Résumé SYE

Loïc Herman

Architecture de base

Séparation des instructions privilégies / sensibles du reste au moyen d'un kernel.
Ainsi, il y a 2 modes d'exécution pour le CPU:

- User mode
- Kernel mode (privileged mode)

Transition user → kernel uniquement via des interruptions logicielles ou matérielles.

Interruptions

Une interruption matérielle vient de l'IRQ (Interrupt Request) du CPU, c'est une interruption asynchrone.

Une interruption logicielle provient de l'exécution d'une instruction, c'est une interruption synchrone.
Quelques types d'interruptions logicielles:

- syscall
- undefined instruction
- division by zero
- data abort

Gestion des interruptions

1. Arrêt de l'exécution du programme en cours
2. Passage en mode kernel
3. Activation de l'ISR (Interrupt Service Routine)
4. Passage en mode user
5. Reprise de l'exécution du programme en cours

Architecture d'un OS

Il existe deux types: **monolith** et **microkernel**.

Architecture monolithique

Système performant car pas de changement de mode d'exécution, tous les sous-système du kernel sont placé dans un seul kernel space. C'est cependant moins sécurisé car un bug dans un sous-système peut planter tout le système.

Architecture microkernel

Système plus robuste car il y a un nombre minime de sous-système est placé dans le kernel space, le reste est placé dans le user space et la communication est faite via d'autres moyens.

Syscalls et images binaires

Permet d'appeler un service du kernel depuis le user space.

Gestion des appels

La fonction d'un appel syscall en C est en fait un *stub*, qui va en fait gérer d'inscrire les arguments nécessaire et faire appel à l'instruction système.

La procédure d'exécution est la suivante:

- Le stub va inscrire dans un registre (SO3: `R7`) le code correspondant au syscall, puis les arguments nécessaires dans les registres suivants (SO3: `R0` à `R3`).
- L'instruction système va ensuite être appelée, et va passer en mode kernel.
- Le kernel va ensuite récupérer le code du syscall et les arguments, puis va appeler la fonction correspondante.
- La fonction va ensuite retourner le résultat dans le registre `R0` et le kernel va retourner en mode user.

Gestion d'une image binaire

Un compilateur va génerer un *fichier objet*, un résultat intérmédiaire contenant les instructions machines ainsi que les données statiques et les espaces statiques réservés. Ce fichier objet n'est pas exécutable en tant que tel, car il manque des informations d'exécution système (librairies à lier dynamiquement, par exemple).

Un linker va ensuite lier les résultats intérmédiaires et les assembler dans un fichier exécutables qui contient les différents fichiers objets ainsi que les informations d'exécution système (librairies à lier dynamiquement, par exemple).

Librairies

Quand un programme fait appel à une librairie externe, le compilateur insère une adresse symbolique que le linker va remplacer par l'adresse non-symbolique de la fonction dans la librairie une fois celle-ci liée.

Processus

Un processus a deux contextes: le contexte d'exécution et le contexte mémoire.

Le contexte d'exécution contient les instructions et registres du CPU. Le contexte mémoire contient les données du processus, les variables, etc.

Format des processus

Les processus sont organisés de manière hierarchique. Au démarrage, après la phase d'initialisation, un premier processus (par exemple, init) est démarré. Les processus suivants seront donc des processus enfants du P_0.

Quand un processus se termine, le processus parent doit être informé afin de pouvoir libérer les ressources allouées au processus enfant.

Un processus parent qui se termine alors qu'il a encore des enfants causera les processus enfants à devenir orphelins, et seront adoptés par un autre processus parent choisi par le système (souvent P_0).

Zones mémoires d'un processus

Un processus a 4 zones mémoires:

1. Stack $T_0, T_1, ..., T_n$ (variables locales, arguments, adresse de retour. Local à un thread, chaque thread a sa propre stack qui s'étend vers le bas)
2. ...
3. Heap (zone d'allocation dynamique globale au contextes d'exécution, s'étend vers le haut)
4. BSS (variables **statiques non**-initialisées)
5. Data (variables **statiques** initialisées et constantes)
6. Code (instructions CPU)

BSS

Variables statiques non-initialisées.

```
static int i;
static char a[12];
```

Data

Variables statiques initialisées.

```
int i = 3;
char a[] = "Hello World";
static int b = 2023;

void foo(void) {
  static int c = 2023;
}
```

