Systèmes d'exploitation

Cours : Robert Strandh

TD : Abdou Guermouche, Martin Raspaud, Robert Strandh

Support de cours

Diaporama, notes de cours.

Code du système Nachos

Bibliographie

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne : “Principes appliqués des systèmes d’exploitation avec Java” ; Vuibert

Andrew Tanenbaum : “Systèmes d’exploitation” ; Pearson 2003

Jean Baco, Tim Harris : “Operating Systems” ; Addison-Wesley 2003

Laurent Bloch : “Les systèmes d’exploitation des ordinateurs” ; Vuibert 2003 (plutôt historique)

Autres documents

Notes de cours en ligne de Willy Rouaix.

Objectifs de l'UE

- Comprendre comment marche un système d'exploitation
- Comprendre pourquoi il est conçu de cette façon

Prérequis

UE : programmation impérative, programmation système, architecture de l'ordinateur, utilisation de systèmes informatique, environnements de développement.

En particulier : bonne connaissance de la programmation en C, maîtrise des outils (GCC, GDB, Emacs, make, CVS), architecture (interruptions, pile d'exécution, passage de paramètres, cache).

Contenu des cours

- Historique
- Structure et fonctionnement d'un système "classique"
- Support matériel
- Processus, processus légers, ordonnancement
- Communication entre processus, synchronisation
- Gestion de la mémoire
- Gestion de fichiers

Contenu des TD

Lecture et modification d'un (simulateur d'un) petit système d'exploitation : Nachos

Travail individuel

Environ 4h par semaine

Lecture de code, programmation, lecture de la littérature.

Contrôle des connaissances

(selon la fiche d'UE)

Projet et DS

Examen (1h30)

Définition “Système d’exploitation”

Un système d’exploitation est une collection de code dont le but est de faciliter l’écriture d’applications mutuellement *indépendants* capables de *partager des données* de manière *maîtrisée* afin de permettre à l’utilisateur de travailler de façon *efficace*.

Nous allons particulièrement regarder une partie du système appelée le “noyau”.

Autres définitions possibles

Dans la littérature, on trouve d'autres définitions :

- un logiciel destiné à faciliter l'utilisation d'un ordinateur (on peut très bien imaginer une application très utilisable sans système d'exploitation)

- un logiciel pour optimiser l'utilisation des ressources comme l'UC, les disques, la mémoire, etc (cette définition est toujours correcte, mais moins importante)

Différentes parties d'un système

- Le noyau
- Démons (anglais: daemon); des programmes dont le but est d'exécuter en permanence afin de fournir certains services (sshd, inetd, etc)
- Programmes utilitaires (grep, sed, awk, etc)
- Interface graphique
- Outils de développement (éditeurs, compilateurs, débogueurs, etc)

Noyau

Le noyau du système est la partie qui est exécutée en mode privilégié.

Il est responsable de présenter aux applications une “machine virtuelle” (ou un API) qui est une extension d’un sous-ensemble du processeur.

En particulier, il doit protéger une application des autres, tout en permettant le partage de données entre les applications.

Faciliter l'écriture d'applications

Le système permet aux auteurs d'applications de ne pas avoir besoin de :

- connaître le fonctionnement exact des périphériques.
- connaître l'adresse où l'application sera chargée en mémoire
- s'inquiéter de la possibilité qu'une application perturbe le fonctionnement d'une autre

Point sur le support matériel

Deux mécanismes sont essentiels :

- Les *interruptions*; peuvent arriver après l'exécution de n'importe quelle instruction, mais jamais au milieu de son exécution;

- Les *appels système*; c'est un cas spécial d'une interruption (provoquée par l'exécution d'une instruction)

Dans les deux cas, le processeur passe du mode *utilisateur* au mode *privilégié* (ou mode *noyau*, ou mode *superviseur*).

Point historique

Nous n'avons pas trop le temps de parler de l'historique des systèmes.

Cela ne veut pas dire que c'est sans importance (au contraire)

Bagage historique

Plusieurs mécanismes dans les systèmes d'aujourd'hui existent uniquement pour des raison historiques. Exemples :

- la notion de *processus* (avec un espace d'adressage séparé) existe uniquement parce que l'espace d'adressage des processeurs était historiquement trop petit. Maintenant, nous avons des espaces de 2^{64} ou en tout cas 2^{48} octets ;

- la notion de *fichier* (séquence d'octets) existe uniquement parce que les bandes magnétiques ne permettaient pas un accès direct comme la mémoire centrale ou le disque.

