**Programmation système en C sous Linux**

**Programmation système sous Linux**

**Sommaire**

* [Notion d'utilisateur](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#utilisateur)
* [Les processus](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#processus)
* [Les Fichiers](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#fichiers)
* [Les Threads](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#threads)
* [Les Signaux](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#signaux)
* [La Mémoire Partagée](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#memoirepartagee)
* [Les Fichiers mappés](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#fichiersmappes)
* [Les Mutex](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#mutex)
* [Les Sémaphores](http://www.blup.fr/programmation-systeme-en-c-sous-linux/#semaphores)

**Notion d'utilisateur**

**Introduction**

Un système d'exploitation gère les ressources matérielles et la confidentialité des utilisateurs.

Les systèmes Unix traditionnels gardent l'information sur les comptes utilisateurs, y compris les mots de passe cryptés, dans un fichier texte appelé*/etc/passwd*. Comme ce fichier est utilisé par beaucoup d'utilitaires, (comme *ls*) pour afficher les permissions des fichiers, ou encore pour associer le numéro d'identification de l'utilisateur (id #) avec son nom, le fichier doit pouvoir être lu par tout le monde. En conséquence de quoi, cela représente un risque pour la sécurité.

Un autre méthode pour stocker l'information sur les comptes utilisateurs, est le format de mot de passe caché. Tout comme la manière traditionnelle, l'information sur les comptes est stockée dans le fichier */etc/passwd* dans un format compatible. Cependant, le mot de passe est rangé comme un simple caractère “x” (ie. en fait non stocké dans ce fichier). Un second fichier,*/etc/shadow*, contient le mot de passe codé de même que toute autre information telle que les valeurs relatives à l'expiration du compte ou du mot de passe, etc. Le fichier */etc/shadow* file ne peut être lu que par le compte root et cela constitue donc un risque moins grand pour la sécurité.

En plus des mots de passe cachés, le fichier */etc/passwd* contient l'information relative au compte et ressemble à cela:

smithj:x:561:561:Joe Smith:/home/smithj:/bin/bash

Chaque champ dans une entrée de mot de passe est séparé par deux points :

- Nom d'utilisateur, jusqu'à 8 caractères (case-sensitive).
- Un “x” dans le champ mot de passe. Les mots de passe sont stockés dans le fichier */etc/shadow*.
- Numéro d'identification de l'utilisateur. Il est attribué par le script *adduser*. Unix utilise ce champ, plus le champ suivant du groupe, pour identifier quels fichiers appartiennent à l'utilisateur.
- Numéro d'identification du groupe. Habituellement, le numéro de groupe sera le même que le numéro d'utilisateur.
- Nom complet de l'utilisateur.
- Répertoire personnel de l'utilisateur. Habituellement, */home/username*.
- Shell de l'utilisateur, souvent fixé à */bin/bash* pour permettre l'accès au shell bash.

**Gestion des mots de passe et groupes en C**

Le fichier *pwd.h* contient les prototypes qui serviront à la manipulation en C :

# include <pwd.h>

struct passwd

 {

 char \*pw\_name; // nom d'utilisateur
 char \*pw\_passwd; // mot de passe
 uid\_t uid; // uid

 gid\_t gid; // gid

 char \*pw\_gecos; // nom réel de la connexion

 char \*pw\_dir; // répertoire de connexion

 char \*pw\_shell; // shell de connexion

 }

void setpwent();

// Permet d'ouvrir le fichier de mots de passe

struct passwd \* getpwent();

// Renvoie un pointeur sur une structure contenant les divers champs
// du fichier passwd. Au premier appel, renvoie le premier enregistrement
// puis les autres lors des appels itératifs.

void endpwent();

// ferme le fichier de mot de passe

Accès direct :

struct passwd \* getpwnam( char \*name );
// renvoie un pointeur sur une structure contenant les divers champs
// de l'enregistrement dans /etc/passwd correspondant au nom d'utilisateur.

struct passwd \* getpwuid( uid\_t uid );
//idem mais avec uid

Le fichier *grp.h* contient les prototypes qui serviront à la manipulation des groupes en C :

# include <grp.h>

struct group

 {

 char \*gr\_name; // nom du groupe

 char \*gr\_passwd; // mot de passe

 gid\_t \*gid; // gid
 char \*\*gr\_nam ;

 }

getgrnam(char \*nam);

// renvoie un pointeur sur structure contenant l'enregistrement

// issu de /etc/group pour le groupe correspondant au nom name

**Utilisateurs, groupes et filesystem**

Chaque système de fichiers tient à jour une table des descripteurs des fichiers qu'utilise le système d'exploitation pour accéder aux fichiers. Cette table se compose pour chaque fichier ou répertoire, d'une entrée appelée inode, repérée par un index appelé le numéro d'inode.

L'inode de chaque fichier contient l'*uid* et le *gid* du propriétaire de ce fichier, et un champ dont les bits définissent qui peut faire quoi avec ce fichier :
- le bit R comme Read: la lecture est autorisée
- le bit W comme Write: l'écriture sur le fichier et la possibilité de le supprimer
- le bit X comme eXecute: pour un fichier, il contrôle la possibilité de l'exécuter, pour un répertoire, il permet ou il interdit l'accès à ce répertoire.
De plus, 3 bits spéciaux permettent une utilisation plus subtile du fichier.

Exemples pour changer les droits d'un fichier grâce à la commande *chmod* :

*chmod 755 <fichier>*

utilisateur : 7 rwx l'utilisateur a tous les droits

groupe : 5 r x le groupe a les droits de lecture et exécution

autres : 5 r x les autres utilisateurs peuvent lire et exécuter

Plus simplement :

*chmod g+w <fichier>* ajoute le droit d'écriture au groupe

*chmod a+r <fichier>* permet à tout le monde de lire le fichier

Cibles : g = groupe

 u = utilisateur

 o = autres (others)

 a = tous (all)

Actions : + = ajouter un droit

 - = retirer un droit

Les droits sont r (lecture), x (exécution) , w (écriture)

**Utilisateurs et processsus**

Supposons que vous souhaitez exécuter un programme dont vous n'êtes pas le propriétaire. On peut se dire que si le programme appartient à Marcello Mastroianni, il aura les mêmes droits que ce dernier. Mais on peut aussi penser que puisque c'est Charlie Chaplin qui l'exécute, il aura les droits de ce dernier, après tout. En réalité, que se passe-t-il ?

