Licence 2 - Informatique
Info4B - Principes des systèmes d'exploitation
Travaux Pratiques

Éric Leclercq

Révision : Mars 2020

Résumé

Ce document contient l'ensemble des exercices de TP du module de Principes des Systèmes d'Exploitation (Info4B). Tous les exercices sont à réaliser et ils sont regroupés thématiquement en TP s'étalant le plus souvent sur plusieurs séances de 2h. Des encadrés vous donnent des éléments complémentaires pour vous aider à réaliser certaines questions spécifiques. Les exercices sur la concurrence réalisés en TD doivent faire l'objet d'une réalisation et de tests (Ferry, Parking, Tramway, etc.). Les questions notées avec (*) sont des questions qui demandent un approfondissement du cours.
Chaque TP doit faire l'objet d'un compte rendu personnel qui pourra être demandé pour compenser légèrement le projet s'il n'a pas le niveau requis. Le fascicule de TP en ligne sera mis à jour assez régulièrement ainsi que le GitHub qui contient des squelettes de programme (https://github.com/EricLeclercq).

Table des matières

1 Création de processus : étude des threads de Java

Ce TP à une durée de deux séances.

Java fournit en standard un certain nombre de classes pour gérer, les fichiers, les communications réseau, les processus, les fenêtres graphiques, etc. Ces diverses fonctionnalités sont bien souvent réalisées au moyen de la notion de thread (présentée dans le chapitre II du cours). Un thread est un processus léger qui partage avec son parent l'ensemble de ses données. L'utilisation de plusieurs threads dans un programme permet :

- **d'éviter les goulets d'étranglement :** avec un seul thread, le programme doit s'arrêter complètement pour attendre des processus lents comme la gestion des accès aux disques durs ou les communications avec d'autres machines. Le programme est inactif jusqu'à ce que ses demandes soit satisfaites. Avec plusieurs threads, l'application peut poursuivre l'exécution des autres threads pendant que le thread qui attend le résultat d'un processus lent est bloqué ;
- **d'organiser le comportement du programme :** avec plusieurs threads on peut décomposer le programme en plusieurs sections indépendantes et assigner différentes priorités à chaque thread afin d'attribuer davantage de temps CPU aux tâches critiques ;
- **d'utiliser plusieurs processeurs ou plusieurs cœurs d'un même processeur.**

Dans cette série d'exercice, on se propose d'étudier la création de threads, l'affectation de priorité et les problèmes d'échange et de partage de données entre plusieurs threads (communication, concurrence et synchronisation).

Interactions entre Java et le système d'exploitation

Java est une technologie définie par la société SUN Microsystems à la fin de l'année 1995 (aujourd'hui racheté par la société ORACLE, éditeur de Systèmes de Gestion de Bases de Données - SGBD). Java est bien plus plus qu'un langage de programmation orienté objet, on parle couramment de la plateforme Java. La plateforme Java s'articule autour de trois composants essentiels :

1. les spécifications du langage de programmation Java ;
2. un ensemble d'interfaces/bibliothèques de programmation d'applications (API) ;
3. une spécification de machine virtuelle.

Le langage Java fournit, en plus des aspects traditionnels des langages de programmation orientés objets, un support de haut niveau pour la programmation réseau et la communication entre objets distribués. C'est également un langage multi-threads au sens ou un programme Java peut avoir plusieurs flots (ou fils) d'exécution.

La plateforme Java contient une API de base et une extension standard. L'API de base fournit un support simple pour le graphisme, les entrées sorties, les fonctions courantes et le réseau (`java.lang`, `java.awt`, `java.io`, `java.net`, etc.). Les extensions incluent des services spécifiques pour le support d'applications d'entreprise JEE (sécurité, interfaces bases de données, environnement d'exécution serveur, etc.). Comme le langage évolue, de nombreux packages faisant partie des extensions sont inclus dans l'API de base.

La machine virtuelle (JVM) est une spécification de machine abstraite constituée d'un chargeur de classe et d'un interpréteur Java capable d'exécuter le code objet (*byte-code*) produit par le compilateur `javac`. Le code objet est indépendant de la plateforme matérielle, il est stocké dans des fichiers `.class`. Le chargeur de classes utilise les fichiers

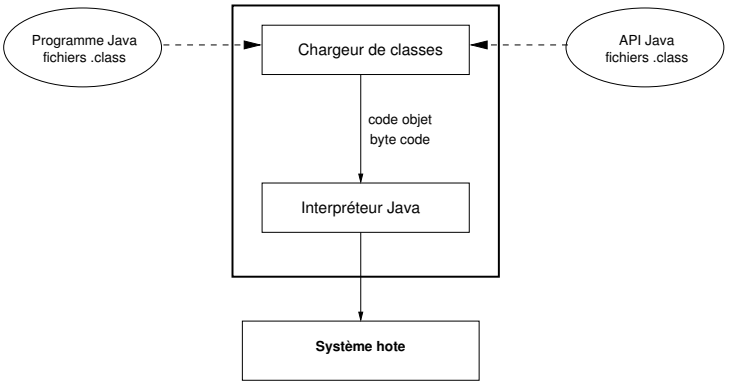

FIGURE 1 – Principe de fonctionnement de la JVM

`.class` et les API pour permettre l'exécution des programmes par l'interpréteur. L'interpréteur peut être un logiciel qui interprète instruction après instruction les commandes du code objet ou un compilateur à la volée qui traduit en code machine le code objet (JIT *Just-In-Time*). Dans certains cas l'interpréteur Java peut être implanté dans une puce spécifique. Afin de pouvoir exécuter un programme Java (`.class`) sur une architecture il faut disposer d'une machine virtuelle spécifique à cette architecture. Le problème de portabilité des programme est donc simplifié puisqu'il suffit de porter la machine virtuelle (quelques Mo de code). La machine virtuelle réalise donc une abstraction du système d'exploitation. Ainsi, on peut, par généralisation, considérer la JVM comme un mini-système d'exploitation. Le processeur sur lequel s'appuie la JVM est un processeur abstrait capable d'exécuter le *byte-code*.

Un instance de la JVM est créée chaque fois qu'une application Java est lancée. L'instance de la JVM commence son exécution lorsque la méthode `main()` est invoquée. C'est également vrai pour les applets qui *a priori* n'ont pas de méthode main puisque celle ci est gérée par le navigateur Web. Si on lance n programmes Java sur une machine, il y aura n machines virtuelles en exécution. C'est même encore plus complexe car la JVM dépend aussi du processeur. Le système d'exploitation GNU/Linux existe pour de nombreux processeurs et par conséquent les différentes JVM sont optimisées pour traduire le byte-code vers le code natif spécifique au processeur.

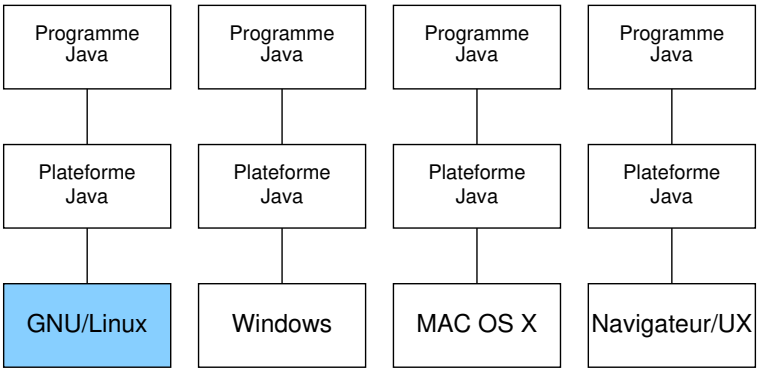

FIGURE 2 – Portabilité de la JVM

L'environnement Java que vous allez utiliser en TP est composé du compilateur Java (`javac`) et de la machine virtuelle (`java`). Le principe de développement du programme est décrit à la figure 1. Il est fortement déconseillé d'utiliser NetBeans car vous aurez du

mal à interpréter et à visualiser les résultats produits lors de l'exécution des programme.

Après chaque exécution de programme, vous devez être en mesure d'interpréter tous les résultats affichés par le programme que vous avez réalisé.

Exercice 1. Création de threads

Au démarrage d'un programme Java, un seul thread est crée. Pour en créer un nouveau, il faut créer un nouvel objet de la classe `Thread` puis le démarrer c'est-à-dire lui demander d'exécuter une portion de code (méthode `run()`). Il existe deux stratégies. Soit le code à exécuter est spécifié dans une classe qui hérite de la classe `Thread`, soit il est spécifié dans une classe qui implémente l'interface `Runnable`. Dans la seconde stratégie, un objet sera passé en paramètre lors de la création du nouvel objet de la classe `Thread`. La première solution a l'avantage de ne manipuler qu'un seul objet mais elle a l'inconvénient d'interdire tout nouvel héritage.

