Programmation Système

Cours 5

Mémoire

Brice Goglin

Copyright

Plan

Partie 1

Gestion de la mémoire

Gestion de la mémoire

- Mémoire système et mémoire utilisateur
- Division de la mémoire utilisateur entre les différents programmes
- Maximiser le nombre de processus pour maximiser l'utilisation du matériel

Besoins

Relocalisation

- Pas de contraintes sur le programmeur
  - Localisation quelconque du programme en mémoire
- Ne pas dépendre des autres programmes résidants en mémoire
- Pouvoir être évincé sur le disque puis ramené en mémoire à un endroit différent

Relocalisation (2/2)

- Ajuster les références internes au programme
  - Données
  - Branchement
- Ajuster les références dans le système
  - Facile si le système gère lui-même la relocalisation

Protection

- Pas d'accès aux données des autres processus ou du système
  - Intentionnel ou par erreur
- Détection à l'avance difficile
  - Relocalisation imprévisible
  - Langages autorisent calcul dynamique d'adresses
- Détection dynamique couteuse logiciellement
  - Support matériel

Partage

- Protection configurable
  - Partage de zones mémoire possible
- Pas de duplication de code
- Partage explicite de données
- Support logiciel trop couteux
  - Support matériel

Organisation logique

- Mémoire linéaire
- Programmes modulaires
  - Protections différentes dans différents modules
  - Ecriture et compilation indépendantes des différents modules
    - Références ajustées dynamiquement
  - Partage de modules parmi différents programmes
- Segmentation

Organisation physique

- Structure hiérarchique de la mémoire
  - Mémoire principale assez rapide, mais limitée et volatile
  - Mémoire secondaire lente mais grande et persistante
- Ne pas contraindre le programmeur
  - Quels modules seront en mémoire principale ?
  - Si plein d'autres programmes ?
- Responsabilité du système d'exploitation

Partitionnement mémoire

Partionnement fixe

- Mémoire physique divisée statiquement en partitions fixes
- Programme chargée dans zone suffisamment grande
- Facile à implanter, peu couteux
- Gaspillage mémoire (*fragmentation interne*)
- Nombre de programmes limité

Partitionnement fixe (2/3) : Taille des partitions

- Tailles identiques
  - Taille du programme ou nombre de programmes très limité
  - Gros gaspillage pour les petits programmes
- Tailles croissantes
  - Moins de fragmentation interne
  - Quand même relativement limité

Partitionnement fixe (3/3) : Algorithme de placement

- Placement dans la partition la plus petite qui peut contenir le programme
- Utiliser une plus grande si aucune petite partition disponible
- Suspendre les processus si aucune partition disponible

Partitionnement dynamique

- Création de partition de taille optimale à la volée
- Fragmentation externe augmente
- Compactage pour réduire fragmentation
  - Gaspillage de temps processeur

Partitionnement dynamique (2/2) : Algorithme de placement

- Premier espace disponible (*first-fit*)
  - simple et rapide
  - un peu de fragmentation
- Meilleur espace disponible (*best-fit*)
  - couteux
  - beaucoup de petite fragmentation
- Nouveau placement quand programme revient du disque vers la mémoire principale

Buddy System

- Avantages partitionnements fixes et dynamiques
  - Pas trop couteux
  - Peu de fragmentation
- Listes de blocs libres de taille 1 à 2^n
- Allocation d'une partition dans le plus petit bloc adapté
- La partie gaspillée est convertie en petits blocs
- Fusion des petits blocs à la désallocation

Pagination

- Découpage en petits *cadres* physiques (*frames*) de taille fixe
  - Chaque *page* du processus va dans un *cadre*
- Pas de fragmentation externe
- Fragmentation interne faible
- Table des pages pour chaque processus
  - Associations page-cadre
- Adresses logiques transparentes page+offset

Segmentation

- Similaire à partitionnement dynamique
- Plusieurs segments par processus
  - Fragmentation externe moindre
- Pas transparent comme pagination
  - Utilisé pour séparer zones de types différents
    - Code, données, ...
- Table des segments, avec leurs longueurs

Chargement et édition de lien

Chargement des programmes

- Assemblage des modules et bibliothèques
- Calcul des différentes adresse mémoire
  - Chargement absolu
    - par le programmeur, à la compilation ou à l'assemblage
  - Chargement relocalisable
    - adresses relatives au début du module
  - Chargement dynamique
    - adresses relatives calculées au moment de l'exécution

