LI356 - Cours
Programmation POSIX – Système

Luciana Arantes et Olivier Marin

Le 3 avril 2011

LI356 – Programmation POSIX/ système

LI356 – Programmation POSIX/ système

- **Responsable : Olivier Marin**
- **Cours : Luciana Arantes et Olivier Marin**
- **TD et TME : Olivier Marin , Mesaac Makpangou et Julien Sopena**
- **Evaluation:**
  - Partiel (30%), TME(10%) Examen (60%).

LI356 – Programmation POSIX/ système

- **Contenu UE:**
  - Introduction
    - Compilation (makefile +gcc), rappels C + Norme POSIX
  - Gestion de Processus
    - Fork, exec, wait
  - Signaux
  - Entrée/Sortie
  - Tubes
  - IPC POSIX
  - Gestion du Temps

Bibliographie

- **L.M. Rifflet et J.B. YUNES.** ***Unix: Programmation et Communication***
- **C. Blaess.** ***Programmation système en C sous Linux.***
- **W. Richards Steven.** ***Advanced Programming un the Unix Environment***

Cours 1 : Introduction

- Définir C / POSIX
- La Norme POSIX
- Rappels
  Makefile
  Environnement de programmation
- Bibliographie

Définir C / POSIX

ou *Pourquoi POSIX ?*

Définir C / POSIX

- **Langage C : une première définition**

"Un langage :
- *structuré,*
- *généralement compilé,*
- *de *haut* niveau"*

Définir C / POSIX

- **Langage C : les origines**

Travaux de Ken Thompson, Dennis Ritchie, Brian Kernighan
- Projet d'OS basé sur Multics
  accès simple aux périphériques
  multi-utilisateurs
  code utilisateur **indépendant du hardware**, i.e. **portable**
- Résultat : UNIX
  Noyau + Compilateur C : **Bibliothèque "standard"**

Définir C / POSIX

- **Langage C : définition *enrichie***

  "Un langage de programmation dont la structure est proche de la machine de von Neumann."

Bibliothèque "standard"

Strict minimum pour interagir avec la machine

ex.: manipulation texte/fichiers, gestion mémoire, gestion du processus, gestion d'événement, mesure du temps de calcul

Tout le reste :

- à la charge du programmeur

Définir C / POSIX

- **Historique Language C**

CPL/BCPL	Travaux MIT, Londres, Cambridge Non typé
B	Adaptation pour *Unics*, Thompson, 1969-1972
NB	Introduction du typage
C	Adaptation pour *Unix*, Ritchie, 1972-1973 Générateur de code pour NB Ajout de fonctionnalités : structures, unions, énumérations définition de type, changement de type (cast)

Définir C / POSIX

- **UNIX :**

  Succès entraîne une multiplication des versions

  UNIX Wars : Berkeley, AT&T, OSF, Novell, X/Open, ...

  Chacune se voit comme *LE* standard

  ⇒ Brouillage de la notion de standard, fin de la portabilité...

- **Retour à la raison : normalisation**

  POSIX : norme ISO "non propriétaire"

  comités indépendants, représentatifs d'un grand nombre de utilisateurs

La Norme POSIX

La Norme POSIX

POSIX : principe

Portable Operating System Interface for Computing Environments

Document de travail

Produit par IEEE

Endossé par ANSI et ISO

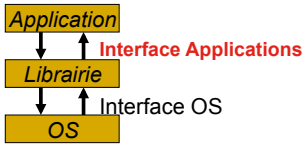
API standard pour applications

Définitions de services

définition du comportement attendu lors d'un appel de service

Portabilité **garantie** pour les codes sources applicatifs qui l'utilisent

contrat *application* / *implémentation* (système)



Macro _POSIX_SOURCE

#define **_POSIX_SOURCE**

La Norme POSIX

POSIX : En fait, un ensemble de standards (IEEE 1003.x)

Chaque standard se spécialise dans un domaine

Entre autres :

1003.1 (POSIX.1) System Application Program Interface (kernel)

1003.2 Shell and Utilities

1003.4 (POSIX.4) Real-time Extensions

1003.7 System Administration

Divisé en sections, 2 catégories de contenu :

- Bla-bla (Préambule, Terminologie, Contraintes, ...)
- Regroupements de services par thème

  Pour chaque service, une définition d'interface

  (*Synopsis, Description, Examples, Returns, Errors, References)*

La Norme POSIX

Sections du standard POSIX.1

1. **General (Scope, Normative References, Conformance)**
2. **Terminology and General Requirements (Conventions, Definitions, General Concepts, Error Numbers, Primitive System Data Types, Environment Description, C Language Definitions, Numerical Limits, Symbolic Constants)**
3. **Process Primitives (Process Creation and Execution, Process Termination, Signals, Timer Operations)**
4. **Process Environment (Process Identification, User Identification, Process Groups, System Identification, Time, Environment Variables, Terminal Identification, Configurable System Variables)**
5. **Files and Directories (Directories, Working Directory, General File Creation, Special File Creation, File Removal, File Characteristics, Configurable Pathname, Variables)**

La Norme POSIX

Sections du standard POSIX.1 (suite)

6. **Input and Output Primitives (Pipes, File Descriptor Manipulation, File Descriptor Deassignment, Input and Output, Control Operations on Files)**
7. **Device- and Class-Specific Functions (General Terminal Interface, General Terminal Interface Control Functions)**
8. **Language-Specific Services for the C-Programming Language (Referenced C Language Routines, C Language Input/Output Functions, Other C Language Functions)**
9. **System Databases (System Databases, Database Access)**
10. **Data Interchange Format (Archive/Interchange File Format)**

La Norme POSIX

Exemple de définition d'interface

NAME

getpid - get the process ID

SYNOPSIS

```
#include <unistd.h>
pid_t getpid(void);
```

DESCRIPTION

The *getpid*() function shall return the process ID of the calling process.

RETURN VALUE

The *getpid*() function shall always be successful and no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

SEE ALSO

exec(), *fork*(), *getpgrp*(), *getppid*(), *kill*(), *setpgid*(), *setsid*(), the Base Definitions volume of IEEE Std 1003.1-2001, *<sys/types.h>*, <unistd.h>

Rappels

Compilation

Rappels : Compilation

Fichier *Makefile* (ou *makefile*)

Constitué de plusieurs règles de la forme

<cible>: <liste_prérequis>

<commandes>

NB : chaque commande est précédée d'une tabulation

Prérequis

<nom_fichier> Le fichier est-il présent ?

<nom_cible> La règle est-elle vérifiée ?

Evaluation d'une règle en 2 étapes

1. Analyse des prérequis (processus récursif)
2. Exécution des commandes

Rappels : Compilation

Exécution d'un programme C : Traducteurs

1. préprocesseur C (cpp)
   substitutions textuelles : constante, macro fonctions, commentaires, inclusion de fichiers, compilation conditionnelle
2. compilateur C (cc1)
   phases : analyses, génération, optimisation
3. assembleur (as)
4. éditeur de liens (ld)

gcc

5. gérant de bibliothèques (ar : ARchiver)
6. interpréteur (bash, …)
7. Bibliothèque libc : bibliothèque standard (partagée)
8. Noyau: gérant de processus, gérant mémoire, …
9. outils d'aide à la mise au point (gdb et fichier de nom 'core', trace, …)

Préprocesseur cpp

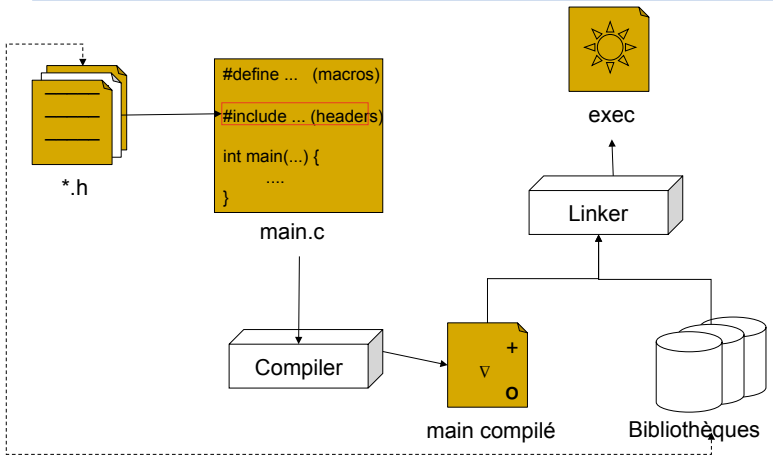
- **permet au programmeur d'inclure des fichiers source dans d'autres fichiers source, de définir des macros, et de procéder à des compilations conditionnelles.**
  - #define, #undef , #include , #include, #ifdet, etc.
  - Exemple compilation conditionnelle :
    - #ifdef AFFICHAGE
      printf ("valeur : %d\n , var);
      #endif
    - Pour l'affichage, il suffit alors de définir AFFICHAGE
      - #define AFFICHAGE
      - -DAFFICHAGE (ligne commande gcc)

Rappels : Compilation

Quelques extensions de noms de fichiers

.c	Source C
.i	Source C prétraité
.h	Fichier préprocesseur (h = header)
.s, .S	Source Assembleur
.o, .so	Fichier objet partagé
.a	Fichier archive (bibliothèque statique)

Rappels : Compilation



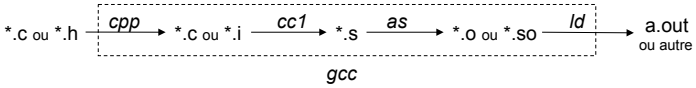
Rappels : Compilation

- **Le préprocesseur *cpp* traite le code source C (*.c*) du programmeur en interprétant les instructions préprocesseur et délivre du code source C pur.**
- **Le compilateur proprement dit, *ccl,* traite le code source C produit par *cpp* et délivre du code assembleur (*.as*)**
- **l'assembleur as traduit le code assembleur produit par *ccl* et délivre du code objet (*.o*)**
- **enfin, l'éditeur de liens *ld* lie les différents modules objets produits par as et délivre du code exécutable (*a.out*)**

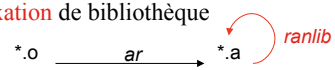
Rappels : Compilation

Quelques maillons de la chaîne de développement

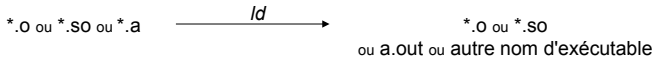
1. Génération d'exécutable



2. Génération et indexation de bibliothèque



3. Edition de liens



Rappels : Compilation

Quelques options de `gcc`

-ansi	Assurer le respect du standard ANSI
-Wall (Warning)	Afficher tous les avertissements générés
-c (cpp + cc1 + as)	Omettre l'édition de liens
-g	Produire des informations de déboguage
-D (Define)	Définir une macro
-M (Make)	Générer une description des dépendances de chaque fichier objet
-H (Header)	Afficher le nom de chaque fichier header utilisé
-I (Include)	Etendre le chemin de recherche des fichiers headers (`/usr/include`)
-L (Library)	Etendre le chemin de recherche des bibliothèques (`/usr/lib`)
-l (library)	Utiliser une bibliothèque (`lib<nom_librairie>.a`) durant l'édition de liens
-o (Output)	Rediriger l'output dans un fichier

Rappels : Compilation

Exemple : projet "HelloWorld"

Rappels : Compilation

Exemple : fichier 'hello.c'

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

void Hello(void){
        printf("Hello World\n");
}
```

Rappels : Compilation

Exemple : fichier 'hello.h'

```
#ifndef H_GL_HELLO
#define H_GL_HELLO

void Hello(void);

#endif
```

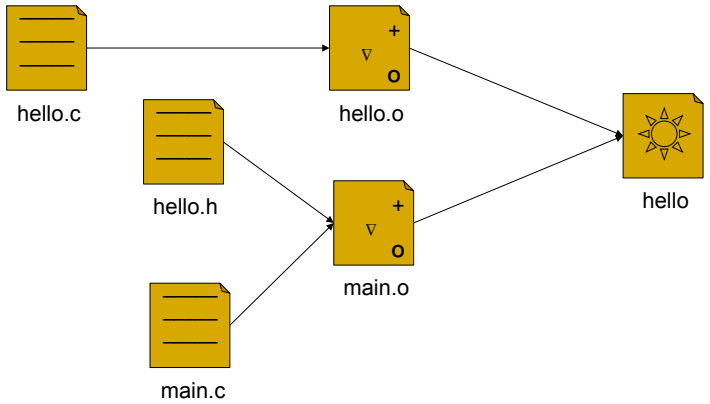

Rappels : Compilation

Exemple : fichier 'main.c'

```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <stdlib.h>
#include "hello.h"

int main(int argc, char * argv[] ){
    Hello();
    return EXIT_SUCCESS;
}
```


Rappels : Compilation

Exemple : projet "HelloWorld" - dépendances



Rappels : Compilation

Exemple "HelloWorld" : fichier 'makefile' minimal

```
hello: hello.o main.o
    gcc -o hello hello.o main.o

hello.o: hello.c
    gcc -o hello.o -c hello.c -Wall -ansi

main.o: main.c hello.h
    gcc -o main.o -c main.c -Wall -ansi
```

Rappels : Compilation

Exemple "HelloWorld" : fichier 'makefile' enrichi - règles complémentaires

```
all: hello

hello: hello.o main.o
	gcc -o hello hello.o main.o

hello.o: hello.c
	gcc -o hello.o -c hello.c -Wall -ansi

main.o: main.c hello.h
	gcc -o main.o -c main.c -Wall -ansi

clean:
	rm -rf *.o hello
```


Rappels : Compilation

Exemple "HelloWorld" : fichier 'makefile' enrichi - variables personnalisées

```
CC=gcc
CFLAGS=-Wall -ansi
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
	$(CC) -o hello hello.o main.o

hello.o: hello.c
	$(CC) -o hello.o -c hello.c $(CFLAGS)

main.o: main.c hello.h
	$(CC) -o main.o -c main.c $(CFLAGS)

clean:
	rm -rf *.o $(EXEC)
```


Rappels : Compilation

Exemple "HelloWorld" : fichier 'makefile' enrichi - variables internes

```
CC=gcc
CFLAGS=-Wall -ansi
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
	$(CC) -o $@ $^

hello.o: hello.c
	$(CC) -o $@ -c $< $(CFLAGS)

main.o: main.c hello.h
	$(CC) -o $@ -c $< $(CFLAGS)

clean:
	rm -rf *.o $(EXEC)
```

$@	*target name*
*$^*	*list of dependencies*
$<	*name of 1st dependency*

Bibliothèque statique

- **Le code du fichier objet est inclus dans l'exécutable.**
- **Option –l*bib:***
  - indique au éditeur de lien de rechercher les symboles dans la bibliothèque nommée lib*bib.a*
- **Création d'une bibliothèque (archive)**
  - **ar** -rs <nom-bibliothèque> <fichiers>
    - L'option *r* indique à *ar* d'ajouter les fichiers dans la bibliothèque et d'en créer une nouvelle, si nécessaire.
    - L'option -*s* génère l'index de l'archive.
    - Autres options:
      -u : mise à jour des fichiers
      - x : suppression des fichiers
  - **ranlib** génère un index du contenu d'une archive, et le stocke dans l'archive.
    - **ranlib** équivaut **ar** -s
  - **nm** –s : lister l'index

Bibliothèque Partagée

- **Les bibliothèques partagées ne sont pas des archives mais des objets relogeables**
  - Bibliothèque utilisée par un programme exécutable, mais n'en faisant pas partie.
    - Shared objects (.so).
  - Plusieurs programmes peuvent utiliser la même bibliothèque.
    - Le binaire généré contient des symboles non-définis mais il sait dans quelle bibliothèque les trouver.
    - Edition de liens sera lancé dynamiquement lorsque le programme sera exécuté. Il localise les bibliothèques partagées et les charge en mémoire.
  - **Avantages:**
    - Le code des fonctions ne sera donc plus dupliqué dans chaque exécutable.
    - Moins d'espace en disque.

Bibliothèque Partagée

- **Création d'une bibliothèque partagée:**
  - Les fichiers doivent être compilés avec l'option *-fPIC* (*Position Independent Code*).
    - compile sans indiquer d'adresse mémoire dans le code
  - La bibliothèque partagée est créée en utilisant l'option *-shared*
- **Exécution**
  - /usr/lib ou /lib (répertoire par défaut)
  - indiquer le chemin vers le répertoire de la bibliothèque pour que celle-ci soit prise en compte par le chargeur.
    - Options gcc: -Wl,-rpath,$HOME/lib
      - *-Wl* : options séparées par virgules sont passé à l'éditeur de lien
      - *-rpath*: répertoire de la bibliothèque

Rappels

Environnement de programmation

Rappels : Environnement de Programmation

Récupération d'arguments

Ligne de commande : `<nom_programme> <liste_arguments>`

ex. : `> myprog toto /usr/local 12`

Copiée par l'OS dans une zone mémoire accessible au processus
En C, récupération au niveau du `main`

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

`argc`	nombre d'arguments (nom du programme inclus)
`argv`	tableau d'arguments (`argv[0]` = nom du programme)

ex. :	`argc`	*4*
	`argv[0]`	*"myprog"*
	`argv[3]`	*"12"*

Rappels : Environnement de Programmation

Récupération des variables d'environnement

Environnement d'un processus :
ensemble d'informations indispensables à son exécution
ex. : `PATH` *emplacement des exécutables*
`PRINTER` *identifiant de l' imprimante par défaut*

En C, accessibles à partir du `main`

```
main(int argc, char *argv[], char **env)
```

`env[i]` couple "<nom_variable>=<valeur>"
ex. : `"PRINTER=rocky.lip6.fr"`
dernier élément du tableau : NULL

Voir aussi les fonctions `setenv` et `getenv` (`stdlib`)
- Environnement hérité du père

Rappels : Environnement de Programmation

Flux standards

Notion de flux standards déjà ouverts

`stdin`	entrée standard	(*par défaut : clavier*)
`stdout`	sortie standard	(*par défaut : écran terminal*)
`stderr`	sortie erreur	(*par défaut : écran terminal*)

Redirection en ligne de commande

`<`	redirection `stdin`
`>` (ou `1>`)	redirection `stdout`
`2>`	redirection `stderr`

ex. : `myprog < myinputfile > r esults/myresults 2 > errors/myerrors`

Entrées et sorties non standards (cf. cours 5 & 6)

Rappels : Environnement de Programmation

Numéros d'erreur (Error Numbers _ cf. POSIX.1 - Section 2.4)

Chaque appel de fonction peut (ou non) générer un numéro d'erreur (> 0)
Récupéré dans la variable externe `errno`
Conservé jusqu'au prochain appel

Constantes symboliques définies dans le standard
ex. : `[EACCES]`, `[ECHILD]`, `[EEXIST]`, `[EINVAL]`
D'autres peuvent être définies dans le système

Bibliographie

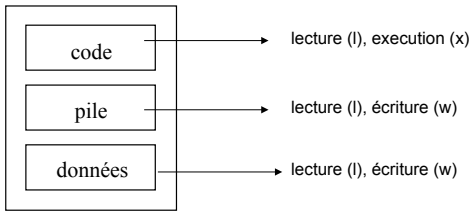
- **Le langage C (C Ansi)**
  Brian Kernighan, Dennis Ritchie, *Masson Prentice Hall*, 1991, 280 p.
- **International Standard ISO/IEC 9945-1, IEEE Std 1003-1**
  IEEE Standards Department, 07/12/1990, 356 p.
- **POSIX Programmer's Guide (POSIX.1)**
  Donald Lewine, *O'Reilly & Associates, Inc.*, 1994, 609 p.
- **POSIX.4.**
  Bill Gallmeister, *O'Reilly & Associates, Inc.*, 1995, 566 p.
- **Programmer avec les outils GNU**
  O'Reilly & Associates, Inc., 1997, 265 p.
- **Méthodologie de la programmation en C, Bibliothèque standard, API POSIX**
  J.-P. Braquelaire, *Dunod, 3e éd.*, Paris 2000, 556 p.

Cours 2
Processus sous UNIX

Processus

- **Processus: entité active du système**
  - correspond à l'exécution d'un programme binaire
  - identifié de façon unique par son numéro : **pid**
  - possède 3 segments :
    - code, données et pile
  - Exécuté sous l'identité d'un utilisateur :
    - propriétaire réel et effectif
      - Groupe réel et effectif
  - possède un répertoire courant

Processus

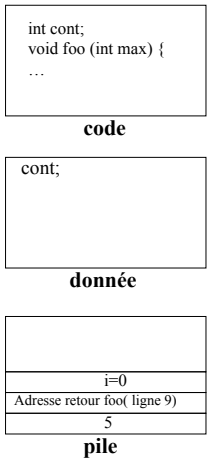


Processus

- **Chaque processus est indépendant**
  - Deux processus peuvent être associés au même programme (code)
  - Synchronisation entre processus (communication)
- **CPU est partagé (temps partagé)**
  - Commutation entre les processus

Execution Programme

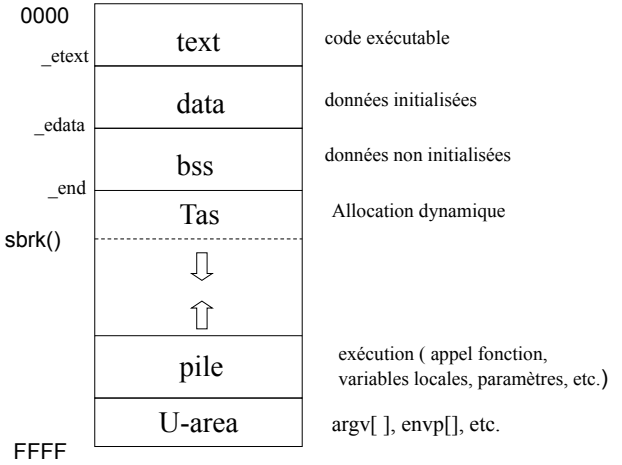
```
1: int cont;

