Licence 2 Informatique

SYSTÈMES 2

Support de cours/TD intégrés et de TD machine

Alain CROUZIL, Jean-Denis DUROU, Jean-Marc PIERSON

{crouzil,durou,pierson}@irit.fr

http://www.irit.fr/enseignement

Avant-propos

Objectif

Ce support est destiné à être utilisé durant les séances de cours/TD intégrés et de TD machine. Il ne constitue pas un support complet dans la mesure où, en séance, vous seront donnés des explications, des exemples, des illustrations et des compléments indispensables. Toutefois, il permet de préparer à l'avance les séances.

Utilisation

Les six premiers chapitres concernent plus spécifiquement les cours/TD intégrés et sont ponctués d'exercices. L'index situé à la fin du document constitue un outil indispensable pour accéder rapidement à la description d'une fonction à partir de son nom.

Signification des pictogrammes

Dans la marge ou près de certains titres, apparaissent les pictogrammes suivants :

✎ désigne un sujet d'exercice de cours/TD intégrés ;

✍ est un repère destiné à l'enseignant.

Bonne lecture !

Table des matières

Chapitre 1
Introduction

1.1 Rôle du système d'exploitation

Le système d'exploitation fournit à chaque utilisateur un environnement de travail adapté à ce qu'il veut faire, c'est-à-dire une machine virtuelle qui est une version simplifiée (abstraction) de l'architecture matérielle. Les fonctions de l'environnement d'exécution de la machine virtuelle sont :

- *la gestion de l'information* :
  - structuration des programmes, des données manipulées par ces programmes, ainsi que des objets manipulés par l'utilisateur ;
  - conservation des fichiers et leur catalogue ;
  - désignation des objets (comment on identifie un fichier, comment on le distingue d'un autre) ;
  - gestion de l'espace d'adressage (permet de stocker, dans la mémoire principale, l'information).
- *la gestion de la communication* pour permettre l'accès :
  - à des machines virtuelles éloignées ou non, à des machines réseaux, c'est-à-dire permettre l'accès à d'autres machines virtuelles, qu'elles soient locales ou distantes ;
  - à des supports externes (disques durs...), via des entrées-sorties ;
  - à des objets, des composants partageables ;
  - à des services ;
  - à des fichiers, qu'ils soient distants ou non.
- *la gestion de l'exécution* pour permettre :
  - de lancer des exécutions mais aussi de suivre ces exécutions de programmes, qu'elles soient faites en séquence, en parallèle, en concurrence sur l'accès à une ressource ;
  - de composer et d'enchaîner les programmes ;
  - de synchroniser des exécutions (quand on fait de la programmation parallèle).

1.2 Machines virtuelles et bon niveau d'abstraction

- *Niveau de l'application particulière* :
  - interaction Homme-machine (IHM), logiciels spécifiques
- *Niveau fichiers* :
  - langage de commande (ex : shell de Bourne) ;
  - interpréteur du langage de commande (ex : `/bin/sh`).
- *Niveau langage machine* :
  - création de l'exécutable et de son image en mémoire (sa représentation dans la mémoire) ;
  - processus : création, états, fin ;
  - entrées-sorties, interruptions.
- *Niveau langage évolué* : C, C++, Java...

1.3 Outils pour l'utilisation des services du système d'exploitation en langage C

Les services du système d'exploitation peuvent être invoqués :
- par l'intermédiaire d'une interface graphique ;
- grâce à un interpréteur de commande ;
- depuis un programme écrit dans un langage évolué.

Dans la suite de ce cours, les services du système Unix seront invoqués depuis des programmes écrits en langage C :
- de manière directe, en utilisant la bibliothèque Unix (l'interface de programmation POSIX) qui donne accès aux « primitives Unix » ;
- de manière indirecte, en utilisant la bibliothèque standard qui offre des outils portables sur tous les systèmes d'exploitation, à la seule condition de disposer d'un compilateur C ANSI.

1.3.1 Interface de programmation POSIX

1.3.1.1 Définition

POSIX (Portable Operating System Interface for unix) est une norme internationale d'« interfaçage » des programmes avec les systèmes d'exploitation. Elle est basée sur le système Unix. La première partie de cette norme, « System Application Program Interface (API) [C Language] », POSIX.1, décrit les fonctions utilisables dans un programme pour interagir avec le système d'exploitation, en utilisant la syntaxe du langage C. Ces fonctions constituent un sous-ensemble portable (95 fonctions) des fonctions présentes dans les bibliothèques « système » des différentes variantes d'Unix. La norme a repris et étendu les fonctionnalités de la bibliothèque standard (`stdio.h`, `time.h`, ...) et a apporté des fonctionnalités nouvelles (gestion de l'arborescence des fichiers, des processus, ...). Toutes ces fonctionnalités peuvent se regrouper en huit parties : paramètres du système (configuration, base de noms, ...), environnement d'un processus (état, temps d'exécution, ...), lancement et terminaison des processus, interaction avec la hiérarchie des fichiers, entrées-sorties de bas niveau (« primitives » d'entrées-sorties), signaux, tubes anonymes, interaction avec le terminal. Viennent se rajouter six parties optionnelles : entrées-sorties asynchrones, gestion des files de messages, processus légers (*threads*), ordonnancement des processus, sémaphores, gestion avancée de la mémoire.

1.3.1.2 Activation

Pour garantir que le compilateur C vérifie qu'un programme respecte la norme POSIX.1, il faut placer, *avant* les inclusions de fichiers d'en-tête (les fichiers d'extension `.h`), la directive :
```
#define _POSIX_C_SOURCE 1
```

1.3.2 Communication avec le système d'exploitation en C ANSI

Nous verrons plus loin comment faire appel indirectement au système de gestion des fichiers avec la bibliothèque standard du langage C. Mais il est aussi possible d'effectuer des interactions simples en utilisant cette bibliothèque.

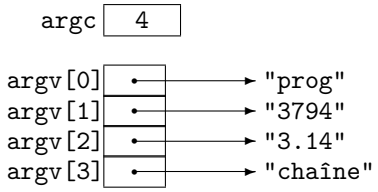
1.3.2.1 Paramètres d'un programme

Il est possible de récupérer les paramètres (arguments) qui sont passés au programme au moment du lancement de son exécution. Pour cela, il faut modifier l'en-tête de la fonction principale et utiliser :

`int main(int argc, char *argv[])`

où
- `argc` contient le nombre de paramètres effectivement passés au programme augmenté de 1 pour le nom du programme exécutable ;
- `argv` est un tableau de `argc` pointeurs vers les chaînes de caractères qui constituent les paramètres effectifs du programme.

Exemple : si `prog` est le nom du programme exécutable correspondant au fichier source `prog.c` et si on lance son exécution en tapant la ligne de commande : `prog 3794 3.14 chaîne`
alors on a :



Ces paramètres peuvent être utilisés dans le programme, sachant qu'il peut être nécessaire d'effectuer des conversions de chaînes de caractères en nombres grâce par exemple aux fonctions `atoi` ou `atof` (déclarées dans `stdlib.h`) ou bien à la fonction `sscanf`.
Exemple :

```
int n;
sscanf(argv[1],"%d",&n);
```

Sous Unix, l'interface POSIX permet de rajouter un paramètre à l'en-tête de la fonction `main` :
`int main(int argc, char *argv[], char *envp[])`
Le paramètre `envp` est un tableau de chaînes de caractères contenant les variables d'environnement au moment de l'exécution, sous la forme `variable=valeur`.

Toutefois, l'accès aux valeurs des variables d'environnement est plus simple si l'on utilise la fonction `getenv` qui est décrite à la page 10 et qui fait partie de la bibliothèque standard.

1.3.2.2 Lancement de commandes du système

La fonction
`#include <stdlib.h>`
`int system(const char *commande)`
permet de lancer l'exécution de la commande `commande`. Si le paramètre `commande` est égal à `NULL`, la fonction `system` retourne une valeur non nulle s'il existe au moins une commande du système qui lui est accessible. Si le paramètre `commande` est différent de `NULL`, la valeur de retour dépend de l'implémentation.

1.3.2.3 Sortie d'un programme

Il y a deux manières de terminer l'exécution d'un programme et de retourner au système un entier appelé *code de retour* :
- en utilisant, dans la fonction `main`, l'instruction
  `return code_retour;`
- en faisant appel, depuis n'importe où dans le programme, à la fonction
  `#include <stdlib.h>`
  `void exit(int code_retour)`

Remarques :

- L'exécution de l'opérateur `return` dans une fonction autre que `main` permet d'interrompre l'exécution de la fonction en question et de rendre la main à la fonction appelante.
- Les conventions des commandes Unix préconisent qu'un programme retourne 0 quand tout s'est bien passé ou un entier positif permettant de décrire une situation anormale.
- Sous Unix, la fonction `exit` fait appel à la primitive :
  `#include <unistd.h>`
  `_exit(int code_retour)`

✎ Exercice 1

écrire un programme C ANSI qui affiche à l'écran ses arguments (un argument par ligne).

1.3.2.4 Accès aux variables d'environnement

La fonction
`#include <stdlib.h>`
`char *getenv(const char *nom_var)`
retourne l'adresse du début de la chaîne de caractères correspondant à la valeur de la variable d'environnement `nom_var`. Si la variable n'existe pas, cette fonction retourne `NULL`.
Exemple : `printf("LOGNAME=%s\n",getenv("LOGNAME"));`
affiche le nom d'utilisateur (par exemple `l2inf200`).

✎ Exercice 2

écrire un programme C ANSI qui prend en paramètre le nom d'une variable d'environnement et qui affiche à l'écran sa valeur.

Remarque : sous l'interpréteur, on peut obtenir l'affichage des variables d'environnement et de leurs valeurs en lançant la commande `env`.

1.3.2.5 Gestion des erreurs

En cas d'erreur, les fonctions de la bibliothèque standard qui font appel indirectement au système (comme les fonctions de gestion des entrées-sorties sur les fichiers) ainsi que la plupart des fonctions de l'API POSIX positionnent la variable globale `errno`. Le fichier `errno.h` contient des messages d'erreur associés aux valeurs de la variable `errno`. La fonction de la bibliothèque standard :
`#include <stdio.h>`
`void perror(const char *message)`
permet d'afficher sur `stderr` la chaîne passée en paramètre suivie du caractère `:`, suivi éventuellement du message associé à l'erreur.