PCB

Le PCB (Process Control Block) est une structure de données qui contient les informations sur un processus. Il est stocké dans le kernel space. Il est surtout utilisé lors d'un changement de contexte. Chaque processus a son propre PCB, qui contient les informations suivantes:

- État du processus
- Espace d'adressage
- Pointeur vers le heap
- Références vers les processus enfants
- Descripteurs des fichiers ouverts
- Contextes d'exécution
  - État du contexte
  - Pointeur sur l'instruction en cours
  - Registres
  - Stack

Forking processes

Le syscall `fork` permet de créer un nouveau processus **enfant**. Toutes les données du processus parent sera copié dans le processus enfant. Le PCB sera donc copié et la MMU devra être reconfigurée pour les adresses du processus enfant. Le processus enfant va ensuite être ajouté à la liste des processus enfants du processus parent.

Attente

Via le syscall `waitpid()`, il est possible d'attendre qu'un processus enfant se termine en référençant son PID. La fonction permettra d'assigner le code de retour du processus enfant à une variable.

Il est obligatoire d'attendre l'exécution d'un processus enfant depuis le parent, afin de s'assurer qu'on ne laisse pas de processus orphelins (qui devront être gérés par le kernel).

Terminaison

Un appel au syscall `exit(int status)` permettra d'arrêter le processus et transmettra le code de retour au processus parent (via une variable du PCB).

Appel à un autre programme

Depuis un processus enfant, on peut faire appel à un syscall de la famille `exec` (en général `execv` ou `execve`). Ce syscall va permettre de charger un nouveau programme dans le processus enfant, et va remplacer le code du processus enfant par celui du nouveau programme.

Changement de contexte

Plusieurs événements peuvent causer un changement de contexte:

- Un processus se termine
- Un processus est mis en attente
- L'ordonnanceur le décide

Une opération de changement de contexte est couteuse, car elle nécessite de sauvegarder le contexte du processus courant dans le PCB et de charger le contexte du processus suivant. Les registres, le cache, et la MMU doivent aussi être reconfigurés, puis le PCB doit aussi être mis à jour.

Les opérations induites par un changement de contexte sont les suivantes:

1. Sauvegarde du contexte du processus courant
2. Chargement du contexte du processus suivant
3. Reconfiguration de la MMU
4. Mise à jour du PCB

États
Un processus peut avoir les états suivants:
- New
- Ready
- Running (peut revenir sur ready dans le cas où le processus est préempté)
- Waiting (uniquement depuis running, et peut revenir seulement sur ready)
- Zombie (état final transitoire avant que le processus parent s'occupe de récupérer le code de sortie et nettoyer définitivement le PCB)

Threading
La création de processus est une opération couteuse, car elle nécessite de copier le PCB et de reconfigurer la MMU. Les threads permettent de créer des processus légers, qui partagent le même espace d'adressage et le même PCB. Les threads sont donc plus légers à créer et à gérer.

Création
Un thread est créé via le syscall `pthread_create()`. Il prend en paramètre un pointeur vers une fonction qui sera exécutée par le thread, ainsi que des arguments pour cette fonction.
Un stack sera donc alloué par le kernel pour ce nouveau contexte d'éxecution.

Le thread aura donc accès au même contexte mémoire que le processus parent, et pourra donc accéder aux mêmes variables globales et au même heap.

Gestion dans le PCB
Le thread recevra un TCB (Thread Control Block) qui sera stocké dans le PCB du processus parent.
Le TCB contient les informations suivantes:

- Pointeur d'instruction
- Registres
- Stack pointer
- État du thread
- Priorité

Ce TCB sera utilisé lors des changements de contexte entre les threads du processus.

Attente d'un thread
Via le syscall `pthread_exit()`, il est possible de terminer un thread et de donner une valeur de retour, qui pourra être récupérée par le processus parent. Cet appel n'est pas obligatoire.

Via le syscall `pthread_join()`, il est possible d'attendre la terminaison d'un thread. La fonction permettra d'assigner la valeur de retour du thread à une variable (si la fonction `pthread_exit()` a été appelée).

File descriptors
Un file descriptor est un entier qui permet d'identifier un fichier ouvert par un processus. Il est unique pour chaque fichier ouvert par un processus.

