Programmation système en C sous UNIX
Manuel des fonctions utiles

Préambule : La plupart des fonctions décrites dans ce document retournent une valeur particulière en cas d'erreur. Dans ce cas il est possible d'avoir plus de détails sur la cause de l'erreur en consultant la variable système **errno**. Il faut pour cela inclure le fichier <error.h> qui définit les codes d'erreurs indiqués pour chaque fonction dans ce document.

1. Processus

1.1. Création de processus

#include <unistd.h>
pid_t **fork**(void);

1.1.1. Description

fork crée un processus fils qui diffère du processus père uniquement par ses valeurs de PID et PPID. Les verrouillages de fichiers et les signaux en attente ne sont pas hérités. Le processus fils démarre à la ligne suivant l'appel de **fork** mais la valeur de retour de **fork** diffère entre le processus père et le processus fils.

1.1.2. Valeur Renvoyée

En cas de succès, le PID du fils est renvoyé au processus père et 0 est renvoyé au processus fils. En cas d'échec -1 est renvoyé au processus père, aucun processus fils n'est créé et **errno** contient le code d'erreur.

1.1.3. Erreurs

ENOMEM : Impossible d'allouer assez de mémoire pour une structure de tâche pour le fils.
EAGAIN : Impossible de trouver un emplacement vide dans la table des processus.

1.2. Terminaison de processus

#include <stdlib.h>
void **exit**(int status);

1.2.1. Description

exit termine normalement le programme en cours. La valeur *status* est renvoyée au processus père (voir 1.5), les constantes **EXIT_FAILURE** et **EXIT_SUCCESS** définies dans **stdlib.h** peuvent être utilisées comme valeurs de retour. Les flux ouverts sont vidés et fermés.

1.3. Numéro de processus

#include <sys/types.h>
#include <unistd.h>
pid_t **getpid**(void);
pid_t **getppid**(void);

1.3.1. Description

getpid retourne le n° du processus actif (PID).
getppid retourne le n° du processus père du processus actif (PPID).

1.4. Mise en attente de processus

#include <unistd.h>
int **pause**(void);
unsigned int **sleep**(unsigned int *nb_sec*);
int **usleep**(useconds_t *nb_microsec*);
#include <time.h>

int **nanosleep**(const struct timespec *_duree_, struct timespec *_reste_);

1.4.1. Description

pause endort le processus appelant jusqu'à ce qu'il reçoive un signal (voir 4).

sleep endort le processus jusqu'à ce qu'une durée de *nb_sec* secondes se soit écoulée ou jusqu'à ce qu'un signal non ignoré soit reçu (voir 4).

usleep fonctionne comme **sleep** mais la durée *nb_microsec* est exprimée en microsecondes.

manosleep fonctionne comme **sleep** mais la durée est indiquée par une structure (pointée par le 1er paramètre : *duree*) contenant un temps en secondes et un temps en nanosecondes. La structure correspondant à ce paramètre est la suivante :

```
struct timespec {
        time_t  tv_sec ;   // indiquant la durée d'attente en secondes
        time_t  tv_nsec ; //  indiquant la durée d'attente en nanosecondes
}
```

Le processus sera donc endormi pour une durée correspondant à la somme des ces 2 champs.

Attention : On ne peut pas utiliser la fonction **alarm** pour générer un signal **SIGALRM** (voir 4.5) pendant que le processus est en attente dans **sleep** ou **usleep**.

1.4.2. Valeur Renvoyée

pause renvoie toujours -1 et **errno** est positionné à la valeur **EINTR**.

sleep() renvoie zéro si le temps prévu s'est écoulé et renvoie le nombre de secondes restantes si **sleep** a été interrompu par un signal.

usleep() renvoie zéro si le temps prévu s'est écoulé et -1 en cas d'erreur. L'erreur contenue dans **errno** sera **EINTR** si **usleep** a été interrompu par un signal et **EINVAL** si le paramètre est incorrect.

nanosleep() renvoie zéro si le temps prévu s'est écoulé et -1 en cas d'erreur. L'erreur contenue dans **errno** sera **EINTR** si **usleep** a été interrompu par un signal et **EINVAL** si le contenu désigné par le 1er paramètre est incorrect. Lorsque l'erreur est **EINTR** la structure pointée par le second paramètre de **nanosleep** (*reste*) contiendra le temps qu'il restait à attendre quand l'attente a été interrompue.

1.5. Attente de processus

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t  wait(int *status)
pid_t  waitpid(pid_t pid, int *status, int options);
```

1.5.1. Description

wait suspend l'exécution du processus courant jusqu'à ce qu'un processus fils se termine ou jusqu'à ce qu'un signal à intercepter arrive. Si un processus fils s'est déjà terminé au moment de l'appel (il est devenu "zombie"), la fonction revient immédiatement et retourne le PID de ce fils. Toutes les ressources utilisées par le fils sont libérées. Si aucun processus fils n'est en cours, **wait** revient immédiatement et sa valeur de retour est -1.

waitpid fonctionne comme **wait** mais permet de désigner le processus à attendre (1er paramètre) et de donner des options d'attente (3ème paramètre). La valeur du 1er paramètre peut être l'une des suivantes :

- -1 attendre la fin de n'importe quel processus fils. C'est le même comportement que **wait.**
- 0 attendre la fin de n'importe quel processus fils du même groupe que l'appelant.
- > 0 attendre la fin du processus fils de numéro *pid*.

La valeur du 3ème paramètre est un OU binaire (opérateur |) entre les constantes suivantes :

WNOHANG : ne pas attendre si aucun fils ne s'est terminé.
WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Si le paramètre *status* (pointeur vers un entier) est non **NULL**, **wait** et **waitpid** y stockent l'information sur la valeur de retour du fils. La valeur de retour mise par le fils lors de l'appel à **exit** est modifiée par le système, pour récupérer la valeur initiale il faut utiliser la fonction **WEXITSTATUS** qui reçoit en paramètre l'entier pointé par *status* et renvoie la valeur initialement mise dans **exit** par le processus fils.

1.5.2. Valeur Renvoyée

En cas de réussite le PID du fils qui s'est terminé est renvoyé, en cas d'échec ou s'il n'y a aucun processus fils à attendre -1 est renvoyé.

1.6. Surcharge de processus

```
#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[]);
```

1.6.1. Description

La famille de fonctions **exec** remplace l'image mémoire du processus en cours par un nouveau processus. Le premier paramètre de toutes ces fonctions est le chemin d'accès du fichier à exécuter.

Les arguments *const char *arg* ainsi que les points de suspension des fonctions **execl**, **execlp**, et **execle** peuvent être vues comme les *arg0*, *arg1*, ..., *argn* de la ligne de commande. Ils constituent une liste de pointeurs sur des chaînes de caractères terminées par le caractère nul ('\0') et sont les arguments du programme à exécuter. Par convention le premier argument *doit* pointer sur le nom du fichier associé au programme à exécuter. La liste des arguments *doit* se terminer par un pointeur **NULL**.

Les fonctions **execv** et **execvp** utilisent un tableau de pointeurs plutôt qu'une liste de paramètres de type pointeurs pour les arguments du programme à exécuter.

La fonction **execle** peut également indiquer l'environnement du processus à exécuter par son dernier paramètre. Ce paramètre est un tableau de pointeurs sur des chaînes de caractères terminées par des caractères nuls ('\0') qui *doit* se terminer par un pointeur **NULL**. Les autres fonctions fournissent au nouveau processus l'environnement constitué par la variable externe *environ*.

Les fonctions **execlp** et **execvp** agiront comme le shell dans la recherche du fichier exécutable si le nom fourni ne contient pas de slash (/). Le chemin de recherche est spécifié dans la variable d'environnement **PATH**. Si cette variable n'est pas définie, le chemin par défaut sera "/bin:/usr/bin:". Si la permission d'accès au fichier est refusée, ces fonctions continueront à parcourir le reste du chemin de recherche. Si aucun fichier n'est trouvé, elles reviendront et **errno** contiendra le code d'erreur **EACCES**.

1.6.2. Valeur Renvoyée

Normalement le processus exécuté par l'une des fonctions de la famille **exec** se termine et la fonction ne retourne jamais. Si l'une des fonctions **exec** revient à l'appelant, c'est qu'une erreur a eu lieu. La valeur de retour est -1 et **errno** contient le code d'erreur.

1.6.3. Erreurs

Toutes ces fonctions peuvent échouer et positionner **errno** comme suit:

EACCES : Le paramètre *file* n'est pas un fichier ou l'autorisation d'exécution est refusée
EPERM : Le système de fichiers est monté avec l'attribut *noexec*.
E2BIG : La liste d'arguments est trop grande.
ENOEXEC : Le fichier exécutable n'est pas dans le bon format, ou est destiné à une autre architecture.
EFAULT : Le paramètre *file* pointe en dehors de l'espace d'adressage accessible.
ENAMETOOLONG : La chaîne de caractères *file* est trop longue.
ENOENT : Le fichier désigné par le paramètre *file* n'existe pas.
ENOMEM : Pas assez de mémoire pour le noyau.
ENOTDIR : Un élément du chemin d'accès n'est pas un répertoire.
ELOOP : Le chemin d'accès au fichier désigné par le paramètre *file* contient une référence circulaire (à travers un lien symbolique)
ETXTBSY : fichier désigné par le paramètre *file* a été ouvert en écriture par un ou plusieurs processus.
EIO : Une erreur d'entrée/sortie de bas niveau s'est produite.
ENFILE : Le nombre maximal de fichiers ouverts sur le système est atteint
EMFILE : Le nombre maximal de fichiers ouverts par processus est atteint.

1.7. Exemples d'utilisation

Processus père et fils :

```
pid_t id; // valeur de retour de la fonction fork

... // partie exécutée par le père seulement
Id = fork(); // A partir de là on 2 processus : le père et son clone de fils, sauf si id = -1
switch (id) { // Pour différencier le rôle des 2 processus
case -1 : // La création du processus a échoué
    printf("Impossible de créer un processus fils\n");
    exit(EXIT_FAILURE);
    break;
case 0 : // fork renvoie 0 au processus fils
    …. // ce que fait seulement le processus fils
    exit(EXIT_SUCCESS); // si le fils doit se terminer là
    break;
default : // fork renvoie le PID du processus fils nouvellement créé au processus père
    …. // ce que fait seulement le processus père
    break;
}
… // partie exécutée par les 2 processus sauf si le fils s'est terminé par exit (comme ci-dessus)
exit(EXIT_SUCCESS); // terminaison du père
```

Processus père avec attente de terminaison du processus fils et fils surchargé :

```
pid_t id;
...
id=fork();
switch (id) {
case -1 :
    printf("Impossible de créer un processus fils\n");
    exit(EXIT_FAILURE);
    break;
case 0 :
    /* processus fils surchargé par le programme "cmde" avec les paramètres "param"  et "-p" */
    execlp("cmde", "cmde", "param" , "-p", NULL); // cmde, arg[0], arg[1], arg[2], fin de liste
    exit(EXIT_SUCCESS); // ne sert que si execlp échoue
    break;
default :
    …. // ce que fait seulement le processus père
    break;
}
```

