The I/O system

The main purpose of the I/O-system is to hide details of specific hardware units from other parts of the kernel.

The I/O-system consists of:

- Buffer cache system
- General driver code
- Drivers for specific hardware devices

Three main types of I/O in FreeBSD (fig. 6.1)

- filesystem
- character devices
- sockets

In most Unix systems disks are accessible both as character device and block device special files.

The block devices were removed from FreeBSD5.2.

I/O buffers

The file system use a buffer cache to reduce the number of data transports between main memory and a device.

- Each buffer is described by a *buf struct*
- A *buf struct* is used both to locate a data block in the cache and to describe a data transport between the main memory and the disk
- Each *buf struct* points to a data area in kernel virtual address space (In FreeBSD 5.2 this data area is a page frame)

The character interface is used by terminals, but also for reading and writing to disk memory without using the buffer cache (raw I/O)

- Raw I/O is used by system utilities like *fsck* and for paging operations to the swap area
- Raw I/O also use a *buf struct* to describe the data transport, but in this case it is not connected to the buffer cache and is called a *swap buffer*
- Raw I/O requires read/write operations to use complete disk blocks

Device Drivers

A device driver is divided into three main sections:

- Autoconfiguration and initialization routines
- Routines for servicing I/O requests (top half)
- Interrupt service routines (bottom half)

I/O Queuing

- Device drivers manage a queue of I/O requests that have not yet completed.
- When an input or output request is received by the top half it is recorded in a data structure (typically a *buf struct*) that is added to the I/O queue.
- When an I/O operation completes, the interrupt service routine removes the request from the queue and wakes top half.
- To prevent the I/O queue from being simultaneously modified from the top half and the bottom half, it has to be locked by a *mutex* when it is updated.

Interrupt Handling

- On receiving a device interrupt, the hardware calls an interrupt handler on an address determined by the interrupt vector table.
- In reality the same interrupt handler is used for all interrupt vectors.
- This common interrupt handler can identify the interrupting device with help of information pushed on the kernel stack by interrupt handling hardware.

The common interrupt handler takes the following actions:

1. Collects the relevant hardware parameters
2. Update statistics on device interrupts
3. Schedule the interrupt service thread for the device
4. Clears the interrupt-pending flag in the hardware
5. Returns from interrupt

Entry Points for Character Devices

Device drivers for character devices have entry points as shown in table 6.1

Entry Points for Disk Device Drivers

In FreeBSD 5.2, disk device drivers are called via the GEOM layer.

In FreeBSD 5.2 device drivers for disk devices contain all the character device entry points but in addition it has a *strategy* entry point.

open: Called for each open system call on the device special file or internally from the *mount* system call.

strategy: A common entry point for read and write operations. Each call to the strategy routine specifies a pointer to a buf struct containing the parameters for an I/O request. The strategy routine is usually called from the block I/O subroutines *bread()* and *bwrite()*.

Close: Called after the final client interested in using the device terminates. Disk devices have nothing to do at close. For magnetic tape, an endmark is usually written to the tape.

Sorting of Disk I/O Requests

- The kernel provides a general *disksort()* routine that can be used by all the disk device drivers.
- The routine sorts the device driver queue using a SCAN algorithm.
- With the BSD filesystem disk sorting is important only when there are multiple simultaneous users of a disk.

Disk Labels

- A disk may be broken up into several partitions, each of which may be used for a separate partition or swap area.
- In order to be able to locate a disk block, the device driver must know the partition layout.

In older Unix systems, the partition table was stored in the device driver. This had some disadvantages:

1. If the partition layout for a disk was changed, the driver had to be recompiled.
2. All disks of the same type needed to have the same partition layout.
3. Unless a standardized partition layout was used, a new version of the kernel had to me modified to be able to use existing disks.

Disk Labels and Partitions

- Today the partition table, sometimes called a disk label, is stored on the disk.
- Information about the disk partition table need to be stored in block 0 on the disk, as this is the only block that can be located without knowing anything about the disk layout.
- Block 0 also stores a simple boot loader.
- For PC machines information about four partitions is stored in MBR (Master Boot Record) in block 0.

For each of the partitions the following information is stored:

- If it is a *primary* or *extended* partition.
- The partition *type*.
- Address to the first block in the partition.
- The partition size.
- One of the partitions is marked as boot partition.

The first block in each partition is a boot block for that partition.

Descriptor Management

- For user processes, all I/O is done through descriptors.
- System calls that refer to open files take a file descriptor as an argument to specify the file.
- The file descriptor is used by the kernel to index into the the *descriptor table* (part of process structure) for the current process.
- Each entry in the descriptor table point to a file structure that points to a vnode or a socket (fig. 6.4).

