Unix : Communication par tubes

Auteur: Zouheir HAMROUNI

4 Les tubes (pipes)

4.1 Introduction

Les tubes (pipes) permettent de faire communiquer des processus selon le schéma générique du producteur-consommateur : un processus écrit (rédacteur), l'autre lit (lecteur). Un tube se comporte comme une file (FIFO) de capacité limitée (quelques dizaines de kilo-octets) où les informations sont introduites à une extrémité et sont extraites à une autre. La communication dans un tube est unidirectionnelle.

Un tube est représenté par un tableau de 2 descripteurs (entiers), par exemple desc[2] :

- desc[1] : descripeur d'écrire dans le tube (avec la primitive write)
- desc[0] : descripteur de lecture dans le tube (avec la primitive read)

La capacité d'un tube est limitée (quelques kilo-octets) :

- Si l'on tente d'écrire dans un tube plein, le processus rédacteur reste bloqué tant que le tube est plein
- Si l'on tente d'écrire dans un tube dont l'extérmité de lecture est fermée (aucun processus n'est prêt à lire), le signal SIGPIPE est envoyé au processus rédacteur, et la primitive write renvoie -1 (erreur)
- Si un processus lit dans un tube vide et si l'extrémité d'écriture est toujours ouverte, le lecteur reste bloqué tant que le tube reste vide
- Si l'on tente de lire dans un tube dont l'extérmité d'écrire est fermée (aucun processus n'est capable d'écrire), la primitive read renvoie 0

Comme pour les autres descripteurs, il y a héritage des tubes créés lors de l'exécution de la primitive fork (et conservation lors de exec). Deux processus pourront donc communiquer via un tube si ce tube a été créé par un ancêtre commun.

4.2 La primitive pipe

int pipe (int pdesc[2]) permet de créer un pipe

- en cas d'erreur le retour est égal à -1
- en cas de succès, le retour est égal à 0 :
  - pdesc[0] est la sortie du tube, i.e. le numéro du descripteur pour la lecture
  - pdesc[1] est l'entrée du tube, i.e. le numéro du descripteur pour l'écriture

Pour que deux processus puissent communiquer via un tube, il faut que ce dernier ait été créé par un ancêtre commun :

- un père et un fils peuvent communiquer via un tube créé par le père avant la création du fils (tube hérité par le fils)
- deux fils peuvent communiquer via un tube créé par leur père avant la création des deux fils (tube hérité par les 2 fils)
- deux processus peuvent communiquer via un tube créé par l'un et hérité par l'autre, ou hérité par les deux d'un ancêtre commun

L'extrémité 1 (écriture) du tube ne doit rester ouverte que dans les processus rédacteurs (plusieurs processus peuvent écrire dans le même tube). Autrement, le processus lecteur peut rester bloqué dans une opération de lecture (extérmité 1 ouverte sans rédacteur).

L'extrémité 0 (lecture) du tube ne doit rester ouverte que dans le processus lecteur. Autrement, tout processus rédacteur peut voir ses données saturer le tube (sans être consommées), et risquer de rester bloqué dans une opération d'écriture.

Même s'il est possible d'avoir plusieurs lecteurs, ce cas est difficile à gérer, car il est compliqué de synchroniser les opérations de lecture et de savoir qui lit quoi (toute lecture consomme les données présentes dans le pipe (en partie ou en totalité).

Pour illustrer le fonctionnement des pipes, on reprend le code du père qui crée trois fils, auquel on ajoute la création d'un pipe par le père avant la création de ses trois fils. Les fils seront rédacteurs (enverront des données), et le père lecteur.

Il est important de noter que :

- le père crée le pipe avant de créer ses trois fils, de manière à ce qu'ils puissent en hériter et l'utiliser
- l'on traite le retour d'échec du pipe (cela peut arriver)
- chaque fils ferme l'extrémité 0 du pipe, car il sera rédacteur
- le père ferme l'extrémité 1 du pipe (lecteur) après la création des fils. Car s'il le faisait avant, cette extrémité serait aussi fermée dans les fils.

Dans cet exemple, le fils envoie 2 fois son pid sous sa forme binaire (4 octets) sans aucune transformation. Les délais d'attente sont choisis de manière à assurer un ordre croissant pour le premier envoi (fils 1, fils 2, fils3), et décroissant pour le second (fils3, fils 2, fils1). Cela donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 pipe : depuis les fils vers le père */

#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 <signal.h>

#define NB_FILS 3     /* nombre de fils */

void handler_sigchld(int num_signal) {
   int wstatus, fils_termine ;
   /* prendre connaissance de la fin des fils */
   while ((fils_termine = (int) waitpid(-1, &wstatus, WNOHANG)) > 0) {
      printf("\nMon fils %d est arrete\n", fils_termine) ;
   }
}

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

    int pipe_f2p[2] ;    /* pipe pour communiquer depuis les fils vers le père* */

   signal(SIGCHLD, handler_sigchld) ;

    retour = pipe(pipe_f2p) ;
    /* Bonne pratique : tester systématiquement le retour des appels système */
    if (retour == -1) {   /* échec du pipe */
        printf("Erreur pipe\n") ;
        /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
        exit(1) ;
    }

