



Programmation Système Unix

DUT Informatique

IUT “A” de Lille, Département informatique

Michaël Hauspie – Bureau 4A49
(`Michael.Hauspie@univ-lille1.fr` –
`http://cristal.univ-lille.fr/~hauspie`)

Partie A

Introduction



IUT A

Organisation

☞ 15 semaines

☞ **Cours magistral**
1h par semaine

Le support sera distribué partiellement au cours du semestre mais n'est pas auto-suffisant

⇒ **VOUS DEVEZ PRENDRE DES NOTES !**

☞ **TP**
3h par semaine

☞ Équipe pédagogique

- Christophe Bacara
- Olivier Fourdrinoy
- Michaël Hauspie
- David Selosse

Évaluation

Contrôle continu sur 3 notes

- ☞ 1 devoir surveillé de C coefficient 1
2h en salle d'examen fin octobre/début novembre
- ☞ 1 devoir surveillé de système coefficient 1
2h en salle d'examen en fin de semestre
- ☞ 1 note de TP coefficient 1
 - Notes des TP (1 point par sujet terminé)
 - Contrôle individuel sur machine

Contenu du cours

☞ Programmation système sous Linux

- Utilisation d'appels système dans des programmes écrits en C
- Respect (autant que possible) des normes ANSI et POSIX

☞ Il ne s'agit pas d'un cours de langage C

- Même s'il commence par quelques rappels
- « Le langage C (C ANSI) » par Brian W. Kernighan et Dennis M. Ritchie (Dunod)

☞ Gestion

- Système de fichiers
- Aspects multi-tâches
- Aspects multi-utilisateurs

Partie B

Rappels : Langage C



IUT A

Cours n° B.1

Rappels de C, structure d'un programme

Présentation générale

- ☞ Programme C \equiv ensemble de fonctions
- ☞ Langage défini par une norme. Nous utiliserons la norme ISO C90 (également appelée ANSI C).
- ☞ Les fonctions (leur code source) sont réparties dans un ou plusieurs fichiers textes
- ☞ Une des fonctions **doit** se nommer `main` :
c'est le code de cette fonction qui sera exécuté par le programme

☛ **Le langage C n'est pas un langage de très haut niveau :**

- Accès bas niveau à l'architecture des machines (code souvent rapide)
- Typage faible
- En général peu de contrôle de cohérence :

les compilateurs considèrent que le programmeur est intelligent et qu'il sait ce qu'il fait !!!

Variables

☞ Définir une **variable** c'est :

- 1 réserver une zone mémoire pour des données d'un certain type
- 2 nommer cette zone pour faire référence à son contenu via un identificateur

☞ Un **type** correspond principalement à la taille des données manipulées
L'opérateur `sizeof()` permet de connaître la taille (en octets) occupée par une variable ou un type.

Peu de types primitifs :

Symbole	Taille occupée	Données représentées
<code>char</code>	1 octet	entier
<code>int</code>	dépend de la machine	entier
<code>float</code>	dépend de la machine	nombre en virgule flottante simple précision
<code>double</code>	dépend de la machine	nombre en virgule flottante double précision

Qualifications

Il est possible de préfixer un type par un ou plusieurs mots réservés de façon à préciser la manière dont une variable doit être considérée (pour l'arithmétique par exemple).

☞ `short` et `long`

☞ `signed` et `unsigned`

☞ `const`

☞ `extern`

☞ `static`

☞ `register`

☞ ...

Généralités

- ☞ Syntaxe des fonctions :

```
type_de_retour nom (liste_des_paramètres) { code }
```

- ☞ **Toute fonction doit être déclarée avant d'être utilisée** (possibilité de simplement prototyper)
- ☞ Les structures de contrôle sont, dans l'ensemble, identique à celles de JAVA
- ☞ Les opérateurs sont, dans l'ensemble, identique à ceux de JAVA
- ☞ **Les définitions de variable doivent être placées AVANT les instructions** (dans les fonctions ou en dehors des fonctions)
- ☞ **Une définition de variable n'initialise pas le contenu de la variable** (valeur inconnue avant première affectation)

```
int une_variable_globale;

void fait_pas_grand_chose(void);

int main (void)
{
    if (une_variable_globale == 3)
    {
        fait_pas_grand_chose();
    }
    return 0;
}

void fait_pas_grand_chose(void)
{
    char c;

    c = 7;

    une_variable_globale = (int) c;
}
```

Affichage formaté

```
int printf(const char *format, ...);
```

- ➡ Définition via l'utilisation des headers :

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

- ➡ Le format représente une suite d'instructions permettant de définir **quoi** et **comment** imprimer.
- ➡ Le format peut inclure :
 - ➡ des caractères classiques
 - ➡ des séquences d'échappement (`\n`, `\t`, etc)
 - ➡ des spécificateurs de conversions :
 - `%d` affichage comme un entier
 - `%c` affichage comme un caractère
 - `%s` affichage comme une chaîne de caractères

À chaque spécificateur doit correspondre un paramètre du bon type

```
#include <stdio.h>

int
main (int argc, char ** argv)
{
    int i = 66;

    char c = 'C';

    char *s = "Bonjour tout le monde";

    printf ("avec %%d : %d\n", i);
    printf ("avec %%c : %c\n", i);
    printf ("avec %%d : %d\n", s);
    printf ("avec %%c : %c\n", s);
    printf ("avec %%s : %s\n", s);
}
```

Chaîne de compilation

Pour passer du fichier source au fichier exécutable plusieurs étapes sont nécessaires :

- ① Utilisation d'un préprocesseur `cpp`
transformation du source par remplacement textuel en C pur
- ② Utilisation d'un compilateur `cc1`
transformation du source en code assembleur
- ③ Utilisation d'un assembleur `as`
transformation du source assembleur en langage machine (objet)
- ④ Utilisation d'un éditeur de liens `ld`
assemblage des différents objets et ajout du code de démarrage

Toutes ces étapes sont généralement masquées par l'outil de développement qui permet de générer le programme exécutable en deux étapes :

① `gcc -c essai.c` étape ① à ③

② `gcc -o essai essai.o` étape ④

⇒ On dit souvent que l'outil de compilation (`gcc`) est un *wrapper*

plus de détails dans le manuel : `gcc(1)`

Options de compilation

Chaque compilateur possède son jeu de fonctionnalités supplémentaire étendant la norme C ANSI (norme ISO C90).

Ces fonctionnalités peuvent être désactivées afin d'assurer un source *portable*.

gcc comporte un **très** grand nombre d'options, dont :

- ☞ `-ansi`
Désactive les fonctionnalités de gcc ne respectant pas la norme ANSI
- ☞ `-pedantic`
Rejette tous programmes ne respectant pas strictement la norme ANSI
- ☞ `-Wall -W`
Active un grand nombre d'avertissement
- ☞ `-Werror`
Transforme tous les avertissements en erreur
- ☞ `-fno-builtin`
Désactive la gestion implicite de certaines fonctions par le compilateur
- ☞ `-g`
Ajoute les informations utiles aux débogueurs dans les fichiers générés

Préprocesseur

- ☞ Le préprocesseur est un outil permettant de transformer du texte via des remplacements
- ☞ Il offre, entre autre, la possibilité dans un fichier source :

- ☛ de définir des *macros* qui seront remplacées par leur valeur partout dans le fichier

```
#define MACRO valeur
```

- ☛ d'inclure le contenu d'un autre fichier

```
#include "fichier"
```

```
#include <fichier>
```

- ☛ de tester l'existence d'une macro

```
#ifdef MACRO
```

```
    /* cette partie restera si MACRO est definie */
```

```
#else
```

```
    /* cette partie restera si MACRO n'est pas definie */
```

```
#endif
```

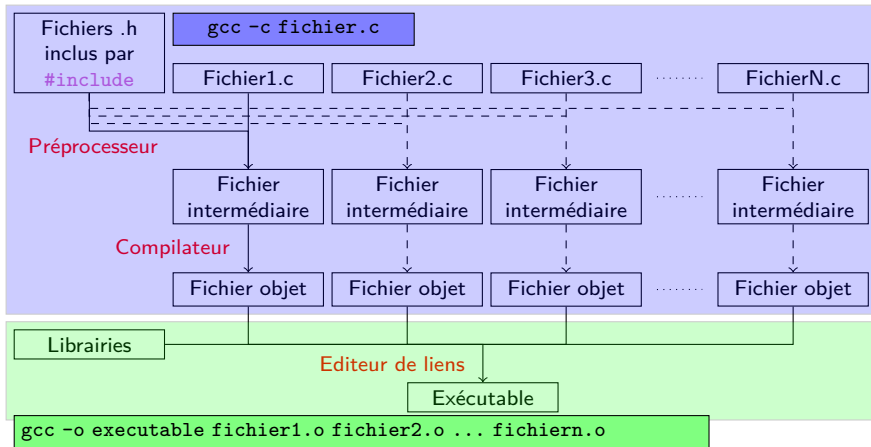
Éditions des liens

- ✎ Pour créer un programme il faut créer un fichier (exécutable) contenant tous les objets (variables ou fonctions) qu'il manipule.
- ✎ Quand un objet est **défini** dans un fichier source le compilateur le définit aussi dans le fichier objet correspondant (l'adresse de l'objet est connu dans ce fichier).
- ✎ Quand un objet est **déclaré** dans un fichier source il n'est pas obligatoirement défini dans le fichier objet correspondant.