Pour créer le nouveau thread il faut utiliser l'un des constructeurs de la classe `Thread`. Les plus couramment utilisés sont :

- `public Thread();`
- `public Thread(String);`
- `public Thread(Runnable, String);`
- `public Thread(Runnable);`
- `public Thread(ThreadGroup, String);`
- `public Thread(ThreadGroup, Runnable);`
- `public Thread(ThreadGroup, Runnable, String);`

Le paramètre `ThreadGroup` permet de regrouper un ensemble de threads et de leur donner des propriétés communes ou de leur appliquer, à tous, une même méthode. Une fois l'objet thread créé au moyen d'un des constructeurs, il faut l'activer (le démarrer) par un appel à la méthode `start()` qui appellera la méthode `public void run()` de l'objet contenant le code à exécuter. Si la méthode `run()` est appelée directement, le code de cette méthode est exécuté dans le thread courant.

```
class monThread extends Thread{
   public void run() {
    for (int i=1 ; i<5 ; i++)
        System.out.println("je␣suis␣le␣thread␣"+getName()+"i="+i);
   }
}

public class exo1{
   public static void main(String args[]) {
    Thread th1 = new monThread();
    Thread th2 = new monThread();
    th1.start();
    th2.start();
  System.out.println("Je␣suis␣le␣thread␣principal");
   }
}
```

Listing 1 – Création de deux threads

1. En vous inspirant du code donné en cours, et du code du listing 1, écrire deux programmes pour créer des threads au moyen des deux méthodes évoquées c'est-à-dire avec `extends Thread` ou bien `implements Runnable`.

2. Taper le programme du listing 1, l'exécuter (enventuellement plusieurs fois), que constatez-vous dans les affichages ? interpréter le résultat.
3. Pourquoi ce programme ne met réellement pas en évidence le fonctionnement multi-tâches du système d'exploitation ? Modifier le programme afin de faire apparaître "l'effet multi-tâches" et la préemption.
4. Modifier les programmes déjà réalisés pour créer 10 threads dans une boucle (sauver les références aux threads dans un tableau).
5. Quel peut être l'inconvénient de la méthode de création et de démarrage des threads (surtout dans le cas d'une utilisation de plusieurs threads en boucle) ?
6. Affecter un nom à chacun des threads crées via la méthode `setName()`, afficher la référence d'un thread, afficher son nom avec la méthode `getName()` ainsi que le nombre de threads avec la méthode de classe `activeCount()`. Lorsqu'un thread est déclaré via l'interface `Runnable`, certaines méthodes de la classe `Thread` ne sont plus accessibles directement (`getName()` par exemple). La gestion des noms associés aux threads est illustrée dans le listing 3 page 8.

Exercice 2. Threads et priorités

Il est possible d'associer une priorité à chaque thread créé, au moyen de méthodes définies dans la classe `java.lang.Thread`.

Au lancement d'un programme un seul thread s'exécute, c'est le thread initial. On peut le contrôler via la méthode de classe `public static Thread currentThread();` (utile par exemple pour utiliser la méthode `getName()` avec des threads implémentant l'interface `Runnable`. Il est ensuite possible de modifier les attributs du thread en le manipulant via les méthodes d'instance de l'objet retourné.

Plus généralement, un objet de la classe `Thread` (ou plutôt une référence sur un objet) permet de contrôler le comportement du thread qui peut être actif, suspendu ou arrêté.

```
class exo2{
 public static void main(String[] args) throws Exception{
  Thread threadInitial = Thread.currentThread();
  // Donner un nom au thread
  threadInitial.setName("Mon thread");
  // Afficher le nom du thread
  System.out.println(threadInitial);
  // Faire dormir le thread 2 sec
  Thread.sleep(2000);
  System.out.println("fin");
 }
}
```

Listing 2 – Modification des propriétés

Il est possible de changer la priorité d'un thread afin qu'il ait une priorité particulière pour accéder au processeur. Par convention, le thread de plus forte priorité accède plus souvent au processeur. Par défaut, un thread a la priorité de son parent.

Pour changer la priorité d'un thread on utilise la méthode `public void setPriority(int)` et pour visualiser la priorité on utilise `public int getPriority()`. Il existe des constantes prédéfinies pour les valeurs des priorités (constantes membres de la classe `Thread`) :

— `public final static int MAX_PRIORITY;`

— `public final static int MIN_PRIORITY;`
— `public final static int NORM_PRIORITY;`

Il n'est pas possible de sortir de ces bornes sous peine de recevoir une exception et il est recommandé d'utiliser les constantes (comme par exemple en affectant la valeur `Thread.MIN_PRIORITY` avec la méthode `setPriority()`) au lieu d'un entier.

Modifier le programme de l'exercice 1 afin de donner différentes priorités aux deux threads. Notez vos observations pour chaque type de priorité testé. Pourquoi le comportement souhaité n'est pas celui observé ?

ENCADRÉ 1 – Commandes systèmes pour observer le comportement des processus

À partir d'un shell d'une fenêtre terminal, vous pouvez utiliser les commandes `top`, `htop` pour observer en continu les processus qui s'éxécutent et la charge globale de la machine (mémoire, CPU, buffer, etc.). La commande `qps` est une version graphique similaire à `htop`.
La commands `ps` et ses options permettent d'avoir le détail sur un processus. `ps -ef` liste tous les processus, `ps -efL` montre aussi les threads.
La commande `pstree` permet de visualiser l'arborescence des processus et `pstree -ap` affiche en plus les identifiants de processus et les threads.
Java propose la commande `jps` pour visualiser les threads des machines virtuelles (utiliser de préférence `jps -l`). La commande `jinfo` est également disponible pour avoir la configuration d'une JVM, `jstat` et `jmap` sont aussi utiles. L'annexe 2 présente des cas d'utilisation plus concrets de ces différentes commandes.

Avant de traiter les 3 exercices suivants lire la partie s'y rapportant en fin de section 1 et le début de la section 2 sur l'utilisation de `synchronized`

Exercice 3. Illustration, une course de thread

Réaliser un programme qui simule une course de threads. Chaque thread démarre et compte jusqu'à 1000. Dans la boucle simulant la course du thread, implanter une attente aléatoire avec la méthode `sleep()` et la méthode `Math.random()`. Le premier thread arrivé à 1000 est déclaré gagnant, néanmoins les autres threads continuent la course.

1. Programmer le course en veillant à ce que votre programme soit en mesure d'établi et d'afficher un classement lorsque tous les threads sont arrivés.
2. Comment implanter un arbitre qui observe la course et établi le classement ?

Exercice 4. Concurrence

Reprendre l'exemple du cours sur la banque et les distributeurs de billets, afin de mettre en évidence le problème de concurrence.

1. Créer un programme qui lance deux threads qui vont retirer 1000 fois 10 pour l'un et 1000 fois 50 pour l'autre, sur un compte qui comporte 800 000. Chaque thread consulte le compte, met le solde dans une variable temporaire, effectue le retrait et communique le solde à la banque. Les deux threads doivent être issus de la même classe, les objets de la classe compte sont instanciés tout comme les threads par le programme principal. Quel doit être le solde du compte ? Qu'observez vous ? Ce résultat est il toujours le même ?

ENCADRÉ 2 – Threads et attente

Il existe plusieurs méthodes pour attendre ou suspendre des threads, nous expérimentons tout d'abord les méthodes `sleep` et `join`. La méthode `sleep(n)` est une méthode de classe définie sur la classe `Thread`. Elle a pour effet de suspendre pendant `n` millisecondes le thread qui l'exécute. Le temps d'attente n'est pas strictement garanti. L'utilisation de la méthode `sleep` doit se faire dans un bloc qui capte les exceptions du type `InterruptedException` de la manière suivante :

```
try{
 Thread.sleep(100);
}catch(InterruptedException e){e.printStackTrace();}
```

La méthode `join` permet d'attendre la fin de l'exécution d'un thread pour continuer les traitements. C'est une méthode bloquante c'est-à-dire que celui qui l'appelle est bloqué en attente de terminaison du thread sur lequel il a appelé la méthode. Par exemple le programme principal peut attendre la terminaison d'un thread (le thread `t1`) de la manière suivante :

```
try{
 t1.join();
}catch(InterruptedException e){e.printStackTrace();}
```

La méthode `isAlive()` appliquée sur `t1` permet de connaître son état avant et après l'appel à `join()`.

2. Modifier le programme afin de mettre en évidence, de manière plus évidente le problème de concurrence (par exemple en introduisant une durée d'attente aléatoire au moment du retrait).
3. Identifier la section critique, puis en utilisant le mécanisme de moniteur (construction `synchronized`), modifier votre programme, afin qu'à un instant donné, un seul thread puisse accéder au contenu du compte. Vérifier que son exécution est correcte.
4. Faire de même en utilisant la notion de verrou (objet `Lock`), voir encadré 3 : qui dispose du verrou ? qui l'utilise ? Critiquer l'utilisation du verrou.
5. Modifier le code pour avoir une méthode `retrait` sur la classe compte, contrôlée par un moniteur.

Exercice 5. Un exemple plus réel de sollicitation du CPU

Vous avez à votre disposition dans les salles de TP, des machines qui possèdent un processeur multi-cœurs (4 par exemple). On veut créer un programme qui permet de calculer les nombres premiers compris entre 1 et 5 000 000 et accélérer le traitement en utilisant tous les cœurs. Pour exploiter plusieurs cœurs du processeur vous utiliserez plusieurs threads. Pour toutes les questions, observer l'exécution de votre programme avec l'utilitaire `htop`.