File structure (file entry)

- The original use of *file struct* was to hold the read/write offset for open files.
- Today file struct also contains a type and an array of function pointers that translate the generic operations on file descriptors to the specific routines for their type.
- The following operations exist:
  - → fo_read
  - → fo_write
  - → fo_ioctl
  - → fo_poll
  - → fo_close

The descriptor type may be VNODE or SOCKET.

Multiplexed I/O

- A process sometimes want to handle I/O on more than one descriptor.
- In this case blocking read operations cannot be used.
- The traditional Unix method to solve this problem have been to use concurrent processes.
- A drawback is that process switching is slow.

FreeBSD provides three methods that permit multiplexing I/O on descriptors:

- Nonblocking I/O
- Signal-driven I/O
- Polled I/O using select or poll

Multiplexed I/O, cont.

Four alternatives that avoid the blocking problem:

1. Set all the descriptors into nonblocking mode.
   - The problem with this is that the process must run continuously (Busy wait !).
2. Enable all involved descriptors to send a signal when I/O can be done.
   - A signal handler is called when I/O can be done.
   - A drawback is that signals are expensive to catch.
3. Have the system provide a system call that can check which descriptors are capable of doing I/O.
   - A drawback is that the process has to do two system calls for each I/O operation.
   - Implemented in FreeBSD via *select*.
4. Use threads.
   - Potential problems are lack of good standards for programming with threads and also the memory demand may be to big if a large number of threads are needed.
   - Implemented in FreeBSD 5.2.

Movement of Data Inside the Kernel

- When writing to block devices, data must first be copied to a kernel buffer before it can be sent to the driver.
- The kernel buffer can be written from several user buffers in one operation (scatter/gather I/O).
- Useful for example to add a header to a network packet.
- Data movements between the kernel and a process is described by a *uio structure.*
- Base address and length for a user mode buffer is given by an *iovec*.
- An *uio structure* contains (fig. 6.6):
  - → Pointer to an array of *iovec structures*
  - → The number of elements in the *iovec* array
  - → Offset in the file
  - → READ/WRITE flag

Movement of Data Inside the Kernel cont.

Copying of data is done by the routine:

```
uiomove(kaddr, nbytes, uio);
```

kaddr specifies the address to a kernel buffer (described by a *buf struct*).

Raw I/O

- Raw I/O do not use the buffer cache, instead data are transferred directly to/from a user buffer.
- The kernel still have to allocate a *buf struct* (swap buffer) to describe the data transport.
- Driver code also requires the user data buffer to be mapped into the kernel virtual address space.
- Read and write operations from a raw device special file calls the *physio()* routine to perform the I/O operation.

FreeBSD Buf struct

```
struct buf {
   b_bcount;   /* Valid bytes in buffer. */
   b_data;     /* Memory adress for data transport */
   b_dev       /* Device to do I/O on */
   b_iocmd     /* I/O operation */
   b_ioflags;  /* flags for I/O operation. */
   b_iooffset  /* offset into file */
   b_resid;    /* Remaining I/O in bytes */
   b_blkno;    /* Underlying physical block number. */
   *b_left;    /* incore hash chains */
   *b_right;   /* implemented with splay trees */
   b_vnbufs;   /* Buffer's associated vnode. */
   b_freelist; /* Free list position if not active. */
   *bio_queue; /* Device driver queue when active */
   b_flags     /* Buffer status flags */
   b_bufsize;  /* Allocated buffer size. */
   b_kvabase;  /* kernel virtual adress (kva) for buffer */
   b_kvasize   /* size of kva for buffer */
   b_saveaddr; /* Original b_addr for physio. */
   b_lblkno;   /* Logical block number. */
   *b_vp;      /* Device vnode. */
   (*b_iodone) /* Function to call upon completion. */
   ...
};
```


Flags for buf struct

b_flags in buf struct use the following flags (among others):

B_CACHE Indicates that the buffer is entirely valid.

B_MALLOC Request that the buffer be allocated from the malloc pool.

B_VMIO Indicates that the buffer is tied into an VM object.

Physio()

Simplified FreeBSD physio()

```
physio(strategy, bp, dev, flags, uio)
   int strategy();
   struct buf *bp;
   dev_t dev;
   int flags;
   struct uio *uio;
{

    bp = getpbuf();  /* allocate swap buffer */
    while (uio is not exhausted) {
       /* set up the buffer */
       bp->b_flags = 0;
       bp->b_bcount = iovp->iov_len;
       bp->b_data = iovp->iov_base;
       lock the part of the user address space
           involved in the transfer;
       vmapbuf(bp, bp->b_count); /* map into kernel VM */
      (*strategy)(bp); /* start transfer */
       wait for  transfer to complete;
       unlock user address space;
       vunmapbuf(bp, bp->b_count); /* unmap from kernel*/
       iolen = bp->b_bcount - bp->b_resid; /* transfered no of bytes
       iovp->iov_len -= iolen;  /* update uio */
       iovp->iov_base += iolen;
    }
    free swap buffer;
}
```


The Virtual filesystem interface

- In BSD4.3 and earlier, the *file struct* directly referenced the *inode*.
- With the advent of multiple filesystem types, this did not work anymore.
- The solution was to add a new object-oriented data structure between *file struct (file entry)* and the *inode*.
- This was first implemented by Sun, who called the new data structure a *vnode (virtual node)*.