➡ Dans le fichier final toutes les adresses des objets doivent être connues.

Pour uniquement **déclarer** un objet il faut le qualifier avec **extern** dans le fichier. Pour **définir** un objet il faut le déclarer et donner sa définition (pour une fonction le code source doit être présent dans le fichier).

Compilation d'un exécutable



Cours n° B.2

Tableaux, structures, chaînes de caractères

Tableaux

- ☞ Un tableau est une zone mémoire contigue contenant plusieurs valeurs d'un même type
- ☞ La taille d'un tableau est fixée lors de la définition de celui-ci.
Elle ne peut pas être modifiée.
- ☞ Les tableaux sont toujours indicés par des valeurs entières de 0 à $\text{taille} - 1$
- ☞ **Il n'y a pas de marqueur de fin de tableau** (donc pas de vérification automatique de dépassement ni à la compilation ni à l'exécution)

Exemple de définitions :

```
int tent[10] ..... tableau de 10 entiers
char tcar[8] ..... tableau de 8 caractères
```

Exemples d'indexations :

```
tent[3] = 1 ..... 4e case du tableau
tcar[6+1] = 'a' ..... dernière case du tableau
```

```
#include <stdio.h>

int main (int argc, char ** argv)
{
    int t[10];

    int i;

    for (i = 0; i < 10; i += 1)
    {
        t[i] = 65 + i;
    }

    /* Correct */
    for (i = 0; i < 10; i ++)
    {
        printf ("%d ", t[i]);
    }
    printf ("\n");

    /* Correct ? */
    for (i = 0; i <= 10; i ++)
    {
        printf ("%c ", t[i]);
    }
    printf ("\n");

    return 0;
}
```

Les structures

Il est possible de créer des objets composites par l'entremise des **structures**.

Déclaration :

```
struct nom_de_la_structure
{
    type_du_membre_1 nom_du_membre_1;
    ...
    type_du_membre_n nom_du_membre_n;
};
```

La structure permet d'associer les membres divers, éventuellement de types différents :

<pre>struct personne { char nom[8]; int numero; };</pre>	ou encore	<pre>struct point { int x; int y; };</pre>
--	-----------	--

Opérations sur les structures

Les seules opérations autorisées sur les structures sont :

- ☞ copie ou affectation d'une structure (permettant d'utiliser une structure comme paramètre ou retour d'une fonction) ;
- ☞ récupération de l'adresse avec l'opérateur & ;
- ☞ accès aux membres.

Attention :

- ☞ **il n'est pas possible de comparer (avec l'opérateur ==) des structures !**
- ☞ il est possible d'initialiser une structure par une liste de valeurs.

Retour sur les structures

```
struct point p1,p2;
```

déclaration de variables

```
struct point origine = {0, 0};
```

initialisation

```
p1.x = 2;
```

accès aux membres

```
p1.y = 3;
```

```
p2 = p1;
```

affectation

```
printf("%d %d\n",p2.x,p2.y);
```

affichage 2 3

```
if (p1 == p2)  
    printf("egal\n");
```

Interdit !!!

Chaînes de caractères

- ☞ On définit les chaînes de caractères par une **convention** :
 - ☞ Les chaînes de caractères sont des tableaux de caractères
 - ☞ Un caractère est un **entier** défini sur un octet dans le code ASCII. Cet entier représente l'indice du caractère représenté dans la table ASCII
 - ☞ Le marqueur de fin est le caractère '`\0`', code ASCII 0
- ☞ Les fonctions d'entrées/sorties utilisent cette convention
- ☞ Les constantes de type chaînes de caractères sont exprimés entre guillemets :
`"bonjour"`

Exemple

```
int sont_egales(const char chaine1[], const char chaine2[])
{
    int i = 0;
    while (chaine1[i] == chaine2[i] && chaine1[i] != '\0')
        i++;
    if (chaine1[i] == chaine2[i])
        return 1; /* chaines gales */
    return 0; /* chaines diffrentes */
}
```

Extrait de table ASCII

Dec.	Hex.	Caractère	Dec.	Hex.	Caractère
65	41	A	66	42	B
67	43	C	68	44	D
69	45	E	70	46	F
71	47	G	72	48	H
73	49	I	74	4A	J
75	4B	K	76	4C	L
77	4D	M	78	4E	N
79	4F	O	80	50	P
81	51	Q	82	52	R
83	53	S	84	54	T
85	55	U	86	56	V
87	57	W	88	58	X
89	59	Y	90	5A	Z

Les définitions de type

Il est possible de nommer des types par l'instruction :

```
typedef un_type nom_de_type;
```

```
typedef int entier;                                entier i = 0;
```

```
typedef char * chaine;                            chaine c = "aaa";
```

```
typedef struct point point;                        point p1;
```

```
typedef struct personne etudiant;                 etudiant e;
```

```
typedef struct {  
    point haut;  
    point bas;  
}  
rectangle r; r.haut.x = 2; ...
```

Cours n° B.3

Pointeurs

Pointeurs

- ☞ Un pointeur est une variable dont le contenu est une adresse-mémoire
- ☞ Un pointeur doit être défini en fonction du type de la donnée stockée à cette adresse mémoire :

```
char * pc;   pointeur de caractères
int * pi;    pointeur d'entier
toto * pt;   pointeur de toto
void * pv;   pointeur de quelque chose
```

- ☞ **L'opérateur unaire «&» donne l'adresse-mémoire d'une donnée.**
 - ☛ Il ne s'applique qu'aux objets stockés en mémoire (variables, éléments de tableaux, etc) et donc pas aux expressions, constantes, etc
- ☞ **L'opérateur unaire «*» donne la donnée stockée à une adresse mémoire.**
 - ☛ C'est un opérateur d'indirection

`char c = 'A';` définition d'une variable

- de type `char` ;
- de nom `c` ;
- de valeur initiale 65 (ou `'A'`)
- d'adresse `&c`

`char * p = &c;` définition d'un pointeur

- de type `char *` ;
- de nom `p` ;
- de valeur initiale `&c`

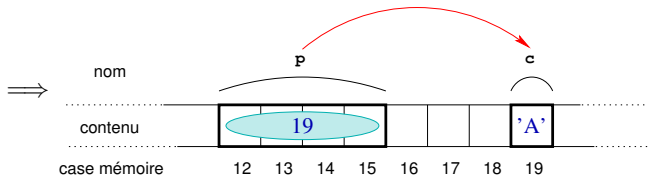
➡ `p` pointe sur `c`

Opérateurs sur les pointeurs

```
char    c;  
char *  p;
```

```
c = 'A';
```

```
p = &c;
```



☞ p et &c ont la même valeur

☞ *p et c ont la même valeur : 65 (qui est représentable par 'A')

```
#include <stdio.h>

int main (int argc, char ** argv)
{
    char c = 'A';

    char* p;

    p = &c;

    printf("c = %c\n", c);
    printf("c = %d\n", c);

    printf("\n");

    printf("*p = %c\n", *p);
    printf("*p = %d\n", *p);

    printf("\n");

    printf("p = %xd\n", p);

    printf("\n");

    printf("&c = %xd\n", &c);

    return 0;
}
```

```
int x=1;
int y=2;
int t[10];
```

```
int * pi; ..... pointeur d'entier
```

```
pi = &x; ..... pi pointe sur x
```

```
y = *pi; ..... y vaut 1
```

```
*pi = 0; ..... x vaut 0
```

```
pi = &t[5]; ..... pi pointe sur t[5]
```

**Un pointeur ne peut pointer qu'un objet du type
pour lequel il a été déclaré !**

Binky!

Pointer Fun with Binky



by Nick Parlante

This is document 104 in the Stanford CS
Education Library — please see
cslibrary.stanford.edu
for this video, its associated documents,
and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright
panel for redistribution terms.
Carpe Post Meridiem!

This is document 104 in the Stanford CS Education Library. Please see <http://cslibrary.stanford.edu/> for this and other free educational materials.
Copyright Nick Parlante 1999.

Passage de paramètres (1)

```
#include <stdio.h>

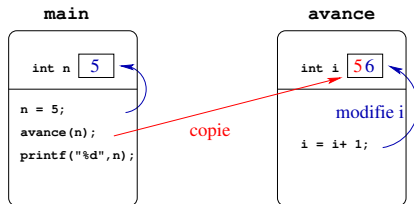
void avance(int i)
{
    i = i + 1;
}

int main (int argc, char **argv)
{
    int n = 5;

    avance (n);

    printf ("%d\n", n);

    return 0;
}
```



➡ Passage d'une valeur

Passage de paramètres (2)

```
#include <stdio.h>

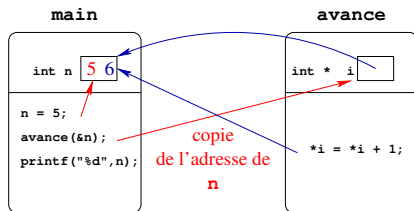
void avance(int *i)
{
    *i = *i + 1;
}

int main (int argc, char **argv)
{
    int n = 5;

    avance (&n);

    printf ("%d\n", n);

    return 0;
}
```



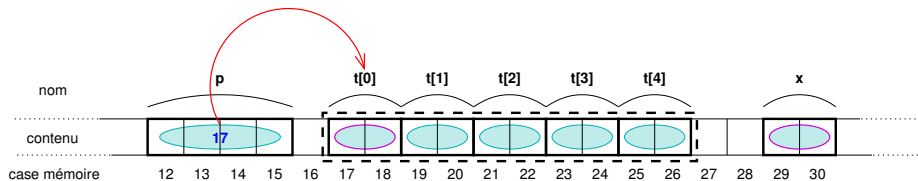
➡ Passage d'une référence

Pointeurs & tableaux (1)

`short t[5];` définir un tableau de 5 entiers

`short * p;` définir un pointeur d'entier

`short x;` définir un entier



`p = &t[0];` `p` pointe vers le premier élément de `t`

`x = *p;` copie le contenu de `t[0]` dans `x`

Pointeurs & tableaux (2)

**La valeur d'une variable de type tableau est
l'adresse du 1^{er} élément du tableau !**

$$p = \&t[0] \quad \equiv \quad p = t$$

Un tableau est une constante

$p++ \Rightarrow \text{OK}$ alors que $t++ \Rightarrow \text{INTERDIT}$

Exemple live

Pointeurs & tableaux (3)

```
char machaine[] = "bonjour";
```

définition d'un tableau de caractères de taille nécessaire pour contenir la chaîne avec lequel le tableau est initialisé.

