

Unix Internals

Module 07

Raju Alluri

askraju @ spurthi.com

Unix Internals, Module 07

- Device Drivers
- Streams

Device Drivers

- Device Drivers
 - Part of Kernel
 - Collection of Data structures & Functions
 - Only module that may interact with a device
- Benefits
 - Isolation of device specific code
 - Easy to add new devices
 - Devices can be developed without kernel code
 - Consistent view of devices to kernel (and hence to users)

Hardware Configuration

- Devices are connect thru Controllers (Adapters)
- A Controller may connect more than one device
 - Has “Control and Status Registers” for each device
 - Control Registers used to perform actions on devices
 - Repeatability of actions?
 - Value written might not be same as value read
 - Status registers used to get status
- Device Data Transfers
 - Programmed I/O
 - DMA
 - DVMA

Device Interrupts

- Devices use interrupts to get CPU attention
- Unix defines ipls (Interrupt Priority Levels)
 - Ranges from zero (lowest) to above
 - Kernel/User code runs at ipls of zero
 - Each Controller/Device uses a fixed ipl to interrupt

Device Driver Framework

- Driver Classification
 - Character Devices: Transfer arbitrary sized data
 - Typically interrupt driven
 - Block Devices: Transfer data in fixed sizes
 - Typically do I/O to paged memory
 - Use buf structures
- Special Drivers:
 - drivers without devices (pseudo drivers)
 - Mem driver: Access physical memory
 - Null driver: data sink, used to write anything to a black hole

Driver Code

- Driver Code
 - Configuration: Boot time, to initialize
 - I/O: by I/O subsystem to read/write
 - Control: control operations like open/close/rewind
 - Interrupts: Device to CPU communication (I/O completion, error status etc.)
- Synchronous
 - I/O & Control Code
- Asynchronous
 - Interrupts

Driver Code

- Driver Routines
 - Top Half: Synchronous Code
 - Execute in Process Context
 - Access address space and u area of the process
 - May result in a sleep if needed
 - Bottom Half: Asynchronous Code
 - Mostly not related to current process
 - Not allowed to access address space and u area of the process
 - Not allowed to sleep

Device Switches

- Block Device

```
struct bdevsw {  
    int (*d_open)();  
    int (*d_close)();  
    int (*d_strategy)();  
    int (*d_size)();  
    int (*d_xhalt)();  
} bdevsw[];
```

Device Switches

- Character Device

```
struct cdevsw {  
    int (*d_open)();  
    int (*d_close)();  
    int (*d_read)();  
    int (*d_write)();  
    int (*d_ioctl)();  
    int (*d_mmap)();  
    int (*d_segmap)();  
    int (*d_xpoll)();  
    int (*d_xhalt)();  
  
    struct streamtab *d_str;  
};  
cdevsw[];
```

I/O Subsystem

- Major/Minor Device Numbers
- Device Files
- The specfs filesystem
- Common s-node
- Device Cloning
- I/O to a character device

Major/Minor Device Numbers

- Major device number
 - Identifies the type of device (i.e. Driver to be used)
 - Typical Usage
 - (*bdevsw[getmajor(dev)].d_open)(dev,)
- Minor device number
 - Identifies specific instance of device
- dev_t is a combination of major/minor device numbers
- A single driver may be given multiple major numbers
- A single device may be given multiple minor numbers

Major/Minor Device Numbers

- SVR4 dev_t: 32 bit info
 - 14 bits for major device number
 - 18 bits for minor device number
 - Internal device numbers
 - Identify the driver (index to driver switches)
 - getmajor(), getminor()
 - External device numbers
 - User visible representation of the device
 - Stored in i-node (i_rdev field) of the device special file
 - getemajor(), geteminor()

Device Files

- Kernel view: Device Numbers
- User view: device files
 - Part of file system name space
 - By convention, use /dev/....
 - Has inode, but no blocks on file system
 - Only super user creates these files using mknod()
 - IFBLK, IFCHR
 - Advantages
 - User programs use same routines for devices and files
 - Device file access thru regular file access control

The specfs File System

- File system is accessed thru vfs/v-node interface
- Vnode of a ufs file points to a vector ufsops
 - Ufsops has pointers to ufslookup(), ufsclose() etc.
- When device files reside on a vfs managed filesystem, how to handle device file info?
 - Vnode for /dev/lp has file type IFCHR
 - Get device numbers from inode
 - Pass them to specvp()
 - Specvp() finds snode for file

I/O to character device

- Driver does most of the work
- When a user opens the file
 - Create snode & common snode
- When a user makes a read
 - From file descriptor, dereference vnode
 - Do VOP_READ on vnode, resulting a call to spec_read()
 - spec_read() uses cdesw[] table because its a character device, and calls d_read() routine
 - d_read() for character device is a synchronous operation

The poll() system call

- To multiplex I/O over several descriptors
 - Checking each file descriptor will block until that descriptor is ready
 - What happens if other descriptors have data before the one you are checking
- Poll() system call

```
poll(struct pollfd *fds, int nfds, int  
     timeout);
```

```
struct pollfd {  
    int fd;  
    short events;  
    short revents;  
};
```

The poll() system call

- Poll() system call

```
poll(struct pollfd *fds, int nfds, int  
     timeout);
```

