République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur
et de la recherche Scientifique
Université Abou Bekr Belkaid Tlemcen
Faculté des sciences de L'ingénieur

Département d'Informatique

Thème

Synchronisation des proc... par thread Java

Présenté par :

* M^elle HADDADI Fatima Zohra
* M^elle ABDELDJELIL Hanane

Encadré par
* Mr BENAMMAR. A

Examiné par
* Mr KADRI .B

Année universitaire 2006 -2007

Remerciement

Louange à Dieu, qui nous a donné la force, le courage et l'espoir nécessaire pour accomplir ce travail et veiner l'ensemble des difficultés.

Ou il nous soit permis d'exprimer nos profondes gratitudes a Monsieur Benammar . A , notre superviseur pour ses conseils, son suivie durant notre projet.

Nous exprimons nos gratitudes,remerciements à Monsieur Kadri . B d'avoir accepté d'examiner notre projet .

Nous remercions aussi tous nos enseignants de promotion de LMD informatique .Et Mr Hadjila pour sa contribution.

Dédicace

A la mémoire de mon très chère père qui m'a appris ce qu'est vraiment la vie.

Je dédie ce travail spécialement à ma très chère mère qu'elle me présente la vraie amie, sœur c'est toute ma vie tout simplement parce qu'elle permise d'accepter tout pour que sa fille réussisse et je veut la dire « je t'aime ***mama*** très fort ».

De même degré je dédie à mes très chers frères ***Ilyes, kheireddine et Fadéla*** « merci pour tout ce que vous faisiez pour moi ».

- ***Ilyes*** qu'il m'a donné tous les sentiments noble et la tendresse d'un frère et le soutien d'un père qu'il est marqué leur présence dans mon cœur par les conseilles qu'il ma apporte, que dieu bénis son chemin et je dédie aussi à sa femme ***Zahira.***
- ***Kheireddine*** j'ai pas trouvé les mots qui traduire ce que je veut le dire, il m'a donné beaucoup de choses... , grâce à lui j'ai pouvait arrivé à faire ce travail .Dieu merci que j'ai pouvait réalisé un de ses rêves.
- Ma très chère sœur ***Fadéla*** qu'elle est toujours présente avec leur prière, pour m'encouragé, me donné l'espoir et les conseilles . Et je dédie à son mari ***Hakim.***

Ce travail est dédié à tous mes neveux ***(Samad, Nassima, Nadir,Ikrame,Amine,Oussama*** et surtout à ma petite nièce ***Kamar)***.

Et aussi à tous mes oncles, tantes et mes cousins.

De même façon, à tous mes amie spécialement mon binôme ***Hanane*** qu'elle partage avec tout les mauvaises et les bonne moments, et aussi à ma deuxième sœur ***Slhem, Djazia*** et mes amis de promotion.

Tous ceux là, et aux autres qui vont se reconnaître eux-mêmes, j'exprime ma profonde

Fatima Zohra

Dédicace

Je dédie ce modeste travail aux plus chers au monde :

A mes parents :

- Mon très cher père qui m'a donné pendant toutes mes années d'études le courage et la force.
- Ma très chère mère, en temoinage de l'amour, du respect et de gratitude que je lui porte.

Ainsi, je dédie ce travail à :

- Mes très chères sœurs : ***AICHA*** et ***AMEL***, a mon frère ***SID-AHMED*** et surtout a le petit ***YACINE***.
- Toute ma famille surtout ma grand-mère.
- Ma très chère sœur, amie et binôme : ***HADDADI FATIMA ZOHRA*** et sa petite famille surtout sa mère et sa petite nièce ***KAMAR.***
- Mes très chères amies : ***MERIEM, SIHEM, FATIMA ZOHRA, DJAZIA, SOUMIA*** et ***IMANE.***
- A toute la promotion 2006-2007 de ***LMD informatique*** surtout ***MERIEM.***

Tout ce qui nous aide de proche ou loin d'accomplir ce travail j'exprime mon profond remerciement.

Hanane

sommaire

****

Chapitre II. Implémentation et commentaires

Introduction générale

Un « système d'exploitation » est le logiciel le plus important de la machine il contrôle les ressources de l'ordinateur et fournit la base sur laquelle seront construits les programmes et les applications.

Deux modes de fonctionnement existent :

- Le mode noyau ou superviseur.
- Le mode utilisateur.

Tout système d'exploitation dépend étroitement de l'architecture de l'ordinateur sur lequel il fonctionne. Il existe cinq générations de système d'exploitation.

Dans ce mémoire on présentera d'une manière bref trois chapitres sur le thème « la synchronisation des processus par threads Java ».ce mémoire est organisé en trois chapitres :

Le premier chapitre sera une présentation du système d'exploitation en se basant sur la troisième génération et l'apparition de la multiprogrammation, suivie d'une petite introduction à l'exclusion mutuelle et la section critique.

Dans le deuxième chapitre on donnera des prévisions sur la synchronisation des processus avec les différentes solutions d'exclusion mutuelle proposés dans la littérature (sémaphore, moniteur, région critique et thread).

Dans le dernier chapitre nous détaillerons les threads Java.

Enfin, nous étudierons comme exemple de synchronisation le problème du producteur consommateur programmé en Java on utilisant JBuilder.

Chapitre I:

Présentation

1. Introduction :

Un système d'exploitation (SE) est un logiciel destiné à facilite et à simplifier l'utilisation d'un ordinateur.
Un système d'exploitation assurer l'interface entre le matériel et l'utilisateur en mettant à sa disposition tout un éventait de service le d déchargeant des spécificités d'accès complexes du matériel. **[1]**

2. Fonctionnalités d'un système d'exploitation :

Un système d'exploitation a pour but :
- de décharger le programmeur d'une tâche de programmation énorme et fastidieuse, et de lui permettre de se concentrer sur l'écriture de son application ;
- de protéger le système et ses usagers de fausses manipulations ;
- d'offrir une vue simple, uniforme et cohérente de la machine et de ses ressources.

On peut considérer un système d'exploitation de deux points de vue, représentés par le schéma ci-dessous,

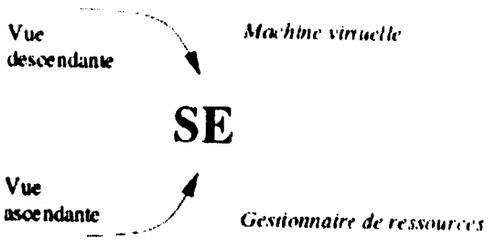


Schéma 1: **[2]**

La machine virtuelle fournit à l'utilisateur :
- une vue uniforme des entrées / sorties ;
- une mémoire virtuelle et partageable ;
- la gestion de fichiers et répertoires ;
- la gestion de droits d'accès, sécurité, et du traitement des erreurs ;
- la gestion de processus ;
- la gestion des communications interprocessus.

En tant que gestionnaire de ressources, le système d'exploitation doit permettre :
- d'assurer le bon fonctionnement des ressources et le respect des délais ;
- l'identification de l'usager d'une ressource ;
- le control des accès aux ressources ;
- l'interruption d'une utilisation de ressource ;
- la gestion des erreurs ;
- l'évitement des conflits.

3. Place du système d'exploitation dans l'ordinateur :

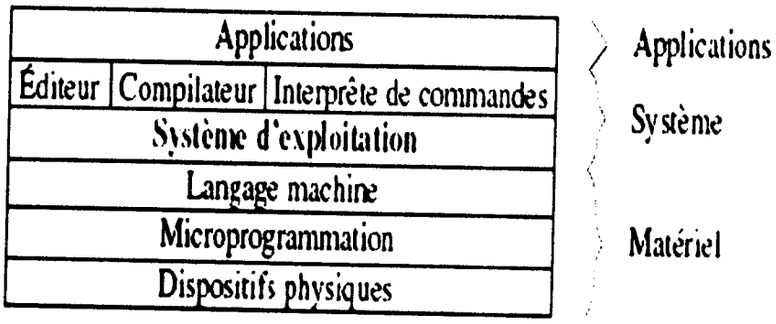


Schéma 2 : **[2]**

Ne sont pas des systèmes d'exploitation :
- l'interprète de commandes ;
- le système de fenêtrage ;
- les utilitaires (cp, chmod, uptime, . . .) ;
- le compilateur (ni sa bibliothèque) ;
- l'éditeur . . .

En fait, tous ces besoin d'un accès privilégié au matériel .en revanche, le système d'exploitation fonctionne typiquement en mode privilégié, pour pouvoir accéder à toutes les fonctionnalités du processeur .Ainsi, le système d'exploitation est protégé par le matériel contre les erreurs de manipulation (mais il existe des système d'exploitation s'exécutant sur du matériel non protégé, comme par exemple le DOS sur les anciens IBM PC). **[2]**

4. Historique des systèmes d'exploitation :

4.1. La première génération (1945 - 1955) :

C'est l'apparition des premiers ordinateurs , à relais et à tubes à vide , programmés par tableaux de connecteurs, puis par cartes perforées au début des années 50 .c'est aussi l'apparition du terme « bug » . **[2]**
Il n'existait pas de système d'exploitation.
- Les utilisateurs travaillaient chacun leur tour sur l'ordinateur qui remplissait une salle entière.
- Ils étaient d'une très grande lenteur. **[3]**

4.2. La deuxième génération (1955 - 1965) :

L'apparition du transistor rendait les ordinateurs plus fiables. Ils pouvaient maintenant être vendus, et l'on vit apparaître pour la première fois la distinction entre constructeur, opérateur, programmeur, et utilisateur.
Les programmes s'exécutaient par lot («batch ») .un interprète de commandes sommaire permettait le chargement et l'exécution des programmes dans l'ordinateur .ainsi, avec le système FMS (« Fortan Monitor System »), on soumettait les travaux de la manière suivante :

$JOB
$FORTRAN
. . . Programme . . .
$LOAD
$RUN
. . . Données . . .
$END

Les gros ordinateurs disposaient typiquement de trios dérouleurs de bande ;
Un pour conserver la bande du système d'exploitation, un pour le programme à exécuter et ses données, et le dernier pour recueillir les données en sortie.
En amont et en aval se trouvaient deux calculateurs plus petits, chargés l'un de transcrire sur bande les cartes perforées apportées par le programmeur, et l'autre d'imprimer sur papier les résultats contenus sur les bandes de sortie de données.

Ces ensembles étaient servis par des opérateurs, dont le rôle était d'alimenter les ordinateurs en bandes, cartes, et papier.

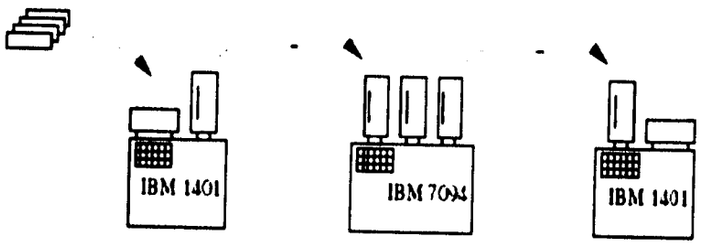


Schéma 3 : **[2]**

Comme la majeure partie du temps du calculateur principal était perdue lors des déplacements des opérateurs, un système de traitement par lot (« batch ») fut mis en place :
Plusieurs travaux (« jobs ») étaient collectés sur une même bande d'entrée, qui était changée une fois par heure et apportée au calculateur principal. Celui-ci lit le premier travail de la bande, et à la fin de chaque travail lit automatiquement le suivant, jusqu'à la fin de la bande.

4.3. La troisième génération (1965 - 1980) :

L'avancée technologique majeure de cette génération est l'apparition des circuits intégrés, qui ont permis de diminuer le rapport coût / performance de plusieurs ordres de grandeur.
La standardisation apportée par les circuits intégrés s'est également appliquée aux machines, avec l'apparition de familles de machines, qui partagent le même langage machine et le même système d'exploitation, pour des puissances et des applications très différentes.

Le problème majeur de cette approche était de pouvoir disposer d'un système d'exploitation efficace sur toutes les machines de la gamme, permettant d'utiliser toute la puissance des gros calculateurs, mais aussi capable de tenir dans la mémoire des plus petits.

Afin de rentabiliser l'utilisation des machines les travaux peuvent être stockés sur le disque de l'ordinateur dès leur arrivée en salle machine, sans passer par des ordinateurs annexes et des manipulations de bandes. Cette technique s'appelle le spoule (francisation de «spool », pour « Simultancouos Peripheral Operation On Line »).
De même, s'est développé l'usage de la multiprogrammation :
La mémoire physique était partitionnée en segments de tailles fixées à l'avance, dont chacune pouvait accueillir un programme différent. Ainsi, lorsqu'une tâche attendait la fin d'une entrée / sortie, le processeur pouvait basculer sur une autre tâche. C'est à ce moment qu'apparurent les premiers mécanismes matériels de contrôle des accès mémoire, pour protéger mutuellement les programmes contre les accès invalides.

L'inconvénient majeur du système de traitement par lots était que le programmeur ne pouvait plus déboguer son programme en temps réel, comme c'était le cas au tout début de l'informatique, lorsque chaque programmeur se voyait attribuer la machine pendant plusieurs heures d'affilée.
Pour retrouver cette interactivité a donc été développés les systèmes à temps partagé.
Le premier, CTSS (« Concurrent Time-Sharing System ») , crée au MIT en 1962 , a été suivi par Multics(« MULTiplexed Information and Computing Service ») , développé aux laboratoires Bell, puis par son descendant Unix, premier système d'exploitation écrit dans un langage de haut niveau , le C. **[2]**

Le développement de système d'exploitation durant les années de la troisième génération passe par les étapes suivantes :

En 1965 : Le MIT s'allie avec General Electric et les Bell Labs d'AT&T dans le projet MULTICS (Multiplexed Information and Computing Service) qui durera plusieurs années pour développer un prototype de nouvel ordinateur ainsi qu'un nouveau système d'exploitation temps partagé (time sharing).
Le MIT et Bell Labs avaient déjà une expérience dans le domaine avec CTSS (MIT Compatible Time-Sharing System) et BESYS. Pour résoudre le problème d'accès concurrent au ressources dans les systèmes d'exploitation multiprogramme, et exécuter les processus en exclusion mutuelle pour assurer la cohérence.

Le but du projet était de créer un système d'exploitation pour ordinateur parfaitement fiable, capable de tourner 24H sur 24, 7 jours sur 7, utilisable par plusieurs personnes à la fois et capable en même temps de faire tourner des calculs en tâche de fond.

- IBM System/360 par Gene Amdahl **1969:**

Le Bell Lab d'AT&T se retire du projet MULTICS, considérant que celui-ci prendrait trop de temps pour arriver à un résultat concret. Un groupe d'informaticiens menés par Ken Thompson et Dennis Ritchie avait commencé à réfléchir à la création d'un nouveau système d'exploitation temps partagé mais leur hiérarchie refusait d'en entendre parler.

