Programmation Répartie
Master 1 Informatique – 4I400

Cours 5 : Processus, Signaux

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Plan

On a vu au cours précédent

- Threads, mutex, conditions : en mémoire partagée !

Aujourd'hui : Posix, Processus, Fichiers, Signaux

- POSIX : norme pour l'échange avec le système
- Création et fin de processus POSIX
- Inter Process Communication
- Signaux

Références :

Cours de P.Sens, L.Arantes (PR <= 2017)

Cppreference

Le man, section 2

POSIX

La Norme POSIX

POSIX : principe

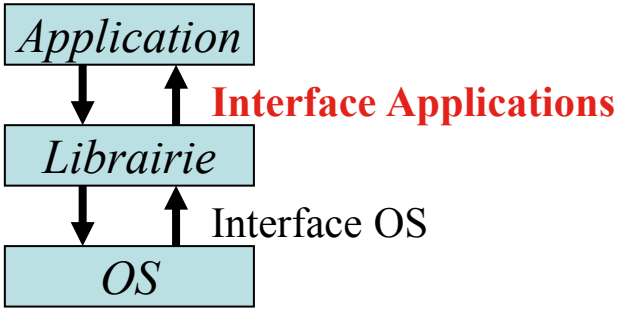
Portable Operating System Interface for Computing Environments

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) sur tous les systèmes POSIX



NB : Windows n'est pas POSIX

Macro _XOPEN_SOURCE

#define **_XOPEN_SOURCE 700**

La Norme POSIX :standard IEEE

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)*

La Norme POSIX : interface

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>

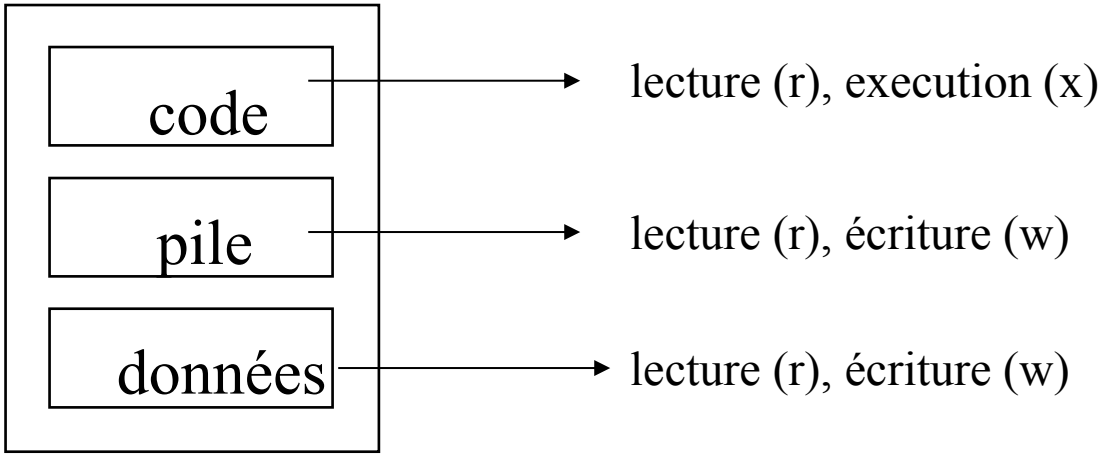
Processus POSIX

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 :Segments



Processus

Chaque processus est indépendant : Espace d'addressage distinct

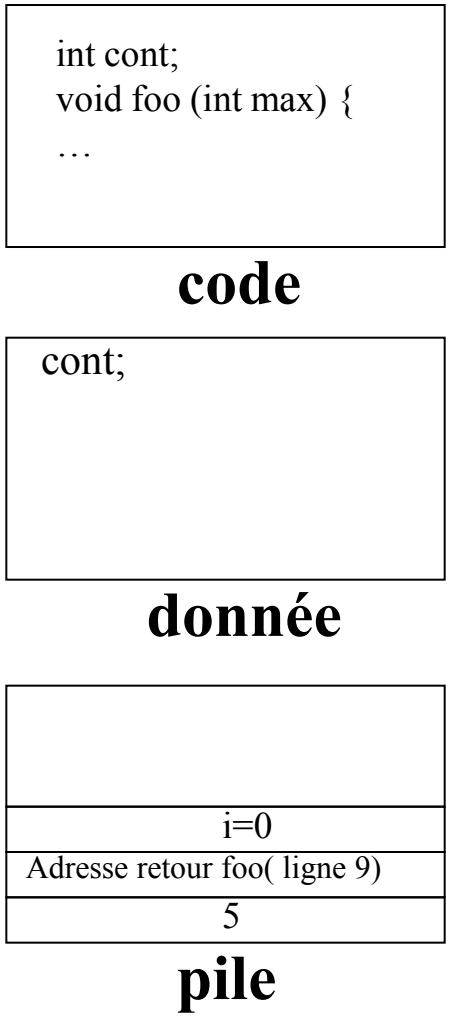
- ✓ Deux processus peuvent être associés au même programme (code)
- ✓ Synchronisation entre processus (communication) possibles

