Environnement Client/Serveur

Cours 1
Rappels sur les systèmes d'exploitations
Généralités Client/Serveur
Communication par mémoire partagée

kn@lri.fr

Contenu du cours

Ce cours est une introduction au modèle Client/Serveur comme structuration d'une application distribuée. Le concentre sur des aspects d'architecture logicielle. Le plan de cours est le suivant :

1. Prélude : Rappels Unix, Généralités Client/Serveur, accès concurrent aux fichiers, fichiers mappés. (1 cours)
2. Systèmes distribués, réseaux (généralité, routage IP, TCP/UDP, couche application HTTP), (3 cours)
3. Java Avancé (Java 8) (1 cours)
4. Concept de RPC (et son utilisation en Java : RMI) (2 cours)
5. Initiation à la programmation JSP (2 cours)
6. Bonus/Révision/Étude de cas (1 cours)

Cours disponible en ligne sur https://www.lri.fr/~kn/teaching_fr.html.
Supports de cours et TP basés sur du matériel de Pierre Vigneras (Miage Apprentissage) et Andreï Paskevich (IUT Orsay)

Modalités de Contrôle des Connaissances (MCC)

2 sessions:

- 1ère session
  - Contrôle continu (1/3) : TP notés (au moins 2, annoncés avec une semaine d'avance)
  - Examen (2/3)
- 2ème session (examen 100%)

Rappel: *La défaillance fait obstacle au calcul de la moyenne et implique l'ajournement. La présence de l'étudiant étant obligatoire en TP [...], plus d'une absence injustifiée dans un enseignement peut entraîner la défaillance de l'étudiant dans l'enseignement concerné*

Esprit des TPs

Vous faire pratiquer du Java à haute dose et de belle manière :

- Rappels de programmation Objet
- Notion de programmation Système et Réseau en Java
- Apprentissage de la bibliothèque standard Java
- Initiation à J2EE
- Apprentissage de *design patterns* courants

Plan

Système d'exploitation

Quelques systèmes:

- Windows XP/NT/2003/7/8, ...
- Linux, FreeBSD, NetBSD, OpenBSD, ...
- MacOS X (basé sur une variante de FreeBSD), ...
- Unix, AIX, Solaris, HP-UX, ...
- Symbian OS (Nokia), iOS, Android, ...

Système d'exploitation

Qu'est-ce qu'un système d'exploitation ?

- c'est un *programme*
- qui *organise* l'accès aux *ressources* de la machine

Quelles sont les ressources d'une machine?

- Processeur (temps d'exécution)
- Mémoire
- Accès aux périphériques de stockage
- Accès aux périphériques d'entrées/sorties
- ...

Système d'exploitation

Haut niveau

Bas niveau

- *Applications*: navigateur Web, éditeur de texte, anti-virus, jeu, compilateur, ...
- ***Système d'exploitation***:
  - Gestion des ressources
  - Interface avec le matériel (pilotes)
- *Matériel*: CPU, mémoire, périphériques, ...

Le système Unix

1965 : MultICS: *Multiplexed Information and Computing Service* (Bell & MIT)

1969 : Unix: 1ère version en assembleur (AT&T)