1. Réaliser une version séquentielle du programme (mono-thread) en utilisant une méthode `boolean isPrime(int nb)`. Mesurer le temps d ?exécution en utilisant la commande `time` d'UNIX.

2. Réaliser une première version parallèle du programme en attribuant l'exploration des intervalles 1 - 2 500 000 à un thread et 2 500 001 - 5 000 000 à un autre thread. Mesurez le temps d'exécution. Quel est l'inconvénient de ce programme ?
3. Réaliser une deuxième version parallèle comportant un index qui indique la progression du calcul. Chaque thread vient consulter l'index pour connaître le nombre qu'il doit tester. L'algorithme en pseudo-code est le suivant :

```
tant que nb < borne sup
 nb=lire la valeur du nombre à tester dans le distributeur (index)
 incrémenter la valeur de l'index du distributeur
 tester si nb est premier
fin pour
```

Utilisez tous les cœurs, mesurez le temps d'exécution de cette version, calculer le facteur d'accélération (*speed-up*). Quel est le *speed-up* théorique maximum ?

Exercice 6. Synthèse (travail personnel)

1. Reprendre l'exercice 6 des TD, sur la fonction `fork()`. Donner une version de chaque programme pour les questions 1 et 4 en utilisant des threads ; au moyen des commandes Unix `pstree -ap` et `htop` observer la création des threads. La JVM gère-t-elle la filiation des threads ?
2. Étudier attentivement le paragraphe sur le complément d'indications pour les exercices 3,4,5 donné à la page suivante. Tester les méthodes `stop()` `suspend()` et `destroy()` en Java (>= 1.5) ? Interprétez et commenter la documentation officielle.
3. (*) Programmez l'algorithme de Petterson vu en cours, pour assurer l'exclusion mutuelle sans utiliser les primitives de synchronisation classiques (verrou, sémaphores, moniteurs), l'appliquer sur l'exercice de la banque.

```
class RunnableJob implements Runnable{
  public void run(){
    Thread thread = Thread.currentThread();
    System.out.println("Je␣suis␣exécuté␣par␣:␣" + thread.getName()
       + "␣dont␣l'id␣est␣" + thread.getId() + "");
  }
}
public class TName{
  public static void main(String[] args) throws
     InterruptedException {
    // on cree une seule tâche runnable
    RunnableJob rj = new RunnableJob();
    // on l'injecte dans un thread
    Thread thread1 = new Thread(rj);
    thread1.start();
    // on crée un autre thread avec un nom
    Thread thread2 = new Thread(rj);
    thread2.setName("thread2");
    thread2.start();
    // puis dans un autre avec une variante du constructeur
    Thread thread3 = new Thread(rj, "thread3");
    thread3.start();
```

```
    // le main affiche le nom du thread qui l'execute
22  Thread currentThread = Thread.currentThread();
    System.out.println("Main thread: " + currentThread.getName() +
       "(" + currentThread.getId() + ")");
24 }
}
```

Listing 3 – Gestion du nom des threads

Compléments et indications pour la réalisation des exercices 3,4,5

La JVM exécute les threads jusqu'à ce que :
- la méthode `exit()` soit appelée ;
- tous les threads qui n'ont pas été marqués *daemon* soient terminés.

Un thread se termine lorsque la méthode `run()` se termine ou lors de l'appel à la méthode `stop()` sur une référence au thread. Il est possible de suspendre momentanément l'exécution d'un thread au moyen de la méthode `suspend()` et de reprendre l'exécution au moyen de la méthode `resume()`. La méthode `destroy()` permet de détruire un thread sans aucune possibilité de réaction et en ne libérant aucun moniteur. La méthode `yield()` permet de rendre la main. Toutes les opérations ne sont possibles que si le thread appelant les méthodes a le droit d'accéder au thread dont on veut modifier l'état. Ceci n'est vrai que si les threads appartiennent au même ThreadGroup.

Chaque thread s'exécutant dans le même espace d'adressage, il est nécessaire d'avoir un mécanisme d'exclusion mutuelle entre threads lors des accès à la mémoire. Java implante le mécanisme de moniteur qui assure que quand un thread rentre dans une portion de code gérée par un moniteur aucun autre thread ne peut y pénétrer.

Java garantit l'atomicité de l'accès et de l'affectation des types primitifs, sauf les long et les double. Ainsi, si deux threads qui modifient de façon concurrente une variable de type double, ils peuvent produire des résultats incohérents. Bien entendu cette règle est valable pour des objets.

Pour cela Java propose une construction syntaxique avec le mot clef `synchronized` qui s'applique à une portion de code relativement à un objet particulier. Pendant l'exécution d'une portion de code synchronisée par un thread A , tout autre thread essayant d'exécuter une portion de code synchronisée sur le même objet est suspendu. Une fois que l'exécution de la portion de code synchronisée est terminée par le thread A , un thread en attente et un seul est activé pour exécuter sa portion de code synchronisée. Ce mécanisme est un mécanisme de moniteur. Il peut y avoir un moniteur associé à chaque objet. Deux constructions sont disponibles :
- `synchronized` est utilisé comme modifieur d'une méthode. Le moniteur est alors associé à l'objet courant et s'applique au code de la méthode ;
- `la construction synchronized(o){...}`. Le moniteur est associé à l'objet o et s'applique au bloc associé à la construction.

```
1 public interface Lock {
  void lock( );
3 void lockInterruptibly( ) throws InterruptedException;
  boolean tryLock( );
5 boolean tryLock(long time, TimeUnit unit) throws
     InterruptedException;
  void unlock( );
```

```
7   Condition newCondition( );
}
```

Listing 4 – Interface Lock

ENCADRÉ 3 – Les verrous (à partir de Java 1.5)

Les outils de synchronisation disponibles à partir de J2SE 5.0 implementent une interface commune : `Lock` dont le détail est donné à l'adresse http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Lock.html et pour une description plus générale du package http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html.

2 Gestion de la concurrence et synchronisation de processus

Nous avons vu que les threads interagissent principalement dans deux situations :

1. lorsqu'ils entrent en concurrence pour l'accès à des ressources (objets communs ou partagés) ;
2. quand ils coopèrent, se coordonnent ou communiquent (via des champs publics, des objets partagés, des fichiers, des messages, des appels de méthodes, etc.).

Ainsi, on retrouve deux niveaux de synchronisation correspondants :

— au problème de concurrence d'accès aux ressources partagées avec la mise en place de l'exclusion mutuelle pour l'accès à des données (objets) ou à l'exécution du code (méthodes) ;

— au problème de la synchronisation des threads sur des événements (attente du résultat d'un thread pour effectuer ou continuer un calcul par exemple).

Les deux niveaux de synchronisation sont résolus dans la langage Java via les moniteurs (moniteurs de Hoare dégénérés). Cependant, depuis Java 1.5 et les versions suivantes, les primitives de synchronisation sont étendues et on retrouve, comme dans de nombreux autres langages, des primitives de bas niveau tels que les verrous (package `java.util.concurrent.locks`).

Pour traiter les problèmes d'exclusion mutuelle, Java propose la définition de sections critiques exprimées au moyen du modificateur `synchronized`. Tout objet Java est créé avec un verrou propre et interne (c'est-à-dire non accessible directement), permettant de réaliser l'exclusion mutuelle. Ainsi, pour assurer qu'un seul thread accède à un objet `o` d'une classe `C` quelconque, on englobe les actions portant sur l'objet dans une section critique. La syntaxe générale d'une section critique est la suivante :

```
synchronized (o) {
    < section critique >
}
```

Une méthode peut aussi être qualifiée de `synchronized`. Il y a, dans ce cas, une exclusion d'accès à l'objet sur lequel on applique la méthode. Attention l'exclusion ne s'étend pas à la méthode elle-même, ce qui signifie qu'elle peut être exécutée concurremment sur des objets différents.

```
synchronized TYPE methode(...) { ... }
```

Ceci est équivalent à la forme précédente :

```
TYPE methode(...) {
     synchronized (this) { ... }
}
```

De plus, chaque classe possède, aussi un verrou spécifique interne, qui s'applique aux méthodes de classe (méthodes déclarées statiques) :

```
class C {
   static synchronized TYPE methode1() { ... }
   static synchronized TYPE methode2() { ... }
}
```

L'utilisation du mot clé `synchronized` assure dans ce cadre l'exclusion mutuelle pour toutes les méthodes statiques synchronisées de la classe `C`.

Les verrous internes sont qualifiés de **récursifs** ou **réentrants**. En effet, si un thread possède un verrou, une deuxième demande provenant de ce même thread peut être satisfaite sans provoquer d'auto-interblocage. Ainsi, le code suivant s'exécute sans problème.

```
synchronized(o) {
 ...
 synchronized(o){
  ... }
 ... }
```

Lors de l'utilisation de plusieurs verrous, le **risque d'interblocage** existe, plus précisément ce risque est présent dès qu'un thread peut utiliser plusieurs verrous. L'extrait de code suivant, exécuté dans 2 threads distincts ne garantit pas l'absence d'interblocage.

```
synchronized (o1) { synchronized (o2) { ... } }
..
synchronized (o2) { synchronized (o1) { ... } }
```

Comme vu en cours, il existe plusieurs stratégies pour éviter l'interblocage. Une solution simple pour garantir l'absence d'interblocage provoqué par des verrous consiste à définir une relation d'ordre sur l'ensemble des verrous et à assurer que les threads acquièrent les verrous par ordre croissant d'importance. Ceci revient à ordonner les objets ou les classes. Dans l'exemple précédent, si la relation d'ordre définie est `o1` $\prec$ `o2`, la deuxième ligne est erronée. Il est alors assez simple de vérifier qu'un code proprement écrit respecte la contrainte d'ordre.