Chaque fichier possède un bit *setuid* qui permet de résoudre le dilemme. Lorsqu'on exécute un programme dont le *setuid* est désarmé (=par défaut), il hérite l'UID de la tâche qui l'a lancé, par exemple le shell de Charlie. En revanche, lorsque ce bit est armé, alors le programme s'exécutera toujours avec l'UID de son propriétaire. Par exemple, lorsqu'un utilisateur utilise la commande *passwd* pour changer son mot de passe, il faut répercuter le changement dans le fichier */etc/passwd*, qu'aucun utilisateur n'a le droit d'écrire. Mais le programme *passwd* appartient au root et son *setuid* est mis, donc la modification peut se faire.

*chmod +s <fichier>* permet de mettre à 1 le bit setuid

Description détaillée :
*suid* : si le *suid* d'un executable est à 1, le processus s'exécute avec les droits uid de celui qui a créé le fichier.
*sgid* : si le *sgid* d'un executable est à 1, le processus associé s'execute avec les droits gid de celui qui a créé le fichier
*sticky bit* : si le sticky bit d'un executable est à 1 (positionné), il restera en mémoire même apres la fin de son execution pour pouvoir être relancé plus rapidement.

Repertoires :
Dans le cas des répertoires, si le sticky bit est à 1 et un des bits w est à 1 alors tout le monde a le droit de créer des fichiers.
Mais pour des raisons de securité, il n'est pas possible d'effacer un fichier sauf si une des conditions est vérifiée :
- l'utilisateur est le propriétaire du fichier
- l'utilisateur a la permission sur le fichier
- l'utilisateur est root
Si le *sgid* est à 1 pour un répertoire, un fichier créé dans ce répertoire prend le groupe du répertoire. Si le créateur fait partie de ce groupe même si ce n'est pas son groupe principal.

Dans un programme, il est possible de changer l'*uid* et le *gid* :

int setgid (gid\_t); // real gid
int setuid (uid\_t); // real uid
int setegid (gid\_t); // effective gid
int seteuid(uid\_t); // effective uid

Seul root a le droit de passer des paramètres quelconques. L'utilisateur a le droit de passer son propre *uid* et *gid*.

Exemple : *suexec* permet au serveur http Apache de s'exécuter avec un uid quelconque (ex: www). Si le script doit lire une base de données ou des fichiers, alors Apache change d'uid grâce à *suexec*.

httpd rwxr-xr-x (propriétaire : www)

suexec rws--x--x (propriétaire : root)

**Les processus**

**Introduction**

Un processus représente l'ensemble programme en cours d'exécution + données de ce programme. Un utilisateur ne peut exécuter qu'un nombre limité de processus.

Le système d'exploitation attribue un numéro unique (*pid*) à chaque processus et conserve en mémoire une arborescence des processus, consultable grâce aux commandes *pstree* (sous Linux) et *ptree* (sous Solaris). La racine de cette arborescence est le processus *init* qui ne se termine jamais, à l'exception de Solaris où *sched* possède le *pid* 0, qui n'a pas de fils et est son propre père.
Les processus sont groupés (un processus est un groupe à lui tout seul).

Quand un processus est tué, son père en est normalement averti. Toutefois, il peut arriver que le père ne soit pas averti de la disparition de l'un de ses fils (notamment dans le cas où ce dernier a été victime d'un brutal 'kill -9'). Le fils est donc toujours référencé alors qu'il n'existe plus en réalité. Le processus defunct (ancien processus fils) ne disparaîtra donc qu'avec la mort du père.

Prenons par exemple la fameuse commande *kill -9 -1*. Elle envoie le signal 9 (SIGKILL) à tous les processus, les tuant sans condition. Par défaut, si aucun numéro de signal n'est précisé, le signal 15 (SIGTERM) est envoyé. Quelle différence entre les signaux 9 et 15 ? Le signal 9 tuera à coup sûr et immédiatement le processus (sans lui demander son avis en quelque sorte), alors que le signal 15 donne l'ordre au processus de se terminer. Le signal 9 est radical : il tue tout et ne laisse même pas le temps au système de prendre bonne note du déloguage.

Le signal 15 fait tout dans les règles, c'est-à-dire qu'il donne l'ordre au processus de s'auto-terminer, ainsi qu'à ses fils (les processus qu'il a lui-même lancé). Ainsi, il se peut que certains fils indignes fassent la forte tête : des processus récalcitrants peuvent survivre.

$ ps aux | grep user | sort -n +4

*sort* est le fils de *grep*, qui est le fils de *ps*, lui même le fils du shell : ils constituent un groupe

La commande *ps* affiche la liste des processus.
*-e* : tous les processus
*-d* : tous sauf leaders de sessions
*-f* : full listing (pid, ppid)
*-l* : long listing (pid, ppid, uid, priorité)
*-j* : (pid, pgid et sid)

**Image mémoire**

La mémoire virtuelle est découpée en pages non contigües qui peuvent être partagées.

La taille mémoire d'un processus est composée de 2 grands éléments : la taille statique et la taille dynamique. La taille statique est liée à l'éxécutable au moment de son lancement, elle comprend :
- la taille du code lié statiquement
- les données statiques
- la pile initiale
La taille dynamique est plus complexe à définir, toutefois, on peut énumérer les grandes composantes suivantes :
- la taille du code lié dynamiquement
- les augmentations de taille de pile si celle ci est dynamique
- les données allouées dynamiquement sur le tas



**Structures de données du système**

Le système conserve la trace de tous les processus. L'arborescence comporte des pointeurs sur les groupes, les sessions, ou encore vers le père et les fils du processus, mais aussi les frères, l'identité de l'utilisateur (réelle ou effective), l'état du processus et l'allocation de la mémoire.

sched.h : struct task\_struct

proc.h : struct proc

**Manipulation**

On ne crée jamais un processus, on le duplique avec l'appel système *fork()*. Après un *fork*, 2 processus identiques existent : tout est dupliqué, y compris les variables globales. Deux cas se présentent : le processus créateur fait son travail, ou le processus créé fait le sien.

