Programmation Parallèle
IUP MIAGe 3
Année 2002-2003
Université Bordeaux I

Alain Griffault

7 mai 2003

Table des matières

Informations générales.

0.1 Avertissement

Ce document correspond à mes notes de cours. Le niveau de rédaction des diverses parties est très inégal, et il comporte certainement quelques erreurs et beaucoup d'oublis.

0.2 Bibliographie

- La programmation parallèle. J.P. Banâtre (Eyrolles).
- Algorithmique du parallélisme (le problème de l'exclusion mutuelle). M. Raynal (Dunod).
- Synchronisation de programmes parallèles. F. André, D. Herman. J.P. Verjus (Dunod).
- Processus Séquentiel Communicants. C.A.R. Hoare (Masson).
- Java : de l'esprit à la méthode. M. Bonjour, G. Falquet, J. Guyot et A. Le Grand (International Thomson Publishing)

0.3 Les pré-requis

- Le langage JAVA.

0.4 Le contrôle

- l'examen : Devoir surveillé.

Chapitre 1

Introduction.

1.1 Pour quoi faire ?

A partir de l'idée simple :*A plusieurs, on est plus efficace*, la programmation parallèle peut être vue comme :*Utiliser plus de moyens (ressources informatiques) pour réaliser plus rapidement une tâche.*

1.1.1 Des calculs indépendants

Soient deux matrices A et B de taille (m,n), et le calcul C=A+B.

Dans un langage de programmation classique :

```
pour i de 1 a m faire
  pour j de 1 a n faire
    c(i,j) = a(i,j) + b(i,j)
```

Bien que les calculs des c(i,j) soient indépendants, la programmation séquentielle nous oblige à fournir l'ordre dans lequel seront faits les calculs.

Dans un langage de programmation parallèle, on écrira :

```
En parallele, pour tous les couples (i,j) dans [1..m,1..n] faire
    c(i,j) = a(i,j) + b(i,j)
```

Le programmeur n'a pas à introduire du "comment" dans son programme, mais seulement du "quoi faire". Le programme est plus proche de la spécification.

1.1.2 Des sources d'information multiples

Soit un système qui contrôle le niveau d'une cuve, en modifiant l'ouverture d'une vanne en fonction d'informations provenant de différents capteurs.

Dans un langage de programmation classique :

```
program controleur
debut
  pour i de 1 à m faire
    si valeur(capteur[i]) != valeurAncienne(capteur[i]) alors
      modifierOuvertureVanne(i,valeur(capteur[i]))
fin
```

Deux défauts principaux :

- Même lorsque toutes les valeurs des capteurs restent inchangées, le programme n'arrête pas de ne rien faire.

– La réactivité peut être mauvaise : N capteurs ⇒ N cycles processeur au pire des cas pour traiter une variation d'un capteur.

Dans un langage de programmation parallèle, on écrira :

```
Tache C[i]
  Pour chaque modification de valeur(capteur[i])
    envoyerMessage(controleur,valeur(capteur[i]));

Tache controleur
  recevoir(C[i],valeur(capteur[i])) -> modifierOuvertureVanne(i,valeur(capteur[i]));
```

Les deux défauts précédents ont disparu.

1.1.3 Gain en efficacité

Soit un problème donné. Un algorithme séquentiel fourni le résultat en un temps t_s. Un algorithme parallèle exécuté sur N processeurs fourni le résultat en un temps t_N.
L'efficacité est le rapport $E = t_s/(N \times t_N)$.
Si $E = 1$, c'est parfait. En réalité, on a toujours $E < 1$ du fait que la répartition du travail entre les N processeurs et la collecte des résultats prennent du temps.

1.2 Où en trouve t'on ?

1.2.1 Les systèmes d'exploitation

Les systèmes multi-tâches et multi-utilisateurs temps partagés essaient d'utiliser au mieux les ressources (notamment le processeur) afin que chacun ait l'impression d'avoir la machine pour lui seul. Les techniques utilisées sont du domaine de la programmation parallèle.

1.2.2 Les calculs

Les calculs météorologiques, d'images de synthèse, de simulation aéronautique, de déformation des structures...sont très “gourmands” en ressource processeurs.

La plus part de ces programmes sont écrits en fortran parallèle et s'exécutent sur des machines SIMD (Single Instruction Multiple Data). Ce type d'ordinateur est parfait lorsque les calculs sont très indépendants.

Les derniers CRAY possèdent plus de 1000 processeurs qui exécutent tous en même temps une même instruction.

Remarque : cela ne sert à rien de prévoir en 30 heures le temps du lendemain.

1.2.3 Les applications temps réel

Ce sont des applications avec des contraintes de temps très fortes (la réaction à un signal provenant d'un capteur ABS sur une voiture doit **obligatoirement** avoir lieu avant un délai très court.

1.2.4 Les télécommunications

Le développement d'un protocole est par nature un programme parallèle. L'émetteur et le récepteur sont généralement sur des machines distinctes, donc sur des processeurs différents.

De plus, il est beaucoup plus facile de développer un protocole en distinguant trois parties : l'émetteur, le récepteur, et une normalisation des messages échangés ; plus tôt que de développer ce même protocole de façon monolithique.

1.2.5 L'animation graphique

Les temps de calculs des images peuvent être longs. L'utilisateur n'apprécie pas de voir une image s'afficher lentement. Une technique consiste à faire calculer la nouvelle image par une tâche dédiée, puis une fois ce travail fait, elle signale à la tâche dédiée à l'affichage qu'elle peut commencer son travail.

Ce découpage rend non seulement le programme plus lisible, mais permet si l'on dispose de plusieurs processeurs, de commencer le calcul de la nouvelle image pendant que la précédente s'affiche.

1.2.6 Les interfaces Homme-Machine

Les interfaces des applications "modernes" sont composés d'une multitude de boutons et autres objets graphiques. L'application doit donc à tout instant pouvoir réagir à n'importe quel action autorisée de l'utilisateur.

Le schéma de conception Vue-Contrôle-Application permet d'écrire un contrôle réactif aux actions effectuées sur la vue. Les composants graphiques notifient le contrôleur lorsqu'ils sont modifiés.

1.2.7 Les applications client-serveur

Les applications client-serveur généralisent le schéma précédent d'un protocole. Une entité (le serveur) échangent des informations avec plusieurs entités (les clients).

Chapitre 2

Les concepts des langages parallèles

2.1 Les caractéristiques d'un langage parallèle

Contrairement à un programme séquentiel qui spécifie une liste ordonnée d'instructions, et qui est déterministe (Sorties = fonction(Entrées)) :

- un programme parallèle est découpé en processus (tâches).
- les processus possèdent des relations de coopération.
- un programme parallèle spécifie l'exécution concurrente de cet ensemble de processus.
- un programme parallèle n'est pas nécessairement déterministe.

Les caractéristiques essentielles d'un langage :

1. l'expression du parallèlisme,
2. les modes de communication,
3. les modes de synchronisation,
4. l'expression du non-déterminisme,
5. éventuellement l'expression du temps.

2.2 L'expression du parallèlisme

2.2.1 Les coroutines

```
Coroutine Producteur
var m : message
debut
tant que vrai faire
     debut
     // construction du message m
     x <- m
     RESUME Consommateur
     fin
fin
```

```
Coroutine Consommateur
var m : message
debut
tant que vrai faire
     debut
     m <- x
     // traitement du message m
     RESUME Producteur
     fin
fin
```

Un programme est décomposé en processus dont un seul est actif à la fois. En fait il s'agit plus de modularité que de parallélisme.

Le langage Modula 2 utilise les coroutines.