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

    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[0]) ;
            pid = getpid() ;
            printf("\n    Processus de pid %d, de pere %d\n", pid, getppid()) ;

            /* écrire 2 fois son pid dans le pipe */
            sleep(fils) ;
            write(pipe_f2p[1], &pid, sizeof(int)) ;
            sleep(2 * (NB_FILS - fils)) ;
            write(pipe_f2p[1], &pid, sizeof(int)) ;
            /* fermer l'extrémité 1 : fin des envois */
            /* la prochaine lecture sur ce pipe renverra 0 */
            close(pipe_f2p[1]) ;

            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }

    /* fermer l'extrémité 1 : le père va lire dans le pipe */
```

```
77      /* fermer l'extrémité 1 : le père va lire dans le pipe */
78      close(pipe_f2p[1]) ;
79
80      /* lire ce qui arrive dans le pipe entier par entier  */
81      while ((read(pipe_f2p[0], &v_lue, sizeof(int)) > 0)) {
82          printf("\nProcessus Principal - recu : %d\n", v_lue) ;
83      }
84      /* fermer l'extrémité 0 */
85      close(pipe_f2p[0]) ;
86      printf("\nProcessus Principal termine.\n") ;
87      return EXIT_SUCCESS ;
88  }
```

A l'exécution, on vérifie bien que la lecture dans le pipe se fait bien selon le modèle FIFO. Le père lit le premier envoi des 3 fils dans un ordre croissant, et le second envoi dans un ordre décroissant (même ordre d'envoi).

Lorsque les données transmises sont constituées de texte ou hétérogènes, la communication se fait en mode cacarctère. Dans le code suivant, chaque fils enrichit les données qu'il envoie avec des messages texte dans lesqels est intégré le pid (transformé en chaine de caractères).

```
 1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
 2  /* 1 pipe : depuis les fils vers le père */
 3
 4  #include <stdio.h>     /* entrées sorties */
 5  #include <unistd.h>    /* pimitives de base : fork, ...*/
 6  #include <stdlib.h>    /* exit */
 7  #include <sys/wait.h>  /* wait */
 8  #include <string.h>    /* opérations sur les chaines */
 9
10  #define NB_FILS 3      /* nombre de fils */
11
12  void handler_sigchld(int num_signal) {
13     int wstatus, fils_termine ;
14     /* prendre connaissance de la fin des fils */
15     while ((fils_termine = (int) waitpid(-1, &wstatus, WNOHANG)) > 0) {
16        printf("\nMon fils %d est arrete\n", fils_termine) ;
17     }
18  }
19
20  int main()
21  {
22      int fils, retour, pid ;
23
24      int pipe_f2p[2] ;     /* pipe pour échanger depuis les fils vers le père* */
25      char message[64] ;
26
27     signal(SIGCHLD, handler_sigchld) ;
28
29      retour = pipe(pipe_f2p) ;
30      /* Bonne pratique : tester systématiquement le retour des appels système */
31      if (retour == -1) {    /* échec du pipe */
32          printf("Erreur pipe\n") ;
33          /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
34          exit(1) ;
35      }
36
37      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
38
39      for (fils = 1 ; fils <= NB_FILS ; fils++) {
40          retour = fork() ;
41
42          /* Bonne pratique : tester systématiquement le retour des appels système
43          if (retour < 0) {    /* échec du fork */
44              printf("Erreur fork\n") ;
45              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
46              exit(1) ;
47          }
48
49          /* fils */
50          if (retour == 0) {
51
52              /* fermer l'extrémité 0 : le fils va écrire dans le pipe */
53              close(pipe_f2p[0]) ;
54              pid = getpid() ;
55              printf("\n    Processus de pid %d, de pere %d\n", pid, getppid()) ;
56
57              /* écrire 2 messages dans le pipe */
58              sleep(fils) ;
59              bzero(message, sizeof(message)) ;    /* vider le buffer */
60              sprintf(message, "Le bonjour du fils %d", fils) ;
61              write(pipe_f2p[1], message, strlen(message)) ;
62
63              sleep(2 * (NB_FILS - fils)) ;
64              bzero(message, sizeof(message)) ;
65              sprintf(message, "Au revoir de la part du fils %d", fils) ;
```

```
            sprintf(message, "Au revoir de la part du fils %d", fils) ;
            write(pipe_f2p[1], message, strlen(message)) ;
            /* fermer l'extrémité 1 : fin des envois */
            close(pipe_f2p[1]) ;

            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }

    /* fermer l'extrémité 1 : le père va lire dans le pipe */
    close(pipe_f2p[1]) ;

