous Linux, leur implémentation actuelle (depuis le noyau 2.6) est la NTPL (Native Posix Thread Library) où à chaque thread utilisateur est associé un thread noyau (unité d'ordonnancement du noyau Linux).

Cela permet d'éviter les problèmes de blocages qui se produisent quand l'implémentation des threads est uniquement au niveau utilisateur et où l'unité d'ordonnancement est le processus. Dans ce cas, un thread d'un processus peut bloquer tous les autres threads du même processus.

Des informations détaillées sur le fonctionnement des pthreads sous Linux sont disponibles dans la page de man **pthreads** (section 7)

Certaines fonctions de la bibliothèque C manipulent des ressources globales. On vérifiera dans le man qu'elles supportent les accès concurrent (MT safe) ou non. Certaines fonctions non MT safe disposent d'une version réentrante, par exemple :

rand() -> **rand_r()**

strerror() -> **strerror_r()**

LES THREADS POSIX

CARACTERISTIQUES D'UN THREAD

- Identifiant `pthread_t idThread;`
- Code `void * fonction(void * arg) { ... }`
- Attributs
 - portée de l'ordonnancement ("scope")
`PTHREAD_SCOPE_SYSTEM`
`PTHREAD_SCOPE_PROCESS`
 - priorité
 - politique d'ordonnancement
`SCHED_OTHER`
`SCHED_FIFO`
`SCHED_RR`



Portée de l'ordonnancement (scope)

La portée de l'ordonnancement précise si le thread est ordonné par rapport à tous les threads de tous les processus (`PTHREAD_SCOPE_SYSTEM`) ou par rapport aux threads de son processus (`PTHREAD_SCOPE_PROCESS`).

Seul l'attribut `PTHREAD_SCOPE_SYSTEM` est implémenté sous Linux.

Politique d'ordonnancement

`SCHED_OTHER` : politique par défaut. Ordonnement temps partagé à priorité dynamique. L'attribut **priorité** n'est pas pris en compte pour cet ordonnancement (tous les threads ont la même priorité de base).

`SCHED_FIFO`, `SCHED_RR` : politiques temps réel.

Pour plus d'information sur les politiques d'ordonnement Linux, se référer à la page de man de la fonction `sched_setscheduler()`

LES THREADS POSIX

CARACTERISTIQUES D'UN THREAD

➤ Attributs

- attachement

attaché : `PTHREAD_CREATE_JOINABLE`

détaché : `PTHREAD_CREATE_DETACHED`

- pile

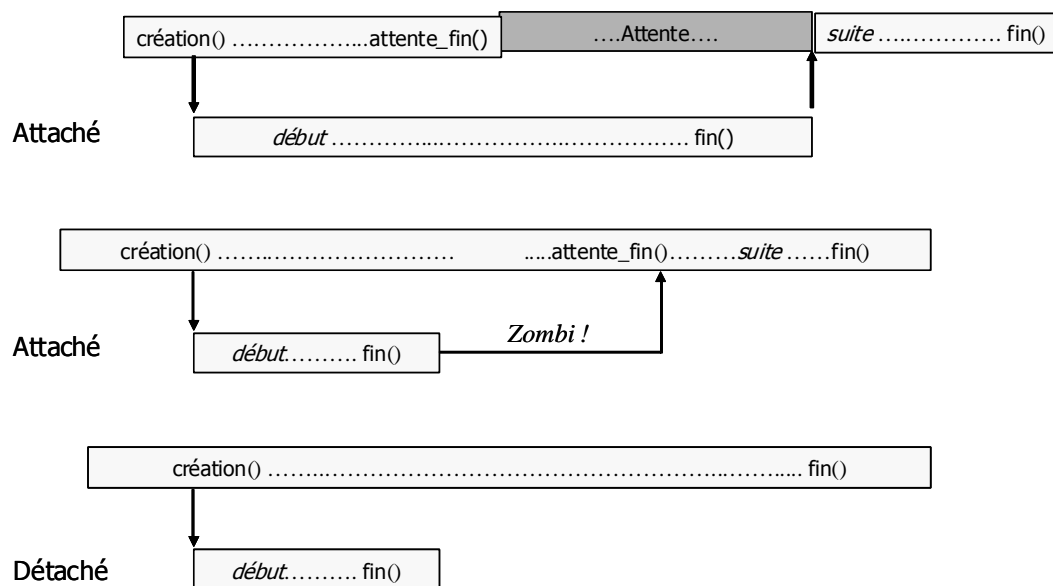
taille

adresse

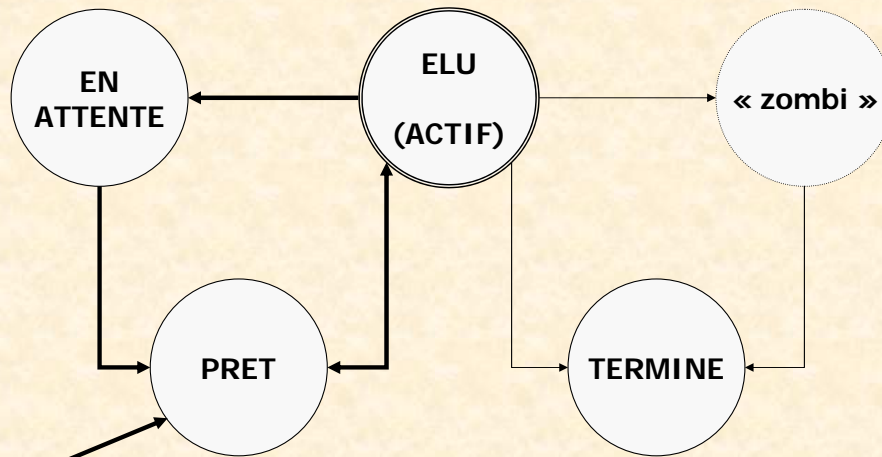


Attachement

Il est possible d'attendre la terminaison d'un thread créé avec l'attribut `PTHREAD_CREATE_JOINABLE` (valeur par défaut) à des fins de synchronisation.



CYCLE DE VIE D'UN THREAD



LES THREADS POSIX

CREATION D'UN THREAD

```
int pthread_create ( pthread_t *tid, pthread_attr_t *attribut,  
                    void * (* nomfonc) (void *arg), void *arg );
```

tid : l'identifiant du thread créé

attribut : les attributs du thread ou NULL (défaut)

nomfonc : le code du thread

arg : la valeur du paramètre de la fonction

valeur de retour : 0 (OK) ou numéro d'erreur

➤ Démarre l'exécution du thread



Exemples

Prototype du code du thread

```
void *foncThread(void *arg);
```

Création sans passage d'argument

```
int main() {  
    int idThread;  
    pthread_create(&idThread, NULL, foncThread, NULL);  
    ...  
    ...  
}
```

Création avec passage d'argument

```
int main() {  
    int idThread;  
    int arg = 2;  
    pthread_create(&idThread, NULL, foncThread, &arg);  
    ...  
    ...  
}
```

LES THREADS POSIX

TERMINAISON D'UN THREAD

`void pthread_exit (void *ret);`

ret : valeur à renvoyer (si le thread est "attaché")

valeur de retour : aucune

- Termine l'exécution du thread appelant
- Si le thread possède l'attribut `PTHREAD_CREATE_JOINABLE` sa valeur de retour peut être récupérée
- **exit()** termine **TOUS** les threads d'un processus



Exemples

Code du thread

```
void *foncThread(void *arg){  
    ...  
    ...  
    pthread_exit(NULL);  
}
```

Création du thread

```
int main() {  
    int idThread;  
    pthread_create(&idThread, NULL, foncThread, NULL);  
    ...  
    ...  
}
```

LES THREADS POSIX

ATTENTE DE LA FIN D'UN THREAD

```
int pthread_join ( pthread_t tid, void **ret );
```

tid : l'identifiant du thread

ret : valeur renvoyée par le thread

valeur de retour : 0 (OK) ou numéro d'erreur

ANNULATION D'UN THREAD

```
int pthread_cancel ( pthread_t tid );
```

tid : l'identifiant du thread

valeur de retour : 0 (OK) ou numéro d'erreur



IDENTIFIANT DU THREAD COURANT

```
pthread_t pthread_self ( void );
```

valeur de retour : l'identifiant du thread courant



L'attente de la fin d'un thread n'est possible que si le thread possède l'attribut **PTHREAD_CREATE_JOINABLE**.

L'annulation d'un thread par un autre thread avec la fonction **pthread_cancel()** doit être utilisée avec précaution pour 2 raisons :

- l'annulation d'un thread n'est possible qu'au moment de l'appel à certaines fonctions dites « cancellation points ». Se référer au man pour la liste de ces fonctions.
- si le thread détient des ressources partagées, celles-ci ne seront pas libérées lors de son annulation, avec pour conséquences possibles des interblocages ou des données incohérentes.

On préférera généralement « demander » au thread de se terminer de lui-même, par exemple en positionnant un flag.

Exemple : création d'un thread et attente de sa terminaison

```
int main() {  
    int idThread;  
    pthread_create(&idThread, NULL, foncThread, NULL);  
    ...  
    pthread_join(idThread, NULL);  
}
```

LES THREADS POSIX

INITIALISATION DES ATTRIBUTS

pthread_attr_init()	attributs par défaut
pthread_attr_setdetachstate()	détaché
pthread_attr_setguardsize()	pile (protection)
pthread_attr_setinheritsched()	héritage de l'ordonnancement
pthread_attr_setschedparam()	priorité
pthread_attr_setschedpolicy()	type d'ordonnancement
pthread_attr_setscope()	portée
pthread_attr_setstackaddr()	pile (adresse)
pthread_attr_setstacksize()	pile (taille)



Remarque

A chaque fonction **pthread_attr_set_XXX()**, correspond une fonction **pthread_attr_get_XXX()** qui permet d'accéder à la valeur courante de l'attribut.

CREATION D'UN THREAD

 **Exemple 21**

EXCLUSION MUTUELLE DES THREADS

- Accès concurrent à une variable globale



Cet exemple illustre la nécessité de synchroniser les accès concurrents à une même ressource (mémoire, fichier, etc ...) pour obtenir un comportement déterministe des programmes.

LES THREADS POSIX

EXCLUSION MUTUELLE DES THREADS : LES MUTEX

- Sémaphore d'exclusion mutuelle
- Utilisé pour résoudre des problèmes d'accès à une ressource critique
- Type de données `pthread_mutex_t mutex;`
- Attributs : il est possible de modifier les attributs d'un mutex, mais ceux-ci ne sont pas standard. On utilisera donc de préférence les attributs par défaut pour assurer la portabilité des applications.



On rappelle qu'un mutex (ou sémaphore d'exclusion mutuelle) est un objet système doté :

- d'un état interne E pouvant prendre les valeurs 0 et 1
- d'une opération de demande de ressource notée P() d'algorithme :
 - tant que E vaut 0
 - attente
 - E = 0
- d'une opération de libération de ressource notée V() d'algorithme :
 - E = 1
 - émission d'un signal de réveil aux tâches en attente sur ce mutex

Il permet d'exclure mutuellement l'exécution de portions de code accédant à une ressource particulière.

EXCLUSION MUTUELLE DES THREADS : LES MUTEX

- Initialisation d'un mutex statique (attributs par défaut)

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- Initialisation d'un mutex dynamique (attributs par défaut)

```
pthread_mutex_t *m;
```

```
m = malloc(sizeof(pthread_mutex_t));
```

```
pthread_mutex_init( &m, NULL );
```

- Destruction d'un mutex dynamique

```
pthread_mutex_destroy( &m );
```

L'état interne du mutex est initialisé à 1 : le mutex est non bloquant à sa création.

EXCLUSION MUTUELLE DES THREADS : LES MUTEX

➤ P(mutex)

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

m : le mutex

valeur de retour : 0 (OK) ou numéro d'erreur

➤ V(mutex)

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

m : le mutex

valeur de retour : 0 (OK) ou numéro d'erreur



Déclaration d'un mutex statique

```
pthread_mutex_t _m = PTHREAD_MUTEX_INITIALIZER;
```

Opération P() : demande de la ressource

```
pthread_mutex_lock(&_m);
```

Opération V() : libération de la ressource

```
pthread_mutex_unlock(&_m);
```

EXCLUSION MUTUELLE DES THREADS : LES MUTEX

- Accès concurrent à une variable globale

Exemple 23



```
unsigned int _cpt; // la ressource a proteger
pthread_mutex_t _m = PTHREAD_MUTEX_INITIALIZER;

void *foncThread1(void *arg){
    unsigned int savcpt;
    while (_cpt<5){
        pthread_mutex_lock(&_m);
        savcpt=_cpt;
        printf("Thread1: _cpt=%u - ", _cpt);
        _cpt++;
        printf(" %u+1=%u\n",savcpt, _cpt);
        pthread_mutex_unlock(&_m);
    }
}

void *foncThread2(void *arg){
    unsigned int savcpt;
    while (_cpt<5){
        pthread_mutex_lock(&_m);
        savcpt=_cpt;
        printf("Thread2: _cpt=%u - ", _cpt);
        _cpt++;
        printf(" %u+1=%u\n",savcpt, _cpt);
        pthread_mutex_unlock(&_m);
    }
}
```

LES THREADS POSIX

SYNCHRONISATION : SEMAPHORES POSIX

- Comptage de ressources
- Synchronisation interne ou interprocessus
- norme Posix temps réel (Posix.1b)
- **errno** positionnée en cas d'erreur
- **#include <semaphore.h>**
- Synchronisation interne :
type de données **sem_t s;**



On rappelle qu'un sémaphore est un objet système doté :

- d'un compteur interne C pouvant prendre des valeurs ≥ 0
- d'une opération de demande de ressource notée P() d'algorithme :
tant que C vaut 0
attente
 $C = C - 1$
- d'une opération de libération de ressource notée V() d'algorithme :
 $C = C + 1$
émission d'un signal de réveil aux tâches en attente sur ce sémaphore

La norme Posix prévoit des **sémaphores nommés** (interprocessus) pour la synchronisation de threads appartenant à des processus différents et des **sémaphores sans nom** (internes) pour la synchronisation des threads d'un même processus. Les sémaphores nommés ne sont disponibles sous Linux que depuis le noyau 2.6 avec l'implémentation NTPL des threads Posix.

