Systèmes d'exploitation
(Module Maîtrise 00-01)
Université Bordeaux I

Alain Griffault

December 22, 2000

Index

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.

Il s'inspire fortement des notes de cours de Jean Betréma, qui dispensait ce module avant moi.

0.2 Bibliographie

1. Silberschatz, Galvin : Principes des systèmes d'exploitation.

2. Tanenbaum : Systèmes d'exploitation.

3. Beauquier, Bérard : Systèmes d'exploitation, concepts et algorithmes.

4. Krakowiak : Principes des systèmes d'exploitation des ordinateurs.

0.3 Les pré-requis

- le module USI
- le module AMP

0.4 Le contrôle

- le contrôle continu : Devoir surveillé proche des TD précédents.
- l'examen : Devoir surveillé (Cours, Exercice TD, Problème réflexion).

Chapter 1

Introduction.

1.1 Machine virtuelle.

La machine réelle comporte principalement : un ou plusieurs processeurs, de la mémoire, un bus, et des périphériques (clavier, souris, écran, disque(s), imprimante, carte(s) réseau).

Machine virtuelle = machine réelle + système d'exploitation ; seule utilisable.

Chaque tâche a l'illusion de disposer:

- d'un processeur,
- d'une mémoire adaptée à ses besoins,
- d'un système de fichiers,
- de communications standardisées avec les périphériques.

Le système d'exploitation peut également être défini comme le gestionnaire des ressources suivantes :

- les processus,
- la mémoire,
- le système de fichiers,
- les entrées/sorties.

D'autres fonctions importantes sont ou non considérées comme faisant partie du système d'exploitation :

- langage(s) de commande,
- interface graphique,
- communications (réseau).

Ce sont des services de plus haut niveau que les précédents : architecture en couches.

1.2 Historique des S.E.

Evolution des systèmes d'exploitation			
	Caractéristiques	Avantages	Inconvénients
Au tout début	Une seule console. Le programmeur est aussi l'opérateur	Très intéractif.	Le temps de préparation est long, et un seul utilisateur à la fois. L'UC est très mal utilisée.
Traitement par lots	L'opérateur est spécialisé et il regroupe les taches en lots.	Meilleure utilisation de l'UC	Plus d'intéractivité. Le "debuggage" passe par un vidage mémoire.
Job Control Language	Un moniteur résident qui enchaine les travaux.		
Traitements hors ligne	Il y a un pré et un post traitement des entrées et des sorties afin que lors de l'exécution, elles soient faites à partir de bandes.	Meilleure utilisation de l'UC qui est moins ralenti par les Entrées et les sorties.	Le temps de restitution du fait des traitements supplémentaires peut être moins bon.
Spooling	Les disques remplacent les bandes. L'accès direct vs l'accès séquentiel.	Le disque bufferise les entrées et les sorties. Le chevauchement des travaux pendant les entrées sorties est possible.	
La multiprogrammation	L'ordonnancement des travaux est réalisé par commutation au moment des entrées sorties, ou par l'utilisateur. Le SE prend des décisions.	Meilleure utilisation de l'UC.	
Le temps partagé	Commutation entrées sorties, et d'autres très fréquentes. L'utilisateur peut pendant l'exécution soit intervenir, soit passer des commandes au système d'exploitation	Très intéractif	L'utilisation de l'UC peut être moins bonne.

Des systèmes d'exploitation spécialisés			
	Caractéristiques	Avantages	Inconvénients
Ordinateur personnel	Sécurité inutile. Mono-tâche : MSDOS. Multi-programmation : Mac-Intosh.	Très intéractif et conviviaux.	Pas de protection du système. Virus
Parallèle multi-processeurs	Calcul vectoriel	Capacité de calcul. Fiabilité.	Programmation spécifique plus difficile.
Système réparti pour réseaux	Processus spécialisés. Découpage fonctionnel des tâches.	Partage des ressources. Fiabilité. Meilleure utilisation du matériel.	
Système temps-réel	Contraintes temporelles. Contrôle-Commande. Systèmes embarqués.		

1.3 Mécanismes d'exécution et de communication

1.3.1 Exécution séquentielle

Le mécanisme général qui permet l'exécution séquentielle d'un programme est le suivant :

- sauvegarde et restauration des contextes à l'appel et au retour des procédures,
- transmission de paramètres entre procédures appelante et appelée,
- gestion d'une zone de travail propre à chaque procédure.

1.3.2 Interruptions, déroutements, appels au superviseur

Pour la programmation d'un système d'exploitation il faut tenir compte de :

- l'asynchronisme, c'est à dire de l'effet sur le déroulement d'une activité, d'événements extérieurs à celle-ci,
- la protection mutuelle entre activités, qui nécessite lors du passage d'une activité à une autre, une modification du contexte plus profonde que pour un simple appel de procédure.

Le schéma d'exécution d'une commutation de contexte :

- Passage du processeur en mode privilégié (superviseur) ;
- Sauvegarde du contexte matériel : compteur ordinal, mot d'état du processeur ;
- Chargement du nouveau contexte depuis une adresse prédéfinie (table des déroutements) ;
- Fin de déroutement : restauration de l'ancien contexte (si l'ordonnanceur n'a pas décidé de changer le processus courant).

Le passage du processeur en mode superviseur modifie le fonctionnement de la mémoire, et permet donc l'accès à une partie de la mémoire invisible depuis un programme utilisateur: protection contre les erreurs de programmation ou les virus.

Certains processeurs possèdent plusieurs modes privilégiés : anneaux concentriques, points d'entrée contrôlés (gateways).

On distingue généralement trois mécanismes pour lesquels le *noyau* du système d'exploitation intervient.

- Déroutement (impossible d'exécuter l'instruction courante).
- Interruption (externe, vient du matériel via le bus).
- Appel au système (instruction de déroutement volontaire du programme).

Il y a souvent des confusions sur ces trois termes. Le terme déroutement unifie quelquefois les trois.

De même, il ne faut pas confondre les bibliothèques de fonctions (procédures) système, qui facilitent la vie (exemple : printf), et la liste (courte) des appels-systèmes.

Généralement, les interruptions sont désactivées pendant le traitement d'un déroutement.

1.4 Noyau et serveurs.

Le noyau est l'ensemble des procédures qui, en mode superviseur, traitent les déroutements ; il assure les fonctions de base : gestion des processus, de la mémoire, et des E/S de bas niveau. Les autres parties du système d'exploitation peuvent être réalisées sous formes de serveurs (exemples : NFS, X-Window). Unix a un gros noyau, on préfère aujourd'hui un micro-noyau (qui gère aussi les messages) et des serveurs. Solution bien adaptée aux réseaux.

Chapter 2

Gestion des Processus.

2.1 Un processus.

2.1.1 Définition.

Processus = programme (code exécutable statique) + contexte dynamique (compteur de programme, état processeur, espace mémoire, fichiers, ...).

2.1.2 Modèle d'un processus.

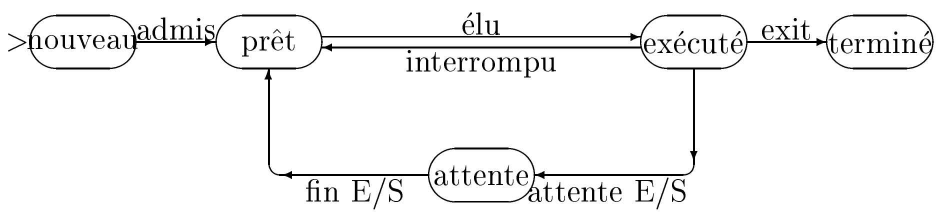

Figure 2.1: Les états et transitions d'un processus

2.2 L'ordonnancement de processus.

2.2.1 Les types d'ordonnancements.

Ils sont distingués suivant les transitions permises dans le graphe d'état d'un processus.

transition "interrompu" interdite sans réquisition (multiprogrammation).

transition "interrompu" autorisée avec réquisition (temps partagé).

2.2.2 Les critères d'ordonnancement.

- l'utilisation maximale de l'UC.
- la capacité de traitement maximale.
- l'utilisation maximale des périphériques.
- la diminution du temps moyen de restitution : le temps mis par un processus pour aller de l'état "nouveau" à l'état "terminé".
- la diminution du temps moyen d'attente : le temps passé dans l'état "prêt".
- la diminution du temps moyen de réponse : dans un processus interactif, le temps entre une fin d'entrée et le début d'une sortie de résultat (pas la fin). C'est une mesure de "réactivité".
- l'équité.

Les objectifs sont contradictoires, il faut des compromis.

2.2.3 Les différentes phases de l'ordonnancement.

le bloc de contrôle d'un processus

Il contient toute l'information nécessaire pour qu'un processus lors d'une reprise se comporte comme s'il n'avait pas été interrompu.

- l'état du processus,
- le compteur d'instructions,
- les registres de l'UC,
- des informations pour l'ordonnanceur (priorité, ...),
- des informations pour la gestion mémoire (valeur des registres base et limite, les tables de pages et/ou de segments ...),
- des informations de comptabilisation (temps processeur écoulé, temps réel écoulé, numéros des processus fils ...),
- des informations sur les E/S (liste des périphériques alloués, liste des fichiers ouverts ...).

la table des processus et les files d'attentes

C'est une structure de stockage pour l'ensemble des blocs de contrôle.

la commutation de l'UC entre processus

2.2.4 Les algorithmes d'ordonnancement.

FIFO. Utilisation maximale de l'UC. Équitable.

Tourniquet (Round Robin). Un paramètre supplémentaire : le quantum (une durée) qui permet d'interrompre un processus en cours d'exécution si sa prochaine E/S est trop éloignée dans le temps (supérieur au quantum).

Équitable. Utilisation de l'UC et temps moyen de réponse dépendent du quantum. Une grande valeur de quantum favorise les processus de type calculs, tandis que une petite valeur favorise les E/S, donc les processus interactifs.

Tourniquet avec priorités statiques. Chaque processus possède une priorité, et chaque priorité possède son propre quantum : grand pour les faibles priorités, et petit pour les hautes priorités. Généralement les processus de calculs sont de faible priorité et les processus interactifs de grande priorité.

Non équitable. Utilisation de l'UC et temps moyen de réponse peuvent être bons.

Tourniquet avec priorités dynamiques. Un paramètre supplémentaire (une durée, un nombre d'interruptions) qui permet de faire passer les processus de la file d'attente de priorité N à celle de priorité $N + 1$.

Équitable. Utilisation de l'UC et temps moyen de réponse sont bons. Diverses stratégies pour la gestion des priorités. Généralement, après une interruption, un processus réintègre la file d'attente correspondante à sa priorité initiale.

PCTE (Plus Court Temps d'Exécution). Non équitable. Temps moyen de restitution optimal. (analogie avec la file d'attente à une caisse de grande surface, où une règle de politesse est de laisser passer quelqu'un qui a peu d'objets si l'on a un plein caddie).

Exemple : deux tâches (A d'une durée de 5 minutes, et B de 30 secondes). A puis B donne un temps moyen de 5 minutes et 15 secondes, alors que B puis A donne seulement 3 minutes.

Implémentable dans les SGBD, mais pas dans un SE généraliste, car le temps du prochain cycle d'exécution n'est pas connu.

Une implémentation est possible en utilisant une fonction d'estimation du temps du prochain cycle. Par exemple t= a * t + b * te. (t : estimation, te : temps effectif, a+b=1).

PCTER (Plus Court Temps d'Exécution Restant). Une variante du précédent. Il y à réquisition pour un nouveau processus dont le temps d'exécution serait plus court que celui "restant" du processus en cours.

2.2.5 L'ordonnancement à plusieurs niveaux.

L'ensemble des processus prêts est souvent trop important pour tenir en RAM. Certains sont donc sur disque. Un processus élu qui est sur disque prend beaucoup plus de temps qu'un processus en RAM pour être chargé dans l'UC.

Le processus élu est donc toujours pris parmi ceux en RAM. L'algorithme est alors non équitable.

La solution consiste à utiliser un deuxième algorithme d'ordonnancement pour gérer les déplacements des processus prêts entre le disque et la RAM.