2.2.2 Le fork

L'instruction fork (en langage C) spécifie à un processus de commencer son exécution. Le processus père peut continuer ou arréter la sienne. Attention au fork récursif.

2.2.3 Les instructions spéciales

```
Cobegin
      Processus1 // Processus2 // ... // ProcessusN
Coend
```

```
Parbegin
      Processus1 // Processus2 // ... // ProcessusN
Parend
```

Ces instructions démarrent tous les processus simultanément, l'ordre dans lequel ils s'exécutent est donc indeterminé.

L'instruction est terminée lorsque TOUS les processus sont terminés.

Les langages ConcurrentPascal, CSP (Concurrent Sequential Processes), OCCAM (langage issu de CSP dédié aux transputers) utilisent ce type d'instruction.

2.2.4 La création dynamique de processus

Suivant les langages, les processus sont des instances d'objets (C, Java, Ada), ou bien des processus déclarés (CSP).

2.3 Les modes de communication/synchronisation

2.3.1 Les variables partagées

Les processus partagent généralement le processeur et une zone mémoire. Nous faisons l'hypothèse qu'il est impossible que les deux processus accèdent au même instant (même cycle du processeur) à la même variable.

Exemple : Soient les deux processus de mise à jour d'un compte bancaire et la variable globale `compte` initialisée à 10.

```
Processus A
var aux : integer
begin
aux := compte;
aux := aux + 10;
compte := aux;
end.
```

```
Processus B
var aux : integer
begin
aux := compte;
aux := aux - 5;
compte := aux;
end.
```

La figure 2.1 représente le graphe des comportements du "programme" constitué des deux processus. Chaque nœud contient les valeurs des trois variables, et les arcs représentent les instructions.

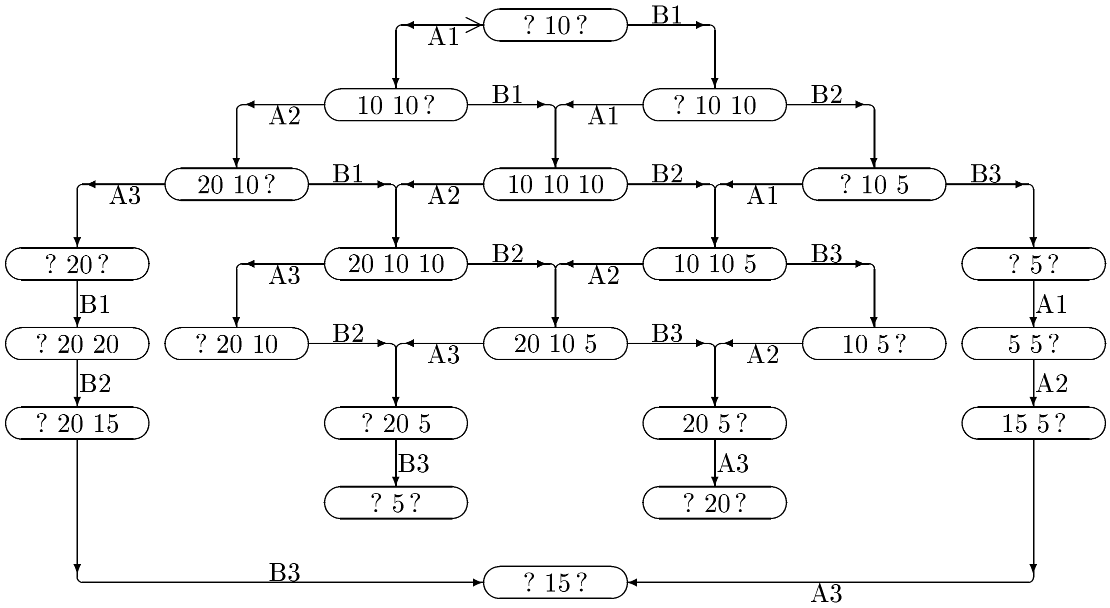

Figure 2.1: Les trois variables.

Le résultat dépend de l'entrelacement des actions, donc de l'ordonnancement des processus. Il y a donc un besoin de contrôler le déroulement d'un processus, c'est à dire l'entrelacement des actions afin de pouvoir garantir le résultat (15 ici).

2.3.2 Les messages

Les primitives de communication inter-processus sont en général :
- emettre(message, destinataire)
- recevoir(message, source).

Les messages permettent de synchroniser des processus distants.

Il existe des variantes en fonction des sémantiques :

- emettre(message, destinataire) bloquant ou non.
- recevoir(message, source) synchrone ou non.

Exemple : Le protocole du bit alterné.

```
Processus Emetteur
begin
parite := true;
while true do
     begin
     recevoir(message); /* producteur */
     repeat
           emettre(message,parite);
           recevoir(ack,ackParite);
     until parite == ackParite;
     parite := not parite;
     end;
end
```

```
Processus Recepteur
begin
ackParite := true;
while true do
     begin
     repeat
           recevoir(message,parite);
           emettre(ack,parite);
     until parite == ackParite;
     emettre(message); /* consommateur */
     ackParite := not ackParite;
     end;
end
```

2.4 L'expression du non-déterminisme

DIJKSTRA (75) propose deux constructions pour introduire le non déterminisme dans l'exécution d'un programme séquentiel. Ces deux structures de contrôle (alternative et répétitive) sont basées sur les commandes gardées.

```
<commandeGardee> ::= <garde> -> <listeCommandes>
<garde>          ::= <expressionBooleenne>|<commandeEntree>|<commandeSortie>
<listeCommandes> ::= <commande>{;<commande>}*
<commande>       ::= <instruction>|<commandeEntree>|<commandeSortie>
<commandeEntree> ::= <idProcessus>?<idMessage>
<commandeSortie> ::= <idProcessus>!<idMessage>
```

Une commande gardée ne peut être exécutée que lorsque la(le) garde est vrai(e). L'instruction (si elle existe) associée à la garde est exécutée, puis la liste de commandes associée.

Les structures de contrôles :

```
<ensembleCommandeGardees> ::= <commandeGardee>{[] <commandeGardee>}*
<alternative>             ::= if <ensembleCommandeGardees> fi
<repetitive>              ::= do <ensembleCommandeGardees> od
```

et leurs sémantiques.

Alternative

1. Evaluation de tous les gardes.
2. Si aucun garde n'est vrai alors ECHEC sinon
   (a) Sélection (non déterministe) d'un garde parmi ceux évalués à vrai.
   (b) Exécution de la commande gardée associée.
   (c) FIN de l'alternative.

Répétitive

1. Evaluation de tous les gardes.
2. Si aucun garde n'est vrai alors FIN sinon
   (a) Sélection (non déterministe) d'un garde parmi ceux évalués à vrai.
   (b) Exécution de la commande gardée associée.
   (c) RETOUR en (1).

Ce concept n'existe que dans peu de langages (ADA, CSP, OCCAM).

2.4.1 Un exercice

```
if
   a < b -> a := a+1;
   []
   a > 0 -> a := a-1;
fi
```

```
do
   a < b -> a := a+1;
   []
   a > 0 -> a := a-1;
od
```

après DO		avant		après IF	
a	b	a	b	a	b
boucle infinie		5	10	6,4	10
0	-10	5	-10	4	-10
boucle infinie		-5	10	-4	10
-5	-10	-5	-10	échec	

2.4.2 Un exemple

Un serveur à l'écoute de ses clients.

```
do
  Client1?requete -> //service au client 1
    []
  Client2?requete -> //service au client 2
    []
  ...
    []
  ClientN?requete -> //service au client N
```

Dans un langage déterministe l'écriture impose un ordre qui ne correspond à rien.

```
if Client1?requete {//service au client 1}
elsif Client2?requete {//service au client 2}
elsif ...
elsif ClientN?requete {//service au client N}
```

2.5 L'expression du temps

Dans certaines applications, il existe des contraintes dites temporelles. Divers concepts peuvent être introduits pour ces phénomênes.