Vnode

The *vnode* contains among other things the following information:

- Flags that may specify for example that that the vnode represents an object that is the root of a file system.
- Reference counts for the number of “users” of the vnode.
- A pointer to the *mount structure* that describes the filesystem that contains the object represented by the vnode.
- A pointer to the set of vnode operations defined for the object.
- A pointer to the *inode or nfsnode* (private information for the object represented by the vnode).
- The type of the underlying object (for example directory or file). This is only an optimization.
- Lists with clean and dirty buffers. The *dirty* list is used by fsync to quickly locate buffers that need to be written back to the disk. The *clean* and *dirty* lists are used when a file is deleted to free all the buffers used by the file.
- The *mount struct* points to a list of vnodes for all open files in the filesystem. Can be used by *sync*.

Vnode Operations

- Which operations that are called via a vnode can be dynamically changed.
- At system boot, every filesystem registers which vnode-operations it supports.
- For every filesystem, an operation vector is built:
  - → For supported operations, it is filled in with the address to the routine implementing the operation.
  - → For not supported operations, a routine which calls a lower level in the vnode-stack may be used or an error routine may be called.

The filesystem code was split in to parts in 4.4BSD:

1. Routines that implement the naming in the filesystem tree.
2. Routines that implement the physical storage of data in a flat name space (block numbers).

Mount Parameters

- Some properties of a filesystem can be specified in the mount command and are stored as flags in the *mount struct.*
- Some things that can be specified:

Noexec Files on this filesystem may not be executed.

Nosuid Set-user-id programs can not be executed on this filesystem. Useful if free mounting of unknown disks is allowed.

Nodev Do not allow special files on this filesystem.

Pathname Translation

At open, a filename given as a path name shall be translated to a pointer to a vnode.

The pathname translation proceeds as follows:

1. The pathname to be translated is copied from the user process to the kernel.
2. The starting point is determined: *root* or *current directory*. The vnode of the appropriate directory becomes the *lookup directory* used in the next step.
3. The vnodes *lookup()* routine is called with one name component as parameter. The underlying filesystem looks up the component in the *lookup directory* and returns its vnode:
   - If the last component: ready
   - If not the last component and not a directory: error!
   - If not the last component and a directory:
     → If mount point: *lookup directory* is set to the mounted filesystem.
     → Otherwise: *lookup directory* becomes the returned vnode.

Filesystem-Independent vnode-services

- The vnode interface also provides a set of management routines that can be used by the client filesystem or other parts of the kernel.
- FreeBSD has a global pool of vnodes that are shared among all the filesystems.

getnewvnode(): Allocate a new vnode. Take from the vnode free-list in LRU order.

inactive(): Called when the vnode usage count drops to zero, due to the close of the last reference to the file. Modified blocks are written back to disk but the blocks are usually left in the buffer cache. The vnode is added to the vnode free-list.

reclaim(): Called by *getnewvnode()* to completely free the vnode from its previous usage. Removes associated cached data blocks from their hash lists.

The Name Cache

To optimize the translation of pathname to vnode, a name cache is used.

Operations:

- Add a name to the cache
- Look up *name,directory-vnode* - return a pointer to a vnode
- Remove a name from the cache

Certain content in this cache becomes invalid when a vnode is reclaimed.

- Each vnode has a list of its entries in the cache.
- Each directory vnode has a second list of all the cache entries for names that are contained within it.
- When a directory vnode is to be purged, all name-cache entries on this second list must be removed.
- A vnode's name-cache entries must be purged each time the vnode is reused by getnewvnode() or if the name is changed for a directory.

The Name Cache - cont.

- The cache management routines also allows for negative caching.
- If a name is looked up in a directory and not found, it can be entered in the cache.
- If the name is later looked up, the cache will inform that the name is NOT in the directory.
- Improves path searching in command shells.

Block I/O interface

The block I/O routines in FreeBSD address logical block numbers in a file.

Bread() Read the specified logical block and return a locked buffer for it.

Breadn() As bread, but also start read-ahead of additional blocks.

A buffer can be released (written to disk) with:

brelse() Return the buffer to the free list an wake up any processes waiting for it. The buffer may not be modified.

bdwrite() Mark the buffer *dirty* and return it to the free list without initiating I/O and also wake up any waiting processes. The buffer is written by *sync* within 30 seconds.

bawrite() Initiates asynchronous write on the buffer.

bwrite() Starts I/O on the buffer and waits for the data to be written to the disk.