Un tableau de file descriptors est stocké dans le PCB du processus, et est initialisé avec les 3 premiers file descriptors: `stdin`, `stdout` et `stderr`.
Ils font référence à une table globale de file descriptors ouverts, qui est gérée par le kernel.

Redirection
Avec le syscall `dup2()`, il est possible de rediriger un file descriptor vers un autre file descriptor.
L'ensemble des redirections sont conservées lors d'un appel à `fork()`.

Pipes
Les pipes permettent de gérer la communication d'un processus à un autre. Ils sont unidirectionnels et sont gérés par le kernel. La capacité d'un pipe est limitée (en général à 4 Ko).

Deux file descriptors sont créés par pipe: `pipe_fd[0]` pour la lecture (éqiv `stdin`), et `pipe_fd[1]` pour l'écriture (éqiv `stdout`).

Création
Il existe deux types de pipes: les pipes anonymes et les pipes nommés.

Pipes anonymes
Les pipes anonymes sont créés via le syscall `pipe()`. Ils sont créés dans le kernel et sont donc partagés entre les processus. Ils sont unidirectionnels et ne peuvent être utilisés que par des processus qui ont un lien de parenté.

Pipes nommés
Les pipes nommés sont créés via le syscall `mkfifo()`. Ils sont créés au moyen d'un fichier virtuel par le kernel dans le système de fichier en userland. Ils peuvent être utilisés par n'importe quel processus.

Signaux
Les signaux sont des interruptions logicielles asynchrones de type event. Ils sont envoyés par le kernel à un processus. Ils peuvent être envoyés par le kernel ou par un autre processus.

Un processus peut ignorer un signal, le masquer, le traiter, ou le laisser par défaut. Dans tous les cas, un handler sera appelé dans le userland.

Il ne peut y avoir qu'un seul signal du même type en attente, et ils seront pris en compte dès que le processus sera dans l'état running.

Envoi et réception
Un signal peut être envoyé via le syscall `kill()`. Il prend en paramètre le PID du processus à qui envoyer le signal, ainsi que le type de signal à envoyer.

Un signal peut être reçu via le syscall `signal()`. Il prend en paramètre un pointeur vers une fonction qui permettra de traiter le signal, ainsi que le type de signal qui doit être pris.

Ignorer un signal
On peut faire un appel à la méthode `signal()` avec le paramètre `SIG_IGN` pour ignorer un signal, ou `SIG_DFL` pour le remettre par défaut.

Sockets
Les sockets sont des objets de communication entre processus. Ils sont créés via le syscall `socket()`, qui prend en paramètre le type de socket à créer (TCP ou UDP).

Serveur
Un serveur TCP est créé via les appels suivants:
1. `ofd = socket()`
2. `bind()`
3. `listen()`
4. `nfd = accept()` (fork automatique)
5. `len = recv()` / `send()`
6. `close(nfd)` (retour à accept)
7. `close(ofd)`

Client

Un client TCP est créé via les appels suivants:

1. `fd = socket()`
2. `connect()`
3. `send()` / `len = recv()`
4. `close(fd)`

Ordonnancement

Il existe différentes mesures pour définir des algorithmes d'ordonnancement, les principales sont:

- Durée d'exécution (burst)
- Délai d'attente (temps passé à READY)

Points de préemption

L'ordonnanceur n'est activé que lors des points de préemption, dans lesquels il pourra décider si le processus en cours doit être préempté ou non. Il faut toutefois noter que seulement les processus à l'état *ready* peuvent être démarrés par l'ordonnanceur.

L'ordonnanceur est préemptif s'il va opérer des changements de contexte.

Types d'ordonnancement

FCFS (First come first served / FIFO)

Ordonnancement non-préemptif, facile à implémenter mais sous-optimal. Une simple file d'attente FIFO gère les processus entrants.

Sous-optimal car un long process peut monopoliser le système au détriment de processus plus courts.

SJF (Shortest job first)

Ordonnancement non-préemptif ou préemptif, qui va donner la priorité aux processus les plus courts d'abord. Difficile à implémenter mais proche de l'optimal.

Dans le cas du mode préemptif la durée considérée lors de l'arrivée d'un nouveau processus est la durée restante du burst. Le délai d'attente est alors optimal.

Risques de **famines** des processus, car les processus courts peuvent monopoliser le système et donc des processus plus long n'auront jamais de temps d'exécution.

