# Unix : Contrôle du mode d'exécution des échanges

Auteur: Zouheir HAMROUNI

## 5 Contrôle du mode d'exécution des échanges

### 5.1 Introduction

Lorsqu'on ouvre un flot de données (en entrée ou en sortie) sur une ressource (fichier, pipe, socket,…), les primitives d'échange read et write s'exécutent par défaut en mode bloquant. Cela signifie qu'une opération de lecture ou d'écriture se termine seulement lorsque l'opération, si elle n'était pas erronée, a pu être correctement exécutée. Ce qui guarantit à l'appelant que l'opération a été exécutée correctement et complètement (sauf cas d'erreur). Cependant un tel mode de fonctionnement implique un blocage possible du processus appelant jusqu'à ce que l'opération puisse être exécutée :

- Pour une lecture d'une ligne tapée au clavier, il faut attendre qu'une telle ligne ait été validée par l'usager
- Pour une lecture dans un pipe, on reste bloqué dans le read tant que le pipe est vide
- Pour une écriture dans un pipe, on peut rester bloqué dans le write si le pipe est plein

Nous avons vu dans l'exemple 4.7 (un père lecteur et 3 fils rédacteurs connectés par 3 pipes) que les messages envoyés par les 3 fils ne sont pas traités dans le bon ordre, car le père restait bloqué sur la lecture du pipe du fils le plus lent.

Pour éviter cela, il est possible d'envisager un deuxième mode d'exécution évitant tout blocage. Il s'agit donc d'un mode immédiat : le noyau essaie d'exécuter l'opération demandée, et effectue un retour immédiat selon la situation :

- sans avoir réalisé l'échange demandé : en lecture, pas de données disponibles ; en écriture, pas d'espace tampon libre pour écrire ;
- en ayant partiellement réalisé l'opération : en lecture, des données étaient disponibles mais en taille plus faible que demandé ; en écriture, l'espace tampon libre était insuffisant ;
- en ayant complètement réalisé l'échange

### 5.2 La primitive fcntl

`int fcntl (int desc, int cmd, int arg)` permet de lire et de demodifier le mode d'exécution des échanges sur le descripteur desc. Le paramaètre cmd peut prendre les les valeurs suivantes :

- F_DUPFD : dupplique le descripteur desc. la valeur de retour est alors un nouveau descripteur avec les mêmes caractéristiques que desc. Cette valeur est égale à la plus petite des valeurs de descripteurs disponibles plus grandes que arg.
- F_GETFL : la valeur de retour indique en retour l'état du descripteur desc. Les valeurs possibles sont :
  - O_RDONLY
  - O_WRONLY
  - O_RDWR
  - O_NONBLOCK (= O_NDELAY)
- F_SETFL : modifie l'état du descripteur desc, avec la valeur de l'argument arg, qui peut être exprimé sous la forme d'un ou logique entre le mode d'accès (O_RDONLY, O_WRONLY, O_RDWR) et O_NONBLOCK. Si O_NONBLOCK est spécifié dans arg, alors l'accès à desc devient non bloquant.

Pour rendre un descripteur desc non bloquant, sans modifier les autres éléments de son état, il suffit d'appeler fcnl de la façon suivante :

`fcntl(desc, F_SETFL, fcntl(desc, F_GETFL) | O_NONBLOCK);`

Le paramètre arg est défini comme un ou logique entre l'état précédent de desc (obtenu avec l'appel `fcntl(desc, F_GETFL)` et O_NOBLOCK ; ce qui permet d'ajouter le mode non bloquant à l'état précédent.

En mode non bloquant, read renvoie :

- -1 avec la variable errno = EAGAIN lorsqu'aucune donnée n'est disponible en lecture
- 0 lorsqu'on ne peut plus lire (pipe fermé en écriture)
- le nombre d'octets lus si des données étaient disponibles

On reprend le code de l'étape 4.7, et on y apporte les modifications suivantes :

- on met les extrémités 0 des 3 pipes en mode non bloquant
- on encapsule les de lecture sur 3 les pipes dans une boucle répéter, dans laquelle on reste jusqu'à ce que tous les pipes aient été fermés en écriture (retour du read = 0)
- on utilise un tableau pour enregistrer l'état des pipes, et une variable pour maintenir à jour le nombre de pipes fermés