    /* lire ce qui arrive dans le pipe */

    /* vider le buffer de lecture pour pouvoir traiter le bloc lu comme une */
    /* chaine de caractères - se terminant par 0 - */
    /* mais on peut aussi ajouter un 0 à la fin du bloc dont la taille est */
    /* connue comme résultat du read */
    bzero(message, sizeof(message)) ;
    while (read(pipe_f2p[0], message, sizeof(message)) > 0) {
        printf("\nProcessus Principal - recu : %s\n", message) ;
        bzero(message, sizeof(message)) ;
    }
    /* fermer l'extrémité 0 */
    close(pipe_f2p[0]) ;
    printf("\nProcessus Principal termine.\n") ;
    return EXIT_SUCCESS ;
}
```

4.3 blocage en lecture

Comme indiqué plus haut, il est primordial que :

- l'extrémité 1 du pipe soit fermée dans tous les processus non rédacteurs
- l'extrémité 0 du pipe soit fermée dans tous les processus non lecteurs

Autrement, on peut se retrouver dans des situations de blocage :

- le processus rédacteur qui risque de saturer le pipe et rester bloqué en écriture, car l'extrémité 0 du pipe est ouverte sans lecteur potentiel
- le processus lecteur risque de rester bloqué en lecture, car l'extrémité 1 du pipe est ouverte sans rédacteur potentiel

Ce second cas peut être mis en évidence en mettant en commentaire l'instruction qui ferme l'extrémité 1 du pipe dans le processus père. Ce dernier se retrouve lecteur et rédacteur en même temps, et après lecture des messages de ses fils, il reste bloqué dans la primitive de lecture car aucune nouvelle écriture ne se fait dans le pipe, alors que l'extrémité 1 reste toujours ouverte (dans le père).

4.4 blocage en écriture

Lorsqu'un processus écrit dans un pipe dont l'extrémité 0 (lecture) est ouverte, mais sans aucun lecteur potentiel (aucun processus n'effectue de lecture), le rédacteur risque de finir par saturer le pipe (dont la capacité est limitée) et rester bloqué dans l'opération d'écriture.

On reprend le code précédent, et on le transforme légèrement :

- chaque fils envoie un message de 1024 octets toutes les secondes, et affiche le numéro du message envoyé
- le père n'effectue aucune lecture dans le pipe, et s'endort pendant une longue durée tout en laissant l'extrémité 0 du pipe ouverte

```
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 pipe : des fils vers le père, avec bloquage en écriture */

#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 <signal.h>

#define NB_FILS 3      /* nombre de fils */

void handler_sigchld(int num_signal) {
   int wstatus, fils_termine ;
   /* prendre connaissance de la fin des fils */
   while ((fils_termine = (int) waitpid(-1, &wstatus, WNOHANG)) > 0) {
```

```c
      printf("\nMon fils %d est arrete\n", fils_termine) ;
   }
}

int main()
{
    int i, fils, retour, pid, num, retw ;
   char message[1024] ;
    int pipe_f2p[2] ;    /* pipe pour communiquer depuis les fils vers le père* */

   signal(SIGCHLD, handler_sigchld) ;
   for (i=0 ; i<sizeof(message) ; i++) {
      message[i]='a' ;
   }
    retour = pipe(pipe_f2p) ;
    /* Bonne pratique : tester systématiquement le retour des appels système */
    if (retour == -1) {    /* échec du pipe */
        printf("Erreur pipe\n") ;
        /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
        exit(1) ;
    }

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

    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[0]) ;
            pid = getpid() ;
         num = 1 ;
            /* écrire sans s'arrêter */
         while(1) {
            sprintf(message, "Le bonjour du fils %d ", fils) ;
            retw = write(pipe_f2p[1], message, sizeof(message)) ;
            printf("\n    Processus de pid %d envoie son message numero %d de tai
            num++ ;
            sleep(1) ;
         }
            /* fermer l'extrémité 1 : fin des envois */
            /* la prochaine lecture sur ce pipe renverra 0 */
            close(pipe_f2p[1]) ;

            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n"
                    getpid(), fils, retour) ;
        }
    }

    /* fermer l'extrémité 1 : le père va lire dans le pipe */
    close(pipe_f2p[1]) ;

    sleep(300) ;

    /* fermer l'extrémité 0 */
    close(pipe_f2p[0]) ;
    printf("\nProcessus Principal termine.\n") ;
    return EXIT_SUCCESS ;
}
```

A l'exécution, on peut vérifier que les fils envoient quelques dizaines de messages avant de rester bloqués dans l'opération d'écriture dans le pipe. On peut en déduire la capacité du pipe en multipliant le nombre de messages envoyés par la taille du message.

4.5 Plusieurs pipes

Nous allons illustrer une situation d'une communication du père vers ses fils, et on utilisera 3 pipes : un pour la communication avec chaque fils. Le père se contentera d'envoyer son pid, sous sa forme binaire, à chacun de ses fils.

Cet exemple illustre la nécessité de bien penser à gérer toutes les extrémités des pipes créés afin de ne pas créer des situations de bloquage.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 pipe par fils : depuis le père vers le fils */

#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 */

#define NB_FILS 3     /* nombre de fils */

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

    int pipe_p2f[NB_FILS+1][2] ;      /* pipe_p2f[0] non utilisé */

    /* Créet NB_FILS pipes */
    for (int i = 1 ; i <= NB_FILS ; i++) {
        retour = pipe(pipe_p2f[i]) ;
        if (retour == -1) {   /* échec du pipe */
            printf("Erreur pipe\n") ;
            exit(1) ;
        }
    }

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

    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 1 de son pipe : le fils est lecteur */
            close(pipe_p2f[fils][1]) ;

            /* fermer les extrémités de tous les autres pipes */
            for (int i = 1 ; i <= NB_FILS ; i++) {
                if (i!=fils) {
                    close(pipe_p2f[i][0]) ;
                    close(pipe_p2f[i][1]) ;
                }
            }
            /* lire ce qui arrrive dans le pipe */

            while (read(pipe_p2f[fils][0], &v_lue, sizeof(int)) > 0) {
                printf("\n     Processus de pid %d - recu : %d\n", getpid(), v_lue
            }
            /* fermer l'extrémité 0 de son pipe */
            close(pipe_p2f[fils][0]) ;

            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }
    /* fermer l'extrémité 0 de tous les pipes : le père est rédacteur */
    for (int i = 1 ; i <= NB_FILS ; i++) {
        close(pipe_p2f[i][0]) ;
    }

    pid = getpid() ;

    /* envoyer son pid à chaque fils */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        sleep(1) ;
        retour = write(pipe_p2f[fils][1], &pid, sizeof(int)) ;
        if (retour < 0) {
            printf("\nProcessus Principal : Erreur write %d.\n", retour) ;
        }
    }
    /* fermer l'extrémité 1 de tous les pipes : fin des envois */
```

```
85      /* les porchains read effectués dans les fils renvoient 0  */
86      for (int i = 1 ; i <= NB_FILS ; i++) {
87          close(pipe_p2f[i][1]) ;
88      }
89
90      printf("\nProcessus Principal termine.\n") ;
91      return EXIT_SUCCESS ;
92  }
```