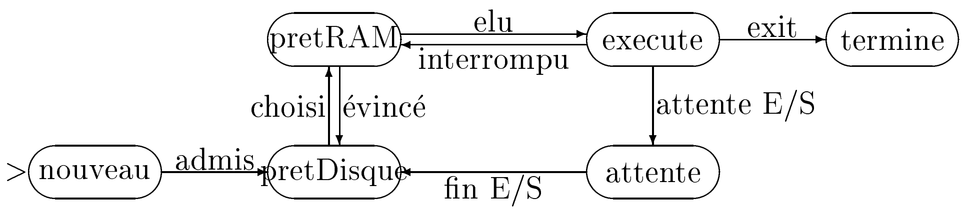

Figure 2.2: Deux niveaux d'ordonnancement

2.2.6 L'ordonnancement sur plusieurs processeurs.

Le problème est plus complexe, il n'y a pas de solution optimale.

2.3 La synchronisation (communication) de processus.

La mémoire est partagée entre plusieurs processus (exemple BD), il y a donc des risques de comportements bizarres.

Soient les deux processus suivant et la variable globale `compte` initialisée à 10.

Processus A	Processus B
var aux : integer	var aux : integer
begin	begin
aux := compte;	aux := compte;
aux := aux + 10;	aux := aux - 5;
compte := aux;	compte := aux;
end.	end.

La figure 2.3 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.

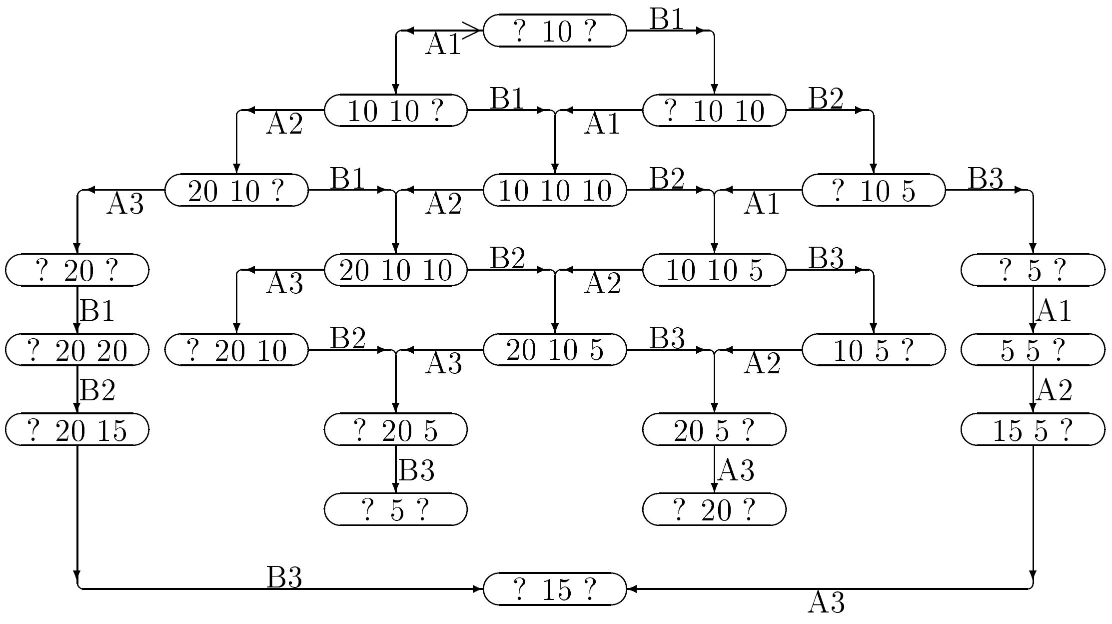

Figure 2.3: Les trois variables.

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.

2.3.1 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.*

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.

Le SE doit fournir des primitives pour synchroniser des processus.

Les primitives matérielles de synchronisation.

Les solutions logicielles font les hypothèses suivantes :

- Interruption toujours possibles.
- Actions atomiques = Lecture, Écriture.

Le problème devient plus simple si on les change.

- Permettre à l'utilisateur d'interdire momentanément 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] initialisés 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 procédures wait et signal manipulent des entiers.

  ```
  procedure wait(var S : entier)        procedure signal(var S : entier)
  debut                                 debut
  ```

```
tant que (S <= 0) faire rien;          S := S + 1
S := S - 1;                            fin
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).
Le SE doit fournir des appels systèmes pour endormir et réveiller des processus.

Les appels systèmes de synchronisation.

On suppose l'existence de deux appels systèmes: d'une part `block` qui permet à un processus de se mettre en état d'attente; d'autre part `wakeup(P)` qui permet de mettre l'état du processus P à prêt.

Avec ces hypothèses, il est assez facile d'éliminer l'attente active. Il suffit de remplacer `faire rien` par `block`, puis ajouter en fin de la phase de sortie une boucle pour réveiller tous les processus "en attente" grâce à un `wakeup(P)`

Cette solution est moyennement satisfaisante, car en fait un seul processus doit être réveillé, les autres feront en effet un nouveau `block`.

2.3.2 Les sémaphores : Dijkstra - 1965

L'idée est de combiner les appels `block` et `wakeup` avec les procédures `wait` et `signal`.

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
```

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.

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`.

Utilisation

- Attention, les sémaphores 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
  debut
  wait(S1);
  ```

  ```
  processus P2
  debut
  wait(S2);
  ```

```
wait(S2);                       wait(S1);
...                             ...
fin                             fin
```

- Les sémaphores permettent l'ordonnancement de tâches.
- Un sémaphore S(p,vide) permet le partage de p ressources indistinguables, en utilisant l'algorithme pour l'exclusion mutuelle.

2.3.3 Autres solutions

Il existe des solutions équivalentes (en terme de puissance de synchronisation) pour la synchronisation de processus.

Les messages

Le SE propose les primitives

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

Les S.E. conçus autour d'un micro-noyau utilise principalement cette technique.
Les messages permettent de synchroniser des processus distants.
Il existe des variantes en fonction des sémantiques :

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

Une solution au problème de la section critique. utilise un processus contrôleur 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
```

Les structures précédentes (sémaphores et messages) sont puissantes, mais encore difficiles d'utilisation. Les suivantes sont plus simples d'emploi.

Les régions critiques conditionnelles

Une variable partagée est déclarée par :

```
var v : shared T;
```

et ne peut être utilisée que de la façon suivante :

```
region v when B do S;
```

Une solution au problème de la section critique Les variables manipulées dans la section critique sont rassemblées dans une structure v déclarée partagée.

```
debut
tant que vrai faire debut
        region v when true do
               Section Critique;
        Section Non Critique
        fin
fin
```

Le risque, pour l'utilisateur, d'erreur d'entrelacement est nettement diminué.

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.
```

Équivalences

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
```

2.3.4 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.

Le Producteur/Consommateur avec buffer limité

Un producteur dépose des messages dans un buffer, duquel un consommateur 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.

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.

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 possible pour ce problème sans faire d'hypothèses supplémentaires. (priorité, accès limité à N-1 à la table).

2.3.5 Remarque

Le cours détaille le problème de la section critique et les sémaphores. N'importe quel autre problème, et n'importe quel autre mécanisme auraient permis la présentation de ce chapitre.

Chapter 3

Gestion de la mémoire

3.1 Le problème

La mémoire est assimilable à un tableau :

SE	Processus utilisateurs

Il faut gérer les emplacements utilisés par les processus utilisateurs, en essayant d'optimiser l'espace disponible.

3.2 Les critères de comparaison des solutions

3.2.1 Monoprogrammation versus multiprogrammation

La monoprogrammation ne permet qu'à un seul processus utilisateur d'être en mémoire à un instant donné, la gestion de la mémoire est énormément simplifiée. C'est le cas de MS-DOS par exemple.

La suite du cours est dans un cadre de multiprogrammation.

3.2.2 La taille d'un processus

La taille d'un processus n'est pas connue au début de l'exécution, il faut donc l'estimer. Les tailles des données statiques et du code sont connues au chargement, celles des données dynamiques (le tas et la pile) sont fixées arbitrairement.

Si un processus devient plus grand que la zone allouée, il faut soit agrandir l'emplacement, soit lui trouver un nouvel emplacement, soit le tuer.

3.2.3 La fragmentation

Définition 3.1 *La fragmentation interne est la quantité :*
$F_i = \sum(tailles\ processus)/\sum(tailles\ allouees)$

La fragmentation interne existe dès lors que l'unité d'allocation de la mémoire diffère de l'unité minimale (ie l'octet).

Définition 3.2 *La fragmentation externe est la quantité :*
$F_e = 1 - (\sum(tailles\ allouees)/(taille\ memoire))$

3.2.4 La protection inter-processus

Un processus ne doit pas endommager ses collègues. Il faut contrôler, **avant exécution** que toutes les adresses accédées par le processus sont à l'intérieur de la zone allouée. Cela protège les utilisateurs entre eux, mais aussi le SE.

Pour cela, il faut un minimum de matériel : un registre *base* et un registre *limite*. Ensuite chaque adresse mémoire utilisée qui n'est pas comprise entre *base* et *limite* provoque la terminaison du processus.

3.2.5 La liaison d'adresse

La liaison d'instructions et de données (adresses logiques dans le programme, adresses physiques pendant l'exécution) peut avoir lieu à n'importe quelle étape :

Un utilisateur écrit un programme qui contient des *adresses symboliques*. Le compilateur peut générer :

1. *des adresses physiques*, le code est alors *absolu*.

2. *des adresses logiques*, le code est alors *translatable*.
   Dans cette situation, le chargeur peut générer :

   (a) *des adresses physiques*, le code est alors *non relogable*.

   (b) *des adresses logiques*, le code est alors *relogable*.
   Dans cette situation, il faut pouvoir au moment de l'exécution faire la liaison d'adresses, cela n'est possible qu'avec du matériel spécifique (registre de translation). C'est le rôle de la MMU. On peut distinguer différentes situations :

      i. *partition unique* : il faut un registre de translation

      ii. *pagination* : l'adresse logique est généralement appelée adresse virtuelle (page, déplacement) : il faut une table des pages et une valeur de translation par cadres de page.

      iii. *segmentation* : l'adresse logique est simplement un déplacement dans le segment associé : il faut une valeur de translation par segment.

3.3 Une première technique : le Va-et-vient

La taille des processus peut être plus grande que la mémoire. C'est l'ordonnancement à plusieurs niveaux.

La question cruciale est la suivante : un processus doit-il toujours être dans le même emplacement mémoire ?

La réponse est : oui si le code n'est pas relogable, non sinon.

3.4 Une technique ancienne : l'allocation contiguë

Les processus constituent un seul bloc non décomposable.

3.4.1 Partitions de taille fixe

SE	P1	P6	vide	P4	vide

Les propriétés :

taille du processus : il est nécessaire de faire du recouvrement lorsque la taille du processus est supérieure à celle de la partition.

fragmentation interne : fonction monotone de la taille de la partition.

fragmentation externe : nulle.

liaison d'adresse : le code doit être translatable.

3.4.2 Partitions de taille variable

SE	P1	P6	vide	P4	vide

Les propriétés :

taille du processus : il est nécessaire d'estimer la taille d'un processus (code, données, tas, pile). Le dépassement est difficile à gérer.

fragmentation interne : fonction monotone de l'estimation de la taille d'un processus

fragmentation externe : est généralement fonction antimonotone de la taille des *zones inoccupées*. Cependant elle peut être réduite à zéro en effectuant *régulièrement* un compactage de la mémoire. Cette solution demande du code relogable et est de plus coûteuse en temps CPU.

liaison d'adresse : le code doit être translatable.

Il faut de plus gérer la liste des *zones libres*. La solution retenue est généralement une liste chaînée, réalisée en utilisant le début des zones libres, et la taille; manipulée par deux algorithmes.

1. Libération d'une zone occupée : fusionner les zones libres contiguës (4 cas).
2. Allocation d'une zone libre : on scinde une zone libre en une zone allouée de la taille estimée et un reste qui est remis dans la liste chaînée des zones libres.

