[next](http://www.malgouyres.org/cours/programmationSysteme/node8.html) [up](http://www.malgouyres.org/cours/programmationSysteme/programmationSysteme.html) [previous](http://www.malgouyres.org/cours/programmationSysteme/node6.html) [contents](http://www.malgouyres.org/cours/programmationSysteme/node1.html)  
**Next:** [Gestion du disque dûr](http://www.malgouyres.org/cours/programmationSysteme/node8.html) **Up:** [Programmation Système](http://www.malgouyres.org/cours/programmationSysteme/programmationSysteme.html) **Previous:** [Communication entre processus](http://www.malgouyres.org/cours/programmationSysteme/node6.html) **[Contents](http://www.malgouyres.org/cours/programmationSysteme/node1.html)**

**Subsections**

* [Pointeurs de fonction](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00710000000000000000)

* [Thread Posix (sous linux)](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00720000000000000000)

* + [Qu'est-ce qu'un thread ?](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00721000000000000000)

* + [Création d'un thread et attente de terminaison](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00722000000000000000)

* + [Exemples](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00723000000000000000)

* [Donnée partagées et exclusion mutuelle](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00730000000000000000)

* [Sémaphores](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00740000000000000000)

* [Exercices](http://www.malgouyres.org/cours/programmationSysteme/node7.html" \l "SECTION00750000000000000000)

**Threads Posix**

**Pointeurs de fonction**

Un pointeur de fonctions en $C$ est une variable qui permet de désigner une fonction $C$. Comme n'importe quelle variable, on peut mettre un pointeur de fonctions soit en variable dans une fonction, soit en paramètre dans une fonction.

On déclare un pointeur de fonction comme un prototype de fonction, mais on ajoute une étoile ($*$) devant le nom de la fonction. Dans l'exemple suivant, on déclare dans le main un pointeur sur des fonctions qui prennent en paramètre un int, et un pointeur sur des fonctions qui retournent un int.

#include <stdio.h>

int SaisisEntier(void)

{

int n;

printf("Veuillez entrer un entier : ");

scanf("%d", &n);

return n;

}

void AfficheEntier(int n)

{

printf("L'entier n vaut %d\(\backslash\)n", n);

}

int main(void)

{

void (\*foncAff)(int); {/\* d�claration d'un pointeur foncAff \*/

int (\*foncSais)(void); {/\*d�claration d'un pointeur foncSais \*/

int entier;

foncSais = SaisisEntier; {/\* affectation d'une fonction \*/

foncAff = AfficheEntier; {/\* affectation d'une fonction \*/

entier = foncSais(); {/\* on ex�cute la fonction \*/

foncAff(entier); {/\* on ex�cute la fonction \*/

return 0;

}

Dans l'exemple suivant, la fonction est passée en paramètre à une autre fonction, puis exécutée.

#include <stdio.h>

int SaisisEntier(void)

{

int n;

printf("Veuillez entrer un entier : ");

scanf("%d", &n);

getchar();

return n;

}

void AfficheDecimal(int n)

{

printf("L'entier n vaut %d\(\backslash\)n", n);

}

void AfficheHexa(int n)

{

printf("L'entier n vaut %x\(\backslash\)n", n);

}

void ExecAffiche(void (\*foncAff)(int), int n)

{

foncAff(n); {/\* ex�cution du param�tre \*/

}

int main(void)

{

int (\*foncSais)(void); {/\*d�claration d'un pointeur foncSais \*/

int entier;

char rep;

foncSais = SaisisEntier; {/\* affectation d'une fonction \*/

entier = foncSais(); {/\* on ex�cute la fonction \*/

puts("Voulez-vous afficher l'entier n en d�cimal (d) ou en hexa (x) ?");

rep = getchar();

{/\* passage de la fonction en param�tre : \*/

if (rep == 'd')

ExecAffiche(AfficheDecimal, entier);

if (rep == 'x')

ExecAffiche(AfficheHexa, entier);

return 0;

}

Pour prévoir une utilisation plus générale de la fonction ExecAffiche, on peut utiliser des fonctions qui prennent en paramètre un void\* au lieu d'un int. Le void\* peut être ensuite reconverti en d'autres types par un *cast*.

void AfficheEntierDecimal(void \*arg)

{

inte n = (int)arg; /\* un void\* et un int sont sur 4 octets \*/

printf("L'entier n vaut \%d\n", n);

}

void ExecFonction(void (\*foncAff)(void\* arg), void \*arg)

{

foncAff(arg); /\* exécution du paramètre \*/

}

int main(void)

{

int n;

...

