Unix : Fichiers

Auteur: Zouheir HAMROUNI

3 Les fichiers

3.1 Introduction

Les fichiers ouverts dans un processus, sont identifiés par un entier appelé descripteur de fichier. Celui-ci est en réalité un simple indice dans un tableau de descripteurs gérés par le noyau. Ce tableau comporte un nombre limité d'entrées (fixé lors de l'installation du noyau, par exemple : 4096), ce qui restreint le nombre de fichiers ouverts simultanément par un processus. À la création d'un processus (fork), celui-ci hérite des descripteurs de fichiers de son père.

Par convention, les trois premiers descripteurs sont toujours ouverts au début d'un processus :

- 0 correspond à l'entrée standard (généralement le clavier)
- 1 correspond à la sortie standard (généralement l'écran)
- 2 correspond à la sortie standard des messages d'erreurs (généralement l'écran)

Dans "unistd.h", trois macros définissent ces descripteurs : STDIN_FILENO, STDOUT_FILENO et STDERR_FILENO.

Les primitives open, read, write et close permettent respectivement la création, la lecture, l'écriture, et la fermeture de fichiers.

Comme précédemment et sauf indication complémentaire, les primitives sont déclarées dans "unistd.h".

3.2 open & close

`int open (char *nomf, int option, int mode)`

Définie dans "fcntl.h", la primitive open permet d'ouvrir l'accès à un fichier (et éventuellement de le créer s'il n'existe pas).

- nomf : constitue une référence relative par rapport au répertoire de travail du processus si cette chaîne ne commence pas par le caractère /, sinon une référence absolue.
- option : détermine le mode d'ouverture du fichier. Il peut prendre une ou plusieurs des constantes symboliques (séparées par des OU bit-à-bit -"|"-) définies dans le fichier "fcntl.h".
  - O_RDONLY : ouverture en lecture
  - O_WRONLY : ouverture en écriture
  - O_APPEND : ouverture en écriture en fin de fichier
  - O_CREAT : création du fichier avec des droits d'accès définis par l'argument "mode"
  - O_EXCL : avec O_CREAT provoque un échec si le fichier existe déjà
  - O_TRUNC : ramène la taille du fichier à zéro si le fichier existe déjà
- mode : (droit d'accès) n'est pris en compte que si le fichier est effectivement créé
  - par exemple : rw- r-- --- peut être spécifié en octal avec 0640

Par exemple : desc = open("toto.txt", O_WRONLY | O_CREAT | O_TRUNC, 0640);

- ouvre le fichier "toto.txt" en écriture s'il existe, en le vidant de son contenu (O_TRUNC). Sans O_TRUNC, les nouvelles écritures viennent remplacer progressivement les données du fichier, mais ne les remplace complètement que si le volume des nouvelles données est supérieur ou égal à celui des anciennes.
- crée le fichier avec les droits rw- r-- --- et l'ouvre en écriture (O_CREAT)

`int close (int desc)` ferme l'accès au fichier référencé par le descripteur desc pour le processus appelant.

3.3 read

Les accès à un fichier se font selon le modèle d'une file séquentielle : une tête de lecture (ou d'écriture) est associé à chaque ouverture. Elle définit la position courante dans le fichier, à partir de laquelle se fera le prochain accès. Initialement positionnée au début du fichier, la tête avance du nombre d'octets lus à chaque opération de lecture (ou d'écriture).

`int read (int desc, char *buffer, int nb_octets)`

permet la lecture de nb_octets octets du fichier référencé par le descripteur desc, à partir de la position courante. La suite d'octets lus est placée dans le tampon buffer. La position courante (tête) progresse du nombre d'octets lus.

La valeur de retour vaut :

- le nombre d'octets effectivement lus (≤ nb_octets) : le nombre d'octets lus peut être strictement inférieur à la taille demandée si l'on se trouve en fin de fichier.
- 0 si la fin de fichier est atteinte dès la lecture du premier octet
- −1 en cas de problème (fichier fermé par exemple)

3.4 Une seule ouverture / lectures concurrentes

A une ouverture est associée une seule tête de lecture. Donc, si plusieurs processus accèdent au même fichier ouvert une seulle fois (dans un père par exemple, puis hérité par ses fils), ils se partagent la même tête de lecture. L'exemple suivant illustre ce partage. On y reprend le code du père qui crée trois fils :