Il n'existe pas d'algorithme optimal d'allocation (vis à vis de la fragmentation externe), mais seulement des heuristiques :

First Fit liste des trous par adresse. Le premier suffisamment grand est choisi.

Best Fit liste des trous par taille croissante. Le premier suffisamment grand est choisi.

Worst Fit liste des trous par taille décroissante. Le premier suffisamment grand est choisi.

Pour un fonctionnement correct, il faut en moyenne, que le nombre de zones libres soit la moitié du nombre de zones allouées.

3.5 Une deuxième technique : la pagination

Une autre amélioration possible (et souvent retenue) est l'utilisation d'allocations non contigües : pagination et/ou segmentation.

L'allocation contiguë est très contraignante et entraîne généralement une fragmentation externe importante, qui de surcroît augmente avec le temps. Une solution consiste à découper en morceaux le processus, et d'allouer indépendemment pour chacun des morceaux une zone mémoire.

3.5.1 Mémoire virtuelle

La mémoire virtuelle est une technique autorisant l'exécution de processus pouvant ne pas être complètement en mémoire.

La mémoire logique (utilisable par le programmeur) est plus grande que la mémoire physique.

Processus i			1	3		5										...
Processus j		2								5			4			...

Il faut gérer l'information :

Processus i

pages	cadres
2	1
3	3
5	5

Processus j

pages	cadres
1	2
10	5
13	4

Mémoire

cadres	processus	ecriture	modifiée
0			
1	i	x	x
2	j	x	
3	i		
4	j	x	x
5	i,j		

Le partage des cadres par plusieurs processus permet :

- le partage de données et ainsi la communication inter processus.
- le partage de code (réentrant) et donc une optimisation importante de la mémoire. Par exemple, les fonctions des bibliothèques dynamiques ne sont chargées qu'une seule fois.

Pour éviter une recherche (même dichotomique) dans la table des pages d'un processus, il est nécessaire d'avoir une table indicée par les pages, et donc un champ qui indique sa présence où non dans un cadre.

Une adresse logique est alors constituée de :

numéro de page	déplacement dans la page
p	d
(m-n) bits	n bits

Le matériel nécessaire à la pagination (schéma page 269)

Supposons une mémoire de $2^{32} = 4Giga$ et une taille des pages de $2^{12} = 4K$. La taille de la table des pages peut posséder $2^{20} = 1M$ entrées, et ne loge donc pas dans une page, elle est donc paginée.

Tables à plusieurs niveaux

Une amélioration consiste à utiliser une table de pages à plusieurs niveaux.

indice page principale	indice page secondaire	décalage	
p1	p2	d	
10	10	12	pour un 80386

La page principale est en mémoire et seule les pages secondaires nécessaires également (entrées de la table primaire marquées présentes).

Mémoire associative

Une autre amélioration consiste à utiliser une mémoire associative. C'est une mémoire cache matérielle avec un nombre d'entrées restreint (32, 64), et qui permet d'associer en un seule cycle le numéro du cadre à un numéro de page, si l'entrée est présente. Ce dispositif est efficace s'il répond à 95/100 des requêtes.

Tables inverses

Avec les adresses 64 bits, les tables de pages deviennent très très grande. Une solution consiste à utiliser une table des pages inversée commune à tous les processus et d'une fonction de hachage.

Les propriétés de la mémoire virtuelle

taille du processus : il n'est plus nécessaire d'estimer la taille d'un processus, on lui fournit des pages au besoin.

fragmentation interne : Statistiquement $F_i = (N \times T)/2$ ou N est le nombre de processus en mémoire, et T la taille d'un bloc (une page). F_i est donc une fonction linéaire de la taille du bloc, il suffit donc de diminuer la taille des blocs pour diminuer la fragmentation interne.

Diminuer la taille des blocs va engendrer d'une part des sur-coûts de gestion des blocs; d'autre part un ralentissement des performances d'entrées-sorties.

Une fois de plus, il faut trouver le bon compromis.

fragmentation externe : nulle : chaque cadre libre est utilisable.

liaison d'adresse : le code doit être relogable. Il faut une table des pages et une valeur de translation par cadres de page.

3.5.2 Défauts de pages

C'est le traitement par logiciel des pages marquées absentes pendant le déroulement d'une instruction. Il faut :

1. arrêter l'instruction,
2. trouver un cadre libre (au besoin en libérer un),
3. charger la page,
4. relancer l'instruction. Il faut pour cela que le processeur ait sauvegarder le contexte du début de l'instruction. (impossible sur 68000)

3.6 Algorithmes de pagination

Lors d'un défaut de page, si tous les cadres sont utilisés, il faut en libérer un afin de pouvoir y mettre la nouvelle page.

Le choix de la page à évincer est crucial.

Soit une mémoire comportant 5 cadres :

Numéro de cadre	1	2	3	4	5
Numéro de page	7	2	5	1	9

et une demande pour la page 3.

3.6.1 Algorithme optimal

La solution optimale consiste à évincer la page dont l'occurrence de la prochaine utilisation est la plus éloignée dans le futur.

Supposons que la suite de références aux pages soit 3, 9, 4, 3, 2, 1, 2, 7...

La page à évincer est la page 5 qui est la seule à ne pas apparaître dans cette liste. La nouvelle configuration mémoire est donc :

Numéro de cadre	1	2	3	4	5
Numéro de page	7	2	3	1	9

Propriétés

L'algorithme précédent :

- minimise le nombre de défauts de page,
- n'est pas implémentable (hormis dans certaines requêtes dans un SGBD), car il fait l'hypothèse de la connaissance du futur.

3.6.2 LRU : Least Recently Used

L'idée est la suivante : futur = miroir du passé.

L'implémentation demande de stocker une date comme attribut de la table des pages, qui est mise à jour à chaque accès. Cette solution demande un accès à l'horloge pour chaque accès, c'est trop coûteux.

Il est possible d'approximer la date avec un tableau de n bits, plus n est grand, meilleur est l'approximation. Chaque accès provoque un décalage vers la droite de toutes les dates (1 pour la page accédée, 0 pour les autres). La plus petite valeur indique la page la moins récemment utilisée.

3.6.3 FIFO

La date de chargement est inscrite (une seule fois) dans la table des pages. la plus ancienne est évincée.

L'algorithme FIFO possède l'anomalie de Belady : Il existe des suites de requêtes s, telles que :
nombreDefautsPage(s, n cadres) < nombreDefautsPage(s, (n+1) cadres).

3.6.4 seconde chance et horloge

Horloge = FIFO + approximation du dernier accès par un seul bit R.

3.6.5 Quelques variantes

Sur la base des algorithmes précédents, il est possible de tenir compte également de :

- du bit de modification de la page. Une page modifiée doit être sauvegardée, ce qui prend du temps. Il est préférable de choisir une page non modifiée, mais attention aux pages de codes qui ne sont jamais modifiées.
- du nombre d'accès dans un intervalle de temps donné.
- du propriétaire des pages. Les pages peuvent être traitées globalement, où bien localement par processus. Dans ce cas on évince une page du processus qui provoque le défaut de page. Attention, pour qu'un processus fonctionne *correctement*, il faut lui assurer en permanence un nombre minimum de pages en mémoire.

3.6.6 Le principe de localité

Statistiquement, on constate que les suites de requêtes émises par un processus ne sont pas aléatoires. Pour chacune des pages accédées, la courbe du nombre d'accès en fonction du temps ressemble à une courbe de Gauss. Cela signifie que lorsque l'on charge une page, il est fort probable qu'elle va être beaucoup utilisée dans les instructions suivantes.

En fonction de ce principe, les algorithmes basés sur la date du dernier accès et/ou sur le nombre d'accès récent sont meilleurs que ceux basés sur la date du chargement et/ou le nombre d'accès depuis le début du processus.

3.6.7 Pré-chargement

Le pré-chargement s'oppose à la pagination à la demande, une page peut être chargée sans qu'elle soit demandée.

L'hypothèse correspond à un principe de localité *physique* et non pas temporel comme le précédent. Si le processus à besoin de la page p, il va sans doute avoir besoin dans peu de temps des pages $p-1$ et $p+1$. Celles-ci sont donc préchargées lors du chargement de p sur défaut de page.

Une autre utilisation du pré-chargement est lié au va-et-vient. Lorsqu'un processus est replacé en mémoire, il est opportun qu'il retrouve tout de suite l'ensemble des pages qu'il avait au moment de sa sauvegarde sur disque.

3.6.8 Spool de cadres libres

Une amélioration consiste à avoir toujours un nombre de cadres libres supérieur à un seuil. Ainsi le traitement d'un défaut de page devient :

1. arrêter l'instruction,
2. charger la page dans un cadre libre,
3. relancer l'instruction. Il faut pour cela que le processeur ait sauvegarder le contexte du début de l'instruction. (impossible sur 68000)

Il faut alors qu'en tâche de fond, un démon libère des cadres lorsque le nombre des libres devient proche du seuil. Il utilise un algorithme parmi les précédents.

1. choisir une page à évincer,
2. la sauvegarder si nécessaire,
3. ajouter le cadre à l'ensemble des cadres libres.

Une seconde optimisation consiste à se souvenir pour chacun des cadres libres de la dernière page contenue. Si celle-ci est de nouveau demandée, il est inutile de la rechargée (le cadre contient la bonne version), il suffit de retourner le bon numéro de cadres.

3.7 Une troisième technique : la segmentation

la segmentation est une technique d'allocation non contigüe.

zone	pagination	segmentation
taille	fixe	variable
nombre	grand	petit (3 pour 80386 : code, données, pile)
découpage	physique	logique (la protection est facilitée)

3.8 Segmentation + Pagination + Va-et-vient

La segmentation permet une protection logique, puis la pagination permet de découper en petit bloc les gros segments.

Une bonne solution à la gestion de la mémoire.

Chapter 4

Le système de fichiers

4.1 Caractéristiques générales

L'utilisateur confond souvent système d'exploitation et système de fichiers, du fait qu'il en est la partie la plus *visible*. Le système de fichiers peut être caractérisé par analogie à la mémoire.

	mémoire vive	mémoire de stockage
taille	T	T x 1000
fréquence des accès	important	faible
persistance	volatile	permanente
accès (en général)	protégé	partagé
découpage logique	segments	fichiers
découpage physique	cadres	blocs

Le fichier peut être vu comme la plus petite quantité d'informations qui peut être sauvegardée en mémoire auxiliaire. Le système de fichiers d'un système d'exploitation est chargé de la gestion des fichiers.

4.2 Attributs d'un fichier

A des fins de gestion, les systèmes attachent à chaque fichiers un ensemble (variable) d'informations :

le nom symbolique est où n'est pas un attribut,

le type du fichier (lorsqu'il existe) est un attribut,

l'emplacement permet la localisation du fichier sur le bon périphérique,

la taille courante et quelquefois maximale,

la protection qui indique le propriétaire, les droits des utilisateurs et/ou des processus.

les dates de création, de modification et quelquefois du dernier accès.

4.3 Opérations sur les fichiers

Le système d'exploitation fournit des appels systèmes pour manipuler les fichiers.

la création d'un fichier consiste à créer en mémoire la structure qui contient les attributs du fichier, puis la sauvegarder en mémoire auxiliaire de façon à pouvoir y accéder ensuite par le nom.

la destruction d'un fichier consiste enlever de la mémoire auxiliaire la structure qui contient les attributs du fichier.

l'ouverture d'un fichier consiste à charger en mémoire les attributs d'un fichiers. Comme plusieurs ouverture d'un même fichier sont possibles simultanément, l'ouverture incrémente généralement un compteur de référence. Ce compteur peut être un attribut du fichier, ou bien un compteur géré par le système pour sa gestion globale des fichiers ouverts.

la fermeture d'un fichier consiste à sauvegarder l'information en mémoire auxiliaire, à mettre à jour les attributs, à sauvegarder la structure en mémoire, puis à décrémenter le compteur de référence. Si la nouvelle valeur est zéro, l'espace mémoire liée à la structure est libéré.

la lecture d'un fichier consiste à retourner l'information *unitaire* (si elle existe) qui suit le pointeur de fichier.

l'écriture d'un fichier consiste à l'information *unitaire* (si elle existe) qui suit le pointeur de fichier.

le positionnement dans un fichier consiste à déplacer le pointeur de fichier à l'emplacement spécifié.

