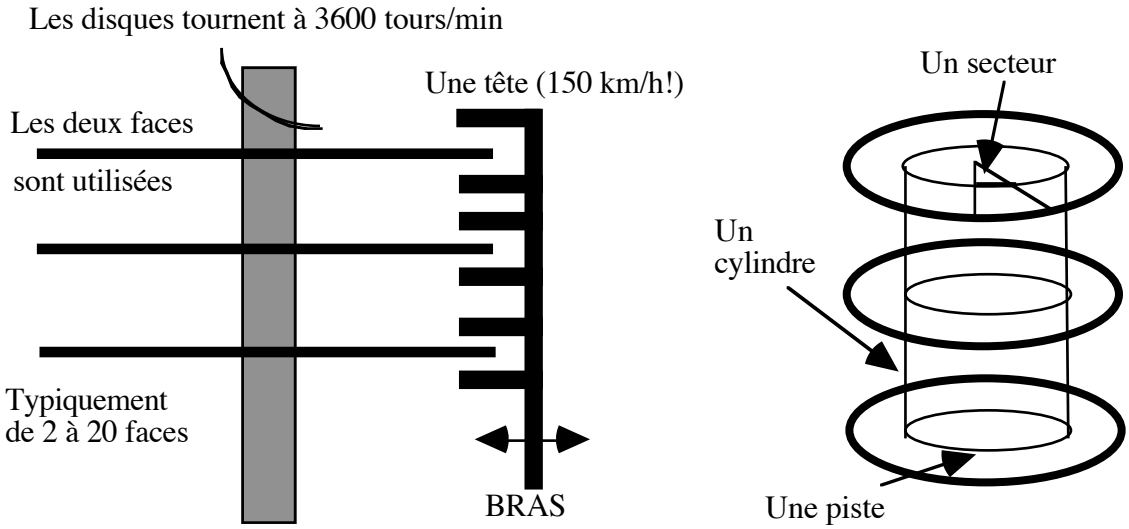
ED 6
Gestion des disques et des fichiers

Exercice 1 :
Comparaison des politiques de gestion du disque



On se propose d'étudier diverses politiques de déplacement du bras, dans le service d'un ensemble de requêtes à un disque. A un instant donné, la file d'attente des requêtes est définie par le tableau suivant :

N° de cylindre demandé	20	16	13	40	12	1	11
Ordre d'arrivée	1	2	3	4	5	6	7

Supposant qu'aucune autre requête n'arrive pendant le service des requêtes en attente et que la position initiale du bras est sur le cylindre 15, on étudiera les politiques suivantes : service à l'ancienneté (FCFS), service sur le cylindre le plus proche (SSTF), service selon la stratégie de l'ascenseur (SCAN) sens initial montant.

Question 1

Rappeler le fonctionnement des différentes politiques.

Question 2

Les résultats des requêtes sont, par souci de cohérence, délivrés dans l'ordre d'arrivée de celles-ci.

Donner l'ordre de service des requêtes et le nombre de cylindres parcourus.

Pour chaque requête et pour les trois stratégies, donner en nombre de cylindres parcourus, la date de délivrance du résultat.

Exercice 2 : Gestion de fichiers UNIX

Un processus Unix veut lire séquentiellement un fichier *Toto* de 8 Mo stocké sur disque, à raison de 256 octets à la fois. Pour localiser un fichier sur disque, le système dispose des tables suivantes :

• une table, que nous appelons TABDIRECT, donnant les numéros de blocs pour les 10 premiers blocs de l'objet externe,

• un numéro de bloc, que nous appelons INDIRECT_1, qui contient les numéros des p blocs suivants de l'objet externe,

• un numéro de bloc, que nous appelons INDIRECT_2, qui contient les p numéros de blocs contenant les numéros des p^2 blocs suivants de l'objet externe,