➡ Il est possible de modifier le contenu du tableau mais **pas** l'espace pointé.

```
char * machaine = "bonjour";
```

définition d'un pointeur de caractère initialisé en pointant sur la chaîne constante "bonjour".

➡ Il est possible de modifier le pointeur pour le faire pointer sur une autre zone mémoire mais il **n'est pas possible de modifier le contenu** de la chaîne constante.

Les pointeurs de structures (1)

```
struct point *ptr1, *ptr;
```

déclaration de pointeurs

```
ptr1 = &p1;
```

affectation de l'adresse

```
(*ptr).x = 5;
```

accès à un membre

```
ptr->y = 6;
```

autre notation

Les pointeurs de structures (2)

```
struct personne {  
    char nom[9];  
    int numero;  
};
```

```
struct personne moi;  
struct personne *per;
```

```
per = &moi;
```

```
scanf("%c",&(per->nom[0])); ...;  
scanf("%c",&(per->nom[7]));  
per->nom[8] = '\\0';
```

```
per->numero = 402;
```

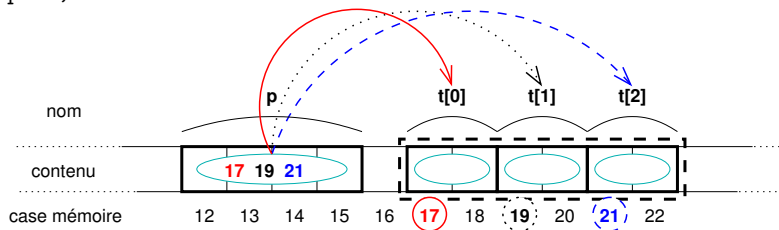
```
printf("%s %d\\n", per->nom,per->numero);
```

Calcul sur les pointeurs

Certaines opérations sont définies sur les pointeurs :

- pointeur + entier
- pointeur - entier
- pointeur - pointeur
- comparaison de pointeurs ($=$, \neq , $<$, \leq , $>$, \geq)

```
p = t;  
p = p + 1;  
p ++;
```



Par convention, on a l'habitude d'affecter 0 ou NULL à un pointeur qui ne pointe rien.

Arguments de ligne de commande

Un programme C peut être appelé avec des paramètres, dans ce cas, deux paramètres sont passés à la fonction `main` :

- ① `argc`, le nombre d'arguments («*mots*») de la ligne de commande ;
- ② `argv`, un tableau de chaînes de caractères qui contiennent les arguments (un argument ou «*mot*» par chaîne) ;

➡ `argv[argc]` vaut `NULL`.

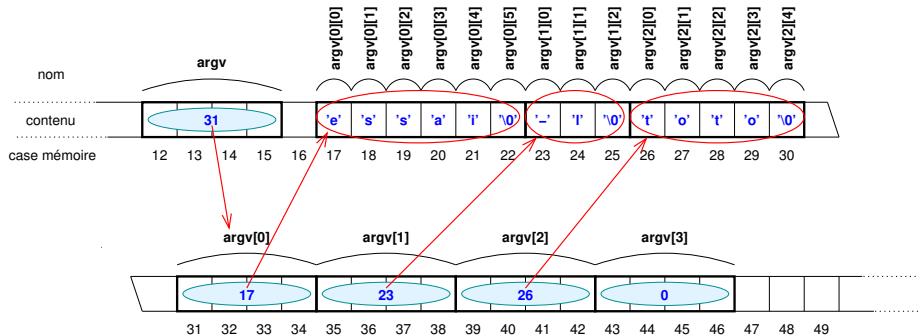
```
#include <stdio.h>

int main (int argc, char *argv[])
{
    while (argc >= 0)
    {
        printf("%s\n",argv[argc]);
        argc -= 1;
    }

    return 0;
}
```



```
$ essai -l toto
```



Cours n° B.4

Allocations mémoire

Problématique

La réservation d'espace via les définitions de variables (simple ou tableau) est appelée **allocation statique** :

la taille mémoire réservée n'est pas modifiable
au cours de l'exécution du programme.

Pour utiliser de l'espace mémoire dont on ne connaît pas la taille avant le début de l'exécution du programme on doit trouver une solution.

La solution naïve c'est de :

- Réserver **statiquement** une grande zone
- Distribuer et Redistribuer **dynamiquement** des bouts de cette zone au cours de l'exécution

Allocation dynamique

La solution naïve a des inconvénients :

- ☞ sous-utilisation de la mémoire
les limites de l'allocation statique s'applique et toute la mémoire de l'ordinateur (ou du processus) n'est pas utilisable (seul le système a accès à toute la mémoire de la machine)
- ☞ mauvaise utilisation de la mémoire
toute la mémoire réservée est contigue, des trous peuvent se former et ne plus être utilisé (fragmentation)

Les systèmes d'exploitations modernes (au moins ceux respectant la norme POSIX) offrent des fonctions d'**allocation mémoire dynamique** :

- ☞ l'espace mémoire n'est pas réservé avant l'appel système correspondant
- ☞ la taille de l'espace mémoire réservée peut-être modifiée au cours de l'exécution du programme sans perte de données
- ☞ le système gère seul la mémoire réservée (problème d'alignement, etc.)
- ☞ les problèmes de fragmentation sont toujours présents (mais en général limités par des algorithmes d'allocation efficaces)

➡ Fonctions de la bibliothèque standard du C (stdlib.h)

malloc

```
void * malloc (size_t size)
```

- ☞ Demande au système de réserver `size` octets contigus en mémoire
- ☞ Retourne l'adresse du début de la zone ou `NULL` si le système n'a pas pu trouver la place disponible
- ☞ Le pointeur doit être converti en pointeur du type des données qui vont être stockées dans cette zone

plus de détails dans le manuel : `malloc(3)`

➡ **malloc n'initialise pas la zone réservée**

calloc

```
void *calloc(size_t nmemb, size_t size)
```

- ☞ Demande au système de réserver une zone de `nmemb` cases contigus en mémoire pouvant toutes contenir `size` octets. Toutes ces cases sont initialisées à 0.
- ☞ Retourne l'adresse du début de la zone ou `NULL` si le système n'a pas pu trouver la place disponible
- ☞ Le pointeur doit être converti en pointeur du type des données qui vont être stockées dans cette zone

plus de détails dans le manuel : `calloc(3)`

realloc

```
void *realloc(void *ptr, size_t size);
```

- ☞ Change la taille de la zone mémoire pointée par `ptr` pour qu'elle occupe `size` octets
- ☞ le contenu des cases mémoires est inchangé (du début du bloc au minimum de l'ancienne et de la nouvelle taille)
- ☞ le pointeur `ptr` doit être le résultat d'un appel précédent d'une fonction de type `malloc` ou `realloc` (ou être `NULL`)
- ☞ retourne un pointeur sur la nouvelle zone mémoire, ou `NULL` si une erreur est survenue ou si `size` valait 0
- ☞ en cas d'erreur, l'ancienne zone mémoire n'est pas modifiée.

plus de détails dans le manuel : `realloc(3)`

free

```
void free(void * ptr)
```

- ☞ Préviens le système que la zone mémoire débutant à l'adresse ptr n'est plus utilisée et que sa réservation peut être libérée
- ☞ La zone pointée par ptr doit avoir été préalablement allouée par un appel à malloc ou calloc

plus de détails dans le manuel : free(3)

Partie C

Programmation Système (débutant)



IUT A

Cours n° C.1

Entrées/Sorties bas niveau

Principes

Rappel du cours de 1^{re} année (système de gestion de fichiers) :

Au niveau noyau un fichier est une suite non-structurée d'octets (*byte stream*)

Le système permet d'accéder directement à cette suite :

- ☞ Sans formatage

- ☞ Sans conversion

 - ➡ Seuls des **octets** sont lus et écrits

- ☞ Sans tampon d'entrées/sorties au niveau du programme

 - ➡ Les octets sont lus et écrits **directement** et **sans intermédiaire**

➡ Les accès sont dits de bas niveau

Accès aux fichiers

Rappel du cours de 1^{re} année (processus) :

Tous les processus gèrent une table stockant le nom des différents fichiers qu'ils utilisent. Chaque index de cette table est appelé un *descripteur de fichiers*.