- le père ouvre un fichier en lecture avant de créer ses trois fils, qui vont par conséquent se partager la même tête de lecture du fichier
- le fichier ouvert en lecture constient les mots : 100 101 ... 111
- chaque fils, qui aura hérité de l'accès au fichier, boucle en alternant une lecture de 4 octets (3 chiffres + un espace) et une période de sommeil jusqu'à ce qu'il ne reste plus rien à lire dans le fichier
- On introduit un petit déphasage initial pour bien séparer dans le temps les actions de lecture des 3 fils : le fils 3 lit le premier mot, suivi du fils2, puis du fils 1 et ainsi de suite.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Fichiers : lecture partagée entre père et fils avec ouverture unique */

#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 */

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, desc_fic, fils_termine, wstatus ;
    int duree_sommeil = 3 ;

    char fichier[] = "fic_centaines.txt" ;
    char buffer[8] ;     /* buffer de lecture */

    /* Initialiser buffer avec 0 */
    bzero(buffer, sizeof(buffer)) ;

    /* ouverture du fichier en lecture */
```

```
25      desc_fic = open(fichier, O_RDONLY) ;
26      /* traiter systématiquement les retours d'erreur des appels */
27      if (desc_fic < 0) {
28          printf("Erreur ouverture %s\n", fichier) ;
29          exit(1) ;
30      }
31
32      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
33
34      for (fils = 1 ; fils <= NB_FILS ; fils++) {
35          retour = fork() ;
36
37          /* Bonne pratique : tester systématiquement le retour des appels système */
38          if (retour < 0) {   /* échec du fork */
39              printf("Erreur fork\n") ;
40              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
41              exit(1) ;
42          }
43
44          /* fils */
45          if (retour == 0) {
46              /* decaler les lectures des differents fils : fils 3, fils 2, fils 1 */
47              sleep(NB_FILS - fils) ;
48              /* lire le fichier par blocs de 4 octets */
49              while (read(desc_fic, buffer, 4) > 0) {
50
51                  printf("     Processus fils numero %d a lu %s\n",
52                          fils, buffer) ;
53                  sleep(duree_sommeil) ;
54                  bzero(buffer, sizeof(buffer)) ;
55              }
56              /* Important : terminer un processus par exit */
57              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
58          }
59
60          /* pere */
61          else {
62              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
63                      getpid(), fils, retour) ;
64          }
65      }
66
67      /* attendre la fin des fils */
68      for (fils = 1 ; fils <= NB_FILS ; fils++) {
69          /* attendre la fin d'un fils */
70          fils_termine = wait(&wstatus) ;
71
72          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
73              printf("\nMon fils de pid %d a termine avec exit %d\n",
74                      fils_termine, WEXITSTATUS(wstatus)) ;
75          }
76          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
77              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
78                      fils_termine, WTERMSIG(wstatus)) ;
79          }
80      }
81      close(desc_fic) ;
82      printf("\nProcessus Principal termine\n") ;
83      return EXIT_SUCCESS ;
84  }
```

A l'exécution on voit bien que :

- les 3 fils ne lisent pas la même chose : le fils 3 lit les 4 premiers octets, le fils 2 les 4 suivants, et ainsi de suite
- l'ensemble des données lues par les 3 fils correspond bien à la totalité du contenu du fichier

Ce qui prouve que les 3 fils utilisent la même tête de lecture.

Si on change la synchronisation temporelle entre les différents processus, les résultats de lecture peuvent être différents, car ils dépendent de l'ordre d'accès au fichier par les différents processus.

3.5 Déplacement de la tête

La tête de lecture s'incrémente automatiquement du nombre d'octets lus, mais peut aussi âtre modifiée grâce à la primitive `lseek`

`off_t lseek(int fd, off_t offset, int mode)`

permet de déplacer la tête d'accès au fichier. L'arguement mode peut prendre les valeurs suivantes :

- SEEK_SET : La tête est placée à offset octets depuis le début du fichier
- SEEK_CUR : La tête est déplacée de offset octets à partir de la position courante
- SEEK_END : La tête est placée à la fin du fichier plus offset octets

En lecture, la nouvelle position de la tâte doit rester comprise entre le début et la fin du fichier.

lseek(), s'il réussit, renvoie le nouvel emplacement, mesuré en octets depuis le début du fichier. En cas d'échec, la valeur (off_t) -1 est renvoyée.

Pour illustrer ceci, nous reprenons le code précédent, en y apportant les modifications suivantes :

- avant la première lecture, le fils 3 positione la tête à début + 4
- après chaque lecture, le fils 1 avance la tête de lecture de 4 octets

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Fichiers : lecture partagée avec ouverture unique et lssek */

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

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, desc_fic, fils_termine, wstatus ;
    int duree_sommeil = 3 ;

    char fichier[] = "fic_centaines.txt" ;
    char buffer[8] ;     /* buffer de lecture */

    /* Initialiser buffer avec 0 */
    bzero(buffer, sizeof(buffer)) ;

    /* ouverture du fichier en lecture */
    desc_fic = open(fichier, O_RDONLY) ;
    /* traiter systématiquement les retours d'erreur des appels */
    if (desc_fic < 0) {
        printf("Erreur ouverture %s\n", fichier) ;
        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) ;
        }