```
int etat ; // valeur de retour du processus fils
wait(&etat) ; // attente de terminaison du processus fils
if (WEXITSTATUS(etat) == EXIT_SUCCESS) printf("Le processus fils s'est terminé sans
erreur\n");
exit(EXIT_SUCCESS); // terminaison du père
```

2. Threads

2.1. Création de thread

#include <pthread.h>
int **pthread_create**(pthread_t * *thread*, pthread_attr_t * *attr*,
void * (**start_routine*)(void *), void * *arg*);

2.1.1. Description

pthread_create créé un nouveau thread s'exécutant en parallèle avec le thread ou le processus appelant. Le nouveau thread exécute la fonction définie par le paramètre *start_routine* en lui passant *arg* comme paramètre (pointeur non défini) si *arg* n'est pas **NULL**. Le nouveau thread s'achève soit explicitement en appelant **pthread_exit()** (voir 2.3) soit implicitement lorsque la fonction *start_routine* s'achève. Ce dernier cas est équivalent à appeler **pthread_exit()** avec la valeur renvoyée par *start_routine* comme code de sortie.

Le paramètre *attr* contient les attributs de création du nouveau thread. Voir **pthread_attr_init()** pour une liste complète des attributs. Le paramètre *attr* peut être **NULL**, auquel cas, les attributs par défaut sont utilisés.

2.1.2. Valeur Renvoyée

En cas de succès, l'identifiant du nouveau thread est stocké à l'emplacement mémoire pointé par le paramètre *thread* et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

2.1.3. Erreurs

EAGAIN : pas assez de ressources système pour créer un processus pour le nouveau thread.

2.2. Attributs de thread

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t **attr*);
int **pthread_attr_destroy**(pthread_attr_t **attr*);
int **pthread_attr_setdetachstate**(pthread_attr_t **attr*, int *detachstate*);
int **pthread_attr_getdetachstate**(const pthread_attr_t **attr*, int **detachstate*);
int **pthread_attr_setschedpolicy**(pthread_attr_t **attr*, int *policy*);
int **pthread_attr_getschedpolicy**(const pthread_attr_t **attr*, int **policy*);
int **pthread_attr_setschedparam**(pthread_attr_t **attr*, const struct sched_param **param*);
int **pthread_attr_getschedparam**(const pthread_attr_t **attr*, struct sched_param **param*);
int **pthread_attr_setinheritsched**(pthread_attr_t **attr*, int *inherit*);
int **pthread_attr_getinheritsched**(const pthread_attr_t **attr*, int **inherit*);
int **pthread_attr_setscope**(pthread_attr_t **attr*, int *scope*);
int **pthread_attr_getscope**(const pthread_attr_t **attr*, int **scope*);

2.2.1. Description

Définir les attributs de création d'un thread revient à remplir la structure *attr* qui est de type **pthread_attr_t** puis à la passer en tant que deuxième paramètre à **pthread_create**(). Une valeur de *attr* égale à **NULL** revient à choisir les paramètres par défaut pour tous les attributs (voir **pthread_attr_init**).

pthread_attr_init initialise la structure d'attributs de thread *attr* et la remplit avec les valeurs par défaut pour tous les attributs (ces valeurs par défaut sont définies plus bas).

Chaque attribut *attrname* (voir plus bas pour une liste de tous les attributs) peut être individuellement modifié en utilisant la fonction **pthread_attr_set***attrname* et récupéré à l'aide de la fonction **pthread_attr_get***attrname.*

pthread_attr_destroy détruit l'objet attribut de thread qui lui est passé en paramètre : il ne peut plus être utilisé à moins qu'il ne soit réinitialisé.

Les structures d'attributs ne sont consultées que lors de la création d'un nouveau thread. La même structure d'attributs peut être réutilisée pour créer plusieurs threads. Modifier une structure d'attributs après un appel à **pthread_create** ne modifie pas les attributs du thread précédemment créé.

Les attributs de thread sont les suivants :

detachstate

Contrôle si le thread créé est dans l'état joignable (valeur **PTHREAD_CREATE_JOINABLE**) ou dans l'état détaché (valeur **PTHREAD_CREATE_DETACHED**).

Valeur par défaut : **PTHREAD_CREATE_JOINABLE**.

Quand un thread est dans l'état joignable, un autre thread peut attendre sa terminaison en utilisant **pthread_join**(). Les ressources d'un thread restent allouées après sa fin jusqu'à ce qu'un autre thread appelle **pthread_join**().

Quand un thread est dans l'état détaché, ses ressources sont immédiatement libérées quand il se termine mais **pthread_join**() ne peut plus être utilisé pour attendre la fin de ce thread.

Un thread créé dans l'état joignable peut ensuite être mis dans l'état détaché en utilisant **pthread_detach**() .

schedpolicy

Change la politique d'ordonnancement pour le thread pour l'une des politiques parmi **SCHED_OTHER** (processus normal, non temps réel), **SCHED_RR** (processus temps réel ordonnancé par round-robin) ou **SCHED_FIFO** (processus temps réel ordonnancé par premier dans la liste = premier exécuté). Les politiques d'ordonnancement temps réel **SCHED_RR** et **SCHED_FIFO** ne sont disponibles que pour les processus possédant les privilèges du super-utilisateur.

Valeur par défaut: **SCHED_OTHER**.

La politique d'ordonnancement d'un thread peut être modifiée après sa création avec **pthread_setschedpolicy**() .

schedparam

Cet attribut est sans signification si la politique d'ordonnancement est **SCHED_OTHER**; il n'a d'importance que pour les politiques temps réel **SCHED_RR** et **SCHED_FIFO**. Il concerne la priorité d'ordonnancement.

Valeur par défaut : priorité à 0.

La priorité d'ordonnancement d'un thread peut être modifiée après sa création avec **pthread_setschedparam**() .

inheritsched

Cet attribut indique si la politique et les paramètres d'ordonnancement pour le nouveau thread sont déterminés par les valeurs des attributs *schedpolicy* et *schedparam* (valeur **PTHREAD_EXPLICIT_SCHED**) ou sont héritées du thread parent (valeur **PTHREAD_INHERIT_SCHED**).

Valeur par défaut : **PTHREAD_EXPLICIT_SCHED**.

scope

Cet attribut définit comment sont interprétés les paramètres d'ordonnancement pour le nouveau thread.

PTHREAD_SCOPE_SYSTEM, signifie que tous les threads sont en compétition avec tous les processus en cours d'exécution pour le temps processeur : les priorités des threads sont interprétées relativement aux priorités de tous les autres processus sur la machine.

PTHREAD_SCOPE_PROCESS, signifie que les threads ne sont en compétition qu'avec les autres threads du même processus : les priorités des threads sont interprétées relativement à celles des autres threads du processus quelle que soit la priorité des autres processus.

Valeur par défaut : **PTHREAD_SCOPE_SYSTEM**.

2.2.2. Valeur Renvoyée

Toutes ces fonctions renvoient 0 en cas de succès et un code non nul en cas d'erreur. En cas de succès, les fonctions **pthread_attr_get***attrname* sauvegardent également la valeur actuelle de l'attribut *attrname* à l'emplacement pointé par leur second paramètre.

2.2.3. Erreurs

La fonction **pthread_attr_setdetachstate** renvoie, en cas de problème, le code d'erreur **EINVAL** si l'argument *detachstate* spécifié n'est ni **PTHREAD_CREATE_JOINABLE** ni **PTHREAD_CREATE_DETACHED**.

La fonction **pthread_attr_setschedparam** renvoie, en cas de problème, le code d'erreur **EINVAL** si la priorité indiquée par *param* n'est pas dans l'intervalle autorisé pour la politique d'ordonnancement courante dans *attr* (1 à 99 pour **SCHED_FIFO** et **SCHED_RR**; 0 pour **SCHED_OTHER**).

La fonction **pthread_attr_setschedpolicy** renvoie l'un des codes d'erreur suivants en cas de problème :

- **EINVAL** : le paramètre *policy* spécifié n'est ni **SCHED_OTHER**, ni **SCHED_FIFO**, ni **SCHED_RR**.
- **ENOTSUP** : le paramètre *policy* spécifié est **SCHED_FIFO** ou **SCHED_RR** mais l'utilisateur effectif du processus appelant n'est pas le super utilisateur.

La fonction **pthread_attr_setinheritsched** renvoie, en cas de problème, le code d'erreur E**INVAL** si le paramètre *inherit* spécifié n'est ni **PTHREAD_INHERIT_SCHED** ni **PTHREAD_EXPLICIT_SCHED**.

La fonction **pthread_attr_setscope** renvoie l'un des codes d'erreur suivants en cas de problème :

- **EINVAL** : le paramètre *scope* spécifié n'est ni **PTHREAD_SCOPE_SYSTEM** ni **PTHREAD_SCOPE_PROCESS**.
- **ENOTSUP** : le paramètre *scope* spécifié est **PTHREAD_SCOPE_PROCESS** mais n'est pas supporté pas cette version d'UNIX.

2.3. Terminaison de thread

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

2.3.1. Description

pthread_exit termine l'exécution du thread.

Le paramètre *retval* est un pointeur sur la valeur de retour du thread. Il peut être consulté par un autre thread qui utilise la fonction **pthread_join()** .

2.4. Détachement de thread

#include <pthread.h>
int **pthread_detach**(pthread_t *th*);

2.4.1. Description

pthread_detach place le thread désigné par *th* dans l'état détaché. Cela garantit que les ressources mémoire consommées par ce thread seront immédiatement libérées lorsque l'exécution de ce thread s'achèvera. Cependant, cela empêche les autres threads d'en attendre la fin en utilisant **pthread_join**.

Un thread peut être créé initialement dans l'état détaché, en utilisant l'attribut **detachstate** dans l'appel de **pthread_create()** . La fonction **pthread_detach** ne s'applique qu'aux threads créés dans l'état joignable et nécessitant d'être mis dans l'état détaché plus tard.

Dès que **pthread_detach** rend la main, tout appel ultérieur à **pthread_join** sur le thread désigné par *th* échouera. Si un autre thread a déjà joint le thread désigné par *th* lorsque **pthread_detach** est appelée, **pthread_detach** ne fait rien, et laisse le thread désigné par *th* dans l'état joignable.

2.4.2. Valeur Renvoyée

En cas de succès, 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

2.4.3. Erreurs

ESRCH : aucun thread ne correspond à celui désigné par *th*
EINVAL : le thread désigné par *th* est déjà dans l'état détaché.

2.5. Attente de terminaison de thread

#include <pthread.h>
int **pthread_join**(pthread_t *th*, void ***thread_return*);

2.5.1. Description

pthread_join suspend l'exécution du thread appelant jusqu'à ce que le thread désigné par *th* achève son exécution ou soit annulé.

Si le paramètre *thread_return* ne vaut pas **NULL**, la valeur renvoyée par le thread désigné par *th* y sera enregistrée. Cette valeur sera soit l'argument passé à **pthread_exit()** , soit **PTHREAD_CANCELED** si le thread désigné par *th* a été annulé.

Le thread désigné par *th* doit être dans l'état joignable : il ne doit pas avoir été détaché par **pthread_detach()** ou par l'attribut **PTHREAD_CREATE_DETACHED** lors de sa création par **pthread_create()** .

Quand l'exécution d'un thread joignable s'achève, ses ressources mémoire (descripteur de thread et pile) ne sont pas désallouées jusqu'à ce qu'un autre thread l'attende en utilisant **pthread_join**. Aussi, **pthread_join** doit être appelée une fois pour chaque thread joignable pour éviter des "fuites" de mémoire.

Au plus un seul thread peut attendre la fin d'un thread donné. Appeler **pthread_join** sur un thread dont un autre thread attend déjà la fin renvoie une erreur.

2.5.2. Valeur Renvoyée

En cas de succès, le code renvoyé par le thread désigné par *th* est enregistré à l'emplacement pointé par *thread_return*, et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

2.5.3. Erreurs

ESRCH : Aucun thread correspondant à *th* n'a pu être trouvé.
EINVAL : Le thread désigné par *th* a été détaché.
EINVAL : Un autre thread attend déjà la fin du thread désigné par *th*.
EDEADLK : Le paramètre *th* désigne le thread appelant lui-même.

2.6. Exemples d'utilisation

Création lancement et attente d'un thread joignable sans paramètre