- `fds`: array of `pollfd` struct elements
- `events`: `POLLIN`, `POLLOUT`, `POLLERR`
- `timeout`: if 0, return immediately
- `timeout`: if `INFTIM` or -1, wait until an event happens

The poll() implementation

- Two key data structures
 - Pollhead: Associated with each device file, maintains a queue of polldat structures
 - Polldat: identifies a process waiting for that device file and interested events.
- Poll() system call implementation
 - First, loop thru all vnodes of device files and do a

```
error = VOP_POLL(vnp, events, anyyet,  
                &revents, &php);
```
 - If there is an event, then call pollwakeup() with pollhead of the device
 - If there is no event, add a polldat for current process to the pollhead

The poll() implementation

- Poll() system call implementation
 - pollwakeup() traverses thru the polldat chain for the pollhead and wakes up each process with event info.

The select() system call

- 4.3 BSD has a select() system call similar to that of poll()
 - Select(nfds, readfds, writefds, exceptfds, timeout);
 - Readfds, writefds exceptfds are pointers to descriptor sets
 - Fixed size arrays (size nfds) where non-zero values indicate file descriptors of interest
 - Operations on descriptor sets
 - FD_SET(fd, fdset)
 - FD_CLR(fd, fdset)
 - FD_ISSET(fd, fdset)
 - FD_ZERO(fdset)

Block I/O

- Block I/O has more involvement of I/O subsystem
- Two types of block devices
 - raw/unformatted
 - Direct access thru device files
 - Those that contain unix filesystems
 - Reading/writing to a ordinary file
 - Reading/writing to a device file
 - Accessing a memory mapped file
 - Paging to/from a swap device

The buf structure

- The buf structure has the following data
 - Major/Minor device number
 - Starting block number of data on device
 - Number of bytes to transfer (multiples of sector size)
 - Location of data in memory
 - Flags (read/write, synchronous or not)
 - Address of completion routine to be called from interrupt handler

The buf structure

- Modern Unix Systems also have
 - Pointer to the vnode of the device file
 - Flags that state the state of the buffer (free/busy, dirty)
 - Aged flag
 - Pointers to keep buffer in on an LRU free list
 - Pointers to chain the buffer in hash queue (vnode/block number)

Device access methods

- Pageout operations
 - Pagedaemon flushes dirty pages to disk (keep mostly used pages in memory) regularly
 - Locates vnode from page structure, invokes VOP_PUTPAGE
 - For device files, it calls `spec_putpage()`, resulting in `d_strategy()` call
 - For ordinary files, it calls `ufs_putpage()`, which calls `ufs_bmap()` (to compute physical block number) and then `d_strategy()`

Device access methods

- Pageout operations
 - Pagedaemon flushes dirty pages to disk (keep mostly used pages in memory) regularly
 - Locates vnode from page structure, invokes VOP_PUTPAGE
 - For device files, it calls `spec_putpage()`, resulting in `d_strategy()` call
 - For ordinary files, it calls `ufs_putpage()`, which calls `ufs_bmap()` (to compute physical block number) and then `d_strategy()`

Device access methods

- Mapped I/O to a file
 - Whenever a process accesses a mmap()ed area and the page is NOT already in memory, a page fault occurs
 - The fault is handled by segvn_fault(), which invokes VOP_GETPAGE()

Device access methods

- Ordinary File I/O
 - Reading a file using `read()` results in an `VOP_READ`.
 - For ordinary files, `VOP_READ()` translates to file system specific read, e.g. `ufs_read()`
 - `ufs_read()` calls `segmap_getmap()` to get the data mapping
 - Calls `uiomove()` to transfer data from file to user space
 - Calls `segmap_release()` to free the mapping. This mapping is cached for any subsequent use of the same page

Device access methods

- Direct I/O to block device
 - Reading a block device using `read()` results in an `VOP_READ`.
 - For device files, `VOP_READ` results in `spec_read()`
 - `spec_read()` behaves almost similar to `ufs_read()`
- Direct I/O to block device using `mmap()`ed I/O
 - Not having a page in memory causes `segvn_fault()`, which would invoke `VOP_GETPAGE`
 - `VOP_GETPAGE` calls `spec_getpage()`
 - `spec_getpage()` calls `d_strategy()` of the device

Raw I/O to block device

- Regular read()/write() calls copy the data twice
 - Once between user process & kernel and then to device
- Raw I/O to device is handled by character interface
 - Do raw i/o using character switch entry of the device
 - Result in calling the d_read() or d_write() (which calls physiock() of kernel)
 - Validates I/O parameters, allocates buf struct
 - Calls as_fault() to cause fault and lock the pages
 - Calls d_strategy() of the device and sleep until I/O completes
 - Unlock user pages and return results

