4I400
Programmation Répartie

MI014 - POSIX

- **Responsable : Emmanuel Saint-James**
- **Cours : Luciana Arantes**
- **TD et TME : E. Saint-James,L.Arantes, E. Dubois, et G. Turchini**
- **Evaluation:**
  - Partiel (30%), TME(10%) Examen (60%).

MI014 - POSIX

- **Cours 1 : Rappels**
  - Introduction, processus et signaux
- **Cours 2 & 3: Processus légers**
- **Cours 4 & 5: IPC POSIX, IPC Système 5 et Gestion de la mémoire**
- **Cours 6 : Gestion de fichiers**
- **Cours 7 & 8 : Communications distantes**
  - Sockets UDP et TCP
- **Cours 9 & 10 : Temps Réel**

Bibliographie

- **Le langage C (C Ansi)**
  Brian Kernighan, Dennis Ritchie, *Masson Prentice Hall*, 1991, 280 p.
- **Unix : Programmation et communication**
  J.-M. Rifflet, J.-B. Yunes, *Dunod,* 2003, 768 p.
- **Méthodologie de la programmation en C, Bibliothèque standard, API POSIX**
  J.-P. Braquelaire, *Dunod, 3e éd.*, Paris 2000, 556 p.
- **Programmation système en C sous Linux**
  C. Blaess, *Eyrolles, 2e éd.*, 2005, 964p.
- **Advanced programming in the Unix environment**
  W. Richard Stevens, *Addison-Wesley*, 1992, 768p.

Cours 1 – Rappels

- **La Norme POSIX**
- **Compilation / Makefile**
- **Processus**
- **Signaux**

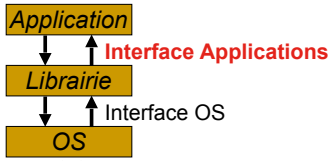
1. La Norme POSIX

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**

1. La Norme POSIX

- **POSIX : En fait, un ensemble de standards (IEEE 1003.x)**
  - Chaque standard se spécialise dans un domain
    - *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)*

1. 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>

IEEE Std 1003.1, 2004 Edition Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved.

2. Compilation/Makefile : 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"*

2. Compilation/Makefile

- **Fichier *Makefile* (ou *makefile*)**
  - Constitué de plusieurs règles de la forme

    <cible>: <liste_prérequis>

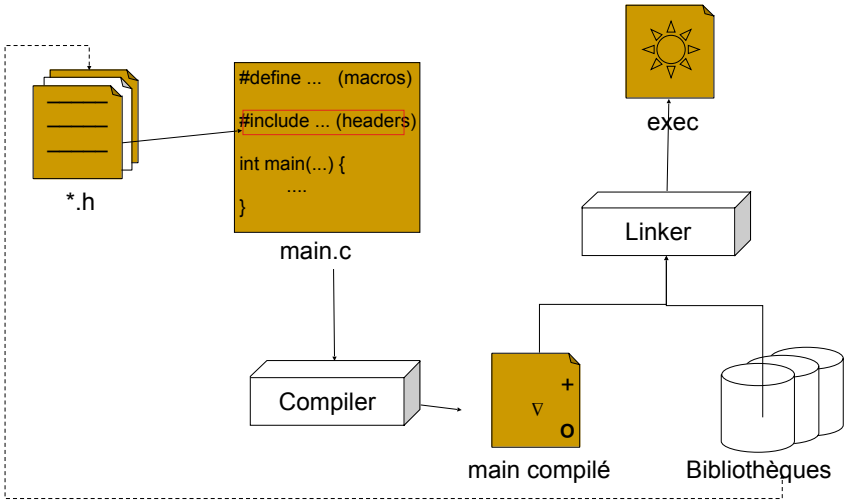
    <commandes>

    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

2. Compilation/Makefile



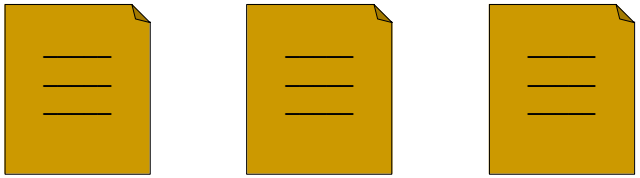
2. Compilation/Makefile

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`)
-o (Output)	Rediriger l'output dans un fichier

2. Compilation/Makefile

Exemple : projet "HelloWorld"



2. Compilation/Makefile

Exemple : fichier 'hello.c'

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

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


2. Compilation/Makefile

Exemple : fichier 'hello.h'

```
#ifndef H_GL_HELLO
#define H_GL_HELLO

void Hello(void);

#endif
```