Edition de liens

- Utilisation de symboles pour référencer les adresses
- Adresses relatives au début du module
- Références non résolues provoquent chargement de modules supplémentaires
  - mise à jour des modules sans changer les applications
  - partage de code facile

Détails de Linux

- Le compilateur précise l'interpréteur
- Le noyau lance l'interpréteur du programme (ld.so)
- Emplacement des librairies : ld.so.conf et ldconfig
- Emplacement des bibliothèques alternatives : LD_LIBRARY_PATH
- Bibliothèque de surcharge : LD_PRELOAD

Partie 2

Mémoire virtuelle

Intérêt et avantages

Intérêt

- Pagination et segmentation avantageux
  - Usage combiné dans les systèmes modernes
  - Ne pas les imposer au programmeur
- Mémoire virtuelle complexe
  - Relation entre support matériel et logiciel
  - Besoins logiciels très grands
    - Sécurité et protection
    - Flexibilité
    - Efficacité

Pagination et segmentation

- Les références mémoire d'un processus sont logiques
  - Traduire à la volée en adresse physiques
  - Support de la relocation après évinçage
- Les processus peuvent être découpés en parties non-contiguës en mémoire physique
  - pages ou segments

Résidence mémoire

- Un processus peut s'exécuter sans que toutes ses parties soient en mémoire
- Le système charge les parties nécessaires au fur et à mesure
  - Exception dans le processeur en cas de défaut de page
  - I/O pour charger les pages en mémoire
    - Processus en attente
- Localité mémoire pour le code et les données
  - Confirmé par l'expérience

Autres avantages

- Les processus utilisent moins de mémoire physique
- Plus de processus peuvent être chargés simultanément
  - Meilleure utilisation du processeur
- Un processus peut utiliser plus de mémoire virtuelle qu'il n'y a de mémoire physique
  - Pas de contraintes sur le programmeur

Support matériel

Table des pages

- Ensemble d'associations page + cadre physique
  - *Page Table Entry*
  - Ensemble de bits décrivant les propriétés
    - *Present*, *Modified*, ...
- Début de la table dans un registre spécial
- Table stockée en mémoire
  - virtuelle pour ne pas monopoliser trop de physique

Table des pages (2/2)

- Traduction par parcours de différents niveaux (entre 1 et 4)
- Exemple avec adresse virtuelle 32 bits
  - 10 bits pour premier niveau de la table
  - 10 bits pour deuxième niveau
  - 12 bits d'offset

Table des pages inversée

- Table des pages normale trop grande
- Une entrée par cadre physique
  - Identifiant du processus propriétaire
  - Page du processus
  - Bits de contrôle
  - Données de chaînage
    - Pour les pages partagées

Translation Lookaside Buffer

- Deux accès mémoire pour chaque accès réel
  - au moins un pour lire la table de pages
- Cache matériel pour l'éviter
- Mapping associatif pour recherche rapide
- TLB dans la *Memory Management Unit*
  - MMU dans le processeur
  - Placés entre caches de niveau 1 et 2

Taille des pages

- Si trop petit
  - Trop de pages, tables trop grandes
  - Moins de localité
    - Plus de défauts de page
  - TLB moins efficace
    - Taille matérielle limitée
- Si trop grand
  - Trop de fragmentation
  - Moins d'impact de la localité
    - Plus de défauts de page

Taille des pages (2/2)

- Taille idéale dépend
  - Mémoire physique disponible
  - Taille des programmes
  - Techniques modernes de programmation
    - Orienté objet, multithreadé, ...
- Tailles multiples
  - Grandes zones contiguës dans grandes pages
  - Supporté par beaucoup d'architectures
    - Peu de support dans les systèmes d'exploitation

Segmentation

- Partitions de taille variable et dynamique
- Table des segments
  - Adresse de base dans un registre
  - Recherche associative tenant compte de la longueur
  - Un seul niveau
- Bits similaires à ceux des tables de pages
- Segments visibles pour le programmeur

Combinaison

- Ensemble de segments découpés en pages
- Une table des segments et plusieurs tables des pages
- Adresse logique segment + page + offset
- Support par la plupart des architectures

Protection et partage

- Protection par les segments
  - Accès invalide si en dehors
- Partage explicite par partage de segments entre processus
- Partage de pages invisible pour le programmeur
- Utilisation d'anneaux privilégiés pour définir les autorisations d'accès à certaines zones

