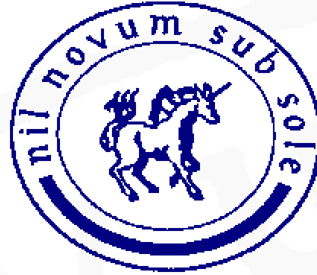


Taller de Sistemas Operativos

Dispositivos de E/S orientados a bloques

Block I/O Layer

- Los block devices se caracterizan por el acceso random de trozos de información de tamaño fijo llamados bloques
- El más común de los block devices son los discos duros
- Otros dispositivos son floppies, CD-Rom, memoria flash
- El otro tipo de dispositivos son los character devices
- En estos los datos se acceden como un stream de datos secuenciales; de a un byte por vez
- El teclado por ejemplo es un character device; El device provee un stream de caracteres que el usuario tipea.
- El driver del disco duro puede pedir leer un bloque cualquiera y luego pedir leer otro no necesariamente consecutivos.

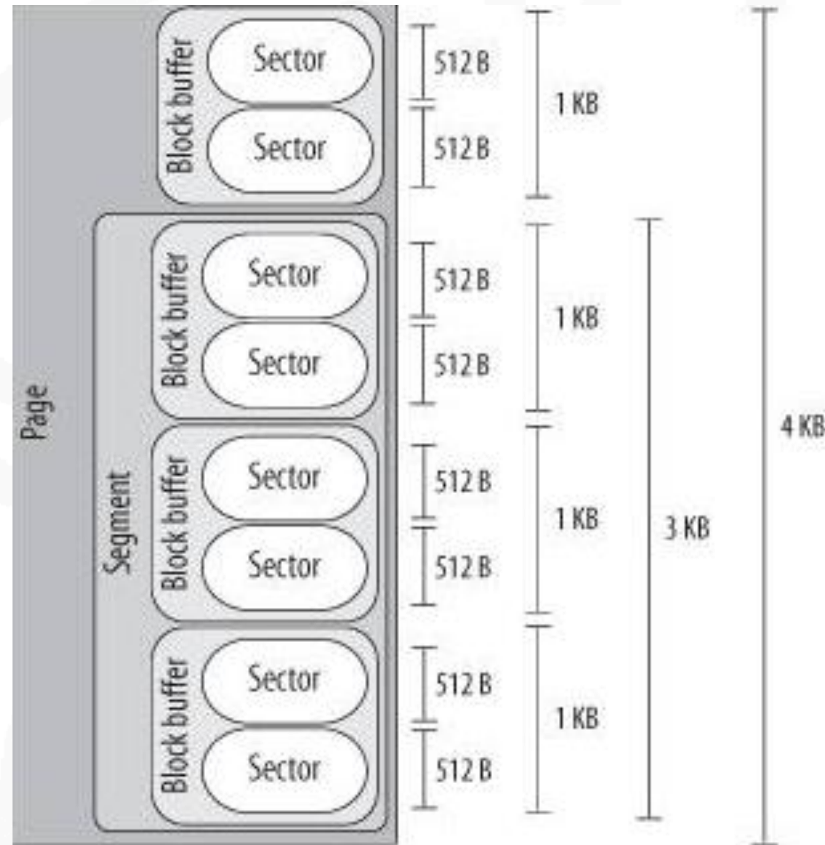
Block I/O Layer

- La administración de los block devices requiere más cuidado, preparación y trabajo que los character devices
- El kernel provee un subsistema entero para el manejo de block devices
- Mejorar la capa de block I/O fue el objetivo principal de la serie de desarrollo 2.5 del kernel