2: void foo (int max) {
3:  int i;
4:  for (i=0; i++; i<max)
5:     printf ("%d \n", i);
6:}

 7: int main (int argc, char* argv []) {
 8: cont=5;
 9:  foo(cont) ;

10: return EXIT_SUCCESS;
11: }
```



Segment de données

- **Data: contient les données initialisées au chargement du processus et les donnes statiques initialisées .**
- **BSS : Données non initialisées.**
- **Tas (heap): zone mémoire utilisée pour stocker les espaces mémoire alloués dynamiquement (ex. malloc(), calloc()).**
  - Elle augmente dans le sens inverse de la pile.

Espace virtuel d'un processus



Allocation dynamique de mémoire

- **#include <stdlib.h>**
  - void * malloc(size_t size)
    - Alloue un bloc de *size* octets
  - void * calloc(size_t nb, size_t size)
    - Alloue un bloc de *nb*size* octets
      - initialisés à 0
  - void * realloc(void* ptr, size_t size)
    - Redimensionne un bloc mémoire
      - Conserve le contenu
  - void free (void * ptr)
    - Libère la mémoire allouée

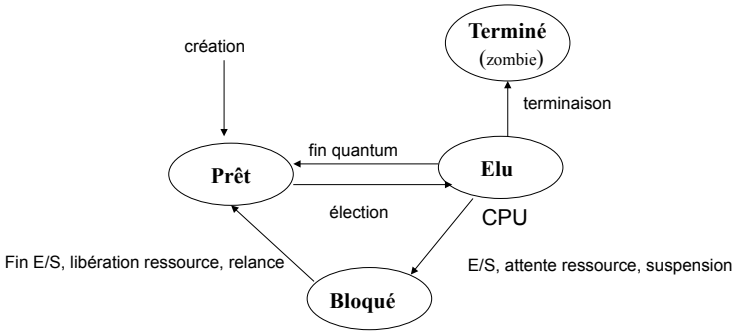
Allocation dynamique de mémoire

- **Augmenter le tas**
  - #include <unistd.h>
  - int brk(void *ptrFin);
    - positionne la fin du tas (le premier mot mémoire hors de la zone accessible) à l'adresse spécifiée par *ptrFin* .
  - void *sbrk(intptr_t *inc)*;
    - incrémente le tas du programme de *inc* octets.
    - Appeler sbrk avec un incrément nul permet d'obtenir l'emplacement de la limite actuelle.

Etats d'un processus

- **En exécution, un processus change d'état**
  - **Elu (running)**: instructions du processus sont exécutées
  - **Bloqué (waiting)** : processus en attente d'une ressource, en suspension
  - **Prêt (ready)**: processus attend d'être affecté au processeur
  - **Terminé (zombie)** : processus a fini son exécution mais son père n'a pas encore pris connaissance de sa terminaison

Etats d'un processus



- **Quantum** : durée élémentaire (e.g. 10 à 100 ms)

Attributs d'un processus

- **Identité d'un processus:**
  - pid: nombre entier
    - POSIX: type *pid_t*
      - <unistd.h> : fichier à inclure
    - **pid_t getpid (void)** :
      - obtention du pid du processus
- **Exemple:**

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv) {
        printf (" pid du processus : %d \n", getpid()) ;
 return EXIT_SUCCESS;
}
```


Attributs d'un processus (cont)

- **Un processus est lié à un utilisateur et son groupe**
  - Réel : Utilisateur (groupe):
    - Droits associé à l'utilisateur (groupe) qui lance le programme
  - Effectif: utilisateur (groupe):
    - Droits associé au programme lui-même
      - identité que le noyau prend effectivement en compte pour vérifier les autorisations d'accès pour les opérations nécessitant une identification
      - Exemple: ouverture de fichier, appel-système réservé.
  - UID (User identifier) GID (group identifier)
    - #include <sys/types.h>
    - Types *uid_t* et *gid_t*

Attributs d'un processus (cont)

- **#include <unistd>**
  - uid_t getuid (void) /* utilisateur réel */
  - uid_t geteuid (void) /* utilisateur effectif */
  - gid_t getgid (void) /* groupe réel */
  - gid_t getegid (void) /* groupe effectif */
    - Réel : celui qui a lance le processus
    - Effectif : propriétaire du programme si bit "Set-UID" positionné
      - Exemple:
        - ls -l /usr/bin/passwd
        - -r-s--x--x 1 root root 23720 2008-01-23 15:08 /usr/bin/passwd
  - Changer les droits du processus:
    - setuid(), seteuid(), setgid(), setegid()

Fork : création d'un processus

- **Primitive *pid_t fork (void)***
  - permet la création dynamique d'un nouveau processus (*fils*) qui s'exécute de façon concurrente avec le processus qui l'a créé (*père*).

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void)
```

  - Processus fils créé est une copie du processus père

Fork : création d'un processus

- **Chaque processus reprend son exécution en effectuant un retour d'appel fork**
  - un seul appel à ***fork***, mais deux retours dans chacun des processus. Valeurs de retour diffèrent selon le processus
    - **0** : renvoyé au processus fils
    - **pid du processus fils** : renvoyé au processus père
    - **-1** : appel à la primitive a échoué
      - errno <errno.h> :
        - ENOMEM : système n'a plus assez de mémoire disponible
        - EAGAIN : trop de processus créés
  - pid_t getppid (void)
    - obtenir le pid du père

Fork : création d'un processus

- **Exemple 1**

test-fork1.c

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
        pid_t pid_fils;
        switch (pid_fils = fork ( ) ) {
          case (pid_t) -1 : perror ("fork"); exit (1);
          case  (pid_t) 0 : printf ("FILS : pid % d; pid pere: %d \n", getpid (), getppid () );
                            return EXIT_SUCCESS;
          default:          printf ("PERE :pid %d; pid pere :%d, pid fils: %d \n", getpid (),
                                getppid (), pid_fils);
                            return EXIT_SUCCESS;
  }
}
```

Fork : création d'un processus

- **Processus père s'exécute après les fils**

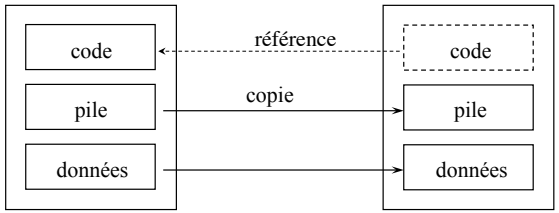
```
>echo $$  /* pid du processus shell */
1089
>test-fork1
FILS : pid 1528; pid père: 1476
PERE : pid 1476; pid père : 1089, pid fils : 1528
```

- **Processus père s'exécute avant les fils**
  - Processus fils devient orphelin.
    - Hérité par le processus *init*, dont le *pid* =1

```
>echo $$  /* pid du processus shell */
1089
>test-fork1
PERE : pid 1476; pid père : 1089, pid fils : 1528
FILS : pid 1528; pid père: 1
```

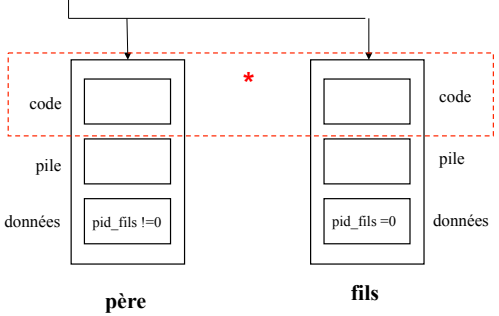

Fork : création d'un processus

- Les deux processus partagent le même code physique.
- Duplication de la pile et segment de données :
  - variables du fils possèdent les mêmes valeurs que celles du père au moment du *fork* ;
  - toute modification d'une variable par l'un des processus n'est pas visible par l'autre.



Fork : création d'un processus

```
int pid_fils;
 main (int arg, *argv []) {
 ….
 if ( ( (pid_fils= fork ( )) == 0)
 {/* fils */
   …
 }
 else {
 …
 }
```



Fork : duplication des données

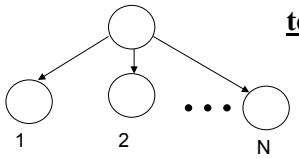
Exemple 2 :

test-fork2.c

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
    int a= 3; pid_t pid_fils;
        a *=2;
        if ( (pid_fils = fork ( ) ) == -1 ) {
              perror ("fork"); exit (1); }
        else
              if (pid_fils == 0) {
                 a=a+3;
                 printf ("fils : a=%d \n", a); }
        else
           printf ("pere : a=%d \n", a);
        return EXIT_SUCCESS;
        }
```

Fork : Exemple 3

- **Un processus crée N fils**



test-fork3.c

Donnez le code

Fork : Exemple 4

Combien de processus sont-ils créés?

test-fork4.c

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
   int i =0 ;
   while (i <3) {
     printf ( "%d ", i);
     i ++;
     if (fork ( ) == -1)
        exit (1);
   }
   printf( "\n ");
   return EXIT_SUCCESS;
}
```


Fork – Exemple 5
Allocation dynamique de mémoire

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

char* ptr1=NULL;
char* ptr2 =NULL;
```

test_fork5.c

```
int main (int argc, char* argv []) {
  pid_t pid;
  ptr1= (char*)malloc(SIZE);
  memcpy(ptr1, (char*)"toto", strlen("toto"));
    if ((pid=fork ( ) )== -1)
      exit (1);
    else
            if (pid != 0){
              /*pere */
              ptr2= (char*) malloc(SIZE);
      memcpy(ptr2, (char*) "titi", strlen("titi"));
      printf ("PERE - ptr1:%s ; ptr2:%s \n", ptr1, ptr2);
            }
     else
              printf ("FILS - ptr1:%s ; ptr2:%s \n", ptr1, ptr2);

    return 0;
}
```


Fork : Héritage

- **Un processus hérite de(s) :**
  - ID d'utilisateur et ID de groupe
    - (réel et effectif)
  - ID de session
  - Répertoire de travail courant
  - Les bits de *umask*
  - Masque de signal et les actions enregistrées
  - Variables d'environnement
  - Mémoire partagée attachée
  - Les descripteurs de fichiers ouverts
  - Valeur de *nice*
  - …

Fork : Héritage

- **Un processus n'hérite pas de(s) :**
  - identité (pid) du processus père
  - temps d'exécution
  - signaux pendants
  - verrous de fichiers maintenus par le processus père
  - alarmes ni temporisateurs
    - fonctions *alarm*, *setitimer*, …

Fork : Héritage

test-env.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#define TAILLE_BUF 1024

char buffer [TAILLE_BUF]; pid_t pid_fils;
int main(int argc, char **argv) {
 if ((pid_fils = fork ()) == -1)
          return EXIT_FAILURE;
else if (pid_fils == 0)
   printf ("FILS: rep. cour.:%s, pid:%d \n",getcwd(buffer,TAILLE_BUF), getpid ()) ;
    else
   printf ("PERE: rep. cour.:%s, pid: %d \n",getcwd (buffer,TAILLE_BUF),getpid ());
return EXIT_SUCCESS;
}
```

```
>test-env
PERE : rep. cour. : /home/arantes/test-fork4 pid:1894
FILS : rep. cour. : /home/arantes/test-fork4 pid: 2534
```

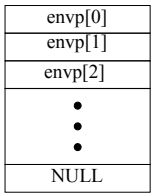

Fork : Héritage – variable d'environnement

- **Les variables d'environnement sont définies sous la forme de chaînes de caractères**
  - NOM=valeur
- **Après un *fork* le processus fils hérite d'une copie des variables d'environnement de son père**
- **Un processus peut modifier, créer, détruire les variables d'environnement**
  - Ceci n'affecte que sa propre copie
  - Fonctions *getenv*, *setenv*, *putenv*
- **Un processus récupère la liste de variables d'environnement :**
  - Variable globale environ (recommander par POSIX)
  - Troisième argument de main, envp

Fork : Héritage – variable d'environnement

```
#define _POSIX_SOURCE 1
#include <stdio.h>
extern char **env;
int main (int argc, char* argv [])
{ int i=0;
   for (i=0; env[i]!=NULL; i++)
    printf ("%s \n", env[i]);
    return 0;
}
```

```
#define _POSIX_SOURCE 1
#include <stdio.h>
int main (int argc, char* argv [], char* envp)
{ int i=0;
   for (i=0; envp[i]!=NULL; i++)
    printf ("%s \n", envp[i]);
    return 0;
}
```



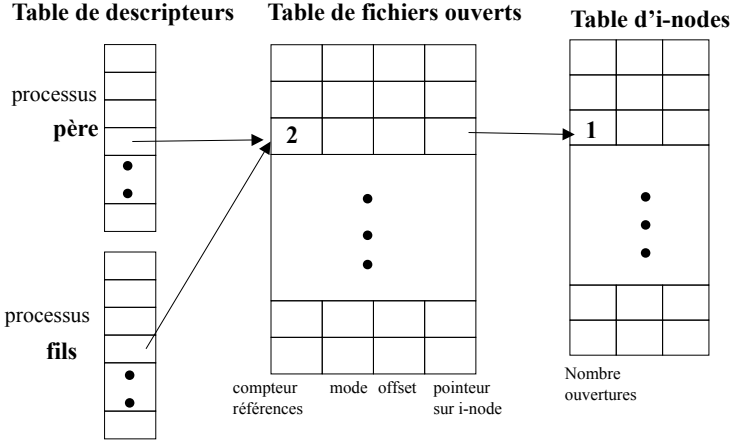
Fork : Héritage – variable d'environnement

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <sys/unistd.h>
#include <stdio.h>
#include <stdlib.h>

char* env; pid_t pid_fils;
```

test-fork5.c

```
int main (int argc, char* argv []) {
   env=getenv ("PATH");
   printf ("PERE: PATH=%s\n\n", env);
                if ( (pid_fils = fork ( ) )  == -1 ) {
                        perror ("fork"); exit (-1); }
                else
                 if (pid_fils == 0) {
                    printf ("FILS: PATH %s \n", env);
                    setenv("PATH",strcat (env,":./"),1);
                    env=getenv ("PATH");
                    printf ("FILS: PATH=%s \n\n", env); }
                 else {
                   sleep (1);
                   env=getenv ("PATH");
                    printf ("PERE: PATH=%s\n", env);
                 }
                 return EXIT_SUCCESS;
}
```


Fork : Héritage - descripteurs de fichier



Fork : Héritage - descripteurs de fichier

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

#define SIZE_TAMPON 100
 char tampon [SIZE_TAMPON]; int status;

 int main (int argc, char* argv []) {
    int fd1, fd2;    int n,i;
  if ((fd1 = open (argv[1], O_RDWR| O_CREAT |
                O_SYNC,0600)) == -1) {
        perror ("open \n");
       return EXIT_FAILURE;
}
if (write (fd1,"abcdef", strlen ("abcdef")) == -1) {
    perror ("write");
    return EXIT_FAILURE; }
```

test-fork6.c

```
if (fork () == 0) {
   /* fils */
     if ((fd2 = open (argv[1], O_RDWR)) == -1) {
        perror ("open \n");
        return EXIT_FAILURE;
     }
     if (write (fd1,"123", strlen ("123")) == -1) {
        perror ("write");
        return EXIT_FAILURE;
     }
     if ((n= read (fd2,tampon, SIZE_TAMPON)) <=0) {
        perror ("fin fichier\n");
       return EXIT_FAILURE;
     }
     for (i=0 ; i<n; i++)
        printf ("%c",tampon [i]);
     printf("\n");
     exit (0);
}
else /* père */
   wait (&status);
 return EXIT_SUCCESS;
}
```

```
>test-fork6 fich
 abcdef123
>cat fich
 abcdef123
```


Terminaison d'un processus

- **Fonction *exit(int val)* ou *return val***
  - val: valeur récupérer par le processus père
  - Possible d'employer les constantes:
    - EXIT_SUCESS
    - EXIT_FAILURE
  - Processus lancé par le shell se termine, code d'erreur disponible dans la variable $?
    - echo $?

Terminaison d'un processus (cont)

- **Lorsqu'un processus se termine normalement :**
  - Toutes les fonctions qui on été enregistrées à l'aide de *atexit()* sont appelées
  - Fermeture de tous les flux dE/S
    - Buffers sont vidés
  - L'appel à *_exit()* (appel système) qui:
    - Ferme les descripteurs de fichiers ouverts
    - Le processus père reçoit un signal SIGCHILD
  - Le processus devient alors zombie

Terminaison de processus (cont.)

- **Processus zombie:**
  - Etat d'un processus terminé tant que son père n'a pas pris connaissance de sa terminaison.
- **Synchronisation père/fils:**
  - En se terminant avec la fonction *exit* ou *return* dans main, un processus affecte une valeur à son *code de retour* :
    - processus père peut accéder à cette valeur en utilisant les fonctions *wait* et *waitpid*.

Terminaison de processus (cont.)

- **Fonction atexit ()**
  - Enregistrer une fonction qui doit être invoquée à la fin du programme

    #include <stdlib.h>

    int atexit (void (*function (void));
    - La fonction **atexit()** enregistre la *fonction* donnée pour que celle-ci soit automatiquement invoquée lorsque le programme se termine normalement avec exit(2) ou par un retour de la fonction **main**. Les fonctions ainsi enregistrées sont invoquées en ordre inverse de leur enregistrement, aucun argument n'est transmis.

Wait : Synchronisation père/fils

- **Primitive pid_t wait (int* status)**

```
#include <sys/types.h>
#include <sys/wait.h>
 pid_t wait (int* status)
```

  - Si le processus appelant :
    - possède au moins un fils *zombie* :
      - la primitive renvoie l'identité de l'un de ses fils zombies et si le pointeur *status* n'est pas NULL, sa valeur contiendra des informations sur ce processus fils.
    - possède des fils, mais aucun n'est dans l'état zombie :
      - Le processus est bloqué jusqu'à ce que:
        - un de ses fils devienne *zombie*
        - il reçoive un signal .
    - ne possède pas de fils
      - l'appel renvoie -1 et errno = ECHILD.

Wait : Synchronisation père/fils

- **Interprétation de la valeur de retour - *int*status***
  - Utilisation des ***macros*** pour des questions de portabilité :
    - Type de terminaison
      - WIFEXITED : non NULL si le processus fils s'est terminé normalement.
      - WIFSIGNALED : non NULL si le processus fils s'est terminé à cause d'un signal
      - WIFSTOPPED : non NULL si le processus fils est stoppé (option WUNTRACED de waitpid)
    - Information sur la valeur de retour ou sur le signal
      - WEXITSTATUS : code de retour si le processus s'est terminé normalement
      - WTERMSIG : numéro du signal ayant terminé le processus
      - WSTOPSIG : numéro du signal ayant stoppé le processus

Wait : Synchronisation père/fils

- **Exemple :**

test-wait.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

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

 pid_t pid_fils; int status;
 if (fork () == 0) {
  printf ("FILS:  pid = %d \n",
           getpid());
  exit (2);
 }
else {
   pid_fils = wait(&status);
   if (WIFEXITED (status) ) {
    printf ("PERE: fils %d termine, status : %d \n",
            pid_fils,  WEXITSTATUS (status));
   return EXIT_SUCCESS;
 }
  else
   return EXIT_FAILURE;
 }
}
```

```
>test-wait
  FILS: pid= 3254
   PERE : fils 3254 termine, status : 2
```


Wait : Synchronisation père/fils

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

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

 pid_t pid_fils; int status;
 if (fork () == 0) {
  printf ("FILS:  pid = %d \n",
           getpid());
  pause ();
  exit (2);
 }
```

test-wait2.c

```
else {
      pid_fils = wait(&status);
      if (WIFEXITED (status) ) {
          printf ( "PERE: fils %d termine avec status %d \n",
                   pid_fils, WEXITSTATUS (status));
          return EXIT_SUCCESS;
      }
     else
          if (WIFSIGNALED (status) ) {
              printf ( "PERE: fils %d termine par signal %d \n",
                       pid_fils, WTERMSIG (status));
              return EXIT_SUCCESS;
          }
      return EXIT_FAILURE;
  }
}
```

```
>test-wait2 &
  FILS: pid= 4897
> kill –KILL 4987
   PERE : fils 4987 termine par signal 9
```


Wait : Synchronisation père/fils

test-wait3.c

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>

#define N 3

int cont =0;
```

Quelle est la valeur affichée de *cont* ?

