Cours de S.E. – le Système de Gestion des Fichiers

Andreea Dragut

Univ. Aix-Marseille

2012

Plan

Resources :

- Richard W. Stevens. Advanced Programming in the Unix Environment. Addisson-Wesley, 1992.
- Andrew Tanenbaum. Systèmes d'exploitation. Pearson, 2003
- Kay A. Robbins, Steve Robbins. UNIX Systems Programming : Communication, Concurrency and Threads. Prentice Hall, 2003.
- Michael Kerrisk. The Linux Programming Interface : A Linux and UNIX System Programming Handbook, 2010

Systèmes d'exploitation — catégories

- généralistes – ordinateurs « classiques »
- spécialisés
  - industriels – usine robotisée
  - application dédiée – caisse enregistreuse de supermarché
- interrogation grandes bases de données (bibliothèques, banques, etc.)
- transactionnels (réservations aériennes, etc.)

Systèmes d'exploitation — catégories

- embarqués (borne SNCF, distributeur billets de banque, terminal encaissement automatique)
- miniaturisé (téléphone cellulaire, agenda électronique, lecteur MP3)
- conditions difficiles (plateforme pétrolière, sous-marin robotisé)
- extrêmement robustes (centrale nucléaire, TGV, poste commande base militaire, objets volants identifiés ou pas)

Définition ?

Pas de définition universellement acceptée pour SE

- faire fonctionner le matériel
  - « allocateur » de ressources
  - gestionnaire de ressources avec un minimum de sécurité
    - arbitre – décide dans les cas conflictuels
    - superviseur/contrôleur de programmes
    - prévient les erreurs et l'utilisation non-conforme
- maintenir un espace virtuel

Composition des SE — logiciels

- logiciels pour effectuer/coordonner/sécuriser les accès aux ressources partagées (CPU, mémoire)
- logiciels dédiés pour piloter les périphériques
  - des procédures bas-niveau exécutées par le CPU pour le dialogue
  - coordonne et gère les échanges, leurs moments, les tampons pour entreposer les données
- logiciels pour organiser et retrouver les données en mémoire (de masse)
  - une structuration plus « haut niveau » de l'information sur le périphérique – ex. – une structure hiérarchique de système de fichiers
- logiciels pour pour l'intéraction homme-machine (interfaces graphiques, texte, etc.) et machine-machine (en réseaux)

Composantes SE

- Composantes principales du SE :
  - le noyau (kernel) – partie centrale du SE contrôlant la plupart des ressources importantes
  - modules (LKM) – étendant les fonctionnalités du noyau
- Autres composantes du SE :
  - shells – permettant le dialogue humains $\leftrightarrow$ SE
  - le système X Window – gérant l'affichage (graphique)
  - gestionnaires de fenêtres (Window Managers) (« shells » pour X Window)

Architectures de noyau

- monolythique : réside en entier et en permanence dans la mémoire vive
  - présente des abstractions du matériel aux programmes
- micro-noyau : les modules sont chargés/dynamiquements en mémoire
  - seulement fonctionnalités « de base » : multitâche, gestion mémoire, communication et synchronisation entre processus
  - le reste (e.g. réseau) – applications (serveurs)
  - utilise principalement le passage de message
  - plus « léger », mais plus lent
- hybrides
- exo-noyaux
  - permettent l'accès direct aux composantes matérielles – bibliothèques de fonctions
  - assurent seulement le multiplexage de cet accès, et la protection mutuelles

Philosophie SE, Unix et Linux

Unix : né dans les annés '70

- hérite des concepts de SE généralistes précédents
- notion de noyau : partition virtuelle de la mémoire vive physique
- innovant
  - la simplification au maximum l'ensemble d'éléments primitifs
  - la définition de leurs relations avec peu de règles
- but – avoir un schéma global facile à maîtriser
- écrit en C, assez ouvert et l'aide en ligne est extrêmement compréhensive – le **man**uel.

Aide en ligne — manuel

Section	Type de commandes
1	commandes et applications utilisateur
2	appels système, codes erreurs noyau
3	fonctions de bibliothèque
4	pilotes de périphériques et protocoles réseau
5	formats de fichiers standard
6	jeux et démos
7	divers fichiers et documents
8	commandes d'administration système
9	divers specs noyau et interfaces

- *man -k*
- plus d'information : *man man*

Philosophie Unix et Linux – processus

- L'ordinateur exécute des programmes, qui transforment des données, le tout reposant en mémoire.
- **processus** — le concept de programme en train d'être exécuté
- Le SE
  - gère ces processus, leur permettant de vivre heureux et de s'épanouir dans un certain espace.

Processus

étudier les processus pour comprendre comment ils

- sont gérés, comment ils
- naissent, vivent, meurent et sont inhumés,
- sont organisés ensemble dans des familles,
- sont pilotés par le système, à travers ce qu'on appelle des signaux, etc.

Philosophie Unix et Linux – **Fichier** –le point d'entrée-sortie système

Notion Unix et Linux de fichier

- Notion basique : une collection de données reposant en **mémoire de masse**
- sans taille
  - un flot de données :pipes, sockets
  - un périphérique permettant l'accès aux composantes matérielles
    - 2 numéros :majeur(type de périph.), mineur (partition logique du périph.)
    - Ex. : 14 : premier port IDE sur la carte mére,
      le "fichier" /dev/disk0 est un accés direct au premier disque dur
      brw-r—— 1 root operator 14, 0 Sep 20 02 :58 disk0
      brw-r—— 1 root operator 14, 1 Sep 20 02 :58 disk0s1
      brw-r—— 1 root operator 14, 2 Sep 20 02 :58 disk0s2

Philosophie Unix et Linux – IPC

- Tâches complexes :
  - mieux d'avoir un ensemble de programmes coopérant **de manière coordonnée**
- Pourquoi ?
  - contrôler le fonctionnement plus facilement,
  - développer les parties indépendamment
  - efficacité pour plusieurs processeurs

Notion d'IPC

- Notion d'IPC : les programmes travaillant en équipe doivent pouvoir communiquer entre eux
  - les processus peuvent s'attendre mutuellement pour
    - partager l'accès à des ressources communes
    - dialoguer de manière cohérente
    - récupérer en sécurité des résultats les uns des autres
- Moyens IPC — paradigmes de contrôle
- sémaphores – les feux dirigeant la circulation
- pipes – téléphones

Opérations d'Entrées/Sorties E/S — organisation système

Travail CPU :

- rapide : calculs, accès mémoire vive,
- *lent* : accès mémoire de masse (disque dur, DVDROM, etc.), accès périphérique (réseau, etc.)

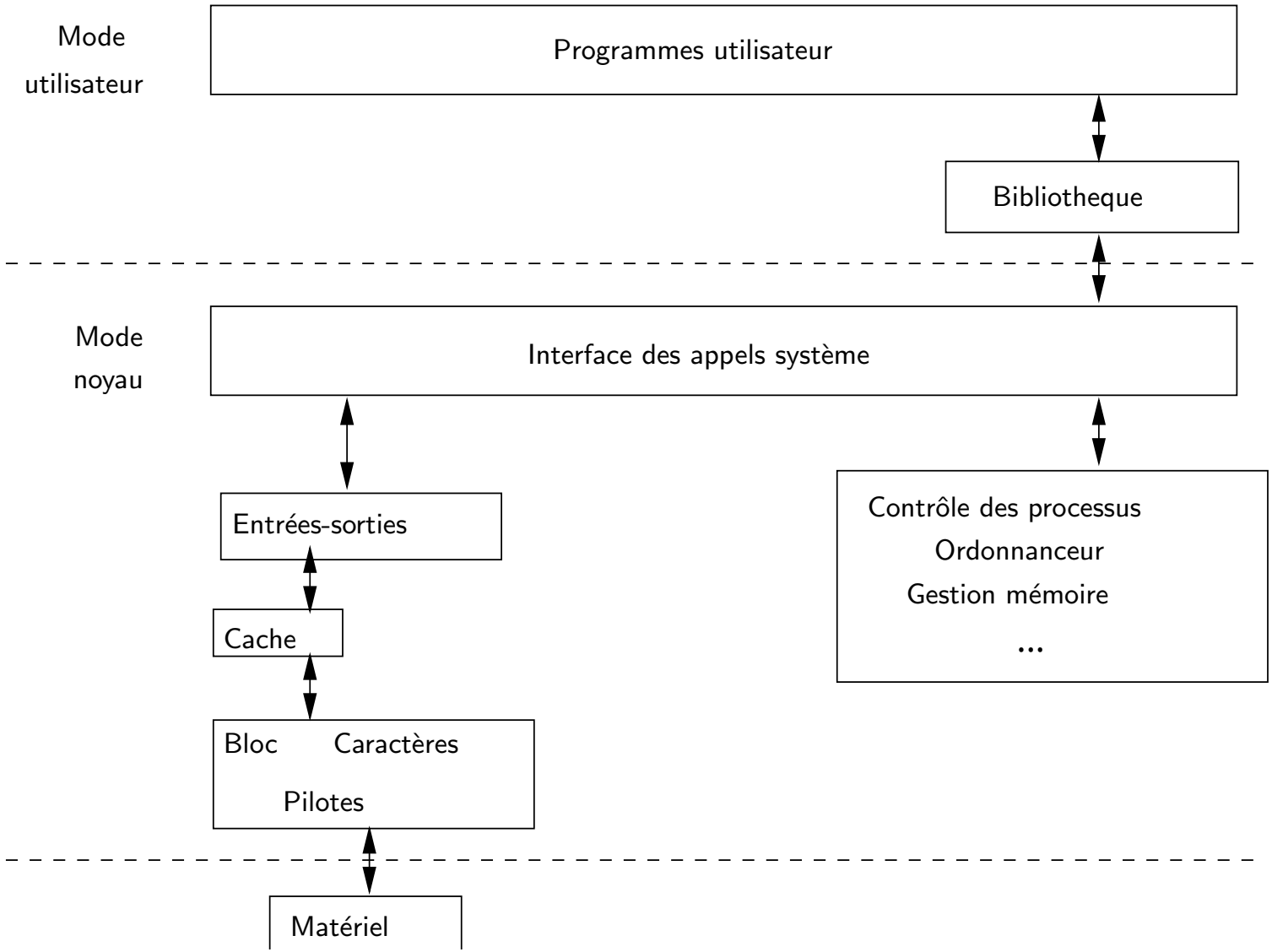
Des lectures/écritures efficaces pour les processus $\Longrightarrow$ **délégation travail** : le processus demandant l'E/S mis en attente, **commutation de contexte**