#include <sys/types.h>
#include <unistd.h>
pid\_t pid;

switch(pid=fork()) {

 case -1: ... break; // erreur

 case 0: ... break; // le fils exécute cette partie

 default: ... break; // le père exécute cette partie

}

Remarque : les pid sont strictement positifs.

Le processus père reçoit l'id de chaque fils. Un processus n'a qu'un seul père. La fonction *getppid()* renvoie à un processus le pid de son père. Le fils et le père s'exécutent indépendamment après *fork*. Si le fils se termine, il restera zombie jusqu'à ce qu'un autre processus le remarque.

exit(int) // ne retourne jamais

abort() // crée un fichier core avant d'appeller exit

L'entier passé à *exit()* transmet de l'information au processus qui attend la fin du processus courant. Par convention, *exit(0)* correspond à une exécution normale sans erreur. Si un processus n'appelle pas *exit*, la valeur renvoyée est aléatoire, sauf si un *return* est effectué dans la fonction *main()*.

**Attendre un processus**

Le shell effectue un *fork* pour exécuter une commande, puis attend la fin de ce processus.
En C, la fonction *wait()* attend et retourne la valeur passée à *exit()* par le fils.

pid\_t wait(int \*);
// attend la fin de n'importe quel fils

// int \* : valeur de retour passée à exit par le fils

pid\_t waitpid(pid\_t, int \*, int);
// attend la fin d'un processus donné

// le 3e paramètre permet de passer des flags

*wait* est bloquante mais retourne immédiatement si le fils a terminé son exécution ou est devenu zombie.
*waitpid* ne bloque pas avec l'option WNOHANG

Remarque : Le processus père tourne sans arrêt. Quand il lance un fils, il récupère un int status :

void verifier(int status) {

 if(WIFEXITED(status)) {

 // le fils a appelé exit

 message("le fils a retourné %d",WEXITSTATUS(status));

 }

}

 **Changer le programme d'un processus**

*fork*, *wait* et *exit* ne permettent pas d'exécuter de nouveaux programmes.
Les appels système *exec\** permettent de remplacer l'image courante du processus par un autre programme.

int execue(char \*path, char \*argv[], char \*envp[]);

 // path : chemin vers exécutable

 // argv : arguments, se terminant par NULL

 // envp : variables système (forme var=val)

// exemple :

 char \*argv[] = {"mv","prog.c","prog.d",NULL};

 char \*envp[] = {"PATH=/bin/usr/bin",NULL};

 execue("/bin/usr/mv",argv,envp);

exec\* ne retourne jamais. Il en existe plusieurs variantes (execlp, execue...) pour lesquelles il faut combiner les lettres suivantes :
*v* : si des tableaux sont passés en argument
*l* : si des listes sont passées
*p* : si seul le nom de fichier est donné
*e* : si l'env est passé en argument

 **Diagnostic**

*wait*, *fork*, *exec* retournent -1 en cas d'erreur et positionnent *errno* (variable globale - *#include <errno.h>*)
*perror()* consulte errno et affiche le message d'erreur
*strerror()* renvoie une chaîne de caractères descriptive

 **Shell et kill**

Stopper tous les processsus : *$ kill -9 -1* envoie le signal de fin (9) à tous les programmes de l'utilisateur
Stopper un processus : *$ kill -STOP <pid*>
Reprendre : *$ kill -CONT <pid>*

**Exemple**

main(int argc, char \*\*argv) {
 int i;
 int nproc=0; // nombre de fils
 int link=1; // qu'est-ce qu'on link ?
 for(i=1;i<argc;i++) {
 pid\_t pid=fork();
 switch(pid) {

 case -1: perror("Erreur"); break;
 case 0: execlp("gcc","gcc","-c",argv[i]); break;
 }
 }
 for(i=1;i<argc;i++) {
 int status;
 wait(&status);
 link=link && WIFEXITED(status) && (WEXITSTATUS(status)==0);
 }
 if(link) {
 char \*\*args;
 args=(char\*\*)calloc(argc+1,sizeof(char\*));
 args[0]="gcc";
 for(i=1;i<argc;i++) args[i]=f(argv[i]);
 // la fonction f remplace .c par .o
 args[argc]=NULL;
 execvp(args[0],args);
 } else {
 printf("Erreur : link");
 }
}

**Les Fichiers**

**Primitives**

Appels système (niveau 2) - *fcntl.h* :
open close dup read write link unlink mknod start
Routines (niveau 3) - *stdio.h* :
fopen fclose freopen fread fwrite fscanf fprintf fputs fgets

**Différences**

Au niveau 2, les entrées/sorties sont effectuées immédiatement.
Au niveau 3, les données sont mises en buffer.

int f;

char tab[256];
if((f=open("fich.txt",O\_CREAT|O\_TRUNC|O\_WRONLY))==0) {
 write(f,tab,15);
 close(f);
}

FILE \*f;
char tab[256];
if((fd=fopen("fich.txt","w"))!=NULL) {
 fwrite(tab,sizeof(char),15,fd);
 fclose(fd);
}

fflush() est automatique à chaque \n (retour à la ligne)
Chaque processus a 3 flux ouverts : stdin, stdout et stderr.

**Descripteurs de fichiers**

Le système maintient une table des fichiers ouverts à 3 niveaux : table de fichiers processus, table de fichiers système et table des inodes. Un appel à*fopen* produit une entrée dans chacune des 3 tables. A la création d'un processus, les entrées du père sont dupliquées et à sa fin elles sont supprimées.

Niveau 3 :
if(freopen("dump.txt","w",stdout)==NULL) {

 perror("...");
} else {
 printf("stoo\n");
}

Niveau 2 :
dup(int) // duplique le descripteur et produit une
 // nouvelle entrée dans la table des fichier
 // du processus mais pas dans celle du système
close(0); // ferme l'entrée standard
dup(); // garantit qu'il va utiliser le premier descripteur libre

*dup2(int a,int b)* fait *close(...); dup(...); close(...);*
a : descripteur à dupliquer
b : descripteur à affecter

**Fork et fichiers**

Le processus fils est une copie du père : il contient les descripteurs du père et peut y accéder.

f=open("f.txt",...);

if(fork()==0) {

 char \*s="fils";

 write(f,s,4);

} else {

 char \*s="pere";

 write(f,s,4);

}

close(f);

Résultat :

perefils (ou) filspere

**Redirection E/S**

Il est possible d'effectuer des redirections d'entrées/sorties.

