Summer Report 1999

Linux Network

Network Architecture Lab
Electrical and Computer Engineering
University of Toronto

Michael Feng
Roy Leung
Andrew Do-Sung Jun

Contents

Chapter 1: Introduction 1

Chapter 2: Convention used in this book 2

Chapter 3: GNU Zebra Daemon 3

- *3.1: Overview* 3
- *3.2: Scheduling in Zebra* 6
  - *3.2.1: Scheduling Basis* 6
  - *3.2.2: Scheduling Scheme and* `thread_fetch` 7
- *3.3: Routing Table* 8
- *3.4: Zebra Protocol* 9
- *3.5: Zebra-Kernel Protocol* 9
- *3.6: Virtual Terminal Interface (VTY)* 10
- *3.7: Thread Master* 10
- *3.8: Thread Life Cycle* 11
- *3.9: Zebra daemon initialization* 13
- *3.10: Important threads running in Zebra* 26
- *3.11: Current status* 31
- *3.12: Supported platforms* 31

Chapter 4: GateD 33

- *4.1: Introduction* 33
- *4.2: Protocol Scheduling* 33
- *4.3: Task* 34
- *4.4: Timer* 35
- *4.5: Job* 35
- *4.6: Memory Management* 36
- *4.7: Routing Table* 36
- *4.8: Interfaces* 37

Chapter 5: Linux Kernel 38

5.1: Processes and tasks 38

5.2: Important data structure 40

5.2.1: The task structure 40

5.2.2: Process relationships 42

5.2.3: Memory Management 43

5.2.4: Process ID 45

5.2.5: Files 46

5.2.6: Timing 48

5.2.7: Inter-process communication 49

5.2.8: Miscellaneous 50

5.2.9: The process table 53

5.2.10: Files and inodes 53

5.2.11: Dynamic memory management 55

5.2.12: Queues and semaphores 56

5.2.13: System time and timers 58

5.3: Main algorithms 60

5.3.1: Signals 60

5.3.2: Interrupts 62

5.3.3: Booting the system 64

5.3.4: Timer interrupt 66

5.3.5: The scheduler 70

5.4: Implementing system calls 73

5.4.1: How do system calls actually work 73

Chapter 6: Network Implementation 76

6.1: Background 76

6.2: Receiving Packets 78

6.2.1: Interrupt Handling 78

6.2.2: Bottom Half Handler 79

6.2.3: Protocol Layer Handling 79

6.2.4: Socket Layer Handling 80

6.2.5: Process Handling 83

6.3: Transmitting packets 83
6.3.1: Process Layer 83
6.3.2: Socket Layer 84
6.4: Making a Connection on an INET BSD Socket 85
6.5: Listening on an INET BSD Socket 87
6.6: Accepting Connection Requests 87
6.7: Protocol Layer 88
6.8: Device Layer 92
Chapter 7: Linux Traffic Control 95
7.1: Queuing Discipline 95
7.1.1: `noop_qdisc`, `noqueue_qdisc`: a special kind of queuing discipline. 99
7.1.2: Queuing Discipline operations 99
7.1.3: Examples 107
7.2: Classes 115
7.3: Filters 120
7.3.1: Types of Filters 121
7.3.2: Creating a new filter node 127
7.4: `tc`, rtnetlink, netlink, and kernel implementation overview 137
7.4.1: Adding a new queuing discipline using `tc` 139
7.5: Modifying queuing discipline inside the kernel 145
Chapter 8: Linux Management Information Base (MIB) 150
8.1: Statistics Structures 150
8.2: Usage 152
8.3: Control Structures 153
Appendix A 158
Appendix B 159
Appendix C 160

Chapter 1

Introduction

In the summer of 1999, our group investigated on Linux's support in quality of service and we also looked at different fields in this area that includes router softwares, Linux network architecture, traffic control, and MIB (Management information base). This document discusses application approaches and kernel techniques that support the above features and they are divided into the following categories:

1. GNU Zebra and Merit GateD.
2. Linux kernel network implementation
3. Linux kernel traffic control
4. Management Information Base

Chapter 2

Conventions used in this Book

The following is a list of the typographical conventions used in this document.

Commands, data structures or fields within data structures are identified by the following font: `variable`.

Functions are identified by the following font: *`function`*.

Chapter 3

GNU Zebra Daemon

This chapter gives an overview of GNU Zebra router software. First, a general overview is described. Then, this remaining of the section is devided into two parts: the first part provides the internal structure of Zebra while the second part discusess the Zebra architecture in more detail. These two parts provides technical background to the readers who are interested in Zebra implementation.

3.1 Overview

Zebra is a distributed multi-server multi-threaded routing software and is a free software distributed under GPL (GNU Public License). Traditional routing softwares (such as GateD) are made as one process program that provides all of the routing protocol functionalities as a whole. Zebra is unique in its design in which it has a process (running in background, thus a daemon) for each protocol. Zebra uses multithread technology under multithread supported Unix kernels. However it can be run under non-multithread supported UNIX kernels.[1] Thus Zebra provides flexibility and reliability. Each module can be upgraded independently of the others, allowing for quick upgrades as well as protection from the case of a failure in one protocol affecting the entire system.

These daemons include Zebra daemon (which acts as a central arbiter) and other protocol daemons such as, a RIP daemon (RIPd), OSPF daemon (OSFPd), and a BGP daemon (BGPd). The organization of these daemons is illustrated in figure 3.1.

[1] At this moment thread library which comes with Linux has some problem for using Zebra, so Zebra allows users to optionally choose thread systems between POSIX thread and Zebra's select system call for multiplexing. Zebra's select system call is set to be default in compilation.

Protocol daemons do not communicate with the kernel directly. Instead, Zebra defines its own set of protocol (known as Zebra Protocol, *see* Zebra Protocol in section 3.4) to handle inter-process communication between the Zebra daemon and other daemons, and then, the Zebra daemon is responsible to communicate with the Linux kernel. As a result, Zebra mimics the client-server model in which the protocol daemons are clients and the Zebra daemon is a server that allocates and distributes services and resources (routing table information) from the kernel to its client.

Each daemon has its own routing table. Zebra daemon is responsible for kernel routing table update (*service*) and its redistribution between different protocol (e.g. RIPd). Other daemons are for protocol handling.

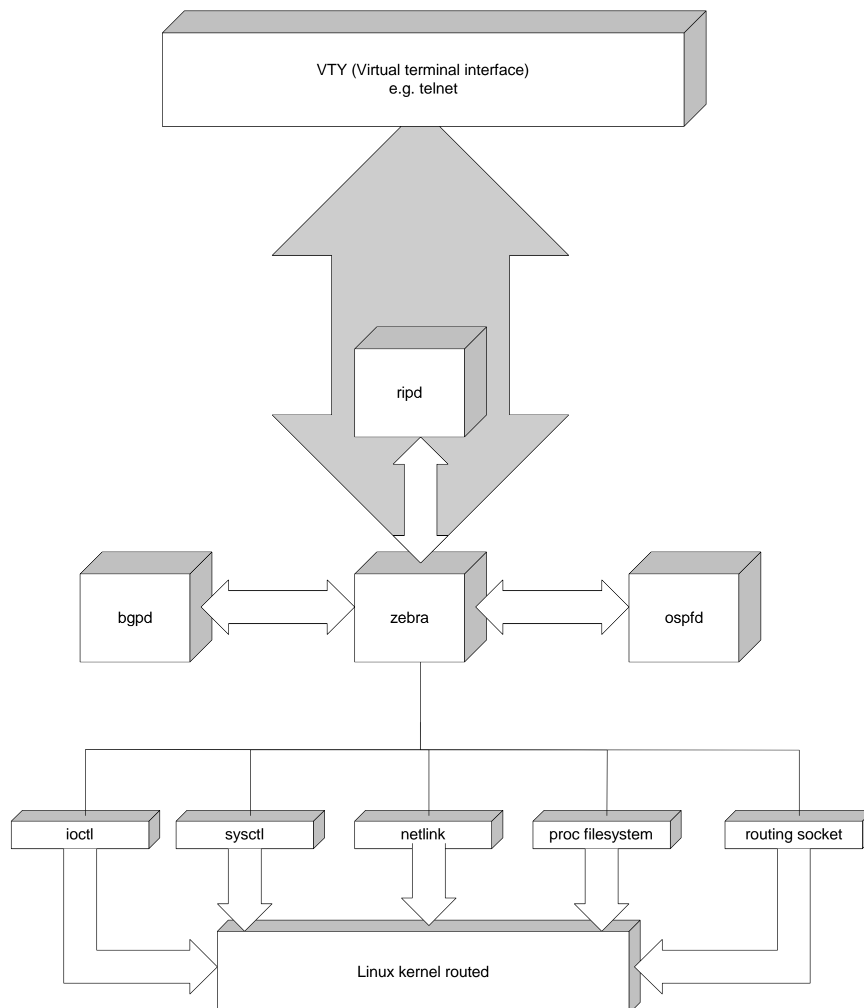


Figure 3.1: the organization of zebra daemons.

Zebra achieves modularity, extensibility, and maintainability in its architecture design. This leads to advantage as well as disadvantages.

Advantage

- *Modularity*: due to the multi-process nature of the Zebra software, it is easily upgraded and maintained. Each protocol can be upgraded separately, leaving the other protocols and the router online. This will save network administrators time in upgrading and maintenance. [4]
- *Speed*: packet routing is carried out at a faster rate than with traditional software. Zebra software allows routers to transfer more data quicker. The need for the ability to transfer large amounts of data quickly is increasing as the internet grows and global networks form. Zebra software will meet that need. [4]
- *Reliability*: in the event of failure of any of the software modules, the router can remain online and the other protocol daemons will continue to operate. The failure can then be diagnosed and corrected without taking the router offline. [4]

Disadvantage

- Inter-process communication (Zebra) v.s. intra-process communication (GateD).
- Kernel routing table updates and redistribution would be slow.

The section that follows contains two parts: the first part contains general information about the Zebra internal organization while the second part concerns more about the low level detail of the Zebra architecture. The second part is only useful if the reader wants to understand Zebra in a low level.

Part 1: Zebra *Internal*

3.2 Scheduling in Zebra

Each daemon in Zebra performs its own scheduling. All daemons use the same scheduling scheme and function.

3.2. 1 Scheduling Basis:

Zebra operation are comprised of a series of function calls, and each function call is stored in a *thread*[2] structure in Zebra sitting inside scheduling queue, waiting for its turn to be dequeued.

The thread structure includes the following information:

- Its ID.
- Its thread type which can be `THREAD_READ`, `THREAD_WRITE`, `THREAD_TIMER`, `THREAD_EVENT`, and `THREAD_UNUSED`.
- A `next` pointer to the next thread.
- A `previous` pointer to the previous thread.
- A pointer to the thread master.
- A pointer to an event function that points to the function which will be inovked at *`thread_calls`*.
- Arguments of the event.
- File description in case of read/write.
- Time value (used by time critical event).

Each daemon contains a thread master that holds important information. Part of this information includes five pointers to scheduling queues pointing to:

1. Event queue (FIFO)
2. Read queue (FIFO)
3. Write queue (FIFO)
4. Timer queue (in the order job service time, earlier comes first)
5. Unuse queue (FIFO)

Although timer queue is used to service packets with real-timer information, Zebra scheduling is non-preemptive and therefore, it cannot guarantees the operation of this kind of *thread* would be served at real-time. Each daemon contains a while loop in its `main` function continuously calls *`thread_fetch`*. *`thread_fetch`* is invoked to dequeue one *thread* from the queues and to call the *thread*'s function by *`thread_call`*.

3.2.2 Scheduling Scheme and `thread_fetch`:

Before *`thread_fetch`* is called inside the daemon, the event queue, read queue, write queue, and timer queue contain *threads* that are waiting to be called. During the call of *`thread_fetch`*, *`thread_fetch`* dequeues a thread at the head of the event queue and this *thread* will be the one that gets executed.[3] If the event queue is empty, all the *threads* in the read queue and write queue are transferred to the event queue. The *threads* in the timer queue also get a chance to transfer to the event queue but unlike the previous case, not all the *threads* move to the event queue. Instead, only the *threads* that should run before the current time are moved to the event queue. For instance, if the current time is 10:00, the *threads* with its timing information earlier or equal to 10:00 get enqueued in the event queue and the remainders will stay behind at the timer queue.

---

[2] This "thread" term is italiced throughout the document to distinguish the thread data structure in Zebra and the normal thread in operating system (that is without italiced).
[3] Execution of a thread structure is not the same as normal execution of a thread in operating system. Instead, it means the function pointed by the thread structure gets executed. The function held by a particular thread structure depends on the nature of the daemon job. (See *Routing Table* for example)

One important point is that the order of *thread* execution in the event queue is not arbitrary. Instead, the *threads* in the read queue have greater privilege than those in the write queue and the timer queue in term of the order of execution; so does the threads in the write queue have a greater execution priority than those in the timer queue. As a result, the read threads get executed earlier than the timer threads.

Another point is that Zebra does not *free* the *threads* whose job has been finished its function execution. Instead, these *threads* are enqueued into the unuse queue and are waiting for reuse which takes place when a daemon requests a new *thread* to hold new job information.

3.3 Routing Table

In RIPd, route updates happen every 30 seconds and a route expires if no update occurs within 180 seconds. This routine is accomplished in Zebra by embedding a timing function (`rip_timer` in Zebra RIPd) in a *thread* and enqueued the *thread* into the timer queue. When this *thread* is fetched, `rip_timer` executes and calls `rip_age_route` to update RIP routes. If the route expires, it will delete the route from the routing table and RIPd will inform Zebra daemon to perform the same action inside the kernel with the use of socket and Zebra protocol. The kernel update is necessary to ensure consistency between the routing tables in Zebra and the kernel.

If there is no modification made to the daemon table, Zebra daemon is responsible for obtaining information from the kernel routing table and updating the daemon table.

Routing table in Zebra is organized as a binary tree, and the location at which a routing entry appears depends on the prefix of that route.

3.4 Zebra Protocol

Zebra provides standard interface for updating kernel routing table and interface look up method. If developers want to write their own routing protocol, Zebra protocol is very useful because there is no need of knowing the detail of OS interface. They only need to send essential information to the Zebra daemon.

Below is the list of Zebra protocol type.

```
ZEBRA_IPV4_ROUTE_ADD
ZEBRA_IPV4_ROUTE_DELETE
ZEBRA_IPV6_ROUTE_ADD
ZEBRA_IPV6_ROUTE_DELETE
ZEBRA_GET_ALL_INTERFACE
ZEBRA_GET_ONE_INTERFACE
ZEBRA_GET_HOSTINFO
```

3.5 Zebra-Kernel Protocol

Zebra supports `ioctl`, `sysctl`, `proc` filsystem, and `netlink`. Appendix A provides details for each protocol.

`Netlink` makes asynchronous communication between kernel and Zebra possible. To begin communication with the kernel, Zebra daemon calls *`kernel_init`* in which it calls *`netlink_socket`* to make a socket for Linux `netlink` interface. After *`netlink_socket`* has been set up, transfer of information between kernel and daemon only involve traditional socket function calls such as *`sento`* and *`receive`*.

3.6 Virtual Terminal Interface (VTY)

Virtual Terminal Interface is command line interface to change and/or view current configuration in Zebra. It allows users to connect to the daemon via telnet protocol. Therefore, it is convenient to make changes to configuration without restarting the daemons.

Part 2: Zebra *Architecture*

3.7 Thread Master

As mentioned in part 1, each daemon in Zebra packet contains a thread master called `master`. It is type of `thread_master` data structure and its structure is shown below.

```
/* Master of threads. */
struct thread_master *master;
```

```
/* Master of the theads. */
struct thread_master
{
  struct thread_list read;
  struct thread_list write;
  struct thread_list timer;
  struct thread_list event;
  struct thread_list unuse;
  fd_set readfd;
  fd_set writefd;
  fd_set exceptfd;
  unsigned long alloc;
#ifdef HAVE_PTHREAD
  pthread_mutex_t lock;
#endif /* HAVE_PTHREAD */
};
```

Without the use of POSIX threading system, the multithreading nature of Zebra requires implementation of scheduling functions on top of the application layer, and `thread_master` is the heart of Zebra scheduling.

The thread master contains five *thread* lists or *thread* queues called `read`, `write`, `timer`, `event`, and `unuse`. The first four queues are used to schedule the order of *thread* execution. `read`, `write`, and `event` are types of FIFO queue while threads in `timer` queue are scheduled in the order of their execution time. The earlier the *threads* need to run, the closer its position to the head of the queue.

The memory spaces held by the *threads* that have been executed their executioin are not freed immediately. Instead, they are queued into `unuse` queue. When process needs to add a new *thread* in the other queues, the `unuse` *thread* are used again without the need to allocate new memory space. This reduces invocation of system calls.

3.8 Thread Life Cycle

The life cycle of a thread is shown in the figure 3.2 and a brief description is outlined below.

1. When one of the daemons wants to create a new *thread*, it may call *`thread_add_read`*, *`thread_add_write`*, or *`thread_add_timer`*, depending on the type of the thread that the function creates.
    - *`thread_add_read`* adds a *thread* to the read queue that is responsible for accepting and reading data from the outside through a socket.
    - *`thread_add_write`* adds a *thread* to the write queue that is responsible for flushing and writing data to the outside through a socket.
    - *`thread_add_timer`* adds a *thread* to the timer queue that is responsible for timing an event such as updating and redistributing table.

2. In all of the calls made above, the function *`thread_new`* is used to allocate memory space for a new *thread*. It first checks if there is any `unuse` *thread* in the `unuse` queue. If there is one, it will use it as the new thread. Otherwise, it will call *`malloc`* to allocate memory space for the new *thread*. The parameters of the *thread* structure

are set to default. Finally, `thread_new` invokes `thread_all_list` to add the new *thread* to the desired thread queue.

3. There is a point at which the Zebra daemon continuously fetches *threads* from the event queue and executes them. Once the *thread* get executed, the *thread's* type is set to `unuse` and transferred to the `unuse` queue.

4. If the event queue is empty, the daemon executes `select` function that monitors "sets" of file descriptors; in particular `readfds`, `writefds`, and `exceptfds`. When `select` returns, `readfds`, `writefds`, and `exceptfds` would have modified to reflect which of the file descriptors selected is ready for reading, writing, and performing exception. Then, the program can test if the file descriptor of a *thread* is ready, by using `FD_ISSET`.