Ce qui donne le codes suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 pipe depuis chaque fils vers le père, en mode non bloquant fcntl */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <string.h>   /* opérations sur les chaines */
#include <fcntl.h>    /* fcntl */
#include <errno.h>

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, pid, nb_pipes_fermes ;

    int pipe_f2p[NB_FILS+1][2] ;    /* pipes pour communiquer depuis les fils vers le père */
    int pipe_ferme[NB_FILS+1] ;

    /* Créet NB_FILS pipes */
    for (int i = 1 ; i <= NB_FILS ; i++) {
        retour = pipe(pipe_f2p[i]) ;
        if (retour == -1) {   /* échec du pipe */
            printf("Erreur pipe\n") ;
            exit(1) ;
        }
    }

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    /* creer NB_FILS */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
```

```
44          if (retour == 0) {
45
46              /* fermer l'extrémité 0 : le fils va écrire dans le pipe */
47              close(pipe_f2p[fils][0]) ;
48              /* fermer les extrémités de tous les autres pipes */
49              for (int i = 1 ; i <= NB_FILS ; i++) {
50                  if (i!=fils) {
51                      close(pipe_f2p[i][0]) ;
52                      close(pipe_f2p[i][1]) ;
53                  }
54              }
55
56              pid = getpid() ;
57              /* écrire 5 fois son pid dans le pipe */
58              for (int i = 1 ; i <= 5 ; i++) {
59                  sleep(2 * (NB_FILS - fils)) ;
60                  printf("    Processus de pid %d effectue un envoi\n", pid) ;
61                  write(pipe_f2p[fils][1], &pid, sizeof(int)) ;
62              }
63
64              /* fermer l'extrémité 1 : fin des envois */
65              close(pipe_f2p[fils][1]) ;
66
67              /* Important : terminer un processus par exit */
68              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
69          }
70
71          /* pere */
72          else {
73              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
74                      getpid(), fils, retour) ;
75          }
76      }
77
78      /* fermer l'extrémité 1 de tous les pipes : le père est lecteur */
79      for (int i = 1 ; i <= NB_FILS ; i++) {
80          close(pipe_f2p[i][1]) ;
81      }
82
83      sleep(1) ;
84      /* mettre les extrémités 0 des pipes en mode non boquant */
85      for (fils = 1 ; fils <= NB_FILS ; fils++) {
86          retour = fcntl(pipe_f2p[fils][0], F_SETFL, fcntl(pipe_f2p[fils][0], F_GETFL) | O_NONBLOCK) ;
87          printf("Processus principal - Echec fcntl\n") ;
88          pipe_ferme[fils] = 0 ;
89      }
90
91      nb_pipes_fermes = 0 ;
92      /* lire ce que les fls envoient */
93      do {
94          for (fils = 1 ; fils <= NB_FILS ; fils++) {
95              if (!pipe_ferme[fils]) {
96                  while ((retour = read(pipe_f2p[fils][0], &pid, sizeof(int))) > 0) {
97                      printf("\nProcessus Principal - recu : %d\n", pid) ;
98                  }
99                  if (retour == 0) {   /* pipe fermé en écriture */
100                     pipe_ferme[fils] = 1 ;
101                     nb_pipes_fermes++ ;
102                     printf("Processus Principal - Fils %d a fermé son pipe\n", fils) ;
103                 }
104                 /* cas d'erreur */
105                 else if (!(retour == -1 &&&& errno == EAGAIN)) {
106                     printf("Processus Principal - Erreur sur lecture pipe du fils %d : \n", fils) ;
107                 }
108                 sleep(1) ;
109             }
110         }
111     } while (nb_pipes_fermes < NB_FILS) ;
112
113     /* fermer l'extrémité 0 de tous les pipes */
114     for (int i = 1 ; i <= NB_FILS ; i++) {
115         close(pipe_f2p[i][0]) ;
116     }
117     printf("\nProcessus Principal termine.\n") ;
```