1. CPU – *donne l'ordre* de transfert
2. périphérique+DMA – *effectuent* le transfert
3. aussi – opération *initiée* par le périphérique lui-même (surtout écriture), e.g. réseau – étapes 2 et 3)

Quand le contrôle revient-il au programme depend du type de fichier

- Classification des périphérique
  - Block Device — groupes d'octets par transfert (disque dur, DVDROM, bande magnétique)
  - Character Device — octet par octet (clavier, souris, port série, périphs USB)
  - Périphériques avec protocoles spéciaux
- DMA — pour des périphériques rapids
  - le contrôleur de périphérique transfère des blocs de données « directement dans la mémore vive », sans faire intervenir le CPU
  - une interruption par bloc
  - le « chipset »– composante matérielle dédiée, qui gère effectivement le traffic – donne au contrôleur l'accès au bus mémoire

Arhictecture simplifie



Fonctionnement du CPU, commutation contexte

- **Assurer protection et cohérence** dans le **contexte multiprocesseur-multiutilisateur**,
- le CPU → deux modes de fonctionnement,
  - utilisateur ou user, ou protégé : accès au matdriel impossible, accès interdit à certaines zones mémoires et aux instructions privilégiées du noyau
  - noyau ou kernel, ou moniteur/superviseur, ou privilégié. – pas de restriction
- Gérés par commutation de contexte
  - interruptions : matérielles, logicielles
  - exceptions, traps

Appels système E/S, commutation contexte

Appels système : code privilégie écrit dans de zones mémoires prefixés.
Ils provoquent une interruption **commutant** l'UC en mode noyau, et **rendant en même temps la main**, i.e. ne gardant plus le contrôle.

- CPU sauvegarde son contexte,
- CPU abandonne temporairement le programme utilisateur
- CPU exécute du code se trouvant à des adresses précises, bien connues (et normalement protégées) — du code du noyau du SE
- il y a un timer de prévention de boucle infinie

Système de Gestion de Fichiers SGF

Unix — vision « unifiée »E/S

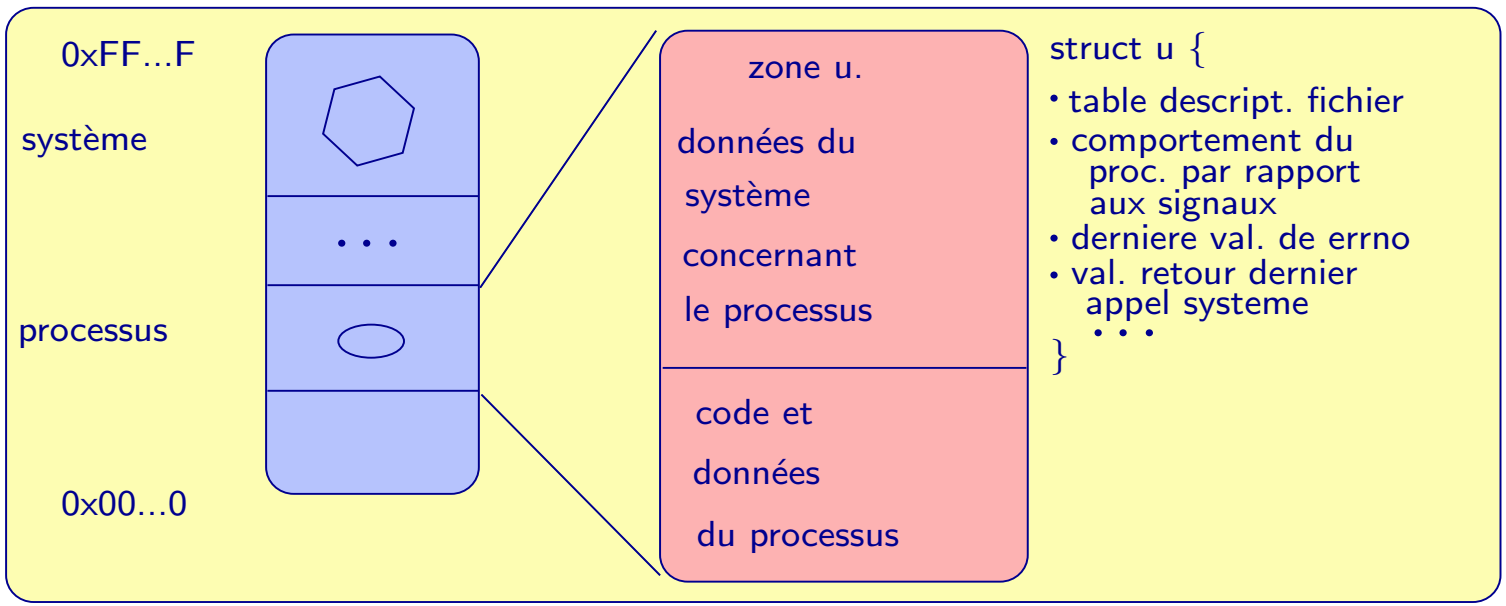
- Fichier : Point d'entrée-sortie
- Informations sur les fichiers
  - `stat()`, `fstat()`
- Création ou acquisition des droits de lecture/écriture d'un fichier existant et attribution d'un **Identifiant unique** – descripteur de fichier
  - `open()`
- Opérations lecture/écriture, positionnement
  - `read()`, `write()`, `lseek()`
- Partage/verrouillement d'un fichier
  - `dup2()`, `fcntl()`
- Fermeture
  - `close()`
- Destruction d'un fichier
  - `unlink()`

C'est quoi un *descripteur de fichier* ?

- *pour chaque ouverture de fichier* – un indice entier d'une table de pointeurs :
  - gestion « session d'ouverture » fichier,
  - même processus $\rightarrow$ plusieurs ouvertures du même fichier $\rightarrow$ plusieurs descripteurs
  - généralise notion de communication — partie des *IPC* —
    - pipes
    - sockets
    - périphériques

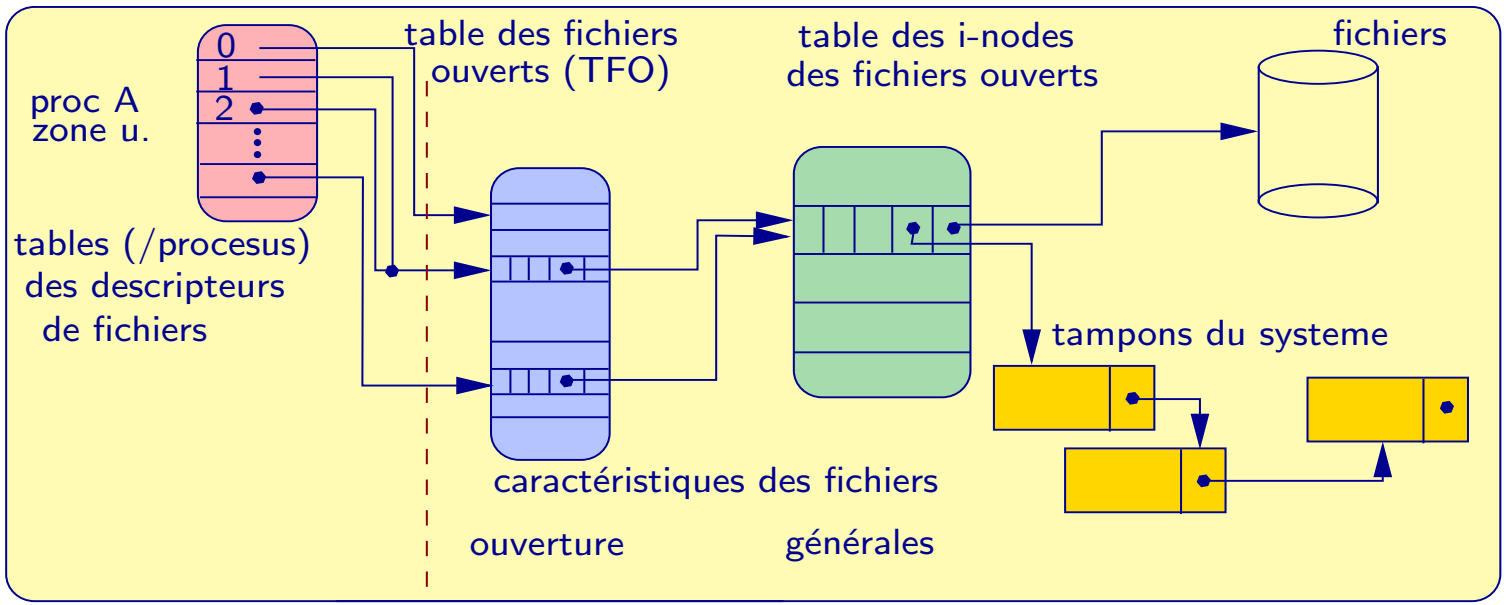
Où est logée la table des descripteurs ? — structure de la mémoire

- mémoire → processus → espace d'adressage virtuel.
- zone haute du mémoire processus — découpe partie réservée au système — zone u-dot, notée u.



Descripteur — i-node

- descripteur de fichier — indice dans une table de pointeurs → table de fichiers ouverts → table des i-nodes des fichiers ouverts.



- i-node/i-nœud ?
  - "nœud d'information" structure de données système de gestion de fichiers qui regroupe des renseignements sur un fichier

Un peu d'ordre – quel renseignement va où ?

