Université de Strasbourg

S31 Sujet N°4
Comprendre les Systèmes
gancarski@unistra.fr

IUT Robert Schuman Université de Strasbourg

Fichiers et pipes en C

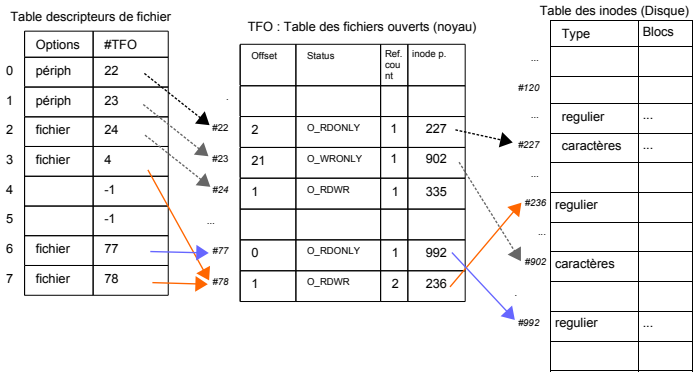
Objectif : Comprendre les entrées sorties en C

Notions : Création et utilisation de fichiers. Communication par pipes

1 Processus et fichiers

1.1 Descripteur de fichiers

Pour qu'un processus ait accès au contenu d'un fichier, il faut qu'il connaisse son implantation physique (par exemple son inode sur le disque). Pour faire le lien entre ce processus et la table des inodes, Unix associe à chaque processus un table de descripteurs de fichier locale à ce processus.



Chaque entrée de cette table pointe sur une entrée (un index) dans une table globale (dans le noyau) qui donne 4 informations :

- l'offset : la position de lecture/écriture (i.e., le nombre de caractères déjà lus/écrits)
- le statut : les droits
- le nombre de références : le nombre de fois où le fichier est ouvert
- l'inode du fichier (ou autre information permettant de trouver le fichier physiquement)

Appels système liés à la gestion de fichiers en C

- int **open**(const char *pathname, mode_t mode) : Ouvre un fichier avec possibilité de le créer si demandé / **close**(int filedescriptor) ;
- ssize_t **read**(int fd, void *buf, size_t count) / **write**(int fd, const void *buf, size_t count) ;

Exemple :

```
char c = 'a';
 // Création et ouverture d'un fichier :
int fd1 = open("fic1.txt", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
write(fd1, &c, 1); // Ecriture d'un caractère dans le fichier
```

Chaque fois qu'un processus ouvre un fichier par `open`, un descripteur lui est affecté et est renseigné dans la table. L'index de la case du tableau est retourné.

→ c'est ce descripteur qui sera utilisé pour lire et écrire dans le fichier par `read` et `write`

Complétez le schéma suite à l'exécution de ce code sachant que l'inode qui a été associé au fichier `fic1.txt` est 120. Remarque : il y a plusieurs possibilités.

1.2 Entrées/Sorties en C

Soient les deux variables `char c` et `char buf[100]`. Soient les codes suivant :

1	2	3	4

`for(int i = 0; i<10;i++){`
`  read(0,&c,1);`
`  write(1,&c,1);`
`}`
	`read(0,buf,10);`
	`write(1,buf,10);`
		`while(read(fd1,&c,1)!=0){`
		`  write(1,&c,1);`
		`}`
			`do {`
			`  read(0,&c,1);`
			`  write(fd1,&c,1);`
			`} while (c != '\n');`

1.2.1 Pour chacun d'eux, dire ce qu'il fait exactement

1.2.2 Comparez les deux premiers codes : avantages et inconvénients respectifs.

2 Pipes et communication inter-processus

Pour communiquer entre eux deux processus peuvent utiliser des pipes. Un pipe est un fichier particulier dont le(s) descripteur(s) pointent sur une entrée dans une table commune (dans le noyau) aux processus. Ainsi quand un processus écrit l'autre peut lire immédiatement et réciproquement.
Il existe deux types de pipes

— Pipe non nommé : le fichier partagé est stocké en mémoire

— Pipe nommé : le fichier partagé est via une inode

2.1 Pipes non nommés

2.1.1 Création

La création d'un pipe non nommé se fait :

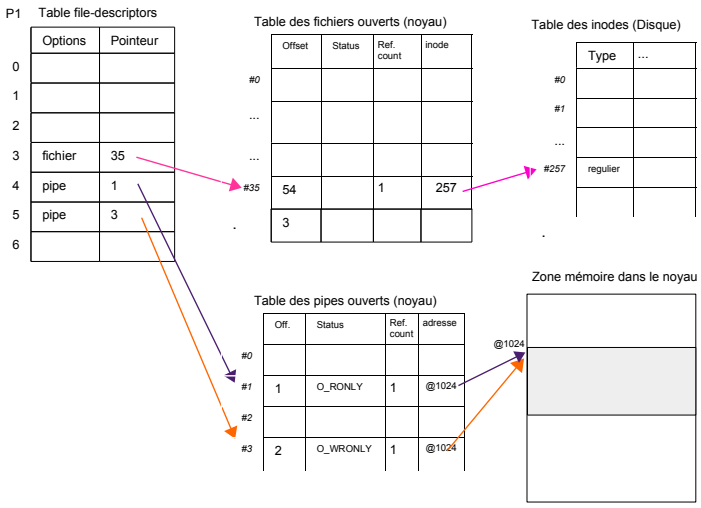
```
int fd[2];    //Les deux descripteurs de fichiers sur le pipe
int r = pipe(fd);  // r est le code d'erreur : 0 si tout s'est bien passé
```

où :