```
118     return EXIT_SUCCESS ;
119 }
```

A l'exécution, on observe bien que les messages sont lus dans le même ordre chronologique d'écriture, et que le processus principal s'arrête bien après que tous ses fils aient arrêté de transmettre.

## 5.3 Synchronisation avec SIGALRM

Dans le code précédent, le porcessus père (lecteur) gérait la lectures des données dans une boucle d'attente active. Mais cette solution n'est pas pratique si ce processus a d'autres traitements à faire. On peut imaginer une solution basée sur une lecture par scrutation régulière :

- le processus père découpe son temps en périodes
- à la fin de chaque période, il va vérifier les données éventuellement arrivées dans les pipes.

Cette synchronisation peut se faire en utilisant la primitive alarm et le signal SIGALRM, et la lecture des données peut être faites toutes les n secondes dans le handler assoicé au signal SIGALRM.

On modifie le code précédent, en :

- en créant un handler pour le signal SIGALRM
- déplaçant la boucle de lecture sur les 3 pipes dans le handler du signal SIGALRM
- en découpant la boucle du processus principal en périodes de 2 secondes (alarm(2)), au bout desquelles le signal SIGARLM va activer le handler, et par conséquent la lecture des données éventuellement arrivées dans les pipes.

Cette solution présente quand même un inconvénient assez important : la férquence des lectures peut être mal adaptée à celle des écritures :

- si la période de scrutation est trop longue, les lectures subissent un décalage important par rapport aux écritures
- Si la période de scrutation est trop faible, plusieurs entrées dans le handler peuvent être inutiles

Cela donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils   */
/* 1 pipe depuis chaque fils vers le père, en mode non bloquant fcntl */
/* Scrutation réguière des pipes avec alarm et SIGALRM                */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <string.h>   /* opérations sur les chaines */
#include <fcntl.h>    /* fcntl */
#include <errno.h>

#define NB_FILS 3     /* nombre de fils */
#define DALARM 3      /* délai d'alarme */

/* Variables globales car utilisées par le handler */
int nb_pipes_fermes ;
int pipe_ferme[NB_FILS+1] ;
int pipe_f2p[NB_FILS+1][2] ; /* pipes pour communiquer depuis les fils vers le père */

/* Traitant du signal SIGALRM */
/* Lecture des données envoyées par les fils */
void handler_sigalrm(int signal_num) {
    int retour, pid, fils ;
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        if (!pipe_ferme[fils]) {
            while ((retour = read(pipe_f2p[fils][0], &pid, sizeof(int))) > 0) {
```

```
28              printf("\nProcessus Principal - recu : %d\n",pid) ;
29          }
30          if (retour == 0) {   /* pipe fermé en écriture */
31              pipe_ferme[fils] = 1 ;
32              nb_pipes_fermes++ ;
33              printf("Processus Principal - Fils %d a fermé son pipe\n", fils) ;
34          }
35          /* cas d'erreur */
36          else if (!(retour == -1 &&&& errno == EAGAIN)) {
37              printf("Processus Principal - Erreur sur lecture pipe du fils %d : \n", fils) ;
38          }
39      }
40  }
41  /* réarmer l'alame */
42  alarm(DALARM) ;
43  return ;
44 }
45
46 int main()
47 {