Pour synchroniser des threads sur des conditions logiques, on dispose du couple d'opérations permettant d'assurer le blocage et le déblocage des threads (`wait()`, `notify()`). Ces opérations sont applicables à tout objet, pour lequel le thread a obtenu au préalable l'accès exclusif. L'objet est alors utilisé comme une sorte de variable condition.

- `o.wait()` libère l'accès exclusif à l'objet et bloque le thread appelant en attente d'un réveil via une opération `o.notify()` ;
- `o.notify()` réveille une thread bloqué sur l'objet (si aucun thread n'est bloqué, l'appel ne fait rien) ;

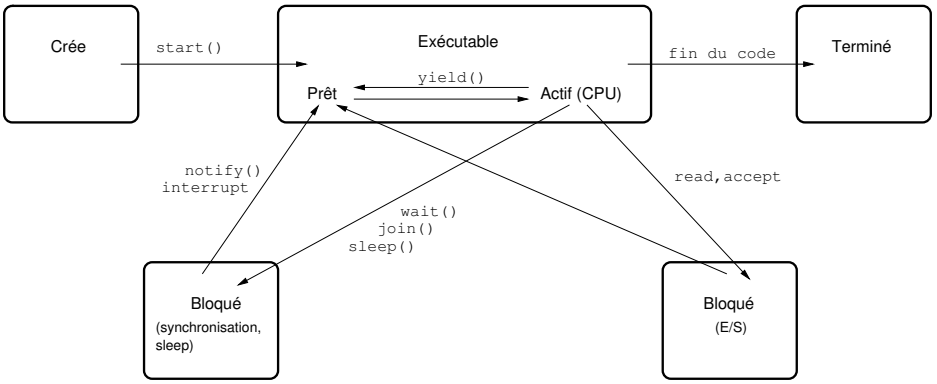

FIGURE 3 – Cycle de vie d'un thread

— `o.notifyAll()` réveille tous les threads bloqués sur l'objet.

Lorsqu'une activité est réveillée, elle est mise en attente de l'obtention de l'accès exclusif à l'objet.

Pour rappel, l'opération `join()` a un rôle particulier : celui d'attendre la fin de l'exécution d'un thread (encradré 2).

L'objectif des exercices de la série suivante est de réaliser un ensemble de primitives permettant la synchronisationn, la coordination ou encore la communication entre plusieurs threads. Ces primitives pourront être réutilisées dans le projet. Vous devez **éviter d'utiliser les classes du package** `java.util.concurrent`. Si vous n'avez pas encore testé l'exercice du *Sleeping Baber*, implanter le programme réalisé en TD avec la méthode `sleep` puis modifier le pour avoir une version utilisant `wait` et `notify`.

Exercice 1. Sémaphore et algorithme de Petterson

1. Implanter en Java le mécanisme de sémaphores vu en cours (opérations `P` et `V`) au moyen d'une classe `Semaphore` et des méthodes appropriées. Vous aurez le choix d'utiliser une attente active (appelée aussi *spin-lock* dans certains langages de programmation) ou une attente passive c'est-à-dire avec une préemption du thread (écrire les deux versions). **Utiliser exclusivement les constructions `synchronized`, `wait`, `notify` et `sleep`** ;
2. Tester vos sémaphores pour assurer l'exclusion mutuelle pour l'exercice de la banque, tester les deux méthodes : l'attente active, et l'attente passive. Suivez l'évolution de l'usage du CPU, conclure.
3. Implanter un mécanisme de verrous. Valider le fonctionnement de votre classe.
4. Définir un package proposant vos différents verrous et sémaphores.
5. Dans votre package ajouter un nouveau type de primitive pour assurer l'exclusion mutuelle correspondant à l'algorithme de Petterson vu en cours. Le principe de cet algorithme est de ne pas utiliser les primitives de base (moniteur, verrou, sémaphore). Le valider sur l'exemple de la banque.

À la fin de cet exercice vous devez avoir au moins 5 programmes utilisant différentes stratégies pour assurer l'exclusion mutuelle et réaliser en toute sécurité les retraits sur le compte bancaire. Vous pouvez aussi vous aider des exercice *Sleeping Barber* et parking vus en TD.

Exercice 2. Synchronisation, problèmes et modèles type

Les processus concurrents qui s'exécutent dans le système d'exploitation peuvent être indépendants ou bien coopérants. Un processus est dit indépendant s'il ne peut pas influencer (ou être influencé par) d'autres processus. Un processus est dit coopérant s'il peut influencer (ou être influencé par) d'autres processus. Les processus qui partagent des données ou les processus qui communiquent sont, par conséquent, des processus coopérants. Ils doivent alors coordonner leurs actions pour réaliser une tâche. Dans les systèmes d'exploitation, la communication inter-processus est généralement nommée IPC (*interprocess communication*). Dans cet exercice on se propose d'étudier différents modes de communication permettant de synchroniser (ou coordonner) les activités des processus.

Réaliser les programmes correspondants aux situations de communication suivantes :

1. **Producteurs/Consommateurs :** dans ce programme producteurs et consommateurs partageant un buffer borné (à une seule place pour commencer puis ensuite a n places), il s'agit pour les producteurs et les consommateurs de coordonner leur activités autour du buffer (c'est-à-dire de se synchroniser) avec les contraintes suivantes :
   — deux producteurs ou consommateurs, ne peuvent pas accéder en même temps au buffer ;
   — si le buffer est plein les producteurs sont bloqués ;
   — si le buffer est vide les consommateurs attendent que des producteurs produisent pour continuer à s'exécuter.
2. **Lecteurs/Rédacteurs :** il s'agit d'un modèle de synchronisation de processus étudié depuis longtemps[1] [Courtois et al., 1971], [Dijkstra, 2001], [Knuth, 1966], qui sert également à tester les mécanismes d'exclusion mutuelle. Supposons qu'une ressource soit manipulée par deux catégories de processus, des lecteurs et des rédacteurs. Les contraintes sont : 1) plusieurs lecteurs doivent pouvoir lire la ressource en même temps ; 2) si un rédacteur est en train de modifier la ressource, aucun autre utilisateur (rédacteur ou lecteur) ne doit pouvoir y accéder.
   (a) Réaliser le programme, instancier plusieurs lecteurs, appliquer la règle selon laquelle les rédacteurs sont mis en attente tant qu'il y a des lecteurs et constater que si les lecteurs effectuent de nombreuses lectures, une situation de famine (des rédacteurs) s'en suit.
   (b) Donner la priorité aux rédacteurs. Que constatez vous ?
   (c) Comment réaliser un système équitable ?
3. **Barrière de synchronisation :** dans ce programme, on exécute les activités[2] pas à pas en utilisant des barrières de synchronisation. Pour passer d'un pas à un autre une activité doit attendre que toutes les autres activités aient terminé le pas précédent. Ainsi, on peut ainsi simuler dans un système asynchrone (par exemple cluster de calcul) des calculs qui nécessitent des parties synchrones (pour l'assemblage de résultats par exemple). Une barrière est un compteur qui peut être incrémenté et décrémenté (mais en restant toujours entre une valeur minimale `valMin` et une valeur maximale `valMax`). Si l'opération d'incrémentation ou de décrémentation ne peut pas se faire sans violer cette contrainte, alors l'opération est rendue bloquante jusqu'à ce que la condition soit validée ;

1. https://cs.nyu.edu/~lerner/spring10/MCP-S10-Read04-ReadersWriters.pdf
2. ensemble d'instructions

4. Utiliser le paradigme producteurs consommateurs pour réaliser une file de messages utilisée par plusieurs processus (lorsque la file est vide ou pleine les opérations d'accès sont bloquantes, c'est-à-dire mises en attente).
5. Utiliser les barrières dans l'exercice de simulation de distributeurs de billets pour attendre la fin des threads avant d'afficher le message du solde restant.

Exercice 3.

Le problème dit *Sleeping-Barber Problem* est un exemple classique de la synchronisation des processus. On considère un salon de coiffure qui comporte une salle d'attente avec n chaises et une autre salle avec un fauteuil de coiffure et un coiffeur. Si il n'y a pas de client, le coiffeur dort. Si un client entre et que toutes les chaises sont occupées, le client s'en va. Si le coiffeur est occupé et qu'au moins une chaise est libre le client s'assied et attend. Si le coiffeur est endormi l'arrivée d'un client le réveille.

1. Réaliser le programme du TD en utilisant un endormissement aléatoire du coiffeur.
2. Modifier le programme pour répondre exactement à la situation décrite dans l'énoncé. Vous aurez besoin d'utiliser les méthodes `wait()` et `notify()`.

Exercice 4. Un timer

Réaliser une classe `timer` qui permet d'instancier des `timers`, c'est-à-dire des objets qui lorsqu'ils sont appelés suspendent le thread qui les appelle et le réveille après un temps donné (passé en paramètre à l'appel).