CPU est partagé (temps partagé)

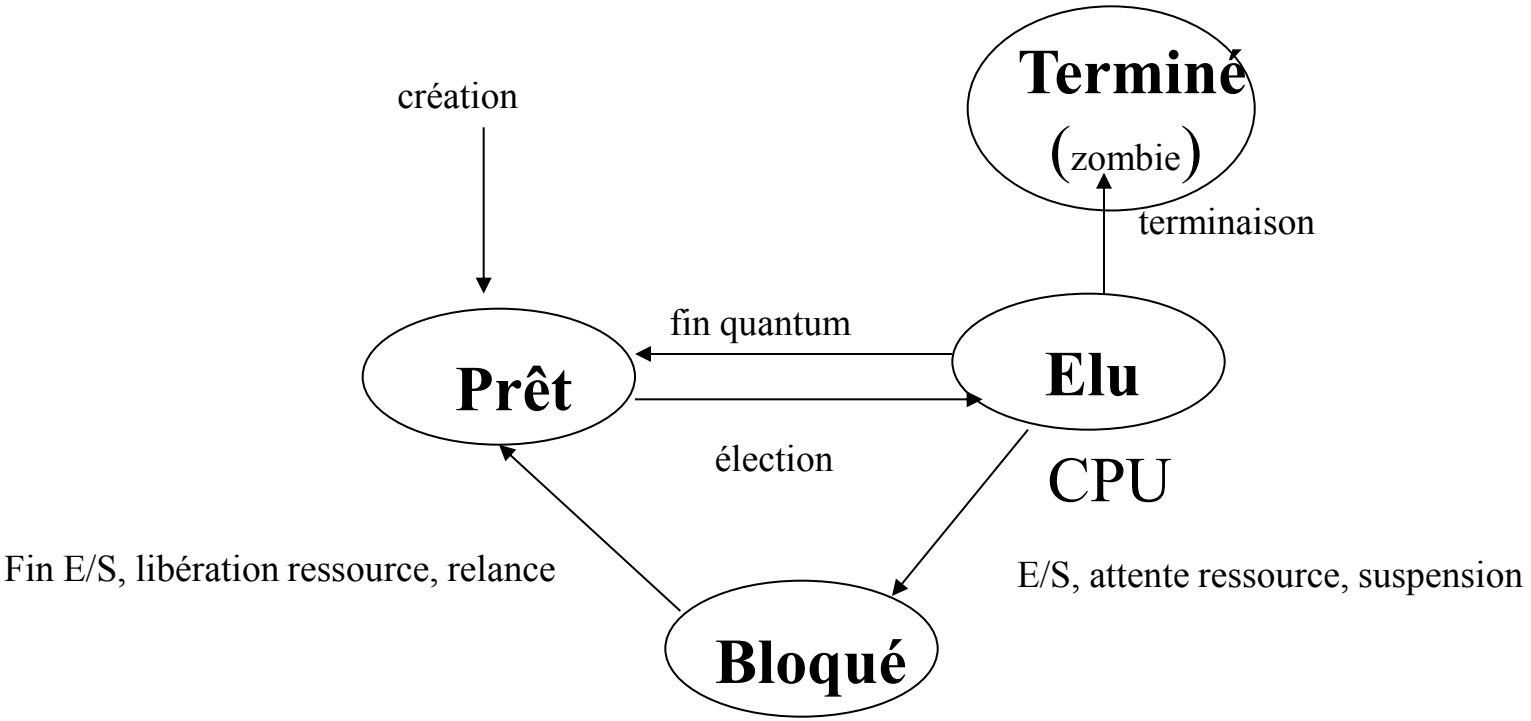
- ✓ Commutation entre les processus

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: }
```



Processus: Etats d'un processus



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

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 _XOPEN_SOURCE 700
#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;
}
```

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*

Fork : création de 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
- ✓Tous les processus POSIX sont issus de fork

création d'un processus (2)