- La notion de priorité permet de choisir en fonction du contexte certaines actions plus tôt que d'autres (OCCAM).
- La notion de délai permet de continuer une exécution lorsqu'un événement ne s'est pas produit avant un délai fixé. Très utilisé dans les protocoles.
- la notion de temporisation permet de retarder une instruction, généralement afin de permettre à un autre processus de faire des choses.

Chapitre 3

Les mécanismes de synchronisations

3.1 Le problème de la section critique

N processus exécutent le programme suivant :

```
debut
tant que vrai faire debut
        "entrer en section critique"
        Section Critique
        "sortir de section critique"
        Section Non Critique
        fin
fin
```

Les objectifs :
- pas 2 processus en S.C en même temps.
- pas d'attente infinie.
- pas d'attente s'il n'y a pas de compétition.
- *équitable si possible.*

3.2 Les solutions logicielles.

Première idée pour 2 processus : un feu vert/rouge. le feu est initialisé à vert.

```
debut
tant que vrai faire debut
        tant que feu = rouge faire rien;
        feu := rouge;
        Section Critique;
        feu := vert;
        Section Non Critique
        fin
fin
```

Les 2 processus peuvent être ensemble en S.C

Deuxième idée : Deux drapeaux Flag[i] signifie que Pi veut entrer et les drapeaux sont initialisés à faux.

```
debut
tant que vrai faire debut
        Flag[i] := true;
        tant que Flag[j] faire rien;
```

```
        Section Critique;
        Flag[i] := false;
        Section Non Critique
        fin
fin
```

Les 2 processus ne peuvent pas être ensemble en S.C

Il n'y a pas d'attente s'il n'y a pas de compétition.

Il y a possibilité d'attente infinie (famine).

la preuve est faite par la construction du graphe du comportement des deux processus.

Troisième idée : une variable qui indique "A qui le tour". `debut`

```
tant que vrai faire debut
        tant que tour = j faire rien;
        Section Critique;
        tour := j ;
        Section Non Critique
        fin
fin
```

2 processus ne peuvent pas être ensemble en S.C

Il n'y a pas d'attente infinie.

Il peut y avoir attente s'il n'y a pas de compétition. C'est un algorithme d'alternance (jeton).

Algorithme de Peterson pour 2 processus. Combiner les deuxième et troisième idées.

```
debut
tant que vrai faire debut
        Flag[i] := true;
        tour := i ;
        tant que (Flag[j] et tour = i) faire rien;
        Section Critique;
        Flag[i] := false;
        Section Non Critique
        fin
fin
```

La preuve sera faite en TD

Section critique pour N processus. Solutions complexes et difficiles à prouver.

3.3 Les primitives matérielles de synchronisation.

Les solutions logicielles font les hypothèses suivantes :
- Interruption toujours possibles.
- Actions atomiques = Lecture, Ecriture.

Le problème devient plus simple si on les change.
- Permettre à l'utilisateur d'interdire momentanement les interruptions (difficile et dangereux).
- Augmenter l'ensemble des actions atomiques (le jeu d'instructions).

fonction TestAndSet permet de modifier un booléen s'il vaut false.

```
fonction TestAndSet(var but : booleen) : booleen
debut TestAndSet := but; but := true; fin
```

Une solution non équitable pour N processus au problème de la section critique avec un booléen lock initialise à false.

```
debut
tant que vrai faire debut
        tant que TestAndSet(lock) faire rien
        Section Critique
        lock := false
        Section Non Critique
        fin
fin
```

Certains processeurs possèdent une instruction TestAndSet.

procedure swap permet l'échange de deux booléens.

```
procedure swap(var a, b : booleen)
debut aux := a; a := b; b := aux fin
```

Une solution non équitable pour N processus au problème de la section critique avec un booléen lock et N booléens key[i] initialises à false.

```
debut
tant que vrai faire debut
        key[i] := true;
        tant que key[i] faire
        swap(lock,key[i]);
        Section Critique;
        lock := false
        Section Non Critique
        fin
fin
```

Certains processeurs possèdent une instruction swap.

les procedures wait et signal manipulent des entiers.

```
procedure wait(var S : entier)
debut
tant que (S <= 0) faire rien;
S := S - 1;
fin
```

```
procedure signal(var S : entier)
debut
S := S + 1
fin
```

Une solution non équitable pour N processus au problème de la section critique avec un entier mutex initialisé à 1.

```
debut
tant que vrai faire debut
        wait(mutex);
        Section Critique;
        signal(mutex);
        Section Non Critique
        fin
fin
```

Aucun processeur ne possède ces instructions.

Il existe des solutions équitables.

Le défaut commun à ces solutions est l'attente active (gaspillage de l'UC).

3.4 Les sémaphores : Dijkstra - 1965

L'idée est d'utiliser les appels systèmes `block` et `wakeup` dans les procédures `wait` et `signal` afin d'éliminer l'attente active.

3.4.1 Définition

Une structure de données

```
type semaphore = (n : entier; l : liste de processus);
```

manipulée uniquement par les actions atomiques `wait` et `signal`.

```
procedure wait(var S : semaphore)
debut
S.n := S.n - 1;
si (S.n < 0) alors debut
        ajouter(moi,S.l);
        block;
        fin
fin
```

```
procedure signal(var S : semaphore)
debut
S.n := S.n + 1;
si (S.n <= 0) alors debut
        enlever(P,S.l);
        wakeup(P);
        fin
fin
```

3.4.2 Une solution au problème de la section critique

Une solution **équitable** pour N processus au problème de la section critique avec un sémaphore mutex initialisé à (1,vide).

```
debut
tant que vrai faire debut
        wait(mutex);
        Section Critique;
        signal(mutex);
        Section Non Critique
        fin
fin
```

La solution est équitable si la liste de processus est une FIFO.

3.4.3 Implémentation

Les sémaphores peuvent être implémentés de la façon suivante :

- Un sémaphore de taille m est implémenté à l'aide de 3 sémaphores binaires et d'un entier.
  Les opérations `wait` et `signal` des sémaphores binaires sont implémentés avec les instructions `TestAndSet` et/ou `Swap`
- Les interruptions sont interdites pendant l'exécution des `wait` et `signal`.

3.5 Les messages

Le système propose les primitives

- emettre(message, destinataire)
- recevoir(message, source).

3.5.1 Une solution au problème de la section critique.

utilse un processus controleur qui délivre les autorisations d'entrée en SC.

```
processus Pi
debut
tant que vrai faire debut
        emettre(demande,controleur);
        recevoir(aquittement,controleur);
```

```
        Section Critique;
        emettre(fin,controleur);
        Section Non Critique
        fin
fin

processus Controleur
debut
tant que vrai faire
        cas de
        recevoir(demande,P1); emettre(aquittement,P1); recevoir(fin,P1);
        ou
        recevoir(demande,P2); emettre(aquittement,P2); recevoir(fin,P2);
        ou
        ...
        ou
        recevoir(demande,Pn); emettre(aquittement,Pn); recevoir(fin,Pn);
        sac
fin
```

3.5.2 Implémentation

Les protocoles pour mettre en œuvre le rendez-vous ne sont pas simples.

3.6 Les moniteurs

Définition : Un moniteur comprend des variables, des conditions, des procédures et un code d'initialisation. Seules les procédures peuvent être manipulées par les processus, et le mécanisme du moniteur garantit l'exécution de chaque procédure en exclusion mutuelle.