Moderne $\neq$ bon

Certains systèmes du passé étaient meilleurs que les systèmes couramment utilisés aujourd'hui :

- le système *Genera* n'avait pas de processus, ce qui simplifie considérablement l'écriture d'applications ;
- le système *Multics* n'avait pas de fichiers, mais une arborescence de vecteurs de mots dans une seule mémoire virtuelle.

Question : pourquoi nous n'utilisons pas ces systèmes aujourd'hui?

Nachos

C'est un *simulateur* d'un système d'exploitation rudimentaire (écrit en C++).

Il permet de comprendre le fonctionnement d'un système typique à travers le suivi de l'exécution (GDB) et la modification du code.

Il est rudimentaire, car il ne contient même pas les appels système les plus fondamentaux (comme `write`), Une application Nachos ne peut donc pas se servir de `printf` par exemple.

Nachos (suite)

Nachos était écrit initialement pour l'architecture MIPS.

Maintenant porté sur x86, mais les applications sont toujours en code MIPS.

Nachos contient donc un simulateur pour cette architecture.

Il faut donc deux compilateurs, un compilateur natif pour le système et un compilateur croisé pour les applications.

Utilisation d'un simulateur simplifie la mise au point (pas d'interruptions aléatoire)

Processus

Un processus est un programme + son état d'exécution.

États possibles : En cours d'exécution (E), bloqué (B), prêt (P).

Passage entre E et P : ordonnanceur.

Passage E $\rightarrow$ B : demande d'entrée/sortie.

Passage B $\rightarrow$ P : demande satisfaite.

Processus (suite)

Chaque processus a son propre *espace d'adressage*.

Aujourd'hui cela est possible grâce à la *pagination*, traitée plus tard (gestion de la mémoire).

Dans le passé, d'autres systèmes était employés, tels que le *va-et-vient*.

Représentation d'un processus

Le système d'exploitation gère une *table de processus*.

On appelle les entrées de cette table *processus* ou *blocs de contrôle de processus*, qui sont des structures ou des instances de classes.

Bloc de contrôle de processus

Un tel bloc contient :

- état du processus
- compteur ordinal
- contenu des registres (y compris compteur ordinal)
- table des pages
- fichiers ouverts
- etc

Commutation (changement de contexte)

Voir `threads/switch.S` de Nachos.

Espaces d'adressage séparés

On utilise une *table des pages* en mémoire (virtuelle).

La référence à la table des pages se trouve dans un registre spécial que l'on peut modifier seulement en mode privilégié.

Pour un processeur moderne, la table contient plusieurs *niveau*.

Translation look-aside buffer (TLB)

Passer par la table des pages à chaque référence serait trop lent.

On utilise un TLB (une sorte de petit cache associatif).

Un mot sur le cache

Il y a souvent deux niveaux de cache aujourd'hui (L1 et L2).

Un défaut de cache peut coûter des centaines de cycles d'horloge.

Le cache peut être indexé soit par des adresses *virtuelles*, soit par des adresses *physiques*.

Dans le premier cas, le cache est consulté d'abord, dans le deuxième cas, la traduction par le TLB est effectuée d'abord.

Processus légers (threads)

Un *processus léger* est similaire à un processus, mais partage potentiellement son espace d'adressage avec d'autres processus légers.

Avantage : partage de mémoire plus facile

Inconvénient : mise au point plus difficile

Réalisation des processus légers

Ne pose pas de problème de principe. Il suffit que deux "processus" partages la même table des pages.

On souhaite souvent se servir de processus léger pour la performance (par exemple dans un serveur Web).

Processus léger Unix

Traditionnellement, Unix n'avez que des processus (pourquoi?).

Implémentation de processus léger dans un processus Unix est (presque) possible.

Bibliothèque de code pour l'utilisation de processus légers.

Une telle implémentation est souvent très rapide.

Ordonnancement (scheduling)

Un *ordonnanceur* (anglais : *scheduler*) est une partie d'un système d'exploitation dont le but est de choisir un processus (léger) *prêt* à exécuter sur un processeur "libre".

Exécution de l'ordonnanceur

l'ordonnanceur exécute :

- lorsqu'un processus cède l'UC parce qu'il termine son exécution,
- lorsqu'un processus cède l'UC parce qu'ilest bloqué (attente de périphériques, attente explicite),
- lorsqu'un processus passe à l'état *prêt* parce qu'il vient d'être créé,
- lorsqu'un processus passe à l'état *prêt* parce que son attente est terminée,
- après une interruption par l'horloge.

