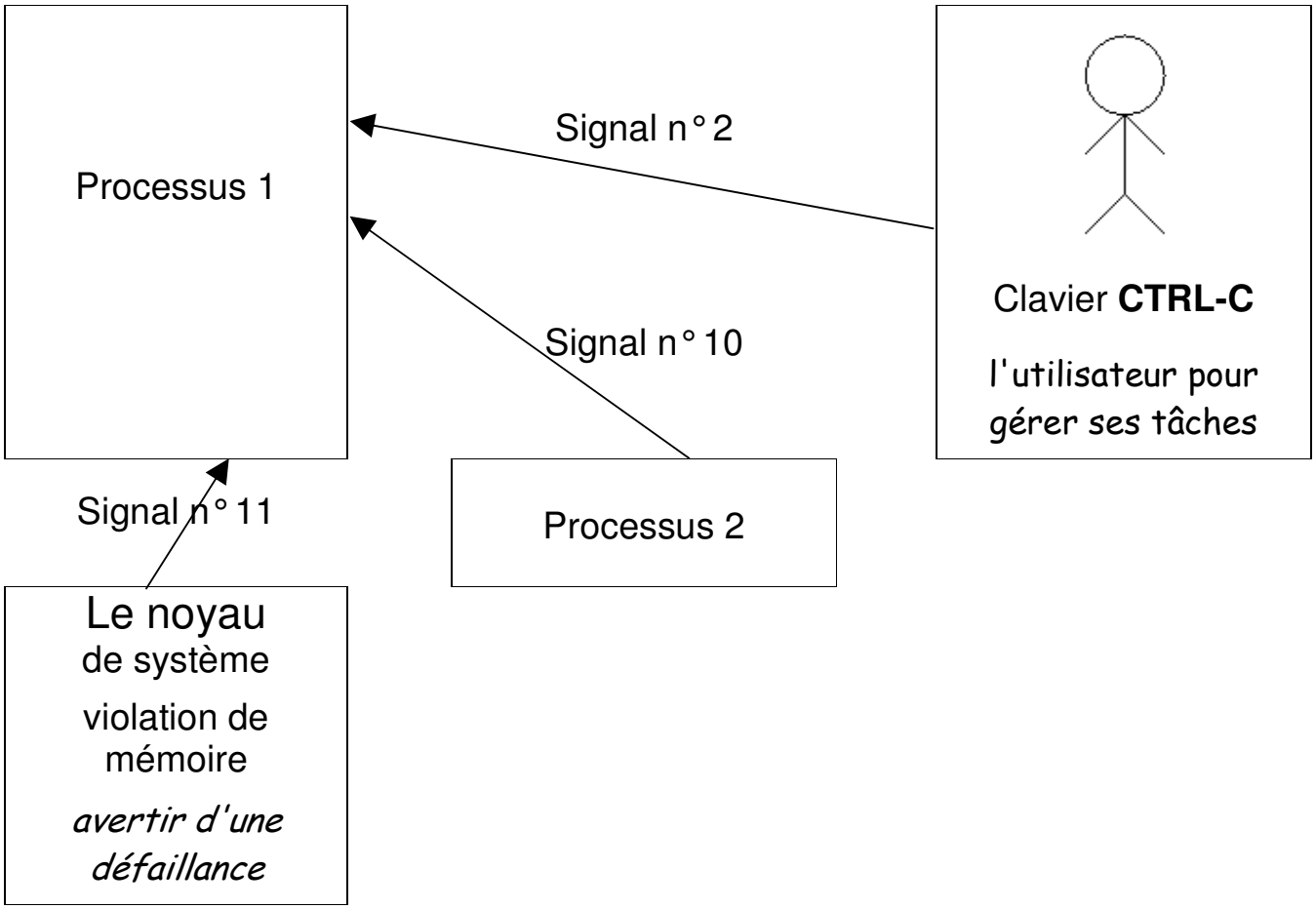
Chapitre 2

Communication interprocessus (suit)

2.3/ Communication par signaux.

2.3.1/ Introduction :

Les signaux (ou les interruptions) servent à informer les processus de l'arrivée d'un événement et quelque chose s'est produit dans le système, on lui associe également l'action à entreprendre :



Références :
L. Bobelin Unix, polycopiés de cours, Polytech, 2011
J. Boukachour, Polycopiés de cours, Université du Havre, 2013.

Les signaux de Unix ont des origines diverses,

les signaux :

- Peuvent être transmis à tout moment ou bien occasionnellement

- Sont émis à **partir :**
  - **du clavier**(suite à la frappe d'une combinaison de touches : par exemple **CTRL-C** pour envoyer le signal SIGINT; CTRL D...)

  - d'un **processus**(par la primitive kill dans un programme C).