RR (Round Robin)

Ordonnancement préemptif, qui va donner un temps d'exécution (*quantum*) à chaque processus. Basé sur FCFS, mais une fois le temps de quantum écoulé, le processus est mis en fin de file d'attente.

Ordonnancement par priorité

Préemptif ou non-préemptif, à priorité constante ou dynamique, une file FIFO est utilisée pour les priorités égales.

Dans le cas où la priorité est statique, il y a un risque de famine des processus à faible priorité. Un système de priorité dynamique avec une augmentation de la priorité plus le processus attent peut être utilisé pour éviter ce problème, le processus retrouvera sa priorité initiale lors de son exécution.

Ordonnancement par files d'attentes

Files d'attentes multiples basé sur l'ordonnancement avec priorité, chaque file d'attente a sa propre politique d'ordonnancement. Une file à basse priorité peut être FCFS, et une file à haute priorité peut être RR.

Chaque processus reçoit une priorité initiale, ne correspondant pas à la priorité du processus mais la priorité assignée à une file d'attente.

Des files d'attentes avec **rétroaction** peut être utilisée pour éviter des famines si la priorité reste constante. Le processus est alors déplacé dans une file d'attente de priorité supérieure après un certain temps d'attente.

Dispositifs de mémoire

L'espace d'adressage de la mémoire est **linéaire**, ainsi plusieurs intervalles d'adresses physiques consécutifs permettent d'accéder à différentes zones mémoires allouées par différents types de périphériques.

Allocation de la mémoire physique

L'adresse de départ d'un processus indique l'adresse physique de départ de la mémoire allouée au processus.

Afin de permettre une défragmentation de la mémoire RAM, l'adresse de base est stockée dans un registre à part et si le programme doit être déplacé alors seulement ce registre doit être mis à jour étant donné que le reste des adresses sont des offsets relatifs à l'adresse de base.

Algorithme d'allocation first-fit

Recherche de la première zone mémoire libre qui peut contenir le processus. On part alors d'un pointeur courant (qui commence sur la première adresse disponible) et une recherche circulaire est effectuée jusqu'à ce qu'une zone mémoire libre de taille suffisante soit trouvée.

Cet algorithme va favoriser un taux de fragmentation élevé, ce qui peut rapidement empêcher l'allocation d'un bloc contigu même si la somme totale des résidus de mémoire est amplement suffisante pour allouer la taille du bloc recherché.

Algorithme d'allocation best-fit

Recherche de la zone mémoire libre la plus petite qui peut contenir le processus. On part alors d'un pointeur courant (qui commence sur la première adresse disponible) et une recherche circulaire est effectuée sur l'entièreté de la mémoire pour trouver la zone générant le résidu le plus petit.

Cet algorithme va favoriser un taux de fragmentation faible, mais va nécessiter plus de temps pour trouver une zone mémoire libre.

Algorithme d'allocation quick-fit

Gestion des zones libres par liste dynamique à deux dimensions. Chaque entrée de liste de 1er niveau correspond à une liste de zones de taille identique. Il est toutefois nécessaire de synchroniser de temps en temps les zones adjacentes et de les fusionner, ce qui devrait être fait en arrière-plan.

La liste est difficile à maintenir de sorte à ce qu'elle garde une taille raisonnable.

Espace d'adressage virtuel

La gestion est opérée par la MMU, qui va traduire les adresses virtuelles en adresses physiques entre le CPU et la mémoire.

L'espace d'adressage représentera donc l'ensemble des adresses virtuelles accessibles pour un processus. Cet espace contiendra donc une partie noyau et une partie utilisateur, la partie noyau aura tout le code noyau et la partie utilisateur n'aura que le code utilisateur ainsi que les données associées au processus courant.

Typiquement, pour 32 bits (4 Go) d'espace d'adressage, 1 Go sera réservé au noyau et 3 Go seront réservés au processus. Dans un système 64 bits, un espace d'adresse canoniques est défini selon un certain nombre de bits de poids faible pour l'espace user, et le kerne est placé dans l'espace d'adresse de haut niveau.

Pagination

Pour éviter de devoir gérer des zones mémoires trop grandes, un système de pagination peut être mis en place et permet de gérer des régions de mémoires (dites **pages**) de taille fixe. La MMU va alors traduire les adresses virtuelles en adresses physiques en utilisant une table de pagination.