```
pthread_attr_t attr; // attributs de création
pthread_t thread_id; // identificateur du thread créé
...
int main() {
    ...
    pthread_attr_init(&attr); // Init des attributs à leur valeur par défaut
    pthread_create(&thread_id, &attr, (void *)mon_thread, NULL); // creation du thread
    ....
    pthread_join(thread_id, NULL); // attente de terminaison du thread sans valeur de retour
    ...
    exit(EXIT_SUCCES);
}
...
void mon_thread() { // executé par le thread
    ...
    pthread_exit(NULL); // terminaison du thread sans valeur de retour
}
```

Création et lancement d'un thread détaché avec paramètre

```
pthread_attr_t attr;
pthread_t thread_id;
...
int main() {
    ...
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    int parametre = 10; // parameter passé au thread créé
    pthread_create(&thread_id, &attr, (void *)mon_thread, (void *)&parametre);
    ....
    exit(EXIT_SUCCES);
}
...
void mon_thread(void * param) { /* executé par le thread */
    int p = * ((int *) param); // récuperation du parametre en tant qu'entier (int)
    ...
    pthread_exit(NULL);
}
```

2.7. Thread et Mutex

#include <pthread.h>
pthread_mutex_t *fastmutex* = PTHREAD_MUTEX_INITIALIZER;

int **pthread_mutex_lock**(pthread_mutex_t **mutex*));
int **pthread_mutex_trylock**(pthread_mutex_t **mutex*);
int **pthread_mutex_unlock**(pthread_mutex_t **mutex*);
int **pthread_mutex_destroy**(pthread_mutex_t **mutex*);

2.7.1. Description

Un Mutex est un sémaphore d'exclusion mutuelle (MUTual EXclusion device). Il permet de protéger des données partagées contre des accès concurrents et d'implémenter des sections critiques.

Un Mutex ne peut être que dans deux états : déverrouillé ou verrouillé. Un thread qui tente de verrouiller un Mutex déjà verrouillé est suspendu jusqu'à ce que le Mutex soit déverrouillé et puisse donc être verrouillé à nouveau.

Les variables de type **pthread_mutex_t** doivent être initialisées de manière statique en utilisant la constante **PTHREAD_MUTEX_INITIALIZER**.

La fonction **pthread_mutex_lock** verrouille le Mutex. Si le Mutex est déverrouillé, il devient verrouillé et **pthread_mutex_lock** rend la main immédiatement. Si le Mutex est déjà verrouillé, **pthread_mutex_lock** suspend le thread appelant jusqu'à ce que le Mutex soit déverrouillé puis elle le verrouille et rend la main.

La fonction **pthread_mutex_trylock** se comporte de la même manière que **pthread_mutex_lock** excepté qu'elle ne bloque pas le thread appelant si le Mutex est déjà verrouillé mais rend la main immédiatement avec le code d'erreur **EBUSY**.

La fonction **pthread_mutex_unlock** déverrouille le Mutex.

La fonction **pthread_mutex_destroy** détruit un Mutex, libérant les ressources qu'il détient. Pour pouvoir être détruit, le Mutex doit être déverrouillé.

2.7.2. Valeur Renvoyée

pthread_mutex_init retourne toujours 0. Les autres fonctions renvoient 0 en cas de succès et un code d'erreur non nul en cas de problème.

2.7.3. Erreurs

La fonction **pthread_mutex_lock** renvoie l'un des codes d'erreur suivants en cas de problème :

- **EINVAL** : le Mutex n'a pas été initialisé.

La fonction **pthread_mutex_trylock** renvoie l'un des codes d'erreur suivants en cas de problème:

- **EBUSY** : le Mutex ne peut être verrouillé car il l'est déjà.
- **EINVAL** : le Mutex n'a pas été initialisé.

La fonction **pthread_mutex_unlock** renvoie le code d'erreur suivant en cas de problème:

- **EINVAL** : le Mutex n'a pas été initialisé.

La fonction **pthread_mutex_destroy** renvoie le code d'erreur **EBUSY** si le Mutex est verrouillé.

2.8. Exemple d'utilisation

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // declaration et initialisation
...
void thread(int *parametre) {
      ...
      pthread_mutex_lock(&mutex); // entrée en section critique
      /* section critique */
      pthread_mutex_unlock(&mutex); // sortie de section critique
      ...
      pthread_exit(NULL);
}

void thread2() {
      ...
      pthread_mutex_lock(&mutex); // entrée en section critique
      /* section critique */
      pthread_mutex_unlock(&mutex); // sortie de section critique
      ...
      pthread_exit(NULL);
}

int main() {
       pthread_attr_t attr;
       pthread_t thread_id1, id2;
       int param_thread1;
       ...
       pthread_attr_init(&attr);
       param_thread1 = 0;
       pthread_create(&thread_id1, &attr, (void *)thread1, &d param_thread1);
       pthread_create(&thread_id2, &attr, (void *)thread2, NULL);
       ...
       pthread_join(thread_id1,NULL);
       pthread_join(thread_id2,NULL);
       pthread_mutex_destroy(&mutex); // suppression du Mutex
       ...
       exit(EXIT_SUCCESS);
}
```

3. Sémaphores

3.1. Création d'un groupe de sémaphores

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int  semget( key_t key, int nsems, int semflg )
```

3.1.1. Description

semget retourne l'identificateur de l'ensemble de sémaphores associé à la valeur de clé *key*. Si l'option **IPC_CREAT** est présente dans le paramètre *semflg*, un nouvel ensemble contenant *nsems* sémaphores est créé à condition que *key* ait la valeur **IPC_PRIVATE** ou qu'aucun ensemble de sémaphores ne soit déjà associé à *key*.

Pendant la création, le paramètre *semflg* définit les permissions d'accès au jeu de sémaphores (pour le propriétaire, le groupe et les autres) en utilisant le même format et la même signification que les droits d'accès

des fichiers (rwxrwxrwx). Les permissions d'exécution ne sont pas utilisées par le système pour un jeu de sémaphores, l'autorisation d'écriture signifie autorisation de modification. C'est pourquoi on peut se contenter d'utiliser les constantes **SEM_A** (permission de modification) et **SEM_R** (permission de lecture).

Le paramètre *nsems* doit être inférieur ou égal au nombre maximal de sémaphores par ensemble (**SEMMSL**).

3.1.2. Valeur Renvoyée

Si l'appel réussit, **semget** renvoie l'identificateur de l'ensemble (un entier positif), sinon **semget** renvoie **-1** et **errno** contient le code d'erreur.

3.1.3. Erreurs

EACCES : Le jeu de sémaphore associé à *key* existe, mais le processus n'a aucun droit d'accès sur lui
EEXIST : Le jeu de sémaphore associé à *key* existe mais l'argument *semflg* précise à la fois **IPC_CREAT** et **IPC_EXCL**.
EIDRM : le jeu de sémaphores est prêt à être détruit
ENOENT : Aucun jeu de sémaphore associé à *key* n'existe et le paramètre *semflg* ne précise pas **IPC_CREAT**.
ENOMEM : Pas assez de mémoire pour créer les structures nécessaires.
ENOSPC : Le nombre maximal de jeux de sémaphores sur le système (**SEMMNI**) est atteint ou le nombre maximal de sémaphores sur le système est atteint (**SEMMNS**).

3.2. Manipulation de sémaphores

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int  semop( int semid, struct sembuf *sops, unsigned nsops )
```

3.2.1. Description

semop effectue des opérations sur les sémaphores sélectionnés dans l'ensemble de sémaphores identifié par le paramètre *semid*. Chacun des *nsops* éléments dans le tableau pointé par *sops* indique une opération à effectuer sur un sémaphore en utilisant une structure définie comme suit :

```
struct sembuf {
        short sem_num;  // Numéro du sémaphore (0 = premier)
        short sem_op;   // Opération sur le sémaphore
        short sem_flg;  // Options pour l'opération
       }
```

Les options possibles pour **sem_flg** sont **IPC_NOWAIT** et **SEM_UNDO**. Si une opération indique l'option **SEM_UNDO**, elle sera annulée lorsque le processus se terminera.

La fonction **semop** assure que les opérations demandées ne seront effectuées que si **toutes** les opérations sont possibles. Dans ce cas, chaque opération est effectuée sur le **sem_num**ème sémaphore de l'ensemble. Le premier sémaphore a le numéro **0.** Pour chaque sémaphore l'opération est l'une des trois décrites ci-dessous :

1°) Si le champ **sem_op** est un entier positif, la fonction ajoute cette valeur à **semval**. De plus si **SEM_UNDO** est demandé, le système met à jour le compteur "undo" du sémaphore. Cette opération n'est jamais bloquante. Le processus appelant doit avoir l'autorisation de modification sur le jeu de sémaphores.

2°) Si **sem_op** vaut zéro le processus attend que **semval** soit nul. Plusieurs cas sont possibles :
Si **semval** vaut zéro, l'opération est effectuée immédiatement
Sinon, si l'on a positionné **IPC_NOWAIT** dans **sem_flg**, l'opération échoue (en annulant les actions précédentes) et **errno** contient le code d'erreur **EAGAIN**.
Autrement **semzcnt** est incrémenté de 1 et le processus s'endort jusqu'à ce que l'un des évènements suivants se produise :

- **semval** devient égal à 0, **semzcnt** est alors décrémenté et la fonction **semop** continue.

- Le jeu de sémaphores est supprimé : la fonction **semop** échoue et **errno** contient le code d'erreur **EIDRM**.
- Le processus reçoit un signal à intercepter, la valeur de **semzcnt** est décrémentée et la fonction **semop** échoue avec **errno** contenant le code d'erreur **EINTR**.

Le processus appelant doit avoir l'autorisation de lecture sur le jeu de sémaphores.

3°) Si **sem_op** est inférieur à zéro, le processus appelant doit avoir l'autorisation de modification sur le jeu de sémaphores. Si **semval** est supérieur ou égal à la valeur absolue de **sem_op**, la valeur absolue de **sem_op** est soustraite de **semval**. Si **SEM_UNDO** est indiqué, le système met à jour le compteur "undo" du sémaphore. Puis la fonction **semop** continue. Autrement si l'on a positionné **IPC_NOWAIT** dans **sem_flg**, la fonction **semop** échoue (annulant les actions précédentes et **errno** contient le code d'erreur **EAGAIN**). Sinon **semncnt** est décrémenté de un et le processus s'endort jusqu'à ce que l'un des évènements suivants se produise :

- **semval** devient supérieur ou égal à la valeur absolue de **sem_op**, alors la valeur **semncnt** est décrémentée, la valeur absolue de **sem_op** est soustraite de **semval** et si **SEM_UNDO** est demandé le système met à jour le compteur "undo" du sémaphore. Puis la fonction **semop** continue.
- Le jeu de sémaphores est supprimé : la fonction **semop** échoue et **errno** contient le code d'erreur **EIDRM**.
- Le processus reçoit un signal à intercepter, la valeur de **semncnt** est décrémentée et la fonction **semop** échoue avec **errno** contenant le code d'erreur **EINTR**.

En cas de succès, le membre **sempid** de la structure **sem** de chacun des sémaphores indiqués dans le tableau pointé par *sops* est rempli avec le PID du processus appelant. Enfin **sem_otime** et **sem_ctime** sont fixés à l'heure actuelle.

3.2.2. Valeur Renvoyée

semop renvoie la valeur **0**, s'il réussit et **-1** s'il échoue auquel cas **errno** contient le code d'erreur.

