Taller de Sistemas Operativos

Administración de memoria (kernel)

Agenda

- Introducción
- Páginas
- Obtener y liberar memoria
- Zonas de memoria
- Slab allocator
- Buddy system
- Zone allocator
- Pools de memoria

Introducción

- La obtención de memoria en el kernel es más problemática que en el espacio de usuario
  - Muchas veces no se puede bloquear
  - No es fácil manejar los errores
- El kernel usa las páginas como la unidad básica de administración de memoria
- Veremos como hace el kernel para administrar la memoria intentando minimizar la fragmentación y optimizando el uso del cache

Páginas

- En intel de 32 bits Linux usa solamente páginas de 4KB
- Cada página en el sistema está representada por una estructura `struct page` que indica:
  - El offset de la página
  - Cuantas veces está referenciada
  - La dirección en memoria virtual de la página
  - Si está sucia, si fue accedida recientemente, si está disponible para swap, etc
- Solamente refiere a páginas físicas (no virtuales)
- Estas estructuras usan aproximadamente el 1% de la memoria del sistema

Páginas

- Campos del `struct page`:

```
page_flags_t flags
atomic_t  _count
atomic_t  _mapcount
struct address_space *mapping
pgoff_t index
struct list_head lru
void * virtual
```

- Macros
  - `virt_to_page`: dirección de page struct a partir de dirección virtual
  - `pfn_to page`: dirección de page struct a partir de número de marco

Zonas de memoria

- La memoria se divide en tres zonas por limitaciones del hardware
  - ZONE_DMA Páginas donde se puede hacer DMA
  - ZONE_NORMAL Páginas normales
  - ZONE_HIGHMEM Páginas mapeadas dinámicamente

- En intel (32 bits):
  - ZONE_DMA < 16 MB
  - ZONE_NORMAL 16 – 896 MB
  - ZONE_HIGHMEM > 896 MB

Zona HIGHMEM

- El espacio de memoria virtual del kernel es de 1GB (el último de los 4)
- El kernel reserva 128MB para sus estructuras por lo que dispone de 896MB para mapear el resto de la memoria
- Si se obtiene una página en memoria alta no se puede obtener su dirección virtual, si su `page struct`
- Parte de los últimos 128MB se usan para mapear temporalmente páginas en memoria alta
- En arquitecturas de 64 bits ZONE_HIGHMEM está vacío

Marcos de página reservados

- El kernel reserva una serie de marcos de página para ser usados en caso de “emergencia”
- Se acude a ellos cuando hay poca memoria y se hace una solicitud de memoria en un código que no puede bloquearse
- Se reserva un mínimo de 2^7 KB y un máximo de 2^{16} KB
- Se calcula como:
  - `ceil(sqrt(16 * memoria directamente mapeada))`
- Se puede ver y cambiar en /proc/sys/vm/min_free_kbytes

Obtener y liberar memoria

- El único mecanismo de obtención de memoria de bajo nivel que ofrece el kernel funciona a nivel de páginas
  - `struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)`
- Obtiene 2^{order} páginas contiguas y retorna el page struct de la primera o NULL en caso de error
- Se puede usar la función `page_address(struct page *page)` para obtener la dirección lógica
- La función `get_zeroed_page(unsigned int gfp_mask)` retorna la dirección lógica de una nueva página rellena de ceros

Obtener y liberar memoria

- Para liberar la memoria se usa:
  - `void __free_pages (struct page * page, unsigned int order)`
  - `void free_pages (unigned long addr, unsigned int order)`
- **Cuidado**: No hay protección de memoria
- Cuando se obtiene una página en HIGHMEM se debe mapear en la memoria del kernel con `kmap(struct page *page)`
- Y se elimina el mapeo con `kunmap(struct page *page)`

Obtener y liberar memoria

- Para obtener memoria con granularidad de bytes se usa:
  - `void * kmalloc (size_t size, int flags)`
- Funciona igual que malloc a nivel de usuario
- Retorna un puntero a una región de memoria contigua con al menos size bytes o NULL en caso de error
- Para liberarla se utiliza:
  - `void kfree (const void *ptr)`
- Existen además vmalloc y vfree
  - Devuelven memoria virtualmente contigua aunque no necesariamente contigua físicamente

Flags (gfp_mask)

