TD2 : Entrées / Sorties

RS : Réseaux et Systèmes

★ **Exercice 1:** On considère le programme ci-contre :

▷ **Question 1:** Qu'affiche-t-il quand `toto.txt` et `titi.txt` existent tous les deux ?

Réponse

Le programme affiche :

`fd2 = 3 (errno = 0)`

En effet, les descripteurs sont choisis par ordre croissant parmi les descripteurs libres. Le premier alloué pour `toto.txt` est 3 (0, 1, 2 sont occupés par stdin et ses amis) ; puis il est libéré par `close` et réalloué pour `titi.txt`.

Fin réponse

```
1  #include <stdio.h>
2  #include <fcntl.h> /* O_RDONLY */
3  #include <errno.h>
4  int main(){
5    int fd1, fd2;
6    fd1 = open("toto.txt", O_RDONLY, 0);
7    close(fd1);
8    fd2= open("titi.txt", O_RDONLY, 0);
9    printf("fd2 = %d (errno=%d)\n", fd2, errno);
10   return 0;
11 }
```

▷ **Question 2:** Quelle différence si `toto.txt` n'existe pas ? Et si `titi.txt` n'existe pas ?

Réponse

Si toto n'existe pas, le close n'est pas valide (mais linux ne dit rien car fermer ce qui n'a jamais été ouvert est une no-operation). Cependant, le printf d'errno ligne 9 ne peut le voir car la valeur d'errno sera écrasée par l'appel open ligne 8.

Et si titi n'existe pas : `fd2 = -1 (errno = 2)` (errno=ENOENT "No such file or directory" sous linux) (et ce que toto.txt existe ou non : errno est écrasé).

Fin réponse

CORRECTION

CORRECTION

★ **Exercice 2:** On suppose que le fichier `toto.txt` contient les six caractères "LAMBDA". On considère les trois programmes ci-après.

▷ **Question 1:** Qu'affichent chacun de ces programmes ? Si vous ne le savez pas (ce n'est pas dans le cours), proposez des hypothèses en les justifiant.

Deux open
```
1  #include <stdio.h>
2  #include <fcntl.h>
3  int main() {
4    int fd1, fd2; char c;
5    fd1=open("toto.txt", O_RDONLY, 0);
6    fd2=open("toto.txt", O_RDONLY, 0);
7    read(fd1, &c, 1);
8    read(fd2, &c, 1);
9    printf("c = %c\n", c);
10   return 0;
11 }
```

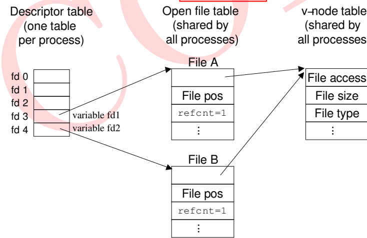
Après fork
```
1  #include <stdio.h>
2  #include <fcntl.h> /* O_RDONLY */
3  int main(){
4    int fd; char c;
5    fd=open("toto.txt", O_RDONLY, 0);
6    if (fork() == 0) {
7      read(fd, &c, 1);
8      return 0;
9    }
10   wait(NULL);
11   read(fd, &c, 1);
12   printf("c = %c\n", c);
13   return 0;
14 }
```

Après dup
```
1  #include <stdio.h>
2  #include <fcntl.h> /* O_RDONLY */
3  int main(){
4    int fd1, fd2; char c;
5    fd1=open("toto.txt", O_RDONLY, 0);
6    read(fd1, &c, 1);
7    fd2 = dup(fd1);
8    read(fd2, &c, 1);
9    printf("c = %c\n", c);
10   return 0;
11 }
```

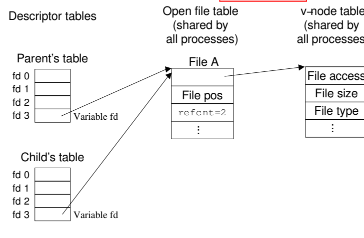
Réponse

La question est de savoir si les descripteurs sont partagés, et comment (ou plus exactement, le pointeur de position : est ce que la lecture sur un descripteur avance les autres ?). Si pointeur de position partagé, `c = A`, si non `c = L`. Ce pointeur est dans l'entrée correspondante dans la table des fichiers ouverts (dans le noyau).

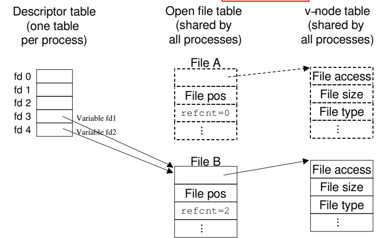
Deux open : `c = L`



Après un fork : `c = A`



Après un dup : `c = A`



Si on fait deux open, le pointeur d'avancement n'est pas partagé puisqu'on crée une nouvelle entrée dans la table des fichiers ouverts à chaque open (ca pointe sur le meme fichier sur disque, mais c'est une autre histoire).

Lors d'un fork, seule la table dans le processus est dupppliqué. Et du coup, le pointeur est partagé puisque c'est la même entrée de la table du noyau qui est utilisée.

Lors d'un dup, on fait pointer la table interne au processus vers la même entrée de la table du noyau (et le pointeur d'avancement est alors partagé). L'autre fichier voit son refcount passer à 0, et il est donc fermé.

Fin réponse

★ **Exercice 3: Redirection et shell.**

Réponse

Il est important de faire les questions les unes après les autres. Si les élèves lisent tout, il est impossible de voir où on va. Il faut les mener pas à pas vers l'objectif : découvrir comment certain programmes classiques sont implémentés sans regarder leur source, simplement expérimentalement.

