Systèmes d'exploitation
(Module Maîtrise 97-98)
Université Bordeaux I

Alain Griffault

27 février 1998

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.

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).

Chapitre 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 gère principalement :

- 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.

Une évolution en présentant les caractéristiques et les défauts majeurs.

1.3 Déroutements.

Le *noyau* du système d'exploitation intervient par déroutement (trap) :

- Exception (impossible d'exécuter l'instruction courante).
- Interruption (externe).
- Appel au système (instruction de déroutement volontaire).

Exécution schématique d'un déroutement :

- 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 MMU, 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).

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

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.

Chapitre 2

Gestion des Processus.

2.1 Un processus.

2.1.1 Définition.

Processus = programme + contexte.

2.1.2 Modèle d'un processus.

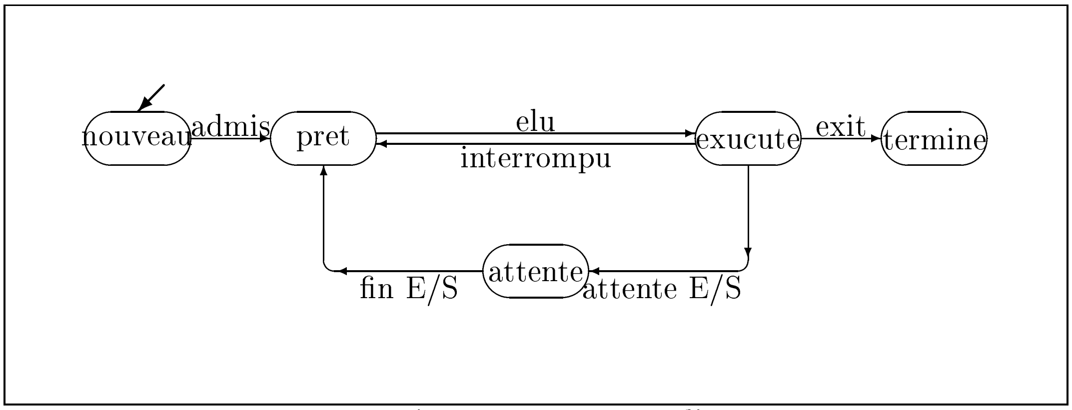

Figure 2.1: Les états et transitions d'un processus

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 différentes phases de l'ordonnancement.

- le changement de contexte.
- la table des processus.
- le bloc de contrôle d'un processus.

2.2.3 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.
- la diminution du temps moyen d'attente.
- la diminution du temps moyen de réponse.
- l'équité.

Les objectifs sont contradictoires, il faut des compromis.

2.2.4 Les algorithmes d'ordonnancement.

FIFO. Utilisation maximale de l'UC. Equitable.

Tourniquet (Round Robin). Equitable. Utilisation de l'UC et temps moyen de réponse dépendent du quantum, ce choix favorise soit les calculs, soit les E/S.

Tourniquet avec priorités statiques. Non équitable. Utilisation de l'UC et temps moyen de réponse peuvent être bons.

Tourniquet avec priorités dynamiques. Equitable. Utilisation de l'UC et temps moyen de réponse sont bons. Diverses stratégies pour la gestion des priorités.

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).

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.

PCTER (Plus Court Temps d'Exécution Restant). Une variante du précédent.

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.