4.6 SIGPIPE

Lorsqu'un processus tente d'écrire dans un pipe dont l'extrémité 0 n'est ouverte dans aucun processus (aucun processus n'est prêt à lire), le signal SIGPIPE est envoyé au processus rédacteur. Le traitement par défaut arrête ce processus.

Pour illustrer cela, on reprend le code précédent et on y apporte la modification suivante : le fils 1 ne lit rien et ferme aussitôt l'extrémité 0 de son pipe. Ceci aura pour effet l'envoi du signal SIGPIPE au père dès qu'il tentera d'écrire dans le pipe réservé au fils 1. Par défaut, le signal SIGPIPE provoque l'arrête du processus qui le reçoit.

```
 1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
 2  /* 1 pipe par fils : depuis le père vers le fils */
 3
 4  #include <stdio.h>    /* entrées sorties */
 5  #include <unistd.h>   /* pimitives de base : fork, ...*/
 6  #include <stdlib.h>   /* exit */
 7  #include <sys/wait.h> /* wait */
 8  #include <string.h>   /* opérations sur les chaines */
 9
10  #define NB_FILS 3     /* nombre de fils */
11
12  int main()
13  {
14      int fils, retour, v_lue, pid ;
15
16      int pipe_p2f[NB_FILS+1][2] ;      /* pipe_p2f[0] non utilisé */
17
18      /* Créet NB_FILS pipes */
19      for (int i = 1 ; i <= NB_FILS ; i++) {
20          retour = pipe(pipe_p2f[i]) ;
21          if (retour == -1) {   /* échec du pipe */
22              printf("Erreur pipe\n") ;
23              exit(1) ;
24          }
25      }
26
27      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
28
29      for (fils = 1 ; fils <= NB_FILS ; fils++) {
30          retour = fork() ;
31
32          /* Bonne pratique : tester systématiquement le retour des appels système
33          if (retour < 0) {   /* échec du fork */
34              printf("Erreur fork\n") ;
35              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
36              exit(1) ;
37          }
38
39          /* fils */
40          if (retour == 0) {
41              /* fermer l'extrémité 1 de son pipe : le fils est lecteur */
42              close(pipe_p2f[fils][1]) ;
43
44              /* fermer les extrémités de tous les autres pipes */
45              for (int i = 1 ; i <= NB_FILS ; i++) {
46                  if (i!=fils) {
47                      close(pipe_p2f[i][0]) ;
48                      close(pipe_p2f[i][1]) ;
49                  }
50              }
51              /* seuls les fils 2 et 3 lisent */
52              if (fils > 1 ) {
53                  /* lire ce qui arrrive dans le pipe */
54                  while (read(pipe_p2f[fils][0], &v_lue, sizeof(int)) > 0) {
55                      printf("\n     Processus de pid %d - recu : %d\n", getpid(),
56                  }
57              }
58              /* fermer l'extrémité 0 de son pipe */
59              close(pipe_p2f[fils][0]) ;
60
61              /* Important : terminer un processus par exit */
62              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
63          }
64
```

```
65          /* pere */
66          else {
67              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n"
68                      getpid(), fils, retour) ;
69          }
70      }
71      /* fermer l'extrémité 0 de tous les pipes : le père est rédacteur */
72      for (int i = 1 ; i <= NB_FILS ; i++) {
73          close(pipe_p2f[i][0]) ;
74      }
75
76      pid = getpid() ;
77
78      /* envoyer son pid à chaque fils */
79      for (fils = 1 ; fils <= NB_FILS ; fils++) {
80          sleep(1) ;
81          retour = write(pipe_p2f[fils][1], &pid, sizeof(int)) ;
82          if (retour < 0) {
83              printf("\nProcessus Principal : Erreur write %d.\n", retour) ;
84          }
85      }
86      /* fermer l'extrémité 0 de tous les pipes : fin des envois */
87      /* les porchains read effectués dans les fils renvoient 0  */
88      for (int i = 1 ; i <= NB_FILS ; i++) {
89          close(pipe_p2f[i][0]) ;
90      }
91
92      printf("\nProcessus Principal termine.\n") ;
93      return EXIT_SUCCESS ;
94  }
```

A l'exécution, on voit bien que le père est arrêté dès qu'il tente la première écriture sur le pipe du fils 1, et que fils 2 et 3 ne reçoivent aucun message.