Exemple : *ls >liste.txt*
n'affichera pas la liste des fichiers à l'écran mais l'écrira dans le fichier liste.txt. En C on peut écrire le code équivalent :

if((pid=fork())==0) {

 int f=open("liste",...);

 close(1);

 dup(f);

 close(f);

 execlp("ls","ls","-l",NULL);
}

**Pipes**

Les pipes permettent la communication entre processus. En effet, lorsqu'un fichier est ouvert en écriture ou en lecture, il est impossible de changer cet état. Un programme doit donc utiliser une autre méthode pour que différents processus puissent communiquer.

int pipe(int fd[2]);

L'appel initialise les deux descripteurs, un pour chaque extrémité : lecture (*fd[0]*) et écriture (*fd[1]*). *pipe* renvoie -1 si une erreur survient.
Un pipe (fifo) est un inode spécial car ce n'est pas un fichier physique et son buffer est de la taille d'une page mémoire (1 Ko sous Linux).

Exemple : grep murf | toto

main(int argc, char \*\*argv) {

 pid\_t pid;

 int p[2];
 int status1, status0;
 pipe(p);
 if((pid=fork())==0) {
 // 1er fils : grep
 if(argc>1) {
 int f=open(argv[1],O\_RDONLY);
 close(0);
 dup(f);
 close(f);
 }
 close(1);
 dup(p[1]); // côté écriture du pipe
 close(p[1]);
 close(p[0]); // on ne lit pas le pipe
 execlp("grep","grep","toto",NULL);
 }
 if((pid=fork())==0) {
 // 2e fils : sort
 if(argc>2) {
 int f=fopen(argv[2],O\_CREAT|O\_TRUNC|O\_WRONLY);
 close(1);
 dup(f);
 close(f);
 }
 close(0);
 dup(p[0]);
 close(p[0]);
 close(p[1]);

 execlp("sort","sort",NULL);
 }
 close(p[0]);
 close(p[1]);
 wait(&status0);
 wait(&status1);
 exit((WIFEXITED(status1) && WEXITSTATUS(status1)==0
 && WIFEXITED(status0) && WEXITSTATUS(status1)==0)
 ? 0 : -1);
}

**Les tubes nommés**

Pour que deux processus puissent communiquer, il faut que ces deux processus aient un ancêtre commun.
Si cette condition n'est pas remplie, alors une solution alternative est l'utilisation de tubes nommés qui remplissent la même fonction que les pipes mais sont représentés sur le système de fichiers et dont l'accès est soumis aux droits classiques. De plus, leur taille peut être ajustée.

**Création**

- par la commande système *mkfifo <nom>*
- dans un programme par l'appel système *mknod()*
L'ouverture, la fermeture, la lecture et l'écriture sont gérées comme pour un fichier normal.

$ mkfifo myfifo

lecteur.c : f=open("myfifo",O\_RDONLY);
 while(read(f,&c,1)) write(1,&c,1);

ecrivain.c: f=open("myfifo",O\_WRONLY);
 while(read(0,&c,1)) write(f,&c,1);

Remaque : les lecteurs reçoivent EOF lorsque tous les écrivains ont fermé le tube (read retourne 0). S'il y a plusieurs lecteurs, ils se partagent les données.

**Les Threads**

**Introduction**

Les threads permettent une exécution parallèle au sein d'un processus avec les avantages suivants :
- un seul espace d'adressage
- une communication rapide
- des segments mémoire visibles depuis tous les threads ("données globales") d'où une communication facile

**Ordonnancement**

* **processus**
· les threads sont ordonnancés au niveau utilisateur
· les processus sont ordonnancés par le système
+ avantage : chaque processus dispose équitablement des ressources
* **lwp** (lightweight processus) :
· les threads sont ordonnancés au niveau utilisateur sur des lwp qui sont ordonnancés par le système (support noyau nécessaire)
· les threads sont multiplexés sur les lwp disponibles
+ avantage : l'occupation CPU est liée au nombre de lwp/processus quel que soit le nombre de threads
- problème : le noyau doit être adapté (utilisé par Solaris - Posix)
* **thread** (linux thread) :
· les threads sont directement ordonnancés par le système : un thread "devient" un processus avec un espace d'adressage virtuel "partagé" avec les autres threads
- problème : un processus avec 10 threads profite 10 fois plus des ressources qu'un processus à 1 thread

**Principes**

Un thread exécute une fonction. Tout programme dispose d'un "thread principal" qui exécute *main()*.

Données :
- de la pile : visibles dans le thread, éventuellement accessibles par les autres
- globales : visibles par tous les threads (à condition de disposer d'un pointeur)
- allouées : visibles par tous les threads (à condition de disposer d'un pointeur)
- privées : accessibles par une clé, visibles uniquement par le créateur

Les attributs permettent de contrôler la terminaison et l'ordonnancement d'un thread.

**Appels système**

**Création**

#include <pthread.h> // compilation avec -lthread
int pthread\_create(pthread\_t \*id, pthread\_attr\_t \*attr,
 void \*(\*f)(void\*), void \*arg);

crée un thread pour le processus et y exécute la fonction f(arg). Si l'*id* est non nul, l'*id* du thread y est stocké. Si attr est NULL, les attributs par défaut sont pris en compte lors de la création, sinon il y a possibilité de paramétrer le nouveau thread.

void pthread\_exit(void \*ret);

termine le thread appelant et retourne le pointeur *ret*.