Dans le cadre de ce cours, nous n'utiliserons que les sémaphores internes.

Les seules différences en terme de code sont leur création et leur destruction :

- sémaphore interne : création par déclaration d'une variable de type **sem_t**
initialisation par appel à **sem_init()**
destruction par **sem_destroy()** ou à la fin du process
- sémaphore interprocessus : création et initialisation par appel à **sem_open()**
destruction par appel à **sem_unlink()**

SYNCHRONISATION : SEMAPHORES POSIX

➤ Initialisation d'un sémaphore

```
int sem_init ( sem_t *sem, int partage, unsigned int value );
```

sem : le sémaphore

partage : 0 (interne au processus) ou $\neq 0$ (interprocessus)

value : valeur initiale, $value \in [0; SEM_VALUE_MAX]$

valeur de retour : -1 (erreur)

➤ Destruction d'un sémaphore

```
int sem_destroy ( sem_t *sem );
```

sem : le sémaphore

valeur de retour : -1 (erreur)



Déclaration d'un sémaphore interne

```
sem_t _s;
```

Initialisation d'un sémaphore interne à la valeur 2

```
sem_init( &_s, 0, 2 );
```

LES THREADS POSIX

SYNCHRONISATION : SEMAPHORES POSIX

➤ P(sémaphore)

```
int sem_wait ( sem_t *sem);  
    sem : le sémaphore  
    valeur de retour : -1 (erreur)
```

➤ P(sémaphore), non bloquant

```
int sem_trywait ( sem_t *sem );  
    sem : le sémaphore  
    valeur de retour : -1 (erreur)
```

➤ V(sémaphore)

```
int sem_post ( sem_t *sem );  
    sem : le sémaphore  
    valeur de retour : -1 (erreur)
```



Remarque

Il existe aussi une fonction `sem_timedwait()` qui permet d'associer un timer à l'attente sur une opération P(). Dans ce cas, l'opération d'attente sera débloquée si le compteur de ressources du sémaphore devient positif ou si le délai d'attente est échu. La valeur du code retour permettra de distinguer les deux cas.

Opération P() : demande d'une ressource

```
sem_wait (&_s) ;
```

Opération V() : libération d'une ressource

```
sem_post (&_s) ;
```

SYNCHRONISATION : SEMAPHORES POSIX

➤ Cohorte



On rappelle qu'une cohorte modélise l'accès à un groupe de N ressources banalisées.

Dans cet exemple, un parking offre N places. On modélisera la ressource « place » par un sémaphore S.

Au début du problème le parking est vide. Le sémaphore sera donc initialisé à N.

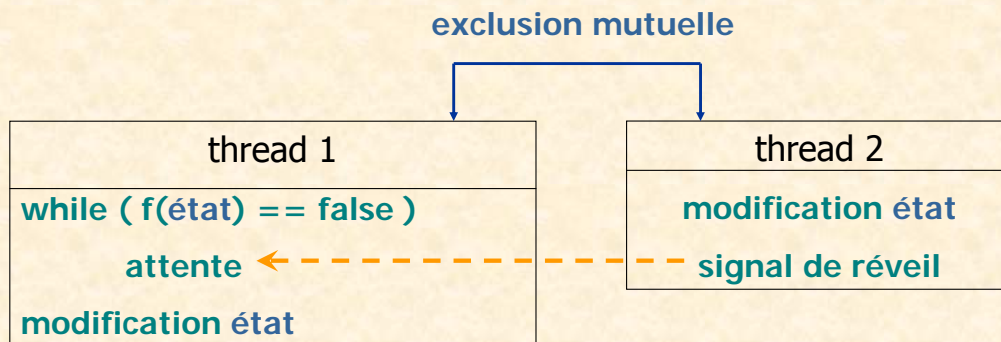
Toute voiture désirant entrer dans le parking doit demander une place. Si aucune place n'est disponible, elle attend. On a ici le schéma d'une opération P(S).

Toute voiture sortant du parking libère une place. On a ici le schéma d'une opération V(S).

LES THREADS POSIX

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Synchronisation de threads par moniteur de Hoare
- Principe



Un **moniteur de Hoare** est une structure algorithmique permettant de regrouper des variables partagées entre plusieurs tâches (variables d'état du moniteur) et les opérations qui les manipulent.

Celles-ci doivent s'exécuter en exclusion mutuelle en évitant l'interblocage : dans le cas où une tâche est en attente d'une ressource, elle doit pouvoir rendre la main aux autres tâches capables de fournir cette ressource.

Il définit un objet nommé variable condition qui est un mécanisme de synchronisation doté d'une opération d'attente, d'une opération de réveil et d'une file d'attente de tâches.

Quand une tâche est en attente, l'exclusion mutuelle est relâchée pour permettre à d'autres tâches de faire évoluer l'état du moniteur.

L'implémentation Posix permettant de réaliser un moniteur de Hoare est la **variable conditionnelle**.

Une variable conditionnelle (correspondant à la variable condition du moniteur) est un objet système comportant une file d'attente de tâches et 2 opérations : attente et signal.

Elle est obligatoirement associée à un mutex.

On notera que, même si l'outil variable conditionnelle n'est pas utilisé pour réaliser un moniteur et ne nécessite donc pas algorithmiquement une exclusion mutuelle, celle-ci reste techniquement obligatoire.

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Création d'une variable conditionnelle

```
pthread_cond_t v = PTHREAD_COND_INITIALIZER;
```

- Attente sur une condition

```
int pthread_cond_wait( pthread_cond_t *v,  
                      pthread_mutex_t *m );
```

v : la variable conditionnelle

m : le mutex associé

valeur de retour : 0 (OK) ou un code erreur



Attente

`pthread_cond_wait()` met en attente la tâche qui l'invoque. Pendant l'attente, le mutex associé est relâché.

Quand la tâche en attente reçoit le signal de réveil, elle reprend automatiquement le mutex et peut donc continuer son exécution en exclusion mutuelle.

```
pthread_cond_t _cond = PTHREAD_COND_INITIALIZER;  
pthread_mutex_t _m = PTHREAD_MUTEX_INITIALIZER;  
int _varEtat;  
...  
pthread_mutex_lock(&_m);  
while (_varEtat != 1)  
    pthread_cond_wait(&_cond, &_m);  
...  
pthread_mutex_unlock(&_m);
```

Remarque : il est techniquement indispensable d'encadrer l'attente par une boucle.

« Spurious wakeups from the `pthread_cond_timedwait()` or `pthread_cond_wait()` functions may occur. Since the return from `pthread_cond_timedwait()` or `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return. »

(extrait du `man pthread_cond_wait`)

LES THREADS POSIX

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Signalement d'une condition (réveil)
 - débloque au moins un des threads en attente

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

v : la variable conditionnelle

valeur de retour : 0 (OK) ou un code erreur

- Signalement d'une condition (réveil)
 - débloque tous les threads en attente

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

v : la variable conditionnelle

valeur de retour : 0 (OK) ou un code erreur



Réveil

pthread_cond_signal() réveille au moins une des tâches en attente sur la variable conditionnelle.

pthread_cond_broadcast réveille toutes les tâches en attente sur la variable conditionnelle.

Si aucune tâche n'est en attente au moment de l'envoi du signal, celui-ci est perdu.

```
pthread_cond_t _cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t _m = PTHREAD_MUTEX_INITIALIZER;
int _varEtat;

...

...

pthread_mutex_lock(&_m);
_varEtat = 1;
pthread_cond_signal(&_cond);
pthread_mutex_unlock(&_m);
```

LES THREADS POSIX

SYNCHRONISATION : VARIABLES CONDITIONNELLES

Thread attendant la condition	Thread signalant la condition
pthread_mutex_lock()	...
while (f(variables état) == 0) {	...
pthread_cond_wait();	pthread_mutex_lock()
<i>Mutex débloqué pendant l'attente</i>	Modification variables état
}	pthread_cond_signal()
Accès variables état	pthread_mutex_unlock()
pthread_mutex_unlock()	...





Exemple 25

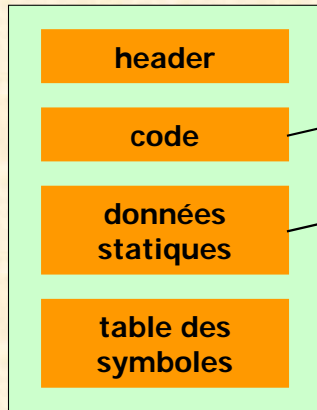
On notera que dans cet exemple, la variable d'état **alerte** ne sert qu'à boucler sur l'attente et n'est pas fonctionnellement nécessaire.

LES PROCESSUS

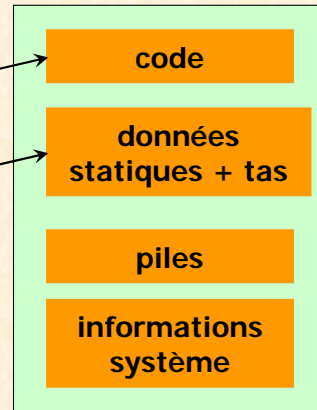
LES PROCESSUS

CREATION D'UN PROCESSUS

Fichier exécutable
(sur disque)



Processus
(en mémoire centrale)



Rappel du cours de Systèmes d'exploitation

Un processus est défini comme « l'image mémoire d'un programme en cours d'exécution »

Son espace d'adressage comprend :

- le code exécutable, partageable avec d'autres processus,*
- la zone de données,*
- les piles utilisateur et système*
- les informations système : environnement, bloc de contrôle ou PCB (Process Control Block)*

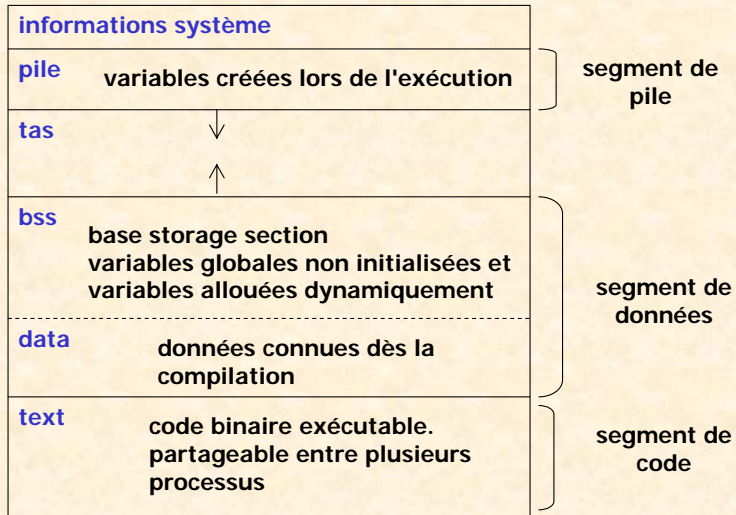
LES PROCESSUS

ESPACE D'ADRESSAGE D'UN PROCESSUS LINUX

taille max. = 3 Go
(noyaux 32 bits)



0



UNIX- Programmation Système

70

Rappel du cours de Systèmes d'exploitation

Sous Linux, la taille maximale de l'espace d'adressage d'un processus dépend de l'architecture du processeur.

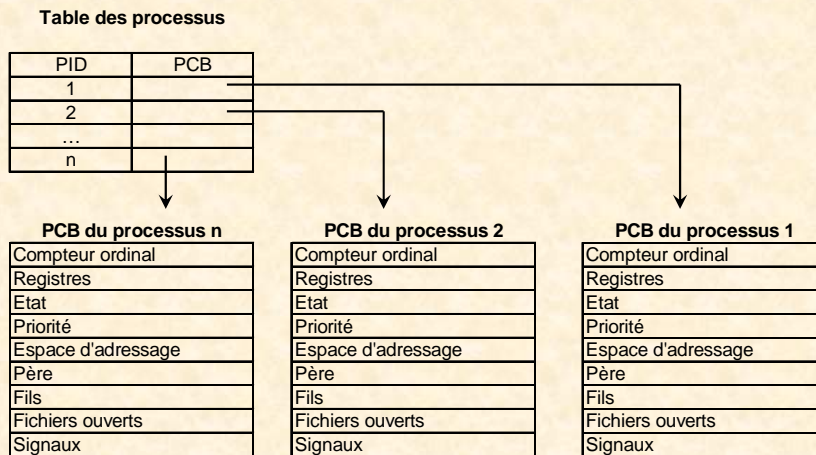
En architecture 32 bits, elle est limitée à 4 Go (dont 3Go utilisateur et 1Go pour les données noyau).

En architecture 64 bits, elle est limitée à 1 To.

LES PROCESSUS

TABLE DES PROCESSUS

➤ Gérée par le système



Chaque PCB (Process Control Block) est décrit sous Linux par une structure `task_struct` (déclarée dans `linux/sched.h`) et est stocké dans l'espace d'adressage du processus. Il contient toutes les informations nécessaires à la gestion du processus (contexte).

La table des processus est donc une structure de données pointant sur les PCB des différents processus.

LES PROCESSUS

CARACTERISTIQUES DES PROCESSUS

- Un numéro d'identification unique : PID (Process Identifier)
- Les ressources allouées (fichiers ouverts, mémoire, ...)
- Les signaux à capturer, à masquer, à ignorer et les actions associées
- Son processus père, ses processus fils,
- Son propriétaire, son groupe,
- Ses variables d'environnement,
- ...



Les caractéristiques présentes dans la structure `task_struct` citées ci-dessus sont celles manipulées par le programmeur.

Le PCB contient en plus toutes les informations nécessaires à la gestion de son cycle de vie (état prêt, actif ou en attente), à la commutation de contexte (état des registres, espace d'adressage) et à l'ordonnancement (priorité, ...).

On notera que le processus n'est pas par définition une unité d'ordonnancement, mais sous Linux, les threads et les processus sont décrits par la même structure de données et ne diffèrent que par la nature de leur espace d'adressage.