Fin réponse

mycat
```
1  #include <errno.h>
2  int main() {
3    char BUFF[1024];
4    int lu;
5
6    while ((lu=read(3,&BUFF,1024))>0) {
7      int a_ecrire=lu;
8      char *pos=BUFF;
9
10     while (a_ecrire) {
11       int ecrit=write(1,pos,a_ecrire);
12       a_ecrire -= ecrit;
13       pos += ecrit;
14     }
15   }
16   if (lu<0)
17     perror("probleme de lecture");
18 }
```

▷ **Question 1:** Quels sont les appels systèmes faits par le shell lorsque l'on tape `cat < toto` ? (question de cours)

▷ **Question 2:** Si le shell suit bien votre hypothèse de la question précédente, que devrait-il se passer si l'on fait `./mycat < toto` ? (*i.e.*, on remplace `cat` par le programme ci-contre)

▷ **Question 3:** Comment expliquez-vous que l'on obtienne l'erreur « Mauvais descripteur de fichier » à la place ? Modifiez votre réponse à la première question si besoin pour l'expliquer.

Réponse

CORRECTION CORRECTION

Question 1 :

```
if (! fork()) {
  fd=open("toto");
  dup2(fd,0);
  execlp("cat","cat",NULL);
}
```

Question 2 : Ça devrait marcher pareil, puisque fd=3, et que le descripteur 3 reste ouvert sur le fichier lui aussi dans le ptit bout de code proposé à la question précédente.

Question 3 : Alors ça veut dire que le shell ferme 3 avant de faire l'exec. Ie, l'hypothèse de la question 1 était pas tout à fait juste. On sait maintenant que bash fait un truc du genre :

```
if (! fork()) {
  fd=open("toto");
  dup2(fd,0);
  close(fd); /* C'est ici que ca change */
  execlp("cat","cat",NULL);
```

Fin réponse

★ **Exercice 4: Réimplémenter popen.**

La fonction `popen` permet de lire la sortie standard d'une commande (ou d'écrire sur son entrée standard) comme s'il s'agissait d'un fichier. Exemple d'ouverture en lecture : `fich = popen("ls -lR","r");`

Exemple d'ouverture en écriture : `fich = popen("ssh neptune","w");`

▷ **Question 1:** Implémentez une fonction permettant de lire la sortie d'une commande (ie, en supposant que le second argument est "r").

Remarque : popen() retourne un `FILE*` utilisable avec `fscanf`. Votre version devra retourner un descripteur pour `read` (c'est plus simple ainsi).

Réponse

Rien de sorcier, c'est juste histoire d'utiliser un pipe(), et de refaire un coup de fork().

```
int mypopen(char *cmd) {
  int fd[2];
  pipe(fd);
  if (fork() > 0) { /* père */
    close(fd[1]);
    return (fd[0]);
  } else { /* fils */
    close(fd[0]);
    dup2(fd[1],1);
    close(fd[1]);
    execlp("sh","sh","-c",cmd,NULL); /* on utilise sh pour parser argv */
  }
}
```

Fin réponse

▷ **Question 2:** Que se passe-t-il si l'on ferme les descripteurs retournés par votre fonction avec `close` ? Proposez une solution à ce problème dans une fonction `mypclose`.

Réponse

Le fils reste zombie vu que personne n'a fait wait.

Il faut une table globale associant fd et pid. Quand on ferme, on fait le waitpid (pour éviter les interactions avec les autres fils de l'utilisateur), puis enfin on ferme le descripteur.

Fin réponse

★ **Exercice 5: Processus et tubes** Considérez le programme suivant. Il crée une série de processus ouvrant des tubes entre eux.

CORRECTION

▷ **Question 1:** Dessinez l'organisation des processus et tubes telle qu'elle est en ligne 26. Il ne vous est pas demandé de dessiner l'historique menant à cette situation (comme à l'exercice précédent), mais bien la situation à l'instant où la ligne 26 est exécutée (en supposant que tous les processus exécutent cette ligne en même temps). Pensez à indiquer quel processus est créé à quel ligne sur votre schéma.

▷ **Question 2:** Décrivez en une phrase ce que font les processus créés par ce programme.

```
 7 int main(int argc,char *argv[ ]) {
 8   int tubA[2], tubB[2], tubC[2];
 9   pipe(tubA); pipe(tubB); pipe(tubC);
10   int n1=0,n2=0;
11   close(0); close(1);
12
13   if ((n1=fork())) {
14     dup2(tubA[0],0);    dup2(tubB[1],1);
15   } else if ((n2=fork())) {
16     dup2(tubB[0],0);    dup2(tubC[1],1);
17   } else {
18     dup2(tubC[0],0);    dup2(tubA[1],1);
19     printf("A");
20     fflush(stdout);
21   }
22   close(tubA[0]); close(tubA[1]);
23   close(tubB[0]); close(tubB[1]);
24   close(tubC[0]); close(tubC[1]);
25
26   /* Dessinez l'état des lieux à ce point */
27
28   char c;
29   while ((read(0,&c,1))>0) { /* Cette boucle ne s'arrête jamais */
30     write(1,&c,1);
31   }
32 }
```

Réponse

Les processus s'organisent en "ronde" et se font passer des caractères.

Le père lit dans tubA et écrit dans tubB.

Le fils lit dans tubB et écrit dans tubC.

Le fils du fils lit dans tubC et écrit dans tubA. Pour démarrer la boucle, il envoie A dans tubA (avec un printf).

Fin réponse

CORRECTION

CORRECTION