Remarques :

* *pthread\_create* est assimilable à *fork* + *exec* et *pthread\_exit* à *exit*, mais au niveau thread
* Un processus contient toujours au moins un thread main() (implicite), les autres threads sont créés par *pthread\_create* (explicite).
* Un thread se termine explicitement avec *pthread\_exit* et implicitement en terminant la fonction.
* Si *main()* se termine, le processus s'arrête et tous les threads disparaissent.
* *pthread\_exit* dans le processus principal attend la fin des autres threads, toujours à la fin du *main()*.

int pthread\_join(pthread\_t id, void \*\*status);

attend la fin d'un thread donné et permet de récupérer ce qui a été passé à*pthread\_exit* si status est non NULL.
*pthread\_join* est assimilable à waitpid, mais au niveau thread.

Exemple :

pthread\_t tid;

char \*filename[FILENAME\_MAX];

struct resultat \*res;

...

pthread\_create(&tid,NULL,analyse,filename);

...

pthread\_join(tid,res);

**Attributs**

struct pthread\_attr\_t { ... };

int pthread\_attr\_init(pthread\_attr\_t \*);
 // initialise les attributs et les valeurs par défaut
int pthread\_attr\_set<attrname>(pthread\_attr\_t\*,int);
 // modifie un attribut
int pthread\_attr\_get<attrname>(pthread\_attr\_t\*,int);
 // récupère la valeur d'un attribut
int pthread\_attr\_destroy(pthread\_attr\_t\*);

Attributs disponibles :
 ° detachstate :
 PTHREAD\_CREATE\_JOINABLE // peut être synchronisé par pthread\_join
 PTHREAD\_CREATE\_DETACHED // tout est désalloué à la fin

Exemple : pthread\_attr\_t st;
 pthread\_attr\_init(&st);
 pthread\_attr\_setdetachstate(&st,PTHREAD\_CREATE\_DETACHED);

 ° schedpolicy :
 SCHED\_OTHER // defaut
 SCHED\_FIFO // temps réel fifo (root seulement)
 SCHED\_RR // temps réel (root seulement)

 ° schedparam
 0 à 99 // priorité d'ordonnancement sans effet avec OTHER

 ° inheritedsched
 PTHREAD\_EXPLICIT\_SCHED
 // paramètre d'ordonnancement donné par schedpolicy et schedparam
 PTHREAD\_INHERIT\_SCHED // hérité du thread parent

 ° scope
 PTHREAD\_SCOPE\_SYSTEM
 // tous les threads sont en compétition avec les autres processus
 PTHREAD\_SCOPE\_PROCESS
 // les threads ne sont en compétition qu'avec les autres threads

Les attributs ne sont consultés qu'à la création. Néanmoins un changement après la création est possible :

int pthread\_detach(pthread\_t \*)

int pthread\_setschedparam(pthread\_t \*, int, struct sched\_param \*);

int pthread\_getschedparam(pthread\_t \*, int \*, struct sched\_param \*);

**Données spécifiques**

Les TSD (Thread Specific Data) sont des variables globales ou statiques ayant des valeurs différentes dans les différents threads (tableau de void\*).
Clés : indices des TSD (communes à tous les threads).

int pthread\_key\_create(pthread\_key\_t \*key, void (\*endfunc)(void\*));
 // crée une clé
int pthread\_key\_delete(pthread\_key\_t key);
 // supprime une clé

La fonction de destruction *endfunc* est appelée à la fin du thread.

int pthread\_setspecific(pthread\_key\_t \*key, void \*pointer);
 // associe une nouvelle valeur à la clé

void \*pthread\_getspecific(pthread\_key\_t key);
 // renvoie la valeur de la clé

**Arrêts autoritaires**

int pthread\_cancel(pthread\_t);

 // ~ kill : un thread en arrête un autre
 // à condition que cet autre soit d'accord

int pthread\_setcancelstate(int new, int \*old);

 // état : PTHREAD\_CANCEL\_ENABLE (défaut)

 PTHREAD\_CANCEL\_DISABLE

int pthread\_setcanceltype(int new, int \*old);

 // type : PTHREAD\_CANCEL\_ASYNCHRONOUS (arrêt immédiat)

 PTHREAD\_CANCEL\_DEFERRED (défaut)

**Gestion de fin de thread**

Une pile de fonctions pouvant être appelées lors de pthread\_exit

void pthread\_cleanup\_push(void(\*func)(void\*),void \*arg);
void pthread\_cleanup\_pop(int execute);

*Push* : empile une fonction, *pop* : dépile

**Exemple**

pthread\_t tid;

char filename[FILENAME\_MAX];

struct resultat \*res;

...

pthread\_create(&tid,NULL,analyse,filename);
// analyse : fonction, filename : argument
...

pthread\_join(tid,&res);

void \*analyse(void \*n) {

 char \*name=(char\*)n;

 char \*buffer;

 struct resultat \*r=(struct resultat\*)malloc(sizeof(struct resultat));

 ...

 buffer=malloc(4096\*sizeof(char));

 pthread\_cleanup\_push(free,buffer);

 ...

 pthread\_cleanup\_pop(1);

 // dépile et exécute la fonction qui avait été empilée
 pthread\_exit(1);

 // ou n'exécute pas la fonction :

 // pthread\_cleanup\_pop(0);

 // free(buffer);

 // pthread\_exit(1);
}

**Les Signaux**

**Introduction**

Un processus peut envoyer des signaux à d'autres processus.
Le destinataire réagit immédiatement, soit en s'interrompant, soit en traitant le signal, ou éventuellement en reprenant son cours.