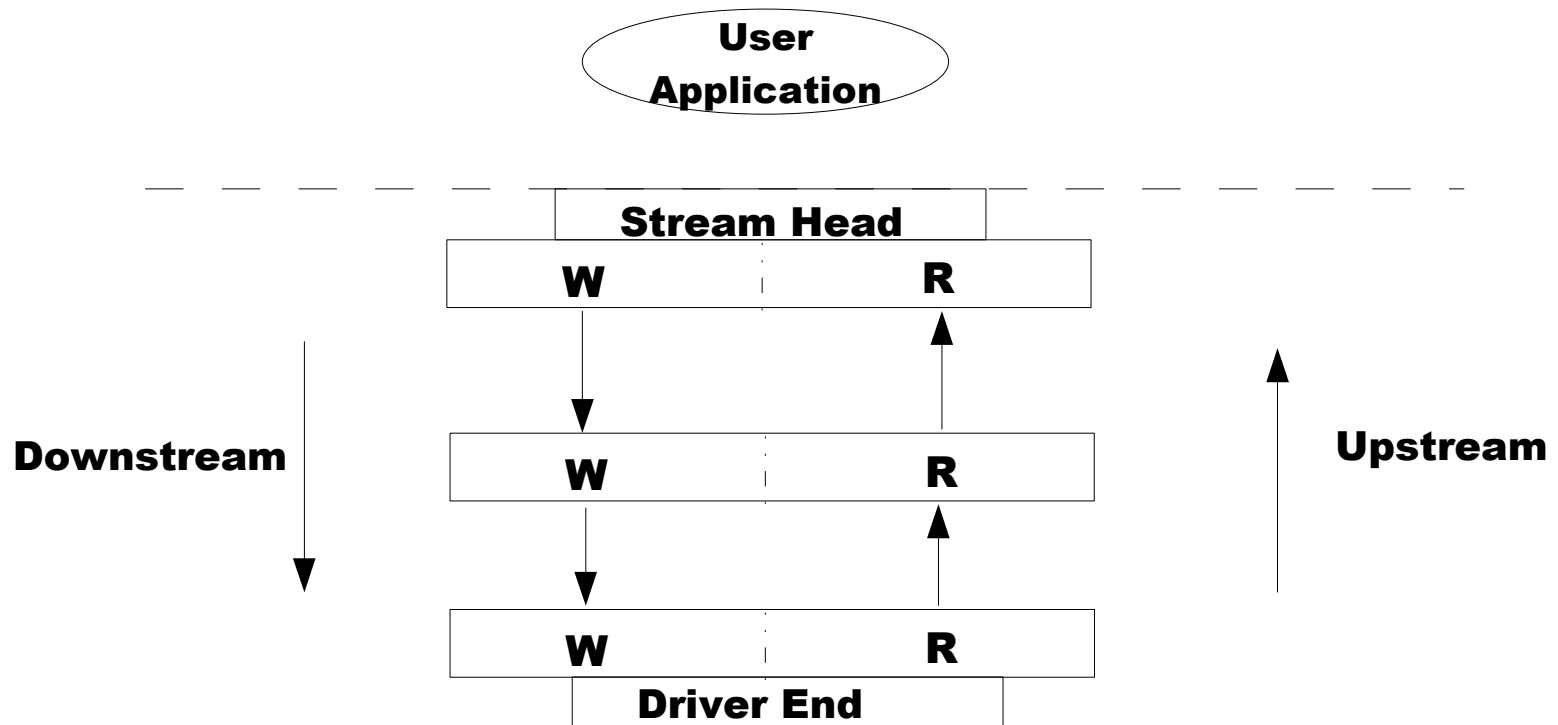
Streams

- Classic Character Drivers have limitations
 - Different vendors (drivers) may replicate the same code, resulting in large size kernels
 - Not buffered and hence inefficient for modern character devices (network interfaces)
 - Limited facilities to applications
- Streams address many of these issues
 - Consist of multiple modules, which can be shared among multiple devices
 - Can provide application level features
 - Provide buffering abilities