CARACTERISTIQUES DES PROCESSUS

➤ Processus père et processus fils

Un processus est toujours créé par un autre processus (à l'exception du processus de démarrage du système)

➤ Lancement des commandes

- commande interne = code interne au shell
---> pas de nouveau processus
- commande externe (exécutable)
---> le shell crée un processus fils
- script
---> selon le mode de lancement

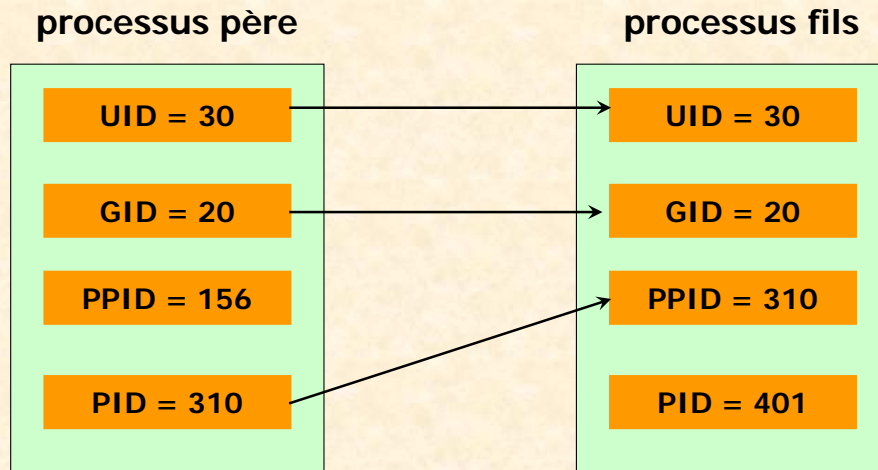
CARACTERISTIQUES DES PROCESSUS

- **PID** (Process IDentifier)
 - numéro unique
 - process **init** : PID 1
- **PPID** (Parent Process IDentifier)
 - numéro du processus père
- **UID** (User IDentifier) = utilisateur réel ---> **droits**
 - identifiant de l'utilisateur ayant lancé le processus
- **GID** (Group IDentifier) = groupe réel ---> **droits**
 - identifiant du groupe

On notera que chaque processus détient la liste de ses processus fils.

LES PROCESSUS

CARACTERISTIQUES DES PROCESSUS : héritage



Les identifiants de l'utilisateur et du groupe réels sont hérités par le processus fils.

DROITS D'ACCÈS AUX RESSOURCES

Pour le propriétaire, le groupe et les autres utilisateurs

- lecture (r)
- écriture (w)
- exécution (x)

Un processus dispose des droits d'accès de son propriétaire effectif

LES PROCESSUS

DROITS D'ACCÈS AUX RESSOURCES

- tout processus lancé par un utilisateur dispose du numéro (RUID) de cet utilisateur (ou utilisateur réel)
- le processus dispose en plus du numéro de l'utilisateur effectif (EUID), celui avec les droits duquel s'exécute le programme.
- En règle générale, RUID = EUID, c'est à dire que le processus se voit attribuer les droits de l'utilisateur qui l'a lancé.
- Les mêmes notions existent au niveau du groupe (RGID, EGID)

DROITS SPÉCIAUX D'ACCÈS AUX RESSOURCES

➤ **bit setuid (s)**

➤ **bit setgid (s)**

➤ **sticky bit**

Appliqués à la place du droit en exécution (x)



bit setuid

- Permet à un processus lancé par n'importe quel utilisateur de s'exécuter avec les droits d'accès du propriétaire du fichier exécutable
- RUID = UID de l'utilisateur, EUID = UID du propriétaire du fichier
- Positionné au niveau des droits du propriétaire à la place du droit en exécution : `chmod u+s fichier`
- Exemple : `passwd` offre les droits de l'utilisateur `root` sur `/etc/passwd`

bit setgid

Permet à un processus lancé par n'importe quel utilisateur de s'exécuter avec les droits d'accès du groupe propriétaire du fichier (RGID != EGID)

Les fichiers créés dans un répertoire `setgid` auront tous le même groupe que le répertoire

Positionné au niveau des droits du groupe à la place du droit en exécution :

`chmod g+s fichier`

sticky bit

Permet de garder le code d'un fichier exécutable en mémoire après la fin de l'exécution (accélération des lancements ultérieurs)

Appliqué à un dossier, permet à un utilisateur d'écrire dans le dossier, mais ne l'autorise à supprimer que les fichiers lui appartenant : `/tmp`

Positionné au niveau des droits des autres utilisateurs à la place du droit en exécution : `chmod o+t fichier`

ACCES AUX CARACTERISTIQUES DES PROCESSUS

➤ Récupération du PID

`pid_t getpid (void);`

valeur de retour : le PID du processus courant

➤ Récupération du PPID

`pid_t getppid (void);`

valeur de retour : le PID du père du processus courant

➤ Récupération du UID

`pid_t getuid (void);`

valeur de retour : le UID du processus courant

➤ Récupération du GID

`pid_t getgid (void);`

valeur de retour : le GID du processus courant



Exemple

```
#include <unistd.h>
#include <stdio.h>
int main(){
    printf("Mon identifiant (PID) est %d\n", getpid());
    printf("Le PID de mon pere(PPID) est %d\n", getppid());
    printf("Mon utilisateur reel (UID) est %d\n", getuid());
    printf("Mon utilisateur effectif est %d\n", geteuid());
    printf("Mon groupe reel (GID) est %d\n", getgid());
    printf("Mon groupe effectif est %d\n", getegid());
    return 0;
}
```

Résultat d'exécution

```
Mon identifiant (PID) est 2306
L'identifiant de mon pere(PPID) est 2078
Mon utilisateur reel (UID) est 501
Mon utilisateur effectif est 501
Mon groupe reel (GID) est 501
Mon groupe effectif est 501
```

Autres fonctions

`geteuid(), getresuid(), getegid(), getresgid()`


LES PROCESSUS

CREATION D'UN PROCESSUS

`pid_t fork (void);`

valeur de retour : le PID du processus fils créé ou -1 (erreur)

Duplique le processus père : le fils en est une copie exacte

- Même code
- Recopie des zones de données
- Recopie de l'environnement :
 - descripteurs de fichiers ouverts
 - répertoire courant
 - priorité
 - masque des signaux
- Partage des pointeurs de fichiers ouverts 



Remarques

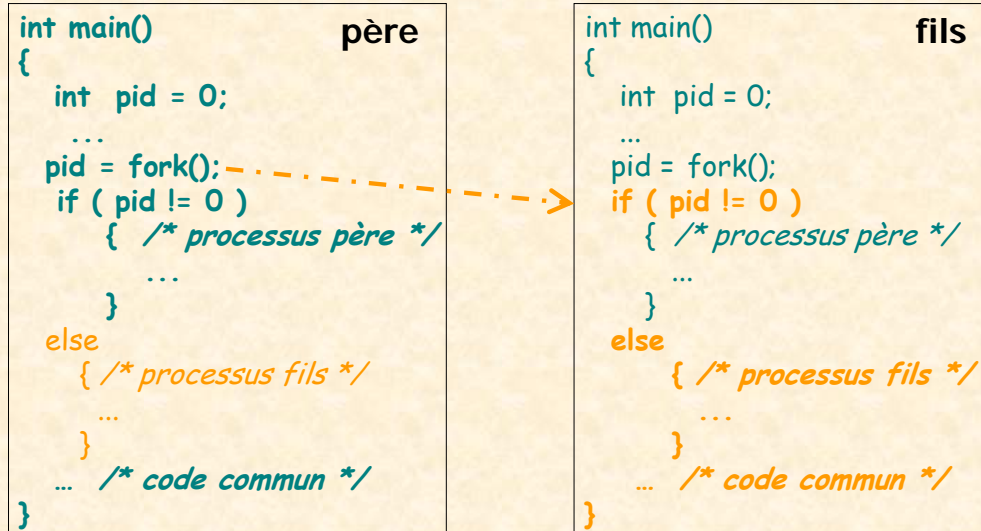
- La valeur de retour de `fork()` vaudra 0 dans le processus fils.
- La fonction `fork()` ne duplique que le thread dans lequel elle est appelée.
- Le code n'est pas dupliqué physiquement. Le PCB du nouveau processus pointe sur les zones mémoire du code du père mais le compteur d'instructions est indépendant.
- La table des descripteurs de fichiers est recopiée mais l'entrée d'un descripteur dupliqué pointe sur la même entrée de la table des fichiers ouverts, ce qui entraîne le partage de la position courante dans le fichier (pointeur de fichier).
- Le code exécuté par le processus fils commence en retour de l'appel à `fork()`.

Utilisation

```
int p;  
...  
p = fork();
```


LES PROCESSUS

CREATION D'UN PROCESSUS



Remarques

- La seule manière de séparer le code devant être exécuté par le père du code devant être exécuté par le fils est de conditionner les instructions du code du fils par la valeur de retour de la fonction `fork()` égale à 0.
- Il faudra donc être vigilant dans la structuration du code !

LES PROCESSUS

TERMINAISON D'UN PROCESSUS

`void exit (int status);`

paramètre : le code retour du processus

- Ne revient jamais au programme appelant
- Libère toutes les ressources du processus
- Les processus fils sont "adoptés" par init (PID 1)
- Remarque : l'instruction **return** de la fonction `main()` invoque **exit()**

Terminaison d'un processus avec création d'un fichier core

`void abort (void);`



exit()

- Termine tous les threads du processus
- Ferme les flux
- Informe son processus père de sa terminaison en lui envoyant le signal **SIGCHLD**

abort()

- Effectue les mêmes traitements que `exit()` et crée un fichier **core** dans le dossier courant. Celui-ci contient l'image mémoire du programme au moment de sa terminaison.

- Si le programme a été compilé avec les options de debug, le fichier **core** pourra être analysé par la commande :

`gdb nom_du_programme core`

- Remarque : pour des raisons évidentes de sécurité, un programme s'exécutant avec le bit setuid ou le bit setgid ne créera pas de fichier **core**.

LES PROCESSUS

SYNCHRONISATION SUR LA FIN D'UN FILS

Quand un processus se termine, il reste à l'état "zombi" tant que son père ne s'est pas synchronisé

```
pid_t wait ( int *status );
```

paramètre : le code retour du processus fils

valeur de retour : le PID du fils ou -1 (pas de fils)

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

pid : >0 -> le pid du processus fils à attendre

status : le code retour du processus fils

options : 0 ou cf. man

valeur de retour : le PID du fils ou -1 (erreur)



- On notera que si un processus père se termine sans s'être synchronisé avec la terminaison de ses fils, ceux-ci sont « adoptés » par le processus `init`.

- Des macros permettent d'analyser la valeur du code retour du processus fils, se référer à la page de man de la fonction `wait()`.

wait()

Si le processus a au moins un fils "zombi" : le fils disparaît de la table des processus

S'il y a au moins un fils en cours d'exécution (et pas de fils zombi) : `wait()` est bloquant en attente du signal de fin du premier fils qui se termine.

Autant d'appels à `wait()` que de fils sont nécessaires

waitpid()

Le paramètre `pid` permet de préciser l'identifiant du processus fils à attendre.

Il peut aussi prendre les valeurs suivantes :

- < -1 : attendre la fin de n'importe lequel des processus fils dont le GID du processus est égal à la valeur absolue de `pid`
- 1 : attendre n'importe lequel des processus fils (identique à `wait()`)
- 0 : attendre n'importe lequel des processus fils dont le GID du processus est égal à celui du processus appelant.

LES PROCESSUS

SYNCHRONISATION SUR LA FIN D'UN FILS

```
int main(void) {
    int status, pid;
    if ( (pid = fork() ) == 0) {
        /* processus fils */
        sleep(5);
        exit(8);
    }
    wait( &status);
    printf("Code retour du fils = %d \n ", WEXITSTATUS(status));
    return 0;
}
```



L'appel à la fonction `wait()` permet d'attendre la fin de l'exécution du processus fils et de récupérer sa valeur de retour.

La macro `WEXITSTATUS()` permet de récupérer la valeur renvoyée par `exit()` ou `return` du processus fils.

Ce programme affiche donc :

Code retour du fils = 8

CREATION DE PROCESSUS



Exemples 5 et 6

LES PROCESSUS

RECOUVREMENT DU CODE D'UN PROCESSUS

- Exécution d'un nouveau programme exécutable sans création de nouveau processus
- Le code et les données du processus courant sont remplacés par ceux du nouveau programme
- Conservation de l'environnement courant (descripteurs de fichiers, signaux, ...)
- Pas de retour à la fonction appelante (code écrasé)
- Famille de fonctions `execXX()`



La fonction `fork()` permet uniquement de créer une copie du processus père.

Si on souhaite lancer un nouvel exécutable, on combinera `fork()` et une des fonction de la « famille » `execXX()` selon la structure suivante :

```
p = fork();
if (p==0) {
    // recouvrement du fils par un nouvel exécutable :
    // appel à une fonction de la famille execXX()
}
// suite du code du père
```

LES PROCESSUS

RECOUVREMENT DU CODE D'UN PROCESSUS

Fonction	Recherche dans le PATH	Passage des arguments	Variables d'environnement
<code>execl ()</code>	NON	Liste	conservées
<code>execlp ()</code>	OUI	Liste	conservées
<code>execle ()</code>	NON	Liste	nouvelles
<code>execv ()</code>	NON	Tableau	conservées
<code>execvp ()</code>	OUI	Tableau	conservées
<code>execve ()</code>	NON	Tableau	nouvelles



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

```
int execl (const char *path, const char *arg, ...);  
int execlp (const char *file, const char *arg, ...);  
int execle (const char *path, const char *arg , ...,  
            char * const envp[]);  
int execv (const char *path, char *const argv[]);  
int execvp (const char *file, char *const argv[]);
```

Exemples

```
execl("/home/joelle/prog", "prog", NULL);  
  
execlp("ls", "ls", NULL);  
  
execlp("cp", "cp", "toto" , "titi", NULL);  
  
char *ligneCmde[] = {"prog", NULL};  
execv("/home/joelle/prog", ligneCmde);
```

RECOUVREMENT DU CODE D'UN PROCESSUS



Exemple 7

COMMUNICATION INTERPROCESSUS

Deux processus s'exécutant en mode utilisateur ne possèdent pas de mémoire commune.