```
int main (int argc, char* argv []) {
   int i=0;  pid_t pid;
    while (i <N) {
      if ((pid=fork ( ) )== 0) {
        cont++;
        break;
      }
      i++;
    }

   if (pid != 0) {
    /* pere */
     for (i=0; i<N; i++)
      wait (NULL);
    printf ("cont:%d \n", cont);
   }
     return EXIT_SUCCESS;
}
```

Waitpid : Synchronisation père/fils

- **Primitive *pid_t waitpid (pid_t pid, int* status, int opt)***

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

  - en bloquant ou non le processus selon la valeur de *opt*, *waitpid* permet de tester la terminaison d'un processus fils d'identité *pid* ou qui appartient au groupe |pid|.
    - *status* possède des informations sur la terminaison du processus en question.

Waitpid : Synchronisation père/fils

- **Valeur du paramètre *pid***
  - > 0 du processus fils
  - 0 d'un processus fils quelconque du même groupe que l'appelant
  - -1 d'un processus fils quelconque
  - < -1 d'un processus fils quelconque dans le groupe |pid|
- **Valeur du paramètre *opt***
  - WNOHANG : appel non bloquant
  - WUNTRACED : processus concerné est stoppé dont l'état n'a pas été encore informé depuis qu'il se trouve stoppé.
- **Code renvoi**
  - -1 : erreur
  - 0 : en cas non bloquant, si le processus spécifié n'a pas terminé
  - *pid* du processus terminé

Waitpid : Synchronisation père/fils

- **Exemple**

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   pid_t pid_fils; int status;
 if ((pid_fils=fork ()) == 0) {
   printf ("FILS: pid=%d \n", getpid());
   sleep (1);
   exit (2);
 }
```

test-waitpid1.c

```
else {
   if (waitpid(pid_fils,&status,WNOHANG) == 0) {
     printf ("PERE: fils n'a pas terminé \n");
     return EXIT_SUCCESS;
     }
   else
     if WIFEXITED (status) {
              printf ("PERE: fils  %d terminé, status= %d \n",
              pid_fils, WEXITSTATUS (status));
              return EXIT_SUCCESS;
     }
     else
     return EXIT_FAILURE;
 }
}
```

```
>test-waitpid1
   FILS : pid =19078
   PERE: fils n'a pas terminé
```


Waitpid : Synchronisation père/fils

- **Exemple**

```
#define _POSIX_SOUCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   pid_t pid_fils; int status;
 if ((pid_fils=fork ()) == 0) {
   printf ("FILS: pid=%d \n", getpid());
    exit (2);
 }
```

test-waitpid2.c

```
else {
   sleep (1);
   if (waitpid(pid_fils,&status,WNOHANG) == 0) {
     printf ("PERE: fils n'a pas terminé \n");
     return EXIT_SUCCESS;
     }
   else
     if WIFEXITED (status) {
              printf ("PERE: fils  %d terminé, status= %d \n",
              pid_fils, WEXITSTATUS (status));
              return EXIT_SUCCESS;
     }
     else
     return EXIT_FAILURE;
 }
}
```

```
>test-waitpid2
   FILS : pid =19084
   PERE: fils 19084 terminé, status 2
```

exec: exécution de nouveaux programmes

- **Primitive exec: recouvrement**
  - permet de remplacer le programme qui s'exécute par un nouveau programme, dont le nom est passé en argument. Le nouveau programme sera exécuté au sein de l'espace d'adressage du processus appelant.
    - Si l'appel à *exec* **réussit**, il ne rend jamais le contrôle au processus appelant.
    - Exemple d'erreur (*errno*):
      - EACCES : pas de permission d'accès au fichier
      - ENOENT : fichier n'a pas été trouvé
      - …

exec : Exécution de nouveaux programmes

- **Six fonctions de la famille exec**
  - *préfixe* = exec
  - plusieurs *suffixes* :
    - Forme sous laquelle les arguments *argv* sont transmis:
      - ***l* : argv** sous forme de liste
      - ***v* : argv** sous forme de tableau (v - vector)
    - Manière dont le fichier à exécuter est recherché par le système:
      - ***p***: fichier est recherché dans les répertoires spécifiés par *$PATH*. Si ***p*** n'est pas spécifié, le fichier est recherché soit dans le répertoire courant soit dans le *path* absolu passé en paramètre avec le nom du fichier.
    - Nouvel environnement
      - ***e***: nouvel environnement transmis en paramètre. Si ***e*** n'est pas spécifié, l'environnement ne change pas.

exec : Exécution de nouveaux programmes

- **argv sous forme de liste :**

```
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[]);
```

- **argv sous forme de tableau :**

```
int execv  (const char *path, char * const argv[]);
int execvp (const char *file, char * const argv[]);
int execve (const char *file, char * const argv[], char * const envp[]);
```

  - Dernier argument doit être NULL

exec : Exécution de nouveaux programmes

- **Exemple : execl**

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
  execl ("/usr/bin/wc","wc", "-w", "/tmp/fichier1", NULL);
  perror ("execl");
  return EXIT_SUCCESS;
}
```

exec : Exécution de nouveaux programmes

- **Exemple : execlp**

```
#define _POSIX_SOURCE 1

#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv) {
  execlp ("wc","wc", "-w", "/tmp/fichier1", NULL);
  perror ("execlp");
  return EXIT_SUCCESS;
}
```


exec : Exécution de nouveaux programmes

- **Exemple : execv**

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
  char *arg_vect [4];
  arg_vect[0] ="wc";
  arg_vect[1] ="-w";
  arg_vect[2] ="/tmp/fichier1";
  arg_vect[3] = NULL;

  execv ("/usr/bin/wc",arg_vect);
  perror ("execv");
  return EXIT_SUCCESS;
}
```


exec : Exécution de nouveaux programmes

sleep_exec.c

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
  char* argv_sleep[]= {"sleep_prog", "2",
                       (char *) NULL };
   printf ("Debut - pid:%d \n", getpid ());
    execv ("./sleep_prog", argv_sleep);
    printf("Fin – pid:%d \n", getpid ());
return EXIT_SUCCESS;
```

sleep_prog.c

```
int main(int argc, char **argv) {
  printf("sleep_prog : debut - pid:%d\n ",
          getpid ());
  sleep (atoi(argv[1]));
  printf("sleep_prog : fin pid:%d\n ",
          getpid ());

  return EXIT_SUCCESS;
}
```

```
> sleep_exec
Debut – pid : 356
sleep_prog : debut – pid: 356
sleep_prog : fin pid:356
```


System ()

- **Contrairement aux fonctions de la famille *exec*, le code du processus appelant n'est pas remplacé**
  - *system ()* invoque le shell en lui transmettant la fonction passée en paramètre.
    - Un *fork ()* est exécute.
      - Le fils lance la commande en appelant le shell "/bin/sh –c commande".
      - Le processus père attend la fin du fils.

System ()

system_exec.c

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
    printf ("Debut - pid:%d \n", getpid ());
     system("./sleep_prog 2");
    printf(" Fin - pid:%d \n", getpid ());
    return EXIT_SUCCESS;
```

```
int main(int argc, char **argv) {
  printf("sleep_prog : debut - pid:%d\n ",
          getpid ());
  sleep (atoi(argv[1]));
  printf("sleep_prog : fin pid:%d\n ",
          getpid ());

  return EXIT_SUCCESS;
}
```

```
> system_exec
Debut – pid : 356
sleep_prog : debut – pid: 374
sleep_prog : fin pid:374
Fin – pid: 356
```


VFORK : création d'un processus

- **Primitive *pid_t vfork (void)***
  - Améliorer les performances du mécanisme de création des processus

```
#include <sys/types.h>
#include <unistd.h>
 pid_t vfork (void)
```

  - Le segment de données n'est pas dupliqué. Le processus fils travaille sur les données de son père.
    - Processus père doit se bloquer jusqu'à ce que le processus fils se termine ou se recouvre (*exec*)

Primitive VFORK

- **Exemple**

test-vfork.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

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

  pid_t pid_fils; int cont = 0;
  if ((pid_fils = vfork ()) == -1) {
    perror ("vfork");
    exit (1);
}
```

```
 if (pid_fils == 0) {
   cont++;
   printf ("fils: cont = %d \n", cont);
   _exit (0);
 }
 else {
   sleep (1);
   cont ++;
   printf ("père: cont = %d \n", cont);
 }
 return EXIT_SUCCESS;
}
```

```
>test-vfork
  fils: cont = 1
  père : cont =2
```

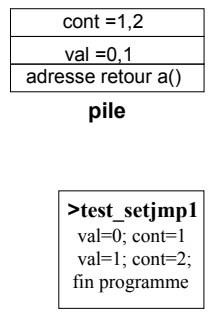

Longmp/setjmp

- **"Goto": utiliser à l'intérieur d'une même imbrication de fonction**
  - #include setjmp.h
  - setjmp (jmp_buf env)
    - Permet de sauvegarder l'état du programme dans *env* du type *jmp_buf*
      - Appel direct a *setjmp* retourne 0
      - Si l'appel est issue d'un *longjmp*, il retourne une autre valeur renovoyée par longjmp.
  - void longjmp(jmp_buf env, int val);
    - Remet le programme dans l'état sauvegardé par le dernier appel à *setjmp* par rapport à la variable env.

Setjmp/letjmp

test_setjmp1.c

```
#include <setjmp.h>
jmp_buf a_buff;

void a () {
  int val = 0; cont=1;
  if (setjmp (a_buff) == 0) {
     printf ("val=%d; cont=%d\n", val, cont);
     val++; cont*=2;
     longjmp(a_buff,1);
  }
  else
     printf ("val=%d; cont=%d\n", val, cont);
}

int main (int argc, char *argv[]){
  a();
  printf ("fin programme \n");
}
```

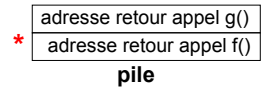



```
>test_setjmp1
  val=0; cont=1
  val=1; cont=2;
 fin programme
```

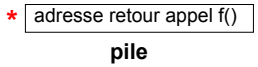

Setjmp/Longjmp

test_setjmp2.c

```
#include <setjmp.h>
jmp_buf env;

void f () {
  if (setjmp (env) == 0) {
     printf ("apres setjmp \n");
     g();
  }
  else
     printf ("apres longjmp\n");
}

g() {
   longjmp(env,1);
}
int main (int argc, char *argv[]){
  f();
  printf ("fin programme \n");
}
```



Contexte sauvegardé en *env*:




```
>test_setjmp2
 apres setjmp
 apres longjmp
 fin programme
```

Cours 3 : Gestion des signaux

- **Mécanisme de communication de base**
  - Un signal est une information transmise à un programme durant son exécution.
    - A chaque signal est associée une valeur entière positive non nulle et strictement inférieure à **NSIG (**constante non POSIX)
    - C'est par ce mécanisme que le système communique avec les processus utilisateurs :
      - en cas d'erreur (violation mémoire, erreur d'E/S),
      - à la demande de l'utilisateur lui-même via le clavier (caractères d'interruption ctrl-C, ctrl-Z...),
      - lors d'une déconnection de la ligne/terminal, etc.
    - Possibilité d'envoi d'un signal entre processus.
    - Traitement par défaut.

Les principaux signaux POSIX

Nom	*Événement*	*comportement*
Terminaison		
SIGINT	ctrl-C	terminaison
SIGQUIT	<QUIT> ctrl-\	terminaison + core
SIGKILL	Tuer un processus	terminaison
SIGTERM	Signal de terminaison	terminaison
SIGCHLD	Terminaison ou arrêt d'un processus fils	ignoré
SIGABRT	Terminaison anormale	terminaison + core
SIGHUP	Déconnexion terminal	terminaison

Les principaux signaux POSIX (cont.)

Nom	*Événement*	*comportement*
Suspension/reprise		
SIGSTOP	Suspension de l'exécution	suspension
SIGTSTP	Suspension de l'exécution (ctrl-Z)	suspension
SIGCONT	Continuation du processus arrêté	reprise
Fautes		
SIGFPE	erreur arithmétique	terminaison + core
SIGBUS	erreur sur le bus	terminaison + core
SIGILL	instruction illégale	terminaison + core
SIGSEGV	violation protection mémoire	terminaison + core
SIGPIPE	Erreur écriture sur un tube sans lecteur	terminaison

Les principaux signaux POSIX (cont.)

Nom	*Événement*	*comportement*
Autres		
SIGALRM	Fin de temporisation	terminaison
SIGUSR1	Réservé à l'utilisateur	terminaison
SIGUSR2	Réservé à l'utilisateur	terminaison
SIGTRAP	Trace/breakpoint trap	terminaison + core
SIGIO	E/S asynchrone	terminaison

SIGNAUX

- **A chaque signal est associé une valeur**
  - **"/usr/include/signal.h"**
  - **Liste des signaux:**

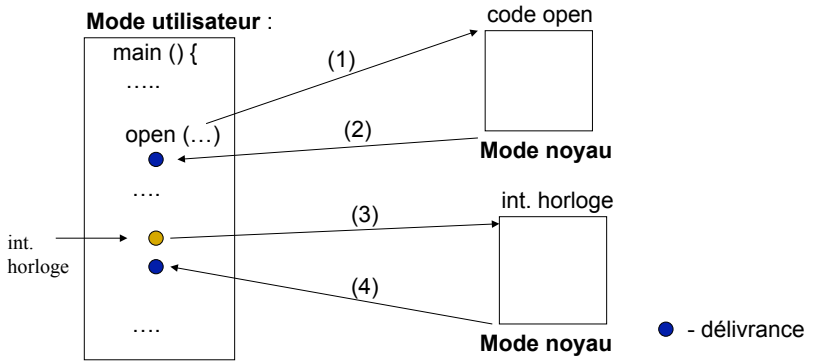
    ```
    $ kill –l
    ```
  - **Utiliser plutôt le nom de la constante au lieu du numéro**
    - **Exemple: SIGKILL (=9), SIGINT (=2), etc.**
      - **kill –KILL <num. proc>; kill –INT <num. proc>**
  - Envoyer un signal revient à envoyer ce numéro à un processus. Tout processus a la possibilité d'émettre à destination d'un autre processus un signal, à condition que ses numéros de propriétaires (UID) lui en donnent le droit vis-à-vis de ceux du processus récepteur.

Signaux - Terminologie

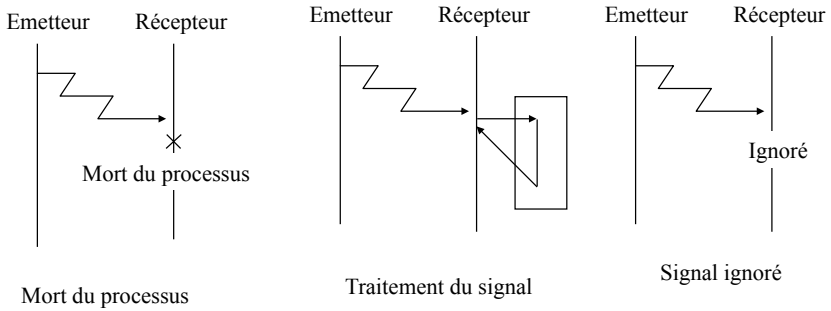
- **Signal pendant**
  - Signal qui a été envoyé à un processus mais qui n'a pas encore été pris en compte.
    - Cet envoi est mémorisé dans le BCP du processus.
    - Si un exemplaire d'un signal arrive à un processus alors qu'il en existe un exemplaire pendant, le signal est **perdu.**
- **Délivrance**
  - Un signal est délivré à un processus lorsque le processus le prend en compte et réalise l'action qui lui est associée.
    - La délivrance a lieu lorsque le processus **passe de l'état actif noyau à l'état actif utilisateur** : retour appel système, retour interruption matérielle, élection par l'ordonnanceur.
- **Signal masqué ou bloqué**
  - La délivrance du signal est ajournée

Délivrance d'un Signal

- Passage du processus de l'état noyau à l'état utilisateur
  - Retour appel système (fonctions), retour interruption, processus vient d'être élu par l'ordonnanceur.



Conséquence de la délivrance d'un signal



- **Ne pas confondre avec les interruptions**
  - Matérielles : int. horloge, int. Disque, etc.

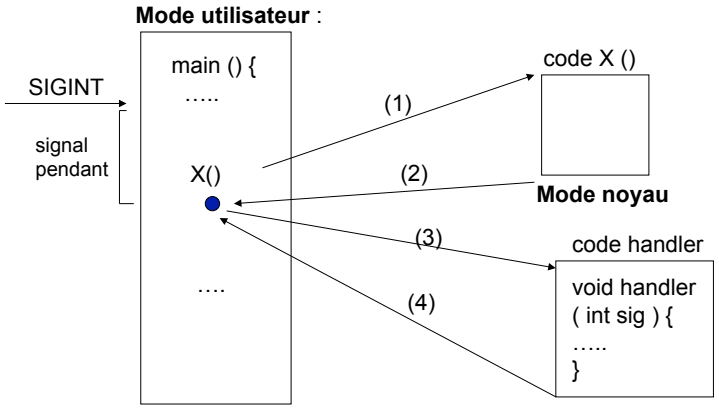
Délivrance d'un signal

- **Comportement par défaut**
  - Terminaison du processus
  - Terminaison du processus avec production d'un fichier de nom *core*
  - Signal ignoré
  - Suspension du processus (*stopped* ou *suspended*)
  - Continuation du processus

- **Installation d'un nouveau handler (sigaction) ***
  - **SIG_IGN** (ignorer le signal)
  - **Fonction** définie par l'utilisateur
  - **SIG_DFL** (restituer le comportement par défaut)

  * Applicable à tous les signaux sauf **SIGKILL**, **SIGSTOP**

Exemple délivrance signal - handler



X = appel système

Délivrance d'un signal –appel système priorité interruptible

- **L'arrivée d'un signal à un processus endormi à un niveau de priorité interruptible le réveille**
  - Processus passe à l'état prêt
  - Le signal sera délivré lors de l'élection du processus
    - Fonction *handler* associée sera exécutée
  - Exemples d'appels système interruptibles:
    - *pause*,
    - *sigsuspend*,
    - *Wait/waitpid*
    - *read*, *write*,
    - etc.

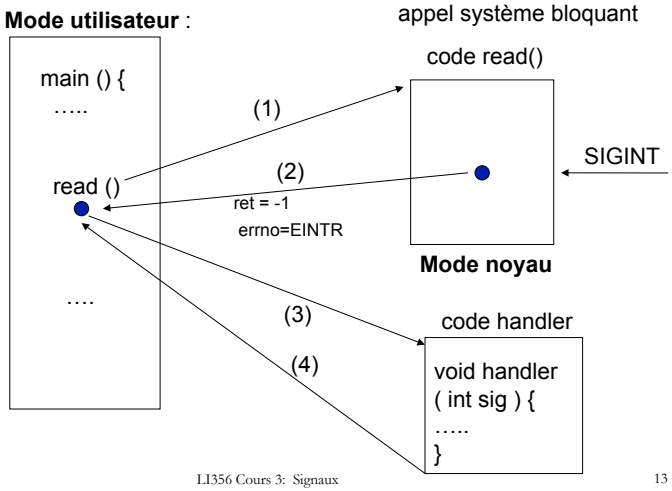
Délivrance d'un signal – appel système priorité interruptible

- **L'arrivée d'un signal sur un appel système interruptible provoque l'arrêt de l'appel**
  - appel système renvoie la valeur -1 signalant un échec.
  - code d'erreur: errno = EINTR .
  - Exemple :

```
ret = read (desc, buffer, TAILLE_BUFFER):
    if ((ret == -1) && (errno == EINTR)
        printf ("signal délivre \n" );
```

  - Possibilité de reprise automatique: *flag* SA_RESTART (*struct sigaction*)

Délivrance d'un signal – appel système priorité interruptible



L'envoi des signaux (kill)

- **Appel système**
  - **int kill (pid_t pid, int signal)**
    - Par défaut la réception d'un signal provoque la terminaison

    pid:
    pid: processus d'identité *pid*
    0 : tous les processus dans le même groupe
    -1 : non défini par POSIX. Tous les processus du système
    < -1 : tous les processus du groupe |*pid*|

    signal:
    valeur entre 0 et NSIG (0 = test d'existence)

- **Commande**
  - $ **kill -l** liste des signaux
  - $ **kill -sig pid** envoi d'un signal

Exemple – kill

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main (int arg, char** argv) {

   printf ("debut application \n");

/* envoyer un SIGINT à soi-même */
  kill(getpid ( ), SIGINT);
  printf ("fin application \n");

   return EXIT_SUCCESS;
}
```


Masquage signaux

- **Signaux bloqués ou masqués**
  - Leur délivrance est différée
  - Même s'ils se trouvent pendants il ne sont pas délivrés
  - Fonction pour masquer et démasquer des signaux
  - Pendant l'exécution du handler associé à un signal, celui-ci est bloqué (norme POSIX)
    - Possibilité de le débloquer dans le handler associé
  - Un processus fils:
    - n'hérite pas des signaux pendants
    - hérite les valeurs du masque de signaux et du handler
    - *fork()* suivi par un *exec() :* réinitialisation dans le fils avec les handlers par défaut.