```

```
45          /* fils */
46          if (retour == 0) {
47              /* decaler les lectures des differents fils : fils 3, fils 2, fils 1, ... */
48              sleep(NB_FILS - fils) ;
49
50              if (fils == NB_FILS) {
51                  lseek(desc_fic, 4, SEEK_SET) ;
52              }
53              /* lire le fichier par blocs de 4 octets */
54              while (read(desc_fic, buffer, 4) > 0) {
55
56                  printf("     Processus fils numero %d a lu %s\n",
57                          fils, buffer) ;
58                  if (fils == 1) {
59                      lseek(desc_fic, 4, SEEK_CUR) ;
60                  }
61                  sleep(duree_sommeil) ;
62                  bzero(buffer, sizeof(buffer)) ;
63              }
64              /* Important : terminer un processus par exit */
65              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
66          }
67
68          /* pere */
69          else {
70              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
71                      getpid(), fils, retour) ;
72          }
73      }
74
75      /* attendre la fin des fils */
76      for (fils = 1 ; fils <= NB_FILS ; fils++) {
77          /* attendre la fin d'un fils */
78          fils_termine = wait(&wstatus) ;
79
80          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
81              printf("\nMon fils de pid %d a termine avec exit %d\n",
82                      fils_termine, WEXITSTATUS(wstatus)) ;
83          }
84          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
85              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
86                      fils_termine, WTERMSIG(wstatus)) ;
87          }
88      }
89      close(desc_fic) ;
90      printf("\nProcessus Principal termine\n") ;
91      return EXIT_SUCCESS ;
92  }
```

A l'exécution, on voit bien que :

- le premier mot n'est lu par aucun des 3 fils, car fils 3 a avancé la tête de +4
- le deuxième, troisième et quatrième mots sont lus respectivement par fils 3, fils 2 et fils 1
- le cinquième mot n'est lu par aucun des 3 fils, car fils 1 a avancé la tête de +4
- le neuvième mot n'est lu par aucun des 3 fils, car fils 1 a avancé la tête de +4

3.6 Plusieurs têtss / lectures concurrentes

Lorsque chaque processus effectue sa propre ouverture du même fichier, chaque processus dispose de sa propre tête de lecture. Par conséquent, les accès effectués par les différents processus sont totalement indépendants. Pour illustrer cela, on reprend le code précédent, et on déplace l'ouverture du fichier dans chacun des fils. Ce qui donne :

```
1 /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2 /* Fichiers : lecture partagée avec ouvertures séparées */
3
4 #include <stdio.h>    /* entrées sorties */
5 #include <unistd.h>   /* pimitives de base : fork, ...*/
6 #include <stdlib.h>   /* exit */
```

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

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, desc_fic, fils_termine, wstatus ;
    int duree_sommeil = 3 ;

    char fichier[] = "fic_centaines.txt" ;
    char buffer[8] ;     /* buffer de lecture */

    bzero(buffer, sizeof(buffer)) ;

    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) {
            /* ouverture du fichier en lecture */
            desc_fic = open(fichier, O_RDONLY) ;
            /* traiter systématiquement les retours d'erreur des appels */
            if (desc_fic < 0) {
                printf("Erreur ouverture %s\n", fichier) ;
                exit(1) ;
            }

            sleep(NB_FILS - fils) ;