  - par la commande **kill** depuis le **shell**

  - ou du **noyau**(lors de la constations d'une anomalie matérielle : Violation mémoire, erreur E/S, division par zéro)

- Permettent d'informer les processus de l'occurrence d'un événement exceptionnel, jamais comme un moyen de communication ordinaire

2.3.2/ Comportement du processus (Réaction aux signaux)

Lors de la réception d'un signal, il y a des actions par défaut suivant le signal :

- Ignorer le signal sans exécuter aucun traitement
- Terminer le programme.
- Un processus peut changer son comportement par défaut lors de la réception d'un signal en indiquant la fonction à exécuter lors de la réception du signal.
- Avec les signaux il n'y a pas de communication de données. Ils sont identifiés par un numéro entier et un nom symbolique décrit dans signal.h.
- Le récepteur ne peut pas déterminer l'identité de l'émetteur.
- L'interruption d'un processus par un signal est **impossible** quand il est en **mode noyau** : Il faut qu'il passe en mode utilisateur.
- Les signaux sont sans effet sur un processus zombi

2.3.3/ Identification des signaux

Unix gère 64 signaux différents, Chaque signal est identifié par un nom et un numéro. On distingue deux classes de signaux :

– Classiques, numérotés de 1 à 31
– Signaux temps réels, numérotés de 32 à 63

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Voici la signification des signaux principaux :

- SIGINT 2 signal d'interruption (provoqué par CTRL-C par exemple) ;
- SIGQUIT 3 caractère quit frapper au clavier (Ctrl - \) ;
- ...
- SIGKILL 9 fin du processus (non déroutable) ;
- SIGUSR1 10 (aussi 30,16) signal émis par un processus utilisateur (disponible pour les applications)
- SIGSEGV 11 violation d'espace mémoire
- SIGUSR2 12 (aussi 31,17) signal émis par un processus utilisateur (disponible pour les applications)
- SIGPIPE 13 écriture dans un tube sans lecteur
- SIGALRM 14 signal déclenché après un certain nombre de secondes (horloge) ;
- SIGTERM 15 fin de processus ;
- ...

- SIGCONT 18 est envoyé à un processus pour lui faire reprendre sonexécution (après un SIGSTOP).
- SIGSTOP 19 (CNTL-Z) pour pouvoir stopper un processus (stopper pour reprendre plus tard, pas arrêter) ;
- ...
- SIGIO 29
- SIGUSR1 30 signal utilisateur 1 (disponible pour les applications) ;
- SIGUSR2 31 signal utilisateur 2 (disponible pour les applications).

- Certains signaux ont des statuts particuliers :

    - SIGKILL ne peut pas être intercepté, bloqué ou ignoré. Cela permet de tuer un signal même en cas de processus récalcitrant.

    - SIGSTOP est dans le même cas, pour pouvoir stopper un processus (stopper pour reprendre plus tard, pas arrêter).

    - SIGCONT il est envoyé à un processus pour lui faire reprendre son exécution (après un SIGSTOP). Il est donc logique qu'il ne puisse être pris en charge par un handler .

2.3.4/ Comment manipuler les signaux

Dans le cas du système Unix/Linux, un processus utilisateur peut **envoyer** un signal à un autre processus.

- Les deux processus appartiennent au même propriétaire,
- ou bien le processus émetteur du signal est le super-utilisateur.

Il existe 2 moyens :

- Par la ligne de commande,
- Par une primitive en c.

Le programmeur peut mettre en place des méthodes (handlers) pour **prendre en charge** chacun des signaux reçus.

2.3.4.1/ Commande Kill (shell)

- La commande kill permet d'envoyer un signal à un processus
- **Syntaxe du kill :** Kill -n°du signal (ou le nom du signal) PID
- **Exemples du Kill :**

  `Kill -9 2235` (envoie du signal 9 au processus 2235)

  le processus recevant ce signal exécutera la fonction correspondante (au signal 9) qui est toujours la terminaison (état zombie)

  `Kill -2 2235` (envoie du signal 2 au processus 2235) (équivalent de CTRL-C)
- Le signal SIGKILL (N°9) ne pas être dérouté. Le comportement par défaut à la réception du signal 9 est la destruction du processus.
- Si le numéro du signal n'est pas précisé, kill envoie le signal 15 (SIGTERM)
- La commande `kill -l` donne la liste des signaux d'un système Unix.

2.3.4.2/ La primitive kill (en C) :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

Il prend en argument le `pid` du processus destinataire du signal et le numéro du signal à envoyer `signal`.

`pid` peut prendre comme valeurs :
. Si pid > 0 : processus pid.
. Si pid = 0 : groupe de l'émetteur.
. Si pid = -1 : tous les processus (seulement root peut le faire).

La valeur de retour de `kill` est 0 en cas de succès ou -1 en cas d'échec.

Dans le bloc de contrôle d'un processus, le noyau mémorise entre autres :
– Les signaux reçus
– Les fonctions de traitement des signaux

La réponse d'un processus à un signal déclenche l'exécution d'une routine standard.