Pour accéder aux données stockées dans un fichier il faut utiliser des appels systèmes pour :

- ① Ouvrir le fichier open
- ② Manipuler le contenu du fichier read ou write
- ③ Fermer le fichier close

- ☞ **Les fichiers seront repérés par leur descripteur**
- ☞ **Tous les fichiers se manipulent de la même manière**
- ☞ **Le nombre de descripteurs disponibles pour chaque processus est limité**

Ouverture de fichier (1)

```
int open(const char * pathname, int flags);
```

☞ `pathname` → chemin (relatif ou absolu) du fichier à ouvrir

☞ `flags` → mode d'accès au fichier (lecture, écriture, etc.)

• <code>O_RDONLY</code>	} combinable avec l'opérateur « »
• <code>O_WRONLY</code>	
• <code>O_RDWR</code>	
• <code>O_CREAT</code>	
• ...	

☞ retourne le descripteur du fichier ou -1 en cas d'erreur

plus de détails dans le manuel : `open(2)`

Ouverture de fichier (2)

```
int open(const char *pathname, int flags, mode_t mode);
```

- ☞ `pathname` → chemin (relatif ou absolu) du fichier à ouvrir
- ☞ `flags` → mode d'accès au fichier (lecture, écriture, etc.)
- ☞ `mode` → droits d'accès au fichier s'il doit être créé
- ➡ retourne le descripteur du fichier ou -1 en cas d'erreur

➡ Les droits effectifs sont calculés en retirant la valeur du masque (`umask`)

plus de détails dans le manuel : `open(2)`

Exemple

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int fd;

    fd = open("/tmp/toto", O_WRONLY | O_CREAT, 0666);

    if (fd == -1)
        printf("Cr ation du fichier /tmp/toto impossible");
    else
        close(fd);

    return 0;
}
```

Traitement des erreurs

- ☞ Lorsqu'une erreur survient lors d'un appel système la variable globale `errno` est fixée
- ☞ La fonction `void perror(const char * s);` permet d'exploiter la valeur de `errno` pour afficher un message expliquant la dernière erreur système survenue

```
...  
if (open("/monfichier", O_WRONLY) == -1)  
{  
    perror("/monfichier");  
    exit(1);  
}  
...
```

```
bash$ ./essai  
/monfichier: No such file or directory
```

À chaque utilisation d'un appel système vous **devrez** traiter le cas d'erreur

Lecture

```
int read(int fd, char * buf, int count);
```

- 👉 `fd` → un descripteur d'un fichier ouvert au moins en lecture
- 👉 `buf` → l'adresse d'une zone mémoire où les octets lus seront stockés par la fonction. La zone pointée **doit avoir été réservée** pour le programmeur (allocation statique ou dynamique)
- 👉 `count` → le nombre d'octets que l'on veut lire
- ➡ retourne :
 - ☞ -1 si une erreur s'est produite
 - ☞ 0 si la fin du fichier est atteinte
 - ☞ le nombre d'octets lus sinon

Les lectures se font de manière séquentielle


plus de détails dans le manuel : `read(2)`

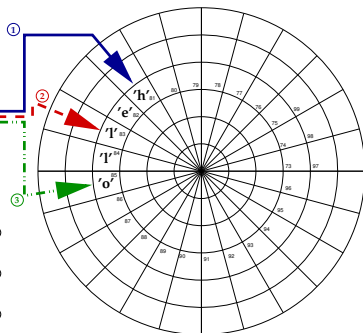
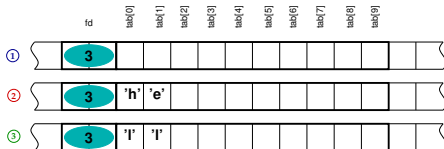
```

char tab[10];
int fd;

fd = open("unfichier", O_RDONLY);
① read(fd, tab, 2);
② read(fd, tab, 2);
③ read(fd, tab, 2);

```

0	/dev/stdin	
1	/dev/stdout	
2	/dev/stderr	
3	unfichier	



Exemple live

Écriture

```
int write(int fd, const char * buf, int count);
```

- ✎ `fd` → un descripteur d'un fichier ouvert au moins en écriture
- ✎ `buf` → l'adresse d'une zone mémoire où les octets à écrire sont stockés et doivent être lus
- ✎ `count` → le nombre d'octets que l'on veut écrire
- ➡ retourne :
 - 👁 -1 si une erreur s'est produite
 - 👁 le nombre d'octets écrits sinon

Les écritures se font de manière séquentielle

plus de détails dans le manuel : `write(2)`

Fermeture

```
int close(int fd);
```

☞ `fd` → un descripteur d'un fichier ouvert

☞ retourne :

☞ -1 si une erreur s'est produite

☞ 0 sinon

plus de détails dans le manuel : `close(2)`

☞ Les fichiers ouverts par un programme sont automatiquement fermés par le système lorsque le processus qui l'exécute se termine

Exemple live

Entrées/Sorties de données quelconques

On peut lire/écrire autre chose que des octets (caractères), en respectant quelques règles :

- ☞ contrôle du formatage des données
- ☞ prendre garde à la taille des données à manipuler
- ☞ attention à **l'alignement** et à **l'endianness** !

```
struct point
{
    int x;
    int y;
};
struct point a;
...
write(f, &a, sizeof(a));
...
read(f, &a, sizeof(struct point));
```

Exemple live

Cours n° C.2

Système de fichiers

Objectifs

- ✎ *Utilisateur* : sauvegarder des données, les organiser, y avoir accès facilement.
Caractérisation par :
 - ★ un nom
 - ★ éventuellement une localisation
- ✎ *Système* :
 - ★ gestion des ressources matérielles
 - ★ stockage d'informations particulières (taille, date de création, droits, ...)
- ✎ *Système multi-utilisateurs* :
 - ★ partage de ressources entre utilisateurs
 - ★ protection des accès

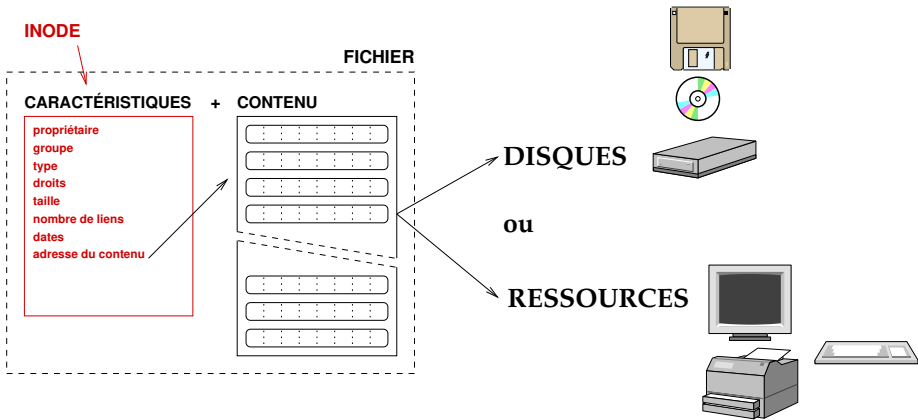
☞ Sous UNIX : **TOUT EST FICHIER**

☞ Le terme *fichier* désigne les ressources :

- matérielles (disquette, disque dur, terminal, ...)
- logicielles (image, son, texte, ...)

☞ Les primitives génériques d'accès aux fichiers permettent de réaliser des opérations de lecture/écriture sur **toutes** les ressources du système

☞ Chaque fichier est associé à une structure décrivant ses caractéristiques (*inode*)



Organisation logique

- ☞ Les *fichiers* UNIX peuvent être des fichiers disques **classiques** ou des fichiers **ressources**.
 - ☞ Plusieurs disques et de nombreuses ressources peuvent être connectés à une machine.
 - ☞ Chaque disque physique peut également être partitionné en plusieurs disques logiques.
 - ☞ À chaque fichier correspond un inode qui contient, entre-autres :
 - l'identification du disque logique du fichier
 - son numéro d'identification dans ce disque logique
- ➡ Tous les fichiers apparaissent à l'utilisateur dans une arborescence unique

Les inodes

Un *inode* est une structure qui contient les informations suivantes :

- ① identification du propriétaire et du groupe propriétaire du fichier ;
- ② type et droits du fichier ;
- ③ taille du fichier en nombre de caractères (si possible) ;
- ④ nombre de liens physiques du fichier ;
- ⑤ trois dates (dernier accès au fichier, dernière modification du fichier, dernière modification de l'inode) ;
- ⑥ adresse des blocs utilisés sur le disque pour ce fichier (pour les fichiers disques) ;
- ⑦ identification de la ressource associée (pour les fichiers spéciaux).

➡ **Chaque fichier est identifié uniquement par son inode**

La structure stat

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

issu du manuel : open(2)

Modification du propriétaire et du groupe (1)

```
int chown(const char *path, uid_t owner, gid_t group);
```

- ☞ `path` → chemin (relatif ou absolu) du fichier à tester
- ☞ `owner` → l'identifiant du nouveau propriétaire
- ☞ `group` → l'identifiant du nouveau groupe
- ➡ retourne 0 si le changement a pu être effectué et -1 sinon

plus de détails dans le manuel : `chown(2)`

**Il faut que le processus soit exécuté par root
pour que l'appel puisse réussir**

Il existe des commandes associées : `chown` et `chgrp`

plus de détails dans le manuel : `chown(1)` et `chgrp(1)`

Vérifier les droits d'accès