3 Création de processus et threads de C et C++

Ces exercices sont prévus pour moins d'une séance (compte 1h15 environ). L'objectif est de comparer le modèle de thread du C (threads POSIX) avec celui de Java. Notamment de s'interroger sur la filiation des threads et des processus. Vous devez interpréter l'affichage des programmes afin de déterminer quels processus ou threads ont été créés, leur filiation puis faire un schéma (arbre). Vous ajouterez ensuite un temps d'attente à la fin des programmes (avec la fonction C `sleep(nbsecondes)` [3].) pour vous permettre de visualiser, dans une autre fenêtre terminal ce qui se passe réellement. Utilisez pour se faire les commandes :

— `ps -efL` pour lister les thread associés aux processus ;
— `pstree -h -p` pour visualiser l'organisation des threads et des processus.

Exercice 1. Création de processus

1. Compiler et exécuter les programmes suivants (le code source est disponible sur le GitHub du cours) : `lst0.c`, `lst1.c`, `lst2.c`. L'encadré 4 précise comment compiler les programmes C.
2. Interpréter leur comportement : réaliser l'arbre de filiation des processus.
3. Reprendre l'exercice de la première séance du TD (exercice 6) qui entraîne une création non contrôlée de 32 processus pour reproduire son comportement sur votre machine en utilisant l'appel système `fork()`.

3. La bibliothèque standard du langage C propose également la fonction `usleep(n)` qui comme la méthode `sleep` de Java endors le thread ou le processus pendant `n` microsecondes

4. Corriger le programme afin d'obtenir uniquement 5 processus fils.

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

int main (void)
{
  int valeur;
  valeur = fork();
  printf ("Valeur␣retournée␣par␣la␣fonction␣fork␣:␣%d\n", (int)
     valeur);
  printf ("Je␣suis␣le␣processus␣:␣%d\n", (int)getpid());
}
```

Listing 5 – Création de processus (`lst0.c`)

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

int main (void)
{
  int valeur;
  valeur = fork();
  if (valeur == 0) sleep (4);
  printf ("␣Valeur␣retournee␣par␣la␣fonction␣fork␣:␣%d\n", (int)
     valeur);
  printf ("Je␣suis␣le␣processus␣:␣%d\n", (int)getpid());
}
```

Listing 6 – Création de processus avec attente (`lst1.c`)

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

int main (void)
{
  int valeur, valeur1 ;
  printf ("␣␣1␣-␣Je␣suis␣le␣processus␣pere␣num␣=␣%d␣\n", (int)
     getpid() );
  valeur = fork();
  printf ("␣␣2␣-␣retour␣fork␣:␣%d␣-␣processus␣num=␣%d␣-␣num␣pere=%d
     ␣\n", valeur, (int)getpid(), (int)getppid() );
  valeur1 = fork();
  printf ("␣␣3␣-␣retour␣fork␣:␣%d␣-␣processus␣num=␣%d␣-␣num␣pere=%d
     ␣\n", valeur1, (int)getpid(), (int)getppid() );
}
```

Listing 7 – Création de processus multiples (`lst2.c`)

Exercice 2. Création de thread sous UNIX

1. Compiler et exécuter les programmes suivants : `lst3.c` et `lst4.c`.

ENCADRÉ 4 – Compilation d'un programme C et manuel en ligne

Pour compiler un programme C sous Unix on utilise la commande :
- `gcc -o prog prog.c`
- où `prog` est le nom de l'exécutable produit.

Si l'option `-o` n'est pas utilisée, l'exécutable sera un fichier nommé `a.out`.

Pour compiler en utilisant des bibliothèques de fonctions (*library)* on utilise l'option `-l`. Par exemple `gcc -o monthread monthread.c -lpthread`.

L'option `-Wall` permet de compiler et d'afficher tous les warnings (avertissements), un programme bien construit ne doit pas produire de warning et surtout pas sur des pointeurs.

Unix et Linux proposent un manuel en ligne pour les commandes et les fonctions C standard. Les section du manuel sont organisées en niveaux de 1 à 7. Le niveau 3 correspond au fonction C. Pour accéder à la description de la fonction `sleep` on utilisera la commande `man 3 sleep`.

2. Interpréter leur comportement : réaliser l'arbre de filiation. Vérifier votre proposition.
3. Créer d'autres threads à partir d'un thread, observer l'arbre de filiation. Que constatez vous ?
4. Reprendre le programme sur la banque et les distributeurs et le réaliser en C en utilisant les sémaphores proposés par la bibliothèque Thread POSIX. Inspirez vous du programme, donné en exemple. Comment passer des paramètres aux threads ?

```
1  #include <stdio.h>
   #include <stdlib.h>
3  #include <pthread.h>

5  void *my_thread_process (void * arg)
   {
7    int i;

9    for (i = 0 ; i < 5 ; i++) {
       printf ("Thread␣%s:␣%d\n", (char*)arg, i);
11     sleep (1);
     }
13   pthread_exit (0);
   }
15
   main (int ac, char **av)
17 {
     pthread_t th1, th2;
19   void *ret;

21   if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
       fprintf (stderr, "pthread_create␣error␣for␣thread␣1\n");
23     exit (1);
     }
25
```

```
  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
27  fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
29  }

31  (void)pthread_join (th1, &ret);
    (void)pthread_join (th2, &ret);
33 }
```

Listing 8 – Création de thread (`lst3.c`)

```
1  #include <stdio.h>
   #include <stdlib.h>
3  #include <pthread.h>
   #include <semaphore.h>
5
   static sem_t my_sem;
7  int the_end;

9  void *thread1_process (void * arg)
   {
11   while (!the_end) {
       printf ("Je t'attend !\n");
13     sem_wait (&my_sem);
     }
15
     printf ("OK, je sors !\n");
17   pthread_exit (0);
   }
19
   void *thread2_process (void * arg)
21 {
     register int i;
23
     for (i = 0 ; i < 5 ; i++) {
25     printf ("J'arrive %d !\n", i);
       sem_post (&my_sem);
27     sleep (1);
     }
29
     the_end = 1;
31   sem_post (&my_sem); /* Pour debloquer le dernier sem_wait */
     pthread_exit (0);
33 }

35 main (int ac, char **av)
   {
37   pthread_t th1, th2;
     void *ret;
39
     sem_init (&my_sem, 0, 0);
41
     if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
43     fprintf (stderr, "pthread_create error for thread 1\n");
       exit (1);
45   }
```

```
47    if (pthread_create (&th2, NULL, thread2_process, NULL) < 0) {
        fprintf (stderr, "pthread_create error for thread 2\n");
49      exit (1);
      }
51
      (void)pthread_join (th1, &ret);
53    (void)pthread_join (th2, &ret);
}
```

Listing 9 – Threads et coordination (`lst4.c`)

Ressources Web (validées) :

- https://openclassrooms.com/fr/courses/1513891-la-programmation-systeme-en-c-sous-unix/1514567-les-threads l'essentiel des fonction pour la gestion des thread en C
- https://www.ibm.com/developerworks/library/l-posix1/index.html une bonne introduction
- https://mtodorovic.developpez.com/linux/programmation-avancee/?page=page_4 pour aller au delà du TP et manipuler les thread en C dans des programmes complexes ;

4 Fichiers et interactions avec le système d'exploitation

Ce TP est prévu pour une séance.

Exercice 1. Lecture de fichier en Java

1. Écrire un programme qui permet de lire ligne à ligne le contenu d'un fichier texte et qui affiche les lignes lues à l'écran ;
2. Le fichier à considérer dans cette seconde question est un fichier contenant des entiers. Le programme doit en afficher la somme. Utilisez les indications suivantes.

Indications :

La classe `StreamTokenizer` permet de réaliser un analyseur lexical simple. Cette analyseur considère plusieurs types d'unités lexicales (lexème ou tokens) :

- nombre
- mot (délimité par les espaces)
- chaîne de caractères (délimitée par des quotes)
- commentaires
- EOF, EOL
- etc.

Il est possible de récupérer le type du lexème dans la variable d'instance `ttype`, la valeur numérique dans `nval` et la valeur de la chaîne dans `sval`. La méthode qui appelle l'analyseur est `nextToken`.

On peut également redéfinir les divers caractères spéciaux qui servent de délimiteurs. Dans la classe, la méthode `wordChars` définit les caractères pour les mots, `resetSyntax` remet à vide tous les caractères spéciaux, `whitespaceChars` définit les caractères qui servent de délimiteurs pour les mots.

Les variables d'instance et les constantes de classes qui vous seront utiles dans l'exercice sont :

ENCADRÉ 5 – Passer des paramètres à un thread, retourner des résultats

On utilise le pointeur non-typé déclaré comme paramètre de la fonction qui est prise en charge par le thread pour désigner une portion mémoire contenant le ou les paramètres. Il peut s'agit par exemple d'un `struct`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void *run(void *i) {
    int a = *((int *) i);
    printf("jai␣recu␣%d␣\n",a);
}
int main() {
    pthread_t thread;
    int valeur = 5;
    int *i ;
    i=&valeur;
    pthread_create(&thread, 0, run, (void *) i);
    sleep(2);
}
```