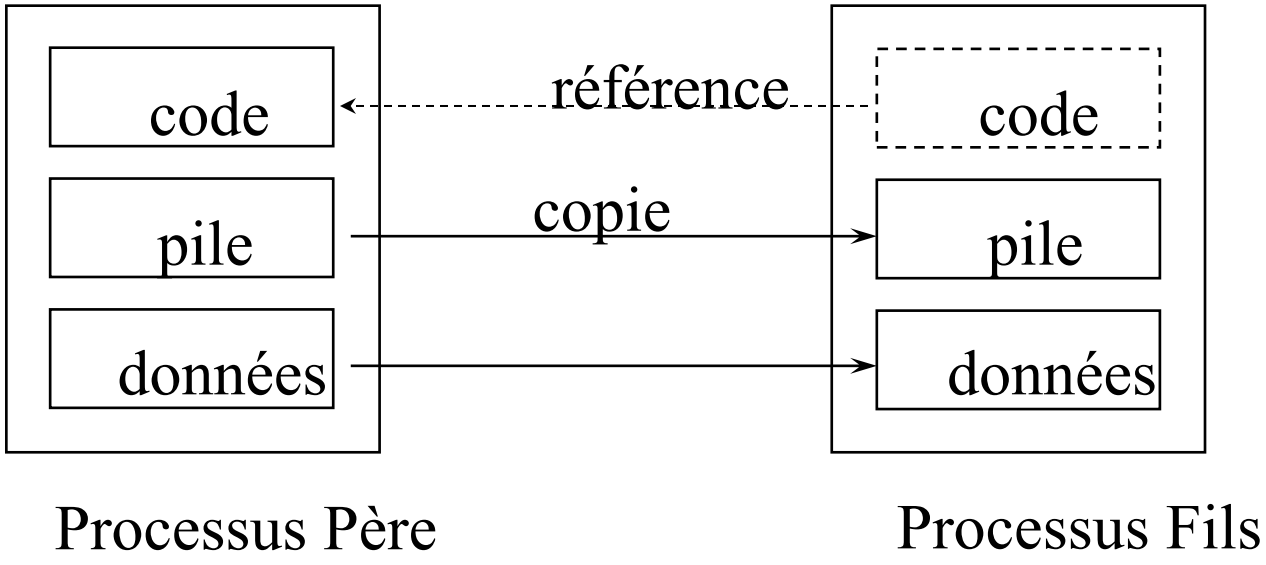
Chaque processus reprend son exécution en effectuant un retour d'appel fork

- ✓ 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

Mnémotechnique : chaque processus a un seul père (getppid), le père obtient le pid du fils créé (car il n'a pas d'API a posteriori pour trouver ses fils)

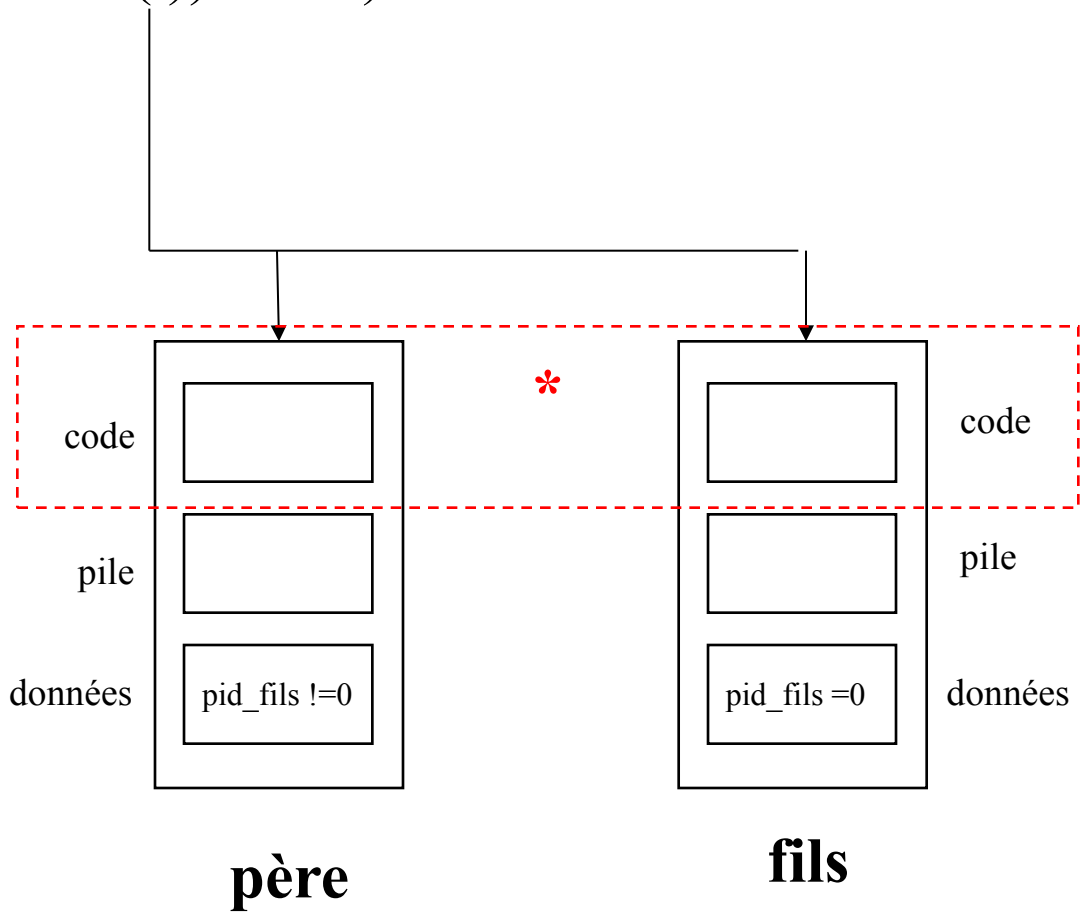
fork

- ✓ 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* : une copie !
  - toute modification d'une variable par l'un des processus n'est pas visible par l'autre.
  - Attention différent de la sémantique thread



