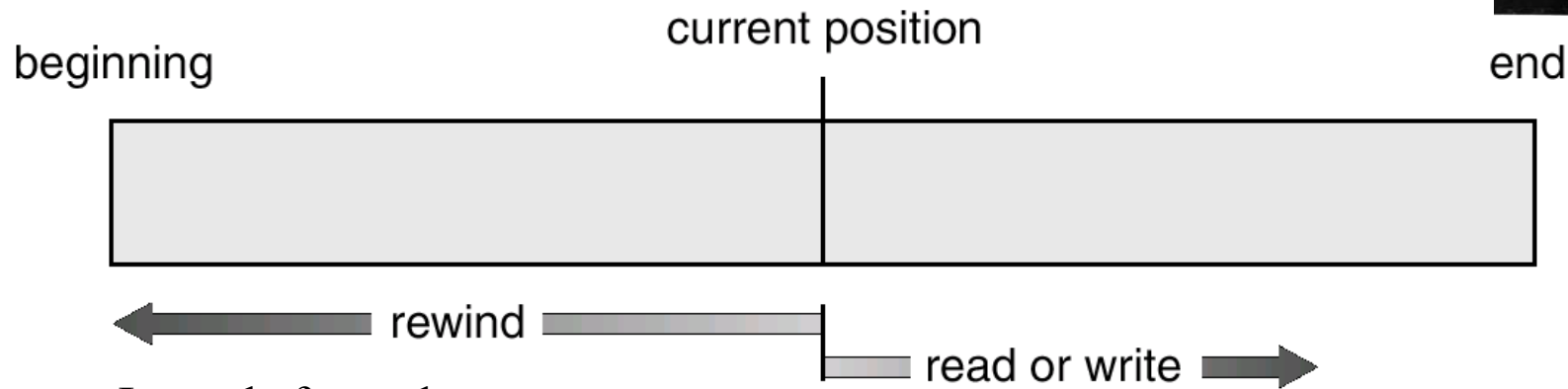
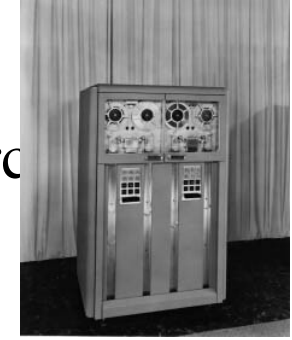