- Se dividen en tres tipos
  - Action: deciden como se obtiene la memoria
  - Zone: de dónde se obtiene la memoria
  - Type: son una combinación de los otros dos para obtener memoria para un uso específico
- Ejemplos de Action:
  - __GFP_RECLAIM: se puede bloquear para liberar memoria
  - __GFP_HIGH: se puede acceder a la memoria de emergencia
  - __GFP_IO: se puede usar I/O a disco
  - __GFP_COLD: no se pueden usar páginas en cache
  - __GFP_NOFAIL: no puede fallar

Flags (gfp_mask)

- Ejemplos de Zone:
  - __GFP_DMA: Obtiene la memoria de la zona DMA
  - __GFP_HIGHMEM: Obtiene la memoria de la zona highmem (con preferencia) o normal
- Ejemplos de Type:
  - GFP_ATOMIC: Alta prioridad, no se puede bloquear, puede acceder a reservas de memoria
  - GFP_NOIO: No se puede acceder a disco, puede liberar páginas limpias o cache
  - GFP_KERNEL: Modo normal, puede bloquearse, le da todas las opciones posibles al kernel
  - GFP_DMA: Memoria para DMA

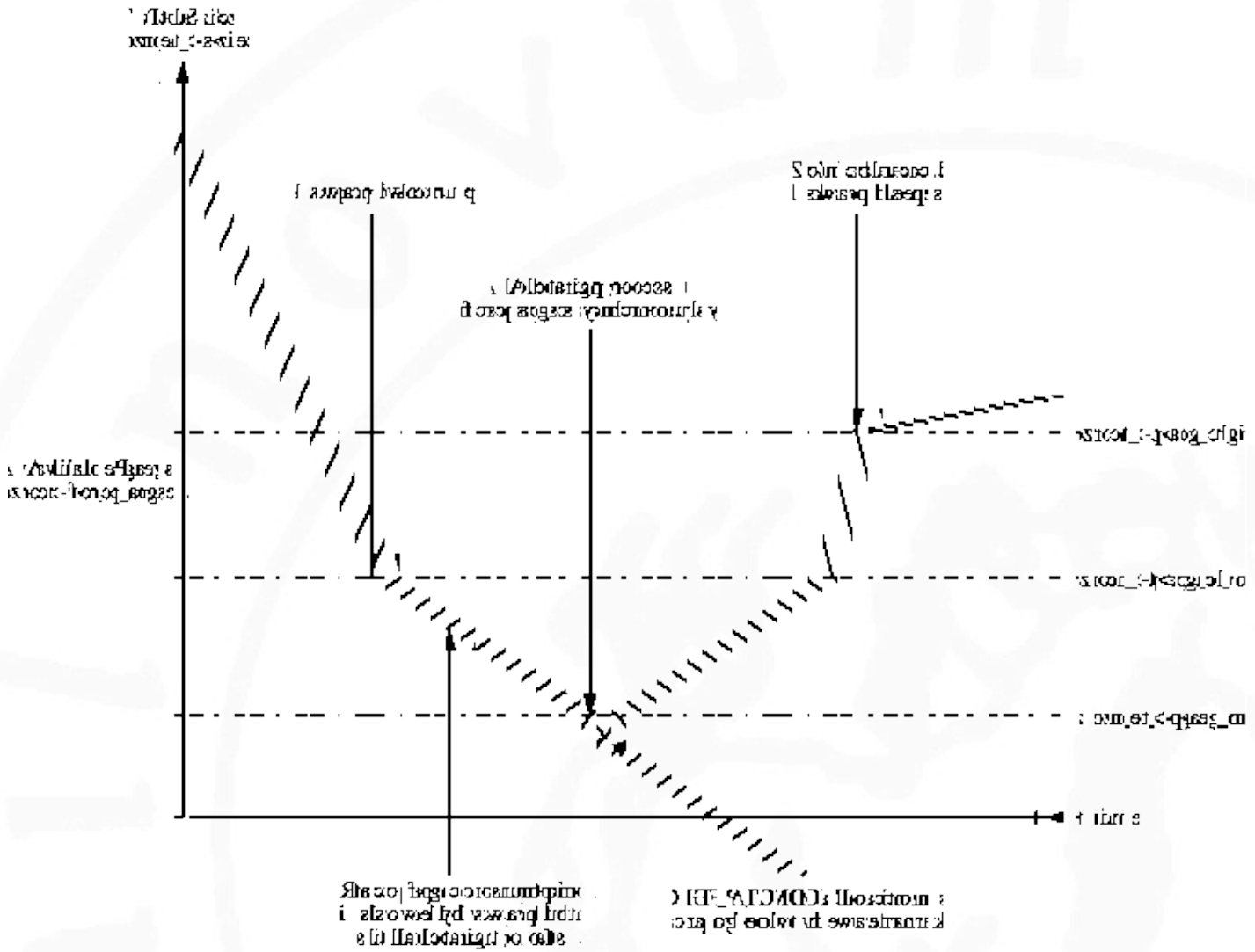
Flags (gfp_mask)