Support logiciel

Clés du design

- La gestion de la mémoire dans un système d'exploitation dépend de
  - l'utilisation de la mémoire virtuelle
    - Support matériel nécessaire
  - l'utilisation de la pagination et/ou de la segmentation
    - Support matériel nécessaire
  - les différents algorithmes utilisés
    - Prendre des décisions en faveur de l'efficacité du système
      - C'est très difficile

Politique de chargement

- Quand charger une page en mémoire ?
- à la demande (*Demand Paging*)
  - Beaucoup de défauts de pages au début
  - Localité réduit les défauts par la suite
- à l'avance (*Prepaging*)
  - Exploitation du matériel (disque)
    - Lecture de plusieurs blocs contigus plus efficace
  - Efficace au début du programme
  - Peut être nuisible par la suite

Politique de placement

- Important pour la segmentation
- Inutile si pagination
- Critique sur NUMA
  - Placer les pages à côté des processeurs qui en ont besoin
  - Problème si allocation et utilisation à des endroits différents
    - Allocation fainéante

Politique de remplacement

- Si la mémoire est pleine
  - Libérer des pages pour en charger des nouvelles
- Gestion des pages résidentes
  - Combien de pages de chaque processus peuvent résider en mémoire ?
  - Supprimer n'importe quelle page ? ou bien une du processus qui veut charger une page ?

Politique de remplacement (2/2)

- Quelle page supprimer ?
  - Celle qui a le moins de chance d'être utilisée dans le futur proche ?
  - Localité tend à ce que ce soit la page la moins récemment utilisée
  - Support matériel nécessaire
- Verrouillage
  - Pages importantes ne peuvent pas être évincées

Algorithmes pour le remplacement

- Optimal
  - Evincer la page qui sera utilisée le plus tardivement
  - Impossible car nécessite de connaître l'avenir
- *Last Recently Used*
  - Algorithme naturel d'après le principe de localité
  - Semble bon d'après expérience
  - Difficile à implanter dans le matériel
    - Marquer les pages

Algorithmes pour le remplacement (2/2)

- *First-In-First-Out*
  - Pages évincées en *Round-Robin* dans un tampon circulaire
  - Simple à implanter dans le matériel
  - Pas efficace
- Politique de l'horloge
  - FIFO utilisant des bits pour se souvenir du passé
  - Simple à implanter dans le matériel
  - Relativement efficace

Page Buffering

- Les pages non-utilisées sont placées dans une liste spéciale mais pas effacées
  - Liste des pages modifiées ou pas modifiées
- En cas de défaut de page, on regarde d'abord si la page est dans la liste
- Optimisations en lien avec le cache
  - Eviter les défauts de cache quand une page est évincée

Politique de nettoyage

- Quand écrire sur le disque une page modifiée ?
- à la demande (*Demand Cleaning*)
  - Quand elle est évincée
  - Défaut de page plus lent
- à l'avance (*Precleaning*)
  - Permet de regrouper les écritures (*Batch*)
  - Dommage si modifiée juste après
- Combinaison avec *Page Buffering*
  - Ecriture régulière et passage en état non-modifié

Degré de multiprogrammation

- Si peu de processus
  - Matériel pas bien utilisé
- Si trop de processus
  - Trop de défaut de page
- Adapter la fréquence des défauts à leur coût
- Suspendre des programmes
  - Ceux dont la probabilité de faute est trop grande

Exemples

Pagination dans SVR4 et Solaris

- Pagination pour les processus et les blocs disque
- Pages décrites par
  - Table de pages
  - Descripteurs de bloc disque
  - Table des cadre physique
  - Table d'utilisation du *swap*
- Remplacement par horloge

Kernel Memory Allocator de SVR4 et Solaris

- Allocation des petites zones
  - Moins d'une page
  - Toutes les structures internes au noyau
- *Lazy Buddy Algorithm*
  - Fusion des blocs pas immédiate
    - Dépassement d'un seuil
    - Quand vraiment nécessaire

Adressage dans Windows

- Espace utilisateur privé pour chaque processus
  - Espace paginé
  - Entre 0 et 2 Go
    - Zones spéciales autour pour détecter les débordement
- Espace système partagé
  - Entre 2 et 4 Go
    - Micronoyau
    - *Executive*
    - Pilotes de périphériques