- Informations uniques, globales au niveau d'un fichier, indépendantes des processus et des ouvertures **sauf son nom**→ *répertoire* :
  - dates de : dernier accès, derniére modification du fichier et de ses attributs
  - droits d'accès,
  - type de fichier, le périphérique où réside le fichier,
  - le nombre de liens vers le fichier,
  - taille, etc.
- Informations pertinentes pour une « session d'ouverture » (appel à `open()`) :
  - mode d'ouverture (lecture, écriture, etc.), drapeaux « status »(close-on-exec) :
  - position courante de lecture/écriture : offset
  - verrous
  - le nombre de descripteurs (d'ouvertures du fichier).

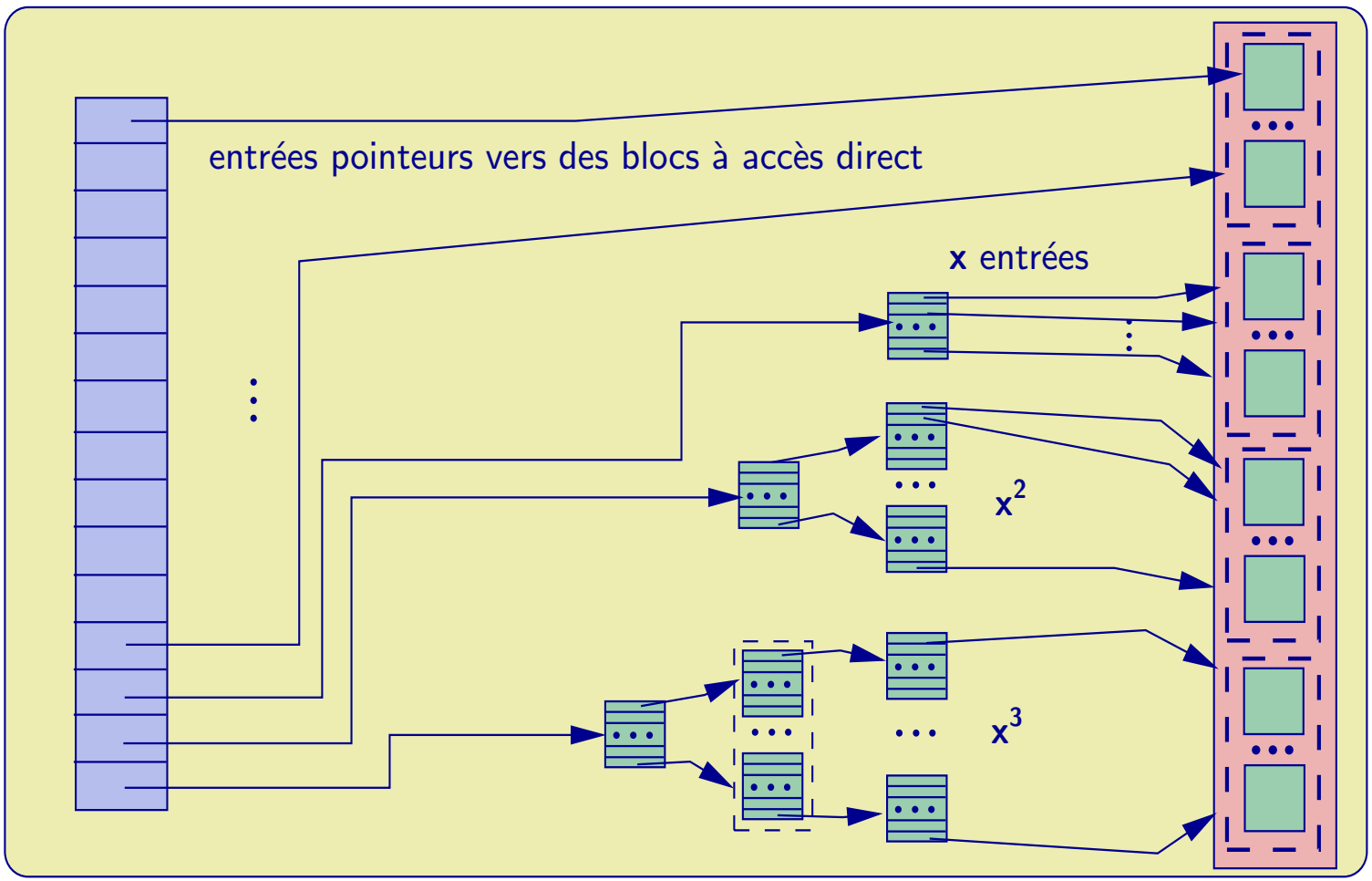
Table des blocs — i-node

Dans `ext2_inode`, `ext3_inode` la table des blocs : nombre fixe d'entrées, 15

- les 12 premières pointent directement sur le bloc correspondant
- les trois autres pointent sur des blocs contenant eux-même des tables de blocs — structure arborescente à plusieurs niveaux, avec $x \geq 256$ tels numéros logeables.
  - entrée no. 12 sur une table d'entrées-pointeurs sur blocs de données — x **blocs**
  - entrée no. 13 sur une table d'entrées-pointeurs sur blocs contenant des tables de blocs avec des pointeurs sur des blocs de données — x^2 **blocs**
  - entrée no. 14 sur une table d'entrées-pointeurs sur blocs contenant des tables de blocs avec des pointeurs sur des tables de blocs avec des pointeurs sur des blocs de données — x^3 **blocs**

Table des blocs

Taille bloc 4Ko, Adresse sur 32bits=4octets : x=1024
Taille max. d'un fichier UNIX : $1024(10 + 1024 + 1024^2 + 1024^3) \backsimeq 4$To



Renseignements sur un i-node

- Un **i-node** est décrit par une `struct stat`

```
struct stat {
 dev_t     st_dev;     // device
 ino_t     st_ino;     // inode
 mode_t    st_mode;    // type de fic. et permissions
 nlink_t   st_nlink;   // nbre liens
 uid_t     st_uid;     // du proprietaire
 gid_t     st_gid;     // du proprietaire
 dev_t     st_rdev;    // type device (si device)
 off_t     st_size;    // taille
 blksize_t st_blksize; // taille pour E/S
 blkcnt_t  st_blocks;  // nbre blocs alloues
 time_t    st_atime;   // dernier acces
 time_t    st_mtime;   // derniere modif
 time_t    st_ctime;   // dernier chgm etat
};
```

Le champ *st_mode* de la structure `struct stat`

- type de fichier : examiné avec des macros `S_ISREG`,`S_ISDIR`,`S_ISCHR()`,`S_ISFIFO`,`S_ISLNK`,`S_ISSOCK`, ...
- droits d'accès, e.g. `S_IRUSR` – lecture pour propriétaire, `S_IWUSR`, `S_IXUSR`, `S_IRGRP`, etc.
- droits d'accès spéciaux– trois bits spéciaux
  - le bit **set UID** (`S_ISUID`)– pour un processus exécutant ce fichier : l'UID effectif ← l'UID du propriétaire du fichier (SUID=l'UID du propriétaire)
  - le bit **set GID** (`S_ISGID`) – même chose pour le GID
  - le bit **sticky** (`S_ISVTX`) – pour un répertoire – tout élément peut en être détruit ou renommé seulement par son propriétaire, ou par celui du répertoire ou par un processus privilégié (utilisé pour /tmp)

Renseignements sur un i-node

- La fonction stat()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *chemFic, struct stat *buf);
```

- le systéme recupere le statut du fichier pointé par chemFic et remplit une variable de type : struct stat ;
- peut être utilisè sans les droits d'accès à la cible , mais il faut des droits d'ouverture sur tous les répertoires intermédiaires y conduisant
- rend -1 si erreur, et *errno* en contient la raison

Gestion des erreurs, valeurs d'*errno*

- un appel système echoué positionne la variable globale *errno* (déclarée dans <errno.h>)
- la strerrorr() (déclarée dans <string.h>) → message explicatif.

```
//... avec les autres d'avant
#include <string.h>
#include <errno.h>
#include <iostream>
//...
  struct stat buf;
  ...
  const int e=stat("fic1.txt", &buf);
  if(e == -1) {
    std::cerr << "ERR␣ouvrt.␣fic1.txt␣lect.:␣"
              << strerror(errno) << "\n";
  }
```

Si $e \rightarrow \boxed{-1}$ souvent *errno* :
ENOENT — fichier inexistant, EACCESS — pas d'accès permis

Wrapper de stat()

- **déclaration** — exceptions classe CExc (*throw* – en-tête).
- **test implicite** valeur de retour stat() — pas d'erreur si et seulement si elle est nulle ;
- **si erreur**, **on leve** **une exception** contenant un maximum d'information — avec l'instruction *throw*.