Anatomía de un block device

- La unidad más pequeña direccionable en un block device se denomina sector
- El tamaño de los sectores es en pot (2), pero 512 bytes es el tamaño más común
- El tamaño de sector es una propiedad del dispositivo
- Muchos dispositivos pueden transferir varios sectores a la vez
- El software impone su propio unidad mínima que se llama bloque.
- Todas las operaciones de disco del kernel son en función del tamaño de bloque
- El tamaño de bloque debe ser múltiplo del sector y no mayor que el tamaño de página
- Los tamaños de bloque más comunes son 512 b, 1 Kb y 4 Kb

Anatomía de un block device



Buffers y Buffer Heads

- Los bloques se guardan en memoria en un buffer. Representa un único bloque de disco en memoria
- Una página puede guardar uno o más bloques en memoria
- A cada buffer se le asocia un descriptor llamado buffer head
- El tip de datos de cada buffer head es `struct buffer head`
- Esta estructura contiene toda la información que el buffer necesita para manipular los buffer y se define en `<linux/buffer_head.h>`

Buffers y Buffer Heads

```
struct buffer_head {
        unsigned long          b_state;          /* buffer state flags */
        atomic_t               b_count;          /* buffer usage counter */
        struct buffer_head     *b_this_page;     /* buffers using this page */
        struct page            *b_page;          /* page storing this buffer */
        sector_t               b_blocknr;        /* logical block number */
        u32                    b_size;           /* block size (in bytes) */
        char                   *b_data;          /* buffer in the page */
        struct block_device    *b_bdev;          /* device where block resides */
        bh_end_io_t            *b_end_io;        /* I/O completion method */
        void                   *b_private;       /* data for completion method */
        struct list_head       b_assoc_buffers;  /* list of associated mappings */
};
```

- El propósito de un buffer head es describir el mapeo entre el bloque en disco y el buffer en memoria

Buffers y Buffer Heads

Status Flag	Meaning
BH_Uptodate	Buffer contains valid data
BH_Dirty	Buffer is dirty (the contents of the buffer are newer than the contents of the block on disk and therefore the buffer must eventually be written back to disk)
BH_Lock	Buffer is undergoing disk I/O and is locked to prevent concurrent access
BH_Req	Buffer is involved in an I/O request
BH_Mapped	Buffer is a valid buffer mapped to an on-disk block
BH_New	Buffer is newly mapped via get_block() and not yet accessed
BH_Async_Read	Buffer is undergoing asynchronous read I/O via end_buffer_async_read()
BH_Async_Write	Buffer is undergoing asynchronous write I/O via end_buffer_async_write()
BH_Delay	Buffer does not yet have an associated on-disk block
BH_Boundary	Buffer forms the boundary of contiguous blocksthe next block is discontinuous

Buffers y Buffer Heads

- Antes del kernel 2.6 era una estructura muy importante
- No solo se usaba para describir el mapeo sino era usada como contenedor para toda la I/O de bloques
- Esto generaba básicamente algunos problemas
  - Los buffer heads eran grandes y difíciles de usar para manipular los datos en término de buffer
  - Se deben romper grandes I/O en múltiples estructuras buffer heads
- A partir del kernel 2.6 se utiliza una nueva estructura `bio`