Pour qu'ils puissent se communiquer des données, ils devront obligatoirement passer par un mécanisme offert par le système.

Dans cette partie, nous verrons les moyens de communication et de synchronisation offerts aux processus tournant sur la même machine puis sur des machines accessibles à travers un réseau.

MECANISMES UNIX



- Communication
 - fichiers
 - tubes (pipelines)
 - tubes nommés
 - mémoire partagée
 - files de messages
 - sockets (réseau)
- Synchronisation
 - ensembles de sémaphores
 - signaux
 - verrous

On distinguera les mécanismes de communication qui permettent d'échanger des données des moyens de synchronisation qui ne véhiculent pas d'information mais qui permettent de gérer la concurrence (sémaphores, verrous) ou de déclencher des traitements de manière asynchrone (signaux).

COMMUNICATION INTER PROCESSUS

CRITERES DE CHOIX



- Communication ou synchronisation
- Volume d'information
- Rapidité des échanges
- Protection des données
- Conservation des données
- Structuration des données

La conception d'une application multiprocessus impose de choisir entre les moyens de communication disponibles.

On peut caractériser ceux-ci à l'aide des critères ci-dessus.

LES FICHIERS



- Critères de choix
 - Volume d'information ---> **important**
 - **Communication** ou synchronisation
 - Rapidité des échanges ---> **lente**
 - Protection des données ---> **à synchroniser**
 - Conservation des données ---> **oui**
(*Données indépendantes des processus*)
 - Structuration des données ---> **non**

Les fichiers permettent de conserver les données de manière permanente, indépendamment du processus qui les a créés.

C'est un moyen de communication « lent » dans la mesure où il implique l'accès à un périphérique.

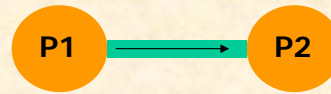
Les accès concurrents à un fichier seront à synchroniser, soit par de l'exclusion mutuelle de code, soit par des verrous posés sur une partie ou sur la totalité d'un fichier.

Sauf dans le cas des bases de données, les données d'un fichier ne sont pas « nativement » structurées. Il appartiendra au programmeur de gérer une éventuelle structuration.

On les utilisera donc pour des données permanentes, indépendantes des processus qui y accèdent.

COMMUNICATION INTER PROCESSUS

LES TUBES



- Critères de choix
 - Volume d'information ---> **moyen**
 - **Communication** ou synchronisation
 - Rapidité des échanges ---> **rapide**
 - Protection des données ---> **synchronisé**
 - Conservation des données ---> **non**
 - Structuration des données ---> **non**



Les tubes (ou pipelines) sont un moyen de communication implanté en RAM.

C'est donc un moyen de communication « rapide ».

Ils fonctionnent sur le mode producteur/consommateur, les accès concurrents sont synchronisés par le système et la lecture est destructrice.

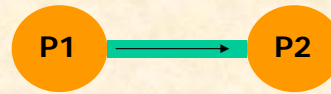
L'atomicité des opérations en cas d'écritures ou de lectures concurrentes n'est garantie que pour une taille de données inférieure à la constante **PIPE_BUF** (sous Linux, **PIPE_BUF** = 4096 octets)

Les données ne sont pas nativement structurées (flot d'octets).

Depuis Linux 2.6.11, la capacité d'un tube est de 65536 octets (dans les versions antérieures, elle était de la taille d'une page).

COMMUNICATION INTER PROCESSUS

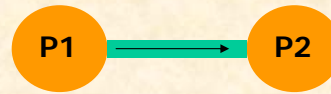
LES TUBES



- Mécanisme de communication unidirectionnelle entre processus
- FIFO implantée en mémoire
- Gérés via le système de fichier
 - descripteurs
 - droits
 - accès par les primitives de gestion des fichiers
 - mais ... pas de déplacement dans un tube

COMMUNICATION INTER PROCESSUS

LES TUBES



Synchronisation de type producteur/consommateur

➤ Lecture

tube vide : attente

fin du tube si pas d'écrivain

➤ Ecriture

tube plein : attente

fin du tube et du processus si pas de lecteur



Synchronisation en lecture

- si le tube contient des données, elles sont consommées
- le tube ne contient pas de données :
 - s'il existe au moins un processus écrivain : attente
 - s'il n'existe plus de processus écrivain* : la lecture signale une fin de fichier. Le tube est supprimé.

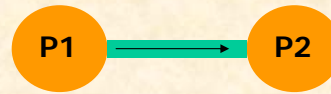
*processus écrivain : processus ayant un descripteur en écriture ouvert sur le tube. La fermeture de tous les descripteurs ouverts en écriture conduit la lecture à détecter la fin du tube.

Synchronisation en écriture

- si il y a de la place disponible dans le tube : écriture
- si le tube est plein : attente
 - s'il n'existe plus de processus lecteur* : le processus écrivain reçoit le signal SIGPIPE qui cause par défaut sa terminaison et affiche le message "Broken pipe". Le tube est supprimé.

*processus lecteur : processus ayant un descripteur en lecture ouvert sur le tube

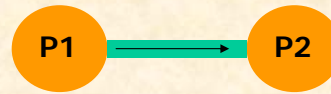
LES TUBES



2 types de tubes

- tube sans nom
 - communication entre processus de même famille
 - accessible uniquement par ses descripteurs
- tube nommé
 - communication entre processus "indépendants"
 - accessible par son chemin d'accès dans le système de fichiers

LES TUBES SANS NOM



➤ Création

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

fd : tableau de 2 descripteurs

fd[0] : lecture

fd[1] : écriture

valeur de retour : 0 (OK) ou -1 (erreur)



Exemples 8 et 9



La fonction `pipe()` crée le tube de communication et ajoute 2 entrées dans la table des descripteurs de fichiers du processus :

- un descripteur qui servira à lire des informations dans le tube,
- un descripteur qui servira à écrire des informations dans le tube.

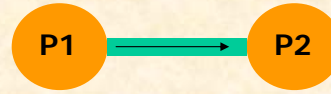
Les valeurs de ces descripteurs sont chargées dans le tableau passé en paramètre à la fonction.

Les tubes sans nom servent à la communication entre processus de la même famille (père et fils). En effet, les descripteurs sont internes et ne seront transmis que par copie de la table des descripteurs de fichiers au moment d'un appel à `fork()` .

```
int p=0, tube[2];
char buf[20];
pipe(tube);
p = fork();
if (p == 0) { // processus fils
    close (tube[0]);
    write (tube[1], "Hello !", 7);
    close (tube[1]);
}
else { // processus pere
    close (tube[1]);
    read (tube[0], buf, sizeof(buf));
    close (tube[0]);
    ...
}
```

COMMUNICATION INTER PROCESSUS

LES TUBES NOMMES



➤ Création

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

```
int mkfifo ( const char *nom, mode_t mode );
```

nom : chemin d'accès du tube

mode : droits d'accès aux fichiers

valeur de retour : 0 (OK) ou -1 (erreur)

OU

```
int mknod ( const char *nom, mode_t mode2 );
```

mode2 = **mode** | S_IFIFO

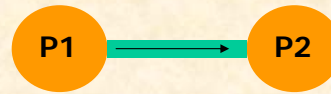


La fonction **mkfifo()** crée un tube de communication en lui donnant un nom. Celui-ci apparaît dans le système de fichiers (fichier spécial) et est indépendant de l'existence du processus l'ayant créé.

Pour l'utiliser, il faudra préalablement l'ouvrir à l'aide de la fonction **open()**.

On supprimera un tube nommé comme un fichier, par appel à la fonction **unlink()**.

LES TUBES NOMMES



Exemple 10

On notera dans cet exemple l'utilisation de la fonction `fdopen()`. Celle-ci permet de récupérer un flux (`FILE *`) à partir d'un descripteur de fichier ouvert :

```
FILE * fdopen ( int fd , const char *mode );
```

fd : descripteur du fichier

mode : mode d'ouverture compatible avec celui de `fd`,
valeurs : cf. modes de `fopen()`

valeur de retour : le flux ou `NULL` (erreur)

LES SIGNAUX



- Critères de choix
 - Volume d'information ---> **pas de données**
 - Communication ou **synchronisation**
 - Rapidité des échanges ---> **asynchrone**
 - Protection des données ---> -
 - Conservation de l'information ---> -
 - Structuration des données ---> -

Les signaux sont difficiles à classer, dans la mesure où ils ne véhiculent pas d'information et sont fondamentalement un mécanisme asynchrone.

Ils permettent de demander à un processus d'exécuter un traitement.

On les classera tout de même dans les moyens de synchronisation, ce qui est effectivement le cas pour un processus père qui attend par l'appel système `wait()` le signal de terminaison d'un de ses processus fils.

LES SIGNAUX



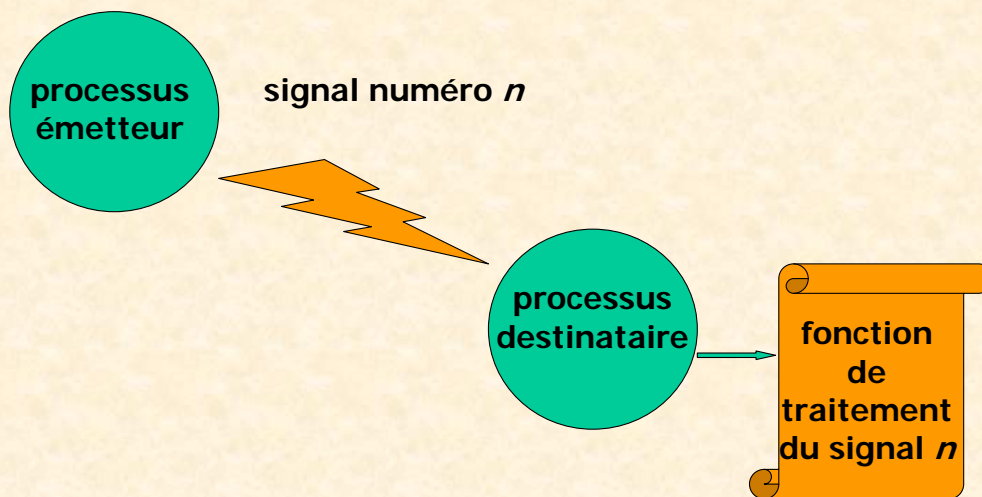
- Interruption logicielle
- Mécanisme asynchrone : à la prise en compte d'un signal, le processus exécute un traitement
- Modes de traitement
 - ignorance volontaire : pas pour tous les signaux
 - traitement par défaut : en général, `exit()`
 - exécution d'une fonction utilisateur (`handle`)

Chaque système Unix définit une liste des signaux disponibles et leur mode de traitement. Un traitement par défaut est associé à chaque signal.

La liste des signaux et des traitements par défaut associés peut être obtenue par la commande `man -s7 signal`

COMMUNICATION INTER PROCESSUS

LES SIGNAUX



La prise en compte d'un signal par le processus destinataire n'est pas immédiate. Sa prise en compte se fera :

- si le processus destinataire est actif : le signal est pris en compte lors d'une transition entre l'état "actif noyau" et l'état "actif utilisateur" (au retour d'un appel système)
- si le processus destinataire est endormi : la réception d'un signal le fait passer à l'état "prêt". Le signal sera traité quand le processus passera à l'état "actif utilisateur".

Il s'agit donc d'un mécanisme asynchrone.

LES SIGNAUX



Catégories de signaux

- exceptions : lancées par le noyau en cas d'exécution anormale (/0, violation de segment, instruction illégale, ...)
- interaction du terminal : CTRL-C, CTRL-Z, ...
- interactions entre processus : synchronisation , réveil, alarme, ...

COMMUNICATION INTER PROCESSUS

QUELQUES SIGNAUX LINUX



Nom	Valeur	Action	Evénement
SIGHUP	1	A	Déconnexion terminal contrôle
SIGINT	2	A	Interruption clavier (CTRL-C)
SIGQUIT	3	A	Quit clavier (CTRL-\)
SIGTRAP	5	C	Trace ou Breakpoint (debug)
SIGABRT	6	C	Envoyé par abort()
SIGKILL	9	AE	Kill
SIGBUS	10,7	A	Bus Error
SIGSEGV	11	C	Erreur de Segmentation
SIGPIPE	13	A	Broken Pipe
SIGALRM	14	A	Timer alarm() échu
SIGTERM	15	A	Fin
SIGUSR1	30,16,10	A	Signal utilisateur 1
SIGUSR2	31,12,17	A	Signal utilisateur 2
SIGCHLD	20,17,18	B	Fin d'un fils



La liste des signaux ci-dessus n'est pas exhaustive. La liste complète peut être obtenu par la commande **man -s7 signal**.

Dans ce tableau, l'action correspond au comportement par défaut :

A : terminaison

B : signal ignoré

C : terminaison avec création d'un fichier core

E : ne peut pas être ignoré par le processus

On notera que pour des raisons de portabilité, on utilisera le mnémonique du signal et non sa valeur (par exemple, **SIGUSR1** au lieu de 16).

Rappel : La commande **kill** permet d'envoyer un signal à partir de la ligne de commande.

kill 312 envoi de **SIGTERM** au processus de pid 312

kill -9 312 envoi de **SIGKILL** au processus de pid 312

LES SIGNAUX



➤ Émettre un signal

```
int kill ( pid_t pid , int sig );
```

pid : processus destinataire(s)

> 0 : PID du destinataire

= 0 : envoi à tous les processus du groupe

< 0 : émis par root : envoi du signal à tous les
processus non système

émis par un autre utilisateur : envoi du signal
à tous ses processus

sig : numéro du signal

valeur de retour : 0 ou -1 (erreur)



```
int pere;  
...  
pere = getppid();  
kill(pere, SIGKILL);
```

LES SIGNAUX



- Émettre un signal



Exemple 14



```
int main(void) {
    int pere;

    if (fork() == 0) {
        /* processus fils */
        pere = getppid();
        kill(pere, SIGTERM);
        while( kill(pere, 0)==0 ) {
            printf("pere non termine\n");
            sleep(1);
        }
        printf("mon nouveau pere %d\n", getppid());
    }
    else {
        /* processus pere */
        sleep(10);
    }
    return 0;
}
```

Résultat de l'exécution

mon nouveau père 1

LES SIGNAUX



- Paramétrer la réception d'un signal

2 méthodes de gestion des signaux

ANSI : gestion simplifiée. Les signaux sont systématiquement délivrés au processus.