- Ils trouvèrent un Dec PDP 7 (ordinateur apparu en 1964, évolution du PDP-1) inutilisé pour mettre leurs idées en pratique. Certaines idées furent héritées du projet MULTICS :

* notion de process
* système de fichiers arborescent
* interpréteur ligne de commande tournant comme un simple programme utilisateur
* représentation simple des fichiers texte et accès généralisé aux périphériques.

D'autres nouvelles idées servirent de principe pour le développement :

- Concevoir les outils comme un ensemble de petits programmes simples ;
- faire en sorte que le résultat d'un programme puisse devenir l'entrée du programme suivant :
- un noyau Unix primitif ;
- un shell ;
- quelques programmes utilitaires ;
- un éditeur et un assembleur furent rapidement mis au point sur le PDP 7.

Ce n'est que par la suite qu'un nom fût trouvé par Brian Kernighan pour ce nouveau système d'exploitation :

UNIX (par opposition au projet MULTICS). Cette version est connue sous le nom Unix Time-Sharing System V1.

1971 : De janvier à mars : portage du tout nouveau système d'exploitation UNIX sur PDP 11/20. Ken Thompson et Dennis Ritchie ont obtenu cette machine en prétextant le développement d'un logiciel de traitement de textes, les responsables du Bell Labs ne voulant plus entendre parler de systèmes d'exploitation suite à l'abandon du projet MULTICS.

Sur cette machine disposant de 24 Ko de mémoire, le noyau Unix occupait 16 Ko, 8 Ko restant disponibles pour les utilisateurs. Le disque dur avait une taille de 512 Ko et les fichiers une taille maximale de 64 Ko.

Le formateur de texte roff fût porté sur cette machine, ce qui permit à trois personnes du service des brevets d'utiliser effectivement la machine comme traitement de textes et ce en même temps que Thompson et Ritchie qui continuaient le développement d'applications.

Le succès de cette expérience a prouvé l'utilité d'UNIX et a rendu possible la poursuite du développement sur PDP 11/45.

Cette version est connue sous le nom UNIX Time-Sharing System V2.

1972: OS/VS1 et OS/VS2 d'IBM.

Le futur MVS **1973 :**
- Le noyau du système d'exploitation UNIX est entièrement réécrit en langage C par Ken Thompson. Au vu de la qualité du résultat, tous les autres outils utilisés sous Unix vont être réécrits en C. Cette version est connue sous le nom Unix Time-Sharing System V4. L'Université de Californie à Berkeley sera la première à disposer d'Unix (sources y compris) en dehors d'AT&T.
- Gary Kildall écrit le premier système d'exploitation pour micros : CP/M (Control Program for Microcomputers).
Il devint le système d'exploitation de prédilection pour les premiers micros ordinateurs à usage professionnel. Au milieu des années 70, il semblait devoir durer définitivement mais le choix d'un interpréteur Basic dans les premiers micros ordinateurs à usage personnel fit qu'il disparut rapidement de la scène.

1974 : Gary Kildall auteur du CP/M, et sa femme fondent Intergalactic Digital Research Inc. (renommé par la suite Digital Research Inc) dans le bus de commercialiser ce système d'exploitation pour micros.

1976 : La société IMSAI lance l'IMSAI 8080, basé sur le processeur Intel 8080 et utilisant le système d'exploitation CP/M. Cette machine contribua au grand succès de CP/M.

1977 : Apple fournit sur disquettes avec le Apple II, un système d'exploitation de disquettes assez rudimentaire mais efficace, le ProDOS.

1978 : Digital Equipment Corporation lance le VAX 11/780, premier ordinateur 32 bits de la longue lignée des VAX tournant sous le système d'exploitation VMS qui se prolonge jusqu'à nos jours. Cette nouvelle gamme de machines 32 bits représente aussi un gros progrès au niveau de cet excellent système d'exploitation écrit par Dave Cuttler.
Il s'agit d'un système multi-tâches, multi-utilisateurs à mémoire virtuelle avec une très bonne gestion de la sécurité et de l'allocation des ressources disque, mémoire et processeur entre les utilisateurs. Son interface en ligne de commande, son aide en ligne et son langage de commande (DCL) très simples et conviviaux facilitent grandement l'utilisation de la machine.

1979 : Apple met sur le marché en février 1979, la version 3.2 de son DOS, AppleDOS 3.2.

MVS/370 d'IBM **1980 :** - Microsoft annonce XENIX OS. En février 1980, Microsoft commence le développement d'un système d'exploitation portable pour les microprocesseurs à 16 bits de type Intel 8086, Zilog Z8000, Motorola M68000 et DEC PDP-11. Il sera multi usagers et multitâches. Ce sera sa propre version du UNIX de AT&T.
En août de la même année, elle en annonce la commercialisation. - IBM recherche un système d'exploitation pour son projet de micro ordinateur.
Ils pensèrent d'abord naturellement au CP/M de Digital Research, le plus répandu.
Gary Kildall n'étant pas la le jour du passage de l'équipe d'IBM (il faisait de l'avion).

En un mois de plus, l'éditeur EDLIN est développé. Il présentera QDOS sous le nom de 86-DOS en Septembre à Microsoft.
En Octobre, Microsoft, cherchant alors dans l'urgence un système d'exploitation pour micro ordinateurs pour satisfaire la demande d'IBM, achète pour 50000$, les droits de 86-DOS. - En France, conception de PROLOGUE : premier système d'exploitation multi utilisateurs sur micro-ordinateur.
Ce développement a été réalisé au sein de BULL MICRAL, filiale du Groupe BULL. **[4]**

4.4. La quatrième génération (1980 - 1990) :

Les ordinateurs personnels

Ils sont dus au développement des **circuits LSI** (Large Scale Integration) contenant des centaines de transistors au cm2.
Ils ont la **même architecture que les mini-ordinateurs** mais leur prix est beaucoup moins élevé.
Il existe deux systèmes d'exploitation principaux : **MS-DOS** (Microsoft Inc.) et **UNIX**.
MS-DOS intègre petit à petit des concepts riches d'UNIX et de MULTICS.
Dans le milieu des années 80, on voit l'**apparition de réseaux d'ordinateurs individuels** qui fonctionnent sous des systèmes d'exploitation en réseau ou des systèmes d'exploitation distribués.

4.5. La cinquième génération (1990 - ????) :

Les ordinateurs personnels portables et de poche **Apparition des PIC** (Personal Intelligent Communicator de chez Sony) et des PDA (Personal Digital Assistant, comme le Newton de chez Apple), grâce à l'intégration des composants et l'arrivée des systèmes d'exploitation de type « micro-noyau ».
Ils sont utiles pour les « **nomades** » et les systèmes de gestion des informations (recherche, navigation, communication).
Ils utilisent la **reconnaissance de caractère** (OCR) et les modes de communication synchrone et asynchrone (mode messagerie).
Très bon marché, ils sont capables de **se connecter à des ordinateurs distants et performants**.
Les **systèmes d'exploitation de type « micro-noyau »** sont modulaires (un module par fonction) ; ils peuvent être réalisés avec plus ou moins de modules et donc adaptables à des très petites machines (PDA et PIC).
Notion de multi-tâches et mono-utilisateur :
Le succès des ordinateurs de ces dernières années tient en partie au développement de systèmes d'exploitation plus conviviaux, plus simples à mettre en œuvre notamment grâce à l'utilisation du graphisme.
La complexité d'un système d'exploitation dépend du nombre d'utilisateurs de l'ordinateur. **[2]**

5. Mono utilisateur :

Sur un système mono-utilisateurs (un seul utilisateur à la fois) le système peut être assez simple si l'utilisateur commet des erreurs (il détruit tous ces fichiers par exemple), il n'aura à s'en prendre qu'à lui-même un ordinateur avec un système mono-utilisateurs peut cependant être utilisé successivement par différents personnes par maladresse ou malveillance, quelqu'un peut consulter ou détruire des informations personnelles.
Dans un système mono-tâche, l'ordinateur exécute un seul programme à la fois, c'est le cas de MS-DOS un des premiers systèmes d'exploitation des PC.
En cours d'édition d'un document par exemple, il est impossible de lancer l'exécution d'un autre programme. Il faut quitter l'éditeur, puis lancer le programme.
Dans un système mono-utilisateur multi-tâche il n'y qu'un seul utilisateur, mais l'utilisateur peut exécuté plusieurs tâches (programmes) se déroulant souvent dans différentes fenêtres. L'utilisation de la souris permet de passer d'une fenêtre à l'autre et donc d'une tâche à l'autre, on peut éditer un programme dans une fenêtre et exécuter un autre programme dans une autre une autre fenêtre on peut aussi en général récupérer des données d'une fenêtre pour les insérer dans une autre ce qui se traduit en terme simple par une copie (d'une fenêtre source dans une zone mémoire de l'ordinateur) et un collet (de la zone mémoire vers la fenêtre destination).

5.1. Processus :

Tous les ordinateurs modèles peuvent faire plusieurs choses en même temps.
Le CPU commute d'un programme à un autre en exécutant chaque un pour un quantum .il est évident qu'un instant données le CPU n'exécute qu'une seul tâche ou programme mais la vitesse a la quelle le CPU commute entre les tâches donne une illusion du parallélisme se qu'on appel le pseudo-parallélisme par contre un système parallélisme réel exécute en même temps un ensembles de tâches ce qui permet d'économisé le temps d'exécution en augmentant la difficulté de gestion.
Des ressources et des processus
Chaque tâche effectue par le système d'exploitation est réalisé à l'aide d'un processus. Un processus est juste un programme en cours d'exécution y' est compris la pile, le segment des données ainsi que les valeurs courantes du registres et les variables.

5.2. Problématique :

Pour protéger les ressources aux accès concurrent des processus afin d'assurer la cohérence pour illustrer le problème de deux processus qui exécutent en concurrence la partie « section critique » et donc conduit à une anomalie. Il est donc nécessaire que le système fournisse un support efficace pour synchroniser l'utilisation de ressources.
En faite tout problème de synchronisation se ramène au problème d'accès concurrent à une variable partagés (Rase Condition).
La solution conceptuelle naturelle à la résolution du problème d'accès concurrent consiste à interdire la modification des données partagés ie définir un mécanisme d'exclusion mutuelle sur des tranches spécifique des codes appelés « section critique ».

6. Exclusion mutuelle :

On parle d'***exclusion mutuelle*** à propos d'une ressource partageable à 1 point d'entrée (imprimante, fichier en écriture...).

L'exclusion mutuelle sert lorsque deux processus ne doivent pas faire la même chose au même moment. Deux utilisateurs par exemple ne peuvent pas imprimer leurs fichiers simultanément sur la même imprimante.

6.1. Programmation de l'exclusion mutuelle :

Une ressource non partageable (simultanément) est dite ***Ressource Critique***. Toute séquence qui l'utilise est dite ***Section Critique***.

6.1.1. La section critique :

- **Définition :**

La **section critique** d'un programme donné, est une suite d'instructions de ce programme dont l'exécution est gérée en **exclusion mutuelle** .Un processus parmi ceux qui s'exécutent en parallèle ne peut entrer en section critique si l'un des autres s'y trouve déjà. Intuitivement, un processus attend dans la section d'entrée qu'une certaine condition soit satisfaite pour entrer en section critique et signale sa sortie dans la section de sortie.

répéter

<section restante>
<section d'entrée>
<section critique>
<section de sortie>

jusqu'à faux;

Si dans certains systèmes il existe des dispositifs implantés au niveau matériel, qui assurent l'indivisibilité des instructions élémentaires, ce n'est pas le cas partout et surtout pas dans les systèmes répartis. Il existe des solutions logicielles (utilisation de variables partagées) pour pallier à ces manques matériel, par exemple : Soit deux processus **P1** et **P2** qui partagent une variable commune nommée ***Tour***, qui peut prendre les valeurs 1 ou 2. La *section d'entrée* consiste à consulter la valeur de ***Tour***. L'exécution de la *section critique* de **Pi** n'est autorisée que si ***Tour*** a la valeur ***i***. Lorsque la section critique est terminée, le processus exécute sa *section de sortie* en affectant à ***Tour*** le numéro de l'autre processus, permettant à celui-ci d'entrer dans sa propre *section critique*.

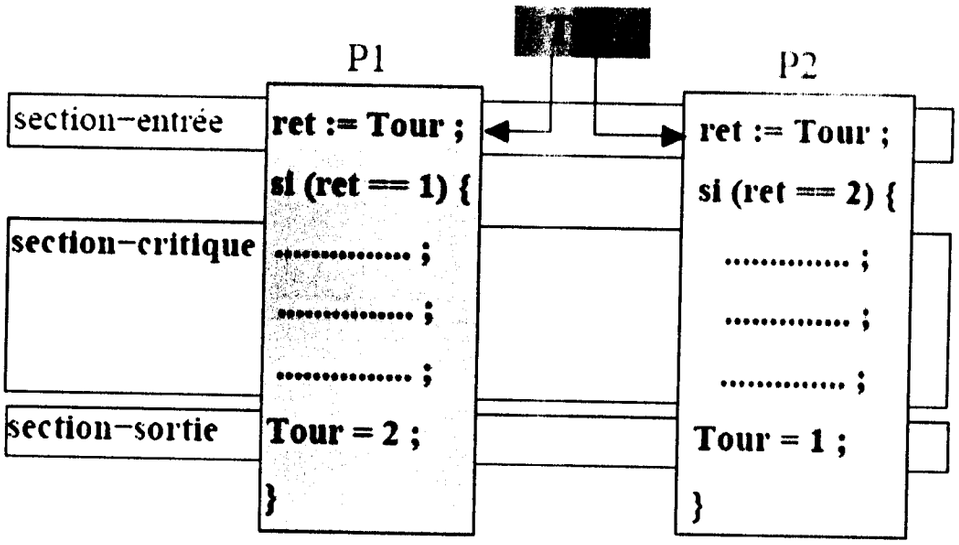


Schéma 4. [5]

Cette solution a des inconvénients en particulier si l'un des processus s'arrête définitivement, alors que la variable ***Tour*** n'a pas été changée.

Pour cela on impose une condition supplémentaire, appelée **condition de progression** :

Définition : un processus bloqué en **section restante** ne peut pas empêcher un autre processus d'entrer en **section critique.** D'autres algorithmes mettent en évidence que le problème de la *section critique* n'est pas simple à résoudre, surtout quand des conditions supplémentaires s'y greffent, comme la **progression** et l'**absence de blocage**.

Une dernière condition, appelée **attente bornée**, permet d'assurer une certaine équité entre les processus :

Définition : Lorsqu'un processus est en attente de sa section critique, il existe une borne supérieure au nombre de fois où d'autres processus exécutent leur section critique.

Cette condition interdite à certains processus d'exécuter itérativement leur section critique, en laissant un processus attendre, pendant une durée arbitrairement longue, l'entrée dans sa propre section critique.