Une page est un ensemble d'octects contigus de taille fixe (en général 4 Ko). Une page virtuelle est mappée sur une page physique et elles sont numérotées.

Pour effectuer la pagination, l'adresse virtuelle est décomposée en deux parties: le numéro de page et l'offset à l'intérieur de celle-ci. La MMU pourra alors retrouver l'adresse physique en utilisant le numéro de page et l'offset et une table des pages.

La répartition des bits de l'adresse virtuelle dépend de la taille des pages, pour des pages de 4 Ko il faudra 12 bits pour encoder l'offset. Les 20 bits restants peuvent donc être utilisés pour encoder le numéro de page.

Pagination à un niveau

La table de pages est une table de PTE (page table entry) contenant le numéro de page physique correspondant pour une page virtuelle. Les PTE peuvent aussi contenir d'autres informations, comme les droits d'accès, le bit de présence, etc. Une PTE occupe en général 32 bits.

Pour effectuer la traduction, l'adresse virtuelle est décomposée en deux parties, n sera le numéro de pages et d l'offset. La PTE correspondant à l'index n est récupéree contenant la valeur p comme adresse de base de la page physique. La MMU concatène ensuite p et d pour former l'adresse physique résultante.

Pagination multi-niveaux

Pour éviter de devoir stocker un table de pages trop grande en mémoire, on peut la diviser en deux (voire plus) niveaux. Le premier niveau contient des PTE qui pointent vers des tables de pages de second niveau, qui contiennent les PTE qui pointent vers les pages physiques.

Caching de PTE (TLB - translation look-aside buffer)

Les TLBs sont des caches de PTE, qui permettent de réduire le temps d'accès à la mémoire. Ils sont stockés dans la mémoire cache de niveau 2 ou 3. Cela permet de réduire le temps d'accès à la mémoire, car la traduction peut être effectuée directement dans le TLB sans devoir accéder à la table des pages.

Gestion des pages physiques

Si la mémoire physique est paginée, il faut alors un système permettant de gérer la disponibilité de ces pages physiques. On peut alors faire appel à un bitmap, qui va représenter l'état de chaque page physique. Un bit à 1 indique que la page est libre, et un bit à 0 indique que la page est occupée. On ajoutera aussi une liste des pages libres qui peuvent contenir d'autres informations concernant la page en question.

Dans le cas où toutes les pages sont occupées, on va devoir remplacer les pages chargées en mémoire par d'autres pages. On va alors faire appel à un algorithme de remplacement de pages (swapping).

Extension de la mémoire

L'opération de swapping permet de déplacer une page physique de la mémoire vers un espace de stockage secondaire (disque dur, SSD, etc.). Cela permet de libérer de la mémoire physique pour d'autres pages.

Ainsi deux mécanismes sous-jacents à la pagination et la mémoire virtuelle sont nécessaires: la faute de page et le swapping.

Faute de page

Une page nécessaire n'est pas présente dans la RAM, donne lieu à une interruption logicielle.

En utilisant les bits d'attributs présents dans la PTE, on peut savoir si la page est présente en RAM ou non en utilisant un bit de validité. Parmis ces bits d'attributs nous trouverons:

- Bit de validité (indique si la page est présente en RAM)
- Bit de référence (indique si la page a déjà été référencée)
- Bit de modification (indique si la page a été modifiée)
- Bit de cache (indique si la page est en cache)
- Bit de protection (indique les droits d'accès à la page - RWX)

Lors de l'accès à une page qui n'est pas présente en RAM, une interruption logicielle est générée. Le kernel va alors charger la page en RAM et mettre à jour la PTE correspondante. Il va ensuite reprendre l'exécution du processus.

Swapping

Dans le cas d'un chargement de page avec une mémoire physique pleine, on peut alors déplacer une page de la RAM vers un disque et charger la nouvelle page à la place.

Il existe deux types de swap: remplacement global et remplacement local. Un remplacement local va uniquement remplacer des pages appartenant au processus fautif.

Il existe plusieurs algorithmes de remplacement de pages, les principaux sont présentés ci-après.

Algorithme FIFO
Un pointeur courant est maintenu pointant sur la page suivant celle qui vient d'être remplacée. Quand une page doit être remplacée, la page pointée par le pointeur courant est remplacée et le pointeur est incrémenté.