Syntaxe d'un moniteur

type *nom du moniteur* = monitor
déclarations de variables
déclarations de conditions
procedure entry *P1(...)*
begin ...end ;
⋮
procedure entry *Pn(...)*
begin ...end ;
begin
code d'initialisation
end.

La synchronisation des processus est obtenu à l'aide des procédures *wait* et *signal* appliquées sur les variables de type condition. L'action X.signal provoque le réveil de tous les processus qui ont auparavant effectué X.wait.

Une solution au problème de la section critique

type SectionCritique = monitor
token : booleen ;
sc : condition
procedure entry EntreeSC

```
        begin
        while not token do sc.wait ;
        token := false ;
        end ;
procedure entry SortieSC
        begin
        token := true ;
        sc.signal ;
        end ;
begin
token := true
end.
```

shared var SC = SectionCritique

Code des processus

```
begin
while true do begin
        code de la section non critique
        SC.EntreeSC ;
        code de la section critique
        SC.SortieSC
        end
end.
```

3.7 Equivalences

En fait, tous ces mécanismes sont équivalents au sens ou n'importe lequel permet d'implémenter tous les autres.

Par exemple, il est possible d'implémenter une région critique conditionnelle à l'aide des sémaphores mutex(1,vide), delai1(0,vide) et delai2(0,vide), et des entiers compteur1 et compteur2 initialisés à 0.

```
debut
wait(mutex);
tant que non B faire debut
        compteur1 := compteur1 + 1;
        si compteur2 > 0 alors
                signal(delai2);
        sinon
                signal(mutex);
        wait(delai1);
        compteur1 := compteur1 - 1;
        compteur2 := compteur2 + 1;
        si compteur1 > 0 alors
                signal(delai1);
        sinon
                signal(delai2);
        wait(delai2);
        compteur2 := compteur2 - 1;
        fin;
S;
si compteur1 > 0 alors
        signal(delai1);
```

```
sinon
        si compteur2 > 0 alors
                signal(delai2);
        sinon
                signal(mutex);
fin
```

3.8 Autres problèmes classiques de synchronisation

Il existe plusieurs problèmes de synchronisation de processus qui permettent de valider, et de comparer les facilités d'utilisation des différents mécanismes.

3.8.1 Le Producteur/Consommateur avec buffer limité

Un producteur dépose des messages dans un buffer, duquel un comsommateur les retire. La synchronisation doit satisfaire les points suivants :

- Le producteur ne peut pas déposer un message lorsque le buffer est plein.
- Le consommateur ne peut pas retirer un message lorsque le buffer est vide.

Une solution avec deux sémaphores

Deux sémaphores `vide (0,vide)` et `plein (n,vide)`

```
Processus producteur
debut
tant que vrai faire debut
        wait(plein);
        deposer(message);
        signal(vide);
        fin
fin

Processus consommateur
debut
tant que vrai faire debut
        wait(vide);
        retirer(message);
        signal(plein);
        fin
fin
```

3.8.2 Les lecteurs/rédacteurs

Un objet est partagé par des lecteurs qui veulent connaître sa valeur, et des rédacteurs qui peuvent modifier sa valeur.

- Plusieurs lectures sont possibles simultanément.
- Une seule modification est permise à la fois, et seulement si aucune lecture n'est en cours.
- Une priorité soit aux lecteurs, soit aux rédacteurs.

Une solution avec un moniteur à un problème plus simple

Le moniteur suivant protège la lecture d'une variable `a` non initialisée.

```
sync : moniteur;                    |  processus P
  var fait : boolean;               |    debut
      fini : condition;             |    ecrire(a);
procedure finEcrire;                |    sync.finEcrire;
  debut                             |    fin
  fait := vrai;                     |
  fini.signaler                     |  processus Q
  fin                               |    debut
procedure debutLire;                |    sync.debutLire;
  debut                             |    lire(a);
  si (non fait) alors               |    fin
    fini.attendre
  fin
procedure init;
  debut
  fait := faux
  fin
```

Une solution avec un moniteur

```
fichier : moniteur;
  var : ecrire : booleen;
        nbLecteurs : entier;
        cEcrire, cLire : condition;
procedure debutLire;                | processus lecteur;
  debut                             |   debut
  nbLecteurs := nbLecteurs + 1;     |   fichier.debutLire;
  si ecrire alors                   |   < accès aux données protégées >
     debut                          |   fichier.finLire;
     cLire.attendre;                |   fin
     cLire.signaler;                |
     fin                            |
  fin                               | processus redacteur;
procedure finLire;                  |   debut
  debut                             |   fichier.debutEcrire;
  nbLecteurs := nbLecteurs - 1;     |   < accès aux données protégées >
  si (nbLecteurs = 0) alors         |   fichier.finEcrire;
     cEcrire.signaler               |   fin
  fin
procedure debutEcrire;
  debut
  si (ecrire ou (nbLecteurs > 0)) alors
     cEcrire.attendre;
  ecrire := vrai;
  fin
procedure finEcrire;
  debut
  ecrire := faux;
  si (nbLecteurs > 0) alors
     cLire.signaler;            // priorité aux lecteurs.
  sinon
     cEcrire.signaler;
  fin
procedure init;
```

```
debut
ecrire := faux;
nbLecteurs := 0;
fin
```

3.8.3 Les philosophes mangeurs de spaghettis

N philosophes alternativement pensent et mangent. Pour manger, ils disposent d'une table circulaire ayant N assiettes et N fourchettes.

– Un philosophe ne mange que lorsqu'il a deux fourchettes.

Il n'existe pas de contrôle réparti possible pour ce problème sans faire d'hypothèses supplémentaires. (priorité, accès limité à N-1 à la table).

Chapitre 4

JAVA : un langage parallèle

4.1 Les threads

Pour le système d'exploitation, un programme Java est un processus. Pour la machine virtuelle Java, un programme Java est souvent composé de plusieurs *threads* qui s'exécutent en parallèle, et est donc perçu comme ceci :

- le contexte du processus, commun à tous les threads,
- des objets communs à tous les threads,
- pour chaque threads :
  - des objets propres,
  - une pile pour l'exécution,
  - un tas pour l'exécution.

Un programme Java est donc spécifié par l'ensemble des exécutions concurrentes des différents threads qui le composent.

4.1.1 Les threads Java

C'est soit une instance de la classe `Thread`, soit une instance d'une classe qui implémente l'interface `Runnable`. On note :

threadID un identificateur unique,

une priorité manipulable par `getPriority()` et `setPriority(nb)`

les méthodes `init()`, `start()`, `stop()`, `destroy()`, `run()`, `yield()`, `join()`, `sleep()`, `sleep(duree)`, ...

4.1.2 Un exemple

La classe PingPong

```
class PingPong extends Thread {
  String word;
  int delay;

  PingPong(String whatToSay, int delayTime ) {
    word = whatToSay;
    delay = delayTime;
    }

  public void run() {
    try {
```

```
        for (;;) {
          System.out.print(word + " ");
          sleep(delay);
          }
      }
      catch (InterruptedException e) {
        return;
      }
    }

    public static void main(String []argv) {
      new PingPong("ping",33).start();
      new PingPong("PONG",100).start();
      }
  }
```

4.1.3 L'ordonnanceur Java

La machine virtuelle Java choisit dans un programme Java le thread en cours d'exécution. L'algorithme est de type tourniquet à plusieurs niveaux. Il choisit donc pour un temps déterminé, un thread parmi les plus prioritaires, puis recommence.