Exemple :

```
int fd;
fd=open("fich",O_RDONLY);
if (fd==-1) perror("fich");
```

Si le fichier `fich` n'existe pas, `perror` va afficher un message ressemblant à :
`fich : No such file or directory`
Remarque : la fonction `open` est décrite de manière détaillée à la page 22.

Chapitre 2
Processus

2.1 Définitions

Un processus est l'instance d'un programme mis sous forme exécutable en mémoire. C'est aussi le déroulement de l'exécution de cette instance par la machine virtuelle :

- niveau utilisateur ;
- niveau système.

Mais encore, c'est aussi un objet actif qui modifie son état et ses données en utilisant la machine virtuelle : un processus peut être représenté par son état, son contexte et son espace d'adressage.

2.2 Création d'un processus et exécution

✍ 1

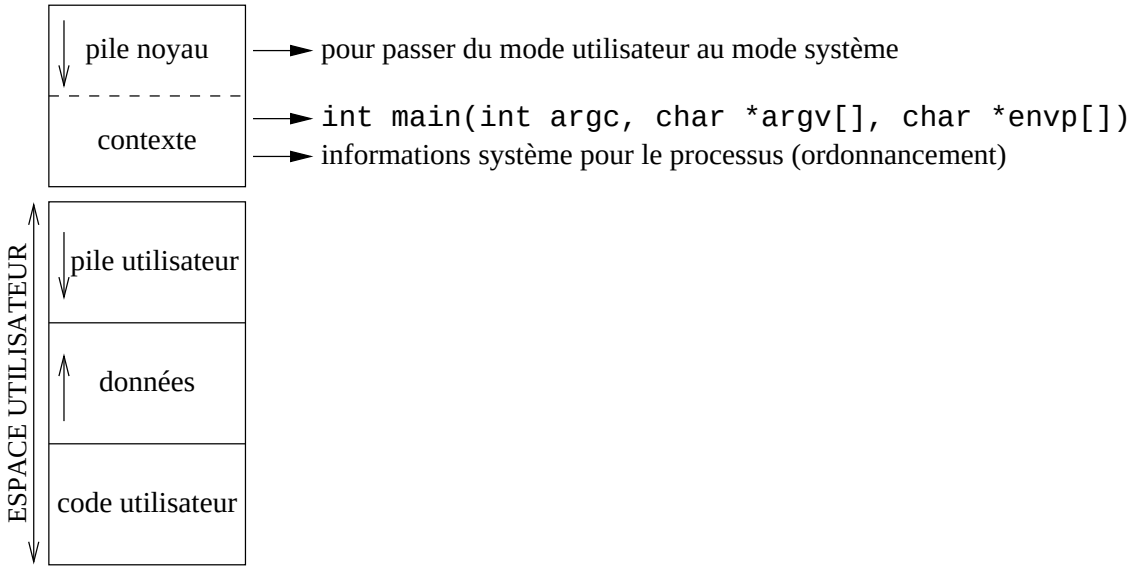
2.3 Image d'un processus sur disque

✍ 2

2.4 Image d'un processus en mémoire virtuelle

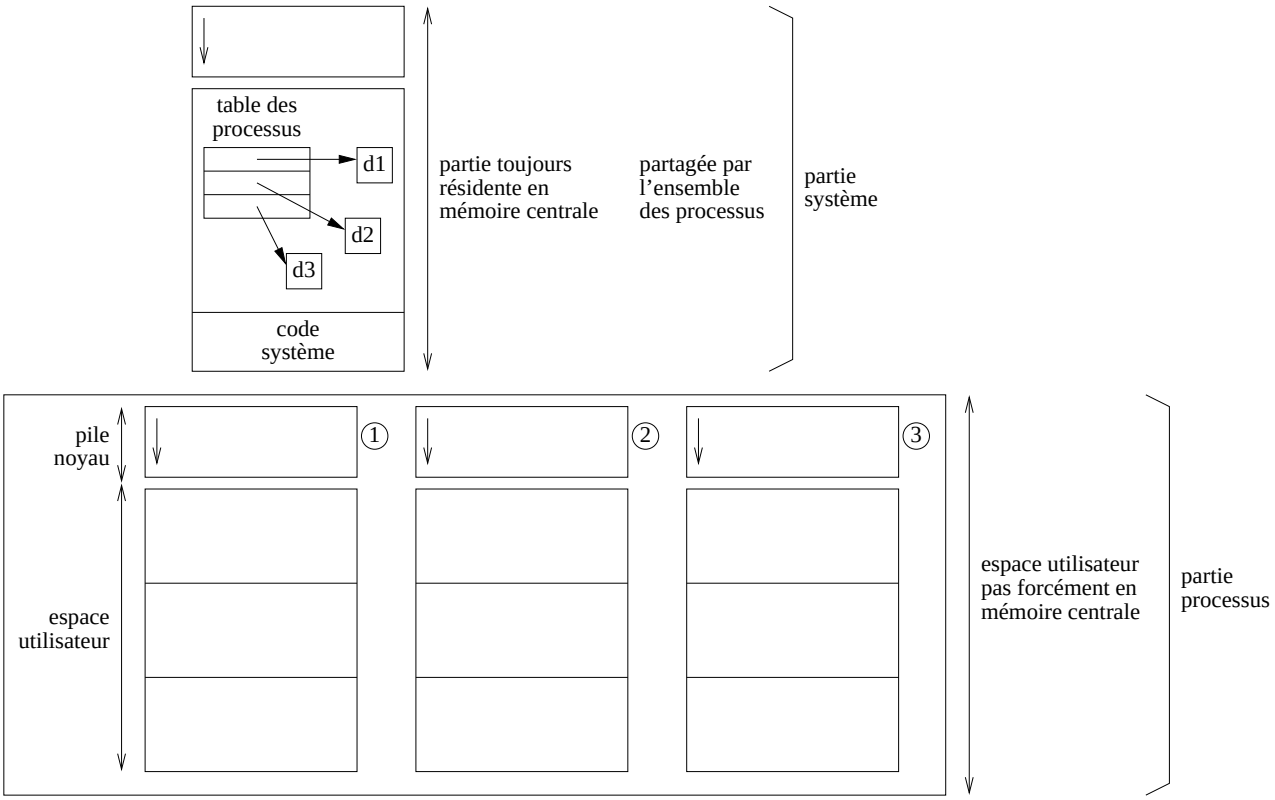
✍ 3

2.5 Espace d'adressage du processus vu d'un système



2.6 Description du processus (*process status*)

- identificateur du processus (`pid`) ;
- identificateur du créateur du processus (`ppid`) ;
- zone mémoire allouée + pointeur de pile d'exécution
- informations d'ordonnancement (priorités) ;
- valeur du compte temps.



- Les processus sont en concurrence.
- à un utilisateur sont associés plusieurs processus.
- Tout processus peut créer un processus $\Longrightarrow$ arborescence de processus : tout le système repose sur ce concept.

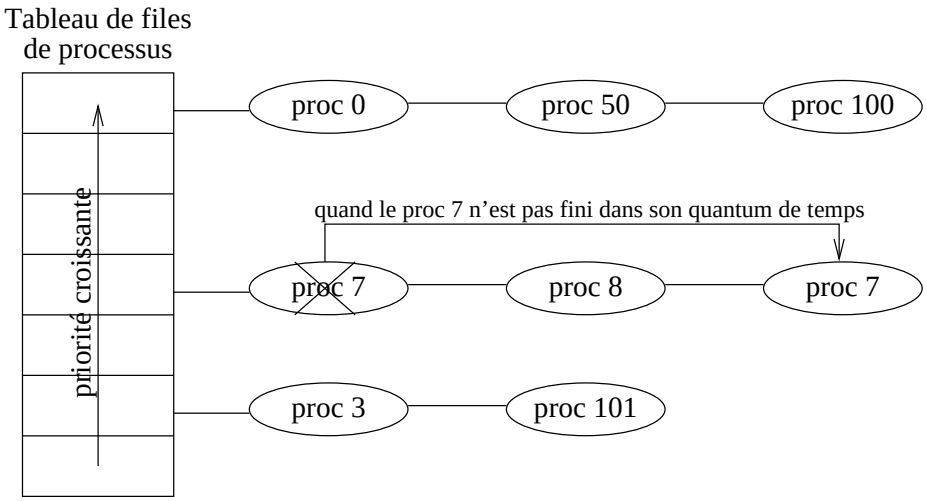
✍ 4

2.7 Ordonnancement

- Choix du processus en exécution (attribution du processeur) ;
- Pour combien de temps : notion de quantum de temps pendant lequel un processus peut s'exécuter.

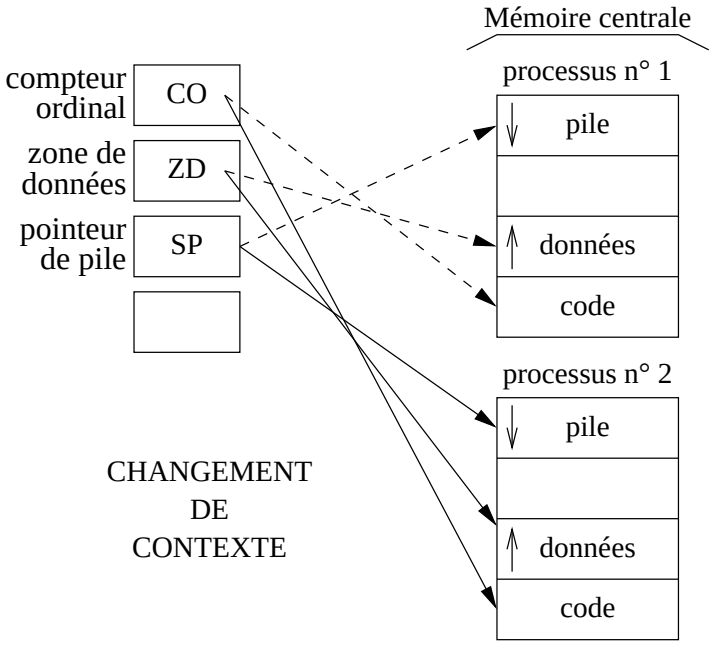
✍ 5

2.8 Files des processus prêts



Remarque : la priorité est fonction de l'utilisateur, du fait qu'il s'agit d'un processus système ou pas.