Posix : Chaque processus définit un masque des signaux : liste des signaux bloqués. Le comportement du processus pendant l'exécution du gestionnaire de signal est paramétrable.



Signaux ANSI

La gestion ANSI des signaux est une gestion simplifiée, peu performante dans la mesure où tous les signaux sont délivrés au processus qui va examiner le traitement à effectuer même si le processus ignore le signal.

Malgré sa disponibilité sur tous les systèmes Unix, le comportement du processus pendant l'exécution du gestionnaire de signal (handler) peut différer. Pour cette raison, elle n'est pas portable.

Sous Linux, quand le gestionnaire de signal est appelé, le comportement du processus vis à vis du signal n'est pas réinitialisé, et la distribution des instances suivantes du signal est bloquée tant que le gestionnaire s'exécute.

Les autres signaux peuvent être distribués.

Signaux Posix

La gestion Posix est plus performante : si un signal est ignoré par un processus, il ne lui sera pas délivré.

Un masque des signaux (liste des signaux bloqués ou non) est applicable à deux niveaux :

- signaux bloqués pour le processus
- signaux bloqués pendant l'exécution d'un gestionnaire de signal.

LES SIGNAUX



➤ Paramétrer la réception d'un signal : C ANSI

Pour chaque signal dont on veut modifier le comportement :

```
void ( * signal ( int sig, void ( *action ( int ) ) ) ( int );
```

sig : numéro du signal

action : nom du gestionnaire de signal *ou*

SIG_DFL : traitement par défaut *ou*

SIG_IGN : signal à ignorer (sauf SIGKILL)

valeur de retour : adresse de la fonction ou -1 (erreur)



Numéro du signal

On utilisera le mnémonique (cf. `man -s7 signal`) et non le numéro explicite du signal pour des raisons de portabilité.

Gestionnaire de signal (handler)

Le gestionnaire de signal est une fonction dont le prototype est imposé :

```
void nom_du_gestionnaire(int numSignal);
```

Aucune valeur n'est renvoyée et le numéro du signal ayant déclenché l'appel du gestionnaire est passé en paramètre.

```
void fonction_usr1(int numS ) {  
    ...  
}  
  
int main(void) {  
    ...  
    signal(SIGUSR1, fonction_usr1);  
    signal(SIGINT, SIG_IGN);  
    ...  
}
```

LES SIGNAUX



- Paramétrer la réception d'un signal : C ANSI



Exemple 12



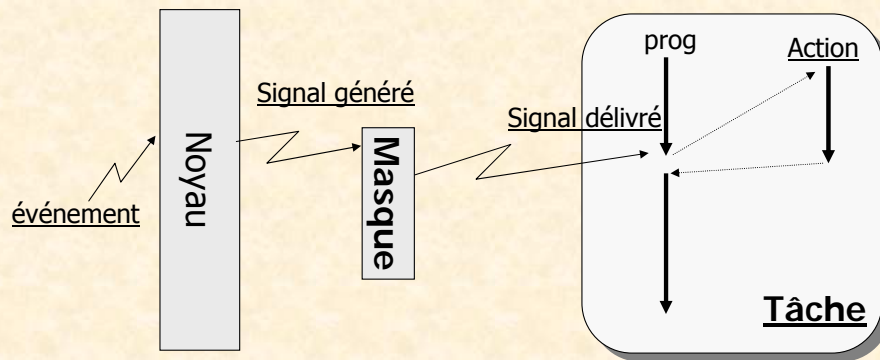
```
unsigned long somme=0;

/* gestionnaire des signaux SIGUSR1 et SIGUSR2 */
void espion(int numSig) {
    printf("Somme = %lu\n", somme);
    switch(numSig){
        case SIGUSR1:
            break;
        case SIGUSR2:
            exit(0);
    }
}

int main(void) {
    signal(SIGINT, SIG_IGN);
    signal(SIGUSR1, espion);
    signal(SIGUSR2, espion);
    printf("Je suis le processus numero: %d\n", getpid());
    while(somme<ULONG_MAX) {
        somme++;
    }
    printf("Calcul termine: %lu", somme);
    return(0);
}
```

LES SIGNAUX

- Paramétrer la réception d'un signal : POSIX



LES SIGNAUX



➤ Paramétrer la réception d'un signal : Posix

1 - Mise en œuvre du masque de blocage des signaux

Préparation du masque

- création de l'ensemble de signaux (tout ou rien)
- ajout ou retrait de signaux

Application du masque

2 - Définition du comportement spécifique des signaux autorisés

LES SIGNAUX



- Paramétrer la réception d'un signal : Posix

1 - Mise en œuvre du masque de blocage des signaux

Préparation du masque

```
int sigemptyset ( sigset_t * ensemble );
```

ensemble : ensemble de signaux
valeur de retour : 0 ou -1 (erreur)

```
int sigfillset ( sigset_t * ensemble );
```

ensemble : ensemble de signaux
valeur de retour : 0 ou -1 (erreur)



Le masque permet de définir le comportement du processus face à tous les signaux existant. Selon le nombre de signaux à paramétrer, on choisira de partir d'un masque vide ou d'un masque contenant tous les signaux.

Création d'un ensemble de signaux vide

```
sigset_t ensemble;  
sigemptyset ( &ensemble );
```

Création d'un ensemble de signaux contenant tous les signaux

```
sigset_t ensemble;  
sigfillset ( &ensemble );
```


LES SIGNAUX



- Paramétrer la réception d'un signal : Posix

1 - Mise en œuvre du masque de blocage des signaux Préparation du masque

int sigaddset (sigset_t * ensemble, int numSignal);

ensemble : ensemble de signaux

numSignal : numéro du signal à ajouter

valeur de retour : 0 ou -1 (erreur)

int sigdelset (sigset_t * ensemble, int numSignal);

ensemble : ensemble de signaux

numSignal : numéro du signal à supprimer

valeur de retour : 0 ou -1 (erreur)



Une fois l'ensemble de signaux initialisé, on choisira d'ajouter ou de supprimer des signaux.

Ajout d'un signal dans un ensemble

```
sigset_t ensemble;  
sigemptyset ( &ensemble );  
sigaddset ( &ensemble, SIGUSR1 );  
sigaddset ( &ensemble, SIGUSR2 );
```

Suppression d'un signal d'un ensemble

```
sigset_t ensemble;  
sigfillset ( &ensemble );  
sigdelset ( &ensemble, SIGINT );
```

LES SIGNAUX



- Paramétrer la réception d'un signal : Posix

1 - Mise en œuvre du masque de blocage des signaux

Application du masque

```
int sigprocmask ( int methode, const sigset_t * ensemble,  
                  sigset_t * ancien );
```

methode : SIG_SETMASK

ensemble : masque des signaux à appliquer

ancien : sauvegarde de l'ancien masque

valeur de retour : 0 ou -1 (erreur)



Une fois l'ensemble de signaux préparé, on l'applique. A partir de ce moment, tous les signaux n'appartenant pas à l'ensemble seront bloqués, sauf ceux ne pouvant pas être ignorés (par exemple **SIGKILL**).

Les signaux autorisés ont leur comportement par défaut.

```
sigset_t ensemble, sauv_ensemble;  
sigfillset ( &ensemble );  
sigdelset ( &ensemble, SIGINT );  
sigprocmask (SIG_SETMASK, &ensemble, &sauv_ensemble);
```

LES SIGNAUX



- Paramétrer la réception d'un signal : Posix

2 - Définition du comportement spécifique d'un signal

```
int sigaction ( int numSig, const struct sigaction * newSigAct,  
               struct sigaction * oldSigAct );
```

numSig : numéro du signal

newSigAct : comportement à appliquer

oldSigAct : sauvegarde de l'ancien comportement

valeur de retour : 0 ou -1 (erreur)



La fonction `sigaction()` permet d'appliquer un comportement à un processus vis-à-vis d'un signal donné. Ce comportement est défini par la structure `sigaction` passée en paramètre.

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

sa_handler : définit le comportement du signal. Valeurs possibles :

SIG_DFL : comportement par défaut,

SIG_IGN : signal bloqué

nom d'une fonction : gestionnaire de signal

sa_mask : ensemble des signaux autorisés ou bloqués pendant l'exécution du gestionnaire de signal. A préparer de la même manière que le masque des signaux du processus.

sa_flags : combinaison OU de flags : cf. man `sigaction`

LES SIGNAUX



- Paramétrer la réception d'un signal : Posix



Exemple 11



```
int main(void) {  
  
    sigset_t masque;  
    struct sigaction act;  
  
    sigemptyset(&masque);  
    sigaddset(&masque, SIGINT);  
    sigprocmask(SIG_SETMASK, &masque, NULL);  
  
    act.sa_handler=espion;  
    act.sa_flags=0;  
    sigfillset(&(act.sa_mask));  
    sigaction(SIGUSR1, &act, NULL);  
    sigaction(SIGUSR2, &act, NULL);  
  
    printf("Je suis le processus numero: %d\n", getpid());  
    while(somme<ULONG_MAX) {  
        somme++;  
    }  
  
    printf("Calcul termine: %lu", somme);  
    return(0);  
}
```

LES SIGNAUX



- Mise en place d'une temporisation



Exemple 13



Mise en place d'une temporisation

La fonction `alarm()` permet de mettre en place une temporisation.

```
unsigned int alarm(unsigned int nb_sec);
```

Au bout de `nb_sec` secondes à partir de l'appel de la fonction, le signal `SIGALRM` est envoyé au processus.

On notera qu'il ne s'agit pas d'une temporisation temps réel et que si le signal est envoyé au processus au minimum `nb_sec` secondes après la mise en place de l'alarme, on ne peut pas prévoir dans quel délai il le prendra réellement en compte.

L'appel `alarm(0)` annule toute temporisation précédemment mise en place.

LES IPC System V



- 3 outils de communication entre processus locaux
 - mémoire partagée
 - ensembles de sémaphores
 - files de message
- Mécanismes communs
 - nommage via une clé
 - persistants
 - commandes de contrôle
 - fichier d'entête : `sys/ipc.h`



Les IPC (Inter Process Communication)

Les IPC System V constituent l'API historique de communication entre processus locaux sous Unix.

Ils offrent trois moyens de communication/synchronisation qui suivent des règles de programmation et de fonctionnement homogènes :

- zones de mémoire partageable entre processus (*shared memory*, *shm*)
- ensembles de sémaphores
- files de messages : mécanisme de boîte aux lettres (*message queue*)

On notera qu'il existe une API de communication interprocessus Posix, plus récente et qui offre les mêmes fonctionnalités que l'API System V.

Pour plus d'information sur cette API se référer au man.

- Mémoire partagée Posix : `man shm_overview`
- Ensembles de sémaphores Posix : `man sem_overview`
- Files de messages Posix : `man mq_overview`

Dans le cadre de ce cours, nous traiterons de l'API System V.

TIRAGE DE CLÉ



- Obtention d'un « nom » unique pour un IPC

`key_t ftok (const char * path, int val);`

`path` : chemin d'accès d'un fichier

`id` : valeur combinée avec `path` pour obtenir la clé

valeur de retour : une clé ou -1 (erreur)

- A chaque combinaison (`path`, `val`) correspond une clé unique
- Mais ... en cas de liens, tirage de la même clé !



Le nommage des IPC System V

Un IPC étant partagé entre plusieurs processus, il faut lui donner un « nom » lors de sa création. Ce nom doit être unique sur le système et connu par tous les processus qui l'utilisent.

Le problème de l'unicité de ce nom est résolu par un mécanisme de tirage d'une valeur selon un algorithme qui fait correspondre à un couple (numéro d'inode d'un fichier, valeur entière) une clé unique. Cette clé constituera le « nom » de l'IPC.

Ce tirage est réalisé par la fonction `ftok()`.

`key_t ftok (const char * path, int val);`

`path` : chemin d'accès d'un fichier existant

`id` : valeur combinée avec `path` pour obtenir la clé. Doit être différent de 0

valeur de retour : une clé ou -1 (erreur)



Exemple 15

```
int main(void) {  
    printf("%d\n", ftok("exemple_pipe.c", 'A'));  
    printf("%d\n", ftok("exemple_pipe.c", 'G'));  
    printf("%d\n", ftok("exemple_fork1.c", 'A'));  
    printf("%d\n", ftok("exemple_fork1.c", 'G'));  
    printf("%d\n", ftok("exemple_pipe.c", 1));  
    printf("%d\n", ftok("exemple_fork1.c", 1));  
    return 0;  
}
```

Résultat d'exécution

```
1090916642  
1191579938  
1090916631  
1191579927  
17174818  
17174807
```


COMMANDES DE CONTRÔLE DES IPC



- Liste des IPC présents

ipcs

Exemple

IPC status from <running system> as of Wed Aug 25 09:05:23

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Message Queues:

Shared Memory:

m	0	0xffffffff	--rw-rw-rw-	luter	profs
---	---	------------	-------------	-------	-------

Semaphores:

s	0	0xffffffff	--ra-r--r--	luter	profs
---	---	------------	-------------	-------	-------



La commande **ipcs** permet de visualiser tous les IPC System V présents sur le système.

On notera qu'il n'existe pas de commande spécifique pour les IPC Posix.