- Flags de cada Type:
  - GFP_ATOMIC: (__GFP_HIGH | __GFP_ATOMIC | __GFP_KSWAPD_RECLAIM)
  - GFP_NOIO: __GFP_RECLAIM
  - GFP_KERNEL: (__GFP_RECLAIM | __GFP_IO | __GFP_FS)
  - GFP_DMA: __GFP_DMA
- Se pueden mezclar con OR

Watermarks

- Cada zona tiene tres marcas de agua llamadas
  - pages_low,
  - pages_min
  - pages_high
- Permiten medir la presión de memoria de la zona
- La cantidad de páginas es pages_min se calcula al inicio del sistema y depende del tamaño de la zona.
- Se calcula inicialmente como tamaño_paginas / 128.
- El valor está entre 20 páginas (80K) y 255 páginas (1MiB).

Watermarks



Uso del stack

- Todas estas formas de pedir memoria manejan solamente memoria dinámica
- En lugar de esto se podría usar el stack
- El problema es que el stack del kernel tiene solamente dos páginas (8K en intel)
- Además no hay control de stack overflow
  - Recordar que al final del stack del kernel está el thread_info
- Conclusión:
  - Solamente se pueden guardar datos pequeños en el stack
  - No se puede usar recursión

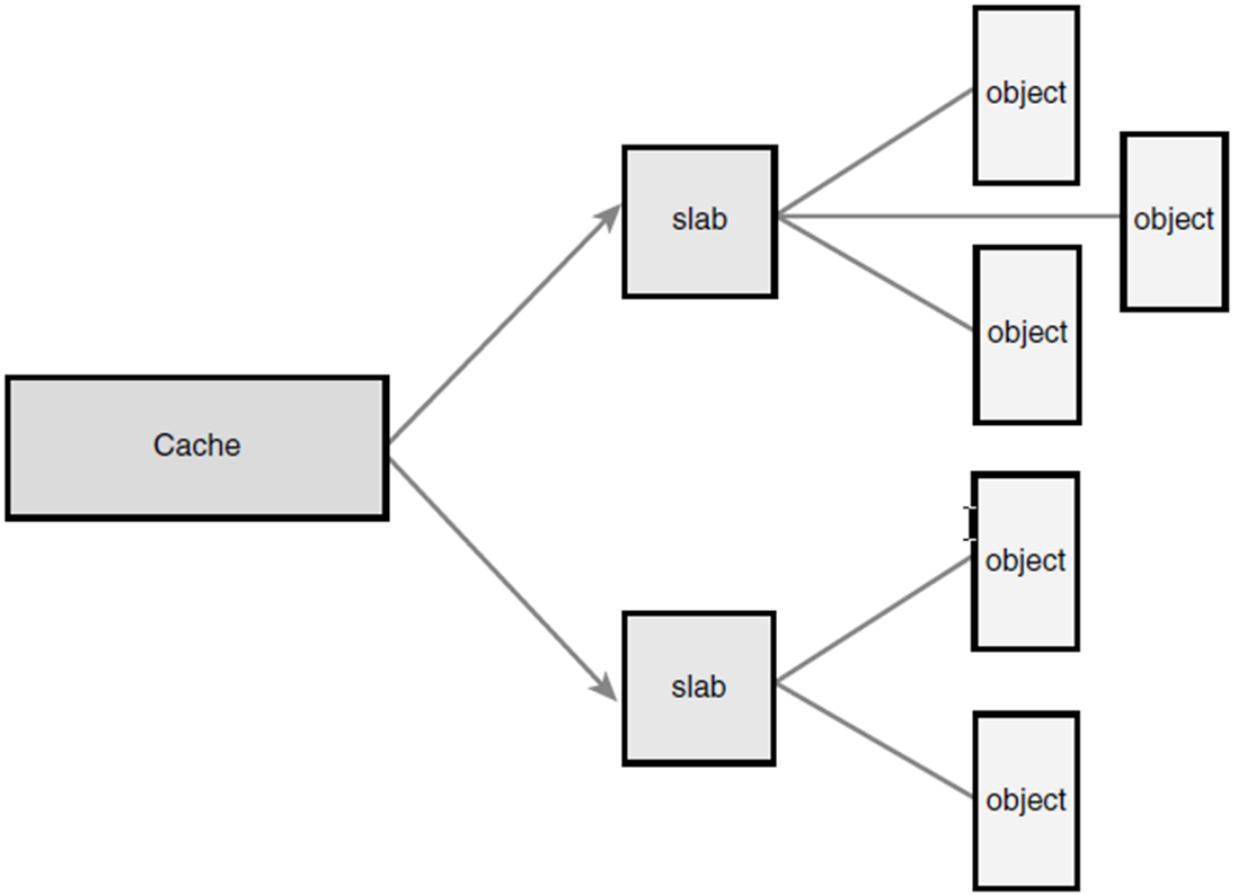
Slab Allocator

- Es una abstracción que proporciona el kernel para manejar la creación y destrucción de estructuras en memoria
- Funciona como un cache de datos de forma que se sacan estructuras creadas del cache y se devuelven al mismo cuando ya no se necesitan