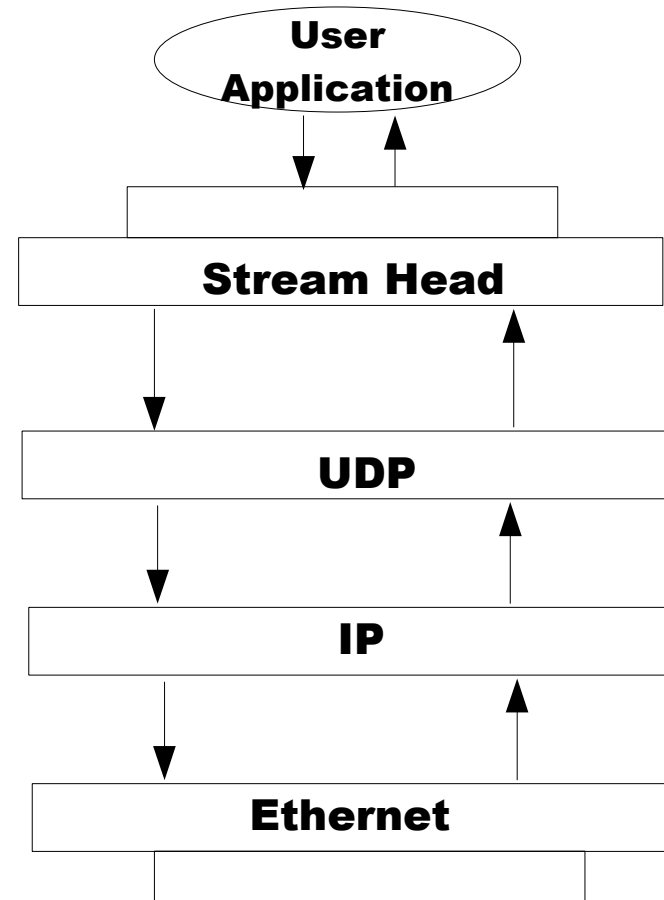
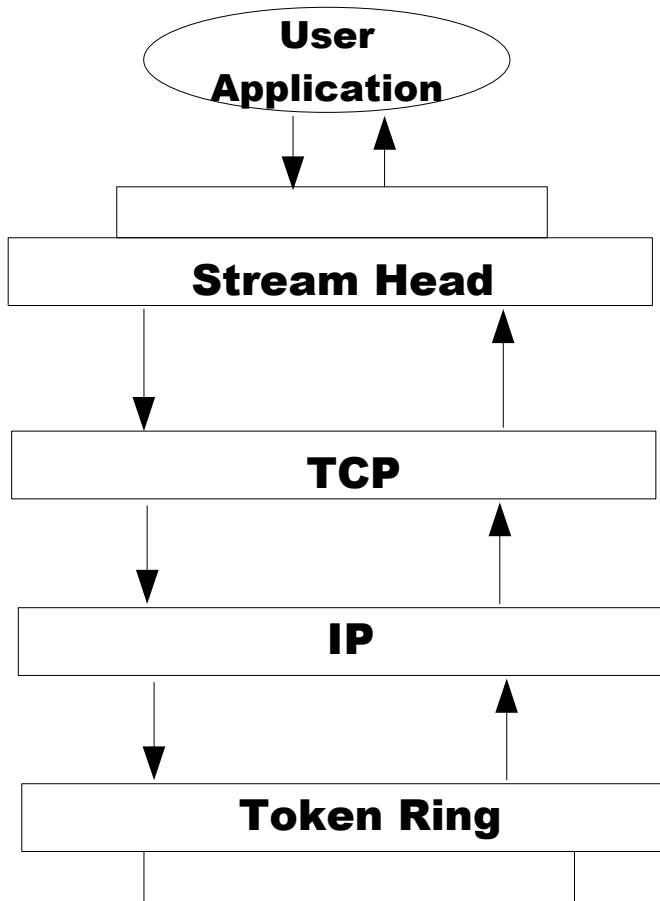
Streams Overview

- Stream Head
 - Interaction with user applications, part of kernel space
- Modules
 - One or more components of the stream
 - Reusable across multiple streams
 - Each module contains a pair of queues
 - Write Queue & Read Queue
- Driver End
 - Interface to the hardware
- Upstream & Downstream
 - Combination of Read Queues & Write Queues of ALL modules