Ordonnancement (contexte historique)

(toujours valable dans certaines situations)

Dans un système de *traitement par lot* (batch), le but traditionnel de l'ordonnancement est de choisir des processus (légers) de manière à optimiser l'exécution d'une collection de tels processus (selon différents critères)

Algorithmes d'ordonnancement

Il y en a plusieurs dans la littérature.

Ils utilisent des structures de données plus ou moins complexes contenant des processus dans l'état prêt.

Chacun est basé sur des critères à optimiser (temps UC, temps d'attente, temps de réponse, etc).

Fréquence de l'ordonnancement

Afin d'avoir un système réactif, l'ordonnanceur doit exécuter assez souvent.

Pour éviter top de perte de performance, il ne doit pas exécuter trop souvent, ni prendre trop de temps à exécuter.

Ordonnancement (suite)

Critères “modernes” sur un ordinateur personnel :

- respecter des délais (le tampon de la carte son ne doit jamais être vide, génération d’images d’un film)
- éviter des pertes de performances (cache, TLB)

Influence de l'architecture sur l'ordonnancement

Il n'y aura probablement plus d'augmentations considérables de la fréquence de l'horloge.

Aujourd'hui, il y a des architecture de type "hyper-thread" et "multi-core".

Un algorithme d'ordonnancement doit en tenir compte.

Synchronisation

Le problème :

Plusieurs processus exécutent simultanément, ce qui peut poser problème si ces processus doivent collaborer. Exemples :

- le problème producteur/consommateur (tubes Unix)
- le problème de la section critique

Producteur/consommateur

(lecture du programme `producer_consumer.c`)

Synchronisation, logiciel

Il existe des *algorithmes* de synchronisation implémentés entièrement en logiciel, mais ces algorithmes ne sont plus d'actualité.

Synchronisation, support matériel

Suppositions :

- une instruction est soit exécutée entièrement, soit pas du tout
- dans un système avec plusieurs processeurs, l'effet global est comme si deux instructions ne puissent pas être exécutées simultanément (voir le protocole de cohérence des caches)
- une interruption peut arriver après l'exécution de n'importe quelle instruction

Instruction : `test-and-set`

Il y a plusieurs variantes.

`TAS addr`

Description :

```
CC[C] = M[addr][0]
M[addr][0] = 1
```

Instructions pour la synchronisation

D'autres instructions possibles :

- test-and-test-and-set
- swap
- compare-and-swap
- fetch-and-add
- load-link/store-conditional

Abstraction test-and-set

On peut donc supposer l'existence d'une fonction C :

```
int
test_and_set(int *lock)
{
  /* use test-and-set instruction */
  /* return CC[C] */
}
```

Exclusion mutuelle avec test-and-set

```
static int lock = 0;
static int count = 0;

static void
increment_count()
{
  while (test_and_set(&lock))
    ;
  count++;
  lock = 0;
}
```

Exclusion mutuelle avec test-and-set

```
static void
decrement_count()
{
  while (test_and_set(&lock))
    ;
  count--;
  lock = 0;
}
```

Sémaphores

Une abstraction de synchronisation.

Un objet contenant un entier $i \geq 0$.

Deux opérations :

`P(s)` : attendre que $i > 0$, puis décrémenter i.

`V(s)` : incrémenter i.

Ces opérations sont *atomiques*.

Utilisation de sémaphores

Exclusion mutuelle :

Selon la tradition, on appelle un sémaphore binaire *mutex* (pour *mutal exclusion*).

```
semaphore mutex = make_semaphore(1);
...
P(mutex);
/* section critique */
...
V(mutex);
```

Utilisation de sémaphores

Producteur/consommateur : (lecture du programme `producer_consumer2.c`)

Lecteurs et Écrivains

(Readers and Writers problem)

Définition : un nombre arbitraire de lecteurs est permis, mais un écrivain doit être seul.

```
semaphore mutex = make_semaphore(1);
semaphore write = make_semaphore(1);
int read_count = 0;
```

Lecteurs et Écrivains

```
void write(...)
{
  P(write);
  /* écrire */
  V(write);
}
```

Lecteurs et Écrivains

```
... read(...)
{
  P(mutex);
  read_count++;
  if(read_count == 1)
    P(write);
  V(mutex);
  /* lire */
  P(mutex)
  read_count--;
  if(read_count == 0)
    V(write);
  V(mutex);
}
```

Interblocage

```
f(...)
{
  P(m1);
  P(m2);
  ...
}

g(...)
{
  P(m2);
  P(m1);
  ...
}
```

Famine

Voir le problème des lecteurs et écrivains.