2.9 Registres de l'unité centrale



Lorsqu'un processus est élu, les registres de l'unité centrale reçoivent les valeurs du nouveau processus. Quand un processus est enlevé de l'unité centrale, il passe de l'état élu (actif) à un autre état (zombie, bloqué...) et son état est sauvegardé dans un PCB (*process control block*). Le PCB contient ce qu'il faut pour relancer un processus (quand il repasse dans l'état élu) :

- état du processus ;
- valeurs des registres de l'unité centrale (UC), dont le CO (adresse de la prochaine instruction à exécuter) ;
- informations sur l'ordonnancement du processus (priorité) ;
- informations sur la mémoire utilisée (notamment les pages utilisées) ;
- informations sur les entrées-sorties (E/S), comme les fichiers ouverts.

2.10 Activité de l'unité centrale

✍ 6

Remarque : les appels systèmes peuvent concerner :

- la gestion des processus : `fork`, `exec`, `malloc`...
- la gestion des fichiers (E/S) : `read`, `open`, `stat`, `fcntl`...
- la gestion du système de fichiers : `mkdir`, `chdir`, `mount`...
- la gestion des protections : `chmod`, `chown`...
- la gestion du temps : `times`...
- la gestion des signaux : `kill`, `sigaction`...

2.11 Gestion des processus avec les primitives de la bibliothèque Unix

2.11.1 Identification du processus courant

Un processus qui crée un autre processus est appelé le *processus père* et le processus créé est le *processus fils*. Un processus appartient à un groupe de processus. Un processus a un propriétaire qui

appartient à un groupe d'utilisateurs.

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

- `pid_t getpid(void)` retourne l'identificateur du processus courant (pid).
- `pid_t getppid(void)` retourne l'identificateur du processus père du processus courant (ppid).
- `pid_t getpgrp(void)` retourne l'identificateur du groupe (pgrp) de processus auquel appartient le processus courant.
- `char *getlogin(void)` retourne l'adresse d'une chaîne de caractères contenant le nom de l'utilisateur (login) connecté sur le terminal contrôlant le processus courant ou `NULL` si cette information n'est pas disponible.
- `uid_t getuid(void)` retourne l'identificateur de l'utilisateur (uid) propriétaire du processus courant.
- `gid_t getgid(void)` retourne l'identificateur du groupe d'utilisateurs (gid) du propriétaire du processus courant.

Remarques :

`pid_t`, `uid_t` et `gid_t` sont des synonymes de types entiers définis dans `sys/types.h`.

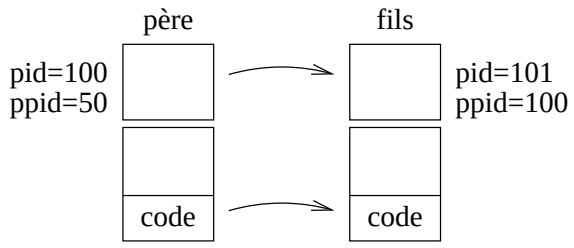
Vous pouvez obtenir votre identificateur et celui de votre groupe avec la commande shell `id`.

2.11.2 Création

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)
```

La fonction `fork` permet de créer un processus fils. Le processus créé (fils) est une copie quasi-conforme (un clone) du processus appelant (père). En effet, le fils hérite, par copie (duplication), de tout le contexte du père, c'est-à-dire de la plupart de ses attributs : environnement (`PATH`, `SHELL`...), priorité, numéro de groupe, segment de pile, segment de données, fichiers ouverts... Il partage le code du père. Le processus fils débute son existence dans la fonction `fork`. à partir de là, les deux processus s'exécutent « en parallèle ». Les principales différences entre les deux processus sont :

- leurs identificateurs,
- la valeur retournée par `fork` :
  - dans le contexte du père : la valeur -1 en cas d'échec, l'identificateur du fils sinon,
  - dans le contexte du fils : la valeur 0,
- la localisation des segments de pile et de données : ces segments sont privés ; à la création du processus fils, ils contiennent les mêmes informations ; par la suite, ils évoluent indépendamment,
- les descripteurs de fichiers (chaque processus possède ses propres descripteurs qui pourraient partager des ressources).

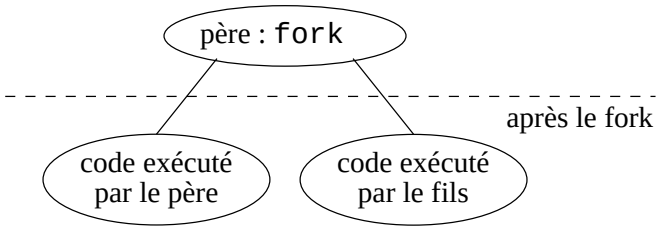


Après la création
les deux processus sont indépendants

Schémas classiques de création d'un processus fils

```
if ((pid=fork())==-1)
{ /* Code exécuté par le père
     en cas de problème      */
  ...
}
else if (pid)
{ /* Code exécuté par le père */
  ...
}
else
{ /* Code exécuté par le fils */
       ...
}
```

```
switch (pid=fork())
{
case -1: /* Code exécuté par le père
            en cas de problème      */
          ...
          break;
case 0 : /* Code exécuté par le fils */
          ...
          break;
default: /* Code exécuté par le père */
          ...
}
```



Remarque : la création du processus fils peut échouer par exemple si la table des processus est pleine (plus le droit de créer des processus) ou si la mémoire virtuelle est pleine.

✎ Exercice 3

écrire un programme qui crée un processus fils. Après la création, le processus père doit afficher son propre identificateur. Le processus fils doit afficher l'identificateur de son père ainsi que son identificateur.

✍ 7

2.11.3 Terminaison d'un processus

```
#include <stdlib.h>
void exit(int compte_rendu)
```

La fonction `exit` termine l'exécution du processus courant et transmet à son père un compte rendu dans l'octet de poids faible de `compte_rendu` : 0 (ou `EXIT_SUCCESS`) pour une terminaison normale, ou un entier dans l'intervalle $[1, 255]$ (par exemple `EXIT_FAILURE`) en cas de problème.

Un processus qui a terminé son traitement s'autodétruit. Ceci conduit :

- à fermer tous les fichiers ouverts du processus
- à réveiller son père (si le père est en attente de la terminaison d'un fils)
- à notifier au père sa terminaison par la transmission d'un compte rendu.

Un processus terminé se retrouve dans l'état de zombie tant que son père n'est pas terminé et qu'il n'a pas pris connaissance de la terminaison de son fils, c'est-à-dire tant qu'il n'a pas attendu la terminaison de son fils (voir la fonction `wait` ci-dessous). Quand un processus crée des processus fils, il est important qu'il attende par la suite leurs terminaisons. Dans le cas contraire, les fils restent sous la forme de zombies durant toute la durée de l'exécution de leur père.

2.11.4 Attente de la terminaison d'un fils

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
pid_t wait(int *pcirconstance)
```

La fonction `wait` endort le processus appelant jusqu'à la terminaison de l'un de ses fils (ou l'arrivée d'un signal non ignoré). Elle retourne :

- le `pid` du fils qui s'est terminé et, si elle est différente de `NULL`, place à l'adresse `pcirconstance` un entier qui décrit la circonstance de cette terminaison. Si cet entier s'appelle `circonstance`, la macro `WEXITSTATUS(circonstance)` permet alors de récupérer le code de retour transmis par le fils (grâce à la fonction `exit` ou à l'instruction `return` dans la fonction `main`).
- la valeur -1 si les terminaisons de tous les fils ont déjà été prises en compte ou s'il n'y a pas de fils.

2.11.5 Endormissement temporaire d'un processus

```
#include <unistd.h>
unsigned int sleep(unsigned int sec)
```

La fonction `sleep` endort le processus appelant durant `sec` secondes. Si le processus est réveillé par un signal avant l'écoulement des `sec` secondes, `sleep` retourne le nombre de secondes qu'il restait à attendre. Sinon, elle retourne 0.

✎ **Exercice 4**

écrire un programme qui permet à quatre processus frères, numérotés 1, 2, 3 et 4, de constituer avec le père, numéroté 0, un anneau unidirectionnel. Entre ces processus, on définit la relation de successeur de la manière suivante :

- successeur(i)=i+1 pour i=0,1,2,3.
- successeur(4)=0

Chaque processus fils doit afficher `N` fois (`N` étant une constante symbolique définie dans le programme) un message accompagné de son identificateur de processus et il retourne un compte rendu égal à son numéro (pas son identificateur). Le père attend, avant de se terminer, la fin de tous ses fils et, à chaque terminaison d'un fils, il affiche l'identificateur du fils terminé ainsi que le code de retour de ce fils.

2.11.6 Primitives de recouvrement

La famille de primitives `exec` permet à un processus de charger en mémoire un nouveau code exécutable. Après un `exec`, le contrôle de l'exécution est donné au programme lancé. Le code et les données étant écrasées, il est impossible de revenir au programme dans lequel se trouve l'appel à la fonction de la famille de primitives `exec`. Le processus garde son identificateur et donc la plupart de ses caractéristiques : il n'y a pas de création de processus.

Les fonctions de la famille diffèrent principalement par la manière dont on passe en paramètres les informations concernant le nouveau code qui va être exécuté. Par exemple, la fonction :

```
#include <unistd.h>
int execl(const char *désignation_fichier, const char *arg, ... , NULL)
```

permet de charger en mémoire le programme défini par :

- `désignation_fichier` qui est la désignation du fichier qui contient le code à exécuter ;
- les paramètres suivants qui sont des chaînes de caractères correspondant aux paramètres de la fonction `main`, c'est-à-dire `argv[0]`, `argv[1]`, `argv[2]`..
- le dernier paramètre qui doit être `NULL` (ou `(char *)0`).

Exemple : pour remplacer le code du processus courant par celui de la commande `ls -l`, il suffit d'utiliser :

```
execl("/bin/ls","ls","-l",NULL);
```

Une variante intéressante est la fonction :

```
#include <unistd.h>
int execlp(const char *nom_fichier, const char *arg, ... , NULL)
```

qui se comporte comme la fonction `execl` à l'exception du premier paramètre `nom_fichier` qui est le nom du fichier exécutable que l'on veut lancer et qui sera recherché dans les répertoires contenus dans la variable d'environnement `PATH`.