Le programme suivant illustre deux modes de transmission de valeurs à partir du thread vers le programme principal : avec une variable globals `message` ou explicitement en demande au thread le résultat avec la méthode `pthread_join`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
char message[]= "Hello␣world!";
void *run_du_thread(void *arg){
 printf("thread␣en␣cours␣d'exécution,␣arg␣=␣%s\n",(char *)arg);
 sleep(5);
 // attention à la taille réservée par message[]
 strcpy(message,"le␣thread");
 pthread_exit("fin␣du␣thread");
}
int main(){
 int res;
 pthread_t le_thread;
 void *resultat_thread;
 res = pthread_create(&le_thread, NULL, run_du_thread, (void *)
    message);
 if (res != 0)
 {
  perror("Echec␣à␣la␣creation␣du␣thread");
  exit(EXIT_FAILURE);
 }
 printf("En␣attente␣de␣la␣terminaison␣du␣thread...\n");
 res = pthread_join(le_thread, &resultat_thread);
 printf("Retour␣du␣thread␣:␣%s␣\n", (char *)resultat_thread);
 printf("Contenu␣de␣message␣:␣%s␣\n", message);
 exit(EXIT_SUCCESS);
}
```

— `double nval` : contient la valeur du nombre si le lexème courant est un nombre ;
— `String sval` : ce champ contient une chaîne de caractères représentant ce mot, si...
— `static int TT_EOF` : représente une constante [4] indiquant que la fin du flux a été lue ;
— `static int TT_EOL` : représente une constante indiquant que la fin de la ligne a été lue ;
— `static int TT_NUMBER` : représente une constante indiquant qu'un nombre a été lu ;
— `static int TT_WORD` : représente une constante indiquant qu'un mot a été lu ;
— `int ttype` : contient le type du lexème venant d'être lu (suite à un appel à la méthode `nextToken`).

Exercice 2. Gestion de chaînes de caractères

Réaliser un programme qui compte le nombre d'occurrences des différents mots d'un texte et qui affiche ensuite pour chaque mot le nombre d'occurrences trouvées.

Exercice 3. Structure d'accès et sérialisation

Réaliser un programme qui permet de gérer un annuaire téléphonique simple. On ne demande que les fonctions de bases et non pas la partie interface utilisateur.

1. Définir la structure de la classe `Personne`, quelques attributs, accesseurs et modificateurs.
2. À partir de la documentation de la JavaDoc résumer les méthodes principales de la classe `Hastable` (https://depinfo.u-bourgogne.fr/doc/j2sdk-1.6.30/docs/api/). En quoi les tables de hachage diffèrent-elles des tableaux et des `ArrayList`.
3. Sachant que la classe `Annuaire` est vue comme un ensemble de personnes, utiliser une table de hachage pour permettre des recherches, selon le nom ou le numéro de téléphone. On suppose qu'il n'y a pas d'homonymes.
4. Implanter le mécanisme de sérialisation pour enregistrer l'annuaire et pouvoir le recharger lors d'une nouvelle exécution du programme.
5. On souhaite effectuer aussi des recherches par ville et pouvoir gérer les homonymes. Faites évoluer votre classe `Annuaire` pour permettre à la hastable de gérer des listes [5] et non plus un objet unique associé à une clé.

Remarques :

À l'issue de ces séries d'exercices pratiques, et avant de commencer les exercices sur la communication réseau, vous devez avoir programmé et testé les exercices de TD concernant :

— *the Sleeping Barber* en respectant les règles de l ?énoncé c'est-à-dire en utilisant un événement pour réveiller le coiffeur. De plus vous devez être en mesure de compter le nombre de clients traités et perdus, d'ajouter des coiffeurs juste en modifiant la méthode `main` et l'instanciation des différent composants de la simulation ;
— le programme de la simulation de parking ;
— le programme du Ferry ou du Tramway ;

4. Comme toutes les constantes de classe définies `public static final` leur utilisation est préfixée par le nom de la classe qui les a défini (dans ce cas `StreamTokenizer.TT_EOF`).
5. dans notre cas des `ArrayList`

— le programme concernant le problème des philosophes, une version faisant apparaître un verrou mortel, une autre version permettant de contourner le problème et une troisième en détectant le problème puis en le corrigeant.

ENCADRÉ 6 – Classe `Hastable` de Java

La classe `Hashtable` du package `java.util` permet de créer des collections d'objets associés à un attribut clé. Une hashtable peut contenir des objets de classe quelconque. Les constructeurs disponibles sont :

— `Hashtable()`
— `Hashtable(n)` ou n est un entier permettant de donné une capacité initiale à la hastable.

Les opérations d'ajout `put()`, de consultation `get()` et de suppression `remove()` utilisent les méthodes `equals()` et `hashCode()` :

— `hashCode()` sert à calculer le code de hachage de la clé d'un objet Java quelconque ;
— `equals()` sert à vérifier l'égalité entre les codes de hachage de deux objets.

Lors du stockage d'un couple (clé, référence objet), l'objet représentant la clé est automatiquement soumise à la méthode `hashCode()` afin d'en récupérer un code de hachage identifiant d'une manière unique l'objet valeur. Ensuite, les codes de hachage sont comparés de telle sorte à vérifier si la clé est déjà présente dans la table de hachage. Si tel est le cas, la valeur donnée remplace l'ancienne.
https://depinfo.u-bourgogne.fr/doc/j2sdk-1.6.30/docs/api/

5 Réseau et socket

Dans ce TP on se propose d'étudier l'utilisation les sockets afin de réaliser des applications clients serveur. Les sockets sont une abstraction permettant de communiquer facilement entre deux machines connectées par un réseau utilisant le protocole TCP/IP. À l'aide des sockets, il est possible de lire et écrire sur des connexions associées à des machines distantes, comme si il s'agissait de simples fichiers (flux), sans se soucier des détails concernant les couches réseau de niveau inférieur.

Ce TP est prévu pour deux séances. Vous devez avoir sous la main les chapitres 8 et 9 et vous inspirer des exemple donnés en cours pour réaliser les question 1,2 et 3 de l'exercice suivant.

Principe des sockets :

Une application serveur ou client est en général dédiée à un service particulier tel que HTTP (*HyperText Transfer Protocol*) ou FTP (*File Transfer Protocol*). En utilisant les sockets serveur, une application qui offre l'un de ces services peut se lier aux applications client qui souhaitent utiliser le service. De même, les sockets client permettent à une application de se lier à des applications serveur offrant ce service.

Implémentation des services :

Les sockets, en tant qu'objet Java, constituent le composant essentiel pour créer des applications serveur ou client. Les sockets sont l'interface entre l'application serveur ou client et la couche réseau du noyau du système d'exploitation. Pour beaucoup de services, tels que HTTP, FTP ou Telnet des programmes serveurs ont été développés et sont proposés en standard dans les systèmes d'exploitation. Si aucun serveur n'est disponible pour le service précis dont vous avez besoin, vous devrez créer votre propre programme serveur ou client. Par exemple, lorsque vous créez une application distribuée, vous pouvez utiliser la bibliothèque de fonctions socket pour communiquer avec les systèmes distants.

Description des protocoles de services :

Avant de créer une application réseau (un serveur ou un client), vous devez concevoir le service que votre application offrira ou utilisera, c'est-à-dire identifier les fonctionnalités que vous proposez. La majorité des services ont des protocoles standard que votre application doit supporter si elle désire les utiliser. Si vous créez une application cliente réseau pour un service standard tel que HTTP, FTP, Finger [6] ou Telnet, vous devez comprendre les protocoles utilisés pour communiquer avec les systèmes distants. Pour chaque type protocole de niveau application (au dessus de TCP/IP) utilisé sur Internet il existe une documentation très précise se rapportant au service que vous comptez implémenter ou utiliser. Ces documentations sont décrites sous forme de standards RFC [7] (*Request For Comment*) et portent des numéros identificateurs et les documents sont librement accessibles (`https://tools.ietf.org/rfc/index`) comme par exemple la RFC sur FTP (`https://tools.ietf.org/html/rfc414`) Dans le cas ou vous voulez implémenter un nouveau service pour une application qui communique avec des systèmes distants, la première étape consiste à concevoir le protocole de communication pour les serveurs et les clients de ce service. Quels messages sont envoyés ? Comment ces messages sont ils structurés ? Comment les informations sont elles codées ?

Dans le langage Java, plusieurs classes du package `java.net` interviennent lors de la réalisation d'une communication. La classe `InetAddress` permet de manipuler des adresses

6. Finger permet de connaître les paramètres d'un utilisateur
7. `https://fr.wikipedia.org/wiki/Liste_de_RFC`

IP. La classe `SocketServer` permet de programmer l'interface côté serveur en mode connecté. La classe `Socket` permet de programmer l'interface côté client et la communication effective via un flux relié au socket. Les classes `DatagramSocket` et `DatagramPacket` permettent de programmer la communication en mode datagramme.

Adresses en Java :

La classe `java.net.InetAddress` permet de manipuler les adresses Internet. Il n'existe pas de constructeur associé, ainsi, pour obtenir une instance de cette classe, on utilise les méthodes de classe suivantes :

- `public static InetAddress getLocalHost()` : retourne un objet contenant l'adresse Internet de la machine locale ;
- `public static synchronized InetAddress getByName(String hostName)` : retourne un objet contenant l'adressse Internet de la machine dont le nom est passé en paramètre ;
- `public static synchronized InetAddress[] getAllByName(String hostName)` : retourne un tableau d'objets contenant l'ensemble des adressses Internet de la / des machine(s) qui répond(ent) au nom passé en paramètre.

Les méthodes que l'on peut appliquer à un objet de la classe `InetAddress` sont :

- `public String getHostName ()` : retourne le nom de la machine dont l'adresse est stockée dans l'objet ;
- `public byte[] getAddress ()` : retourne l'adresse internet stockée dans l'objet sous forme d'un tableau de 4 octets ;
- `public String toString ()` : retourne une chaîne de caractères qui liste le nom de la machine et son adresse.

Veiller à bien propager ou capter les exceptions qui peuvent être déclenchées par l'utilisation des méthodes.