Programmation système

Laurent Mascarilla

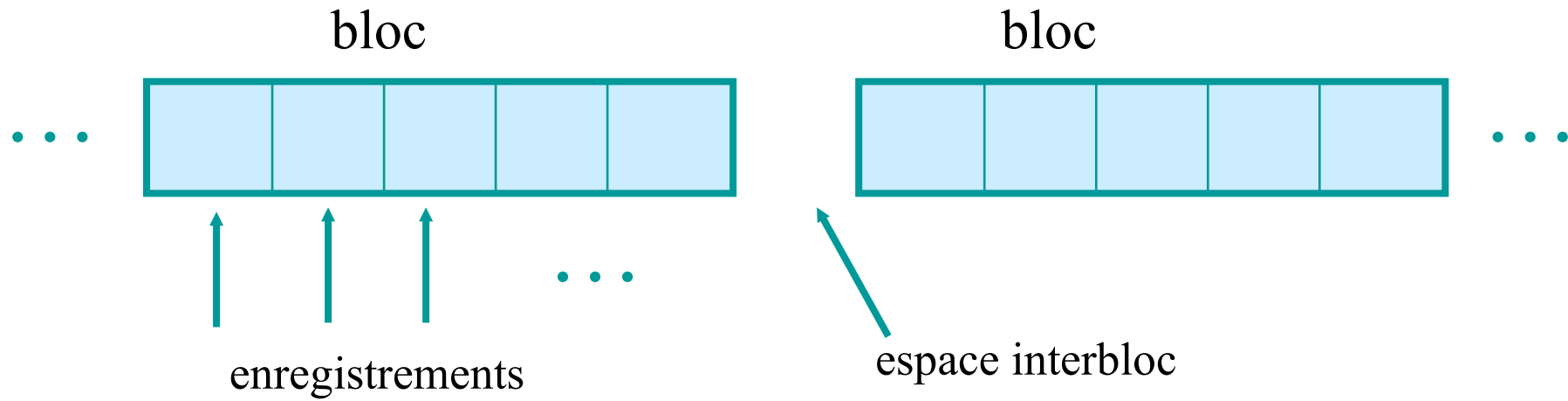
2019-2020

Fichiers à accès séquentiel (arcs rubans)



La seule façon de retourner en arrière est de retourner au début (rembobiner, rewind)

En avant seulement, 1 seul enreg. à la fois



BIBLIOTHEQUE E/S

- Appels système: read(), write(), open(), close()
 - accès par les descripteurs
 - définition des zones de réception/émission
 - nb d 'octets transmis
 - pas de formatage des E/S, pas de conversion
- Appels fonctions en bibliothèque stdio.h
 - gestion tampon utilisateur
 - réduction du temps UC /appel système

- Principe du tampon
 - en lecture:
 - tampon se remplit par une lecture
 - les caractères sont récupérés dans ce tampon jusqu'à épuisement
 - en écriture
 - les caractères remplissent le tampon
 - le tampon plein est vidé par une écriture
 - Mémoire cache avant écriture disque
- Structure FILE dans stdio.h pour la gestion du tampon
 - pointeurs du tampon du fichier
 - n° du descripteur

- Bibliothèque stdio.h

- fopen()
- fread()
- fwrite()
- fclose()
- fflush()
-

Déjà vue en cours de C : manipulent
des **FILE * aussi appelés flux**

Opérations de base (système) sur un fichier

- Ouverture d'un fichier : **open**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *chemin,
         int typeOpen,
         mode_t droitsFic);
```

Descripteur du fichier
ou **-1** en cas d'erreur

Nom du fichier à ouvrir

Mode d'ouverture :

O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT

Si création : droits appliqués à la
création (p. ex. 0750)

- **O_APPEND**

Le fichier est ouvert en mode « ajout »: la tête de lecture/écriture est placée à la fin du fichier

- **O_CREAT**

Créer le fichier s'il n'existe pas.

- **O_EXCL**

S'assurer que cet appel crée le fichier : si cet attribut est spécifié en conjonction avec O_CREAT et si le fichier existe déjà, open() échouera.

- **O_TRUNC**

Si le fichier existe, est un fichier régulier, et est ouvert en écriture (O_RDWR ou O_WRONLY), il sera tronqué à une longueur nulle.

-

Peuvent être combinés : **O_RDWR | O_CREAT**

mode/flags

Mode fopen()	Flags open()	Effet
r	O_RDONLY	Lit à partir du b=début
w	O_WRONLY O_CREAT O_TRUNC	Écrit à partir du début du fichier Créé ou efface (si existant)
a	O_WRONLY O_CREAT O_APPEND	Écrit à la fin. Crée le fichier s'il n'existe pas
r+	O_RDWR	Lit et écrit à partir du début Contenu préservé puis écrasé
w+	O_RDWR O_CREAT O_TRUNC	Lecture+écriture. Fichier créé sinon effacé
a+	O_RDWR O_CREAT O_APPEND	Lecture+ajout. Fichier créé sinon contenu préservé Si on lit : au début, si on écrit : à la fin. Toujours une seule tête lecture/écriture.

- Création d'un fichier : **creat**

creat() est équivalent à open() avec l'attribut flags égal à **O_CREAT | O_WRONLY | O_TRUNC**.

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

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

