Taller de Sistemas Operativos

Administración de memoria (usuarios)

Agenda

- Introducción
- Espacio de memoria de los procesos
- Regiones de memoria
- Manejo del heap
- Page frame reclaiming algorithm (PFRA)
- Out of memory killer
- Swap

Introducción

- Vimos las formas en que el kernel accede a la memoria (`alloc_pages`, `kmalloc`, `vmalloc`) y como estas funciones retornan directamente referencias a memoria
- Esto funciona porque:
  - El kernel es el componente de mayor prioridad del S.O. y se trata de darle memoria lo más pronto posible
  - No se necesita ninguna protección en el acceso a memoria (el kernel confía en sí mismo)
- Sin embargo con los procesos de usuario se tiene que:
  - Los pedidos de memoria no se consideran urgentes
  - Se necesita capturar todos los errores en el acceso a memoria

Espacio de memoria de los procesos

- El espacio de memoria de un proceso consiste en un una serie de intervalos de memoria física que el proceso tiene permitido usar
- El kernel los representa mediante *regiones de memoria* que están definidas por:
  - Dirección de inicio
  - Largo
  - Permisos
- Tanto el inicio como el largo son múltiplos de 4K (están alineados con las páginas)

Regiones de memoria

- Cada región de memoria contiene una sección diferente del código y los datos del programa:
  - El código del programa (*text section*)
  - Las variables globales inicializadas (*data section*)
  - Las variables globales sin inicializar (se mapea por defecto una página con ceros)
  - El *stack* (también se mapea con ceros inicialmente)
  - Las bibliotecas dinámicas cargadas
  - Los archivos mapeados a memoria
  - Los segmentos de memoria compartida

Regiones de memoria

- Un proceso adquiere nuevas regiones de memoria cuando:
  - Se crea el proceso
  - Cuando se invoca `exec*` (se conserva el PID pero cambia la memoria)
  - Cuando se mapea un archivo a memoria
  - Cuando se agranda el *stack* de usuario
  - Cuando se agranda el área de memoria dinámica (*heap*) con `malloc()`o similar
  - Cuando se crea una región de memoria compartida vía IPC

Regiones de memoria

- La memoria de los procesos es descripta por la estructura `mm_struct`, que es apuntada por el campo `mm` del PCB
- Algunos de sus campos son:

`struct vm_area_struct * map`	Lista de regiones
`struct rb_root mm_rb`	Árbol de regiones
`atomic_t mm_users`	Cantidad de usuarios (*threads*)
`int map_count`	Cantidad de regiones
`unsigned long start_brk`	Inicio del heap
`unsigned long brk`	Fin del heap
`unsigned long start_stack`	Inicio del stack
`unsigned long stack_vm`	Cantidad de páginas del *stack*

