Gestion de la Mémoire
2ème partie: Mémoire pour le kernel

Linux et pagination

- Le kernel fait un usage important du système de pagination
  - Plus répandu que les segments sur les processeurs
- Le code du kernel et de ses structures sont dans des pages physiques (cadres) réservées
  - Les pages de ces cadres ne peuvent jamais être swapées
- Habituellement, le kernel est en RAM à partir de l'adresse 0x00100000
- Nombre total de pages: dépend de la configuration
- Pourquoi pas à l'adresse 0?
  - A cause de l'architecture PC

Linux et pagination

- Le cadre 0 est utilisé par le BIOS pour stocker la configuration détectée
- Les adresses 0x000a0000 à 0x000fffff sont réservées aux routines du bios et pour mapper la mémoire interne des cartes ISA
- Les autres pages sont réservées pour divers systèmes
- Pour éviter les problèmes, le kernel n'utilise donc pas la zone < 1MB
- Taille des cadres:
  - 4K ou 4M possibles
  - Utilisation de 4K
    - Souvent multiple de taille des blocs sur le disque => pratique pour le swap

Kernel et pages

- Le kernel doit gérer l'état de toutes les pages
  - Différence entre les cadres pour le noyau et pour les processus
  - Occupation d'un cadre
- Utilisation d'un tableau
  - Chaque entrée (descripteur de page) correspond à un cadre
  - Utilisation d'une *struct page*

```
typedef struct page {
        struct list_head list;
        struct address_space *mapping;
        unsigned long index;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;
        struct list_head lru;
        struct page **pprev_hash;
        struct buffer_head * buffers;
        void *virtual;
} mem_map_t;
```

Structure page

- *count* : Compteur d'utilisation
  - mis à 0 si la page est libre
  - tout autre valeur si utilisée par un processus ou le kernel
- *list* : liste chaînée des cadres
- flags : jusqu'à 32 flags décrivant l'état du cadre
  - *PG_error* : erreur de lecture de ce cadre
  - *PG_locked* : le cadre ne peut être swappée
  - *PG_slab* : le cadre est dans un slab

Allocation de cadres

- Les cadres sont allouées grâce à 4 fonctions et macros
- *__get_free_pages(gfp_mask, order)*
  - Fonction pour demander 2^{order} cadres contigus
- *__get_dma_pages(gfp_mask, order)*
  - Macro pour obtenir des cadres utilisables pour du DMA
- *__get_free_page(gfp_mask)*
  - Macro pour obtenir une unique page
- *get_free_page(gfp_mask)*
  - Fonction qui invoque *__get_free_page* et remplis le cadre obtenu de 0
- Paramètres
  - *__GFP_WAIT* : le kernel peut vider des cadres existants pour satisfaire la demande. Le demandeur est donc mis en attente
  - *__GFP_IO* : Une I/O physique est autorisée pous libérer des cadres et satisfaire cette demande.
  - *__GFP_DMA* : Le cadre est utilisable pour du DMA
  - *__GFP_HIGH, __GFP_MED, __GFP_LOW* : niveau de priorité, le *low* est en général pour les processus en *user mode*

Libération de cadres

- Il existe 3 macros/fonctions pour libérer les cadres
- *free_pages(addr, order)*
  - Vérifie le descripteur de page correspondant au à l'adresse physique *addr*
  - Si le cadre n'est pas réservé, le *count* est décrémenté
  - Si *count* atteint 0, alors le kernel suppose que les 2^{order} cadres ne sont plus utilisés
  - Le descripteur est alors placé dans la liste des cadres libres
- *__free_page(p)*
  - Similaire à la précédente, mais utilise un pointeur sur descripteur de page
- *free_page(addr)*
  - Libère l'unique page à adresse

Allocations contigues

- Le kernel alloue des cadres contigus
- Risque de fragmentation externe
- 2 façons d'éviter ça
  - Utiliser le circuit de pagination du cpu pour maper des adresses linéaires contigues vers des cadres non contigus
  - Maintenir une liste de blocs de cadres contigus libres et éviter de casser de gros blocs pour de petites allocations
- Linux utilise la 2ème solution
  - Recourir au circuit de pagination n'est pas toujours possible (DMA)
  - Meilleure gestion des *translations lookaside buffers* si cadres consécutifs

The Buddy System Algorithm

- Tous les cadres libres sont mis dans des groupes de tailles différentes
  - 1, 2, 4, 8, 16, 32, 64, 128, 256 et 512 cadres contigus
- Ces groupes sont maintenus dans des listes (une par taille de groupe)
- L'adresse physique du premier cadre du bloc est un multiple de la taille du groupe
  - Ex: l'adresse initiale d'un bloc de 16 cadres est un multiple de 16×2^{12}
- Algorithme
  - Une demande est faite pour *n* cadres consécutifs
  - Si *n* a une liste correspondante, on l'utilise
    - Sinon, utilisation de la liste supérieure la plus proche
  - Allocation des cadres nécessaire
    - Les cadres restants sont mis dans la liste correspondante