Gestion des Signaux

	Signaux pendants	Signaux masqués	handler	Masque à la capture	
1	0/1	0/1	**handler1 ()/NULL**	**0/1, 0/1, 0/1, …**	**0/1**
2	0/1	0/1	**handler2 () /NULL**	**0/1, 0/1, 0/1, …**	**0/1**
NSIG	0/1	0/1	**handlerNSIG () /NULL**	**0/1, 0/1, 0/1, …**	**0/1**
				1 2 3 …	NSIG

• Informations maintenues par le système

Manipulation des ensembles de signaux

- **Fonctions qui ne changent pas les signaux eux-mêmes mais permettent de manipuler des variables "ensembles de signaux".**
  - int sigemptyset(sigset_t *set);
  - int sigfillset(sigset_t *set);
  - int sigaddset(sigset_t *set, int sig);
  - int sigdelset(sigset_t *set, int sig);
  - int sigismember(sigset_t *set, int sig);

    (retourne !=0 si signal présent)

Masquage des signaux

- **Blocage des signaux:**
  - Un processus peut installer un masque de signaux à l'exclusion de SIGKILL et SIGSTOP
  - Le traitement des signaux est retardé
    - signal pendant.
  - Un processus fils hérite le masque de signaux mais non pas les signaux pendants
  - Liste des signaux pendants bloqués:
    - **int sigpending (sigset_t *set);**

Masquage des signaux

- **Appel à la fonction sigprocmask**
- **int sigprocmask(int how, const sigset_t *set, sigset_t *old);**

  how : SIG_BLOCK : bloquer en plus les signaux positionnés dans set
  SIG_UNBLOCK : démasquer
  SIG_SETMASK : bloquer uniquement les signaux dans set

  set : masque de signaux

  old: valeur du masque antérieur, si non NULL

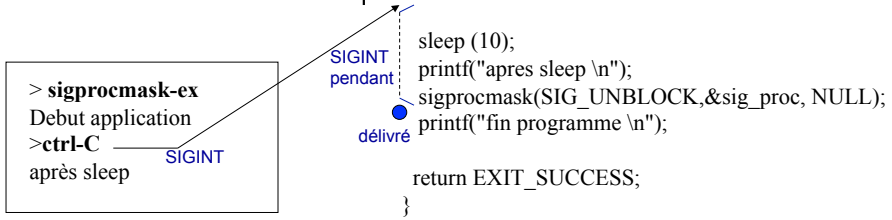
  - Le nouveau masque est formé par *set,* ou composé par *set* et le masque antérieur

Exemple – masquage signaux

```
#define _POSIX_SOURCE 1

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

sigprocmask-ex.c

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

 sigset_t sig_proc;
 printf ("Debut application \n");

sigemptyset(&sig_proc);
sigaddset (&sig_proc, SIGINT);
sigprocmask(SIG_BLOCK,&sig_proc, NULL);

 sleep (10);
 printf("apres sleep \n");
 sigprocmask(SIG_UNBLOCK,&sig_proc, NULL);
 printf("fin programme \n");

 return EXIT_SUCCESS;
}
```



Exemple – signaux pendants

```
int main (int argc, char* argv []) {

        sigset_t  sig_set;     /* liste des signaux bloqués */

        sigemptyset (&sig_set);
        sigaddset (&sig_set,SIGINT);

        /*masquer le signal SIGINT */
        sigprocmask (SIG_SETMASK, &sig_set, NULL);

        kill (getpid (), SIGINT);

        /* obtenir la liste des signaux pendants */
         sigpending (&sig_set);

        if (sigismember ( &sig_set,SIGINT) )
          printf("SIGINT pendant \n");

        return EXIT_SUCCESS;
}
```


Changement du traitement par défaut

```
struct sigaction { void (*sa_handler)  () ; /*fonction */
                   sigset_t sa_mask;  /* masque des signaux */
                   int sa_flags; /* options */
```

- **Le comportement que doit avoir un processus lors de la délivrance d'un signal est décrit par la structure sigaction**
  - *sa_handler* :
    - fonction à exécuter, SIG_DFL (traitement par défaut), ou SIG_IGN (ignoré le signal)
  - *sa_mask* : correspond à une liste de signaux qui seront ajoutés à la liste de signaux qui se trouvent bloqués lors de l'exécution du *handler*.
    - *sa_mask U {sig}*:
    - Le signal en cours de délivrance est **automatiquement** masqué par le handler
  - *sa_flags*: différentes options

struct sigaction (cont.)

- **Quelques options pour *sa_flags***
  - *SA_NOCLDSTOP* : Le signal SIGCHLD n'est pas envoyé à un processus lorsqu'un de ses fils est stoppé.
  - *SA_RESETHAND* : Rétablir l'action à son comportement par défaut une fois que le gestionnaire a été appelé
  - SA_RESTART: Un appel système interrompu par un signal capté est repris au lieu de renvoyer -1.
  - *SA_NOCLDWAIT*: Si le signal est SIGCHLD, le processus fils qui se termine ne devient pas ZOMBIE
  - etc.
- **La plupart des options ne sont pas dans la norme POSIX**

Changement du traitement par défaut

- **int sigaction (int sig, struct sigaction *act, struct sigaction *anc);**
  - **Permet l'installation d'un handler *act* pour le signal *sig***
    - *act* et *anc* pointent vers une structure du type *struct sigaction*
    - La délivrance du signal *sig,* entraînera l'exécution de la fonction pointée par *act->sa_handler,* si non NULL
    - *anc:* si non NULL, pointe vers l'ancienne structure sigaction

Exemple changement traitement par défaut (sigaction)

```
#define _POSIX_SOURCE 1

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
}
```

sigaction-ex.c

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

  sigset_t sig_proc;
  struct sigaction action;

    sigemptyset(&sig_proc);
    action.sa_mask=sig_proc;
    action.sa_flags=0;
    action.sa_handler = sig_hand;

    sigaction(SIGINT, &action,0);

    kill (getpid(), SIGINT);
    printf("fin programme \n");

  return EXIT_SUCCESS;
```

```
> sigaction-ex
signal reçu 2
fin programme
```


Attente d'un SIGNAL

- **Processus passe à l'état « stoppé ». Il est réveillé par l'arrivée d'un signal non masqué**
  - **int pause (void)**
    - Ne permet ni d'attendre l'arrivée d'un signal de type donné, ni de savoir quel signal a réveillé le processus.
  - **int sigsuspend (cons sigset_t *p_ens)**
    - Installation du masque des signaux pointé par *p_ens*. Le masque d'origine est réinstallé au retour de la fonction.

Exemple – sigaction + sigsuspend

```
void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
}

int main(int argc, char **argv) {
  sigset_t sig_proc;
  struct sigaction action;

     sigemptyset(&sig_proc);

     /* changer le traitement */
     action.sa_mask=sig_proc;
     action.sa_flags=0;
     action.sa_handler = sig_hand;
  sigaction(SIGINT, &action,NULL);
```

```
/* masquer SIGINT */
     sigaddset (&sig_proc, SIGINT);
     sigprocmask (SIG_SETMASK,
&sig_proc, NULL);

     /* attendre le signal SIGINT */
     sigfillset (&sig_proc);
     sigdelset (&sig_proc, SIGINT);
     sigsuspend (&sig_proc);

    return EXIT_SUCCESS;
}
```

Attente d'un signal (exemple1)

```
pid_t pid_fils;
void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
}

int main(int argc, char **argv) {
  sigset_t sig_proc;
  struct sigaction action;
     sigemptyset(&sig_proc);

     /* changer le traitement */
     action.sa_mask=sig_proc;
     action.sa_flags=0;
     action.sa_handler = sig_hand;

     sigaction(SIGUSR1, &action,0);
```

Fils se bloque pour toujours ?

```
if ( (pid_fils= fork ()) == 0) {
    sleep (1);
    printf("fils: après sleep \n");
    pause ();
    printf ("reprise fils \n");
 }
 else{
   kill (pid_fils, SIGUSR1);
   wait(NULL);
    printf ("fin pere \n");
 }
    return EXIT_SUCCESS;
}
```

sigurs1-ex1.C

Attente d'un signal (exemple2)

```
pid_t pid_fils;
void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
}

int main(int argc, char **argv) {
  sigset_t sig_proc;
  struct sigaction action;
     sigemptyset(&sig_proc);

     /* changer le traitement */
     action.sa_mask=sig_proc;
     action.sa_flags=0;
     action.sa_handler = sig_hand;

     sigaction(SIGUSR1, &action,0);
}
```

Fils se bloque pour toujours ?

```
if ( (pid_fils= fork ()) == 0) {
    sleep (1);
    printf("fils: après sleep \n");
    sigfillset (&sig_proc);
    sigdelset (&sig_proc, SIGUSR1);
    sigsuspend (&sig_proc);
    printf ("reprise fils \n");
 }
 else{
   kill (pid_fils, SIGUSR1);
   wait(NULL);
    printf ("fin pere \n");
 }
    return EXIT_SUCCESS;
}
```

sigurs1-ex2.C

Attente d'un signal (exemple3)

```
pid_t pid_fils;
void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
}
int main(int argc, char **argv) {
  sigset_t sig_proc;
  struct sigaction action;

   sigemptyset(&sig_proc);
    action.sa_mask=sig_proc;
    action.sa_flags=0;
    action.sa_handler = sig_hand;
   sigaction(SIGUSR1, &action,0);
```

sigaddset (&sig_proc, SIGUSR1);
sigprocmask (SIG_SETMASK, &sig_proc, NULL);

Fils se bloque pour toujours ?

```
if ( (pid_fils= fork ()) == 0) {
    sleep (1);
    printf("fils: après sleep \n");
    sigfillset (&sig_proc);
    sigdelset (&sig_proc, SIGUSR1);
    sigsuspend (&sig_proc);
    printf ("reprise fils \n");
 }
 else{
   kill (pid_fils, SIGUSR1);
   wait(NULL);
    printf ("fin pere \n");
 }
    return EXIT_SUCCESS;
}
```

sigurs1-ex3.C

Perte de signaux pendants

- **Pas de mémorisation du nombre de signaux pendants**

```
int cont; pid_t pid_fils;

void sig_hand(int sig){
 if (sig == SIGUSR1)
   cont++;
 else {
   printf ("nombre SIGUSR1 recu: %d \n", cont);
   exit (0);
 }
}

int main(int argc, char **argv) {
  sigset_t sig_proc; int i;
  struct sigaction action;

   sigemptyset(&sig_proc);
     action.sa_mask=sig_proc;
     action.sa_flags=0;
     action.sa_handler = sig_hand;
     sigaction(SIGUSR1, &action,0);
     sigaction(SIGINT, &action,0);

  if ( (pid_fils= fork ()) == 0) {
    while (1)
      pause ();
  }
  else{
    for (i=0; i<20; i++)
        kill (pid_fils, SIGUSR1);
    kill (pid_fils, SIGINT);
    wait (NULL);
   return EXIT_SUCCESS;
  }
}
```

sig_contUSR1.c

```
>sig_contUSR1
  nombre SIGUSR1 reçu:4
```

Signal SIGCHLD

- **Signal envoyé automatiquement à un processus lorsque l'un de ses fils se termine ou lorsque l'un de ses fils passe à l'état stoppé (réception du signal SIGSTOP ou SIGTSTP).**
- **Le comportement par défaut est d'ignorer le signal**
- **En captant ce signal, un processus peut prendre en compte le "moment" où la terminaison de son fils s'est produite.**
- **Elimination du fils zombie**
  - wait() / waitpid () , wait3()

Signal SIGCHLD- Exemple

```
void sig_hand(int sig){
      printf ("signal reçu %d \n",sig);
 if (sig == SIGCHLD)
    wait (NULL)
}

    int main(int argc, char **argv) {
     sigset_t sig_proc;
     struct sigaction action;

        sigemptyset(&sig_proc);

     /* changer le traitement */
         action.sa_mask=sig_proc;
         action.sa_flags=0;
         action.sa_handler = sig_hand;
      sigaction(SIGCHLD, &action,NULL);


    if (fork() != 0)
          sleep (1);
       return EXIT_SUCCESS;
    }
```


Signaux SIGSTOP/SIGTSTP, SIGCONT et SIGCHLD

- **Processus s'arrête (état bloqué) en recevant un signal SIGSTOP ou SIGTSTP**
- **Processus père est prévenu par le signal SIGCHLD de l'arrêt d'un de ses fils**
  - Comportement par défaut : ignorance du signal
  - Relancer le processus fils en lui envoyant le signal SIGCONT

SIGSTOP/SIGCONT, SIGCHLD Exemple

```
#define _POSIX_SOURCE 1
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>


pid_t pid_fils;
void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
   kill (pid_fils, SIGCONT);
}

int main(int argc, char **argv) {
    sigset_t sig_proc;
    struct sigaction action;

    sigemptyset(&sig_proc);

    /* changer le traitement */
    action.sa_mask=sig_proc;
    action.sa_flags=0;
    action.sa_handler = sig_hand;
    sigaction(SIGCHLD, &action,0);

     if ( (pid_fils= fork ()) == 0) {
         kill (getpid(), SIGSTOP);
         printf ("reprise fils \n");
      }
    else{
       wait (NULL);
       printf ("fin pere \n");
   }
     return EXIT_SUCCESS;
}
```

sig_STOP.c

```
> sig_stop
  signal reçu 20
  reprise fils
   fin père
```

Session de processus

- **Partitionnement des processus du système en sessions**
  - Tout processus appartient exactement à une session
    - Même que celle de son père
  - Processus *leader* d'une session est celui qui a crée la session
    - Primitive de création d'une session: *pid_t setsid (void);*
    - Session identifiée par le PID du leader
  - Lorsque le leader d'une session se termine, tous les processus de la session reçoivent un signal **SIGHUP**
    - processus continuent d'appartenir à la même session qui n'a plus de leader ni de terminal de contrôle.

Session de processus

- Le leader acquiert un terminal de contrôle en ouvrant */dev/tty*
- Un terminal ne peut être terminal de contrôle de plus d'une session
- Exemple :
  - Utilisateur se connecte au système
    - Le processus *login-shell* créé est le leader d'une nouvelle session
      - Terminal de l'utilisateur = terminal de contrôle.
  - Ensemble de processus lancés depuis un *shell* sont sous la même session et terminal de contrôle

Groupe de processus

- **Partitionnement d'une session en groupe de processus**
  - Distinguer parmi les processus attachés à un terminal, ceux qui sont interactifs (*avant-plan*) de ceux qui ne le sont pas (*arrière-plan*)
  - Tout processus appartient, à un instant donné, à un groupe de processus
    - A sa naissance, un processus appartient au même groupe que son père
    - *Leader du groupe* : processus qui a crée le groupe
      - pid_t getpgrp(void)
    - Rattachement d'un processus à groupe:
      - setpgid (pid_t pid, pid_t gpid)
        - Si le groupe de processus *gpid* n'existe pas, il est créé et le processus appelant en devient le leader du groupe.

Groupe de processus (cont.)

- **Groupe de processus en avant-plan (*foreground*)**
  - Un unique groupe des processus en avant-plan par session
  - Processus peuvent accéder au terminal
  - Processus destinataires des signaux SIGINT (ctrl-C), SIGQUIT (ctrl-\) et SIGTSTP (ctrl-Z).
- **Groupe de processus en arrière-plan (*background*)**
  - Ne peuvent pas accéder au terminal
  - Processus ne reçoivent pas les signaux SIGINT (ctrl-C), SIGQUIT (ctrl-\) et SIGTSTP (ctrl-Z) issus d'un terminal.
  - On appelle *job (travail)* un processus en arrière-plan ou suspendu.

Signal ctrl-C envoyé depuis le terminal de contrôle d'un session

- **Signal ctrl-C est reçu par tous le processus en avant-plan (foreground) sous ce terminal**

```
pid_t pid_fils;
void sig_hand(int sig){
  printf ("%d: signal %d \n", getpid(), sig);
}
int main(int argc, char **argv) {
  sigset_t sig_proc;
  struct sigaction action;

     sigemptyset(&sig_proc);
     action.sa_mask=sig_proc;
     action.sa_flags=0;
     action.sa_handler = sig_hand;

     sigaction(SIGINT, &action,0);
```

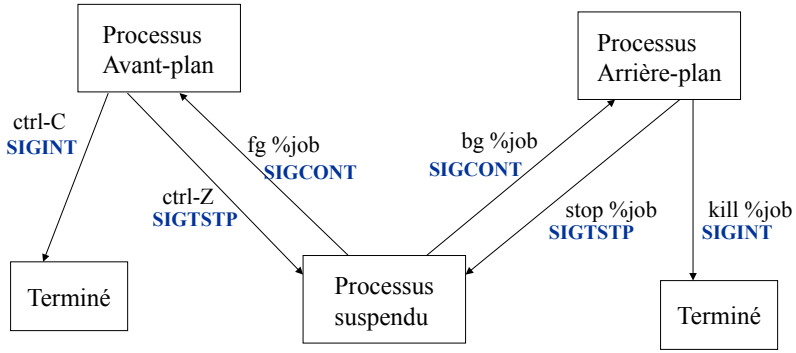
```
if ( (pid_fils= fork ()) == 0) {
    printf("fils: %d\n", getpid( ));
  }
  else{
   printf("père: %d \n", getpid( ));
  }
  sleep (30);
    return EXIT_SUCCESS;
}
```

teste_sig_fg.c

```
> teste_sig_fg
   fils :3736
   père : 3825
>ctrl-C
   3736: signal 2
   3825: signal 2
```


Gestion de Jobs

- **Gestion par des signaux**



> jobs /* commande pour obtenir la liste des jobs */

Utilisation des temporisateurs (*alarm* et *settimer*)

- **But : Interrompre le processus au terme d'un délai**
  - Processus arme un temporisateur (timer). Lorsque le délai fixé arrive à son terme, le processus reçoit un signal.
  - Un seul temporisateur par processus
  - Utilisation des fonctions *alarm* ou *setitimer*
    - ***alarm*** : temps réel mais la résolution est en secondes.
      - Signal reçu : SIGALRM
    - ***setitimer*** : permet de définir de temporisateurs de différents types avec une résolution plus fine que la seconde.
      - Signal reçu : SIGALRM, SIGTVALRM ou SIGPROF
  - Terminaison du processus est le traitement par défaut du signal reçu

alarm( ) - SIGALRM

- **alarm(int sec);**
  - Durée exprimée en secondes
    - Temps-réel (wall-clock time) dont la résolution est à la seconde
  - Un SIGALRM est généré à son terme
  - Un seul temporisateur par processus
    - Une nouvelle demande annule la précédente.
    - Un appel avec la valeur 0 annule la demande en cours.

alarm( ) : SIGALRM (Exemple)

```
void sig_hand(int sig){
   printf ("signal reçu %d \n",sig);
   alarm (1);
 }

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

  sigset_t sig_proc;
  struct sigaction action;
  sigemptyset(&sig_proc);
```

```
action.sa_mask=sig_proc;
action.sa_flags=0;
action.sa_handler = sig_hand;
sigaction(SIGALRM, &action,0);
 alarm (1);
 while (1)
     pause ();
 return EXIT_SUCCESS;
}
```

sig_ALRM.c

```
>sig_ALRM
  signal reçu 20
  signal reçu 20
  signal reçu 20
 …..
```


setitimer ( ) SIGALRM, SIGVTALRM, SIGPROF

- **Résolution plus fine que la seconde**
  - Utilise deux structures définies dans <sys/times.h>
    - Durée :

```
struct timeval {
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds */
};
```

    - Intervalle de réarmement

```
struct itimerval {
  struct timeval it_interval; /* next value */
  struct timeval it_value; /* current value */
 };
```

    - Timer périodique :
      - Une écheance à *it_value* puis une toutes les *it_interval*
        - *it_value* = 0 : annulation; *it_interval*=0 : pas de réarmement

setitimer ( ) SIGALRM, SIGVTALRM, SIGPROF

- **Primitive *setitimer* permet trois type d'alarmes**

```
#include <sys/time.h>
int setitimer (int type, struct itimerval * new, struct itimerval *old);
```

TYPE	TEMPORISATION	SIGNAL
ITER_REAL	Temps réel	SIGALRM
ITER_VIRTUAL	Temps en mode utilisateur	SIGVTALRM
ITER_PROF	Temps CPU total	SIGPROF

Exemple *setitimer* - SIGVTALRM

```
#define TICK ((double) sysconf(_SC_CLK_TCK))
struct itimerval itv;
struct tms start, end;
struct sigaction action;

void handler(int sig) {
  times(&end);
  printf(" Temps usager %4.2f\n",
      (end.tms_utime - start.tms_utime) /TICK) ;
  times(&start);
}
```

sig_VTALRM.c

```
>sig_ VTALRM
   Temps usager : 1.50
   Temps usager: 0.50
   Temps usager: 0.50
  …..
```

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

action.sa_handler = handler;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
sigaction(SIGVTALRM, &action,
        (struct sigaction *) 0);

 itv.it_value.tv_sec = 1;
 itv.it_value.tv_usec = 500000;

 itv.it_interval.tv_sec = 0;
 itv.it_interval.tv_usec = 500000;
 times(&start);
 setitimer(ITIMER_VIRTUAL, &itv,
          (struct itimerval *)0);
 while (1);
 return EXIT_SUCCESS;
  }
```

SIGSEGV – violation de mémoire

- **Point de rupture (*breakpoint*)**
  - Plus petite adresse non utilisée de l'espace de données
  - La mémoire est découpée en blocs de taille fixe appelés pages.
    - Unité d'allocation.
    - Protection assurée au niveau de chaque page.
- **Signal SIGSEGV**
  - Indique toutes les violations de mémoire lorsque le processus accède en écriture à une adresse en dehors de son espace d'adressage.