ExecFonction(AfficheEntierDecimal, (void\*)n);

...

}

On peut utiliser la même fonction ExecFonction pour afficher tout autre chose que des entiers, par exemple un tableau de float.

typedef struct

{

int n; /\* nombre d'éléments du tableau \*/

double \*tab; /\* tableau de double \*/

}TypeTableau;

void AfficheTableau(void \*arg)

{

inte i;

TypeTableau \*T = (TypeTableau\*)arg; /\* cast de pointeurs \*/

for (i=0 ; i<T->n ; i++)

{

printf("%.2f", T->tab[i]);

}

}

void ExecFonction(void (\*foncAff)(void\* arg), void \*arg)

{

foncAff(arg); /\* exécution du paramètre \*/

}

int main(void)

{

TypeTableau tt;

...

ExecFonction(AfficheTableau, (void\*)&tt);

...

}

**Thread Posix (sous linux)**

**Qu'est-ce qu'un thread ?**

Un *thread* (ou *fil d'exécution* en français) est une parie du code d'un programme (une fonction), qui se déroule parallèlement à d'autre parties du programme. Un premier interêt peut être d'effectuer un calcul qui dure un peu de temps (plusieurs secondes, minutes, ou heures) sans que l'interface soit bloquée (le programme continue à répondre aux signaux). L'utilisateur peut alors intervenir et interrompre le calcul sans taper un ctrl-C brutal. Un autre intérêt est d'effectuer un calcul parallèle sur les machines multi-processeur. Les fonctions liées aux thread sont dans la bibliothèque pthread.h, et il faut compiler avec la librairie libpthread.a :

$ gcc -lpthread monprog.c -o monprog

**Création d'un thread et attente de terminaison**

Pour créer un thread, il faut créer une fonction qui va s'exécuter dans le thread, qui a pour prototype :

void \*ma\_fonction\_thread(void \*arg);

Dans cette fonction, on met le code qui doit être exécuté dans le thread. On crée ensuite le thread par un appel à la fonction pthread\_create, et on lui passe en argument la fonction ma\_fonction\_thread dans un pointeurs de fonction (et son argument arg). La fonction pthread\_create a pour prototype :

int pthread\_create(pthread\_t \*thread, pthread\_attr\_t \*attributs,

void \* (\*fonction)(void \*arg), void \*arg);

Le premier argument est un passage par adresse de l'identifiant du thread (de type pthread\_t). La fonction pthread\_create nous retourne ainsi l'identifiant du thread, qui l'on utilise ensuite pour désigner le thread. Le deuxième argument attributs désigne les attributs du thread, et on peut mettre NULL pour avoir les attibuts par défaut. Le troisième argument est un pointeur sur la fonstion à exécuter dans le thread (par exemple ma\_fonction\_thread, et le quatrième argument est l'argument de la fonction de thread.

Le processus qui exécute le main (l'équivalent du processus père) est aussi un thread et s'appelle le *thread principal*. Le thread principal peut attendre la fon de l'exécution d'un autre thread par la fonction pthread\_join (similaire à la fonction wait dans le fork. Cette fonction permet aussi de récupérer la valeur retournée par la fonction ma\_fonction\_thread du thread. Le prototype de la fonction pthread\_join est le suivant :

int pthread\_join(pthread\_t thread, void \*\*retour);

Le premier paramètre est l'dentifiant du thread (que l'on obtient dans pthread\_create), et le second paramètre est un passage par adresse d'un pointeur qui permet de récupérer la valeur retournée par ma\_fonction\_thread.

**Exemples**

Le premier exemple crée un thread qui dort un nombre de secondes passé en argument, pendant que le thread principal attend qu'il se termine.