Exemple : l'appel précédent peut donc être remplacé par :

```
execlp("ls","ls","-l",NULL);
```

Si tout se passe bien, les fonctions de la famille `exec` ne retournent jamais puisque c'est le nouveau code qui remplace l'ancien. Si une erreur se produit (exécutable inexistant par exemple), alors elles retournent -1.

Attention : la bibliothèque standard propose la fonction `system` qui a été décrite au paragraphe 1.3.2.2, page 9, et qui permet de lancer l'exécution de la commande. Mais la fonction `system` crée un processus fils car elle lance un nouveau shell pour interpréter la commande.

✎ Exercice 5

Que produit l'exécution du fragment de programme suivant :

```
printf("Maintenant :\n");
execlp("date","date",NULL);
printf("Terminé\n");
```

✎ Exercice 6

Soit le programme suivant :

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
  printf("%d\n",(int)getpid());
  return 0;
}
```

Si `pid` est le nom du fichier exécutable produit par la compilation de ce programme, que produit l'exécution du programme suivant :

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
  printf("%d\n",(int)getpid());
  printf("%d\n",execlp("pid","pid",NULL));
  return 0;
}
```

Voici un exemple d'utilisation des primitives `fork`, `wait` et `exec` dans lequel les contrôles d'erreur ainsi que les inclusions des fichiers d'en-tête ont été omis : ✍ 8

```
int main(void)
{
  int pid=fork();
  if (pid==0)     /* on est dans le fils */
    execl("/bin/ls","ls","-l",NULL);
  else
    if (pid>0)    /* on est dans le père */
      wait(NULL); /* le père attend la fin du fils */
  exit(0);        /* terminaison du père (le fils n'exécute pas cette instruction) */
}
```

✎ Exercice 7

écrire un programme C qui réalise un travail similaire à celui de l'interpréteur de commandes lorsqu'on lui demande d'exécuter la séquence de commandes suivante :

```
ls -l
pwd
```

L'interpréteur doit créer un processus fils, lui faire exécuter la commande `ls -l` et attendre la fin de ce processus fils. Puis, il crée un nouveau processus fils, lui fait exécuter la commande `pwd` et attend sa fin.

Chapitre 3
Primitives Unix (POSIX.1) de manipulation des fichiers

3.1 Introduction

On appelle « primitives » ou « appels système » des services offerts par le noyau et qui sont activés par des appels de fonctions. On ne s'intéresse ici qu'au sous-ensemble que l'on appelle parfois « entrées-sorties de bas niveau » qui permettent d'interagir avec le système de gestion des fichiers. On ne verra que quelques fonctions et quelques paramètres (voir L3, M1, etc.).

3.2 Représentation d'un ensemble par un entier

En langage C, on peut représenter un ensemble par un entier dont les bits positionnés à 1 dans sa représentation binaire indiquent les éléments présents. Chaque élément est représenté par une puissance de 2.

Exemple :

- *Définition du type*
```
typedef unsigned char Soupe; /* Ensemble pouvant comporter jusqu'à 8 éléments */
```
- *Définition des éléments*
```
#define PATATES   1    /* 00000001 */
#define CAROTTES  2    /* 00000010 */
#define NAVETS    4    /* 00000100 */
#define OIGNONS   8    /* 00001000 */
#define FEVES    16    /* 00010000 */
#define POIREAUX 32    /* 00100000 */
#define CELERI   64    /* 01000000 */
#define COURGE  128    /* 10000000 */
```
- *Déclaration d'un ensemble*
```
Soupe s;
```
- *Construction d'un ensemble :* opérateur `|` (OU logique bit à bit)
```
s=CAROTTES|NAVETS|OIGNONS|POIREAUX;    /* 00101110 */
```
- *Détermination de l'appartenance d'un élément à un ensemble :* opérateur `&` (ET logique bit à bit)
```
if (s&OIGNONS)
  printf("Il y a des oignons dans la soupe.\n");
```

Cette technique est par exemple utilisée au niveau des paramètres de la fonction `open` qui fait partie de la bibliothèque Unix de manipulation des fichiers.

3.3 Primitives d'entrées-sorties

3.3.1 Ouverture d'un fichier

```
#include <sys/types.h> /* types implémentés différemment selon la version d'Unix */
#include <sys/stat.h> /* constantes symboliques pour les droits d'accès */
#include <fcntl.h> /* constantes symboliques pour les différents types d'ouverture */
int open(const char *chemin, int mode)
int open(const char *chemin, int mode, mode_t droits)
int creat(const char *chemin, mode_t droits)
```

- `chemin` : chaîne de caractères contenant la désignation du fichier.
- `mode` : entier obtenu par la combinaison de constantes symboliques grâce à l'opérateur « ou bit à bit » (`|`) permettant de décrire le mode d'ouverture du fichier. Il doit comporter l'une des constantes suivantes :
  - `O_RDONLY` : ouverture en lecture
  - `O_WRONLY` : ouverture en écriture
  - `O_RDWR` : ouverture en lecture/écriture

  auxquelles on rajoute n'importe quelle combinaison des constantes suivantes :
  - `O_APPEND` : concaténation des données en fin de fichier
  - `O_CREAT` : création du fichier lorsqu'il n'existe pas
  - `O_EXCL` : provoque une erreur si le fichier existe (`open` retourne `-1`)
  - `O_TRUNC` : efface le contenu du fichier s'il existe
  - `O_SYNC` : écriture immédiate sur disque en vidant les tampons
  - ...
- `droits` : si `O_CREAT` est utilisé, il faut fournir un troisième paramètre qui indique les droits du fichier créé. Il est construit grâce à l'opérateur `|` et aux constantes suivantes :
  - `S_IRUSR` : lecture autorisée pour le propriétaire
  - `S_IRGRP` : lecture autorisée pour le groupe
  - `S_IROTH` : lecture autorisée pour les autres
  - `S_IWUSR` : écriture autorisée pour le propriétaire
  - `S_IWGRP` : écriture autorisée pour le groupe
  - `S_IWOTH` : écriture autorisée pour les autres
  - `S_IXUSR` : exécution autorisée pour le propriétaire
  - `S_IXGRP` : exécution autorisée pour le groupe
  - `S_IXOTH` : exécution autorisée pour les autres
  - `S_IRWXU` : lecture, écriture et exécution autorisées pour le propriétaire
  - `S_IRWXG` : lecture, écriture et exécution autorisées pour le groupe
  - `S_IRWXO` : lecture, écriture et exécution autorisées pour les autres

  Remarque : `S_IRWXU == S_IRUSR|S_IWUSR|S_IXUSR`

  La fonction `open` retourne le descripteur interne du fichier, c'est-à-dire l'indice (entier naturel) dans la table des descripteurs associée au processus ou -1 en cas de problème.

  La fonction `creat` est équivalente à la fonction `open` en mode `O_CREAT|O_WRONLY|O_TRUNC`

Remarque : les descripteurs de fichiers correspondant aux unités standard sont les constantes `STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO` définies dans `unistd.h` et qui valent respectivement 0, 1 et 2. Ces trois unités standard sont automatiquement ouvertes au lancement du programme.

3.3.2 Fermeture d'un fichier

```
#include <unistd.h>
int close(int fd)
```
Cette fonction ferme le fichier de descripteur `fd` (en fait, elle libère le descripteur `fd`). Elle retourne 0 si la fermeture s'est bien passée ou -1 en cas d'erreur (descripteur incorrect par exemple).

3.3.3 Lecture dans un fichier

```
#include <unistd.h>
ssize_t read(int fd, void *p, size_t nboctets)
```
Cette fonction essaie de lire `nboctets` octets dans le fichier de descripteur `fd` et les range à partir de l'adresse `p`. Elle retourne le nombre d'octets effectivement lus.
Remarque : `size_t` est un type entier retourné par l'opérateur `sizeof` et `ssize_t` est un type entier retourné par la fonction `read`.

3.3.4 écriture dans un fichier

```
#include <unistd.h>
ssize_t write(int fd, const void *p, size_t nboctets)
```
Cette fonction essaie d'écrire `nboctets` octets situés à partir de l'adresse `p` dans le fichier de descripteur `fd`. Elle retourne le nombre d'octets effectivement écrits.

✎ Exercice 8
écrire un programme C qui effectue la copie du contenu d'un fichier par blocs de 512 octets. Ce programme prend deux arguments : le nom du fichier source et le nom du fichier destination. Si le fichier destination n'existe pas, il doit être créé avec le droit en lecture pour le propriétaire et le groupe et le droit en écriture pour le propriétaire. S'il existe, son contenu doit être « écrasé ». Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, fichier inaccessible, etc.).

3.3.5 Contrôle de la position courante

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t decalage, int origine)
```
Cette fonction modifie la position courante d'un fichier de descripteur `fd`. La nouvelle position est située à `decalage` octets de l'origine qui peut être : `SEEK_SET` (début), `SEEK_CUR` (position courante) ou `SEEK_END` (fin). Elle retourne la nouvelle position courante exprimée en nombre d'octets depuis le début du fichier ou -1 en cas d'erreur.

✎ Exercice 9
Retour à une position mémorisée
On suppose que le descripteur `fd` est associé à un fichier ouvert en lecture.

1. écrire l'instruction permettant de mémoriser dans une variable `position` la position courante dans le fichier.
2. écrire l'instruction permettant de revenir à une position mémorisée dans la variable `position`.

✎ **Exercice 10**

Détermination de la taille d'un fichier

écrire l'instruction permettant de récupérer dans une variable `taille` la taille (nombre d'octets) d'un fichier ouvert en lecture et de descripteur `fd`.

✎ **Exercice 11**

Accès direct au contenu d'un fichier binaire

On suppose que le descripteur `fd` est associé à un fichier binaire ouvert en lecture et contenant des `int`. écrire une fonction qui retourne le `n`$^{\text{ième}}$ `int` de ce fichier.

3.4 Duplication de descripteur

Chaque processus possède une table de descripteurs de fichiers. Il est possible de dupliquer un descripteur de fichier. Les deux descripteurs désigneront alors le même fichier. Ils pourront être utilisés de manière interchangeable.