            /* lire le fichier par blocs de 4 octets */
            while (read(desc_fic, buffer, 4)>0) {

                printf("     Processus fils numero %d a lu %s\n",
                        fils, buffer) ;
                sleep(duree_sommeil) ;
                bzero(buffer, sizeof(buffer)) ;
            }

            close(desc_fic) ;

            /* 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) ;
        }
    }
    /* attendre la fin des fils */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        /* attendre la fin d'un fils */
        fils_termine = wait(&wstatus) ;

        if WIFEXITED(wstatus) {   /* fils terminé avec exit */
            printf("\nMon fils de pid %d a termine avec exit %d\n",
                    fils_termine, WEXITSTATUS(wstatus)) ;
        }
        else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
            printf("\nMon fils de pid %d a ete tue par le signal %d\n",
                    fils_termine, WTERMSIG(wstatus)) ;
        }
```

```
81      }
82
83      printf("\nProcessus Principal termine\n") ;
84      return EXIT_SUCCESS ;
85  }
```

A l'exécution, on voit bien que chaque fils lit la toalité du fichier, preuve que chaque processus dispose de sa propre tête de lecture qu'il ne partage pas avec ses frères.

Même si on modifie la synchronisation temporelle des différents processus, les résultats de lecture restent les mêmes : chaque processus lit dans le bon ordre la totalité du fichier.

3.7 write

`int write (int desc, char *buffer, int nb_octets)`

permet d'écrire, à partir de la position courante dans le fichier accessible via le descripteur desc, les nb_octets premiers octets du tampon buffer. La valeur de retour est le nombre d'octets effectivement écrits ; elle vaut −1 en cas d'erreur. La position courante de la tête d'écriture progresse du nombre d'octets écrits.

Lorsque plusieurs processus écrivent dans le même fichiers, deux comportements sont possibles :

- si l'ouverture du fichier a été faite une fois dans un seul processus, et l'accès hérité par des fils, les processus se partagent la même tête d'écriture, et les écritures s'ajoutent les unes aux autres (dans l'orde des écritures)
- si chaque processus effectue sa propre ouverture, chaque processus dispose de sa propre tête d'écriture, et les écritures peuvent s'écraser mutuellement : la dernière écriture à une position "i" écrase la précédente

Les 2 programmes suivants illustrent ces deux comportements.

3.8 Une seule tête / écritures concurrentes

Dans l'exemple suivant :

- le fichier est ouvert en écriture par le père avant la création des 3 fils
- les fils effectuent, chacun, 4 écritures dans le fichier de manière entrelacée : fils 3 écrit un mot de 4 octets, puis fils 2, puis fils 1, ...

```
1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2  /* 1 fichier : 1 seule ouverture en écriture partagée */
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  #include <fcntl.h>    /* opérations sur les fichiers */
10
11 #define NB_FILS 3     /* nombre de fils */
12
13 int main()
14 {
15     int fils, retour, desc_fic, fils_termine, wstatus, ifor ;
16     int duree_sommeil = 3 ;
17
18     char fichier[] = "fic_res_ouv_uni.txt" ;
19     char buffer[8] ;     /* buffer de lecture */
20
21     bzero(buffer, sizeof(buffer)) ;
22
23     /* ouverture du fichier en ecriture, avec autorisations rw- -r- ---*/
24     /* avec création si le fichier n'existe pas : O_CREAT */
```