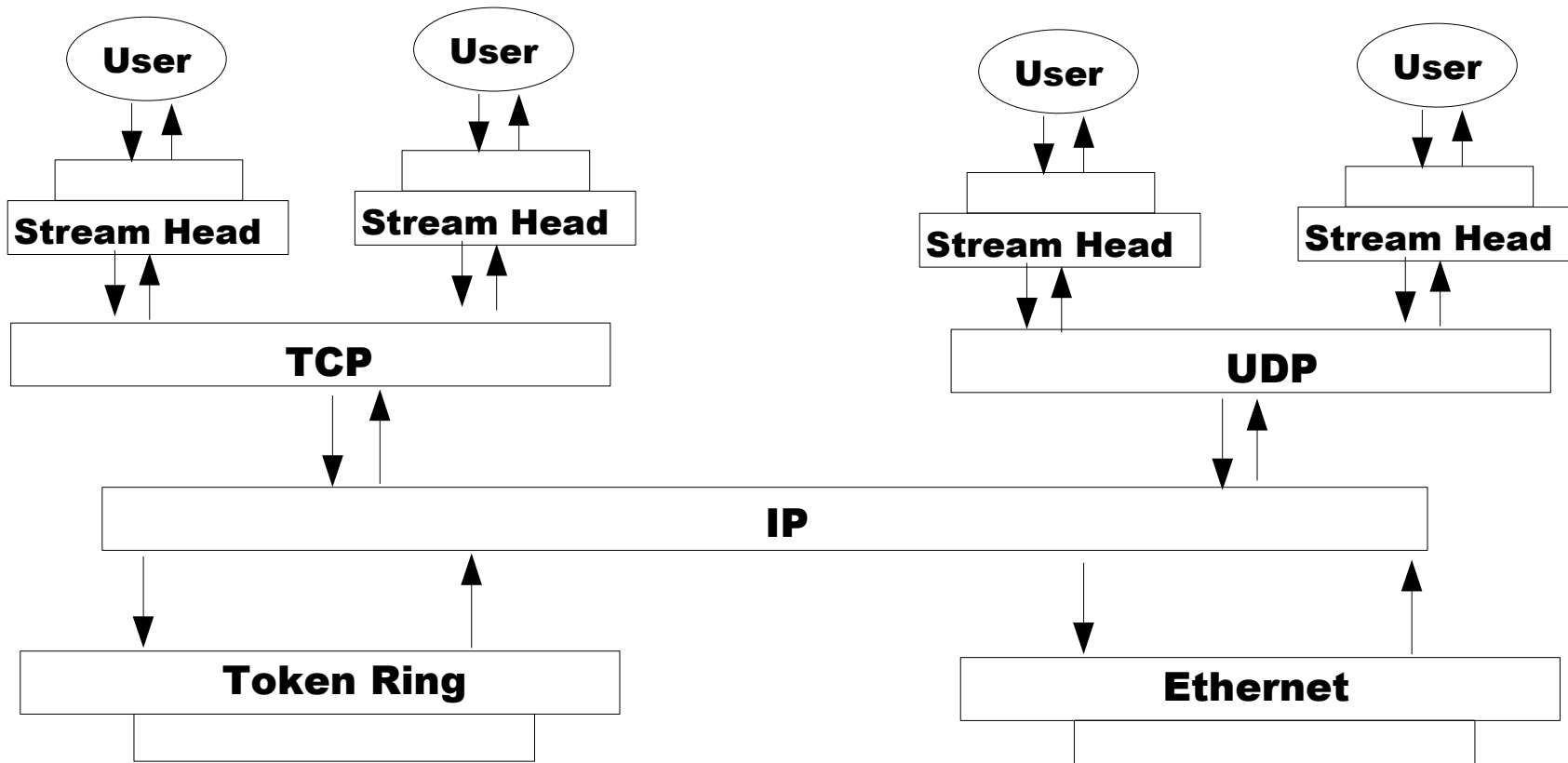
Streams Overview



Reusable Modules



Reusable Modules



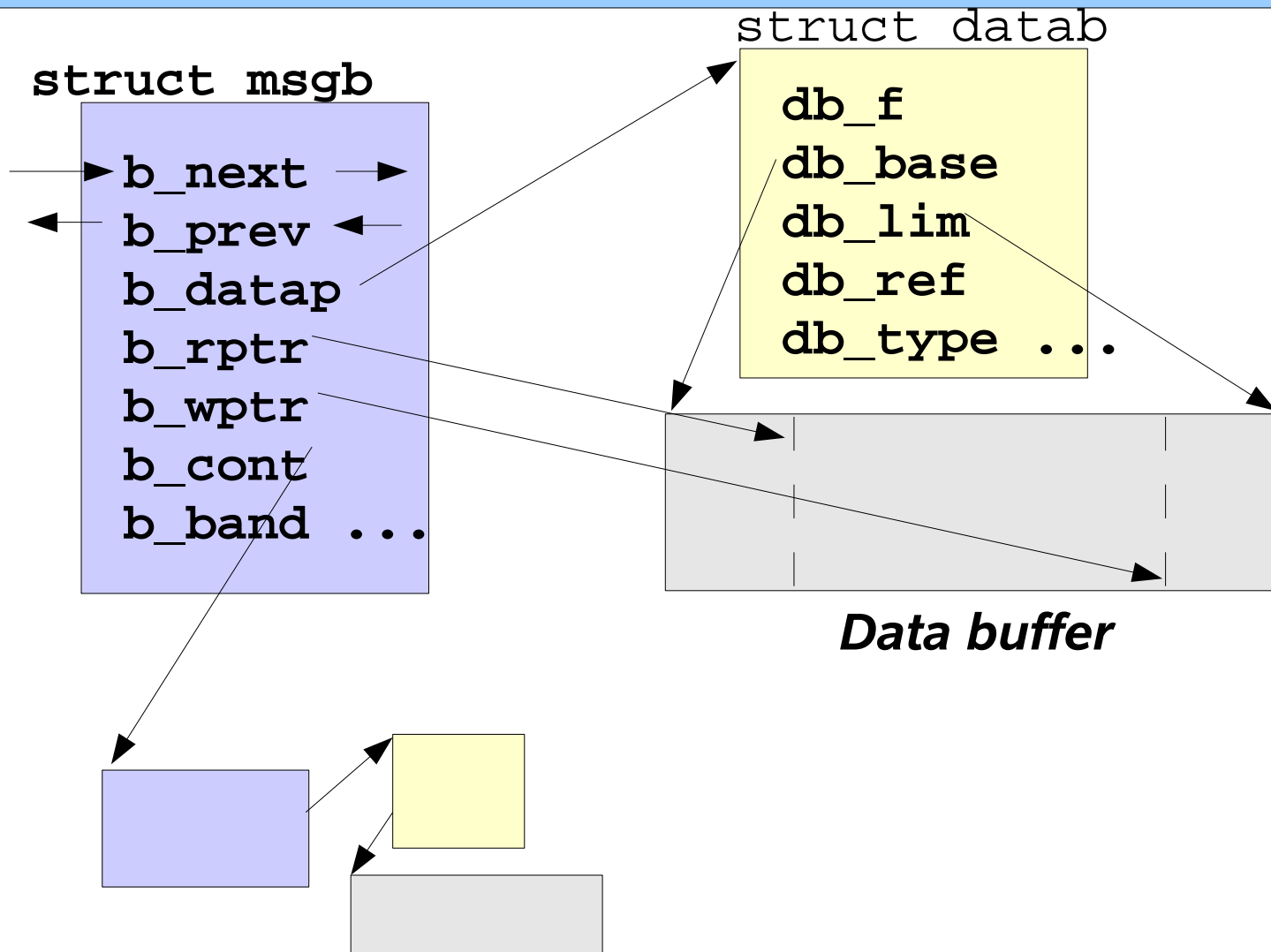
Streams Overview

- Multiplexing Modules/Drivers
 - Drivers/Modules that can connect to more than one Driver/Module at the top or bottom
 - Fan-in (Upper) multiplexer
 - One that can connect to more than one modules above it
 - Fan-out (Lower) multiplexer
 - One that can connect to more than one modules below it
- Messages & Queues
 - Passing messages is the only form of communication
 - Messages are processed thru queues at each module