Fork : création d'un processus

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



Fork - exemple

```
#define _XOPEN_SOURCE 700
#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;
}
```

Fils voit a = 6 + 3

Père voit a= 6

test-fork1.c

Processus: Fork exemple

Combien de processus sont créés?

test-fork2.c

```
#define _XOPEN_SOURCE 700
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

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

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 (shared memory anonyme)
- ✓ **Les descripteurs de fichiers ouverts** (dont pipe anonymes)
- ✓ Valeur de *nice*
- ✓ ...

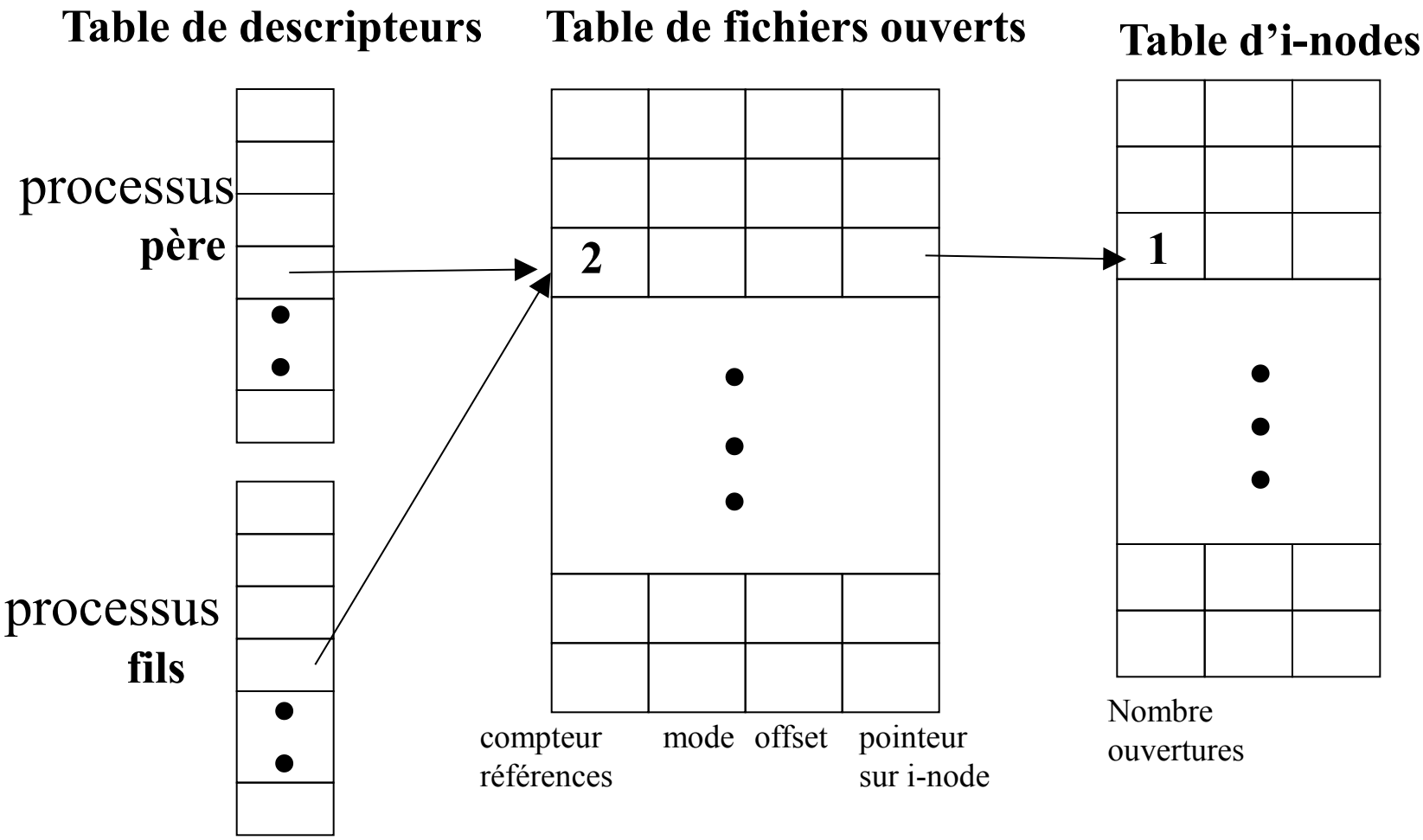
Fork - Héritage

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 de descripteurs de fichier

Les processus fils partagent l'offset dans les fichiers ouverts avec le père



Fork - Héritage de descripteurs de fichier

test-fork3.c

```
#define _XOPEN_SOURCE 700
#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;
}
```

```
>test-fork3 fich
  abcdef123
>cat fich
  abcdef123
```

Terminaison et Wait

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 : zombie et wait

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*.

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

test-wait.c

```
#define _XOPEN_SOURCE 700
#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

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

#define N 3

int cont =0;
```

Quelle est la valeur affichée de *cont* ?

test-wait3.c

```
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

test-waitpid1.c

```
#define _XOPEN_SOURCE 700
#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);
  }
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
```

exec : changer de programme

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
    - …

Attention sauf erreur, on ne revient pas de exec.