1972-73 : Unix réécrit en C

1976 : Invention de TCP/IP

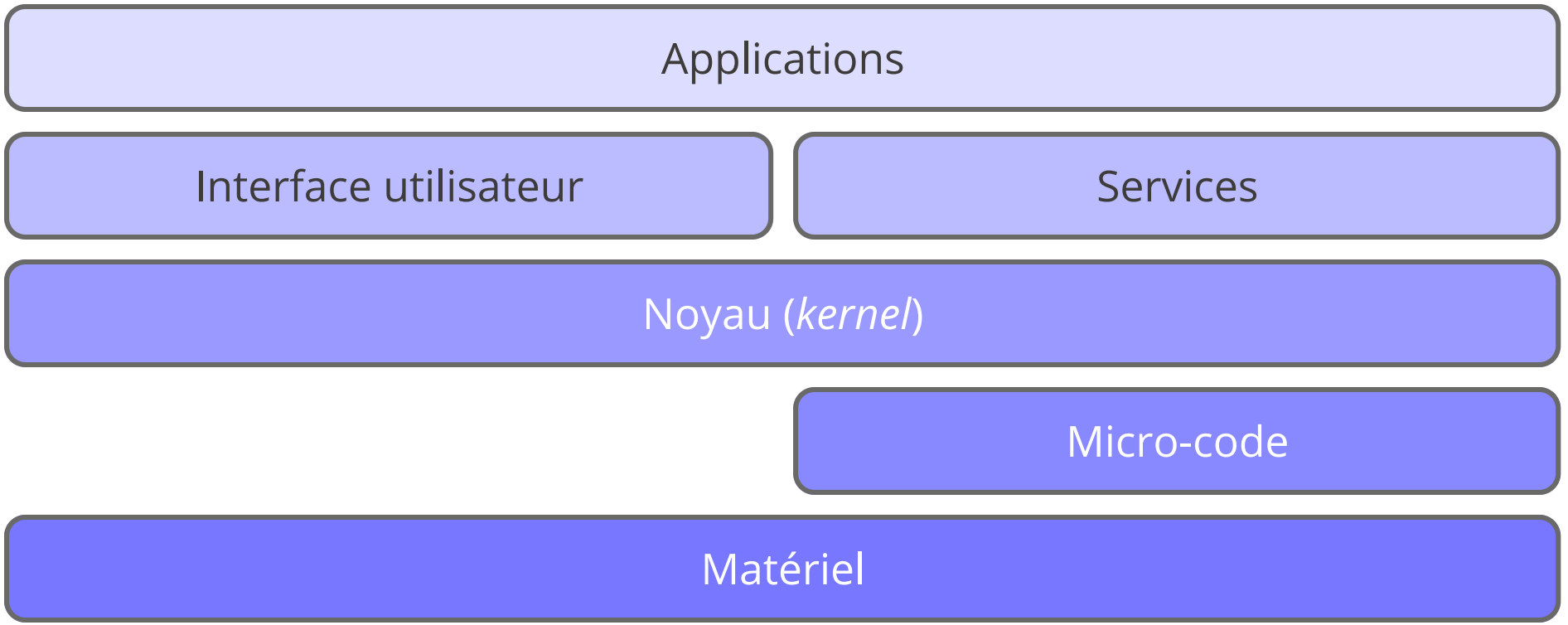
1977 : *Berkeley Software Distribution* (BSD)