3.2.3. Erreurs

E2BIG : le champ *nsops* est supérieur à **SEMOPM**, le nombre maximal d'opérations par appel système.
EACCES : Le processus appelant n'a pas les permissions d'accès nécessaires.
EAGAIN : Une opération a échoué et **IPC_NOWAIT** a été indiqué dans l'argument *sem_flg***.**
EFAULT : le champ *sops* pointe en dehors de l'espace d'adressage accessible.
EFBIG : La valeur du champ **sem_num** est inférieure à 0 ou supérieure ou égale au nombre de sémaphores dans l'ensemble.
EIDRM : Le jeu de sémaphores a été supprimé.
EINTR : Un signal a été reçu pendant l'attente.
EINVAL : L'ensemble de sémaphores n'existe pas ou le champ *semid* est inférieur à zéro ou le champ *nsops* n'a pas une valeur positive.
ENOMEM : Pas assez de mémoire pour allouer les structures nécessaires.
ERANGE : la somme des champs **semop** et **semval** est supérieure à **SEMVMX.**

3.3. Contrôle de sémaphores

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int  semctl( int semid, int semno, int cmd, union semun arg )
```

3.3.1. Description

semctl effectue l'opération de contrôle indiquée par *cmd* sur l'ensemble des sémaphores identifié par *semid* ou sur le *semno*^{ème} sémaphore de cet ensemble. Le premier sémaphore de l'ensemble est identifié par la valeur **0** dans l'argument *semno.*

Le type de l'argument *arg* est une union que le programme utilisateur doit déclarer :

```
union semun {
    int val;    // utilisé par SETVAL seulement
```

```
    struct semid_ds *buf;       // utilisé par IPC_STAT et IPC_SET
    ushort *array;              // utilisé par GETALL et SETALL
};
```

Les valeurs autorisées pour l'opération *cmd* sont :

IPC_STAT

Copier dans la structure pointée par *arg.buf* la structure de données concernant l'ensemble de sémaphores. Le paramètre *semno* est alors ignoré. Le processus appelant doit avoir des privilèges de lecture sur le jeu de sémaphores

IPC_SET

Fixer la valeur de certains champs de la structure **semid_ds** pointée par *arg.buf* dans la structure de contrôle de l'ensemble de sémaphores, en mettant à jour son membre **sem_ctime.** Les champs de la structure **struct semid_ds** fournie dans *arg.buf* et copiés dans la structure de l'ensemble sont :

sem_perm.uid
sem_perm.gid
sem_perm.mode

L'UID effectif du processus appelant doit être soit celui du Super-User soit celui du créateur ou du propriétaire du jeu de sémaphores. Le paramètre *semno* est ignoré.

IPC_RMID

Supprimer immédiatement l'ensemble de sémaphores en réveillant tous les processus en attente. Ils obtiendront un code d'erreur et **errno** aura la valeur **EIDRM.** L'UID effectif du processus appelant doit être soit celui du Super-User soit celui du créateur ou du propriétaire du jeu de sémaphores. Le paramètre *semno* est ignoré.

GETALL

Renvoyer la valeur **semval** de chaque sémaphore de l'ensemble dans le tableau *arg.array*. Le paramètre *semno* est ignoré. Le processus appelant doit avoir des privilèges de lecture sur le jeu de sémaphores.

GETNCNT

Renvoyer la valeur de **semncnt** pour le *semno*^{ème} sémaphore de l'ensemble (i.e. le nombre de processus en attente d'une incrémentation du champ **semval** du *semno-ième* sémaphore). Le processus appelant doit avoir des privilèges de lecture sur le jeu de sémaphores.

GETPID

Renvoyer la valeur de **sempid** pour le *semno*^{ème} sémaphore de l'ensemble (i.e. le PID du processus ayant exécute le dernier appel système **semop** sur le *semno-ième* sémaphore). Le processus appelant doit avoir des privilèges de lecture sur le jeu de sémaphores.

GETVAL

Renvoyer la valeur du champ **semval** du *semno*^{ème} sémaphore de l'ensemble Le processus appelant doit avoir des privilèges de lecture sur le jeu de sémaphores.

GETZCNT

Renvoyer la valeur du champ **semzcnt** du *semno*^{ème} sémaphore de l'ensemble. (c'est à dire le nombre de processus attendant que le champ **semval** du *semno*^{ème} sémaphore revienne à 0). Le processus appelant doit avoir des privilèges de lecture sur le jeu de sémaphores.

SETALL

Positionner le champ **semval** de tous les sémaphores de l'ensemble en utilisant le tableau *arg.array* , et en mettant à jour le champ **sem_ctime** de la structure **semid_ds** de contrôle du jeu de sémaphores. Les processus en attente sont réveillés si **semval** devient 0 ou augmente. Le paramètre *semno* est ignoré. Le processus appelant doit avoir des privilèges d'écriture sur le jeu de sémaphores.

SETVAL

Placer la valeur *arg.val* dans le champ **semval** du *semno*^{ème} sémaphore de l'ensemble en mettant à jour le champ **sem_ctime** dans la structure **semid_ds** associée au jeu de sémaphores. Le processus appelant doit avoir des privilèges d'écriture sur le jeu de sémaphores Les processus en attente sont réveillés si **semval** devient 0 ou augmente.

3.3.2. Valeur Renvoyée

En cas d'échec la fonction **semctl** renvoie **-1** et **errno** contient le code d'erreur. Autrement, **semctl** renvoie une valeur non négative dépendant du paramètre *cmd* :

GETNCNT : la valeur de **semncnt**.
GETPID : La valeur **sempid**.
GETVAL : La valeur **semval**.
GETZCNT : La valeur **semzcnt**.

3.3.3. Erreurs

EACCES : Le processus appelant n'a pas les privilèges nécessaires pour exécuter la commande *cmd*.
EFAULT : *arg.buf* ou *arg.array* pointent en dehors de l'espace d'adressage accessible.
EIDRM : L'ensemble de sémaphores a été supprimé.
EINVAL : *cmd* ou *semid* a une valeur illégale.
EPERM : Le paramètre *cmd* contient les commandes **IPC_SET** ou **IPC_RMID** mais l'UID effectif du processus appelant n'a pas les privilèges adéquats.
ERANGE : Le paramètre *cmd* contient les commandes **SETALL** ou **SETVAL** et la valeur de **semval** (pour l'ensemble ou pour certains sémaphores) est inférieure à 0 ou supérieur à la valeur **SEMVMX**.

3.4. Création de clés

```
# include <sys/types.h>
# include <sys/ipc.h>
key_t   ftok(char *pathname, char proj )
```

3.4.1. Description

ftok convertit le nom et le chemin d'accès d'un fichier existant (*pathname*) ainsi qu'un identificateur de projet (*proj***)** en une clé de type **key_t**. Le fichier doit exister mais n'est pas utilisé par **ftok**.

3.4.2. Valeur Renvoyée

Si elle réussit, la fonction **ftok** renvoie une valeur de type **key_t**, sinon elle renvoie -1, et **errno** contient un code d'erreur :

ENOENT : Un composant de *pathname* n'existe pas ou il s'agit d'une chaîne vide.
ENOTDIR : Un composant du chemin d'accès n'est pas un répertoire.
ELOOP : Trop de liens symboliques rencontrés dans le chemin d'accès.
EACCES : Autorisation refusée.
ENOMEM : Pas assez de mémoire.
ENAMETOOLONG : Nom de fichier trop long

3.5. Exemples d'utilisation

Création d'un groupe de 5 sémaphores avec clé privée :

```
#define NOMBRE_SEMAPHORES  5
int semidgr5 = semget(IPC_PRIVATE, NOMBRE_SEMAPHORES,
                      IPC_CREAT | IPC_EXCL | SEM_R | SEM_A);
```

Création d'un seul sémaphore avec clé non privée :

```
#define INDEX  64100
key_t cle = ftok ("fichier_existant" , INDEX); // Création de la clé depuis un nom de fichier
int semid = semget(cle, 1, IPC_CREAT | IPC_EXCL | SEM_R |SEM_A);
```

Récupération du groupe de 5 sémaphores avec clé privée par un processus fils :

```
int semidgr5 = semget(IPC_PRIVATE, 5, 0); // Récupération de l'ID du groupe de 5 sémaphores
```

Récupération d'un seul sémaphore avec clé non privée par un processus quelconque :

```
#define INDEX  64100
key_t cle = ftok ("fichier_existant" , INDEX ); // Création de la clé depuis le nom de fichier
int semid = semget(cle, 1, 0); // Récupération de l'ID du sémaphore seul dans le groupe
```

Initialisation à 1 du 3ème sémaphore du groupe de 5 désigné par *semidgr5* :

```
union semun { // Déclaration du paramètre pour initialisation (union)
   int  val;    // Valeur pour SETVAL
   struct semid_ds *buf;    // structure pour IPC_STAT, IPC_SET
   unsigned short  *array;  // Tableau pour GETALL, SETALL
   struct seminfo  *__buf;  // Tampon pour IPC_INFO (spécifique à Linux)
```

```
} valinit;

valinit.val = 1; // valeur initiale du sémaphore = 1
semctl(semidgr5, 2, SETVAL, valinit); // Initialiser le 3ème sémaphore
```

Opération P sur le 2ème sémaphore du groupe de 5 désigné par *semidgr5* :

```
struct sembuf operationP; // définition de l'opération P
operationP.sem_num = 1; // L'opération porte sur le 2ème sémaphore
operationP.sem_op = -1; // Pour l'opération P la valeur est -1
operationP.sem_flg = 0; // Options normales
semop(semidgr5, &operationP, 1); // Effectuer l'opération P avec blocage éventuel
```

Opération V sur le sémaphore seul désigné par *semid* :

```
struct sembuf operationV; // définition de l'opération V
operationV.sem_num = 0; // L'opération porte sur le seul sémaphore
operationV.sem_op = 1; // Pour l'opération V la valeur est 1
operationV.sem_flg = 0; // Options normales
semop(semid, &operationV, 1); // Effectuer l'opération V
```

Suppression du groupe de 5 sémaphores désigné par *semidgr5* :

```
semctl(semidgr5, 0, IPC_RMID, 0); // Suppression du groupe de sémaphores
```

4. Signaux

4.1. Prise en compte de signaux par "signal"

#include <signal.h>
void (***signal**(int *signum*, void (**handler*)(int)))(int);

signal est une fonction qui accepte deux paramètres (un entier et un pointeur de fonction) et retourne un pointeur vers la fonction précédemment associée au gestionnaire de signaux.

Remarque : Il est actuellement conseillé de ne plus utiliser la fonction **signal** dont le fonctionnement n'est pas tout à fait identique selon les versions de linux. On lui préfèrera la fonction **sigaction** (voir plus loin).

4.1.1. Description

signal installe un nouveau gestionnaire (2ème paramètre) pour le signal numéro *signum*. Le gestionnaire de signal peut être soit une fonction spécifique de l'utilisateur soit l'une des constantes **SIG_IGN** ou **SIG_DFL**.

Lors de l'arrivée d'un signal correspondant au numéro *signum*, les évènements suivants se produisent : Si le gestionnaire correspondant est configuré avec **SIG_IGN**, le signal est ignoré. Si le gestionnaire vaut **SIG_DFL**, l'action par défaut pour le signal est associée. Si le gestionnaire est dirigé vers une fonction *handler*(), alors *handler*() est appelé avec l'argument *signum*.

Utiliser une fonction comme gestionnaire de signal est appelé "intercepter ou capturer le signal". Les signaux **SIGKILL** et **SIGSTOP** ne peuvent être ni ignorés ni interceptés.

4.1.2. Valeur Renvoyée

signal renvoie un pointeur vers la fonction précédente du gestionnaire de signaux ou **SIG_ERR** en cas d'erreur.

4.2. Prise en compte de signaux par "sigaction"

#include <signal.h>
int **sigaction**(int *signum*, const struct sigaction **act*, struct sigaction **oldact*);

4.2.1. Description

L'appel système **sigaction** sert à modifier l'action effectuée par un processus à la réception d'un signal spécifique.

signum indique le signal concerné, à l'exception de **SIGKILL** et **SIGSTOP**.

Si *act* est différent de **NULL**, la nouvelle action pour le signal *signum* est définie par *act*. Si *oldact* est différent de **NULL**, l'ancienne action est sauvegardée dans *oldact*.

La structure **sigaction** est définie par :

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

Le champ *sa_handler* indique l'action affectée au signal *signum*, et peut être **SIG_DFL** pour l'action par défaut, **SIG_IGN** pour ignorer le signal ou un pointeur sur une fonction de gestion de signaux de type *void fct(int)*.

Le champ *sa_mask* définit le masque des signaux à bloquer pendant l'exécution de la fonction associée au signal. Le signal de numéro *signum* est automatiquement bloqué à moins que **SA_NODEFER** soit mis dans le champ *sa_flags* (voir ci-dessous). On peut initialiser le champ *sa_mask* à l'aide des fonctions **sigemptyset**, **sigfillset**, **sigaddset** ou **sigdelset** définies comme suit :