Implémentation de sémaphores

(lecture du programme `semaphore.c`)

Autres primitives de synchronisation

La notion de sémaphore n'est pas la seule possibilité.

Variables de condition

Une *variable de condition* est une abstraction de synchronisation à utiliser avec un mutex.

Il y a deux opérations sur les variables de condition :

```
wait(mutex, cond)
```

```
notify(cond)
```

Variables de condition

L'opération `wait` fait `V(mutex)` et attend la notification (le processus est bloqué). Une fois la notification arrive, `wait` fait automatiquement un `P(mutex)`.

L'opération `notify` réveille un processus bloqué (s'il y en a).

Utilisation de variables de condition

Exemple : producteur/consommateur (non borné)

```
consume(...)
{
  P(mutex);
  while(empty(q))
    wait(mutex, cond);
  dequeue(q);
  V(mutex);
}
```

Utilisation de variables de condition

```
produce(...)
{
  P(mutex);
  enqueue(q);
  notify(cond);
  V(mutex);
}
```

Constructions linguistiques

Il est difficile de programmer avec des primitives de synchronisation.

Une solution possible : augmenter les langages de programmation avec des constructions (syntaxe) pour simplifier l'utilisation.

Exemple : *moniteur*

Moniteur

Un *moniteur* est similaire à une *classe* dont un sous-ensemble des méthodes sont exécutées dans le contexte d'un mutex.

Certains problèmes de synchronisation sont impossibles à résoudre avec ce mécanisme seul.

C'est pourquoi un moniteur est utilisé avec des variables de condition.

Moniteur en Java

(lecture du programme `monitor.java`)

Difficultés des constructions linguistiques

Ça marche uniquement si :

- le langage contient déjà les constructions adéquates, ou
- le langage permet des *extensions syntaxiques*.

Gestion de la mémoire

Problèmes de la multi programmation

- un programme chargé à n'importe quelle adresse en mémoire
- éviter la fragmentation de la mémoire
- taille des programmes supérieure à celle de la mémoire physique
- bibliothèques partagées
- code partagé
- la sécurité (protection).

Édition de liens

Un fichier *objet* (.o) contient des trous.

Chaque trou est étiqueté avec :

- un *segment* (code, données)
- taille du trou
- décalage depuis le début du segment

Édition de liens

L'éditeur de liens combine plusieurs fichiers objets pour en créer un fichier *exécutable*.

Il combine les segments du même type des différents fichiers objets.

Linking loader

Problème (rappel) : un programme chargé à n'importe quelle adresse en mémoire.

Une solution est d'appeler l'éditeur de lien à chaque chargement du programme.

Segmentation

Utiliser un registre de base pour chaque segment. Toute adresse émise par le programme doit être relative à l'un de ces registres.

Adressage relatif au compteur ordinal

Anglais: PC-relative addressing,

Chaque adresse émise par le programme est relative à la valeur actuelle du compteur ordinal.

Ça marche bien pour le code et moins bien pour les données.

Le code est facile à déplacer.

Pagination

Une *adresse virtuelle* est divisée en deux parties : numéro de page et décalage.

Le numéro de page est traduit par le mécanisme de pagination en *numéro de cadre*.

Protection : la table des page contient des *bits de protection* (lecture, écriture, exécution) pour chaque page.

Pagination à la demande

Une partie (certaines pages) du programme se trouve sur mémoire secondaire (disque).

Pour les pages sur disque, l'adresse indiquée dans la table des pages est une adresse *disque*.

Une tentative d'accéder à une page sur disque provoque un appel au système d'exploitation.

Traitement d'un défaut de page

- Vérifier si la référence est *valide*. Si non alors terminer le processus.
- Si aucun cadre libre n'existe, choisir un cadre *victime* à écrire sur disque (si modifié) et libérer la cadre.
- Lancer une opération de lecture de la page sur disque dans le cadre libre, et bloquer le processus.
- Lorsque la lecture se termine, modifier la table des page et réveiller le processus.