1985 : Unix System V

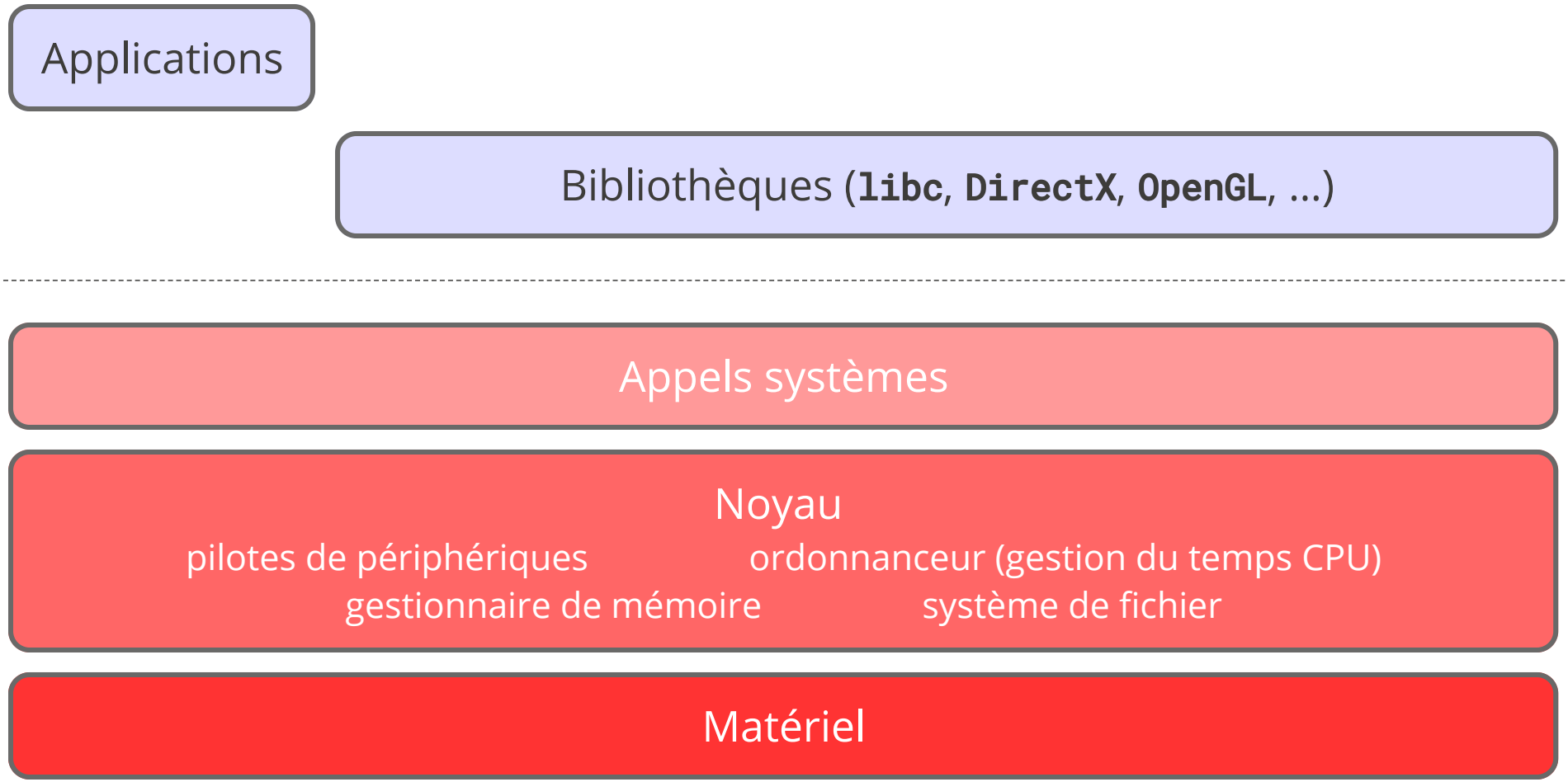
1988 : Minix

1992 : Linux

Unix : architecture



Zoom sur le noyau



Le standard POSIX

Portable Operating System Interface : ensemble de standards spécifiés par l'IEEE qui définissent les API standards pour interagir avec un sytème d'exploitation :

- Systèmes de fichiers (chemins, types, permissions, ...)
- Manipulation des fichiers (primitives de lectures et d'écritures)
- Gestion des processus et des *threads* (création, partage de mémoire, *mutex*, ...)
- Gestion de la mémoire (allocation et desallocation mémoire, mémoire partagée, mémoire paginée, ...)
- Communication entre processus (signaux, *sockets*, ...)

La plupart des systèmes d'exploitation font plus que ce qui est dans le standard

Java étant porté sur beaucoup de systèmes (même non-POSIX), il fournit une couche d'abstraction très semblable à POSIX.