Buffer pool

- A buffer with valid content is on exactly one *bufhash* list (fig. 6.8).
- The lists are hashed on <vnode, logical block-number>
- A buffer is removed from the hash list only when the content becomes invalid or it is reused for other data.
- In FreeBSD the hash lists are implemented as splay trees.

Besides on a hash list the buffer also is on exactly one free list:

Locked: Buffers on this list are locked in the cache. Used in FreeBSD for blocks being written in the background.

Dirty: Buffers that contain modified data not written to the disk. When too many buffers are dirty the *buffer daemon* is started.

Clean: Buffers with content that is known but not modified, or rewritten to disk.

Empty: The empty buffers are just headers that have no memory associated with them.

When a new buffer is needed it is taken from the *empty* list, unless the maximum number of buffers is already allocated. In this case new buffers are taken from the *clean* list.

Bio Routines

bread - block read

in data: struct vnode * vp, daddr_t blkno, int size,

Out data: struct buf * bp – Pointer to a locked buffer

```
bread(...)
{
   bp = getblk(vp, blkno, size) /* get buffer */
   if (buffer data not valid) {
       VOP_STRATEGY(vp, bp); /* fill buffer */
   }
   return;
}
```

Bio Routines, cont.

getblk - returns a pointer to a locked buffer

```
struct buf *
getblk(vp, blkno, size)
{
   if (buffer in hash list)
       return buffer;
   else {
      bp = getnewbuf(size); /* get buffer header and kva*/
      insert buffer into hash list;
      allocbuf(bp, size); /* allocate physical memory */
   }
}
```

Allocbuf reserves physical memory for a buffer. For blocks in files, page frames from the virtual memory are used.

Allocbuf()

Allocbuf() allocates physical memory for a buffer. Virtual kernel addresses for the buffer has been allocated by *getnewbuf()*.

- For logical blocks in a file, page frames from the virtual memory system are used as buffer memory.
- Allocbuf() also calls *pmap_qenter()* to map the buffer into kernel virtual memory.

Filesystem meta data like inodes and bitmaps do not exist in the virtual memory.

- In this case, allocbuf() calls *malloc* to allocate memory for the buffer
- If a buffer already have too much memory, allocbuf() can free unneeded memory.

Stackable filesystems

- It can be desirable to be able to provide new filesystem features without modifying the existing stable code.
- One approach is to provide a mechanism for stacking several filesystems on top of each other.
- To make this possible the vnodes need to be stackable.
- The stackable vnodes in FreeBSD is taken from 4.4BSD with minor modifications.
- The bottom of a vnode stack tends to be a disk-based filesystem.
- The layers above typically transform their arguments and pass on to a lower layer.
- To connect a new vnode-layer in the file system, the *mount* command is used.

Stackable filesystems, cont.

- Commands are called via the operations vector in a specific vnode.
- Thus, every level in the vnode stack have its own version of the command code.
- When a command is called (for example open or read) via a vnode, that vnode has several options:
  - → Do the requested operation and return a result.
  - → Pass the operation without change to the next-lower level in the vnode stack. When the operation returns from the lower vnode, it may modify the results or simply return them.
  - → Modify the parameters provided with the request and pass the operation to the next-lower level. The result from the lower level may be modified or returned without change.

If the call propagates to the bottom of the vnode-stack without any layer taking action on it, an error code “*operation not supported*” is returned.

Stackable filesystems, cont.

In FreeBSD, the operations vector is dynamically built at boot time.

Together with stacked vnodes, this gives two problems:

1. The filesystem must be able to bypass operations that are not defined in the filesystem implementation.
2. It must be possible to handle parameters to not implemented operations (which are of unknown type).

Solution to these problems:

1. Not defined operations are handled by a bypass operation.
2. To be able to handle the parameters, they are always packed into an argument structure that is passed as a single parameter to the vnode operation (fig. 6.11)

The Union filesystem

- The union filesystem can mount a filesystem atop of another existing filesystem.
- Unlike normal mounts, both filesystems are visible after the mount (fig. 6.13).
- The union filesystem is implemented using stackable vnodes.
- All levels below the top layer are read-only.
- If a file residing in a layer is opened for writing it is copied to the top level.
- Removal of a file at a lower level is tricky, because this level may not be written.
- It is solved by placing a whiteout file at the top level.
- The without file have an inode number of 1 and the same component name as the file it is blocking.

The Union filesystem, cont.

Possible uses for the union filesystem:

- To compile programs for several architectures from a common NFS-mounted source text and get the object files in local directories.
- To compile sources on a CD-ROM without having to copy the source texts.