```
#include <signal.h>
int sigemptyset(sigset_t *masque)
int sigfillset(sigset_t *masque)
int sigaddset(sigset_t *masque int sugnum)
sigdelset(sigset_t *masque int sugnum)
```

La fonction **sigemptyset** bloque tous les signaux
La fonction **sigfillset** débloque tous les signaux
La fonction **sigaddset** bloque le signal dont le numéro est donné en second paramètre
La fonction **sigdeldset** débloque le signal dont le numéro est donné en second paramètre

Toutes ces fonctions renvoient 0 si elles réussissent et -1 si le numéro de signal est invalide dans ce cas *errno* contient le code d'erreur **EINVAL**.

Le champ *sa_flags* définit le comportement du gestionnaire de signaux. Il est formé par un OU binaire (|) entre les options suivantes :

SA_NOCLDSTOP
: Si *signum* vaut **SIGCHLD**, bloquer les signaux de notification d'arrêt d'un processus fils (quand le fils reçoit un signal **SIGSTOP**, **SIGTSTP**, **SIGTTIN** ou **SIGTTOU**).

SA_RESETHAND
: Rétablir l'action à son comportement par défaut une fois que le gestionnaire a été appelé (c'est le comportement par défaut avec la fonction **signal**)

SA_RESTART
: Redémarrer automatiquement les appels systèmes longs interrompus par l'arrivée du signal (par exemple getchar()). Si **SA_RESTART** n'est pas positionné les appels systèmes longs sont considérés comme terminés après le traitement du signal.

SA_NODEFER
: Autoriser la réception d'un signal pendant l'exécution de la fonction associée à ce même signal.

4.2.2. Valeur Renvoyée

La fonction **sigaction** renvoie 0 si elle réussit et -1 si elle échoue, dans ce cas **errno** contient le code d'erreur.

4.2.3. Erreurs

EINVAL : Un signal invalide est indiqué. Cette erreur se produit également si l'on tente de modifier l'action associée à **SIGKILL** ou **SIGSTOP**.
EFAULT : *act* ou *oldact* pointent en-dehors de l'espace d'adressage accessible.
EINTR : L'appel système a été interrompu.

4.2.4. Les signaux de la norme POSIX

Le tableau ci-dessous décrit les principaux signaux définis par la norme POSIX ainsi que les actions qui leur sont normalement associées (les signaux sont définis dans le fichier **signal.h**):

Signal	Action	Commentaire
SIGHUP	T	connexion avec le terminal fermée, ou utilisateur déconnecté
SIGINT	T	Interruption au clavier (Ctrl C)
SIGQUIT	D	Quitter au clavier (Ctrl \)
SIGILL	D	Instruction inconnue
SIGABRT	D	Signal émis par la fonction abort
SIGFPE	D	Erreur lors de calcul en réels
SIGKILL	T *	Tuer un processus (kill)
SIGBUS	D	Erreur d'adressage sur le bus
SIGSEGV	D	Violation de protection mémoire
SIGPIPE	T	Tube (pipe) cassé: écriture dans un tube sans lecteurs
SIGALRM	T	Signal de délai (timer) fonction alarm
SIGTERM	T	Terminaison normale de processus
SIGUSR1	T	Signal d'utilisateur 1
SIGUSR2	T	Signal d'utilisateur 2
SIGCHLD	I	Processus fils stoppé ou terminé
SIGSTOP	S *	Stopper le processus
SIGTSTP	S	Stopper le processus au clavier (Ctrl Z)
SIGTTIN	S	Entrée au clavier pour un processus en arrière plan
SIGTTOU	S	Sortie au terminal par un processus en arrière plan

T : Terminer le processus
I : Ignorer le signal
D : Terminer le processus et faire une image de mémoire (core dump)
S : Stopper le processus
* : Le signal ne peut ni être ignoré ni être capturé

4.3. Masquage de signaux

```
#include <signal.h>
int sigprocmask(int mode, const sigset_t *masque, sigset_t *oldmasque);
```

4.3.1. Description

Cette fonction permet modifier le masque de signaux c'est-à-dire de définir l'ensemble des signaux actuellement bloqués. Le paramètre *mode* peut prendre les valeurs suivantes :

SIG_BLOCK
Les signaux indiqués dans le paramètre *masque* sont bloqués. Ils viennent donc s'ajouter à l'ensemble des signaux actuellement bloqués.

SIG_UNBLOCK
Les signaux indiqués dans le paramètre *masque* sont débloqués. Ils viennent donc s'enlever de l'ensemble des signaux actuellement bloqués.

SIG_SETMASK
Le paramètre *masque* définit le nouvel ensemble de signaux bloqués.

Le contenu du paramètre *masque* peut être défini à l'aide des fonctions **sigemptyset**, **sigfillset**, **sigaddset** ou **sigdelset** (voir 4.2.1). Si le paramètre *masque* est **NULL**, le masque de signaux n'est pas modifié mais la valeur actuelle du masque de signaux est stockée dans *oldmasque* (s'il n'est pas **NULL**).

Si le paramètre *oldmasque* est différent de **NULL**, la valeur précédente du masque de signaux est stockée dans *oldmasque*.

4.3.2. Valeur Renvoyée

sigprocmask renvoie 0 si elle réussit et -1 si le paramètre *mode* n'a pas l'une des 3 valeurs définies ci-dessus, dans ce cas *errno* contient le code d'erreur **EINVAL**.

4.4. Génération de signaux

#include <signal.h>
int **kill**(pid_t pid, int sig);

4.4.1. Description

kill peut être utilisé pour envoyer n'importe quel signal à n'importe quel processus ou groupe de processus. Le signal envoyé sera pris en compte s'il n'a pas été bloqué (voir 4.3).

Si *pid* est positif, le signal *sig* est envoyé au processus *pid*.

Si *pid* vaut zéro, alors le signal *sig* est envoyé à tous les processus appartenant au même groupe que le processus appelant.

Si *pid* vaut -1, alors le signal *sig* est envoyé à tous les processus sauf le premier (init) dans l'ordre décroissant des numéros dans la table des processus (par ex: shutdown envoie le signal **SIGTERM** à tous les processus).

Si *pid* est inférieur à -1, alors le signal *sig* est envoyé à tous les processus du groupe -*pid*.

4.4.2. Valeur Renvoyée

En cas de réussite 0 est renvoyé, en cas d'échec -1 est renvoyé et **errno** contient le code d'erreur.

4.4.3. Erreurs

EINVAL : Numéro de signal invalide.
ESRCH : Le processus ou le groupe de processus n'existe pas. Un processus existant peut être un zombie, c'est à dire qu'il s'est déjà terminé mais que son père n'a pas encore récupéré sa valeur de retour avec **wait()**.
EPERM : Le processus appelant n'a pas l'autorisation d'envoyer un signal à l'un des processus concernés. Pour qu'un processus ait le droit d'envoyer un signal à un autre processus *pid* il doit avoir des privilèges de Super-Utilisateur, ou avoir un UID réel ou effectif égal à l'ID réel ou sauvegardé du processus récepteur.

4.5. Génération retardée de signal

#include <unistd.h>
unsigned int **alarm**(unsigned int *nb_sec*);
useconds_t **ualarm**(useconds_t *usecs*, useconds_t *intervalle*);

4.5.1. Description

alarm programme une temporisation pour envoyer un signal **SIGALRM** au processus en cours dans *nb_sec* secondes. Si le signal **SIGALRM** n'est pas bloqué, intercepté ou ignoré, sa réception terminera le processus.
Si *nb_sec* vaut zéro, aucune alarme n'est planifiée. Ceci permet d'annuler une programmation antérieure.

ualarm programme une temporisation pour envoyer un signal **SIGALRM** au processus en cours dans *usescs* microsecondes. Si le signal **SIGALRM** n'est pas bloqué, intercepté ou ignoré, sa réception terminera le processus. Si le second paramètre est non nul, le signal **SIGALRM** sera renvoyé chaque *intervalle* microsecondes.

Si *usescs* vaut zéro, aucune alarme n'est planifiée. Ceci permet d'annuler une programmation antérieure.

Les programmations successives d'alarmes ne sont pas empilées, chaque appel de **alarm** ou de **ualarm** annule l'éventuelle programmation précédente.

4.5.2. Valeur renvoyée

alarm renvoie le nombre de secondes qu'il restait de la programmation précédente (qui vient donc d'être annulée par le nouvel appel de **alarm**) ou zéro si aucune alarme n'avait été planifiée auparavant ou si la précédente était terminée.

ualarm renvoie le nombre de microsecondes qu'il restait de la programmation précédente (qui vient donc d'être annulée par le nouvel appel de **ualarm**) ou zéro si aucune alarme n'avait été planifiée auparavant ou si la précédente était terminée.

4.6. Exemples d'utilisation

Envoi et récupération d'un signal SIGUSR1 entre 2 processus père et fils

```
struct sigaction sig_pere;  // paramètre pour sigaction

void signal_pere(int sig) { // traitement par le fils du signal du père
     ... // ce que l'on fait quand le signal est reçu
}

int main () {
pid_t id;
int etat;

/* on crée le processus fils */
id=fork();
switch (id) {
case -1 :
          printf("Impossible de créer un processus fils\n");
          exit(EXIT_FAILURE);
          break;
case 0 : // processus fils
sig_pere.sa_handler = signal_pere; // fonction à exécuter quand le signal est reçu
sig_pere.sa_flags = SA_RESTART ; // non interruption des fonctions système
sigemptyset (&sig_pere.sa_mask); // masque de signaux
sigaction(SIGUSR1, &sig_pere, NULL); // mise en place de la fonction
          ... // ce que fait le processus fils, pendant ce temps le signal pourra être reçu et traité
          exit(EXIT_SUCCESS);
          break;
default :// processus père
          ....
          kill(id, SIGUSR1); // envoi du signal au fils
          ...
          /* attente de terminaison du fils pour éviter les zombies */
          wait(&etat);
          exit(EXIT_SUCCESS);
          break;
          }
}
```

Envoi et récupération d'un signal retardé

