# Chapitre 2
# Accès aux ressources

## 1. Les ressources

Un programme s'exécute dans un environnement. L'environnement met des ressources à disposition du programme pour qu'il s'exécute. Les différents types de ressources sont décrits dans ce chapitre.

### 1.1 Les différents types de ressources

Les ressources mises à disposition d'un programme sont relativement peu nombreuses. On peut distinguer les ressources privées à un programme et les ressources partagées par plusieurs programmes.

Dans la première catégorie, la première d'entre elles est la mémoire qui sert à exécuter le programme. La mémoire est segmentée en différentes entités qui ont une utilisation spécifique. Le code exécuté est séparé des données manipulées. Le programme peut demander de la mémoire allouée dynamiquement en fonction des besoins au système d'exploitation. Ce dernier alloue un bloc de mémoire qui est géré par le programme via la bibliothèque standard C, en utilisant les fonctions d'allocations dynamiques `malloc` et `free`. Le programme dispose d'un autre segment où sont stockées les variables locales aux fonctions, ou procédures, ainsi que le passage de paramètres.

L'autre ressource non partagée est le thread. Le thread est une unité d'exécution qui vit au sein d'un programme. Le programme démarre toujours avec une unité d'exécution, il peut demander la création de threads, au système d'exploitation, pour effectuer des calculs simultanés. Le programme libère cette ressource par un appel système ou à la fin de l'exécution de la partie du programme exécutée par le thread.

En dehors de la mémoire et des threads, toutes les ressources sont externes au programme. Cependant, certaines peuvent être vues comme des zones mémoires. Notamment, les mémoires partagées sont des zones de mémoires allouées, à la demande d'un programme, par le système. Elles sont rendues accessibles aux programmes en demandant le « mapping » d'une mémoire partagée par une référence système. En pratique, ce genre de ressource est assez peu utilisée, probablement par méconnaissance de ce genre de fonctionnalité.

Une autre ressource vue comme une zone mémoire est le mapping de fichier en mémoire. Il permet un accès aléatoire, c'est le système d'exploitation qui gère la synchronisation entre le contenu de la mémoire et le contenu du fichier.

Enfin, la dernière ressource disponible est le mapping direct d'un périphérique en mémoire. L'intérêt est que le programme fonctionne dans l'espace utilisateur plutôt que dans l'espace superutilisateur qui est réservé au système d'exploitation. C'est une fonctionnalité plutôt réservée au développement de pilotes de périphériques.

Les autres ressources sont disponibles sous la forme de canaux d'entrées-sorties. C'est un mécanisme très général. Une ressource, fichier, communication réseau ou périphérique, est demandée au système. Ce dernier fourni, au retour de la demande de la ressource, un identifiant au programme qui pourra effectuer des opérations de lecture et d'écriture sur la ressource. D'autres opérations sont disponibles sur les canaux pour effectuer la configuration de la ressource, par exemple, les réglages du débit d'une liaison série.

C'est le programme qui gère l'ensemble de ces ressources, dans les limites autorisées par le système d'exploitation. Notamment, il doit libérer les ressources lorsqu'elles ne sont plus utilisées. Le système d'exploitation libérera toutes les ressources utilisées par le programme lorsque celui-ci a fini son exécution. Dans le cas contraire, les ressources non libérées provoqueront un épuisement des ressources, ce qui entraînera un ralentissement général.

## 1.2 Comment sont-elles manipulées ?

Les ressources sont accessibles soit directement, soit au travers d'appels système. Les ressources mémoires sont librement manipulables par le programme. Le programme ne peut pas écrire dans toutes les zones mémoires pour des questions de sécurité imposée par le système d'exploitation.

Les autres ressources sont manipulées via des appels systèmes. Les appels sont principalement l'ouverture (ou la création), la fermeture, l'écriture et la lecture. D'autres opérations existent, comme la suppression, ou les opérations de configuration de la ressource (droits des fichiers, configuration du canal de communication). Pour les opérations de lecture et d'écriture, un tampon, mis à disposition par le programme, sert de réceptacle à l'information ou contient l'information à envoyer. Pour les opérations annexes, les opérations dépendent des fonctions systèmes disponibles. Elles nécessitent un paramètre en renseignant une donnée structurée complexe allouée par le programme et mise à disposition au système d'exploitation. À noter que pour les opérations d'ouverture, fermeture, lecture et écriture, les appels sont similaires sur les différents systèmes d'exploitation.