```
Stat(const char* chemFic, struct stat*buf) throw(CExc){

 if(::stat(chemFic, buf)) throw CExc("stat()",chemFic);
} // Stat()
```

- Constructeur exception — stat(), chemin d'access au fichier *chemFic*.
- CExc – notre classe pour les exceptions

Exceptions, espaces de noms — detail TDTP

- Bases
  - `exception` — classe standard, mais très pauvre —
  - notre `CExc`
    - dérivée de la classe `exception`
    - données-membres pour stocker les renseignements du contexte (nom de fonction, paramètres, raison d'erreur, etc.)
    - méthode `_Edit()`, affichant ces renseignements ainsi que *errno*, appelée par la surcharge de << pour `CExc`
- Espaces de noms
  - Limiter conflits noms → isoler les noms dans des ensembles.
  - accès —
    - préfixer l'identificateur,
    - annoncer avec `using`.

Wrappers — detail dans TDTP

- Exceptions de C++, → enrober chaque fonction système dans une fonction à nous,
  - **étudie** *la valeur de retour* et **l'erreur** éventuelle suite appel
  - construit et lève une **exception** appropriée → **transporte renseignements** (nom de fichier, ou descripteur de fichier, erreur, etc.).
  - est **appelée** dans un `try-catch` → filtrage des exceptions.
- **nom** wrappers — même nom que la fonction système enrobée, majuscule : `Stat()`

Exceptions — **arrêtent automagiquement toute fonction en cours**, jusqu'au premier `try-catch` englobant **prévu pour la capturer**.

Remarque

S'il n'y a pas *de tel* `try-catch`, *alors 'Abort'.*

Exemple dialogue – copie de fichiers

Shell :

`machine %` cp fichierSource fichierDestination

Programme ?
Utiliser `open()`, `close()`, `read()`, `write()` pour

1. **ouvrir fichiers** en **lecture**, respectivement **écriture**
2. **lire octets** : **fichier source** → zone mémoire
3. **écrire octets** : zone mémoire → **fichier destination**
4. **recommencer au pas 2** tant que ∃ à lire
5. **fermer fichiers**.

Remarque

Aucune importance

- *signification contenu*
- *« organisation » logique*

Fichier = succession d'octets.

Programme de copie de fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
....
#include CExc.h
int main() {
  const int fdSource = Open("ficSource.txt",O_RDONLY);
  const int fdDest   = Open("ficDest.txt",
                            O_WRONLY | O_CREAT,0700);
  const size_t  NbBytes  = 512; char Tampon[NbBytes];
  size_t nbLus = 0;
  while( (nbLus = Read(fdSource,Tampon,NbBytes)) > 0){
      Write(fdDest, Tampon, nbLus);
  }
  Close(fdSource);Close(fdDest);
  return 0;
}
```

Commentaire sur le programme de copie de fichier

- Inclure en-têtes
- **Ouverture** fichiers → noms.
- **Lecture**, **écriture** → file descriptors
- Lecture : fichier → zone mémoire réservée programme
- Écriture : zone mémoire → autre fichier
- Fermeture accès.
- Travail → `read()` et `write()` – noyau

Questions simples

- Généralité ? open() est-il pareil pour tout type de « fichier » ?
- Paramètres — `open()`, `close()`, `read()`, `write()` ?
- Choix valeurs paramètres ?
- Comment ça marche ?

Quelques réponses — ouverture fichier

- Ouverture de fichier, obtention de descripteur de fichier : fonction open()

```
#include <fcntl.h>
int open(const char *path, int oflag);
int open(const char *path, int oflag, mode_t mode);
```

- Nom fichier — NTCS *path*, '\0'
- *oflag*
  - genre (lecture, écriture)
  - manière ouverture (création, tronquature, etc.)
  - comportement (écriture immédiate ou différée, etc.)
- *mode* → SEULEMENT création O_CREATparmi les drapeaux : droits d'accès
- *valeur de retour* :
  - −1 si erreur,
  - positive → *descripteur de fichier*

Drapeaux ouverture fichier

- genre :
  - `O_RDONLY` **lecture uniquement**
  - `O_WRONLY` **écriture uniquement**
  - `O_RDWR` **lecture et/ou écriture**
- manière ouverture :
  - `O_CREAT` (écriture : `O_WRONLY` ou bien `O_RDWR`) : **crée le fichier** lors de l'ouverture, **s'il n'existe pas** ; (si pas spécifié, alors **échec si le fichier n'existe pas**),
  - `O_TRUNC` (écriture : `O_WRONLY` ou bien `O_RDWR`) : **tronque le fichier**
  - `O_EXCL` (en association avec `O_CREAT` : **crée le fichier s'il n'existe pas**, sinon **échec** — assure l'ininterruptibilité — cohérence concurrence processus)

L'open() — comment s'en sert-on ?

- Plusieurs options *oflag* → OU binaire — opérateur |.
  - O_WRONLY, etc. — ensembles disjoints bits 1, | → union d'ensembles.
  - Par exemple, O_WRONLY vaut 1 et O_CREAT vaut 64 = 2^6.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main() {
  const int fdSource = Open("ficSource.txt",O_RDONLY);
  const int fdDest   = Open("ficDest.txt",
                            O_WRONLY | O_CREAT,0700);
```

- variables (NTCS) pour noms de fichiers,
- Attention au transtypages char * ⇔ string

Fichier Unix – identifiant unique– différents protocoles E/S

en fonction de leur débit et de la sûreté de la communication :

- Protocole bloquant – J'attends quelque chose et je pars avec la première tranche qu'on m'a donnée :Block Device, Character Device
- Protocole non-bloquant – Je regarde, et repars tout de suite, avec ou sans : pipes, sockets
- Protocole asynchrone – **Dis-moi quand tu seras prêt** : sockets
- Pour les fichiers — sur disque, DVDROM, etc.
  - seulement protocole bloquant – service « rapide »
  - entrelacement E/S-calcul pour plusieurs processus

Drapeaux ouverture fichier– Continuation

- comportement des opérations dédiées seulement à certains types de fichiers :
  - `O_APPEND` écriture **en fin de fichier**
  - `O_SYNC` **écriture immédiate, non-tamponnée** — trés lent, mais sûr
  - `O_NONBLOCK` — sockets et pipes (et non pas pour fichiers) — plus tard

Lecture depuis un fichier

- Lecture → **descripteur de fichier** : fonction `read()`

```
#include <unistd.h>


ssize_t Read(int fd, void *buf, size_t NbALire);
```

- `read()` — **ESSAIE de lire** *NbALire* octets : descripteur de fichier *fd*, → la zone d'écriture : mémoire pointée par *buf*. **Il faut la réserver !**
- *buf* — pointer vers au moins *NbALire* octets **déjà réservés**
- *fd* — descripteur de fichier **déjà ouvert en lecture**.
- Rend nombre octets lus, ou zéro si fin de fichier, ou -1 si erreur.
- Appels successifs — suite — **indicateur interne de position** (offset).
- Aussi **sockets**, **pipes**, **terminaux**, etc. — comportement différent : si `O_NONBLOCK` (lors de l'`open()`), alors si pas de données → -1 ; plus tard.

Écriture dans un fichier

- Écriture sur un **descripteur de fichier** (pas un nom de fichier) : fonction `write()`

```
#include <unistd.h>
ssize_t Write(int fd,const void *buf,size_t NbAEcrire);
```

- `write()` **ESSAIE d'écrire** *NbAEcrire* octets : la zone de lecture : zone mémoire (non-modifiable) pointée par *buf* → descripteur de fichier *fd*
- *buf* — pointer vers au moins *NbAEcrire* octets **déjà réservés**,
- *fd* — descripteur de fichier **déjà ouvert en écriture**.
- Rend nombre d'octets écrits (≥zéro), ou -1 si erreur.
- Des appels successifs — suite — **indicateur interne de position**.
- Aussi **sockets**, **pipes**, **terminaux**, etc. — comportement différent : si `O_NONBLOCK` (lors de l'`open()`), alors si tampons pleins, etc. → -1 ; plus tard.

Le `write()` — comment s'en sert-on ?

- *Tampon* — réserve mémoire — lecture dedans — écriture depuis.

```
// ...
const size_t  NbBytes  = 512;char Tampon [NbBytes];
ssize_t nbLus = 0;
while( (nbLus = Read(fdSource,Tampon,NbBytes)) > 0){
    Write(fdDest, Tampon, nbLus);
}
// ...
```

- `write()` → *valeur de retour* de `read()`.
- Boucle → fin de fichier → `read()` rend zéro
- Astuce C/C++ : affectation dans la foulée de *nbLus*.
- Gestion des erreurs — sommaire : arrêt en cas de problème.

Fermeture de l'accès à un fichier

- Fermeture d'un **descripteur de fichier** (pas un nom de fichier) : `close()`

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

- `close()` **ferme** le **descripteur de fichier** *fd* (réutilisable, autres `open()`, etc.)
- *valeur de retour* — zéro si succès, et -1 sinon.
- Sans `O_SYNC` $\rightarrow$ écritures tamponnées, et ce n'est même pas le succès de `close()` qui indique l'écriture réelle
- ne pas négliger le cas d'erreur — parfois seulement là — erreurs d'écriture d'un `write()` précédent.
- Étude erreurs

Positionnement dans un fichier

- Positionnement de la position de la prochaine écriture ou lecture dans un fichier : `lseek()`, utilisant son **descripteur de fichier**

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int direction);
```

- `lseek()` **fixe** la position courante d'E/S pour le **descripteur de fichier** *fd*
- rend *la nouvelle position* relative au début du fichier, -1 si erreur
- Trois valeurs possibles pour *direction* :
  - `SEEK_SET` exactement *offset*
  - `SEEK_CUR` position actuelle + *offset*
  - `SEEK_END` taille du fichier + *offset*
- positionnement au delà de la fin du fichier (sans changer sa taille), et création de trous par écriture forcee.
- `off_t` est un sous-type du long int
- erreurs : mauvais arguments, ou opération impossible (sur un pipe, un tty, etc.), ou overflow de `off_t`

Manipulation des drapeaux d'un fichier - `fcntl()`

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);// operation
int fcntl(int fd, int cmd, long arg);//parametre
```

- *cmd* pour drapeaux d'un **descripteur de fichier** *fd*
  - `F_GETFL` obtention (valeur de retour)
  - `F_SETFL` changement selon *arg* :
    - `O_APPEND`,
    - `O_DIRECT` (minimisation tampons système),
    - `O_NOATIME` (non-changement temps accès `read()`),
    - `O_NONBLOCK`, `O_ASYNC` (pour terminaux, sockets, etc.)
- LA fonction pour un "file control" complet, ses arguments depend de la "cmd" utilisée

OPTIONNEL – manipulation des paramètres d'un fichier – suivi

- Avertissement lorsqu'un autre processus tente d'ouvrir ou effectuer une troncation d'un fichier, avec *cmd* :

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);// operation
int fcntl(int fd, int cmd, long arg);// parametre
```

- Valeurs de *cmd* :
  - F_SETLEASE pour établir (ou supprimer) le suivi, selon *arg* :
    - F_RDLCK simple notification
    - F_WRLCK pareil, mais réussira uniquement si aucun autre processus n'a déjà ouvert le fichier
    - F_UNLCK suppression
  - F_GETLEASE pour savoir si on est en train de suivre (*valeur de retour* parmi celles pour *arg* pour F_SETLEASE)