Algorithme LRU
Une version simplifiée consiste à mettre le bit de référence à 1 lors de l'accès à une page. Un timer est ensuite utilisé pour remettre le bit de référence à 0 après un certain temps. On sélectionne ensuite la première page avec le bit de référence à 0.

Algorithme de la seconde chance
Version améliorée de FIFO, en examinant le bit de référence de la PTE. Si une page a été référencée elle est préservée et son bit de référence est remis à 0. Si une page a un bit de référence à 0, elle est remplacée.

Algorithme WSClock
Algorithme fonctionnant exclusivement en remplacement local. Un working set est utilisé et contient l'ensemble des pages les plus récemment utilisées sur une fenêtre d'observation d'une certaine taille.

Avec TVC le temps virtual du processus et TDU le temps de la dernière utilisation d'une page, une page est dans l'ensemble de travail si $\text{TVC} - \text{TDU} \leq \Delta$.

L'algorithme est le suivant:

- Pointeur courant sur une page physique
- Tester le bit de référence
  - Si 1, remettre le bit de référence à 0 et incrémenter le pointeur (seconde chance)
  - Si 0 et que $\text{TVC} - \text{TDU} \leq \Delta$, incrémenter le pointeur
    Autrement, c'est la page à remplacer

Stockage secondaire

Dans un disque dur, un secteur est généralement constitués de 512 octets.

Il existe deux formes d'adressage:
- CHS: on référence directement un secteur par sont numéro de cylindre, de tête et de secteur
- LBA: chaque secteur est numéroté de façon linéraire et le disque s'occupe de traduire en CHS

Il existe deux formats d'organisation des secteurs:
- MBR: Master Boot Record, le secteur LBA 0 est recherché par le BIOS et contient la description des partitions
- GPT: GUID Partition Table, plus récent et plus flexible, permet de gérer plus de partitions

Structure d'une partition

1. Bootstrap
2. Superbloc
3. Gestion espace libre
4. Métadonnées
5. Répertoire racine
6. Répertoires et fichiers

Adressage

Chaque bloc FS est identifié par un numéro de bloc. Un bloc FS est une unité de stockage de taille fixe, en général 4 Ko. Pour retrouver une adresse LBA à partir d'un numéro de bloc, on utilise la formule suivante:
$\text{LBA} = \text{bloc} \cdot \left(\frac{\text{taille bloc}}{\text{taille secteur}}\right) + 1$. L'offset de 1 étant pour prendre en compte le secteur 0 utilisé par le MBR. On donne ainsi le premier numéro de secteur. Pour une taille de secteur de 512 octets et des blocs de 4Ko, il faudra ensuite donner l'intervalle complet des huit adresses.

FAT

Dans un système d'allocation par liste chaînées, chaque bloc contient un pointeur vers le bloc suivant, et il peut contenir des informations relatives aux données utilisateur et aussi des métadonnées système. Dans les métadonnées de la partition, on trouve alors la référence vers le premier bloc du fichier, qui contient un pointeur vers le bloc suivant, etc.

Cela permet une bonne gestion de l'espace disque, mais il est difficile de faire de l'accès direct. Aussi, une corruption d'un bloc conduit à une rupture de la chaîne et d'une perte partielle des données du fichier.

Une FAT permet donc de sortir les pointeurs de la chaîne des blocs de données en les stockant dans une table. Il y a donc deux paramètres: le nombre de bits pour encoder le numéro de bloc, et la taille des clusters (blocs).

Calculs de taille

Pour un disque de 1 To et 2^{28} bits d'adressage, il faudra au minimum des clusters de $\frac{2^{40}}{2^{28}} = 2^{12} = 4$ Ko pour encoder l'ensemble des adresses.

La taille minimum d'un fichier sera la taille minimale d'un cluster, donc 4 Ko.

Comme il y a 28 bits d'adressage et 2^{28} entrées dans la FAT, il faut $28 \cdot 2^{28}$ bits pour la stocker, soit $28 \cdot 256$ Mo $= 7$ Go. En termes de blocs dans notre système, cela prendrait $\frac{28 \cdot 2^{28} \text{ bits}}{2^{12} \text{ octets/blocs} \cdot 2^{3} \text{ bits/octet}} = 28 \cdot 2^{13}$ blocs.