On ajoute, maintenant, un traitant du signal SIGPIPE. Ce traitaant se contente d'afficher le numéro du siganl reçu et redonne la main au processus interrompu. Le père va donc pouvoir poursuivre ces émissions vers les autres fils. On remarquera à l'exécution que le signal SIGPIPE porte bien le numéro 13, et que le write renvoie bien -1 lorsque le pipe est fermé en lecture.

Il est donc important de traiter lesignal SIGPIPE pour éviter au rédacteur de subir un arrêt brutal à cause d'un pipe fermé en lecture.

```
 1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
 2  /* 1 pipe par fils : depuis le père vers le fils, sigpipe traité */
 3
 4  #include <stdio.h>     /* entrées sorties */
 5  #include <unistd.h>    /* pimitives de base : fork, ...*/
 6  #include <stdlib.h>    /* exit */
 7  #include <sys/wait.h>  /* wait */
 8  #include <string.h>    /* opérations sur les chaines */
 9
10  #define NB_FILS 3      /* nombre de fils */
11
12  /* Traitant du signal SIGPIPE */
13  void handler_sigpipe(int signal_num) {
14      printf("\n      Processus de pid %d : J'ai reçu le signal %d\n",
15              getpid(),signal_num) ;
16      return ;
17  }
18
19  int main()
20  {
21      int fils, retour, v_lue, pid ;
22
23      int pipe_p2f[NB_FILS+1][2] ;       /* pipe_p2f[0] non utilisé */
24
25      /* associer un traitant au signal SIGPIPE */
26      signal(SIGPIPE, handler_sigpipe) ;
27
28      /* Créet NB_FILS pipes */
29      for (int i = 1 ; i <= NB_FILS ; i++) {
30          retour = pipe(pipe_p2f[i]) ;
31          if (retour == -1) {    /* échec du pipe */
32              printf("Erreur pipe\n") ;
33              exit(1) ;
34          }
35      }
36
37      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
38
39      for (fils = 1 ; fils <= NB_FILS ; fils++) {
40          retour = fork() ;
41
42          /* Bonne pratique : tester systématiquement le retour des appels système
43          if (retour < 0) {    /* échec du fork */
```

```
44              printf("Erreur fork\n") ;
45              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
46              exit(1) ;
47          }
48
49          /* fils */
50          if (retour == 0) {
51              /* fermer l'extrémité 1 de son pipe : le fils est lecteur */
52              close(pipe_p2f[fils][1]) ;
53
54              /* fermer les extrémités de tous les autres pipes */
55              for (int i = 1 ; i <= NB_FILS ; i++) {
56                  if (i!=fils) {
57                      close(pipe_p2f[i][0]) ;
58                      close(pipe_p2f[i][1]) ;
59                  }
60              }
61              if (fils > 1) {
62                  /* lire ce qui arrrive dans le pipe */
63
64                  while (read(pipe_p2f[fils][0], &v_lue, sizeof(int)) > 0) {
65                      printf("\n     Processus de pid %d - recu : %d\n", getpid(),
66                  }
67              }
68              /* fermer l'extrémité 0 de son pipe */
69              close(pipe_p2f[fils][0]) ;
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      /* fermer l'extrémité 0 de tous les pipes : le père est rédacteur */
82      for (int i = 1 ; i <= NB_FILS ; i++) {
83          close(pipe_p2f[i][0]) ;
84      }
85
86      pid = getpid() ;
87
88      /* envoyer son pid à chaque fils */
89      for (fils = 1 ; fils <= NB_FILS ; fils++) {
90          sleep(1) ;
91          retour = write(pipe_p2f[fils][1], &pid, sizeof(int)) ;
92          if (retour < 0) {
93              printf("\nProcessus Principal : Erreur write %d.\n", retour) ;
94          }
95      }
96      /* fermer l'extrémité 0 de tous les pipes : fin des envois */
97      /* les porchains read effectués dans les fils renvoient 0  */
98      for (int i = 1 ; i <= NB_FILS ; i++) {
99          close(pipe_p2f[i][0]) ;
100     }
101
102     printf("\nProcessus Principal termine.\n") ;
103     return EXIT_SUCCESS ;
104 }
```

4.7 Synchronisation des lectures

Dans le cas d'un processus lecteur sur plusieurs pipes, se pose la question : comment va-t-il gérer l'ordre des lectures ?

Cette question n'est pas facile à traiter à cause du caractère bloquant (par défaut) du read. Le processus lecteur peut se trouver bloqué en lecture d'un pipe dans lequel rien n'arrive, alors que les données des autres pipes restent non consommées.