Remarque : Si l'ouverture charge l'ensemble du fichier en mémoire, les opérations de lecture, écriture et positionnement sont inutiles. Deux défauts, des parties du fichier sont chargées inutilement, et les écritures ne sont effectives que lors de la fermeture du fichier.

4.4 Structures, types et accès d'un fichier

4.4.1 Structures d'un fichier

L'unité d'information contenu dans un fichier est soit l'octet soit un enregistrement. L'enregistrement est utilisé en gestion principalement. La lecture et l'écriture manipulent le contenu par unité.

4.4.2 Types d'un fichier

C'est un attribut qui donne une indication sur le format du fichier, et donc sur son utilisation possible. Les systèmes d'exploitation varient sur ce point.

Unix reconnaît trois types de fichier : fichiers réguliers, répertoires et fichiers spéciaux (fichiers fictifs en liaison avec un pilote de périphérique).

MS DOS reconnaît les fichiers `.com`, les fichiers `.exe` et `.bat`. Attention les extensions `.doc, .xls` ne sont pas des extensions du système, mais utilisées par les outils.

Macintosh associe non seulement à tous les fichiers un type, mais aussi des informations sur le programme créateur.

4.4.3 Accès à un fichier

Accès séquentiel est la méthode la plus simple et le plus utilisé. La recherche est coûteuse.

Accès direct dit aussi aléatoire, est utile pour accéder rapidement à de grandes quantités d'informations. Le fichier est composé d'enregistrements de longueur fixe, et la méthode se base sur le modèle de disque. Les systèmes d'exploitation ne supportent pas tous l'accès direct.

Accès indexé est une méthode *intermédiaire*, elle améliore la recherche en construisant un index pour les fichiers, afin de pouvoir se positionner rapidement à certains endroits, à partir desquels l'accès séquentiel est utilisé.

4.5 Gestion du contenu d'un fichier

Le pilote de disque permet de voir un fichier comme une suite de blocs numérotés. Le bloc est l'analogue de la page : taille fixe et petite.

Une première solution consiste à utiliser la fin d'un bloc pour indiquer le numéro du suivant, mais cela est trop pénalisant pour beaucoup d'algorithmes. Le système doit gérer une description des blocs d'un fichier.

4.5.1 Première technique : la table

MS-DOS utilise une table d'allocation des fichiers (FAT), avec une entrée par bloc. (partie droite du schéma page 394). Le défaut majeur est la limitation (FAT12 fabrique des partitions de 32 Mega, FAT16 , FAT32), et soit la place utilisée en mémoire si elle est totalement chargée, soit le nombre d'accès qui peuvent être nécessaire pour un gros fichier.

La table est généralement stockée sur un (où plusieurs) des premiers blocs du disque.

4.5.2 Deuxième technique : le nœud d'informations

On ajoute une indirection systématique (le temps d'accès au premier bloc sera plus long), qui permet d'accéder à une structure qui contient toute l'informations (les attributs) d'un fichier. Pour des raisons d'efficacité, cette structure est de taille fixe. (schéma page 397)

Unix répartit les i-nodes sur l'ensemble du disque, afin que le l'i-node d'un fichier soit *proche* de son premier bloc.

4.6 Gestion des noms des fichiers : les répertoires

L'utilisateur ne veut pas connaître le numéro du premier bloc d'un fichier, ou bien le numéro son i-node pour y accéder. Il veut le faire par un nom symbolique. Ce nom ne peut donc pas être stocké dans le fichier, où dans l'i-node, ce n'est donc pas généralement un attribut.

Les noms des fichiers sont stockés dans des répertoires.

MS DOS définit un répertoire comme un fichier d'enregistrement : chaque enregistrement comprend le nom et le numéro du premier bloc. C'est un enregistrement, la longueur du nom est donc fixe. Pour des raisons d'efficacité, le nombre d'entrées était également borné. (partie gauche schéma page 394)

Unix utilise un fichier standard (unité = octet) pour ses répertoires, et n'impose aucune limite.

4.7 Gestion des répertoires

Un répertoire étant un fichier, il est naturel que le nom d'un répertoire puisse apparaître comme une entrée dans un répertoire. Cela conduit à une structure arborescente du système de fichiers.

MS DOS utilise cette structure arborescente.

Unix offre le concept de liens pour transformer cette arborescence en graphe. Les liens physiques sont une nouvelle entrée dans un répertoire donnant un deuxième nom symbolique (un deuxième chemin) à un même i-node. Les liens symboliques sont une nouvelle entrée spéciale dans un répertoire avec un nouvel i-node. C'est le contenu du fichier qui doit être interprété comme un chemin pour atteindre le *vrai* nom symbolique.

Les liens transforme l'arbre en graphe. La restriction (faite en Unix) qui interdit de faire des liens physiques sur des i-nodes de répertoires, transforme l'arbre en graphe acyclique. Les graphes cycliques sont problématiques pour les parcours du système de fichiers.

Un attribut *compteur de références* est mis à 1 lors de la création de l'inode, incrémenté à chaque nouveau lien physique, décrémenté à chaque *rm*. La destruction de l'inode est effective lorsque le compteur vaut 0.

4.8 Gestion des blocs

Lors de la gestion du contenu d'un fichier, nous avons dit que le pilote de disque permet de voir celui-ci comme une suite de blocs numérotés.

Les tables d'allocation et les nœuds d'informations permettent de gérer les blocs d'un fichier, sans se préoccuper des emplacements physiques des blocs. Cette abstraction est insuffisante dans le cas des SGBD pour lesquels il est nécessaire pour des raisons d'efficacité, de gérer les emplacements physiques, afin d'obtenir des allocations *contiguës* de blocs pour une même table.

4.8.1 Ordonnancement du disque

Un disque est formé de plateaux comportant chacun deux faces. Chacune des f faces est découpée d'une part en p pistes; d'autre part en s secteurs. On définit ainsi p cylindres qui correspondent intuitivement aux f pistes *alignées* et $f \times p \times s$ blocs. Un bloc, qui est l'unité d'information pour le transfert avec la mémoire, correspond à *l'intersection* d'un secteur et d'une piste. Chaque bloc b_i est identifié par un triplet (p_i, f_i, s_i).

Les blocs sont accédés au moyen de têtes de lecture/écriture montées sur un (ou plusieurs) bras. Quelque soit la technologie utilisée, il faut un certain temps pour lire et/ou écrire l'information contenu dans un bloc. Ce temps de recherche se décompose en :

temps de positionnement qui correspond au déplacement du bras pour que les têtes de lecture soit sur le bon cylindre. Il peut donc être nul. On note t_p le temps pour passer d'un cylindre au cylindre voisin.

temps de latence qui correspond au positionnement de la tête sur le début du bon secteur. Ce temps dépend de la vitesse de rotation du disque. On note t_l le temps pour passer d'un secteur au secteur voisin.

temps de transfert qui correspond au temps de lecture/écriture du bloc. Ce temps dépend de la vitesse de rotation du disque et de la taille du secteur. On note t_t le temps de transfert d'un secteur

Le temps de transfert est une donnée du disque, qui ne peut être modifié par logiciel, et généralement, le temps de latence maximal (une rotation - 1 secteur) est très inférieur au plus petit temps de positionnement non nul (d'une piste à une voisine).

Une approximation du temps de recherche entre les blocs (p_i, f_i, s_i) et (p_j, f_j, s_j) est donc $t_r = |p_i - p_j| \times t_p + |s_i - s_j| \times t_l$

4.8.2 Allocation contiguë et blocs consécutifs

Soit le bloc (p_i, f_i, s_i), tous les blocs (p_i, f_j, s_{i+1}) sont à un temps de recherche $t_r = t_l$. En réalité, pour des raisons autres (alignement plus ou moins parfait des têtes de lecture, temps entre deux transferts qui ne peut être nul), ce n'est pas tout à fait vrai. En pratique, les blocs qui minimisent le temps de recherche à partir de (p_i, f_i, s_i) sont :

1. le bloc (p_i, f_i, s_j) qui correspond à la rotation pendant le temps nécessaire entre deux transferts. C'est la notion de blocs entrelacés.
2. ainsi de suite pour tous les blocs même face, même piste.
3. les blocs (p_i, f_j, s_j) qui ne nécessitent qu'un réalignement (un temps t_a) des têtes.
4. les blocs des cylindres voisins (p_{i-1}, f_j, s_j) et (p_{i+1}, f_j, s_j)
5. ainsi de suite pour les autres cylindres.

Une meilleure approximation du temps de recherche est donc $t_r(b_i, b_j) = |p_i - p_j| \times t_p + (f_i \neq f_j) \times t_a + |s_i - s_j| \times t_l$

Étant donné un disque et ses caractéristiques, il est donc possible de définir une fonction de coût pour le temps de recherche d'un bloc, de définir la notion de blocs successifs, et donc d'avoir une politique d'allocation contigüe. Le système VMS/CMS d'IBM le permet. Les administrateurs de bases de données peuvent être également amenés à ce type de travail. Les problèmes vu au cours du chapitre sur la mémoire sont présents : estimation taille d'un fichier, fichier qui dépasse son allocation, fragmentations interne et externe. Les algorithmes First Fit, Best Fit et Worst Fit peuvent être utilisés, ainsi que le compactage de disque.

4.8.3 Algorithmes d'accès aux blocs

Lors d'un défaut de page, la mémoire demande au pilote du disque de lui retourner le contenu d'un bloc identifié par un triplet (p_i, f_i, s_i).

Plus généralement, à un instant donné, le pilote a en attente une liste de requêtes constituée d'identificateurs de blocs.

FIFO : la requête la plus ancienne

Cet algorithme très simple ne tient pas compte de la structure d'un disque. Il est équitable, mais assez inefficace.

PCTR : Plus Court Temps de Recherche

Considérons la fonction $t_r(b_i, b_j) = |p_i - p_j| \times t_p + (f_i \neq f_j) \times t_a + |s_i - s_j| \times t_l$. Cette propriété fournit directement l'algorithme suivant, connu sous le nom de PCTR.

```
debut
tant que vrai faire debut
   min = MaxInt;
   pour bj dans la liste des blocs en attente faire
      si tr(bi,bj) < min alors debut
         min = tr(bi,bj);
         prochainBloc = bj;
         fin
   lire/ecrire prochainBloc
   bi = prochainBloc;
   fin
fin
```

PCTR est optimal pour le temps de recherche mais malheureusement non équitable. Il n'est donc pas utilisé en pratique, mais sert de référence.

Remarque : $t_p >> t_a >> t_l \Rightarrow t_r(b_i, b_j) \sim (|p_i - p_j|, (f_i \neq f_j), |s_i - s_j|)$.

On obtient la variante (plus efficace)

```
debut
tant que vrai faire debut
   min = (MaxInt,MaxInt,MaxInt);
   pour bj dans la liste des blocs en attente faire
      si (|p_i - p_j|, (f_i \neq f_j), |s_i - s_j|) < min alors
         debut
         min = (|p_i - p_j|, (f_i \neq f_j), |s_i - s_j|);
         prochainBloc = bj;
         fin
   lire/ecrire prochainBloc
   bi = prochainBloc;
   fin
fin
```

Remarque : si l'on maintient un tableau trié des blocs en attente, une recherche dichotomique accélère encore la recherche.

Les algorithmes de balayage

Un bloc $b_i = (p_i, f_i, s_i)$ peut être également identifié par $b_i = (p_i \times f \times s) + (f_i \times s) + s_i$. Un bloc est alors un numéro entre 0 et $(p \times f \times s) - 1$.