```
#include <unistd.h>
int dup(int fd)
int dup2(int fdsource, int fddestination)
```

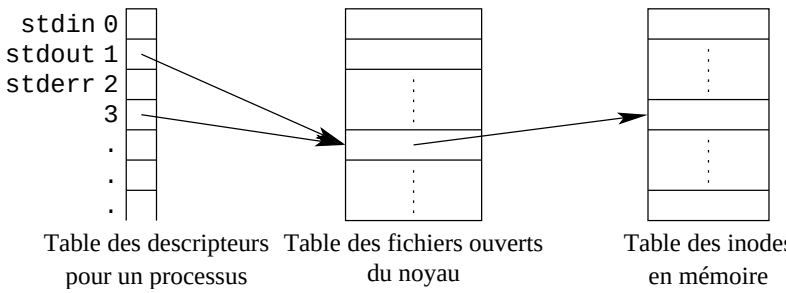
La fonction `dup` utilise le premier descripteur libre pour le nouveau descripteur. La fonction `dup2` duplique le descripteur `fdsource` dans le descripteur `fddestination` après l'avoir fermé si nécessaire. Ces fonctions retournent le nouveau descripteur ou -1 en cas de problème (dans ce cas, la fonction `perror` peut être utilisée).

Exemple : redirection de la sortie standard vers un fichier.

```
int d;
d=open("fich", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
if (d==-1)
  perror("échec car ");
else
{
  printf("Bonjour\n");
  dup2(d,1); /* ou bien dup2(d, STDOUT_FILENO); ou bien close(1); dup(d); */
  printf("Bonsoir\n");
}
```

Lors de l'exécution de cet extrait de programme, `Bonjour` est affiché à l'écran et `Bonsoir` est écrit dans le fichier `fich`.

Remarque : pour faire une redirection avec écriture en fin de fichier (équivalente à l'utilisation de `>>` en shell), il suffit de remplacer `O_TRUNC` par `O_APPEND`.



✎ **Exercice 12**

écrire un programme C qui lance l'exécution d'une commande en réalisant les redirections du type : `commande < fich_entree >> fich_sortie`

Ce programme prend trois arguments : la commande, le nom du fichier vers lequel doit être redirigée l'entrée standard et le nom du fichier vers lequel doit être redirigée la sortie standard (avec ajout en fin de fichier). L'exécution de la commande doit être déléguée à un processus fils. Les redirections doivent être réalisées par duplication des descripteurs de fichiers. Si le fichier de sortie n'existe pas, il doit être créé avec le droit en lecture pour tous et le droit en écriture pour le propriétaire. Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, fichier inaccessible, etc.).

3.5 Suppression d'un fichier (lien physique)

```
#include <unistd.h>
int unlink(char *designation)
```

Cette fonction retourne 0 en cas de succès ou -1 en cas d'échec.
La fonction `remove` de la bibliothèque standard fait appel à `unlink`.

Chapitre 4
Structure interne du système de fichiers d'Unix

4.1 Introduction

Un fichier Unix est une suite finie d'octets. Il est matérialisé par des blocs du disque. à chaque fichier est associée une petite table appelée *inode* (ou nœud d'index).

Un inode contient des informations concernant le fichier ainsi que 10 adresses (numéros) de blocs de données et 3 adresses de blocs d'adresses (pointeurs à simple, double et triple indirection).

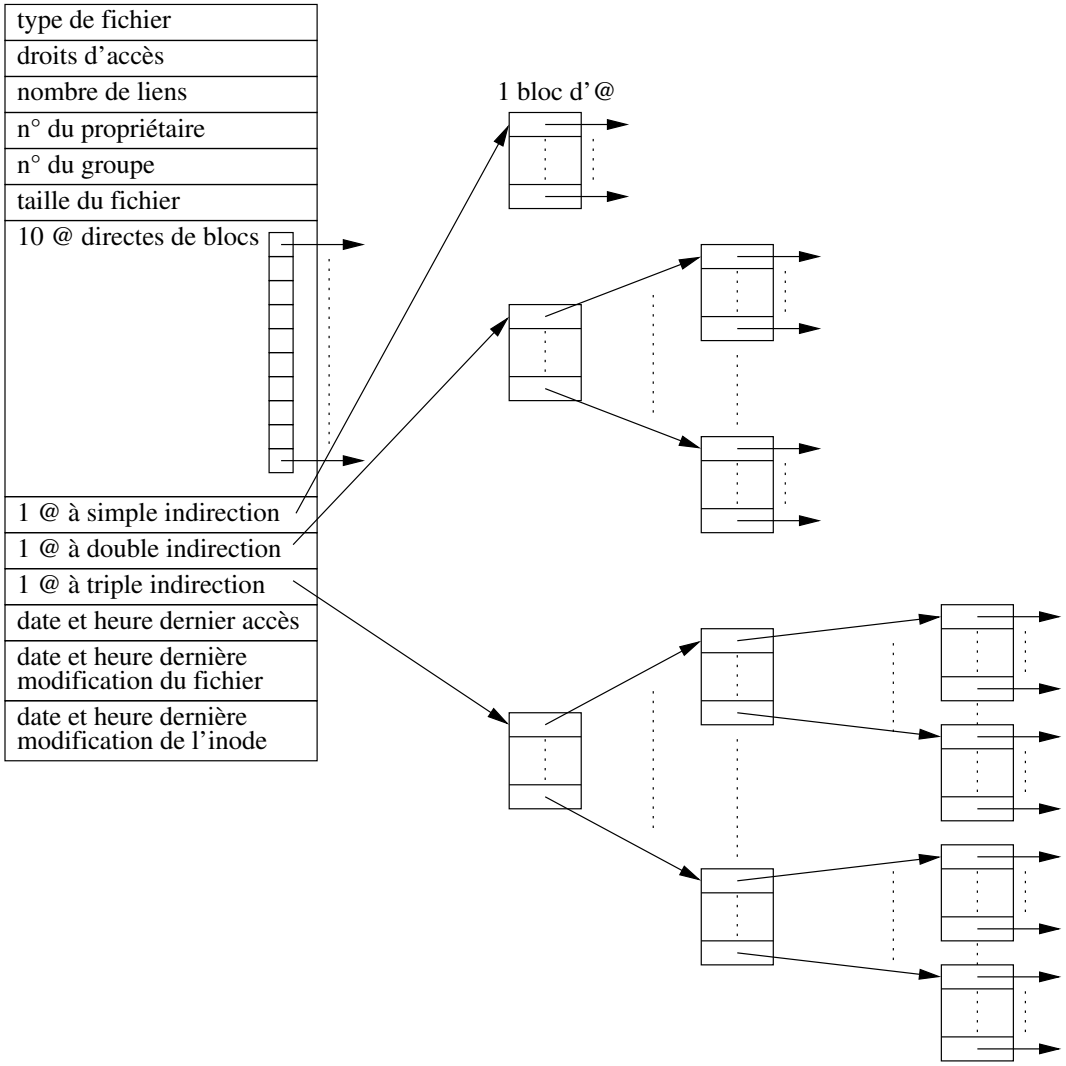
4.2 Structure de base du système de fichiers

n° de bloc	Contenu
0	Bloc de boot (+ identification du disque)
1	Superbloc : • date de dernière mise à jour du SF • nombre de fichiers que peut contenir le SF • taille du SF • pointeur sur la liste des blocs libres
2 ⋮ k	Table des inodes (informations concernant les fichiers)
$k+1$ ⋮	Blocs de données

La taille d'un bloc dépend du système (exemples : 512 octets, 8192 octets, ...). Le nombre d'inodes (le nombre maximum de fichiers) est déterminé à la création du SF. Sous Linux, la commande `dumpe2fs -h /dev/sda1` affiche les caractéristiques du SF installé sur le périphérique `sda1`.

4.3 Structure d'un inode

Chaque inode est repéré dans la table par son numéro (index). Le numéro 2 est la racine du SF. La commande `ls -li` affiche les numéros d'inodes.



✎ Exercice 13

Compléter les zones laissées en pointillés dans le texte suivant.

- Si taille(fichier) $\leq$ .. blocs, les adresses des blocs de données sont entièrement contenues dans l'inode.
- Si taille(fichier) $>$ .. blocs, le système attribue au fichier un nouveau bloc du disque contenant les adresses (numéros) des blocs de données.
  Supposons qu'un bloc peut contenir 256 adresses de blocs.
- Alors, si taille(fichier) $\leq$ ..... blocs, on n'a pas besoin d'utiliser les pointeurs à double et triple indirection.
- De même, si taille(fichier) $\leq$ ..... blocs, on n'a pas besoin du pointeur à triple indirection.
- La taille maximale d'un fichier est donc, dans ce cas, de ..... blocs.

4.4 Accès aux caractéristiques d'un fichier

Il est possible d'accéder aux informations contenues par l'inode d'un fichier grâce aux fonctions suivantes :

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *NomFich, struct stat *pInfos)
int fstat(int DescFich, struct stat *pInfos)
```

Ces fonctions récupèrent des informations contenues dans l'inode d'un fichier et les stockent dans la structure dont l'adresse est passée en paramètre. La fonction `stat` prend en entrée le nom du fichier alors que la fonction `fstat` utilise le descripteur du fichier, ce qui sous-entend que le fichier doit avoir été ouvert. Ces fonctions retournent -1 en cas de problème ou 0 sinon. La structure `struct stat` contient (au moins) les champs suivants :

```
struct stat
{
  dev_t   st_dev;   /* Id du système de fichiers contenant le fichier          */
  ino_t   st_ino;   /* N° de l'inode du fichier                                */
  mode_t  st_mode;  /* Type et mode du fichier                                 */
  nlink_t st_nlink; /* Nb de liens physiques sur le fichier                    */
  uid_t   st_uid;   /* Id de l'utilisateur propriétaire du fichier             */
  gid_t   st_gid;   /* Id du groupe du propriétaire du fichier                 */
  off_t   st_size;  /* Taille du fichier en octets                             */
  time_t  st_atime; /* Instant du dernier accès au fichier                     */
                    /* (création, lecture, écriture,                           */
                    /*  lecture des caractéristiques,...)                      */
  time_t  st_mtime; /* Instant de la dernière modification des données         */
  time_t  st_ctime; /* Instant de la dernière modification des caractéristiques */
                    /* (création, écriture, changement des droits,             */
                    /*  changement de propriétaire, ...)                       */
};
```

- Le champ `st_mode` contient plusieurs informations dont le type du fichier et ses droits d'accès.
  - Pour obtenir le type du fichier, on fait appel aux macros suivantes (on suppose que `Infos` est une variable de type `struct stat`) :
    `S_ISREG(Infos.st_mode)` $\neq 0$ si c'est un fichier ordinaire,
    `S_ISFIFO(Infos.st_mode)` $\neq 0$ si c'est un fichier spécial tube,
    `S_ISCHR(Infos.st_mode)` $\neq 0$ si c'est un fichier spécial caractère,
    `S_ISBLK(Infos.st_mode)` $\neq 0$ si c'est un fichier spécial bloc,
    `S_ISDIR(Infos.st_mode)` $\neq 0$ si c'est un répertoire.
  - Pour obtenir les droits d'accès, on utilise l'opérateur `&` et les constantes correspondantes.