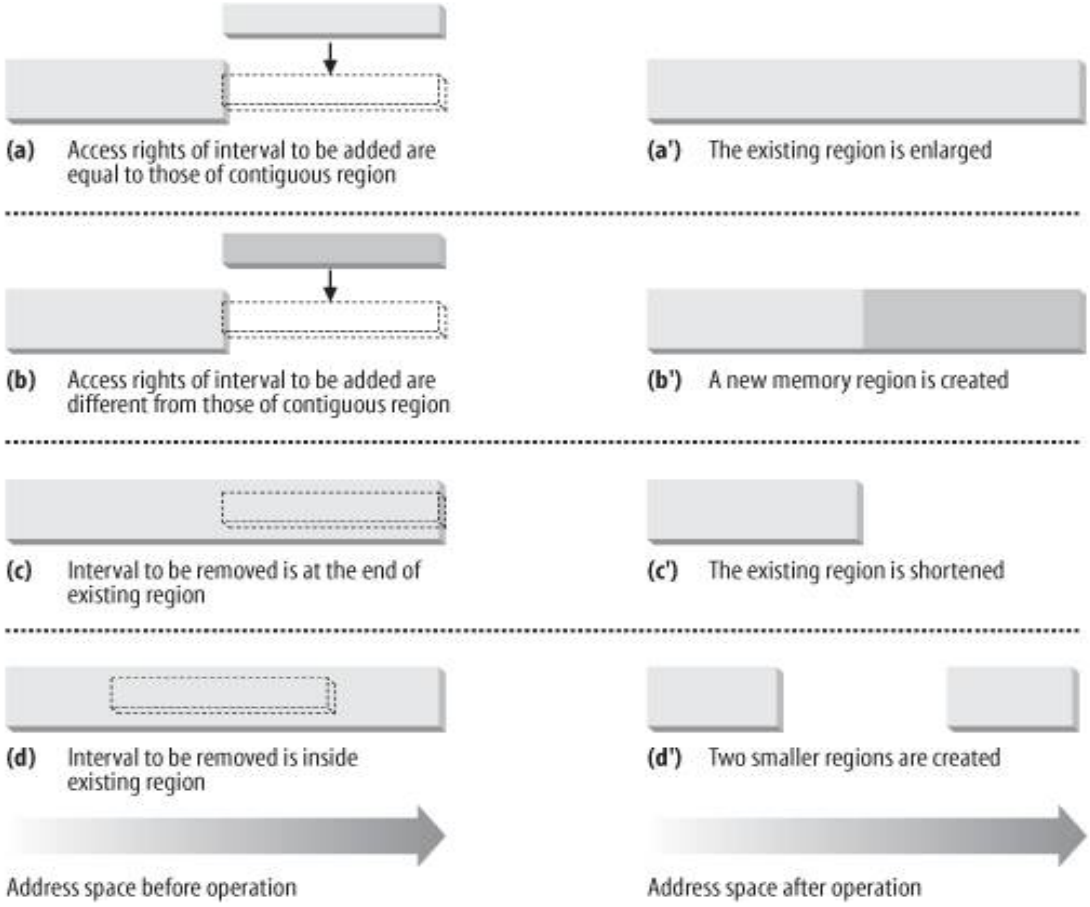
Regiones de memoria

- Cada región se representa por la estructura `vm_area_struct`
- Algunos de sus campos son:

`unsigned long vm_start`	Dirección de inicio
`unsigned long vm_end`	Dirección de fin
`pgprot_t vm_page_prot`	Permisos de la página
`struct file * vm_file`	Archivo mapeado (si corresponde)
`struct vm_area_struct vm_next *`	Próxima región del proceso

- El tamaño de la región está dado por `vm_start` – `vm_end`
- Las regiones nunca se solapan y el kernel trata de unirlas si se tienen dos regiones contiguas con los mismos permisos

Regiones de memoria



Fuente: Understanding the linux kernel, 3ra edición

Regiones de memoria

- Las regiones de memoria de un proceso están en una lista encadenada en orden ascendente por dirección de memoria
- Un proceso no puede tener mas de 65530 regiones
  - Esto se puede ver y cambiar desde `/proc/sys/vm/max_map_count`
- Si hay muchas regiones buscar en la lista no es eficiente, por eso se mantiene además un *red-black tree* de punteros a las regiones
- Al insertar o borrar de la lista se buscan los punteros anterior y siguiente en el árbol y se actualiza sin recorrer la lista

Permisos de las regiones de memoria

- Linux permite definir permisos de lectura, escritura y ejecución sobre las páginas
- Además se puede especificar si la página va a ser compartida
- Sin embargo el control de los accesos a memoria lo realiza el *hardware* por lo que se deben mapear los permisos definidos a los que proporciona la arquitectura
- En Intel cada entrada de la tabla de páginas solamente tiene dos bits para:
  - lectura / escritura
  - acceso modo usuario / supervisor

Permisos de las regiones de memoria

- Las 16 opciones de lectura, escritura, ejecución y compartición se mapean así:
  - Si la página tiene acceso de escritura o compartición se prende el bit de lectura/escritura
  - Si la página tiene acceso de lectura o ejecución pero no tiene acceso de escritura o compartición se apaga el bit de lectura/escritura
  - Si la CPU tiene el bit NX y no tiene permiso de ejecución este bit se desactiva
  - Si no tiene ningún permiso se marca la página como no presente
  - El bit de modo usuario está siempre prendido
- Además es posible que una página esté protegida contra escritura cuando no debe ser compartida (recordar *copy on write*)