Algorithmes de remplacement des pages

Le problème : choisir une page *victime*.

Le but est de minimiser le *taux de défaut de pages*.

Algorithme optimal

[mais malheureusement impossible à implémenter] Choisir comme victime la page qui sera utilisé le plus tard dans le futur.

FIFO

Une file de page est maintenue.

L'anomalie de Belady : avec FIFO, le taux de défaut des pages peut *augmenter* avec plus de mémoire physique disponible.

Moins récemment utilisée

Anglais : Least Recently Used (LRU)

Choisir comme victime la page la moins récemment utilisée.

Malheureusement, ce serait trop coûteux.

LRU approximatif

Le matériel maintient un *bit de référence*, mis à jour à chaque accès mémoire.

Le système s'en sert pour déterminer une approximation de la date d'utilisation de la page.

Algorithmes :

- Registre de décalage
- Deuxième chance (ou "horloge")

D'autres algorithmes

LFU, MFU

Working Set

Une définition approximative.

C'est l'ensemble des page dont un processus a besoin pour travailler.

Si la somme des WS dépasse la taille de la mémoire physique, le système passe son temps dans la pagination.

Effets sur la programmation

Localité du programme (même considération que pour le cache).

Verrouillage de pages

Pour certaines pages, il n'est pas souhaitable de les expulser de la mémoire physique :

- la page contenant le code qui implémente la pagination et certaines autres parties du système d'exploitation.
- des pages cibles d'un transfert d'entrée/sortie d'accès direct à la mémoire (DMA).

Organisation de la table des pages

- tables des pages inversée
- table de hachage

Bibliothèques

"Linkers and Loaders" John R. Levine, Morgan Kaufmann 2000 (environ 35 €)

http://www.iecc.com/linker/

Trois types :

- non partagées
- partagée, statiques
- partagée, dynamiques

Bibliothèques non partagées

Une collection de *modules* dans un fichier *archive*.

Un module peut contenir du code et/ou des données.

L'éditeur de liens choisit seulement les modules nécessaires de manière itérative.

Ici "nécessaire" signifie "afin de résoudre une référence à un symbole" (par exemple `printf` ou `malloc`).

Cette sémantique permet à une fonction définit par le programmer de supplanter la version dans la bibliothèque (cas fréquent de `malloc` et `free`)

Problèmes des bibliothèques non partagées

Espace disque (négligeable)

Espace mémoire (à cause de la multi programmation).

Bibliothèques partagées statiques

Une partie de l'espace d'adressage de chaque processus est consacrée aux bibliothèques partagées.

Cela nécessite une gestion globale de l'espace d'adressage des bibliothèques.

Deux parties d'une bibliothèque statique

Le fichier exécutable à mapper dans chaque processus (grâce à `mmap`).

Un fichier à utiliser par l'éditeur de liens pour résoudre des références.

Mise à jour de la bibliothèque

Problème : il ne faut pas que les adresses des symboles dans la bibliothèque changent.

Solution : construire une *table d'indirection*:

```
putchar:        jmp _putchar
getchar:        jmp _getchar
printf:         jmp _printf
...

_putchar(...)
{
  ...
}
...
```

Problèmes de sémantique

Impossible de faire référence de la bibliothèque à un symbole défini par le programmeur d'une application.

En particulier, impossible pour un symbole dans l'application de supplanter un symbole dans la bibliothèque.

Bibliothèques partagées dynamiques

Une bibliothèque dynamique est un fichier ELF (par exemple) à mapper dans l'espace d'adressage du processus.

La bibliothèque peut contenir du code et des données.

Premier problème à résoudre : indépendance de la position dans l'espace.

Indépendance de la position (PIC)

Pas de problèmes de références de l'application au code ou aux données de la bibliothèque, car l'éditeur de lien peut modifier les adresses de l'application.

Exemple : biblio.c

Pas de problème non plus de références de la bibliothèque au code de la bibliothèque (adressage relatif au compteur ordinale).

Problèmes de références de la bibliothèque aux données de la bibliothèque et de l'application.

Problème de références d'une bibliothèque au code d'une autre bibliothèque.

Références aux données de l'application

La taille du code est connue. On connaît donc la distance entre une instruction quelconque et le début du segment de données.

L'éditeur de liens crée une table globale (GOT, global offset table) au début du segment de données.

À l'exécution, la GOT contiendra les adresses des données de l'application. Elle est remplie par l'*éditeur de liens dynamique*.