La estructura `bio`

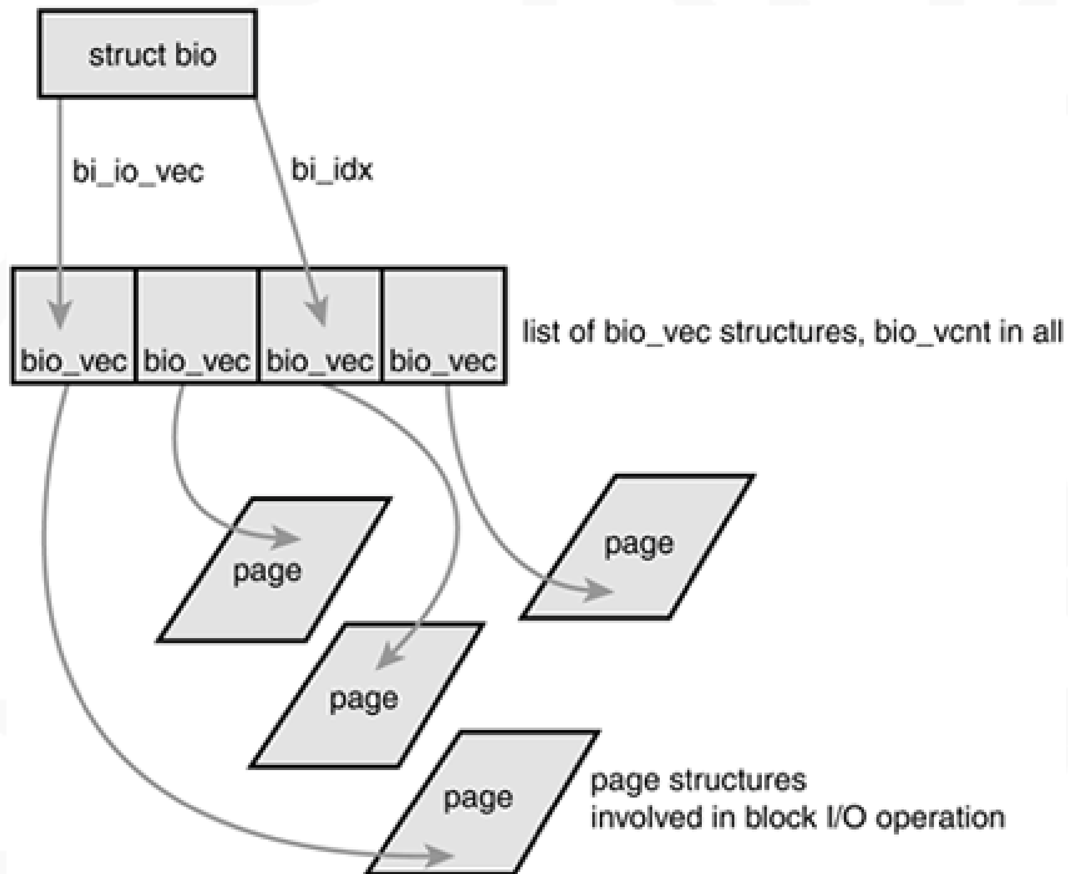
- El contenedor para las operaciones de I/O es la estructura `bio`
- Es definida en `<linux/bio.h>`
- Representa las block I/O activas como una lista de segmentos
- Un segmento es un conjunto de buffers contiguos en memoria
- Como se permite que los buffer se describan de a conjuntos, esta estructura permite al kernel operaciones de block I/O dede distintos lugares en memoria
- Este tipo de I/O se denomina scatter–gather I/O
- El propósito primario de la estructura es representar las operaciones de block I/O corrientes.

La estructura `bio`

```
struct bio {
        sector_t              bi_sector;           /* associated sector on disk */
        struct bio            *bi_next;            /* list of requests */
        struct block_device   *bi_bdev;            /* associated block device */
        unsigned long         bi_flags;            /* status and command flags */
        unsigned long         bi_rw;               /* read or write? */
        unsigned short        bi_vcnt;             /* number of bio_vecs off */
        unsigned short        bi_idx;              /* current index in bi_io_vec */
        unsigned short        bi_phys_segments;    /* number of segments after coalescing */
        unsigned short        bi_hw_segments;      /* number of segments after remapping */
        unsigned int          bi_size;             /* I/O count */
        unsigned int          bi_hw_front_size;    /* size of the first mergeable segment */
        unsigned int          bi_hw_back_size;     /* size of the last mergeable segment */
        unsigned int          bi_max_vecs;         /* maximum bio_vecs possible */
        struct bio_vec        *bi_io_vec;          /* bio_vec list */
        bio_end_io_t          *bi_end_io;          /* I/O completion method */
        atomic_t              bi_cnt;              /* usage counter */
        void                  *bi_private;         /* owner-private method */
        bio_destructor_t      *bi_destructor;      /* destructor method */
};
```