Messages

- Simplest Message
 - struct msgb
 - struct datab
 - Data buffer
- Multipart messages
 - Several of above triplets
 - Useful in layered protocol implementations where each layer adds/removes a message triplet
- Virtual copying
 - Struct datab has a reference count field (db_ref) and this struct can be used by multiple msgb structures

Messages



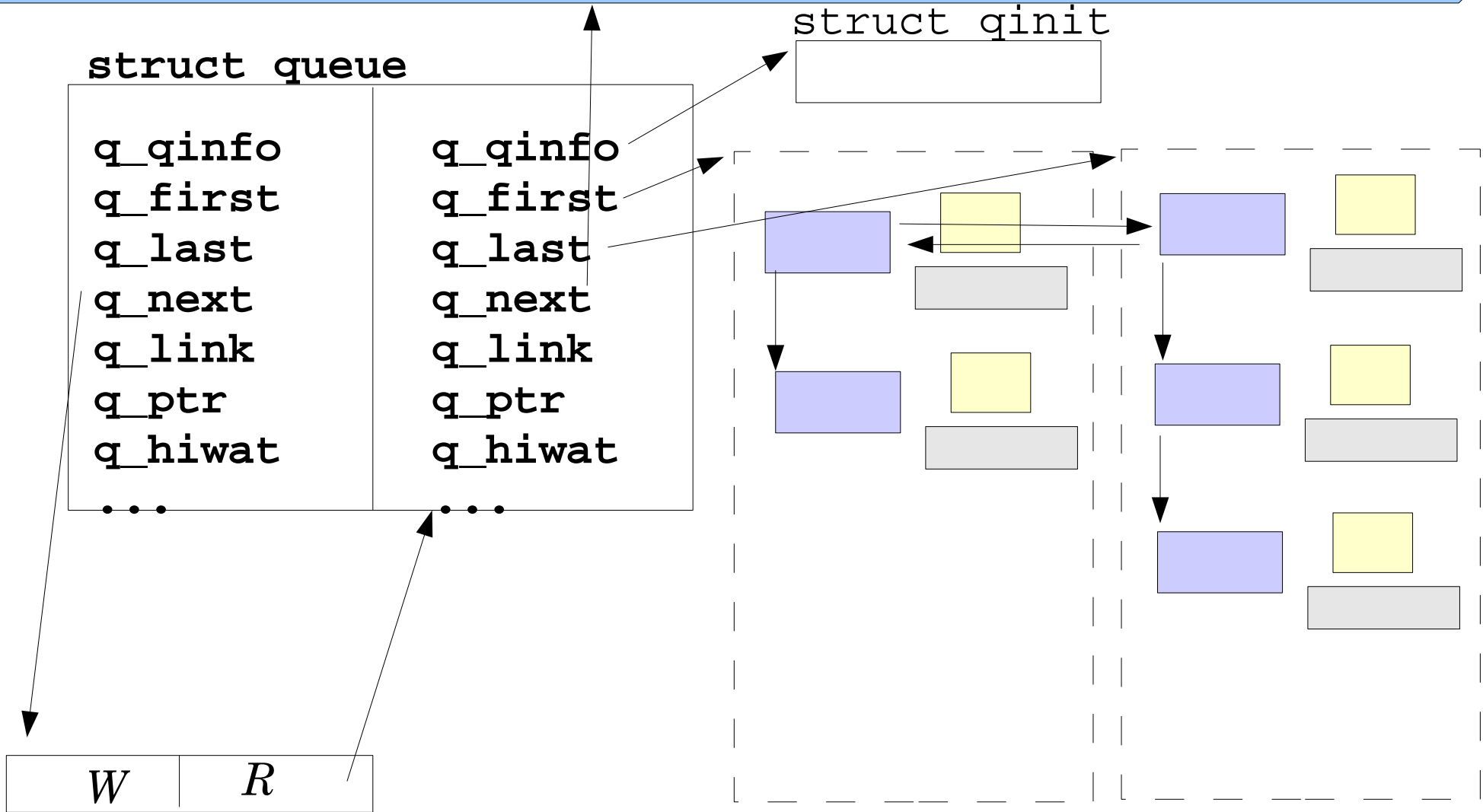
Messages

- Message Types
 - Some types are for upstream, some are for downstream
 - For full list, refer to p553-554 of “Unix Internals”