6.2. Conditions de réalisation de l'exclusion mutuelle :

Exclusion	A tout instant, un processus au plus est en section critique
Accès	Si des processus demandent la section critique, et si la section critique est libre, l'un de ceux qui demandent doit y entrer au bout d'un temps fini.
Indépendance	Le blocage par cette section critique doit être indépendant des autres types de blocage : par exemple, si un processus est terminé, ou bloqué en attente d'une imprimante, il ne doit pas empêcher un autre processus d'utiliser un CD ROM.
Uniformité	Aucun processus ne doit jouer de rôle privilégié.

Ces règles doivent être satisfaites par les solutions envisagées pour l'exclusion mutuelle. La dernière n'est pas strictement indispensable, mais elle permet une programmation plus claire et systématique.

La mention pendant un temps fini est importante : elle indique que l'attente infÿÿie est évitée, sÿÿs donner la moindre ÿÿécision sur la durée de l'attente. Que le processus utilise la ressource pendant un dixième de seconde ou pendant trois jours importe peu ; les processus doivent fonctionner sans faire d'hypothèses sur les durées. **[5]**

7. Conclusion :

On a parler dans ce chapitre de l'évolution de système d'exploitation depuis **1945** et on a remarquer que dans la troisième génération : une caractéristique très importante d'un système d'exploitation multiprogramme concerne l'accès aux données et ressources qui doivent être contrôler par le système d'exploitation donc , ils ont développés plusieurs méthodes pour traiter cette problématique tel que (notion de verrouillage , sémaphore , moniteur . . .etc), et ce qu'on verrons dans le chapitre suivant.

// * Les références * // :

[1] : cours du système d'exploitation de troisième année informatique LMD.

[2] : http// : metalab.unc.edu

[3] : Cours Systèmes d'exploitation, François Bourdon, IUT de Caen, département de informatique.

[4] : http // : www.tt-hardwar.com

[5] : http// : www.dil.univ-mrs.fr

Chapitre II :

Synchronisation des processus

1. introduction :

Lorsqu'une ressource est critique (un seul point d'entrée) on parle d'Exclusion mutuelle. On appelle section critique la partie d'un programme ou la ressource est seulement accessible par le processus en cours. Il faut s'assurer que deux processus n'entrent jamais en même temps en section critique sur une même ressource. A ce sujet, aucune hypothèse ne doit être faite sur les vitesses relatives des processus. L'exclusion mutuelle sert donc lorsque deux processus ne doivent pas faire la même chose au même moment. Elle doit garantir l'entrée en section critique par un processus au plus. Pour que l'exclusion mutuelle soit réalisée. **[1]**

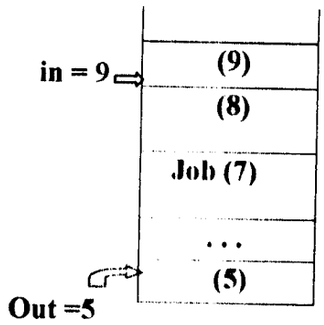
2. la synchronisation :

Pour protéger les ressources partagés aux accès concurrent afin d'assurer la cohérence pour utiliser le problème de partage de donnée on va étudier le problème de sérialisation du travaux d'impression, les tâches à imprimé sont stockées dans le répertoire de stol, tel que chaque tâche reçoit un numéro qui définis son ordre d'exécution, pour administrer ce répertoire deux variables partagés **in** et **out** qui contient respectivement le numéro de tâche à attribué et le numéro de tâche à imprimé.

Variable global ⟹ **int in, out;**
{ local_in = in

Placer_job (local_in)

Interruption ⟹ **in = local_in +1 ;}**



Le problème apparus quand deux processus exécutent en concurrence cette partie du code en cas où l'un des deux processus est interrompu juste après la rupture de la variable **global in** donc un autre processus va lire cette même valeur donc il aura deux tâches avec le **même** numéro d'ordre **local_in** ce qui conduit à une anomalie dans l'impression .Il est donc nécessaire que le système fournisse un support sur une efficace pour synchroniser l'utilisation de la variable partage **in.** En fait, tout problème de synchronisation se ramène au problème d'accès concurrent à une variable partagée (Race Conditions). **[2]**

2.1. Les mécanisme de synchronisation :

Les mécanismes de synchronisation sont utilisés pour résoudre les problèmes de S.C et plus généralement pour bloquer et débloquer des processus suivant certaines conditions.

Les verrous : permettent de bloquer tout ou une partie d'un fichier. Ces blocages peuvent être réalisés soit pour les opérations de lecture, soit d'écriture, soit pour les deux.

Les sémaphores : sont un mécanisme plus général, ils ne sont pas associés à un type particulier de ressource et permettent de limiter l'accès concurrent à une section critique à un certain nombre de processus.

Les signaux : sont à l'origine destinés à tuer (terminer) un processus dans certaines conditions, par exemple le signal SIGSEGV tue un processus qui effectue un accès à une zone de mémoire qu'il n'a pas alloué. Les signaux peuvent cependant être déroutés vers d'autres fonctions. Le blocage d'un processus se fait alors en demandant l'attente de l'arrivée d'un signal et le déblocage consiste à envoyer un message au processus.

Le problème des mécanismes de synchronisation est que les processus ne sont bloqués que s'ils les utilisent. De plus, leur utilisation est difficile et entraîne des problèmes d'interblocage (tous les processus sont bloqués). **[3]**

2.2. Synchronisation des processus :

C'est dans le cadre de processus partageant une même zone mémoire que le mot << concurrence >> prend tout son sens : les divers processus impliqués sont en concurrence pour l'accès à cette unique ressource qu'est la mémoire. Au problème du partage des ressources vient s'ajouter celui du manque de maîtrise de l'alternance et du temps d'exécution des processus concurrents.

Le système qui gère l'ensemble des processus peut à tout moment interrompre un calcul en cours. Ainsi lorsque deux processus coopèrent, ils doivent pouvoir garantir l'intégrité des manipulations de certaines données partagées. Pour cela, un processus doit pouvoir rester maître de ces données tant qu'il n'a pas achevé un calcul ou toute autre opération (par exemple, une acquisition sur un périphérique). Pour garantir l'exclusivité d'accès aux données à un seul processus, on met en oeuvre un mécanisme dit d'exclusion mutuelle. **[4]**

2.3. Les solutions d'exclusion mutuelle :

2.3.1. Masquage d'interruption :

Elle consiste à masqué (interdire) les interruption avant d'entrée dans la section critique se qui a assure qu'un seul processus est en section critique .mais cette solution ne satisfait pas la condition de section critique « un processus en section critique ne doit pas bloqué le reste du processus système » puisque un processus bloqué en section critique peut bloqué l'exécution des reste processus.

2.3.2. Variable de verrouillage :

Elle consiste à prévoir pour chaque section critique une variable de verrouillage (drapeau) qui mise **1** par le processus demandant la section critique juste avant de y entrées et mise à **0** après avoir sortir à cette section critique.

```
{
 L :while (verrou=1) { } ;
M(I) ;                          Masquage
d'interruption
If (verrou=1)
{
Goto L;
DMI ();                         démasquage
}
Verrou=1;
<section critique>
Verrou=0;
}
```

Une amélioration consiste à ajouter une instruction de masquage des interruptions pour partager le test de la variable verrou .ce qui fait de cette amélioration efficace pour l'exclusion mutuelle similaire aux instructions indivisible **test and set.**

2.3. 3. Variable d'alternance :

Le problème dans les solutions précédents est que les deux processus modifié la même variable au même temps pour éviter cela on peut mettre en œuvre une variable tour qui définit le processus qui a le droit d'entrée dans la section critique. Cette solution ne satisfait pas à la condition de section critique « un processus en dehors de la section critique ne doit pas bloquer autre processus en attente de cette section critique ».

2.3.4. Les instructions TAS :

Cette solution est basé sur un support matériel ie une instruction machine TAS qui charge le contenue d'un mot mémoire dans un registre de CPU puis il le teste ensuite il modifier ce mot mémoire. Les opérations de chargement, test, mise à jour ce son des instructions indivisible.

```
Int verrou ;

Entrer section :

    TAS reg, verrou ;

CMP reg, 0 ;

JNE entrer section ;

Ret

Quitter section :

Mov verrou, 0 ;

Ret ;
```

2.4. Bilans des algorithmes :

Si on analyse les algorithmes précédents, on remarque qu'il utilise une boucle while pour attendre la libération de la section critique ce qu'il veut dire que le processeur est modélisé pendant la durée une quantum de ce processus pour boucler dans le vide .on appel cette boucle une attente active qui cause :

- La monotonématique du bus d'adresse par les tests.
- Ralentissement des autres processus
- Coût additionnel dû à l'ordonnancement des processus attendant la même condition.

Pour résoudre ces inconvénient et éviter cette attente active en faisons dormir le processus attendant cette S.C jusqu'à ce qu'elle soit libérer on utilisant une file d'attente gérer par une politique FIFO si plusieurs sections critiques existent on utilisent une file d'attente pour chacune .La solution consiste à attacher une variable booléenne à chaque section critique qui indique son état. Le processus demandant cette section critique teste cette variable s'elle est vraie (libre). Il passe au S.C sinon il est bloqué il met en file d'attente jusqu'à ce qu'il soit réveillé (débloqué) par un autre processus sortant de cette S.C on utilisant par exemple deux primitives Sleep et Wakeup cette idée à donner lieu à la création des sémaphores. **[2]**

2.5. Synchronisation avec Sémaphore :

2.5.1. Définition :

La notion de sémaphore a été introduite dans les années 1968 par Dijkstra. On appelle sémaphore, le regroupement

- *d'une variable entière:* ***val(S)***
- *d'une file d'attente:* ***F(S)***
- *de 2 primitives* ***P*** *et* ***V*** *tels que:*

• procedure **P** {	• procédure **V** {
val(S)=1;	**V** incrémente Val(S) (Val(S) =Val(S) +1)
P décrément val(S) (val(S) =val(S)-1)	si (val(S) est inférieure ou égal à 0) alors
Si(val(S) est inférieur à 0) alors	on débloque un processus dans la file d'attente F(S)
le processus appelant la sémaphore est mis dans la file d'attente F(S)	**V**.état=prêt ;}
P.état=bloqué ;}	

Les sémaphores peuvent être utilisés pour résoudre le problème de l'exclusion mutuelle.

Avant d'entrer dans sa section critique, tout processus doit faire une demande explicite. Cette demande peut être bloquante (si la section critique est occupée). Intuitivement cette demande se traduit par l'exécution de l'opération P, seule opération bloquante sur les sémaphores.
Le processus sortant doit libérer la section critique permettant à un autre processus bloqué à l'entrée d'y accéder. Intuitivement cette libération se traduit par l'exécution de l'opération V, seule opération pouvant s'accompagner d'un déblocage de processus.

La valeur du compteur représente le nombre de tickets disponibles. Elle détermine par conséquent le nombre d'autorisations pouvant être accordées en même temps aux processus pour franchir cette barrière. Mais le nombre de processus autorisé de passer dans la section critique en même temps est égal à **1** (exclusion mutuelle). Donc, cette valeur ne devra jamais dépasser la valeur 1.
Il faut utiliser un sémaphore **(mutex)** initialisé à **1**

Les deux sections de ce protocole se définissent de la manière suivante : Prologue (section d'entrée) :
MUTEX.P ( ) ;
/*plus besoin d'une boucle pour faire attendre le processus. L'opération P bloque le progrès du processus si la section critique n'est pas libre*/

Epilogue (section d'entrée) :

MUTEX .V() ;
/*s'il y a un processus en attente alors il sera automatiquement débloqué*/

Pour établir la validité de cette solution, il faut montrer :

- Qu'à tout instant, un processus au plus se trouve dans la section critique
- Lorsqu'aucun processus ne se trouve dans la section critique, l'entrée en section critique se fait au bout d'un temps fini. [7]

2.5.2. Caractéristiques des sémaphores :

Un sémaphore S créer par une déclaration spécifier ça valeur initiale (le nombre de processus qui peuvent coexiste au même temps dans la S.C ;

- La politique de gestion de la file d'attente est FIFO pour qu'il n'aura pas un risque de famine.
- **P** et **V** sont les sous opérations applicable sur un sémaphore, un sémaphore ne peut donc être testé, ni recevoir une valeur sauf l'hors de son initialisation.
- Un processus bloqué par **P** ne peut être réveillé que par un autre processus exécutant **V** sur le sémaphore.
- Les opérations **P** et **V** sur un sémaphore S sont exécutant en exclusion mutuelle, pour effectuer ça on peut masquer les interruptions ou bien on utilise les instructions TAS ;
- Lorsque une opération **V** débloque un processus celui passe à l'état prêt et son ordonnancement est effectué sur Ordonnanceur système.
- Afin d'assurer la cohérence on doit équilibrés le nombres d'exécution de **P**(S) et **V**(S). [2]

Pour étudier la réalisation de l'exclusion mutuelle :

On va démontrer que le nombre de processus se trouvant en section critique à un instant donné est toujours inférieur ou égal à **1.**

Le nombre de processus se trouvant dans la section critique est donné par :

- nombre de processus ayant franchi le sémaphore
- nombre de processus ayant quitté la section critique

C'est à dire : **NF – NV**

Par ailleurs, nous savons que NF = min (NP, NV + 1) **/*compteur0==1*/**
Donc, **NF = NV + 1** ce qui implique **NF – NV = 1**
Par conséquent, l'exclusion mutuelle est réalisée.

Pour étudier la vérification de la condition de progression .On va effectuer un raisonnement par l'absurde :

- Si aucun processus ne se trouve dans la section critique alors on aurait :

  NF – NV == 0 c'est à dire NF == NV

- S'il existe de processus attendant derrière le sémaphore MUTEX, alors on aurait :

  NF < NP

Mais, **NF == min (NP, NV + 1).** Dans ce cas, **NF** serait égale à **NV + 1**

Donc, **(NF == NV + 1)** et **(NF == NV)** en même temps. D'où la contradiction. Par conséquent, la condition de progression est réalisée.

2.5.3. Inconvénients :

Les sémaphores sont souvent difficiles à utiliser car leur dissémination dans les programmes n'est pas aisée. Les programmes deviennent vite illisibles et difficiles à mettre au point. **[6]**

2.6. Les régions critiques conditionnelles :

2.6.1. Définition :

Le concept de région critique conditionnelle **[Bri 73] [Hoa 72]** permet de surmonter ces difficultés, en offrant une notation structurée pour la spécification de la synchronisation. Des variables partagées sont explicitement placées dans des groupes appelés ressources. Chaque variable peut se trouver dans au plus une ressource et peut être accédée uniquement au moyen d'instructions contenues dans une région critique conditionnelle **(CCR :Conditional Critical Region)** prenant le nom de la ressource . L'exclusion mutuelle est garantie par le fait que l'exécution de différentes instructions relatives à une même région ne peuvent s'entrelacer. Des conditions de synchronisation sont autorisées au moyen d'empressions booléennes, explicites dans des instructions **CCR.**Une ressource **R** contenant les variables **V1, V2, V3, . . ., Vn** est déclaré ainsi : Ressource **R : V1, V2,..., Vn ;** De cette façon, les variables associées à **R** ne peuvent être accédées par des instructions, que dans des **CCR** identifiées par **R.** de telles instructions ont la forme suivante :

region R when B do S end region :

- **P** entre dans la section critique et test **B** .
- Si **R** est vrai **P** exécute S en exclusion mutuelle et libère les processus bloqué puis quitte la région critique.
- Si **R** est faux **P** sort de la région critique et ce bloque jusqu'à ce que la condition réveille par un autre processus.