5. After performing `select`, all the *threads* inside the read queue that have their `readfds` flag ready, are moved to event queue. All the *threads* inside write queue that has its `writefds` flag ready, are moved to the event queue. Other *threads* that

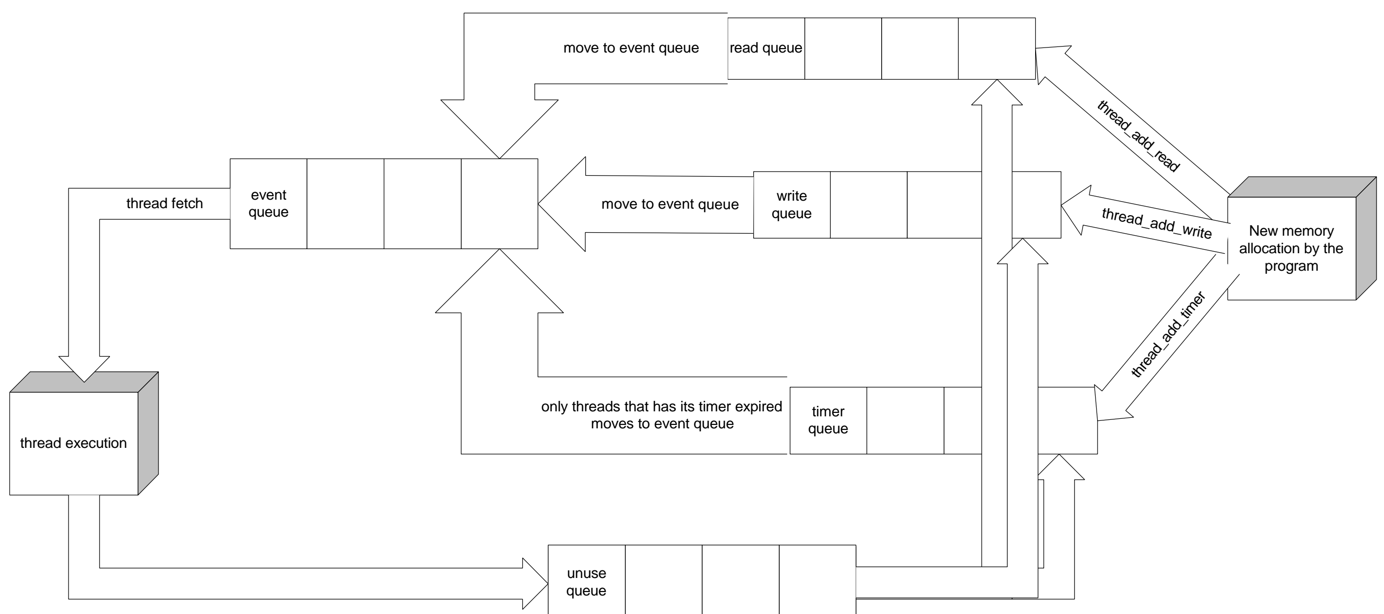


Figure 3.2: Life Cycle of *threads*.

are not ready for I/O remain in their own queues. Once the *threads* in the read and write queue are moved to the event queue, their `readfds` and `writefds` flags are cleared. `exceptfds` are not used in Zebra.

6. Only timer *threads* that have their timers expired relative to current clock time are moved to the event queue.

7. Once this is done, the *thread* at the head of the event queue is fetched and gets executed if there is any. If the head of the event queue is empty, repeat step 3 to 5 indefinitely.

3.9 Zebra Daemon Initizatation

This section introduces the execution sequence of Zebra daemon. This introduction also discusses some of the data structure in Zebra daemon and provides brief explanation of their uses. Figure 3.5 provides a flow chart to illustrate the Zebra daemon execution sequence.

The following outlines the execution sequence of Zebra daemon.

1. Call *`openzlog`* to allocate `ZLOG` *socket* and open the socket via the function call *`opensock`*.
   - This opens a log file to report information during the daemon execution, e.g. I/O error.

2. Capture option parameter by *`get_opt_long`* function call and set mode or option variable appropriately.

   Mode and option include:

`'-b'` `'—b'`	Runs in batch mode. 'zebra' parse configuration file and terminate immediately.

`-r'` `` `—r' ``	When progrm terminates, retain added route by zebra.
`` `d' `` `` `—d' ``	Runs in daemon mode.
`` `f FILE' `` `` `—config_file=FILE' ``	Set configuration file name.
`` `-l' `` `` `—log_mode' ``	Set verbose log mode flag.
`` `-P PORT' ``	Set vty's port number.

`` `—vty_port=PORT' ``
`` `-v' ``
`` `—version' ``
	Print program version.

`` `-h' `` `` `—help' ``	Display this help and exit.

3. Call *`log_init`* to set global variable `logfp` to `stdout`.

4. Make master thread emulator by allocating and initializing memory space for global variable called `master`.

5. Call *`signal_init`* which
   - Set signal option so that whenever the program receives interrupt signal `SIGINT`, it will execute *`sigint`*.
   - Set signal option so that whenever the program receives termination signal `SIGTERM`, the program will execute *`sigint`*.

- Set signal option so that whenever the program receives ‘broken pipe’ signal `SIGPIPE`, the program will execute `SIG_IGN`.

  About `sigint`

  - *`signit`* calls `zlog()` to print out current time information and string “terminating on signal” on the zebra log file.
  - If `retain_mode` is disabled, *`rib_close`* will be called to delete all added route and to close RIB.

6. Call *`cmdinit`* to

- Initialize global host variable (known as `host`)’s information including `name`, `password`, `enable`, `logfile`, `config`, and `lines`.
- Allocate memory space and initialize global command vector `cmdvec`. (The default size is the size of a vector with only 1 index.) Statistic for vector and vector data allocations are updated (each increment by 1). The data structure of `cmdvec` is shown on the figure 3.3.

- Install command nodes of the global command vector `cmdvec` including `view` node, `enable` node, `auth` node, `auth enable` node, and `config` node by calling *`install_node`* command.

- Also install node’s basic command element including
  - `config_enable_cmd,`
  - `config_exit_cmd,`
  - `config_help_cmd,`
  - `show_version_cmd,`
  - `config_terminal_cmd,`
  - `config_exit_cmd,`
  - `config_help_cmd,`
  - `config_list_cmd,`

- show_running_config_cmd,
- config_write_file_cmd,
- config_write_memory_cmd,
- copy_runningconfig_startupconfig_cmd,
- show_version_cmd,
- config_end_cmd,
- config_exit_cmd,
- config_help_cmd,
- config_list_cmd,
- hostname_cmd,
- no_hostname_cmd,
- password_cmd,
- enable_password_cmd,

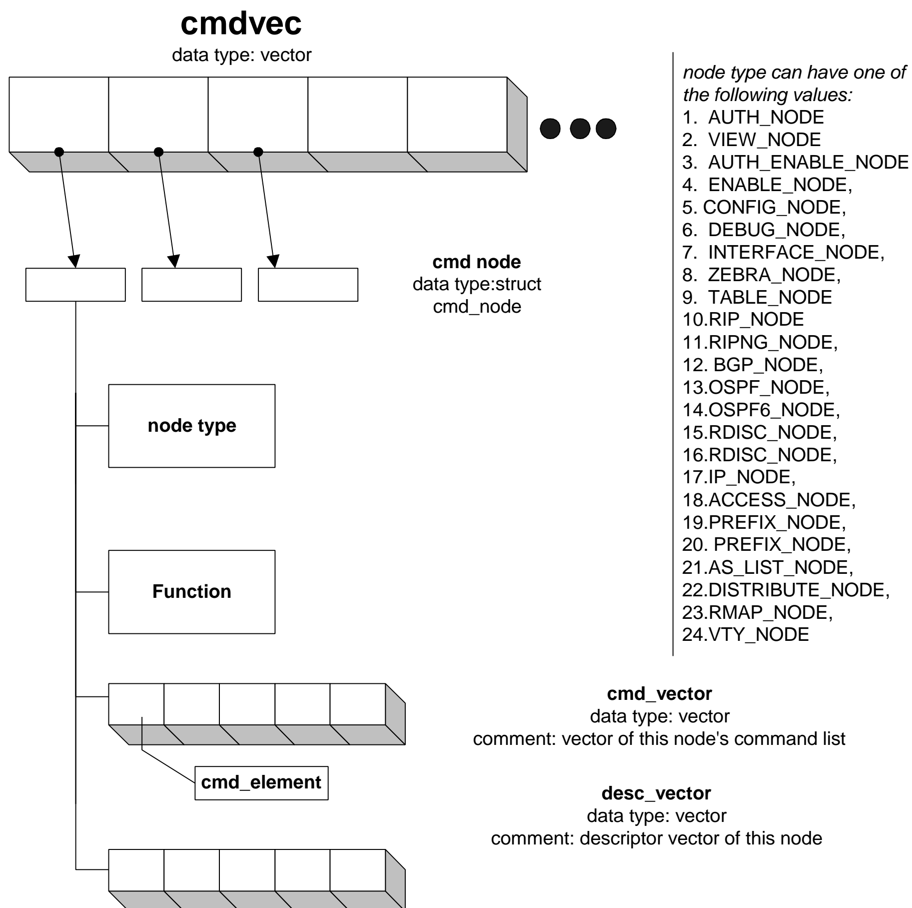


Figure 3.3: cmdvec data structure.

- config_lines_cmd,
- config_log_cmd,
- config_log_file_cmd,
- service_password_encrypt_cmd,
- no_service_password_encrypt_cmd

by calling install_element().

- Set random number generator (*rand*) seed to current time by calling *srand*(*time*(NULL)).

7. Call *vtyinit* to
   - Allocate memory space and initialize global vty vector vtyvec. (The default size is the size of a vector with only 1 index.) Statistic for vector (vector_alloc) and vector data (vector_data_alloc) allocations are updated (each incremented by 1).

   - Install bgp top command node named vty_node by calling *install_node*
   - Install command elements of the command nodes by calling *install_element*. The new elements include:
     - config_who_cmd *for* view_node,
     - configwho_cmd *for* enable_node,
     - line_vty_cmd,
     - service_advanced_vty_cmd,
     - no_service_advanced_vty_cmd,
     - config_end_cmd,
     - config_exit_cmd
     - config_help_cmd,
     - exec_timeout_cmd,
     - vty_access_class_cmd,

- no_vty_access_class_cmd.

8. Call *memory_init* to
   - Install command elements including
     - show_memory_cmd *for* view_node,
     - show_memory_cmd *for* enable_node .

9. Call *zebra_init* which
   - Initializes the global list variable client_list. (update statistic bookkeeping for list: list_alloc)
   - Make zebra socket for communication with other daemons such as ripd.
     - Create a socket called accept_sock .
     - Create an address structure with zebra port number 2600 and bind it to the socket.
     - Call *zebra_event* to add a read thread whose function sets to *zebra_accept*.

10. Call *rib_init* to
    - Allocate and initialize the global route table ipv4_rib_table by calling *route_table_init*. The structure of a routing table is shown on the figure 3.4.
    - Install command elements including
      - show_ip_cmd *for* view_node.
      - show_ip_cmd *for* enable_node.

      If HAVE_IPV6 is defined in 'making' zebra. It will also install
      - show_ipv6_cmd *for* view_node.
      - show ipv6_cmd *for* enable_node.

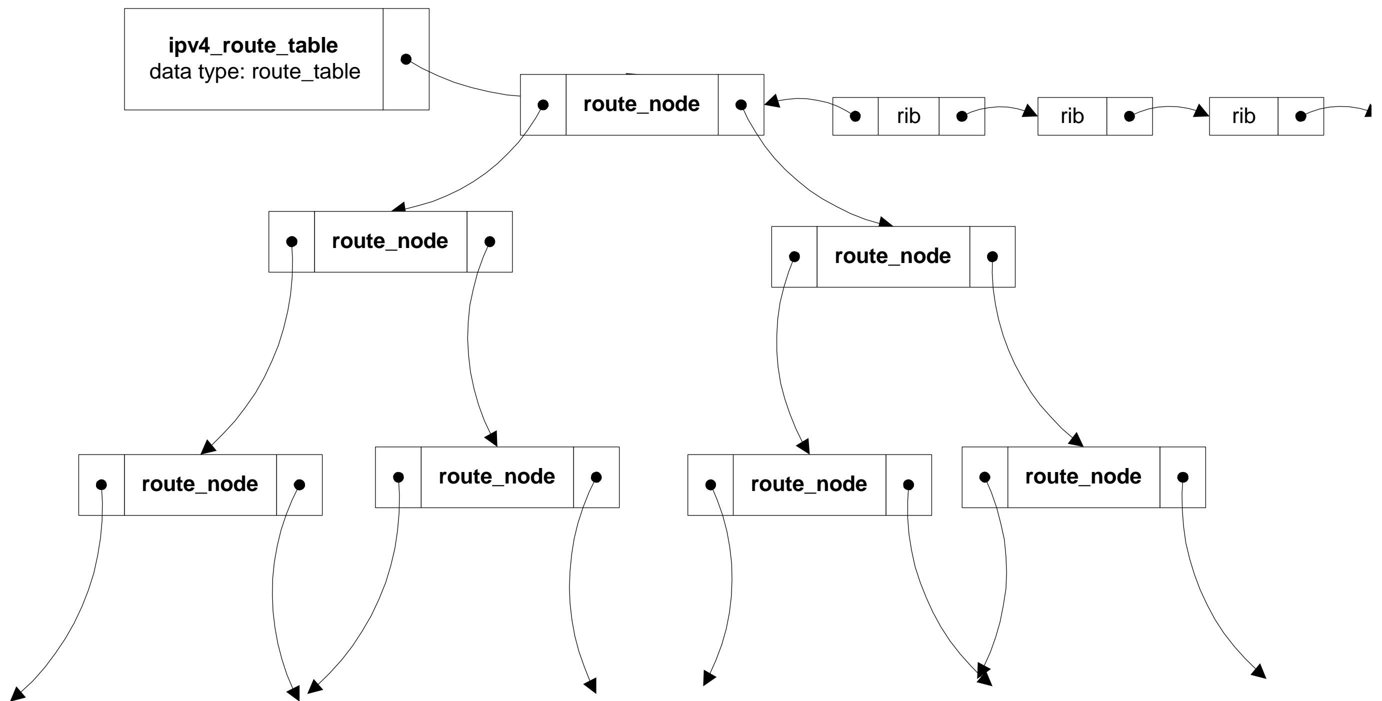


Figure 3.4: route table data structures

11. Call *zebra_if_init* to initialize zebra interface.
    - Call *if_init* to initialize `if_list`, the global interface list.
    - Set interface master called `if_master` 's `if_new_hook` to `if_zebra_new_hook`.
    - Set interface master called `if_master` 's `if_delete_hook` to `if_zebra_delete_hook`.
    - Install command node `interface_node` by calling *install_node*.
    - Install elements by calling *install_element* that includes
        - `show_interface_cmd` *for* `view_node`,
        - `show_interface_cmd` *for* `enable_node`,
        - `interface_cmd`,
        - `config_end_cmd`,
        - `config_exit_cmd`,
        - `config_help_cmd`,
        - `interface_desc_cmd`,

- no_interface_desc_cmd,
- mulicast_cmd,
- no_multicast_cmd,
- shutdonw_if_cmd,
- no_shutdonw_if_cmd,
- ip_address_cmd,
- no_ip_address_cmd.

12. Call *access_list_init* to install access node by *install_node* call.
    - Install vty related elements such as
        - access_list_cmd
        - no_access_list_cmd

13. Call *kernel_init* to make one of the three kind for kernel socket, including

    1. socket

    - Call *routing_socket* to create kernel routing update socket called routing. The socket family type is AF_ROUTE, and its type is SOCK_RAW to access internal interface. (Super user access required)

    2. Netlink

    - Create socket interface to kernel called netlink whose family is AF_NETLINK and type is SOCK_RAW (super user required).
    - The address structure is sockaddr_nl whose family is AF_NETLINK, its group is set to 0, and nl_groups is RTMGRP_IPV6_ROUTE + RTMGRP_IPV4_ROUTE.
    - Bind this address structure to the new socket.
    - call *thread_add_read* to add a thread to the read queue with function, with function call *kernel_read* and fd set to netlink.sock

    3. icotl

    - dummy function which does nothing upon its call.

Note: depending on which interface used in Zebra, different global variables are used. For example, if netlink is used, the global variable netlink is defined. If socket is used, routing is defined.

14. Call *interface_list*. Depending on what interface is used, different set of function is called:

    1. *interface_list* for sysctl, which is a interface listing up function.
        - Fetch interface information into allocated buffer by sysctl.
        - Parse both interfaces and addresses.
        - Free sysctl buffer.
    2. interface_list for netlink, which in turn calls *intreface_lookup_netlink*.
        - Get interface information by calling *netlink_request* and *netlink_parse_info*.
        - Get IPv4 address of the interfaces by calling *netlink_request* and *netlink_parse_info*.
        - If ipv6 is defined, it will get Ipv6 address of the interfaces by calling *netlink_request* and *netlink_parse_info*.
    3. *interface_list* does nothing, it's only a dummy interface.

15. Call *route_read*

For netlink

- Call *netlink_route_read*
- Call *netlink_request* and *netlink_parse_info(netlink_routing_table)*

- *netlink_request* send netlink socket a request message for routing table.
- *netlink_parse_info* waits to receive message from netlink for response of previous request. If the response is received collectively, call *netlink_routing_table* to look up routing table. Inside *netlink_routing_table*, it may call *rib_add_ipv4*.

- *rib_add_ipv4* adds prefix into rib. If there is a same type prefix, then we assume it as implicit replacement for the route.

For /proc filesystem

- call *proc_route_read* to
  - Open /proc filesystem.
  - Call *fgets* and *sscanf* to get the needed information.
- Inside *proc_route_read*, it may call *rib_add_ipv4*.
- It will call to *proc_ipv6_route_read* support ipv6.

For sysctl, it read routing table information by calling *sysctl*.

16. Call *hostinfo_get* to
  - host information logging.
  - set global host information hinfo 's ipforward to the output of ipforward which includes
    - open a datagram socket
  - If ipv6 is defined, it sets hinfo's ipv6 to 1 and calls *ipforward_ipv6* to set hinfo's ipv6forward. Otherwise, it sets it to 0.

17. Call *sort_node* to sort each node's command element according to command string.

18. Call *vty_read_config* which
  - Test where configuration file is (it actually chooses from three choices config_file, config_current, and config_default).
    - config_file is the file location and name that was given as a parameter of the zebra daemon execution.
    - config_current is the default file name in the current directory with the same directory the daemon was started.
    - config_default was the default directory and file name.

- Execute *vty_read_file* which read up configuration from file name
- Execute *config_from_file* which in term execute *cmd_execute_command_strict* that runs command if there is a match. The function get executed is the function that is pointed by the function pointer of the command element in cmdvec.
- Set the global config_file to the one that has been read successfully: config_file, config_current, or config_default.

19. Calls up *rib_weed_tables* which in turn calls *rib_weed_table* with ipv4_rib_table to clean up table and to delete all routes from unmanaged tables.
    - Transverse the rib tables and check each rib's table parameter. If the 'table' parameter is not equal to rtm_table_default or not equal to RT_TABLE_MAIN, then delete that rib entry and free it.

    If ipv6 defined, repeat the same procedure with ipv6_rib_table.

20. If batch_mode was set, exit the program.

21. Otherwise, call daemon to daemonize itself.

22. Call *vty_serv_sock* to
    - Make a new socket called accept_sock by calling *sockunion_stream_socket*.
    - Call *sockopt_reuseaddr* to allow reuse of sever socket.
    - Call *sockunion_bind* to bind socket to universal address and given port which is 2601 (ZEBRA_VTY_PORT by default).
    - Call *listen* to socket under queue 3.
    - Add vty sever event by calling *vty_event* with parameter set to VTY_SERV which in term calls *thread_add_read* with function parameter set to *vty_accept* to add thread in read queue.

23. while loop encountered which indefinitely call *thread_fetch* which

- Check event queue first. If there is an event, dequeue it by calling *thread_trim_head*. Move the thread from event queue to unuse queue. Return the thread for *thread_call*.
- If no thread found in event queue, then move to retry region.
- Compute time_min which is the difference between the current clock time and the time stored in the sand time parameter of the thread at the head of timer queue. time_wait = time_min.
- Call select to monitor sets of file descriptors within time_wait amount of time. When select returns, readfds, writefds, exceptfds will be modified to reflect which of the file descriptors you selected is ready for reading, waiting, and exception. You can test them with the macro *FD_ISSET*.
- Call *FD_ISSET* to check if any thread in read queue is ready for reading. If so, move all those thread from read queue to event queue. It also calls *FD_CLR* to remove that specific thread's flag set on the readfd.
- Call *FD_ISSET* to check if any thread in write queue is ready for writing. If so, move all those threads from write queue to event queue. It also calls *FD_CLR* to remove that specific thread's flag set on the writefd.
- Get the current time and store it in time_now. Then move all the thread that has its sand time expires to event thread.
- Call *thread_trim_head* to remove head thread of the event queue. If the queue is empty, jump to retry label. Otherwise, move it to unuse queue and return that head thread for *thread_call* function.

24. *thread_call* is used to execute the function stored in the thread.