```
int creat(const char *ref,  
          mode_t droitsFic);
```

Nom du fichier à créer

Droits appliqués à la création

Descripteur ou -1 en cas
d'erreur

- Lecture dans un fichier : **read**

```
#include <unistd.h>  
int read(int desc,  
        void *buffer,  
        size_t nb);
```

Descripteur du fichier

Adresse de la variable dans laquelle on stocke le résultat de la lecture

Nombre d'octets à lire

Nb d'octets lu
-1 en cas d'erreur
0 si fin de fichier

Attention : **read** lit et déplace le curseur !

Opérations de base sur un fichier

- Ecriture dans un fichier : **write**

```
#include <unistd.h>  
int write(int desc,  
          void *buffer,  
          size_t nb);
```

Descripteur du fichier

Adresse de la variable qui
contient ce que l'on veut écrire

Nombre d'octets à écrire

Nb d'octets écrits
-1 en cas d'erreur

Attention : **write** écrit et déplace le curseur !

- Déplacement de la position courante : **lseek**

```
#include <unistd.h>  
off_t lseek(int desc,  
             off_t depl,  
             int vers);
```

Descripteur du fichier

Valeur du déplacement (en octet)
par rapport à *vers*

Nouvelle position par
rapport au début du
fichier
-1 en cas d'erreur

- **SEEK_SET** : déplacement par rapport au début du fichier
- **SEEK_CUR** : déplacement par rapport à la position courante
- **SEEK_END** : déplacement par rapport à la fin du fichier

- Fermeture : **close**

```
#include <unistd.h>  
int close(int desc);
```

-1 en cas d'erreur
0 si ok

descripteur du fichier à fermer

- Dupliquer un descripteur de fichier : **dup** et **dup2**

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

```
int dup(int oldfd);
```

```
int dup2(int oldfd, int newfd);
```

Ancien descripteur à dupliquer

nouveau descripteur

dup et dup2 renvoient le nouveau descripteur, ou -1 s'ils échouent, auquel cas errno contient le code

- **dup** et **dup2** créent une copie du descripteur de fichier oldfd.
- Après un appel réussi à **dup** ou **dup2**, l'ancien et le nouveau descripteurs peuvent être utilisés de manière interchangeable. Par exemple si le pointeur de position est modifié en utilisant **lseek** sur l'un des descripteurs, la position est également changée pour l'autre.
- **dup** utilise le plus petit numéro inutilisé pour le nouveau descripteur.

- **dup2** transforme newfd en une copie de oldfd, fermant auparavant newfd si besoin est.
- Par défaut un processus dispose de
 - stdin dont le fd est 0
 - stdout dont le fd est 1
 - Stderr dont le fd est 2

- Association flux / descripteur de fichier : **fdopen**

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

```
FILE *fdopen (int desc, const char *mode);
```

("r", "r+", "w", "w+", "a", ou "a+")
doit être compatible avec celui
du descripteur de fichier

descripteur de fichier ouvert

- Association descripteur de fichier/flux : **fileno**

```
#include <unistd.h>  
int fileno (FILE *flux);
```

Flux ouvert

descripteur de fichier
ou -1 en cas d'erreur

- Liens matériels (hard links) entre fichiers

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

```
int link (char* nom_original, char* nom_nouveau);
```



Fichier existant



0
ou -1 en cas d'erreur

Le « nouveau » correspond à un nom différent mais au même inode que « original » : c'est le même contenu
D'où la notion de **compteur de lien** (incrémenté à chaque nouveau lien)


En shell :

```
$ ln nom_original nom_nouveau
```

- Liens matériels (hard links) entre fichiers

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

```
int unlink (char* nom_fichier);
```



0
ou -1 en cas d'erreur

Le **compteur de lien** est décrémenté et le fichier détruit quand il est à 0.