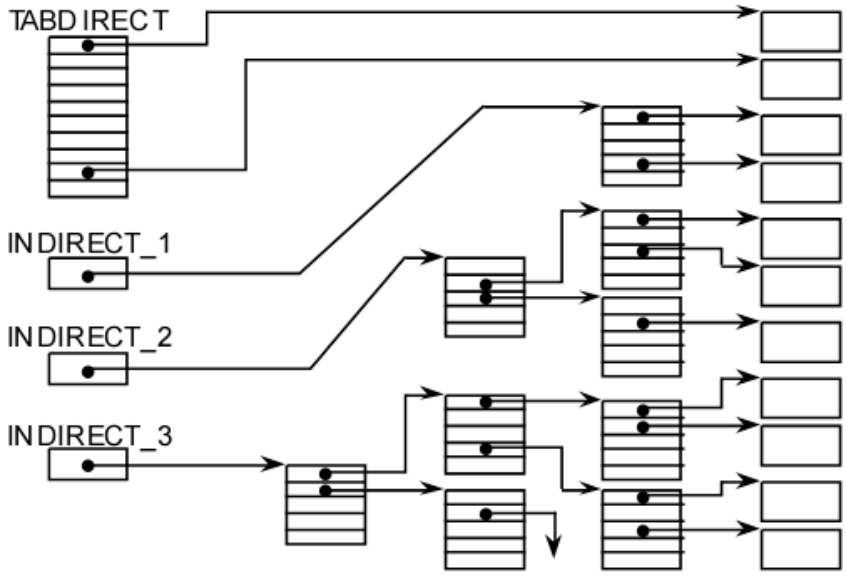
• un numéro de bloc, que nous appelons INDIRECT_3, qui contient les p numéros de blocs contenant, au total, p^2 numéros de blocs contenant les numéros des p^3 blocs suivants de l'objet externe.

Avec des blocs de 1024 octets et un numéro de bloc de 4 octets, le paramètre p est égal à 256. Par ailleurs.

Le processus qui veut lire le fichier *Toto* fera donc 32768 demandes de lecture successives.

On suppose qu'il n'y a qu'un seul processus dans le système, que le temps d'accès moyen au disque est de 40 ms, et que le système n'utilise pas de tampons de bloc disque, ce qui implique que chaque fois qu'une information située dans un bloc disque est nécessaire, ce bloc doit être lu depuis le disque.

Évidemment le descripteur d'un fichier ouvert, c'est-à-dire les informations TABDIRECT, INDIRECT_1, INDIRECT_2 et INDIRECT_3, restent en mémoire centrale.



1- *Décrire ce qui se passe lors des deux premières demandes de lecture de 256 octets, puis lors de la 5^{ième} demande.*

2- *Décrire ce qui se passe lors des 41^{ième} et 45^{ième} demandes de lecture de 256 octets.*

3- *Décrire ce qui se passe lors des 1065^{ième} et 1066^{ième} demandes de lecture de 256 octets.*

4- *Décrire ce qui se passe lors des 2089^{ième} et 2090^{ième} demandes de lecture de 256 octets.*

5- *En déduire le nombre total d'accès disque nécessaires et le temps d'attente d'entrées-sorties.*

Exercice 3 : Duplication de descripteurs d'entrée-sortie

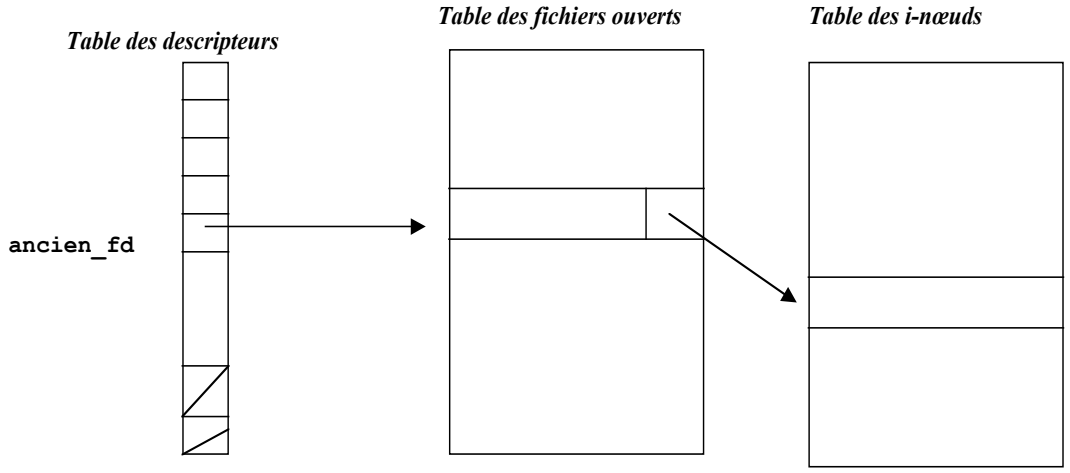
Unix dispose d'une fonction système *dup*() qui permet de dupliquer les descripteurs de fichiers.

```
#include <unistd.h>
```