```
1 InetAddress adresseLocale = InetAdress.getLocalHost ();
  InetAddress adresseServeur = InetAdress.getByName("depinfo.u-
     bourgogne.fr");
3 System.out.println(adresseServeur);
```

Listing 10 – Exemple de manipulation d'adresses

Services et ports :

La plupart des services standards sont associés, par convention, à des numéros de ports précis. Pour l'instant, nous considérons le numéro de port comme un code numérique pour un service donné qui permet au noyau de délivrer les données aux processus concernés. Si vous implémentez un service standard (décrit par une RFC), les sockets du système d'exploitation fournissent des méthodes de recherche de numéro de port pour le service à partir de son nom. Si vous offrez un service nouveau, vous pouvez spécifier son numéro de port dans un fichier `SERVICES` dans le répertoire windows, sur les systèmes d'exploitation Windows et dans le fichier services du répertoire `/etc` sous Unix. Les 1024 premiers ports sont réservés pour le système d'exploitation et les services standards. Les autres ports (codés sur 2 octets) sont utilisables pour les programmes utilisateur.

Types de connexions par socket :

Les connexions par socket sont de trois types indiquant la façon dont la connexion a été ouverte et la catégorie de connexion :

- **connexions client :** les connexions client connectent un socket client sur le système local à un socket serveur sur un système distant. Les connexions client sont lancées par le socket client. En premier lieu, le socket client doit décrire le socket

serveur auquel il souhaite se connecter c'est-à-dire fournir l'adresse IP de la machine serveur et le port. Le socket client envoie ensuite une demande de connexion sur le socket serveur et, lorsqu'il l'a trouvé, ce dernier valide la connexion et informe le client. Le socket serveur peut ne pas établir immédiatement la connexion. Les sockets serveur, hébergés par la noyau, gèrent une file d'attente des demandes de clients et établissent la connexion lorsqu'ils le peuvent. Lorsque le socket serveur accepte la connexion du client, il envoie au socket client une description complète du socket serveur auquel il se connecte et la connexion est finalisée par le client ;

- **connexions d'écoute :** les sockets serveur génèrent des demi-connexions passives qui restent à l'écoute des requêtes des clients. Les sockets serveur associent une file d'attente à leurs connexions d'écoute. La file d'attente enregistre les requêtes de connexion lorsqu'elles lui parviennent. Lorsque le socket serveur accepte une demande de connexion client, il forme un nouveau socket pour se connecter au client pour que la connexion d'écoute reste ouverte afin d'accepter d'autres requêtes de clients ;
- **connexions serveur :** les connexions serveur sont formées par des sockets serveur lorsque le socket d'écoute accepte une requête du client. La description du socket serveur ayant effectué la connexion au client est envoyée au client lorsque le serveur accepte la connexion. La connexion est établie lorsque le socket client reçoit cette description et effectue véritablement la connexion.

Lorsque la connexion au socket client est complètement réalisée, la connexion serveur est identique à une connexion client : les deux extrémités ont les mêmes possibilités.

La classe `ServerSocket`

La classe `public class java.net.ServerSocket` implante un objet ayant un comportement de serveur communiquant via un socket. Une implantation standard du service socket existe, elle peut être redéfinie en donnant une implantation spécifique sous la forme d'un objet de la classe `SocketImpl`. Les constructeurs `ServerSocket` sont les suivants :

- `public ServerSocket(int port)`
- `public ServerSocket(int port, int backlog)`
- `public ServerSocket(int port, int backlog, InetAddress bindAddr)`

Ces constructeurs créent un objet serveur à l'écoute du port spécifié. La taille de la file d'attente des demandes de connexion peut être explicitement spécifiée via le paramètre `backlog`. Si la machine possède plusieurs adresses IP, on peut aussi restreindre l'adresse sur laquelle on accepte les connexions. Nous présentons uniquement les opérations de base de la classe `ServerSocket` :

- `public Socket accept()` : cette méthode essentielle permet l'acceptation d'une connexion d'un client. Elle est bloquante, mais l'attente peut être limitée dans le temps par l'appel préalable de la méthode `setSoTimeout` ;
- `public void setSoTimeout(int timeout)` : cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La valeur par défaut 0 équivaut à l'infini. À l'expiration du délai de garde, l'exception `java.io.InterruptedIOException` est levée ;
- `public void close()` : fermeture du socket d'écoute.

Les méthodes suivantes retrouvent l'adresse IP ou le port d'un socket d'écoute :

- `public InetAddress getInetAddress()`
- `public int getLocalPort()`

La classe `java.net.Socket`

La classe `java.net.Socket` est utilisée pour la programmation des sockets connectés, côté client et côté serveur. Comme pour la classe serveur, nous utiliserons l'implantation

standard bien qu'elle soit redéfinissable par le développement d'une nouvelle implantation de la classe `SocketImpl`. Côté serveur, la méthode `accept` de la classe `ServerSocket` renvoie un socket de service connecté au client. Côté client, on utilise les constructeurs suivants :

- `public Socket(String host, int port)`
- `public Socket(InetAddress address, int port)`
- `public Socket(String host, int port, InetAddress localAddr, int localPort)`
- `public Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)`

Les deux premiers constructeurs construisent un socket connecté à la machine et au port spécifiés, par défaut, la connexion est de type TCP (donc fiable). Les deux autres constructeurs permettent de fixer l'adresse IP et le numéro de port utilisés côté client plutôt que d'utiliser un port disponible quelconque.

La communication effective sur un socket utilise la notion des flux (`OutputStream` et `InputStream`). Les méthodes suivantes sont utilisées pour obtenir les flux en entrée et en sortie :

- `public InputStream getInputStream()`
- `public OutputStream getOutputStream()`

Les flux ainsi obtenus servent de base à la construction d'objets de classes plus abstraites telles que `DataOutputStream` et `DataInputStream` (pour le JDK v1), ou `PrintWriter` et `BufferedReader` (JDK 2 c'est-à-dire J2SDK 1.2 et suivants). Les opérations de lecture sur ces flux sont bloquantes. Il est néanmoins possible de fixer une délai de garde à l'attente de données (`public void setSoTimeout(int timeout)`. Les méthodes suivantes permettent d'obtenir les éléments constitutifs de la liaison établie :

- `public InetAddress getInetAddress()` : fournit l'adresse IP distante
- `public InetAddress getLocalAddress()` : fournit l'adresse IP locale
- `public int getPort()` : fournit le port distant
- `public int getLocalPort()` : fournit le port local

La méthode `close` ferme la connexion et libère les ressources du système associées au socket.

Socket en mode datagramme via la classe `DatagramSocket`

La classe `java.net.DatagramSocket` permet d'envoyer et de recevoir des paquets (datagrammes UDP). Il s'agit de messages non fiables, non ordonnés et dont la taille est assez faible. Ainsi lors de l'utilisation de datagrammes il y a possibilités de pertes et/ou de duplication, les messages peuvent être reçus dans un ordre différent de celui d'émission. Ces problèmes doivent être gérés par le programmeur. Les principaux constructeurs de la classe `DatagramSocket` sont :

- `public DatagramSocket()`
- `public DatagramSocket(int port)`

Ils permettent de construire un socket datagramme en spécifiant éventuellement un port sur la machine locale. Par défaut, un port disponible quelconque est choisi. Pour l'émission/réception les méthodes suivantes sont fournies :

- `public void send(DatagramPacket p)`
- `public void receive(DatagramPacket p)`

Ces deux opérations permettent d'envoyer et de recevoir un paquet (c'est-à-dire un objet de la classe `DatagramPacket`) Un paquet possède une zone de données et éventuellement une adresse IP associée avec un numéro de port (destinataire dans le cas `send`, émetteur dans le cas `receive`). Les principales méthodes sont :

- `public DatagramPacket(byte[] buf, int length)`

— `public DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
— `public InetAddress getAddress()`
— `public int getPort()`
— `public byte[] getData()`
— `public int getLength()`
— `public void setAddress(InetAddress iaddr)`
— `public void setPort(int iport)`
— `public void setData(byte[] buf)`
— `public void setLength(int length)`

Il est possible de connecter un socket datagramme à un destinataire. Dans ce cas, les paquets émis sur le socket seront toujours pour l'adresse spécifiée. La connexion simplifie l'envoi d'une série de paquets, ainsi il n'est plus nécessaire de spécifier l'adresse de destination pour chacun. La déconnexion supprime l'association (le socket redevient disponible comme dans l'état initial).

— `public void connect(InetAddress address, int port)`
— `public void disconnect()`

Ressources :