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

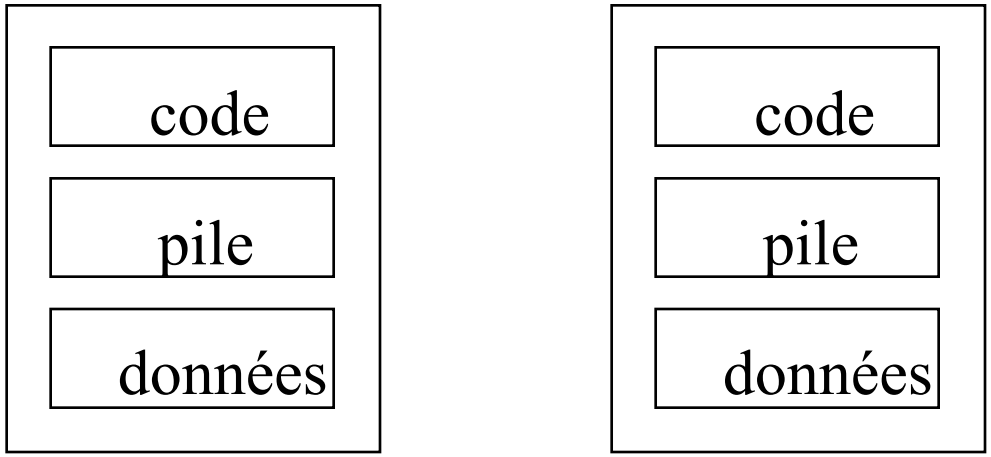
Exemple : execl

```
#define _XOPEN_SOURCE 700
#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;
}
```

Inter Process Communication

Communication entre processus

Comment les processus peuvent communiquer, synchroniser ou partager des données?



Processus P1 **Processus P2**

Processus ne partagent pas leur segments de données

Communiquer *entre* processus

- Plus difficile qu'en thread :
  - Pas d'espace d'adressage commun
- Différents mécanismes utilisés (POSIX) :
  - Fichiers
  - Signaux
  - Tubes et tubes nommés (pipe)
  - Segment de mémoire partagée ou mmap
  - Files de messages
  - Sockets
- Et aussi des primitives de synchronisation inter process
  - Semaphore

IPC : Outils et principes

Inter-Process Communication (IPC)

- ✓ Modèle processus : moyen d'isoler les exécutions
  - Canaux de communication basiques : `wait/exit`, `kill`, ...
- ✓ Pb : Nécessité de communication/synchronisation étroite entre processus
  - Canaux basiques pas toujours suffisants
  - Solutions par fichiers (*eg. tubes*) peu efficaces et pas forcément adaptées (*eg. priorités*)
- ✓ Trois mécanismes de comm/synchro entre processus locaux via la mémoire
  - Les files de messages
  - La mémoire partagée
  - Les sémaphores

Observation: l'identifiant de l'IPC doit commencer par "/ " (e.g. "/file")

Signaux

Signaux

Mécanisme de communication de base inter processus POSIX

- ✓ 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 = mourir, rien.

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

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

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

Les signaux C++11

Constant	Explanation
SIGTERM	termination request, sent to the program
SIGSEGV	invalid memory access (segmentation fault)
SIGINT	external interrupt, usually initiated by the user
SIGILL	invalid program image, such as invalid instruction
SIGABRT	abnormal termination condition, as is e.g. initiated by std::abort()
SIGFPE	erroneous arithmetic operation such as divide by zero

Définis dans <signal>

- Sous ensemble compatible POSIX
- Deux primitives seulement :
  - signal (positionne un handler)
  - raise (envoie un signal au processus **courant**)
- Pas d'interprocessus dans C++11