Le concept de *fichier*

Un fichier est une *collection d'informations numériques* réunies sous un même *nom* et enregistrée sur un support de stockage

- Manipulable comme une unité
- Selon les systèmes, le *nom* a plus ou moins d'importance
- possède un type

Le concept de *fichier*

Ne pas confondre:

- type du fichier: il influe sur le comportement du système (fichier « normal », répertoire, lien (raccourcis), fichier système, ...). C'est une méta-donnée conservée par le système de fichier
- type du contenu: le type des *données* contenues dans le fichier:
  - DOS puis Windows: l'extension (les 3 derniers caractères après le « . ») détermine le type de contenu
  - MacOS puis OS X/iOS: les premiers octets du fichier déterminent son type
  - Premiers octets ou extension, selon les interfaces utilisées

Les attributs d'un fichier

Nom **:**

Propriétaire **:** utilisateur qui possède ce fichier

Groupe **:** groupe d'utilisateurs qui possède ce fichier

Emplacement **:** localisation du fichier sur le support physique

Taille **:** en octet (peut être la taille réelle ou la taille occupée sur le support)

Permissions **:** « qui a quel droit » sur le fichier (lecture, écriture, exécution, ...)

Type **:**

Dates **:** dernier accès, dernière modification, création, ...

Organisation logique des fichiers

Usuellement, les fichiers sont regroupés en *répertoires*. Les répertoires sont imbriqués les uns dans les autres de manière a former une *arborescence*.

Sous Unix il y a un répertoire racine, « */* » (*slash*) qui contient toute l'arborescence du système.
Chaque utilisateur possède aussi un répertoire personnel

Noms de fichiers et chemins

Un chemin est une *liste de répertoire* à traverser pour atteindre un fichier ou répertoire donné. Sous Unix, le séparateur de chemin est le « / »
Les chemins absolus commencent par un **/** et dénotent des fichiers à partir de la racine. Exemple:

```
/home/kim/Documents/ECS/cours01.pdf
```

Les chemins relatifs dénotent des fichiers à partir du répertoire courant. Exemple:

```
Documents/ECS/cours01.pdf
```

si on se trouve dans le répertoire **/home/kim**

Les noms spéciaux: « . » dénote le répertoire courant, « .. » le répertoire parent, « ~ » le répertoire de l'utilisateur et « ~**toto** » le répertoire de l'utilisateur **toto**

Droits et propriétés des fichiers

Sous Unix un utilisateur est identifié par son *login* (ou nom d'utilisateur). Chaque utilisateur est dans un *groupe principal*.
Chaque fichier appartient à un utilisateur et à un groupe.
Chaque fichier possède 3 permissions pour son propriétaire, son groupe et tous les autres. Les permissions sont lecture, écriture, exécution (plus d'autres non abordées dans ce cours).

Permission	**fichier**	**répertoire**
lecture (r)	lire le contenu du fichier	lister le contenu du répertoire
écriture (w)	écrire dans le fichier	supprimer/renommer/créer des fichiers dans le répertoire
exécution (x)	exécuter le fichier (si c'est un programme)	rentrer dans le répertoire

```
$ ls -l
drwxr-x---  9 kim prof 4096 Sep  7 21:31 Documents
```

La commande *chmod*

`chmod` *permissions* $chemin_1$ … $chemin_n$

modifie les permissions des fichiers *1* à *n*. La chaîne ***permissions*** est soit une suite de modifications de permissions *symbolique* soit l'ensemble des permissions données de manière *numérique*:

```
chmod 755 fichier.txt
chmod u-w,a+x,g=w fichier.txt
```

Permissions numériques

On groupe les *bits* de permissions par trois puis on convertit en décimal:

Utilisateur			Groupe			Autres		
r	w	x	r	w	x	r	w	x
1	1	0	1	0	0	0	0	0
6			4			0		

Le fichier est lisible et modifiable mais pas exécutable par son propriétaire, lisible pour le groupe. Les autres ne peuvent ni le lire ni le modifier.

Permissions symboliques

```
cible modifieur permission
```

- *cible* : u (utilisateur), g (groupe), o (others), a (all)
- *modifieur* : + (autorise), - (interdit), = (laisse inchangé)
- *permission* : r (lecture), w (écriture), x (exécution)

Exemple:

```
chmod u+rw,u-x,g+r,g-wx,o-rwx fichier.txt
```

Programmes et Processus

Programme : séquences d'instructions effectuant une tâche sur un ordinateur.

Exécutable : fichier binaire contenant des instructions machines interprétables par le microprocesseur.