```
struct sigaction sig_alarme, ancien; // paramètre et sauvegarde de sigaction

void alerte(int signal) { /* fonction appelée par alarm */
```

```
    ... // Ce que l'on fait quand le délai est terminé
}

int main() {
...
    sig_alarme.sa_handler = alerte; // fonction appelée à la fin du délai
    sig_alarme.sa_flags = SA_RESTART;
    sigemptyset (&sig_alarme.sa_mask);
sigaction(SIGALRM, &sig_alarme, &ancien); // traitement du signal retardé
alarm(10); // le signal SIGALRM sera envoyé dans 10 secondes
.... // ce que fait le programme, pendant ce temps le signal pourra être reçu et traité
alarm(0); // arrêt du délai : le signal SIGALRM  ne sera plus envoyé
    sig_alarme.sa_handler = SIG_DFL;
sigaction(SIGALRM, &ancien, NULL); // retour à la prise en compte par défaut
...
exit(EXIT_SUCCESS);
}
```

5. Tubes (pipes)

5.1. Création

#include <unistd.h>
int **pipe**(int *filedes*[2]);

5.1.1. Description

pipe crée une paire de descripteurs de fichiers, pointant sur un i-noeud (inode) de tube, et les place dans un tableau *filedes* à 2 valeurs : *filedes*[0] est utilisé pour la lecture et *filedes*[1] pour l'écriture.

En général deux processus (créés par **fork**) vont se partager le tube et utiliser les fonctions **read** et **write** pour se transmettre des données.

5.1.2. Valeur renvoyée

pipe renvoie 0 s'il réussit ou -1 s'il échoue auquel cas **errno** contient le code d'erreur.

5.1.3. Erreurs

EMFILE : Trop de descripteurs de fichiers sont utilisés par le processus.
ENFILE : La table système pour les tubes est pleine.
EFAULT : *filedes* est invalide.

5.2. Fermeture

#include <unistd.h>
int **close**(int *fd*);

5.2.1. Description

close ferme le descripteur *fd*, de manière à ce qu'il ne référence plus aucun tube.
Pour un tube on a deux descripteurs de fichiers que l'on peut fermer indépendamment l'un de l'autre. En général l'un des processus ferme le descripteur utilisé en lecture tandis que l'autre ferme celui utilisé en écriture.

Si *fd* est la dernière copie d'un descripteur de tube donné, sa fermeture libère les ressources qui lui sont associées.

5.2.2. Valeur renvoyée

close renvoie 0 s'il réussit ou -1 si le descripteur de fichier *fd* est invalide, dans ce cas **errno** vaut **EBADF**.

5.3. Lecture et écriture

Elles font appel aux fonctions **read** et **write** utilisées également pour les fichiers :

```
#include <sys/types.h>
#include <unistd.h>
ssize_t   read(int fd, void *buf, size_t count);
ssize_t   write(int fd, const void *buf, size_t count);
```

5.3.1. Description

read lit au maximum *count* octets dans le tube associé au descripteur *fd* et les place dans le buffer pointé par *buf. S*i *count* vaut zéro, **read** renvoie zéro et n'a pas d'autre effet.
write écrit *count* octets pris dans le buffer pointé par *buf* dans le tube associé au descripteur *fd.*

5.3.2. Valeur renvoyée

S'il échoue **read** renvoie -1 auquel cas **errno** contient le code d'erreur.
Sinon, **read** renvoie le nombre d'octets lus et avance le curseur de lecture de ce nombre. Le fait que le nombre renvoyé soit plus petit que le nombre demandé n'est pas une erreur cela se produit s'il n'y avait pas suffisamment d'octets à lire dans le tube.

write renvoie le nombre d'octets écrits (0 signifiant aucune écriture) ou -1 s'il échoue auquel cas **errno** contient le code d'erreur.

5.3.3. Erreurs

EINTR : **read** (resp **write**) a été interrompu par un signal avant d'avoir eu le temps de lire (resp. d'écrire) quoi que ce soit.
EAGAIN : On utilise une lecture (resp. une écriture) non bloquante (attribut **O_NONBLOCK** du descripteur de tube) et aucune donnée n'est disponible (resp. il n'y a plus de place dans le tube).
EBADF : *fd* n'est pas un descripteur valide ou n'est pas ouvert en lecture (resp. en écriture).
EFAULT : *buf* pointe en dehors de l'espace d'adressage accessible.
EINVAL *fd* correspond à un objet ne permettant pas l'écriture.
EPIPE *fd* est connecté à un tube (pipe) dont l'autre extrémité est fermée. Quand ceci se produit, le processus écrivain reçoit un signal **SIGPIPE.** S'il intercepte, bloque ou ignore ce signal, **EPIPE** est renvoyé.

5.4. Exemples d'utilisation

Création, lecture et fermeture d'un pipe

```
int main () {
    int tube[2]; // un tube correspond à 2 descripteurs (un pour lire et un pour écrire)
    int valeur;
    ...
    if (pipe(tube)==-1) {
            printf("création du tube impossible\n");
            exit(EXIT_FAILURE);
    }
    ...
    close(tube[1]); // ce programme n'écrit pas dans le tube
    ....
    read(tube[0],&valeur, sizeof(int)); // lire un entier dans le tube
    ...
    close(tube[0]); // fermeture du tube en lecture
    ...
```

```
        exit(EXIT_SUCCESS);
    }
```

Lecture avec arrêt lors de la fermeture du pipe par l'autre processus

```
    while (read(tube_donnees[0], &valeur,sizeof(int)) != -1) {
        /* lecture des éléments dans le tube jusqu'à ce qu'il soit fermé par l'autre processus */
        ...
    }
    close(tube_donnees[0]);
```

6. Messages

6.1. Création

include <sys/types.h>
include <sys/ipc.h>
include <sys/msg.h>
int **msgget** (key_t *key*, int *msgflg*)

6.1.1. Description

msgget renvoie l'identificateur de la file de messages associée à la clé *key*. Une nouvelle file de messages est créée si le paramètre *msgflg* contient la constante **IPC_CREAT** que la parmètre *key* a la valeur **IPC_PRIVATE** ou qu'aucune file de message n'est déjà associée au paramètre *key*. La fonction **msgget** échouera si le paramètre *msgflg* contient à la fois **IPC_CREAT** et **IPC_EXCL** et qu'une file de messages associée au paramètre *key* existe déjà.

Lors de la création, le paramètre *msgflg* définit les permissions d'accès à la file de messages (pour le propriétaire, le groupe et les autres) avec le même format et la même signification que les permissions d'accès dans les appels **open()** ou **creat()** (bien que la permission d'exécution ne soit pas utilisée). On peut le plus souvent se limiter à utiliser les constantes **MSG_R** (lecture autorisée) et **MSG_W** (écriture autorisée).

6.1.2. Valeur renvoyée

msgget renvoie l'identificateur de la file de messages (un entier positif) s'il réussit. En cas d'échec -1 est renvoyé et **errno** contient le code d'erreur.

6.1.3. Erreurs

EACCES : Une file de messages associée à la clé *key* existe mais le processus appelant n'a pas de permissions sur cette file.
EEXIST : Une file de messages associée à la clé key existe et *msgflg* contient à la fois **IPC_CREAT** et **IPC_EXCL**.
EIDRM : la file de messages est en cours de suppression.
ENOENT : Aucune file de messages n'existe associée à la clé *key* et *msgflg* ne contient pas **IPC_CREAT**.
ENOMEM : Pas assez de mémoire pour les nouvelles structures de données.
ENOSPC : Le nombre maximum de files de messages sur le système (**MSGMNI**) est atteint.

6.2. Opérations sur les messages

include <sys/types.h>
include <sys/ipc.h>
include <sys/msg.h>
int **msgsnd**(int *msqid*, struct msgbuf * *msgp*, size_t *msgsz*, int *msgflg*)
ssize **msgrcv**(int *msqid*, struct msgbuf * *msgp*, size_t *msgsz*, long *msgtyp*, int *msgflg*)

6.2.1. Description

Pour envoyer ou recevoir un message, le processus appelant alloue une structure comme celle-ci :

```
struct msgbuf {
  long  mtype;    // type de message ( doit être > 0 )
  char  mtext[msgsz]; // contenu du message
};
```

Le tableau de caractères *mtext* a une taille définie par *msgsz*, valeur entière non négative.

Le champ *mtype* doit avoir une valeur strictement positive qui puisse être utilisée par le processus lecteur pour la sélection de messages (voir plus bas).
Le processus doit avoir une permission d'écriture sur la file pour envoyer un message et une permission de lecture pour en recevoir un.

La fonction **msgsnd** insère une copie du message pointé par le paramètre *msgp* dans la file dont l'identificateur est indiqué par la valeur du paramètre *msqid*.

Le paramètre *msgflg* précise le comportement de l'appel système si l'insertion du nouveau message nécessite plus de *msg_qbytes* dans la file. En indiquant **IPC_NOWAIT** le message ne sera pas envoyé et la fonction échouera en indiquant **EAGAIN** dans **errno**. Sinon, le processus sera suspendu jusqu'à ce que la condition de blocage soit levée (auquel cas le message sera envoyé et la fonction réussira) ou que la file soit supprimée (auquel cas la fonction échouera et **errno** contiendra **EIDRM**) ou que le processus reçoive un signal à intercepter (auquel cas la fonction échouera et **errno** contiendra **EINTR**).

Si la fonction réussit, la structure de file de messages sera modifiée comme suit :
msg_qnum est incrémenté de 1.
msg_stime est rempli avec l'heure actuelle.

La fonction **msgrcv** lit un message depuis la file indiquée par *msqid* dans la structure **msgbuf** pointée par le paramètre *msgp* en extrayant le message en cas de réussite.

Le paramètre *msgsz* indique la taille maximale en octets du champ *mtext* de la structure pointée par le paramètre *msgp*. Si le contenu du message est plus long que *msgsz* octets et si le paramètre *msgflg* contient **MSG_NOERROR**, alors le message sera tronqué (et la partie tronquée sera perdue). Sinon le message ne sera pas extrait de la file et la fonction échouera en indiquant **E2BIG** dans **errno.**

Le paramètre *msgtyp* indique le type de message désiré :
- Si *msgtyp* vaut 0, le premier message est lu.
- Si *msgtyp* est supérieur à 0, alors le premier message de type *msgtyp* est extrait de la file. Si *msgflg* contient **MSG_EXCEPT** l'inverse est effectué : le premier message de type différent de *msgtyp* est extrait de la file.
- Si *msgtyp* est inférieur à 0, le premier message de la file avec un type inférieur ou égal à la valeur absolue de *msgtyp* est extrait.

Le paramètre *msgflg* est composé d'un OU binaire (|) entre les constantes suivantes :
- **IPC_NOWAIT** si aucun message du type désiré n'est présent, la fonction échoue et **errno** est fixé à **ENOMSG**.
- **MSG_EXCEPT** utilisé avec *msgtyp* supérieur à 0 pour lire les messages de type différent de *msgtyp*.
- **MSG_NOERROR** tronque sans erreur les messages trop longs

Si aucun message du type requis n'est disponible et si on n'a pas précisé **IPC_NOWAIT** dans le paramètre *msgflg*, le processus appelant est bloqué jusqu'à l'occurrence d'un des événements suivants :
- Un message du type désiré arrive dans la file.
- Le processus appelant reçoit un signal à intercepter : la fonction échoue et **errno** contient **EINTR**.

Si la fonction réussit, la structure décrivant la file de messages est mise à jour comme suit :
- *msg_lrpid* est rempli avec le PID du processus appelant.
- *msg_qnum* est décrémenté de 1
- *msg_rtime* est rempli avec l'heure actuelle.

6.2.2. Valeur renvoyée

En cas d'échec les deux fonctions renvoient -1 et **errno** contient le code d'erreur. Sinon **msgsnd** renvoie 0 et **msgrvc** renvoie le nombre d'octets copiés dans la table *mtext*.

6.2.3. Erreurs

Pour msgsnd :
EAGAIN : Le message n'a pas pu être envoyé à cause de la limite *msg_qbytes* pour la file et de la présence de **IPC_NOWAIT** dans *mgsflg.*
EACCES : le processus appelant n'a pas de permissions de lecture dans la file.
EFAULT : *msgp* pointe en dehors de l'espace d'adressage accessible.
EIDRM : La file de message a été supprimée
EINTR : Un signal est arrivé avant d'avoir pu écrire quoi que ce soit.
EINVAL : *msqid* ou *mtype* ou *msgsz* sont invalides.
ENOMEM : pas assez de mémoire.

Pour msgrcv :
E2BIG : message trop long et **MSG_NOERROR** n'a pas été requis.
EACCES : Le processus appelant n'a pas de permission de lecture dans la file.
EFAULT : *msgp* pointe en dehors de l'espace d'adressage
EINTR : Un signal est arrivé avant d'avoir pu lire quoi que ce soit.
EINVAL : *msgqid* ou *msgsz* sont invalides.
ENOMSG : **IPC_NOWAIT** a été requis et aucun message du type réclamé n'existe dans la file.

6.3. Contrôle de messages

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>
int   msgctl( int msqid, int  cmd, struct msqid_ds *buf )
```

6.3.1. Description

msgctl permet d'effectuer l'opération indiquée par *cmd* sur la file de messages ayant l'identificateur *msqid.* Les valeurs possibles de *cmd* sont :

IPC_STAT Copier les informations depuis la structure représentant la file de messages dans la structure pointée par *buf.* L'appelant doit avoir des privilèges d'accès en lecture sur la file de messages.