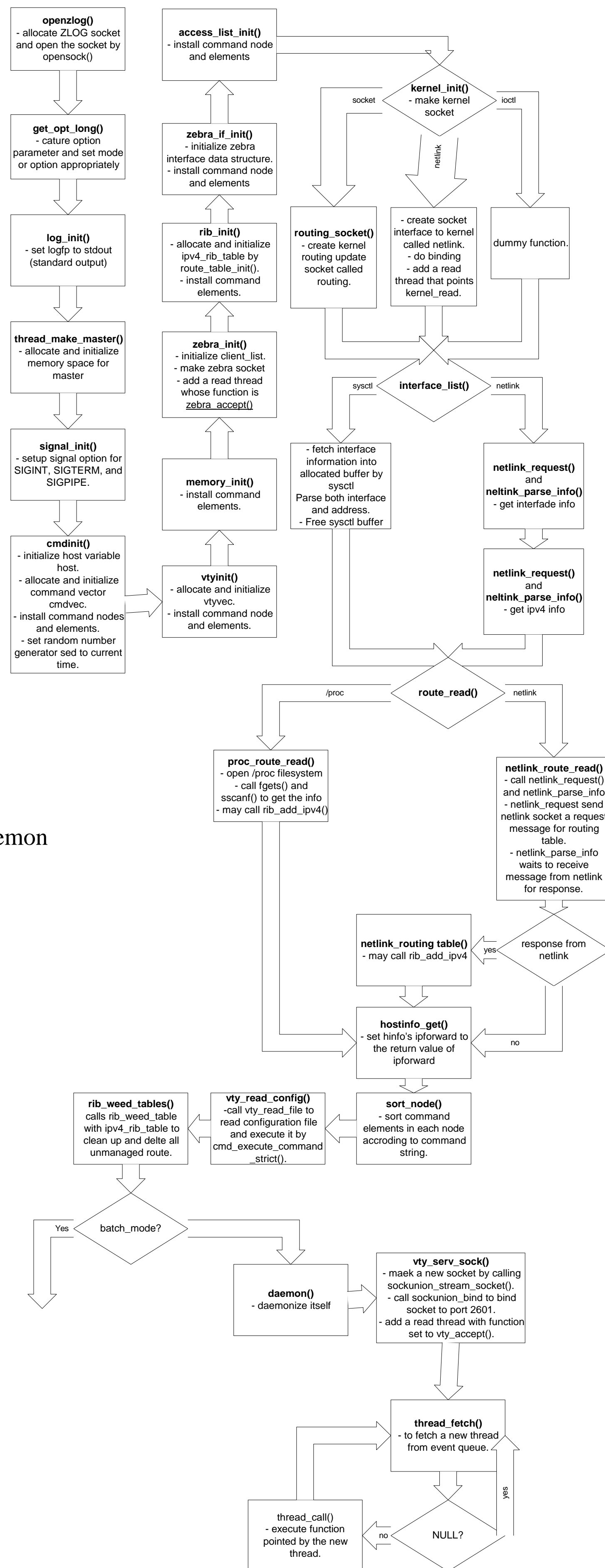


Figure 3.5: Zebra daemon execution sequence

3.10 Important threads running in Zebra

The previous section describes the actions that take place during the initialization. After the initialization, Zebra begins to interact with its outside world (e.g. kernel, vty, and other daemons) by executing *threads* in its scheduler.

Zebra initialization creates three *threads* in the read queue and they are listed according to the order they are created and thus in the order of their execution.

1. The first *thread* has its function argument points to `zebra_accept`.
2. The second *thread* has its function argument points to `kernel_read`.
3. And the last *thread* has its function argument points to `vty_accept`.

From their function names, it is obvious that these three functions are used to accept messages from the daemons, kernel, and the vty respectively.

About `zebra_accept`

`zebra_accept` is a handler to zebra service request. It uses the socket and port setup that was created in `zebra_init` during the state of initialization. By use of socket and `accept` function call, the Zebra daemon establishes a connection with the other daemons. Notice `accept` is in block mode and therefore blocks the current running *thread* until a packet is received. This may seen to be a problem at first sight but it is not. Remember that `select` was used inside `thread_fetch` to monitor which *threads* are ready for reading or writing. *Threads* are only moved to the event queue and get fetched if their corresponding file descriptors indicate them to be ready for further processing. As a result, the above case of blocking the running *thread* will not happen. When `accept` returns, the new socket descriptor for the connection between the Zebra daemon and the client is assigned to the variable `client_sock`. The `accept` fills in the address structure with the client's IP address and port number, as well as the size (the number of bytes) of client address information.

If *accept* is done successfully, Zebra will invoke *client_new* to create a new client. The creation of the client involves:

- Setting the client's fd (file descriptor) to the new socket descriptor client_sock.
- Allocating new input and output buffer called ibuf and obuf. The size of the buffer is 4096 bytes.
- Setting its rtm_table to rtm_table_default.
- Adding the new client to the tail of the global client_list. client_list is used to take tack of the clients created in zebra daemon.
    - Creating a new client would also means creating a set of threads for its own use. So, it is necessary to add a read thread that points to function *zebra_read* to allow the new client to send information to zebra and allows zebra to read them by *zebra_read*.
- Adding a read *thread* that point to zebra_accept (the running thread) with client = NULL and sock = accept_sock. So, when the function finishes, this *thread* moves to unuse queue and a new *thread* is created to continuously look for other client.

About *kernel_read*

kernel_read is used for kernel route reflection. This document only discusses the use of netlink in *kernel_read*.

Inside *kernel_read*, the socket that was created in *kernel_init* is retrieved and stored in the variable called sock. The retrieved socket is a kernel socket that will be used to communicate between the Zebra daemon and the kernel. Once the retrieval is done, *kernel_read* calls *netlink_parse_info(*netlink_route_change) in which *netlink_parse_info* receives a message through the socket (sock) and pass it to *netlink_route_change* for further analysis of the message. According to

message information, *netlink_route_change* may perform addition or deletion of routing information in the route table.

Finally, *kernel_read* add another read *thread* that points to itself to the read queue. Thus, zebra daemon will continue to monitor the incoming packet from the kernel.

About *vty_accept*

vty_accept is used to accept connection from the vty client. It first retrieves socket descriptor that was created during *vty_serv_sock* and stores it into `accept_sock`. Then, it calls *sockunion_accept* to receive information (such as port number and address structure) from the vty client. *sockunion_accept* also returns a new socket descriptor to the new vty client and is held by the variable called `vty_sock`.

After creating a new socket descriptor, a new vty session is created by calling *vty_create*. Inside the call of *vty_create*, a new vty structure is allocated and its parameters are set to the default value. For example, its file descriptor is set to `vty_sock`.

- The new vty structure is then put into *vty_sock*th slot of the global vector variable called `vtyvec`.
- `vtyvec` holds the vty that has been created so far inside the Zebra.
- Check if the vty's password has been set already. If no password has not been set, vty is not available.
- Call *vty_hello* to say hello to vty interface.
- Setting up terminal which includes
  - Call *vty_will_echo* to send `WILL_TELOPT_ECHO` to remote sever.
  - Call *vty_do_window_size* to use window size.

  - Call `vty_prompt` to put out prompt and wait input from the user at the vty terminal.
- Call `vty_event` twice to create one write thread and one read thread.
  - The first call is to add a write thread that points to function `vty_flush`. When zebra daemon wants to pass information to the other daemon, this write thread is fetched to flush output buffer of the vty to the vty.
  - The second one is to add a thread the points to `vty_read`.
- Each vty has a timeout period which is set to 300 seconds by default.
- Call `thread_add_timer` to add a new timer thread with its expiration time set to current clock time + vty's timeout. It is added into the timer queue of the scheduler. When its timer expires, it will be fetched and will terminate the vty connection.
- Finally, `thread_add_read` is called to create a new read thread that points to the function itself (`vty_accept`).

About `zebra_read`

Similar to the thread function above, the first thing task is to get its thread data such as socket descriptor and its client information, client.

- Its call `stream_read` to obtain length, command, and the rest of the packet content.
- Once the above information has been received, `zebra_read` uses the command in a switch-case construct to decide what kind of function the client requests the zebra to do.

Request can be

- ZEBRA_IPV4_ROUTE_ADD → *zebra_read_ipv4*
- ZEBRA_IPV4_ROUTE_DELETE → *zebra_read_ipv4*
- ZEBRA_IPV6_ROUTE_ADD → *zebra_read_ipv6*
- ZEBRA_IPV6_ROUTE_DELETE → *zebra_read_ipv6*

- ZEBRA_GET_ALL_INTERFACE → *zebra_request_all_interface*
- ZEBRA_GET_ONE_INTERFACE → doing nothing
- ZEBRA_GET_HOSTINOF → *zebra_request_hostinfo*
- ZEBRA_REDISTRIBUTE_ADD → *zebra_redistribute_add*
- ZEBRA_REDISTRIBUTE_DELETE → *zebra_redistribute_delete*
- `default` → `zlog "Zebra received unknown command"`

- Call *`stream_reset`* to reset the buffer pointers.

- Call *`zebra_event`* to add another *`zebra_read`* thread to the read queue of the scheduler.

About *`vty_flush`*

`vty_flush` is to flush buffer to the vty.

- Call *`buffer_flush_window`* to calculate size of the output and then call *`buffer_flush_vty`* to flush buffer to the file.
- If the vty's output buffer is empty and the vty's status is `VTY_CLOSE`, then close the vty by calling *`vty_close`*. Otherwise, if vty's output buffer is empty and its status is not `VTY_CLOSE`, set vty's status to VTY_NORMAL.
- If the vty's output buffer is not empty, then set its status to `VTY_MORE`.
- Return 0.

About *`vty_read()`*

`vty_read` is to read data via vty socket.

- Retrieve vty's socket.

- Call read to read raw data from socket.
- If the read results no data, set the vty's status to VTY_CLOSE.
- According to the status or escape flag and buffer content, different action is performed under the case statement.
- Finally, check if the vty's status is VTY_CLOSE. If so, call *vty_close* to close the socket. Otherwise, call *vty_event* twice. One for VTY_WRITE and the other for VTY_READ. (*See* above.)

About *vty_timeout*

vty_timeout: when time out occur, output message then close connection.

- Clear buffer by calling *buffer_reset*.
- Call *vty_out* to output "Vty connection is timed out."
- Close connection by setting the vty's status to VTY_CLOSE and call *vty_close* to close the vty.
- Return 0.

3.11 Current status

Zebra is still in the development stage; once the final release (version 1.0) is completed it will be available at the GNU ftp server. This report is written based on the June 1999's latest version of Zebra, beta-0.67. The current version is version 0.78, which has the bugs fixed and includes additional features such as SNMP.

You could join the Zebra mailing list for documentation and support. The address is zebra@zebra.org. You could also send or subscribe to bug-zebra@gnu.org for bug reports.

3.12 Supported platforms

- GNU/Linux 2.0.X and 2.2.X
- FreeBSD 2.2.8, 3.1, and 4.X.

- NetBSD 1.4
- OpenBSD 2.4

Chapter 4

GateD

The chapter discusses another Linux routing software, GateD. It provides a general introduction in the beginning and then, outlines general aspects in GateD, such as the operation of jobs in its scheduler.

4.1 Introduction

GateD (or "GateDaemon") is a software package used to interconnect packet-switched networks. The "Merit GateD Consortium" divides its GateD software by protocols: Unicast, Multicast, IPv6, and RSd. Current releases of GateD are available only to GateD Consortium members, however, academic groups can join the membership for free.

Unlike Zebra, GateD is a single daemon that can run multiple protocols at the same time. GateD supports a wide variety of protocols such as RIP, EGP, BGP, OSPF, RIPng, OSPFv6, and BGP with multicasting.

In this chapter, we focus on the operation of RIP in GateD (Unicast) and explore only the high level details of the software and how they are used.

4.2 Protocol Scheduling

Protocol scheduling is done in a continuous loop in `main`, it is responsible for handling queuing and scheduling functions. A high level detail is illustrated in figure 4.1.

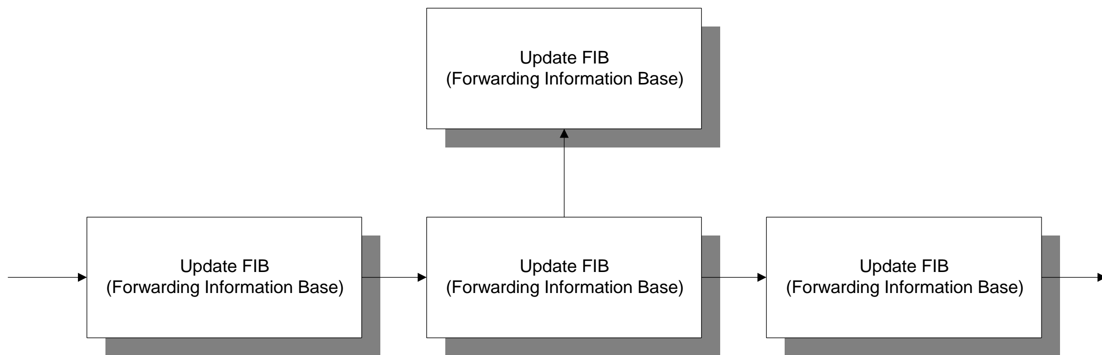


Figure 4.1: High detail of protocol scheduling.

4.3 Task

GateD provides a platform for prototyping and supports multiple routing protocols. This is achieved with subroutines called tasks. Task is a sequence of related primitive operations that are important in completing a request from the program. Different protocol has different task and it is non-preemptive. It usually associates with a UNIX socket, eg. When packets arrive on a socket, the task module calls a handler *rip_update* to perform the read update.

A protocol creates a task and inserts it into the global list of task by calling *task_create* and reserves a socket by calling *task_set_socket*. The operating system can read the task directly from *task_recv_buffer*, and protocols can schedule and read on the task socket by using *task_set_recv*.

There are two global task queues: task_read_queue and task_write_queue. In addition, each task structure includes the following fields:

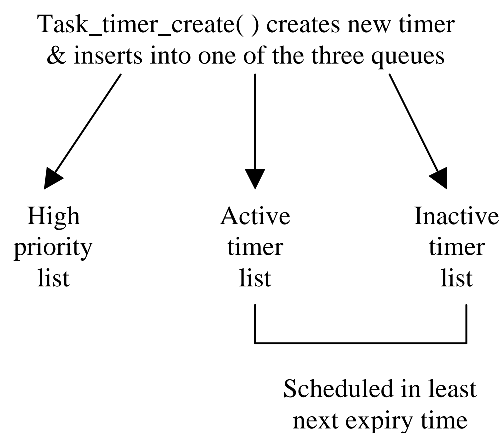
- Pointer to protocol implementation state block
- Socket descriptor
- Pointer to protocol specific routines:
- Read/write routing protocol message

- Flash changes to forwarding table
- Schedule timers and jobs

4.4 Timer

A timer (non-preemptive) schedules periodic computations and it is generally used for monitoring connection status and time outs on connections. It has a time limit and the system load affects its expiry.

GateD allows protocol to specify priority to its timers and allows it to ignore the operating system timer expiry notification. Timer expiry is checked using polling to all active timers in the main loop. A timer dispatcher, `task_timer_dispatch`, traverses through the timer list according to priority to schedule the next timer.



4.5 Job

Job is a unit of computation scheduled at a later time. There are 2 types of jobs:

- Foreground – foreground jobs are queued in a FIFO list, it is time limited; an example would be a job to close a peer connection. The main loop call

`task_job_fg_dispatch` to schedule foreground jobs, and the queue runs to completion.

- Background – background jobs are queued in a priority list with priority ranging from 0-7. They are used for longer running computations. The main loop call `task_job_bg_dispatch` to remove the highest priority bg job from the queue, and returns after completing that job.

4.6 Memory Management

GateD uses the OS memory management function, *`malloc`*, for variable sized allocation. However, GateD has its own memory management routines, like *`task_block`*, for highly space-optimization and for allocating fixed size blocks of 32 or 64 bytes. Protocols can register with the memory manager by calling `task_block_init`.

4.7 Routing Table

In GateD, changes to route database must be made by atomic changes (lock/unlock). Whenever the database is unlocked, the route database module informs all protocol instances of all changes. The routing database in GateD is a single radix tree for all the routes; one per address family.

Updating the router's RIB and the kernel's routing table are done using timer handler or flash handler. Each internal node of the contains a pointer to an `rt_head` struct which contains destination address, a back pointer to the internal node in a radix tree. A list of changes of the database is stored in the `rt_list` structure. In addition, the protocols maintain their own maps of routes in the database (this can be accessed in a byte array stored in the route entry's `rt_head`). Furthermore, the routing table allows multiple routes per destination and they are aged unless the `RTS_NOAGE` flag is specified.

4.8 Interfaces

During initialization, GateD finds all active interfaces and creates interface structures for them. Every minute these interfaces are checked for a change in status, but new interfaces are not detected. However, if no packets are received within the time out period, the routes to this interface will be deleted.

Chapter 5

Linux Kernel

This chapter describes mechanisms inside the Linux kernel. Understanding these structures and how they interact provides a basic foundatation to the understanding in the following chapters. Readers who have background in Linux kernel and are only interested in Linux network are recommended to skip this chapter and move onto the next section. The chapter begins with a look at the various critical data structure such as task structure and file structure that forms fundamental building block in the kernel. We then look at main algorithm for the process management and this topic includes the use of signal and interrupts in system booting and ths scheduler. Finally We discuss the implementation of system calls that make a range of services in the operation system available to the processes,

5.1 Processes and tasks

Process can takes one of a number of states. Figure 1 shows the most important of these. The arrows in this diagram show the possible changes of state. The following states are possible.

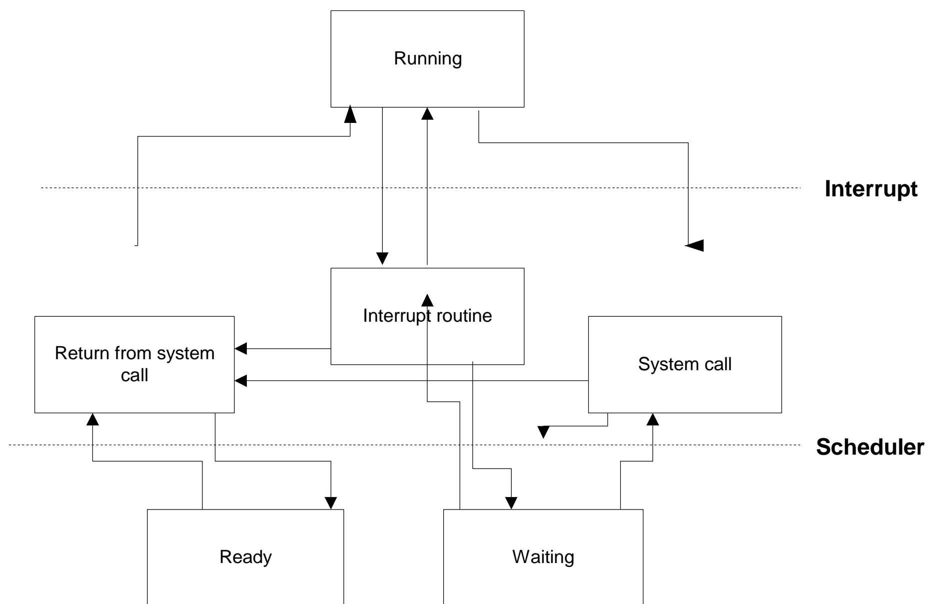


Figure 5.1: Process states in Linux

Running	The task[4] is active and running in the non-privileged User Mode and this state can only be exited via an interrupt or a system call.
Interrupt routine	The interrupt routines become active when the hardware signals an exception condition.
System call	System calls are initiated by software interrupts. A system call is able to suspend the task to wait for an event.
Waiting	The process is waiting for an external event. Only after this has occurred will it continue.
Ready	The process is competing for the processor, which is however occupied with another process at the present time.
Return from system call	The state is automatically adopted after every system call and after some interrupts. At this point, checks are made as to whether the scheduler needs to be called and whether there are signals to process. The scheduler can switch the process to the 'Ready' state and activate another process.