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.**

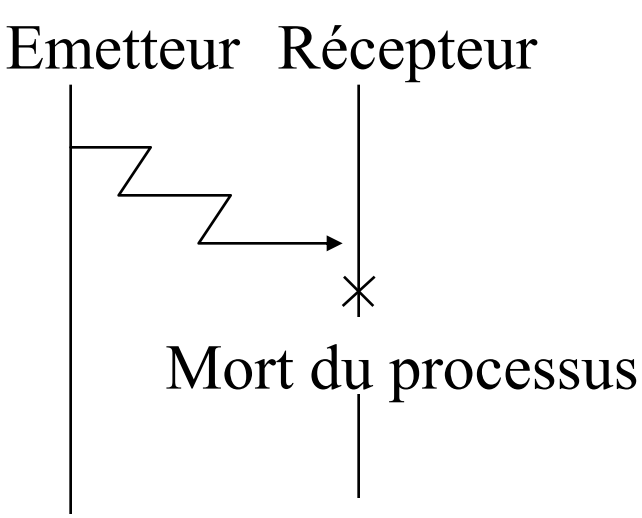
Délivrance

- 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**.**

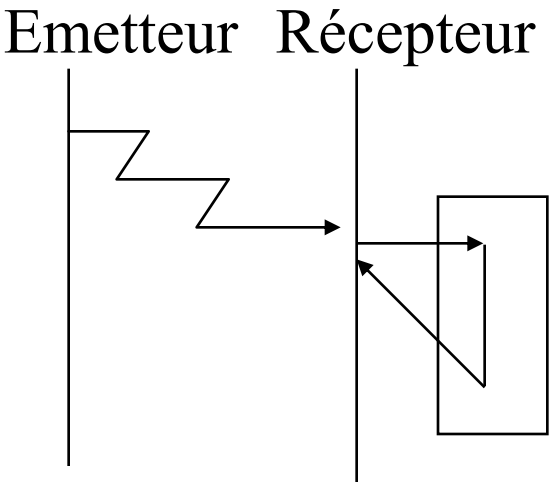
Signal masqué ou bloqué

- La délivrance du signal est ajournée

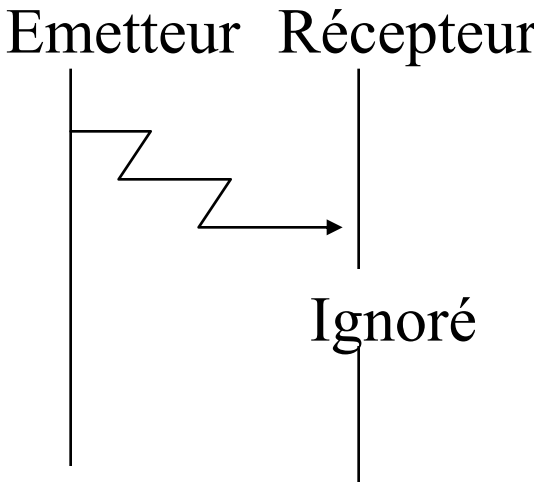
Conséquence de la délivrance d'un signal



Mort du processus



Traitement du signal



Signal ignoré

Ne pas confondre avec les interruptions

- ✓ Matérielles : int. horloge, int. Disque, etc.

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) *

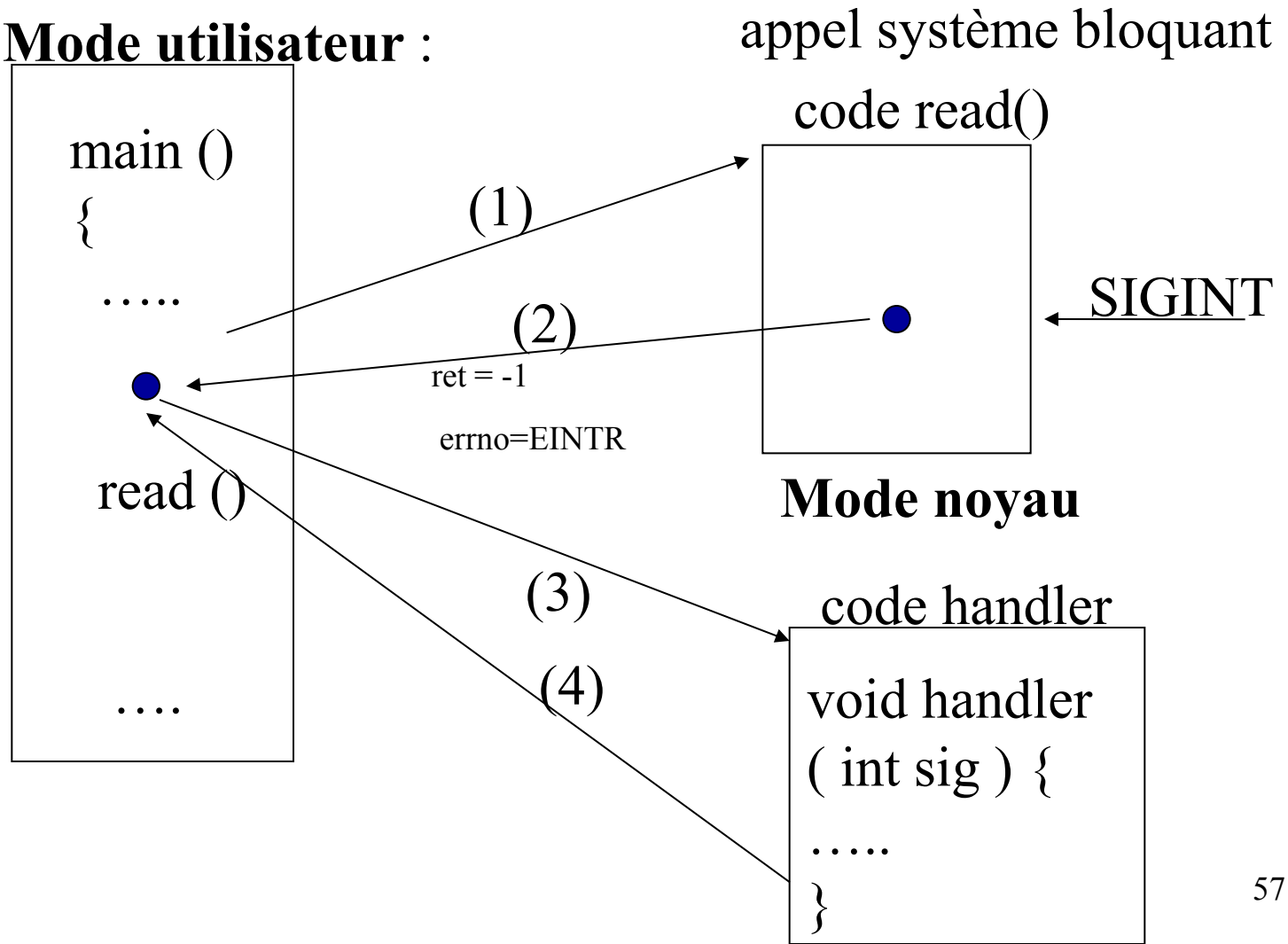
- ✓ **SIG_IGN** (ignorer le signal)
- ✓ **Fonction** définie par l'utilisateur
- ✓ **SIG_DFL** (restituer le comportement par défaut)
  - Applicable à tous les signaux sauf **SIGKILL**, **SIGSTOP**
- Utiliser la fonction **signal** pour positionner un handler en C++11