- `int dup(int ancien_fd);`

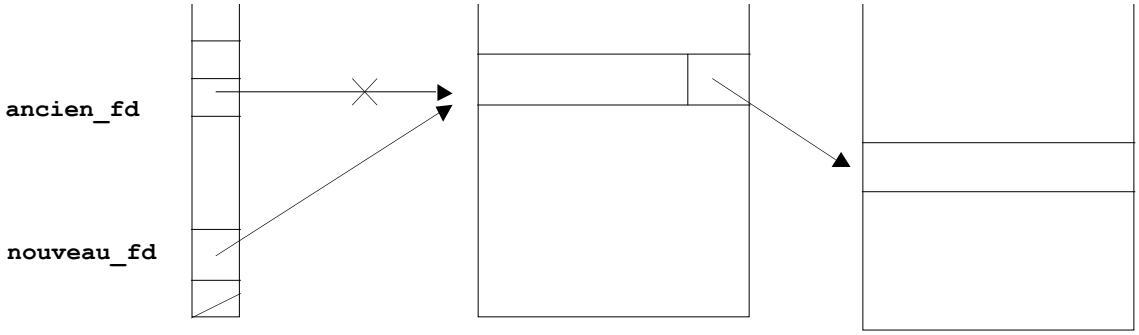
`dup` duplique le descripteur `ancien_fd` et retourne un autre descripteur associé au même fichier ouvert et -1 en cas d'erreur.

Le schéma suivant montre l'état des tables systèmes avant exécution de la primitive `dup(ancien_fd)`.



L'état des tables est le suivant après exécution de l'instruction :

`nouveau_fd=dup(ancien_fd) ;`



L'objectif du programme suivant est de créer deux processus P1 et P2. Ces deux processus ont pour rôle de réaliser la commande : « ls –l | wc –l ». Pour cela, P1 doit exécuter « ls –l » et P2 doit exécuter « wc –l ».

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
int fd;

if(fd=open("Temp",O_RDWR) == -1) {perror("Erreur dans l'ouverture de
Temp"); exit(1); }

switch(fork())
{ case -1: {perror("erreur sur fork"); exit(2); }
  case 0 : /* processus fils */
          /*fermer stdout pour rediriger la sortie standard sur le
fichier*/
           close(STDOUT_FILENO);
           dup(fd);   /* fd pointe sur la std out*/
      close(fd); /*l'ancien fd ne sert plus =>on peut le fermer*/
            execlp("ls", "ls", "-l", NULL);
            perror("ERREUR execlp par le fils");
            exit(2);
  default : /* processus pere */
              /*fermer stdin pour rediriger l'entree standard sur le
fichier*/
              close(STDIN_FILENO);
              dup(fd);  /* fd pointe sur la std in */
      close(fd); /*l'ancien fd ne sert plus =>on peut le fermer*/
            execlp("wc", "wc", "-l", NULL);
            perror("ERREUR execlp par le pere");
            exit(2);
    }
}
```

1- *Expliquer la commande : ls -l | wc -l*

Etant donné que P2 a besoin du résultat d'exécution de P1 pour lancer sa commande, la programme crée un fichier temporaire temp qui servira de pipe sans toutefois obliger les processus à utiliser explicitement le fichier temporaire.

2- *Expliquer comment le programme procède pour exécuter ces commandes en agissant uniquement sur les descripteurs de fichiers de chaque processus.*