***Thread* :** plus petite unité de traitement (≡ séquence d'instructions) pouvant être ordonnancée par le système d'exploitation.

Processus : instance d'un programme (≡ « un programme en cours d'exécution »). Un processus est constitué de un ou plusieurs *threads*.

Différence fondamentale : deux threads peuvent partager des variables (écrire dans la même variable « **x** » du programme. Deux processus ne peuvent partager que des fichiers.

Signaux

L'OS peut envoyer des *signaux* à un processus. Sur réception d'un signal, un processus peut interrompre son comportement normal et exécuter son *gestionnaire de signal*. Quelques signaux:

Nom	Code	Description
HUP	1	demande au processus de s'interrompre
INT	2	demande au processus de se terminer
ABRT	2	interrompt le processus et produit un *dump*
KILL	9	interrompt le processus immédiatement
SEGV	11	signale au processus une erreur mémoire
STOP	24	suspend l'exécution du processus
CONT	28	reprend l'exécution d'un processus suspendu

Plan

Client / Serveur ?

Le modèle Client/Serveur est un modèle d'architecture logicielle qui permet d'organiser des applications distribuées. Ce modèle s'applique à un cas bien particulier :

- Une certaine ressource (page Web, carte son, carte graphique, annuaire, base de données, ...) est disponible
- Plusieurs processus veulent avoir accès à la ressource

Le modèle Client/Serveur s'organise en :

- Un serveur : un processus particulier qui gère l'accès à la ressource (autorise l'accès, gère les différentes demandes, applique une politique de sécurité pour l'accès, ...)
- Des clients : des processus qui demandent au serveur l'accès à la ressource

Applications distribuées ?

Attention à ne pas confondre plusieurs concepts bien distincts :

programmation distribuée
: modèle où le système de calcul est composé de nœuds distincts ne partageant pas de mémoire (généralement connectés en réseau).

programmation parallèle
: modèle où certaines sous-tâches du programme principal sont faites en parallèle dans un soucis de gain performance. Par exemple sur des processeurs modernes (superscalaires):

```
x = z + 35;
y = z - 22;
```

Les deux résultats sont calculés en même temps.

programmation concurrente
: modèle de programmation où plusieurs actions peuvent se produire au même moment. Exemple : un système d'exploitation est un programme concurrent, il exécute des tâches (par exemple exécuter des programmes utilisateurs, dessiner l'interface graphique, etc...) en donnant l'illusion qu'elles se déroulent en même temps (même sur un système uniprocesseur).

Et le modèle Client/Serveur dans tout ça ?

Programmation distribuée : le client et le serveur sont souvent (mais pas obligatoirement) sur des machines différentes. Ils doivent donc communiquer l'un avec l'autre.

Question : comment deux programmes peuvent-il communiquer ?

On va tenter de répondre à cette simple questions au cours des 10 séances de cours !

Exemple de situation

On considère une application d'«envoi de messages». On se place dans le cadre simple suivant : les deux utilisateurs sont sur la même machine.

- Un processus serveur « S_{alice} » attend les messages pour l'utilisateur **`alice`** et les affiche
- Un processus client « C_{alice} » envoie un message de l'utilisateur **`alice`**
- Chaque utilisateur (**`alice`**, **`bob`**, ...) de la machine lance un serveur et un client

Concrètement, le processus C_{bob} veut « envoyer » la chaine de caractères **`"Bonjour, Alice"`** au processus S_{alice}. Comment faire ?

Fichiers partagés

On peut supposer que S_{alice} lit dans un fichier (par exemple **`~/messages`**) et que C_{bob} écrit dedans. Quels problèmes cela pose-t-il ?

- Problèmes de permissions (il faut les bons droits sur le fichier, mais difficile à gérer finement avec les droits standards Unix)
- Problèmes d'accès concurrents (C_{bob} et $C_{charlie}$ envoie un message à **`alice`**)
- Problèmes de performances (écrire/lire/ouvrir/fermer des fichiers est coûteux)

Utilisations de signaux

Le client peut envoyer un signal Unix au serveur (par exemple **USR1** ou **USR2**. Problèmes ?