```
En shell :  
$ rm nom_fichier
```

- Liens symboliques (soft links) entre fichiers

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

```
int symlink (char* nom_original, char* nom_nouveau);
```

Fichier existant

0
ou -1 en cas d'erreur

Ici le nouveau nom contient seulement le chemin vers l'original :

- On peut effacer le nouveau sans affecter l'original.
- Si l'original est effacé, le nouveau existe toujours mais indique un fichier qui n'existe plus...

Plus souple (fonctionne même entre partitions distantes)

En shell :

```
$ ln -s nom_original nom_nouveau
```

Répertoire

Nom fichier	i-noeud
.	602213
..	649297
un	649298
deux	649300
lien_sur_un	649298
lien_sur_deux	649300
autre_lien_sur_deux	649300

Table des i-nœuds

...	
649298	●
649299	
649300	●
...	

(copie de /etc/host.conf)

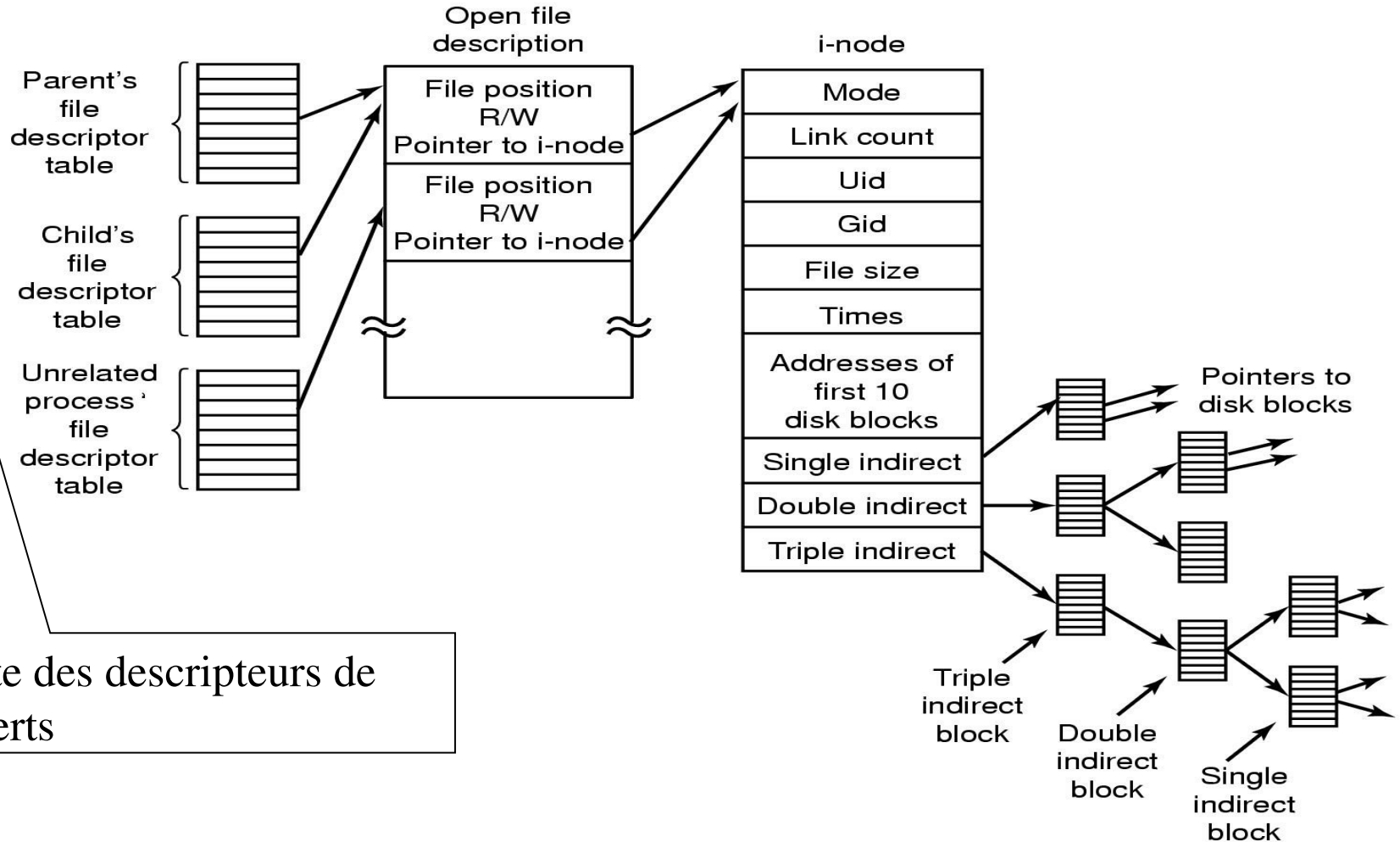
(copie de /etc/services)

