CHAPITRE 1

FOURNIR À L'UTILISATEUR
UNE MACHINE VIRTUELLE
ET
UN ENVIRONNEMENT D'EXÉCUTION DE PROGRAMMES

Plan

Manuel 1

Machine virtuelle et environnement utilisateur

1. Environnement de travail adapté au besoin de l'utilisateur

On attend du système d'exploitation qu'il fournisse à l'utilisateur de l'ordinateur un cadre de travail adéquat. En particulier lorsque l'utilisateur veut écrire et exécuter des programmes, ce cadre doit comprendre une image simplifiée de l'architecture matérielle, qu'on appelle une architecture virtuelle, et un environnement d'exécution qui lui apporte les principales fonctionnalités complémentaires qui lui sont nécessaires.

Le rôle du système est de fournir à chaque utilisateur une machine virtuelle.

1.1. Fonctions de l'environnement d'exécution de la machine virtuelle

L'environnement de travail doit faciliter la mise en place et le contrôle des informations à traiter, la communication et de l'exécution.

Cela comprend au moins une interface de gestion de l'information, une interface de gestion des communications et une interface de contrôle d'exécution.

1.1.1. Fonction de gestion de l'information

Il s'agit de fournir une aide pour la structuration des programmes, des données, des objets de l'utilisateur et pour leur conservation dans des fichiers et pour la structuration de catalogues. Cette fonction comporte la désignation de ces objets pour les repérer lors de leur appel, donc la création et la gestion d'espaces d'adressage, de catalogues de fichiers ou de serveurs de noms d'objets. Elle fournit les interfaces pour le partage et le contrôle d'accès aux objets partagés.

1.1.2. Fonction de gestion des communications

Il s'agit de permettre l'accès à d'autres utilisateurs ou plus exactement à leurs machines virtuelles, qu'elles soient locales, distantes ou mobiles, à des supports externes, via des entrées-sorties, à des objets, des composants partageables, des services, des fichiers qu'ils soient locaux, distants ou mobiles, à des groupes de diffusion ou de coopération.

1.1.3. Fonction de contrôle d'exécution

Il faut permettre de lancer et de suivre des exécutions de programmes en séquence, en parallèle, en concurrence, de composer et d'enchaîner des programmes, de synchroniser des exécutions. On doit y trouver des aides pour la mise au point, pour le traitement des défaillances, pour la pose et l'exploitation de points de reprises, pour la mesure du temps.

1.2. Niveau d'abstraction et visibilité de l'environnement

L'environnement de travail dépend du niveau d'abstraction où se place ce travail.

Si l'utilisateur se situe dans une application particulière, c'est celle-ci qui définit l'environnement utilisable et l'interface homme machine (IHM)

Si l'utilisateur se place au niveau des fichiers et du langage de commande, c'est le script écrit en ce langage qui manipule l'environnement. Les objets sont des fichiers de stockage ou des fichiers exécutables ou interprétables.

Un niveau important est celui du langage machine ou d'un langage, comme le C, qui donne accès aux conventions de l'architecture et de la machine virtuelles.

Quand l'utilisateur écrit des programmes dans un langage évolué comme Ada, Java, Modula, C++, il utilise l'environnement qui est défini par le langage.

D'autres niveaux, comme ceux des plates-formes Corba, Java, Dcom, définissent aussi des environnements pour gérer des objets locaux ou distants (bus logiciel et appels de méthodes standard)

La gestion de l'environnement peut être explicitement faite par l'utilisateur ou son programme. Cela passe par des appels aux primitives systèmes.

La gestion de l'environnement peut être implicite avec mise en place automatique par un compilateur ou son moteur d'exécution ("run time"), par un interpréteur ou par le système. Cette gestion suit des conventions prédéfinies au niveau du langage de programmation ou de commande (appel de bibliothèque standard dont les programmes contiennent les appels aux primitives système).