Exemple : biblio1.c

Références aux données de la bibliothèque

L'adresse de chaque variable est fixe par rapport au début du segment de données, et donc par rapport à la GOT.

Exemple : biblio2.c

Édition de liens paresseuse

Raison : la plupart des fonctions d'une bibliothèque ne sont jamais appelées, et il peut y en avoir un nombre considérable.

Il est donc inutile pour l'éditeur de liens dynamique de remplir les adresses de toutes les fonctions.

Table de pointeurs sur fonctions (PLT)

Dans le segment code de chaque fichier (application ou bibliothèque), l'éditeur de liens génère une PLT.

```
PLT0:  pushl GOT+4
       jmp *GOT+8
PLT1:  jmp *GOT+12
       push #reloc_offset
       jmp PLT0
PLT2:  jmp *GOT+16
       push #reloc_offset
       jmp PLT0
...
```

GOT+4 contient un identifiant du fichier, GOT+8 contient l'adresse de l'éditeur de liens dynamique.

Avantages des bibliothèques dynamiques

Code partagé par toutes les applications.

Sémantique plus proche de celle des bibliothèques non partagées.

Possibilité de rajouter des bibliothèques à l'exécution (plug-ins) grâce à `dlopen`, etc.

Inconvénients

Performance :

- lors du démarrage de l'application
- appels de fonctions indirectes
- indirection pour les références aux données globales

Pourquoi cette complexité

C'est parce que nous avons des un espace d'adressage pour chaque processus.

Et ça, c'est parce que l'espace d'adressage physique du processeur est insuffisant pour la multi programmation, même avec des processeurs 32bits.

Question : Qu'est-ce qui va se passer avec les processeurs de 64bits.

Gestion du disque

Raisons de l'existence du disque :

- le disque coûte moins cher que la mémoire centrale par unité de stockage
- le disque est non volatile

Attention, la situation peut changer dans quelques années seulement avec la mémoire MRAM.

Structure physique

Plusieurs disques chacun avec un bras pour déplacer la tête de lecture/écriture.

Déplacement des bras grâce à un solénoïde.

Environ 10000 rotations par minute (environ 200 par seconde).

Environ 10ms pour déplacer le bras d'un endroit à un autre (selon la distance).

L'unité d'accès est le "secteur" (presque toujours 512 octets).

Contrôleur

Il s'agit de circuits, physiquement intégrés avec le disque permettant la communication avec l'unité centrale.

Aujourd'hui, les contrôleurs sont de plus en plus "intelligents" et contiennent une certaine quantité de mémoire RAM.

Gestion de l'espace libre

Vecteur contenant un bit par secteur indiquant si le secteur est libre.

Liste chaînée de secteurs libres avec variations.

Liste chaînée de zones de secteurs libres.

Allocation d'espace

Contiguë.

Chaînée.

Table d'allocation de fichiers.

Indexée.

Allocation Contiguë

Un fichier est une zone linéaire de secteurs.

Problèmes de fragmentation.

Accès rapide.

Difficulté de déterminer a priori la taille d'un fichier.

Allocation Chaînée

Un fichier est une liste chaînée de secteurs.

Accès non séquentiel au fichier n'est pas efficace.

Il n'est pas pratique d'avoir un pointeur dans chaque secteur.

Table d'allocation de fichiers (FAT)

Variation de la méthode chaînée.

Les pointeurs sont alloués séparément dans une table sur disque.

La table contient une entrée pour chaque secteur du disque.

La difficulté d'accès séquentiel existe toujours.

Allocation indexée

Le premier secteur d'un fichier contient une table de pointeurs sur les secteurs du fichier.

Accès direct simple.

Perte d'espace considérable pour les petits fichiers.

Il faut permettre à l'index de dépasser un secteur (index chaîné, multi niveaux, combinaison des deux).

Ordonnancement du disque

Il s'agit d'optimiser les accès lorsque plusieurs requêtes d'E/S existent simultanément, à cause de la multi programmation.

Attention: De plus en plus, l'ordonnancement et fait par des logiciels contenus dans le contrôleur du disque.

FIFO

C'est la méthode la plus simple, la première requête arrivée est la première servie.

Peut provoquer beaucoup de mouvements inutiles du bras.

Minimisation de déplacement

On connaît[*] la position du bras après la requête précédente.

On satisfait la requête en attente ayant la position la plus proche de la précédente.

Cette méthode peut provoquer la *famine*.