Un signal ne transporte que son numéro :
*$ kill -9 1234*   envoie le signal SIGKILL (9) au *pid* 1234
*$ kill -STOP 1234*   (arrêter) *$ kill -CONT 1234*   (continuer)

**Appels système**

#include <signal.h>
int kill(pid\_t pid, int s);

Exemples :
Lorsqu'un processus fils se termine, il envoie un signal SIGCHILD à son père.
Lorsqu'un processus écrit dans un *pipe* sans lecteur, il reçoit SIGPIPE.
Certaines applications envoient un signal au processus dont le terminal est le terminal de contrôle.
Ctrl+C = SIGINT
Ctrl+\ = SIGQUIT

**Réaction d'un processus**

Un processus est paramétrable pour traiter les signaux. Plusieurs comportements sont paramétrables :

- ignorer : pas de réaction
- comportement par défaut : termine et produit un *core dumped*- appeler une fonction puis reprendre normalement
   ce comportement peut être mis en place par l'utilisation d'un gestionnaire de signal (*handler*) :

 void (\*signal(int signum,void(\*sighandler)(int)))(int);
 typedef void (\*SIGHANDLER)(int);
 SIGHANDLER signal(int,SIGHANDLER);

L'appel système *signal()* installe un nouveau gestionnaire de signal pour le signal possédant le numéro *signum*. Le gestionnaire est défini pour *sighandler*qui peut être soit une fonction définie par l'utilisateur, soit SIG\_IGN pour ignorer, soit SIG\_DFL (défaut). Seule exception, les comportements de SIGKILL et SIGSTOP ne peuvent être redéfinis ou ignorés.
La fonction *signal()* renvoie la valeur précédente du gestionnaire de signal, ou SIG\_ERR s'il y a erreur.

**Exemple**

main() {

 pid\_t pid;
 void (\*prec)(int);
 switch(pid=fork()) {
 case 0: // fils
 break;
 case 1: // père
 signal(SIGUSR1,prec);
 // le père n'a pas besoin de ce signal
 kill(pid,SIGUSR1);
 break;
 }
}

void handler(int nsig) {
 signal(nsig,handler);
 printf("le processus %d a recu %d\n",getpid(),nsig);
}

**Blocage**

Un processus peut bloquer la réception d'un signal grâce à *sigprocmask()* : le signal est reçu mais ne sera pas traité tant qu'il est bloqué.

**Structures**

Chaque entrée de la table des processus comporte pour chaque signal
- 1 bit indiquant si le signal a été reçu et reste à traiter
- 1 bit indiquant si le signal a été bloqué
- 1 structure *sigaction* indiquant le comportement à adopter (ignorer, défaut,*handler*) et diverses informations concernant le traitement.

**Remarques**

* *kill(pid, sig)* permet d'envoyer le même signal à un groupe de processus
- si pid > 0 alors le signal est envoyé au processus unique de numéro *pid*
- si pid = 0 alors le signal est envoyé à tous les processus du groupe du processus courant
- si pid = -1 alors le signal est envoyé à tous les processus du groupe sauf le premier
* après *fork()* le processus fils a un comportement identique à celui de son père vis à vis des signaux
* après *exec()*, les signaux ignorés continuent à l'être, les autres reprennent leur comportement par défaut, car les *handlers*sont remplacés.
* la primitive *alarm()* permet de programmer l'envoi du signal SIGARLM après un délai donné
* la primitive *pause()* bloque le processus appelant jusqu'à réception d'un signal

**La Mémoire Partagée**

**Introduction**

Un processus s'exécute dans un espace d'adressage (virtuel) de 2 ou 4 Go. Toutes les adresses utilisées n'ont de sens que dans ce processus. Ce dernier n'occupe réellement que des tranches de cet espace car certaines adresses sont invalides. Le système maintient pour chaque processus une structure de données décrivant l'occupation de la mémoire (espace d'adressage virtuel).

Il est possible d'allouer dynamiquement des places libres de cet espace et d'y placer un morceau de mémoire, un fichier, etc... La mémoire est découpée en pages et la correspondance mémoire virtuelle -> mémoire physique est faite par le matériel (MMU : Memory Managing Unit) avec une table de pages.

Conséquence importante : La même page physique peut apparaître dans l'espace virtuel de plusieurs processus à des adresses éventuellement différentes.

**Segments de mémoire partagée**

Un segment de mémorie partagée est un segment de mémoire du système apparaissant dans l'espace d'adressage d'un ou de plusieurs processus. L'identification est faite à deux niveaux :
- niveau utilisateur : une clé numérique par segment
- niveau système : un identificateur entier
La clé doit être connue de plusieurs processus; elle apparaît généralement dans un fichier de configuration.
Un segment de mémoire partagée est comme un fichier, il dispose de l'*id* du propriétaire et de droits d'accès.

**Création**

#include <sys/shm.h>

#include <sys/ipc.h>

int shmget(key\_t key, int size, int flag);

*shmget()* retourne un *id* entier utilisé par les autres appels
*size* est la taille du segment à allouer, arrondi à un multiple de la taille d'une page
*flag* est un OR logique admettant : IPC\_CREAT (création d'un shm), 12 bits de poids faible (permissions), IPC\_EXCL (échec si le segment existe déjà). Exemple : IPC\_CREAT | 0666 correspond à un accès en lecture et écriture pour tout le monde.

**Identification sans création**

Pour identifier un segment, il suffit d'utiliser *shmget()* sans le flag IPC\_CREAT. Dans les deux cas, le segment est créé, on récupère l'*id* système mais il n'est pas installé dans la mémoire virtuelle.

**Installation**

char \*shmat(int id, char \*ad, int flag);

*shmat()* attache le segment identifié par *id* au segment de données du processus appelant et retourne l'adresse où est placée le segment, ou (char\*) -1 en cas d'erreur.
*ad*est l'adresse à laquelle installer le segment (cette adresse doit être libre ou nulle si le système doit choisir)
*flag*est nul ou vaut SHM\_RDONLY si on veut imposer un accès en lecture seule

**Désinstallation**

int \*shmdt(char \*ad);

*shmdt()* détache le segment à l'adresse *ad* et renvoie 0 en cas de succès ou 1 en cas d'échec. Toute tentative d'accès ultérieure provoque SIGSEGV. La carte des processus et la table des pages sont mises à jour.

**Destruction**

On ne peut pas détruire instantanément un segment car d'autres processus sont encore susceptibles de l'utiliser.

int shmctl(int id, int cmd, struct shmid\_ds \*buf);

*shmctl()* marque le segment identifié par *id*comme étant "à détruire". L'entier*cmd*représente la commande à exécuter, dont les valeurs peuvent être :
- IPC\_STAT : permet de récupérer une description dans *buf*
- IPC\_SET : modifie les permissions d'accès
- IPC\_RMID : détruit ou marque à détruire, dans ce cas *buf* peut être NULL
Le shm sera détruit lorsque tous les processus l'auront détaché. Le créateur, le propriétaire et le super-utilisateur peuvent marquer le segment à détruire.