48      int fils, retour, pid ;
49
50      /* Créet NB_FILS pipes */
51      for (int i = 1 ; i <= NB_FILS ; i++) {
52          retour = pipe(pipe_f2p[i]) ;
53          if (retour == -1) {   /* échec du pipe */
54              printf("Erreur pipe\n") ;
55              exit(1) ;
56          }
57      }
58
59      /* associer un traitant au signal SIGALRM */
60      signal(SIGALRM, handler_sigalrm) ;
61
62      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
63
64      /* creer NB_FILS */
65      for (fils = 1 ; fils <= NB_FILS ; fils++) {
66          retour = fork() ;
67
68          /* Bonne pratique : tester systématiquement le retour des appels système */
69          if (retour < 0) {   /* échec du fork */
70              printf("Erreur fork\n") ;
71              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
72              exit(1) ;
73          }
74
75          /* fils */
76          if (retour == 0) {
77
78              /* fermer l'extrémité 0 : le fils va écrire dans le pipe */
79              close(pipe_f2p[fils][0]) ;
80              /* fermer les extrémités de tous les autres pipes */
81              for (int i = 1 ; i <= NB_FILS ; i++) {
82                  if (i!=fils) {
83                      close(pipe_f2p[i][0]) ;
84                      close(pipe_f2p[i][1]) ;
85                  }
86              }
87
88              pid = getpid() ;
89              /* écrire 5 fois son pid dans le pipe */
90              for (int i = 1 ; i <= 5 ; i++) {
91                  sleep(2 * (NB_FILS - fils)) ;
92                  printf("    Processus de pid %d effectue un envoi\n", pid) ;
93                  write(pipe_f2p[fils][1], &pid, sizeof(int)) ;
94              }
95
96              /* fermer l'extrémité 1 : fin des envois */
97              close(pipe_f2p[fils][1]) ;
98
99              /* Important : terminer un processus par exit */
100             exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
101         }
```

```
102
103         /* pere */
104         else {
105             printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
106                     getpid(), fils, retour) ;
107         }
108     }
109
110     /* fermer l'extrémité 1 de tous les pipes : le père est lecteur */
111     for (int i = 1 ; i <= NB_FILS ; i++) {
112         close(pipe_f2p[i][1]) ;
113     }
114
115     /* mettre les extrémités 0 des pipes en mode non boquant */
116     for (fils = 1 ; fils <= NB_FILS ; fils++) {
117         retour = fcntl(pipe_f2p[fils][0], F_SETFL, fcntl(pipe_f2p[fils][0], F_GETFL) | O_NONBLOCK) ;
118         pipe_ferme[fils] = 0 ;
119         if (retour == -1 ) {
120             printf("Processus principal - Echec fcntl\n") ;
121         }
122     }
123
124     nb_pipes_fermes = 0 ;
125     alarm(DALARM) ;
126     /* Faire ce qu'on a faire sans se soucier des lectures dans les pipes */
127     do {
128         /* attente d'un signal : peut être remplacé par tout autre traitement */
129         pause() ;
130     } while (nb_pipes_fermes < NB_FILS) ;
131
132     /* fermer l'extrémité 0 de tous les pipes */
133     for (int i = 1 ; i <= NB_FILS ; i++) {
134         close(pipe_f2p[i][0]) ;
135     }
136     printf("\nProcessus Principal termine.\n") ;
137     return EXIT_SUCCESS ;
138 }
```