signal reçu dans un appel système

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



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

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;
}
```

Equivalent à raise(SIGINT)

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.

Signaux : masquage

- On utilise un struct sigset_t
  - Représente un ensemble de signaux
  - Jeu de primitives pour initialiser : sigfillset, sigaddset…
- On bloque les signaux avec sigprocmask
  - Sauf SIGKILL et SIGSTOP
  - Les signaux masqués deviennent pendants
    - On perd les doubles notifications
    - sigpending : permet de les interroger

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

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;
}
```

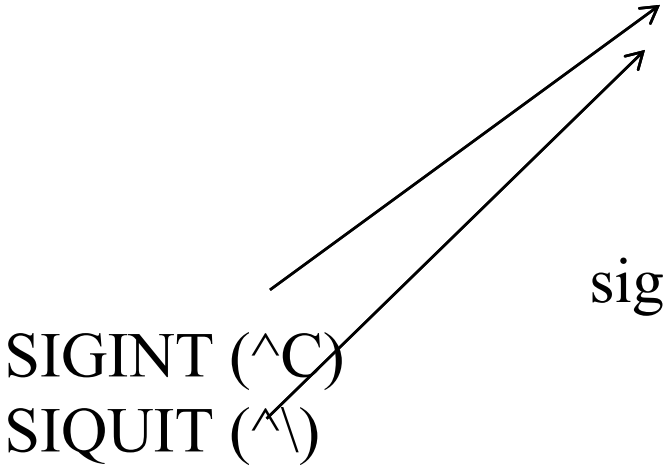
Signaux : Attente d'un SIGNAL

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.
  - On fait donc un « fill » puis un « delset » des signaux intéressants

Signaux : sigsuspend

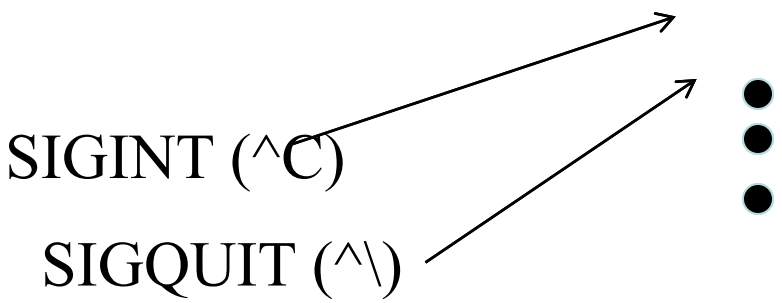



```
int main(int argc, char **argv) {
    ●
    ●
    ●
sigsuspend (&sig_proc);
    ●
    ●
    ●
    return EXIT_SUCCESS;
}
```

Processus se termine avant le sigsuspend

Signaux : Exemple – sigprocmask + sigsuspend

```
int main(int argc, char **argv) { sigset_t sig;
    /* masquer SIGINT et SIGQUIT*/
    sigempty (&sig);
    sigaddset (&sig, SIGINT); sigaddset (&sig, SIGQUIT);
    sigprocmask (SIG_SETMASK, &sig, NULL);
```



Signaux pendants

```
    /* démasquer SIGINT et SIGQUIT*/
    sigfillset (&sig);
    sigdelset (&sig, SIGINT); sigdelset (&sig, SIGQUIT);
    sigsuspend (&sig);

    return EXIT_SUCCESS;
}
```

Processus se termine lors du sigsuspend

Signaux : Changement du traitement par défaut

```
struct sigaction { void (*sa_handler)  () ; /*fonction */
                   sigset_t sa_mask;  /* masque des signaux */
                   int sa_flags; /* options, utiliser 0 */
```

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
  - On peut passer un masque initialisé mais vide.
- ✓ *sa_flags*: différentes options

Essentiellement, on passe un pointeur de fonction à invoquer sur réception du signal.