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

void \*ma\_fonction\_thread(void \*arg)

{

int nbsec = (int)arg;

printf("Je suis un thread et j'attends %d secondes\(\backslash\)n", nbsec);

sleep(nbsec);

puts("Je suis un thread et je me termine");

pthread\_exit(NULL); {/\* termine le thread proprement \*/

}

int main(void)

{

int ret;

pthread\_t my\_thread;

int nbsec;

time\_t t1;

srand(time(NULL));

t1 = time(NULL);

nbsec = rand()%10; {/\* on attend entre 0 et 9 secondes \*/

{/\* on cr�e le thread \*/

ret = pthread\_create(&my\_thread, NULL,

ma\_fonction\_thread, (void\*)nbsec);

if (ret != 0)

{

fprintf(stderr, "Erreur de cr�ation du thread");

exit (1);

}

pthread\_join(my\_thread, NULL); {/\* on attend la fin du thread \*/

printf("Dans le main, nbsec = %d\(\backslash\)n", nbsec);

printf("Duree de l'operation = %d\(\backslash\)n", time(NULL)-t1);

return 0;

}

Le deuxième exemple crée un thread qui lit une valeur entière et la retourne au main.

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

void \*ma\_fonction\_thread(void \*arg)

{

int resultat;

printf("Je suis un thread. Veuillez entrer un entier\(\backslash\)n");

scanf("%d", &resultat);

pthread\_exit((void\*)resultat); {/\* termine le thread proprement \*/

}

int main(void)

{

int ret;

pthread\_t my\_thread;

{/\* on cr�e le thread \*/

ret = pthread\_create(&my\_thread, NULL,

ma\_fonction\_thread, (void\*)NULL);

if (ret != 0)

{

fprintf(stderr, "Erreur de cr�ation du thread");

exit (1);

}

pthread\_join(my\_thread, (void\*)&ret); {/\* on attend la fin du thread \*/

printf("Dans le main, ret = %d\(\backslash\)n", ret);

return 0;

}

**Donnée partagées et exclusion mutuelle**

Lorsqu'un nouveau processus est créé par un fork, toutes les données (variables globales, variables locales, mémoire allouée dynamiquement), sont dupliquées et copiées, et le processus père et le processus fils travaillent ensuite sur des variables différentes.

Dans le cas de threads, la mémoire est *partagée*, c'est à dire que les variables globales sont partagées entre les différents threads qui s'exécutent en parallèle. Cela pose des problèmes lorsque deux threads différents essaient d'écrire et de lire une même donnée.

Deux types de problèmes peuvent se poser :

* Deux threads concurrents essaient en même temps de modifier une variable globale ;
* Un thread modifie une structure de donnée tandis qu'un autre thread essaie de la lire. Il est alors possible que le thread lecteur lise la structure alors que le thread écrivain a écrit la donnée à moitié. La donnée est alors incohérente.

Pour accéder à des données globales, il faut donc avoir recours à un mécanisme d'exclusion mutuelle, qui fait que les threads ne peuvent pas accéder en même temps à une donnée. Pour celà, on introduit des données appelés *mutex*, de type pthread\_mutex\_t.

Un thread peut verrouiller un mutex, avec la fonction pthread\_mutex\_lock(), pour pouvoir accéder à une donnée globale ou à un flot (par exemple pour écrire sur la sortie stdout). Une fois l'accès terminé, le thread dévérouille le mutex, avec la fonction pthread\_mutex\_unlock(). Si un thread A essaie de vérouiller le un mutex alors qu'il est déjà verrouillé par un autre thread B, le thread A reste bloqué sur l'appel de pthread\_mutex\_lock() jusqu'à ce que le thread B dévérouille le mutex. Une fois le mutex dévérouillé par B, le thread A verrouille immédiatement le mutex et son exécution se poursuit. Cela permet au thread B d'accéder tranquillement à des variables globales pendant que le thread A attend pour accéder aux mêmes variables.