```
25      /* avec vidange (raz du contenu) si le fichier existe: O_TRUNC */
26      desc_fic = open(fichier, O_WRONLY | O_CREAT | O_TRUNC, 0640) ;
27
28      /* traiter systématiquement les retours d'erreur des appels */
29      if (desc_fic < 0) {
30          printf("Erreur ouverture %s\n", fichier) ;
31          exit(1) ;
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              /* decaler les écritures des differents fils : fils 3, fils 2, fils 1, ... */
49              sleep(NB_FILS - fils) ;
50
51              /* effectuer 4 ecritures dans le fichier */
52              for (ifor = 1 ; ifor <= 4 ; ifor++) {
53                  bzero(buffer, sizeof(buffer)) ;
54                  sprintf(buffer, "%d-%d\n", fils,ifor) ;
55                  write(desc_fic, buffer, strlen(buffer)) ;
56                  printf("     Processus fils numero %d a ecrit %s\n",
57                          fils, buffer) ;
58                  sleep(duree_sommeil) ;
59              }
60              /* Important : terminer un processus par exit */
61              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
62          }
63
64          /* pere */
65          else {
66              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
67                      getpid(), fils, retour) ;
68          }
69      }
70      /* attendre la fin des fils */
71      for (fils = 1 ; fils <= NB_FILS ; fils++) {
72          /* attendre la fin d'un fils */
73          fils_termine = wait(&wstatus) ;
74
75          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
76              printf("\nMon fils de pid %d a termine avec exit %d\n",
77                      fils_termine, WEXITSTATUS(wstatus)) ;
78          }
79          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
80              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
81                      fils_termine, WTERMSIG(wstatus)) ;
82          }
83      }
84      close(desc_fic) ;
85      printf("\nProcessus Principal termine\n") ;
86      return EXIT_SUCCESS ;
87  }
```

Après exécution, on voit peut vérifier que le fichier généré contient tous les mots écrits par les 3 fils (3 x 4), dans l'ordre chronolique d'écriture.

3.9 Déplacement de la tête d'écriture

Comme vu plus haut, la primitive lseek permet aussi de déplacer la tête d'écrire. Elle permet même de placer la tête au-delà de la fin actuelle du fichier, mais cela ne modifie la taille du fichier

que si une nouvelle écriture est réalisée à partir de la nouvelle position. Dans ce cas, les nouveaux octets générés par le déplacement sont mis à 0.

Pour illustrer celà, on reprend l'exemple précédent, et y effectue 2 ajouts :

- avant chaque écriture, le fils 2 déplace la tête d'écriture de -4 octets
- après chaque écriture, le fils 1 déplace la tête d'écriture de +4 octets

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* 1 fichier : 1 seule ouverture en écriture partagée et lseek */

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

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, desc_fic, fils_termine, wstatus, ifor ;
    int duree_sommeil = 3 ;

    char fichier[] = "fic_res_ouv_uni_lseek.txt" ;
    char buffer[8] ;     /* buffer de lecture */

    bzero(buffer, sizeof(buffer)) ;

    /* ouverture du fichier en ecriture, avec autorisations rw- -r- ---*/
    /* avec création si le fichier n'existe pas : O_CREAT */
    /* avec vidange (raz du contenu) si le fichier existe: O_TRUNC */
    desc_fic = open(fichier, O_WRONLY | O_CREAT | O_TRUNC, 0640) ;

    /* traiter systématiquement les retours d'erreur des appels */
    if (desc_fic < 0) {
        printf("Erreur ouverture %s\n", fichier) ;
        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) {
            /* decaler les écritures des differents fils : fils 3, fils 2, fils 1, ... */
            sleep(NB_FILS - fils) ;

            /* effectuer 4 ecritures dans le fichier */
            for (ifor = 1 ; ifor <= 4 ; ifor++) {
                // fils 2 recule la tete de 4 octets
                if (fils == 2) {
                    lseek(desc_fic, -4, SEEK_CUR) ;
                }
                bzero(buffer, sizeof(buffer)) ;
                sprintf(buffer, "%d-%d\n", fils,ifor) ;
                write(desc_fic, buffer, strlen(buffer)) ;
                printf("     Processus fils numero %d a ecrit %s\n",
                        fils, buffer) ;
                // fils 1 avance la tete de 4 octets
                if (fils == 1) {
```

```
65                      lseek(desc_fic, 4, SEEK_CUR) ;
66                  }
67                  sleep(duree_sommeil) ;
68              }
69              /* Important : terminer un processus par exit */
70              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
71          }
72
73          /* pere */
74          else {
75              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
76                      getpid(), fils, retour) ;
77          }
78      }
79      /* attendre la fin des fils */
80      for (fils = 1 ; fils <= NB_FILS ; fils++) {
81          /* attendre la fin d'un fils */
82          fils_termine = wait(&wstatus) ;
83
84          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
85              printf("\nMon fils de pid %d a termine avec exit %d\n",
86                      fils_termine, WEXITSTATUS(wstatus)) ;
87          }
88          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
89              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
90                      fils_termine, WTERMSIG(wstatus)) ;
91          }
92      }
93      close(desc_fic) ;
94      printf("\nProcessus Principal termine\n") ;
95      return EXIT_SUCCESS ;
96  }
```