4.2 La synchronisation en Java

Elle est basée sur une implémentation des moniteurs de Hoare d'une part, et des insctructions conditionnelles d'autre part.

4.2.1 un block synchronisé

Le block d'instruction est ininterruptible par toute instruction qui essaierai d'accéder à l'objet sur lequel porte la condition.

```
synchronized(compte){
   aux = compte;
   aux = aux + 10;
   compte = aux;
   }
```

Intuitivement, ce block d'instruction s'exécute ainsi :

```
  {
   compte.wait();
   aux = compte;
   aux = aux + 10;
   compte = aux;
   compte.signal();
   }
```

Les méthodes wait() et signal() étant applicables sur des objets *synchronisés*

Attention, pour ne pas avoir de problèmes, il faut que tous les threads qui manipulent `compte` le fasse dans un block synchronisé.

4.2.2 un moniteur

C'est un objet dont les attributs sont non accessibles, et toutes ses méthodes publiques sont synchronisées.

```
public class Compte {
    private int contenu = 0;
    public Compte(int init) {
        contenu = init;
    }
    public synchronized String toString() {
        return " valeur compte: " + contenu;
    }
    public synchronized int getCompte() {
        return contenu;
    }
    public synchronized boolean modifCompte(int modif) {
        if (contenu+modif >0) {
           contenu += modif;
           return true;
           }
        else
           return false;
    }
}
```

Dans un moniteur, deux méthodes synchronisées ne peuvent s'exécuter simultanément. Ainsi toute instance de la classe `Compte` contient une variable entière manipulable uniquement en exclusion mutuelle.

Chapitre 5

Les utilisations possibles

5.1 Avertissement

Attention, les sémaphores et toutes les primitives de synchronisation permettent de synchroniser des processus, mais en aucun cas ne garantissent que cette synchronisation soit exempte de problème.

Soient les deux sémaphores S1(1,vide) et S2(1,vide). les processus P1 et P2 suivant peuvent se bloquer mutuellement.

```
processus P1                  processus P2
debut                         debut
wait(S1);                     wait(S2);
wait(S2);                     wait(S1);
...                           ...
fin                           fin
```

5.2 L'ordonnancement de tâches

Les sémpahores permettent l'ordonnancement de tâches.

5.2.1 Un premier exemple : un graphe acyclique

Une solution consiste à utiliser un sémaphore (initialisé à 0) par arête dans le graphe d'ordonnancement des tâches.

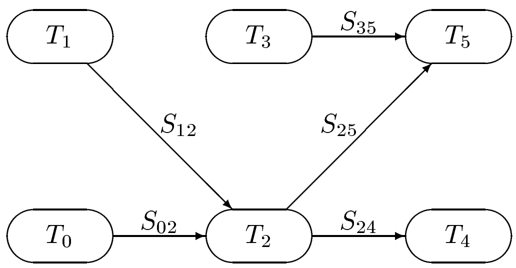

Figure 5.1: Ordonnancement avec 5 sémaphores.

Avec ces sémaphores, l'ordonnancement est obtenu, si chaque tâche exécute les codes suivants.

T_0	T_1	T_2	T_3	T_4	T_5
code	code	P(S_{02})	code	P(S_{24})	P(S_{25})
V(S_{02})	V(S_{12})	P(S_{12})	V(S_{35})	code	P(S_{35})
		code			code
		V(S_{24})			
		V(S_{25})			

5.2.2 Un second exemple : un graphe quelconque

Une solution consiste à utiliser :

- un sémaphore (initialisé à 1) pour une arête dans chaque cycle élémentaire du graphe d'ordonnancement des tâches.
- un sémaphore (initialisé à 0) pour toutes les autres arêtes.

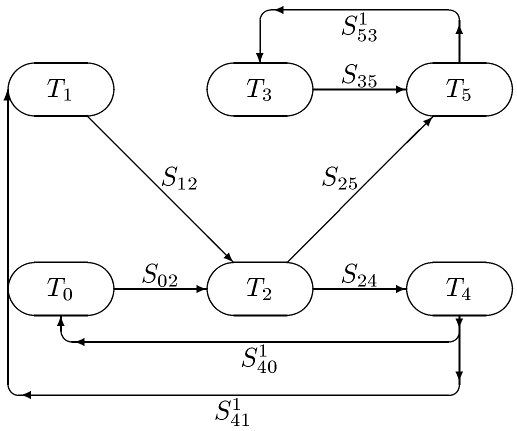

Figure 5.2: Ordonnancement avec 8 sémaphores.

Avec ces sémaphores, l'ordonnancement est obtenu, si chaque tâche exécute les codes suivants.

T_0	T_1	T_2	T_3	T_4	T_5
P(S_{40})	P(S_{41})	P(S_{02})	P(S_{53})	P(S_{24})	P(S_{25})
code	code	P(S_{12})	code	code	P(S_{35})
V(S_{02})	V(S_{12})	code	V(S_{35})	V(S_{40})	code
		V(S_{24})		V(S_{41})	V(S_{53})
		V(S_{25})			

5.2.3 Mise en œuvre en Java

La classe Semaphore

```
class Semaphore {
    private int n = 1;
    Semaphore(int val){
        if (val >= 0)
            n = val;
    }

    synchronized void p() throws InterruptedException { // autre nom de wait()
        n--;
        if (n < 0)
            wait();
```

```
    }

    synchronized void v() { // autre nom de signal()
        n++;
        // if (n <= 0) // ce test est inutile car notify est sans effet sinon.
        notify();
    }
 }
```

La classe Tache

```
class Tache extends Thread {

    private Semaphore[] amont;
    private Semaphore[] aval;
    private int threadId;
    private int duree;

    Tache(String who,
          int id,
          int duree,
          Semaphore[] entrant,
          Semaphore[] sortant) {
        setName(who);
        threadId = id;
        this.duree = duree;
        amont = entrant;
        aval = sortant;
    }

    public void run() {
        try {
            // boucle infinie avec un compteur de passage
            for(int occurence=1; ; occurence++){
                // protocole de debut de tâche
                for(int i = 0; i < amont.length; i++)
                    amont[i].p();
                // corps de la tâche
                System.out.println("Tache "+getName()+
                                   " : debut de l'occurence "+occurence);
                try{
                    sleep(duree);
                }
                catch(InterruptedException e){}
                System.out.println("               ... fin de la tache "+
                                   getName());
                // protocole de fin de tâche
                for(int i = 0; i < aval.length; i++)
                    aval[i].v();
            }
        }
        catch(InterruptedException ie){}
        finally{}
    }
}
```