2. Environnement primitif de la famille Posix, Unix, Linux

2.1. Notion de processus

Le processus est le concept de base de la concurrence qui sert à repérer l'exécution de la suite des actions définie par un programme. C'est donc tout à la fois :

- la forme prise par l'instance d'un programme pendant son exécution dans la machine,
- le déroulement temporel de cette instance dans une abstraction de cette machine, appelée machine virtuelle,
- l'objet système qui repère cette évolution, objet actif dont l'état évolue dans le temps et qui est caractérisé par un état, un contexte et un espace d'adressage.

Un processus hérite de deux constructeurs :

- le programme source qui définit le besoin particulier de l'utilisateur,
- le système d'exploitation hôte qui met en place une machine virtuelle générique.

2.2. Chaîne de production de processus

La chaîne de construction de programme prépare le processus selon le besoin de l'utilisateur et en tenant compte des conventions du système hôte. Il y a toujours une étape de la construction qui est conservée dans les fichiers de l'utilisateur et qui aboutit à un fichier exécutable. Cet exécutable comprend le programme découpé en entités logiques qui ont des propriétés différentes : code invariant, données constantes, données modifiables, table des symboles permettant des liens vers des modules externes,...Quand l'exécution du fichier exécutable du programme est lancé, il faut le transformer en processus. Il est alors "chargé" par le système d'exploitation. Celui-ci lui applique les transformations finales qui vont permettre de suivre son exécution et de gérer au mieux les ressources mémoire et processeur qu'il faudra lui attribuer pour que celle-ci se fasse.

2.3. La machine virtuelle associée au processus

Le "chargement" des différentes entités d'un programme respecte les conventions d'utilisation de l'espace d'adressage disponible avec le nombre de bits utilisés dans l'architecture (8 bits, 16 bits, 32 bits, 64 bits d'adressage). Cet espace d'adressage disponible est aussi appelé mémoire virtuelle. Il faut y placer les entités du processus, celles qui sont définies par le fichier exécutable du programme (les entités logiques du programme), et celles qui sont nécessaires à l'allocation de mémoire pour les données (données statiques non initialisées à la compilation, données dynamiques gérées en pile ou en tas). Le résultat est une "image" du processus, conservée dans un fichier du système. Le moment du chargement de cette image en mémoire centrale physique dépend de la nature du système d'exploitation (système avec chargement total initial ou à chargement à la demande).

L'espace d'adressage contient aussi une zone réservée au code et aux données du système. Dans cette zone, on distingue une partie utilisée par le système pour gérer le processus et une partie pour gérer l'ensemble des processus et les ressources partagées par eux (partie noyau). Cette partie noyau est commune à tous les processus, donc à tous leurs espaces d'adressages. On a alors un ensemble d'espaces d'adressages qui ont tous une racine commune. Tout processeur possède deux modes d'exécution au moins, le mode utilisateur et le mode système et seul le mode système permet d'atteindre la zone de l'espace d'adressage réservée au système. On contrôle ainsi l'utilisation de cet espace d'adressage en l'interdisant aux programmes des utilisateurs.

Chaque processus est repéré par le système par un certain nombre d'attributs dont un certain nombre figurent dans la structure appelée "descripteur de processus", observable en mode utilisateur.

2.4. Les primitives de création et de gestion de processus

Dans Unix, un utilisateur peut être associé à un ensemble de processus. (Dans IBM/VM ou VAX/VMS, au contraire, un utilisateur ne met en jeu qu'un processus).

Tout processus peut créer d'autres processus. Cela engendre une arborescence de processus. Tout le système repose sur ce concept d'arborescence, y compris les processus permanents, appelés aussi démons du système.

La création de processus se fait par la primitive "fork" qui crée un nouvel espace d'adressage et un nouveau processus fils qui est un clone du père. Le contexte du père est logiquement recopié dans celui du fils : code, données, fichiers ouverts, environnement, terminal, ... Seuls diffèrent le nom du processus et le nom de son père.