◆ Pas possible d'associer une donnée à un signal (donc on n'est juste capable d'envoyer un petit entier au serveur)

La communication par signaux est donc utilisée uniquement pour envoyer à un processus des messages « convenus » (relis ton fichier de configuration, arrête toi, relance toi, ...). L'envoi de signal est par contre très efficace.

Utilisations de tuyaux (*pipes*)

On peut créer des *pipes* entre deux processus (par exemple c'est ce que fait le *Shell* lorsque l'on écrit :

```
commande1 | commande2
```

Un *pipe* est crée en lecture pour **commande2** (relié à son entrée standard) et en écriture pour **commande1** (relié à sa sortie standard). Problème ?

- Connection point à point, il faut créer autant de tuyaux que de clients
- Unidirectionnel : on écrit d'un côté et on lit de l'autre, il faut donc créer un deuxième tuyaux du serveur vers le client
- Possible uniquement entre processus fils d'un processus donné : deux processus arbitraires ne peuvent pas communiquer via un *pipe*.

La communication par *pipe* est par contre performante

Utilisation des Fichiers partagés, revisitée

Certains outils des systèmes d'exploitation vont nous permettre de palier à deux points négatifs de la communication par fichiers :

1. Les verroux de fichiers (règle les problèmes d'accès concurrents)
2. La mémoire mmapée (règle les problèmes de performance)

Vérroux de fichier

Il est possible de poser un verrou (ou plusieurs) sur un fichier. Un verrou peut être :

- Exclusif : un seul processus peut accéder au fichier
- Partagé : plusieurs processus peuvent accéder au fichier

De plus, on a une granularité fine : on peut vérouiller une portion d'un fichier uniquement (par exemple les octets 20 à 42). On possède trois primitives pour manipuler les verrous :

- Acquérir un verrou (en précisant la portion et le mode) et attendre jusqu'à ce que ce soit possible
- Acquérir un verrou (en précisant la portion et le mode) si possible et renvoyer une erreur si impossible
- Relacher un verrou acquis

En Java (1/2)

La classe `FileLock` représente les verrous, la méthode `.release()` permet de les relacher. Pour acquérir un verrou sur un fichier, on doit d'abord créer un objet de la classe `FileChannel` (par exemple à partir du nom de fichier). Une fois un tel objet créé, on peut utiliser les deux méthodes suivantes :

- `.lock(long position, long size, boolean shared)` : essaye d'acquérir un verrou sur le fichier entre les positions **`position`** (incluse) et **`position + size`** (exclue). Si **`shared`** vaut vrai, alors le verrou est partagé, sinon il est exclusif. Le programme est bloqué jusqu'à ce qu'on puisse obtenir le verrou.
- `.tryLock(long position, long size, boolean shared)` : essaye d'obtenir un verrou (avec les mêmes paramètres que `.lock` mais renvoie immédiatement soit le verrou (s'il a pu être acquis) soit **`null`** si ce n'est pas possible.

En Java (2/2)

Il existe une variante **`.lock()`** (resp. **`.tryLock()`**) sans paramètre qui correspond à **`.lock(0, N, false)`** (resp. **`.tryLock(0, N, false)`**), où N est la taille du fichier).

Fichier MMAPé

Un fichier associé à zone mémoire (*memory-mapped file*) est un fichier auquel le système d'exploitation associe une zone mémoire (un tableau d'octets). On n'accède plus alors dans le fichier par des primitives **`write()`** ou **`read()`** mais directement en écrivant/lisant dans le tableau. Le système d'exploitation s'occupe alors de détecter les mises à jour en mémoire et de les répercuter sur le fichier.

Le principal avantage de cette technique est la très grande efficacité du procédé, à peine un peu plus coûteux que d'écrire dans un tableau en C, le système d'exploitation s'occupe seul d'écrire les données sur le disque au moment opportun.