[4] The term 'task' and 'process' are used interchangeably in Linux terminology.

5.2 Important data structure

5.2.1 The task structure

The description of the characteristics of a process is given in the structure `task_struct`.

```
struct task_struct {
      volatile5 long state;
```

The state variable contains a code for current state of the process and can be one of the following values.

State definition	Description
`TASK_RUNNING`	The process is waiting for the CPU to be assigned or it is running.
`TASK_INTERRUPTIBLE`	The process is waiting for certain events (known as blocking system calls) and is therefore at present idle. This process *can be* reactivated by signals.
`TASK_UNINTERRUPTIBLE`	The process is waiting for certain events (known as blocking system calls) and is therefore at present idle. This kind of process state is waiting directly or indirectly for a hardware condition and therefore *will not* accept any signal.[6]
`TASK_STOPPED`	A process has been halted, either after receiving an appropriate signal (`SIGSTOP, SIGSTP, SIGTTIN` or `SIGTTOU`) or when the process is being monitored by another process using the *`ptrace`* system call and has passed control to the monitoring process.
`TASK_ZOMBIE`	A process has been terminated but which must still have its task structure in the process table.
`TASK_SWAPPING`	Not yet used in Linux kernel 2.0.

[5] The keyword volatile indicates that the state variable can be altered asynchronously from interrupt routines.

```
long counter;
long priority;
```

The `counter` variable holds the time in 'ticks' for which the process can still run before a mandatory scheduling action is carried out. The scheduler uses the counter value to select the next process. `counter` thus is a mechanism to represent dynamic priority of a process, while `priority` holds the static priority of a process. The scheduling algorithm uses priority to derive a new value for counter when necessary.

```
unsigned long signal;
unsigned long blocked
```

The `signal` variable contains a bit mask for signals that has been received by the process; `blocked` contains a bit mask for all the signals the process intends to handle later – that is, those for which processing is at present blocked. The size of these two depend upon the word size of the processor (in this case 32 bits). This signal flag is evaluated in the routine *`ret_from_sys_call`*, which is called after every system call and after slow interrupts.

```
unsigned long flags;
int errno;
long debugreg[8];
```

`flags` may contain the combination of the following system status flags

Flags name	Value	Description
PF_ALIGNWARN	0x00000001	Print alignment warning messages.
PF_PTRACED	0x00000010	They indicate that the process is being monitored by another process with the aid of the system call *`ptrace`*.
PF_TRACESYS	0x00000020	
PF_FORKNOEXEC	0x00000040	Forked but didn't exec.
PF_SUPERPRIV	0x00000100	Used super-user privileges.
PF_DUMPCORE	0x00000200	Dumped core.
PF_SIGNALED	0x00000400	Killed by a signal.
PF_STARTING	0x00000100	Indicates that the process is just being initiated.
PF_EXITING	0x00000200	Indicates that the process is just being terminated.
PF_USEDFPU	0x00100000	Process used the FPU this quantum (SMP only)
PF_DTRACE	0x00200000	Delayed trace (used on m68k)

[6] Note the difference between `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE`. The first one is "*can be*" and the latter one is "*will not*".

The `errno` variable holds the error code for the last faulty system call. On return from the system call, this is copied into the global variable `errno`.

The `debugreg` variable contains the 80x86's debugging registers. These are at present used only by the system call *`ptrace`*.

5.2.2 *Process relationships*

All processes are kept in a doubly linked list with the help of the two following components

```
struct task_struct *next_task, *prev_task;
struct task_struct *next_run,  *prev_run;
```

The start and end of this list are held in the global variable `init_task`.

There are 'family relationships' between the processes, which are represented by the following components:

`struct task_struct *p_opptr`	Original parent
`struct task_struct *p_pptr`	Parent
`struct task_struct *p_cptr`	Youngest child
`struct task_struct *p_ysptr`	Younger sibling
`struct task_struct *p_osptr`	Older sibling

The child processes for the same parent processes are similarly linked together as a doubly linked list by `p_ysptr` (next younger sibling) and `p_osptr` (next older sibling).

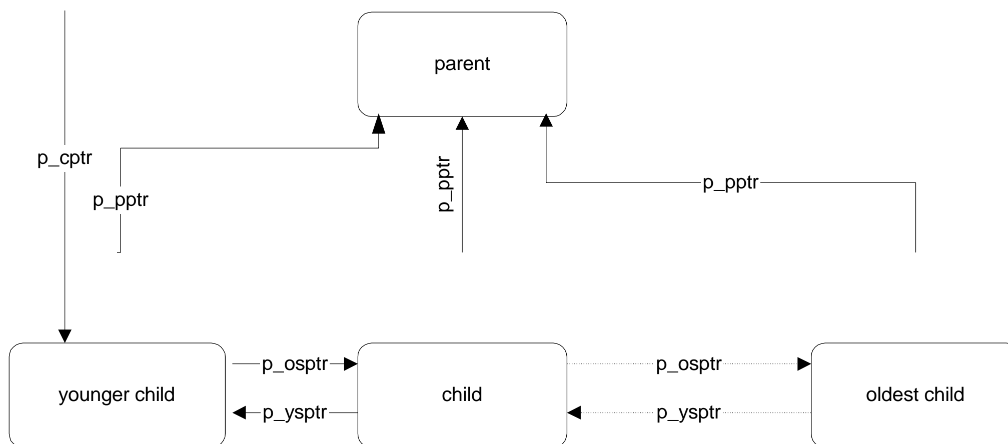


Figure 5.2: Family relationship between processes

5.2.3 Memory Management

A process's virtual memory contains executable code and data from many sources, such as 1. Program image that is loaded; 2. New (virtual) memory allocated during execution of a process; 3. Shared libraries that can be used by several running processes at the same time. Linux uses a technique called *demand on paging* where the virtual memory of a process is brought into physical memory only when a process attempts to use it. Figure 3 illustrates a general picture of a process's virtual memory.

The Linux kernel needs to manage all of the area of virtual memory, and the content of each process's virtual memory is described by a `mm_struct` data structure.

```
struct mm_struct *mm;
```

The process's `mm_struct` data structure also contains information about loaded executable image and a pointer to the process's page tables.

```
unsigned long context;
pgd_t * pgd;
```

In addition, part of the components inside `mm_struct` are:

```
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk, start_stack, start_mmap;
unsigned long arg_start, arg_end, env_start, env_end;
```

which describe the start and size of the code and data segments of the program currently running.

It also contains pointers to a list of `vm_area_struct` data structures, each representing an area of virtual memory within the process. To speed up access to virtual memory, Linux arranges the `vm_area_struct` data structure into AVL (Adelson-Velskii and Landis) tree. This tree is arranged so that each `vm_area_struct` (or node) has a left and a right pointer to its neighboring `vm_area_struct` structure. The left pointer points to node with a lower starting virtual address and the right pointer points to a node with a higher starting virtual address.

Two more components in `task_struct` that are related to memory management are `kernel_stack_page` and `saved_kernel_stack`.

When a process is operating in System Mode, it needs its own process stack. The address of the stack is stored in `kernel_stack_page`. For the MS-DOS emulator, there is another stack pointer `saved_kernel_stack`, which stores the old stack pointer.

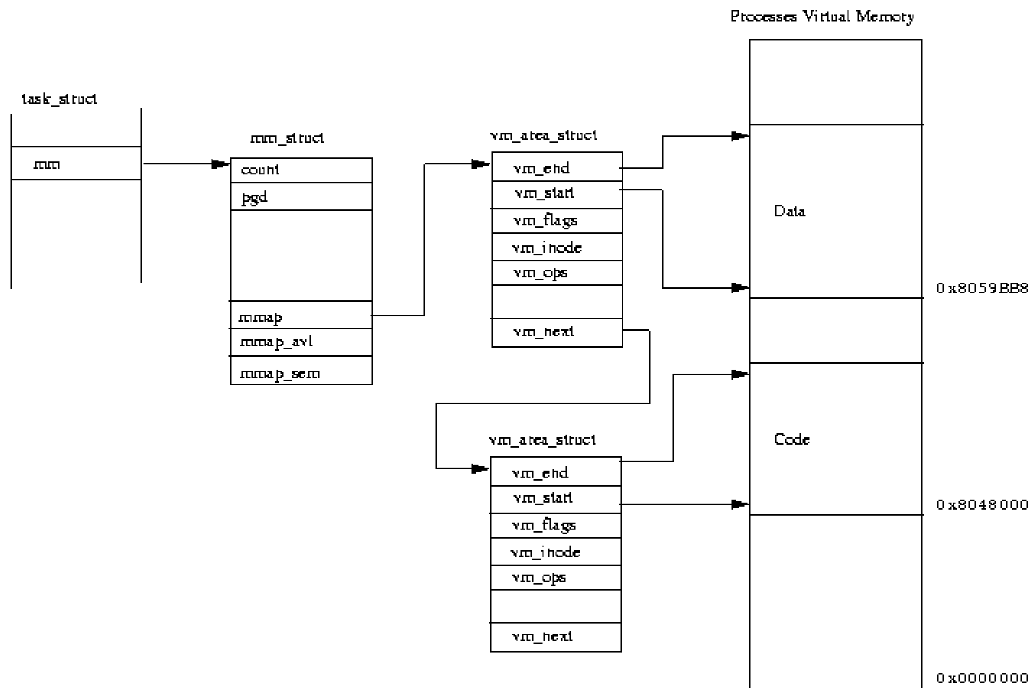


Figure 5.3: Linux memory data structure relationship

5.2.4 *Process ID*

Every process has its own process ID number, `pid`, and is assigned to a process group, `pgrp`, and a session, `session`. Every session has a leader process, `leader`.

```
int pid;
int pgrp;
int session;
int leader;
```

To handle access control, every process has a user ID, `uid`, and a group ID, `gid`. These are inherited by the child process from the parent process when a new process is created by the *fork* system call. For actual access control, the effective user ID, `euid`, and the effective group ID, `egid`, are used. `fsuid` is used whenever identification is required by the file system. As a general rule, `(uid==euid)&&(gid==egid)`and `(fsuid==euid)&&(fsgid==egid)`.

Exceptions arise for so-called set-UID programs, where the values of euid and fsuid, or those of egid and fsgid, and set to the user ID and the group ID for the owner of the executable file. This makes a controlled distribution of privileges possible e.g. modification of user password requires super user privilege[7].

Linux's setfsuid system call allows fsuid to be altered without changing euid and this means that daemons can limit their rights when accessing file system with setfsuid (to the rights of the user for whom they are providing services), but they will retain their privileges.

Linux allows a process to be assigned to a number of user groups at the same time. These groups are considered during checking the access permission to files. Each process may belong to a maximum of NGROUPS (i.e. 32) groups, which are held in the groups component of the task structure.

```
int    groups[NGROUPS];
```

5.2.5 Files

There are two data structures that describe file system specific information for each process in the system. The first, the fs_struct contains pointers to this process's VFS inodes and its umask. The second data structure, the files_struct, contains information about all of the files that this process is currently using.

[7] The operating system will use the real Ids to identify the *real user* for things such as process accounting or sending mail, and the effective Ids to determine what additional permissions should be granted to the process. Most of the time the real and effective Ids for a process are identical. However, there are occasions when non-privileged users on a system must be allowed to access/modify privileged files (such as the password file). To allow controlled access to key files, Unix has an additional set of file permissions (known as set-user-ID (SUID) and set-group-ID (SGID) that can be specified by the file's owner. When indicated, these permission tell the operating system that, when the program is run, the resulting process should have the privileges of the owner/group of the program (versus the real user/group privileges associated with the process).

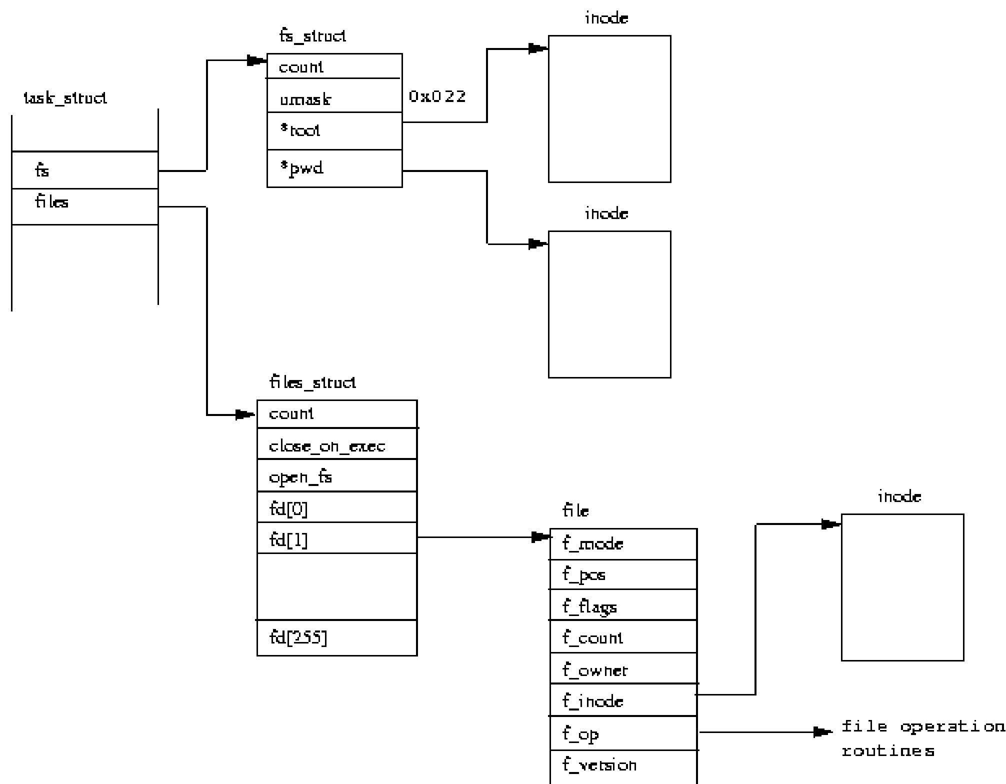


Figure 5.4: Linux file data structure.

fs_struct:

The file-system-specific data are stored in the substructure:

```
struct fs_struct fs[1];
```

This contains the four components:

count	The count variable is reserved for future expansions.
umask	A process can affect the access mode of newly created files via the system call *umask*. The values set using *umask* are also stored in the component umask.
pwd	Under Unix, every process has a current directory, pwd, which is required when resolving relative pathnames and can be changed by means of system call *chdir*.
root	Every process has in addition its own root directory – root – that is used in resolving absolute pathnames.

files_struct:

A process opening a file with *open* or *creat* is given a file descriptor by the kernel to use in referencing the file in the future. The file descriptors are assigned to the files under Linux via the `fd[]` field in the substructure:

```
struct files_struct *files;
```

File descriptors are used as an index in the `fd[]` field, which locates the file pointer assigned to the file descriptor, and with its help the file itself can then be accessed.

The `files_struct` has four components:

open_fds	A bit mask of all file descriptors used
close_on_exec	Contains a bit mask of all file descriptors, which is used to close the file descriptor when the system call *exec* is issued.
fd_set	This data type is large enough to hold NR_OPEN (256) bits.
count	Served as a reference counter.

Every file has its own descriptor and the `files_struct` contains pointers to up to 256 `file` data structures, each describing a file being used by this process. More information is provided in section "Files and inodes".

5.2.6 Timing

Under Linux, times are always measured in 'ticks'. These ticks are generated by a timer chip every 10 milliseconds and counted by the timer interrupt.

```
long utime, stime, cutime, cstime, start_time;
```

utime	hold the time the process has spent in User Mode.
stime	hold the time the process has spent in System Mode.
cutime	hold the totals of the corresponding times for all child processes in User

	Mode.
`cstime`	hold the totals of the corresponding times for all child processes in System Mode.
`start_time`	contains the time at which the current process was generated.

In addition to these accounting timers, Linux supports process specific *interval* timers. A process can use these timers to send itself various signals each time that they expire. Three sorts of interval timers are supported:

Real	The timer ticks in real time, and when the timer has expired, the process sends a `SIGALRM` signal.
Virtual	This timer only ticks when the process is running and when it expires it sends a `SIGVTALRM` signal.
Profile	These timer ticks both when the process is running and when the system is executing on behalf to the process itself. `SIGPROF` is signaled when it expires.

One or all of the interval timers may be running and Linux keeps all of the necessary information in the process's `task_struct` data structure.

```
unsigned long timeout, policy, rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
```

The components `it_real_value`, `it_prof_value` and `it_virt_value` contain the time in ticks until the timer will be triggered. The components `it_real_incr`, `it_prof_incr` and `it_virt_incr` hold the values required to reinitialize the timers after they run out. `real_timer` is used for the implementation of the real-time interval timer.

5.2.7 Inter-process communication

The Linux kernel implements a system of inter-process communication to provide services such as semaphores.

A process can occupy a semaphore. If other processes also wish to occupy this semaphore, they are halted until the semaphore is released. This uses the component

```
struct sem_queue *semsleeping;
```

When the process is terminated, the operating system must release all semaphores occupied by the process. The component

```
struct sem_undo *semundo;
```

contains the information required for this.

5.2.8 Miscellaneous

The following components do not fit in any of the above groups:

```
struct wait_queue *wait_chldexit;
```

A process executing the system call *`wait4`* must be halted until a child process terminates. It joins the `wait_chldexit` wait queue in its own task structure, sets the status flag to the value `TASK_INTERRUPTIBLE` and passes control to the scheduler. When a process terminates it signals this to its parent process via this queue.

Every process can decide how it wishes to react to signals. This is specified in the `sigaction` structure.

```
struct sigaction sigaction[32];
```

Amongst other things, it contains either the address of a routine that will handle the signal or a flag which tells Linux that the process either wishes to ignore this signal or let the kernel handle the signal for it.

Every process can check its limits for the use of resources by means of the system call *setrlimit* and *getrlimit*. These are stored in the `rlim` structure.

```
struct rlimit rlim[RLIM_NLIMITS];

int exit_code, exit_signal;
```

holds the return code for the program and the signal by which the program has been aborted.

```
unsigned long personality;
```

`personality` is used to give a precise description of the characteristics of the version of Unix on which this process is run. For standard Linux programs, `personality` takes the value `PER_LINUX` (defined as 0 in `<linux/personality.h>`).

```
int dumpable:1;
int did_exec:1;
```

The `dumpable` flag indicates whether a memory dump is to be executed by the current process if certain signal occurs.

A rather obscure semantic in the POSIX standard requires, when calling *setpgid*, to distinguish whether a process is still running the original program or whether it has loaded a new program with the system call `execve`. This information is monitored using the flag `did_exec`.

```
struct desc_struct *ldt;
```

This entry has been included especially for the WINE Window emulator, which needs more information and different memory management routines as compared with a standard Linux program.

```
struct linux_binfmt *binfmt;
```

`binfmt` describes the functions responsible for loading the program.

```
struct thread_struct tss;
```

The `thread_struct` structure holds all the data on the current processor status at the time of the last transition from User Mode to System Mode. All the processor registers are saved here to enable them to restore on return to User Mode. Inside `thread_struct`, there are components:

```
struct vm86_struct * vm86_info;
unsigned long screen_bitmap;
unsigned long v86flags, v86mask, v86mode;
```

to describe the 8086 emulation implemented by the system call *`vm86`* .

Linux supports three scheduling algorithms: a classical scheduling (`SCHED_OTHER`) and two real-time scheduling algorithms (`SCHED_RR` and `SCHED_FIFO`). Each process can be assigned to one of these scheduling classes `policy` with its real-time priority `rt_priority`.

```
unsigned long policy; /* SCHED_FIFO, SCHED_RR, SCHED_OTHER */
unsigned long rt_priority;
```

The linux kernel supports SMP (Symmetric Multi-Processing) and therefore, each process needs to know on which processor it is running.

```
#ifdef __SMP__
	int processor;
	int last_processor;
	int lock_depth;
#endif
```

5.2.9 *The process table*

Every process occupies exactly one entry in the process table, and the process table is statically organized as a doubly linked circular list:

```
struct task_struct *task[NR_TASKS];
```

`next_task` and `prev_task` in each `task_struct` allows kernel routine to transverse through the process table. The external variable `init_task` points to the start of the doubly linked circular list.

```
struct task_struct init_task;
```

This is initialized with the first task `INIT_TASK` when the system is booted. Once the system has been booted, this is only responsible for the use of unclaimed system time (the idle process). The entry `task[0]` is the `INIT_TASK`.