SIGSEGV – violation de mémoire

- **Primitives pour augmenter l'espace de données**
  - Le point de rupture est positionné au placement demandé mais l'unité d'allocation est une page
    - **brk(const void *adr);**
      - Attribue au point de rupture l'adresse spécifiée par *adr*.
    - **sbrk(int offset)**
      - Ajout de *offset* bytes à la valeur courante du point de rupture.
      - valeur renvoyée:
        - Valeur antérieure avant modification du point de rupture

SIGSEGV - exemple

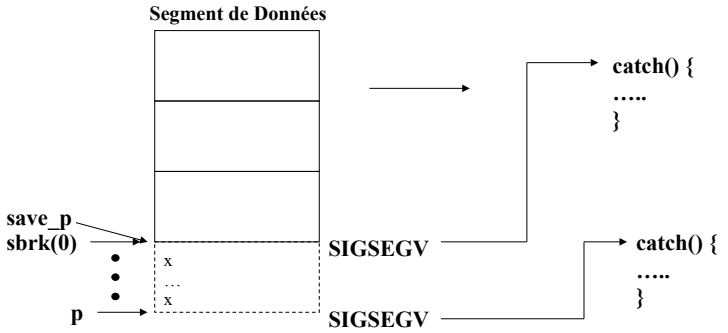
sig_SEGV.c

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

struct sigaction sa;
char *p;

void catch(int signum) {
   static int *save_p = NULL;
   if (save_p == NULL) {
        save_p = (int*)p; brk(p+1);
   }
  else {
     printf("Taille page %d\n", (p - (char*)save_p) );
     exit(0);
  }
}
```

```
int main(int argc, char *argv[]) {

sa.sa_handler = catch;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;

sigaction(SIGSEGV, &sa, NULL);

p = (char*)sbrk(0);
while (1) *p++ = 'x';

}
```

```
>sig_SEGV
Taille page: 4096
```


SIGSEGV – exemple (cont)



- sbrk(0) : point de rupture courant
- brk(p+1) : positionne le point de rupture à p+1, mais l'unité d'allocation est une page
- taille page = 4096 = p /*deuxième SIGSEGV*/ – save_page /* premier SIGSEGV */

Le contrôle de point de reprise

- **Fonctions qui permettent le positionnement sur la pile d'exécution en assurant la sauvegarde et la restauration des masques de signaux :**
  - **int sigsetjmp** (sigjmp_buf env, int ind)
    - Sauvegarder dans l'objet *env* la valeur d'environnement.
    - Si *ind* < > nul, le masque courant est sauvegardé.
    - Valeur de retour :
      - Appel direct à *sigsetjmp* : 0
      - Appel indirect provoqué depuis *siglongjmp* : *val*
  - **int siglongjmp** (sigjmp_buf env, int val)
    - Restaurer un environnement sauvegardé au moyen de la fonction *sigsetjmp*.
    - *val* est la valeur donnée à l'appel ainsi simulé.
    - Un appel à la fonction *siglongjmp* suppose que le paramètre *env* a été initialisé précédemment par un appel direct à *sigsetjmp*.

Exemple – sigsetjmp et siglongjmp

```
int sig;
sigjmp_buf env;

void sig_hand(int sig){
  siglongjmp (env,sig);
}

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

  struct sigaction action;
  action.sa_handler = sig_hand;
  sigaction (SIGINT, &action, NULL);

  if ((sig = sigsetjmp (env,1)) == SIGINT)
     printf ("SIGINT reçu \n");
  else
    pause ();
  return EXIT_SUCCESS;
}
```


Fonctions non POSIX : *raise*

- **int raise (int sig);**
  - Envoie le signal *sig* donné au processus courant.
  - Equivaut à *kill (getpid(), sig)*

```
void sig_hand(int sig){
  printf ("signal reçu %d \n",sig);
}

int main(int argc, char **argv) {
  sigset_t sig_proc;
  struct sigaction action;

     sigemptyset(&sig_proc);
```

```
/* changer le traitement */
     action.sa_mask=sig_proc;
     action.sa_flags=0;
     action.sa_handler = sig_hand;
  sigaction(SIGINT, &action,NULL);

  raise (SIGINT);
  return EXIT_SUCESSS
```


Fonctions non POSIX: *signal*

- **typedef void (*sighandler_t)(int);**

  sighandler_t signal(int *sig*, sighandler_t *handler*);
  - installe un nouveau gestionnaire pour le signal numéro *sig*. Le gestionnaire de signal est *handler* qui peut être soit une fonction , soit une des constantes **SIG_IGN** ou **SIG_DFL**.
  - **VALEUR RENVOYÉE**
    - valeur précédente du gestionnaire de signaux, ou **SIG_ERR** en cas d'erreur.

Fonctions non POSIX : *signal*

- **Limitations:**
  - Impossible d'apposer un masque de signaux pendant l'exécution du *handler*
  - Le traitement par défaut est réinstallé après l'exécution du handler

```
void sig_hand(int sig){
  printf ("singal reçu %d \n",sig);
}

int main(int argc, char **argv) {
   signal(SIGINT, &sig_hand);
   kill (getpid(), SIGINT);
   return EXIT_SUCCESS;
}
```


Fonctions non POSIX: *siginterrupt*

- **int siginterrupt(int *sig*, int *flag*);**
  - modifie le comportement d'un appel système interrompu par le signal *sig*.
    - Si *flag* vaut 0 alors l'appel système interrompu par le signal *sig* sera relancé automatiquement
    - Equivaut à positionner le drapeau *sa_flags* de la structure *sigaction* à la valeur SA_RESTART, si flag égal à 0; sinon le positionner à ~SA_RESTART.
  - Utiliser plutôt *sigaction* avec le flag SA_RESTART au lieu de *siginterrupt*.

Manipulation de signaux - Shell

- Commande **trap**
  - **trap** *liste_commande signum1 signum2 ...*
    - Association d'un traitement à un ou plusieurs signaux
  - **trap**
    - Liste des traitements associés
  - **trap** *" " signum1 signum2 ...*
    - Ignorer les signaux
  - **trap** *signum1 signum2*
    - Revenir au traitement par défaut

Manipulation de signaux - Shell

- **Exemple :**

> trap pwd 2 3
> trap "echo signal 15 reçu" 15
> kill -2 387 /* 387 : pid du bash */
/home/arantes/Enseignement/LI356
> kill -15 387 /*387 : pid du bash */
signal 15 reçu
> trap 2 3 15 /* traitement par défaut */

- **kill -2 387** /* 387 : pid du bash */
  - Bash tué

Trap – exemple scrip shell

Processus pere.sh

```
#! /bin/bash
trap "echo : $$ a recu SIGINT" SIGINT
echo "debut du pere $$"
sleep 2
./fils.sh
sleep 2
echo pere fini
```

Processus fils.sh

```
#! /bin/bash
trap "echo : $$ a recu SIGINT" SIGINT
echo "debut du fils $$"
sleep 30
echo fils fini
```

- **SIGINT envoyé lorsque les processus fils et père sont vivants**
  - Reçu par tous les processus rattachés au terminal qui ne sont pas en arrière-plan.

```
debut du pere 2227
debut du fils 2229
> ctrl-C
2229 a recu SIGINT
fils fini
2227 a recu SIGINT
pere fini
```


Limites des Signaux

- **Quelques limitations de signaux:**
  - Aucune mémorisation du nombre de signaux reçus
  - Aucune mémorisation de la date de réception d'un signal
    - les signaux seront traités par ordre de numéro.
  - Aucun moyen de connaître le PID du processus émetteur du signal.

Cours 4 : Entrées / Sorties

- **Primitives d'entrées-sorties POSIX**
  `unistd.h, sys/stat.h, sys/types.h, fcntl.h`
  - Constituent l'interface avec le noyau Unix (**appels systèmes**)
    permettent l'utilisation des services offerts par le noyau.
  - Portabilité des programmes sur Unix.

- **Bibliothèque d'entrées-sorties standard C**
  `stdio.h`
  - + grand niveau de portabilité : indépendance du système.
  - Surcouche d'optimisation (eg. suite d'appels à *write*)
    accès asynchrones, bufferisés et formatés (type).

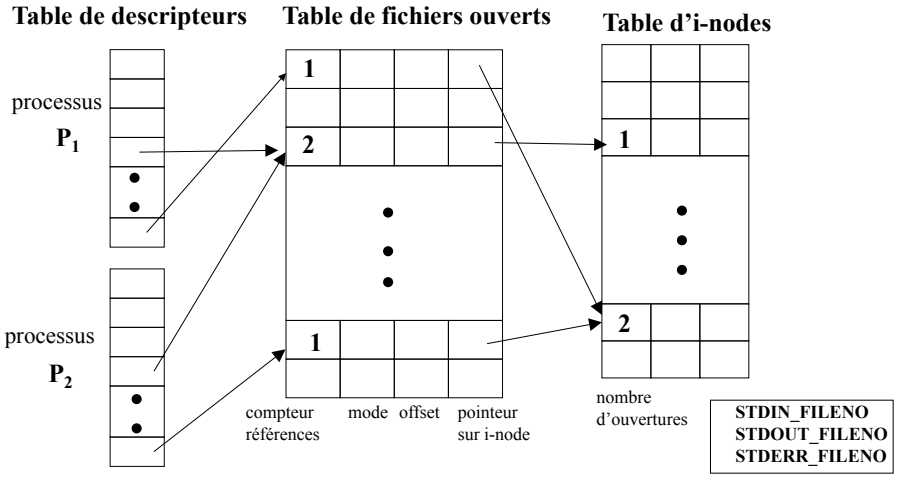
Fichiers d'en-tête

- **Types de base universels (= portables).**
  *eg. FILE**
- **Constantes symboliques.**
  eg. NBBY (8)
- **Structures et types utilisés dans le noyau.**
  eg. struct stat
- **Prototypes des fonctions.**
  *eg. FILE *fopen(const char *, const char *);*

Quelques constantes de configuration (POSIX)

- **LINK_MAX**
  nb max de liens physiques par i-node (8).
- **PATH_MAX**
  longueur max pour le chemin (nom) d'un fichier (255).
- **NAME_MAX**
  longueur max des noms de liens (14).
- **OPEN_MAX**
  nb max d'ouvertures de fichiers simultanées par processus (16).

Organisation des Tables



Quelques erreurs associées aux E/S

#include <errno.h>

extern int errno;

- **EACCESS** : accès interdit.
- **EBADF**: descripteur de fichier non valide.
- **EEXIST** : fichier déjà existant.
- **EIO**: erreur E/S.
- **EISDIR**: opération impossible sur un répertoire.
- **EMFILE**: trop de fichiers ouverts pour le processus (> OPEN_MAX).
- **EMLINK** : trop de liens physiques sur un fichier (> LINK_MAX).
- **ENAMETOOLONG** : nom fichier trop long (>PATH_MAX)
- **ENOENT** : fichier ou répertoire inexistant.
- **EPERM** : droits d'accès incompatible avec l'opération.

Consultation de l'i-node (stat)

- **Structure stat** <sys/stat.h>

```
struct stat {
        dev_t       st_dev;     /* device file resides on */
        ino_t       st_ino;     /* the file serial number */
        mode_t      st_mode;    /* file mode */
        nlink_t     st_nlink;    /* number of hard links to the file*/
        uid_t       st_uid;     /* user ID of owner */
        gid_t       st_gid;     /* group ID of owner */
        dev_t       st_rdev;    /* the device identifier*/
        off_t       st_size;     /* total size of file, in bytes */
        unsigned long  st_blksize; /* blocksize - file system I/O*/
        unsigned long  st_blocks;  /* number of blocks allocated */
        time_t      st_atime;   /* file last access time */
        time_t      st_mtime;   /* file last modify time */
        time_t      st_ctime;   /* file last status change time */
}
```


Type de fichier

Champ *st_mode* de *struct stat*
Type : masque S_IFMT (POSIX : macros)

- **Fichiers réguliers : données (S_IFREG)**
  - macro: S_ISREG (t)
- **Répertoires (S_IFDIR)**
  - macro: S_ISDIR (t)
- **Tubes FIFO (S_FIFO)**
  - macro: S_ISFIFO (t)
- **Fichiers spéciaux : périphs bloc (S_IFBLK) ou caractère (S_IFCHR)**
  - macro: S_ ISBLK (t) et S_ ISCHR (t)
- **Liens symboliques (S_IFLNK)**
  - macro: S_ISLNK (t)
- **Sockets (S_IFDOOR)**
  - macro: S_ISSOCK (t)

Droits d'accès

- **Propriétaire, groupe et autres** **(Champ *st_mode* de *struct stat*)**
  - lecture, écriture et exécution

	Propriétaire	**Groupe**	**Autres**
Lecture	S_IRUSR	S_IRGRP	S_IROTH
Ecriture	S_IWUSR	S_IWGRP	S_IWOTH
Exécution	S_IXUSR	S_IXGRP	S_IXOTH
Les trois	S_IRWXU	S_IRWXG	S_IRWXO

> ls - l **rwxr-xr--**

S_IRWXU| S_IRGRP | S_IXGRP| S_IROTH

Fonctions de consultation de l'i-node

- **Obtention des caractéristiques d'un fichier**
  - **int stat(const char *file_name, struct stat *buf);**
  - **int fstat(int fdes, struct stat *buf);**
    - Résultats récupérés dans une *struct stat*
- **Test des droits d'accès d'un processus sur un fichier**
  - **int access (const char* pathname, int mode);**
    - mode : **R_OK**, **W_OK**, **X_OK**, **F_OK**
      (droit de lecture, écriture, exécution, existence).

Exemple - stat

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
 struct stat stat_info;

 if ( stat (argv[1], &stat_info) == -1)
   { perror ("erreur stat");
    return EXIT_FAILURE;
   }

 if (S_ISDIR (stat_info.st_mode) )
    printf ("fichier répertoire\n");

 printf ("Taille fichier : %ld\n", (long)stat_info.st_size);

 if (stat_info.st_mode & S_IRGRP)
   printf ("les usagers du même goupe peuvent lire le fichier\n");

 return EXIT_SUCCESS;
}
```


Manipulation de liens physiques

- **Création d'un lien physique sur un répertoire**
  - **int link (const char *origine, const char *cible)**
    - permet de créer un nouveau lien physique
    - contraintes
      - *origine* ne peut pas être un répertoire
      - *cible* ne doit pas exister

```
> ln Fic1 Fic2
> ls -ia
```

24	.
43	..
78	Fic1
78	Fic2

- **Suppression d'un lien physique**
  - **int unlink (const char *ref)**
    - supprime le lien associé à *ref*
    - fichier supprimé si:
      - nombre de liens physiques sur le fichier est nul
      - nombre d'ouvertures du fichier est nul
- **Changement de nom de lien physique**
  - **int rename (const char *ancien, const char *nouveau)**
    - *nouveau* ne doit pas exister
    - impossible de renommer . et ..

code renvoi : 0 (succès) ; -1 (erreur)

Changement d'attributs d'un i-node

- **Droits d'accès**
  - **int chmod (const char* reference, mode_t mode);**
  - **int fchmod (int descripteur, mode_t mode);**
    attribution des droits d'accès *mode* au fichier :
    - de nom *reference*
    - associé à *descripteur*
- **Propriétaire**
  - **int chown (const char* reference, uid_t uid, gid_t gid);**
  - **int fchown (int descripteur, uid_t uid, gid_t gid);**
    modification du propriétaire *uid* et du groupe *gid* d'un fichier

code renvoi : 0 (succès) ; -1 (erreur)

Exemple - chmod

test-chmod.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
   if (chmod (argv[1], S_IRUSR | S_IWUSR| S_IRGRP | S_IWGRP | S_IROTH |
     S_IWOTH) == 0)
        printf ("fichier %s en lecture-ecriture pour tous les usagers \n ", argv[1]);
  else { perror ("chmod");
         return EXIT_FAILURE;
       }
   return EXIT_SUCCESS;
}
```


Primitives de base (1)

- **Ouverture d'un fichier : open**
  - **int open (const char* reference, int flags);**
  - **int open (const char* reference, int flags, mode_t droits);**
    - renvoie un numéro de descripteur
    - **flags**: – **O_RDONLY** : ouverture en lecture
      – **O_WRONLY** : ouverture en écriture
      – **O_RDWR** : ouverture en lecture-écriture
      – **O_CREAT**: création d'un fichier s'il n'existe pas
      – **O_TRUNC**: vider le fichier s'il existe
      – **O_APPEND** : écriture en fin de fichier
      – **O_SYNC** : écriture immédiate sur disque
      – **O_NONBLOCK** : ouverture non bloquante
    - **droits**: lecture, écriture, exécution

code renvoi :
descripteur (succès)
-1 (erreur)

Primitives de base (2)

- **Fermeture de fichier : close**
  - **int close (int descripteur);**
    - Ferme le descripteur correspondant à un fichier en désallouant son entrée de la table des descripteurs du processus.
    - Si nécessaire, mise à jour table des fichiers et table des i-nodes.
- **Création d'un fichier**
  - **int creat (const char* reference, mode_t droits);**
    correspond à l'appel suivant:
    open (reference, int flags, O_WRONLY | O_CREAT | O_TRUNC, droits);

Primitives de base (3)

- **Lecture dans un fichier : `read`, `readv`, `pread`**
  - **ssize_t read (int desc, void* tampon, size_t nbr);**
    - Demande de lecture d'au + *nbr* caractères du fichier correspondant à *desc.*
    - Les caractères lus sont écrits dans *tampon.*
    - Renvoie le nombre de caractères lus ou -1 en cas d'erreur.
    - La lecture se fait à partir de la **position courante**
      offset de la *Table des Fichiers Ouverts ;* mise à jour après la lecture.
  - **ssize_t readv (int desc, const struct iovec* vet, int n);**
    - Données récupérées dans une *struct iovec* de taille *n.*

```
struct iovec {
   void *iov_base;
   size_t iov_len; }
```

  - **ssize_t pread (int desc, void* tampon, size_t nbr, off_t pos);**
    - Lecture à partir de la position *pos* ; *offset* n'est pas modifié.

Primitives de base (4)

- **Ecriture dans un fichier : `write`, `writev`, `pwrite`**
  - **ssize_t write (int desc, void* tampon, size_t nbr);**
    - Demande d'écriture de *nbr* caractères contenus à partir de l'adresse *tampon* dans le fichier correspondant à *desc*.
    - Renvoie le nombre de caractères écrits ou -1 en cas d'erreur.
    - L'écriture se fait à partir de la fin du fichier (O_APPEND) ou de la position courante.
    - Modifie le champ *offset* de la *Table des Fichiers Ouverts.*
  - **ssize_t writev (int desc, const struct iovec* vet, int n);**
  - **ssize_t pwrite (int desc, void* tampon, size_t nbr, off_t pos);**

Exemple – open, read et write

test-rw.c

```
#define _POSIX_SOURCE 1
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define SIZE_TAMPON 100
char tampon [SIZE_TAMPON];
int main (int argc, char* argv []) {
  int fd1, fd2;    int n,i;

  fd1 = open (argv[1], O_WRONLY|O_CREAT|
              O_SYNC,0600);
  fd2 = open (argv[1], O_RDWR);

  if ( (fd1== -1) || (fd2 == -1)) {
     printf ("open %s" ,argv[1]);
    return EXIT_FAILURE;
}
```

```
  if (write (fd1,"abcdef", strlen ("abcdef")) == -1) {
     perror ("write");
     return EXIT_FAILURE;
  }
  if (write (fd2,"123", strlen ("123") ) == -1) {
     perror ("write");
     return EXIT_FAILURE;
  }
  if ((n= read (fd2,tampon, SIZE_TAMPON)) <=0) {
      perror ("fin fichier\n");
      return EXIT_FAILURE;
  }
  for (i=0 ; i<n; i++)
    printf ("%c",tampon [i]);

  return EXIT_SUCCESS;
}
```


Primitives de base (5)

- **Manipulation de l'offset: `lseek`**
  - **off_t lseek (int desc, off_t position, int origine);**
    - Permet de modifier la position courante *(offset*) de l'entrée de la *Table de Fichiers Ouverts* associée à *desc*.
    - La position courante prend comme nouvelle valeur : *position* + *origine.*
    - **origine**:
      - **SEEK_SET**: 0 (début du fichier)
      - **SEEK_CUR** : Position courante
      - **SEEK_END** : Taille du fichier
    - Renvoie la nouvelle position courante ou -1 en cas d'erreur.

Exemple – lseek

test-lseek.c

```
#define _POSIX_SOURCE 1
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define SIZE_TAMPON 100
  char tampon [SIZE_TAMPON];
int main (int argc, char* argv []) {
 int fd1, fd2;    int n,i;

 fd1 = open (argv[1], O_WRONLY|O_CREAT|
             O_SYNC,0600);
 fd2 = open (argv[1], O_RDWR);

 if ( (fd1== -1) || (fd2 == -1)) {
   printf ("open %s" ,argv[1]);
   return EXIT_FAILURE;
}
```

```
  if (write (fd1,"abcdef", strlen ("abcdef")) == -1) {
     perror ("write");
     return EXIT_FAILURE;
  }
 if (write (fd2,"123", strlen ("123") ) == -1) {
     perror ("write");
     return EXIT_FAILURE;
 }
/* déplacement au début du fichier */
 if (lseek(fd2,0,SEEK_SET) == -1) {
   perror ("seek");
   return EXIT_FAILURE;
}
 if ((n= read (fd2,tampon, SIZE_TAMPON)) <=0) {
    perror ("fin fichier\n");
    return EXIT_FAILURE; }
 for (i=0 ; i<n; i++)
    printf ("%c",tampon [i]);
 return EXIT_SUCCESS;
}
```

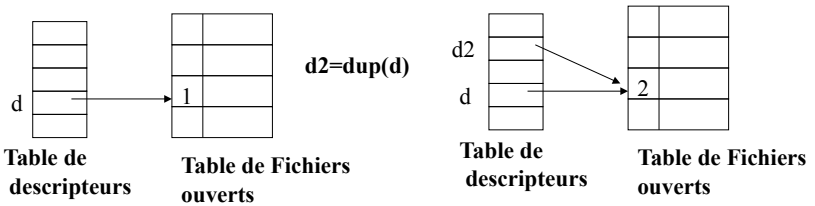
Exemple – fork

test-fork.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

#define SIZE_TAMPON 100
 char tampon [SIZE_TAMPON];

 int main (int argc, char* argv []) {
   int fd1, fd2;   int n,i;
  if ((fd1 = open (argv[1], O_RDWR| O_CREAT |
              O_SYNC,0600)) == -1) {
       perror ("open \n");
      return EXIT_FAILURE;
}
if (write (fd1,"abcdef", strlen ("abcdef")) == -1) {
   perror ("write");
   return EXIT_FAILURE; }
if (fork () == 0) {
   /* fils */
     if ((fd2 = open (argv[1], O_RDWR)) == -1) {
       perror ("open \n");
       return EXIT_FAILURE;
     }
     if (write (fd1,"123", strlen ("123")) == -1) {
       perror ("write");
       return EXIT_FAILURE;
     }
     if ((n= read (fd2,tampon, SIZE_TAMPON)) <=0) {
       perror ("fin fichier\n");
       return EXIT_FAILURE;
     }
     for (i=0 ; i<n; i++)
       printf ("%c",tampon [i]);
     exit (0);
 }
 else /* père */
   wait (NULL);
 return EXIT_SUCCESS;
}
```


Duplication de descripteur

- **La primitive dup**
  - **int dup (int desc);**
    - Recherche le + petit descripteur disponible dans la table des descripteurs du processus et en fait un synonyme de *desc*.
  - **int dup2 (int desc, int desc2);**
    - Force le descripteur *desc2* à devenir synonyme de *desc*.