Exemple en Java (naïf)

Supposons que l'on veuille lire un tableau de 100 entiers stocké dans un fichier **`tab.data`**, incrémenter tous les entiers, et sauver le tableau modifié dans le même fichier

```
DataInputStream din = new DataInputStream(new FileInputStream("tab.data"));

int tab[] = new int[100];

for (int i = 0; i < 100; i ++)
     tab[i] = din.readInt() + 1;

din.close();

DataOutputStream dout = new DataOutputStream(new FileOutputStream("tab.data

for (int i = 0; i < 100; i ++)
    dout.writeInt(tab[i]);

dout.close();
```

Combien de mémoire utilise-t-on ?

Exemple en Java (naïf)

1. En premier lieu, pour des raisons de performances, le système d'exploitation ne lit jamais un fichier octet par octet. Il charge un morceau du fichier dans un *buffer* interne d'un seul coup (1ère copie)

2. Ensuite notre programme lit le fichier et stocke le résultat de la transformation un tableau Java (2ème copie)

3. Enfin, le système d'exploitation n'écrit jamais octet par octet dans un fichier, il stocke les écritures successives dans un *buffer* interne et les écrit en block sur le disque (3ème copie).

On a donc copié 3 fois les données en mémoire. La primitve **`mmap`** permet d'exposer au programmeur les *buffer* systèmes et donc de se passer de une copie (si le fichier de sortie est différent du fichier d'entrée) ou de deux copies (si le fichier d'entrée est le même que le fichier de sortie)

Exemple en Java (efficace)

```
FileChannel in = FileChannel.open(Paths.get("tab.data"),
                                  StandardOpenOption.READ,
                                  StandardOpenOption.WRITE);

MappedByteBuffer buff = in.map(MapMode.READ_WRITE,0, 400);

in.close();

IntBuffer ibuff = buff.asIntBuffer();

for (int i = 0; i < 100; i ++)
    ibuff.put(i, ibuff.get(i) + 1);
```

Exemple en Java (efficace)

`FileChannel.open`
: permet d'ouvrir un fichier. Le fichier doit être représenté par un objet **`Path`**, la classe utilitaire **`Paths`** permet de créer un tel objet à partir du nom de fichier. On doit donner des options lors de l'ouverture du fichier : **`StandardOption.READ`** (fichier ouvert en lecture), **`StandardOption.WRITE`** (fichier ouvert en ecriture), et pleins d'autres (voir la Javadoc).

`FileChannel.map(mode, position, size)`
: Permet d'obtenir un tableau d'octets représentant le fichier. Le mode peut être **`MapMode.READ`** ou **`MapMode.READ_WRITE`**. On peut restreindre le tableau demandé à un certain fragment du fichier

`.asIntBuffer()`
: permet de renvoyer une vue « tableau d'entier » du tableau d'octet (les octets sont lus 4 par 4 et combinés pour en faire des entiers).

`.get/.set`
: permet de lire et d'écrire dans le tableau

Exemple en Java (efficace)

Remarques :

◆ Si on ne demande pas les bons modes, alors des exceptions sont levées lors des opérations invalides (par exemple tenter d'écrire dans un fichier ouvert en lecture)
◆ Un tableau d'octet mappé existe même une fois que le fichier est fermé. Il ne disparaît que lors que le *garbage collector* de Java le supprime car il n'est plus utilisé.
◆ Il existe plein d'options et méthodes, en particulier pour forcer le système à écrire les modifications du tableau sur le disque

Exemple initial revisité

- S_{alice} « mape » le fichier **~/message**, sans vérou et affiche son contenu s'il change
- C_{bob} prend un verrou exclusif sur **~alice/message**, écrit son message et rend le verrou
- $C_{charlie}$ prend un verrou exclusif sur **~alice/message**, écrit son message et rend le verrou

Charlie ne pourra pas écrire tant que Bob tiens le verrou (problème de concurrence réglé)
Le tout sera très efficace car les fichiers sont mapés
Deux problèmes subsistent :

1. Le fichier **message** doit être accessible en écriture à tous si les utilisateurs ne sont pas dans les mêmes groupes
2. Que se passe-t-il si $C_{charlie}$ obtient le verrou avant que S_{alice} n'affiche le message ? le message de Bob est écrasé !