Il est possible de modifier la réponse en définissant une fonction spécifique, appelée `handler`

ATTENTION : cet appel système est particulièrement mal nommé (kill : tuer), car il ne tue que très rarement un processus

Exemple d'envoi d'un signal

```
/* sig1.c */
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
void main(void) {
pid_t p;
int etat;
if ((p=fork()) == 0) { /* processus fils qui boucle */
while (1);
exit(2);
}
/* processus pere */
sleep(6);
printf("envoi de SIGUSR1 au fils %d\n", p);
kill(p, SIGUSR1);
// bloque l'appelant (pere) et selectionne le fils
p = waitpid(p, &etat, 0);
printf("etat du fils %d : %d\n", p, etat);
}
```

Voici le résultat de l'exécution :

```
cc  sig1.c  -o  sig1

./sig1  &
[1] 9211
nakechbm@corton:~/s4#/tp5$ envoi de SIGUSR1 au fils 9212
etat du fils 9212 : 10

[1]+  Termine 23              ./sig1
```

2.3.5/ Traitement des signaux

Méthodes de traitement :

- Traitement standard

- Traitement par une fonction définie par l'utilisateur (un handler).

Règles générales :

- A chaque type de signal est associé un handler par défaut. Le plus commun est la terminaison du programme (kill -9).

- Un processus peut ignorer un signal en lui associant le handler,

- si le processus exécute un programme utilisateur : traitement immédiat du signal reçu,

- s'il se trouve dans une fonction du système (ou system call) : le traitement du signal est différé jusqu'à ce qu'il revienne en mode user, c'est à dire lorsqu'il sort de cette fonction.

2.3.5.1 Traitement standard du signal.

Deux stratégies de gestion des signaux :

- SIG_DFL traitement du signal par défaut
- SIF_IGN signal est ignoré

Exemple 1 : Ci-dessous un programme non interruptible avec CTRL-C ; SIGINT est ignoré. *Le programme doit être arrêté avec kill -9 no.PID. Par contre, si l'appel signal() est mis en commentaire, le programme s'arrête bien avec CTRL-C.*

```
// sig3_1.c
#include<signal.h>
void main (){signal (SIGINT,SIG_IGN) ;
for ( ; ;) printf(" boucle infinie \n"); }
```

Exemple 2:

```
/* sig3_2.c */
#include <stdio.h>
#include <signal.h>
int main(void)
{    signal(SIGINT, SIG_DFL);
/* traitement standard  du signal  SIGINT   */
    printf(" Dans le programme principal (man) :\n ");
  printf("appui sur Ctrl-C pour envoyer le signal n° 2 : \n");
  printf("ce signal va provoquer le traitement \n
standard : signal(SIGINT, SIG_DFL) \n");
    printf(" qui arrêtera le processus \n");
    printf(" Avant la boucle :\n ");
    for (;;) { }
    printf(" apres la boucle :\n ");return 0;}
```

Voici le résultat de l'exécution :

```
./sig3_2
 Dans le programme principal (man) :
 appui sur Ctrl-C pour envoyer le signal n° 2 :
ce signal va provoquer le traitement
 standard : signal(SIGINT, SIG_DFL)
 qui arrêtera le processus
 Avant la boucle :
^C
```

2.3.5.2 Mise en place d'un handler

Un handler est la fonction qui décrit la suite des instructions à effectuer lors de la réception d'un signal.

Les signaux (autres que SIGKILL, SIGCONT et SIGSTOP) peuvent avoir un handler spécifique installé par un processus :

La primitive `signal()`

```
#include<signal.h>
signal(numero_signal, methode_de_traitement);
```

- Elle installe le handler spécifié par `methode_de_traitement` pour le signal `numero_signal`.

- La fonction de handler (**`methode_de_traitement`**) est exécutée par le processus à la délivrance d'un signal (**`numero_signal`**). A la fin de l'exécution de cette fonction, l'exécution du processus reprend au point où elle a été suspendue.

Exemples de mise en place handler

Exemple1 : de mise en place handler

```
/* sig4_1.c */
#include <stdio.h>
#include <signal.h>
void traitement_interruption(int signum)
{
printf("Procédure simulant le traitement d'interruption
\n");}
int main(void)
{    printf(" Dans le programme principal (man) \n ");
    printf("appui sur Ctrl-C pour envoyer le signal n°
    2 : \n");
    signal(SIGINT, traitement_interruption);
    printf(" avant la boucle :\n ");
    for (;;) { }
    printf(" après la boucle :\n ");
    return 0;}
```

Voici le résultat de l'exécution :

```
cc  sig4_1.c  -o  sig4_1
./sig4_1
 Dans le programme principal (man) :
  avant la boucle :
^C appui sur Ctrl-C
Arret au prochain coup
^C
```