Pour particulariser le fils, on utilise une primitive de la famille "exec" qui permet de placer en mémoire virtuelle, dans la partie code de l'espace d'adressage, un nouveau code exécutable qui écrase l'ancien. Cette fois le contexte du processus ne change pas. La combinaison d'un "fork" suivi d'un "exec" chez le fils est l'équivalent du lancement d'un processus nouveau pour exécuter une fonction spécifique.

Un processus se termine quand il exécute la primitive "exit()". Cela passe une information d'état au système et place le processus dans l'état zombie tant que son père n'a pas enregistré cette terminaison.

Un processus attend la fin de son fils par une primitive “wait” qui lui permet de récupérer la variable d’état émise par ”exit”. Si un père se termine avant son fils, celui-cidevient orphelin et est adopté par le processus permanent “init”.

2.5. La famille des processus de l'utilisateur et du système

Ces primitives sont utilisées par le système pour vérifier le mot de passe d’un utilisateur et, en cas de succès, pour lancer l’interpréteur de commande (“shell”) particulier à cet utilisateur. Elles servent aussi à créer un processus pour exécuter une commande dans un espace d’adressage différent de celui de l’interpréteur de commande et pour procurer ainsi une certaine isolation entre les codes des commandes successives.

2.6. Ordonnancement et commutation de processus

Comme ce système encourage la création de processus, il comprend plus de processus que de processeurs et il faut ordonnancer l’allocation du (ou des) processeur(s) aux processus. Les processus qui attendent le processeurs sont des processus prêts qui sont gérés par priorités et à l’ancienneté quand plusieurs processus ont la même priorité. Les processus utilisateurs ont même priorité, mais ils sont gérés selon la méthode du tourniquet avec un quantum de temps de service. Un processus qui est bloqué par une primitive d’entrée-sortie ou par un ”wait” n’est pas mis dans les files des processus prêts.

Les processus passent par divers états que gère le système : “prêt” (ou éligible), “élu”, “bloqué”, “créé”, “zombie”. Dans un système qui déplace les processus par va et vient (“swap in, swap out”) avec une mémoire secondaire, les états “prêt” et “bloqué” ont un double, “prêt S” et “bloqué S”.

La liste des processus et leurs caractéristiques peuvent être lus en mode utilisateur par la commande “ps”.

L’allocation dynamique du processeur aux processus entraîne des commutations de contexte et un certain surcoût de traitement.

3. Processus et concurrence dans les langages de programmation

Il y a d’autres manières de créer des processus que par le “fork”. Les langages de programmation définissent un type particulier de procédure qui décrit le code du processus (type task en Ada) ou bien font hériter un objet ou une classe d’une classe processus prédéfinie (classe Thread en Java).

3.1. Langage servant à décrire les algorithmes présentés dans le cours

Les algorithmes concurrents présentés tout au long du cours sont écrits dans un pseudo langage qui utilise la lexicographie et la syntaxe introduites vers 1958 pour le langage Algol 60, et qui a été utilisée pour de nombreux langages comme Pascal, Simula (le premier langage a objets) et Ada. Cette syntaxe, très professionnelle, accompagnée d’un typage fort des données, n’a cessée d’être améliorée et fiabilisée. Sa lisibilité et sa clarté en font le meilleur outil pédagogique pour un langage impératif. Son utilisation est facilitée quand on peut utiliser un éditeur syntaxique (c’est le cas avec Ada)

On y trouve la structure de bloc, des paquetages (modules permettant d’encapsuler des objets et des types), des tâches (représentation des processus), la déclaration d’interface d’exportation et l’importation d’éléments externes (données, types, procédures, objets, paquetages).

3.2. Langage C avec des API Posix

Le langage C a été introduit en 1978, en même temps qu’Unix, et il dispose d’une syntaxe beaucoup plus artisanale, ce qui n’a pas empêché son succès. Ce succès vient aussi de son emploi avec les API pour les processus de la famille Unix/Posix/Linux. Il est utilisé aussi avec les notions de processus légers, ou activités, ou “threads”, qui ont été introduites pour apporter de la concurrence à l’intérieur d’un même espace d’adressage. La création de “thread”

est précédée par la définition de la fonction qui décrit le fonctionnement du “thread” et n’introduit pas de création d’un nouvel environnement comme avec “fork” et ”exec”. Tous les “threads” d’un même processus se partagent l’environnement du processus et ils doivent gérer explicitement les données partagées. Cette extension est utilisée dans l’architecture Posix (par exemple les systèmes répartis Chorus, Mach ou Amoeba) et dans l’architecture Windows NT.