**Exemple**

key\_t cle;

pid\_t pid;

int id;

char \*ad;

*ecrit.c :*
if(argc!=4) {

 fprintf(stderr,"Usage %s <cle> <pid> <message>\",argv[0]);
 exit(1);
}

sscanf(argv[1],"%d",&cle);
sscanf(argv[2],"%d",&pid);
if((id=shmget(cle,1024,IPC\_CREAT|666))==1) {
 perror("shmget");
 exit(1);
}
if((ad=shmat(id,NULL,0))==(char\*)-1) {
 perror("shmat");
 exit(1);
}
strcat(ad,"|msg=");
strcat(ad,argv[3]);
if(pid!=0) kill(pid,SIGUSR1);

*lecteur.c :*
if(argc!=2) {
 fprintf(stderr,"Usage %s <cle>\n",argv[0]);
 exit(1);
}
sscanf(argv[1],"%d",&cle);
id=shmget(cle,1024,IPC\_CREAT|666); // +test
ad=shmat(id,NULL,SHM\_RDONLY); // +test
signal(SIGUSR1,handler);
while(1) {
 pause(); // if(sig==SIGUSR1)
 printf("%s",ad);
}

Si \*ad=="\0" alors il se produit une *segmentation fault*.

**Remarques**

Un segment peut être attaché plusieurs fois à un même processus (ex : en lecture/écriture et en lecture seule).
Le segment est une zone de mémoire "brute", mais on peut y accéder de manière structurée

struct s { struct s \*ps;

 int i; ps=(struct s\*)shmat(...);

 double d; ps[15].d=3.5;

}

On ne peut forcer *malloc()* à allouer dans le segment.
Après *fork()*, le fils attache les mêmes segments que son père.
Après *exec()* ou *exit()*, les segments sont détachés.
Les conflits d'accès ne sont pas gérés, d'où la nécessité d'utiliser des sémaphores.

**Les Fichiers mappés**

**Introduction**

Il est parfois intéressant, plutôt que de travailler directement sur le contenu d'un fichier, d'en projeter une image en mémoire, sur laquelle on oeuvre ensuite comme avec des variables normales. Cette projection permet de manipuler le fichier beaucoup plus rapidement et simplement qu'avec de véritables lectures et écritures sur le disque.

**Mmap**

**Création**

#include <sys/mman.h>

void \*mmap(void \*a, size\_t l, int p, int flags, int fd, off\_t offset);

*a* : adresse à laquelle on souhaite placer le contenu du fichier, ou NULL pour laisser le noyau décider
*l* : taille de la zone (octets)
*p* : mode de protection (OU binaire)
PROT\_EXEC : on peut exécuter du code dans la zone (assimilé à PROT\_READ sur x86)
PROT\_READ : on peut lire le contenu de la zone mémoire
PROT\_WRITE : on peut écrire dans la zone mémoire; le noyau synchronisera le fichier par la suite.
*flags* : paramètres de la projection
MAP\_MIXED : n'utiliser que l'adresse indiquée (déconseillé)
MAP\_SHARED : tous les processus mappant le fichier partagent les mêmes pages physiques
MAP\_PRIVATE : projection privée
*fd* : descripteur de fichier (qui doit être ouvert)
*offset* : déplacement initial dans le fichier
*mmap()* renvoie MAP\_FAILED en cas d'erreur, ou l'adresse de la projection en cas de succès.

Le contenu du fichier apparaît comme une zone de mémoire gérée par le système, accessible aléatoirement et dont les pages sont chargées et swappées à la demande. Si la projection est partagée, toute modification de la zone de mémoire est visible immédiatement aux autres processus.

**Remarques**

Le code d'un processus est projeté en mémoire et chargé uniquement lorsqu'il est utilisé.
L'appel système *int munmap(void \*a, size\_t l)* supprime la zone de mémoire désignée de l'espace d'adressage.
Si le contenu n'est pas sauvegardé, il est perdu.

int msync(void \*a, size\_t l, int flags);

Permet la sauvegarde sur le disque du contenu du fichier mappé.
*flags*: MS\_ASYNC : la sauvegarde est prévue, retourne immédiatement
MS\_SYNC : la sauvegarde est immédiate et la fonction retourne uniquement lorsque la tâche est accomplie
MS\_INVALIDATE : si un autre processus mappe le même fichier, les pages seront rafraîchies à son prochain accès et il aura accès aux modifications

La projection est automatiquement détruite à la fin du processus, tandis que la fermeture du fichier ne supprime pas la projection.
Sur certains systèmes, *mmap()* est le seul moyen de créer des segments de mémoire.

**Les Mutex**

**Introduction**

Au plus, deux threads accèdent simultanément à une même ressource (périphérique, donnée). Pour assurer l'exclusion mutuelle (un seul accès à un moment donné) il existe des mécanismes de protection. Les mutex sont adaptés aux threads et définis dans la librairie des pthreads : ils permettent un accès concurrent de plusieurs threads afin de sécuriser des sections critiques qu'un seul thread peut exécuter à la fois. Un mutex est un sémaphore binaire.

**Mutex POSIX**

**Création**

int pthread\_mutex\_create(pthread\_mutex\_t \*, mutex\_attr\_t \*);

**Demande**

int pthread\_mutex\_lock(pthread\_mutex\_t \*);

**Restitution**

int pthread\_mutex\_unlock(pthread\_mutex\_t \*);

**Destruction**

int pthread\_mutex\_destroy(pthread\_mutex\_t \*);

**Demande non bloquante**

int pthread\_mutex\_trylock(pthread\_mutex\_t \*);

 exemple :

 if(pthread\_mutex\_trylock(&mutex)==0) {

 // section critique

 } else {

 ...