COMMANDES DE CONTRÔLE DES IPC



➤ Supprimer un IPC

ipcrm -option numeroIPC

option : s -> ensemble de sémaphores

m -> mémoire partagée

q -> file de message

Exemple

ipcrm -s 200



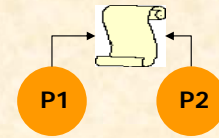
Les IPC étant indépendants de l'existence des processus, ils doivent être supprimés explicitement (commande **ipcrm** ou appel à la fonction de suppression). Autrement, ils ne seront effacés qu'au prochain redémarrage du système.

Un utilisateur ne pourra supprimer un IPC que si il en est propriétaire.

La suppression définitive d'une file de messages ou d'un ensemble de sémaphores est immédiate (même s'ils sont en cours d'utilisation par un processus).

La suppression définitive d'une zone de mémoire partagée n'interviendra que lorsqu'elle ne sera plus attachée à aucun processus.

MÉMOIRE PARTAGÉE



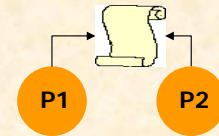
➤ Critères de choix

- Volume d'information ---> **faible**
- **Communication** ou synchronisation
- Rapidité des échanges ---> **rapide**
- Protection des données ---> **non synchronisé**
- Conservation des données ---> **non**
- Structuration des données ---> **oui**

Une zone de mémoire partagée peut être vue comme une « variable globale » à plusieurs processus.

Il s'agit donc d'un moyen de communication en RAM, rapide, accessible comme toute variable et ne possédant pas de mécanisme de synchronisation intégré.

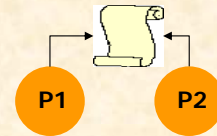
MÉMOIRE PARTAGÉE



- "Allocation dynamique" d'une zone de mémoire système
- Accessible par les processus
 - disposant de son nom (clé)
 - autorisés (droits d'accès)
- Principe
 1. Création ou ouverture de la zone de mémoire
 2. Attachement à l'espace mémoire du processus
 3. Manipulation par son adresse (variable)

Toutes les fonctions de gestion des zones de mémoire partagée sont déclarées dans le fichier d'entête **sys/shm.h**

MÉMOIRE PARTAGÉE



➤ Création ou ouverture d'une zone de mémoire partagée

```
int shmget ( key_t cle, size_t nbOctets, int flags );
```

cle : clé unique ou IPC_PRIVATE

nbOctets : taille de la zone en octets

flags : Combinaison OU de constantes:

IPC_CREAT : création ou ouverture

IPC_EXCL : création exclusive. Échoue si existe déjà droits d'accès (si IPC_CREAT)

valeur de retour : l'identifiant de la zone ou -1 (erreur)



La fonction `shmget()` permet de créer une nouvelle zone de mémoire partagée (cas où le flag `IPC_CREAT` est positionné) ou d'ouvrir une zone existante. Elle renvoie un identifiant qui permettra de la manipuler.

Dans le cas d'une zone de mémoire partagée destinée à être utilisée entre processus de la même famille (père, fils), le processus père créera la zone avant le lancement des fils. Ceux-ci hériteront ainsi de son identifiant. La création d'une clé externe devient inutile. On utilisera alors la valeur `IPC_PRIVATE` à la place de la clé.

La création n'est possible que si la zone de mémoire n'existe pas. Dans le cas d'une demande de création sur une zone existante, la fonction renverra un identifiant pour cette zone (ouverture), sauf si le flag `IPC_EXCL` a été positionné.

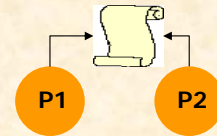
La taille de la zone demandée à l'ouverture doit être \leq à la taille fixée à la création.

Seuls les droits en lecture et écriture sont pertinents pour les zones de mémoire partagée.

```
int id;  
key_t cle;  
cle = ftok( "toto", 2);  
  
id = shmget(cle, 256, IPC_CREAT | 0600);
```

COMMUNICATION INTER PROCESSUS

MÉMOIRE PARTAGÉE



➤ Attachement d'une zone de mémoire partagée

```
void * shmat ( int id, const void *adresse, int flags );
```

id : identifiant de la zone de mémoire

adresse : adresse d'attachement ou **NULL** (le système la choisit)

flags : Combinaison OU de constantes ou **0**: SHM_RND, SHM_RDONLY

valeur de retour : l'adresse de la zone ou -1 (erreur)

➤ Détachement d'une zone de mémoire partagée

```
int shmdt (const void *adresse );
```

adresse : adresse d'attachement

valeur de retour : 0 ou -1 (erreur)



Attachement

Afin de pouvoir utiliser la zone de mémoire partagée comme une « variable », un processus doit disposer d'une adresse rattachée à son propre espace d'adressage.

shmat () renvoie cette adresse d'attachement.

Détachement

shmdt () détache le segment de mémoire partagée de l'espace mémoire du processus. Ce détachement est automatique si le processus se termine.

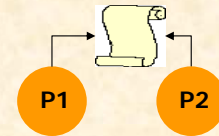
On notera qu'une demande de suppression d'une zone de mémoire partagée ne sera effective que quand tous les processus attachés auront appelé

shmdt () ou seront terminés.

```
int id;
key_t cle;
char * addr;
cle = ftok( "toto", 2);
id = shmget(cle, 256, IPC_CREAT | 0600);
addr = (char *) shmat(id, NULL, 0);
...
shmdt(addr);
```

COMMUNICATION INTER PROCESSUS

MÉMOIRE PARTAGÉE



➤ Opération de contrôle

```
int shmctl ( int id, int cmd, struct shmid_ds * infos);
```

id : identifiant de la zone de mémoire

cmd : opération de contrôle (IPC_RMID, IPC_STAT, IPC_SET)

infos : informations. NULL si cmd = IPC_RMID

valeur de retour : 0 ou -1 (erreur)



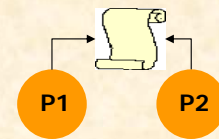
Les opérations de contrôle offertes par `shmctl()` sont la suppression (`IPC_RMID`), la récupération des caractéristiques (`IPC_STAT`), la modification de caractéristiques (`IPC_SET`). Les deux dernières opérations utilisent la structure `shmid_ds`, définie sous Linux de la manière suivante :

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* Permissions d'accès */
    int shm_segsz; /* Taille segment en octets */
    time_t shm_atime; /* Heure dernier attachement */
    time_t shm_dtime; /* Heure dernier détachement */
    time_t shm_ctime; /* Heure dernier changement */
    unsigned short shm_cpid; /* PID du créateur */
    unsigned short shm_lpid; /* PID du dernier opérateur */
    short shm_nattch; /* Nombre d'attachements */
    /* --- Les champs suivants sont privés ----- */
    unsigned short shm_npages; /* Taille segment en pages */
    unsigned long *shm_pages; /* Taille d'une page (?) */
    struct shm_desc *attaches; /* Descript. attachements */
};
```

La récupération des caractéristiques (`IPC_STAT`) renseigne les champs publics de la structure, la modification (`IPC_SET`) permet de modifier les permissions d'accès.

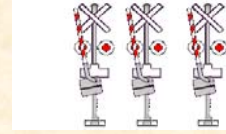
Suppression : `shmctl(id, IPC_RMID, NULL);`

MÉMOIRE PARTAGÉE



Exemple 16

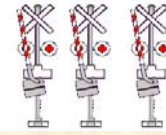
ENSEMBLES DE SEMAPHORES



- Critères de choix
 - Volume d'information ---> **pas de données**
 - Communication ou **synchronisation**
 - Rapidité des échanges ---> -
 - Protection des données ---> -
 - Conservation des données ---> -
 - Structuration des données ---> -

Les ensembles de sémaphores offrent une synchronisation interprocessus de type sémaphore ou mutex.

ENSEMBLES DE SEMAPHORES



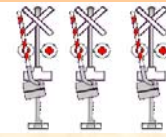
- Gestion simultanée de n opérations sur sémaphores
- Mutex ou compteur : valeur ≥ 0
- Enchaînement
 - Création ou ouverture de l'ensemble
 - Initialisation des sémaphores (si création)
 - Opérations atomiques sur m sémaphores parmi les n
 $P(S_i)$ ou $V(S_i)$

Plusieurs opérations peuvent être effectuées simultanément et de manière atomique sur l'ensemble de sémaphore, ce qui permet d'éviter des cas d'interblocage :

Si toutes les opérations demandées peuvent être effectuées, elles seront validées simultanément.

Si au moins une opération ne peut être effectuée, aucune ne sera validée.

ENSEMBLES DE SEMAPHORES



- Fonctions déclarées dans sys/sem.h
- Création ou ouverture d'un ensemble de sémaphores

int **semget** (key_t cle, int nbSem, int flags);

cle : clé unique ou IPC_PRIVATE

nbSem : nombre de sémaphores de l'ensemble

flags : Combinaison OU de constantes:

IPC_CREAT : création ou ouverture

IPC_EXCL : création exclusive. Échoue si existe déjà
droits d'accès

valeur de retour : l'identifiant de l'ensemble ou -1 (erreur)



La fonction **semget** () permet de créer un nouvel ensemble de sémaphores (cas où le flag **IPC_CREAT** est positionné) ou d'ouvrir un ensemble existant. Elle renvoie un identifiant qui permettra de le manipuler.

Dans le cas d'un ensemble de sémaphores destiné à être utilisé entre processus de la même famille (père, fils), le processus père créera l'ensemble de sémaphores avant le lancement des fils. Ceux-ci hériteront ainsi de son identifiant. La création d'une clé externe devient inutile. On utilisera alors la valeur **IPC_PRIVATE** à la place de la clé.

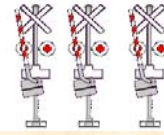
La création n'est possible que si l'ensemble de sémaphores n'existe pas. Dans le cas d'une demande de création sur un ensemble existant, la fonction renverra un identifiant pour cet ensemble (ouverture), sauf si le flag **IPC_EXCL** a été positionné.

Le paramètre nombre de sémaphores de l'ensemble n'est pertinent que lors de la création. La valeur interne des sémaphores est indéterminée tant qu'ils n'auront pas été initialisés par un appel à la fonction **semctl** () .

Seuls les droits en lecture et écriture sont pertinents pour les ensembles de sémaphores.

```
int id;  
key_t cle;  
cle = ftok( "toto", 2);  
  
id = semget(cle, 3, IPC_CREAT | 0600);
```

ENSEMBLES DE SEMAPHORES



➤ Opérations de contrôle sur un ensemble de sémaphores

`int semctl (int id, int semNum, int cmd, autre paramètre);`

id : identifiant de l'ensemble

semNum : numéro du sémaphore concerné (de 0 à n - 1)
ou non pertinent si opération sur l'ensemble

cmd : opération

SETVAL : initialisation d'1 sémaphore,

SETALL : initialisation de tous les sémaphores,

IPC_RMID : suppression de l'ensemble, etc ...

autre param. : facultatif selon l'opération demandée.

Type : union **semun** à déclarer

val. de retour : valeur du sémaphore (GETVAL) ou -1 (erreur)



Les principales opérations de contrôle offertes par `semctl()` sont l'initialisation (**SETVAL**, **SETALL**), la suppression (**IPC_RMID**), la récupération de caractéristiques (**IPC_INFO**, **SEM_INFO**, **IPC_STAT**), la modification de caractéristiques (**IPC_SET**).

C'est une fonction à nombre de paramètres variable. Le dernier argument n'est pertinent que pour certaines opérations.

Dans ce cas, le programme **doit** définir le type de cet argument de la manière suivante :

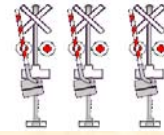
```
union semun {
    int val; /* Pour SETVAL */
    struct semid_ds *buf; /* Pour IPC_STAT et IPC_SET */
    unsigned short *array; /* Pour GETALL et SETALL */
    struct seminfo *__buf; /* Pour IPC_INFO */
};
```

La structure `semid_ds` est définie sous Linux de la manière suivante :

```
struct semid_ds {
    struct ipc_perm sem_perm; /* permissions */
    time_t sem_otime; /* Heure dernier semop */
    time_t sem_ctime; /* Heure dernière modification */
    unsigned short sem_nsems; /* N° du semaphore */
};
```

On notera que la lecture des valeurs internes (**GETVAL**, **GETALL**) ne garantit pas que ces valeurs n'aient pas évolué lors de leur utilisation.

ENSEMBLES DE SEMAPHORES



➤ Opérations P et V sur les sémaphores d'un ensemble

int semop (int id, struct sembuf *sOps, size_t nbOps);

id : identifiant de l'ensemble

sOps : tableau de *nbOps* structures. Chaque structure décrit une des opérations.

nbOps : nombre d'opérations

valeur de retour : 0 ou -1 (erreur)



Le détail des opérations à effectuer est décrit par les éléments du tableau de structures passé en argument (**sOps**). Chaque structure décrit une opération.

```
struct sembuf {  
    unsigned short sem_num; /* Numéro du sémaphore */  
    short sem_op; /* Opération sur le sémaphore */  
    short sem_flg; /* Options pour l'opération */  
};
```

sem_op : >0 pour une opération V(), <0 pour une opération P()

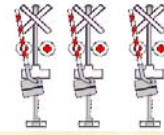
sem_flg : 0, IPC_NOWAIT, SEM_UNDO

Exemple

Ce code effectue une opération P() respectivement sur le premier et le troisième sémaphore d'un ensemble d'identifiant *id*.