- Al dar una interface general el kernel tiene control sobre el tamaño del caché y puede reducirlo cuando hay poca memoria
- Además permite reducir la fragmentación

Slab Allocator

- El slab allocator divide la memoria en caches, uno para cada tipo de objeto
- Cada cache tiene además un constructor (puede ser null)
- A su vez los caches se dividen en slabs que son un grupo de páginas (generalmente una)
- Cada slab contiene varias instancias del objeto y puede estar en uno de tres estados: full, partial, empty
- Cuando se requiere un objeto nuevo se saca de:
  - un slab parcial o
  - si no hay ninguno parcial de uno vacío o
  - si no hay ninguno vacío se crea uno nuevo

Slab Allocator



Fuente: Linux Kernel Development, 3ª edición

Slab Allocator

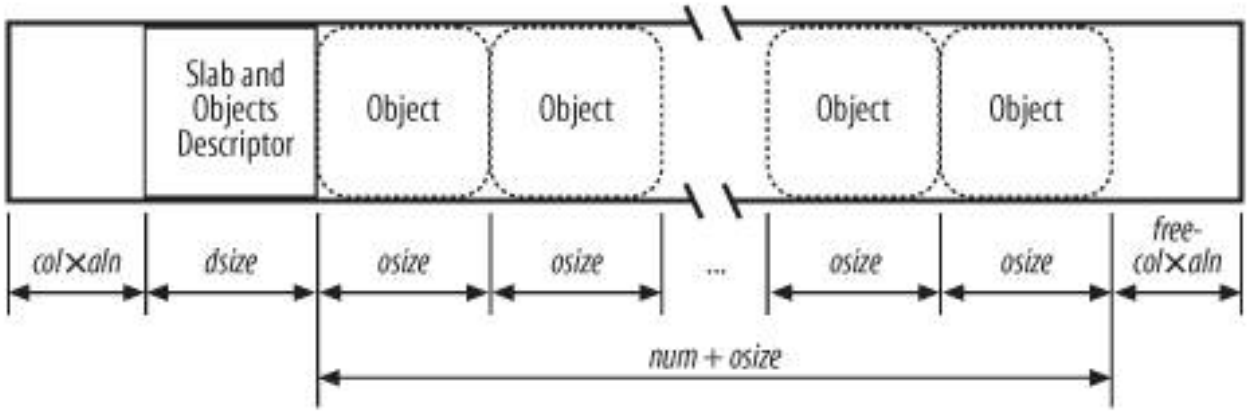
- Cada caché está representado por un `struct kmem_cache_s` que tiene las tres listas de *slabs*
- Cada slab se representa por un `struct slab` que puede estar dentro o fuera del propio *slab*
- Las páginas para los *slabs* se obtienen usando *alloc_pages*
- Las *flags* del pedido de memoria se pasan específicamente al solicitar un objeto del cache
- Los *slabs* se liberan si hay muchos vacíos o con una operación que ejecuta periódicamente