```
$ cp /etc/services ./un
$ cp /etc/host.conf ./deux
$ ln un lien_sur_un
$ ln deux lien_sur_deux
$ ln deux autre_lien_sur_deux
```

d'après C. Blaes

Partage de fichiers entre processus père et fils

- Le fork duplique la table des descripteurs de fichiers du processus père.



Le fils hérite des descripteurs de fichiers ouverts

Exemple: partage de fichiers père/fils

```
int outfd = open("fileout", O_RDWR+O_CREAT);
int infd = open("filein", O_RDWR);
int outfile = fdopen(outfd, "w");
```

filein :
abcdefghi

```
fprintf(outfile, "123");
write(outfd, "456", 3);
read(infd, buffer, 3);
```

```
printf("Before fork read %s\n", buffer);
int p = fork();
```

```
if (p > 0) {
    write(outfd, "p7p8", 4);
    fprintf(outfile, "p9\n");
    read(infd, buffer, 3);
    printf("Parent after fork read %s\n",
buffer);
```

```
} else {
    write(outfd, "c7c8", 4);
    fprintf(outfile, "c9\n");
    read(infd, buffer, 3);
    printf("Child after fork read %s\n",
buffer);
}
```


Exemple: partage de fichiers

```
int outfd = open("fileout", O_RDWR+O_CREAT);  
int infd = open("filein", O_RDWR);  
int outfile = fdopen(outfd, "w");
```

filein :
abcdefghi

```
fprintf(outfile, "123");  
write(outfd, "456", 3);  
read(infd, buffer, 3);
```

```
printf("Before fork read %s\n", buffer);  
int p = fork();
```

```
if (p > 0) {  
    write(outfd, "p7p8", 4);  
    fprintf(outfile, "p9\n");  
    read(infd, buffer, 3);  
    printf("Parent after fork read %s\n",  
buffer);  
}
```

```
else {  
    write(outfd, "c7c8", 4);  
    fprintf(outfile, "c9\n");  
    read(infd, buffer, 3);  
    printf("Child after fork read %s\n",  
buffer);  
}
```

```
$ ./fork_file  
Before fork read abc  
Parent after fork read def  
Child after fork read ghi  
$ cat fileout  
456p7p8123p9  
c7c8123c9
```

Gestion des erreurs

perror - Afficher un message d'erreur système.

SYNOPSIS

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

```
void perror(const char *s);
```

```
#include <errno.h>
```

```
const char *sys_errlist[];  
int sys_nerr;  
int errno;
```

DESCRIPTION

affiche un message sur la sortie d'erreur standard, décrivant la dernière erreur rencontrée durant un appel système ou une fonction de bibliothèque

le numéro d'erreur est obtenu à partir de la variable externe **errno**, qui contient le code d'erreur lorsqu'un problème survient, mais **qui n'est pas effacé lorsqu'un appel est réussi.**

Exemple :

```
if ((fd = open(nom_fichier, mode)) == -1) {  
    perror("open");  
    exit(EXIT_FAILURE);  
}
```

En cas d'erreur affiche :

open: Aucun fichier ou répertoire de ce type