Avec **S :** une liste d'instructions .des variables locales aux processus exécutant peuvent également apparaître dans l'instruction **CCR.**

Edison :

dans le langage **Edison [Bri 82] ,P.Brinch Hansen** propose une simplification du concept de région critique conditionnelle , en arguant par ailleurs , que l'occasionnelle inefficience était, d'importance mineure sur un microprocesseur .étant entendu aussi , que l'on envisageait alors de fédérer plusieurs microprocesseurs .

Dans la proposition originale des régions critiques conditionnelles, des opérations sur des variables ressources (shared) différentes, comme dans l'exemple ci-dessus :

Region R1 when B1 do S1 end region

et

Region R2 when B2 do S2 end region

Peuvent se dérouler concurremment. En Edison par contre, les régions critiques conditionnelles prennent une forme simplifiée: **when B do SL end**

Où **SL** est une liste d'instructions. L'exécution de toutes les constructions when s'effectue strictement une à la fois .si plusieurs processus nécessitent l'évaluation ou la réévaluation concurrente des conditions d'ordonnancements, celle-ci est en fait réalisée une à la fois, selon un ordre équitable, par exemple cyclique.

Dans des système bien conçus on a pu établir que chaque processus opère la plupart du temps sur ses variables propres (locales) et uniquement une faible fraction de son temps d'exécution est consacrée à l'échange de données avec d'autre processus. L'additionnelle autoconsommation nécessaire à l'évaluation des expressions est donc acceptable, pratiquement **[Bri 82]. [7]**

Region R do S await B : le processus P exécute S de façon inconditionnelle puis ce bloque jusqu'à ce que la condition soit vrai. Les régions critiques sont implémenter on utilisant les sémaphores par exemple Région **R** do S peut être implémenter on associons à chaque variables partagé **V** une sémaphore Mutex initialisé à1.

P(Mutex) S **V**(Mutex) **Région V when b do S** peut être implémenté comme suite : Var Mutex, S : sémaphore ; np: entier; Mutex=1; S=0; np=0; ***A***:**P** (Mutex) Si b=vrai alors Début S ; Pour i :=1 à np faire **V**(S) ;	np=0 ; **V** (Mutex) ; Fpour Sinon Début Np++ ; **V** (Mutex) ; **P**(s) ; Aller à ***A*** ; Fin Fsi.

. [2]

2.6.2. Remarques et commentaires :

Bien que les régions critiques conditionnelles disposent de nombreux éléments positifs, elles restent coûteuses à implanter. En effet, les conditions dans des instructions CCR peuvent contenir des références à des variables locales obligeant ainsi chaque processus à évaluer ses propres conditions. Ce qui sur un système multiprogrammé entraîne de nombreux changements de contextes (context switching), parmi lesquels plusieurs inutiles, car le processus veillé peut trouver sa condition encore une fois fausse.Toutefois, si chaque processus est exécuté sur son propre processeur et est que la mémoire est partagée (et partiellement locale à chaque processeur), comme sur les systèmes multiprocesseurs, les instructions **CCR** peuvent être implantées à peu de frais par l'utilisation de l'attente active.

Il convient de relever que malgré qu'aucun des exemples présentés n'en ait contenu, des régions critiques conditionnelles peuvent néanmoins être emboîtées .cela n'est toutefois admis qu'avec des ressources distinctes .par ailleurs ,il faut prendre garde à éviter les inter blocages que l'imbrication peut induire . Des conditions toujours vraies, du genre :

region R when TRUE do . . . end region

Cette forme syntaxique peut naturellement être abrégée par la plus simple qui suit :

Region R do . . . end Region

Pour une introduction nous avons préféré la première, plus explicite. L'inspiration PORTAL, d'importer systématiquement tous les objets globaux dans les procédures ou processus n'a souvent pas été respectée.

2.7. Les moniteurs :

2.7.1. Définition:

Les moniteurs sont des structures qui regroupent des invariables partagées par plusieurs processus et les instructions qui les manipulent. Dans certains langages tel que : Concurrent Pascal, le moniteur est défini comme un type, dans la partie déclaration du programme. Les variables partagées sont donc encapsulées dans le moniteur. Ces données sont accédées par l'intermédiaire d'un ensemble de méthodes publiques qui opèrent sur ces données.

Un moniteur présente un ensemble d'opérations définies par le programmeur à qui l'on fournit l'exclusion mutuelle à l'intérieur du moniteur. Ce concept a été propose par **C.A.R. HOARE** en **1974** et **P.BRINCH-HANSEN** en **1975.**

2.7.2.Propriétés de moniteurs :

Une procédure définie à l'intérieur d'un moniteur ne peut accéder qu'aux variables qui sont déclarées localement à l'intérieur du moniteur et à tous les paramètres formels qui sont passés aux procédures.

La structure du moniteur ne permet qu'à un seul processus à la fois d'être actif à l'intérieur d'un moniteur. Par conséquent, le programmeur n'a pas besoin de coder explicitement la synchronisation ; elle est bâtie dans le type moniteur.

Sa structure = procédure sans paramètres

A l'intérieur du moniteur, nous distinguons les différentes parties suivantes.

Déclaration des variables partagée qui ne sont pas accessibles en dehors du moniteur

Des procédures et des fonctions internes au moniteur. Elles sont les seules à manipuler des variables partagées. Leurs paramètres constituent le lien avec le programme.

2.7.3. Format de l'appel:

nom_du_moniteur. nom_de_la_procedure.

Tant que < le magasin_est_ouvert > faire

si < entrée > alors Nombre_de_clients.incrémenter

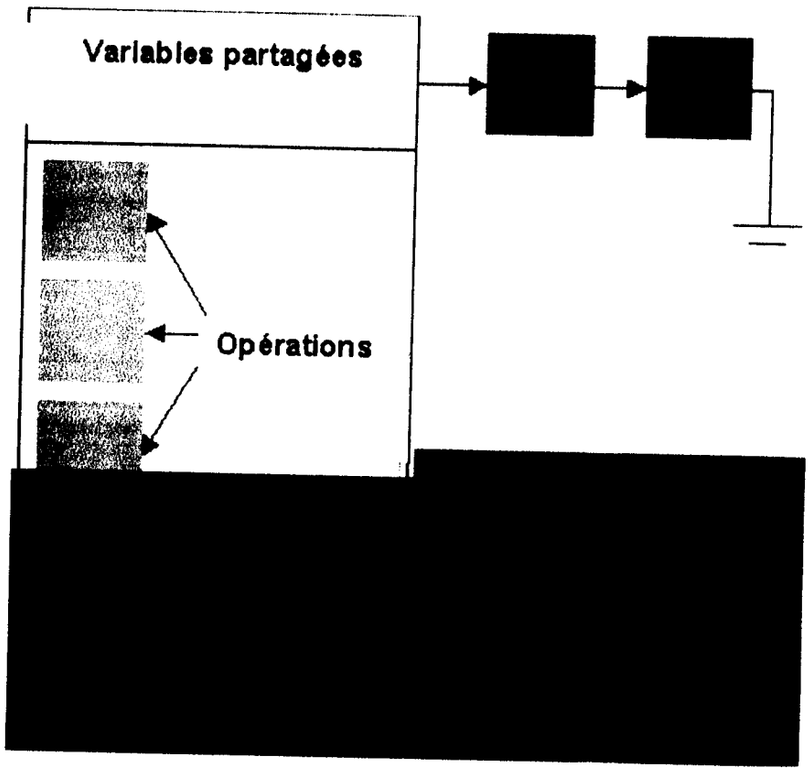
Les variables de type condition ont un rôle particulier dans les moniteurs par le biais des opérations spéciales qui sont invoquées sur elles (**wait et signal**). Un programmeur qui doit écrire son schéma de synchronisation sur mesure peut définir une ou plusieurs variables de type **condition :**

Condition x, y;
L'opération: **x.wait** signifie que le processus invoquant cette opération est suspendu jusqu'à ce qu'un autre processus invoque **x.signal**.

L'opération **x.signal** reprend l'exécution d'un processus. Si aucun processus n'est suspendu, alors l'opération signal n'a pas d'effet ; c'est à dire que l'état de x est maintenu comme si l'opération n'avait jamais été exécutée. Opposons cette opération à l'opération V sur les sémaphores qui en affecte toujours l'état.
- **signal-and-wait** - soit **P** attend que **Q** quitte le moniteur, soit il attend une autre condition ;
- **signal-and-continue** - soit **Q** attend que **P** quitte le moniteur, soit il attend une autre condition.
-x.Empty : Booléenne cette fonction retourne vraie si la file d'attente x est vide et faux sinon .
Il existe des arguments raisonnables pour choisir l'une ou l'autre des solutions. Puisque **P** s'exécute déjà dans le moniteur, **signal-and-continue** semble plus raisonnable.
Cependant, si nous autorisons le processus **P** à continuer, alors la condition logique pour laquelle **Q** attendait peut ne plus être valable au moment où **Q** reprendra son exécution.

signal-and-wait était défendu par Hoare, principalement du fait que l'argument précédent en sa faveur se traduisait directement en règles de preuves simples et élégantes.

// Schéma de moniteur //



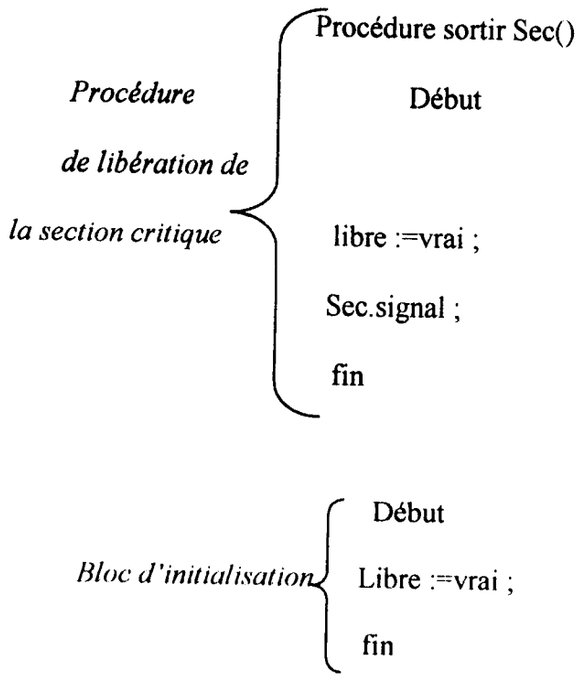
. [6]

Exemple d'utilisation des moniteurs : Moniteur Sect_crt ;Var Sec : Condition ;

Procédure entrer Sec ( )

Procédure d'entrer En Section Critique

```
Début
  Si (non libre) alors
    Sec.wait ;
  Libre :=faux ;
fin
```



Processus P
Sect.entrer
<S.C>
Sect.sortie . [2]

2.7.4. Avantages des moniteurs :

Les sections critiques sont transformées en fonctions ou procédures d'un moniteur. Elles ne sont plus dispersées .La gestion de ces sections n'est plus à la charge de l'utilisateur. Elle est réalisée par l'implantation du moniteur. Elle garantit qu'au plus 1 processus à la fois peut accéder à cette structure. Le moniteur tout entier est implanté comme une section critique.

Quand un processus désire exécuter une opération concernant une variable partagée, il appelle une procédure particulière d'un moniteur. Cette procédure sera exécutée seulement si le moniteur est disponible. Si au contraire le moniteur est occupé le BCP du processus est placé dans une file d'attente associée au moniteur.

Dès que ce dernier (le moniteur) est libéré, un processus est choisi de la file et la procédure invoquée et exécutée ; De cette façon, le problème de la section critique (l'exclusion Mutuelle) est réglé. Mais, pour obtenir la puissance d'expression de sémaphores et résoudre les problèmes de synchronisation il a fallu définir et introduire un nouveau type: Condition.

2.7.5. Implantation des moniteurs (Utilisant les sémaphores) :

Une implantation possible, qui assure l'usage du moniteur en exclusion mutuelle, consiste à associer à chaque moniteur, un sémaphore "Ex_Mut" initialisé à 1.

Comme c'est le cas d'une S.C., l'entrée dans le moniteur est précédée d'une opération **Ex_Mut.P()**; la sortie du moniteur est suivie de **Ex_Mut.V()**.
La file d'attente pour l'entrée dans le moniteur est celle du sémaphore Ex_Mut. Cependant,

Lorsqu'on choisit de faire attendre un processus ayant exécuté une opération **Signal** sur une variable condition il faut définir une file d'attente supplémentaire à l'intérieur du moniteur.

La file d'attente d'un autre sémaphore ("f") joue ce rôle. que processus après Un Signal sera placé en attente sur la file de ce sémaphore. La longueur de cette file est repérée par une variable entière : Longueur. Le compilateur remplace chaque procédure du moniteur par :

Ex_Mut.P () **corps de la procédure** **Si Longueur > 0 Alors {** **f.V ();** **Longueur --;** **}** **Sinon Ex_Mut.V ();** De cette façon, lorsqu'un processus sort du moniteur, il réveille en priorité un processus de la file du sémaphore file. Si cette file est vide, il réveille un des processus en attente à l'entrée sur la file du sémaphore **1**	Ex_Mut. L'implantation des variables de type condition se fait de manière analogue à l'aide des sémaphores et des variables entières. Ces dernières représentent la longueur de la file d'attente du sémaphore. Le compilateur remplace donc chaque procédure du moniteur par : Ex_Mut.P() **corps de la procédure** **Si Longueur > 0 Alors {** **f.V() ;** **Longueur -- ;** **}** **2**
Sinon Ex_Mut.V() ; Ex_Mut est un sémaphore initialisé à **1.** f est un sémaphore initialisé à **0.** Toute variable de type condition sera implantée en utilisant un sémaphore initialisé à 0. Ainsi, Si on considère par exemple la déclaration suivante: **Condition C ;** L'opération C.Signal sera remplacée par le compilateur par la séquence suivante : **Si ! (C.Vide()) Alors {** **C.V() ;** **f.P() ;** **}** **3**	Si la file de C n'est pas vide alors on en réveille un processus ; le processus appelant se bloque dans la file f L'opération C.Wait sera, quant à elle, remplacé par le compilateur par la séquence suivante : **Si !(f.Vide()) f.V() ;** **Sinon Ex_Mut.V() ;** **C.Wait ;** **4**

Cette opération permet de libérer le moniteur et bloquer le processus appelant. [7]

2.7.6. Les inconvénient des moniteurs :

- Il nécessite un langage de moniteur ou il sont définies ex (concurrent pascal).
- Un signal de réveille non attendue est perdu.
- Le cas ou plusieurs processus attendre le même signal n'est pas traité.
- Il ne permettre pas l'échange d'information entre les processus.
- Il son inutilisable dans le système distribuer. [2]

2.8. Les threads :

2.8.1. Définition :

Tout le monde a entendu parler de système multi-tâches. Un tel système est capable d'exécuter plusieurs programmes en parallèle sur une même machine. Mais attention : dans la quasi totalité des cas il n'y a jamais deux programmes différents qui s'exécutent au même instant. La raison en est simple : la plupart du temps, une machine n'a qu'un seul processeur et ce dernier n'est capable de réaliser qu'une seule chose à la fois. Mais alors, comment se fait t'il qu'on arrive malgré tout à exécuter plusieurs applications en même temps.