[*] attention au contrôleur

Méthode circulaire (C-LOOK)

On satisfait la requête ayant la plus petite adresse *supérieure ou égale à la précédente.*

Si aucune adresse n'est supérieure ou égale à la précédente, alors on satisfait la requête avec la plus petite adresse.

Variations : SCAN, C-SCAN, LOOK.

Choisir un algorithme

L'efficacité dépend de plusieurs facteurs :

- la méthode d'allocation d'espace du disque
- le type d'accès aux fichiers
- l'endroit où se trouvent les répertoires
- le type d'utilisation du système (serveur, personnel)
- etc

Optimisation des rotations

Difficile en général, sauf pour le contrôleur.

Fichiers

Abstractions de la mémoire

Un système d'exploitation typique présente deux types d'abstractions très différentes de la mémoire :

- la mémoire utilisée pour les objets d'un langage de programmation (tableaux, structures, objets, etc).
- la mémoire contenant les fichiers.

Pour des raisons historique, la sémantique des deux est différente.

Sémantique de la première mémoire

Accès direct :

```
a[i] = b[j]
```

Les données sont volatiles.

Sémantique de la deuxième mémoire

Accès plutôt séquentiel :

```
putchar(c)
write(fd, buffer, n)
```

Accès direct possible mais différent :

```
lseek(fd, offset, SEEK_SET)
```

De plus en plus, on utilise mmap.

Fichiers

C'est une unité de stockage permanent d'information.

Un fichier peut contenir du code ou des données.

Le *type* du fichier détermine les opérations possibles.

Le système d'exploitation peut reconnaître peu ou beaucoup de types de fichiers différents.

Fichiers

Le système Unix ne reconnaît essentiellement qu'un seul type de fichiers : une suite d'octets.

D'autres systèmes permettent des fichiers contenant de *enregistrements* avec une méthode d'accès indexée.

Ce sont les *applications* Unix qui implémentent d'autres types de fichiers.

Avantage des fichiers Unix

Relativement facile à implémenter.

Permet un traitement uniforme par un grand nombre d'outils.

Inconvénients des fichiers Unix

Un outil inadapté peut être appliqué à un fichier (impression d'un fichier binaire, par exemple)

Chaque application doit implémenter son propre format de fichier.

Opérations sur les fichiers

- Création (`creat`)
- Écriture (`write`)
- Lecture (`read`)
- Positionnement (`lseek`)
- Suppression (`unlink`)

et

- ouverture (`open`)
- fermeture (`close`)

Rôle de `open` et `close`

Évite la nécessité de chercher les répertoire à chaque accès.

L'ouverture crée un descripteur de fichier utilisé par les autres opérations.

Cohérence

Visibilité des modifications d'un fichier par d'autres utilisateurs ayant le même fichier ouvert.

Unix : Une modification est immédiatement visible.

Andrew File System : Les modifications sont visible après la fermeture du fichier.

Sémantique alternative

Une seule mémoire persistante (pas besoin de “sauver...”).

Peu de systèmes populaires s’en servent.

Mais voir le système Multics.

Répertoires

Un répertoire traduit un *nom* (d'un fichier ou d'un autre répertoire) en un objet (fichier ou répertoire).

C'est donc le type abstrait *dictionnaire* (la clé est une chaîne de caractères) avec plusieurs implémentations différentes (vecteur (trié ou non), table de hachage, arbre, etc).

Systèmes de fichiers

- arbre
- graphe sans cycles
- graphe général
- marqueurs

Arbre

- Facile à implémenter
- Facile à déterminer si un fichier peut-être supprimé
- Un fichier se trouve à *un seul* endroit

Graphe sans cycles

- Un fichier peut avoir plusieurs *noms* différents
- Difficile à détecter la création de cycles
- Unix permet le partage de fichiers uniquement (pas de répertoires)
- Pour cela, on utilise des *compteurs de références*

Unix permet également des *liens symboliques*.

Graphe général

- Nécessite un ramasse-miettes
- Peu de systèmes d'exploitation le font

Marqueurs

Voir : http://www.shirky.com/writings/ontology_overrated.html

Essence : Il est impossible d'établir des catégories arborescentes.

Exemple : Un article sur la programmation. Dans articles/programmation ou programmation/articles?

Solution: Associer une collection de marqueurs (tags) avec chaque objet (fichier).