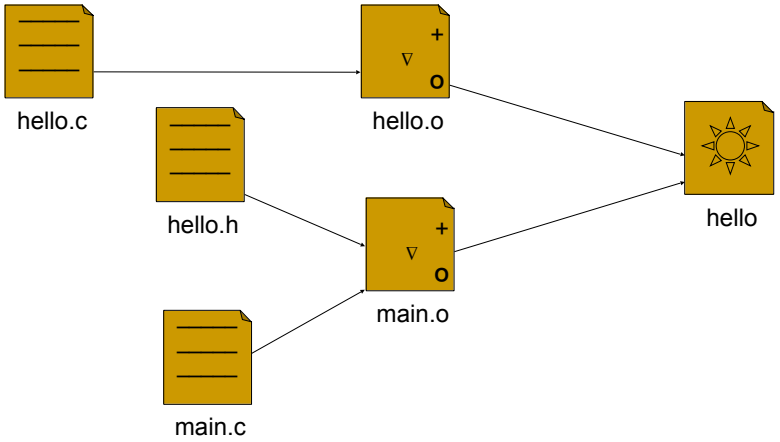

2. Compilation/Makefile

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;
}
```

2. Compilation/Makefile

Exemple : projet "HelloWorld" - dépendances



2. Compilation/Makefile

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
```


2. Compilation/Makefile

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)
```


2. Compilation/Makefile

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

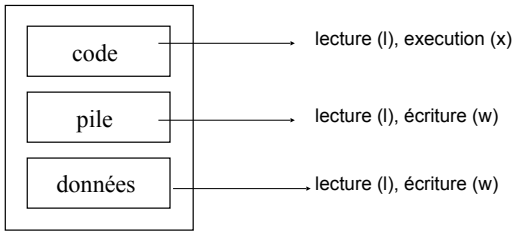
```
CC=gcc
CFLAGS=-Wall -ansi
EXEC=hello

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*

3. 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

3. Processus

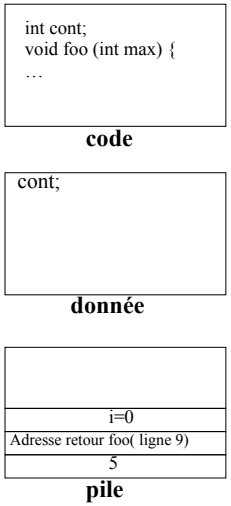


Processus

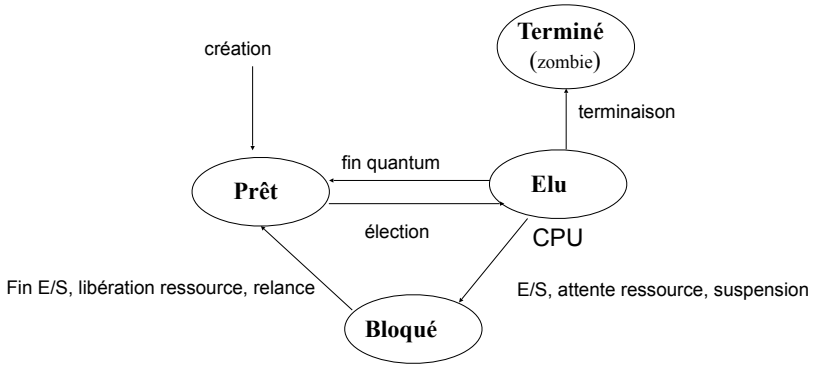
- **Chaque processus est indépendant**
  - Deux processus peuvent être associés au même programme (code)
  - Synchronisation entre processus (communication)
- **CPU est partagé (temps partagé)**
  - Commutation entre les processus

3. 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: }
```



3. Processus: Etats d'un processus



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

3. Processus: 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;
}
```


3. ProcessusAttributs 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*

3. Processus : 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)
```

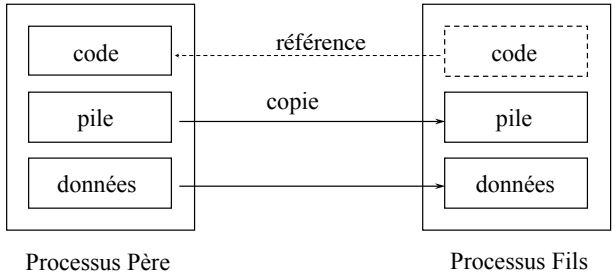
  - Processus fils créé est une copie du processus père

3. Processus: 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

3. Processus: fork

- 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.



3. Processus: Fork - exemple

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 []) {
   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;
       }
```

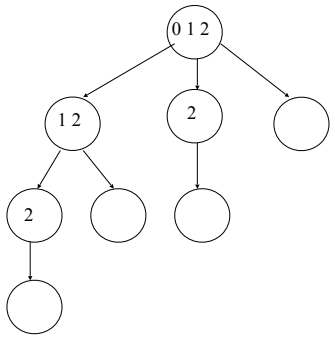

3. Processus: Fork exemple

Combien de processus sont-ils créés?