http://java.sun.com/docs/books/tutorial/networking/index.html
http://java.sun.com/docs/books/tutorial/networking/sockets/readingWriting.html
http://java.sun.com/j2se/1.4.2/docs/index.html

Exercice. Programme `echo` largement amélioré

Vous devrez, à la fin des deux séances consacrées à ce TP, avoir réalisé un programme de chat complet, multi-clients.

1. Réaliser les deux programmes vus en cours pour établir une communication entre un processus client et un processus serveur :
   — les deux processus échangent des messages sous forme de lignes de texte ;
   — le processus client commence par émettre un message et le serveur lui répond par un écho de cette ligne concaténée avec le mon du serveur.
   — au bout de n échanges, le client envoie une commande de terminaison (`END`) et ferme la connexion. Il est possible comme dans beaucoup de protocoles d'utiliser plusieurs sockets comme par exemple un socket pour les données et un autre socket pour les commandes. Dans notre cas il y aurait un socket pour les messages et un autre pour les commandes.

   Vous pouvez tester le serveur en utilisant la commande `telnet` du système d'exploitation suivie des arguments nom de machine cible et port de destination ;
2. Développer le serveur de façon à ce qu'il puisse gérer plusieurs clients simultanément : un thread sera dédié à chaque connexion client et gérera les flux. Chaque client à un nom (ou pseudo) indiqué lors du lancement par un argument sur le ligne de commande ;
3. Modifier vos programmes afin de réaliser un outil de chat simple qui diffuse à tous les clients les messages envoyés par l'un d'entre eux. Les différentes connexions (flux) des clients sont stockées dans un tableau (commun ou pas). Les échanges de messages entre un client et le serveur et entre les clients doivent se faire avec des objets sérializables qui contiennent le pseudo et le texte du message ;
4. Le serveur propose plusieurs canaux de discussion, un canal sera sélectionné lors de la connexion (après une éventuelle sélection d'un canal parmi la liste proposée). En

option, un utilisateur devrait pouvoir créer un canal et gérer des abonnements. Dans cette question vous commencez à établir un protocole entre les clients et le serveurs en vous servant de l'objet message que vous avez crée à la question précédente. Quelques indications pour définir le protocole :
   — les commande peuvent être : `CREATE CHANNEL`, `SET CHANNEL`, `SUBSCRIBE CHANNEL`, `UNSUBSCRIBE CHANNEL`, toutes avec comme paramètre le nom du canal, `LIST` pour avoir la liste des canaux existant, `WHO` pour savoir qui est connecté, ces deux dernières commandes n'ont pas de paramètre ;
   — les messages sont alors composés de 4 attributs : le pseudo, la commande, le paramètre de la commande, le texte du message ;
   — au lieu d'utiliser des chaînes de caractère pour les commandes, vous pouvez les analyser sur le client et code chaque commande dans le message par un entier.
5. À des fins de traçabilité, le serveur doit enregistrer l'ensemble des messages reçus de chaque client avec leur date et heure, ainsi que les informations de connexion déconnexion et les stocker dans un fichier.
6. Comment chiffrer les échanges entre les clients et le serveur ? Quel est l'impact du chiffrement sur votre programme ?
7. On ne souhaite plus utiliser de programme serveur centralisé mais un seul programme qui joue le rôle de client et serveur. Ainsi tous les participant sont au même niveau (réseau de pair à pair). Proposer une architecture, définir les classes pour retrouver les mêmes fonctionnalités que dans la version précédente.

6 Exercices de synthèse

Exercice 1

Vous devez écrire le code Java du programme correspondant aux comportements suivants. Les lignes de code réalisant chaque élément (de 1 à 7) doivent être mises en évidence par un commentaire. De la même manière, mettre en évidence les éléments permettant de gérer la concurrence.

1. Créer et lancer deux threads d'une même classe (`TSearch`) à partir du programme principal (`main`), celui-ci aura également instancié un objet de la classe `DB` qui encapsule l'accès à des données textuelles volumineuses.
2. Les threads disposent d'une référence à un objet de la classe `DB` et recherchent respectivement le nombre d'occurrences des mots `Python` et `Scala` au moyen de la méthode `rechercher` de la classe `DB`. La méthode `rechercher` s'arrête à chaque occurrence trouvée et retourne une référence à un objet (on ne l'utilise pas par la suite). Si on relance la méthode, elle continue sa recherche à partir de la position à laquelle elle s'était arrêtée. La méthode retourne `null` si la recherche ne donne plus rien c'est-à-dire si elle a atteint la fin des données.
3. Les threads s'arrêtent si ils ont trouvé plus de 1 000 occurrences d'un mot ou bien si un autre objet appelle la méthode `arrêter` définie sur la classe `TSearch` ou encore si la recherche est terminée.
4. À chaque occurrence de mot trouvée chaque thread incrémente un compteur commun (classe `CptOccurrence` contenant un entier).
5. Le programme principal attend la terminaison des threads pour afficher la valeur du compteur, c'est-à-dire le nombre d'occurrences des mots `Python` ou `Scala`.

6. Le programme principal écrit la valeur du compteur dans un fichier.
7. Un troisième thread est lancé, il arrête les threads `TSearch` si ils n'ont pas terminé leur exécution au bout de 4 minutes.

Exercice 2. Synchronisation de processus, rendez-vous

On considère deux threads A et B, le thread A exécute une méthode composée de deux portions de code a_1 et a_2, le thread B suit le même schéma, les portions de code sont b_1 et b_2.

On veut garantir que la portion de code a_1 sera exécutée avant la portion de code b_2 et que la portion de code b_1 sera exécutée avant la portion a_2.

1. Proposez un programme dans la syntaxe Java utilisant les primitives de synchronisation `wait` et `notify`.
2. Reprendre le même programme en utilisant deux sémaphores afin d'indiquer respectivement que A ou B est arrivé au point de rendez-vous.
3. Un ordre particulier des primitives de synchronisation peut-il provoquer un verrou mortel ?

Références

[Bloch, 2003] Bloch, L. (2003). Les systèmes d'exploitation des ordinateurs. *Histoire, fonctionnement, enjeux.*

[Courtois et al., 1971] Courtois, P.-J., Heymans, F., and Parnas, D. L. (1971). Concurrent control with readers and writers. *Communications of the ACM*, 14(10) :667–668.

[Dijkstra, 2001] Dijkstra, E. W. (2001). Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer.

[Knuth, 1966] Knuth, D. E. (1966). Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5) :321–322.

[Knuth, 1998] Knuth, D. E. (1998). *The Art of Computer Science, Vol. 3 : Sorting and Searching.* Addison-Wesley.

[Silberschatz, 2008] Silberschatz, A. (2008). Principes des systèmes d'exploitation avec java.

[Silberschatz et al., 2014] Silberschatz, A., Galvin, P. B., and Gagne, G. (2014). *Operating system concepts essentials.* John Wiley & Sons, Inc.

Annexe 1 - Comment arrêter un thread

Comme nous l'avons vu dans le premiers exercices de TP, un thread meurt lorsqu'il a fini d'exécuter la méthode run. Il existe cependant des situations pour lesquelles on souhaite pouvoir demander à un thread de s'arrêter (dans un état cohérent).

La classe `Thread` propose deux méthodes qui peuvent être utilisées pour arrêter un thread : `stop` et `destroy` qui permettent respectivement d'arrêter et de détruire un thread. Il ne s'agit pas d'une bonne pratique, d'ailleurs, les deux méthodes `stop` et `destroy` sont dépréciées (*deprecated*) depuis plusieurs version du jdk. De plus utiliser un arrêt brutal d'un thread peut conduire à des situations inconsistantes (certaines variables ont été modifiées mais pas d'autres), voire provoquer un blocage par perte ou oublis d'un verrou.

Annexe 2 - Observer le comportement des processus

Commande strace

La commande `strace` surveille les appels système et les signaux d'un programme spécifique. Elle est utile pour comprendre la séquence d'exécution d'un processus du début à la fin.

```
leclercq@leclercq-Precision-5530:~/temp$ strace ls
execve("/bin/ls", ["ls"], 0x7ffc9e953160 /* 54 vars */) = 0
brk(NULL)                               = 0x558dd18bf000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=140261, ...}) = 0
mmap(NULL, 140261, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f3d49b39000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20b\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=154832, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3d49b37000
mmap(NULL, 2259152, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f3d4970d000
mprotect(0x7f3d49732000, 2093056, PROT_NONE) = 0
```

Par défaut, `strace` affiche tous les appels système, pour afficher uniquement un appel système spécifique, utilisez l'option `strace -e`.

Pour lister les processus :

```
top -c -b | head -50
top -b | head -50
ps -eo pid,ppid,%mem,%cpu,cmd --sort=-%cpu | head
ps -eo pid,ppid,%mem,%cpu,comm --sort=-%cpu | head
```

Pour lister les fichiers ouverts :

```
lsof -u root | less
lsof -i 4 | grep '192.168.0.2'
lsof -p pid
lsof -t /tmp/fichier.txt
lsof -i :8000-8005 # pour les ports
```