La classe Ordonnanceur

```java
import java.util.*;

class Ordonnanceur {

    // Cet algorithme est correct si :
    // dans chaque composante fortement connexe élémentaire,
    // il y a un arc dont le semaphore est initialise à 1".

    private Tache[] tache;

    Ordonnanceur() {
        graphe();
        for (int i=0; i<tache.length; i++)
            tache[i].start();
    }

    private void graphe(){
        Semaphore[][] semaphore = new Semaphore[6][6];
        semaphore[0][2] = new Semaphore(0);
        semaphore[1][2] = new Semaphore(0);
        semaphore[2][4] = new Semaphore(0);
        semaphore[2][5] = new Semaphore(0);
        semaphore[3][5] = new Semaphore(0);
        semaphore[4][0] = new Semaphore(1);
        semaphore[4][1] = new Semaphore(1);
        semaphore[5][3] = new Semaphore(1);
        Semaphore[] avant, apres;
        tache = new Tache[6];
        // la tâche 0
        avant = new Semaphore[1];
        avant[0] = semaphore[4][0];
        apres = new Semaphore[1];
        apres[0] = semaphore[0][2];
        tache[0] = new Tache("T0",0,50,avant,apres);
        // la tâche 1
        avant = new Semaphore[1];
        avant[0] = semaphore[4][1];
        apres = new Semaphore[1];
        apres[0] = semaphore[1][2];
        tache[1] = new Tache("T1",1,75,avant,apres);
        // la tâche 2
        avant = new Semaphore[2];
        avant[0] = semaphore[0][2];
        avant[1] = semaphore[1][2];
        apres = new Semaphore[2];
        apres[0] = semaphore[2][4];
        apres[1] = semaphore[2][5];
        tache[2] = new Tache("T2",0,25,avant,apres);
        // la tâche 3
        avant = new Semaphore[1];
        avant[0] = semaphore[5][3];
        apres = new Semaphore[1];
```

```
        apres[0] = semaphore[3][5];
        tache[3] = new Tache("T3",0,50,avant,apres);
        // la tâche 4
        avant = new Semaphore[1];
        avant[0] = semaphore[2][4];
        apres = new Semaphore[2];
        apres[0] = semaphore[4][0];
        apres[1] = semaphore[4][1];
        tache[4] = new Tache("T4",0,25,avant,apres);
        // la tâche 5
        avant = new Semaphore[2];
        avant[0] = semaphore[2][5];
        avant[1] = semaphore[3][5];
        apres = new Semaphore[1];
        apres[0] = semaphore[5][3];
        tache[5] = new Tache("T5",0,75,avant,apres);
    }

    public static void main(String []args) {
        new Ordonnanceur();
    }
}
```

Un ordonnancement possible

```
Tache T0 : debut de l'occurence 1
Tache T1 : debut de l'occurence 1
Tache T3 : debut de l'occurence 1
              ... fin de la tache T0
              ... fin de la tache T3
              ... fin de la tache T1
Tache T2 : debut de l'occurence 1
              ... fin de la tache T2
Tache T4 : debut de l'occurence 1
Tache T5 : debut de l'occurence 1
              ... fin de la tache T4
Tache T0 : debut de l'occurence 2
Tache T1 : debut de l'occurence 2
              ... fin de la tache T5
Tache T3 : debut de l'occurence 2
              ... fin de la tache T0
              ... fin de la tache T1
Tache T2 : debut de l'occurence 2
              ... fin de la tache T3
              ... fin de la tache T2
Tache T4 : debut de l'occurence 2
Tache T5 : debut de l'occurence 2
              ... fin de la tache T4
Tache T0 : debut de l'occurence 3
Tache T1 : debut de l'occurence 3
              ... fin de la tache T5
Tache T3 : debut de l'occurence 3
              ... fin de la tache T0
              ... fin de la tache T1
```

```
Tache T2 : debut de l'occurence 3
              ... fin de la tache T3
              ... fin de la tache T2
Tache T4 : debut de l'occurence 3
Tache T5 : debut de l'occurence 3
```

5.3 Le partage de ressources indistinguables

Un sémaphore S(p,vide) permet le partage de p ressources indistinguables, en utilisant l'algorithme pour l'exclusion mutuelle.

5.3.1 Exemple : un stock d'outils en prêt

Les sémaphores `marteau(10,vide)`, `scie(5,vide)`, `echelle(2,vide)`…et les processus :

```
Processus Ouvrier
debut
wait(marteau);
wait(echelle);
<premier travail);
signal(echelle);
wait(scie);
<second travail>;
signal(scie);
signal(marteau);
fin
```

Attention aux interblocages, les solutions pour éviter les interblocages sont difficiles (algorithme du banquier). Prendre les ressources toujours dans le même ordre minimise les risques.

5.3.2 Mise en œuvre en Java

La classe Macon

```java
class Macon extends Thread {
    static int numero = 1;
    private int threadId;
    private Chantier chantier;
    Macon(Chantier lieu) {
        setName("Macon");
        threadId = numero++;
        chantier = lieu;
    }

    public void run() {
        try {
            for(;;){
                // premier travail
                chantier.metre.p();
                System.out.println(getName()+" "+threadId+" mesure.");
                try{sleep(50);}
                catch(InterruptedException e){}
                chantier.metre.v();
                // second travail
```

```
                chantier.echelle.p();
                chantier.marteau.p();
                chantier.truelle.p();
                System.out.println(getName()+" "+threadId+" monte un mur.");
                try{sleep(200);}
                catch(InterruptedException e){}
                chantier.truelle.v();
                chantier.marteau.v();
                chantier.echelle.v();
            }
        }
        catch(InterruptedException ie){}
        finally{}
    }
}
```

La classe Chantier

```
import java.util.*;

class Chantier {

    public Semaphore clef, echelle, marteau, metre, tournevis, truelle;

    Chantier() {
        truelle = new Semaphore(2);
        marteau = new Semaphore(4);
        echelle = new Semaphore(3);
        metre = new Semaphore(2);
        tournevis = new Semaphore(3);
        clef = new Semaphore(3);
        new Macon(this).start();
        new Macon(this).start();
        new Electricien(this).start();
        new Electricien(this).start();
        new Carreleur(this).start();
        new Menuisier(this).start();
    }

    public static void main(String []args) {
        new Chantier();
    }
}
```

Une planification possible du chantier

```
Macon 1 mesure.
Macon 2 mesure.
Electricien 1 pose les plafonniers.
Electricien 2 pose les plafonniers.
Macon 1 monte un mur.
Carreleur 1 pose le carrelage au sol.
Electricien 1 puis les prises.
Electricien 2 puis les prises.
Macon 1 mesure.
```

```
Macon 2 monte un mur.
Menuisier 1 pose les portes.
Macon 1 monte un mur.
Electricien 1 pose les plafonniers.
Carreleur 1 puis le carrelage mural.
Electricien 1 puis les prises.
Macon 2 mesure.
Macon 2 monte un mur.
Electricien 2 pose les plafonniers.
Macon 1 mesure.
Menuisier 1 et les fenetres.
Carreleur 1 pose le carrelage au sol.
Macon 1 monte un mur.
Macon 2 mesure.
```

5.3.3 Ordonnancement et partage de ressources

Il suffit de combiner les deux techniques précédentes.

5.4 L'animation graphique

L'animation consiste généralement à dessiner *souvent* une image, celle-ci ayant plus ou moins été modifiée entre-temps.

La programmation parallèle permet le découpage de l'animation en plusieurs activités : d'une part le calcul de la nouvelle image, d'autre part son affichage.

5.4.1 L'exemple d'une horloge

Cet exemple est en très grande partie décrit dans le livre *Java : de l'esprit à la méthode. M. Bonjour, G. Falquet, J. Guyot et A. Le Grand (International Thomson Publishing)*

L'affichage de l'heure se fait dans un applet. Dans un applet, la méthode `repaint()` est appelée chaque fois que cela est nécessaire ; et son code fait un appel à la méthode `update`, puis à la méthode `paint()`. Ce sont ces deux méthodes que l'on peut redéfinir.

Une premiere horloge dans un applet

```
import java.awt.*;
import java.util.Date;

public class Animation
    extends java.applet.Applet {

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;

    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.setFont(timesRoman24Gras);
        maintenant = new Date();
        g.drawString(maintenant.toString(),10,40);
    }
}
```

Defauts : Il n'y a pas d'animation. Pour avoir l'heure il faut *jouer* avec l'applet.

Une seconde horloge dans un applet

```
import java.awt.*;
import java.util.Date;

public class Animation
    extends java.applet.Applet {

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;

    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.setFont(timesRoman24Gras);
        while (true) {
            maintenant = new Date();
            g.drawString(maintenant.toString(),10,40);
        }
    }
}
```

Defauts : Il y a de l'animation, mais l'heure devient illisible.