Manejo del heap

- Cada proceso tiene un *heap* desde donde el cual se satisfacen los pedidos de memoria dinámica (es otra región)
- El mismo está delimitado por `start_brk` y `brk`
- Las funciones que agrandan el heap está implementadas basándose en los *system calls* `brk()` y `mmap()`
  - `brk()` aumenta el tamaño del *heap*
  - `mmap()` mapea un espacio de disco en memoria
- Ambas funciones dan memoria con granularidad de páginas

Manejo del heap

- `brk()`verifica si la nueva dirección del tope del *heap* está en la misma página que la anterior:

```
newbrk = (addr + 0xfff) & 0xfffff000;
oldbrk = (mm->brk + 0xfff) & 0xfffff000;
if (oldbrk == newbrk) {
        mm->brk = addr;
        ...
}
```

- En caso afirmativo no pide memoria nueva
- En realidad se usa la macro PAGE_ALIGN que está implementada en assembler

PFRA

- PFRA = *Page Frage Reclamation Algorithm*
- Linux en general no realiza controles muy estrictos en cuanto a la cantidad de memoria asignada a usuarios o caches
- Esto permite aprovechar al máximo la memoria RAM y además es más eficiente
  - Con poca carga casi toda la memoria es usada en caches y buffers
- Solo cuando la memoria empieza a ser poca Linux decide recuperar parte de la misma

PFRA

- El PFRA decide que páginas memoria puede recuperar y con ella rellena las listas del *buddy system*
- Se saca memoria de los caches y de los procesos de usuario
- Las páginas en uso se dividen en cuatro clases:
  - *Unreclamaible*: no se pueden recuperar
  - *Swappable*: se pueden copiar al swap
  - *Syncable*: se sincronizan con el disco si es necesario
  - *Discardable*: se pueden recuperar de inmediato
- El algoritmo debe lograr un balance entre servidores que piden mucha memoria y máquinas de usuario que piden menos memoria pero la necesitan rápido

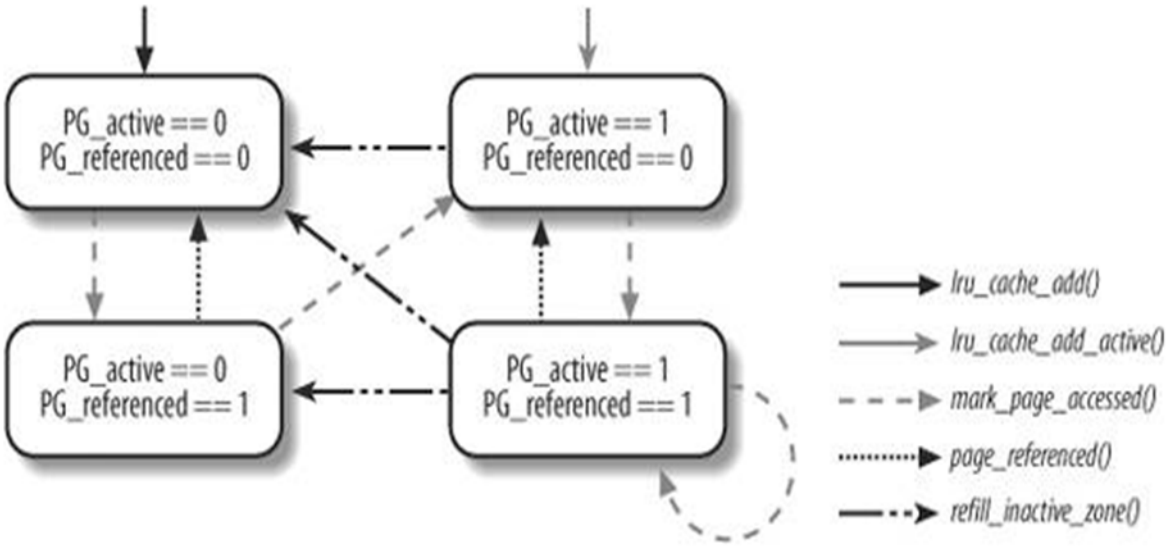
PFRA

- El diseño es completamente empírico
- Las ideas básicas del algoritmo son:
  - Liberar primero las páginas descartables
  - Todas las páginas de los procesos pueden ser recuperadas
  - Se pueden recuperar páginas compartidas pero hay que des mapearlas de todas las tablas
  - Se recuperan las páginas menos usadas primero (LRU)
  - No siempre es factible implementar esto realmente, depende del *hardware*