Deux algorithmes sont basés sur cette propriété. Le premier est dit de balayage circulaire, et le second de balayage.

```
debut
tant que vrai faire
   pour i de 0 a (p*f*s)-1 faire
      si i est dans la liste des blocs en attente alors
         lire/ecrire bloc bi
fin
```

```
debut
tant que vrai faire debut
   pour i de 0 a (p*f*s)-1 faire
      si i est dans la liste des numéros de blocs en attente alors
         lire/ecrire bloc bi
   pour i de (p*f*s)-1 a 0 faire
      si i est dans la liste des numéros de blocs en attente alors
         lire/ecrire bloc bi
   fin
fin
```

Si l'on maintient la liste des requêtes triée par numéro, ces deux algorithmes peuvent être améliorés.

```
debut
tant que vrai faire debut
   i = premier(liste);
   tant que i <= dernier(liste) faire debut
      lire/ecrire bloc i;
      i = suivant(liste);
      fin
   fin
fin
```

```
debut
tant que vrai faire debut
   i = premier(liste);
   tant que i <= dernier(liste) faire debut
      lire/ecrire bloc i;
      i = suivant(liste);
      fin
   i = dernier(liste);
   tant que i >= premier(liste) faire debut
      lire/ecrire bloc i;
      i = précédent(liste);
      fin
   fin
fin
```

Les algorithmes de balayage sont les plus utilisés aujourd'hui. Ils sont équitables. Ils sont également utilisés pour la gestion des ascenseurs.

4.8.4 Gestion des blocs libres

Le système doit gérer (affecter, libérer) les blocs libres. L'ensemble des blocs libre peut être vu comme une liste soit de triplet $b_i = (p_i, f_i, s_i)$, soit de numéro $b_i = (p_i \times f \times s) + (f_i \times s) + s_i$.

Plusieurs gestions sont possibles :

un vecteur binaire 0101110000111001110...

> **Avantages** simple, efficace pour l'allocation de n blocs contigüe.
>
> **Défauts** doit résider en mémoire pour être efficace, sa taille est alors pénalisante.
>
> C'est la solution retenue par le système Macintosh, avec une optimisation de lecture du vecteur par mot.

un vecteur de comptage qui liste des couples (adresse de blocs, nombres de blocs libres successifs). C'est une variante du vecteur binaire.

une liste chaînée des blocs.

Avantages simple,

Défauts inefficace pour l'allocation d'un grand nombre de blocs.

C'est la solution retenue par MS-DOS, car elle est gratuite avec la FAT. L'espace libre est un fichier particulier.

une liste de groupement de blocs, consiste à utiliser le contenu des blocs libres comme adresses des blocs libres suivants.

Avantages le codage est compact, la recherche d'un grand nombre de blocs libres est rapide.

Défauts la gestion est plus complexe.

La gestion des fichiers Unix est très complexe. Elle a évoluée avec le temps (sans que les utilisateurs s'en aperçoivent). Elle a permis de passer d'un taux d'utilisation de la largeur de bande (débit maximal du transfert) de 3% à 30%

4.8.5 Gestion des blocs défectueux

Les disques neufs possèdent des blocs défectueux, et en cours d'utilisation des blocs le deviennent également. Cette liste peut être gérée de différente manière, mais c'est du ressort du contrôleur et non du système.

statiquement Le formatage reconnaît les blocs défectueux, et ils ne sont pas mis dans la première description des blocs libres. Le système n'a donc jamais accès à ces blocs. C'est la solution retenue par les disques IDE.

dynamiquement Au formatage, le contrôleur livre une description incomplète des blocs libres. Le contrôleur en conserve certains pour rediriger un bloc qui deviendrait défectueux, sans que le système s'en aperçoive. Cela peut diminuer les performances en cas d'allocation contiguë, mais peu si les blocs réservés sont répartis équitablement dans le disque. C'est la solution retenue par les disques SCSI.

4.8.6 Cohérence des blocs

Les fichiers ouverts et les répertoires possèdent généralement deux images : une sur disque et une en mémoire plus actuelle.

Une panne de l'ordinateur (matérielle ou logicielle) peut donc avoir des conséquences graves. Un bloc peut être libre et pris, où n'apparaître nul part.

Généralement un programme spécifique vérifie au démarrage le statut de chacun des blocs. En fonction des algorithmes d'allocation de blocs, certaines erreurs peuvent où ne peuvent pas être réparés. Pour ces dernières, le système d'exploitation va faire en sorte de les minimiser : écriture systématique des blocs fichiers modifiés et des blocs i-nodes concernés.

4.9 Une amélioration : les buffers

Pour améliorer la bande passante, des techniques de mémoire cache sont souvent mises en place. Un cache sur le contrôleur du disque permet souvent de stocker une piste complète (la piste du dernier bloc demandé). Une partie de la mémoire principale peut mémoriser des blocs et une partie du disque peut servir de zone de recopie avec un algorithme LRU.

Le défaut principal est l'incohérence encore plus grande en cas de panne. Différentes techniques pour minimiser les conséquences : écriture périodique des caches vers le stockage permanent, écriture d'un journal (SGBD).

4.10 Une contrainte : la sécurité

Il faut protéger l'information contre les dégâts physiques (fiabilité) et les accès inappropriés (protection).

4.10.1 La fiabilité

On peut citer :

les sauvegardes qui permettent la restauration du systèmes de fichiers dans un état cohérent d'un passé plus ou moins lointain.

les tableaux de disques redondants (RAID) permettent de dupliquer l'information sur un ensemble de disque, de façon à pouvoir reconstruire l'information si l'un tombe en panne.

la mémoire stable fait en sorte que les transferts de données défectueux soit détectés pour être repris.

4.10.2 La protection

La protection consiste à protéger les objets du système en fonction d'un environnement que l'on appelle domaine d'exécution. Un objet est donc un fichier, une imprimante, où plus généralement une ressource, et un domaine est un utilisateur, où bien un processus.

De manière très générale, la protection est donc une matrice de droits d'accès, qui indique dans chaque place les opérations permises.

	F1	F2	F3	Imprimante	D1	D2	D3	D4
D1	lire		lire			commuter		
D2				imprimer			commuter	commuter
D3		lire	exécuter					
D4	lire écrire		lire écrire		commuter			

Cette matrice est beaucoup trop complexe d'implémentation et d'utilisation. Il faut, pour chaque fichier, décrire la colonne correspondante.

Les systèmes d'exploitation implémente donc une version plus ou moins dégradé de cette matrice.

MS DOS n'offre pas de protection du fait que ce système a été conçu pour des ordinateurs personnels non relié à un réseau. Seul les fichiers cachés ont un semblant de protection.

Unix utilise une liste d'accès dégradée (propriétaire, groupe, autre) par fichiers, avec trois types d'accès (lire, écrire, exécuter). L'avantage est le codage compact et de taille fixe qui permet que cela soit un attribut du fichier.

Mach utilise des listes de capacités. Une liste de capacité d'un domaine est une liste d'objets, avec les opérations permises. La liste est protégée et maintenue par le système, accédée indirectement par l'utilisateur.

clés et verrous chaque objet possède une liste de verrous, et chaque domaine une liste de clés.

SGBD indique pour chaque table les droits. Ceux-ci ont une valeur par défaut modifiable par l'instruction GRANT.

HTML permet deux protections : une par adresse de machine, et une par mot de passe.

mot de passe

4.10.3 Les vers et virus

Un ver est un processus utilisant le mécanisme de génération dynamique pour détruire la performance du système. Un des plus célèbre est le ver Internet du 2 Novembre 88, produit par Morris, étudiant de l'université Cornell. Il est basé sur :

1. le fichier `.rhost` qui permet des `rlogin` sans mot de passe.
2. le fichier `password` et d'un algorithme de *crackage*
3. les commandes `finger` et `sendmail` qui étaient alors *buggées*

Il consiste, grâce au bug, à envoyer sur une machine distante un programme C court, qui se compile, qui s'exécute et dont le résultat est le téléchargement du vers complet; puis de recommencer plus loin après examen des fichiers `.rhost` et `password`.

(schéma page 466)

Un virus est un fragment de code inséré dans un programme légitime. L'exécution du programme commence par l'exécution du virus, puis reprend son cours normal. Ce problème concerne les systèmes qui ne protègent pas les programmes contre l'écriture.

Chapter 5

Les Entrées/Sorties

5.1 Structure des Entrées/Sorties

Un système est constitué d'une UC et d'un certain nombre de contrôleurs de périphériques connectés par un bus commun. Chaque contrôleur est chargé d'un où plusieurs périphériques.

5.1.1 Contrôleurs et pilotes

La partie électronique d'un contrôleur est composée de registres spécialisés et de mémoire locale.

La partie logicielle du contrôleur a la responsabilité de déplacer les données entre les périphériques contrôlés et sa mémoire locale.

Le pilote de périphérique qui fait partie du système dialogue avec le contrôleur. Il lui donne des ordres (généralement codés sur un octet); bloque ou non le processus qui a généré l'interruption d'entrée/sortie; attend le résultat (ou le code d'erreur) fournit par le contrôleur, met le résultat *au bon endroit*; et enfin réveille si nécessaire le processus.

Deux problèmes majeurs :

une gestion des E/S indépendante des périphériques dont l'objectif est de faciliter le travail de l'utilisateur. Cela passe par les bibliothèques de fonctions (ex printf);

les débits de transferts différents des périphériques font que le mécanisme des interruptions pour transférer les données vers le processus en attente est correctement efficace pour les périphériques lents (clavier) mais très pénalisant en UC pour les périphériques rapides (disque).

5.1.2 La structure DMA

Cette structure permet un accès direct à la mémoire pour le contrôleur sans passer par le mécanisme d'interruption. Le pilote doit configurer les registres, et les fournir au contrôleur. Le contrôleur DMA dépose sur le bus les données qui se copie directement en mémoire. Lorsque le transfert est terminé, le contrôleur le signale au pilote. Il n'y a qu'une seule interruption pour le transfert d'un bloc.

Cela justifie la notion d'entrelacement vue au chapitre précédent.

5.2 Quelques périphériques

5.2.1 Le disque

Déjà vu au chapitre précédent.

5.2.2 L'horloge

Elle fournit des services :

- la date et l'heure.
- des statistiques par processus, utilisés par exemple pour approximer l'algorithme PCTE d'ordonnancement des processus.
- des délais, utilisés dans l'algorithme du tourniquet. Après chaque quantum, il faut interrompre le processus en cours.
- des alarmes, utilisées par les commandes `at` et `crontab`, qui permettent l'exécution de processus à des moments déterminés.

Par exemple, Unix gère :

- un compteur de secondes depuis le 1 janvier 1970 sur 32 bits. Cela permet environ 100 ans d'utilisation. Ce compteur est utilisé pour la date, l'heure et les alarmes, mais n'est pas assez précis pour l'ordonnancement.
- un échéancier qui stocke des délais décrémentés régulièrement pour être comparés à zéro (et non les dates qu'il faudrait comparer avec l'heure) pour les prochaines commandes à exécuter.

5.2.3 Terminaux

Cela comprend les terminaux d'entrées (clavier, souris, ...) et les terminaux de sorties (écran, ...).

Le clavier

Deux modes de fonctionnement sont possibles : le mode caractère immédiat, qui pour chaque touche enfoncée, provoque une interruption pour traitement du caractère; et le mode ligne qui stocke dans une mémoire locale tous les caractères tapés jusqu'au prochain caractère `Entrée`.

Il faut noter la présence de caractères spéciaux dont le traitement diffère : ils ne sont pas envoyer au programme de traitement, mais sont interprétés comme des instructions pour le système d'exploitation.

L'écran

Deux techniques pour afficher de l'information sur un écran.

la gestion du curseur consiste à pouvoir positionner le curseur (et donc la prochaine sortie) sur l'écran. La technique utilise des séquences d'échappement (`'ESC [ 2 J'` efface souvent l'écran) qui définissent les capacités d'affichage du terminal (inverse vidéo, clignotement,...). Elles sont décrites dans le fichier `/etc/termcap`.

la gestion de l'image consiste à maintenir une image mémoire de l'image écran. Une image bitmap est une image mémoire qui utilise un bit par pixel. Pour un écran normal, (1000×1000), il faut environ 125 K-octets. Pour une image couleur RGB, on utilise un octet par couleur, soit 24 bits par pixel, soit environ une mémoire vidéo de 3 M-octets. La sortie consiste alors à écrire en mémoire, puis à rafraîchir l'image (avec des optimisations).