Exemple2 : Mise en place handler : Déroutement

Ci-dessous un programme provocant une division par zéro :

```
/*sig4_2.c */
#include<stdlib.h>
#include<stdio.h>
#include <unistd.h>
#include <signal.h>
main() { int a, b, Resultat;
         printf("Taper a : "); scanf("%d", &a);
         printf("Taper b : "); scanf("%d", &b);
         Resultat = a/b;
        printf("La division de a par b = %d\n", Resultat);}
```

Voici le résultat de l'exécution :

```
./sig4_2
Taper a : 5
Taper b : 0
Exception en point flottant
```

Nous allons mettre en place un handler permettant un traitement personnalisé de la division par zero :

```
/*sig4_3.c */
#include <signal.h>
#include <stdio.h>
void Hand_sigfpe() {
    printf("\nErreur division par 0 !\n");
    exit(1);
}
main() {
    int a, b, Resultat;
    signal(SIGFPE, Hand_sigfpe);
    printf("Taper a : "); scanf("%d", &a);
    printf("Taper b : "); scanf("%d", &b);
    Resultat = a/b;
    printf("La division de a par b = %d\n", Resultat);
}
```

Voici le résultat de l'exécution :

```
./sig4_3
Taper a : 5
Taper b : 0
Erreur division par 0 !
```

Info2 #, Programmation système TP5, Signaux première partie

Enseignant : M. Nakechbandi

Exercice 1

1. Compiler, exécuter et tester le programme `sig3_1.c` (page 11)
2. Modifier ce programme pour ignorer les cinq premiers CTRL-C (signal `SIGINT`) et réagit au sixième.
3. Idem modifier le programme `sig3_2.c` pour que le programme s'arrête quand on frappe CTRL-\ (signal `SIGQUIT`)

Exercice 2

1. Compiler, exécuter et tester le programme `sig4_1.c` (page 13)
2. Modifier ce programme (`ex2.2.c`) pour que le processus exécute 3 handlers sur réception de `SIGINT` de `SIGUSR1` ou de `SIGUSR2`, un message différent selon le signal sera affiché.

Exercice 3

1. Compiler, exécuter et tester le programme `sig1.c` (page 9)
2. S'inspirer de ce programme pour écrire un nouveau programme (`ex3_2.c`) prenant un pid en paramètre. Ce programme saisit un caractère au clavier qui peut être '+', '-' ou 'A'. Cette saisie se fait en boucle jusqu'à ce que l'utilisateur tape 'A'. Après chaque saisie et en fonction du caractère tapé par l'utilisateur, le programme envoie les signaux suivants :

   - si le caractère est '+' , le programme envoie le signal `SIGUSR1`
   - si le caractère est '-', le programme envoie le signal `SIGUSR2`
   - si le caractère est 'A', le programme envoie le signal `SIGINT`

Ces signaux sont envoyés au processus dont le pid est passé en paramètre.

Tester ce programme en couple avec le programme de la question 2.2 (`ex3_2.c`)

Aide :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/signal.h>
int main(int argc, char* argv[]) {
  int pid; //pid du pss qui affiche les valeurs
  char ordre;
  pid = atoi(argv[1]);
  do {printf("que voulez-vous faire ? \n");
      ordre = getchar();
      getchar( ); /*pour éliminer le retour-chariot*/
      /*on envoie le bon signal en fonction du caractère saisi*/
```

```
      if (ordre == '+') { kill(... ,...);}
      if (ordre == '-') { kill(... ,...);}
      if (ordre == 'A') { kill(... ,...);}
     }
  while (ordre != 'A');
}
```

3. Tester ce programme en couple avec le programme de la question 2.2 d'un autre utilisateur. Quelle est votre conclusion

Exercice 4

1. Ecrire un programme e(`ex4_1.c`) qui
   - affiche par défaut et à l'infini la valeur d'une variable val (initialisée à 1)
   - sur réception du signal CTRL-C, augmente la valeur de val de 1
   - sur réception du signal CTRL-\, diminue la valeur de val de 1

Aide :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/signal.h>
int val;
void augmenter(int sig) {  val++;}
void diminuer(int sig) {  val--;}
int main() {
  val = 1;
  signal(..., augmenter);
  signal(..., diminuer);
  while (1)
    {
      printf("val = %d\n",val);
      sleep(1);    } }
```

2. Modifier le programme précédent (`ex4_2.c`) pour qu'il prenne en paramètres les bornes min. et max. Lorsque la borne max. (ou min) est atteinte le programme s'arrête. **Indications :**
   - `min` et `max` pouvant être lu comme paramètres. Utiliser :
     ```
     if (argc != 3) {printf("Erreur dans le nb d'arguments
     !\n"); exit(1);}
      min = atoi(argv[1]);
      max = atoi(argv[2]);
      ...
     ```
   - S'inspirer de l'exemple `sig3_2.c` page 11 pour arrêter le programme avec un traitement standard du signal en utilisant `signal(SIGINT, SIG_DFL)` ...