- Los campos más importantes son `bi_io_vec`, `bi_cnt`, `bi_idx`

La estructura `bio`



La estructura `bio`

- El campo `bi_io_vec` apunta a un array de estructuras `bio_vec`
- Cada `bio_vec` es un vector <page,offset,len> que describe un segmento específico
- La estructura `bio_vec` se define en `<linux/bio.h>`
- En cada operación de block I/O hay `bi_vcnt` vectores en el array `bio_vec` comenzando con `bi_io_vecs`
- Cada pedido de block I/O se representa con una estr `bio`
- Cada pedido incluye 1 o más bloques que son guardados en un array de estructuras `bio_vec`
- El campo `bi_idx` es usado para apuntar a la `bio_vec` actual en la lista .
- La estructura `bio` lleva un contador de referencias en el campo `bi_cnt (bio_get y bio_put)`

Buffer Heads vs bio

- La estructura `bio` representa una operación de I/O que puede incluir una o más páginas en memoria.
- La estructura `buffer_head` representa un buffer único que describe un único bloque.
- El uso de `buffer_head` resulta en una división innecesaria en trozos de a un bloque para luego volver a reensamblarlos
- La estructura `bio` puede describir bloques discontiguos
- Buffer Heads parte innecesariamente las operaciones de I/O
- El concepto de `buffer head` se sigue necesitando
- Se utiliza como descriptores de bloque para el mapeo entre bloques de disco y páginas
- La estructura `bio` es un array de vectores describiendo uno o más segmentos de datos para una única operación de block I/O

Request Queues

- Los dispositivos de bloques mantienen request queues para guardar las I/O pendientes
- Cada solicitud es representada por la estructura `request_queue` definida en `<linux/blkdev.h>`
- Se encadena en una lista doblemente encadenada
- Las solicitudes son agregadas por código de más alto nivel
- Mientras la cola no es vacía, el device driver asociado con la cola, toma el primer elemento y le envía al dispositivo asociado
- Cada elemento de la cola es del tipo `struct request`
- Cada solicitud puede estar compuesta por más de una estructura `bio`

Schedulers de I/O

- Enviar las solicitudes de block I/O en el orden que el kernel las genera resulta en una performance horrible
- Una de las operaciones más lentas son los seeks en disco
- Minimizar la cantidad de seeks es crucial para la performnace
- Se realizan operaciones llamads merging y sorting para mejorar la performance del sistema
- El subsistema del kernel que hace esta tarea se llama I/O scheduler
- El I/O scheduler divide el recurso de I/O sobre disco entre los pedidos de block I/O pendientes en la request queue
- Virtualiza los block devices entre múltiples solicitudes de block I/O
- Se realiza con el objetivo de minimizar seeks y asegurar la mayor performance a nivel de disco

Schedulers de I/O

- Un I/O scheduler trabaja administrando la request queue de un block device
- Decide el orden de las solicitudes en la cola y cuando es despachada hacia el dispositivo
- Intenta minimizar la cantidad de seeks
- Puede ser "injusto" con algunas solicitudes para mejorar la performance del sistema
- Dos acciones primarias (merging and sorting)
- Merging es juntar dos o más solicitudes en una sola
- Si dada una solicitud, existe otra en la cola para acceder a un sector adyacente en disco, entonces las dos solicitudes son "merged" en una única
- Haciendo merging se minimiza la cantidad de operaciones sobre el dispositivo y la cantidad de seeks

Schedulers de I/O

- En el caso que no se pueda hacer “merge” se buscan solicitudes a lugares “similares” en el disco
- Se inserta esta solicitud en la request queue en un lugar “cercano” a otras solicitudes que acceden a lugares “cercanos” de disco
- La request se mantiene ordenada
- Esta operación se denomina “sorting”
- El objetivo no es solo minimizar la cantidad de seeks sino también mantener la cabeza del disco moviéndose en línea recta.
- Esto es similar al algoritmo usado en ascensores (elevators)
- Por este motivo a los I/O schedulers se los llaman también elevators.