Linux 2.0 supports multi-processing (SMP) and therefore, there is a global variable called `current` that points to the current task for each processor.

```
#define current (0+current_set[smp_processor_id()])
struct task_struct *current_set[NR_CPUS];
```

5.2.10 *Files and inodes*

Inodes contain information such as the file's owner and access rights. There is ex*actly one* inode entry in the kernel for each file in a system. File structures (data structures of the type `struct file`), on the other hand, contain the view of a process on these files (represented by inodes). This view on the file includes attributes, such as the mode in which the file can be used (read, write, read + write), or the current position of the next I/O operation.

File structure

The file structure contains the following components.

`f_mode`	Describes the access node in which the file was opened (read-only, read + write or write only).
`f_pos`	Holds the position of the read/write pointer at which the next I/O operation will be carried out. The value is updated by every I/O operation and by the system calls *`lseek`* and *`llseek.`* The `f_pos` is type of `loff_t`, which is a 64-bit word and therefore, is able to handle 2 gigabytes (2^{31} bytes).
`f_flags`	Contains additional access flags. These can be set when a file is open with the system call `open` and later read and modified using the system call `fcntl.`
`f_count`	Serves as a reference counter. A number of file descriptors may refer to the same file structure. When a file is opened, `f_count` is initialized to 1. Every time the file descriptor is copied (by the system calls `dup,` `dup2,` or `fork`) the reference counter is incremented by 1, and every time a file is closed (using the system call `close,` `_exit` or `exec`) it is decreased by 1. The file structure is only released once there is no longer any process referring to it (i.e. `f_count == 0).`
`f_next,` `f_prev`	All file structures present in the system form part of a doubly linked list through their components `f_next` and `f_prev`. The global variable `struct file *first_file;` constitutes the start of this list.
`f_inode`	refers to the actual descriptor of the file.
`f_op`	Refers to a structure of function pointers referencing all file operations.

Inodes

Many of the components of the `inode`'s structure can be polled via the system call *`stat.`* The inode data structure contains the following component.

`i_dev`	A description of the device (the disk partition) on which the file is located.
`I_ino`	Identifies the file within the device. It refers to the block number of the data structure on the hard disk, describing the file on the external memory device.
`I_mode`	Describes the access permissions to the file.
`I_uid`, `I_gid`	describes its owner (user and group)
`I_size`	The size in bytes
`I_mtime`	The times of the last modification
`I_atime`	The last access time.
`I_ctime`	The last modification time to the inode.

```
struct inode_operations *i_op;
```

Similar to the file structure, the inode also has a reference to a structure containing pointers to functions that can be used on inode.

5.2.11 Dynamic memory management

Under Linux, memory is managed on a page basis. One page contains 2^{12} bytes. The basis operations to request a free page are the functions

```
#define __get_free_page(priority) __get_free_pages((priority),0,0)
#define __get_dma_pages(priority, order)
__get_free_pages((priority),(order),1)
extern unsigned long __get_free_pages(int priority, unsigned long
gfporder, int dma);
```

`priority`	Controls the behavior of `__gret_free_page()` if not enough pages are free in main memory. `priority` can takes one of the following values: `GFP_BUFFER`, `GFP_ATOMIC`, `GFP_KERNEL`, `GFP_NOBUFFER`, and `GFP_NFS`.
`gfporder`	describes the number of pages to be reserved, which is 2^{order}.
`dma`	If `dma` is not equal to 0, memory is requested that can be addressed by the DMA environment.

In Linux kernel, there is a similar provision of *malloc* and *free* that are often used in C programming, and they are:

```
void *kmalloc(size_t size, int priority);
void kfree(void * ptr);
```

The argument `priority` indicates how *kmalloc* is to request new pages of memory using *get_free_page*. *kmalloc* can request blocks of memory up to an extent of 128 Kbytes.

5.2.12 Queues and semaphores

Often a process will be dependent on the occurrence of certain conditions. For instance, a process is waiting for a page to be loaded into the real memory by the DMA. This waiting is implemented in Linux by means of wait queues. A wait queue is a cyclical list containing pointers to the process table.

```
struct wait_queue {
      struct task_struct * task;
      struct wait_queue * next;
};
```

A process can be added or removed from the wait queue by the following functions:

```
void add_wait_queue(struct wait_queue **queue, struct wait_queue
*entry))

void remove_wait_queue(struct wait_queue **queue, struct wait_queue
*entry))
```

The `queue` variable contains the wait queue to be modified, and `entry` the entry to be added or removed.

A process wishing to wait for a specific event enters itself in a wait queue of this type and relinquishes control. There is a wait queue for every possible event (and semaphores). When the relevant event occurs, all the processes in its wait queue are reactivated and can resume operation.

```
void sleep_on(struct wait_queue **p)
void interruptible_sleep_on(struct wait_queue **p)
```

These set the process status (`current->state`) to `TASK_UNINTERRUPTIBLE` or `TASK_INTERRUPTIBLE` respectively, enter the current process (`current`) in the wait queue and call the scheduler. The process then voluntarily relinquishes control.

It is only reactivated when the status of the process is set to `TASK_RUNNING`. This is generally done by another process calling the functions

```
void wake_up(struct wait_queue **q)
void wake_up_interruptible(struct wait_queue **q)
```

to 'wake up' all the processes entered in the wait queue.

With the aid of wait queues, Linux also provides semaphores. These are used to synchronize access by various kernel routines to shared data structures.[8]

```
struct semaphore {
      int count;
      struct wait_queue * wait;
};
```

A semaphore is taken to be occupied of `count` has a value less than or equal to 0. All the processes wishing to occupy the semaphore enter themselves in the wait queue. They are then notified when it is released by another process. There are two auxiliary functions to occupy or release semaphores:

```
extern inline void down(struct semaphore * sem)
```

[8] These semaphores should not be confused with semaphores provided for user program in Unix System V.

```
{
      if (sem->count <= 0)
            __down(sem);
      sem->count--;
}

extern inline void up(struct semaphore * sem)
{
      sem->count++;
      wake_up(&sem->wait);
}

void __down(struct semaphore * sem)
{
      struct wait_queue wait = { current, NULL };
      add_wait_queue(&sem->wait, &wait);
      current->state = TASK_UNINTERRUPTIBLE;
      while (sem->count <= 0) {
            schedule();
            current->state = TASK_UNINTERRUPTIBLE;
      }
      current->state = TASK_RUNNING;
      remove_wait_queue(&sem->wait, &wait);
}
```

5.2.13 System time and timers

There is only one internal time base in Linux, which is measured in ticks[9] elapsed since the system was booted. It is generated by a timer chip in the hardware and counted by the timer interrupt in the global variable `jiffies`.

Sometime, there is a time gap before the next set of data can be sent when a slow device is being used. To support this, Linux provides a facility to initiate functions at a defined future time. Two forms of timer are used.

On the one hand, there is a older timer mechanism, has a static array of 32 pointers to `imer_struct` data structures and a mask of active timers, `timer_active`.

```
struct timer_struct {
      unsigned long expires;
      void (*fn)(void);
} timer_table[32];
```

[9] One tick equal to 10 milliseconds.

Each entry is given a pointer to a function fn and a time expires at which the function is to be called. A bit field

```
unsigned long timer_active;
```

indicates which entries in the timer_table[] are valid. This kind of timer is obsolete and only used for certain device drivers.

For normal applications, there is another newer interface of the form:

```
struct timer_list {
      struct timer_list *next;
      struct timer_list *prev;
      unsigned long expires;
      unsigned long data;
      void (*function)(unsigned long);
};
```

The entries next and last in this structure are used for internal management of all the timers in a doubly linked sort list. Entries are held in ascending expiry time order. At the start of this list is the variable timer_head. The component expires gives the time at which the function function to be called with the argument data. The two functions

```
void add_timer(struct timer_list * timer);
int  del_timer(struct timer_list * timer);
```

are used in the administration of the timer list.[10]

[10] As an argument for add_timer(), expire describes the time interval after which the timer is to run out. Once add_timer() has entered the structure on the list, expire signifies the time at which the function is to be called.

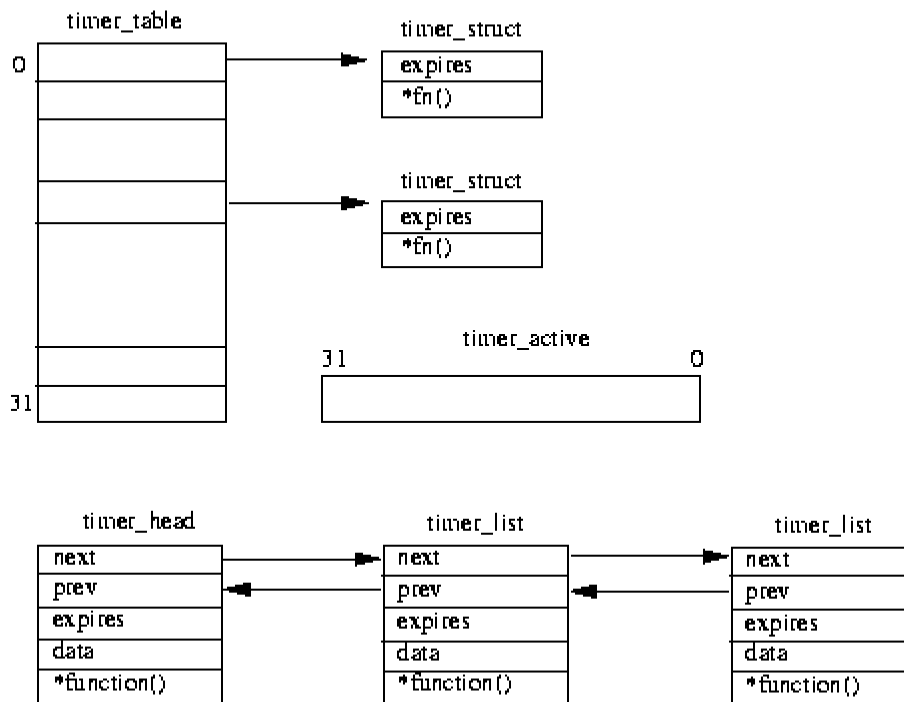


Figure 5.5: timer structure.

5.3 *Main algorithms*

5.3.1 *Signals*

The kernel uses signals to inform processes about certain events. The user typically uses signals to abort processes or to switch interactive programs to a defined state.

All signals are sent by the function `send_sig`. There are three arguments involved: the first argument gives the signal number; the second one provides a description of the process which is to receive the signal (or, more precisely, a pointer to the entry for the process in the task structure); the third argument gives the priority of the sender.

At present, only two priorities are supported. The signal can be sent from a process, or it can be generated by the kernel. The kernel can send a signal to any process, while a normal user process is only allowed to do so if it possess superuser rights or have the same UID and GID as the receiving process. An exception to this is the SIGCONT signal, which may be sent from any process in the same session.

Processes can choose to ignore most of the signals that are generated, with two exception: neither the SIGSTOP signal (which causes a process to halt its execution) nor the SIGKILL signal (which causes a process to exit) can be ignored.

If a blocked signal is generated, it remains pending until it is unblocked.

If there is authority to send a signal and the receiving process is not inclined to ignore this signal, it is sent to the process and is done by setting the bit for the signal number in the signal component of the task structure for the receiving process. This indicates that the signal has been sent. In other words, there is no immediate treatment of the signal by the receiving process: this happens only after the scheduler has returned the process to TASK_RUNNING state.

When the process is reactivated by the scheduler, but before it is switched from system mode to user mode, the routine *ret_from_sys_call* is run. If signals are waiting for the current process, this calls the *do_signal* function, which takes over the actual signal handling. The *do_signal* function manipulates the stack and register of the process so that the process's instruction pointer is set to the first instruction in the signal handling routine, and the parameters for the signal handling routine are added to the stack. When the process resumes operation, it appears to it as if the signal handling routine has been called like a normal function.

A process can specify which signals are to be blocked while a signal handling routine is running. This is implemented by the signal mask current->blocked before

calling the user-defined signal handling routine. The signal mask must be restored to its original state after the signal handling routine has terminated and this is done by placing the system call `sigreturn`, as the return address of the signal handling routine, on to the stack. This then takes care of the clearing-up operation at the end of the signal handling routine.

If a number of signal handling routines need to be called, a number of stack frames are set up. As a result, the signal handling routines are executed one after the other.

About `fore_sig`

In addition, the kernel has the possibility of sending signals by means of the `force_sig` function. This ensures that the signal is delivered even when the process has blocked the signal or wants to ignore it.

5.3.2 Interrupts

Interrupts are used to allow the hardware to communicate with the operating system. There are two types of interrupt in Linux: slow and fast.[11]

Slow interrupts

Slow interrupts are the usual kind. When slow interrupts are being dealt with, other interrupts are legal. After a slow interrupt has been processed, additional activities requiring regular attention are carried out by the system. The processing of an interrupt involves are described below in more details.

[11] We could even say there are three, with the thrid type represented by system calls, which are also triggered via interrupts.

First, all the registers are saved with `SAVE_ALL` and receipt of the interrupt is confirmed to the interrupt controller with `ACK`. At the same time, further receipt of interrupts of the same type is blocked.

The nesting depth of the interrupts is noted in the variable `intr_count`, after which further interrupts are enabled and the interrupt routine itself is called. A copy of the set of registers is provided for the interrupted process and the registers are used by some of the interrupt handlers to determine whether the interrupt has interrupted the user process or the kernel. Once the interrupt routine has been successfully executed, the interrupt controller is informed that interrupts of this type can again be accepted. In addition, the interrupt counter is decremented. A jump into the assembler routine `ret_from_sys_call` is then made. This function takes care of administration tasks after any slow interrupt or system call. It restores the registers saved with `SAV_ALL` and carries out the `iret` required at the end of an interrupt routine.

Fast interrupt

Fast interrupts are used for short, less complex tasks. While they are being handled, any other interrupts are blocked, unless the handling routine involved explicitly enables them.

First, only registers that are modified by a normal C function are saved. (It uses `SAVE_MOST` instead of previous `SAVE_ALL`.) The interrupt controller is informed by `ACK` and the variable `intr_count` is incremented. However, no further interrupts are accepted before the interrupt handler itself is called (*`sti`* is not called). This completes the interrupt handling. The interrupt controller is informed that interrupts can again be accepted. The interrupt counter is decremented. `RESTORE_MOST` returns the saved registers to their previous values and then calls *`iret`* to continue the interrupted process.

5.3.3 *Booting the system*

LILO is responsible to finds the Linux kernel and loads it into memory. It then begins at the entry point `start:` which is held in the `arch/i386/boot/setup.S` file. This file contains assembler code responsible for initializing the hardware. Once the essential hardware parameters have been established, the process is switched into Protected Mode by setting the protected mode bit in the *machine status word.* The assembler instruction

```
jmp 0x1000, KERNEL_CS
```

then initiates a jump to the start address of the 32-bit code for the actual operating system kernel and continues from `startup_32:` in the file `arch/i356/kernel/head.S`. More hardware is initialized (in particular the MMU (page table), the co-processor and the interrupt descriptor table) and the environment (stack, environment, and so on) required for the execution of the kernel's C functions.

Once the initialization is complete, the first C function, *`start_kernel`* from `init/main.c` is called. It first saves all the data the assembler code has found about the hardware up to that point. Next all areas of the kernel are then initialized.

```
asmlinkage void start_kernel(void)
{
	memory_start = paging_init(memory_start,memory_end);
	trap_init();
	init_IRQ();
	sched_init();
	time_init();
	parse_options(command_line);
	init_modules();
	memory_start = console_init(memory_start,memory_end);
	memory_start = pci_init(memory_start,memory_end);
	memory_start = kmalloc_init(memory_start,memory_end);
	sti();
	memory_start = inode_init(memory_start,memory_end);
	memory_start = file_table_init(memory_start,memory_end);
	memory_start = name_cache_init(memory_start,memory_end);
	mem_init(memory_start,memory_end);
	buffer_init();
	sock_init();
	ipc_init();
…
```

The process now running is process 0 and it generates a kernel thread which executes the *init* function.

```
kernel_thread(init, NULL, 0);
```

As a result, process 0 is only concerned with using up unused CPU time.

```
cpu_idle(NULL);
```

The *init* function carries out the remaining initialization and it starts the `bdflush` and `kswap` daemons which are responsible for synchronization of the buffer cache contents with the file system and for swapping.

```
kernel_thread(bdflush, NULL, 0);
kernel_thread(kswapd, NULL, 0);
```

Then the system call setup is used to initialize the file systems and to mount the root file system.

```
setup();
```

The system now attempts to establish a connection with the console and to open the file descriptor 0, 1, and 2.

```
if ((open("/dev/tty1",0_RDWR,0) < 0) &&
   (open("/dev/ttys0", 0_RDWR,0) < 0))
      printk("Unable to open an initial console.");
(void) dup(0);
(void)dup(0);
```

Then the system attempts to execute one of the programs `/etc/init`, `/bin/init` or `/sbin/init`. These usually start the background process running under Linux and make sure that the `getty` program runs on each connected terminal, so that a user can log in to the ststem.

```
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);
```

If none of the above programs exists, an attempt is made to process `/etc/rc` and subsequently start a shell so that the superuser can repair the system.

```
        pid = kernel_thread(do_rc, "/etc/rc", SIGCHLD);
        if (pid>0)
              while (pid != wait(&i))
                    /* nothing */;
  }

  while (1) {
        pid = kernel_thread(do_shell,
              execute_command ? execute_command : "/bin/sh",
              SIGCHLD);
        if (pid < 0) {
              printf("Fork failed in init\n\r");
              continue;
        }
        while (1)
              if (pid == wait(&i))
                    break;
        printf("\n\rchild %d died with code %04x\n\r",pid,i);
        sync();
  }
  return -1;
}
```

5.3.4 Timer interrupt

The system time is usually implemented by arranging the hardware to trigger an interrupt at specified intervals. Under Linux, system time is measured in ‘ticks’ since the system was started up. One tick represents 10 milliseconds. The time is stored in the following variable

```
unsigned long volatile jiffies;
```

which should only be modified by the timer interrupt. This time variable only provides an internal time base.

Applications use the ‘actual time’ which are held in variable

```
volatile struct timeval xtime;
```

which is also updated by the timer interrupt.

The interrupt routine (*do_timer*) proper simply updates the variable `jiffers` and marks the bottom half routine of the timer interrupt as active. This bottom half routine is called by the system at a later point after handling other interrupt. Since several timer interrupt occur before the handling routines become active, the timer interrupt increments the variables

```
unsigned long lost_ticks;
unsigned long lost_ticks_system;
```

so that these can later be evaluated in the bottom half routines.