```
int access(const char *pathname, int mode);
```

☞ `pathname` → chemin (relatif ou absolu) du fichier à tester

☞ `mode` → droits à tester sur le fichier.

- `R_OK`, test d'accès en lecture
- `W_OK`, test d'accès en écriture
- `X_OK`, test d'accès en exécution
- `F_OK`, test d'existence du fichier

} combinable avec l'opérateur «|»

➡ retourne 0 si l'accès est autorisé et -1 sinon

plus de détails dans le manuel : `access(2)`

Exemple live

Types de fichier

Ils permettent un niveau d'abstraction de plus :

☞ **fichier réguliers**

Le contenu est une suite d'octets non structurée classique. La taille est connue et elle permet de trouver la fin du fichier sur le disque.

☞ **répertoires**

Le contenu est structuré comme une liste d'entrées

☞ **fichiers spéciaux**

Le contenu correspond à une ressource du système. Ils permettent des accès par blocs (disques, etc.) ou par octets (par *caractères*) (terminaux, imprimantes, etc.)

☞ **liens symboliques**

Le contenu est interprété comme le chemin vers un autre fichier

☞ **tubes**

Ils permettent la communication entre processus

☞ **sockets**

Ils permettent la communication au sens général

Exemple

TABLE DES INODES

Inode	Caractéristiques	Contenu
0		
1		
24801	... Répertoire ...	
24802	... Régulier ...	toto est là
24803	... Lien ...	tutu
24804	... Répertoire ...	

RÉPERTOIRE : /tmp

NOM	INODE
.	24801
..	2
toto	24802
titi	24803
tutu	24802
tata	24804

RÉPERTOIRE : /tmp/tata

NOM	INODE
.	24804
..	24801
toto	24815
foo	24802

nombre de liens physique \equiv nombre de noms

Exemple

```
bash$ ls -l /dev
```

```
brw-rw---- 1 root disk      3,  1 May  5 1998 hda1
brw-rw---- 1 root disk     3, 10 May  5 1998 hda10
brw-rw---- 1 root disk     3, 11 May  5 1998 hda11
brw-rw---- 1 root disk     3, 12 May  5 1998 hda12
brw----- 1 root floppy    2,  0 May  5 1998 fd0
drwxr-xr-x 2 root root      0 Jan  9 11:23 pts/
```

```
bash$ ls -l /dev/pts
```

```
crw--w---- 1 root tty      136,  0 Jan  9 13:19 0
crw--w---- 1 root tty      136,  1 Jan  9 13:16 1
```

Accès aux caractéristiques d'un fichier

```
int stat(const char *file_name, struct stat *buf);
```

- 👉 `file_name` → chemin (relatif ou absolu) du fichier à caractériser
- 👉 `buf` → adresse d'une zone de type `struct stat` qui sera remplie avec les caractéristiques du fichier
- ➡ retourne 0 en cas de succès et -1 en cas d'erreur

- 👉 le processus doit posséder les droits d'accès en recherche sur tous les répertoires du chemin spécifié
- 👉 le pointeur `buf` doit pointer sur une structure `stat`

plus de détails dans le manuel : stat(2)

Déterminer le type

Pour déterminer le type d'un fichier il suffit d'observer le contenu du champ `st_mode` d'une structure `stat`

Des macros, applicables à ce champ, permettent de tester facilement le type d'un fichier (technique des masques) :

- ☞ `S_ISLNK(m)` est-ce un lien symbolique ?
- ☞ `S_ISREG(m)` est-ce un fichier régulier ?
- ☞ `S_ISDIR(m)` est-ce un répertoire ?
- ☞ `S_ISCHR(m)` est-ce un fichier caractères ?
- ☞ `S_ISBLK(m)` est-ce un fichier blocs ?
- ☞ `S_ISSOCK(m)` est-ce une socket ?
- ☞ ...

Exemple live

Fichiers catalogues (répertoires)

Les données dans un fichier catalogue correspondent grossièrement à une liste de couples (entrées) :

- ☞ nom une chaîne de caractères
- ☞ numéro d'inode un entier

Les structures exactes de la liste et des entrées dépend du système de fichiers. Pour éviter les appels bas-niveaux (dépendant du système de fichiers) on utilise les fonctions de la librairie standard C.

➡ Interface indépendante du système

Les données des répertoires sont manipulées via des pointeurs vers une structure spécifique : DIR *

Ouverture

```
DIR * opendir(const char *name);
```

- ☞ `name` → chemin (relatif ou absolu) du répertoire à ouvrir
- ☞ retourne un pointeur vers un flux de répertoire ou NULL si une erreur s'est produite

- ① ouvre le répertoire
- ② alloue un espace de type DIR pour ce répertoire
- ③ positionne la structure DIR pour qu'elle représente la première entrée du répertoire
- ④ renvoie l'adresse de cet espace (ou NULL si impossible)

plus de détails dans le manuel : opendir(3)

Fermeture

```
int closedir(DIR *dir);
```

- 👉 name → chemin (relatif ou absolu) du répertoire à ouvrir
- ➡ retourne 0 si la fermeture s'est bien déroulée et -1 sinon

- ① ferme le répertoire
- ② libère l'espace utilisé par la structure DIR pour ce répertoire
- ③ renvoie 0 si tout s'est bien passé ou -1 sinon

plus de détails dans le manuel : closedir(3)

Lecture de répertoire

```
struct dirent *readdir(DIR *dir)
```

- ☞ `dir` → pointeur vers le flux de répertoire à parcourir
- ☞ retourne un pointeur vers une structure de description d'entrée de répertoire ou NULL si une erreur s'est produite

- ① lit l'entrée courante
- ② modifie le flot pour pointer vers l'entrée suivante du répertoire
- ③ retourne l'adresse de cette entrée ou NULL si la fin est atteinte

plus de détails dans le manuel : `readdir(3)`

Entrée de catalogue

```
struct dirent
{
    ino_t            d_ino;        /* numero d'inode */
    off_t            d_off;        /* offset to the next dirent */
    unsigned short int d_reclen;    /* length of this record */
    unsigned char     d_type;       /* type of file */
    char             d_name[256];  /* filename */
}
```

Exemple live

Autres manipulations de répertoires

- ☞ Création de répertoire vide

```
int mkdir(const char *pathname, mode_t mode);
```

- ☞ Suppression de répertoire vide

```
int rmdir(const char *pathname);
```

plus de détails dans le manuel : mkdir(2), rmdir(2)

Il existe des commandes associées : mkdir, rmdir

plus de détails dans le manuel : mkdir(1), rmdir(1)

⚠ Attention aux effets de bord d'autres fonctions
(la création d'un fichier modifie un répertoire par exemple)

Fichiers liens

- ☞ Les données dans un fichier lien correspondent grossièrement à un chemin vers un autre fichier
- ☞ Les fonctions «*usuelles*» suivent les liens symboliques :
Les liens sont transparents pour l'utilisateur

Création d'un lien symbolique :

```
int symlink(const char *oldpath, const char *newpath);
```

plus de détails dans le manuel : `symlink(2)`

- ☞ L'option «*-s*» de la commande «*ln*» permet également de créer un lien symbolique
- ☞ Consultation des caractéristiques du fichier lien :

```
int lstat(const char *file_name, struct stat *buf);
```

☛ La fonction `stat` appliquée à un lien ne donne pas l'information sur le fichier lien mais sur le fichier vers lequel pointe le lien

Lecture d'un lien

```
int readlink(const char *path, char *buf, size_t bufsiz);
```

- 👉 `path` → chemin du fichier lien à lire
- 👉 `buf` → l'adresse d'une zone mémoire où les octets lus seront stockés par la fonction
- 👉 `busiz` → taille de l'espace réservé à l'adresse `buf`
- ➡ retourne le nombre de caractères lus ou -1 en cas de problème

Place la valeur du lien dans le buffer `buf` qui a la taille `bufsiz`, si `buf` est trop petit, la chaîne est tronquée.

Attention la zone remplie ne comporte pas de «\0» à la fin.

plus de détails dans le manuel : `readlink(2)`

Partie D

Programmation Système (avancé)



IUT A

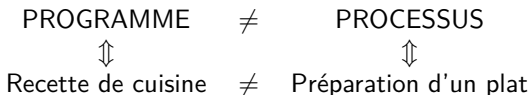
Cours n° D.1

Processus

Rappels

Un **programme** est une suite d'instructions que le système doit faire accomplir au processeur pour résoudre un problème particulier. Ces instructions sont rangées dans un fichier.

Un **processus** correspond au déroulement (*l'exécution*) d'un programme par le système dans un environnement particulier.



Définition

- ☞ Objet dynamique correspondant à l'exécution d'un programme.
- ☞ Un processus possède un espace d'adressage qui définit l'ensemble des objets qui lui sont propres (instructions et données).
- ☞ Le processus peut s'exécuter dans deux modes différents :
 - ☛ en **mode utilisateur**, le processus exécute des instructions du programme et accède aux données de son espace d'adressage.
 - ☛ en **mode noyau**, le processus exécute des instructions du noyau et a accès à l'ensemble des données du système (par exemple lors des appels système).
- ☞ Chaque processus possède un espace d'adressage de données propres, plusieurs processus peuvent partager le même programme (code réentrant).

Généralités

- ☞ Tout processus peut créer de nouveaux processus.
- ☞ Tout processus (sauf le premier) est créé par un appel à la primitive `fork`.
- ☞ La primitive `fork` a pour effet de dupliquer le processus appelant.
- ☞ Les processus sont organisés en arborescence en fonction de leur processus créateur appelé `père`.
- ☞ Le noyau du système a en charge la gestion des différents processus et le partage des ressources entre-eux, en particulier l'`ordonnancement` des processus : choisir parmi les processus en attente celui qui doit être activé.

Caractéristiques (1)

- ➡ Identité du processus (`pid`).
- ➡ Identité du père du processus (`ppid`).
- ➡ Liens avec les utilisateurs :
 - ☛ propriétaire réel/effectif du processus
 - ☛ groupe réel/effectif du processus

Remarque. Un processus ayant des droits privilégiés peut modifier ses propriétaires et groupes réels ou effectifs (procédure utilisée à la connexion d'un utilisateur).

Caractéristiques (2)

- ☞ Le répertoire de travail du processus.
- ☞ Le groupe de processus et la session auxquels le processus appartient.
- ☞ La date de création du processus.
- ☞ Les temps CPU consommés par le processus en modes utilisateur et noyau ainsi que par ses fils terminés.
- ☞ Le masque de création des fichiers.
- ☞ La table des descripteurs de fichiers.
- ☞ L'état du processus.
- ☞ L'événement attendu par le processus s'il est à l'état endormi.
- ☞ Les informations pour le traitement des signaux.
- ☞ Les verrous sur les fichiers.
- ☞ ...

Fonctions d'accès aux caractéristiques