OPTIONNEL – manipulation des paramètres d'un fichier – suivi

- Avertissement lors d'changement dans un répertoire : *cmd* est alors F_NOTIFY, et *arg*

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd); // operation
int fcntl(int fd, int cmd, long arg);//parametre
```

  - DN_ACCESS – un fichier a été lu
  - DN_MODIFY – un fichier a été modifié (écrit, tronqué)
  - DN_CREATE – un fichier a été crée
  - DN_DELETE – un fichier a été enlevé
  - DN_RENAME – un fichier a eu son nom changé dans ce répertoire
  - DN_ATTRIB – les attributs d'un fichier ont été changés (permissions, propriétaire, temps, etc.)

OPTIONNEL – manipulation des paramètres d'un fichier – autres

- On peut examiner/changer le bit « close-on-exec »(`F_GETFD` et `F_SETFD`)
- On peut dupliquer un descripteur de fichier (`F_DUPFD`)
- On peut demander au système d'être averti lorsque des opérations d'E/S deviennent possibles (non pas pour les fichiers usuels, mais pour les terminaux, les sockets, etc.) : *cmd* sera `F_GETOWN`, `F_SETOWN`, `F_GETSIG`, `F_SETSIG`, etc.
- On peut également poser des verrous indicatifs (advisory) ou impératifs (mandatory) pour réserver l'accès à un fichier – à voir plus tard (IPC)

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, struct flock *lock);
```

Répertoire – L'organisation hiérarchique du SGF

- liste chaine d'**entrées de répertoire** `struct dirent` – fichiers, liens, fifos, peripheriques ou autres sous-répertoires
- dans la `struct dirent` : le champ POSIX *char d_name [NAME_MAX+1]* → obtention du nom effectif de l'élément

```
struct dirent {

    ino_t          d_ino; /*inode number */
    off_t          d_off; /*offset to the next dirent*/
    unsigned short d_reclen;/*length of this record */
    unsigned char  d_type;  /*type of file */
    char           d_name[256];/*filename */
};
```

Répertoire – nom d'un fichier – i-node

- stockent pour tout objet du SGF un **i-node** – un identificateur numérique « numéro de sécurité sociale »
- le seul endroit où l'on associe le **nom** de l'objet avec son **contenu** (par le biais de l'**i-node**)
- les fonctions « à utiliser » (POSIX, etc.) sont man 3 – bibliothèque
- la création, la destruction et la modification des éléments de (donc écriture du) répertoire se fait avec `open()` (ou `creat()`), `mkdir()`, et respectivement `unlink()` et `rmdir()`, et respectivement `rename()`

Manipulation de répertoire – création

- mkdir() (la commande éponyme du shell en est une sorte de wrapper)

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *nomDir, mode_t mode);
```

- *mode* est similaire à celui de open()
- rend 0 si succès, −1 si erreur, et *errno* en contient la raison

Manipulation de répertoire – ouverture pour la lecture

- opendir() (man 3 bibliothèque – utilise open() 2 interdit d'ouverture en écriture d'un répertoire

```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(const char *nomDir);
```

- rend un pointeur vers un flot de répertoire — DIR classe opaque
- on utilise dans la programmation interne un descripteur de fichier special obtenu par dirfd()
- rend 0 si succès, −1 si erreur, et *errno* en contient la raison

Manipulation de répertoire – lecture, parcours

- readdir() (man 3 – utilise en fait getdents()qu'on peut pas appeler)

```
#include <dirent.h>
#include <sys/types.h>
struct dirent *readdir(DIR *dir);
```

- une structure spéciale struct dirent pour obtenir les éléments du répertoire, un par un
- rend 0 si la fin de la liste est atteinte, ou bien si erreur
- $\Rightarrow$ il faut mettre *errno* à zéro explicitement avant un appel de readdir() et tester après *errno* contient un code d'erreur pour decider
- groupe supplémentaire de fonctions d'accès : rewinddir(), telldir(), seekdir(), etc.

```
DIR *dp;                    /** * flot de repertoire */
struct dirent *de;          /** * entree de repertoire */
struct stat s;
...
dp= Opendir (workingDir);
len_workingDir=strlen(workingDir);
while((de = Readdir(dp)) != NULL) {
    /* Sans .. et . */
    if((strcmp(de->d_name,"..")!=0)&&
        (strcmp(de->d_name,".")!=0)){

        /* Construction du chemin*/
        len= strlen(de->d_name);
        chemFic=(char*)malloc(len+
                              len_workingDir+2);
        strcpy(chemFic, workingDir);
        strcat(chemFic, "/");
        strcat(chemFic, de->d_name);

        Stat(chemFic, &s);
```

Manipulation de répertoire – fermeture

- La fonction `closedir()` (man 3) ferme le flot de répertoire

```
#include <dirent.h>
#include <sys/types.h>
int closedir(DIR *dir);
```

- rend 0 si succès, −1 si erreur, et *errno* en contient la raison

Manipulation de répertoire – renommage, déplacement

- rename() change le nom d'un fichier ou répertoire, etc., le changeant d'emplacement du répertoire initial au répertoire destination (si différence il y a)

```
#include <unistd.h>
int rename(const char *ancNm,const char *nvNm);
```

- si *nvNom* existe déjà, l'opération est atomique (aucun processus essayant d'y accéder n'échouera à cause de son absence), sauf pour certaines erreurs (voir le man...)
- rend 0 si succès, −1 si erreur, et *errno* en contient la

Manipulation de répertoire – destruction de fichier

- unlink() – enlève le nom spécifié du répertoire où il se trouve, et décremente son compteur de liens (physiques – car les symboliques ne sont pas répertoriées)

```
#include <unistd.h>
int unlink(const char *cheminFichier);
```

- si le compteur est maintenant nul et qu'aucun processus n'a le fichier ouvert, le fichier en question est vraiment supprimé et l'espace qu'il utilisait est rendu disponible
- si fichier est un lien symbolique, alors il est supprimé
- renvoie -1 si erreur et *errno* en contient la raison
- ne peut pas être utilisé pour les répertoires

Manipulation de répertoire – destruction

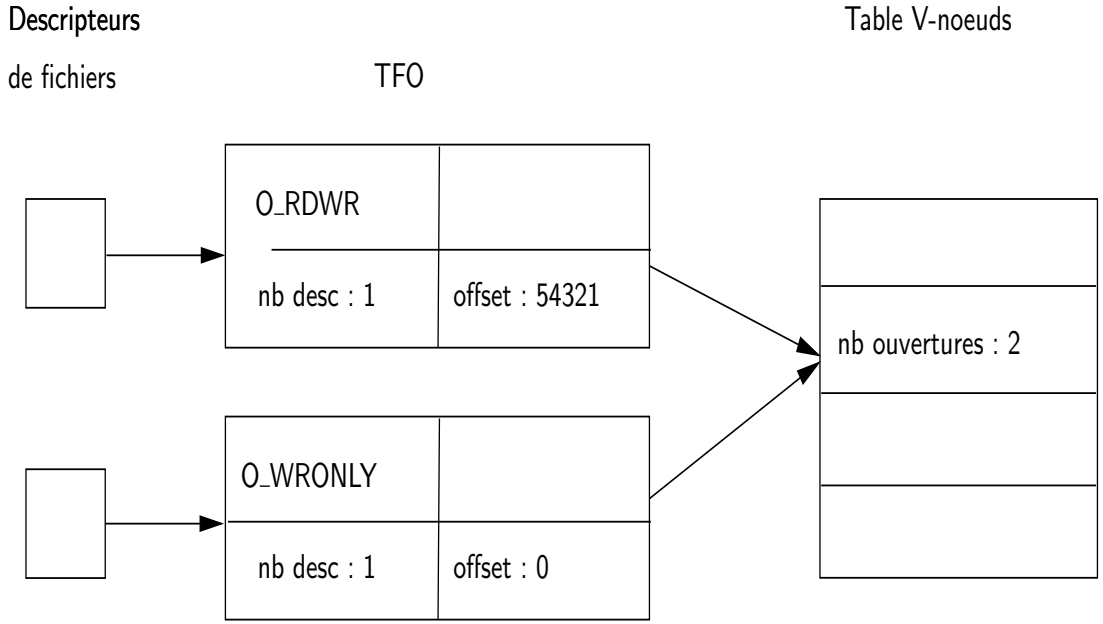
- rmdir() – supprime un répertoire, seulement lorsqu'il est vide (du répertoire qui le contient, bien entendu)