— fd[0] est le descripteur de fichier permettant de lire le pipe

— fd[1] est le descripteur de fichier permettant d'écrire dans le pipe

Ainsi sur le schéma montrant la situation après le pipe



on voit :

- qu'une zone mémoire dans le noyau a été allouée pour le pipe
- que deux entrées dans la table des pipes non nommés ont été associées au pipe
- que deux descripteurs ont été réservés : fd[0] = 4 et fd[1] = 5.

Un pipe étant un fichier la lecture et l'écriture se font par `read` et `write` mais ces deux opérations sont bloquantes :

- Si le pipe est vide, le processus est bloqué dans le `read` jusqu'à ce qu'il puisse lire le nombre de caractères demandés
- Si le pipe est plein, le processus est bloqué dans le `write` ce qu'il puisse écrire le nombre de caractères demandés

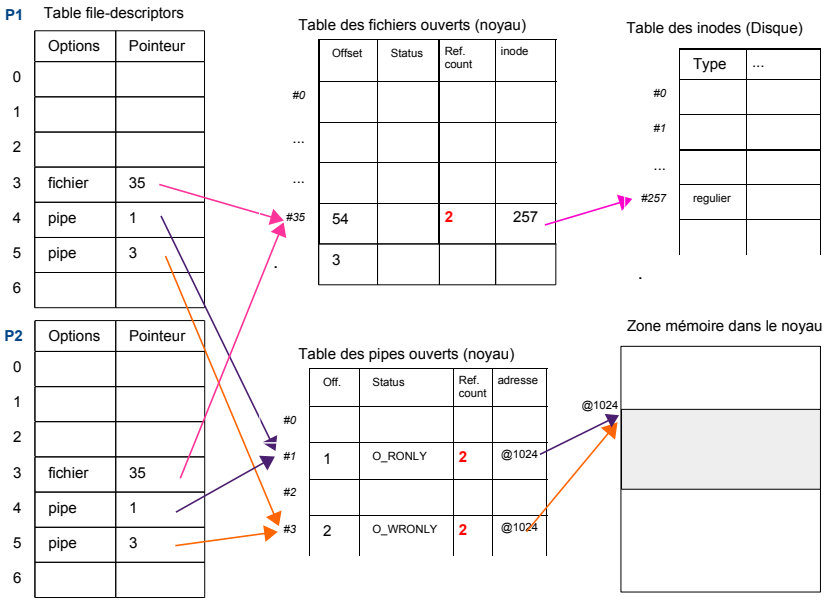
Un descripteur d'un pipe peut être fermé par un processus par `close` :

- Si d'autres processus utilisent le pipe, celui reste ouvert pour eux.
- Un processus ne peut pas rouvrir un descripteur qu'il a fermé

2.1.2 Partage du pipe

Pour accéder (lire/écrire) au pipe, il faudrait que les autres processus aient chacun deux descripteurs de fichier qui pointent sur les entrées de la table des pipes (dans l'exemple : #1 et #3). Or, il est impossible pour un processus de forcer un descripteur de fichier à pointer sur une entrée particulière de cette table.

Solution : Utiliser le fait qu'un `fork` recopie la table des descripteurs de fichiers.



2.2 Premiers pas

Le fichier `td4_ex2.2 (squelette).c` contient le code suivant :

```
int main(int argc, char *argv[]){
   char c;
   ... fd[2];    // Pipe entre le père et le fils

  // Création du pipe
  pipe(...);
  if (fork() == 0) {
    ...
    write( ... , ... , ...);
    printf("Caractère écrit par le fils : %c \n",c); }
  else {
    read( ... , ... , ... );
    printf("Caractère lu par le père : %c \n",c);
  }
}
```

2.2.1 - Compléter ce programme (remplacer les ...) pour que la sortie ressemble à ceci :
`Caractère écrit par le fils : a`
`Caractère lu par le père : a`

2.2.2 - Modifier ce programme pour que le fils envoie la chaîne de caractères (Max : 100 caractères) passée en argument de l'appel. La sortie correspondant à `a.out "J'adore ce TD"` doit ressembler à ceci :
`Texte écrit (14 caractères) par le fils : J'adore ce TD`
`Caractères (14 caractères) lus par le père : J'adore ce TD`
Le processus fils écrit les caractères d'un coup, le processus père écrit les caractères un après l'autre

2.2.3 - Écrire un programme qui permet au processus fils d'envoyer un entier (inférieur à 10^{10}) et au père de le lire

2.2.4 - Modifier le programme précédent en supposant que la taille maximale de la chaîne de caractères envoyée par le fils n'est plus bornée (le père ne sait pas a priori quelle peut être la taille de celle-ci).