## 5.4 Synchronisation avec SIGUSR1

La solution précédente (scrutation régulière) présente un inconvénient assez important : la férquence des lectures peut être mal adaptée à celle des écritures

- si la période de scrutation est trop longue, les lectures subissent un décalage important par rapport aux écritures
- Si la période de scrutation est trop faible, plusieurs entrées dans le handler seront inutiles

Pour traiter la donnée arrivée sur un pipe imméditament après son envoi par le fils, on peut en visager d'en informer le lecteur dès que le rédacteur ait fini d'écrire. Ceci peut être fait en lui envoyant un signal, par eexemple le signal SIGUSR1 ou SIGUSR2 prévus pour l'usage de l'utilisateur.

On transforme légèrement le code précédent :

- le handler de SIGALRM n'est pas modifié et devient handler pour SIGUSR1
- le fils envoie le signal SIGUSR1 à son père à la fin de chaque écriture dans le pipe
- le père n'a plus à synchronier son temps avac alarm. Il se consacre à ses traitements, et ne sera interrompu qu'à la réception du signal SIGUSR1 lui indiquant l'arrivée d'une donnée sur l'un des pipes.

```
1 /* Exemple d'illustration des primitives Unix : Un père et ses fils   */
2 /* 1 pipe depuis chaque fils vers le père, en mode non bloquant fcntl */
3 /* SIGUSR1 envoyé par le fils après chaque écriture dans le pipe      */
4
5 #include <stdio.h>    /* entrées sorties */
6 #include <unistd.h>   /* pimitives de base : fork, ...*/
7 #include <stdlib.h>   /* exit */
8 #include <sys/wait.h> /* wait */
```

```c
#include <string.h>   /* opérations sur les chaines */
#include <fcntl.h>    /* fcntl */
#include <errno.h>

#define NB_FILS 3     /* nombre de fils */

/* Variables globales car utilisées par le handler */
int nb_pipes_fermes ;
int pipe_ferme[NB_FILS+1] ;
int pipe_f2p[NB_FILS+1][2] ; /* pipes pour communiquer depuis les fils vers le père */

/* Traitant du signal SIGUSR1 : activé par l'envoi di SIGUSR1 par un fil */
/* Lecture des données envoyées par les fils */
void handler_sigusr1(int signal_num) {
    int retour, pid, fils ;
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        if (!pipe_ferme[fils]) {
            while ((retour = read(pipe_f2p[fils][0], &pid, sizeof(int))) > 0) {
                printf("\nProcessus Principal - recu : %d\n",pid) ;
            }
            if (retour == 0) {   /* pipe fermé en écriture */
                pipe_ferme[fils] = 1 ;
                nb_pipes_fermes++ ;
                printf("Processus Principal - Fils %d a fermé son pipe\n", fils) ;
            }
            /* cas d'erreur */
            else if (!(retour == -1 &&&& errno == EAGAIN)) {
                printf("Processus Principal - Erreur sur lecture pipe du fils %d : \n", fils) ;
            }
        }
    }
    return ;
}

int main()
{
    int fils, retour, pid ;

    /* Créet NB_FILS pipes */
    for (int i = 1 ; i <= NB_FILS ; i++) {
        retour = pipe(pipe_f2p[i]) ;
        if (retour == -1) {   /* échec du pipe */
            printf("Erreur pipe\n") ;
            exit(1) ;
        }
    }

    /* associer un traitant au signal SIGUSR1 */
    signal(SIGUSR1, handler_sigusr1) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    /* creer NB_FILS */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {

            /* fermer l'extrémité 0 : le fils va écrire dans le pipe */
            close(pipe_f2p[fils][0]) ;
            /* fermer les extrémités de tous les autres pipes */
            for (int i = 1 ; i <= NB_FILS ; i++) {
                if (i!=fils) {
                    close(pipe_f2p[i][0]) ;
                    close(pipe_f2p[i][1]) ;
                }
```

```
 83             }
 84
 85             pid = getpid() ;
 86             /* écrire 5 fois son pid dans le pipe */
 87             for (int i = 1 ; i <= 5 ; i++) {
 88                 sleep(2 * (NB_FILS - fils)) ;
 89                 printf("    Processus de pid %d effectue un envoi\n", pid) ;
 90                 write(pipe_f2p[fils][1], &pid, sizeof(int)) ;
 91                 kill(getppid(), SIGUSR1) ;
 92             }
 93
 94             /* fermer l'extrémité 1 : fin des envois */
 95             close(pipe_f2p[fils][1]) ;
 96
 97             /* Important : terminer un processus par exit */
 98             exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
 99         }
100
101         /* pere */
102         else {
103             printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
104                     getpid(), fils, retour) ;
105         }
106     }
107
108     /* fermer l'extrémité 1 de tous les pipes : le père est lecteur */
109     for (int i = 1 ; i <= NB_FILS ; i++) {
110         close(pipe_f2p[i][1]) ;
111     }
112
113     /* mettre les extrémités 0 des pipes en mode non boquant */
114     for (fils = 1 ; fils <= NB_FILS ; fils++) {
115         retour = fcntl(pipe_f2p[fils][0], F_SETFL, fcntl(pipe_f2p[fils][0], F_GETFL) | O_NONBLOCK) ;
116         pipe_ferme[fils] = 0 ;
117         if (retour == -1 ) {
118             printf("Processus principal - Echec fcntl\n") ;
119         }
120     }
121
122     nb_pipes_fermes = 0 ;
123     /* Faire ce qu'on a faire sans se soucier des lectures dans les pipes */
124     do {
125         /* attente d'un signal : peut ête remplacé par tout autre traitement */
126         pause() ;
127     } while (nb_pipes_fermes < NB_FILS) ;
128
129     /* fermer l'extrémité 0 de tous les pipes */
130     for (int i = 1 ; i <= NB_FILS ; i++) {
131         close(pipe_f2p[i][0]) ;
132     }
133     printf("\nProcessus Principal termine.\n") ;
134     return EXIT_SUCCESS ;
135 }
```

A l'exécution, on peut vérifier que la lecture des données arriavnt dans les pipes est plus régulière, et est bien synchronisée par rapport aux envois.