La primitive `sigaction` est une autre structure décrit le comportement utilisé pour le traitement d'un signal :

```
struct sigaction {
        void (*sa_handler) ();
        sigset_t sa_mask;
        int sa_flags;}
```

- **sa_handler** : fonction de traitement (ou `SIG_DFL` et `SIG_IGN`) , il est positionné par `sigaction`
- **sa_mask** : ensemble de signaux supplémentaires à bloquer pendant le traitement
- **sa_flags** : différentes options

Exercice 5 :

5.1/ Analyser puis compiler, exécuter et tester le programme `ex5_1.c` ci-dessous. Tester plusieurs fois en envoyant au processus les signaux `SIGINT` puis `SIGUSR1`.(d'une fentre `kill -10 pid`)

```
/* Exercice ex5_1.c : */
#include<stdio.h>
#include<sys/types.h>
#include<sys/signal.h>
            /* handler (fonction) associé au signal SIGINT */
void traitement (int sig)
      {  printf("signal SIGINT reçu !\n"); }
int main()
{
  struct sigaction action;
            /* structure permettant de construire un handler*/
  sigset_t masque;
            /* masque contenant les éventuels signaux masqués*/
                  /* pendant l'execution du handler */
                    /* on initialise le masque à vide : */
                 /* on en souhaite masquer aucun signal */
  sigemptyset(&masque);    /* on remplit la structure du handler*/
  action.sa_handler = traitement;
                              /* la fonction associée au signal */
  action.sa_mask = masque;    /* le masque des signaux bloqués */
  action.sa_flags = 0;        /* d'eventuelles options */
                              /* on installe le handler */
  /* cela veut dire qu'on l'associe a un signal paticulier :ici SIGINT*/
  /* cela veut aussi dire qu'on rend le handler actif */

  sigaction(SIGINT, &action, NULL);
  printf("Mon pid est %d\n", getpid());
  while (1) {  printf("un tour! \n");   sleep(1);   }
}
```

C'est un programme de test pour l'installation d'un handler de signal en utilisant la primitive `sigaction`. Ce programme boucle à l'infini. Pour l'arrêter, il faut récupérer le pid du processus et le tuer dans une autre fenêtre par un `kill -9 pid`.
Pour tester ce programme, il suffit de le lancer et de taper de temps à autres CTRL+C. Au lieu de s'arrêter, le programme affiche le message "signal SIGINT reçu !"

5.2 / Idem que l'exercice ex2.2 (de lundi), modifier ce programme (`ex5_2.c`) pour que le processus exécute 3 handlers sur réception de `SIGINT` de `SIGUSR1` ou de `SIGUSR2`, un message différent selon le signal sera affiché.

5.3 / Modifier le programme précédent 5.1 (`ex5_3.c`) afin que le processus ignore le signal `SIGUSR1`. Faire des tests en envoyant différents signaux à différents moments.
Indication : on ajoute :

```
action.sa_handler = SIG_IGN;
sigaction(SIGUSR1, &action, NULL);
```

Exercice 6 : Refaire l'exercice 4 de tp de lundi (`ex6.c`) en utilisant la structure `sigaction`. Rappelons que cet exercice :

- affiche par défaut et à l'infini la valeur d'une variable val (initialisée à 1)
- sur réception du signal CTRL-C, augmente la valeur de val de 1
- sur réception du signal CTRL-\, diminue la valeur de val de 1

Exerce 7 : Utilisation du masque
Modifier le programme de l'exercice 5.1 (`ex7.c`) précédent pour que le processus, au lieu de boucler indéfiniment, attende jusqu'à ce qu'il ait reçu le signal SIGINT (et uniquement celui là). La réception du signal SIGINT doit toujours provoquer l'exécution du handler *traitement*. Tester en envoyant éventuellement une ou plusieurs fois le signal SIGUSR1 avant d'envoyer le signal SIGINT.

Indication : On remplace la boucle `while (1)` par

```
sigfillset(&masque);
sigdelset(&masque,SIGINT);
sigsuspend(&masque);
```

Explication : On utilise ici la primitive `sigsuspend`, il faut préparer le masque, on met tous les signaux sauf SIGINT, `sigsuspend` bloque le processus en attente de tout signal ne figurant pas dans le masque.

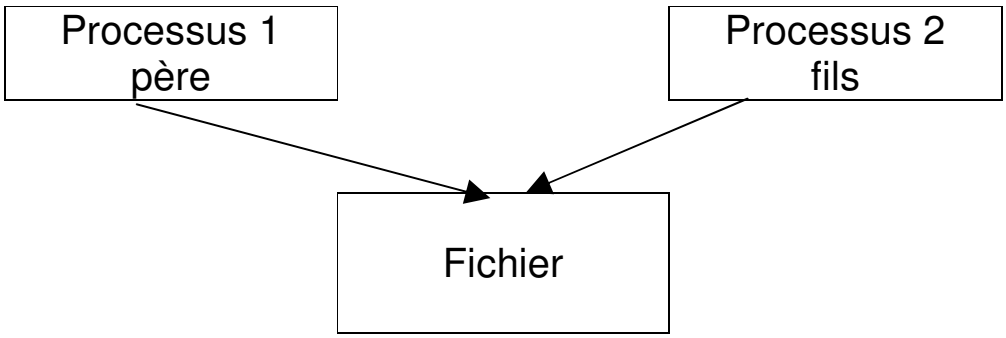
Chapitre 2

Communication interprocessus (suit)

2.4/ Communication par tube

2.4.1/ Introduction.