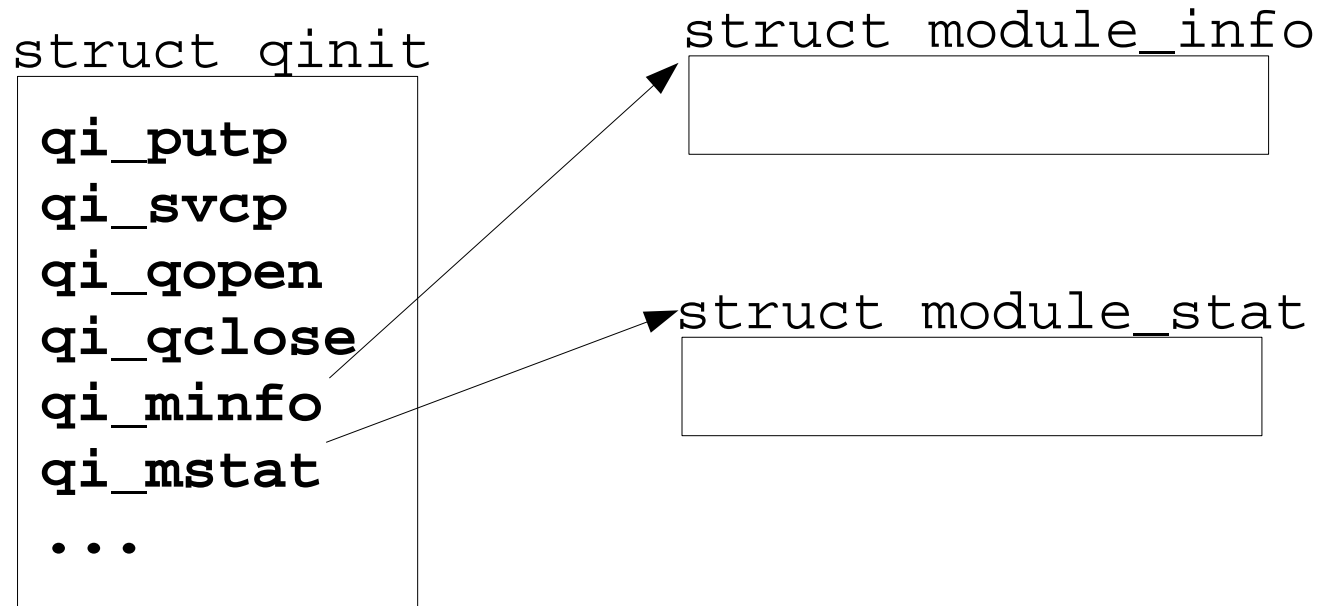
Queues

- Each module has two queues
- Each queue consists of zero or more messages lined up (message queue) for processing
- Struct `qiinit`
 - Has a set of methods & pointers (`qi_putp`, `qi_srvp`, `qi_qopen`, `qi_qclose`, `qi_mstat`, `qi_minfo`)
 - Open and close are called by processes synchronously
 - Put processes the message immediately, when possible. Else, adds message to message queue
 - Service method handles messages in message queue (delayed processing)

Queues



Queues



Stream I/O

- User does a write/putmsg system call for writing to device
 - Stream head allocates a message and copies data to it
 - Sends it downstream to the next queue
 - Eventually, data reaches driver
- Queues pass message to the next one using putnext()
 - Identify the next queue using q_next
 - Results in invoking put procedure of the next queue