```
pid_t getpid(void);  
pid_t getppid(void);  
uid_t getuid(void);  
uid_t geteuid(void);  
gid_t getgid(void);  
gid_t getegid(void);  
int chdir(const char *path);  
char *getcwd(char *buf, size_t size);
```

...

identité du processus
identité du processus père
identité du propriétaire réel
identité du propriétaire effectif
groupe propriétaire réel
groupe propriétaire effectif
change le répertoire courant
rép. courant

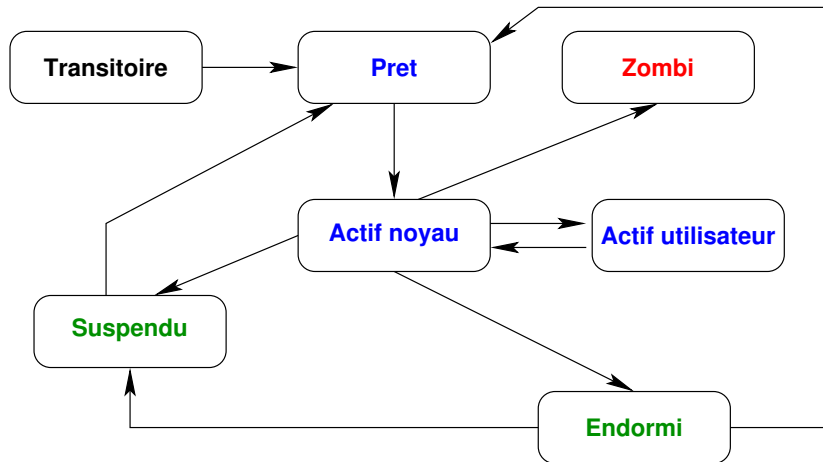
État d'un processus

Au cours de sa vie, un processus passe par différents états :

- ☞ **TRANSITOIRE**
à la création ou lors de la création d'un fils
- ☞ **PRÊT** (R)
prêt à passer en mode d'exécution
- ☞ **ACTIF** (R)
en cours d'exécution (mode noyau ou utilisateur)
- ☞ **ENDORMI** (S ou D)
en attente d'un événement (attente d'entrées/sorties, d'un signal, de la terminaison d'un processus, etc)
- ☞ **SUSPENDU** (T)
- ☞ **ZOMBI** (Z)
processus terminé mais dont le père n'a pas encore pris connaissance de la terminaison

La commande `ps` permet, entre autre, de connaître l'état d'un processus

Changements d'états



Organisation mémoire

Le processus est constitué de 4 segments mémoire :

- ① le **bloc de contrôle** qui contient les informations utiles au système (descripteurs de fichiers, signaux, identité, priorité, etc). Il a une taille fixe quel que soit le processus.
- ② les **instructions** qui appartiennent à l'espace d'adressage du processus et peuvent être partagées entre plusieurs processus si le code est réentrant. C'est un espace de taille fixe pour un programme donné. (`.text`)
- ③ les **données** manipulées par le programme qui appartiennent par excellence à l'espace d'adressage du processus et dont la taille varie au grès des allocations mémoire (`.data`, `.bss`, `.rodata`)
- ④ La **pile** dont la taille varie en fonction de l'imbrication des appels de fonctions (variables locales, sauvegarde des contextes, etc).

fork

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

- ➡ retourne 0 dans le fils, le `pid` du fils dans le père ou -1 en cas d'erreur
- ☞ permet de créer un processus fils
- ☞ après la création, les deux processus semblent avoir exécuté l'appel à la primitive `fork`, chacun des processus continue son exécution à partir de l'instruction qui suit le `fork` ;
- ☞ la valeur de retour de `fork`, permet de différencier le père du fils :
 - ☞ 0 dans le fils,
 - ☞ le `pid` du fils dans le père ;

plus de détails dans le manuel : `fork(2)`

Exemple

```
int main(void)
{
    printf("Tout seul");
    fork();
    printf("Nous sommes deux!\n");
    return 0;
}
```

Affiche :

```
$ ./testfork
```

```
Tout seul
```

```
Nous sommes deux!
```

```
Nous sommes deux!
```

Exemple live

Génétique de processus

Le processus fils hérite les caractéristiques de son père excepté :

- ☞ le pid du fils est différent de celui du père (le pid étant l'identifiant d'un unique processus) ;
- ☞ le pid du père ;
- ☞ les temps CPU (ils sont mis à 0 pour le fils) ;
- ☞ les verrous sur les fichiers ;
- ☞ les signaux pendants (voir cours sur les signaux) ;
- ☞ la priorité (la priorité est utilisée pour l'ordonnancement, la priorité du fils est initialisée à une valeur standard lors de sa création).

Terminaison des processus

- ☞ Tout processus UNIX possède une valeur de retour (valeur de retour de la fonction `main`, ou code utilisé pour la fonction `exit`) à laquelle son père peut accéder.
- ☞ Tout processus se terminant passe dans l'état **zombi** (état **Z** indiqué par commande `ps`), jusqu'à ce que son père prenne connaissance de sa terminaison.
- ☞ Le mécanisme de processus **zombi** permet à un processus d'accéder au code de retour de ses processus fils de manière asynchrone.

Remarque. Si le processus père se termine sans avoir pris connaissance de la terminaison d'un de ses fils, celui-ci est adopté par le processus de `pid 1` qui prend connaissance du code de retour du fils et lui permet ainsi de se terminer.

Primitive wait

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

- ☞ si le processus ne possède aucun fils, la fonction retourne -1 ;
- ☞ si le processus possède des fils qui ne sont pas en l'état zombi, le processus est bloqué jusqu'à passage d'un des fils dans l'état zombi (ou réception d'un signal) ;
- ☞ si le processus possède au moins un fils zombi, la fonction retourne le pid de l'un des fils zombi **choisit par le système !** Dans ce cas :
 - le processus zombi considéré disparaît de la liste des processus (maintenue par le système),
 - si l'adresse status est différente de NULL, elle reçoit les informations sur la terminaison du processus zombi.

Primitive waitpid

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

- ☞ permet de tester la terminaison d'un processus spécifié en bloquant ou non le processus appelant ;
- ☞ le paramètre `pid` peut avoir les valeurs suivantes :
 - `<-1` tout processus fils dans le groupe `|pid|`,
 - `-1` tout processus fils,
 - `0` tout processus fils du même groupe que l'appelant,
 - `>0` processus d'identité `pid` ;
- ☞ le paramètre `options` est une combinaison (|) des constantes `WNOHANG` (appel non bloquant), et `WUNTRACED` ;
- ☞ la fonction retourne :
 - `-1` en cas d'erreur,
 - `0` en cas d'échec en mode non bloquant (le processus demandé existe mais n'est ni bloqué ni stoppé),
 - le `pid` du fils en cas de succès.

Interprétation du code de retour

La valeur reçue par l'intermédiaire de la fonction `wait` n'est **PAS** directement la valeur retournée par le processus fils ! Les fonctions suivantes doivent être utilisées pour tester cette valeur (entière) :

- ☞ `WIFEXITED(mystat)` vrai si le fils s'est terminé normalement ;
- ☞ `WEXITSTATUS(mystat)` fournit le code de retour du fils (les 8 bits de poids faible) uniquement s'il s'est terminé normalement ;
- ☞ `WIFSIGNALED(mystat)` vrai si le fils s'est **terminé** à cause d'un signal ;
- ☞ `WTERMSIG(mystat)` retourne le numéro du signal ayant provoqué la terminaison du fils si le fils s'est **terminé** à cause d'un signal ;
- ☞ `WIFSTOPPED(mystat)` vrai si le fils est **stoppé** (utilisable uniquement avec l'option `WUNTRACED`) ;
- ☞ `WSTOPSIG(mystat)` retourne le numéro du signal qui a stoppé le fils si le fils a été **stoppé** par un signal.

Exemple live

Commandes exec*