PFRA

- El algoritmo se invoca en tres casos:
  - Se detecta falta de memoria (ej. error en *alloc_pages*)
  - Se debe guardar toda la memoria para hibernar la máquina
  - Invocación periódica (*kswapd* y *cache reaper*)
- Todas las páginas de los procesos se agrupan en dos listas: la activa y la inactiva
- Hay un par de estas listas (*LRU list*) por cada zona de la memoria
- El PFRA solo elige páginas de la lista de inactivas
- Lo fundamental es como se mueven páginas de una lista a la otra

PFRA



- *PG_active* es un *flag* que indica si la página está en la lista de páginas activas
- *PG_referenced* se usa para que se necesiten dos accesos para mover una página a la lista de activas
- Aquí se ve como se mueven las páginas entre los cuatro estados posibles

PFRA

- `mark_page_accessed()` se invoca cada vez que el kernel determina que una página está siendo accedida por un proceso de usuario, por ejemplo:
  - Al cargar una página del *heap*
  - Al cargar una página de un archivo mapeado a memoria
  - Cuando se carga una página de memoria compartida IPC
  - Cuando se trae una página del *swap*
- `page_referenced()` retorna 1 si el bit de referencia está prendido en la estructura o en la tabla de páginas
  - Desactiva el bit `PG_referenced` de la página y además resetea el bit en todas las tablas de páginas que la referencian

PFRA

- `refill_inactive_zone()` es la función que mueve páginas a la lista de inactivas
- Funciona en forma incremental: primero prueba pocas páginas y si tiene problemas para encontrar memoria busca en áreas más grandes de la memoria
- Swap tendency: determina si la función mueve cualquier página o solo las que están en caches (si es < 100)
  - *Swap tendency* = mapped ratio / 2 + distress + swappiness
  - *Mapped ratio* es el porcentaje de las páginas usadas que pertenecen a procesos de usuario
  - *Distress* depende de cuantas páginas pudo recuperar la última vez (entre 0 y 100)
  - *Swappiness* en general vale 60 (/proc/sys/vm/swappiness)

PFRA

- La liberación efectiva de la memoria la hace la función `try_to_free_pages()`
- Intenta liberar al menos 32 páginas llamando a `shrink_caches()` y a `shrink_slab()` repetidamente
- Si no lo logra llama a `out_of_memory()`
- `shrink_caches()` llama a `shrink_zone()` para cada zona de la memoria la cual a su vez termina llamando a `refill_inactive_zones()` si no hay marcos libres
- Estas funciones crean una lista de páginas candidatas las cuales son efectivamente liberadas por `shrink_list()`
- Aquí se llama a `page_referenced()` en cada página de la lista

Out of memory (OOM) killer

- Se activa cuando el PFRA no puede recuperar memoria a pesar de estar en máximo *distress*
- Evita situaciones en que no se pueda ejecutar nada por falta de memoria
- Selecciona un proceso y lo mata
  - Debe ser un proceso con mucha memoria
  - No debe haber estado corriendo por mucho tiempo
  - Debe tener baja prioridad estática
  - No debe tener permisos de root
  - No debe acceder directamente al hardware (ej. Xserver)