2.3 Echanges entre processus

On suppose que chacun des deux processus doit faire une boucle "infinie" consistant à envoyer deux entiers à l'autre processus puis à lire les deux entiers venant de l'autre processus.

2.3.1 - Quel est le problème ?

2.3.2 - Comme résoudre ce problème ?

2.3.3 - Implantez cette solution en TP

3 Serveur de calcul

Il semblerait qu'avec ce savoir, vous pouvez sans problème produire la version finale V2b. On précise que dans cette version, c'est le serveur qui affiche le résultat.

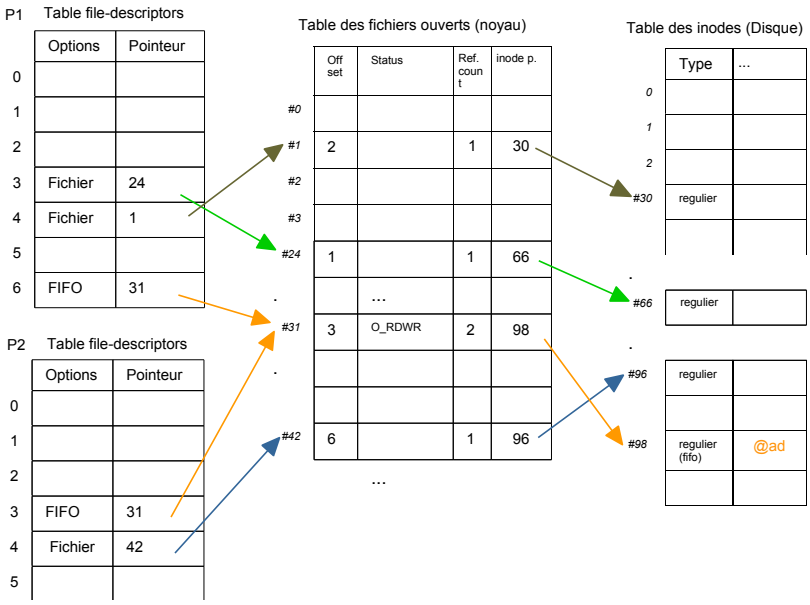
4 Pipes nommés

Contrairement aux pipes non nommés, un pipe nommé nécessite une inode car il est créé et géré par le système de fichier

- `int mknod(const char *pathname, mode_t mode, dev_t dev);`
- `int mkfifo(const char *pathname, mode_t mode);`

L'appel système `mkfifo` crée un fichier spécial FIFO (tube nommé) semblable à un tube (pipe) sauf qu'un tel fichier FIFO est inséré dans le système de fichiers : n'importe quel processus peut l'ouvrir en lecture ou écriture (en fonction des droits UNIX classiques). Par défaut :

- Les deux extrémités doivent être ouvertes avant de pouvoir effectuer une opération d'écriture ou de lecture : Si un processus qui vient d'ouvrir le tube nommé écrit dans le tube avant qu'il n'y ait de lecteur (qu'un processus ait ouvert le tube en lecture), cela engendre un signal SIGPIPE (tube détruit) et donc un erreur !
- L'ouverture du tube en lecture est bloquante tant qu'il n'y a pas ouverture en écriture et vice versa (le producteur et le consommateur sont alors synchronisés)



Un pipe nommé peut être ouvert, lu ou écrit comme un fichier ordinaire via les appels systèmes de bas niveau (`open, read, write`) ou les fonctions de la librairie C (`fopen, fscanf, fprintf`)

4.1 - Complétez ces 2 programmes (P1 ne pourra envoyer que des messages inf. à 100 caractères)

- P1.c

```
if (mkfifo("./FP1.pipe",                    ...                    ) == -1) {
      if (errno == EEXIST)   { /* nop : c'est juste que le fichier existe déjà */ }
     else {  perror("Création du fichier de pipe nommé : "); exit(errno); }
};
 int FP1 = open(... , ... );      // ouverture en écriture seule
char * c = "Hello";  write(... , ... ,...);
close( ... );
}
```

- P2.c

```
if (mkfifo("./FP1.pipe",                    ...                    ) == -1) {
      if (errno == EEXIST)   { /* nop : c'est juste que le fichier existe déjà */ }
     else  {  perror("Création du fichier de pipe nommé : "); exit(errno); }
};
int FP2 = open( ... , ... );   // ouverture en lecture seule
 char d[...];
 read( ... , ... , ...);
 printf(" %d caratères lus : %s \n",  (int)strlen(d), d);
 close( ...);
}
```

4.2 - Reprendre le programme de l'exercice 3.2b et le modifier de façon à ce que les pipes utilisés soient des pipes nommés

Remarque : Lorsque les processus échangent de données via le FIFO, le noyau transmet toutes les données sans les écrire dans le système de fichiers. Ainsi, le fichier spécial FIFO n'a pas de contenu réel dans le système de fichiers ; l'entrée du système de fichiers sert simplement comme point de référence afin que les processus puissent accéder au pipe en utilisant une entrée dans le système de fichiers