IPC_SET Ecrire la valeur de certains champs de la structure **msqid_ds** pointée par *buf* dans la structure représentant la file de messages en mettant à jour le champ *msg_ctime*. Les champs à copier depuis la structure **struct msqid_ds** pointée par *buf* sont :
msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes
L'UID effectif du processus appelant doit être soit celui du Super-User soit celui du créateur ou du propriétaire de la file de messages. Seul le Super-User peut augmenter la valeur de *msg_qbytes* au-dessus de la valeur définie dans la constante système **MSGMNB**.

IPC_RMID Effacer immédiatement la file de messages et ses structures de données en réveillant tous les processus écrivains et lecteurs en attente. Ils obtiendront un code d'erreur et **errno** aura la valeur **EIDRM**. L'UID effectif du processus appelant doit être soit celui du Super-User soit celui du créateur ou du propriétaire de la file de messages.

6.3.2. Valeur renvoyée

msgctl renvoie 0 s'il réussit ou -1 s'il échoue auquel cas **errno** contient le code d'erreur.

6.3.3. Erreurs

EACCES : Le paramètre *cmd* indique l'opération **IPC_STAT** mais le processus appelant n'a pas d'accès en lecture sur la file de messages *msqid.*
EFAULT : Le paramètre *cmd* indique l'une des opérations **IPC_SET** ou **IPC_STAT** mais *buf* pointe en dehors de l'espace d'adressage accessible.
EIDRM : La file de messages a déjà été supprimée.
EINVAL : *cmd* ou *msqid* ont une valeur illégale.
EPERM : Le paramètre *cmd* indique l'une des opérations **IPC_SET** ou **IPC_RMID** mais l'UID effectif du processus appelant n'a pas assez de privilèges pour réaliser la commande. C'est aussi le cas d'un processus non Super-User tentant d'augmenter *msg_qbytes* au-dessus de la valeur définie dans la constante système **MSGMNB**.

7. Mémoire

7.1. Mémoire non partagée

#include <stdlib.h>
void ***calloc** (size_t nmemb, size_t size);
void ***malloc** (size_t size);
void ***realloc** (void *ptr, size_t size);
void **free** (void *ptr);

7.1.1. Description

calloc alloue la mémoire nécessaire pour un tableau de *nmemb* éléments, chacun d'eux représentant *size* octets et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros.

malloc alloue *size* octets et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.

realloc modifie la taille du bloc de mémoire pointé par *ptr* pour l'amener à une taille de *size* octets. **realloc** conserve le contenu de la zone mémoire minimum entre la nouvelle et l'ancienne taille. Le contenu de la zone de mémoire nouvellement allouée n'est pas initialisé. Si *ptr* est **NULL**, l'appel de **realloc** est équivalent à **malloc(size)**. Si *size* vaut zéro, l'appel est équivalent à **free**(ptr). Si *ptr* n'est pas **NULL**, il doit avoir été obtenu par un appel antérieur à **malloc**, **calloc** ou **realloc**.

free libère l'espace mémoire pointé par *ptr* qui a été obtenu lors d'un appel antérieur à **malloc**, **calloc** ou **realloc**. Si le pointeur *ptr* n'a pas été obtenu par l'un de ces appels ou s'il a déjà été libéré avec **free**, le comportement est indéterminé. Si *ptr* est **NULL**, aucune tentative de libération n'a lieu.

7.1.2. Valeur Renvoyée

Pour **calloc** et **malloc**, la valeur renvoyée est un pointeur sur la mémoire allouée ou **NULL** si la demande échoue.

realloc renvoie un pointeur sur la mémoire nouvellement allouée qui peut être différent de *ptr* ou **NULL** si la demande échoue ou si *size* vaut zéro. Si **realloc** échoue, le bloc mémoire original reste intact, il n'est ni libéré ni déplacé.

free ne renvoie pas de valeur.

7.2. Mémoire partagée

Allouer un segment de mémoire partagée

#include <sys/ipc.h>
#include <sys/shm.h>
int **shmget**(key_t *clé*, int *size*, int *shmflg*);

7.2.1. Description

La fonction **shmget** renvoie l'identificateur du segment de mémoire partagée associé à la valeur du paramètre *clé*. Un nouveau segment mémoire de taille *size*, arrondie au multiple supérieur de **PAGE_SIZE**, est créé si *clé* a la valeur **IPC_PRIVATE** ou si aucun segment de mémoire partagée n'est déjà associé à *clé* alors que **IPC_CREAT** est présent dans *shmflg*.

shmflg est composé d'un OU binaire (|) entre les constantes suivantes :

- **IPC_CREAT** : pour créer un nouveau segment. Sinon **shmget** recherchera le segment associé à *clé*, vérifiera que l'appelant a la permission de recevoir l'identifiant *shmid* associé au segment et contrôlera que le segment n'est pas détruit.
- **IPC_EXCL** : est utilisé avec **IPC_CREAT** pour demander l'échec de la création si le segment existe déjà.
- **mode d'accès** (les 9 bits de poids faibles) : indiquent les permissions pour le propriétaire, le groupe et les autres. Actuellement la permission d'exécution n'est pas utilisée par le système. On peut utiliser les constantes **SHM_R** (permission en lecture) et **SHM_W** (permission en écriture).

Si un nouveau segment est créé, les permissions d'accès de *shmflg* sont copiées dans le membre *shm_perm* de la structure **shmid_ds** décrivant le segment. Cette structure est définie ainsi :

```
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 ipc_perm {
  key_t  key;
  ushort uid;   // UID et GID effectifs du propriétaire
  ushort gid;
  ushort cuid;  // UID et GID effectif du créateur
  ushort cgid;
  ushort mode;  // Mode d'accès sur les 9 bits de poids faible
  ushort seq;   // Numéro de séquence
};
```

De plus, durant la création, le système initialise la structure **shmid_ds** associée au segment comme suit :

- *shm_perm.cuid* et *shm_perm.uid* contiennent l'UID effectif de l'appelant.
- *shm_perm.cgid* et *shm_perm.gid* contiennent le GID effectif de l'appelant.
- Les 9 bits de poids faibles de *shm_perm.mode* contiennent les 9 bits de poids faibles de *shmflg*.
- *shm_segsz* prend la valeur *size*.
- *shm_lpid*, *shm_nattch*, *shm_atime* et *shm_dtime* sont mis à 0.
- *shm_ctime* contient l'heure actuelle

Après un **fork** le fils hérite des segments de mémoire partagée.
Après un **exec** tous les segments de mémoire partagée sont détachés (pas détruits).
Lors d'un **exit** tous les segments de mémoire partagée sont détachés (pas détruits).

7.2.2. Valeur renvoyée

Un identificateur de segment *shmid* valide est renvoyé en cas de réussite, sinon -1 est renvoyé et **errno** contient le code d'erreur.

7.2.3. Erreurs

EINVAL : **SHMMIN** > *size* ou *size* > **SHMMAX** ou *size* plus grand que la taille du segment.
EIDRM : Le segment est détruit.
ENOSPC : Tous les ID de mémoire partagée sont utilisés ou l'allocation d'un segment partagé de taille *size* dépasserait les limites de mémoire partagée du système.
ENOENT : Aucun segment n'est associé à *clé*, et **IPC_CREAT** n'était pas indiqué.
EACCES : L'appelant n'a pas les autorisations d'accès au segment.
ENOMEM : Pas assez de mémoire.

Opérations sur la mémoire partagée

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/shm.h>
char  *shmat( int shmid, char *shmaddr, int shmflg )
int   shmdt( char *shmaddr)
```

7.2.4. Description

shmat attache le segment de mémoire partagée identifié par *shmid* au segment de données du processus appelant. L'adresse d'attachement est indiquée par *shmaddr* avec les critères suivants :

- Si *shmaddr* vaut **NULL**, le système essaye de trouver une zone libre.
- Si *shmaddr* n'est pas nul et si **SHM_RND** est indiqué dans *shmflg*, l'attachement a lieu à l'adresse *shmaddr* arrondie au multiple inférieur de **SHMLBA**. Si **SHM_RND** n'est pas indiqué *shmaddr* doit être alignée sur une frontière de page et l'attachement a lieu à cette adresse.

Si **SHM_RDONLY** est indiqué dans *shmflg*, le segment est attaché en lecture seulement et le processus doit disposer de la permission de lecture dessus (une tentative d'écriture lèvera le signal **SIGSEGV**). Sinon le segment est attaché en lecture et écriture et le processus doit disposer des deux permissions d'accès. Il n'y a pas de notion d'écriture seule pour les segments de mémoire partagée.

Le segment est automatiquement détaché quand le processus se termine. Le même segment peut être attaché à la fois en lecture seule et en lecture/écriture. Il peut également être attaché en plusieurs endroits de l'espace d'adressage du processus.

Si **shmat** réussit, le membre *shm_atime* de la structure **shmid_ds** associée au segment de mémoire partagée correspond à l'heure actuelle.

shmdt détache le segment de mémoire partagée situé à l'adresse indiquée par *shmaddr*. Le segment doit être effectivement attaché et l'adresse *shmaddr* doit être celle renvoyée précédemment par **shmat**.

Quand **shmdt** réussit, les membres de la structure **shmid_ds** associée au segment de mémoire partagée sont mis à jour ainsi :

shm_dtime correspond à l'heure actuelle.
shm_lpid contient le PID de l'appelant.
shm_nattch est décrémenté de 1. S'il devient nul et si le segment est marqué pour destruction, il est effectivement détruit.

La région occupée de l'espace d'adressage du processus est libérée.

7.2.5. Valeur renvoyée

Les deux fonctions renvoient -1 si elles échouent auquel cas **errno** contient le code d'erreur. Sinon **shmat** renvoie l'adresse d'attachement du segment de mémoire partagée et **shmdt** renvoie 0.

7.2.6. Erreurs

Pour shmat:

EACCES : L'appelant n'a pas les permissions d'accès nécessaires pour l'attachement.
EINVAL : *shmid* est invalide, *shmaddr* est mal alignée ou l'attachement a échoué.
ENOMEM : Pas assez de mémoire.
EINVAL : Pas de segment attaché à l'adresse *shmaddr*.

Contrôle de la mémoire partagée

#include <sys/ipc.h>
#include <sys/shm.h>
int **shmctl**(int *shmid*, int *cmd*, struct shmid_ds **buf*);

7.2.7. Description

shmctl permet à l'utilisateur d'obtenir des informations concernant un segment de mémoire partagée ainsi que de fixer le propriétaire le groupe et les permissions d'accès à ce segment. Cette fonction permet également de détruire un segment.

Les informations concernant le segment identifié par *shmid* sont renvoyées dans une structure *shmid_ds* (voir 7.2.1).

Les commandes *cmd* suivantes sont disponibles :

- **IPC_STAT** : permet de récupérer dans le buffer *buf* les informations concernant le segment de mémoire partagée. L'appelant doit avoir la permission d'utilisateur correspondant aux champs *uid*, *gid*, ou *mode* de la structure *shm_perms*. Seuls les 9 bits de poids faibles sont utilisés dans *mode*. Le membre *shm_ctime* est aussi mis à jour. L'appelant doit être le créateur du segment, son propriétaire ou le Super-User.
- **IPC_RMID** : permet de considérer un segment comme prêt pour la destruction. Il sera détruit effectivement après le dernier détachement (quand le membre *shm_nattch* de la structure *shmid_ds* associée vaudra zéro). L'appelant doit être le créateur du segment, son propriétaire ou le Super-User.

Attention : même après le dernier détachement, le contenu du segment n'est pas effacé par le système. Un processus réalisant à nouveau un attachement récupèrera son contenu. Il est à la charge du processus l'utilisateur d'écraser le contenu du segment s'il ne veut pas qu'il persiste.

7.2.8. Valeur renvoyée

shmctl renvoie 0 s'il réussit et -1 s'il échoue auquel cas **errno** contient le code d'erreur.

7.2.9. Erreurs

EACCES : *cmd* a la valeur **IPC_STAT** mais *shm_perm.modes* ne permet pas la lecture du segment *shmid*.
EFAULT : *cmd* a la valeur **IPC_SET** ou **IPC_STAT** mais *buf* pointe en-dehors de l'espace d'adressage accessible.
EINVAL : *shmid* n'est pas un identificateur de segment valide ou *cmd* n'est pas une commande reconnue.
EIDRM : *shmid* pointe sur un segment détruit.
EPERM : *cmd* a la valeur **IPC_SET** ou **IPC_RMID** mais l'appelant n'est ni le propriétaire du segment, ni son créateur, ni le Super-User.

7.3. Exemples d'utilisation

Programme qui crée une zone de mémoire partagée et y écrit