Le code suivant illustre cette situation : Trois pipes sont créés pour communiquer des fils vers le père. Ce dernier lit d'abord tout ce que le fils 1 lui envoie, puis fait de même avec le fils 2 et le fils 3. Mais, de l'autre côté, l'ordre chronologique des émissions est inversé : le fils 1 est le plus à émettre. Mais le processus lecteur ne connait pas cette information.

```
1 /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2 /* 1 pipe depuis chaque fils vers le père, retard de lecture */
3
4 #include <stdio.h>    /* entrées sorties */
5 #include <unistd.h>   /* pimitives de base : fork, ...*/
6 #include <stdlib.h>   /* exit */
7 #include <sys/wait.h> /* wait */
8 #include <string.h>   /* opérations sur les chaines */
9
```

```c
#define NB_FILS 3     /* nombre de fils */


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

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

    /* 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()) ;

    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]) ;
                }
            }

            pid = getpid() ;

            /* écrire 5 fois son pid dans le pipe */
            for (int i = 1 ; i <= 5 ; i++) {

                sleep(2 * (NB_FILS - fils)) ;
                printf("    Processus de pid %d effectue un envoi\n", pid) ;
                write(pipe_f2p[fils][1], &pid, sizeof(int)) ;
            }

            /* fermer l'extrémité 1 : fin des envois */
            close(pipe_f2p[fils][1]) ;

            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n"
                    getpid(), fils, retour) ;
        }
    }

    /* fermer l'extrémité 1 de tous les pipes : le père est lecteur */
    for (int i = 1 ; i <= NB_FILS ; i++) {
        close(pipe_f2p[i][1]) ;
    }

    /* lire ce que les fls envoient */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {

        while (read(pipe_f2p[fils][0], &pid, sizeof(int)) > 0) {
            printf("\nProcessus Principal - recu : %d\n", pid) ;
        }
    }
    /* fermer l'extrémité 0 de tous les pipes */
    for (int i = 1 ; i <= NB_FILS ; i++) {
        close(pipe_f2p[i][0]) ;
    }
    printf("\nProcessus Principal termine.\n") ;
    return EXIT_SUCCESS ;
}
```

A l'exécution, on voit bien que tous les messages du fils 3 sont traités en dernier alors qu'il était le plus rapide à effectuer ses envois.

On peut corriger en partie cette situation, en allant lire à chaque passage un seul messages sur chaque pipe : on répète une lecture sur chaque pipe jusqu'à ce que tous les pipes soient vides. Celà améliore légèrement la situation, mais les messages du fils 3 restent toujours retardés par la lenteur du fils 1. On verra dans le chapitre suivant comment traiter plus finement cette question.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 pipe par fils : depuis le fils vers le père, lecture alternée  */

#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 */

#define NB_FILS 3      /* nombre de fils */


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

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

    /* 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()) ;

    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]) ;
                }
            }

            pid = getpid() ;

            /* écrire 5 fois son pid dans le pipe */
            for (int i = 1 ; i <= 5 ; i++) {

                sleep(2 * (NB_FILS - fils)) ;
                printf("    Processus de pid %d effectue un envoi\n", pid) ;
                write(pipe_f2p[fils][1], &pid, sizeof(int)) ;
            }

            /* fermer l'extrémité 1 : fin des envois */
            close(pipe_f2p[fils][1]) ;

            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }

    /* fermer l'extrémité 1 de tous les pipes : le père est lecteur */
```

```
 78      for (int i = 1 ; i <= NB_FILS ; i++) {
 79          close(pipe_f2p[i][1]) ;
 80      }
 81
 82      sleep(2) ;
 83
 84      do {
 85          nb_recus = 0 ;    /* nombre de messages reçus à chaque passage */
 86          /* lire ce que les fls envoient un par un */
 87          for (fils = 1 ; fils <= NB_FILS ; fils++) {
 88
 89              if (read(pipe_f2p[fils][0], &pid, sizeof(int)) > 0) {
 90                  printf("\nProcessus Principal - recu : %d\n", pid) ;
 91                  nb_recus ++ ;
 92              }
 93          }
 94      } while (nb_recus > 0) ;
 95
 96      /* fermer l'extrémité 0 de tous les pipes */
 97      for (int i = 1 ; i <= NB_FILS ; i++) {
 98          close(pipe_f2p[i][0]) ;
 99      }
100      printf("\nProcessus Principal termine.\n") ;
101      return EXIT_SUCCESS ;
102  }
```

4.8 Dupplication de descripteur

Il est possible de dupliquer un descripteur, c'est-à-dire de créer un nouveau point d'accès à un fichier ou un pipe déjà ouvert.

La primitive `int dup(int desc)` fournit un nouveau descripteur ayant exactement les mêmes caractéristiques que le descripteur desc (qui doit nécessairement exister). Elle garantit que la valeur restituée est la plus petite possible. Autrement dit c'est le descripteur libre de plus petit indice qui est choisi et ré-alloué. "dup" renvoie le nouveau descripteur en cas de succès, ou −1 en cas de problème.

Un cas d'utilisation classique est la redirection de la sortie standard (1) vers un fichier accessible par un descripteur "fdesc". Pour cela on ferme le descripteur 1, qui devient le plus petit descripteur disponible, et on dupplique le descripteur du fichier avec la primitive dup(fdsc). En cas de succès, le descripteur 1 constitue un nouveau point d'accès au fichier, ce qui redirige toute écriture sur la sortie standard vers le fichier. Dans la séquence suiante, l'affichage du second message se fait dans le fichier accessible par fdesc et non plus à l'écran.