Exemple – dup2

Redirection de stdout

```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
int fd1;
int main (int argc, char* argv []) {

 if ((fd1 = open (argv[1], O_WRONLY| O_CREAT,0600)) == -1) {
     perror ("open \n");
     return EXIT_FAILURE;
  }

 printf ("avant le dup2: descripteur %d \n", fd1);
 dup2 (fd1, STDOUT_FILENO);
printf ("après le dup2 \n");

 return EXIT_SUCCESS;
}
```


Liens symboliques

- **int symlink (const char* reference, const char* lien);**
  - créer un lien symbolique sur le fichier *reference*
- **int lstat (const char* reference, struct stat* pStat);**
- **ssize_t readlink (const char* ref, char* tampon, size_t taille);**
  - récupère à l'adresse *tampon* la valeur du lien symbolique (son contenu)
- **lchmod (const char* reference, mode_t mode);**
- **lchown (const char* reference, uid_t uid, gid_t gid);**

Les entrées-sorties sur répertoires (1)

- **Ouverture d'un répertoire**
  - **DIR * opendir (const char* reference);**
    - ouvre en lecture le répertoire de référence *reference*
    - lui alloue un objet du *type DIR*, dont l'adresse est renvoyée au retour
- **Lecture d'une entrée**
  - **struct dirent* readdir (DIR *pDir);**
    - lit l'entrée courante du répertoire associé à *pDir*
    - place le pointeur sur l'entrée suivante
    - renvoie un pointeur de type *struct dirent*

```
struct dirent {
   …
   char d_name [];
}
```

    - renvoie NULL en cas d'erreur ou lorsque la fin de fichier est atteinte

Les entrées-sorties sur répertoires (2)

- **Rembobinage**
  - **void rewinddir (DIR *pDir);**

    Repositionne le pointeur des entrées associé à *pDir* sur la 1re entrée dans le répertoire.
- **Fermeture**
  - **int closedir (DIR *pDir);**

    Ferme le répertoire associé à *pDir*.

    Les ressources allouées au cours de l'appel à *opendir* sont libérées.

Exemple – Lister le contenu d'un répertoire

test-listdir.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

DIR *pt_Dir;
struct dirent* dirEnt;
```

```
int main (int argc, char* argv []) {
 if ( ( pt_Dir = opendir (argv[1]) ) == NULL) {
    perror ("opendir");
     return EXIT_FAILURE;
 }
  while ((dirEnt= readdir (pt_Dir)) !=NULL)
    printf ("%s\n", dirEnt->d_name);
 closedir (pt_Dir);
 return EXIT_SUCCESS;
}
```

```
>test-listdir rep1
   .
   ..
   fich1
   fich2
```


Ecritures dans les répertoires

- **Création d'un répertoire**
  - **int mkdir (const char* ref, mode_t mode);**
    - Création d'un répertoire vide en spécifiant les droits.
- **Supprimer un répertoire**
  - **int rmdir (const char* ref);**
- **Renommer un répertoire**
  - **int rename (const char* ancien, const char* nouveau);**

code renvoi : 0 (succès) ; -1 (erreur)

Obtention/modification des attributs d'un fichier

- **Fonction fcntl**
  - Permet de modifier des attributs associés à un descripteur
    - mode d'ouverture
    - duplication du descripteur
    - verrouillage des zones du fichier
  - Signature dépend de la valeur de `cmd`
    - `int fcntl(int fd, int cmd);`
    - `int fcntl(int fd, int cmd, long arg);`
    - `int fcntl(int fd, int cmd, struct flock *lock);`

Obtention/modification des attributs d'un fichier

- **Fonction fcntl**
  - **Argument *`cmd`* :**
    - F_GETFD : obtenir la valeur des attributs du descripteur
    - F_SETFD : modifier les attributs du descripteur
    - F_GETFL : obtenir la valeur des attributs du mode d'ouverture
    - F_SETFL : modifier le mode d'ouverture
      `O_APPEND, O_NONBLOCK, O_NDELAY, O_SYNC`
    - F_DUPFD : duplication de descripteur dans le + petit descripteur disponible
      `fcntl (fd, F_DUPFD, 0);` est équivalent à *`dup (fd);`*
    - F_SETLK, F_SETLKW, F_GETLK : verrouillage
  - **Code de retour :**
    - F_DUPFD : le nouveau descripteur, sinon -1 en cas d'erreur (voir errno)
    - F_GETFD, F_GETFL: valeur des attributs, sinon -1 en cas d'erreur (voir errno)
    - F_SETFD, F_SETFL : 0 en cas de succès, -1 en cas d'erreur (voir errno)

Verrou sur fichier

- **Fonction fcntl**
  Le verrouillage est consultatif !
  Verrouillage effectif ⇒ sémaphores
  - **Type de verrou**
    - Exclusif (write) : aucun autre processus ne peut verrouiller une zone avec verrou exclusif
    - Partagé (read) : tout autre processus peut verrouiller une zone avec verrou partagé
  - **`struct flock`**

```
off_t l_start;    /* starting offset */
off_t l_len;      /* len = 0 means until end of file */
pid_t l_pid;      /* lock owner */
short l_type;     /* lock type: F_RDLCK, F_WRLCK, F_UNLCK */
short l_whence;   /* type of l_start */
```

  - **Argument *`cmd`***
    F_GETLK : récupérer le verrou courant sur la zone définie par `lock`
    F_SETLK : (dé)verrouiller la zone définie par `lock`
    F_SETLKW : idem, mais attente bloquante jusqu'à ce que la zone soit libérée d'autres verrous

Exemple – Fonction fcntl

fcntl-test.c

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main (int argc, char* argv []) {
 int fd1, mode_ouv;

 if ((fd1 = open (argv[1], O_RDWR)) == -1){
  perror ("open"); return EXIT_FAILURE;
 }

 if (write (fd1, "abc", 3) == -1) {
  perror ("write"); return EXIT_FAILURE;
 }

mode_ouv = fcntl (fd1, F_GETFL);
 mode_ouv |= O_APPEND;
 fcntl (fd1,F_SETFL, mode_ouv);

  if (write (fd1, "xyz", 3) == -1) {
   perror ("write"); return EXIT_FAILURE;
  }

  close (fd1);
  return EXIT_SUCCESS;
}
```

La bibliothèque E/S standard C

- **Constitue une couche au-dessus des appels système correspondant aux primitives de base d'E/S POSIX.**
- **But : travailler dans l'espace d'adressage du processus**
  - E/S dans des tampons appartenant à cet espace d'adressage
  - Objet de type FILE, obtenu lors de l'appel à la fonction *fopen* :
    - permet de gérer le tampon associé au fichier
    - possède le numéro du descripteur du fichier
      - **STDIN_FILENO = stdin**
      - **STDOUT_FILENO = stdout**
      - **STDERR_FILENO = stderr**
  - `fflush` force l'écriture du contenu du tampon dans les caches système

La bibliothèque E/S standard C

- **Fichier <stdio.h>**

  Constantes:
  - **NULL** : adresse invalide
  - **EOF** : reconnaissance de fin de fichier
  - **FOPEN_MAX**: nb max de fichiers manipulables simultanément
  - **BUFSIZ**: taille par défaut des tampons

La bibliothèque E/S standard C

- **Fichier <stdio.h>**
  - **Types:**
    - **FILE**: type dédié à la manipulation d'un fichier
      Gère le tampon d'un fichier ouvert.
    - **fpos_t**: position dans un fichier
    - **size_t**: longueur du fichier
  - **Objets prédéfinis de type FILE*:**
    - **stdin:** objet d'entrée standard
    - **stdout** : objet de sortie standard
    - **stderr :** objet de sortie-erreur standard

Fonctions de base (1)

- **Ouverture d 'un fichier**
  - **FILE* fopen (const char*reference, const char *mode);**
    - Arguments
      - *reference* chemin d'accès au fichier
      - *mode* mode d'ouverture
    - Renvoie un pointeur vers un objet *FILE* associé au fichier, NULL si échec.
    - Association d'un *tampon* pour les lectures/écritures, et d'une *position courante*.
    - *mode*:

`r`	**lecture seulement.**
`r+`	**lecture et écriture sans création ou troncature du fichier.**
`w`	**écriture avec création ou troncature du fichier.**
`w+`	**lecture et écriture avec création ou troncature du fichier.**
`a`	**écriture en fin de fichier ; création si nécessaire.**
`a+`	**lecture et écriture en fin de fichier ; création si nécessaire.**

Fonctions de base (2)

- **Nouvelle ouverture d'un fichier**
  - **FILE* freopen (const char* reference, const char *mode, FILE* pFile);**
    - Associe à un objet déjà alloué une nouvelle ouverture.
    - Redirection d'E/S.
    - **Exemple: Redirection sortie standard**
      - freopen ("fichier1","w", stdout);
- **Obtention du descripteur associé à l'objet FILE**
  - **int fileno (FILE* pFile);**
- **Obtention d'un objet du type FILE à partir d'un descripteur.**
  - **FILE *fdopen (const int desc, const char *mode);**
    - Le *mode* d'ouverture doit être compatible avec celui du descripteur.

Exemple – fdopen

fdopen-test.c

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

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

int fd;
FILE *ptFile;
 if ( (fd = open (argv[1], O_RDWR |
  O_CREAT)) == -1) {
  perror ("open"); exit (1);
}

 if ((ptFile = fdopen (fd,"w+")) == NULL) {
  perror ("fdopen"); exit (1);
}

 if (write (fd,"ab",2)==-1) {
    perror ("write");exit (1);
 }

  if (fputs ("cd",ptFile) == -1) {
     perror ("fputs");exit (1);
  }
  return (EXIT_SUCCESS);
  }
```


Fonctions de base (3)

- **Test de fin de fichier**
  - **int feof (FILE *pFile);**
    - associé aux opérations de lecture
    - renvoie une valeur ≠ 0 si la fin de fichier associée à *pFile* a été détectée
- **Test d'erreur**
  - **int ferror (FILE *pFile);**
    - renvoie une valeur ≠ 0 si une erreur associée à *pFile* a été détectée
- **Fermeture d'un fichier**
  - **int fclose(FILE *pFile);**
    - ferme le fichier associé à *pFile*.
    - Transfert de données du tampon associé.
    - Libération de l'objet *pFile*.
    - Renvoie 0 en cas de succès et EOF en cas d'erreur.

Gestion du tampon

- **A chaque ouverture de fichier**
  tampon de taille BUFSIZ est automatiquement alloué
- **Association d'un nouveau tampon:**
  - **int setvbuf(FILE *pFile, char* tampon, int mode, size_t taille);**
    - Permet d'associer un nouveau tampon de taille *taille* à *pFile*.
    - Critère de vidage (*mode*)
      - _IOFBF: lorsque le tampon est plein
      - _IOLBF: lorsque le tampon contient une ligne ou est plein
      - _IONBF: systématiquement
- **Vidage du tampon**
  - **int fflush (FILE *pFile);**
    Si *pFile* vaut NULL, tous les fichiers ouverts en écriture sont vidés

Fonctions de base (4)

- **Lecture**
  - **Un caractère**
    - **int fgetc (FILE* pFile);**
      - retourne le caractère suivant du fichier sous forme entière
        EOF en cas d'erreur ou fin de fichier
      - **int getchar (void)** équivalent à **fgetc(stdin)**;
  - **Une chaîne de caractères**
    - **char *fgets (char *pChaine, int taille, FILE* pFile);**
      - lit au + *taille-1* éléments de type *char* à partir de la *position courante* dans *pFile*
      - arrête la lecture si *fin de ligne (\n*, incluse dans la chaîne)
        ou *fin de fichier* est détectée
      - renvoie NULL en cas d'erreur ou fin de fichier
        - Test avec *feof* ou *ferror.*

Exemple fgetc et fgets

fgetc-s-test.c

```
#define POSIX_SOURCE 1
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define TAILLE_BUFF 100

int main (int argc, char ** argv) {
 char c;
 char buff[TAILLE_BUFF];
 FILE *ptLire;

 if ( (ptLire = fopen (argv[1], "r")) == NULL) {
  perror ("fopen"); exit (1);
 }
```

```
/* lecture d'un caractère */
   if ((c=fgetc(ptLire))!= EOF)
      printf ("%c",c);

 /* lecture d'une chaîne */
     if (fgets (buff,TAILLE_BUFF, ptLire)
       !=NULL)
            printf("%s\n",buff);

fclose (ptLire);
  return (EXIT_SUCCESS);
}
```


Fonctions de base (5)

- **Lecture (cont)**
  - **lecture d'un tableau d'objets**

    size_t fread (void *p, size_t taille, size_t nElem, FILE* pFile);
    - Lit au + *nElem* objets à partir de la position courante dans *pFile*.
    - Tableau des objets lus sauvegardé à l'adresse *p*.
    - Chaque objet est de taille *taille.*
    - Retourne le nombre d'objets lus
      0 en cas d'erreur ou fin fichier (test *feof* ou *ferror).*

Fonctions de base (6)

- **Lecture (cont)**
  - **lecture formatée**

    int fscanf (FILE* pFile, const char *format, …);
    - Lit à partir de la *position courante* dans le fichier pointé par *pFile.*
    - *format* : procédures de conversion à appliquer aux suites d'éléments de type *char* lues.
    - **scanf** équivaut à **fscanf** sur **stdin**.
    - Retourne le nombre de conversions réalisées ou EOF en cas d'erreur.

Exemple – scanf

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <stdlib.h>

 int ret, var_int;
char var_char, var_string[10];

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

 ret = scanf ("%c %s %d", &var_char, var_string, &var_int);
 if (ret == 3)
  printf ("var_char=%c, var_string = %s, var_int = %d\n", var_char,
           var_string,var_int);
 else{
    fprintf (stderr, "erreur: ret = %d\n", ret);  return EXIT_FAILURE;
  }
   return EXIT_SUCCESS;
}
```


Fonctions de base (7)

- **Ecriture**
  - **un caractère**
    - **int fputc (int car, FILE* pFile);**
      - Écrit le caractère *car* dans le fichier associé à *pFile*.
      - Renvoie EOF en cas d'erreur ou 0 sinon.
      - **int putchar (int)** équivalent à **fputc** sur **stdout**.
  - **une chaîne de caractères**
    - **int fputs (char *pChaine, FILE* pFile);**
      - Écrit la chaîne *pChaine* dans le fichier associé à *pFile*.
      - Le caractère nul de fin de chaîne n'est pas écrit.
      - Renvoie EOF en cas d'erreur ou 0 sinon.

Exemple – fputc et fputs

fputc-s-test.c

```
#define POSIX_SOURCE 1
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>

int main (int argc, char ** argv) {
 FILE *ptEcr;

 if ( (ptEcr = fopen (argv[1], "w+")) == NULL) {
  perror ("fopen"); exit (1);
 }

 return (EXIT_SUCCESS);
}
```

```
/* écriture d'un caractère */
   if ((fputc('a',ptEcr))== EOF) {
      perror ("fputc");
      exit (1);
   }

  /* ecriture d'une chaîne */
   if (fputs ("bcd", ptEcr) == EOF) {
      perror ("fputs");
      exit (1);
   }

  fclose (ptEcr);
 return (EXIT_SUCCESS);
}
```


Fonctions de base (8)

- **Ecriture (cont)**
  - **Ecriture d'un tableau d'objets**

    size_t *fwrite (void *p, size_t taille, size_t nElems, FILE* pFile);
    - Écrit *nElems* objets de taille *taille* à partir de la *position courante* dans *pFile*
    - Le tableau d'objets à écrire est à l'adresse *p*.
    - Retourne le nombre d'objets écrits
      une valeur inférieure à *nELems* en cas d'erreur.

Fonctions de base (9)

- **Ecriture (cont)**
  - **Ecriture formatée**
    - **int printf (const char *format, ….);**
    - **int fprintf (FILE* pFile, const char *format, ….);**
      - Ecrit dans un fichier associé à *pFile* les valeurs des arguments converties selon le format en chaînes de caractères imprimables.
      - **printf** équivaut à **fprintf** sur **stdout**.
      - Retourne le nombre de caractères écrits ou un nombre négatif en cas d'erreur.

Exemple – fgets et fputs (fscp)

Copier le fichier argv[1] vers argv[2]

```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <stdlib.h>

#define TAILLE_TAMPON 100

 FILE *fd1, *fd2;
 int nombre_car;
char tampon [TAILLE_TAMPON];

int main (int argc, char* argv []) {
  fd1 = fopen (argv[1],"r");
  fd2 = fopen (argv[2], "w");

  if ( (fd1== NULL) || (fd2 == NULL)) {
    fprintf (stderr, "erreur fopen");
    return EXIT_FAILURE;
  }
  while (fgets (tampon, TAILLE_TAMPON, fd1) !=NULL)
    if ( fputs (tampon, fd2) == EOF){
      fprintf (stderr, "erreur fwrite\n");
      return EXIT_FAILURE;
    }
  fclose (fd1);
  fclose (fd2);

  if (ferror (fd1) ) {
    fprintf (stderr, "erreur lecture \n");
    return EXIT_FAILURE;
  }
  else
    return EXIT_SUCCESS;
  }
```


Exemple: fread et fwrite

Copier le fichier argv[1] vers argv[2]

```
#define _POSIX_SOUCE 1

#include <stdio.h>
#include <stdlib.h>

#define TAILLE_TAMPON 100

FILE *fd1, *fd2;
 int nombre_car;
char tampon [TAILLE_TAMPON];

int main (int argc, char* argv []) {
  fd1 = fopen (argv[1],"r");
  fd2 = fopen (argv[2], "w");

  if ( (fd1== NULL) || (fd2 == NULL)) {
    fprintf (stderr, "erreur fopen");
    return EXIT_FAILURE;
  }
  while ((nombre_car = fread (tampon,   sizeof(char),
                TAILLE_TAMPON, fd1)) >0)
if ( fwrite (tampon, sizeof(char),nombre_car, fd2) !=
                nombre_car) {
      fprintf (stderr, "erreur fwrite\n");
      return EXIT_FAILURE;
}

  fclose (fd1);
  fclose (fd2);

if (ferror (fd1) ) {
    fprintf (stderr, "erreur lecture \n");
   return EXIT_FAILURE;
}
else
  return EXIT_SUCCESS;
 }
```


Fonctions de base (10)

- **Manipulation de la position courante**
  - **int fseek (FILE *pFile, long pos, int origine);**
    - positionne le curseur associé à *pFile* à la position *pos* relative à *origine*
    - origine: SEEK_SET, SEEK_CUR, SEEK_END
    - retourne une valeur non nulle en cas d'échec, 0 sinon
  - **void rewind (FILE *pFile);**
    - est équivalent à **fseek (pFile, 0L, SEEK_SET);**
  - **long ftell (FILE *pFile);**
    - retourne la position courante associée à *pFile*
      -1 en cas d 'erreur

Cours 5 : Tubes anonymes et nommés

- **Mécanisme de communications du système de fichiers**
  - I-node associé.
  - Type de fichier: S_IFIFO.
  - Accès au travers des primitives ***read*** et ***write***.
- **Les tubes sont unidirectionnels**
  - Une extrémité est accessible en ***lecture*** et l'autre l'est en ***écriture***.
  - Dans le cas des tubes anonymes, si l'une ou l'autre extrémité devient inaccessible, cela est irréversible.

Tubes anonymes et nommés

- **Mode FIFO**
  - Première information écrite sera la première à être consommée en lecture.
- **Communication d'un flot continu de caractères (stream)**
  - Possibilité de réaliser des opérations de lecture dans un tube sans relation avec les opérations d'écriture.
- **Opération de lecture est destructive :**
  - Une information lue est extraite du tube.

Tubes anonymes et nommés

- **Capacité limitée**
  - Notion de tube plein (taille : PIPE_BUF).
  - Écriture éventuellement bloquante.
- **Possibilité de plusieurs lecteurs et écrivains**
  - Nombre de lecteurs :
    - L'absence de lecteur interdit toute écriture sur le tube.
    - Signal SIGPIPE.
  - Nombre d'écrivains :
    - L'absence d'écrivain détermine le comportement du *read:* lorsque le tube est vide, la notion de fin de fichier est considérée.

Les tubes anonymes

- **Primitive pipe**
- **Pas de nom**
  - Impossible pour un processus d'ouvrir un pipe anonyme en utilisant *open*.
- **Acquisition d'un tube:**
  - Création : primitive *pipe*.
  - Héritage : *fork*, *dup*
    - Communication entre père et fils.
    - Un processus qui a perdu un accès à un tube n'a plus aucun moyen d'acquérir de nouveau un tel accès.