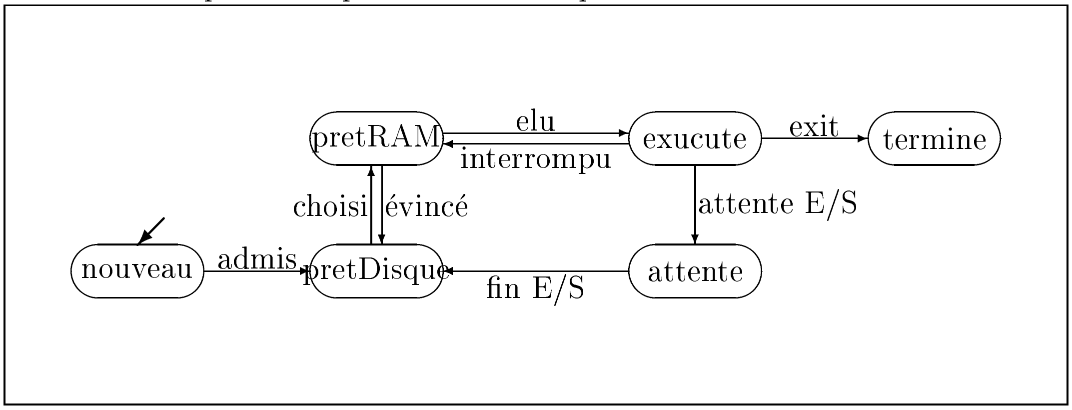

Figure 2.2: Deux niveaux d'ordonnancement

2.3 La synchronisation (communication) de processus.

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

Exemple d'une variable compte modifiée par deux processus, 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, Ecriture.

Le problème devient plus simple si on les change.

- Permettre à l'utilisateur d'interdire momentanement les interruptions (difficile et dangereux).
- Augmenter l'ensemble des actions atomiques (le jeu d'instructions).

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

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

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

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

Certains processeurs possèdent une instruction TestAndSet.

procedure swap permet l'échange de deux booléens.

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

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

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

Certains processeurs possèdent une instruction swap.

les procedures wait et signal manipulent des entiers.

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

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

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

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

Aucun processeur ne possède ces instructions.

Il existe des solutions équitables.

Le défaut commun à ces solutions est l'attente active (gaspillage de l'UC).
Le SE doit fournir des appels systèmes pour endormir et réveiller des processus.

Les appels systèmes de synchronisation.

On suppose l'existance 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 reveiller tous les processus "en attente" grâce à un `wakeup(P)`

Cette solution est moyennement satisfaisante, car en fait un seul proceesus doit être reveillé, 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 deux façons :

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

- Les sémpahores 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

- emettre(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 :

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

Une solution au problème de la section critique. utilse un processus controleur qui délivre les autorisations d'entrée en SC.

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

```
        Section Non Critique
        fin
fin

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

Equivalence de modèles Il est possible d'implémenter les messages à l'aide des sémaphores et réciproquement.

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.

Equivalence 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
```

Les moniteurs

Un moniteur est déclaré ainsi :

2.3.4 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 comsommateur les retire. La synchronisation doit satisfaire les points suivants :

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

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.

Chapitre 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 Allocation contiguë

Les processus constituent un seul bloc non décomposable.

3.2.1 Partitions de taille fixe

SE	P1	P6	vide	P4	vide

Le défaut majeur de cette approche est la nécessité de faire du recouvrement lorsque la taille du processus est supérieure à celle de la partition.

3.2.2 Partitions de taille variable

SE	P1	P6	vide	P4	vide

Le compactage de la mémoire, après chaque libération, est trop coûteux. Liste chaînée de zones libres, réalisée en utilisant le début des zones libres.

1. Libération : fusionner les zones libres contiguës (4 cas).
2. Allocation : on estime la taille du processus (code, données, tas, pile), puis on scinde une zone libre en une zone allouée et un reste (libre).

   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.

3. En moyenne, le nombre de zones libres est la moitié du nombre de zones allouées.

3.2.3 Fragmentation

Interne $\sigma(taillesprocessus)/\sigma(taillesallouees)$

externe $\sigma(taillestrous)/(taillememoire)$

3.2.4 Va-et-vient

Cela correspond à 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.

3.2.5 Les problèmes

Taille d'un processus

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

La protection

Il faut controler, **avant exécution** que toutes les adresses accedées par le processus sont à l'intérieur de la zone allouée. Cela protège les utilisateurs entre eux, mais aussi le SE.