Une troisième horloge dans un applet

```
import java.awt.*;
import java.util.Date;

public class Animation
    extends java.applet.Applet {

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;

    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.setFont(timesRoman24Gras);
        while (true) {
            maintenant = new Date();
            g.clearRect(0,0,size().width,size().height);
            g.drawString(maintenant.toString(),10,40);
        }
    }
}
```

Defauts : Il y a de l'animation, mais trop. Cela donne un effet de clignotement.

Une quatrième horloge dans un applet

```
import java.awt.*;
import java.util.Date;

```

```
public class Animation
    extends java.applet.Applet {

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;

    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.setFont(timesRoman24Gras);
        while (true) {
            maintenant = new Date();
            g.clearRect(0,0,size().width,size().height);
            g.drawString(maintenant.toString(),10,40);
            try{Thread.sleep(1000);}
            catch(InterruptedException e){}
        }
    }
}
```

Defauts : Tout semble parfait, mais le contrôle de l'applet est en fait un peu perdu. Par exemple, la réaction à la destruction est très lente.

Une cinquième horloge dans un applet

```
import java.awt.*;
import java.util.Date;

public class Animation
    extends java.applet.Applet
    implements Runnable{

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;
    Thread actif;

    public void start() {
        if (actif == null) {
            actif = new Thread(this);
            actif.start();
        }
    }

    public void stop() {
        if (actif != null) {
            actif.stop();
            actif = null;
        }
    }

    public void run() {
        while (true) {
            repaint();
            try{Thread.sleep(1000);}
            catch(InterruptedException e){}
```

```
        }
    }

    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.setFont(timesRoman24Gras);
        maintenant = new Date();
        g.drawString(maintenant.toString(),10,40);
    }
}
```

Defauts : Aucun, tout est parfait.

Une sixième horloge dans un applet

```
import java.awt.*;
import java.util.Date;

public class Animation
    extends java.applet.Applet
    implements Runnable{

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;
    Thread actif;
    Image imgTampon;
    Graphics gTampon;

    public void init() {
        imgTampon=createImage(size().width,size().height);
        gTampon=imgTampon.getGraphics();
    }

    public void start() {
        if (actif == null) {
            actif = new Thread(this);
            actif.start();
        }
    }

    public void stop() {
        if (actif != null) {
            actif.stop();
            actif = null;
        }
    }

    public void run() {
        gTampon.setColor(Color.red);
        gTampon.setFont(timesRoman24Gras);
        while (true) {
            gTampon.clearRect(0,0,size().width,size().height);
            maintenant = new Date();
            gTampon.drawString(maintenant.toString(),10,40);
```

```
                repaint();
                try{Thread.sleep(1000);}
                catch(InterruptedException e){}
            }
        }

        public void paint(Graphics g) {
            g.drawImage(imgTampon,0,0,this);
        }
    }
```

Avantages : Utilisation d'un double buffer.

Une septième horloge dans un applet

```
import java.awt.*;
import java.util.Date;

public class Animation
    extends java.applet.Applet
    implements Runnable{

    Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
    Date maintenant;
    Thread actif;
    Image imgTampon;
    Graphics gTampon;

    public void init() {
        imgTampon=createImage(size().width,size().height);
        gTampon=imgTampon.getGraphics();
        System.out.println("parametres " + paramString());
    }

    public void start() {
        if (actif == null) {
            actif = new Thread(this);
            actif.start();
        }
    }

    public void stop() {
        if (actif != null) {
            actif.stop();
            actif = null;
        }
    }

    public void run() {
        gTampon.setColor(Color.red);
        gTampon.setFont(timesRoman24Gras);
        while (true) {
            gTampon.clearRect(0,0,size().width,size().height);
            maintenant = new Date();
```

```
                gTampon.drawString(maintenant.toString(),10,40);
                repaint();
                try{Thread.sleep(1000);}
                catch(InterruptedException e){}
            }
        }

        public void update(Graphics g) {
            //g.clipRect(0,0,size().width,size().height); //450,100
            g.clipRect(205,0,150,100);
            // seul le chiffre unité des secondes est redessiné
            paint(g);
        }

        public void paint(Graphics g) {
            g.drawImage(imgTampon,0,0,this);
        }
    }
```

Avantages : Restriction de la zone réaffichée.

5.5 Les applications client-serveur

Les deux exemples sont en très grande partie décrits dans le livre *Java : de l'esprit à la méthode. M. Bonjour, G. Falquet, J. Guyot et A. Le Grand (International Thomson Publishing)*

5.5.1 L'exemple d'un service de traduction

Une connexion simple de type Client-Serveur : le client envoie un message qui est retourné en majuscule. Le client termine la connexion par le mot "fin".

Un tel serveur maintient une connexion avec chaque client sans avoir une veritable gestion des clients. Il ne mémorise pas (ou très peu) d'informations sur eux.

L'architecture d'un serveur sans état

Mise en œuvre en Java

Le serveur

```
import java.io.*;
import java.net.*;

public class Serveur extends Thread {

    protected ServerSocket ecoute;

    public Serveur(int port) {
        try {
            ecoute = new ServerSocket(port);
        }
        catch(IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Serveur en ecoute sur le port : "+port);
        this.start();
    }

    public void run() {
        try {
            while (true) {
                Socket client = ecoute.accept();
                Connexion c = new Connexion(client);
            }
        }
        catch(IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void main(String[] argv) {
        if (argv.length != 1) {
            System.err.println("Usage : java Serveur <port>");
        }
        else {
            new Serveur(Integer.parseInt(argv[0]));
        }
    }
}
```

La connexion

```
import java.io.*;
import java.net.*;

class Connexion extends Thread {

    protected Socket client;
    protected BufferedReader in;
```

```
    protected PrintStream out;

    public Connexion(Socket clientSocket) {
        client = clientSocket;
        try {
            in = new BufferedReader(new InputStreamReader(client.getInputStream()));
            out = new PrintStream(client.getOutputStream());
        }
        catch(IOException e) {
            try {client.close();}
            catch(IOException el) {}
            System.err.println(e.getMessage());
        }
        this.start();
    }

    public void run() {
        String ligne;
        try {
            while (true) {
                ligne = in.readLine();
                if (ligne.toUpperCase().compareTo("FIN")==0)
                    break;
                out.println(ligne.toUpperCase());
            }
        }
        catch(IOException e) {
            System.out.println("connexion :"+e.toString());
        }
        finally {
            try {client.close();}
            catch(IOException e) {}
        }
    }
}
```

Le client

```
import java.io.*;
import java.net.*;

public class Client {

    public static void main(String[] argv) {
        Socket s=null;
        if (argv.length != 2) {
            System.err.println("Usage : java Client <hote> <port>");
        }
        else {
            try {
                s = new Socket(argv[0],Integer.parseInt(argv[1]));
                BufferedReader canalLecture =
                    new BufferedReader(new InputStreamReader(s.getInputStream()));
                BufferedReader console =
```

```
                    new BufferedReader(new InputStreamReader(System.in));
                PrintStream canalEcriture =
                    new PrintStream(s.getOutputStream());
                System.out.println("Connexion etablie : " +
                                   s.getInetAddress()+ " port : " +
                                   s.getPort());
                String ligne;
                while (true) {
                    System.out.print("?");System.out.flush();
                    ligne = console.readLine();
                    canalEcriture.println(ligne);
                    ligne=canalLecture.readLine();
                    if (ligne==null) {
                        System.out.println("Connexion terminee");
                        break;
                    }
                    System.out.println("!"+ligne);
                }
            }
            catch (IOException e) {
                System.err.println(e);
            }
            finally {
                try {
                    if (s!=null)
                        s.close();
                }
                catch (IOException e2) {}
            }
        }
    }
}
```

Une session possible

```
[alain@localhost Exo1]$ java Serveur 5000&
[1] 4276
[alain@localhost Exo1]$ Serveur en ecoute sur le port : 5000

[alain@localhost Exo1]$ java Client localhost 5000
Connexion etablie : localhost/127.0.0.1 port : 5000
?gjglkgli
!GJGLKGLI
?;b;gl glkjgh:;, jhgguigilugy
!;B;GL GLKJGH:;, JHGGUIGILUGY
?jkk
!JKK
?fin
Connexion terminee
```

5.5.2 L'exemple d'un service de télédiscussion

Une télédiscussion : chaque client diffuse des messages vers tous les clients connectés.
Supposons que l'on souhaite développer l'application 'à partir du code précédent.