```
#include <sys/stat.h>
#include <sys/types.h>
int rmdir(const char *nomDir);
```

- rend 0 si succès, −1 si erreur, et *errno* en contient la

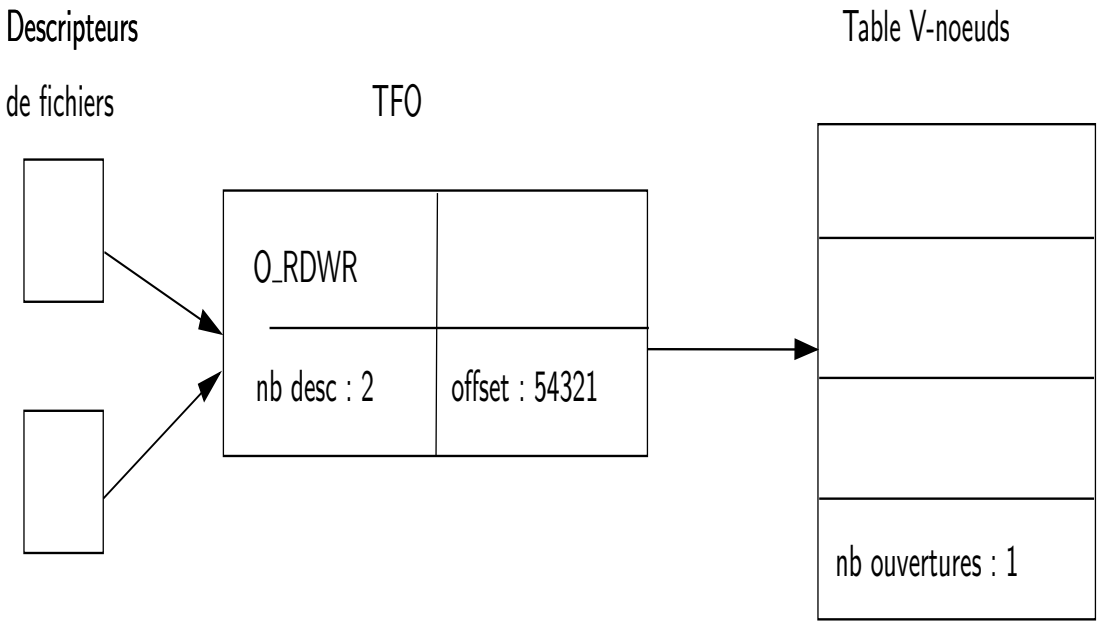
Partage du pointeur de lecture d'un fichier

- dans le meme processus : `dup2()`, `dup()`,`fcntl()` → obtenir copies *descripteurs de fichier*.
- dans deux processus distincts
  - **créés** uniquement **par copie** — `fork()` → duplique zone mémoire d'un processus ⟹ **fils** → reçoit (entre autres) du **père** copie *descripteurs de fichier* ouverts.
  - **modifiés par chargement** d'un exécutable (compilé, ou script) — `exec()` → laisse ouverts les *descripteurs de fichier* qui ne sont pas ouverts avec le drapeau close-on-exec.


Descripteurs
de fichiers
TFO
Table V-noeuds
O_RDWR
nb desc : 1
offset : 54321
O_WRONLY
nb desc : 1
offset : 0
nb ouvertures : 2

Pointeurs independants pour la lecture d'un fichier

Plusieurs appel d'open() – plusieurs éléments de la table des fichiers ouverts désigneront alors un même **i-node**



Duplication descripteurs de fichier

- dup2() – fait en sorte que *newfd* soit une copie de *oldfd* (et ferme *newfd* auparavant s'il était ouvert)

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

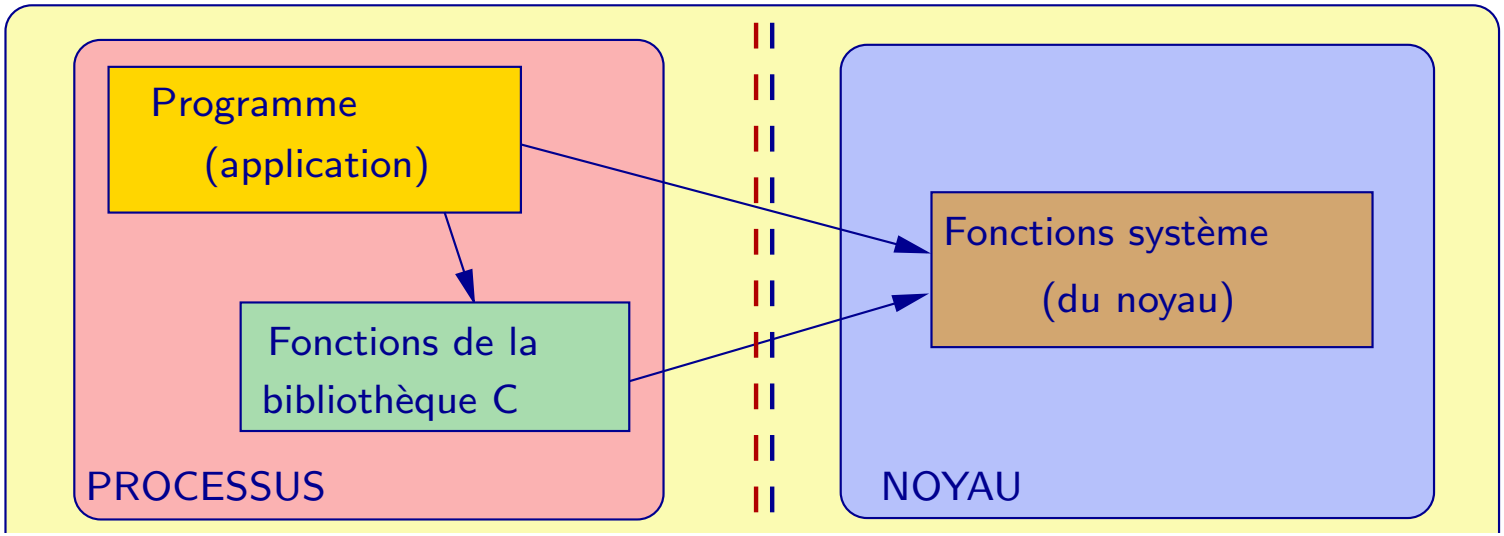
- Attention : faire un close(*newfd*) explicite avant le dup2() si *newfd* était ouvert
- En cas d'erreur – *valeur de retour* -1 et *errno* en contient la raison

Destruction de descripteurs

- `close()`
  - fermeture effective du canal E/S, libération élément table des descripteurs
  - décrémentation compteur références élément table des fichiers ouverts, et si arrivé à zéro, libération également
  - si le fichier réréncé avait été `unlink()`é, et qu'on vient de fermer le dernier descripteur ouvert dessus, alors suppression effective du fichier
- fermeture automatique par le système
  - lors de la terminaison d'un processus (normale ou pas) – comme `close()`, pour tous les éléments non-nuls de la table des descripteurs
  - lors d'un `exec()` (remplacement image processus par un nouveau programme, pour les close-on-exec

Fonctions système et fonctions de bibliothèque

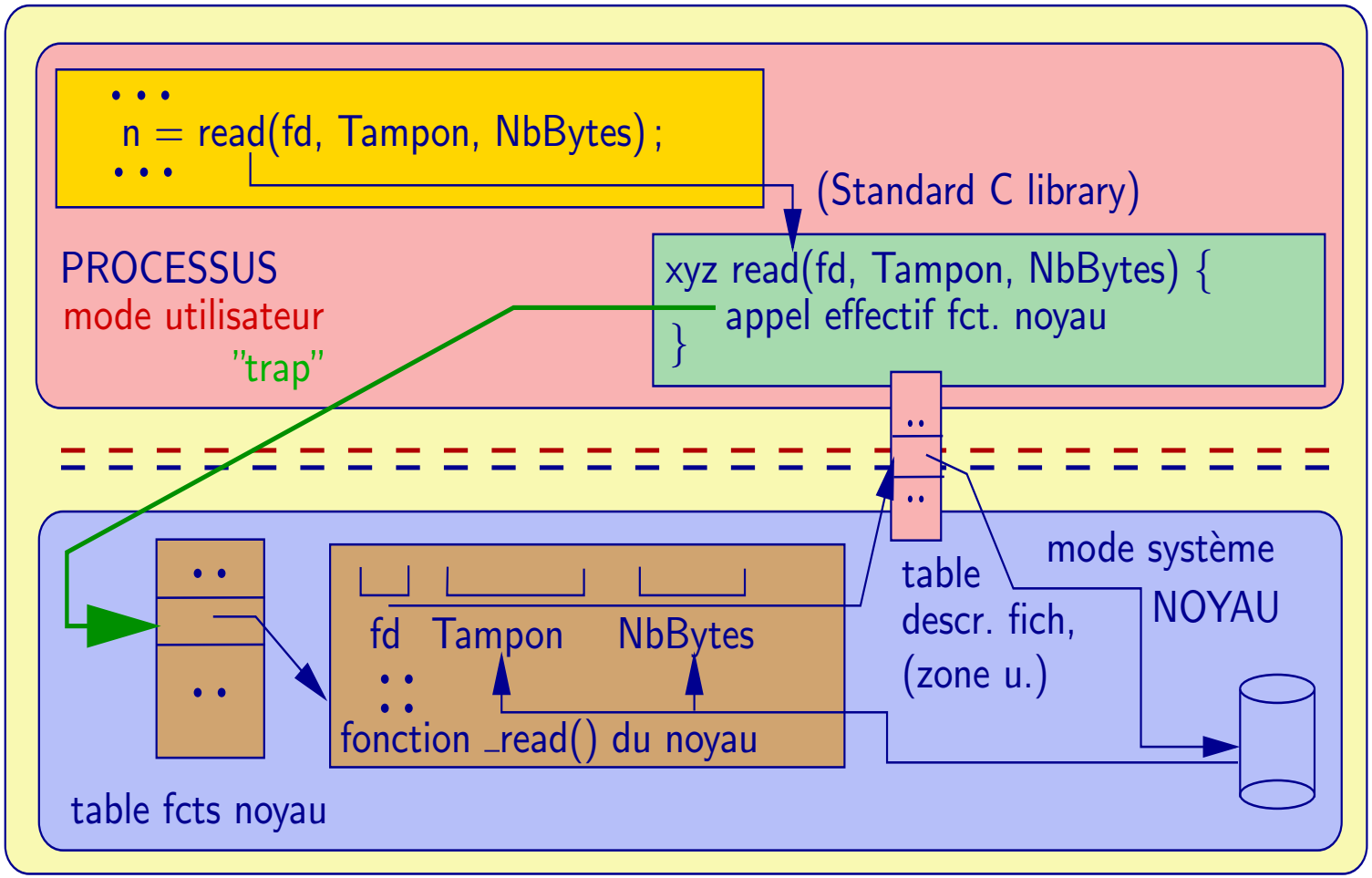
- **être précis** – distinguer
  - fonctions système
  - fonctions de bibliothèque standard C.
- travail effectif → noyau → fonctions système.
- bibliothèque standard → services supplémentaires : conversion, adaptation
- Exemple : `malloc()` → bibliothèque standard C
  - appelle `sbrk()` du noyau,