Un fichier ouvert par un processus est accessible par ses descendants.



Problème : Ouvrir un fichier par deux processus en même temps génère éventuel problème de conflits et d'intégrité des données

Références :
G. Hansel , J.-M. Champarnaud , Passeport pour Unix et C, Ed. Broché ,1999

Exemple :

```
  /* coherence_fichier.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
char chaine[20];
int main(void)
{int descript;
descript = open("toto",O_RDWR,0);
if (fork() == 0)
       { /* processus fils  */
         write(descript, "abcd", 4);
         sleep(2);
         read(descript, chaine, 10);
         printf("\n1ere chaine lue par le fils: %s\n",chaine);
          read(descript, chaine, 10);
         printf("2eme chaine lue par le fils: %s\n",chaine);}
else    { /* processus père  */
          sleep(1);
          read(descript, chaine, 10);
      printf("\n \n1ere chaine lue par le pere: %s\n",chaine);
          write(descript, "ABCD", 4);
          read(descript, chaine, 10);
          printf("2eme chaine lue par le pere: s\n",chaine); }
return 0;}
```

Voici la trace de l'exécution :

```
cat toto
    01234567890123456789
    AZERTYUIOPazertyuiop
    00000000000000000000
./coherence_fichier
    1ere chaine lue par le pere: 4567890123
    2eme chaine lue par le pere: 89
    AZERTYU
    1ere chaine lue par le fils: IOPazertyu
    2eme chaine lue par le fils: iop
    000000
cat toto
    abcd4567890123ABCD89
    AZERTYUIOPazertyuiop
    00000000000000000000
```

La communication par **socket** déjà vue pouvant régler les conflits : Un processus (client) désirant accéder aux données doit soumettre une requête (via la socket) au processus (serveur) responsable de la gestion des données. Le serveur examine la requête et fournit (via la socket) les données correspondant à la requête.

Communication par tubes.

Ce mécanisme permet à deux processus de communiquer entre eux : il s'agit d'un fichier dans lequel l'un des processus écrit des données et l'autre les lit. On distingue :

- les tubes **sans nom** qui ne permettent la communication qu'entre les processus nés d'un même ancêtre après la création du tube par cet ancêtre ou entre ces processus et leur ancêtre ;

- les **tubes nommés** qui apparaissent avec leur nom comme des éléments de l'arborescence des fichiers et n'ont pas la restriction précédente.

2.4.2/ Tube sans nom

Les tubes sont gérés par le Système de **Gestion de Fichiers** mais diffèrent sur certains points des fichiers ordinaires. Les points suivants sont **communs** au tubes et aux **fichiers ordinaires** :

- Représentation par un i-nœud;

- Manipulation par l'intermédiaire de descripteurs et utilisation des appels système `read, write, dup, close, fstat et fcntl;`

- Le nombre maximum de caractères qu'il est possible de lire ou d'écrire en une fois est égal `PIPE_BUF`, constante définie dans le fichier `limits.h` (souvent 4K ou 8K);

- Deux processus indépendants peuvent accéder à un fichier par l'un de ses noms externes ; **un tube n'ayant pas de non** cela leur est impossible; l'accès à un tube **n'est possible** qu'aux processus partageant le nom local du tube, il s'agit donc nécessairement d'un processus descendants nés après la création du tube;

- Les tubes sont gérés en FIFO (first in first out) : le processus envoie une suite d'octets dans le tube, ces octets sont lus processus lecteur dans l'ordre où ils ont été écrits. Une conséquence, l'appel système `lseek` ne s'applique pas aux tubes.

Synchronisation des lectures et écritures

Définition : On appelle **lecteur** (resp. **écrivain**) tout possédant un descripteur en lecture (resp. écriture) sur un tube.

1. Lecture: si un processus demande la lecture de n caractères dans un tube en contenant m 2: 1, il y a lecture de inf(n, m) carateres.

2. Tube vide: si un processus demande une lecture dans un tube vide pour lequel il existe au moins un écrivain, il est en sommeil jusqu'à ce qu'une écriture ait lieu.

3. Fin de tube : si un processus demande une lecture dans un tube vide pour lequel il n'existe plus d'écrivain, le read retourne 0.

4. Ecriture : si un processus demande l'écriture de n caractères (`n ≤ PIPE_BUF`) dans un tube pour lequel il existe au moins un lecteur, le système garantit que ces `n` caractères seront écrits de façon consécutive dans le tube, même si l'écriture se fait en plusieurs fois.