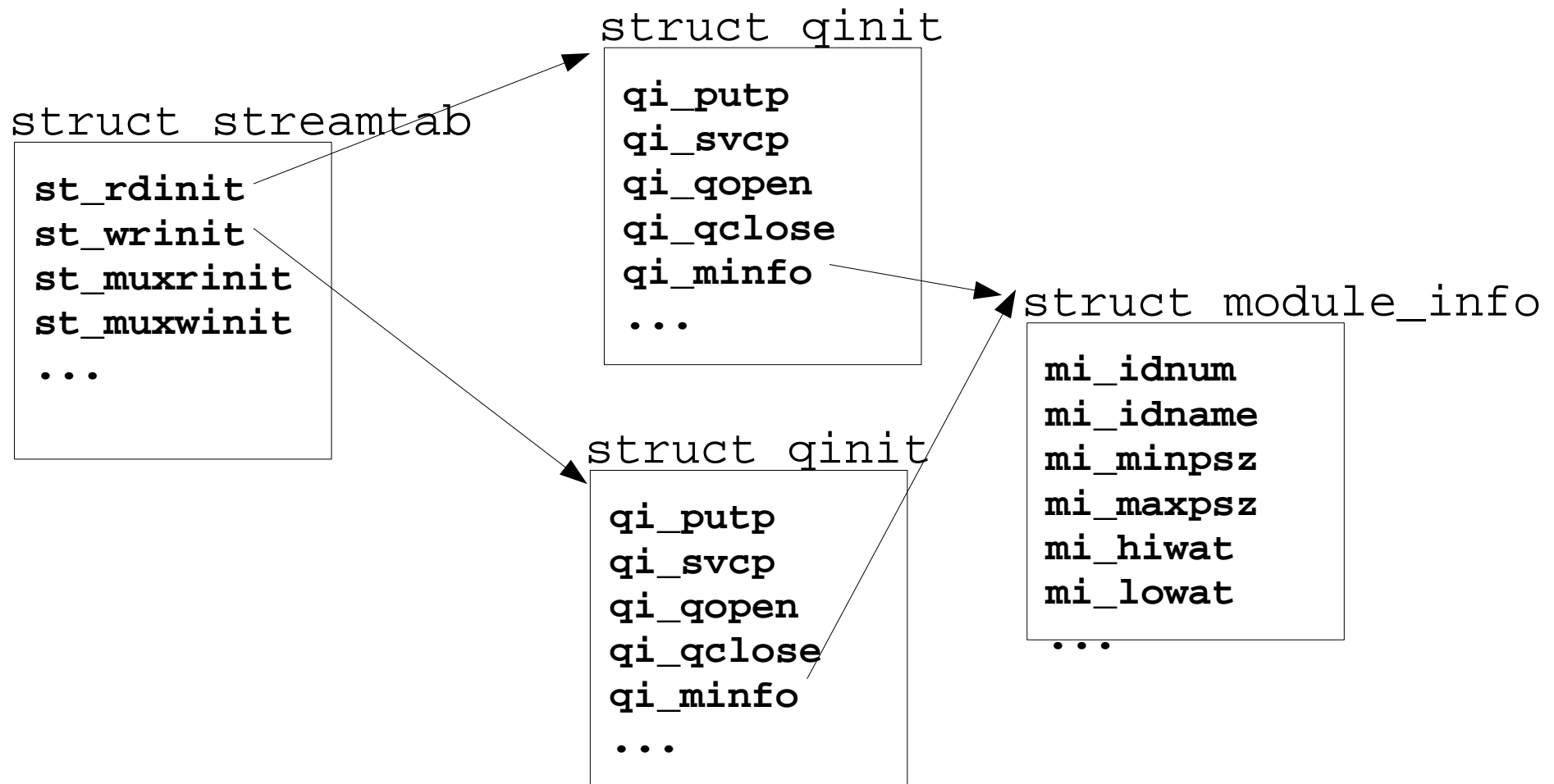
Stream I/O

- Nature of procedures
 - Put and service procedures are non-blocking
 - Both of them keep the messages in queue if processing cannot be done rightaway
 - Need own memory allocation procedures that are non-blocking
 - allocb()
 - bufcall()
 - Service procedures are scheduled in system context, not the process context

Configuration and Setup

- Each module has three configuration structures
 - streamtab, qinit, module_info
 - Streamtab contains two pointers to qinit structures
 - Qinit structures point to module_info
 - module_info contains default parameters of the module, which are copied to the queue structures upon opening the module.
 - These parameters in queue structures can be overwritten later by ioctl() calls

Structs for Configuring a Module/Driver

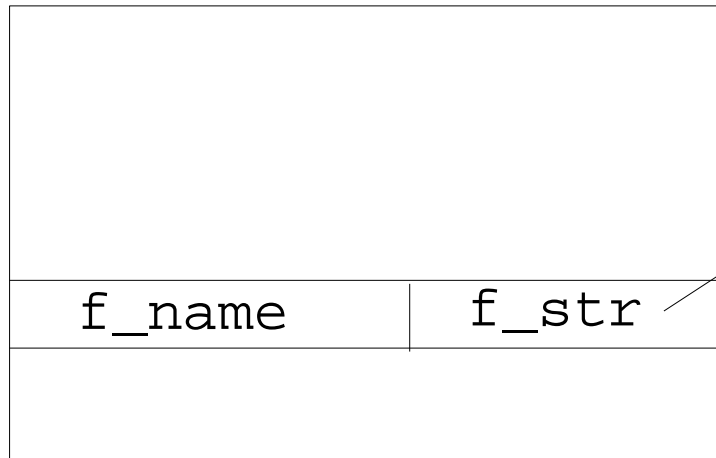


Configuration and Setup

- Streams modules
 - Managed by fmodsw[] switch
 - Identified by f_name (mi_idname of the module_info)
 - f_str pointer points to related streamtab
- Streams drivers
 - The d_str pointer in cdevsw is NOT NULL and points to the streamtab structure
 - Configuration includes
 - Create appropriate device files
 - Use right device numbers

Configuring a Module/Driver

fmodsw[]



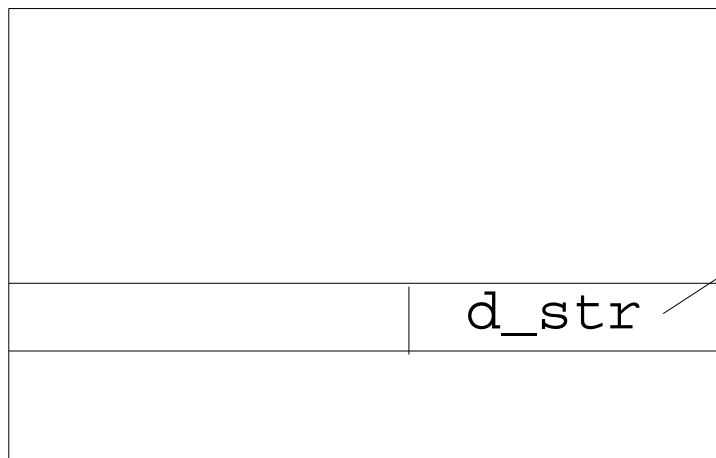
Module

streamtab

qinit
(read)

qinit
(write)

cdevsw[]



Driver

streamtab

qinit
(read)

qinit
(write)