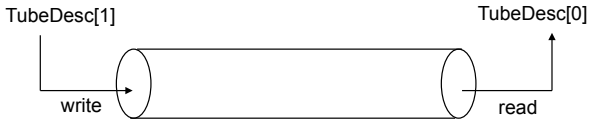
Les tubes anonymes

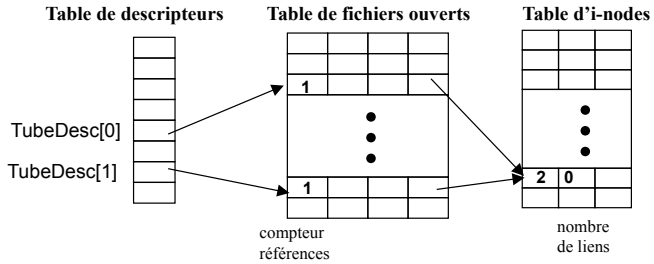
#include <unistd.h>

int pipe (int *TubeDesc);

- en cas de succès: appel renvoie 0
  - TubeDesc[0] : descripteur de lecture
  - TubeDesc[1] : descripteur d'écriture
- en cas d'échec : appel renvoie -1
  - errno = EMFILE (table de descripteurs de processus pleine).
  - errno = ENFILE (table de fichiers ouverts du système pleine).

Les tubes anonymes





Les tubes anonymes

- **Opérations autorisées**
  - ***read***, ***write*** : lecture et écriture
    - Opérations bloquantes par défaut
  - ***close***: fermer des descripteurs qui ne sont pas utilisés
  - ***dup***, ***dup2*** : duplication de description; redirection
  - ***fstat***, ***fcntl***: accès/modification des caractéristiques
- **Opérations non autorisées**
  - *open, stat, access, link, chmod, chown, rename*

Les tubes anonymes (fstat)

- **Accès aux caractéristiques d'un tube**

```
struct stat stat;

int main (int argc, char ** argv) {
  int tubeDesc[2];

 if (pipe (tubeDesc) == -1) {
    perror ("pipe"); exit (1);
}

 if ( fstat (tubeDesc[0],&stat) == -1) {
  perror ("fstat"); exit (2);
 }

if (S_ISFIFO (stat.st_mode)) {
   printf ("il s'agit d'un tube \n");
   printf ("num. inode %d \n", (int)stat.st_ino);
   printf ("nbr. de liens %d \n", (int)stat.st_nlink);
   printf ("Taille : %d \n", (int) stat.st_size);
}

return EXIT_SUCCESS;
}
```

Les tubes anonymes (lecture)

- **Lecture dans un tube d'au plus TAILLE_BUF caractères**
  - **read (tube[0], buff, TAILLE_BUF);**

- **si le tube n'est pas vide** et contient *taille* caractères, lire dans *buff* min (taille, TAILLE_BUF). Ces caractères sont extraits du tube.
- **si le tube est vide**
  - **si le nombre d'écrivains est nul**
    - fin de fichier; *read* renvoie 0
  - **sinon**
    - **si la lecture est bloquante** (par défaut),
      le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide ou qu'il n'y ait plus d'écrivains;
    - **sinon**
      retour immédiat; renvoie -1 et errno = EAGAIN**.**

Les tubes anonymes (écriture)

- **Ecriture de TAILLE_BUFF caractères dans un tube:**
  - **write (tube[1], buff, TAILLE_BUF);**
    - Ecriture sera **atomique** si **TAILLE_BUF** < **PIPE_BUF.**

- **si le nombre de lecteurs dans le tube est nul**
  - signal **SIGPIPE** est délivré à l'écrivain (terminaison du processus par défaut); si SIGPIPE capté, fonction *write* renvoie -1 et **errno = EPIPE**.
- **sinon**
  - **si l'écriture est bloquante**
    - le retour du *write* n'a lieu que lorsque **TAILLE_BUF** caractères ont été écrits**.**
  - **sinon**
    - **si (TAILLE_BUF <= PIPE_BUF)**
      s'il y a au moins TAILLE_BUFF emplacements libres dans le tube
      écriture atomique est réalisée;
      sinon
      renvoie -1, **errno = EAGAIN.**
    - **sinon**
      le retour est un nombre inférieur à **TAILLE_BUF**

Les tubes anonymes (exemple fork)

- **Communication entre processus père et fils**

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define S_BUF 100

int main (int argc, char ** argv) {
      int tubeDesc[2];
      char buffer[S_BUF];
      int n; pid_t pid_fils;

       if (pipe (tubeDesc) == -1) {
         perror ("pipe"); exit (1);
       }
if ( (pid_fils = fork ( )) == -1 ) {
        perror ("fork"); exit (2);
}
```

```
 if (pid_fils == 0) { /*fils */
    if (( n = read (tubeDesc[0],buffer, S_BUF)) == -1) {
        perror ("read"); exit (3);
    }
    else {
        buffer[n] = '\0'; printf ("%s\n", buffer);
    }
      exit (0);
}
else {/*père */
       if ( write (tubeDesc[1],"Bonjour", 7)== -1) {
            perror ("write"); exit (4);
       }
 wait (NULL);
 }
 return (EXIT_SUCCESS); }
```

Affichage fils:
Bonjour

Les tubes anonymes (exemple 2 fork)

- **Communication entre père et fils : blocage**

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define S_BUF 100

int main (int argc, char ** argv) {
      int tubeDesc[2];
      char buffer[S_BUF];
      int n; pid_t pid_fils;

       if (pipe (tubeDesc) == -1) {
         perror ("pipe"); exit (1);
       }
if ( (pid_fils = fork ( )) == -1 ) {
        perror ("fork"); exit (2);
}
```

```
if (pid_fils == 0) { /* fils*/
  for (i=0; i<2; i++)
    if (( n= read (tubeDesc[0], buffer, S_BUF)) == -1){
            perror ("read"); exit (3);
     }
    else { buffer[n] = '\0'; printf ("%s\n",buffer); }
   exit (0);
 }
  else { /* père */
    for (i=0; i<2; i++)
      if ( write (tubeDesc[1],"Bonjour",7) == -1 {
            perror ("write"); exit (4);
      }
    wait (NULL);
 }
return (EXIT_SUCCESS); }
```

Affichage fils:
BonjourBonjour

• Processus père et fils bloqués

Les tubes anonymes (exemple)

- **Ecriture dans un tube sans lecteur** **test-sigpipe.c**

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void sig_handler (int sig) {
 if (sig == SIGPIPE)
   printf ("eciture dans un tube sans lecteurs \n");
}

int main (int argc, char ** argv) {
  int tubeDesc[2]; struct sigaction action;
  action.sa_handler= sig_handler;
  sigaction (SIGPIPE, &action, NULL);

if (pipe (tubeDesc) == - 1) {
  perror ("pipe");
  exit (1);
}
 close (tubeDesc[0]); /* sans lecteur */

 if ( write (tubeDesc[1],"x", 1) == -1)
  perror ("write");

 return EXIT_SUCCESS;
}
```

```
> test-sigpipe
    eciture dans un pipe sans lecteurs
     write: Broken pipe
```


Les tubes anonymes

- ***read* et *write* sont des opérations bloquantes par défaut**
- **fonction *fcntl*:**
  - permet de les rendre non bloquantes

```
int tube[2], attributs;
….. pipe (tube); ….
/* rendre l'écriture non bloquante */
attributs = fcntl (tube[1], F_GETFL);
attributs |=O_NONBLOCK;
fcntl (tube[1], F_SETFL,attributs);
…..
```


Les tubes anonymes (fdopen)

- **Fonctions de haut niveau : *fdopen***
  - Obtenir un pointeur sur un objet du type *FILE*.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main (int argc, char ** argv) {
int tube[2]; char car;

 FILE *ptLire;
  if (pipe (tube) == -1) {
   perror ("pipe"); exit (1);
  }

  if ((ptLire = fdopen (tube[0],"r")) == NULL) {
    perror ("fdopen"); exit (2);
  }

if (write (tube[1],"x",1)!=1) {
   perror ("write"); exit (3);
}

if (fscanf (ptLire, "%c", &car) == -1) {
    perror ("fscanf"); exit (4);
}
 else
   printf ("caractere lu: %c\n", car);
 return (EXIT_SUCCESS);
}
```

```
> test-fdopen
    caractere lu : x
```

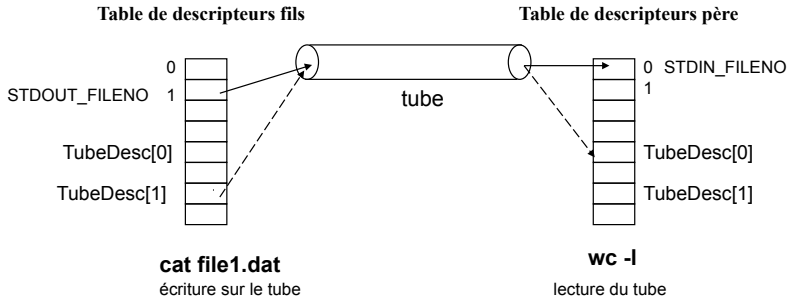

Les tubes anonymes (dup et close)

- ***close***
  - Fermeture des descripteurs qui ne sont pas utilisés.
- ***dup, dup2***
  - Duplication des descripteurs.
  - Rediriger les entrées-sorties standard d'un processus sur un tube.

```
int tube[2], attributs;
 …..
  pipe (tube); ….
  dup2(tube[0], STDIN_FILENO);
  close (tube[0]);
…
```

Les tubes anonymes (exemple de redirection)

- **/* nombre de lignes d'un fichier */**
  - cat file1.dat | wc -l



Les tubes anonymes (exemple de redirection)

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char ** argv) {
int tubeDesc[2]; pid_t pid_fils;

 if (pipe (tubeDesc) == -1) {
   perror ("pipe");
   exit (1);
}

 if ( (pid_fils = fork ( )) == -1 ){
   perror ("fork");
   exit (2);
}
```

```
if (pid_fils == 0) { /* fils */
    dup2(tubeDesc[1],STDOUT_FILENO);
    close (tubeDesc[1]); close (tubeDesc[0]);
    if (execl ("/bin/cat", "cat", "file1.dat",NULL) == -1) {
        perror ("execl"); exit (3);
    }
 }
 else { /* père */
        dup2(tubeDesc[0],STDIN_FILENO);
        close (tubeDesc[0]);
        close (tubeDesc[1]);
       if (execl ("/bin/wc", "wc", "-l", NULL) == -1) {
          perror ("execl"); exit (3);
       }
 }
 return (EXIT_SUCCESS);
}
```


Tubes nommés

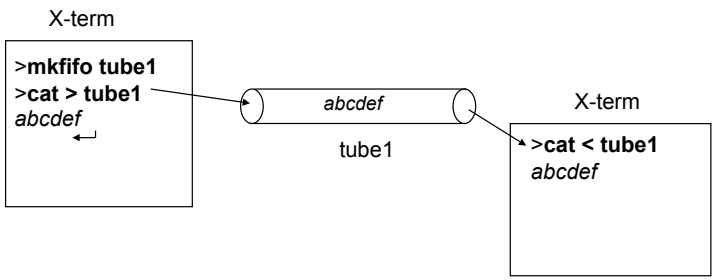
- **Permettent à des processus sans lien de parenté de communiquer en mode flot (stream).**
  - Toutes les caractéristiques des tubes anonymes.
  - Sont référencés dans le système de gestion de fichiers.
  - Utilisation de la fonction *open* pour obtenir un descripteur en lecture ou écriture.
  - ls -l

    prw-rw-r-- 1 arantes src 0 Nov 9 2004 tube1

Tubes nommés (mkfifo)

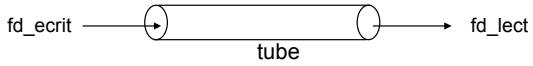
- **Création d'un tube nommé**
  - **mkfifo [-p] [-m mode]** ***référence***
    - -m mode : droits d'accès (les mêmes qu'avec chmod).
    - -p : création automatique de tous les répertoires intermédiaires dans le chemin *référence*.
- **Fonction**
  - **int mkfifo (const char *ref, mode_t droits);**
    - *ref* définit le chemin d'accès au tube nommé et *droits* spécifie les droits d'accès.
    - Renvoie 0 en cas de succès; -1 en cas d'erreur.
      - errno = EEXIST, si fichier déjà créé.

Tubes nommés (mkfifo)



Tubes nommés (open)

- **Par défaut bloquante (rendez-vous):**
  - Une demande d'ouverture en lecture est bloquante s'il n'y a aucun écrivain sur le tube.
  - Une demande d'ouverture en écriture est bloquante s'il n'y a aucun lecteur sur le tube.




```
int fd_lect, fd_ecrit;
fd_lect = open ("tube", O_RDONLY);
fd_ecrit = open ("tube", O_WRONLY);
```


Tubes nommés (open)

- **Ouverture non bloquante**
  - Option O_NONBLOCK lors de l'appel à la fonction *open*
    - **Ouverture en lecture** :
      - Réussit même s'il n'y a aucun écrivain dans le tube.
      - Opérations de lectures qui se suivent sont non bloquantes.
    - **Ouverture en écriture** :
      - Sur un tube sans lecteur, l'ouverture échoue: valeur -1 renvoyée.
      - Si le tube possède des lecteurs, l'ouverture réussit et les écritures dans les tubes sont non bloquantes.

Tube nommé (suppression du nœud)

- **Un nœud est supprimé quand:**
  - Le nombre de liens physiques est nul.
    - Fonction *unlik* ou commande *rm.*
  - Le nombre de liens internes est nul.
    - Nombres de lecteurs et écrivains sont nuls.
- **Si nombre de liens physiques est nul, mais le nombre de lecteurs et/ou écrivains est non nul**
  - Tube nommé devient un tube anonyme.

Tubes nommés (écrivain)

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

#define S_BUF 100
int n ;
char buffer[S_BUF];
int main (int argc, char ** argv) {
  int fd_write;

   if ( mkfifo(argv[1],
      S_IRUSR|S_IWUSR) == -1) {
        perror ("mkfifo");
        exit (1);
   }
```

```
if (( fd_write = open (argv[1],
     O_WRONLY)) == -1) {
       perror ("open");
       exit (2);
}

if (( n= write(fd_write,"Bonjour", 7)) == -1) {
   perror ("write");
   exit (3);
}

close (fd_write);
return EXIT_SUCCESS;

}
```


Tubes nommés (lecteur)

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#define S_BUF 100
int n ; char buffer[S_BUF];

int main (int argc, char ** argv) {
int fd_read;
   if (( fd_read = open (argv[1],
        O_RDONLY)) == -1) {
        perror ("open"); exit (1)
}
```

```
if (( n= read (fd_read, buffer, S_BUF)) == -1){
            perror ("read");
            exit (2);
}
else {
        buffer[n] = '\0';
        printf ("%s\n",buffer);
}
close (fd_read);
return EXIT_SUCCESS;
}
```


Tubes nommés: interblocage (ouverture bloquante)

PROCESSUS 1 :

```
int main (int argc, char ** argv) {
  int fd_write, fd_read;

 if ( (mkfifo("tube1",S_IRUSR|S_IWUSR) == -1) ||
 (mkfifo("tube2",S_IRUSR|S_IWUSR) == -1)) {
       perror ("mkfifo"); exit (1);
 }

 if (( fd_write = open ("tube1", O_WRONLY)) == -1) {
       perror ("open"); exit (2);
 }

 if (( fd_read = open ("tube2", O_RDONLY)) == -1) {
       perror ("open"); exit (3);
 }
.....
 return EXIT_SUCCESS;
}
```

PROCESSUS 2 :

```
int main (int argc, char ** argv) {
   int fd_write, fd_read;

   if (( fd_write = open ("tube2", O_WRONLY))
       == -1) {
             perror ("open"); exit (2);
    }

    if (( fd_read = open ("tube1", O_RDONLY))
      == -1) {
             perror ("open"); exit (3);
    }
   .....
   return EXIT_SUCCESS;
}
```

Licence Informatique **L3**
UE LI356

Cours 6

COMMUNICATIONS INTER-PROCESSUS

1. Files de messages
2. Segments de mémoire partagée
3. Sémaphores

Olivier Marin

olivier.marin@lip6.fr

IPC : Outils et principes

Inter-Process Communication (IPC)

Modèle processus : moyen d'isoler les exécutions
- Distinction des ressources, états, ...
- Canaux de communication basiques : `wait`, `kill`, ...

Pb : Nécessité de communication/synchronisation étroite **entre pcs** ≠
- Canaux basiques pas toujours suffisants
- Solutions par fichiers (*eg. tubes*) peu efficaces et pas forcément adaptées (*eg. priorités*)

Trois mécanismes de comm/synchro entre pcs **locaux** via la mémoire
- Les files de messages
- La mémoire partagée
- Les sémaphores

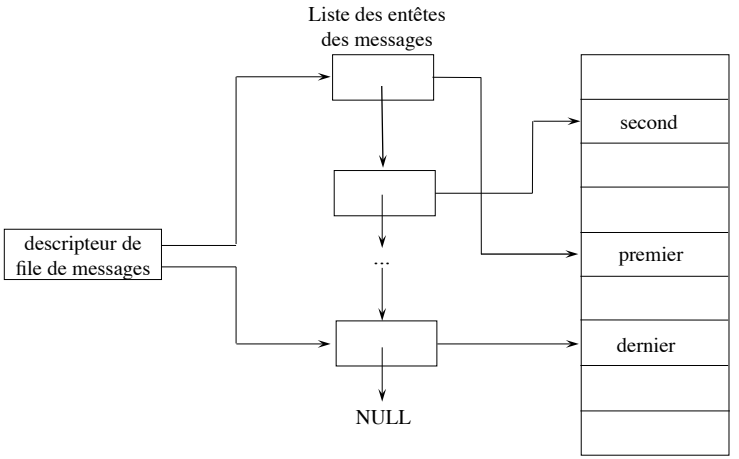
Effets des appels système

Appel à :

- `fork()`
  - Héritage de tous les objets IPC par le fils
- `exec()` ou `exit()`
  - Tous les accès à des objets IPC sont perdus

    ATTENTION : les objets ne sont pas détruits
  - Dans le cas de la mémoire partagée, le segment est détaché

Les files de messages



Les files de messages

Principe

Liste chaînée de messages
- Conservée en mémoire
- Accessible par +sieurs pcs

Message

Structure complexe définie par l'utilisateur

Doit comporter un indicateur du type (~ *priorité*) de message

Fonctionnement

accès FIFO (par défaut) + accès par type

limites : nb de msgs (`MSGMAX`), taille totale en nb de bytes (`MSBMNB`)

limite atteinte ⇒ écriture bloquante (par défaut)

file vide ⇒ lecture bloquante (par défaut)

Les files de messages

Avantages

Amélioration par rapport au concept de tube
- organisée en messages
- contrôle sur l'organisation (priorités)

Simplicité

Proche du fonctionnement naturel d'une application

Désavantages

Reste basée sur du FIFO

Impossible d'organiser les accès différemment (*eg. pile)*

Pas d'accès concurrents à une même donnée

Performances limitées

2 recopies **complètes** par msg : expéditeur → cache système → destinataire

Files de messages POSIX

Fichier <mqueue.h>

Accès

mq_open ⇒ créer / ouvrir une file en mémoire

mq_close ⇒ fermer l'accès à une file

mq_unlink ⇒ détruire une file

mq_getattr ⇒ obtenir les attributs de la file (taille, mode d'accès, ...)

mq_setattr ⇒ modifier le mode d'accès (O_NONBLOCK)

Opérations sur une file

mq_send ⇒ déposer un message

mq_receive ⇒ retirer un message

mq_notify ⇒ demander à être prévenu de l'arrivée d'un message

Attributs d'une file de messages

```
struct mq_attr {
  [...]
  long mq_maxmsg;  //max nb of msgs in queue
  long mq_msgsize; //max size of a single msg
  long mq_flags;   //behaviour of the queue
  long mq_curmsgs; //nb of msgs currently in queue
  [...]
};
```

`mq_flags = O_NONBLOCK` ou `0`

Ouverture d'une file de messages

```
#include<mqueue.h>
mqd_t mq_open(char* name, int flags, struct mq_attr* attrs);
```

- Crée une nouvelle file ou recherche le descr. d'une file déjà existante
- Retourne un descripteur (castable en int) positif en cas de succès, -1 sinon
- *flags* idem open
- *attrs* peut être rempli avant pour définir des valeurs (sauf `mq_flags`)
  `mq_flags` modifiable avec

```
mq_getattr(mqd_t mq_descr, struct mq_attr* attrs);
mq_setattr(mqd_t mq_descr, struct mq_attr* new_attrs
           struct mq_attr* old_attrs);
```


Fermeture d'une file de messages

```
int mq_close(mqd_t mqdescr);
```

- retourne 0 en cas de succès, -1 sinon.
- Automatiquement appelé lors de la terminaison du pcs
- Pas d'effet sur l'existence ou le contenu de la file

```
int mq_unlink(char* mqname);
```

- retourne 0 en cas de succès, -1 sinon.
- Détruit la file associée à *mqname* ainsi que son contenu
  Après l'appel, plus aucun pcs ne peut ouvrir la file
  Destruction effective une fois que ts les pcs qui ont accès ont appelé mq_close

Ajout de message

```
int mq_send(mqd_t mqdescr, const char* msg_data,
            size_t msg_length, unsigned int priority);
```

- retourne 0 en cas de succès, -1 sinon.
- *msg_data* le contenu du message
- *msg_length* la taille du message
- *priority* sa priorité, 0 ≤ priority ≤ MQ_PRIOMAX (≥ 32, déf. dans *limits.h*)
  Si priority > MQ_PRIOMAX, l'appel échoue
- File ordonnée par priorités, en FIFO pour les msgs de même prio
- Appel bloquant si la file est **pleine** et O_NONBLOCK non spécifié

Retrait de message