Pour déclarer et initialiser un mutex, on le déclare en variable globale (pour qu'il soit accessible à tous les threads) :

pthread\_mutex\_t my\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

La fonction pthread\_mutex\_lock(), qui permet de verrouiller un mutex, a pour prototype :

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

Il faut éviter de verrouiller deux fois un même mutex dans le même thread sans le déverrouiller entre temps. Il y a un risque de blocage définitif du thread. Certaines versions du système gèrent ce problème mais leur comportement n'est pas portable.

La fonction pthread\_mutex\_unlock(), qui permet de déverrouiller un mutex, a pour prototype :

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

Dans l'exemple suivant, différents threads font un travail d'une durée aléatoire. Ce travail est fait alors qu'un mutex est verrouillé.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

pthread\_mutex\_t my\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void\* ma\_fonction\_thread(void \*arg);

int main(void)

{

int i;

pthread\_t thread[10];

srand(time(NULL));

for (i=0 ; i<10 ; i++)

pthread\_create(&thread[i], NULL, ma\_fonction\_thread, (void\*)i);

for (i=0 ; i<10 ; i++)

pthread\_join(thread[i], NULL);

return 0;

}

void\* ma\_fonction\_thread(void \*arg)

{

int num\_thread = (int)arg;

int nombre\_iterations, i, j, k, n;

nombre\_iterations = rand()%8;

for (i=0 ; i<nombre\_iterations ; i++)

{

n = rand()%10000;

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

printf("Le thread num�ro %d commence son calcul\(\backslash\)n", num\_thread);

for (j=0 ; j<n ; j++)

for (k=0 ; k<n ; k++)

{}

printf("Le thread numero %d a fini son calcul\(\backslash\)n", num\_thread);

pthread\_mutex\_unlock(&my\_mutex);

}

pthread\_exit(NULL);

}

Voici un extrait de la sortie du programme. On voit qu'un thread peut travailler tranquillement sans que les autres n'écrivent.

...

Le thread numéro 9 commence son calcul

Le thread numero 9 a fini son calcul

Le thread numéro 4 commence son calcul

Le thread numero 4 a fini son calcul

Le thread numéro 1 commence son calcul

Le thread numero 1 a fini son calcul

Le thread numéro 7 commence son calcul

Le thread numero 7 a fini son calcul

Le thread numéro 1 commence son calcul

Le thread numero 1 a fini son calcul

Le thread numéro 1 commence son calcul

Le thread numero 1 a fini son calcul

Le thread numéro 9 commence son calcul

Le thread numero 9 a fini son calcul

Le thread numéro 4 commence son calcul

Le thread numero 4 a fini son calcul

...

En mettant en commentaire les lignes avec pthread\_mutex\_lock() et pthread\_mutex\_unlock(), on obtient :

...

Le thread numéro 9 commence son calcul

Le thread numero 0 a fini son calcul

Le thread numéro 0 commence son calcul

Le thread numero 1 a fini son calcul

Le thread numéro 1 commence son calcul

Le thread numero 4 a fini son calcul

Le thread numero 8 a fini son calcul

Le thread numéro 8 commence son calcul

Le thread numero 8 a fini son calcul

Le thread numéro 8 commence son calcul

Le thread numero 1 a fini son calcul

Le thread numéro 1 commence son calcul

Le thread numero 3 a fini son calcul

Le thread numéro 3 commence son calcul

Le thread numero 3 a fini son calcul

Le thread numéro 3 commence son calcul

Le thread numero 5 a fini son calcul

Le thread numero 9 a fini son calcul

...

On voit que plusieurs threads interviennent pendant le calcul du thread numéro 9 et 4.

**Sémaphores**

En général, une section critique est une partie du code où un processus ou un thread ne peut rentrer qu'à une certaine condition. Lorsque le processus (ou un thread) entre dans la section critique, il modifie la condition pour les autres processus/threads.

Par exemple, si une section du code ne doit pas être exécutée simultanément par plus de n threads. Avant de rentrer dans la section critique, un thread doit vérifier qu'au plus n-1 threads y sont déjà. Lorsqu'un thread entre dans la section critique, il modifie la conditions sur le nombre de threads qui se trouvent dans la section critique. Ainsi, un autre thread peut se trouver empêché d'entrer dans la section critique.