5.3 Le serveur X

Une solution pour avoir une gestion des E/S indépendante des périphériques, consiste à interposer un logiciel entre d'une part l'application cliente; d'autre part l'écran, le clavier et la souris.

Un serveur X est un programme qui peut dialoguer avec plusieurs clients X. Un client X est une interface graphique pour une application quelconque.

Toutes les entrées-sorties de l'application ne sont pas adressées directement aux périphériques, mais au serveur X, et c'est celui-ci qui est chargé du dialogue avec les périphériques. La portabilité est traitée par le serveur et non pas par l'application. Une conséquence est la possibilité des entrées-sorties distantes.

Au niveau de l'implémentation :

- le protocole X,
- les bibliothèques de fonctions (Motif, Xlib ...),
- la boucle de gestion des événements.

Au niveau de l'utilisation :

- les ressources qui paramètrent le comportement du serveur,
- le gestionnaire de fenêtres pour l'aspect et le comportement des applications.
- les émulateurs de terminaux pour les interpréteurs de commandes.

Chapter 6

La gestion des ressources (l'interblocage)

6.1 Les ressources

Un système est constitué d'un certain nombre de ressources (une UC, une mémoire, un écran, des imprimantes, des disques, des descripteurs de fichiers, des sémaphores) qui peuvent être utilisées par des processus concurrents.

Ces ressources sont de natures différentes, elles peuvent être caractérisées par :

partageable/non partageable : plusieurs processus peuvent lire un même fichier sans que cela pose problème, alors qu'il n'est pas souhaitable que plusieurs processus écrivent *en même temps sur une imprimante.*

distinguable/indistinguable un processus qui désire imprime souhaite disposer d'une imprimante, peu importe laquelle si plusieurs peuvent rendre le service.

réquisitionnable/non réquisitionnable l'unité centrale est souvent réquisitionnable, mais que signifie : *réquisitionner un descripteur de fichiers?*

Pour utiliser une ressource, un processus doit :

1. effectuer une requête (appel système), puis attendre l'acceptation (où le refus),

2. utiliser la ressource,

3. libérer la ressource (appel système) lorsqu'il n'en a plus besoin.

Un système d'exploitation doit donc gérer au mieux ses ressources. Le problème majeur est l'interblocage.

6.2 L'interblocage

6.2.1 Un exemple

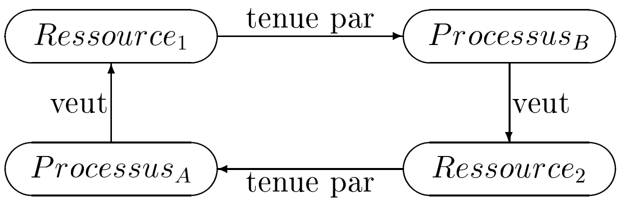

Figure 6.1: un interblocage

6.2.2 Définition

Un ensemble de processus est dans une situation d'interblocage quand chaque processus de l'ensemble attend un événement qui ne peut être produit que par un autre processus de l'ensemble.

6.2.3 Une caractérisation

Pour un ensemble de processus et un ensemble de ressources, la libération d'une ressource étant toujours possible, les événements attendus sont les acceptations et les refus pour les requêtes.

propriété : L'ensemble de processus est dans une situation d'interblocage si et seulement si les quatre conditions suivantes sont satisfaites.

Exclusion mutuelle : les ressources partageables peuvent être utilisées simultanément par plusieurs processus; elles ne provoquent donc ni refus, ni attente. **Il faut donc que des ressources soient non partageables, c'est à dire accessibles en exclusion mutuelle.**

Tenir et attendre : si aucun processus ne peut détenir simultanément deux ressources, et qu'aucun ne conserve indéfiniment une ressource, il ne peut y avoir d'interblocage. **Il faut donc qu'un processus ait besoin de deux ressources simultanément, c'est à dire qu'il puisse tenir une ressource et attendre une autre.**

Pas de réquisition : si un processus peut prendre les ressources détenues par les autres sans leurs accords, il ne peut être dans une situation d'interblocage. **il faut donc que les ressources soient non réquisitionnables.**

Attente circulaire : soit le graphe d'allocation des ressources : graphe bipartite (processus, ressources) et les étiquettes `veut` et `tenue par`. Si ce graphe est sans cycle, et qu'aucun processus ne conserve indéfiniment une ressource, alors la définition de l'interblocage n'est pas vérifiée. **Il faut donc un cycle représentant une attente circulaire dans le graphe d'allocation des ressources.**

6.2.4 Le cadre de l'étude

Remarque : Si l'attente circulaire est vérifiée, c'est qu'il existe un processus qui puisse tenir une ressource et attendre une autre. L'étude de l'interblocage est donc : Étant donné un système composé de processus et de ressources non partageables et non réquisitionnables, existe t-il un cycle dans le graphe d'allocation des ressources.

L'interblocage est un vrai problème pour les systèmes d'exploitation, son traitement peut être préventif où curatif, avec évidemment des avantages et des inconvénients.

6.3 Les politiques de détection et reprise

Le principe consiste à détecter, à intervalles réguliers, les attentes circulaires, puis de permettre la reprise par une action curative.

```
debut
tant que vrai faire debut
     trouverAttenteCirculaire;
     traiterCycle;
     attendre delaiPeriode;
     fin
fin
```

`traiterCycle` utilise généralement une des deux méthodes suivantes :

reprise préemptive : le système provoque une préemption en retirant une ressource à un processus. Ce processus continue comme il peut.

reprise par élimination : le système tue un processus, afin que les ressources qu'il détenait soit libérées.

6.3.1 Cas des ressources distinguables

Soit un système et son graphe d'allocation des ressources, `trouverAttenteCirculaire` est facilement implémentable :

```
debut
tant qu'il existe un sommet sans prédécesseur faire debut
     enlever les arcs sortants de ce sommet;
     enlever le sommet;
     fin
attenteCirculaire = non (graphe résultant est vide)
fin
```

6.3.2 Cas général

Il existe des ressources non distinguables (ex : un bloc du disque libre) pour le problème de l'interblocage. Dans ce cas, le graphe des allocations est complexe, et une représentation matricielle est plus aisée et plus efficace.

Un système est composé de m processus et de n types de ressources. Le problème de l'interblocage est décrit par : un vecteur $E = (e_1, e_2, \ldots, e_n)$ des ressources existantes, une matrice $T = (t_{11}, \ldots, t_{mn})$ des ressources tenues, et une matrice $R = (r_{11}, \ldots, r_{mn})$ des requêtes. On note $O = (0, 0, \ldots, 0)$

`trouverAttenteCirculaire(E,T,R)` peut alors s'écrire :

```
debut
D = E − Σ_{i=1}^{i=m} T_i ;
tant qu'il existe un processus P_i tel que O < R_i ≤ D faire debut
    D = D + T_i ;
    R_i = O ;
    fin
attenteCirculaire = (E ≠ D);
fin
```

D représente les ressources disponibles. L'idée intuitive est donc : si le système satisfait la requête R_i, alors P_i peut terminer son exécution, et rendre ainsi les ressources T_i qu'il détient.

Un exemple

Soit un système composé de 3 processus et de 4 types de ressources.

$E = (5, 2, 3, 1)$, $T = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$, $R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$.

L'algorithme `trouverAttenteCirculaire(E,T,R)` donne :

1. $D = (2, 1, 0, 0)$
2. $O < R_3 \leq D$, donc $R_3 = O$ et $D = (2, 2, 2, 0)$
3. $O < R_2 \leq D$, donc $R_2 = O$ et $D = (4, 2, 2, 1)$
4. $O < R_1 \leq D$, donc $R_1 = O$ et $D = (5, 2, 3, 1)$
5. pas d'attente circulaire

Pour $R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & \mathbf{1} \end{pmatrix}$.

L'algorithme `trouverAttenteCirculaire(E,T,R)` donne :

1. $D = (2, 1, 0, 0)$

2. attente circulaire

Pour $R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ \mathbf{3} & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$.

L'algorithme `trouverAttenteCirculaire(E,T,R)` donne :

1. $D = (2, 1, 0, 0)$
2. $O < R_3 \leq D$, donc $R_3 = O$ et $D = (2, 2, 2, 0)$
3. attente circulaire

6.3.3 Le maintien de l'information

`trouverAttenteCirculaire` utilise soit un graphe, soit des matrices. Cette information peut être obtenue de deux façons :

- elle est maintenue en permanence à jour par le système (en fait un processus particulier), en tenant compte de tous les appels systèmes (requêtes et libérations).
- elle est calculée chaque fois que nécessaire en utilisant les tables de ressources présentes dans les contextes de chaque processus (liste des périphériques utilisés, listes des fichiers ouverts ...).

Les deux solutions sont coûteuses en temps CPU, la première utilise également de la mémoire en permanence.

6.4 Les politiques d'évitement

Pour éviter de devoir choisir entre reprise préemptive et reprise par élimination, il suffit d'éviter les interblocages. Malheureusement, ce n'est pas simple comme le montre l'exemple suivant :

On utilise encore le système composé de 3 processus et de 4 types de ressources.

$E = (5, 2, 3, 1)$, $T = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$, $R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$.

L'algorithme `trouverAttenteCirculaire` nous dit qu'il n'y a pas d'interblocage, et que l'on peut satisfaire aux demandes de P_3. Faisons l'hypothèse que P_3 ne termine pas aussitôt, mais qu'il souhaite maintenant une ressource de type 4. La situation est $T = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 2 & 2 & 2 & 0 \end{pmatrix}$, $R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$, qui est une situation d'interblocage car $D = (0, 0, 0, 0)$.

Pour éviter l'interblocage, il ne fallait donc pas donner les ressources à P_3, mais c'était la seule évolution possible, c'était donc la requête précédente qui n'aurait pas du être satisfaite...

Plus généralement, toutes les solutions d'évitement doivent connaître l'avenir, et ne sont donc pas implémentable directement. Il existe des algorithmes d'évitement dès que l'on a une connaissance même partielle du futur.

6.4.1 L'algorithme du banquier (Dijkstra)

Nous faisons l'hypothèse qu'au début de son exécution, un processus déclare un vecteur $Max = (m_i, \ldots, m_n)$ qui indique, pour chaque type de ressource, le nombre maximum dont il pourra avoir besoin lors de son exécution.

Une situation saine, est une situation dans laquelle même si tous les processus veulent le maximum de ressources, il n'y a pas d'interblocage. A partir d'une telle situation, même dans le pire des cas, on est capable d'ordonner les processus afin que tous terminent.

L'algorithme suivant détermine si une situation est saine ou non.

```
situationSaine(E,Max,T)
debut
situationSaine = non (trouverAttenteCirculaire(E,T,Max-T));
fin
```

L'algorithme d'évitement du banquier s'écrit alors :

```
debut
tant que vrai faire
     pour chaque requête Ri faire
           si situationSaine(E,Max,T+Ri) alors debut
              T = T + Ri;
              Ri = 0;
              fin
fin
```

Supposons qu'au même instant, tous les clients d'une banque décident de retirer tous l'argent qu'ils possèdent sur tous leurs comptes. Si le banquier peut satisfaire ces requêtes, c'est que la situation de la banque est saine. C'est cette analogie qui donne le nom à cet algorithme.