```
fdesc = open("fichier", O_WRONLY | O_CREAT | O_TRUNC, 0640);
printf("Message 1"\n");
close(1);
int nouv_desc = dup(fdesc);
printf("Message 2"\n");
```

La primitive `int dup2(int desc, int nv_desc)` permet de forcer nv_desc comme synonyme de desc. Si nv_desc est un descripteur ouvert alors il est préalablement fermé. "dup2" renvoie le nouveau descripteur en cas de suucès, ou −1 en cas de problème.

L'écriture de l'exemple précédent se trouve plus simple avec dup2.

```
fdesc = open("fichier", O_WRONLY | O_CREAT | O_TRUNC, 0640);
printf("Message 1"\n");
int nouv_desc = dup2(fdesc, 1);
printf("Message 2"\n");
```

Pour illuster ces mécanismes de façon plus complète, on utilse un programme appelé "factorielle" qui lit une suite d'entiers au clavier, en calcule la factorielle et affiche le résultat à l'écran :

```
<<< Entrez la valeur dont vous souhaitez calculer la factorielle (>=0) :
    Ou une valeur négative pour arrêter le programme : 4
>>> La factorielle de 4 est 24
<<< Entrez la valeur dont vous souhaitez calculer la factorielle (>=0) :
    Ou une valeur négative pour arrêter le programme : 7
>>> La factorielle de 7 est 5040
<<< Entrez la valeur dont vous souhaitez calculer la factorielle (>=0) :
    Ou une valeur négative pour arrêter le programme : -1
FIN
```

Et on propose d'écrire un programme qui, sans modifier le programme factorielle, lance le calcul sur des données se trouant dans un fichier, filtre les résultats pour n'en garder que les lignes significatives, et les enregistrer dans un fichier résultats. Pour cela, ce programme :

- redirige le contenu d'un fichier (suite d'entiers) vers l'entrée de l'exécutable "factorielle",
- en rédirige, via un pipe, les résultats vers la commande grep, pour en sélectionner les lignes contenant ">>>"
- et redirige les résultas du grep vers un fichier résultats

Ce qui est équivalent à l'exécution de la ligne de commande suivante :

```
./factorielle < fic_entrees | grep '>>>' > fic_resultats
```

Au final, le fichier "fic_resultats" ne contiendra que les lignes contenant ">>>".

```
>>> La factorielle de 4 est 24
>>> La factorielle de 7 est 5040
```

Ce qui donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Redirections : stdin, pipe et stdout */

#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>    /* opérations sur les fichiers */

int main()
{
    int retour ;
   int desc_ent, desc_res, dupdesc ;    /* descripteurs de fichiers */
   char fic_ent[] = "fic_ent_fact" ;
   char fic_res[] = "fic_res_fact" ;

    int pipe_cmd[2] ;   /* pipe entre deux commandes */

   /* ouvrir le fichier des entrées */
    desc_ent = open(fic_ent, O_RDONLY) ;
   /* traiter systématiquement les retours d'erreur des appels */
    if (desc_ent < 0) {
        printf("Erreur ouverture %s\n", fic_ent) ;
        exit(1) ;
    }

   /* ouvrir le fichier résultats */
    desc_res = open(fic_res, O_WRONLY | O_CREAT | O_TRUNC, 0640) ;
   /* traiter systématiquement les retours d'erreur des appels */
    if (desc_res < 0) {
        printf("Erreur ouverture %s\n", fic_res) ;
        exit(2) ;
    }

    retour = pipe(pipe_cmd) ;
    /* traiter systématiquement le retour des appels système */
    if (retour == -1) {   /* échec du pipe */
        printf("Erreur pipe\n") ;
        exit(3) ;
    }

   /* excéution de : ./factorielle < fic_ent_fact | grep '>>>' > fic_res_ent */
   /* fils : ./factorielle < fic_ent_fact | pere : grep '>>>' > fic_res_ent */
    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(4) ;
    }

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

        /* fermer l'extrémité 0 du pipe : le fils va écrire dans le pipe */
        close(pipe_cmd[0]) ;
      /* rediriger sdout vers l'entrée pipe_cmd[1] */
      /* écririture sur stdout => écririture sur pipe_cmd[1] */
      dupdesc = dup2(pipe_cmd[1], 1) ;
      if (dupdesc == -1) {   /* échec du dup2 */
         printf("Erreur dup2\n") ;
         exit(5) ;
      }

      /* rediriger sdtin vers desc_ent */
      /* lecture sur std => lecture sur desc_ent */
      dupdesc = dup2(desc_ent, 0) ;
      if (dupdesc == -1) {   /* échec du dup2 */
         printf("Erreur dup2\n") ;
         exit(5) ;
      }
      execl("./factorielle","factorielle", NULL) ;
```

```
75        /* on ne se retrouve ici que si exec echoue */
76          /* perror : affiche un message relatif à l'erreur du dernier appel systàm
77          perror("     exec ") ;
78          exit(6) ;
79      }
80
81      /* pere */
82      else {
83        /* fermer l'extrémité 1 : le fils va lire dans le pipe */
84          close(pipe_cmd[1]) ;
85        /* rediriger sdin vers pipe_cmd[0] */
86        /* lecture sur std => lecture sur  pipe_cmd[0] */
87        dupdesc = dup2(pipe_cmd[0], 0) ;
88        if (dupdesc == -1) {    /* échec du dup2 */
89           printf("Erreur dup2\n") ;
90           exit(7) ;
91        }
92
93        /* rediriger sdout vers desc_res */
94        /* écririture sur stdout => écririture sur desc_res */
95        dupdesc = dup2(desc_res, 1) ;
96        if (dupdesc == -1) {    /* échec du dup2 */
97           printf("Erreur dup2\n") ;
98           exit(7) ;
99        }
100       execlp("grep","grep", ">>>", NULL) ;
101       /* on ne se retrouve ici que si exec échoue */
102         /* perror : affiche un message relatif à l'erreur du dernier appel systàm
103         perror("     exec ") ;
104         exit(8) ;
105     }
106     return EXIT_SUCCESS ;
107 }
```