La difficulté est qu'on ne peut pas utiliser une simple variable comme compteur. En effet, si le test sur le nombre de thread et la modification du nombre de threads lors de l'entrée dans la section critique se font séquentiellement par deux instructions, si l'on joue de malchance un autre thread pourrait tester le condition sur le nombre de threads justement entre l'exécution de ces deux instructions, et deux threads passeraient en même temps dans la section critiques. Il y a donc nécessité de tester et modifier la condition de manière atomique, c'est à dire qu'aucun autre processus/thread de peut rien exécuter entre le test et la modification. C'est une opération atomique appelée *Test and Set Lock*.

Les sémaphores sont un type sem\_t et une ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. On entre dans une section critique si la valeur du compteur est strictement positive.

Pour utiliser une sémaphore, on doit le déclarer et l'initialiser à une certaine valeur avec la fonction sem\_init.

\inte sem\_init(sem\_t \*semaphore, \inte partage, {\bf unsigned} \inte valeur)

Le premier argument est un passage par adresse du sémaphore, le deuxième argument indique si le sémaphore peut être partagé par plusieurs processus, ou seulement par les threads du processus appelant (partage égale 0). Enfin, le troisième argument est la valeur initiale du sémaphore.

Après utilisation, il faut systématiquement libérer le sémaphore avec la fonction sem\_destroy.

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

Les primitives de bases sur les sémaphores sont :

* sem\_wait : Reste bloquée si le sémaphore est nul et sinon décrémente le compteur (opération atomique) ;
* sem\_post : incrémente le compteur ;
* sem\_getvalue : récupère la valeur du compteur dans une variable passée par adresse ;
* sem\_trywait : teste si le sémaphore est non nul et décrémente le sémaphore, mais sans bloquer. Provoque une erreur en cas de valeur nulle du sémaphore. Il faut utiliser cette fonction avec précaution car elle est prompte à générer des bogues.

Les prototypes des fonctions sem\_wait, sem\_post et sem\_getvalue sont :

\inte sem\_wait (sem\_t \* semaphore)

\inte sem\_post(sem\_t \*semaphore)

\inte sem\_getvalue(sem\_t \*semaphore, \inte \*valeur)

**Exemple.** Le programme suivant permet au plus n sémaphores dans la section critique, où n en passé en argument.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t semaphore; {/\* variable globale : s�maphore \*/

void\* ma\_fonction\_thread(void \*arg);

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

{

int i;

pthread\_t thread[10];

srand(time(NULL));

if (argc != 2)

{

printf("Usage : %s nbthreadmax\(\backslash\)n", argv[0]);

exit(0);

}

sem\_init(&semaphore, 0, atoi(argv[1])); {/\* initialisation \*/

{/\* cr�ation des threads \*/

for (i=0 ; i<10 ; i++)

pthread\_create(&thread[i], NULL, ma\_fonction\_thread, (void\*)i);

{/\* attente \*/

for (i=0 ; i<10 ; i++)

pthread\_join(thread[i], NULL);

sem\_destroy(&semaphore);

return 0;

}

void\* ma\_fonction\_thread(void \*arg)

{

int num\_thread = (int)arg; {/\* num�ro du thread \*/

int nombre\_iterations, i, j, k, n;

nombre\_iterations = rand()%8+1;

for (i=0 ; i<nombre\_iterations ; i++)

{

sem\_wait(&semaphore);

printf("Le thread %d entre dans la section critique\(\backslash\)n",

num\_thread);

sleep(rand()%9+1);

printf("Le thread %d sort de la section critique\(\backslash\)n",

num\_thread);

sem\_post(&semaphore);

sleep(rand()%9+1);

}

pthread\_exit(NULL);

}

Exemples de trace :

\ gcc -lpthread semaphore.c -o semaphore

\ ./semaphore 2

Le thread 0 entre dans la section critique

Le thread 1 entre dans la section critique

Le thread 0 sort de la section critique

Le thread 2 entre dans la section critique

Le thread 1 sort de la section critique

Le thread 3 entre dans la section critique

Le thread 2 sort de la section critique

Le thread 4 entre dans la section critique

Le thread 3 sort de la section critique

Le thread 5 entre dans la section critique

Le thread 4 sort de la section critique

Le thread 6 entre dans la section critique

Le thread 6 sort de la section critique

Le thread 7 entre dans la section critique

Le thread 7 sort de la section critique

Le thread 8 entre dans la section critique

...