Détails de Linux

Généralités sur la mémoire dabs Linux

- Surtout de la pagination
- Un peu de segmentation quand l'architecture le nécessite
  - Toute la mémoire pour le noyau
  - La mémoire utilisateur pour les processus
  - Quelques segments spéciaux

Espace d'adressage

- Espace utilisateur privé
  - Entre `0` et `PAGE_OFFSET` (3 Go sur IA32)
  - Divisé en différentes zones
  - *Swappable*
- Espace noyau partagé
  - Au dessus de `PAGE_OFFSET`
  - Accessible uniquement en mode réel (privilégié)
  - Pas *swappable*

Structures mémoire

- Espace d'adressage (`struct mm_struct`)
  - Zones utilisateur (`struct vm_area_struct`)
    - sauf pour les threads noyaux
  - Table de pages (`pgd_t`, `pmd_t`, `pte_t`)
- Cadre physique (`struct page`)

Espace utilisateur

- Espace divisé en différentes zones
  - struct vm_area_struct
  - Caractérisée par localisation, protection, verrouillage
  - Méthodes décrivant comment les manipuler
    - Par exemple en cas de défaut de page
- Détaillé dans /proc/<pid>/maps
- Code en lecture seule
- Données en lecture et écriture

Espace utilisateur (2/2)

- Code placé en bas
- Tas juste au dessus
  - Données globales
  - malloc() utilise sbrk() pour l'agrandir
    - Modification de la VMA du tas dans do_brk()
- Pile tout en haut
  - S'agrandit automatiquement
- Grand espace libre entre les deux

Mapping en espace utilisateur

- VMA correspond au mapping d'un fichier
  - Mapping *anonyme* pour les données normales
    - Pile et tas
  - Mapping du programme
    - En lecture seule pour le code
    - En lecture et écriture pour les données
  - Mapping des bibliothèques
    - En lecture seule pour le code
    - En lecture et écriture pour les données

Modifications de l'espace utilisateur

- Mapping explicite d'un fichier par l'utilisateur
  - Appels système mmap(), mremap(), munmap()
- Routines noyau pour manipuler les VMA
  - do_mmap(), do_mremap(), do_munmap()
  - Fichiers mappés sous la pile
    - Avec les bibliothèques
- Utilisé par malloc() pour les grandes allocations
- Permet de partager de la mémoire
  - Les mapping peuvent être privés ou partagés

Accès aux données utilisateurs

- Espace d'adressage réellement divisé entre parties utilisateur et noyau
  - Cas particuliers 4Go/4Go
- Pas accès à la mémoire noyau en mode utilisateur
- Accès à la mémoire utilisateur en mode noyau
  - Adresses non déréférençables
  - `get/put_user(addr, length)`
  - `copy_from/to_user(to, from, length)`
  - Mapping des pages cibles

Cadres physiques dans Linux

- Grand tableau linéaire de `struct page` décrivant tous les cadres physiques
  - Indexé par *Page Frame Number*
  - Stockés après `mem_map`
- Pas de pointeurs vers la page virtuelle dans la plupart des cas
  - Car le cadre peut être partagé
- Bits de protections
- Pointés par la table de pages

Tables de pages

- 3 (ou 4) niveaux selon les noyaux
  - Niveaux réels dépendent de l'architecture
- *Page Global Directory*
  - `pgd = pgd_offset(mm, address)`
- *Page Middle Directory*
  - `pmd = pmd_offset(pgd, address)`
- *Page Table Entry*
  - `pte = pte_offset(pmd, address)`
- Cadre physique
  - `page = pte_page(pte), pfn = pte_pfn(pte)`

Tables de pages (2/2)

- `pgd = pgd_offset(mm, address)`
- `pmd = pmd_offset(pgd, address)`
- `pte = pte_offset_map(pmd, address)`
  - `pte_unmap(pte)`
- `page = pte_page(pte)`
- Vérification des différentes étapes
  - Existence : `pgd/pmd/pte_none()`
  - Présence en mémoire : `pgd/pmd/pte_present()`
  - Validité : `pgd/pmd_bad()`

Espace mémoire du noyau

- Mapping linéaire de la mémoire physique
  - virt_to_phys() et phys_to_virt() valides
    - Translation de PAGE_OFFSET
  - Physiquement contigu
  - Limité à 896 Mo sur IA32
    - Toute la mémoire physique n'y est pas
- Espace *vmalloc*
- Mappings fixes
  - Mémoire des périphériques et *High Memory*