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 i =0 ;
   while (i <3) {
     printf ( "%d ", i);
     i ++;
     if (fork ( ) == -1)
        exit (1);
   }
   printf( "\n ");
   return EXIT_SUCCESS;
}
```

Arborescence



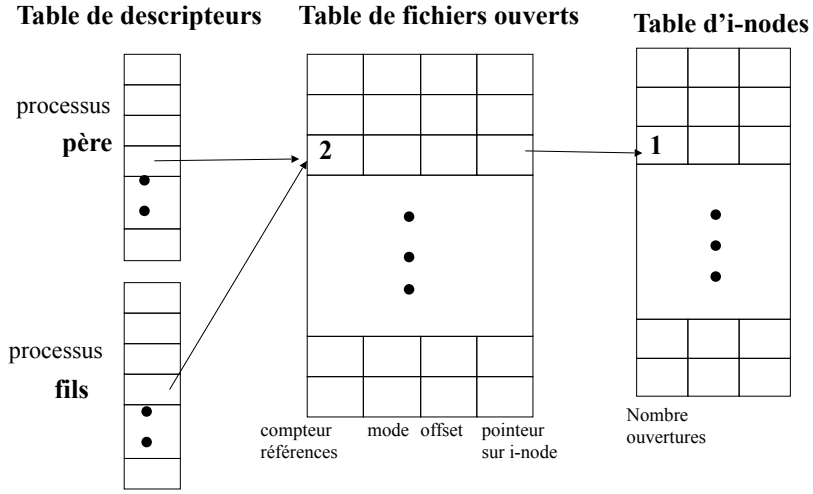
3. Processus: Fork - Héritage

- **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*
  - ...

3. Processus: 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*, …

3. Processus: Fork - Héritage de descripteurs de fichier



3. Processus: Fork - Héritage de descripteurs de fichier

test-fork3.c

```
#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; }
```

```
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;
}
```


3. Processus : Fork - Héritage de variable d'environnement

test-fork4.c

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

char* env; pid_t pid_fils;
```

```
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;
}
```

3. Processus : 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 $?

3. Processus : Terminaison d'un processus

- **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*.

3. Processus: 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.

3. Processus: 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

3. 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;
  }
}
```


3. Processus: 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.

3. Processus: 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

3. Processus: 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;
  }
}
```

3. Processus : 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
      - …

3. Processus : 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.

3. Processus : 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

3. Processus : 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;
}
```

3. Processus : 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;
}
```


4. 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.

4. Signaux: 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

4. Signaux: Les principaux signaux POSIX

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

4. Signaux: Les principaux signaux POSIX

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

4. SIGNAUX

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

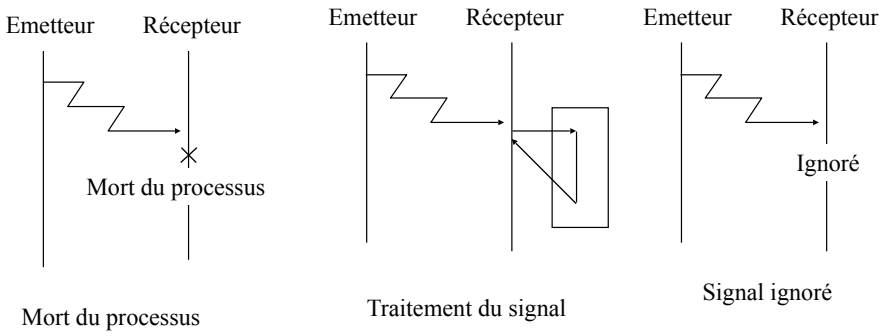
    ```
    $ kill –l
    ```

  - **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.

4. 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

4. SIGNAUX: Conséquence de la délivrance d'un signal



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

4. SIGNAUX: 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**

4. Signaux : 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.

4. Signaux : 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

4. Signaux : 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 du masque de signaux et du handler
    - *fork()* suivi par un *exec() :* réinitialisation dans le fils avec les handlers par défaut.

4. Signaux : 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)

4. Signaux : 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);**

4. Signaux : 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

4. Signaux : Exemple – masquage signaux

sigprocmask-ex.c

```
#define _POSIX_SOURCE 1

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

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;
}
```


4. Signaux : 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

4. Signaux : 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**

4. Signaux : 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

4. Signaux : 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;
```


4. Signaux : 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.

4. Signaux : 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;
```


4. Signaux : 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;
}
```

4. Signaux : 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 ()

4. Signaux : 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;
     }
```


4. 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

4. Signaux : 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

4. Signaux : Utilisation des temporisateurs (*alarm et setitimer*)

- **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 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

4. Signaux : *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.

4. Signaux *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

4. Signaux: *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