Autre exemple avec trois threads dans la section critique :

\ ./semaphore 3

Le thread 0 entre dans la section critique

Le thread 1 entre dans la section critique

Le thread 2 entre dans la section critique

Le thread 1 sort de la section critique

Le thread 3 entre dans la section critique

Le thread 0 sort de la section critique

Le thread 4 entre dans la section critique

Le thread 2 sort de la section critique

Le thread 5 entre dans la section critique

Le thread 3 sort de la section critique

Le thread 6 entre dans la section critique

Le thread 6 sort de la section critique

Le thread 7 entre dans la section critique

Le thread 5 sort de la section critique

Le thread 8 entre dans la section critique

Le thread 4 sort de la section critique

Le thread 9 entre dans la section critique

Le thread 9 sort de la section critique

Le thread 1 entre dans la section critique

Le thread 1 sort de la section critique

Le thread 2 entre dans la section critique

Le thread 7 sort de la section critique

Le thread 0 entre dans la section critique

Le thread 8 sort de la section critique

Le thread 6 entre dans la section critique

Le thread 2 sort de la section critique

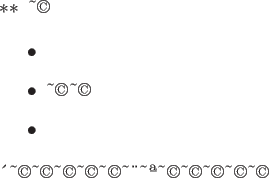
Le thread 3 entre dans la section critique

Le thread 3 sort de la section critique

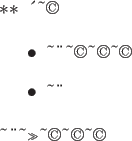
**Exercices**

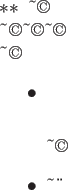
\begin{exercice}{*} \'Ecrire un programme qui crée un thread qui prend en paramètre un tableau d'entiers et l'affiche dans la console. \end{exercice}

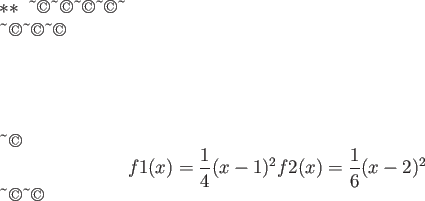
\begin{exercice}{*} \'Ecrire un programme qui crée un thread qui alloue un tabl... ...iers aléatoires entre 0 et 99, et retourne le tableau d'entiers. \end{exercice}

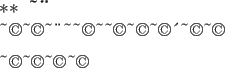


\begin{exercice}{**} {\bf a)} Reprendre la fonction de thread de génération d'... ... cas d'annulation, et doit afficher le résultat du calcul sinon. \end{exercice}









\begin{exercice}{***} {\bf (Problème de l'émetteur et du récepteur)} Un threa... ...dent et que le récepteur ne lise pas deux fois le même octet ? \end{exercice}

\begin{exercice}{***} {\bf (Problème des producteurs et des consommateurs)} Des... ...cteurs restent bloqués en attendant que des places se libèrent. \end{exercice}

\begin{exercice}{***} {\bf (Problème des lecteurs et des rédacteurs)} Ce probl... ... écrivains attendent un temps aléatoire entre 1 et 10 secondes. \end{exercice}

[next](http://www.malgouyres.org/cours/programmationSysteme/node8.html) [up](http://www.malgouyres.org/cours/programmationSysteme/programmationSysteme.html) [previous](http://www.malgouyres.org/cours/programmationSysteme/node6.html) [contents](http://www.malgouyres.org/cours/programmationSysteme/node1.html)  
**Next:** [Gestion du disque dûr](http://www.malgouyres.org/cours/programmationSysteme/node8.html) **Up:** [Programmation Système](http://www.malgouyres.org/cours/programmationSysteme/programmationSysteme.html) **Previous:** [Communication entre processus](http://www.malgouyres.org/cours/programmationSysteme/node6.html) [**Contents**](http://www.malgouyres.org/cours/programmationSysteme/node1.html)

*Rémy Malgouyres 2012-09-19*

v