La plupart des systèmes d'exploitation sont équipés d'un Ordonnanceur de tâches. Ce composant logiciel a pour mission de donner à tour de rôle le processeur aux programmes (on dit au processus) en cours d'exécution. La durée pendant laquelle un processus est actif est en fait très courte et c'est grâce à cette activation cyclique et brève que l'utilisateur a l'impression que plusieurs choses sont en cours d'exécution sur sa machine.

Bien souvent le problème est encore plus complexe. En effet, sur les systèmes récents, le processus est en fait inactif : il permet juste de partager les données d'une application. Et bien, pour les threads , En fait, une application qui s'exécute est représentée par un processus et par au moins un thread (un processus pouvant contenir plusieurs threads). La meilleure traduction française de thread me semble être une tâche.

Un programme est souvent conçu comme une suite (séquentiel) d'instructions, il est possible de concevoir des programmes ou plusieurs tâches se déroulent simultanément, en parallèle ; ces différentes tâches portent le nom de thread et on dit alors que l'application est multithread.

Ce parallélisme ne pourra être effectif que si la machine utilisée est multiprocesseur si ce n'est pas le cas, les différent tâches se partagent successivement le processeur nous supposerons que la machine utilisé est monoprocesseur pour obtenir des applications multithread.

On démarre en générale avec une méthode main ,un premier thread qui peut en lancer d'autre sans s'interrompre pour autant , les nouveaux threads peuvent à leur tour démarrer d'autre threads un thread est quelque fois aussi appelé processus léger ou file d'exécution , un programme multithread est aussi dit concurrent .lorsque l'application est multithread , Il faut suivant synchroniser les accès aux données partagées de façon à conserver leurs cohérence . par ailleurs ,les différent threads doivent pouvoir se coordonner ,s'attendre ,s'alterner,être interrompus définitivement ,être interrompus provisoirement et alors éventuellement être repris . . . etc. **[8]**

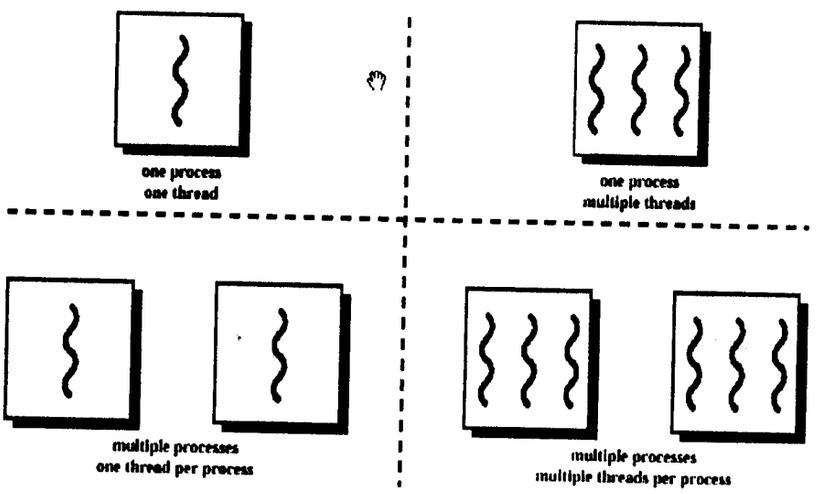


Schéma : **[5]**

2.8.2. Les avantages du multi thread :

* Cela dépend essentiellement du matériel avec lequel le programme est exécuté.
Sur certaines architectures telles que les stations multi processeur, le multi thread permet d'exécuter physiquement plusieurs opérations indépendantes en parallèle au même moment.
* Sur une machine dotée de deux processeurs, une application threadée effectuant des calculs intensifs, sera exécutée presque deux fois plus rapidement.
* Le gain de performance car la gestion du multi thread impose une certaine quantité de code indispensable.

2. 8.3. Caractéristique des threads :

- Le thread partage le même espace d'adressage ce qui signifie simplicités de partage des données et communication et ressources.

- La création des threads est plus rapide que la création des processus (la rapidité due de la non duplication de l'espace d'adressage).
- Une commutation des processus ça due aux les valeurs des registres ne sont pas commuter.
- Amélioration des performances en cas de blocage.

On comparons la programmation multi processeur et multi-thread il se voie que la programmation multi-thread est plus efficace mais la gestion de partage du donnée est de parallélisme elle est faite par le programmeur.

Problématique :

Le partage des données : des variables sont partagées par tous les threads donc elles doivent être protéger contre les accès concurrent on utilisons les sémaphores.
La coopération entre les threads : un mécanisme de synchronisation doit être trouvé pour gérer la coopération. La gestion des erreurs :il y a des signaux qui sont destinées au processus donc il faut décéder lequel entre les threads va traiter ce signal.

2. 8.4. Les états d'un thread :

Ils sont les même que les états des processus c'est-à-dire prêt, bloquer et active.

2. 8.5. Conseils pour la programmation :

Chaque variable globale devient une section critique donc il faut la protéger.

Ne pas créer plusieurs threads parce que sa diminuée les performances et augmente la complexité de leurs gestions et synchronisation.

2.8.6. Les threads UNIX :

Les processus sont des entités à priori indépendantes .chaque processus dispose de ses données et de sa pile d'exécution et exécute un code non modifiable qui peut être partagé .si on veut que les processus puisent communiquer ou partager des informations il faut le demander explicitement à l'aide d'un des moyen (tubes, file de message, mémoires partagées).Une autre façon de procéder est de créer des threads encore appelés processus légers ou fils d'exécution .LINUX implémente la norme Posix.1c appelée Pthreads (Posix threads) sous forme d'une bibliothèque (LINUX Threads).les threads partagent le même espace d'adressage . Chaque thread peut donc accéder aux variables globales du programme qui le crée. Cependant, les différents threads crées se déroulement concurremment et on retrouve, le problème d'accès concurrent à une variable partagée entre plusieurs threads.

La commutation d'un thread à l'autre est en général moins coûteuse en temps CPU que la commutation de processus (crées avec fork ( )).la norme Posix peut être implémentée de diverses façon suivant le S.E.

2.8.7. Les fonction concernant les threads en UNIX :

Un thread est identifié par une variable de type pthread-t (un entier long positif sous LINUX).

2.8.8 Les operations appliqué aux threads:

- **Creation d'un thread pthread-create:**Le thread crée s'exécute concurremment avec le thread qui l'a crée.

Fin de thread : pthread-exit : cette fonction termine le thread en cours d'exécution, une valeur de retour peut communiquer et récupérée par un autre thread par l'intermédiaire de la fonction pthread-join ().

- **Attente la fin d'un thread :**La fonction pthread-join ( ) permet au thread concurrent d'attendre la fin d'un autre thread et de récupérer la valeur communiquer lors de pthread-exit ( ).
- **Détacher un thread :**récupérer les ressources d'un thread dès sa fin (pile et variables privés).on peut pas attendre un thread détaché. Il ne peut donc pas communiquer sa valeur de retour à l'aide de pthread-join ( ) pthread-detach ( ).
- **Fournir son identifiant:**pthread + pthread-self ( );
- **Détruire un thread :** pthread-concel

2.8.9. Principe de moniteur:

Un moniteur s'apparente à un objet qui encapsule les données partagées par plusieurs threads (programmation multi-thread) et définit des méthodes (fonctions) de traitement de ces variables partagées : accès ou modification. Les méthodes d'un moniteur sont en exclusion mutuelle .une seule méthode s'exécute à la fois, ce qui garantie l'intégrité des données.

les threads utilisateurs de l'objet moniteurs ne peuvent pas accéder directement aux données du moniteur .comme en programmation objet, les threads doivent utiliser exclusivement les méthodes du moniteur.

Il faut pourtant pouvoir synchroniser les threads ie arrêter momentanément l'exécution d'un thread. Ceci se réalise à l'aide de variables de type condition sorte de variables s'apparentant plutôt à une liste de threads bloqués sur cette condition .la variable condition est exclusivement prise en compte par deux fonctions.

- une fonction wait ( ) : sur la condition qui à pour effet de bloquer le thread effectuant cette opération sur la condition du moniteur (ajout du thread courant dans la liste des threads en attente sur cette condition).
- Une fonction signal ( ) : sur la condition qui libère un des processus de la liste de la condition s'il y en à un, sinon signal ( ) ne fait rien .si un thread est débloqué, il reprend son exécution à son point d'arrêt une variante de signal appelée broadcast ( ) existe (pthread norme Posix.1c) qui libère tous les threads en attente de la condition.

3. Les algorithmes :

On prend le problème de producteur / consommateur Il s'agit d'un problème ou un producteur produit les informations qui seront consommer par un consommateur, le producteur et le consommateur partagent un tampon dont lequel le producteur mise à la disposition de consommateur les caractères produits.Exemple : -consommateur : écran d'affichage. - producteur : clavier.

3.1. En sémaphore : La solution consiste à protéger ce tampon au accès concurrents on utilisant les sémaphores.

```
Var tampon : tableau [0 ...n-1] caractères ;
(N : est la taille du tampon).
n vide , n plein : sémaphore ;
n vide = n ;(* nombre de places vides *) ;
n plein = 0 ;(* nombre de plein).
Processus Producteur ;
Car : caractère ;
Tête : entier ;
Tête := 0 ;

Début
 Répétez
    Produire (Car) ;
    P (n vide) ;
   tampon [Tête] : = Car ;
   Tête : = (Tête+1) mod N ;
   V (n plein) ;
jusqu'à faux ;

Fin.
```

```
Processus Consommateur ;
Car : caractère ;
Queue : = 0 ;

Début
Répétez
  P (n plein) ;
 Car : = tampon [Queue] ;
 Queue = (Queue+1) mod n ;
V (n vide) ;
Consommateur (Car) ;
 Jusqu'à faux ;
Fin.
```

8.2. En région critique :

```
Var Tampon : Shared Record
T : Tableau [0.. N-1] : Char ;
CPT : Integer ;
end ;
CPT : = 0 ;

 Processus Producteur ;
Car : Caractère ;
Tête : Integer ;
Tête : = 0 ;

  Repeat
    Produire (Car);
Region Tampon when CPT < N do
          Begin
        Tampon [Tete] = Car ;
        Tete = (Tete +1) mod N;

        CPT: = CPT +1;
End;
          UNTIL False.

 Processus Consommateur ;
Car :   Char ;
Queue: Integer;
Queue: = 0;

Repeat
 Region Tampon when CPT > 0 do
              Begin
        Car = Tampon [Queue] ;
      Queue = (Queue + 1) mod N ;
        CPT : = CPT - 1 ;
          end ;
        Consommer (Car) ;

          UNTIL False .
```

3.2. En moniteur :

```
Producteur
Répétez
        C : = produire ( ) ;
        prod- cons .Ajouter (c) ;
 Until false .
Consommateur
Répétez
C : = prod- cons . Retirer ( ) ;
Consommer(C) ;
Until false.
Moniteur   prod-cons ;
Var        const  N ;
           Int    Cpt ;
  cons , prod : condition ;
T : Array  [0 .. N]  of  integer ;
Procedure  Ajouter  ( C )
If  Cpt > = N then  prod . wait ;
If  Cpt > = N then  prod . wait ;
        T [Cpt] = C ;
        Cpt = (Cpt + 1) mod N ;
          cons . signal ;
        end ;
       Procedure   Retirer (C)
        Begin
          If  Cpt + 0  then
            cons. wait;
                Cpt -- ;
                C = T [Cpt];
            prod. signal;
             Return (C);
        end;

       Begin

        Cpt: =0;

        end.
```

4. Conclusion:

Les processus d'un système n'exécutent pas d'une manière résolue, ils ont besoin de coopérer afin d'exécuter un certain nombre de tâche, d'ou la nécessiter de synchronisation, nous avons vu dans ce chapitre les différents solutions qui existent pour résoudre ce problème à la fin on a parler de thread UNIX. La manipulation des threads UNIX nécessite des opérations de blocage et déblocage appliqué sur les sémaphores, au contraire et comme on va voir dans le chapitre suivant l'utilisation des threads java est plus facile car il existe une classe prédéfinie dans le paquetage java.lang.thread.

// * Les références * //

[1] : http// :www.supinfo-projects.com

[2] : cours (S.E) deuxième année informatique LMD 2005/2006

[3] : http// :fr.wikipedia.org

[4] : http// :www.pps.jussieu.fr

[5] : cours (S.E) deuxième année IUT de Caen, informatique (François Bourdon)

[6]: http//:www.lb .refer.org

[7]: tours programmation concurrente et temps réel, chap. 06

[8] : ouvrage : langage java

Chapitre III:

Implémentations & commentaires

1. Introduction :

En programmation il y'a aussi des modes, et actuellement ce sont les langages orientés objet qui ont le vent en poupe. Il y a bien sûr des raisons objectives à cela, et l'utilisation généralisée des interfaces graphiques n'y est pas étrangère.

Tout le monde a entendu parler de système multi-tâches. Un tel système est capable d'exécuter plusieurs programmes en parallèle sur une même machine. Dans la quasi totalité des cas il n'y a jamais deux programmes différents qui s'exécutent au même instant. La raison en est simple : la plupart du temps, une machine n'a qu'un seul processeur et ce dernier n'est capable de réaliser qu'une seule chose à la fois. [1]

2. Choix de langage :

Un langage a été créé en Janvier 1995 chez Sun : **Java**. Il permet le parallélisme grâce à la technique des processus légers (Threads), il est défini pour la programmation d'interfaces, il est orienté objets, il est portable, le kit de développement de Sun est gratuit. Pour ces raisons, il a connu un développement fulgurant, bien plus rapide que pour tous les langages qui l'ont précédé. Ces mêmes raisons nous ont amené à le choisir comme outil de programmation de notre application on utilisant JBuilder7.