5. Tube plein : si un processus demande une écriture dans un tube plein pour lequel il existe au moins un lecteur, il est mis en sommeil jusqu'à ce qu'une lecture ait lieu.

6. Tube sans lecteur : si un processus demande une écriture dans un tube pour lequel il n'existe plus de lecteur, le noyau lui envoie le signal `SIGPIPE`, ce qui, par défaut, cause la terminaison du processus écrivain (avec affichage du message `Broken pipe`).

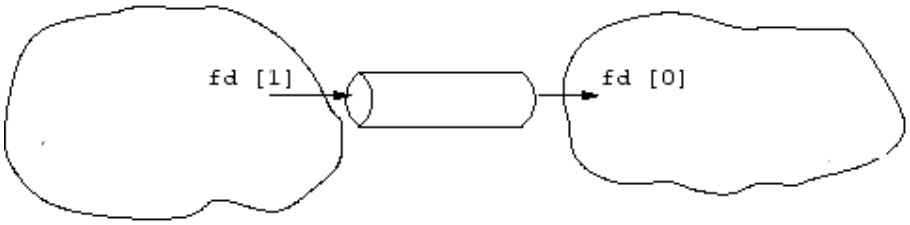
7. Une écriture pourra être suspendue une ou plusieurs fois pour qu'une ou plusieurs lectures libèrent de la place dans le tube.

Création d'un tube sans nom

La création d'un tube sans nom s'effectue par l'appel système `pipe`, déclaré dans le fichier `unistd.h` avec le prototype :

```
int pipe (ind fd[2]) ;
```

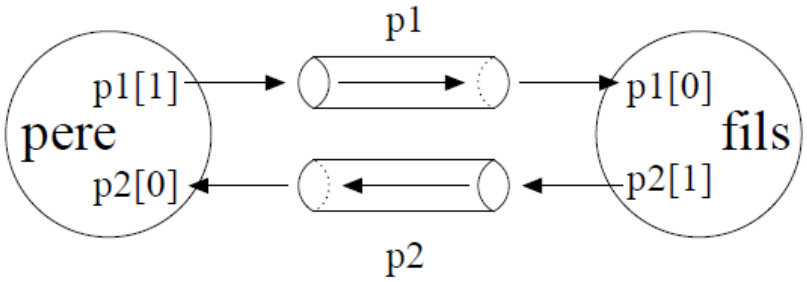
Description : pipe crée deux descripteurs de fichiers, `fd[0]` et `fd[1]`.



`fd[0]` est ouvert en lecture (resp. écriture) sur le tube et `fd[1]` est ouvert en écriture (resp. lecture) sur le tube.

Le retour de pipe est 0 en cas de succès, -1 en cas d'échec.

Exemple 1: On crée deux tubes p1 et p2 pour faire communiquer les deux processus :
– le père a accès en écriture sur p1 et en lecture sur p2,
– le fils a accès en écriture sur p2 et en lecture sur p1.



Le père lit au clavier un entier puis l'envoie via le tube p1 au fils. Le fils le lit, calcule le double puis l'envoie au père via le tube p2 qui l'affiche.

```
// exemple1.c
#define NB_ENTIERS 5

void fils(int p1[2], int p2[2]){
 int i, nombre,nombre1;
 close(p1[1]);      /* Fermeture du tube p1 en ecriture pour le fils */
 close(p2[0]);      /* Fermeture du tube p2 en lecture pour le fils */
   read(p1[0], &nombre, sizeof(int));
                        /* Lecture nombre sur p1 */
   nombre1= nombre * 2;                 /* Calcul du double */
   printf("Le double calculé par le fils = %d\n",nombre1);
   write(p2[1], &nombre1, sizeof(float));
                                /* Ecriture double sur p2 */
   close(p1[0]);    /* Fermeture du tubep1 en ecriture pour lefils. */
   close(p2[1]);    /* Fermeture du tube p2 en lecture pour le fils. */
   exit(0);
}

int main(){
       int i, nombre, nombre_double,
       p1[2],                  /* Descripteurs du tube p1 */
       p2[2];                  /* Descripteurs du tube p2 */
       pipe(p1);               /* Creation du tube p1 */
       pipe(p2);               /* Creation du tube p2 */
       if(fork()==0)           /* Creation du fils */
          fils(p1, p2);        /* Code du fils */
       else {                  /* Code du pere */
            close(p1[0]);
                    /* Fermeture du tube p1 en lecture pour le pere */
            close(p2[1]);
            printf("Entrez un entier\n");     /* Demande d'un entier */
            scanf("%d",&nombre);              /* Saisie d'un entier */
           write(p1[1], &nombre, sizeof(int));
                           /* Ecriture de l'entiersur p1 */
            close(p1[1]);
                 /* Fermeture du tube p1 en ecriture pour le pere  */
            printf("Le double est:\n");
            read(p2[0], &nombre_double, sizeof(int));
                              /*Lecture d'un entier sur p2 */
           printf("%d",nombre_double);      /* Affichage de l'entier */
           printf("\n");      /* Important pour forcer l'affichage */
           close(p2[0]);
                   /*Fermeture du tube p2 en lecture pour le pere */
         }
         return 0;
}
```