- La diffusion est difficile car le serveur ne connaît pas les clients, il faut donc une structure pour les enregistrer.
- La gestion de cette structure est délicate car le serveur ne connaît pas la durée de connexion des clients. Cette structure gérée par le serveur est en fait modifiée par les clients, elle doit donc pouvoir être verrouillée.
- Les messages doivent être reçus dans le même ordre par tous les clients. Ils faut donc que les différentes connexions se synchronisent.
- Un client décide de ses envois de messages, mais pas de ses réceptions, car elles sont déclenchées par les autres clients. Le client ne peut donc pas exécuter alternativement un envoi et une réception.
- Un client dissocie l'envoi d'un message et la réception des autres par l'utilisation de deux tâches distinctes.

Un tel serveur qui gère de l'informations sur les clients est dit avec état.

L'architecture d'un serveur avec état

Mise en œuvre en Java

Le serveur

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Serveur extends Thread {

    protected ServerSocket ecoute;
    protected Vector connexions;
    protected Nettoyeur nettoyeur;

    public Serveur(int port) {
        try {
            ecoute = new ServerSocket(port);
        }
        catch(IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Serveur en ecoute sur le port : "+port);
        connexions = new Vector();
        nettoyeur = new Nettoyeur(this);
        this.start();
    }

    public void run() {
        try {
            while (true) {
                Socket client = ecoute.accept();
                System.out.println("Demande de connexion... ");
                Connexion c = new Connexion(client, nettoyeur, this);
                synchronized (connexions) {connexions.addElement(c);}
                c.start();
            }
        }
        catch(IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void main(String[] argv) {
        if (argv.length != 1)
            System.err.println("Usage : java Serveur <port>");
        else
            new Serveur(Integer.parseInt(argv[0]));
    }
}
```

La connexion

```
import java.io.*;
```

```
import java.net.*;
import java.util.*;

class Connexion extends Thread {

    protected Socket client;
    protected Serveur serveur;
    protected Nettoyeur nettoyeur;
    protected BufferedReader in;
    protected PrintStream out;

    public Connexion(Socket clientSocket, Nettoyeur n, Serveur s) {
        client = clientSocket;
        nettoyeur = n;
        serveur = s;
        try {
            in = new BufferedReader(new InputStreamReader(client.getInputStream()));
            out = new PrintStream(client.getOutputStream());
        }
        catch(IOException e) {
            try {client.close();}
            catch(IOException el) {}
            System.err.println(e.getMessage());
        }
    }

    public void run() {
        String ligne;
        Connexion c;
        try {
            while (true) {
                ligne = in.readLine();
                // il faut se prémunir d’une propre déconnexion brutale.
                if (ligne != null)
                    synchronized(serveur.connexions) {
                        for (int i = 0; i<serveur.connexions.size();i++) {
                            c = (Connexion) serveur.connexions.elementAt(i);
                            // et des autres également.
                            if (c.isAlive())
                                c.out.println(ligne);
                        }
                    }
                if ((ligne == null)||(ligne.endsWith("FIN"))) // OU évalué
                    break;
            }
        }
        catch(IOException e) {}
        finally {
            try {client.close();}
            catch(IOException e) {}
            finally {
                synchronized(serveur.connexions) {
                    System.out.print("Fin de connexion...");
```

```
                    serveur.connexions.notify();
                }
            }
        }
    }
}
```

Le nettoyeur

```
import java.io.*;
import java.net.*;
import java.util.*;

class Nettoyeur extends Thread {

    protected Serveur serveur;

    protected Nettoyeur(Serveur s) {
        serveur = s;
        this.start();
    }

    public void run() {
        Connexion c;
        while (true) {
            try {
                synchronized(serveur.connexions)
                    serveur.connexions.wait();
            }
            catch(InterruptedException e) {}
            synchronized(serveur.connexions) {
                for (int i = serveur.connexions.size()-1;i>=0;i--) {
                    c = (Connexion) serveur.connexions.elementAt(i);
                    if (!c.isAlive()) {
                        serveur.connexions.removeElementAt(i);
                        System.out.println("OK");
                    }
                }
            }
        }
    }
}
```

Le client

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class Client extends Frame{

    Socket s;
    PrintStream canalEcriture;
    TextField entree;
```

```
    TextArea visu;
    Button envoi, stop;
    Panel boutons;
    String nom;
    Ecouteur ecoute;

    public Client(String hote, int port, String who) {
        super("Client "+who);
        try {
            nom = who;
            s = new Socket(hote,port);
            canalEcriture = new PrintStream(s.getOutputStream());
            entree = new TextField();
            entree.addActionListener(new ActionListenerEnvoi());
            visu = new TextArea();
            visu.setEditable(false);
            this.setLayout(new BorderLayout());
            this.add("North",visu);
            this.add("Center",entree);
            boutons = new Panel();
            envoi = new Button("envoi");
            envoi.addActionListener(new ActionListenerEnvoi());
            stop = new Button("stop");
            stop.addActionListener(new ActionListenerStop());
            boutons.add(envoi);
            boutons.add(stop);
            this.add("South",boutons);
            this.pack();
            this.show();
            visu.setText("Connexion : "+
                         s.getInetAddress()+
                         " port : " +
                         s.getPort());
            ecoute = new Ecouteur(s,visu);
        }
        catch (IOException e) {
            visu.setText(e.toString());
        }
    }

    public static void main(String[] argv) {
        Socket s=null;
        if (argv.length != 3)
            System.err.println("Usage : java Client <hote> <port> <nom>");
        else
            Frame f = new Client(argv[0],Integer.parseInt(argv[1]),argv[2]);
    }

    class ActionListenerEnvoi implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            canalEcriture.println(nom+">"+entree.getText());
            entree.setText("");
        }
```

```
    }

    class ActionListenerStop implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            canalEcriture.println(nom+">FIN");
            System.exit(0);
        }
    }
}
```

L'écouteur

```
import java.io.*;
import java.net.*;
import java.awt.*;

class Ecouteur extends Thread {

    BufferedReader entree;
    TextArea visu;

    public Ecouteur(Socket s,TextArea out) {
        try {
            entree = new BufferedReader(new InputStreamReader(s.getInputStream()));
            visu = out;
            this.start();
        }
        catch (IOException e) {System.err.println(e);}
    }

    public void run() {
        String ligne;
        try {
            while (true) {
                ligne = entree.readLine();
                if (ligne == null)
                    break;
                visu.append("\n"+ligne);
            }
        }
        catch(IOException e) {visu.setText(e.toString());}
        finally {visu.setText("connexion interrompue par le serveur");}
    }
}
```