`lost_ticks` counts the timer interrupts that have passed since the last call of the bottom half routine, whereas *lost_ticks_systems* counts the timer interrupts during whose occurrence the interrupted process was in System Mode.

```
void do_timer(struct pt_regs * regs)
{
	(*(unsigned long *)&jiffies)++;
	lost_ticks++;
	if (should_run_timers(lost_ticks))
		mark_bh(TIMER_BH);
	if (!user_mode(regs)) {
		lost_ticks_system++;
		if (prof_buffer && current->pid) {
			extern int _stext;
			unsigned long ip = instruction_pointer(regs);
			ip -= (unsigned long) &_stext;
			ip >>= prof_shift;
			if (ip < prof_len)
				prof_buffer[ip]++;
		}
	}
	if (tq_timer)
		mark_bh(TQUEUE_BH);
}
```

The real work is then carried by the bottom half routines of the timer interrupt.

```
static void timer_bh(void)
{
	update_times();
	run_old_timers();
	run_timer_list();
}
```

Here, *run_old_timers* and *run_timer_list* process the functions for updating the system-wide timers, which are, timer_struct and timer_list. *update_times* is responsible for updating the times.

```
static inline void update_times(void)
{
	unsigned long ticks;

	ticks = xchg(&lost_ticks, 0);

	if (ticks) {
		unsigned long system;

		system = xchg(&lost_ticks_system, 0);
		calc_load(ticks);
		update_wall_time(ticks);
		update_process_times(ticks, system);
	}
}
```

Here, *xchg* is a function that reads the memory address specified in the first argument and sets the value specified in the second argument in an *atomic* way. The use of atomicity guarantees that no ticks are lost even if a new timer interrupt occurs during the processing of this routine.

update_wall_time updates the *real time* xtime, while update_process_time updates the times of the current process.

First, the counter component of the current task structure is updated. When counter becomes zero, the time slice of the current process has expired and the scheduler is activated at the next opportunity.

```
static void update_process_times(unsigned long ticks, unsigned long
system)
{
	unsigned long user = ticks - system;
	if (p->pid) {
		p->counter -= ticks;
		if (p->counter < 0) {
			p->counter = 0;
			need_resched = 1;
		}
```

Then, the `utime` and `stime` components of the task structure are updated for statistical purpose.

```
	current->utime +=user;
	current->stime += system;
```

It is possible to limit a process 's 'CPU consumption' resource and this is done by means of the system call *setrlimit*, which can also be used to limit other resources of a process. The time limit is checked in the timer interrupt, and the process is either informed via the `SIGXCPU` signal or aborted by means of the `SIGKILL` signal.

```
	psecs = (p->stime + p->utime) / HZ;
	if (psecs > p->rlim[RLIMIT_CPU].rlim_cur) {
		/* Send SIGXCPU every second.. */
		if (psecs * HZ == p->stime + p->utime)
			send_sig(SIGXCPU, p, 1);
		/* and SIGKILL when we go over max.. */
		if (psecs > p->rlim[RLIMIT_CPU].rlim_max)
			send_sig(SIGKILL, p, 1);
```

Subsequently, the interval timers of the current task must be updated. When these have expired, the task is informed by a corresponding signal.

```
	unsigned long it_virt = p->it_virt_value;
	if (it_virt) {
		if (it_virt <= ticks) {
			it_virt = ticks + p->it_virt_incr;
			send_sig(SIGVTALRM, p, 1);
		}
		p->it_virt_value = it_virt - ticks;
	}
	unsigned long it_prof = p->it_prof_value;
	if (it_prof) {
		if (it_prof <= ticks) {
```

```
                it_prof = ticks + p->it_prof_incr;
                send_sig(SIGPROF, p, 1);
        }
        p->it_prof_value = it_prof - ticks;
}
```

5.3.5 *The scheduler*

The scheduler is responsible for allocating the computing time to the individual processes. Linux supports various scheduling classes which can be selected via the `sched_setscheduler` system call.

On the one hand, there are real-time[12] processes in the scheduling classes `SCHED_FIFO` and `SCHED_RR`.[13] On the other hand, there exists the scheduling class `SCHED_OTHER` which implements a classic Unix scheduling algorithm. However, every real-time process has a higher priority than any process of the scheduling class `SCHED_OTHER`.

The Linux scheduling algorithm is implemented in the *`schedule`* function. It is called at two different points.

1. There are system calls which call the *`schedule`*, usually indirectly by calling *`sleep_on`*.
2. After every system call and after every slow interrupt, the flag `need_resched` is checked by the *`ret_from_sys_call`* routine (just before a process is returned to process mode from system mode). If it is set, the scheduler is also called from here.

The *`schedule`* function consists of three parts.

[12] Real time does not mean 'hard real time' with guaranteed process switching and reation times, but 'soft real time'.

[13] The difference between `SCHED_FIFO` and `SCHED_RR` is that a process of the `SCHED_FIFO` class can run until it relinquishes control or until a process with higher real-time priority wishes to run. A process of the `SCHED_RR` class is also interrupted when its time slice has expired or there are process of the same real-time priority.

1. Those routines that must be called regularly are started.
2. The process with the highest priority is determined. Here, *real-time* processes always take precedence over 'normal' ones.
3. The new process becomes the current process.

```
asmlinkage void schedule(void)
{
      int c;
      struct task_struct * p;
      struct task_struct * prev, * next;
      unsigned long timeout = 0;
      prev = current;
      next = &init_task;
```

First the bottom halves of the interrupt routines are called, then all routines that are registered for the scheduler in the task queue. Both kinds of routines are time-uncritical routines.

```
      if (bh_active & bh_mask) {
            intr_count = 1;
            do_bottom_half();
            intr_count = 0;
      }

      run_task_queue(&tq_scheduler);
```

If `schedule` was called because the current process must wait for an event, it is removed from the run queue. If the current task belongs to the SCHED_RR scheduling class and the task's time slice has expired, it is placed at the end of the run queue and thus after all other read-to-run tasks belonging to the SCHED_RR scheduling class.

```
      if (p;rev->state != TASK_RUNNIGN)
      {
            del_from_runqueue(prev);
      }
      else if (prev->policy == XCHED_RR && prev->counter == 0)
      {
            prev->counter = prev->policy;
            move_last_runqueue(prev);
      }
```

The scheduling algorithm is then carried out; that is, the process in the run queue that has the highest priority is sought.

```
restart_reschedule:
      next = &init_task /* next process */
      next_p = -1000;   /* and its priority */
      for (p = init_task->next _run; \
            p != &init_task; p = p->next_run)
      {
            if (p->policy != SCHED_OTHER)
                  weight = 1000 + p->rt_priority;
            else
                  weight = p->counter;
            if (weight  > next_p)
            {
                  next_p = weight; next = p;
            }
      }
```

If `next_p` is greater than 0, a suitable candidate is found. If `next_p` is less than 0, there is no read-to-run process and we must activate the idle task. In both cases, `next` points to the task to be activated next.

If `next_p` is equal to 0, there are ready-to-run processes, but we must recalculate their dynamic priorities (the value of `counter`). The `counter` value of all other processes are recalculated as well. Then we restart the scheduler, but this time with more chance to success.

```
if (next_p ==0)
{
      for_each_task(p)
      {
            p->counter = (p->counter /2) + p->priority;
      }
      goto restart_rescheduler;
}
```

At this point, either `next` contains a ready-to-run process (`next_p > 0`), or there is no ready-to-run process (`next_p <0`) and `next` points to `init_task`. In any case, the task pointed to by `next` will be activated:

```
        if (prev != next)
              switch_to(prev,next);
        }
```

5.4 Implementing system calls

5.4.1 How do system calls actually work

A system call works on the basis of a defined transition from User Mode to System Mode and in Linux, this is only possibly using interrupts. The interrupt 0x80 is therefore reserved for system calls.

The user will always call a library function to carry out a certain task. This library function (as a rule generated from the `_syscall` macros in `<asm-i386/unistd.h>`) writes its arguments and the number of the system call to defined transfer registers and then triggers the 0x80 interrupt. When the relevant interrupt service routine returns, the return value is read from the appropriate transfer register and the library function terminates.

The actual work of the system calls is taken care of by the interrupt routine that starts at the entry address *`_system_call()`*, held in the `arch/i386/kernel/entry.S` file. This file is written entirely in assembler.

For better readability, it will be illustrated here by a C equivalent.

The parameters `sys_call_num` and `sys_call_args` represent the system call number and its argument.

```
PSEDUO_CODE system_call( int sys_call_num, sys_call_args)
{
 _system_call:
```

First, all the registers for the process are saved.

```
SAVE_ALL; /* macro in entry.S */
```

If `sys_call_num` represents a legal value, the handling routine assigned to the system call number is called. The handling routine is inside the `sys_call_table[]` field. If the process's `PF_TRACESYS` flag is set, it is monitored by its parent process and is taken care of by the `syscall_trace` function (`arch/i386/kernel/ptrace.c`), which amends the state of the current process to `TASK_STOPPED`, SENDS A `sigtrap` signal to the parent process and calls the scheduler. The current process is interrupted until the parent process reactivate it. Now, the parent process has total control over the behavior of the child process.

The actual work of the system call is now complete, but there may still be some administrative tasks to deal with. One of those is to perform bottom half routine. The function *`do_bottom_half`* calls all the bottom halves marked as being active.

If scheduling has been requested (`need_resched!=0`), the scheduler is called. This causes another process to become active. The *`schedule`* function will only return once the process has been reactivated by the scheduler. If signals have been sent to the current process and the process has not blocked receipt of them, they are now processed by *`do_signal`*.

This completes the necessary work, and the system call (or interrupt) returns. All the registers are now restored and the interrupt routine is then terminated by the assembler instruction `iret`.

Example of simple system calls

nice

`nice` takes as its argument a number by which the static priority of the current process is to be modified. The process argument must be checked. *`suser`* is used to check whether

the current process has superuser privileges; this is used because only the superuser is allowed to raise his/her own priority. The new priority for the process is then calculated and a check is made to ensure that the new priority is within a reasonable range.

Chapter 6

Network Implementation

Networking in Linux is a very large topic. It can go from basic local network to the very popular Internet. Recent years, Linux has gained its popularity in providing simple and complex network solutions to both individuals and organizations. In this chapter, we will highlight on the characteristic of the Linux network implentation.

6.1 Background

Almost all processes communicate with each other using sockets and the support of the underlying protocol layers. Linux implements the Internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software connected only with BSD sockets. Supporting this is the INET socket layer that manages the communication end points of the IP based protocols TCP and UDP. The IP layer contains code implementing the Internet Protocol. This code attaches IP headers to transmitted data and decides how to route incoming IP packets to either the TCP or UDP layers. Underneath the IP layer, supporting all of Linux'snetworking are the network devices.

Every network device in the Linux system is represented by a device data structure. This is often referred to as a network interface, meaning the interface to the network rather than to the hardware.

Figure 6.1 illustrates the network layer model described above.

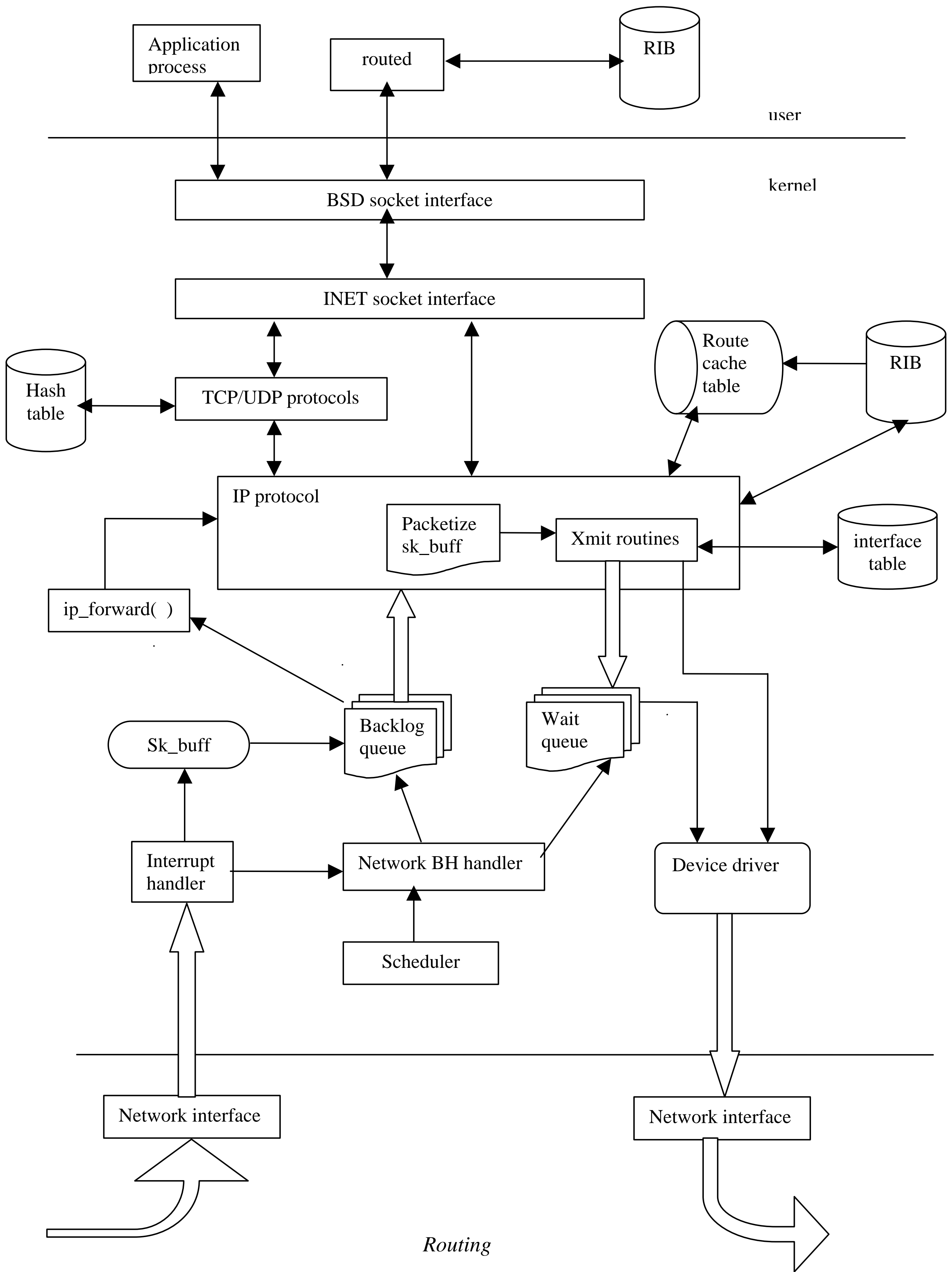


Figure 6.1: Linux network.

6.2 Receiving Packets

6.2.1 Interrupt Handling

When packets arrive at the network interface card, an interrupt is triggered. For an ethernet card, this is handled by an interrupt handler called *ei_interrupt*. Then, it calls ei_*receive* to handle the packet with reference to the network device, and it setups a new sk_buff data structure to hold the new packet.

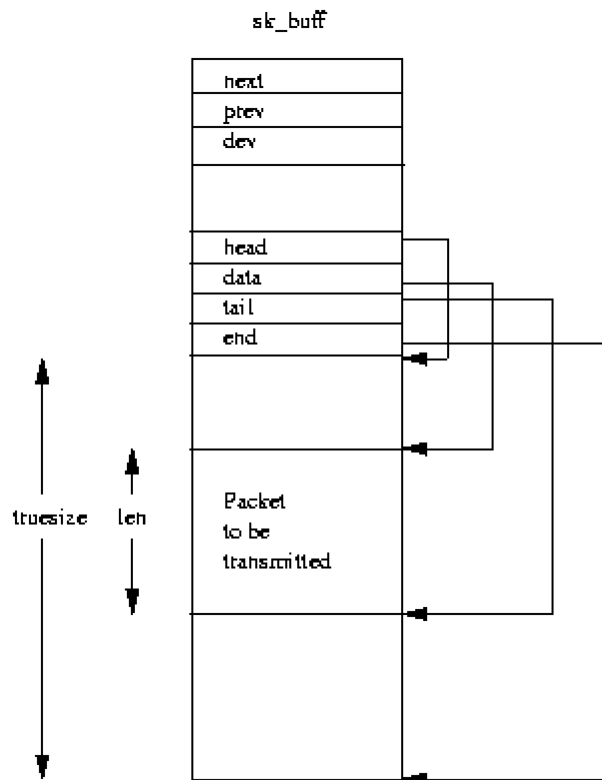


Figure 6.2: sk_buff data structure

Later, *wd_block_input* is called to write the received packet to the new sk_buff. Finally, before the interrupt is finished *netif_rx* in net/core/dev.c is called to add the sk_buff to the backlog queue and set the bh_mask and bh_active flags. There is only one backlog queue in the entire system for receiving

packets. If the backlog queue grows too large then backlog queue will drop any newly received sk_buffs.

6.2.2 Bottom Half Handler

A bottom half handling routine is the way in Linux kernel to defer the "important" part of the interrupt handle to be done later; so that the interrupt would not take up a lot of CPU time. The scheduler activates the network bottom half (BH) handler by calling *do_bottom_half*. This BH handler processes any network packets waiting to be transmitted before processing the backlog queue and determines which protocol layer they are to be passed to. The network bottom half function *net_bh* in net/core/dev.c is invoked when the mask marker is set.

This function also sets the raw pointer of union h in the sk_buff to the beginning of the protocol packet header. The packet_type in the header decides which receive function for the protocol should be called. The packet_type data structure contains the protocol type, a pointer to a network device, a pointer to the protocol's receive data processing routine, and a pointer to the next packet_type.

After all that, *ip_rcv* in net/ipv4/ip_input.c is called. It checks the packet's header for correctness and handling routines for the IP options. If the packet is for the local host then it passes to the upper layer. Otherwise, *ip_forward* is called to rebuild and retransmit the packet. Also if the packet is part of a fragmented data, then *ip_defrag* is called for reassembling.

6.2.3 Protocol Layer Handling

Packets for rebuilding and retransmission are done in the IP layer. When *ip_forward* is called to forward incoming packets, this function checks whether the packet's time to live has been exceeded. Then an exit route is worked out. And now a new packet is now constructed, consisting of the contents of the old one, including the IP

header. The hardware now holds the address of the next computer on the packet's route to its destination.

If the packet is to be passed to the upper level then the raw pointer is now set to the start of the header for the next protocol. If the packet is for TCP then *tcp_rcv* is called. Afterward, *get_tcp_sock* is called to determine which port number and INET socket is for and pass the packet there.

If the incoming packet is a fragment, the IP layer creates a new ipq data structure when the fragment is first received. Then it is put in the IP queue waiting for assembly. When all fragments have been received, they are combined into a single `sk_buff`. There is a timer for this assembly; if it expires, then the packet is marked as lost.

After all that, *tcp_data* is called to check if fresh data is received and discard duplicates. The TCP and UDP layers must do a hash lookup in the socket hash table in order to forward the received packet to the correct INET socket. When an address is bounded to a socket, this information is stored in a hash table. UDP has one hash table while TCP has several.

6.2.4 Socket Layer Handling

After all the protocol layers have finished with the received packet, INET socket's *data_ready* is called to wake up all the processes waiting at the socket. IP address bound to socket is saved in its `recv_addr` and `saddr` fields, and they are later used for hash lookup.

BSD socket provides a generic interface to the application processes while the INET socket provides specific interface that manages the communication end points for the TCP & UDP protocols. When an incoming (TCP) packet is received, the TCP layer creates a new socket and copies the incoming `sk_buff` which contains a pointer to the new socket to the receive queue for the listening socket.

Linux supports several classes of socket and these are known as *address families.* This is because each class has its own method of addressing its communications. Linux supports the following socket address families or domains:

UNIX	Unix domain sockets,
INET	The Internet address family supports communications via TCP/IP protocols
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

Stream

These sockets provide reliable two ways sequenced data streams with a guarantee that data cannot be lost, corrupted or duplicated in transit. Stream sockets are supported by the TCP protocol of the Internet (INET) address family.

Datagram

These sockets also provide two ways data transfer but, unlike stream socket, there is no guarantee that the messages will arrive. Even if they do arrive there is no guarantee that they will arrive in order or even not be duplicated or corrupted. This type of socket is supported of UDP protocol of the Internet address family.

Raw

This allows processes direct (hence “raw”) access to the underlying protocols. It is, for example, possible to open a raw socket to an ethernet device and see raw IP data traffic.

Reliable Delivered Messages

These are very like datagram sockets but the data is guaranteed to arrive.

Sequenced Packets

These are like stream sockets except that the data packet sizes are fixed.