```
int execvp(const char *file, char *const argv[]);
```

- ☞ recouvre le programme en cours qui exécute l'appel par le programme mentionné en premier paramètre ;
- ☞ le paramètre `argv` représente le tableau d'arguments du programme à appeler (`argv[0]` doit être le nom du programme et le tableau doit se terminer par `NULL`) ;
- ☞ il n'y a pas de création de processus, le processus poursuit son exécution mais exécute désormais le nouveau programme ;
- ☞ retourne `-1` en cas d'erreur ;
- ☞ si l'appel n'a pas déclenché d'erreur, **il n'y a pas de retour** ;
- ☞ la particularité de `execvp` est de rechercher le fichier à exécuter dans les répertoires contenus dans le `PATH` (comme le fait le `shell`) s'il n'y a pas de `'/'` dans le nom de fichier spécifié.

Remarque. Voir aussi les autres fonctions de la famille par `man exec`.

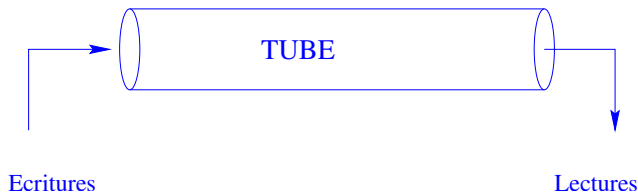
Exemple live

Cours n° D.2

Communication Inter-processus, les tubes

Généralités (1)

- ➡ Les tubes sont des mécanismes permettant aux processus de communiquer entre-eux.
- ➡ Les tubes appartiennent au système de fichiers UNIX, *i.e.* ils sont décrits par un i-nœud.
- ➡ Les tubes peuvent donc être manipulés par l'intermédiaire de descripteurs de fichiers et par exemple des primitives d'entrées/sorties bas niveau `read` et `write`.
- ➡ Les tubes sont des moyens de communication unidirectionnels.



Généralités (2)

- ☞ Un tube correspond au plus à deux entrées dans la table des fichiers ouverts (une entrée en lecture et une entrée en écriture).
- ☞ Les données ne sont pas formatées, elles apparaissent comme un flot de caractères. Le tube est géré en file, *i.e.* la première donnée écrite dans le tube est également la première donnée lue.
- ☞ Attention, un tube a une capacité **finie** !
- ☞ Le nombre de lecteurs d'un tube est le nombre de descripteurs associés à l'entrée en lecture sur le tube. Si ce nombre est nul, il est impossible d'écrire dans le tube.
- ☞ Le nombre de rédacteurs d'un tube est le nombre de descripteurs associés à l'entrée en écriture sur le tube. Si ce nombre est nul, la primitive `read` détecte une **fin de fichier**.

Création d'un tube

La primitive

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

- ✎ permet de créer un tube ;
- ✎ à pour paramètre un tableau de deux entiers qui va permettre de stocker les deux descripteurs ;
- ✎ range dans `fildes[0]` le descripteur permettant de lire dans le tube et dans `fildes[1]` le descripteur permettant d'écrire dans le tube ;
- ✎ retourne 0 en cas de succès et -1 en cas d'erreur.

Tubes vs fichiers

- ☞ Comme les tubes n'ont pas de noms, il est impossible de les ouvrir grâce à la primitive `open`. En conséquence, un processus peut acquérir un descripteur sur un tube, soit en le créant, soit par héritage.
- ☞ Seuls le processus ayant créé le tube et sa descendance peuvent y accéder. Si un processus perd son accès au descripteur sur le tube (par exemple par un appel à `close`), il n'a aucun moyen de le récupérer par la suite.
- ☞ Les accès en lecture sont effectués par des appels à `read` et `write`, comme pour les fichiers.
- ☞ Les lectures dans un tube sont « effaçantes », il s'agit bien d'une extraction des données du tube et non d'une consultation (des fonctions comme `lseek` n'ont aucun sens dans le cas des tubes).

Exemple live

Pourquoi fermer les descripteurs ?

```
int main (void) {
    int pid;
    int fd[2];

    if (pipe(fd) == -1) {
        perror("Creation du tube ");
        exit(1);
    }
    pid = fork();
    if (pid == 0) {
        printf("Processus fils de pid %d\n",getpid());
        close(fd[0]);
        write(fd[1], "Bonjour", 8);
        close(fd[1]);
    }
```

Pourquoi fermer les descripteurs ?

```
else {  
    char mot [9];  
    int n;  
    printf("Processus pere de pid %d\n",getpid());  
    close(fd[1]);  
    while ((n = read(fd[0], mot, 8)) != 0) {  
        mot[n] = '\\0';  
        printf("Pere >> mot recu %s\n",mot);  
    }  
    printf("Pere >> terminaison\n");  
}  
return 0;  
}
```

Pourquoi fermer les descripteurs ?

☛ avec l'instruction `close(fd[1]);` :

```
bash$ a.out
Processus pere de pid 20920
Processus fils de pid 20921
Pere >> mot reçu : Bonjour
Pere >> terminaison
bash$
```

☛ sans l'instruction `close(fd[1]);` :

```
bash$ a.out
Processus pere de pid 20928
Processus fils de pid 20929
Pere >> mot reçu : Bonjour
```

☛ le processus ne se termine pas !

Lecture sur un tube

- ☞ L'appel à la primitive `read` est, par défaut, bloquant (*i.e.* le processus effectuant l'appel à `read` est arrêté jusqu'à ce qu'il y ait quelque chose à lire dans le tube).

- ☞ Comme dans le cas des fichiers, un appel

```
read(fd[0], buffer, size)
```

demande la lecture de `size` octets qui seront placés à l'adresse `buffer`. La valeur retournée est le nombre d'octets effectivement lus (*i.e.* disponibles dans le tube au moment de la lecture).

- ☞ Si le nombre d'écrivains est nul, `read` détecte la fin de fichier et retourne 0.

**Il faut toujours fermer tous les descripteurs inutiles
dès que possible !!!**

Écriture dans un tube

L'écriture est réalisée par appel à la primitive `write`.

- ☞ Si le nombre de lecteurs est nul, le rédacteur reçoit un signal SIGPIPE (et se termine si le signal n'est pas traité).
- ☞ Si le nombre de lecteurs n'est pas nul, le processus écrit les données dans le tube.

```
int main (void) {  
    int pid;  
    int fd[2];  
    if (pipe(fd) == -1) {  
        perror("Creation du tube ");  
        exit(1);  
    }  
    close(fd[0]);  
    write(fd[1], "Bonjour", 7);  
    close(fd[1]);  
    printf("Terminaison\n"); return 0;  
}
```

```
bash$ a.out  
Broken pipe
```

Les tubes dans les commandes

```
bash$ ls /etc | more
```

```
DIR_COLORS
```

```
HOSTNAME
```

```
X11/
```

```
a2ps-site.cfg
```

```
a2ps.cfg
```

```
adjtime
```

```
--More--
```

👉 frappe d'un Ctrl-z par l'utilisateur

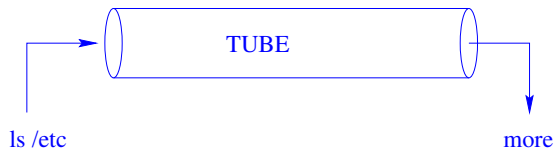
```
Suspended
```

```
[7] 21369 21370
```

```
bash$ fg
```

```
ls -F /etc | more
```

```
aliases
```



Primitive dup

```
int dup(int oldfd);
```

- ☞ crée une copie d'un descripteur de fichier ;
- ☞ retourne le nouveau descripteur (le plus petit libre) ;
- ☞ retourne -1 en cas d'erreur ;
- ☞ l'ancien descripteur et le nouveau peuvent être utilisés de manière interchangeable, ils partagent la position courante dans le fichier.

Exemple live - Redirection de la sortie standard

Les tubes nommés

Les **tubes nommés** ou **fifo** ont été introduits dans la version III d'UNIX.

- ☞ Leur but est de permettre à des processus sans lien de parenté particulier de communiquer par l'intermédiaire de tubes.
- ☞ Ils ont toutes les caractéristiques des tubes et ont en plus une référence dans le système de fichiers.
- ☞ Tout processus connaissant la référence d'un tube peut l'ouvrir avec la primitive `open`.
- ☞ Les fichiers correspondant à des tubes sont identifiés par `ls`.

Exemple.