    Exemple : `(Infos.st_mode & S_IRUSR)` $\neq 0$ si le propriétaire a le droit en lecture sur le fichier.
- Les champs `st_atime`, `st_mtime` et `st_ctime` contiennent des instants (dates et heures) sous la forme d'entiers (nombre de secondes écoulées depuis le 01/01/1970, c'est-à-dire le « big-bang » Unix). Pour les afficher sous une forme plus lisible, on peut utiliser la fonction standard `ctime` :
  ```
  #include <time.h>
  char *ctime(const time_t *pTemps)
  ```
  Exemple : `printf("%s",ctime(&Infos.st_atime));`

4.5 Contenu des répertoires

Un répertoire est un fichier qui contient le nom des éléments contenus dans ce répertoire ainsi que leurs inodes. Par exemple, le contenu d'un répertoire contenant les fichiers `fich1.txt` et `fich2.txt` et le répertoire `REP` peut être représenté par la table suivante :

Nom	N° inode
.	8186182
..	1784306
REP	8333283
fich1.txt	8186211
fich2.txt	8186327

4.6 Parcours d'un répertoire

Le parcours d'un répertoire peut se faire en utilisant les directives :

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

et en utilisant les fonctions suivantes :

- `DIR *opendir(const char *DesignationRep)`
  ouvre le répertoire de nom `DesignationRep` et retourne son identificateur ou `NULL` en cas de problème.
- `struct dirent *readdir(DIR *Rep)`
  retourne un pointeur vers une structure qui représente l'élément courant ou `NULL` en cas de problème ou si la fin du répertoire est atteinte, et passe à l'élément suivant. La structure `struct dirent` contient au moins le champ `d_name`, de type tableau de caractères, qui est le nom de l'élément (fichier ou répertoire) sous la forme d'une chaîne de caractères.

  Attention : la fonction `readdir` peut ou non, selon les implémentations (c'est le cas sur `azteca`), retourner les éléments de nom `"."` et `".."`
- `void rewinddir(DIR *IdRep)`
  remet la position courante au début du répertoire d'identificateur `IdRep`.
- `int closedir(DIR *IdRep)`
  ferme le répertoire d'identificateur `IdRep` et retourne -1 en cas de problème ou 0 sinon.

✎ **Exercice 14**

écrire une fonction d'en-tête :

```
long TailleRepCour(void)
```

qui retourne le nombre total d'octets occupés par les fichiers présents dans le répertoire courant (les éventuels sous-répertoires ne doivent pas être examinés).

4.7 Parcours d'une sous-arborescence

La fonction de parcours d'une sous-arborescence de racine `Rep` doit être récursive et suivre l'algorithme général suivant :

```
Parcourir(Rep)
{
  Pour tout Element de Rep
```

```
    Si Element est différent de "." et de ".."
      Si Rep/Element est un répertoire
        Parcourir(Rep/Element);
      Sinon
        Traiter(Rep/Element);
}
```

L'algorithme précédent nécessite de construire des désignations, c'est-à-dire de manipuler des chaînes de caractères. Pour cela, on utilise les fonctions de manipulation des chaînes de caractères de la bibliothèque standard.

On peut utiliser la fonction `strcpy` qui permet de copier une chaîne de caractères et la fonction `strcat` qui permet de concaténer deux chaînes de caractères.
Exemple :

```
#include <string.h>
...
char ch1[512],ch2[512];
strcpy(ch1,"Il fait "); /* ch1 contient "Il fait " */
strcpy(ch2,ch1); /* ch1 contient "Il fait " et ch2 contient "Il fait " */
strcpy(ch1,"beau "); /* ch1 contient "beau" et ch2 contient "Il fait " */
strcat(ch2,ch1); /* ch2 contient "Il fait beau" et ch1 contient "beau" */
```

Pour comparer deux chaînes de caractères, on peut utiliser la fonction `strcmp` qui retourne 0 quand les deux chaînes passées en paramètres sont identiques ou un entier non nul quand elles sont différentes.

✎ Exercice 15

écrire une fonction qui affiche à l'écran les désignations de tous les fichiers contenus dans la sous-arborescence dont le répertoire racine est passé en paramètre.

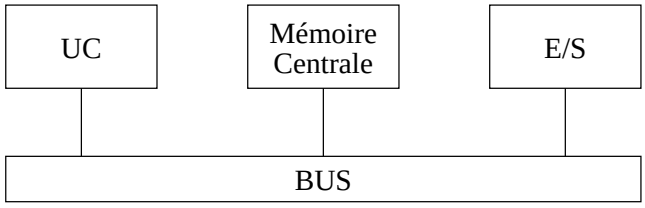
Supposons que l'on se trouve dans la configuration suivante : le répertoire `TEST` contient le fichier `fich.txt`, le répertoire `REP1` et le répertoire `REP2` ; le répertoire `REP1` contient les fichiers `fich11.txt` et `fich12.txt` ; le répertoire `REP2` contient le fichier `fich2.txt`. Si le répertoire `TEST` est passé en paramètre à la fonction, elle doit alors afficher :

```
TEST/REP1/fich11.txt
TEST/REP1/fich12.txt
TEST/REP2/fich2.txt
TEST/fich.txt
```

Chapitre 5
Gestion de la mémoire : mémoire virtuelle et allocation non contiguë

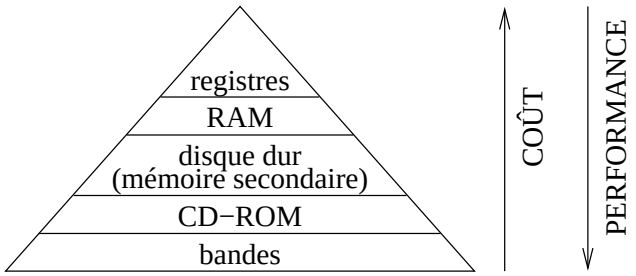
5.1 Introduction

Pour qu'un programme (processus) soit exécuté, il faut qu'il soit en mémoire centrale.



Tous les processus ne rentrent pas en mémoire centrale car elle a une taille limitée.

5.1.1 Hiérarchie des mémoires



5.1.2 Mémoire virtuelle

C'est le support de l'ensemble des informations potentiellement accessibles. La mémoire virtuelle est utilisée pour les raisons suivantes :

- rapport coût/performance ;
- les processus trop gros ne peuvent par rentrer en mémoire centrale $\Longrightarrow$ chargement partiel + chargement à la demande.

La mémoire virtuelle permet d'utiliser de la mémoire secondaire comme si c'était de la RAM (mémoire centrale).

5.1.3 Objectifs

- fournir un espace d'adressage indépendant de la mémoire physique ;
- pouvoir exécuter des programmes dont la taille excède la taille de la mémoire physique.

5.1.4 Réalisation de la mémoire virtuelle

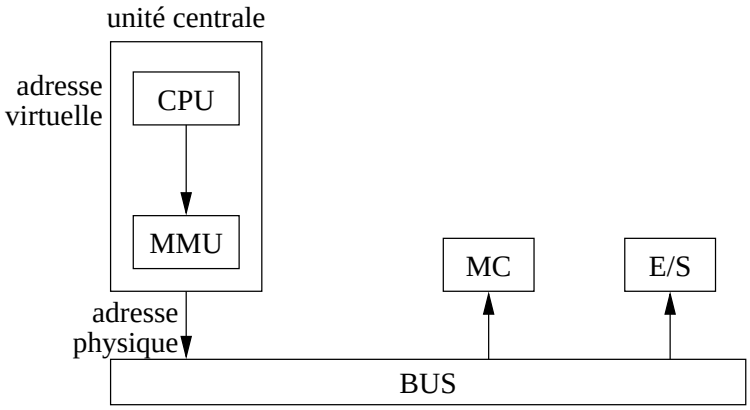
- représentation physique : mémoire centrale + mémoire secondaire ;
- gestion basée sur la pagination, l'allocation par blocs, la segmentation.

5.2 Pagination

Le mécanisme de *pagination* permet d'allouer aux processus un espace de *mémoire physique* (la mémoire centrale ou RAM) constitué de parties *non contiguës*. La mémoire (l'espace d'adressage) physique est découpée en blocs de taille fixe appelés *pages physiques*, *cadres*, *cases* ou *frames*. La mémoire qui est directement adressée par les processus, appelée *mémoire virtuelle*, est elle-même découpée en blocs de taille fixe appelés *pages virtuelles* ou, plus simplement, *pages*. Pour simplifier, la taille d'une page virtuelle est la même que celle d'une page physique (sous Linux, la commande `getconf PAGESIZE` affiche cette taille).