The Buddy System Algorithm

- Exemple: Une demande de 128 cadres arrive
- On regarde dans la liste des 128, si non vide, c'est bon
- Sinon, utilisation de la liste des 256
  - Si non vide, allocation des 128
  - Ajout d'un nouveau bloc de 128 à la liste 128.
- Sinon utilisation de la liste des 512
  - Si non vide, allocation des 128
  - Ajout d'un bloc de 256 et d'un bloc de 128 (384)
- Sinon, pas assez de cadres libres
  - Arrêt de l'algorithme
  - Retour d'erreur

Libération de cadres

- Le nom de l'algorithme vient en fait de la façon dont la libération des cadres est effectuée
- L'algorithme essaie de fusionner des pairs de blocs « amis » (*buddy*) libres
  - 2 blocs de taille b deviennent un bloc de taille $2b$
- 2 blocs sont « amis » si
  - Ils sont de même taille
  - Ils sont à des adresses physiques contigues
  - L'adresse physique du premier cadre du premier bloc est multiple de $2b \times 2^{12}$
- Algorithme itératif
  - Si la fusion a réussi avec des blocs de taille b, on continue avec $2b$

Gestion des zones mémoires

- Une zone mémoire est un nombre arbitraire de cellules mémoires ayant des adresses physiques contigues
  - Pas forcément un nombre exacte de cadres
- Le *buddy* permet de gérer les demandes importantes
- Mais comment gérer des demandes de quelques octets?
- Ajout d'une structure pour gérer l'allocation dans un cadre
  - Mais si demande de *n* octets, alors nécessite *k* octets dans les meta-structure pour gérer allocation
  - Fragmentation interne
- Solution non satisfaisante

Slab Allocator

- Inventé en 1994 pour Solaris 2.4
- Basé sur plusieurs constatations
- Le type de données stockées peut affecter la façon dont l'allocation est effectuée (*get_free_page* remplit le cadre de 0)
  - Les zones mémoire peuvent être vues comme des objets, avec des constructeurs et des destructeurs
  - Quand une zone est libérer, l'objet la représentant est gardé pour être réutilisé
- Le kernel a tendance a demander des zones mémoire du même type fréquemment
  - Création d'un process nécessite des tables…
- Les demandes peuvent être classées suivant leur fréquence
  - Les requêtes fréquente de taille donnée peuvent être gérées en créant des objets de la bonne taille

Slab Allocator

- Le *slab allocator* groupe tous les objets dans des caches
- Chaque cache contient des collections d'objets de même type
  - Libres ou utilisés
  - Chaque collection est un *slab*
- La liste des caches est visible dans */proc/slabinfo*
- Le *slab allocator* utilise le *buddy allocator* pour obtenir des cadres
- Et il ne rend pas les cadres spontanément
  - Demande explicite du kernel
  - Seulement si le *buddy* ne peut satisfaire une demande et si le *slab* est vide

Utilisation du *slab allocator*

- Création d'un cache

```
kmem_cache_t * kmem_cache_create(
                    name,size,offset,flags,
                    constr,destr)
```

- Paramètres
  - name : nom des objets (pour debug)
  - size : taille individuelle de chaque objet
  - offset : décallage du premier objet par rapport au premier cadre (pour optimisation)

Utilisation du *slab allocator*

- Paramètres
  - flags : contrôle de l'allocation
    - SLAB_NO_REAP : empêche le système de réduire le cache si besoin de mémoire
    - SLAB_HW_ALIGN : aligne chacun des objets en mémoire en augmentant si nécessaire leur taille
    - SLASH_CACHE_DMA : les objets sont stockés dans une zone mémoire compatible DMA
  - void (*constructor) (void *, kmem_cache_t *, unsigned long flags)
    - Pointeur vers constructeur
- Constructeur/Destructeur
  - Aucun, ou Constructeur, ou Constructeur/Destructeur
  - Le constructeur est appelé lors de l'allocation d'un nouvel objet (mais aucune garantie temporelle)
  - Destructeur appelé après libération d'un objet
  - Les flags sont passés par le slab allocator pour indiquer des propriétés (SLAB_CTOR_ATOMIC, SLAB_CTOR_CONSTRUCTOR…)

Utilisation du *slab allocator*

- Un cache ne contient par défaut aucun objet
- Création d'un objet

  void *kmem_cache_alloc(kmem_cache_t *cache, int flags)
  - Pointeur vers le cache
  - Flags qui seront utilisés si allocation mémoire nécessaire
- Libération d'un objet

  void *kmem_cache_free(kmem_cache_t *cache, const void *obj)
- Suppression du cache

  void *kmem_cache_destroy(kmem_cache_t *cache)
  - Ne réussit que si aucun objet n'est utilisé
  - Vérifier le code de retour!

Allocation non contigue

- Il est préférable d'allouer la mémoire de façon contigue
  - Meilleure utilisation du cache
- Mais on peut aussi avoir un espace linéaire contigu, et des cadres non contigus
  - Limite la fragmentation externe
- Possibilité d'allouer de la mémoire dans l'espace *virtuel*
- La taille d'un espace non contigu doit être multiple de 4K
- Les informations d'allocation sont maintenues dans une liste simple de *struct vm_struct* contenant
  - L'adresse linéaire de la première cellule mémoire
  - La taille de la zone (+4096)
  - Pointeur vers suivant
- *vmalloc et vfree…*
- Peu utilisés en pratique