I-nodes

Dans un système d'allocation indexée, une table d'index est utilisée pour stocker dans chaque entrée une adresse d'un bloc de données. Les blocs sont ordrés selon les index de la table et il n'y a pas de chaînes. La table d'index est contenue dans un bloc spécial appelé i-node.

L'i-node contient toutes les métadonnées, une adresse sur 32 bits et les adresses des blocs de données. Une i-node peut faire référence à un bloc direct, un bloc indirect, un bloc double indirect ou un bloc triple indirect. L'indirection de bloc permet d'augmenter l'espace d'adressage disponible et ainsi de gérer des fichiers de plus grande taille.

Le nombre d'indirections est déterminé lors du formatage de la partition, et une configuration mixte n'est en général pas possible.

En plus des métadonnées, une i-node référence donc 12 blocs directs, 1 bloc indirect, 1 bloc double indirect et 1 bloc triple indirect (une i-node ne contient que un bloc d'indirection de chaque type, et ils sont optionnels en fonction de la configuration lors du formatage).

Calculs de taille

Un bloc direct peut référencer uniquement 4 Ko, donc une i-node peut référencer 48 Ko de données directement.

Un bloc indirect simple ajoute 1024 adresses supplémentaires pour une adresse codée sur 32 bits et des blocs de 4 Ko, soit une extension de 4 Mo (en plus des 48 Ko).

Un bloc double indirect permet donc $1024 \cdot 1024$ adresses supplémentaires, sommant à $48 \text{ Ko} + 4 \text{ Mo} + 4 \text{ Go}$.

Si une i-node de 1 Ko continent 984 octets de métadonnées, alors avec un adressage direct il y a $\frac{1024 - 984 \text{ octets}}{4 \text{ octets/adresse}} = 10$ adresses et il est ainsi possible d'indexer des fichiers de 10 Ko maximum.

En adressage indirect simple, avec 9 blocs directs et 1 bloc indirect, on aura une taille maximale de $9 \cdot 1 \text{ Ko} + \frac{1024 \text{ octets}}{4 \text{ octets/adresse}} \cdot 1 \text{ Ko} = 9 + 2^8 \text{ adresses} \cdot 1 \text{ Ko} = 265 \text{ Ko}$.

En adressage indirect double, avec 8 blocs directs, 1 bloc indirect et 1 bloc double indirect, on aura une taille maximale de $8 \cdot 1 \text{ Ko} + 256 \text{ Ko} + 2^8 \cdot 2^8 \cdot 1 \text{ Ko} = 264 \text{ Ko} + 64 \text{ Mo}$.

En adressage indirect triple, avec 7 blocs directs, 1 bloc indirect, 1 bloc double indirect et 1 bloc triple indirect, on aura une taille maximale de $7 \cdot 1 \text{ Ko} + 256 \text{ Ko} + 2^{16} \text{ Ko} + 2^8 \cdot 2^8 \cdot 2^8 \text{ Ko} = 264 \text{ Ko} + 64 \text{ Mo} + 16 \text{ Go}$

Liens

Il existe deux types de liens: les liens durs et les liens symboliques.

Un lien symbolique est une référence logique définie par un fichier spécial, ainsi cela permet d'avoir des références circulaires détectables, des lien vers des répertoires et une gestion des liens morts.

Un lien dur est une référence physique; il y a donc deux entéres de répértoire et il est impossible de distinguer le fichier lié du fichier original (les métadonnées et les blocs sont identiques). Il n'est pas possible de hardlink un répertoire, et la notion de lien mort n'existe pas car elle est gérée en interne par le système de fichiers.

Liens et i-nodes

Un lien symbolique est un fichier spécial qui contient une adresse vers un autre fichier. Donc ce fichier dispose de sa propre i-node et de ses propres métadonnées, si le fichier cible disparaît alors le fichier de type lien pointera sur l'entrée inexistante. Le système de fichiers restent toutefois cohérent au point de vue des entrées de répertoire.

Pour un lien dur, l'i-node du fichier cible est référencé par une autre entrée de répertoire. Ainsi, les deux entrées de répertoire pointent vers la même i-node, une variable de compte est alors incrémentée dans l'i-node pour compter le nombre d'entrées faisant référence à celle-ci.

Dans linux, les répertoires spéciaux `.` et `..` sont des liens durs. Ainsi, chaque création d'un sous-répertoire aura pour effet d'incrémenter le compteur de liens de l'i-node du répertoire parent.