Après exécution, on découvre dans le fichier généré (visualison avec la commande hd) :

1. les mots écrits par le fils 3 ont disparu : ils ont été écrasés par les écritures du fils 2 qui déplaçait la tête d'écriture de -4 octets
2. après chaque mot écrit par le fils 1, apparaissent 4 octets initialisés à 0 : en faisant un déplacement de +4 en fin de fichier, le fils ajoutait 4 octets supplémentaires
3. mais le dernier déplacement de +4 effectué par le fils 1 n'a pas été confirmé par une écriture sur ou au delà des 4 octets ajoutés ; ce qui a eu pour effet de ne pas enregistrer ces 4 octets.

3.10 Plusieurs têtes / écritures concurrentes

Dans l'exemple suivant :

- chaque fils effectue sa propre ouverture du fichier en écriture
- les fils effectuent, chacun, 4 écritures dans le fichier de manière entrelacée : fils 3 écrit un mot de 4 octets, puis fils 2, puis fils 1, ...

```
1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2  /* 1 fichier : plusieurs ouvertures, écritures concurrentes */
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  #include <fcntl.h>    /* opérations sur les fichiers */
10
11 #define NB_FILS 3     /* nombre de fils */
12
13 int main()
14 {
15     int fils, retour, desc_fic, fils_termine, wstatus, ifor ;
16     int duree_sommeil = 3 ;
17
```

```
18      char fichier[] = "fic_res_ouv_sep.txt" ;
19      char buffer[8] ;     /* buffer de lecture */
20
21      bzero(buffer, sizeof(buffer)) ;
22
23      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
24
25      for (fils = 1 ; fils <= NB_FILS ; fils++) {
26          retour = fork() ;
27
28          /* Bonne pratique : tester systématiquement le retour des appels système */
29          if (retour < 0) {   /* échec du fork */
30              printf("Erreur fork\n") ;
31              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
32              exit(1) ;
33          }
34
35          /* fils */
36          if (retour == 0) {
37              /* ouverture du fichier en lecture */
38              desc_fic = open(fichier, O_WRONLY | O_CREAT | O_TRUNC, 0640) ;
39              /* traiter systématiquement les retours d'erreur des appels */
40              if (desc_fic < 0) {
41                  printf("Erreur ouverture %s\n", fichier) ;
42                  exit(1) ;
43              }
44
45              /* decaler les écritures des differents fils : fils 3, fils 2, fils 1, ... */
46              sleep(NB_FILS - fils) ;
47
48              /* effectuer 4 ecritures dans le fichier */
49              for (ifor = 1 ; ifor <= 4 ; ifor++) {
50                  bzero(buffer, sizeof(buffer)) ;
51                  sprintf(buffer, "%d-%d\n", fils,ifor) ;
52                  write(desc_fic, buffer, strlen(buffer)) ;
53                  printf("     Processus fils numero %d a ecrit %s\n",
54                          fils, buffer) ;
55                  sleep(duree_sommeil) ;
56              }
57
58              close(desc_fic) ;
59              /* Important : terminer un processus par exit */
60              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
61          }
62
63          /* pere */
64          else {
65              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
66                      getpid(), fils, retour) ;
67          }
68      }
69      /* attendre la fin des fils */
70      for (fils = 1 ; fils <= NB_FILS ; fils++) {
71          /* attendre la fin d'un fils */
72          fils_termine = wait(&wstatus) ;
73
74          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
75              printf("\nMon fils de pid %d a termine avec exit %d\n",
76                      fils_termine, WEXITSTATUS(wstatus)) ;
77          }
78          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
79              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
80                      fils_termine, WTERMSIG(wstatus)) ;
81          }
82      }
83
84      printf("\nProcessus Principal termine\n") ;
85      return EXIT_SUCCESS ;
86  }
```

Après exécution, on découvre dans le fichier généré seulement 4 mots écrits par le fils 1. Bizarre ? non ?