Exemple 2 : tube avec exec (simulation de `ps | wc -l`)

```
// exemple2.c   Pere et fils communique par tube
// le fils execute ps, le resultat de ps est communiqué par tube au
pere, le pere execute wc -l
// les entrée/sortie standard sont redirigées (par dup) vers le
tube
#include <stdio.h>
#include <stdlib.h>              /*    declaration de system */
#include <unistd.h>              /* declaration de fork, pipe */
#include <sys/wait.h>
int tube [2] ;
main()
{int i;
if(pipe(tube) == -1) {           /* creation du tube */
   fprintf(stderr, "Ouverture de tube impossible\n");
   exit(1);
}
i = fork();                      /* naissance d'un fils */
if (i == 0) {                    /* processus fils */
        /* redirection de la sortie standard */
        close(1); dup(tube[1]);
        close(tube[0]); close(tube[1]);
        execlp("ps", "ps" , 0);  /* execution de la commande ps */
}
else {                           /* processus pere */
                        /* redirection de l'entree standard */
        close(0); dup(tube[0]);
        close(tube[0]); close(tube[1]);   /* ESSENTIEL */
        execlp("wc", "wc", "-l", 0);      /* execution de we -1 */
      }
}
```

2.4.3/ Tube nomé

Création d'un tube nommé

Un tube nommé (encore appelé fifo) est un tube qui possède un **nom externe** apparaissant dans l'arborescence de fichiers. Il est repérable dans la sortie commande `ls -l` par la lettre `p` en début d'affichage. A la différence tube sans nom, un tube nommé permet de faire **communiquer des processus indépendants.**

La **création d'un tube nommé** se fait, soit de manière externe par la commande shell **`mknod`**, soit par programme par l'appel système de même nom (**`mknod()`**). La commande Shell `mknod` s'emploie avec la syntaxe `mknod fifo p`

Exemple :

```
// exemple3.c
#include <stdio.h>
#include <unistd.h>      /* declarations de read et write  */
#include <fcntl.h>       /* declaration de open  */
#include <string.h>      /* declaration de strcmp  */
#include <sys/types.h>
#include <sys/stat.h>
char commande [50] ;
main (int argc, char *argv[])
{  int desc, i;
   FILE *IN;
   if (strcmp(argv[1], "marchand") == 0) {
        mknod("carnet", 0666 | S_IFIFO, 0);
        IN = fopen("carnet", "r");
        fscanf(IN, "%s", commande);
        printf("Bien recu commande de %s\n", commande);
        exit(0);
   }
   else {                  /* c'est un client  */
                desc = open("carnet", O_WRONLY, 0);
                write(desc, argv[1], strlen(argv[1]));
                exit (0) ;
        }  }
```

Exemple d'exécution : On exécute ce programme dans deux fenêtres de commande :

```
Processus 1 en utilisant le terminal n° 1

./exemple3  marchand
Bien recu commande de frigo
```

```
Processus 2 en utilisant le terminal n° 2

./exemple3     frigo

ll
prw-r--r-- 1 nakechbm cadre    0 oct.  12 16:04 carnet
-rwxr-xr-x 1 nakechbm cadre 8222 oct.  12 16:52 exemple3
-rw-r--r-- 1 nakechbm cadre  763 oct.  12 16:51 exemple3.c
```

Info2 #, Programmation système TP6, communication par tube

Enseignant : M. Nakechbandi

Exercice 1

1. Compiler, exécuter et tester le programme **`exemple1.c`** (page 25)

2. Modifier précédent (enregistrer sous **`ex1_2.c`**) pour que un autre fils (ou un petit fils) participe au calcul en recevant du père le même nombre et calculant le carré l'envoie au père qui l'affiche.

3. Modifier le programme précédent (enregistrer sous **`ex1_3.c)`** pour que le fils après avoir calculé le double, l'envoie à l'autre fils (ou au petit fils) qui calcule le carré puis l'envoie au père qui l'affiche.

Exercice 2

1. Compiler, exécuter et tester le programme **`exemple2.c`** (page 26)

2. Modifier précédent (enregistrer sous **`ex2_2.c`**) pour simuler la commande : `ls -l | wc -l`

3. `Idem,` modifier **`exemple2.c`** (enregistrer sous **`ex2_3.c`**) pour simuler la commande : `ps aux | grep nakech | wc -l` (tester sous corton)

4. Modifier ce programme (enregistrer sous **`ex2_4.c`**) pour que le paramètre "nakech" est sais au clavier.

Ne pas oublier d'envoyer un compte rendu à `nakech@free.fr`
sujet : tp6_1_votre_nom