  - `sbrk()` augmente ou diminue taille totale de la mémoire virtuelle allouée au processus.
  - `malloc()` gère cet espace,
  - de nombreuses variantes.



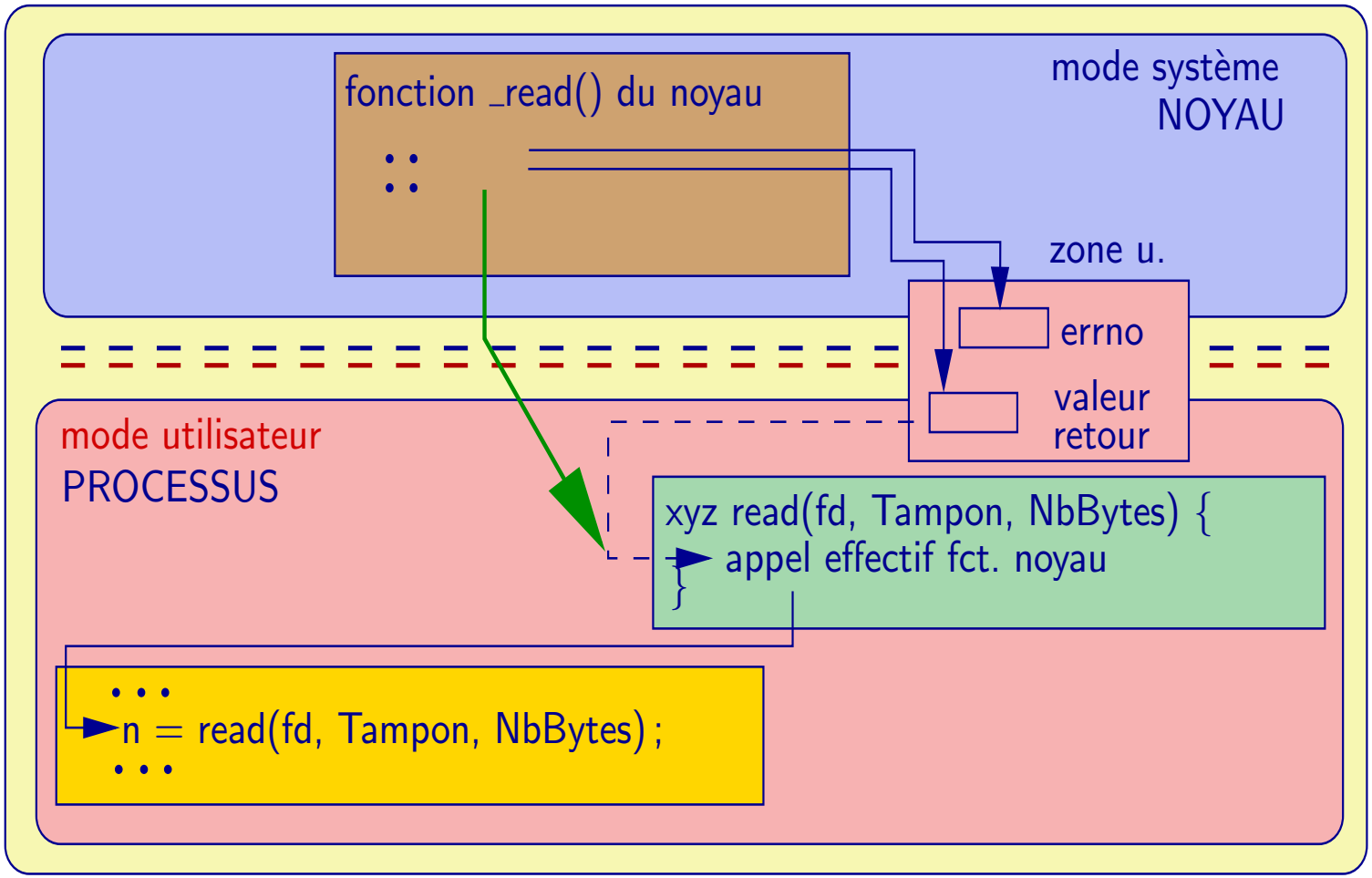
Coup d'oeil sous le capot de ces questions système

- Opérations fichiers → noyau.
- Programme utilisateur → dépose requêtes
- commutation de contexte :
  - CPU mode noyau — sans restrictions
  - CPU mode user — certaines restrictions (E/S, etc.)
- changement de contexte ⟹ sauvegardes registres CPU, etc. → **temps**, **ressources**
- pierre angulaire des SE.

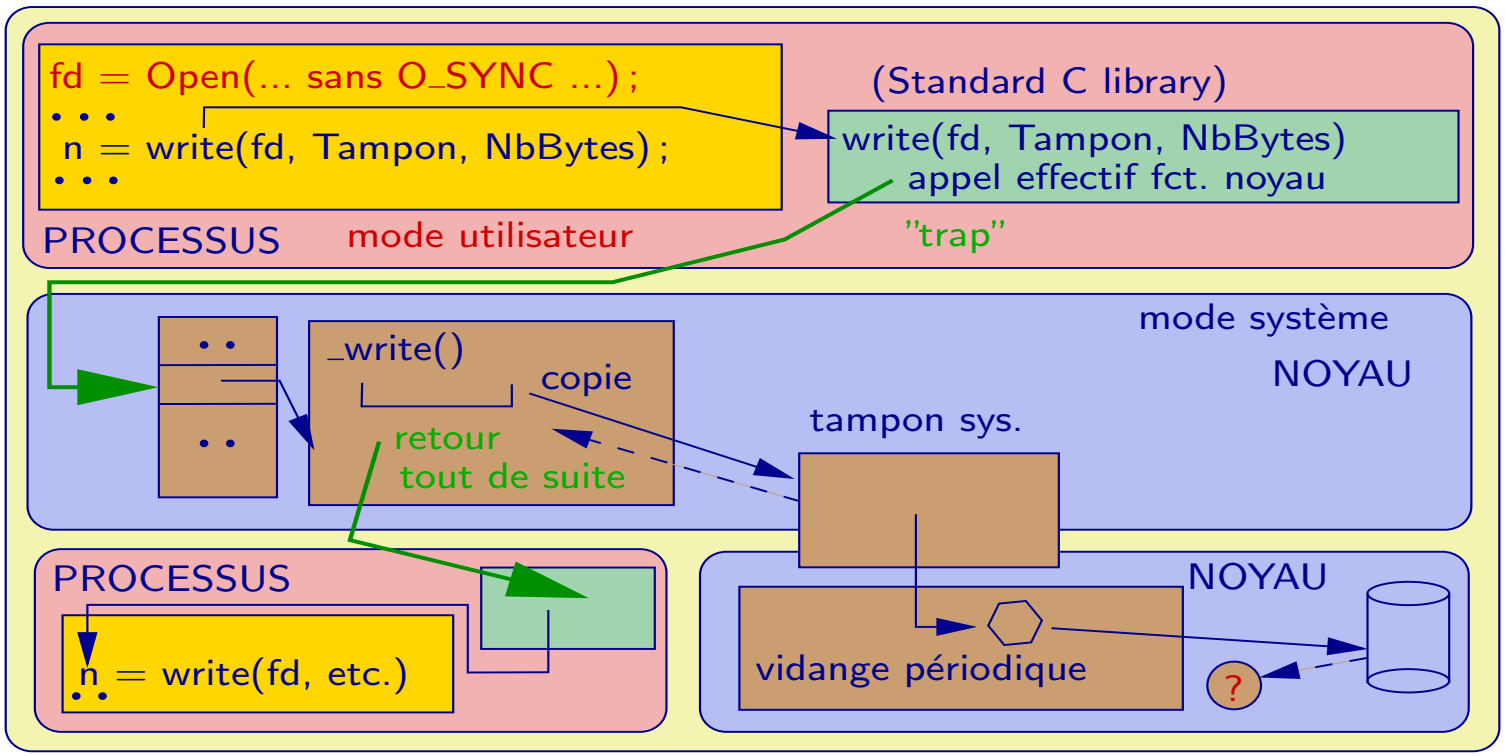
Exemple — `read()`



Suite de l'exemple de `read()`

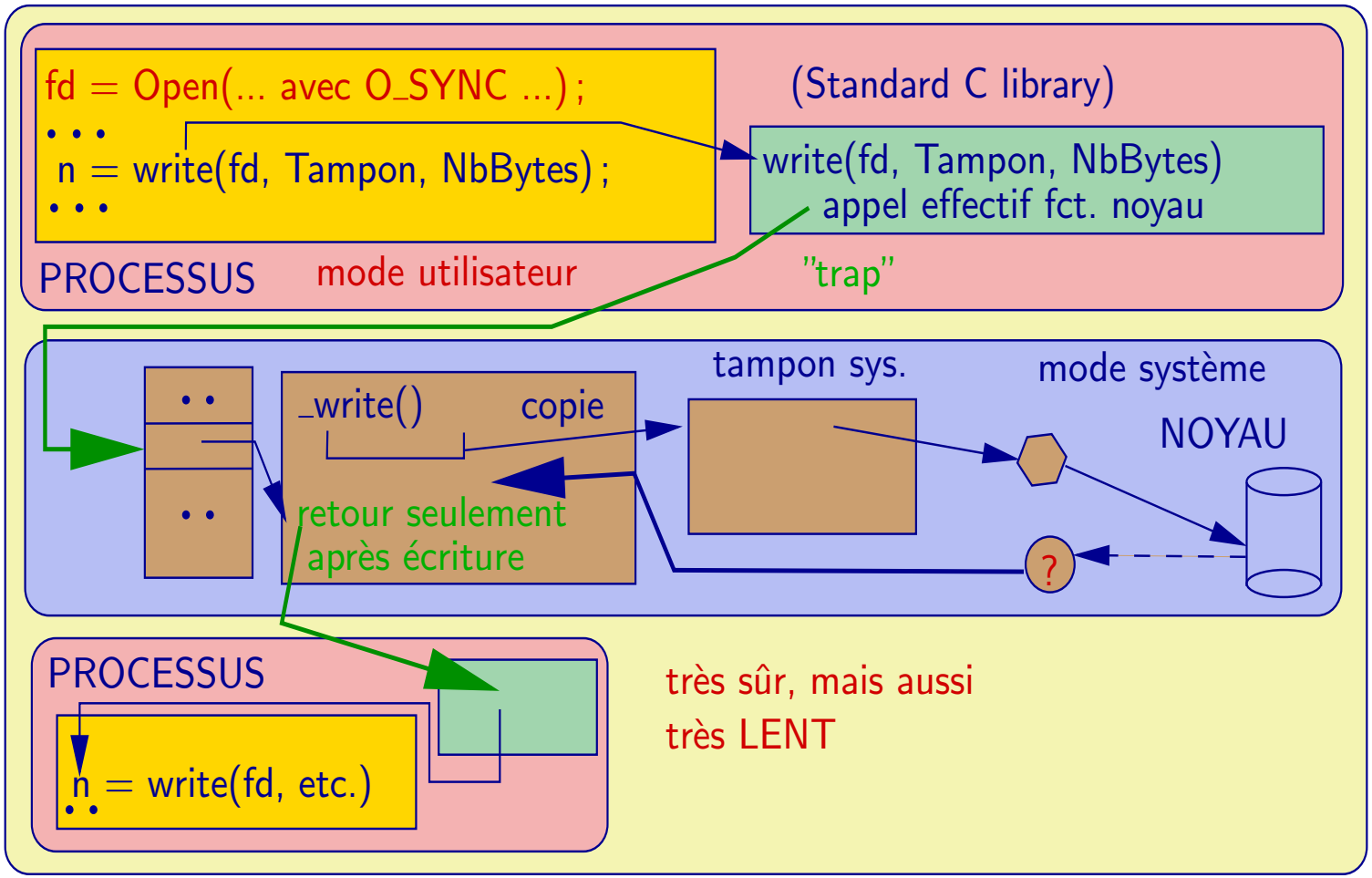


Exemple — `write()`, après `open()` sans `O_SYNC`



Les éventuels problèmes lors de l'écriture physique sur disque peuvent survenir APRÈS le retour de `write()` dans le programme de l'utilisateur ⇒ perte d'informations.

Exemple — `write()`, après `open()` avec `O_SYNC`



Le passage des paramétres des appels système

- dans des registres — problème si plus de paramètres que de registres
- sauvegardés dans un bloc/table/zone mémoire dont l'adresse est mise dans un registre
- empilés sur la pile par le programme, lus et dépilés par le SE

Détails internes du déroulement de `open()`

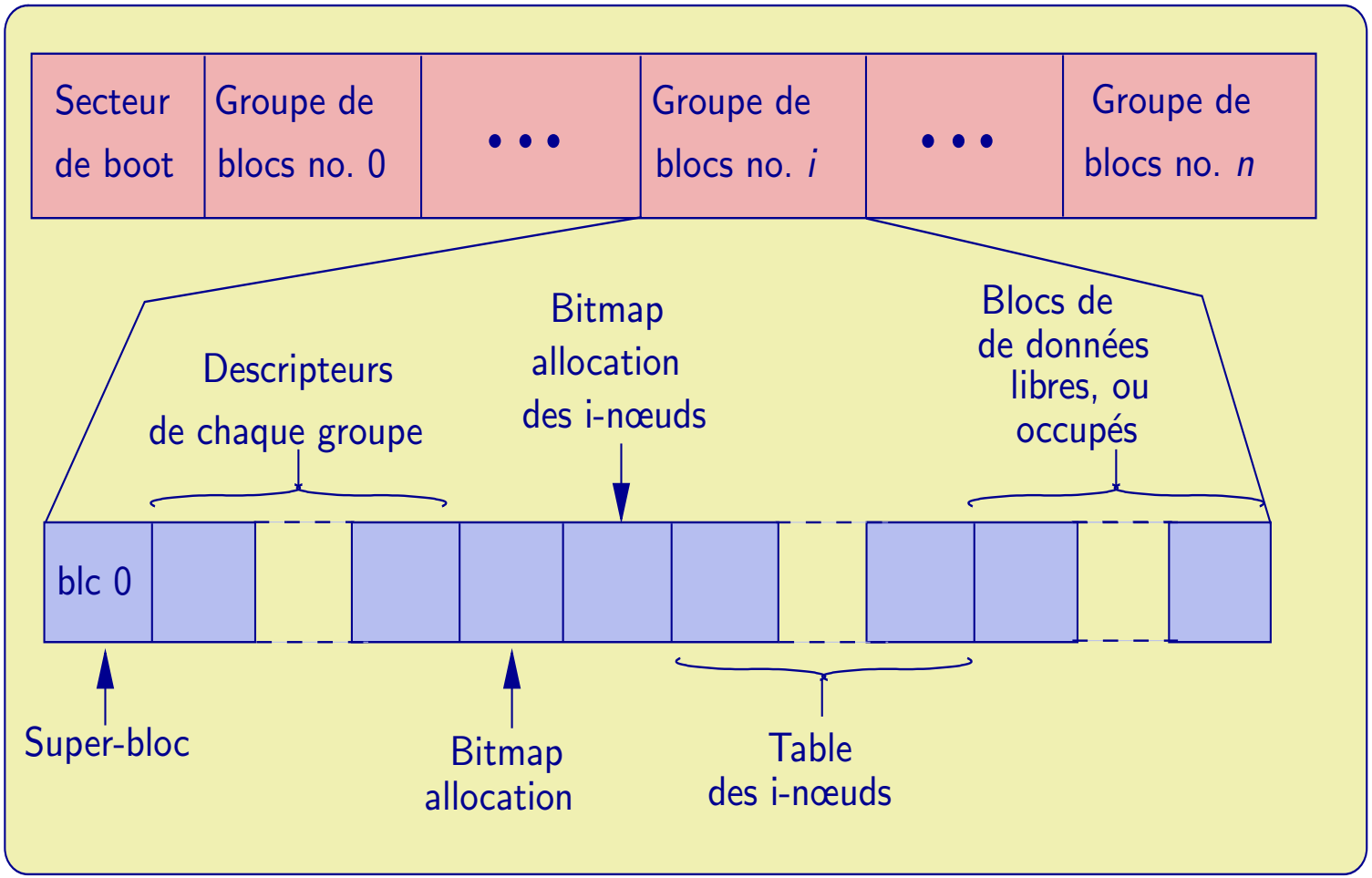
fichier existant

- obtention de l'**i-node** du fichier en question par le système :
  - s'il est dans la table des inodes en mémoire – ok
  - sinon, le système y alloue une entrée et le recopie du disque (`iget()`
- vérification droit d'accès dans le mode demandé
- allocation (si besoin) d'une entrée dans la table des fichiers ouverts du système
- allocation place dans table des descripteurs de fichier du processus
- renvoi de l'indice dans cette table, ou de -1 si échec, etc.

Détails internes du déroulement de `close()`

- si descripteur invalide, renvoi de -1
- si l'ouverture était en écriture, le dernier bloc du tampon-cache est marqué à écrire
- libération entrée dans la table des descripteurs
- décrémentation du compteur de descripteurs, de la table des fichiers ouverts ; s'il arrive à zéro, alors
  - libération de l'entrée de la table des fichiers ouverts
  - décrémentation du compteur d'ouvertures de l'i-node en question, logé en mémoire ; s'il arrive à zéro, alors
    - libération de l'entrée dans la table des i-nodes en mémoire ; si de plus le compteur de liens de l'i-node est à zéro, alors récupération de l'i-nide et des blocs

Structure de l'espace disque



Structure de l'espace disque