Plan

`final`

Que signifie le mot clé **`final`** sur ?

une classe
: On ne peut pas dériver la classe (ex : **`String`**)

une méthode
: On ne peut pas redéfinir (*override*) la méthode. Que signifie redéfinir ?

- écrire une méthode avec exactement le même prototype dans une classe dérivée.

un attribut
: L'attribut peut être initialisé au plus tard dans le constructeur :

- si son type est primitif (**`boolean`**, **`char`**, **`int`**, ...) alors l'attribut est une constante
- si l'attribut est un objet, alors c'est une référence constante vers cet objet! (l'état interne de cet objet peut toujours être modifié, cf. les tableaux)

une variable locale ou un paramètre
: on ne peut pas modifier la valeur de la variable après initialisation

Utilité

◆ Contrôle du code utilisateur. Ex: beaucoup de composants de Java (y compris de sécurité) utilisent la classe **`String`**. Si on pouvait dériver la classe et changer ses méthodes (par exemple **`equals`**) le code utilisant **`String`** n'aurait plus aucune garantie (pour les classes et les méthodes **`final`**)

◆ Sûreté et documentation : un attribut ou une variable déclarés **`final`** ne peuvent être modifiés. Le programmeur déclare que ce sont des constantes

◆ Efficacité : un attribut **`final`** étant constant, l'optimiseur de la JVM connait sa valeur et faire des optimisations. Une classe ou une méthode **`final`** ne pouvant pas être dérivée/redéfinie, l'optimiseur peut déterminer son code de manière certaine

finally

(malgré la similitude de nom, ça n'a rien à voir avec **`final`**.)

Opération courante : on souhaite écrire dans un fichier, puis le refermer. Cependant, il faut aussi traiter le cas des exceptions correctement :

```
PrintStream out = …;
try {

   out.println(…);
   out.close();

} catch (IOException e) { out.close(); }
```

Le code de **`out.close()`** est dupliqué. Si le code était plus important, ce ne serait pas maintenable.

finally

On peut utiliser la clause **finally** dans un **try/catch** pour écrire du code qui sera exécuté obligatoirement :

- à la fin du bloc, si aucune exception n'est levée
- après le code du **catch** si une exception est rattrapée
- avant de quitter le bloc (ou la méthode) si l'exception n'est pas rattrapée

```
PrintStream out = …;
try {

   out.println(…);

} catch (IOException e) {
} finally {  out.close(); }
```

On utilise cette construction lorsque l'on souhaite libérer une ressource (fichier, connexion réseau ou à une base de donnée, ...) avant de poursuivre le programme, quelle que soit la manière dont il se poursuit (normalement, exception rattrapée, exception propagée).

static

Que signifie le mot clé **static** sur ?

une méthode

la méthode n'est pas appliquée à un objet :

- Elle n'a pas accès aux attributs non statiques de l'objet

un attribut

L'attribut est une propriété de la classe et non pas d'un objet particulier. Il ne peut être initialisé que :

- au moment de sa déclaration
- dans un bloc statique

```
class A {
static HashMap<Integer, String> map;
  static {
    map = new HashMap<>();
    map.put(new Integer(1), "A");
    map.put(new Integer(2), "B");
```

Utilité

- Partager un même objet (par exemple un compteur) entre tous les objets d'une même classe
- Similaire à une variable globale (si l'attribut est public static)
- Similaire à une constante globale (si l'attribut est public static final

Attention les champs statiques sont initialisés au chargement de la classe par la JVM, donc bien avant que le moindre code utilisateur ait été exécuté.

Quizz sur les Interfaces

Soit une interface :

```
interface I extends J {
            public R1 meth(T1 t1, …, Tn tn) throws E1, …, Ek
      }
```

on considère une classe **`C`** qui implémente **`I`**