```
bash$ ls -l
```

```
...
```

```
prw-r--r--  1 hauspiem infoens      0 2008-11-19 17:29 tube
```

Création d'un tube nommé

La primitive :

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

- ☞ permet de créer un tube nommé ;
- ☞ `pathname` désigne le nom (nom relatif ou nom absolu) du fichier associé au tube ;
- ☞ `mode` désigne le mode du fichier ainsi créé (comme pour `open`) ;
- ☞ attention, le tube est simplement **créé**, il faut ensuite **l'ouvrir** pour pouvoir l'utiliser.

Utilisation d'un tube nommé

- ☞ Un processus connaissant le « nom » d'un tube peut l'ouvrir grâce à la primitive `open` (si les droits du tube le permettent).
- ☞ Attention, `open` est dans ce cas bloquant (une ouverture en lecture est bloquante tant qu'il n'y a aucun écrivain sur le tube et vice et versa).
- ☞ Le descripteur de fichier obtenu par `open` peut ensuite être utilisé comme tout autre descripteur.
- ☞ Il ne faut pas oublier de supprimer les fichiers associés aux tubes lorsque l'on en a plus besoin :

```
int unlink(const char *pathname);
```

permet de supprimer « proprement » `pathname` du système de fichiers (lire `man 2 unlink`).

Exemple live

Cours n° D.3

Communication Inter-processus, les signaux

Signal

- ☞ Un signal est une **sonnerie d'alerte** qui est *entendue* par un processus
- ☞ Il signale un événement particulier
- ☞ Un signal peut être envoyé :
 - ▷ par une commande
 - ▷ par un appel système dans un programme
- ⇒ les signaux sont des moyens de communiquer avec les processus
- ☞ Chaque signal prévient d'un événement différent.
- ☞ Un signal peut être déclenché *artificiellement* même si l'événement ne s'est pas produit.
 - ⇒ La seule information dont le processus recevant le signal dispose est le nom de celui-ci, il ne possède aucun moyen de savoir si l'événement associé s'est réellement produit.

⇒ Le signal peut être vu comme une interruption logicielle

État des signaux

Lorsqu'un signal est envoyé à un processus, celui-ci doit le gérer correctement. Un signal peut être dans différents états :

- ✎ un signal **pendant** est un signal qui a été envoyé à un processus mais qui n'a pas encore été pris en compte par celui-ci. **Attention**, un seul signal de chaque type peut être pendant, si un deuxième signal du même type arrive, il est perdu (pour les signaux « classiques ») ;
- ✎ un signal est **délivré** au processus lorsque celui-ci en prend connaissance. Le processus exécute alors la fonction de traitement du signal ;
- ✎ un signal peut être **bloqué** ou **masqué**, dans ce cas, il ne sera pas délivré au processus.

Noms des signaux

- ✎ Pour augmenter la portabilité des applications, les signaux sont nommés par des constantes ayant un nom « évocateur » (par exemple SIGTERM ou SIGKILL).
- ✎ Les signaux peuvent provenir d'un événement extérieur au processus, par exemple envoyés par un utilisateur ou par un autre processus, ou provenir d'un événement intérieur au processus ayant provoqué une erreur (division par zéro, violation d'une zone mémoire, ...).

Exemple. L'envoi du signal SIGSEGV à un processus provoque sa terminaison ainsi que la création d'un fichier core. Il est utilisé lors d'une violation mémoire par le processus mais peut aussi être envoyé par l'utilisateur sans qu'aucune violation de la mémoire ne se soit produite.

Principaux signaux

- ☞ **SIGILL** détection d'une instruction illégale
- ☞ **SIGKILL** signal de terminaison
- ☞ **SIGTERM** signal de terminaison
- ☞ **SIGUSR1** signal réservé utilisateur
- ☞ **SIGUSR2** signal réservé utilisateur
- ☞ **SIGCHLD** signal de terminaison d'un fils
- ☞ **SIGSTOP** signal de suspension
- ☞ **SIGCONT** signal de continuation pour un processus suspendu.

liste complète avec la commande : `kill -l`

plus de détails dans le manuel : `signal(7)`

Envoyer un signal : kill()

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

- 👉 pid → désigne le(s) processus à prévenir. Si pid
 - ▷ > 0, il désigne un processus ;
 - ▷ 0, il désigne le groupe auquel le processus courant appartient,
 - ▷ -1, il désigne tous les processus,
 - ▷ < -1, il désigne tous les processus du groupe |pid| ;
- 👉 sig → le signal à envoyer.
- ➡ retourne 0 en cas de succès et -1 en cas d'échec

Attention. Un processus ne peut envoyer un signal qu'à un processus appartenant au même utilisateur.

plus de détails dans le manuel : kill(2)

Envoyer un signal : kill

```
kill [-signal] pid ...
```

- ☞ permet d'envoyer un signal spécifique à un processus ou à un groupe de processus ;
- ☞ par défaut, le signal TERM est envoyé ;
- ☞ les arguments pid peuvent être :
 - le pid d'un processus,
 - le numéro d'un job,
 - -1, le signal s'adresse à tous les processus,
 - -n, le signal s'adresse à tous les processus du groupe |n| ;
- ☞ signal désigne un signal par son nom (ex : KILL) ou par son numéro (ex : 9).

Attention. Un utilisateur ne peut envoyer un signal qu'à un processus lui appartenant.

Réception d'un signal

- ➡ À chaque signal est associé un **handler** (une fonction) définissant le comportement d'un processus recevant ce signal, le handler par défaut est désigné par `SIG_DFL`.
- ➡ Le comportement dépend du type de signal reçu :
 - ☛ terminaison du processus (ex : `SIGKILL`);
 - ☛ terminaison du processus avec création d'un fichier core (ex : `SIGSEGV`);
 - ☛ signal ignoré (ex : `SIGCHLD`);
 - ☛ suspension du processus (ex : `SIGSTOP`);
 - ☛ reprise d'un processus suspendu (ex : `SIGCONT`).
- ➡ Le handler `SIG_IGN` est également prédéfini : son action est simplement d'ignorer un signal.

Traitement d'un signal

À la prise en compte d'un signal par un processus, le traitement associé à ce signal est exécuté, le déroulement normal du processus est donc interrompu et reprend éventuellement après traitement du signal.

- ☞ Le signal est délivré au processus lorsque celui-ci passe du mode noyau au mode utilisateur, le processus ne peut pas être interrompu lorsqu'il se trouve en mode noyau.
- ☞ Si le processus est endormi à un niveau de priorité interruptible (ex : attente de signal), le processus passe dans l'état prêt et il reçoit le signal lorsqu'il repasse en état actif. Si le signal est ignoré, le processus reprend le cours de son exécution (il peut éventuellement se rendormir) sinon, après le traitement du signal, l'appel système sur lequel le processus était en attente renvoie -1.
- ☞ Si le processus est stoppé, SIGKILL et SIGTERM le terminent, SIGCONT le réveille, les autres signaux lui sont délivrés au réveil.

Mise en place d'un handler

Il est possible de modifier le handler d'un signal pour le processus courant

```
int sigaction(int signum,  
              const struct sigaction *act,  
              struct sigaction *oldact);
```

- ☞ `signum` → le signal pour lequel le handler doit être installé
- ☞ `act` → l'action qui doit être installée pour le traitement du signal
- ☞ `oldact` → l'action précédente utilisée pour ce signal est sauvegardée dans `oldact`
- ➡ retourne 0 en cas de succès et -1 en cas d'échec

plus de détails dans le manuel : `sigaction(2)`

struct sigaction

```
struct sigaction
{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

- ☞ le champ `sa_handler` désigne l'« action » qui est associée au signal, il peut être `SIG_DFL` pour l'action par défaut, `SIG_IGN` pour ignorer le signal ou un pointeur vers la fonction à appeler pour traiter le signal ;
- ☞ le champ `sa_mask` donne le masque des signaux qui doivent être bloqués pendant le traitement, par défaut, le signal associé à ce traitement est bloqué, le type `sigset_t` désigne un ensemble de signaux ;

- ☞ le champ `sa_flags` indique des options qui vont modifier le traitement du signal (à combiner avec des « ou » bit à bit) :
 - ☞ `SA_ONESHOT` ou `SA_RESETHAND` restaurent le comportement par défaut dès que le handler a été appelé,
 - ☞ `SA_NOMASK` ou `SA_NODEFER` permettent que le signal soit reçu pendant l'exécution de son propre traitement,
 - ☞ ...

Manipulation des ensembles de signaux

Un certain nombre de fonctions permettent de manipuler les signaux :

- ☞ `int sigemptyset(sigset_t *set);` initialise à « vide » un ensemble de signaux;
- ☞ `int sigfillset(sigset_t *set);` initialise à « plein » un ensemble de signaux;
- ☞ `int sigaddset(sigset_t *set, int signum);` ajoute un signal à un ensemble;
- ☞ `int sigdelset(sigset_t *set, int signum);` efface un signal d'un ensemble;
- ☞ `int sigismember (const sigset_t *set, int signum);` teste si un signal appartient à un ensemble.

Chacune retourne :

- ☞ -1 en cas d'erreur;
- ☞ 0 en cas de succès sauf `sigismember` qui retourne 1 si le signal appartient à l'ensemble et 0 sinon.

Écriture d'un handler

Ne pas mettre n'importe quoi dans un gestionnaire de signal :

- ✎ en C ansi – modification de variables globales de type `sig_atomic_t` déclarées volatile ;
- ✎ en utilisant le blocage des signaux, il est possible d'accéder à différents types de données ;
- ✎ appels de fonctions, appels système : si la fonction n'est pas réentrante :
 - ➡ problèmes en perspective !!!
 - ➡ vérifier dans les pages de manuel si les fonctions utilisent des variables statiques,
 - ➡ quelques appels systèmes réentrants – sleep, open, read, write, close, access, stat, fork, getpid, wait, waitpid, pipe, dup, kill, sigaction,
 - ➡ **ne jamais** utiliser malloc.

Bloquage de signaux

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- ☞ Permet de bloquer ou débloquent la réception d'un ensemble de signaux par le processus courant ;
- ☞ `how` définit s'il faut bloquer ou débloquent les signaux de l'ensemble `set` ;
 - `SIG_BLOCK` : bloque les signaux de l'ensemble `set`,
 - `SIG_UNBLOCK` : débloquent les signaux de l'ensemble `set`,
 - `SIG_SETMASK` : définit l'ensemble des signaux bloqués comme étant l'ensemble `set`.

Exemple live