Linus Elevator

- El primer scheduler de I/O se llama Linus elevator
- Era el scheduler por defecto en Linux 2.4
- Linus elevator realiza merging y sorting
- Cuando llega una nueva solicitud de I/O se verifica si es un posible candidato para hacer merging
- Si no es posible hacer merging, se busca un punto de inserción el la cola de solicitudes
- Si se encuentra un lugar, se inserta allí, sino se inserta al final de la cola
- Si una solicitud existente es anterior a un determinado tiempo, la nueva solicitud se inserta al final de la cola
- Esto evita que muchas solicitudes en lugares cercanos de disco pospongan idefinidamente (starve) a solicitudes sobre otros lugares de disco.

Linus Elevator

- Todo lo anterior mejora la latencia, pero puede terminar en posposición indefinida; lo cual es un un arreglo necesario de la versión 2.4 del kernel
- En resumen
- Si la solicitud es adyacente a una en la lista, se hace merge
- Si una solicitud en la lista es suficientemente vieja, la nueva solicitud es insertada al final
- Si se encuentra un lugar adecuado en la lista, la nueva solicitud es insertada allí
- Si no se puede insertar, se inserta al final de la lista

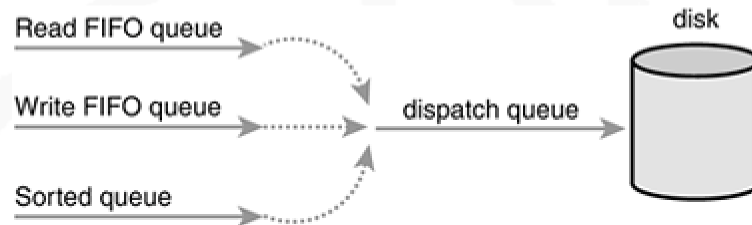
Deadline I/O Scheduler

- Pensado para prevenir la posposición indefinida del Linus elevator
- En situaciones de carga de I/O pesada a sectores adyacentes, pedidos a sectores lejanos son pospuestos
- Las operaciones de write pueden ser pospuestas con respecto a la aplicación que las inicia
- Las lecturas son distintas, por lo gral cuando una aplicación hace un read quiere esperar a obtener el resultado
- Las lecturas tienden a ser dependientes unas de otras
- El deadline I/O scheduler implementa algunas características para asegurar que la posposición de solicitudes (en especial de los reads) es minimizada
- Reducir la posposición tiene un costo al rendimiento global del sistema de I/O (throughput)

Deadline I/O Scheduler

- En el deadline I/O scheduler cada solicitud se le asigna un tiempo de expiración
- Por defecto
  - 500 ms para los read
  - 5 seg para los write
- Se trabaja de manera similar al Linus elevator manteniendo las solicitudes ordenadas por sector de disco. Se le denomina cola ordenada
- Cuando llega una nueva solicitud se intenta hacer merging y sorting
- Además se inserta las solicitudes de read en una cola FIFO (read queue) y las solicitudes de write en otra cola FIFO (write queue)

Deadline I/O Scheduler



- En condiciones normales el deadline I/O scheduler toma las solicitudes de la cola ordenada.
- Como resultado se minimizan los seeks
- Si una solicitiud en cualquiera de las colas FIFO expira el scheduler empieza a tomar solicitudes de las colas FIFO
- De esta manera el scheduler intenta asegurarse que la solicitud no se encuentra mucho más en cola que su tiempo de expiración
- El scheduler no da garantías estrictas sobre la latencia de las solicitudes