5.2.1 Transformation des adresses

Une page virtuelle peut correspondre à une page physique quelconque. Le mécanisme de pagination assure à travers le matériel (MMU : *memory management unit*) la correspondance entre une adresse de la mémoire virtuelle et une adresse de la mémoire physique. Ce mécanisme utilise la *table des pages* qui est associée à chaque processus et qui est, elle-même, stockée en mémoire physique.



$$\text{adresse virtuelle} = \underbrace{\text{adresse de page}}_{\downarrow} + \underbrace{\text{déplacement dans la page}}_{=}$$

$$\text{adresse physique} = \overbrace{\text{adresse de cadre}} + \overbrace{\text{déplacement dans le cadre}}$$

5.2.2 Table des pages virtuelles

	Présente	Modifiée	Protection	n° page physique
0	oui			0
1	oui			3
2	non			
3	oui			2
4	non			
5	non			
6	oui			1

5.2.3 Défaut de page

La mémoire virtuelle peut être de taille très importante et beaucoup plus grande que la taille de la mémoire physique. à un instant donné de l'exécution d'un processus, toutes ses pages ne sont pas obligatoirement en mémoire physique ; certaines peuvent être conservées sur la mémoire secondaire (le disque). Quand un processus accède à une page qui n'est pas en mémoire physique, on dit qu'il y a un *défaut de page*. L'instruction est interrompue et le système doit alors allouer un cadre à cette page. Si aucun cadre n'est libre, il réquisitionne un cadre occupé par une autre page : c'est le *remplacement de page*. Différents algorithmes peuvent être utilisés pour déterminer quelle page va être remplacée. Si la page remplacée a été modifiée par rapport à sa version sur la mémoire secondaire, elle doit être réécrite en mémoire secondaire. Enfin, l'instruction qui a provoqué le défaut de page peut être exécutée. Ce mécanisme s'appelle la *pagination à la demande*.

✎ Exercice 16

Une mémoire virtuelle a une taille de page de 1024 mots, huit pages virtuelles et quatre pages physiques. Sa table des pages est la suivante :

Page virtuelle	*Page physique*
0	3
1	1
2	pas en mémoire physique
3	pas en mémoire physique
4	2
5	pas en mémoire physique
6	0
7	pas en mémoire physique

1. Donnez la liste des adresses qui provoqueront un défaut de page.
2. Quelles sont les adresses physiques correspondant aux adresses virtuelles 0, 3728, 1023, 1024, 1025, 7800, 4096 ?

✎ Exercice 17

Dans un système paginé, le surcoût dû à un défaut de page est de 20 microsecondes et le temps d'accès à un mot de la mémoire centrale de 500 nanosecondes. Calculer le temps d'accès moyen d'un tel système, en fonction du taux de défaut de page, sachant que la table des pages est entièrement en mémoire centrale. Quel est le taux maximum de défaut de page qui maintient le temps d'accès moyen en dessous de 1,2 microsecondes ?

5.2.4 Algorithmes de remplacement

Quand une page n'est pas en mémoire centrale, le système doit :

- la chercher en mémoire secondaire et la charger en mémoire centrale ;
- s'il n'y a plus de place en mémoire centrale :
  - choisir un cadre en mémoire centrale ;
  - éventuellement, le sauvegarder en mémoire secondaire ;
  - le remplacer par la page recherchée.

Il existe plusieurs stratégies pour déterminer quelle est le cadre qui va être remplacé. Par exemple :

OPT – L'algorithme de remplacement OPT, ou algorithme optimal, consiste à remplacer la page à laquelle on accédera le plus tard (dont la prochaine référence est la plus tardive). Cet algorithme n'est pas applicable en pratique car on ne peut pas prévoir les accès futurs, mais il permet de comparer les performances des autres algorithmes à ses performances qui sont les meilleures.

RANDOM – L'algorithme de remplacement RANDOM consiste à choisir au hasard la page qui va être remplacée.

FIFO – L'algorithme de remplacement FIFO (*first in first out*), ou algorithme chronologique de chargement, consiste à remplacer la page qui est en mémoire depuis le plus longtemps. Le matériel doit donc maintenir une file des numéros de pages dans l'ordre où elles ont été chargées en mémoire.

LRU – L'algorithme de remplacement LRU (*least recently used*), ou algorithme chronologique d'utilisation, consiste à remplacer la page qui n'a pas été utilisée depuis le plus longtemps (dont la dernière référence est la plus ancienne). Le matériel doit donc maintenir la liste des pages ordonnées par leur dernier moment d'utilisation.

évidemment, les remplacements entraînent un surcoût.

✎ Exercice 18

Une mémoire centrale comprend 25 cadres. Cinq processus s'exécutent simultanément. Chaque processus dispose de 5 cadres et produit la liste de références (en numéros de pages) : $0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, etc$ L'algorithme de remplacement utilisé est LRU et la mémoire est intialement vide.

1. Quel est le taux de défaut de page provoqué par cette exécution ?
2. On introduit un $6^{\text{ème}}$ processus, analogue aux 5 autres quant à la suite de références produites. On attribue maintenant à chaque processus 4 cadres. Calculer le nouveau taux de défaut de page.
3. Décrire sur cet exemple le phénomène de l'écroulement.

✎ Exercice 19

Un programme a un espace de mémoire virtuelle de 512 mots. On considère la suite d'adresses virtuelles : 34, 123, 145, 510, 456, 345, 412, 10, 14, 12, 234, 236, 412.

1. Donner la suite des numéros de pages virtuelles référencées, sachant qu'elles comportent 100 mots.
2. Le programme dispose de 300 mots en mémoire centrale. Calculer le taux de défaut de page pour les algorithmes FIFO, LRU et OPT.

✎ Exercice 20

On a constaté que le nombre d'instructions exécutées entre deux défauts de page est directement proportionnel au nombre de cadres alloués à un programme. Si on double la mémoire disponible, on double le temps entre deux défauts de page consécutifs. Supposez qu'une instruction normale requiert 1 microseconde et qu'un défaut de page en requiert 2001. Si le programme s'exécute pendant 60 secondes et s'il fait 15000 défauts de page, quelle serait sa durée d'exécution si on avait deux fois plus de mémoire ?

✎ Exercice 21

S'il faut une microseconde pour exécuter une instruction et si le traitement d'un défaut de page entraîne un surcoût de n microsecondes, donner la formule du temps d'exécution moyen d'une instruction s'il se produit en moyenne un défaut de page après k instructions.

✎ Exercice 22

Le taux d'activité moyen du processeur est donné par :

$$taux = t/(t + pT) = 1/(1 + pS)$$

où t est le temps moyen d'exécution d'une instruction, T le temps moyen de traitement d'un défaut de page, p la probabilité globale (sur l'ensemble des processus) d'un défaut de page. Le rapport $S = T/t$ est du même ordre de grandeur que le rapport du temps de transfert d'une page depuis le disque sur le temps d'accès en mémoire principale ; sa valeur est comprise entre 1000 et 10000.

1. Que peut-on dire de p pour que le taux reste voisin de 1 ?
2. Comment évolue le taux en fonction de p ?

Chapitre 6
Fonctions de la bibliothèque standard pour la manipulation des fichiers

6.1 Introduction

Ce chapitre décrit les outils de gestion des fichiers de la bibliothèque standard du langage C ANSI.
Même si, au bout du compte, tout est binaire, on distingue généralement deux types de fichiers :

- les fichiers de texte : chaque octet qui les compose représente un caractère et ces caractères sont organisés en lignes (le contenu du fichier apparaît de manière lisible dans la fenêtre d'un éditeur de texte) ;
- les fichiers binaires : ce sont tous les autres fichiers (leur contenu apparaît sous la forme de caractères illisibles dans la fenêtre d'un éditeur).

La différence entre les deux types de fichiers réside dans la manière dont les informations y sont codées.
Exemple : l'entier 24 est codé par :

- `00110010 00110100` dans un fichier de texte : un octet pour le code ASCII du caractère `2`, qui vaut 50, et un octet pour le code ASCII du caractère `4`, qui vaut 52 ;
- `00000000 00000000 00000000 00011000` dans un fichier binaire si les entiers sont codés sur quatre octets.

Même si ce n'est pas une obligation, on n'utilise généralement pas les mêmes fonctions pour réaliser des entrées-sorties dans les fichiers de texte et les fichiers binaires.

Les fonctions de gestion des fichiers nécessitent la directive `#include <stdio.h>`.

6.2 Ouverture

La manipulation d'un fichier va se faire à travers un identificateur de fichier (aussi appelé *flux* ou *stream*) de type `FILE *`.

La fonction
`FILE *fopen(const char *nom, const char *mode)`
ouvre le fichier de désignation `nom` dans le mode `mode` et retourne son identificateur ou `NULL` s'il y a un problème.

Le `mode` est une chaîne de un, deux ou trois caractères :

- le premier caractère indique si le fichier doit être ouvert en lecture (`r` pour *read*), en écriture (`w` pour *write*) ou en écriture en fin de fichier (`a` pour *append*) :
  - `r` : si le fichier existe, on est positionné au début ; sinon, l'ouverture échoue et `fopen` retourne `NULL` ;
  - `w` : si le fichier existe, son contenu est écrasé ; sinon, le fichier est créé ;
  - `a` : si le fichier existe, on est positionné à la fin ; sinon, le fichier est créé ;

- le deuxième caractère, qui peut être ajouté au premier, est le signe `+` qui précise que l'on souhaite effectuer une mise à jour (lecture et écriture) ;
- le troisième caractère indique si le fichier doit être considéré comme un fichier de texte (`t` pour *text*), qui est le type par défaut, ou comme un fichier binaire (`b` pour *binary*).

Exemple :
```
FILE *id_fich;
id_fich=fopen("data.txt","rt");
if (id_fich==NULL) perror("data.txt ");
```
Remarque : trois identificateurs de fichiers particuliers sont disponibles et n'ont besoin ni d'être ouverts, ni d'être fermés : `stdin` (entrée standard), `stdout` (sortie standard) et `stderr` (sortie standard des erreurs).

6.3 Fermeture

La fonction
`int fclose(FILE *id_fich)`
ferme le fichier d'identificateur `id_fich` et retourne 0, si tout se passe bien, ou `EOF` s'il y a un problème.
Remarques :
- Il ne faut fermer un fichier que s'il est ouvert.
- Il est conseillé de fermer un fichier dès que son traitement est terminé.

6.4 Lecture

6.4.1 Lecture dans un fichier de texte

6.4.1.1 Lecture d'un caractère

La fonction
`int fgetc(FILE *id_fich)`
lit le caractère courant dans le fichier d'identificateur `id_fich` et retourne ce caractère ou `EOF` si la fin du fichier est atteinte.
Remarques : la macro `getc` effectue la même chose que `fgetc` et la macro `getchar` effectue la même chose que `fgetc(stdin)`.