Swapping

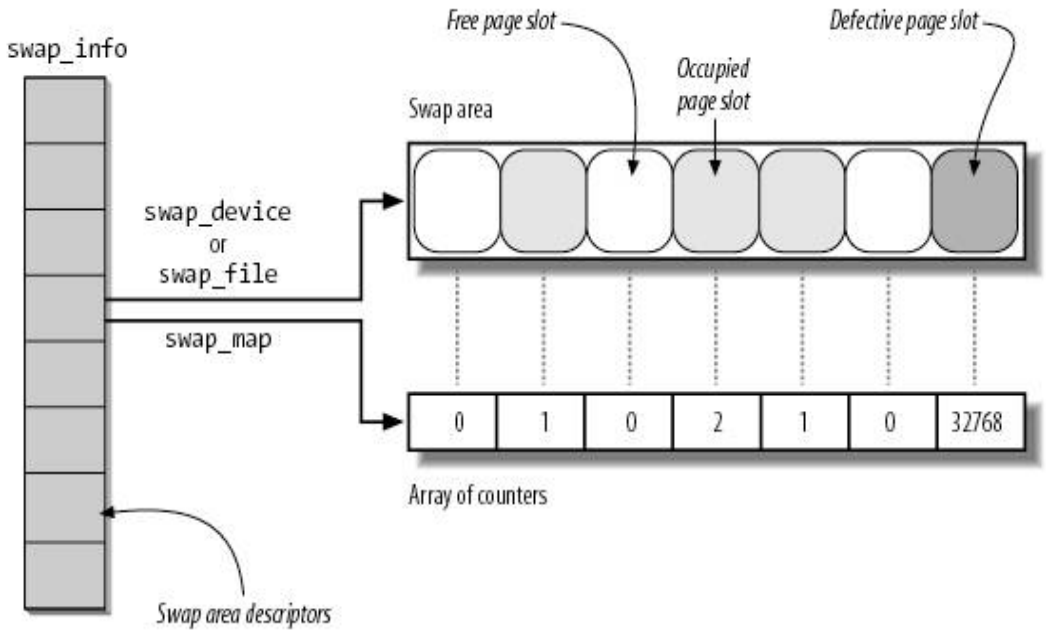
- Funciones fundamentales:
  - Definir áreas de *swap* en el disco para guardar las páginas
  - Administrar el espacio en estas áreas de *swap*
  - Proveer funciones para swap in y *swap out*
  - Identificar en la tabla de páginas el lugar donde está guardada la página en el *swap*
- Como vimos no se encarga de decidir que marcos van al *swap*
- Permite aumentar la cantidad de procesos concurrentes en el sistema (a costa de empeorar la performance)

Áreas de swap

- Normalmente se implementa como una partición en un disco
- Puede ser un archivo también
- Están formados por una serie de marcos de página. El primero se usa para guardar información administrativa
  - Datos del *filesystem*
  - Cantidad de marcos
  - Cantidad de marcos defectuosos
- Se pueden tener varias áreas de *swap* con distintas prioridades

Áreas de swap

- Además se tiene una estructura en memoria que indica si cada marco está usado y cuantos procesos comparten esa página



Fuente: Understanding the linux kernel, 3ra edición

Swap cache

- La transferencia de páginas entre la memoria y el disco puede traer problemas de concurrencia
  - Mientras un proceso está bloqueado leyendo una página del disco otro proceso intenta traerla de nuevo
  - Un proceso intenta traer del *swap* una página que está siendo sacada por el PFRA
- Para solucionar esto se creo el *cache*
  - Antes de empezar una operación de swap se registra en el caché a esa página
  - Antes de tocar el disco siempre se chequea que la página afectada no este ya en el *cache*

Swap token

- Es un método para evitar el *swap thrashing* (hiperpaginación)
- Ya no existe en los kernels nuevos
- Hay un solo *swap token* en el sistema que se le asigna a algún proceso
- El proceso que tiene el *token* evita que el PFRA le saque páginas
- La idea es que alguien pueda trabajar sin problemas y, con suerte, termine y libere algo de memoria
- Cada vez que se produce una falla de página se ejecuta `grab_swap_token()` para ver si se le da el *token* al proceso actual

Swap token

- Antes de asignar el *token* se verifica:
- Hayan pasado al menos dos segundos desde la última invocación a `grab_swap_token()`
- El proceso que tiene el *token* no ha generado fallos de página desde la última invocación de `grab_swap_token()`
- El proceso que tiene el *token* lo ha tenido el token por más *swap_token_timeout ticks* (/proc/sys/vm/swap_token_timeout)
- El proceso actual no ha tenido el token recientemente
- Al morir el proceso se le saca el *token* y se asigna en la próxima invocación a `grab_swap_token()`