- Le disque — découpé en blocs
- Chaque bloc — multiple d'un secteur (1K, 2K ou 4K), valeur fixée à la génération du système de fichiers (FS)
- Le bloc
  - la plus petite unité d'espace disque allouable
  - deux états : libre ou alloué — bit 0 ou 1 dans la bitmap
- Les blocs — groupés en groupes de blocs
- Le premier secteur du disque (Master Boot Record)
  - les renseignements sur le partitionnement du disque
  - éventuellement un programme pour charger et démarrer un SE.
- Le FS ext2 $\rightarrow$ beaucoup de redondances $\Rightarrow$ robustesse

Structure d'un groupe de blocs

- commence par le super-bloc (répété à l'identique dans chaque groupe) — lequel contient les infos globales du FS
- ensuite les descripteurs de tous les groupes
- suivis des bitmaps des blocs alloués, et des i-nœuds alloués
- et de la table des i-nœuds
- et enfin des blocs de données
- Un descripteur de groupe → les offsets en blocs de ces éléments (structure `ext2_group_desc`) et des compteurs (blocs libres, etc.)
- Le super-bloc → compteurs, et des tailles (nbre blocs/groupe, etc.) – structure `ext2_super_block`.

Partitionnement des disques

- Division du disque
- le BIOS des PC demande au plus quatre partitions dites primaires, dont une peut être logique, et alors contenir en fait un grand nombre de partitions secondaires, numérotées à partir de 5
- Linux : **/dev/hda** – disque, **/dev/hda1** – partition
- Linux : **/dev/hda**, **/dev/hdb**, etc. pour disques IDE, **/dev/sda**, **/dev/sdab** pour disques SCSI, etc.
- sur d'autres systèmes Unix, noms plus compliqués

Types de systèmes de fichiers

- UFS – Unix File System – ancient
- VFS – Virtual File System – code générique concernant la gestion de fichiers, qui finit par faire appel au code des « file system drivers », qui implémentent un système de fichiers concret
  - ext2, ext3, reiserfs, afs, nfs, samba (SMB/CIFS), etc.
- **journalisation** : technique de gestion des modifications des données disque pour améliorer la fiabilité :
  - copy-on-write – on copie les i-nodes et blocs de données modifiées dans des nouveaux blocs sur le disque, et on met à jour la liste des i-nodes en conséquence
  - après un crash – on rejoue les dernières transactions, sans parcourir tous les blocs
  - risque : fragmentation des blocs

Types de systèmes de fichiers

- ext2
  - les blocs sont mis en groupes
  - les méta-données (superbloc) écrites de manière asynchrone $\Longrightarrow$ *fsck* au reboot
- ext3
  - journalisation rajoutée – plus besoin de *fsck* au reboot