6.4.1.2 Lecture d'une chaîne de caractères

La fonction
`char *fgets(char *ch, int n, FILE *id_fich)`
lit au maximum `n`−1 caractères dans le fichier d'identificateur `id_fich`, s'arrête si elle rencontre le caractère `'\n'`, les range (y compris l'éventuel `'\n'`) dans la chaîne d'adresse `ch` en complétant par le caractère `'\0'`. Elle retourne l'adresse de la chaîne ou `NULL` s'il y a un problème ou si la fin de fichier a été atteinte.

6.4.1.3 Lecture formatée

La fonction
`int fscanf(FILE *id_fich, const char *format[,liste_d_expressions])`
se comporte de la même manière que `scanf` mais effectue la lecture dans le fichier d'identificateur
✍ 9 `id_fich` au lieu du clavier.

6.4.2 Lecture dans un fichier binaire

La fonction
`size_t fread(void *pt, size_t taille, size_t nb_infos, FILE *id_fich)`
lit dans le fichier d'identificateur `id_fich` `nb_infos` informations (blocs d'octets) de `taille` octets chacunes et les range en mémoire à l'adresse `pt` (la zone mémoire doit avoir été préalablement allouée). Elle retourne le nombre d'informations effectivement lues. Ce nombre peut être inférieur à `nb_infos` si la fin du fichier est atteinte. ✍ 10

6.5 écriture

6.5.1 écriture dans un fichier de texte

6.5.1.1 écriture d'un caractère

La fonction
`int fputc(int c, FILE *id_fich)`
écrit dans le fichier d'identificateur `id_fich` le caractère `c` (convertit en `unsigned char`) et retourne le caractère écrit ou `EOF` s'il y a un problème.

6.5.1.2 écriture d'une chaîne de caractères

La fonction
`int fputs(const char *ch, FILE *id_fich)`
écrit dans le fichier d'identificateur `id_fich` la chaîne d'adresse `ch` et retourne `EOF` s'il y a un problème ou une valeur négative dans le cas contraire.

6.5.1.3 écriture formatée

La fonction
`int fprintf(FILE *id_fich, const char *format[,liste_d_expressions])`
se comporte de la même manière que `printf` mais effectue l'écriture dans le fichier d'identificateur `id_fich` au lieu de l'écran.
Remarque : on utilise très souvent `fprintf` pour afficher les messages d'erreur sur `stderr`.
Exemple : ✍ 11

```
FILE *f;
f=fopen("toto.txt","rt");
if (f==NULL) fprintf(stderr,"Fichier toto.txt inexistant.\n");
```

6.5.2 écriture dans un fichier binaire

La fonction
`size_t fwrite(void *pt, size_t taille, size_t nb_infos, FILE *id_fich)`
écrit dans le fichier d'identificateur `id_fich` `nb_infos` informations (blocs d'octets) de `taille` octets chacunes situées en mémoire à l'adresse `pt`. Elle retourne le nombre d'informations effectivement écrites. Ce nombre peut être inférieur à `nb_infos` dans le cas d'un disque saturé. ✍ 12

6.6 Gestion de la position courante

à chaque fichier ouvert est associé un « pointeur de fichier » qui donne la position courante dans le fichier, c'est-à-dire le rang du prochain octet à lire ou à écrire. Après chaque opération de lecture ou

d'écriture, ce « pointeur » est automatiquement incrémenté du nombre d'octets transférés. On parle d'*accès séquentiel* au fichier.

6.6.1 Détection de fin de fichier

La fonction
`int feof(FILE *id_fich)`
retourne une valeur non nulle si la fin du fichier d'identificateur `id_fich` est atteinte, ou 0 sinon.
Attention : la valeur de retour de `feof` n'est valide qu'après avoir effectué au moins une lecture.
Remarque : dans le cas d'une lecture dans un fichier binaire, la valeur de retour de la fonction `fread` peut remplacer avantageusement l'appel à la fonction `feof`.

✎ Exercice 23
Exemple de lecture séquentielle d'un fichier de texte
En n'utilisant que les fonctions de la bibliothèque standard, écrire un programme qui parcourt le fichier dont le nom est passé en paramètre. On suppose qu'il s'agit d'un fichier de texte contenant des entiers séparés par au moins un caractère séparateur (espace, tabulation, nouvelle ligne...). Le programme doit lire de manière séquentielle chaque entier et l'afficher à l'écran.

6.6.2 Positionnement dans un fichier

Il est possible d'accéder à un fichier de manière directe (accès direct) en modifiant sa position courante grâce à la fonction
`int fseek(FILE *id_fich, long decalage, int depart)`
où
- `id_fich` est l'identificateur du fichier ;
- `decalage` est le nombre d'octets dont on veut se décaler par rapport à la position `depart` ;
- `depart` peut être égal à :
  - `SEEK_SET` ou `0` : début de fichier,
  - `SEEK_CUR` ou `1` : position courante dans le fichier,
  - `SEEK_END` ou `2` : fin de fichier.

La fonction `fseek` retourne 0 si tout s'est bien passé ou -1 sinon, en particuler dans le cas où on essaie de revenir en arrière avant le début du fichier. Néanmoins, quand on essaie d'avancer au-delà de la fin du fichier, `fseek` retourne 0 (tout se passe comme si on « piétinait » à la fin du fichier).

La fonction
`void rewind(FILE *id_fich)`
place le pointeur du fichier d'identificateur `id_fich` à son début.
Remarque : `rewind(f);` ⇔ `fseek(f,0,SEEK_SET);`

La fonction
`long ftell(FILE *id_fich)`
retourne la position courante du fichier d'identificateur `id_fich` ou -1 s'il y a un problème.

✎ Exercice 24
Retour à une position mémorisée
On suppose que l'identificateur `f` est associé à un fichier ouvert en lecture. En n'utilisant que les fonctions de la bibliothèque standard :

1. écrire l'instruction permettant de mémoriser dans une variable `position` la position courante dans le fichier.
2. écrire l'instruction permettant de revenir à une position mémorisée dans la variable `position`.

✎ **Exercice 25**

Détermination de la taille d'un fichier

En n'utilisant que les fonctions de la bibliothèque standard, écrire l'instruction permettant de récupérer dans une variable `taille` la taille (nombre d'octets) d'un fichier ouvert en lecture et d'identificateur `f`.

✎ **Exercice 26**

Accès direct au contenu d'un fichier binaire

On suppose que l'identificateur `f` est associé à un fichier binaire ouvert en lecture et contenant des `int`. En n'utilisant que les fonctions de la bibliothèque standard, écrire une fonction qui retourne le $n^{\text{ième}}$ `int` de ce fichier.

6.7 Réouverture d'un identificateur de fichier

La fonction

`FILE *freopen(const char *nom, const char *mode, FILE *id_fich)`

ferme le fichier associé à l'identificateur `id_fich` et ouvre, en l'associant à l'identificateur `id_fich`, le fichier nommé `nom` en mode `mode`. Cette fonction retourne l'identificateur du fichier ou `NULL` s'il y a un problème.

Exemple : redirection de la sortie standard vers un fichier.

```
/* Affichage d'un message sur stdout */
void Bonjour(void)
{
  printf("Bonjour !\n");
}
...
Bonjour(); /* Affiche Bonjour ! à l'écran */
freopen("fich.txt","wt",stdout);
Bonjour(); /* écrit Bonjour ! dans le fichier fich.txt */
...
```

6.8 Suppression d'un fichier

La fonction

`int remove(const char *nom_fichier)`

supprime le fichier de nom `nom_fichier` et retourne 0 ou une valeur non nulle s'il y a un problème.

6.9 Conseils

Le tableau ci-dessous fait une synthèse des principales fonctions généralement utilisées pour effectuer des lectures, des écritures et pour détecter la fin de fichier en fonction du type de fichier.

	Lecture			**écriture**	
Type fichier	*Mode ouverture*	*Fonctions*	*Détection fin de fichier*	*Modes ouverture*	*Fonctions*
Texte	`"rt"`	`fgetc`, `fgets` `fscanf`	`feof`	`"wt"`, `"at"`	`fputc`, `fputs` `fprintf`
Binaire	`"rb"`	`fread`	Valeur retournée par `fread`	`"wb"`, `"ab"`	`fwrite`

6.10 Remarques sur les fichiers binaires

- Accès plus rapide au contenu d'un fichier binaire :
  - pas de conversion des valeurs numériques ;
  - taille fixe des informations, donc pas de séparateur à gérer.
- Accès direct plus naturel. Par exemple, si on veut lire les 5 derniers entiers d'un fichier :
  `fseek(id_fich,-5*sizeof(int),SEEK_END);`
  Avec un fichier de texte, c'est beaucoup plus compliqué...
  Remarque : les fonctions `fseek` et `ftell` sont surtout utiles pour des fichiers binaires.
- Mais, problèmes de portabilité : pour un type donné, la représentation peut varier au niveau :
  - du nombre d'octets occupés (format) ;
  - du codage ;
  - de l'ordre des octets (*little endian* / *big endian*).

✎ Exercice 27

écrire une fonction d'en-tête :
`int Text2Bin(char NomFichSource[], char NomFichDest[])`
qui lit les entiers contenus dans le fichier de *texte* `NomFichSource` et les écrit, sous forme de `int`, dans le fichier *binaire* `NomFichDest`. On suppose que, dans le fichier de texte, les entiers sont séparés par au moins un caractère séparateur. Si le fichier `NomFichDest` n'existe pas, il doit être créé. S'il existe, son contenu doit être « écrasé ». Cette fonction doit retourner le nombre d'entiers qui ont été convertis ou un nombre négatif en cas d'erreur.

✎ Exercice 28

écrire une fonction d'en-tête :
`int Bin2Text(char NomFichSource[], char NomFichDest[])`
qui effectue la conversion inverse de la fonction `Text2Bin`, c'est-à-dire qui lit les `int` contenus dans le fichier *binaire* `NomFichSource` et les écrit, un par ligne, dans le fichier de *texte* `NomFichDest`. Si le fichier `NomFichDest` n'existe pas, il doit être créé. S'il existe, son contenu doit être « écrasé ». Cette fonction doit retourner le nombre d'entiers qui ont été convertis ou un nombre négatif en cas d'erreur.

✎ Exercice 29

écrire une fonction d'en-tête :
`int Nieme(char NomFich[], int n)`
qui retourne le n$^{\text{ème}}$ `int` du fichier *binaire* `NomFich` ou un nombre négatif en cas d'erreur (on suppose que les entiers contenus dans le fichier sont positifs).

Index