L'explication est assez simple, car chaque processus utilise sa propre tête d'écriture :

1. le fils 3 écrit son premier mot à la position 1, et sa tête d'écriture passe à la position 2
2. puis, le fils 2 écrit son premier mot à la position 1 (sa tête d'écriture n'a pas bougé), et écrase donc le mot écrit par le fils 3
3. puis, le fils 1 écrit son premier mot à la position 1 (sa tête d'écriture n'a pas bougé), et écrase donc le mot écrit par le fils
4. à la fin de cette séquence, les 3 têtes d'écriture se trouvent à la position 2, et le même effet se reproduit

3.11 Calcul distribué du max d'un tableau

Comme indiqué au début de ce cours, 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 fournis par ses fils et calculera le maximum parmi ces valeurs

Mais comment les fils vont-ils communiquer leur résultat au père ?

On utilisera un fichier partagé en écriture par les fils, avec un seul curseur pourque les écritures ne s'écrasent pas mutuellement. Ce fichier sera ouvert en lecture par le père pour récupérer les résultats enregistrés par les fils. Ce qui donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Calcul distribué du maximum d'un tableau : communication par fichier */

#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 */

#define NB_FILS 3     /* nombre de fils */
#define NB_FLEM_FILS 100000
#define NB_ELEM NB_FILS*NB_FLEM_FILS

/* calcul du max d'un sous-tableau */
int cal_max_tab(int tab[], int i1, int i2) {
    int i, max ;

    max = tab[i1] ;

    for (i = i1 + 1 ; i < i2 ; i++) {
        if (tab[i]>max) {
            max = tab[i] ;
        }
    }
    return max ;
}

int main()
{
    int fils, retour, desc_fic, fils_termine, wstatus, max, max_lu ;

    int tab[NB_ELEM] ;

    char fichier[] = "fic_3f_maxtab" ;

    /* initialiser le tableau */
    for (int i=0 ; i < NB_ELEM ; i++) {
        tab[i] = i+1 ;
    }

    /* ouvrir le fichier en écriture */
```

```
43      desc_fic = open(fichier, O_WRONLY | O_CREAT | O_TRUNC, 0640) ;
44      /* traiter systématiquement les retours d'erreur des appels */
45      if (desc_fic < 0) {
46          printf("Erreur ouverture %s\n", fichier) ;
47          exit(1) ;
48      }
49
50      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
51
52      for (fils = 1 ; fils <= NB_FILS ; fils++) {
53          retour = fork() ;
54
55          /* Bonne pratique : tester systématiquement le retour des appels système */
56          if (retour < 0) {   /* échec du fork */
57              printf("Erreur fork\n") ;
58              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
59              exit(1) ;
60          }
61
62          /* fils */
63          if (retour == 0) {
64              /* calculer le max du sous-tableau */
65              max = cal_max_tab(tab, (fils-1)*NB_FLEM_FILS, fils*NB_FLEM_FILS) ;
66              /* enregistrer le max en binaire */
67              write(desc_fic, &max, sizeof(int)) ;
68              /* Important : terminer un processus par exit */
69              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
70          }
71
72          /* pere */
73          else {
74              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
75                      getpid(), fils, retour) ;
76          }
77      }
78
79      /* fermer le fichier ouvert en ecriture */
80      close(desc_fic) ;
81
82      /* ouvrir le fichier en lecture */
83      desc_fic = open(fichier, O_RDONLY) ;
84      /* traiter systématiquement les retours d'erreur des appels */
85      if (desc_fic < 0) {
86          printf("Erreur ouverture %s\n", fichier) ;
87          exit(1) ;
88      }
89
90      max = 0 ;
91      /* attendre la fin des fils */
92      for (fils = 1 ; fils <= NB_FILS ; fils++) {
93          /* attendre la fin d'un fils */
94          fils_termine = wait(&wstatus) ;
95
96          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
97              printf("\nMon fils de pid %d a termine avec exit %d\n",
98                      fils_termine, WEXITSTATUS(wstatus)) ;
99          }
100         else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
101             printf("\nMon fils de pid %d a ete tue par le signal %d\n",
102                     fils_termine, WTERMSIG(wstatus)) ;
103         }
104         /* lire les nouvelles valeurs communiquées par les fils */
105         /* et calculer le max intermédiaire */
106         while (read(desc_fic, &max_lu, sizeof(int))>0) {
107             if (max_lu > max) {
108                 max = max_lu ;
109             }
110         }
111     }
112     close(desc_fic) ;
113     printf("\nProcessus Principal termine. Max = %d\n", max) ;
114     return EXIT_SUCCESS ;
115 }
```