En ce qui concerne les accès mémoire, ils sont de deux types. Soit les accès portent sur des variables utilisées dans les algorithmes, soit ils portent sur des blocs de mémoire alloués. En ce qui concerne les variables, le Java dispose du modificateur `volatile` emprunté au C. Toute variable membre d'une classe peut être déclarée `volatile`. En C, le modificateur `volatile` est très utilisé pour la gestion des périphériques. Ce modificateur renseigne le compilateur sur la stratégie à adopter pour le traitement d'une variable. Simplement, `volatile` empêche le compilateur à maintenir la variable dans un des registres du microprocesseur et oblige le programme à lire la variable et à l'écrire conformément à la manière décrite par le programme. Dans le cas des périphériques, le contenu à une adresse varie de manière indépendante au programme. Par conséquent, en maintenant le contenu dans un registre du microprocesseur, la valeur contenue dans le registre et la mémoire peuvent différer. Voici un exemple pour bien montrer la différence avec deux fonctions C écrites. La première utilise une variable volatile, contrairement à la seconde.

```
int fctvol() {
    volatile int j;
    int i;
    j = 0;
    for(i=0; i<100; i++) {
        j = j*2 + i;
    }

    return j;
}
```

```
fctvol:
    subq.l #4,%sp
    clr.l (%sp)
    clr.l %d1
.L2:
    move.l (%sp),%d0      LECTURE MEMOIRE J
    add.l %d0,%d0                 J*2
    add.l %d1,%d0                 J*2+I
    move.l %d0,(%sp)      ECRITURE MEMOIRE J
    addq.l #1,%d1         i <- i + 1
    moveq #100,%d0
    cmp.l %d1,%d0         Comparaison i avec 100
    jne .L2               i !=100 -> .L2
    move.l (%sp),%d0
    addq.l #4,%sp
    rts
```

```
int fct() {
    int j;
    int i;
    j = 0;
    for(i=0; i<100; i++) {
        j = j*2 + i;
    }

    return j;
}
```

```
fct:
    move.l %d2,-(%sp)
    clr.l %d1
    clr.l %d0
.L7:
    add.l %d0,%d0                 J*2
    add.l %d1,%d0                 J*2+1
    addq.l #1,%d1
    moveq #100,%d2
    cmp.l %d1,%d2
    jne .L7
    move.l (%sp)+,%d2
    rts
```

Dans le flot d'instruction assembleur **m68k**, la première boucle commence à `.L2` et se termine à `jne .L2`. Le corps de la boucle contient une lecture et une écriture de la variable `j` (cf. commentaire). Dans la seconde fonction, le corps de la boucle est compris en `.L7` et `jne .L7`, il n'y a aucune lecture et écriture de la mémoire, la variable `j` est maintenue dans le registre `%d0`. En effet, le niveau d'optimisation du programme est important (la ligne de commande `m68k-linux-gnu-gcc-5 -C -O4 -S -fomit-frame-pointer fvol.c` génère le code assembleur avec un niveau d'optimisation de 4), le compilateur cherche à maintenir au maximum les données dans le microprocesseur.

La notion `volatile` est la même en Java. Elle est moins utile en C, puisqu'une variable `volatile` est très utile pour la gestion des périphériques. L'exemple précédent montre que lorsque le niveau d'optimisation est important, une variable peut être stockée dans un registre du microprocesseur. Ce qui peut être le cas lorsque la JVM optimise le code. L'ajout de l'attribut `volatile` oblige la JVM à lire et écrire systématiquement la variable. Comme l'optimisation du code Java est difficile à accéder, l'exemple est montré via un programme en C.

Le modificateur "volatile" peut être utilisé dans la programmation concurrentielle pour effectuer des contrôles d'intégrité des modifications. Un exemple très simple :

```
public class VolatileVar {

    private volatile  int stampId = 0;

    public void modification() {
        int j = ++stampId;
        stampId = j;

        // perform modifications
        if(j!=stampId) {
            throw new IllegalStateException();
        }
    }
}
```

Dans le corps de la méthode `modification`, le `stampId` est incrémenté et affecté à une variable interne à la méthode. Ensuite, les modifications sont effectuées. Si à l'issue des modifications, le `stampId` a changé, une exception est lancée. L'utilisation de `volatile` garantit que l'accès à la variable `stampId` ne sera pas optimisé. Lorsque la probabilité de collision est faible, cette stratégie évite le recours aux verrous système.

L'utilisation de l'attribut `volatile` n'a de sens que sur les types basiques primaires (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` et `double`). Ils n'ont pas vraiment d'utilité sur les objets sauf éventuellement pour accéder à des membres en accès direct (sans passer par l'appel d'une méthode).

L'autre alternative à l'utilisation des variables volatiles est l'utilisation des verrous système. Le système d'exploitation offre un moyen générique utilisant une instruction du microprocesseur qui garantit l'atomicité des modifications. C'est la base de contrôle d'accès mis en place dans la programmation concurrente. Rien n'empêche d'utiliser cette instruction microprocesseur directement dans le programme, mais cela suppose d'avoir une implémentation par type d'architecture de machine.

Le modificateur diamétralement opposé à `volatile` est `final`. Il indique que la variable ne peut être affectée qu'une seule fois. Lors d'un passage de paramètre, ce modificateur indique que le paramètre ne pourra pas être modifié.

En renseignant correctement les variables par rapport à ces deux modificateurs, la JVM peut effectuer des optimisations sur la manière d'accéder une variable. Cela reste limité, en effet, le contenu d'un objet, même s'il est déclaré « final » peut être modifié en appelant des méthodes de l'objet. Une stratégie consiste à créer un objet en lecture seule, ce qui peut être réalisé en rajoutant des contraintes d'utilisation sur les informations, mais n'apportera aucune aide à la JVM. L'immutabilité d'un objet n'existe pas réellement, contrairement au C++ où l'on peut utiliser le mot-clé `const`. Par exemple, le code suivant ne pourra pas être compilé :

```
class ConcurrentAccess {
private:
    int a;
public:
    int & value() {return a;}
};

void userClass(ConcurrentAccess const * a) {
    a->value() = 2; // compilation error
}
```

En effet, l'instruction `a->value()` référence une variable interne déclarée avec `const`. Tous les appels de méthodes deviennent alors `const`. Ce genre de fonctionnalité n'existe pas en Java :

```
public class FinalParam {
    private int a;

    public void setA(int a) {
        this.a = a;
```