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

Exemple sigaction

sigaction-ex.c

```
#define
_XOPEN_SOURCE 700

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

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(SIGINT, &action,NULL);

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

  return EXIT_SUCCESS;
```

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

SIGSTOP SIGCONT et SIGCHLD

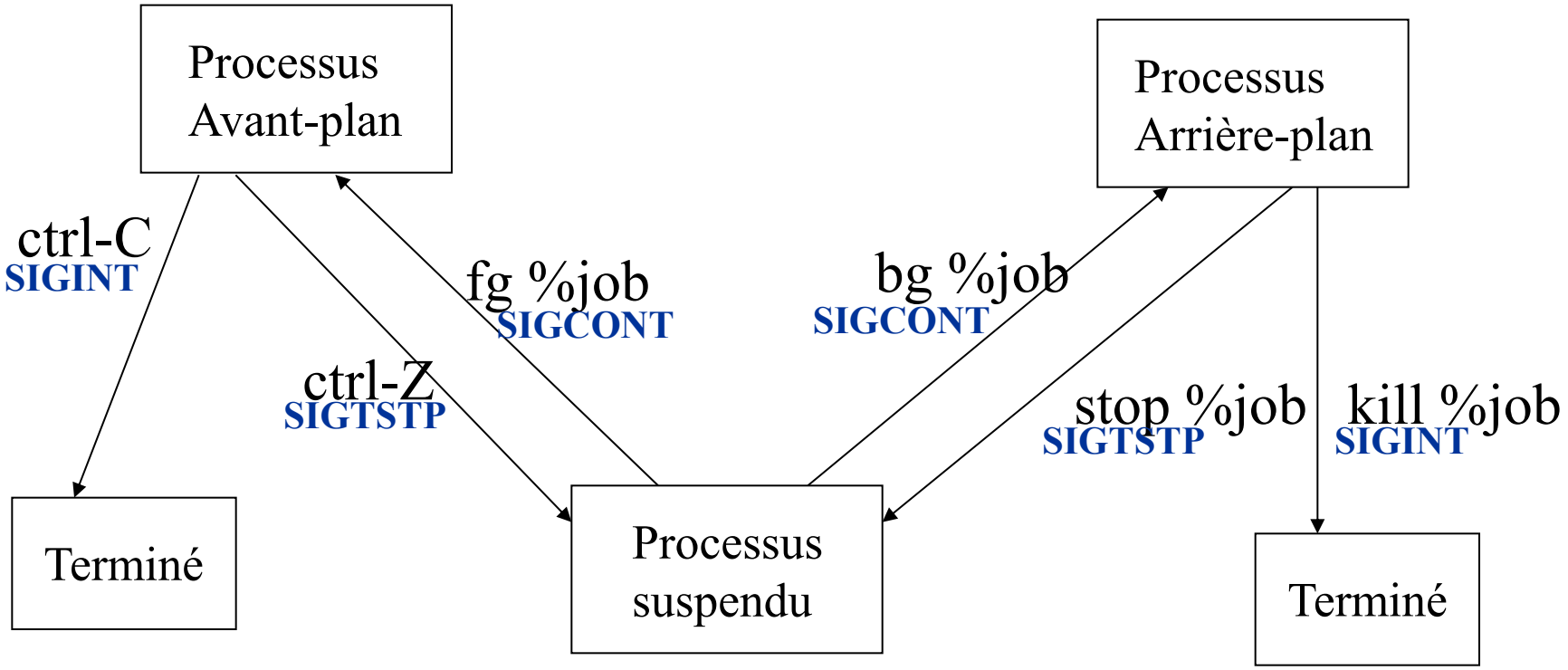
Processus s'arrête (état bloqué) en recevant un signal SIGSTOP

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
- ✓ Observation :
  - en fait le processus père reçoit un SIGCHLD a chaque fois qu'un de ses fils change de status (exemple, processus fils redémarre en recevant un SIGCONT)

Gestion de Jobs

Gestion par des signaux



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

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.

<signal>

- En C++11, principalement intercepter les signaux du système
  - SIGSEGV, SIGILL, SIGFPE... : fautes du programme
  - SIGINT, SIGTERM : messages du système, ^C
- Pour cela : sig_handler signal (int signal, sig_handler handler)
  - Signal : un signal, e.g. SIGINT
  - Sig_handler : le gestionnaire
    - SIG_DFL, SIG_IGN : default et ignore, deux macros
    - void fun(int sig);
- raise (int sig) : permet de s'envoyer un signal.