Packet
This is not a standard BSD socket type, it is a Linux specific extension that allows process to access packets directly at the device level.

Like the virtual filesystem, Linux abstracts the socket interface with the BSD socket layer being concerned with the BSD socket interface to the application programs which is in turn supported by independent address family specific software. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family. This association is made via cross-linking data structures and tables of address family specific support routines.

When the kernel is configured, a number of address families and protocols are built into the `protocols` vector. Each is represented by its name, for example "INET" and the address of its initialization routine. When the socket interface is initialized at boot time, each protocol's initialization routine is called. For the socket address families, this results in tem registering a set of protocol operations. This is a set of routines, each of which performs a particular operation specific to that address family. The registered protocol operations are kept in the `pops` vector, a vector of pointers to `proto_ops` data structure.

The `proto_ops` data structure consists of the address family type and a set of points to socket operation routines specific to a particular address family. The `pops` vector is indexed by the address family identifier, for example the Internet address family identifier (`AF_INET` is 2).

The INET socket layer supports the Internet address family which contains the TCP/IP protocols. Its interface with BSD socket layer is through the set of Internet address family socket operations which it registers with the BSD socket layer during

network initialization. These are kept in the `pops` vector along with the other registered address families. The BSD socket layer calls the INET layer socket support routines from the registered INET `proto_ops` data structure to perform work for it. For example a BSD socket create request that gives the address family as INET will use the underlying INET socket create function. The BSD socket layer passes the `socket` data structure representing the BSD socket to the INET layer in each of these operations. This linkage can be seen in figure above. It links the `sock` data structure to the BSD socket data structure using the data pointer in the BSD `socket`. This means that subsequent INET socket calls can easily retrieve the `sock` data structure. The `sock` data structure's protocol operations pointer is also set up at creation time and it depends on the protocol requested. If TCP is requested, then the `sock` data structure's protocol operations pointer will point to the set of TCP protocol operations needed for a TCP connection.

6.2.5 Process Handling

After the process has been woken up or if data are already in the buffer on the read call, they are copied to the user space memory. To receive data, a process must call *`read`* which calls the underlying functions: *`sys_read`*, *`sock_read`*, *`inet_rcvmsg`*, and *`tcp_rcvmsg`*.

For routed, it normally listens on UDP socket 520 for routing information packets. When updates apply, routed records the change in its internal table and then update the kernel's routing table via the `ioctl` system call through BSD sockets.

6.3 Transmitting packets

6.3.1 Process Layer

A process calls *`write`* to send a message to another process locally or on another network. Subsequently, *`sock_write`* is called to talk to BSD socket. It searches for the socket structure associated with the inode struct. Then the parameters of the write operation are transferred into a message structure.

6.3.2 Socket Layer

Routes are added and deleted via *ioctl* requests to the BSD socket interface and then they are passed onto the protocol to process. INET layer manipulates routing table by using *ip_rt_ioctl*.

Creating a BSD Socket

The requested address family is used to search the pops vector a matching address family. It may be that a particular address family implemented as a kernel module and, in this case, the kerneld daemon must load the module before we can continue. A new socket data structure is allocated to represent the BSD socket. Actually the socket data structure is physically part of the VFS inode data structure and allocating a socket really means allocating a VFS inode.

The newly created BSD socket data structure contains a pointer to the address family specific socket routines and this is set to the proto_ops data structure retrieved from the pops vector. Its type is set to the socket type requested; one of SOCK_STREAM, SOCK_DGRAM and so on. The address family specific creation routine is called using the address kept in the proto_ops data structure.

A free file descriptor is allocated from the current processes fd vector and the file data structure that it points at is initialized. This includes setting the file operations pointer to point to the set of BSD socket interface and it will in turn pass them to the supporting address family by calling its address family operation routines.

Binding an Address to an INET BSD Socket

In order to be able to listen for incoming internet connection requests, each server must create an INET BSD socket and bind its address to it. The bind operation is mostly

handled within the INET socket layer with some support from the underlying TCP and UDP protocol layers. The socket having an address bound to cannot be being used for any other communication. This means that socket's state must be TCP_CLOSE. The IP address bound to is saved in the sock data structure in the recv_addr and saddr fields. These are used in hash lookups and as the sending IP address respectively.

As packets are being received by the underlying network devices, they must be routed to the correct INET and BSD sockets so that they can be processed. For this reason UDP and TCP maintain hash tables which are used to lookup the addresses within incoming IP messages and direct them to the correct socket/sock pair.

UDP maintains a hash table of allocated UDP ports, the udp_hash table. This consists of pointers to sock data structures indexed by a hash function based on the port number. As the UDP hash table is much smaller than the number of permissible port numbers (udp_hash is only 128 or UDP_HTABLE_SIZE entries long) some entries in the table point to a chain of sock data structures linked together using each sock's next pointer.

TCP is much more complex as it maintains several hash tables. However, TCP does not actually add the binding sock data structure into its hash tables during the bind operation, it merely checks that the port number requested is not currently being used. The sock data structure is added to TCP's hash table during the *listen* operation.

6.4 Making a Connection on an INET BSD Socket

Once a socket has been created and, provided it has not been used to listen for inbound connection requests, it can be used to make outbound connection requests. For connectionless protocols like UDP, this socket operation does not do a whole lob but for connection oriented protocols like TCP it involves building a virtual circuit between two application.

An outbound connection can only be made on an INET BSD socket that is in the right state; that is to say one that does not already have a connection established and one that is not being used for listening for inbound connections. This means that the BSD `socket` data structure must be in state `SS_UNCONNECTED`. The UDP protocol does not establish virtual connection between applications, any messages sent are datagram, one off messages that may or may not reach their destinations. It does, however, support the *connect* BSD socket operation. A connection operation on a UDP INET BSD socket simply set up the addresses of the remote application; its IP address and its IP port number. Additionally it sets up a cache of the routing table entry so that UDP packets sent on this BSD socket do not need to check the routing database again (unless this route becomes invalid). The cache routing information is pointed at from the `ip_route_cache` pointer in the INET `sock` data structure. If no addressing information is given, this cached routing and IP addressing information will be automatically be used for messages sent using this BSD socket. UDP moves the `sock`'s state to `TCP_ESTABLISHED`.

For a connect operation on a TCP BSD socket, TCP must build a TCP message containing the connection information and send it to IP destination given. The TCP message contains information abut the connection, a unique starting message sequence number, the maximum size message that can be managed by the initiating host, the transmit and receive window size and so on. Within TCP, all messages are numbered and the initial sequence number is used as the first message number. Linux chooses a reasonably random value to avoid malicious protocol attacks. As the TCP `sock` is now expecting incoming messages, it is added to the `tcp_listening_hash` so that incoming TCP messages can be directed to this `sock` data structure. TCP also starts timers so that the outbound connection request can be timed out if the target application does not respond to the request.

6.5 *Listening on an INET BSD Socket*

Once a socket has had an address bound to it, it may listen for incoming connection requests specifying the bound addresses. A network application can listen on a socket without first building an address to it; in this case the INET socket layer finds an unused port number (for this protocol) and automatically bind it to the socket. The listen socket function moves the socket into state `TCP_LISTEN` and does any network specific work needed to allow incoming connections.

For UDP socket, changing the socket's state is enough but TCP now adds the socket's `sock` data structure into two hash tables as it is now active. These are the `tcp_bound_hash` table and the `tcp_listening_hash`. Both are indexed via a hash function based on the IP port number.

Whenever an incoming TCP connection request is received for an active listening socket, TCP builds a new `sock` data structure to represent it. This `sock` data structure will become the bottom half of the TCP connection when it is eventually accepted. It also clones the incoming `sk_buff` containing the connection request and queues it onto the `receive_queue` for the listening `sock` data structure. The clone `sk_buff` contains a pointer to the newly created sock data structure.

6.6 *Accepting Connection Requests*

UDP does not support the concept of connections, accepting INET socket connection requests only applies to the TCP protocol as an accept operation on a listening socket causes a new `socket` data structure to be cloned from the original listening `socket`. The accept operation is then passed to the supporting protocol layer, in this case INET to accept any incoming connection requests. The INET protocol layer will fail the accept operation if the underlying protocol, say UDP, does not support connections. Otherwise the accept operation is passed through to the real protocol, in this case TCP. The accept

operation can be either blocking or non-blocking. In the non-blocking case if there are no incoming connections to accept, the accept operation will fail and the newly created `socket` data structure will be thrown away. In the blocking case the network application performing the accept operation will be added to a wait queue and then suspended until a TCP connection request is received. Once a connection request has been received, the `sk_buff` containing the request is discarded and the `sock` data structure is returned to the INET socket layer where it is linked to the new `socket` data structure created earlier. The file descriptor (`fd`) number of the new socket is returned to the network application, and the application can then use that file descriptor in socket operation on the newly created INET BSD socket.

6.7 Protocol Layer

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This making passing data buffers between the protocols difficult as each layer needs to find where its particular protocol header and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses socket buffers or `sk_buffs` to pass data between the protocol layers and network device drivers. `sk_buffs` contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or "methods".

The `sk_buff` has four data pointer, where are used to manipulate and manage the socket buffer's data:

head	points to the start of the data area in memory. This is fixed when the `sk_buff` and its associated data block is allocated.
data	points at the current start of the protocol data. This pointer varies depending on the protocol layer that currently owns the `sk_buff`.
tail	points at the current end of the protocol data. Again, this pointer varies depending on the owning protocol layer.

end	points at the end of the data area in memory. This is fixed when the `sk_buff` is allocated.

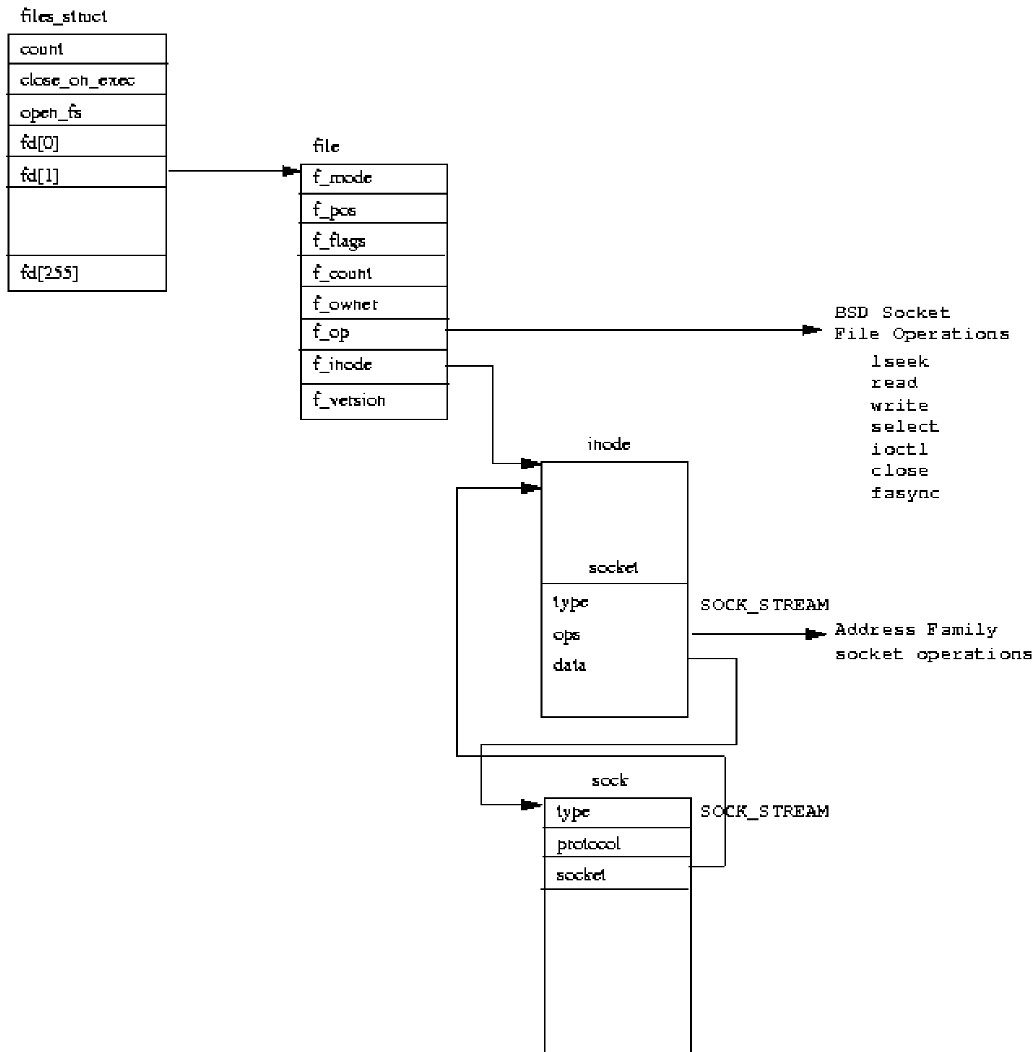


Figure 6.3: socket structure

There are two length field `len` and `truesize`, which describe the length of the current protocol packet and the total size of the data buffer respectively. The `sk_buff` handling code provides standard mechanisms for adding and removing protocol headers

and tails to the application data. These safely manipulate the data, tail and len fields in the sk_buff:

push

This moves the data pointer towards the start of the data area and increments the len field. This is used when adding data or protocol headers to the start of the data to be transmitted,

pull

This moves the data pointer away from the start, towards the end of the data area and decrements the len field. This is used when removing data or protocol headers from the start of the data that has been received,

put

This moves the tail pointer towards the end of the data area and increments the len field. This is used when adding data or protocol information to the end of the data to be transmitted,

trim

This moves the tail pointer towards the start of the data area and decrements the len field. This is used when removing data or protocol tails from the received packet.

Now, *do_tcp_sendmsg* is called. It uses *wmalloc* to allocate memory for the sk_buff. An sk_buff is built to contain the data and various headers are added by the protocol layers as it passes through them. Then *tcp_build_header* and *ip_build_header* are called to build the packet header and initialize the sk_buff.

So far the data to be sent is still in the process space, *memcpy_fromfs* is called to copy the data from there to the TCP segment. Next, *tcp_send_skb* is called to add a variety of protocol-specific information to the packet stored in sk_buff.

IP layer calls *ip_rt_route*, looks at the info in the cache or FIB and determines which route to be taken. This info includes the source IP address, the network device data structure, maximum message size, a reference count, a usage count, a last usage

timestamp, and sometimes a pre-built hardware header. The hardware header is cached with the routes because it must be appended to each IP packet transmitted on this route.

Whenever an IP route is looked up, the route cache is first checked for a matching route and a `rtable` data structure is returned. If there is no matching route in the route cache then the FIB is searched for a route.

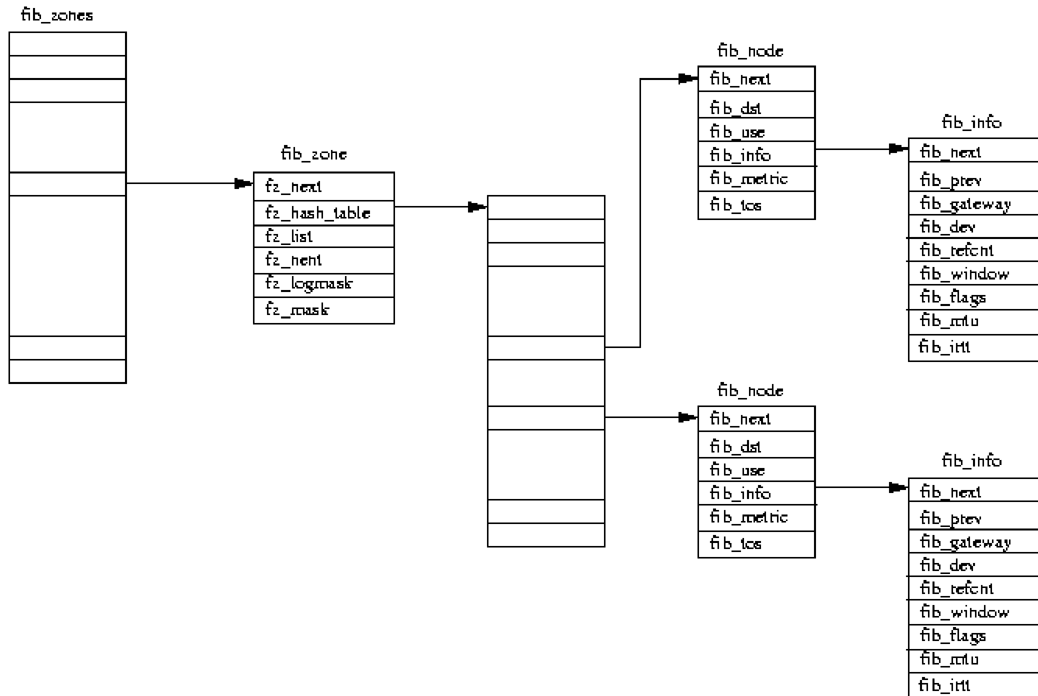


Figure 6.4: FIB structure

Each IP subnet is represented by a `fib_zone` data structure. All routes to the same subnet are described by pairs of `fib_node` and `fib_info` data structures queued onto the `fz_list` of each `fib_zone` data structure. If there are several routes to a subnet, then each route is guaranteed to use a different gateway.

If a route is in the FIB and not in the route cache, then a new entry is generated and added into the route cache for this route. The route cache table is called `ip_rt_hash_table`; it contains pointers to chains of `rtable` data structures. The

index into the routing table is a hash function based on the least significant two bytes of the IP address. The route cache is derived from the forwarding database and represents its commonly used entries.

Later, *ip_queue_xmit* is called to slot the packet into a wait queue (sk_buff link list) for transmission. This function also checks if the computer to be sent is still reachable. The packet is then passed to the function *dev_queue_xmit* which in turn calls *do_dev_queue_xmit* for an interface lookup. If the network device is ready for transmission then the function *hard_start_xmit* is called to transmit the packet immediately. Otherwise, the packet is put into a wait queue.

6.8 Device Layer

Finally, it activates the network device driver for the actual transmission. This calls for the function *ei_start_xmit* to pass the data to the network adapter which in turn sends it to the ethernet. We will describe this procedure in more detail; let us picture it as a packet going from the IP layer to the device layer.

When *dev_queue_xmit* is called, it first checks whether the network device is a software device, ie. no physical hardware such as the loopback device. If it is then just call *hard_start_xmit* which points to *dev_loopback_xmit* to send a packet on a loopback device. Subsequently *netif_rx* is called to place the packet in the backlog queue. Hence, no queuing discipline is involved here.

On the other hand, let's assume that the network device is an ethernet card, then it calls *q->enqueue* which really point to the function *dev->qdisc->enqueue*, to queue the packet for transfer (queuing discipline and traffic control will be explained in the next section with more detail). Then it call *qdisc_wakeup* to notify the device.

Note:

dev_open is called to setup the device interface for use; *device->init* is called only once to initialize the device structure, such as device.pkt_queue, it tells you the number of packets that are queued up on that device.

When you call *dev_activate* to start the device, no queueing discipline is attached to it. Therefore, this function creates a default, i.e. pfifo_fast. pfifo_fast is a 3-band FIFO queue; it is old style, but should be a bit faster than generic prio+fifo combination.

qdisc_base is a list of all installed queueing disciplines and later queueing discipline can be registered or unregistered through *register_qdisc* and *unregister_qdisc*. If the system supports netlink, then a queuing discipline can be created or changed with *qdisc_create* and *qdisc_change*.

Let's again assume that the queuing discipline is of priority type, then *enqueue* is actually point to *prio_enqueue*. It calls *prio_classify* to determine which class to enqueue the packet within the queuing discipline. But, *tc_classify* is the main classifier routine. It scans the classifier chain attached to this qdisc.

Within a queuing discipline, there may be more than one class and each class can be represented by one or more logical bands. Every scheduler will map logical priority bands to real traffic classes, if no other precise mechanism is used to classify packets. In priority qdisc, the maximum defined band is TCQ_PRIO_BANDS or 16. But in sch_prio.c, the default is defined to have 3 priority bands.