(utilisation de CVS)

Différents systèmes de fichiers normalisés

Ext2, Ext3, Minix, ISO 9660, ECMA 119, BSD

Ext2

Afin d'améliorer la performance, on n'utilise pas l'unité *secteur*, mais l'unité *bloc* qui contient typiquement 4 ou 8 secteurs.

L'information d'un fichier est contenue dans une structure appelée *inode*.

Inode

Une centaine d'octets.

Contenu :

- diverses informations (mode, propriétaire, taille, dates, etc)
- 12 pointeurs sur des blocs de données
- un pointeur sur un répertoire d'un seul niveau
- un pointeur sur un répertoire de deux niveaux
- un pointeur sur un répertoire de trois niveaux

Ext2

Le disque (ou la partition) est divisé en *groupes de blocs*.

Contenu :

- Une copie du *superbloc*
- La liste de tous les descripteurs de groupes du système
- Un bitmap pour les blocs libres du groupe
- Un bitmap pour les *inodes* libres du groupe
- La table des inodes
- Les blocs

Répertoire Ext2

Un fichier contenant une suite d'entrées.

Chaque entrée contient :

- un numéro d'inode
- le nombre d'octets de l'entrée
- le nombre d'octets dans le nom du fichier
- le type du fichier (fichier normal, répertoire, ...)
- les caractères du nom

L'utilisation du champ "nombre d'octets de l'entrée" permet de supprimer une entrée plus rapidement.

Il y a également la possibilité d'utiliser une table de hachage.

Ext3

C'est un système de fichiers journalier basé sur ext2.

Le journal (log) permet de récupérer en cas d'un crash.

Protection

- permet le partage contrôlé de ressources entre utilisateurs
- protège le système contre des programmes défectueux
- protège le système et les données des utilisateurs contre des attaques intentionnelles

Objets protégés

- fichiers
- pages virtuelles (partage de code et de bibliothèques)
- périphériques
- inodes
- etc

Domaine de protection

Un concept abstrait qui détermine l'ensemble des droits d'un processus.

Ces droits peuvent changer pendant l'exécution du processus.

Pour Unix, un domaine de protection est un *utilisateur*.

Problèmes concrets

Un utilisateur ne doit pas pouvoir écrire directement sur disque, mais il faut le faire indirectement.

Comment permettre à un autre utilisateur d'accéder à des données, mais uniquement à travers un programme particulier (SGBD)?

Comment à la fois permettre et restreindre le droit de changer le domaine?

Liste d'accès

Chaque objet contient une liste de domaines avec les droits de chaque domaine.

Cette liste peut devenir assez longue. Plusieurs méthodes existent pour éviter ce problème.

Capacités

Chaque domaine contient une liste de tous les objets avec les droits sur chaque objet.

Une capacité est similaire à un pointeur protégé et contenant des indications de droit d'accès.

Le fait de posséder la bonne capacité donne le droit d'accès à l'objet.

Annulation de droits accordés

Facile dans le cas de la liste d'accès.

Plus difficile pour les capacités.

Protection Unix

Le domaine principal est l'utilisateur (effectif).

Le domaine peut changer grâce à la notion de *setuid*.

Les objets (fichiers) contiennent une version condensée de la liste d'accès (propriétaire, groupe, monde).

Une autre composante du domaine est le mode du processeur (utilisateur, privilégié).

Protection Unix

Lors de l'ouverture d'un fichier, le processus obtient une *capacité* sous la forme d'un descripteur de fichier (file descriptor).

C'est pour éviter la nécessité de vérifier les droits avant chaque accès.

Perspectives

Évolution grâce à certains progrès technologiques :

- espace d'adressage de 2^64 octets.
- MRAM
- multi processeurs
- mémoire transactionnelle

Espace d'adressage

Élimine le besoin d'espaces d'adressages séparés.

MRAM

Permet la persistance universelle.

Élimine le besoin de systèmes de fichiers journaliers.

La persistance peut faire en sorte que les capacité deviennent la méthode de protection préférée.

Multi processeurs

Nécessite le découpage de l'exécution d'un programme en plusieurs processus légers afin de profiter de la capacité des processeurs.

Le système doit faciliter l'utilisation de processus légers.

Mémoire transactionnelle

Facilite la synchronisation de processus.

Permet d'éviter le verrouillage dans un grand nombre de cas.

Organisation des données

Élimination du système de fichiers arborescent et de la notion de répertoire.

Systèmes basés sur l'utilisation de marqueurs possible.