Les API de gestion de concurrence apparaissent dans les bibliothèques du système.

3.3. Ada et Java

Ada et Java sont des langages modernes qui ont intégré complètement dans le langage la notion de processus et de gestion de concurrence. Elles ne s’exprime plus par des API de bibliothèque, mais par des clauses du langage, analysées par le compilateur. On exprime la concurrence à plus haut niveau. C’est le compilateur qui génère les instructions d’appel à un moteur d’exécution (“run time”), soit construit spécifiquement, soit obtenu par des appels à une bibliothèque système (qui fournit éventuellement les API Posix).

Ada déclare des objets de type tâche (qui jouent le rôle de processus) en séparant l’interface d’exportation et l’implantation (le corps) et les tâches sont lancées automatiquement à la fin de la partie déclaration dans laquelle elles sont instantiées. On a des objets actifs, déclarés comme tels, les tâches, et des objets passifs, utilisés par les tâches.

En Java il n’y a qu’une sorte d’objet, et les “threads” (qui jouent le rôle de processus) sont les objets qui héritent d’une classe standard, la classe Thread laquelle contient des méthodes spécifique, comme la méthode run. Une classe qui hérite de la classe Thread doit redéfinir la méthode “run” pour spécifier le code du “thread” qui sera créé. Le lancement du “thread” créé doit se faire par l’appel de la méthode “start” qui appartient à la classe Thread, tout comme la méthode “join” qui permet à un “thread” d’attendre le signal de fin d’exécution d’un autre “thread”. La fin d’attente peut être déclenchée par une interruption et cela conduit à utiliser la clause qui permet d‘attraper une exception : “try{...} catch(InterruptedException e){...}”. On retrouvera cette forme dans la gestion de concurrence quand il faudra réveiller des “threads” endormis.

L’absence de séparation entre objet actif et objet passif rend plus difficile de faire apparaître explicitement les entités actives. Cela conduit à définir une interface “Runnable” dans Java ou à définir des notions de “containers” dans les outils employant Java (comme les EJB ou “Entreprise Java Beans”).

4. Bilan de la notion de processus

Le processus est le concept qui représente une activité séquentielle en concurrence avec d’autres.

Les relations entre processus définissent leurs modes de création et de destruction, les pouvoirs qu’ils ont les uns sur les autres et les interactions mutuelles.

Les processus doivent être représentés dans le système d’exploitation qui gère leur exécution. Cette représentation doit permettre d’indiquer les ressources nécessaires et les modifications du contexte du processus pendant son évolution.

Un élément important du contexte est l’état du processus qui indique si le processus est demandeur ou non de ressources et en particulier d’un processeur.

La notion de processus apparaît sous divers vocables, comme task, process, thread.

La modélisation des processus concurrents se fait avec divers modèles comme la théorie des files d’attente, les réseaux de Petri ou les compteurs abstraits.

Cette notion de processus a évolué pour faire apparaître un grain d’activité plus fin, le processus léger qui introduit de la concurrence à l’intérieur d’un acteur doté d’un espace d’adressage. Elle devrait évoluer pour exprimer une séquence d’actions qui se déroulent sur divers sites d’un système réparti et qui sont mises en relation par des messages. Ces processus lourds ont été appelés messages activité dans Chorus.

Pour la suite du cours, on distingue pour le processus

- deux états intrinsèques qui dépendent explicitement des instructions de son programme : soit il est bloqué soit il est actif,
- deux sous-états de l’état actif, utiles pour l’ordonnancement du processeur : prêt et élu