2.1. Quelques éléments de Java

Les langages orientés objet partagent un ensemble de caractéristiques essentielles :

- **Description objets.** La description des données est sous forme d'objets ; elle ressemble à un record Pascal ou une struct C. Mais le point essentiel est que les procédures qui les manipulent leur sont associées. Un objet est décrit comme une classe, comprenant les données et les procédures qui les traitent.
- **Hiérarchie**. Un objet étant défini, il est possible de préciser certaines caractéristiques en définissant de nouveaux objets ayant les mêmes caractères, plus de nouveaux. Ces caractères sont définis par de nouvelles données, ou par de nouvelles procédures. Par exemple un objet EtreVivant peut être précisé (redéfini) en Oiseau.
- **Encapsulation**. Parmi les données qui constituent l'objet, certaines sont invisibles des l'extérieur de l'objet (donc impossible à manipuler par des procédures définies ailleurs).
- **Héritage**. Lorsqu'on redéfinit un objet, à partir d'un ancien, il hérite de toutes les caractéristiques de l'ancien : données et procédures.
- **Polymorphisme**. Deux procédures ayant le même rôle dans des objets différents peuvent porter le même nom. Dans l'exemple donné ci-dessus, l'objet EtreVivant possède une procédure SeDeplacer. En le redéfinissant comme Oiseau, cette procédure précisera que le déplacement est un vol.

Le langage Java adopte la syntaxe de C, pour un apprentissage plus rapide par le plus grand nombre. La définition d'un objet (d'un type d'objets) est matérialisée par une classe (un objet est une instance de cette classe ; on peut en créer autant qu'on veut).

Chapitre III : Implémentation et commentaires

Dans l'implémentation originale de Java, produite par Sun, chaque classe est hébergée par un fichier, portant le même nom (ce qui facilite le travail de l'éditeur de liens). L'ensemble des classes qui composent un programme ou un fragment de programme est regroupé dans une structure plus large, le paquetage. Un paquetage permet de distribuer des parties de programme constituant des bibliothèques.

Java contient de nombreux paquetages prédéfinis, qui permettent de traiter les entrées/sorties, le réseau, les fichiers, et toutes choses utiles à la programmation. La portabilité d'un programme est basée sur ces paquetages, en particulier sur AWT (Abstract Windowing Toolkit) qui offre des fenêtres, menus déroulants, boutons, cases à cliquer etc. et donc évite d'utiliser ces même éléments définis par un système d'exploitation particulier. La machine virtuelle Java assure la traduction dynamique de ces outils Java en leur équivalent sur la machine où le programme se déroule. AWT est aujourd'hui remplacé par une boîte à outils beaucoup plus puissante nommée Swing.

Pour traiter les erreurs, Java propose le mécanisme des exceptions. Lorsqu'on détecte une erreur, par exemple dans une analyse syntaxique très profonde, il serait difficile de la remonter au niveau supérieur. C'est pourtant ce qu'il faut faire. Les exceptions automatisent ce traitement. Là où l'erreur est détectée, on jette une exception, et on la récupère là où on souhaite la traiter. Entre les deux, la remontée est automatique.

Le nom de la classe représente, pour un objet, son type. Lorsqu'une classe en étend une autre, un objet de cette classe possède également le type de la super-classe. Mais puisqu'une interface est une classe creuse, elle définit également un type. [2]

3. Les threads en java:

3.1. Introduction:

Depuis l'apparition du premier PC, la programmation a beaucoup évoluée.
Beaucoup de détails techniques sont profondément liés aux systèmes d'exploitation et ses particularités.
Alors que les premiers systèmes d'exploitation tels que le DOS n'étaient prévu que pour exécuter une unique application à un instant précis, les systèmes plus modernes supportent presque tous le multi tâche.
Ceci signifie qu'ils sont capables d'exécuter plusieurs choses en parallèle.

3.2. Définition des threads :

Le multi threads est une solution technique pour pouvoir faire différentes opérations en même temps.

Lorsque plusieurs choses doivent être réalisées simultanément, les threads sont l'une des meilleures solutions.
Un processus peut gérer plusieurs threads, chaque thread exécute un ensemble d'opérations logiquement regroupées.

L'environnement de la Machine Virtuelle Java est multi threads. L'équivalent de thread pourrait être tâche en français, mais pour éviter la confusion avec la notion de systèmes multitâches, on emploiera le mot thread plutôt que tâche. Le fait que Java permettent de faire tourner en parallèle plusieurs threads lui donne beaucoup d'intérêt : Ceci permet par exemple de lancer le chargement d'une image sur le Web (ce qui peut prendre du temps), sans bloquer votre programme qui peut ainsi effectuer d'autres opérations.

3.3. Les avantages des threads :

Une liste des avantages sur l'utilisation des « threads » :

- Permet de rendre le programme plus ergonomique et intelligent en gardant une IHM (Interface Homme Machine) toujours active pendant que d'autres opérations sont effectuées.
- Les opérations effectuées par notre application pourront être arrêtées « proprement ».
- Permet de lancer des opérations qui attendent une action extérieure, attente d'une connexion réseau par exemple, sans bloquer entièrement l'application.
- Les « threads » peuvent posséder des priorités permettant de rendre certaines parties du programme plus importantes que d'autres
- Le temps processeur est réparti périodiquement pour chaque partie du programme.

3.4. Caractéristique des threads :

1. Les threads partagent le même espace d'adressage ce qui signifie simplicités de partage des données, communication et ressources.
2. La création des threads est plus rapide que la création des processus (la rapidité due de la non duplication de l'espace d'adressage).
3. Une commutation des processus qui est due aux les valeurs des registres ne sont pas commuter.
4. Amélioration des performances en cas de blocage

On comparons la programmation multi processeur et multi thread il se voie que la programmation multi thread est plus efficace mais la gestion de partage du donnée est de parallélisme est faite par le programmeur. [3]

3.5. Création et démarrage d'un thread:

Il existe, en Java, deux techniques pour créer un thread. Soit on dérive notre thread de la classe **java.lang.Thread**, soit on implémente l'interface **java.lang.Runnable**.

Les différences sont subtiles, mais note déjà que si on implémente l'interface Runnable, il nous reste toujours la possibilité de dériver d'une autre classe (n'oublions pas qu'en Java, il n'y a pas d'héritage multiple). De plus, selon la technique que nous choisissons, nous aurons plus ou moins de facilité pour partager (ou non) des données entre différents threads.

3.5.1. Créer un thread en dérivant de la classe java.lang.Thread

Cette classe permet donc, si l'on en hérite de pouvoir créer un objet de Thread. En effet, la classe Thread n'est certes pas abstraite, mais il faut tout de même redéfinir la méthode *run* (qui de base ne fait rien). Cette méthode publique n'attend aucun paramètre et ne retourne rien.

3.5.2. Créer un thread en implémentant l'interface java.lang.Runnable:

La seconde technique consiste donc à implémenter l'interface java.lang.Runnable, pour implémenter une interface, on doit de coder TOUTES les méthodes qui y sont définies. L'interface Runnable est en fait très simple dans le sens où elle ne définit qu'une unique méthode : la méthode *run*. Mais, dans notre classe (qui implémente l'interface) on n'a pas de méthode *start* qui aurait pu permettre le démarrage de notre thread.

3.6. Les avantages et les inconvénients de deux techniques

Donc deux techniques principales peuvent être utilisées pour créer un thread. Résumons leurs avantages et leurs inconvénients.

	avantages	**inconvénients**
Extends Java.lang.Thread	Chaque thread a ses données qui lui sont propres	On ne peut plus hériter d'une autre classe.
Implemente Java.lang.Runnable	L'héritage reste possible .En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de votre classe sont partagés pour tous les threads qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

Notez aussi qu'il est malgré tout possible de partager des données avec la première technique étudiée : il suffit dans ce cas de définir des attributs statiques. Un attribut statique est partagé par toutes les instances d'une même classe.

3.7. Gestion des threads:

Une fois les threads créés et lancés, on peut avoir besoin de les contrôler afin d'affiner le comportement de votre application. Il est vrai qu'on a pas, en Java, un contrôle absolu sur l'exécution de vos threads, mais pas mal de choses restent malgré tout réalisables. Pour que les choses soient claires, il nous faut, dans un premier temps, mieux comprendre le cycle de vie d'un thread. Par la suite, nous regarderons quelques unes des possibilités de contrôle de plus prés.

3.8. Cycle de vie d'un thread:

Un thread peut passer par différents états, tout au long de son cycle de vie. Le diagramme suivant illustre ces différents stades ainsi que les différentes transitions possibles. Nous reprendrons ensuite en détail chaque point du diagramme, pour mieux comprendre chaque étape de vie d'un thread

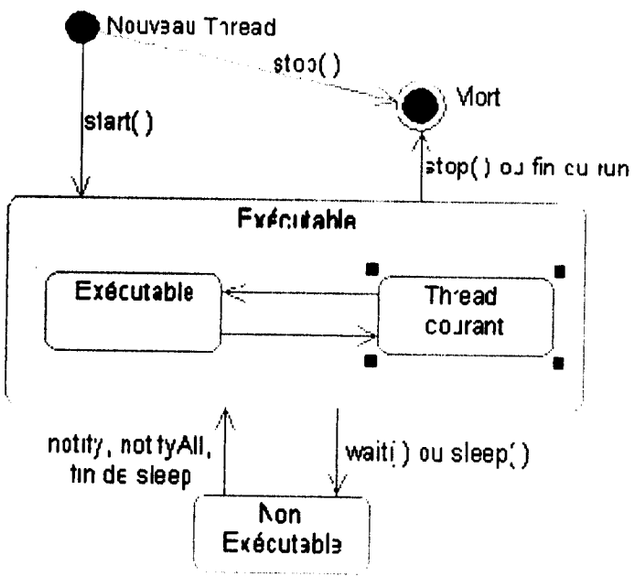


Schéma1.cyle de vie d'un thread [2]

Start () : Cette méthode permet de démarrer effectivement le thread sur lequel elle est invoquée, ce qui va provoquer l'appel de la méthode run () du thread. Cet appel est obligatoire pour démarrer l'exécution d'un thread. En effet, la création d'un objet de classe Thread ou d'une classe dérivée de Thread (par exemple, grâce à l'appel new Thread ()) ne fait que créer un objet sans appeler la méthode run ().

stop() : Cette méthode permet d'arrêter un thread en cours d'exécution. Elle est utile principalement pour stopper des threads dont la méthode run () ne se termine jamais (comme par exemple, dans la classe TimeCounter, la méthode run () n'a pas de raison de s'arrêter d'elle-même puisqu'aux dernières nouvelles, on n'a pas encore trouver le moyen d'arrêter le temps !).

Sleep () : Cette méthode static permette d'arrêter le thread courant pendant un certain laps de temps, pour permettre ainsi à d'autres threads en attente, de s'exécuter. Par exemple, une fois mis à jour une horloge par un thread, celui-ci peut arrêter son exécution pendant une minute, en attendant la prochaine mise à l'heure.

Yield (): Cette méthode static permet au thread courant de céder le contrôle pour permettre à d'autres threads en attente, de s'exécuter. Le thread courant devient ainsi lui-même en attente, et regagne la file d'attente. De manière générale, vos threads devraient s'arranger à effectuer des séries d'instructions pas trop longues ou à entrecouper leur exécution grâce à des appels aux méthodes sleep () ou yield (). Il faut éviter de programmer des séries d'instructions interminables sans appel à sleep () ou yield (), en pensant qu'il n'y aura pas d'autres threads dans votre programme. La Machine Virtuelle Java peut avoir elle aussi des threads système en attente et votre programme s'enrichira peut-être un jour de threads supplémentaires.

setPriority () : Permet de donner une priorité à un thread. Le thread de plus grande priorité sera toujours exécuté avant tous ceux en attente.

Join () : permet la mise en attente du thread en cours d'exécution jusqu'a ce que le thread appliquant la méthode join soit terminé, ce thread bloque le thread en cours d'exécution et à la fin débloquera ce thread.

3.9. Gestion de la priorité d'un thread:

En Java, on peut jouer sur la priorité de notre threads. Sur une durée déterminée, un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread. Il exécutera donc, globalement, plus de code.

La priorité d'un thread va pouvoir varier entre 0 et 10. Mais , il n'est en aucun cas garanti que le système hôte saura gérer autant de niveaux de priorités. Des constantes existent et permettent d'accéder à certains niveaux de priorités : MIN_PRIORITY (0) - NORM_PRIORITY (5) - MAX_PRIORITY (10).

3.10. Gestion d'un groupe de threads:

Une autre possibilité intéressante consiste à regrouper différents threads. Dans un tel cas, on peut invoquer un ordre sur l'ensemble des threads du groupe, ce qui peut dans certains cas sérieusement simplifier notre code.Pour inscrire un thread dans un groupe, faut t'il que le groupe soit initialement créé.

Pour ce faire, il vous faut instancier un objet de la classe ThreadGroup. Un fois le groupe créé, vous pouvez attacher vos threads à ce groupe.
L'attachement d'un thread à un groupe se fait via un constructeur de la classe Thread. Par la suite, un thread ne pourra en aucun cas changer de groupe. Si nous avons mis en oeuvre de l'héritage, n'oubliez pas que dans ce cas tous les constructeurs parents sont invoqués lors de la création d'un objet.

Une fois tous threads attachés à groupe, nous pouvons alors invoquer les méthodes de contrôle d'exécution des threads sur l'objet de groupe. Les noms des méthodes sont identiques à la classe Thread : *suspend()*, *resume()*, *stop()*, ...[2]

3.11. La synchronisation des threads :

Tous les threads d'une même Machine Virtuelle partage le même espace mémoire, et peuvent donc avoir accès à n'importe quels méthode ou champ d'objets existants. Ceci est très pratique, mais dans certains cas, on peut avoir besoin d'éviter que deux threads n'aient accès n'importe quand à certaines données. Si par exemple, un thread `threadCalcul` a pour charge de modifier un champ `var1` qu'un autre thread `threadAffichage` a besoin de lire pour l'afficher, il semble logique que tant que `threadCalcul` n'a pas terminé la mise à jour ou le calcul de `var1`, `threadAffichage` soit interdit d'y avoir accès. Nous aurons donc besoin de synchroniser nos threads

3.11.1. Notions de verrous:

L'environnement Java offre un premier mécanisme de synchronisation (les verrous) Chaque objet Java possède un verrou et seul un thread à la fois peut verrouiller un objet. Si d'autres threads cherchent à verrouiller le même objet, ils seront endormis jusqu'à que l'objet soit déverrouillé. Cela permet de mettre en place ce que l'on appelle plus communément une section critique.

Pour verrouiller un objet par un thread, il faut utiliser le mot clé **synchronized**. En fait, il y a deux façons de définir une section critique. Soit on synchronise un ensemble d'instructions sur un objet, soit on synchronise directement l'exécution d'une méthode pour une classe donnée. Dans le premier cas, on utilise l'instruction *synchronized*.