```
#define TAILLE_MEM 100
```

```
int main () {
char *adresse; // pointeur sur la mémoire partagée
key_t cle; // clé de partage de la zone de mémoire partagée
int partage;

cle=ftok("nom_de_fichier",1); // création de la clé
// création de la mémoire partagée en mode 666 (rwrwrw)
partage=shmget(cle, TAILLE_MEM, IPC_CREAT|IPC_EXCL|0666);
adresse=shmat(partage, NULL, SHM_W); // attachement de la mémoire en lecture/écriture
....
strcpy(adresse,"message"); // écriture d'une chaîne en mémoire partagée
...
shmdt(adresse); // détachement de la mémoire partagée
exit(EXIT_SUCCESS);
}
```

Programme qui récupère une zone de mémoire partagée et y lit

```
int main () {
key_t cle;
int partage;
char *adresse;

cle=ftok("nom_de_fichier ",1); // création de la même clé
partage=shmget(cle, TAILLE_MEM, 0); // accès à cette mémoire partagée
adresse=shmat(partage, NULL, SHM_R); // attachement de la mémoire en lecture
....
/* récupération du contenu de la mémoire partagée */
printf("j'ai lu : %s\n",adresse);
....
exit(EXIT_SUCCESS);
}
```

Pour supprimer la zone de mémoire partagée mettre dans un programme :

```
shmctl(partage, IPC_RMID, NULL); //suppression de la mémoire partagée
```

8. Sockets

Une socket est une structure de données permettant la communication entre deux processus distants ou non (appelés client – celui qui demande - et serveur celui qui offre un service). Cette socket est associée à un numéro de port(codé sur un entier sur 16 bits). Les 1024 premiers ports sont assez chargés et correspondent par exemple à des services tels que ftp (21), telnet (23), smtp (25), etc... Les ports suivants correspondent à des services ou des applications plus ou moins connues tels que nfs (2049), xwindows (6000), etc. Un client désirant se connecter à un serveur doit connecter à un ports ouvert en utilisant un des ports de sa machine (port local).

8.1. Création d'une socket

La création d'une socket se fait à l'aide de la fonction **socket()**, dont la définition est contenue dans **sys/socket.h** et **sys/types.h**. La valeur de retour, comme pour un fichier (mais sous Unix, tout est fichier…) est un **file descriptor (fd)**.

Exemple :

```
fd = socket(int domain, int type, int protocol);
```

Le premier argument de cette fonction, domain, définit le domaine d'adressage de la socket, qui va sélectionner la famille de protocoles à utilise, ces domaines peuvent être :

- AF_INET (Arpa Internet protocols) : Ce domaine d'adressage sera le plus courant.
- AF_UNIX (Unix internal protocols) : Ce domaine sera utilisé essentiellement pour la communication inter-processus.

Le type, quand à lui, définit le mode de communication à employer, à savoir :
- SOCK_STREAM : Pour le mode connecté.
- SOCK_DGRAM : Pour le mode non connecté.

Le protocole définit le protocole à utiliser pour la socket :
- 0 : Le choix du protocole se fera en fonction du domaine d'adressage choisi, tcp pour le mode connecté et udp pour le mode non connecté.
- IPPROTO_UDP : Pour le protocole UDP
- IPPROTO_TCP : Pour le protocole TCP.

Rq : il existe d'autres domaines, mode de communications et protocoles, mais ils sont peu utilisés.

Première socket:
fd = socket(AF_INET, SOCK_STREAM, 0);

Comme pour un fichier, file descriptor (fd) pour accéder à la socket. Si la création de la socket échoue, la fonction renvoie une valeur négative.

Et comme un fichier, dès que possible, on ferme la socket.
int close(int fd);
int shutdown(int fd, int direction);

La direction, pour shutdown, représente le niveau de fermeture de la socket :
- 0 : Rend impossible toute opération de lecture des données de la socket.
- 1 : Rend impossible toute opération d'écriture de données sur la socket.
- 2 : Rend impossible toutes opération de lecture ou d'écriture sur la socket.

A noter que **shutdown(fd, 2)** est la même chose que **close(fd)**.

8.1. Initialisation d'une socket Client

Pour connecter deux sockets, il est nécessaire de déclarer et définir la structure **sockaddr**, représentant l'adresse de la socket distante:

```
struct sockaddr_in {
      short sin_family;
      u_short.sin_port;
      struct in_addr sin_addr;
      char sin_zero[8];
}
```

- sin_family : représente le domaine d'adressage de la socket, AF_INET, AF_UNIX, etc.
- sin_port : représente le port à utiliser pour la connexion
  Par sécurité, ilvaut mieux passer la fonction htons pour convertir un numéro de port au format réseau: int htons(int port);
- sin_addr : Il représente l'adresse IP de la machine distante avec laquelle on veut communiquer, cette adresse est au format réseau, la fonction **gethostbyname** s'occupe de la conversion, mais nous devrons d'abord définir une structure, **hostent**, pointant sur l'adresse du serveur.
- sin_zero[8] : Il s'agit là d'un tableau de 8 caractères nuls, servant... A rien du tout...

Exemple :
```
struct sockaddr_in addy;
```

```
struct hostent *serveur // contient l'adresse du serveur distant.
serveur = gethostbyname("iparla.iutbayonne.univ-pau.fr");
addy.sin_family = AF_INET; //.
addy.sin_port = htons(1111);
addy.sin_addr = *(struct in_addr *)serveur->h_addr
      /* Rq :Ici ça se corse, mais c'est pas si dur que ça en fait.
serveur->h_addr est un des membres de la structure hostent, il contient
l'adresse IP du serveur, convertie au format réseau et récupéré par la
fonction gethostbyname().*/
```

Pour initialiser le membre sin_addr de la structure sockaddr on peut utiliser la fonction C bzero :
bzero((char *)&addy,sizeof(addy));

8.1. Connexion

La fonction permettant la connexion est :

int connect(int fd, struct sockaddr *ptr_address, int lg_address);

- fd est le nom du file descriptor de la socket,.
- struct sock_addr *ptr_address désigne la structure préalablement défini, qui contient les informations sur la socket distante.
- lg_address est la taille de notre structure, on utilisera l'opérateur sizeof() pour la calculer.

La fonction connect renvoie une valeur inférieure à 0 si la connexion a échoué

Exemple :
```
if(connect(fd, (struct sockaddr *)&addy, sizeof(addy)) < 0){
   printf("Connection impossible !\n");
}
```

8.1. Serveur – Attachement socket/port

Le but d'un serveur est d'accepter des connections sur un port précis, et de renvoyer aux clients les résultat du service demandé. Il faut pour cela créer une socket, lui assigner une adresse afin de la rendre accessible via la fonction bind() :

int bind(int fd, struct sockaddr *ptr_address, int addrlen);

-fd correspond au file descriptor de la socket.

- struct sockaddr *ptr_address correspond à la structure sockaddr du serveur sur laquelle nous définirons le port à mettre en écoute, le mode d'adressage etc...

- addrlen correspond a la taille en octets de la structure, on utilisera l'opérateur sizeof() pour la calculer.

bind() renvoie une valeur négative si l'assignation n'a pas pu marcher.

Exemple :
```
int fd;
fd = socket(AF_INET, SOCK_STREAM, 0);
if(fd < 0){
  print("Impossible de créer la socket\n");
  exit(-1);
}
struct sockaddr_in serveur;
serveur.sin_family = AF_INET;
serveur.sin_port = htons(1356);
serveur.sin_addr.s_addr = INADDR_ANY;
if((bind(fd, (struct sockaddr *)&serveur, sizeof(struct sockaddr))) < 0)  {
// Liaison entre la socket et le port
```

```
   printf("Impossible d'assigner la socket a une adresse\n");
   exit(1);
}
```

8.1. Ecoute sur un port

Une socket « écoute » à l'aide de la fonction listen les éventuelles demandes de connexions sur le port dans la structure.

int listen(int fd, int maxsocket);

Le premier argument contient le file descriptor de la socket, le second le nombre maximum de clients pouvant établir une connexion simultanément. Le fonction retourne une valeur négative en cas d'impossibilité d'écoute

8.1. Dialogue

Le dialogue se fait généralement à l'intérieur d'une boucle acceptant les demandes de connexions distantes, à l'aide de la fonction accept() :

int accept(int fd, struct sockaddr *ptr_addr_client, int *ptr_addrlen_client);

- fd représente file descriptor de la socket mise en écoute avec listen(),
- struct sockaddr *ptr_addr_client représente la structure sockaddr pour le client
  - il faudra définir une nouvelle structure.
- int *ptr_addrlen_client représente le pointeur vers un entier contenant la taille des deux structures sockaddr, client et serveur.

La fonction accept renvoie une valeur négative en cas d'erreur

```
int fa, size;
struct sockaddr_in client;
size = sizeof(struct sockadr_in); // La taille des structure sockaddr
client et serveur.
while(1){
   if((fa = accept(fd, (struct sockaddr *)&client, &size)) < 0) {
      fprintf(stderr, "Impossible d'accepter la socket distante\n");
      exit(-1);
   }
}
```

8.1. Envoi/Réception données/messages

Coté client et serveur, les fonctions sont identiques.

int write(int fd, void *ptr_buffer, size_t nb_char);
int send(int fd, void *ptr_buffer, size_t nb_char, option);

write() : le premier argument correspond au file descriptor de la socket avec laquelle nous désirons communiquer. Le second argument correspond à un pointeur vers un tableau de caractères, contenant les données à envoyer. Le troisième argument correspond aux nombres de caractères du tableau à envoyer à la socket.
send(), le dernier paramètre pris en charge permet de spécifier un type de données à envoyer, ce type pouvant être :

- 0 : Cela signifie qu'aucune option est passé à la fonction.
- MSG_OOB : Cela signifie que les données à transmettre sont des données urgentes.

Pour la réception des messages :

int read(int fd, void *ptr_buffer, size_t nb_char);
int recv(int fd, void *ptr_buffer, size_t nb_char, int option);

Ces deux fonctions utilisent le même principe que les deux précédentes, à la différence qu'elles lisent dans la socket un nombre nb_char de caractères, qui seront stockés dans le tableau pointé par *ptr_buffer dans la fonction. Autre petit détail, une option supplémentaire existe pour la fonction recv(), il s'agit de MSG_PEEK, servant à lire des données sans les enregistrer dans un buffer.

Ces deux fonctions sont bloquantes, et ne retourne quelque chose qu'une fois qu'il y a quelque chose à lire.