High Memory

- Cadres physiques non mappés dans l'espace linéaire du noyau
- N'importe quelle cadre physique peut être mappé temporairement
  - addr = kmap(page) et kunmap(page)
  - addr = kmap_atomic(page, type) et kunmap_atomic(addr, type)
    - Visible sur un seul processeur
    - Ne doit pas durer longtemps
  - Nombre limité

Mapping de pages utilisateur dans le noyau

- Manipulation intense de pages utilisateur dans le noyau
- get_user_pages() donne les cadres physiques (struct page)
- Mapping dans le noyau par kmap
- Possibilité de déréférencer comme n'importe quelle structure noyau
- Demapper (kunmap) puis relâcher les pages
  - page_cache_release(page)

Mapping linéaire de la mémoire

- Code tout en bas
- Ensemble de pages disponibles
  - Peuvent être allouées dans le noyau
    - alloc_pages et free_pages
  - Peuvent être utilisées pour les espaces utilisateur
- Ensemble de *Slab Caches* utilisant pages du mapping linéaire
  - Ensemble de zones de tailles fixes préallouées
  - *Buddy Algorithm*

Allocation mémoire dans le noyau

- Allocation d'une structure de type précis
  - kmem_cache_alloc(cache, flags) et kmem_cache_free(cache, ptr)
- Création d'un cache
  - kmem_cache_create et kmem_cache_destroy
- Allocation d'une structure de taille précise
  - kmalloc(taille, flags) et kfree(ptr)
  - Implanté par un cache de taille 2^n supérieure
    - de 32 octets à 128ko

Allocation mémoire dans le noyau (2/2)

- Différents *flags* pour décrire ce que le *Slab Allocator* peut tenter pour allouer la mémoire
  - Exemples de flags internes
    - __GFP_WAIT si on peut bloquer
    - __GFP_IO si on peut faire des entrées/sorties
    - __GFS_HIGHMEM si on peut prendre des pages *HighMem*
  - Exemples de flags manipulés par le programmeur
    - GFP_KERNEL pour les allocations normales
    - GFP_ATOMIC dans un contexte risqué
    - GFS_NOFS dans un contexte d'accès aux fichiers

Zones de mémoires non-contigues

- *Slab Allocator* limité aux zones assez petites
- Zones physiquement contiguës pas toujours utiles
- ptr = vmalloc(size) et vfree(ptr)
  - Alloue des pages quelconques
    - alloc_pages et __GFP_HIGHMEM
  - Les mappe contigument dans l'espace *Vmalloc*
    - Dans l'espace du noyau
- Utilisé pour charger les modules

Le *Swap*

- Pages transférées sur une partition de swap
  - Zones anonymes des processus
  - Mapping privés dans les processus
- Partition divisée en slots de la taille d'une page
- Mapping partagés et cache de fichiers écrits sur leur partition d'origine
- Libération des pages non récemment utilisées
  - Listes de pages actives et inactives

Le *Swap* (2/2)

- Quand swapper ?
  - Quand une allocation est impossible
  - `kswapd` vérifie régulièrement qu'il y a suffisamment de pages libres
- `try_to_free_pages()`
- `shrink_caches()`
  - Réduire les *Slab Caches*

Laziness

- Ne rien faire avant que ce ne soit nécessaire
  - Allouer réellement les pages
    - Attente d'un défaut de page
  - Libérer les pages inutiles
    - Attente qu'il n'y ait plus de mémoire libre
    - Linux utilise souvent toute la mémoire disponible
  - Dupliquer les pages partagées
    - Attente de la première modification concurrente
- Moins d'utilisation processeur et mémoire

Exceptions et défaut de page

- `handle_mm_fault(mm, vma, addr, wr)`
  - `handle_pte_fault()`
- Page pas encore réellement allouée
  - `do_no_page()`
- Page d'un fichier pas encore réellement mappée
  - `do_file_page()`
- Page swappée
  - `do_swap_page()`
- Page partagée mais privée
  - `do_wp_page()`

Copy-on-Write

- Pages dupliquées au dernier moment
  - A la première modification
- Permet un partage de code facile
  - Bibliothèques système
  - Après `fork()`
- Utilisé pour allouer des pages initialisées à 0