```
int mq_receive(mqd_t mqdescr, const char* msg_data,
               size_t msg_length, unsigned int *priority);
```

- retourne le nb de bytes lus en cas de succès, -1 sinon.
- *msg_data* le contenu du message
- *msg_length* la taille du message
  Si msg_length > mq_attr.mqmsgsize, l'appel échoue
- *priority* sa priorité
- Appel bloquant si la file est **vide** et O_NONBLOCK non spécifié

Notification d'arrivée de message

```
int mq_notify(mqd_t mqdescr, const struct sigevent* notification);
```

- retourne 0 en cas de succès, -1 sinon
- Appel non bloquant
- Un seul pcs par file peut demander à être notifié
- Notification si aucun pcs n'est bloqué en attente de msg
- Après notification, désenregistrement de la demande
  - ⇒ pr être notifié à chq arrivée de msg, il faut refaire un appel après chq notification

```
struct sigevent { [...]
   int sigev_notify //Notification type
                        (SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD)
   int sigev_signo  //Signal number
   union sigval sigev_value                          //Notif. data
   void (*)(union sigval) sigev_notify_function    //Notif. function
[...] };
```


Segments de mémoire partagée

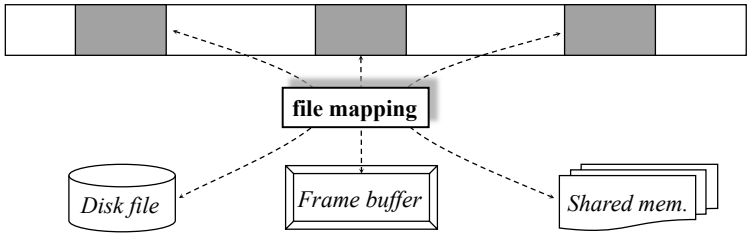
Principe

Zone mémoire attachée à un pcs mais accessible pour d'autres pcs

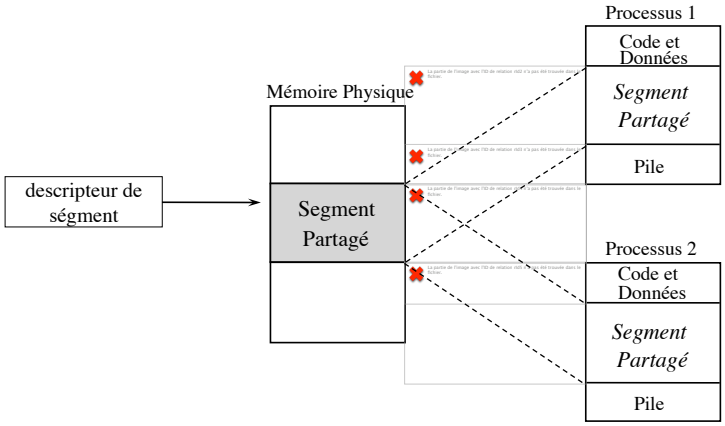
Liée à un autre service : **file mapping**

Etablissement d'une correspondance (attachement) entre :

- Un fichier (ou un segment de mémoire)
- Une partie de l'espace d'adressage d'un processus réservée à cet effet



Segments de mémoire partagée



Segments de mémoire partagée

Avantages

Accès totalement libre

Chq pcs détermine à quelle partie de la structure de données il accède

Efficacité

Pas de recopie mémoire : ts les pcs accèdent **directement** au même segment

Désavantages

Accès totalement libre

Pas de synchro implicite comme pr les tubes et les files de msgs

⇒ Synchro doit être explicitée (sémaphores ou signaux)

Pas de gestion de l'adressage

Validité d'un pointeur limitée à son esp. d'adressage

⇒ Impossible de partager des ptrs entre pcs

Mémoire partagée POSIX

Fichier <sys/mman.h>

Fonctions contenues dans la bibliothèque librt (real-time)

```
$ gcc -Wall -o monprog monprog.c -lrt
```

Accès

- shm_open ⇒ créer / ouvrir un segment en mémoire
- close ⇒ fermer un segment
- mmap ⇒ attacher un segment dans l'espace du processus
- munmap ⇒ détacher un segment de l'espace du processus
- shm_unlink ⇒ détruire un segment

Opérations sur un segment

- mprotect ⇒ changer le mode de protection d'un segment
- ftruncate ⇒ allouer une taille à un segment
- msync ⇒ mettre à jour la sauvegarde stable associée au segment

Mémoire partagée POSIX

Savoir si un système implémente la mémoire partagée POSIX

Fichier <unistd.h>

mmap	_POSIX_MAPPED_FILES ou _POSIX_SHARED_MEMORY_OBJECTS
munmap	_POSIX_MAPPED_FILES ou _POSIX_SHARED_MEMORY_OBJECTS
shm_open	_POSIX_SHARED_MEMORY_OBJECTS
shm_unlink	_POSIX_SHARED_MEMORY_OBJECTS
ftruncate	_POSIX_MAPPED_FILES ou _POSIX_SHARED_MEMORY_OBJECTS
mprotect	_POSIX_MEMORY_PROTECTION
msync	_POSIX_MAPPED_FILES et _POSIX_SYNCHRONIZED_IO

Ouverture / Destruction d'un segment de mémoire partagée

```
#include<sys/mman.h>
int shm_open(const char *name, int flags, mode_t mode);
```

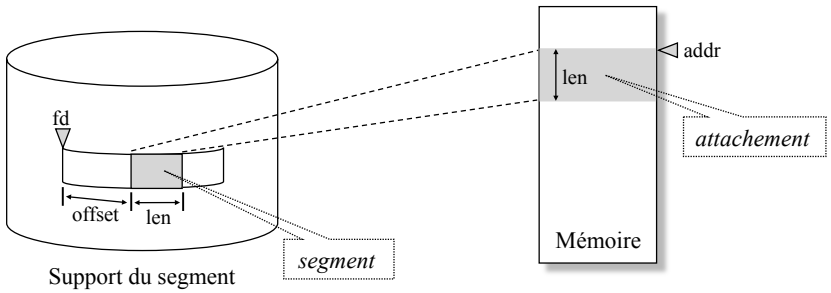
- Crée un nouveau segment de taille 0
  ou recherche le descr. d'un segment déjà existant
- Retourne un descripteur positif en cas de succès, -1 sinon
- *flags* idem open
- *mode* idem chmod

```
int shm_unlink(const char *name);
```

Idem mq_unlink

Attachement d'un segment de mémoire partagée

```
#include<sys/mman.h>
#include<sys/types.h>
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```



Attachement / Détachement d'un segment de mémoire partagée

```
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```

- Retourne NULL en cas d'échec,
  l'@ d'un attachement de taille *len* à partir *d'offset* ds le segment de descr *fd* sinon
- *addr* addresse où attacher le segment en mémoire ; 0 ⇒ choix du système
- *prot* protection associée (PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE)
- *flags* mode de partage
  MAP_SHARED : modifs visibles par tous les pcs ayant accès (partage)
  MAP_PRIVATE : modifs visibles par le pcs appelant uniquement(shadow copy)
  MAP_FIXED : force l'utilisation d'*addr*

```
int munmap(caddr_t addr, size_t len);
```

- Détruit l'attachement de taille *len* à l'addresse *addr*
- Retourne -1 en cas d'échec, 0 sinon

Opérations sur un segment de mémoire partagée

```
void * mprotect(caddr_t addr, size_t len, int prot);
```

- Modifie la protection associée au segment
  PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE (grave utile !)
- Retourne -1 en cas d'échec, 0 sinon

```
int ftruncate(int fd, off_t length);
```

- Définit la taille du segment de descr. *fd*
  nouvelle taille = length
  si (ancienne taille > nouvelle taille), alors les données en excédent sont perdues
- Retourne -1 en cas d'échec, 0 sinon

```
int msync(void *addr, size_t len, int flags);
```

- Met à jour le segment associé à l'attachement d'addresse *addr* et de taille *len*
- *flags* MS_SYNC, MS_ASYNC
- Retourne -1 en cas d'échec, 0 sinon

Exemple

```
int *sp;
int main() {
        int fd;
        /* Creer le segment monshm, ouverture en R/W */
        if ((fd = shm_open("monshm", O_RDWR | O_CREAT, 0600)) == -1) {
                perror("shm_open");
                exit(1);}

        /* Allouer au segment une taille pour stocker un entier */
        if (ftruncate(fd, sizeof(int)) == -1) {
                perror("ftruncate");
                exit(1);}

        /* “mapper” le segment en R/W partagé */
        if ((sp = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0))== MAP_FAILED) {
                perror("mmap");
                exit(1);}

        /* Accès au segment */
        *sp = 10;

        /* “detacher” le segment */
        munmap(sp, sizeof(int));

        /* detruire le segment */
        shm_unlink(“monshm”);
        return 0;
}
```


Les sémaphores

Principe (Dijkstra)

Mécanisme de synchronisation

- accès concurrents à une ressource partagée (*eg. segment de mémoire*)
- solution au problème de l’exclusion mutuelle

Structure sémaphore

- un compteur : nb d'accès disponibles avant blocage
- une file d’attente : pcs bloqués en attente d'un accès

Les sémaphores

Fonctionnement

- Demande d'accès (P - *proberen* ou *"puis-je ?"*)
  - Décrémentation du compteur
  - Si compteur < 0, alors blocage du pcs et insertion ds la file
- Fin d'accès (V - *verhogen* ou *"vas-y"*)
  - Incrémentation du compteur
  - Si compteur ≤ 0, alors déblocage d'un pcs de la file

Blocage, déblocage et insertion des pcs ds la file sont des ops implicites

Les sémaphores

Dysfonctionnements

Liés à leur utilisation franchement pas intuitive

Interblocage

2 pcs P et Q sont bloqués en attente
P attend que Q signale sa fin d'accès et Q attend que P signale sa fin d'accès

Famine

Un pcs est bloqué en attente d'une fin d'accès qui n'arrivera jamais

Sémaphores POSIX

2 types de sémaphores :

- Sémaphores nommés
  - Portée : tous les processus de la machine
  - Primitives de base : **sem_open, sem_close**, **sem_unlink**, sem_post, sem_wait
- Sémaphores anonymes (*memory-based*)
  - Portée : processus avec filiation, uniquement threads dans linux
  - Primitives de base : **sem_init, sem_destroy**, sem_post, sem_wait

Inclus dans la bibliothèque des pthreads

```
$  gcc -Wall -o monprog monprog.c -lpthread
```


Création de sémaphore nommé

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, mode_t mode, int value);
```

- Crée ou ouvre le sémaphore de nom *name*
- *oflag, mode* idem open
- *value* valeur initiale du compteur

Retourne un pointeur sur le sémaphore, NULL en cas d'erreur

Ex : Creation d'un sémaphore initialisé à 10

```
sem_t *s;
s = sem_open(« monsem », O_CREAT | O_RDWR, 0600, 10);
```

Création de sémaphore anonyme

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

Crée et initialise le sémaphore *sem*

- *pshared* != 0
  ⇒ partage entre processus de la même famille (threads sous linux !)
- *value* valeur initiale du sémaphore

Retourne -1 en cas d'erreur, 0 sinon

Ex : création de sémaphore partagé, initialisé à 10

```
sem_t s;
sem_init(&s, 1, 10);
```


Opérations sur sémaphore

« P » : `int sem_wait(sem_t *sem);`

Il existe un « P » non bloquant...

`int sem_trywait(sem_t *sem);`

« V » : `int sem_post(sem_t *sem);`

Retournent -1 en cas d'erreur, 0 sinon

Fermeture / Destruction

Sémaphore nommé :

1) Fermer le sémaphore

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

2) Détruire le sémaphore

```
int sem_unlink(const char *name);
```

Sémaphore anonyme :

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


Exemple : sémaphores nommés

```
...
int main() {

        sem_t *smutex;

      /* creation d'un semaphore mutex initialisé à 1 */
      if ((smutex = sem_open("monsem", O_CREAT  |O_EXCL | O_RDWR , 0666, 1)) == SEM_FAILED) {
         if (errno != EEXIST) {
               perror("sem_open"); exit(1);
         }
          /* Semaphore deja créé, ouvrir sans O_CREAT */
         smutex = sem_open(("monsem", O_RDWR);
       }


        /* P sur smutex */
      sem_wait(smutex);

      /* V sur smutex */
      sem_post(smutex);

      /* Fermer le semaphore */
       sem_close(smutex);

      /* Detruire le semaphore */
      sem_unlink("monsem");
      return 0;
}
```

Licence Informatique **L3**
UE LI356

Cours 7
GESTION DU TEMPS

1. Principes
2. Horloges
3. Temporisateurs

Olivier Marin

olivier.marin@lip6.fr

Système Temps Réel : Principes

Définition d'un système Temps Réel (TR)

Système dont le résultat dépend à la fois :
- de l'exactitude des calculs
- et du temps mis à produire les résultats

Notion d'échéance

Contrainte de temps bornant l'occurence d'un événement (production de résultat, envoi de signal, ...)

"*Introduction aux systèmes temps réel*", C. Bonnet & I. Demeure, Ed. Hermès/Lavoisier 1999

Problématiques Temps Réel

• Garantir qu'un événement se produit à une date donnée, ou avant une échéance donnée

➢ Contraintes périodiques : le service doit être rendu selon un certain rythme
eg. toutes les X ms
➢ Contraintes ponctuelles : lorsque l'évt Y se produit, il doit être traité dans un tps limité

• Garantir qu'un évt A se produit avant un évt B

• Garantir qu'aucun évt externe à l'application TR ne retardera les pcs importants

• Garantir qu'un ordonnancement entre plusieurs tâches est effectivement possible

Causes de ces problématiques
- E/S hardware & E/S utilisateur
- Journalisation de données
- Exécution de tâches de fond

POSIX Temps Réel

POSIX.4 : POSIX Real-Time Scheduling Interfaces

- Gestion dynamique de l'ordonnancement
- Horloges et temporisateurs évolués
- Ajout d'E/S asynchrones
- Signaux TR

Mais pas tout à fait finalisé
Par exemple, pas de gestion explicite des échéances

Gestion du temps

Mesure du temps
- La période minimale dépend de la puissance (cadence) du processeur
- Décomposition du temps en ticks horloge
  - Constante HZ dans <sys/param.h>
  - Période du tick = 1 000 000 / HZ

Horloges
- Font partie intégrante d'UNIX depuis le début
  - 3 horloges, dont une principale : horloge globale (`ITIMER_REAL`)
  - 2 fonctions principales : `time` & `gettimeofday`
- Le monde selon UNIX est né le 1er janvier 1970 à 00:00am (*Epoch*)

Temporisateurs
- 2 types : ponctuel, périodique
- UNIX possède un timer ponctuel de base : la fonction `sleep`

Gestion du temps

Fonctionnalités POSIX manquantes dans UNIX

- Définir un nombre d'horloges supérieur à 3
- Établir une mesure de temps inférieure à la microseconde
  - La majorité des processeurs actuels peut compter en nanosecondes
- Déterminer le nombre de débordements d'un temporisateur
  - ie. le temps écoulé depuis la dernière échéance **traitée**
- Choisir le signal indiquant l'expiration du temporisateur
  - UNIX par défaut : `SIGALRM`

Horloges POSIX

Fonctionnement

- Autant d'horloges que définies dans <time.h>
  - Nombre d'horloges et leur précision dépend de l'implémentation
  - Un identifiant par horloge (type `clockid_t`)
  - Une horloge POSIX **doit** être fournie par l'implém : `CLOCK_REALTIME`
- Structure de comptabilisation du temps à granularité en nanosecondes

```
struct timespec {
     time_t tv_sec;  /* secondes dans l'intervalle */
     time_t tv_nsec; /* NANOsecondes dans l'intervalle */
};
```

- Fonctions d'utilisation

```
include <time.h>
int clock_settime(clockid_t, const struct timespec *);
int clock_gettime(clockid_t, struct timespec *);
int clock_getres(clockid_t, struct timespec *);
```


Horloges POSIX

Fonctions d'utilisation

- Renvoient -1 en cas d'échec, 0 sinon
- Récupération de la précision (*eng : resolution*)

```
int clock_getres(clockid_t cid, struct timespec *res);
```

  - *cid* identifiant de l'horloge dont on cherche la précision
  - *res* résultat : précision de l'horloge
- Récupération de l'heure courante

```
int clock_gettime(clockid_t cid, struct timespec *cur_time);
```

  - *cid* identifiant de l'horloge dont on veut obtenir l'heure
  - *cur_time* résultat : heure courante
- Changement de l'heure courante

```
int clock_settime(clockid_t cid, struct timespec *new_time);
```

  - *cid* identifiant de l'horloge dont on veut changer l'heure
  - *new_time* nouvelle heure courante après mise à jour

Exemple

```
#include <time.h>

…

struct timespec what_time_is_it;

if (clock_gettime(CLOCK_REALTIME, &what_time_is_it) == -1) {
    perror("clock_gettime");
    exit(1);
}
printf("temps écoulé depuis Epoch : %d nanosecondes",
         what_time_is_it.tv_sec*1E9 + what_time_is_it.tv_nsec);

…
```


Temporisateurs POSIX

Fonctionnement

POSIX autorise N temporisateurs par processus
minimum 32, maximum `TIMER_MAX` (<`limits.h`>)

Chaque temporisateur est un élément distinct dans le système
- identifiant unique
- événement spécifique déclenché à l'échéance
- ressources associées (à libérer après utilisation, donc...)

Structure de description de temporisateur

```
struct itimerspec {
     struct timespec it_value;    /* première échéance */
     struct timespec it_interval; /* échéances suivantes */
};
```

`it_interval` équivalent à 0 nanosecondes => temporisateur ponctuel

Temporisateurs POSIX

Attente contrôlée

```
int clock_nanosleep(clockid_t cid, int flags,
        const struct timespec *rqtp, struct timespec *rmtp);
```

- *cid* identifiant de l'horloge régissant le temps
- *flags* mode de temporisation
  - `TIMER_ABSTIME` Temps absolu (ie. date précise, construite avec `mktime`)
  - 0 Temps relatif (ie. à partir de l'appel)
- *rqtp* échéance du réveil
- *rmtp* temps restant jusqu'à l'échéance si un signal a interrompu le sommeil

Renvoie 0 si le temps requis est écoulé, un code d'erreur sinon

Temporisateurs POSIX

Création & destruction

Création de temporisateur

```
int timer_create(clockid_t cid, struct sigevent *evp, timer_t *tid);
```

- *cid* identifiant de l'horloge régissant le temporisateur
- *evp* événement déclenché lorsque le temporisateur arrive à échéance
- *tid* identifiant du temporisateur créé

Destruction de temporisateur (implicite à la terminaison du processus propriétaire)

```
int timer_delete(timer_t tid);
```

- *tid* identifiant du temporisateur à détruire

Renvoient -1 en cas d'échec, 0 sinon

Temporisateurs POSIX

Manipulation

Renvoient -1 en cas d'échec, 0 sinon

Armement de temporisateur

```
int timer_settime(timer_t timerid, int flags,
        const struct itimerspec *value, struct itimerspec *ovalue);
```

- *timerid* identifiant du temporisateur à armer
- *flags* mode de temporisation
  - `TIMER_ABSTIME` Temps absolu (ie. date précise, construite avec `mktime`)
  - `0` Temps relatif (ie. à partir de l'armement)
- *value* échéances (première et suivantes) du temporisateur
- *ovalue* temps restant jusqu'à la prochaine échéance **courante** (ie. avant réarmement)

Renvoie -1 en cas d'échec, 0 sinon

Temporisateurs POSIX

Manipulation (suite)

Consultation de temporisateur

```
int timer_gettime(timer_t timerid, struct itimerspec *t_remaining);
```

- *timerid* identifiant du temporisateur consulté
- *t_remaining* temps restant jusqu'à la prochaine échéance

Renvoie -1 en cas d'échec, 0 sinon

Détermination du débordement

```
int timer_getoverrun(timer_t timerid);
```

- *timerid* identifiant du temporisateur consulté

Renvoie le nombre de déclenchements non traités, -1 sinon

A chaque traitement d'événement déclenché par une échéance, ce nombre est remis à 0

Permet de pallier l'impossibilité de comptabiliser le nombre de signaux reçus

Exemple

```
#include <time.h>
#include <signal.h>

…

timer_t tmr_expl;
struct sigevent signal_spec;
signal_spec.sigev_signo = SIGRTMIN; /* signal utilisateur POSIX.4 */
timer_create(CLOCK_REALTIME, &signal_spec, &tmr_expl);

struct itimerspec new_tmr, old_tmr;
new_tmr.it_value.tv_sec = 1;
new_tmr.it_value.tv_nsec = 0;
new_tmr.it_interval.tv_sec = 0;      /* temporisateur ponctuel */
new_tmr.it_interval.tv_nsec = 0;     /* temporisateur ponctuel */
timer_settime(tmr_expl, 0, &new_tmr, &old_tmr);

pause();

…
```


Conclusion

POSIX.4 = Outils de construction de systèmes temps réel

Gestion du temps

Il existe beaucoup d'autres éléments POSIX.4

Ordonnancement

Verrouillage mémoire : `mlock`, `mlockall`, `munlock`

E/S asynchrones

Signaux temps réel

- Files d'attentes de signaux
- Signaux avec priorité
- Passage de paramètres plus significatifs au handler (en tout cas + d'info qu'un entier)
- N signaux utilisateur (de `RTMIN` à `RTMAX`)