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 :

compilation code absolu, non relogable.

chargement code translatable, non relogable.

exécution code relogable. Cela demande du matériel spécial.

3.3 La pagination : une allocation non contiguë

L'allocation contiguë est très contraignante. Une solution consiste à découper en morceaux le processus.

3.3.1 Mémoire virtuelle

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

La mémoire logique est plus grande que la mémoire physique.

Principe de la pagination, MMU. Table des pages : chaque entrée spécifie aussi si la page est présente, protégée, modifiée, ou utilisée.

3.3.2 Défauts de pages

Traités par logiciel (système d'exploitation). Il faut ensuite relancer l'instruction.

3.3.3 Tables à plusieurs niveaux

Permettent de réduire considérablement la taille de la table des pages. Découpage 10-10-12 (intel 80386) pour une table à deux niveaux, 8-6-6-12 sur Sparc.

3.3.4 Mémoire associative

Cache ; hits et miss. Machine MIPS : les échecs de recherche dans le cache sont traités par le logiciel.

3.3.5 Changement de contexte

Chaque processus a ses propres tables de pages ; ou bien leur sauvegarde et leur rechargement font partie du changement de contexte (lourd), ou bien la MMU gère une table de contextes (Sparc). Penser aussi à invalider les entrées de la mémoire associative, si celle-ci ne contient pas de champ "processus".

3.3.6 Tables inverses

Pour adressage 64 bits. Une fonction de hachage permet de retrouver le numéro de page virtuel. Table des pages manquantes sur disque.

3.4 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.

3.4.1 Algorithme optimal

La solution optimale consiste à évincer la page dont l'occurence 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 apparaitre 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. Il est malheureusement non implémentable (hormis dans certaines requêtes dans un SGBD), car il fait l'hypothèse de la connaissance du futur.

3.4.2 LRU

Est une bonne approximation de l'algorithme optimal si le principe de localité est respecté.

3.4.3 FIFO, horloge

Horloge : une page avec R=1 est conservée, avec remise à zéro de R. Dit aussi algorithme de la seconde chance. Variante avec deux aiguilles, qui permet de paramétrer la notion d'accès récent.

3.4.4 Compteur de références

Pour assurer le vieillisement, faire un décalage à droite, et placer R dans le bit de poids fort du compteur.

3.4.5 Algorithmes globaux/locaux

On peut traiter globalement toutes les pages de la mémoire, ou traiter à part les pages attribuées à un processus.

3.4.6 Pré-chargement

S'oppose à la pagination à la demande.

3.4.7 Démon

Chargé de libérer des cadres, à intervalles réguliers, si le nombre de cadres libres tombe au-dessous d'un seuil, jusqu'à ce qu'il devienne supérieur à un autre seuil.

3.5 La segmentation : une autre allocation non contiguë

Un segment est une unité logique, de grande taille, caractérisé par sa base et sa taille (variable). Possède ses propres modes de protection. Rend inutile les translations d'adresses pendant le chargement du code. La segmentation est bien adaptée à la projection de fichiers en mémoire (MULTICS).

Chapitre 4

Fichiers

4.1 Caractéristiques

Informations de grande taille, permanentes et partagées. L'utilisateur confond souvent système d'exploitation et système de fichiers.

4.2 Structure

L'unité est l'octet ou l'enregistrement. Accès séquentiel, aléatoire ou indexé.

4.3 Opérations

Ouverture, lecture, écriture, positionnement. Autre solution : projeter le fichier en mémoire ; mais on ne contrôle plus les recopies entre mémoire vive et disque.

4.4 Blocs

Le pilote de disque permet de le voir comme une suite de blocs numérotés.

Allocation contiguë rare, sauf pour des applications particulières, comme les bases de données. Le chaînage bloc par bloc ralentit trop l'accès direct, sauf si la table d'allocation est en mémoire (MS-DOS). Solution Unix : l'inode contient la liste des premiers blocs, puis un bloc indirect, etc...