Dans le second cas, on utilise le qualificateur *synchronized* sur la méthode considérée. Le tableau suivant indique la syntaxe à utiliser dans les deux cas.

synchronized(object) {
// Instructions de manipulation d'une ressource partagée.
}
	public **synchronized** void meth(int param) {
	// Le code de la méthode synchronizée.
	}

3.11.2. Utilisation de synchronized :

La synchronisation des threads se fait grâce au mot clé `synchronized`, employé principalement comme modificateur d'une méthode. Soient une ou plusieurs méthodes `methodeI ()` déclarées `synchronized`, dans une classe `Classe1` et un objet `objet1` de classe `Classe1` : Comme tout objet Java comporte un verrou (*lock* en anglais) permettant d'empêcher que deux threads différents n'aient un accès simultané à un même objet.

L'une des méthodes `methodeI () synchronized` est invoquée sur `objet1`, deux cas se présentent :

Soit `objet1` n'est pas verrouillé : le système pose un verrou sur cet objet puis la méthode `methodeI ()` est exécutée normalement. Quand `methodeI ()` est terminée, le système retire le verrou sur `Objet1`.
La méthode `methodeI ()` peut être récursive ou appeler d'autres méthodes `synchronized` de `Classe1` ; à chaque appel d'une méthode `synchronized` de `Classe1`, le système rajoute un verrou sur `Objet1`, retiré en quittant la méthode. Quand un thread a obtenu accès à un objet verrouillé, le système l'autorise à avoir accès à cet objet tant que l'objet a encore des verrous (réentrance des méthodes `synchronized`).

- Soit `objet1` est déjà verrouillé : Si le *thread courant* n'est pas celui qui a verrouillé `objet1`, le système met le *thread courant* dans l'état *bloqué*, tant que `objet1` est verrouillé. Une fois que `objet1` est déverrouillé, le système remet ce thread dans l'état *exécutable*, pour qu'il puisse essayer de verrouiller `objet1` et exécuter `methodeI ()`.

Si une méthode `synchronized` d'une classe `Classe1` est aussi `static`, alors à l'appel de cette méthode, le même mécanisme s'exécute mais cette fois-ci en utilisant le verrou associé à la classe `Classe1`.
Si `Classe1` a d'autres méthodes qui ne sont pas `synchronized`, celles-ci peuvent toujours être appelées n'importe quand, que `objet1` soit verrouillé ou non.

3.11.3. Synchronisation avec wait () et notify () :

`synchronized` permet d'éviter que plusieurs threads aient accès en même temps à même objet, mais ne garantit pas l'ordre dans lequel ces méthodes seront exécutées par des threads.

Pour cela, il existe plusieurs méthodes de la classe Object qui permettent de mettre *en attente* volontairement un thread sur un objet (avec les méthodes `wait ()`), et de prévenir des threads en attente sur un objet que celui-ci est à jour (avec les méthodes `notify ()` ou `notifyAll()`).

Ces méthodes ne peuvent être invoquées que sur un objet verrouillé par le *thread courant*, c'est-à-dire que le *thread courant* est en train d'exécuter une méthode ou un bloc `synchronized`, qui a verrouillé cet objet. Si ce n'est pas le cas, une exception Illegal MonitorStateException est déclenchée.

Quand `wait ()` est invoquée sur un objet `objet1` (`objet1` peut être `this`), le *thread courant* perd le contrôle, est mis *en attente* et l'ensemble des verrous d'`objet1` est retiré. Comme chaque objet Java mémorise l'ensemble des threads mis *en attente* sur lui, le *thread courant* est ajouté à la liste des threads *en attente* de `objet1`. `Objet1` étant déverrouillé, un des threads *bloqués* parmi ceux qui désiraient verrouiller `objet1`, peut passer dans l'état *exécutable* et exécuter une méthode ou un bloc `synchronized` sur `objet1`.

Un thread `thread1` mis *en attente* est retiré de la liste d'attente de `objet1`, quand une des trois raisons suivantes survient :

- `thread1` a été mis *en attente* en donnant en argument à `wait ()` un délai qui a fini de s'écouler.
- Le *thread courant* a invoqué `notify ()` sur `objet1`, et `thread1` a été choisi parmi tous les threads *en attente*.
- Le *thread courant* a invoqué `notifyAll ()` sur `objet1`.

`thread1` est mis alors dans l'état *exécutable*, et essaye de verrouiller `objet1`, pour continuer son exécution. Quand il devient le *thread courant*, l'ensemble des verrous qui avait été enlevé d'`objet1` à l'appel de `wait ()`, est remis sur objet1, pour que `thread1` et `objet1` se retrouvent dans le même état qu'avant l'invocation de `wait ()`. [4]

4. Implémentation de l'application de producteur / consommateur :

4.1. Introduction :

Pour mettre en oeuvre un exemple de synchronisation un peu évolué, nous allons considérer l'exemple classique de producteur/consommateur. Dans un tel cas, une ressource partagée est utilisée par des threads qui produisent et par des threads qui consomment.

Une mémoire "tampon" est gérée par un producteur d'objets et un consommateur. Cette mémoire ne peut contenir qu'un objet.

On synchronise l'accès à cette mémoire (production et consommation non simultanées).

Si le producteur doit déposer un objet alors qu'il en existe déjà un dans la mémoire tampon, il attend ... blocage !

Même type de blocage pour un consommateur qui attend le dépôt d'un objet.

Chapitre III : Implémentation et commentaires

4.1. Relâchement d'exclusion :

La solution du problème précédent passe par la mise en œuvre d'une procédure de relâche d'exclusion :

- L'attente du Producteur ou du Consommation se fait au moyen de la méthode `wait()` (susceptible de déclencher une exception `InterruptedException`) qui relâche l'exclusion sur l'objet.
- Un processuÿÿsort de l'attente lorsqu'uÿÿÿÿtre processus exécute la méthode `notify()` (qui relance *un* processus en attente) ou la méthode `notifyAll()` (qui relance *tous* les processus en attente).

Le fonctionnement :

- Le producteur fonctionne en flux tendu avec un stock très limité.
- Le consommateur consomme le produit à son rythme (il est parfois un peu lent).
- Le producteur doit attendre le consommateur pour produire.

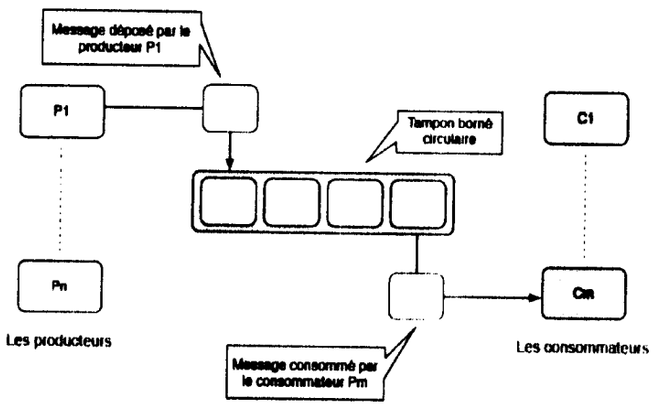


Schéma2.schéma de principe [4]

Le principe de notre programme est de synchroniser entre un producteur qui produit des objets (messages données) et les places dans une file d'attente et un consommateur qui prend un message (caractère par caractère) de la file, L'accès à la file doit être atomique et la file est de taille limitée le rythme de consommation est aléatoire.
Pour implémenter notre objet partagé, nous allons coder trois classes Producteur, Consommateur et Buffer.

La classe Buffer :

C'est une classe hérite de la classe Stack qui représente un LIFO et elle même hérite de la classe Vector, on utilise le Buffer pour posé des messages par le producteur tant que le Buffer n'est pas plein,et retirer un message par le consommateur tant qu'il n'est pas vide.

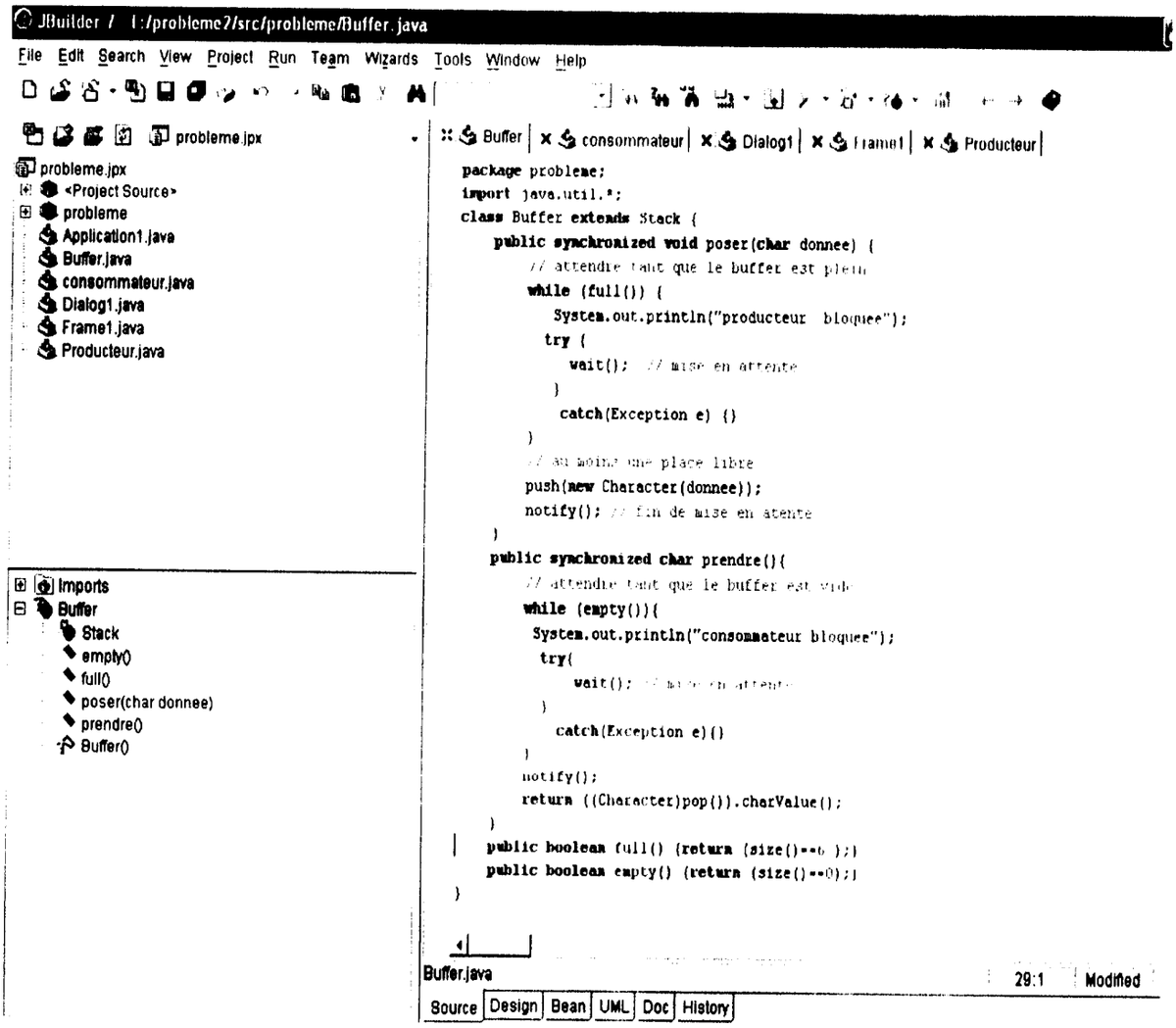


Figure1.ligne de code de la classe Buffer

Maintenant que notre ressource partagée est prête, il ne nous reste plus qu'à coder nos producteurs et nos consommateurs. Dans les deux cas, ces deux types (classes) de composants partagent tous certaines caractéristiques : ils travaillent tous sur la même pile et dans les deux cas, cadencer les choses via un Thread. sleep pourra permettre une bonne lisibilité des résultats sur la console. Nous utilisons ici l'héritage pour définir le tronc commun à tous nos threads.

Par la suite, on dérive de cette classe nos deux types de threads :

La classe Producteur : Les producteurs (classe *Producteur*) qui vont produire des messages entrées et donc les empiler (tant que cette dernière (le Buffer) n'est pas pleine)

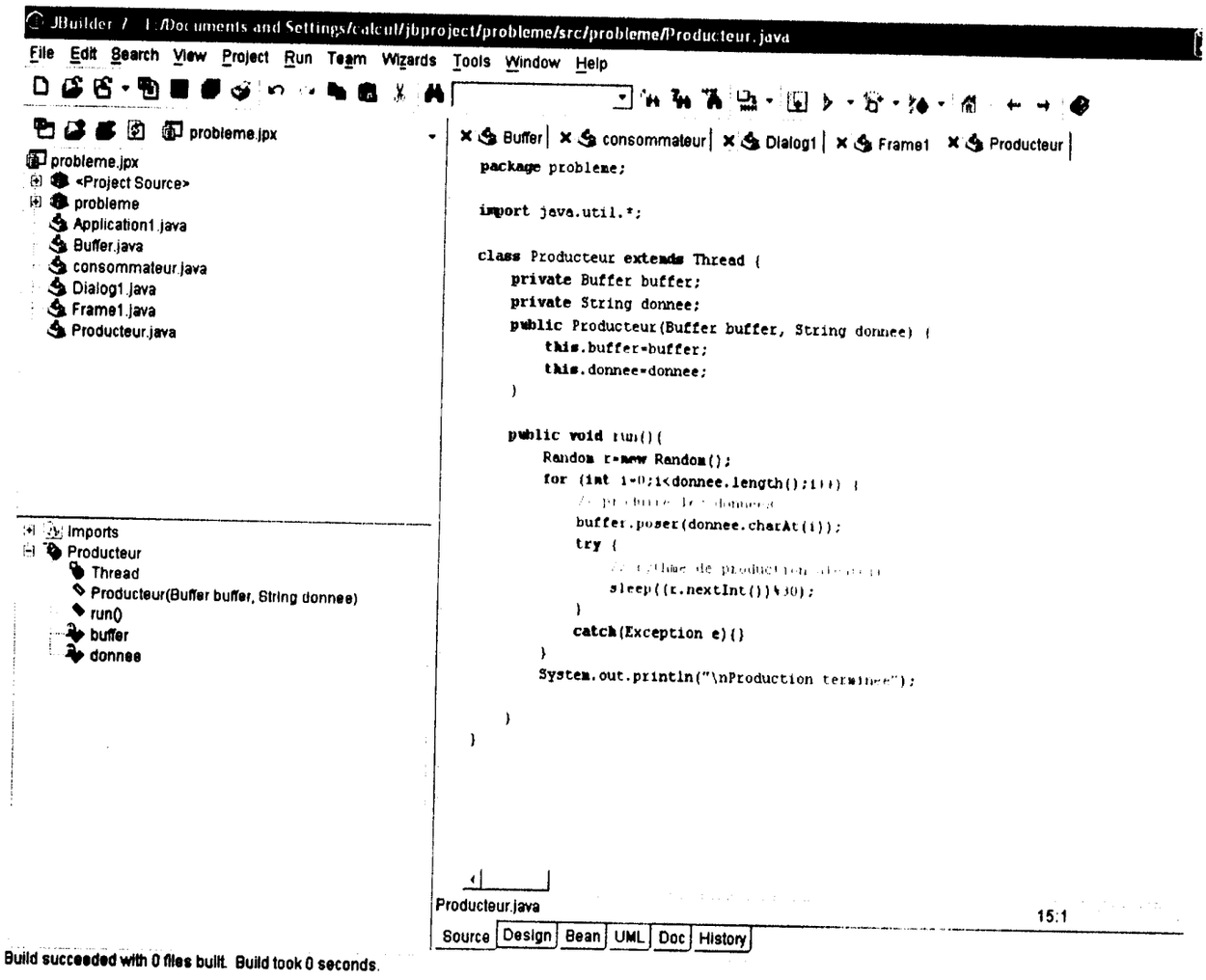


Figure2.ligne de code de la classe Producteur

La classe Consommateur : Les consommateurs (class *Consommateur*) qui vont lire des caractères sur le Buffer (tant que cet dernière n'est pas vide).