 }

**Mutex (non POSIX)**

Mêmes fonctions, sans le préfixe *pthread*.

int mutex\_create(mutex\_t \*m, int use, void \*attr);

*use* : type de mutex souhaité, toujours USYNC\_THREAD
*attr* : attributs (NULL)

**Exemples**

pthread\_mutex\_t c\_lock;

int lines,chars;

void \*wc(void \*arg) {

 // ouverture fichier

 // lecture des lignes

 pthread\_mutex\_lock(&c\_lock);

 lines+=...

 chars+=...

 pthread\_mutex\_unlock(&c\_lock);

 pthread\_exit();

}

main() {

 int lines=0; chars=0;

 pthread\_mutex\_create(&c\_lock,NULL);

 for(...)

 pthread\_create(...);

 for(...)

 pthread\_join(...);

 pthread\_mutex\_destroy(&c\_lock);
}

**Les Sémaphores**

**Introduction**

Un sémaphore est un nombre entier positif ou nul représentant une quantité de ressources disponibles. Ceci permet une mise en attente lorsque cette ressource est indisponible. Un lien de parenté est obligatoire pour l'utilisation d'un sémaphore.

**Création**

#include <semaphore.h>
int sem\_init(sem\_t \*sem, int pshared, unsigned int val);

*sem* : sémaphore à initialiser
*pshared* doit être différent de 0 si partage entre plusieurs processus
*val* : valeur initiale du sémaphore décrémentée chaque fois qu'un thread pénètre dans la portion critique du programme et incrémenté à chaque sortie de cette zone critique. L'entrée dans la portion critique ne peut se faire que si le compteur est strictement positif, ainsi la valeur initiale du compteur représente le nombre maximal de threads simultanément tolérés dans la zone critique.

**Demande**

int sem\_wait(sem\_t \*sem);

Bloque tant que la valeur du sémaphore est nulle, puis le compteur est décrémenté. Il existe une version non bloquante qui renvoie -1 si le compteur n'est pas supérieur à zero.

int sem\_trywait(sem\_t \*sem);

**Restitution**

int sem\_post(sem\_t \*sem);

Incrémente le compteur.

**Consultation**

int sem\_getvalue(sem\_t \*sem, int \*valeur);

Stocke l'état du compteur dans le pointeur en second argument.

**Destruction**

int sem\_destroy(sem\_t \*sem);

**Exemple**

sem\_t s;

routine(void \*num) {
 sem\_wait(&s);
 printf("Thread %d dans portion critique\n",(int)num);
 sleep(aleatoire());
 printf("Thread %d sort\n",(int)num);
 sem\_post(&s);
 sleep(aleatoire());
}

int main(void) {
 int i;
 pthread\_t thread;

 sem\_init(&s,0,3);

 for(i=0;i<2;i++) {

 pthread\_create(&thread, NULL, routine, (void \*)i);

 }

}

**Sémaphores IPC**

Les sémaphores IPC sont des objets partagés entre processus distincts sans lien de parenté, tout comme les segments de mémoire partagés, et définis par une clé unique. Les opérations offertes par les IPC permettent de manipuler des ensembles de sémaphores. Il est possible de demander en une fois des opérations P() ou V() indépendantes sur chaque sémaphore d'un ensemble. Ces opérations sont liées : le noyau les réalisera toutes ou n'en réalisera aucune.

**Création**

#include <sys/sem.h>
int semget(key\_t key, int nombre, int flags);

*nombre* : nombre de sémaphores dans l'ensemble
*flags* : IPC\_CREAT : crée l'ensemble s'il n'existe pas
IPC\_EXCL : échec si l'ensemble existe déjà

**Modification**

La fonction semctl() permet de consulter ou de modifier le paramétrage d'un jeu de sémaphore mais également de fixer l'état du compteur. Un objet de type semid\_ds est associé à l'ensemble et contient uid, gid, mode, nsems. Cette fonction n'est pas définie dans les fichiers d'en-tête et doit être déclarée manuellement.

int semctl(int semid, int semno, int cmd, union semun arg);

union semun {

 int valeur;

 struct semid\_ds \*buffer;

 unsigned short int \*table;

}

*semid* : id système du jeu de sémaphores (retourné par semget)
*semno* : numéro du sémaphore dans l'ensemble de sémaphores (0 : premier)
*cmd* :
IPC\_STAT : remplir *buffer* avec le paramétrage de l'ensemble de sémaphores
IPC\_SET : utiliser *buffer* pour paramétrer les autorisations d'accès sur l'ensemble
IPC\_RMID : suppression de l'ensemble en réveillant tous les processus en attente
GETALL : recopier la valeur de tous les sémaphores dans *table*
SETALL : fixer les compteurs des sémaphores avec les valeurs contenues dans *table*
GETVAL : lire la valeur du sémaphore dont le numéro est indiqué
SETVAL : fixer la valeur du sémaphore dont le numéro est indiqué

**Utilisation**

int semop(int semid, struct sembuf \*ops, unsigned int nombre);

struct sembuf {

 short sem\_num; // numéro du sémaphore concercné
 short sem\_op; // valeur numérique de l'opération à réaliser
 short sem\_flg; // attributs pour l'opération

}

*ops* : table de structures sembuf, opérations à effectuer
*sem\_flags* : IPC\_NOWAIT : l'opération ne sera pas bloquante même si *sem\_op*est négatif ou nul

Lorsque le champ *sem\_op* d'une structure *sembuf* est strictement positif, le noyau incrémente le compteur interne associé au sémaphore de la valeur indiquée et réveille les processus en attente.
Lorsque le champ*sem\_op* est strictement négatif, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit supérieur à*sem\_op* puis il décrémente le compteur de cette valeur avant de continuer l'exécution du processus.
Lorsque le champ *sem\_op* est nul, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit supérieur à *sem\_op* puis il décrémente le compteur de cette valeur avant de continuer l'exécution du processus.

Remarque : l'appel à *semop()* ne retourne que si toutes les opérations peuvent être effectuées

**Précautions**

Tout repose sur la bonne volonté du programmateur :
- le séquencement lock/unlock doit être correct
- ne jamais terminer de thread sans avoir locké les mutex
- ne jamais locker un mutex que l'on ne possède pas
La variable mutex/sémaphore globale doit être visible et utilisée par tous les processus qui doivent être initialisés correctement puis détruits à la fin.
Deux threads peuvent s'interbloquer.
Lorsque plusieurs threads attendent sur un mutex libéré, le thread réveillé est choisi aléatoirement.
Possibilité de demander à un sémaphore de se bloquer.