## 5.5 La primitive select

Une autre manière de rendre les lectures (et les écritures) non boquantes consiste à utiliser la primitive select :

`int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`

permet de surveiller plusieurs descripteurs, attendant qu'au moins l'un des descripteurs soit prêt pour certaines classes d'opérations d'entrées-sorties (lecture, écriture). Un descripteur de fichier est considéré comme prêt s'il est possible d'effectuer l'opération d'entrées-sorties correspondante (par exemple, un read) sans bloquer.

- Le type fd_set décrit un tableau descripteurs. Différentes macros permettent de les manipuler facilement :
  - FD_ZERO (&fdset) : initialise un ensemble de descripteurs fdset à vide
  - FD_SET (fd, &fdset) : positionne le descripteur fd dans l'ensemble fdset
  - FD_CLR (fd, &fdset) : supprime le descripteur fd de l'ensemble fdset
  - FD_ISSET (fd, &fdset) : retoure une valeur non nulle si fd est dans fdset, 0 sinon
- readfs représente l'ensemble des descripteurs à surveiller pour la lecture (caractères disponibles en lecture)
- writefds représente l'ensemble des descripteurs à surveiller pour l'écriture (écriture possible)
- exceptfds représente l'ensemble des descripteurs à surveiller pour les exceptions
- nfds indique le numéro du plus grand descripteur des 3 ensembles, plus 1
- timeout indique la limite supérieure du temps passé dans select() avant son retour.
  - Elle se compose de deux champs : tv_sec (nombre de secondes) et tv_usec ( nombre de microsecondes)
  - Elle peut être nulle, ce qui conduit select() à revenir immédiatement.
- La valeur de retour indique le nombre total de descripteurs prêts

- Les ensembles readfds, writefds, et exceptfds contiennet les descripteurs détectés prêts, et que l'on peut reconnaitre en utilisant la macro FD_ISSET
- Si la valeur de retour est égale à −1 alors une erreur a été détectée par select. Dans ce cas le contenu des descripteurs en sortie n'est pas interprétable. La variable errno décrit alors la nature de l'erreur et contient la dernière erreur système s'étant produite (le message associé peut être imprimé en utilisant perror)

- errno = EBADF : un des descripteurs positionnés dans un des ensembles de descripteurs est invalide
- errno = EFAULT : erreur dans le passage des arguments du select
- errno = EINTR : select interrompu par l'arrivée d'un signal
- errno = EINVAL : valeur de timeout non correcte

ATTENTION : l'appel à select modifie les ensembles readfds, writefds, et exceptfds; ainsi que le timeout. Il faut donc les réinitialiser avant chaque nouvel appel à select.

On reprend le code 5.2 et on remplace l'utilisation du fcntl par select, en :

- utilisant un ensemble readfds qui contient les extrémités de lecture des pipes qui sont actifs en écriture (côté fils)
- et en allant vérifier régulièrement les descripteurs prets en lecture (données disponibles)

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 pipe de chaque fils vers le père - lecture non bloquante avec select*/

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <string.h>   /* opérations sur les chaines */
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, pid, nlu, nb_pipes_fermes ;
    int pipe_ferme[NB_FILS+1] ;
    int pipe_f2p[NB_FILS+1][2] ;    /* pipes pour communiquer depuis les fils vers le père */

    struct timeval timeout ;       /* temps du select */
    fd_set readfds ;              /* ensemble de descripteurs à lire */


```