```
struct sembuf tabSembuf[2];  
...  
tabSembuf[0].sem_num = 0;  
tabSembuf[0].sem_op = -1;  
tabSembuf[0].sem_flg = 0;  
tabSembuf[1].sem_num = 2;  
tabSembuf[1].sem_op = -1;  
tabSembuf[1].sem_flg = 0;  
semop(id, tabSembuf, 2);
```

ENSEMBLES DE SEMAPHORES



||  **Exemples 18, 19, 20**

FILES DE MESSAGES



- Critères de choix
 - Volume d'information ---> **moyen**
 - **Communication** ou synchronisation
 - Rapidité des échanges ---> **oui**
 - Protection des données ---> **oui**
 - Conservation des données ---> **non**
 - Structuration des données ---> **oui**

Une file de messages peut être vue comme une « boîte aux lettres » partagée entre plusieurs processus.

Il s'agit d'un moyen de communication en RAM, rapide, à lecture destructrice et fonctionnant selon le schéma producteur/consommateur.

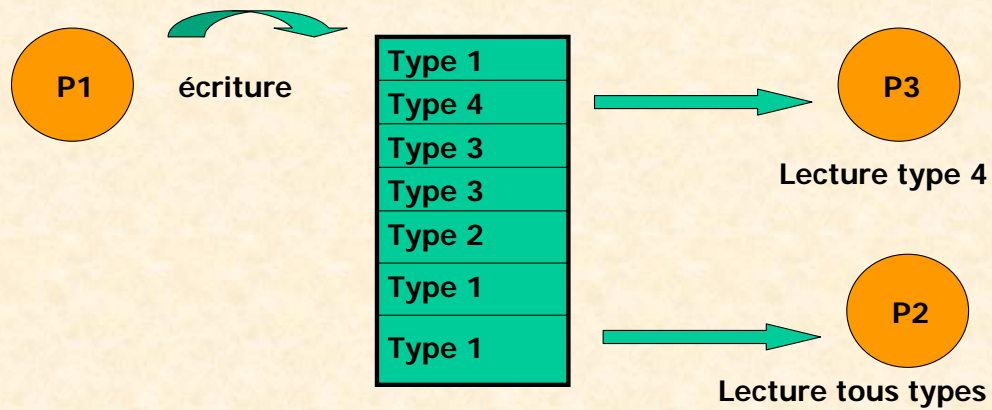
A la différence du pipeline, les données ne sont plus du flot d'octet mais des messages.

Les files de messages sont indépendantes des processus.

COMMUNICATION INTER PROCESSUS

FILES DE MESSAGES

- Mécanisme de boîte aux lettres
- Message = type (numéro) + données



UNIX- Programmation Système

136

Chaque message a un « type » : il s'agit d'un numéro. Lors de la lecture d'un message, un processus peut choisir le type de message qu'il souhaite retirer de la file. Il lira alors le plus ancien message de ce type.

FILES DE MESSAGES



- Fonctions déclarées dans sys/msg.h
- Création ou ouverture d'une file de messages

`int msgget (key_t cle, int flags);`

cle : clé unique ou `IPC_PRIVATE`

flags : Combinaison OU de constantes:

`IPC_CREAT` : création ou ouverture

`IPC_EXCL` : création exclusive. Échoue si existe déjà
droits d'accès

valeur de retour : l'identifiant de l'ensemble ou -1 (erreur)



La fonction `msgget ()` permet de créer une nouvelle file de messages (cas où le flag `IPC_CREAT` est positionné) ou d'ouvrir une file existante. Elle renvoie un identifiant qui permettra de la manipuler.

Dans le cas d'une file de messages destinée à être utilisée entre processus de la même famille (père, fils), le processus père créera la file avant le lancement des fils. Ceux-ci hériteront ainsi de son identifiant. La création d'une clé externe devient inutile. On utilisera alors la valeur `IPC_PRIVATE` à la place de la clé.

La création n'est possible que si la file n'existe pas. Dans le cas d'une demande de création sur une file existante, la fonction renverra un identifiant pour celle-ci (ouverture), sauf si le flag `IPC_EXCL` a été positionné.

Seuls les droits en lecture et écriture sont pertinents pour les files de messages .

```
int id;  
key_t cle;  
cle = ftok( "toto", 2);  
  
id = msgget(cle, IPC_CREAT | 0600);
```

FILES DE MESSAGES



➤ Envoi d'un message

```
int msgsnd ( int id, const void * message, int taille, int flags );
```

id : identifiant de la file de messages
message : adresse du message à envoyer, sous la forme

```
struct monMessage {  
    long type; -> le type du message, >0  
    ...       -> les données utilisateur  
};
```

taille : nombre d'octets des données utilisateur
flags : 0 ou IPC_NOWAIT (non bloquant)
valeur de retour : 0 (OK) ou -1 (erreur)



La fonction `msgsnd()` permet de déposer un message dans une file. La file de message reçoit une copie des données passées dans la structure message.

```
struct myMsg {  
    long type;  
    char data[10];  
} m;  
int id;  
...  
strcpy(m.data, "bonjour");  
m.type = 1;  
msgsnd(id, &m, strlen(m.data)+1, 0);
```

FILES DE MESSAGES



➤ Lecture d'un message

```
int msgrcv ( int id, void * message, int taille,  
                                                    long type, int flags );
```

id : identifiant de la file de messages

message : adresse de stockage du message lu

taille : nombre d'octets des données utilisateur à lire

type : 0 : le premier message disponible
 > 0 : premier message disponible du type donné
 < 0 : premier message disponible du plus petit
 type <= au type donné

flags : MSG_EXCEPT (exclusion du type donné) ou
 IPC_NOWAIT (non bloquant) ou MSG_NOERROR
 (troncature en cas de message trop long)

valeur de retour : 0 (OK) ou -1 (erreur)



La fonction `msgrcv()` permet de lire un message dans une file. Si aucun message du type demandé n'est disponible, la fonction est par défaut bloquante, sauf si le flag `IPC_NOWAIT` a été positionné.

```
struct myMsg {  
    long type;  
    char data[10];  
} m;  
int id;  
...  
msgrcv(id, &m, 10, 0, 0);
```

FILES DE MESSAGES



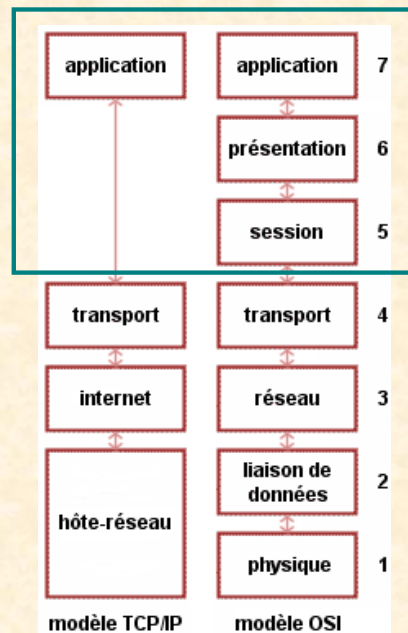
Exemple 17

COMMUNICATION RESEAU

Deux processus s'exécutant sur des machines distantes peuvent communiquer via le réseau.

LES MODELES RÉSEAU

TCP/IP vs OSI



Application

Logique du service rendu à l'utilisateur

Présentation

Conversion, cryptage, formatage, compression de données

Session

Organisation et synchronisation des échanges applicatifs : gestion de la connexion (transactions), gestion du dialogue, gestion des incidents.
Protocoles : NetBIOS, RPC, TLS, ...

Transport

Assure l'acheminement des données d'un processus à un autre processus.
Protocoles : TCP, UDP

Internet

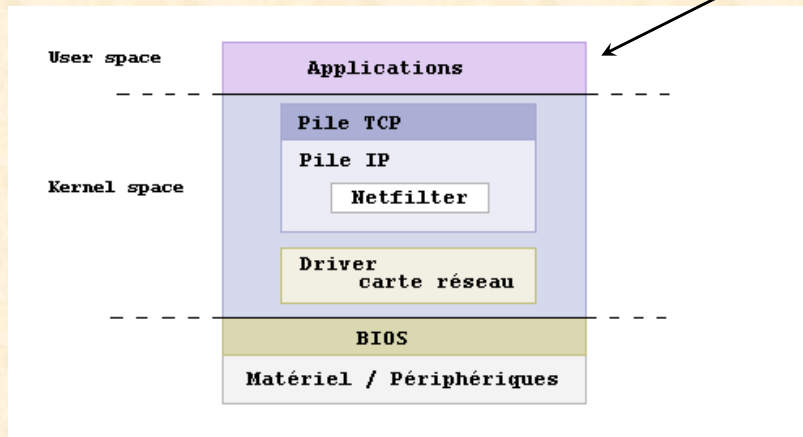
Interconnexion de réseaux. Assure l'adressage et l'acheminement des données vers la machine destinataire. Protocoles : IP + RIP, OSPF (protocoles de routage) , ...

Hôte-Réseau

Permet l'émission de données via le réseau. Protocoles : Ethernet, ...

MISE EN OEUVRE

La pile réseau Linux



Dans le modèle TCP/IP, les processus tournant en mode utilisateur s'adressent à la couche transport (« pile TCP ») via les appels système de l'API socket.

LES SOCKETS

Mécanisme système permettant la communication entre tâches distantes

Offre une interface d'accès au service de transport

- en mode connecté (TCP)
- en mode non connecté (UDP)

+++ Outil disponible quelque soit l'environnement

-- Code lourd, répartition figée



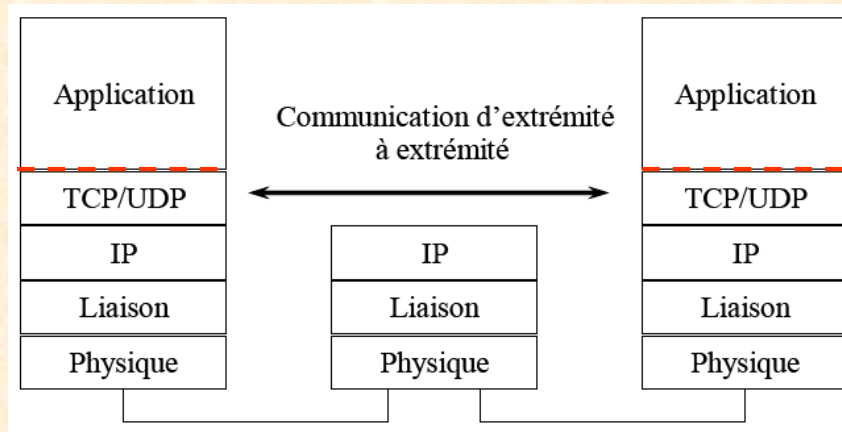
Les sockets sont des appels système qui offrent un point d'accès à la couche transport. Ils sont disponibles dans tous les systèmes d'exploitation.

Deux protocoles sont disponibles pour la couche transport : TCP et UDP.

TCP (Transmission Control Protocol) est un protocole de transport orienté flot d'octets. Il permet une communication fiable en mode connecté. On l'utilisera pour le transfert de grandes quantités de données à la fois (1 requête = n messages) ou nécessitant un accusé de réception.

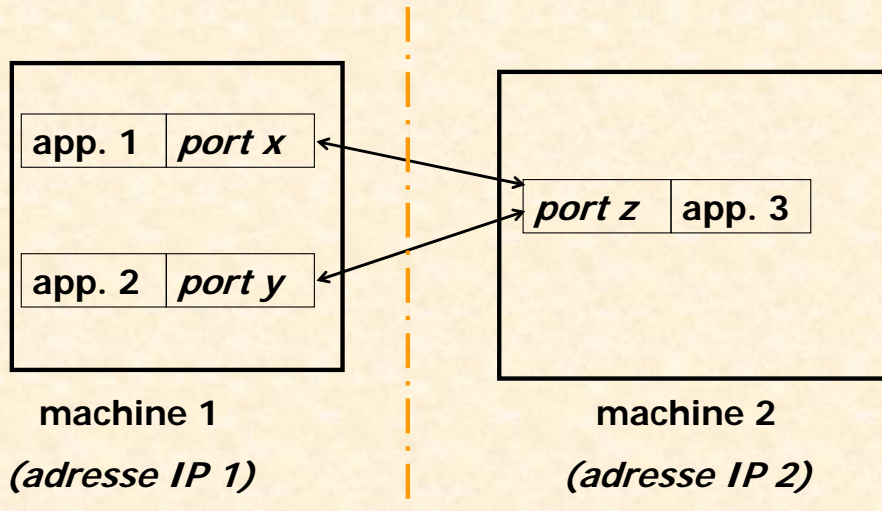
UDP (User Datagram Protocol) est un protocole de transport orienté transfert de messages. Il n'intègre pas de contrôle ni d'acquittement, la responsabilité du bon acheminement des données est à la charge de l'application. Il permet des communication point à point ou multipoint, hors connexion.

LES SOCKETS dans le modèle TCP/IP



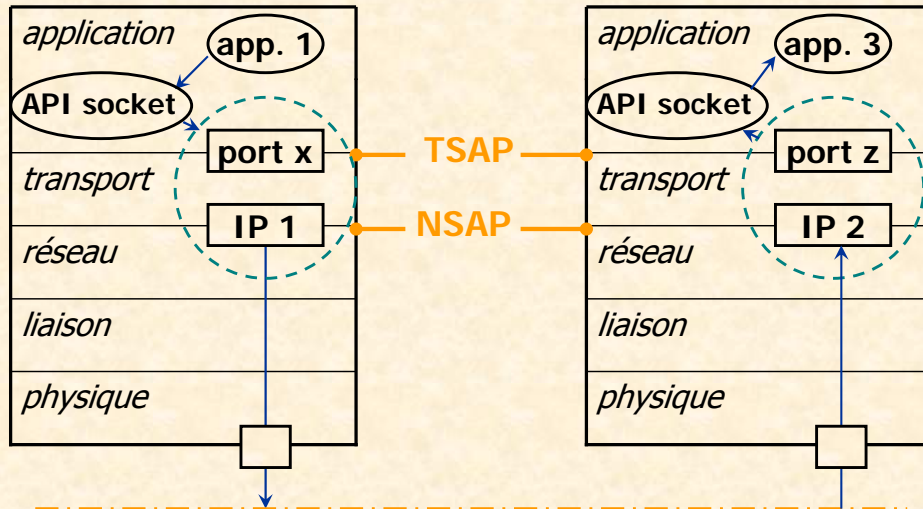
- - - - - : interface socket

Communication entre applications



Une application est identifiée sur le réseau par l'adresse IP de son interface réseau et par son numéro de port.

Communication entre 2 applications



TSAP : point d'accès à la couche transport (numéro de port)

NSAP : point d'accès à la couche réseau (adresse IP)

COMMUNICATION RESEAU : Les sockets

Socket = objet géré par le système d'exploitation et caractérisé par :

Adresse IP

identification de l'interface réseau (machine)

format : aa.bb.cc.dd (IPv4)

Numéro de port

identification de la tâche (ou de l'application)

format : entier (plage utilisateur > 1024)

Protocole de transport

TCP, UDP ou pas de protocole (raw)



L'adresse IP permet d'identifier la carte réseau.

La commande `ifconfig -a` permet de visualiser les interfaces réseau disponibles sur une machine.

Le numéro de port permet d'identifier l'application. Une même application peut utiliser plusieurs numéros de port.

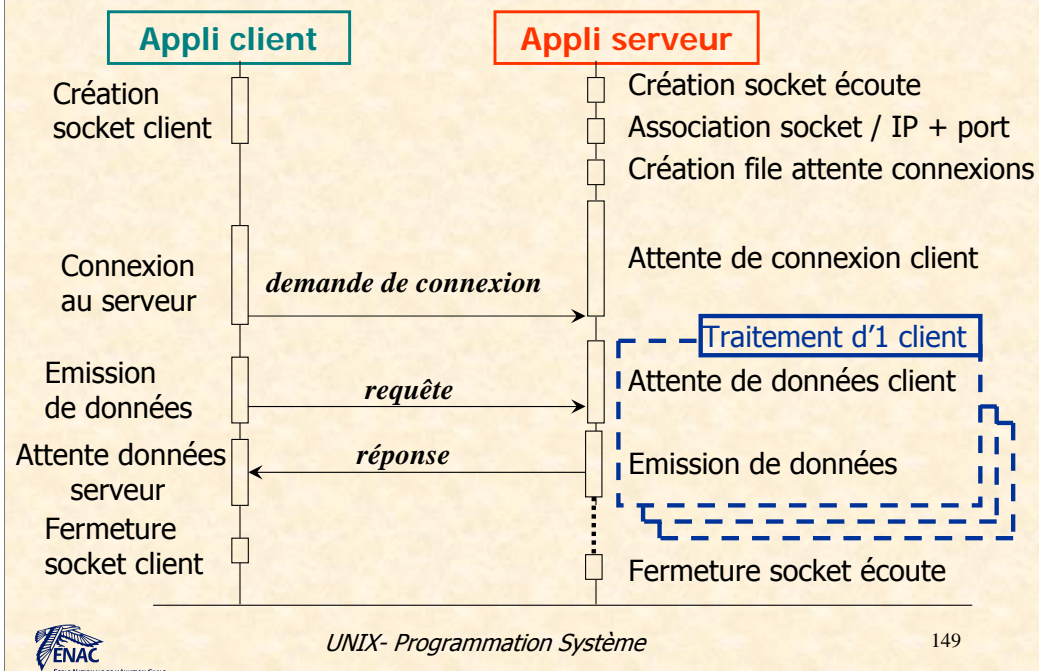
Sous Linux, la liste des ports réservés se trouve dans le fichier `/etc/services`.

Extrait de `/etc/services` :