4.9 Calcul distribué du max d'un tableau

Comme indiqué au début de ce titoriel, nous allons utiliser le code étudié pour implanter un calcul distribué simple : le calcul du maximum d'un tableau.

- Le tableau sera déclaré et initialisé dans le père et donc connu par tous les fils
- chaque fils sera chargé du calcul du maximum d'une partie du tableau
- le père collectera les résultats de ses fils et calculera le maximum parmi ces valeurs

On utilisera un seul pipe pour que les fils (rédacteurs) puissent communiquer leur résultat au père (lecteur).

Ce qui donne le code suivant :

```
1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2  /* Calcul distribué du maximum d'un tableau : communication par pipe */
3
4  #include <stdio.h>    /* entrées sorties */
5  #include <unistd.h>   /* pimitives de base : fork, ...*/
6  #include <stdlib.h>   /* exit */
7  #include <sys/wait.h> /* wait */
8  #include <string.h>   /* opérations sur les chaines */
9
10 #define NB_FILS 3     /* nombre de fils */
11 #define NB_FLEM_FILS 100000
12 #define NB_ELEM NB_FILS*NB_FLEM_FILS
13
14 /* calcul du max d'un sous-tableau */
15 int cal_max_tab(int tab[], int i1, int i2) {
16     int i, max ;
17
18     max = tab[i1] ;
19
20     for (i = i1 + 1 ; i < i2 ; i++) {
21         if (tab[i]>max) {
22             max = tab[i] ;
23         }
24     }
25     return max ;
26 }
27
28 int main()
29 {
30     int fils, retour, fils_termine, wstatus, max, max_lu, nb_resultats ;
31
32     int tab[NB_ELEM] ;
33
34     int pipe_f2p[2] ;
35
36     /* initialiser le tableau */
37     for (int i=0 ; i < NB_ELEM ; i++) {
38         tab[i] = i+1 ;
```

```
39      }
40
41      pipe(pipe_f2p) ;
42
43      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
44
45      for (fils = 1 ; fils <= NB_FILS ; fils++) {
46          retour = fork() ;
47
48          /* Bonne pratique : tester systématiquement le retour des appels système
49          if (retour < 0) {   /* échec du fork */
50              printf("Erreur fork\n") ;
51              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
52              exit(1) ;
53          }
54
55          /* fils */
56          if (retour == 0) {
57              close(pipe_f2p[0]) ;
58              /* calculer le max du sous-tableau */
59              max = cal_max_tab(tab, (fils-1)*NB_FLEM_FILS, fils*NB_FLEM_FILS) ;
60              /* enregistrer le max en binaire */
61              write(pipe_f2p[1], &max, sizeof(int)) ;
62              close(pipe_f2p[1]) ;
63              /* Important : terminer un processus par exit */
64              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
65          }
66
67          /* pere */
68          else {
69              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n"
70                      getpid(), fils, retour) ;
71          }
72      }
73      close(pipe_f2p[1]) ;
74      max = 0 ;
75      nb_resultats = 0 ;
76      /* lire les résultats envoyés par les fils, et calculer le max */
77      while (read(pipe_f2p[0], &max_lu, sizeof(int))>0) {
78          nb_resultats++ ;
79          if (max_lu > max) {
80              max = max_lu ;
81          }
82      }
83
84      /* attendre la fin des fils */
85      for (fils = 1 ; fils <= NB_FILS ; fils++) {
86          /* attendre la fin d'un fils */
87          fils_termine = wait(&wstatus) ;
88
89          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
90              printf("\nMon fils de pid %d a termine avec exit %d\n",
91                      fils_termine, WEXITSTATUS(wstatus)) ;
92          }
93      }
94      close(pipe_f2p[1]) ;
95      printf("\nProcessus Principal termine - Nombre resultats recus %d - Max clacu
96              nb_resultats, max) ;
97      return EXIT_SUCCESS ;
98  }
```