Deadline I/O Scheduler

- Es capaz de satisfacer las solicitudes sobre su expiración
- Esto previene la posposición de las solicitudes
- Como las solicitudes de lectura tiene una expiración menor que las de escritura, nos podemos asegurar que los writes no posponen a los reads.
- Esto minimiza la latencia de los read
- Este scheduler reside en `drivers/block/deadline-iosched.c`

Anticipatory I/O Scheduler

- El deadline I/O scheduler hace un gran trabajo para minimizar la latencia de los read a costo de rendimiento global (throughput)
- Qué sucede durante una I/O pesada de escritura … ?
- El anticipatory I/O scheduler usa el deadline como base
- Implementa tres colas como el deadline
- El gran cambio es el agregado de la heurística de “anticipación”
- Intenta minimizar la “tormenta” de seeks que acompaña a las solicitudes de read en conjunto con otra actividad de I/O
- Cuando llega una solicitud de read, la sirve como siempre.
- Cuando la solicitud es servida, en vez de seguir atendiendo otras solicitudes espera por unos milisegundos (6)
- En este tiempo hay buena probabilidad que la aplicación envíe otra solicitud de lectura.

Anticipatory I/O Scheduler

- Las solicitudes a sectores adyacentes en disco son atendidas inmediatamente
- Luego que el período de espera termina, el scheduler reanuda su trabajo donde lo dejó
- Es importante destacar que el tiempo dedicado en la “anticipación” bien vale la pena si minimizan por lo menos un modesto porcentaje de seeks como resultado de atender reads
- La clave de sacarle el máximo provecho del anticipatory I/O scheduler es poder predecir el comportamiento de I/O de aplicaciones y filesystems
- Esto es realizado mediante un conjunto de estadísticas y heurísticas asociadas
- Intenta minimizar tanto la latencia en los read como la cantidad de seeks

Anticipatory I/O Scheduler

- Este scheduler se comporta bien en varios escenarios de I/O
- Es el scheduler de I/O por defecto en Linux 2.6
- Reside en `drivers/block/as-iosched.c`

CFQ I/O Scheduler

- Complete Fair Queuing (CFQ) I/O scheduler es específico para un tipo de carga de I/O pero provee buena performance para diferentes tipos de cargas
- Clasifica las solicitudes de I/O por el proceso que origina la I/O
- Dentro de cada cola de realiza merging y sorting
- La diferencia con CFQ es que existe una única cola por proceso que genera I/O
- El CFQ scheduler sirve las colas en modo round robin un número de solicitudes de I/O por cola y luego la siguiente
- Esto logra equidad a nivel de proceso
- El escenario de uso de este scheduler es multimedia
- En la práctica, CFQ se comporta bien en muchos escenarios
- Reside en `drivers/block/cfq-iosched.c`

Noop I/O Scheduler

- No hace demasiado
- No hace sorting ni utiliza otra técnica de prevención de seeks
- No necesita implementar algoritmos para minimizar la latencia de las solicitudes
- Solamente realiza merging
- Básicamente mantiene las solicitudes en un orden casi FIFO
- El escenario de uso es para dispositivos realmente random-access como tarjetas de memoria flash
- Si un block device tiene poco o nada de overhead asociado al hacer seek, el Noop I/O scheduler es el candidato ideal
- Reside en `drivers/block/noo-iosched.c`

Selección del I/O Scheduler

- En tiempo de boot con la opción `elevator=e` donde `e` es uno de los siguientes

Parameter	I/O Scheduler
as	Anticipatory
cfq	Complete Fair Queuing
deadline	Deadline
noop	Noop

- `elevator=deadline` utiliza el deadline I/O scheduler para todos los block devices
- En tiempo de ejecución

```
lx:> echo deadline > /sys/block/sda/queue/scheduler
lx:> cat /sys/block/sda/queue/scheduler
noop anticipatory [deadline] cfq
```

- Habilita el deadline I/O scheduler en el dispositivo sda