Un exemple

Soit le système décrit par $E = (5,5,5,5)$, $Max = \begin{pmatrix} 3 & 2 & 3 & 1 \\ 1 & 3 & 2 & 3 \\ 2 & 2 & 2 & 2 \end{pmatrix}$ et l'évolution :

T	R	D	$remarques$
$\begin{pmatrix} 1 & 1 & 2 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	O	$(3,2,1,4)$	$situationSaine(E, Max, T)$
$\begin{pmatrix} 1 & 1 & 2 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 2 & 0 & 2 \\ 2 & 0 & 0 & 1 \end{pmatrix}$	$(2,2,0,3)$ $(3,0,1,2)$ $(1,2,1,3)$	$situationSaine(E, Max, T + R1)$ $situationSaine(E, Max, T + R2)$ $situationSaine(E, Max, T + R3)$
$L'algorithme\ du\ banquier\ accepte\ la\ requete\ de\ P1$			
$\begin{pmatrix} 2 & 1 & 3 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} & & & \\ 0 & 2 & 0 & 2 \\ 2 & 0 & 0 & 1 \end{pmatrix}$	 $(2,0,0,1)$ $(0,2,0,2)$	$Pas\ de\ requete\ de\ P1$ $non\ situationSaine(E, Max, T + R2)$ $non\ situationSaine(E, Max, T + R3)$
$L'algorithme\ du\ banquier\ fait\ attendre\ P2\ et\ P3$			
$\begin{pmatrix} 2 & 1 & 3 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 2 & 0 & 2 \\ 2 & 0 & 0 & 1 \end{pmatrix}$	$(2,2,0,3)$ $(2,0,0,1)$ $(0,2,0,2)$	$situationSaine(E, Max, T + R1)$ $non\ situationSaine(E, Max, T + R2)$ $non\ situationSaine(E, Max, T + R3)$
$L'algorithme\ du\ banquier\ accepte\ la\ requete\ de\ P1$			

6.4.2 Contraintes d'implémentation

Un algorithme d'évitement ne peut pas travailler à intervalles réguliers, il doit donner **son accord pour toutes les requêtes**, il doit donc maintenir en permanence ses données. Sa complexité fait qu'il est très coûteux en UC. De plus tous les processus ne sont pas capables de fournir en début d'exécution le vecteur des ressources maximales. L'algorithme du banquier n'est donc pas exploitable dans un système généraliste.

6.5 Les politiques de prévention

Une autre solution pour éviter de devoir choisir entre reprise préemptive et reprise par élimination, donc pour éviter les interblocages, consiste à faire en sorte qu'une des quatre conditions de l'interblocage soit toujours fausse.

6.5.1 L'exclusion mutuelle

Le problème vient de l'existence de ressources non partageables accessibles uniquement en mode exclusif. Une technique permet de *transformer* une ressource non partageable en une ressource partageable : le *spool géré par un démon.*

Intuitivement un spool est une zone mémoire (un buffer) qui met en attente (non bloquante) des requêtes, qui seront traitées lorsque la ressource sera disponible. C'est la technique utilisée pour les impressions.

En théorie, ce n'est pas une solution, car le buffer est une nouvelle ressource non partageable : c'est le problème du producteur/Consommateur avec un buffer limité.

En pratique, cette solution *retarde* le problème de l'interblocage, qui survient lorsque le buffer est plein. Il suffit donc d'avoir un buffer *suffisamment grand.*

6.5.2 Tenir et attendre

Une solution consiste à demander qu'un processus demande au début toutes les ressources dont il a besoin. Ainsi, il n'aura pas à faire de nouvelle requêtes. Deux défauts : d'une part le processus garde les ressources beaucoup plus de temps que nécessaire; d'autre part la famine est possible.

Une autre solution consiste à libérer toutes les ressources avant d'effectuer une nouvelle requête, même si celle-ci concerne des ressources identiques. Le processus grade moins longtemps les ressources, mais encore deux défauts majeurs : d'une part la famine demeure toujours possible; d'autre part entre deux *possessions* d'une même ressource, celle-ci a pu être modifiée par un tiers.

6.5.3 Pas de réquisition

Le problème vient des ressources déjà allouées et qui ne peuvent être réquisitionnées. Une solution consiste à déclarer les ressources détenues par un processus en attente de ressources comme réquisitionnables. Lors d'une requête d'un autre processus, elles seront réquisitionnées si elles lui permettent de ne pas attendre. Cette solution possède deux défauts : d'une part la famine est possible, d'autre part, il faut pouvoir *sauvegarder* puis *restaurer* la ressource car elle a pu être modifiée par un tiers.

6.5.4 Attente circulaire

Le problème vient du fait que les processus peuvent requérir les ressources dans n'importe quel ordre. Une solution consiste donc à ordonner les ressources (un numéro unique par type de ressource) et d'obliger les processus à demander les ressources en respectant cet ordre. Cette solution ne provoque pas de famine.

Deux défauts : d'une part le processus garde les ressources beaucoup plus de temps que nécessaire; d'autre part elle n'est pas facile à faire respecter par les utilisateurs, ni facile à intégrer dans les compilateurs.

6.6 Les politiques laxistes

En fait, pour deux raisons, les concepteurs de systèmes d'exploitation ignorent en partie le problème de l'interblocage.

- l'évitement (et même la détection) sont des solutions complexes et coûteuses en CPU.
- la probabilité d'un interblocage est faible en pratique.

Les solutions retenues sont en fait une combinaison des politiques de reprise et de prévention.

6.6.1 Mise en œuvre de la détection et reprise

La détection est assurée par l'utilisateur où l'ingénieur système. La reprise est une reprise par élimination d'un processus.

6.6.2 Mise en œuvre de la prévention

Exclusion mutuelle : on utilise un spool pour toutes les ressources qui le permettent.

Tenir et attendre : un protocole à deux phases est souvent mis en place pour les transactions dans les SGBD.

Pas de réquisition : la solution proposée ne peut pas s'appliquer à toutes les ressources, mais c'est la solution utilisée avec l'UC et ses registres lors de chaque entrée-sortie.

Attente circulaire : la solution ne peut pas être respectée par tous les utilisateurs, mais on peut demander aux programmeurs du noyau de la respecter. C'est la solution utilisée dans Linux comme l'indique cet extrait du fichier `dma.c`.

```
/* $Id: dma.c,v 1.7 1994/12/28 03:35:33 root Exp root $
 * linux/kernel/dma.c: A DMA channel allocator. Inspired by linux/kernel/irq.c.
 *
 * Written by Hennus Bergman, 1992.
 */

#include <linux/kernel.h>
#include <linux/errno.h>
#include <asm/dma.h>
#include <asm/system.h>


/* A note on resource allocation:
 *
 * All drivers needing DMA channels, should allocate and release them
 * through the public routines ‘request_dma()’ and ‘free_dma()’.
 *
 * In order to avoid problems, all processes should allocate resources in
 * the same sequence and release them in the reverse order.
 *
 * So, when allocating DMAs and IRQs, first allocate the IRQ, then the DMA.
 * When releasing them, first release the DMA, then release the IRQ.
 * If you don’t, you may cause allocation requests to fail unnecessarily.
 * This doesn’t really matter now, but it will once we get real semaphores
 * in the kernel.
 */
```

Chapter 7

Les systèmes distribués

7.1 Introduction

7.1.1 Les motivations

On peut citer :

- Le partage de ressources (imprimante) qui limite les investissements.
- L'accélération du calcul par répartition sur plusieurs processeurs qui rend possible certaines études.
- La fiabilité grâce à la redondance des ressources qui permet une meilleure disponibilité.
- Les communications par l'échange de messages, par le transport de fichiers qui augmentent la convivialité.

7.1.2 Une classification

Flynn a établi une classification qui tient compte du nombre de processeurs (un ou plusieurs) et du nombre de mémoires (une ou plusieurs). Cela donne les quatre classes suivantes :

SISD Single Instruction, Single Data : les machines "classiques". Tous les chapitres précédents s'y rapportent.

SIMD Les calculateurs vectoriels.

MISD les bi-processeurs actuels.

MIMD les réseaux : c'est l'objet de ce qui suit.

7.2 La structure des réseaux

7.2.1 Les topologies

C'est la structure physique du réseau. On distingue :

Complètement connectés : le graphe est complet, les temps de communications sont constants, très grande fiabilité, très coûteux.

Partiellement connectés : le graphe est incomplet, les temps de communications sont fonctions de la longueur des chemins et de l'existence de *goulots d'étranglement*. La fiabilité dépend du nombre d'arcs (où de machines) qu'il faut *casser* pour obtenir deux sous-réseaux.

Pour des raisons de coûts, les réseaux sont toujours partiellement connectés, mais structurés.

Réseaux hiérarchiques : le graphe est une arborescence. Le défaut majeur est le *goulot d'étranglement* de la racine, et sa non fiabilité.

Réseaux en étoile : le graphe est un étoile. La machine centrale est critique : c'est le *goulot d'étranglement* et sa panne rend le réseau totalement déconnecté.

Réseaux en anneau : le graphe est un anneau unidirectionnel ou bidirectionnel. Le temps de communication est fonction de la longueur de l'anneau, la fiabilité est faible

Réseaux de bus à accès multiples : les machines déposent des messages sur un bus, qui devient une ressource particulière, avec donc une politique d'accès. On peut retrouver toutes les structures précédentes.

Réseaux hybrides : est le réseau obtenu par regroupement de plusieurs réseaux.

7.2.2 Les types de réseaux

On distingue :

Réseaux locaux (LAN) : conçus pour couvrir une petite zone géographique. Le matériel utilisé est du haut de gamme : Les temps de communications sont bons et la fiabilité est bonne. C'est souvent un réseau de bus à accès multiples qui est facile à gérer : ajout et retrait de machines et de périphériques. (Ethernet, FDDI, ATM)

Réseaux à grande distance (WAN) : connectent des réseaux locaux entre eux en utilisant des services moins fiables (lignes téléphoniques). Arpanet, Internet.

7.2.3 Les communications

Les problèmes spécifiques :

Le nommage et l'interprétation des noms doit permettre à deux processus de se localiser afin de pouvoir communiquer. La structure de nommage DNS.

Les stratégies de routage : (fixe , circuit virtuel (durée d'une session), dynamique (pour chaque paquet))

Les stratégies de connexion : (circuits, messages, paquets)

La gestion des conflits : (CSMA/CD, jeton, emplacements (ATM))

7.3 Les structures des systèmes répartis

7.3.1 Les systèmes d'exploitation de réseaux

Ils fournissent un environnement à l'utilisateur qui permet :

La connexion à distance : pour pouvoir utiliser une machine dont on connaît l'existence. (telnet)

Le transfert de fichiers : entre plusieurs machines (ftp)

Peu transparent à l'utilisateur, qui doit soit connaître plusieurs S.E. (telnet), soit maîtriser des commandes supplémentaires (ftp) et dupliquer de l'information (cohérence).

7.3.2 Les systèmes d'exploitation répartis

Les utilisateurs doivent accéder aux ressources distantes de la même manière que les ressources locales.

Migration de données (ftp automatisé)

Migration de calculs (RPC)

Migration de processus (extension des RPC)

7.3.3 Les services à distance

Les appels de procédures distantes (RPC) schéma page 514

Les threads (processus légers).

- un processus peut comporter plusieurs threads : chacun possède son compteur, ses registres … mais l'espace mémoire est commun.

- soit l'ordonnanceur devient un ordonnanceur de threads ($\neq$ variantes), soit un programme spécial qui ordonnance les threads d'un même processus. (deux niveaux d'ordonnancement : plus facile, mais pas meilleur).
- Il est du ressort du programmeur du processus de gérer les communications inter-threads : synchronisations avec sémaphores du fait de l'espace mémoire commun.
- très pratique pour l'écriture de serveurs. Par exemple, un serveur de fichiers possède trois threads :
  - un pour analyser les requêtes.
  - un pour retourner les blocs déjà en mémoire.
  - un pour récupérer les blocs sur disque.

  ainsi, une requête qui nécessite un accès disque (normalement bloquante pour les autres) n'est plus bloquante.

7.3.4 La robustesse

Le réseau peut tomber en panne (liaison, site, perte d'un message). Il faut pouvoir détecter et corriger ses défaillances.

Détection des défaillances est un problème difficile. Il est généralement impossible de connaître la cause. Les techniques : poignée de mains à intervalles réguliers, second chemin,...

Reconfiguration tables de routages, reconstruction de l'anneau, élection ...

Reprise après défaillance diffusion, mise à jour (des défaillances dans l'intervalle, des courriers, ...), et reconfiguration.

7.4 Les systèmes de fichiers répartis

C'est un système de fichiers dont les clients, les serveurs et les unités de stockages sont répartis entre les machines.

7.4.1 Nommage et transparence

Le nommage est la correspondance entre les objets physiques et logiques.

La transparence doit cacher l'endroit où se trouve le fichier dans le réseau. Son nom ne fait pas référence à l'emplacement physique, et ne doit pas être changé quand son emplacement physique change.