```
25      /* Créet NB_FILS pipes */
26      for (int i = 1 ; i <= NB_FILS ; i++) {
27          retour = pipe(pipe_f2p[i]) ;
28          if (retour == -1) {   /* échec du pipe */
29              printf("Erreur pipe\n") ;
30              exit(1) ;
31          }
32      }
33
34      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
35
36      for (fils = 1 ; fils <= NB_FILS ; fils++) {
37          retour = fork() ;
38
39          /* Bonne pratique : tester systématiquement le retour des appels système */
40          if (retour < 0) {   /* échec du fork */
41              printf("Erreur fork\n") ;
42              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
43              exit(1) ;
44          }
45
46          /* fils */
47          if (retour == 0) {
48
49              /* fermer l'extrémité 0 : le fils va écrire dans le pipe */
50              close(pipe_f2p[fils][0]) ;
51              /* fermer les extrémités de tous les autres pipes */
52              for (int i = 1 ; i <= NB_FILS ; i++) {
53                  if (i!=fils) {
54                      close(pipe_f2p[i][0]) ;
55                      close(pipe_f2p[i][1]) ;
56                  }
57              }
58
59              pid = getpid() ;
60
61              /* écrire 5 fois son pid dans le pipe */
62              for (int i = 1 ; i <= 5 ; i++) {
63                  sleep(2 * (NB_FILS - fils)) ;
64                  printf("    Processus de pid %d effectue un envoi\n", pid) ;
65                  write(pipe_f2p[fils][1], &pid, sizeof(int)) ;
66              }
67
68              /* fermer l'extrémité 1 : nb_pipes_fermes des envois */
69              close(pipe_f2p[fils][1]) ;
70
71              /* Important : terminer un processus par exit */
72              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
73          }
74
75          /* pere */
76          else {
77              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
78                      getpid(), fils, retour) ;
79          }
80      }
81
82      /* fermer l'extrémité 1 de tous les pipes : le père est lecteur */
83      for (fils = 1 ; fils <= NB_FILS ; fils++) {
84          close(pipe_f2p[fils][1]) ;
85          pipe_ferme[fils] = 0 ;
86      }
87
88      /* initialiser les varaibles de gestion des échanges */
89      nb_pipes_fermes = 0 ;
90      for (fils = 1 ; fils <= NB_FILS ; fils++) {
91          pipe_ferme[fils] = 0 ;
92      }
93
94      /* lire ce que les fls envoient */
95      do {
96          /* initialiser l'ensemble des descripteurs a ecouter */
97          FD_ZERO(&readfds) ;
98          for (fils = 1 ; fils <= NB_FILS ; fils++) {
```

```
 99             if (pipe_ferme[fils] == 0) {
100                 FD_SET(pipe_f2p[fils][0], &readfds) ;
101             }
102         }
103         timerclear(&timeout) ;
104         timeout.tv_sec = 1 ;
105         /* vérifier si un des descripteurs de readfds est pret */
106         retour = select(FD_SETSIZE, &readfds, NULL, NULL, &timeout) ;
107         switch(retour) {
108             case -1: /* erreur */
109                 perror("select ") ;
110             case 0: /* rien a lire */
111                 break ;
112             default:
113                 /* au moins un descripteur est pret */
114                 for (fils = 1 ; fils <= NB_FILS ; fils++) {
115                     /* lire le descripteur pret */
116                     if (FD_ISSET(pipe_f2p[fils][0], &readfds)) {
117                         nlu = read(pipe_f2p[fils][0], &pid, sizeof(int)) ;
118                         /* donnee valide */
119                         if (nlu > 0) {
120                             printf("\nProcessus Principal - recu : %d\n", pid) ;
121                         }
122                         /* pipe ferme - fin de fichier */
123                         else {
124                             pipe_ferme[fils] = 1 ;
125                             nb_pipes_fermes++ ;
126                             printf("Processus Principal - Fils %d a fermé son pipe\n", fils) ;
127                         }
128 
129                     }
130                 }
131         }
132         sleep(2) ;
133     } while (nb_pipes_fermes < NB_FILS) ;
134 
135     /* fermer l'extrémité 0 de tous les pipes */
136     for (int i = 1 ; i <= NB_FILS ; i++) {
137         close(pipe_f2p[i][0]) ;
138     }
139     printf("\nProcessus Principal termine.\n") ;
140     return EXIT_SUCCESS ;
141 }
```

Même si les deux primitives (fcntl et select) nous permettent de ne pas rester bloqué en lecture sur un descripteur non prêt, leurs comportements présentent quelques petites différences :

- fcntl rend le read non bloquant, alors que le select ne modifie pas le mode du read, qui reste toujours bloquant. Donc, attention à ne pas exécuter plusieurs read sur un descripteur indiqué comme prêt par le select.
- select permet de faire durer l'écoute pendand le délai exprimé dans timeout, alors que le read non bloquant grâce au fcntl génère un retour immédiat

Les mêmes améliorations de la synchronisation par SIGALRM ou SIGUSR1 peuvent être apportés à ce code.