```
ftp-data    20/tcp
ftp         21/tcp
ssh        22/tcp      # SSH Remote Login Protocol
ssh        22/udp      # SSH Remote Login Protocol
telnet     23/tcp
smtp       25/tcp      mail
www        80/tcp      http    # WorldWideWeb HTTP
www        80/udp      # HyperText Transfer Protocol
pop3       110/tcp      # POP version 3
pop3       110/udp
```

COMMUNICATION RESEAU : Les sockets

Mise en œuvre du modèle client-serveur (TCP)



Le modèle client-serveur définit une relation asymétrique entre 2 applications : un serveur et un client.

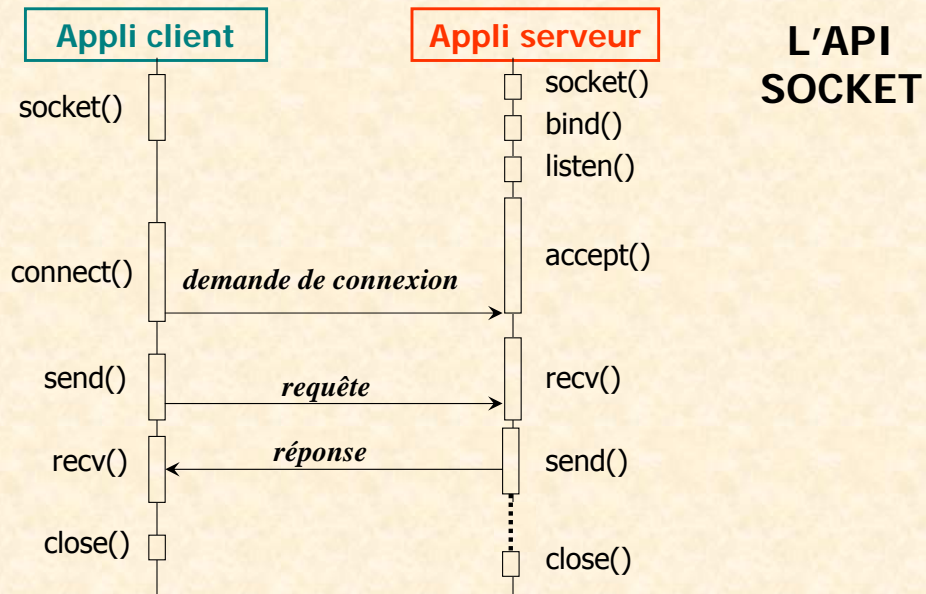
Le rôle du serveur est d'attendre les requêtes du client et de les traiter.

En mode connecté (protocole TCP), le client et le serveur doivent avoir établi une session avant de pouvoir échanger des informations. Le serveur attend donc une demande de connexion du client. Dès qu'il l'a reçue et acceptée, la session est établie et restera ouverte jusqu'à sa fermeture explicite ou jusqu'à la terminaison d'une des deux applications.

Un serveur peut traiter les clients successivement ou en parallèle (serveur multitâche).

COMMUNICATION RESEAU : Les sockets

Mise en œuvre du modèle client-serveur (TCP) :



FONCTIONS DE L'API SOCKET

Création d'une socket

```
int socket (  
    int af,           (1)  
    int type,         (2)  
    int protocol );   (3)
```

(1) Famille d'adresse (AF_INET : ipv4, AF_INET6 : ipv6, ...)

(2) Type de socket : SOCK_STREAM (TCP) , SOCK_DGRAM (UDP),
SOCK_RAW (accès direct à la couche réseau :
IP, ICMP)

(3) Protocole : 0 pour TCP et UDP

Valeur de retour : descripteur de la socket ou -1 (erreur)



La fonction **socket** () renvoie au programme appelant un descripteur créé dans la tables des descripteurs de fichier du processus.

```
int sock;  
sock = socket(AF_INET, SOCK_STREAM, 0);
```

FONCTIONS DE L'API SOCKET

Demande de connexion (client)

int **connect** (

```
    int s,                                (1)  
    const struct sockaddr *sin,          (2)  
    socklen_t lg );                     (3)
```

(1) Identifiant de la socket à connecter

(2) Adresse IP et numéro de port du serveur

(3) sizeof(struct sockaddr)

Valeur de retour : 0 ou -1 (erreur)

FONCTIONS DE L'API SOCKET

Adresse et port du serveur (client)

```
struct sockaddr_in {  
    short sin_family;           (1)  
    unsigned short sin_port;    (2)  
    struct in_addr sin_addr;    (3)  
    char sin_zero[8];          (4)  
};
```

- (1) La famille d'adresse (AF_INET : ipv4, AF_INET6 : ipv6)
- (2) Le numéro de port du serveur au format réseau
- (3) l'adresse IP du serveur au format réseau
- (4) 0



On notera que la structure `sockaddr_in` est une redéfinition du type `sockaddr` requis par `connect()` et que le type `in_addr` est défini ainsi :

```
struct in_addr {  
    uint32_t s_addr; /* adresse IP dans l'ordre des octets  
                     réseau */  
};
```

La représentation interne des nombres entiers peut varier selon l'architecture matérielle. En particulier, entre une machine de représentation little endian et une machine de représentation big endian, l'ordre des octets est inversé.

Afin de garantir l'interopérabilité des applications entre machines hétérogènes, la transmission des nombres entiers se fait sous un format normalisé : le format « réseau ».

Avant l'émission, tout nombre entier représenté sur plus d'un octet doit être converti du format de la machine locale (host) au format réseau.

De même, dès la réception, il devra être converti du format réseau au format de la machine réceptrice.

FONCTIONS DE L'API SOCKET

Fonctions de conversion

- Conversion nombre entier (format host → format réseau)

`uint16_t htons (uint16_t hostShort);`

`uint32_t htonl (uint32_t hostLong);`

- Conversion nombre entier (format réseau → format host)

`uint32_t ntohl (uint32_t netLong);`

`uint16_t ntohs (uint16_t netShort);`



Ces fonctions servent à convertir des entiers courts (suffixe **s**) ou longs (suffixe **l**) du format machine (**h**) au format réseau (**n**) et réciproquement.

En particulier, le numéro de port sera traduit au format réseau pour la demande de connexion.

```
int sock;
struct sockaddr_in sin;
...
sin.sin_port = htons(9999); /* port 9999 */
...
connect(sock, (struct sockaddr*)&sin, sizeof(sin));
```

FONCTIONS DE L'API SOCKET

Fonctions de conversion

- Conversion de l'adresse IP au format réseau

```
unsigned long inet_addr (  
    const char * adresseIP ;           (1)  
);
```

(1) L'adresse IP au format aa.bb.cc.dd

Valeur de retour L'adresse IP au format réseau ou -1 (erreur)

Remarque : -1 correspond à l'adresse IP 255.255.255.255 !
Il est préférable d'utiliser `inet_aton()`



```
int sock;  
struct sockaddr_in sin;  
...  
sin.sin_port = htons(9999);  
sin.sin_addr.s_addr = inet_addr("10.3.6.254");  
sin.sin_family = AF_INET;  
connect(sock, (struct sockaddr*)&sin, sizeof(sin));
```

FONCTIONS DE L'API SOCKET

Lecture dans une socket (client et serveur)

```
int recv (  
    int s,                (1)  
    char * buf,           (2)  
    int len,              (3)  
    int flags );          (4)
```

(1) Identifiant de la socket de communication

(2) Le buffer de réception

(3) Le nombre d'octets à lire

(4) 0 ou combinaison OU de flags (cf. doc API)

Valeur de retour Le nombre d'octets effectivement reçus



```
int sock;  
int nbRecus;  
char buf[255];  
...  
nbRecus = recv(sock, buf, sizeof(buf), 0);
```

FONCTIONS DE L'API SOCKET

Ecriture dans une socket (client et serveur)

int **send** (

int s,	(1)
char * buf,	(2)
int len,	(3)
int flags);	(4)

(1) Identifiant de la socket de communication

(2) Le buffer d'émission

(3) Le nombre d'octets à envoyer

(4) 0 ou combinaison OU de flags (cf. doc API)

Valeur de retour Le nombre d'octets effectivement envoyés



```
int sock;  
char message[255];  
...  
send(sock, message, strlen(message), 0);
```

FONCTIONS DE L'API SOCKET

Attachement de la socket d'écoute (serveur)

int bind (

```
    int s,                                (1)
    const struct sockaddr * name,         (2)
    socklen_t namelen );                 (3)
```

(1) Identifiant de la socket d'écoute

(2) L'adresse IP de l'interface (ou INADDR_ANY) et le port

(3) sizeof(struct sockaddr)

Valeur de retour 0 : OK -1 : erreur



La fonction `bind()` permet d'assigner une adresse IP et un numéro de port à une socket. La valeur `INADDR_ANY` signifie que la socket pourra utiliser toutes les interfaces locales.

L'appel à `bind()` est obligatoire pour caractériser la socket d'écoute du serveur.

L'attachement explicite de la socket du client n'est pas obligatoire, celle-ci se verra attribuer automatiquement une interface et un numéro de port disponibles.

```
int sock;
struct sockaddr_in sin;
...
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_family = AF_INET;
sin.sin_port = htons(9999);

bind(sock, (struct sockaddr *)&sin, sizeof(sin));
```

FONCTIONS DE L'API SOCKET

Mise en écoute (serveur, protocole TCP)

int listen (

int s, (1)

int backlog); (2)

(1) Identifiant de la socket d'écoute

(2) Le nombre maximum de connexions en file d'attente

Valeur de retour 0 : OK, -1 : erreur



La fonction `listen()` permet de désigner une socket comme étant une socket destinée à recevoir des demandes de connexions entrantes (socket d'écoute) et de spécifier la taille maximale de la file d'attente des demandes de connexions.

```
int sock;
```

```
...
```

```
listen(sock, 10);
```

FONCTIONS DE L'API SOCKET

Prise en compte de la connexion client (serveur)

```
int accept (  
    int s,                                (1)  
    struct sockaddr * infoSockCli ,      (2)  
    socklen_t * taille                    ); (3)
```

(1) Identifiant de la socket d'écoute

(2) Les informations du client

(3) Adresse d'une variable contenant sizeof(struct sockaddr)

Valeur de retour Id. de la socket de communication avec le client



La fonction **accept ()** permet d'attendre une demande de connexion sur une socket d'écoute bloquante. Si une demande est présente dans la file, elle crée une nouvelle socket qui permettra d'assurer les échanges avec le client et retourne son descripteur.

```
int sock, csock;  
int taille;  
struct sockaddr_in csin;  
...  
taille = sizeof(csin);  
csock = accept(sock, (struct sockaddr *)&csin, &taille);
```


FONCTIONS DE L'API SOCKET

Quelques autres fonctions ...

`ssize_t recvfrom (int sockfd, void *buf, size_t len, int flags,
struct sockaddr *src_addr, socklen_t *addrlen);`

Réception en mode connecté (si `src_addr=NULL`, identique à `recv`) ou non

`ssize_t sendto (int sockfd, const void *buf, size_t len, int flags,
const struct sockaddr *dest_addr, socklen_t addrlen);`

Emission en mode connecté (si `dest_addr=NULL`, identique à `send`) ou non

`int shutdown (int sockfd, int how);`

Fermeture d'une socket (Selon *how*, fermeture d'1 côté ou des 2)