Une solution : un service de nommage.

7.4.2 L'accès aux fichiers

Utilisation des RPC. Les mémoires caches permettent de minimiser le trafic sur le réseau, mais posent le problème de la cohérence des différentes mémoires caches.

7.4.3 Service avec ou sans état

Un service de fichiers avec états établit une connexion entre le client et le fichier. Le client à une référence à l'inode.

Un serveur sans état traite les requêtes indépendemment les unes des autres, qu'elles proviennent d'un même client ou non.

7.4.4 La duplication des fichiers

Une solution pour diminuer les temps d'accès, mais qui d'une part, demande plus de disques, d'autre part, demande une gestion de la cohérence.

7.4.5 Un exemple : NFS

Network File System (SUN). schéma page 548.

Montage, auto-montage, client/serveur, RPC.

7.5 La coordination répartie

Le besoin de synchroniser et de faire communiquer des processus existe également dans un système réparti.

7.5.1 Les ordres sur les événements

Il est nécessaire, pour traiter un certain nombre de problèmes, d'avoir un ordre global sur les événements d'un système.

L'ordre physique

Peut être construit à partir de : adresse machine, numéro de processus, horloge locale.

Problème : pas cohérent (du fait des dérives) avec l'envoi et la réception de messages.

UTC : Temps Universel Coordonné suffit pour beaucoup de problèmes (dates des fichiers ...), mais insuffisant pour section critique, interblocages... car les délais de transmission ne sont pas connu exactement.

Un ordre logique : les horloges de Lamport

Une relation R telle que :

1. Pour chaque processus P, si a est antérieur à b, alors $R(a_P) < R(b_P)$
2. Pour chaque couple de processus (P, Q), si P envoie à Q un message m, alors $R(envoi_P(m)) < R(reception_Q(m))$

L'algorithme de Lamport :

1. A chaque création de processus P_i, initialiser un compteur c_i à 0.
2. A chaque événement significatif de P_i, incrémenter c_i.
3. Estampiller chaque envoi de message m par P_i du compteur c_i.
4. A chaque réception de message (m, E) par un processus P_i, celui ci effectue l'opération $c_i = max(c_i, E) + 1$.

7.5.2 Le problème de l'exclusion mutuelle

L'approche centralisée

Un coordinateur : 3 messages par entrée (éventuellement un acquittement de la demande).
Pb : si le coordinateur tombe en panne.

L'approche répartie

Une solution basée sur les horloges logiques avec $2 \times (N - 1)$ messages par entrée en section critique.

L'approche par circulation d'un privilège

Un jeton matérialise l'accès à la section critique. Le jeton circule de processus en processus généralement sur un circuit.

7.5.3 Le problème de l'interblocage

Les politiques de prévention

Les solutions vues au chapitre sur l'interblocage sont plus ou moins adaptables :

Exclusion mutuelle : on utilise un spool pour toutes les ressources qui le permettent.

Tenir et attendre : un protocole à deux phases est souvent mis en place pour les transactions dans les SGBD.

Pas de réquisition : la solution proposée (un processus déclare réquisitionnables ses variables) ne peut pas s'appliquer à toutes les ressources, et peut provoquer la famine.

Attente circulaire : la solution de l'ordre sur les ressources du réseau est difficile à mettre en œuvre.

L'algorithme suivant qui utilise des horloges logiques permet un contrôle des réquisitions qui ne provoque pas de famine. L'idée est la suivante :

1. A chaque processus P_i est associée une estampille E_i qui correspond à son instant de création.

2. L'estampille détermine la priorité du processus, afin qu'un processus retardé deviennent de plus en plus prioritaire.

Il existe deux schémas de mise en œuvre :

wait-die (attendre-mourir) n'utilise pas la réquisition. Dans ce schéma, si P_i désire une ressource détenue par P_j deux cas :

1. P_i attend (un certain temps) si $E_i < E_j$.
2. P_i meurt (il rend ses ressources un certain temps) si $E_i > E_j$.

wound-wait (blessé-attendre) utilise la réquisition. Dans ce schéma, si P_i désire une ressource détenue par P_j deux cas :

1. P_i attend (un certain temps) si $E_i > E_j$.
2. P_i blesse P_j (il le force à rendre ses ressources) si $E_i < E_j$.

Ces schémas sont sans famine si les processus blessés et morts recommencent avec la même estampille après un délai qu'il faut estimer, mais peuvent être très inefficaces.

Les politiques de détection

La détection consiste toujours à trouver un cycle dans le graphe d'allocation des ressources.

Le problème Il est facile pour chaque site S_k de maintenir un graphe d'attente local qui contient tous les arcs P_i, P_j dès que P_i (local ou non) désire une ressource R du site S_k, tenue par P_j (local ou non).

Ces graphes vérifient les propriétés :

1. si l'un deux possède un cycle, il y a un interblocage.
2. il n'y a pas d'interblocage si l'union de l'ensemble des graphes est sans cycle.

Il est donc nécessaire de construire le graphe d'attente global.

Un exemple d'interblocage.

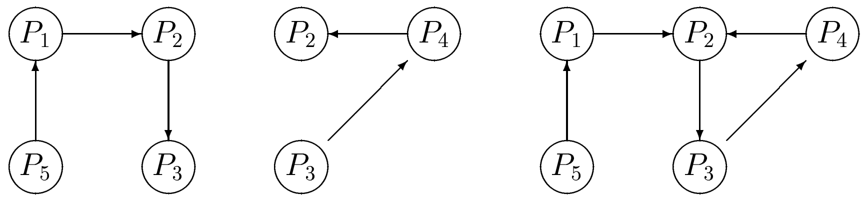

Figure 7.1: Les graphes d'attentes locaux et l'union

L'approche centralisée

Un processus coordonnateur est chargé de gérer le graphe global. Celui-ci peut être maintenu en permanence ou bien calculer périodiquement.

Le défaut majeur est qu'il est impossible d'avoir le graphe réel du fait du temps pris par les communications. Une des conséquences est la détection de faux interblocages (une libération est arrivée trop tard).

Un exemple : Soit la situation suivante :

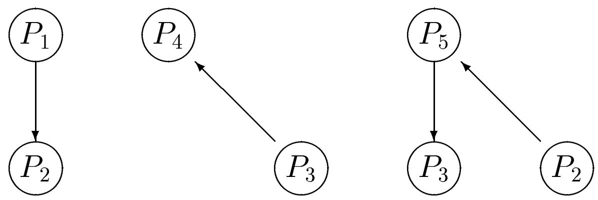

Figure 7.2: Les graphes d'attentes locaux et le graphe global

Supposons maintenant le scénario suivant :

1. P_2 libère la ressource du premier site.
2. P_2 demande une ressource du second site tenue par P_3

et enfin supposons que du fait du réseau le message de la demande arrive au coordinateur avant le message de libération. La situation est alors :

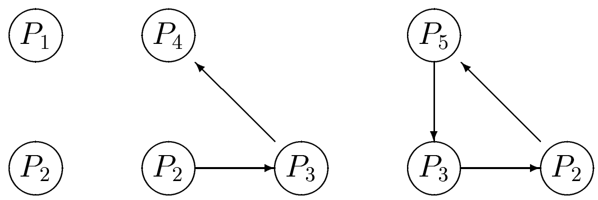

Figure 7.3: Un faux interblocage

Il existe des solutions centralisées basées sur les horloges logiques.

L'approche répartie

Chaque site S_k maintient un graphe composé de :

1. Les arcs P_i, P_j dès que P_i (local ou non) désire une ressource R du site S_k, tenue par P_j (local ou non).

2. Les arcs P_i, P_{ex} dès que P_i local désire une ressource d'un autre site, tenue par un processus quelconque non local.

3. Les arcs P_{ex}, P_i dès que P_{ex} non local désire une ressource R du site S_k, tenue par P_i (local ou non).

Dans ce graphe une attente d'un processus P_i non local sur une ressource locale détenue par P_j apparaît deux fois : P_i, P_j et P_{ex}, P_j

Ce graphe possède les propriétés :

1. si l'un deux possède un cycle sans P_{ex}, il y a un interblocage.

2. si l'un deux possède un cycle avec P_{ex}, il y a risque d'interblocage.

Dans le second cas, notons $C_k = P_{ex}, P_{k_1}, ..., P_{k_n}, P_{ex}$ le cycle du site S_k. Il faut noter que le site S_k connaît exactement la composition de P_{ex} qui débute le cycle, mais pas celle de P_{ex} qui termine le cycle. Le site déclenche l'algorithme suivant :

1. S_k communique le cycle C_k au site S_q qui détient la ressource convoitée par le processus P_{k_n};

2. le site S_q met à jour son graphe. Trois cas :

   (a) il n'existe pas de cycle, il n'y a pas d'interblocage.

   (b) il existe un cycle sans P_{ex}, il y a un interblocage.

   (c) il existe un cycle avec P_{ex}, le site S_q continue l'algorithme de détection vers un nouveau site.

3. les P_{ex} sont à chaque itération de *moins en moins nombreux*, il y a donc un site sur lequel l'algorithme s'arretera, et qui conclura par interblocage ou non.

En fait, s'il y a interblocage *distribué*, plusieurs sites vont posséder un cycle local avec P_ex, il est donc inutile que tous déclenchent l'algorithme. Une optimisation possible utilise un ordre sur les processus :

- le site S_k observe le cycle $C_k = P_{ex}, P_{k_1}, ..., P_{k_n}, P_{ex}$,

- le site S_k déclenche l'algorithme si et seulement si $k_n < k_1$.

Un exemple : Considérons la situation du vrai interblocage initial. Supposons que :

- les processus P_1, P_3, P_5 sont du site S_1,

- les processus P_2, P_4 sont du site S_2,

- les arcs localisent les ressources.

Nous avons alors :

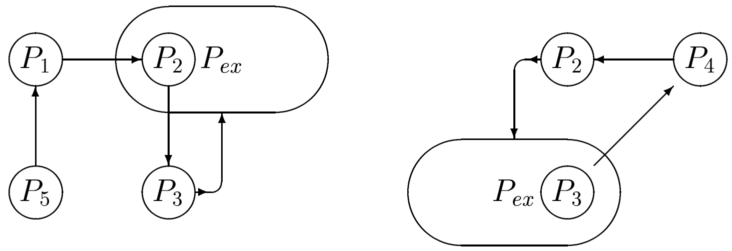

Figure 7.4: Les graphes d'attentes locaux

L'algorithme produit :

1. Les deux sites constatent un cycle :
   - S_1 voit le cycle $P_{ex} = P_2, P_3, P_{ex}$,
   - S_2 voit le cycle $P_{ex} = P_3, P_4, P_2, P_{ex}$,
2. le site S_2 envoie à S_1 le cycle $P_{ex} = P_3, P_4, P_2, P_{ex}$,
3. le site S_1 met à jour son graphe.
4. le site S_1 détecte l'interblocage.

7.5.4 Autres problèmes de coordination réparties

On peut citer :

L'Atomicité : les transactions sont des suites d'actions qui n'en font qu'une. C'est à dire que soit toutes les actions terminent, soit aucune. Les actions peuvent concernées des ressources et des processus sur plusieurs sites. La solution la plus souvent mise en œuvre est le protocole de validation à deux phases.

L'élection : de nombreux algorithmes répartis utilisent un coordinateur. Celui-ci peut tomber en panne, il est alors nécessaire d'initialiser un nouveau coordinateur. Le problème est complexe car :

- il faut être certain que le coordinateur est en panne,
- il faut connaître tous les coordinateurs potentiels.

Une solution est basée sur :

- un processus qui pense que le coordinateur est en panne se proclame nouveau coordinateur, et diffuse l'information à ses voisins;

- un processus qui reçoit un message d'élection participe à l'élection, soit il sait que le coordinateur n'est pas en panne, soit il accepte le nouveau coordinateur, soit il se déclare nouveau coordinateur car il est prioritaire au processus qui lui à envoyer le message. Le processus diffuse cette nouvelle information;
- un processus est élu lorsqu'il à reçu un accord de tous ses voisins.