Slab Coloring

- En general los objetos en cada *slab* están alineados con las palabras de memoria (comienzan en una posición múltiplo de 4 en máquinas de 32 bits)
- Esto desperdicia espacio a cambio de mayor eficiencia en el acceso
- Sin embargo resulta en que dos objetos con el mismo offset en dos *slabs* probablemente compartan la misma línea de cache
- Esto se trata de evitar asignándole un color a cada *slab*

Slab Coloring

- La idea es repartir el espacio que sobra en el *slab* para desfasar la posición de los datos adentro del mismo
- Si hay varios colores disponibles cada *slab* nuevo se crea con uno distinto en forma circular



Fuente: Understanding the linux kernel, 3ra edición

NUMA

- Linux divide la memoria en una serie de nodos donde el tiempo de acceso de cierta CPU a cada dirección de cada nodo es el mismo
- Se trata de poner las estructuras más usadas por cada CPU en la memoria más cercana a ella
- Cada nodo está a su vez dividido en zonas
- En intel de 32 bits hay un solo nodo con toda la memoria

Buddy System

- Implementa la estrategia que utiliza el kernel para reservar grupos de páginas contiguas
- Intenta prevenir la fragmentación externa devolviendo páginas físicamente contiguas en lo posible
- Los marcos de página libres se agrupan en 11 grupos de 1, 2, 4, 8, ... hasta 1024 marcos libres (de 4 KB a 4 MB)
- Cada grupo de marcos de cierto tamaño tiene un “compañero” contiguo de igual tamaño

Buddy System

- Cuando se solicita cierta cantidad de memoria (potencia de 2) se busca en el grupo correspondiente y si hay se devuelve directamente
- En otro caso se busca en un grupo más grande y si se encuentra uno libre se parte en trozos: uno se devuelve y el otro va a la lista correspondiente (más chica)
- Al retornar un bloque de memoria, si su compañero está libre se une con el para formar un bloque más grande
- En caso de éxito se sigue agrupando recursivamente
- Hay un buddy system distinto por cada zona

Cache por CPU

- El cache tiene varios marcos de páginas reservados
- Se utiliza para satisfacer pedidos de una sola página (es una operación muy común)
- En realidad hay dos caches por CPU:
  - Hot: su contenido está en una de las líneas del cache del CPU
  - Cold: su contenido no está mapeado en el caché del CPU
- Las páginas de este cache se obtienen y se retornan al buddy system

Zone Allocator

- Es el *front end* de todo el mecanismo de asignación de marcos del kernel
- Sus objetivos son:
  - Conservar los marcos de página reservados
  - Llamar al proceso que realiza swap a disco cuando hace falta memoria y el proceso que llama se puede bloquear
  - Conservar la zona DMA

Zone Allocator

- El proceso de ejecutar `alloc_pages` es:
  - Se busca memoria libre en las tres zonas
  - Si no se encuentra se invoca *kswapd* y se vuelve a probar
  - Si sigue sin encontrar puede recurrir a la reserva o dar error
- El *slab allocator* lo usa para obtener nuevos marcos para sus caches

Pools de memoria

- Permiten reservar una serie de marcos para ser usados en caso de emergencia
- Solo puede ser usada por su dueño (el que la creó)
- No confundir con la reserva general del kernel
- Se representan con una estructura `mempool_t` y se tienen las siguientes operaciones:
  - `mempool_create()`
  - `mempool_destroy()`
  - `mempool_alloc()` → busca primero fuera del pool
  - `mempool_free()` → si el pool está lleno libera la memoria

Pools de memoria

- Los campos de `mempool_t` son:
  - `spinlock_t lock`
  - `int min_nr` : cantidad mínima de elementos
  - `int curr_nr` : cantidad actual de elementos
  - `void ** elements`
  - `void * pool_data` :datos privados del dueño
  - `mempool_alloc_t * alloc` : función para pedir memoria
  - `mempool_free_t * free` : función para devolver la memoria
  - `wait_queue_head_t wait` : cola de espera en caso de que no haya lugar en el pool

Esquema de administración de memoria