Gestion des blocs libres par liste, ou par table booléenne.

Pour une meilleure efficacité, les blocs d'un fichier doivent se trouver sur des cylindres voisins.

4.5 Noms et catalogues

Une entrée de catalogue peut comporter, outre le nom, les attributs (dates, droits d'accès, taille, propriétaire) et le numéro du premier bloc (MS-DOS), ou bien simplement un numéro d'inode (Unix). La solution Unix permet plusieurs chemins d'accès (liens durs), et exige un compteur de références dans l'inode, pour savoir quand le fichier n'est plus accessible.

Liens symboliques : fichiers spéciaux, contenant un chemin d'accès.

4.6 Cohérence

Un bloc peut manquer (ni libre ni occupé), ou être à la fois libre et occupé. Anomalies dans les compteurs de références. Vérification et réparation périodiques.

4.7 Buffers

Antémémoire, ou cache. Gestion LRU. Recopie périodique sur disque (sync) ou immédiate (MS-DOS, write-through).

4.8 Sécurité

Droits d'accès, listes de capacités. Ver et virus. Ver Internet du 2 Novembre 88, produit par Morris, étudiant de l'université Cornell.

Chapitre 5

Entrées/Sorties

5.1 Contrôleurs

Adresses des registres mappées en mémoire. Vecteur d'interruption. Mémoire tampon, code détecteur d'erreurs et accès DMA pour les contrôleurs de disques ; entrelacement des secteurs.

5.2 Pilotes

Fragment de logiciel adapté au contrôleur. Transforme les requêtes de haut niveau (numéro de bloc $\longrightarrow$ piste, secteur). Traite les interruptions et les erreurs.

5.3 Couches logicielles supérieures

Dans le système, logiciel indépendant des périphériques. Allocation et libération de buffers. Périphériques blocs et caractères.

Les logiciels utilisateurs font de la mise en forme (printf), ou gèrent des files d'attente (spool).

5.4 Ordonnancement du disque

Temps de recherche, temps de latence, et temps de transfert. FIFO, PCTR, et balayage (ou ascenseur). Antémémoire.

5.5 Horloges

Interruptions à intervalles réglables.

Mise à jour du temps : 32 bits ne suffisent pas pour un temps précis à 1/50ème de seconde près.

Gestion des alarmes par une liste chaînée ordonnée et “différentielle”.

Compteurs de garde (watchdog timers).

5.6 Terminaux

Liaisons séries : UART = Universal Asynchronous Reveiver/Transmitter. RS/232.

Terminaux mappés en mémoire ; bitmaps, pixels.

Claviers : modes cru et cuit. Conversion entre événements (enfoncement ou relâchement d'une touche) et codes ASCII.

5.7 Serveurs X

Interposer un logiciel entre le client, d'une part, l'écran, le clavier et la souris d'autre part. Boucle de gestion des événements.

Chapitre 6

Interblocages

6.1 Conditions d'interblocage

1. Exclusion mutuelle.
2. Tenir et attendre.
3. Pas de réquisition.
4. Cycle

6.2 Détection et reprise

Détection de cycle dans un graphe. Algorithme dans le cas de plusieurs ressources de même type. Vecteur E des ressources existantes, matrice T des ressources tenues, vecteur D des ressources disponibles, matrice R des requêtes.

6.3 Evitement

Configurations sûres : graphique dans le cas de deux processus. Algorithme du banquier : on remplace, dans l'algorithme de détection d'interblocages, la matrice des requêtes, par $M - T$, où M est la matrice des besoins maximaux.

6.4 Prévention

Eliminer une des conditions de l'interblocage :

1. Exclusion mutuelle : spool.
2. Tenir et attendre : protocole deux phases.
3. Pas de réquisition : ?
4. Cycle : ordonner les ressources, et n'allouer qu'une ressource de numéro supérieur à ceux des ressources détenues.