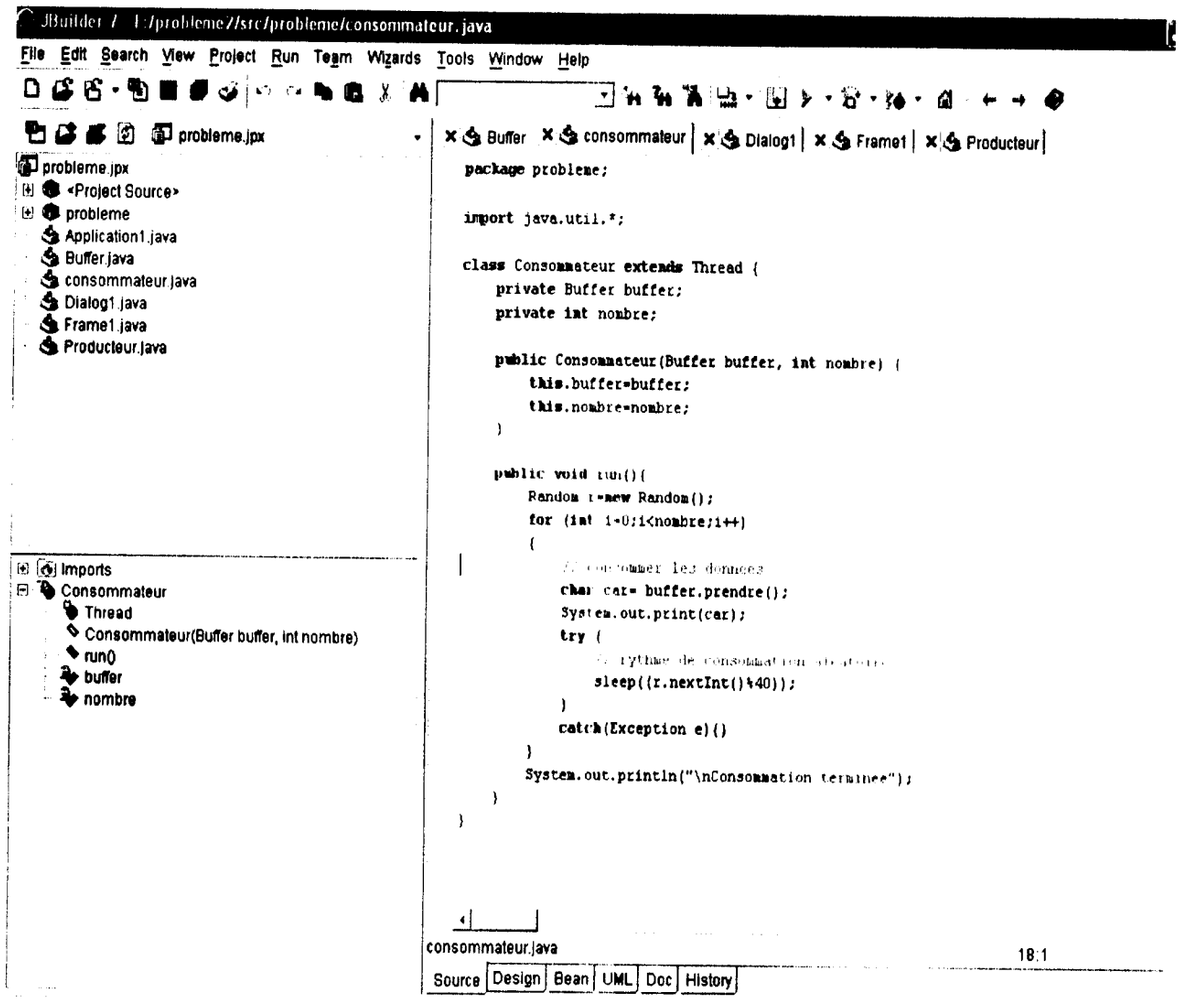


Figure3.ligne de code de la classe Consommateur

La classe Dialog1 :

Maintenant il ne reste plus qu'à coder une classe de démarrage afin d'initier le démarrage des threads à synchroniser sur le Buffer et l'affichage.

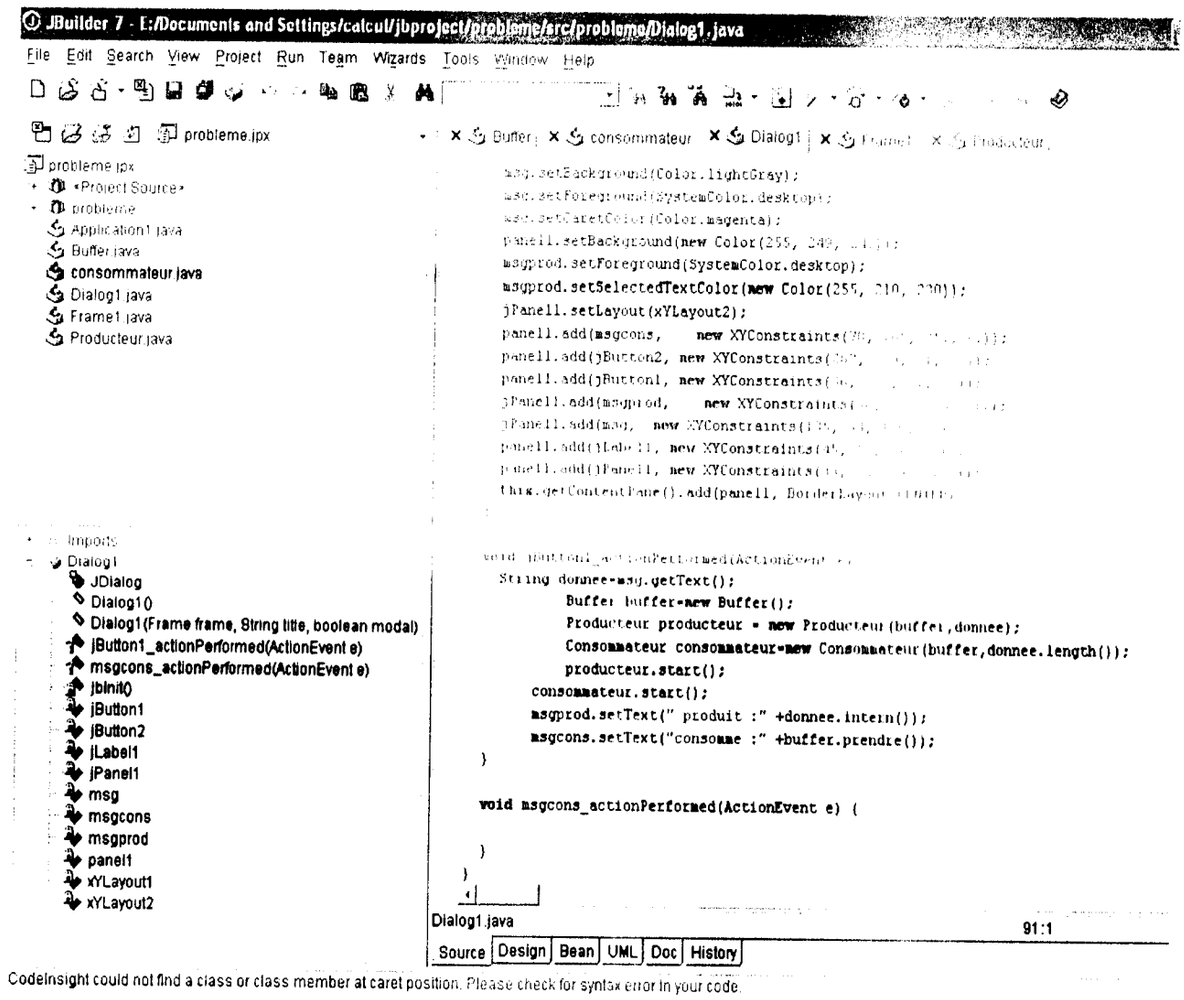
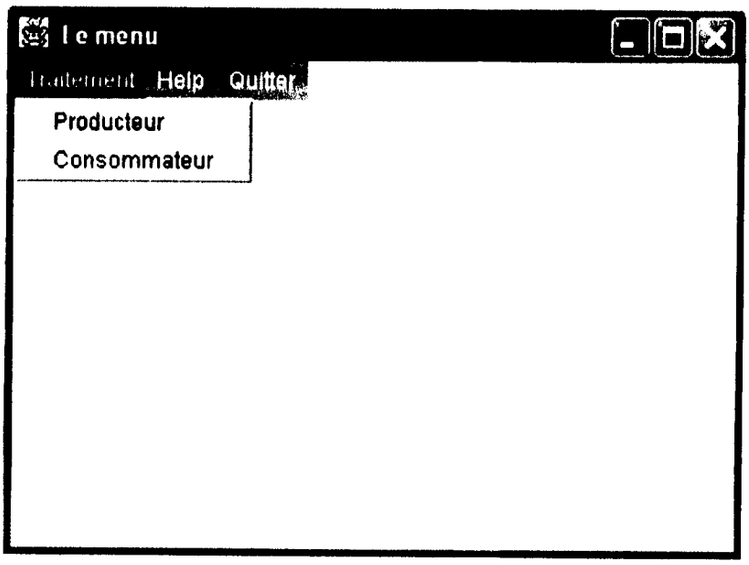


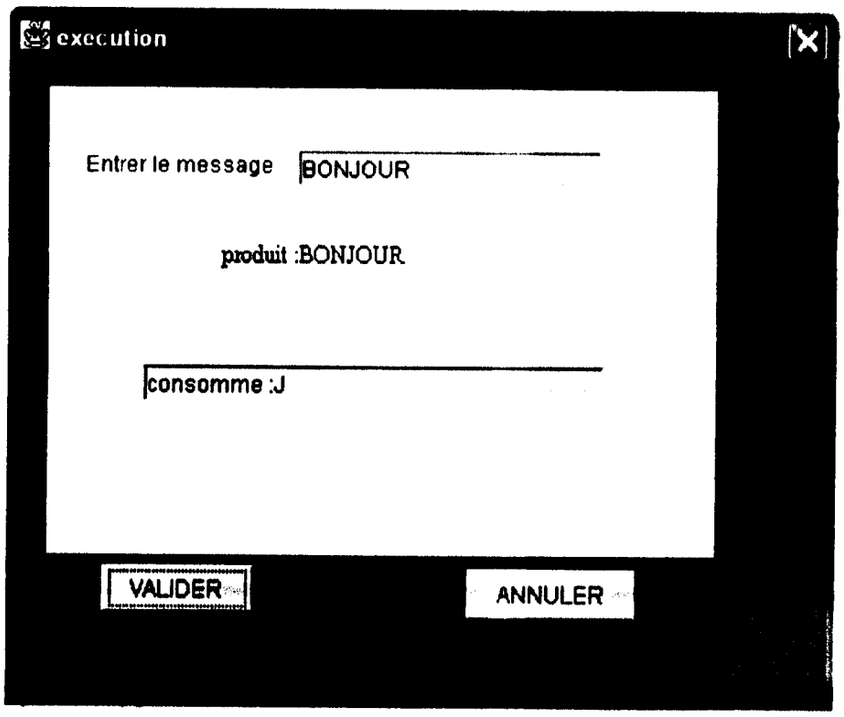
Figure 4.ligne de code de la classe Dialog1

L'interface :

On fait comme première interface le menu de ce programme qui contient de :
Traitement et help, le help c'est des commentaires du programme pour l'aide,
Et le Traitement contient producteur & consommateur, Exit c'est pour quitter ou sorte de l'exécution.



Lorsqu'on clique sur producteur & consommateur on obtient le Dialog d'exécution donc il faut entrer le message qu'on veut le produire, voici un exemple qui donne l'exécution suivante :



On a le message entré c'est BONJOUR et on remarque q'on a deux bouton VALIDER et ANNULER :

VALIDER : c'est pour lancer le producteur et consommateur dans notre exemple le caractère consommé est J ;

ANNULER : pour annuler l'exécution et retourne au menu du programme.

5. Conclusion :

Nous venons donc de voir qu'en Java, il est possible de créer des threads. Deux techniques sont proposées. Chacune d'entre elle ayant des avantages et des inconvénients (nous avons dans ce chapitre, volontairement un peu utilisé les deux techniques). De plus nous avons vu qu'il été possible de contrôler, plus ou moins, finement l'exécution de threads. Il existe même la notion de groupes de threads permettant une gestion en lots de threads.

Mais comme dans tout langage, s'il n'était pas possible de synchroniser les accès concurrents aux ressources partagées, cette solution serait trop souvent inutile. En conséquence, la notion de verrous (sections critiques) et un mécanisme de synchronisation (mise en sommeil et réveil de threads).

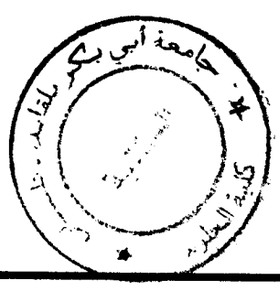
// * Les références * // :

[1] : http// : www.dil.univ-mrs.fr

[2] : http// : www.infini-fr.com

[3] : cours du système d'exploitation de troisième année informatique LMD

[4] : http// : www.eteks.com



Conclusion générale

Depuis l'apparition de concepts d'exclusion mutuelle en troisième génération des système d'exploitation, différents solutions ont été proposés dans la littérature : sémaphore, moniteur, région critique et threads.

Dans ce modeste travail on a essaie de faire une étude de problème de synchronisation et les solutions proposés, on a aussi programmé en Java la solution de l'un des problèmes les plus connue (producteur consommateur) .Ce qui nous a permet l'appréhender le langage de programmation Java ainsi que les supports de développement de ce langage (JBuilder) qui a été utilisé comme base pour la programmation de notre solution.

En fin, nous espérons que ce modeste travail peut être plus amélioré par les promotions futures.