Later *prio_enqueue* calls *qdisc->enqueue* to enqueue the packet to the internal queue of the queuing discipline. If it is successful then it increase sch->stats.bytes, sch->stats.packets, sch->q.qlen, and then return 1; otherwise the packet is dropped and increase sch->stats.drops and then return 0.

The priority queuing discipline is initialized by calling *prio_init*, it assigns all the operation links by calling *qdisc_create_dflt*. *qdisc_create_dflt* is responsible for initializing the sch structure and associates its pointers such as sch->ops, sch->enqueue, sch->dequeue, sch->dev.

When *dev_queue_xmit* calls *qdisc_wakeup*, *qdisc_restart* is called subsequently. *qdisc_restart* can also be called by a watchdog timer or by *qdisc_run_queues* from the *net_bh*. There's a periodic watchdog timer, *dev_do_watchdog*, that is responsible for recovery from hard or soft device bugs. If an error occurs it will eventually call *qdisc_restart* to transmit the packet.

dev_init_scheduler is called to initialize the scheduler for a device which associate it with the noop scheduler. The noop scheduler is the the best scheduler, recommended for all interfaces under all circumstances. It is fast and cheap. It has the following functions: noop_enqueue, noop_dequeue, noop_requeue.

qdisc_restart calls *prio_dequeue* which walks through all levels of priority bands and dequeue the first available packet for transferring. Then it calls *dev_queue_xmit_nit* to setup the packet for hardware transmission such as stamp the current time on the it. Next, *dev->hard_start_xmit* (it will point to *ei_start_xmit* for an ethernet card) is called to transmit the packet. Then ei_block_output, an ethernet device driver routine for a specific card, is called to place the packet on the wire. If however the device is busy, then it will call q->ops->requeue for requeuing and return –1.

Chapter 7

Linux Traffic Control

Through the discussion of Linux network implentation, we are ready to move on to the fascinating world of traffic control. Linux traffic control appears as the bottom layer in the overall layer model. Its existence makes Linux an attractive operating system to provide quality of service in many areas. This chapter introduces the basic elements of traffic control: queuing discipline, class, and filter. We examine each by looking at their structures and their implementation. Note that this chapter only provides information on the implentation detail. Readers who are interested in the actual use of them (such as `tc` software) may want to refer to [6].

7.1 Queuing Discipline

Queuing Discipline is one of the fundamental components in Linux traffic control. In Linux kernel, it is defined as `struct Qdisc` and its components are shown below. Linux kernel supports seven kinds of queuing discipline which are Class Based Queuing (CBQ), Clark-Shenker-Zhang scheduler (CSZ), Random Early Detection queue (RED), Stochastic Fairness Queuing discipline, (SFQ), Token Bucket Filter queue (TBF), First-In-First-Out (FIFO), and Priority queuing.

```
struct Qdisc
{
	struct Qdisc_head h;
	int               (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
	struct sk_buff *  (*dequeue)(struct Qdisc *dev);
	unsigned          flags;
#define TCQ_F_BUILTIN    1
#define TCQ_F_THROTTLED  2
	struct Qdisc_ops  *ops;
	struct Qdisc      *next;
	u32               handle;
	atomic_t          refcnt;
	struct sk_buff_head     q;
```

```
        struct device             *dev;

        struct tc_stats           stats;
        unsigned long             tx_timeo;
        unsigned long             tx_last;
        int                 (*reshape_fail)(struct sk_buff *skb, struct Qdisc *q);

        /* This field is deprecated, but it is still used by CBQ
         * and it will live until better solution will be invented.
         */
        struct Qdisc              *__parent;

        char                data[0];
};
```

The following provides a description of each component inside the queuing discipline data structure.

Each queuing discipline provides a set of functions to control its operation and is put together inside `struct Qdisc_ops` ***ops**. This operation pointer is assigned to the queuing discipline when the kernel calls *qdisc_create* to create a new queuing discipline.

A Linux kernel may contain more than one instance and kind of queuing discipline and each instance of a queuing discipline is identified by a 32-bit numbers, which is composed of a major and a minor number. The ID of a queuing discipline always has its minor number equal to zero. This type of numerical identity is called **handle** in Linux terminology.

Queuing disciplines are often organized as a linked list. An example of this is `qdisc_list` which occurs in every Linux network device; `qdisc_list` is responsible to link all queuing disciplines that are used in the device's queuing procedure. The **next** pointer in the queuing discipline data structure (`struct Qdisc`), thus, is pointing to the next queuing discipline in the same `qdisc_list`.

The **enqueue** and **dequeue** pointers are function pointers that are pointing to the functions *sched*_enqueue and *sched*_dequeue (where *sched* can be `prio` if the

queue is priority type or can be `pfifo` if it is FIFO type). These function pointers are also used by the `enqueue` and `dequeue` pointers in the queuing discipline's `ops` data structure. In other words, the `enqueue` and `dequeue` function pointers in both the data sturctures `struct qdisc` and in the `struct Qdisc_ops` are sharing same function.

In each queuing discipline data structure (`struct Qdisc`), there is a header data structure (`struct` **`Qdisc_head`** `h`) pointing to its own `Qdisc`. Some of these header structures are linked together in a linked list (lead by the global variable `qdisc_head`) when their queuing discipline to which it is pointing contain packet buffer waiting to be sent. So, the linked list `qdisc_head` can be considered as the main transmission queue.

One of the main use of `qdisc_head` occurs in sending a packet. When the kernel is going to send a packet, it calls *`dev_queue_xmit`* which invokes the `enqueue` of the device's queuing discipline and subsequently calls `qdisc_wakeup`. `qdisc_wakeup` immediately calls *`qdisc_restart`* which is the main function to poll queuing disciplines and to send packet. `qdisc_restart` first tries to obtain a packet from the queuing discipline of the device, and if it succeeds, it invokes the device's *`hard_start_xmit`* function to actually send the packet. If sending fails for some reason, the packet is returned to the queuing discipline via its `requeue` function. The return value of *`qdisc_restart`* will determine whether the queuing discipline's `qdisc_head` will be enqueued at the head of the main transmission queue. For example, if the return value is greater than 0, then *`qdisc_wakeup`* will check if the queuing discipline's `qdisc_head` has already been in the transmission queue. If it is not, it will enqueue the `qdisc_head` to the transmission queue. The return value of *`qdisc_restart`* is as follows:

```
Returns:  0  - queue is empty.
            >0  - queue is not empty, but throttled.
          <0  - queue is not empty. Device is throttled, if dev->tbusy != 0.
```

At a later time, the main transmission queue will be processed or used via these three functions: *`qdisc_run_queues`*, *`dev_do_watchdog`*, and *`dev_deactivate`*.

As its name implied, *dev_deactivate* is used whenever a device is going to be de-activated. *dev_deactivate* removes the Qdisc_head in the dev's qdisc from the main transmission queue qdisc_head. It changes the device's qdisc to noop_qdisc and calls *qdisc_reset*.

qdisc_run_queues is called from *net_bh* to scan the transmission queue and to transmit all packets for each device. After sending all the packets in a device, the qdisc_head of the device's qdisc is then removed from the main transmission queue. struct sk_buff_head **q** is the actual queue for storing sk_buff. struct tc_stats holds generic queue statistics and these queue statistics are listed in Table 7.1.

Table 7.1:

__u64 bytes;	Number of enqueued bytes
__u32 packets	Number of enqueued packets
__u32 drops	Packets dropped because of lack of resources
__u32 overlimits	Number of throttle events when this flow goes out of allocated bandwidth
__u32 bps	Current flow byte rate
__u32 pps	Current flow packet rate
__u32 qlen	
__u32 backlog	

The data of a queuing discipline is pointing to sched_data.

For example, the data for a priority queuing discipline is prio_sched_data. This prio_sched_data contains information about the number of bands/filters inside the queuing discipline, the filter list, the default mapping between the priority and classes, and the child queuing discipline.

fifo_sched_data is the data element in the FIFO queuing discipline; it contains information about the length of the queue in packet for pfifo and in byte for bfifo.

Table 7.2:

CBQ	cbq_sched_data

CSZ	csz_sched_data
FIFO	fifo_sched_data
RED	red_sched_data
TBF	tbf_sched_data
SFQ	sfq_sched_data

7.1.1 `noop_qdisc, noqueue_qdisc`*: a special kind of queuing discipline*

noqueue_qdisc is very seldom used: it is only used at *dev_activate* to assign to the device's sleeping queue (dev->qdisc_sleeping) when the transmission length of the device is set to 0, which is very unusual.

noqueue_qdisc queuing discipline structure has no function for its enqueue, dequeue, and requeue. However, its queuing operation pointer points to noop_qdisc.

In contrast, Linux uses noop_qdisc more often. When a queuing discipline has been assigned to noop_qdisc, it usually means that this queuing discipline is not initialized and not assigned to a *real* queuing discipline. noop_qdisc is not *real* in the sense that it does nothing useful in its normal operation such as its *enqueue, dequeue,* and *requeue*.

Its *enqueue* simply frees the packet buffer; its *dequeue* and *requeue* returns NULL immediately when they're called; *See register_qdisc, dev_graft_qdisc, dev_activate, dev_deactivate, dev_init_scheduler, dev_shutdown.*

7.1.2 Queuing Discipline operations

The data structure Qdisc_ops provides information on a set of queuing discipline operations.

```
struct Qdisc_ops
{
      struct Qdisc_ops   *next;
      struct Qdisc_class_ops   *cl_ops;
      char               id[IFNAMSIZ];
```

```
        int                 priv_size;

        int                 (*enqueue)(struct sk_buff *, struct Qdisc *);
        struct sk_buff *    (*dequeue)(struct Qdisc *);
        int                 (*requeue)(struct sk_buff *, struct Qdisc *);
        int                 (*drop)(struct Qdisc *);

        int                 (*init)(struct Qdisc *, struct rtattr *arg);
        void                (*reset)(struct Qdisc *);
        void                (*destroy)(struct Qdisc *);
        int                 (*change)(struct Qdisc *, struct rtattr *arg);

        int                 (*dump)(struct Qdisc *, struct sk_buff *);
};
```

Before using a queuing discipline, the queuing discipline must be registered to the kernel by the function *register_qdisc*.

Information about register_qdisc

register_qdisc is to register a queuing discipline. It first checks whether the request queuing discipline has already been registered in the kernel. If this is the case, it returns -EEXIST. Otherwise, it checks if *enqueue, requeue*, and *dequeue* of the request queuing discipline is NULL. Any of these function pointers that was set to NULL is assigned to *noop_qdisc_ops->enqueue*, *noop_qdisc_ops->requeue*, and *noop_qdisc_ops->dequeue* respectively. If this operation succeeds, return 0.

*register_qdisc*is used in initialization function at sched_api.c and other module initialization functio (e.g. *init_module*).

This kind of queuing discipline registration results in placing the queuing discipline operation at the tail of the global linked list in the kernel called qdisc_base. As a result, qdisc_base contains all the queuing discipline operations that can be used by any queuing discipline. An interesting point about queuing discipline is that there is no component inside the structure Qdisc that is used to identify the type of that queuing discipline. Instead, the type of queuing discipline operation to which the queuing

discipline is pointing (by `struct Qdisc_ops *ops` in `struct Qdisc`) identifies its behavior, such as FIFO, and priority.

The first element inside structure `Qdisc_ops` is a **`next`** pointer to another `Qdisc_ops`. This `next` pointer is used to link all queuing discipline operation that has been registered in the kernel.

The next element inside `Qdisc_ops` is another type of operation pointer called `Qdisc_class_ops`. This type of structure contains a set of operation pointers related to a particular class.

The identity of a queuing discipline is stored in `char id[IFNAMSIZ]`. It is usually compared with the option parameter a pointer to `struct rtattr`. The table below shows the ids for each of the queuing discipline.

Table 7.3:

Queuing Discipline	**id**
pFIFO	"pfifo"
bFIFO	"bfifo"
CBQ	"cbq"
CSZ	"csz"
PRIO	"prio"
RED	"red"
SFQ	"sfq"
TBF	"tbf"

The rest is the function pointer for a queuing discipline; this document will use *priority* as an illustration. *enqueue* enqueues a packet with the queuing discipline. If the queuing discipline has classes, the *enqueue* function first selects a class and then invokes the *enqueue* function of the corresponding queuing discipline/class for further enqueuing.

For priority queuing, the *enqueue* function pointer points to the function *prio_enqueue*.

```
static int
prio_enqueue(struct sk_buff *skb, struct Qdisc* sch)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	struct Qdisc *qdisc;

	qdisc = q->queues[prio_classify(skb, sch)];

	if (qdisc->enqueue(skb, qdisc) == 1) {
		sch->stats.bytes += skb->len;
		sch->stats.packets++;
		sch->q.qlen++;
		return 1;
	}
	sch->stats.drops++;
	return 0;
}
```

prio_enqueue enqueues a packet with the queuing discipline. It first finds out the queue to which this packet should enqueue, by calling *prio_classify* function. Then, it calls the *enqueue* function of the corresponding queuing discipline for further enqueuing. *prio_classify* will be discussed later.

dequeue returns the next packet eligible for sending. If the queuing discipline has no packets to send (e.g. because the queue is empty or because they're not scheduled to be sent yet), *dequeue* returns `NULL`.

The *dequeue* function pointer for priority queuing points to the function *prio_dequeue*.

```
static struct sk_buff *
prio_dequeue(struct Qdisc* sch)
{
	struct sk_buff *skb;
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	int prio;
	struct Qdisc *qdisc;

	for (prio = 0; prio < q->bands; prio++) {
		qdisc = q->queues[prio];
		skb = qdisc->dequeue(qdisc);
		if (skb) {
			sch->q.qlen--;
			return skb;
```

```
            }
        }
        return NULL;

}
```

prio_dequeue dequeues the next packet by traversing the child queue from its base priority to its highest utilized priority (`qdisc->data.bands`, and not necessarily the highest priority.) For each child queue, it calls the *dequeue* of the corresponding child queuing discipline for further dequeuing. Once a packet is found to be send, it returns the packet; if there is no packet to be sent, *prio_dequeue* returns `NULL`.

requeue puts a packet back into the queue after dequeuing it with *dequeue*. This differs from *enqueue* in that the packet should be queued at exactly the place from which it was removed by *dequeue*, and that it should not be included in the statistics of cumulative traffic that has passed the queue, because that was already done in the *enqueue* function.

requeue points to *prio_requeue* function.

```
static int
prio_requeue(struct sk_buff *skb, struct Qdisc* sch)
{
        struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
        struct Qdisc *qdisc;

        qdisc = q->queues[prio_classify(skb, sch)];

        if (qdisc->ops->requeue(skb, qdisc) == 1) {
                sch->q.qlen++;
                return 1;
        }
        sch->stats.drops++;
        return 0;
}
```

drop drops one packet from the queue. The function pointer `drop` points to the function `prio_drop`.

```
static int
prio_drop(struct Qdisc* sch)
{
```

```
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	int prio;
	struct Qdisc *qdisc;

	for (prio = q->bands-1; prio >= 0; prio--) {
		qdisc = q->queues[prio];
		if (qdisc->ops->drop(qdisc)) {
			sch->q.qlen--;
			return 1;
		}
	}
	return 0;
}
```

The prio_drop function scans through each of its child queuing discipline and invokes drop function of the corresponding child queuing discipline.

init initializes and configures the queuing discipline.

```
static int prio_init(struct Qdisc *sch, struct rtattr *opt)
{
	static const u8 prio2band[TC_PRIO_MAX+1] =
	{ 1, 2, 2, 2, 1, 2, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 };
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	int i;

	for (i=0; i<TCQ_PRIO_BANDS; i++)
		q->queues[i] = &noop_qdisc;

	if (opt == NULL) {
		q->bands = 3;
		memcpy(q->prio2band, prio2band, sizeof(prio2band));
		for (i=0; i<3; i++) {
			struct Qdisc *child;
			child = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops);
			if (child)
				q->queues[i] = child;
		}
	} else {
		int err;

		if ((err= prio_tune(sch, opt)) != 0)
			return err;
	}
	MOD_INC_USE_COUNT;
	return 0;
}
```

The *prio_init* first initializes all child queuing discipline to be noop_qdisc to avoid leaving some child queuing discipline un-initialized. Afterward, *prio_init*

checks if the `opt` argument passed by the caller function is `NULL` or not. This `opt` specifies the desired architecture of the queuing discipline from the user and *prio_init* calls *prio_tune* to follow the `opt` to set up the queuing discipline. In some occasion, the `opt` may be `NULL` and the default setup of the queuing discipline takes place.

By default, the mapping between priority and queuing discipline is:

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
qdisc	1	2	2	2	1	2	0	0	1	1	1	1	1	1	1	1

For example, a packet with priority equal to 3 (`skb->priority == 3`) will be enqueued into the third child queue (child queue #2). A packet with priority equal to 6 will be enqueue to the first child queue (child queue #0).

After setup this mapping, the `bands` (the number of bands) is set to 3 and the first three child queues are set up by the function *qdisc_create_dflt*. The supplied arguments `sch->dev` and `&pfifo_qdisc_ops` indicates that the default child queues are to be first-in-first-out queues. If all the action executes successfully, *prio_init* returns 0.

`reset` returns the queuing discipline to its initial state. All queues are cleared, timers are stopped, etc. Also, the `reset` functions of all queuing disciplines associated with classes of this queuing discipline are invoked.

```
static void
prio_reset(struct Qdisc* sch)
{
	int prio;
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;

	for (prio=0; prio<q->bands; prio++)
		qdisc_reset(q->queues[prio]);
	sch->q.qlen = 0;
}
```

prio_reset is a very simple function; it scans through the child queue and call *reset* of the corresponding child queuing discipline. Finally, it resets the queuing length value back to zero.

destroy removes a queuing discipline. It removes all classes and possibly also all filters, cancels all pending events and returns all resources held by the queuing discipline (except for the data structure describing the queuing discipline itself).

```
static void
prio_destroy(struct Qdisc* sch)
{
	int prio;
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;

	for (prio=0; prio<q->bands; prio++) {
		qdisc_destroy(q->queues[prio]);
		q->queues[prio] = &noop_qdisc;
	}
	MOD_DEC_USE_COUNT;
}
```

Similar to *prio_replace*, *prio_destroy* scans through the child queue (from priority zero to highest utilized priority) and call *qdisc_destroy* to each of them and then, set each child queue to point to *noop_qdisc*.

dump returns diagnostic data used for maintenance. Typically, the *dump* functions return all sufficiently important configuration and state variables.

```
static int prio_dump(struct Qdisc *sch, struct sk_buff *skb)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	unsigned char	 *b = skb->tail;
	struct tc_prio_qopt opt;

	opt.bands = q->bands;
	memcpy(&opt.priomap, q->prio2band, TC_PRIO_MAX+1);
	RTA_PUT(skb, TCA_OPTIONS, sizeof(opt), &opt);
	return skb->len;

rtattr_failure:
	skb_trim(skb, b - skb->data);
	return -1;
}
```

`prio_dump` is responsible for setting up the message structure `sk_buff` and the responsibility of sending this structure is taken by another function. `prio_dump` first create a structure `tc_prio_qopt` that is used to store the number of bands and the mapping between priority and bands for the current queuing discipline. Once the structure has been set up, it is put into `sk_buff` by the function `RTA_PUT`. Finally, it returns the length of the packet message structure.

The function `tc_fill_qdisc` is the only one that calls `prio_dump`; its main purpose is to set up a reply message back to the upper layer in the kernel.

7.1.3 Examples

After understanding the building blocks and fundamental functions that are used in queuing discipline, it is time to give an example to illustrate how the structures and functions are used together in traffic control.

The first example describes the procedure of sending a packet through the queuing discipline; the second example illustrates how a queuing discipline is created.

The first example uses the priority queuing discipline as an illustration and its structures is shown below in figure 7.1.

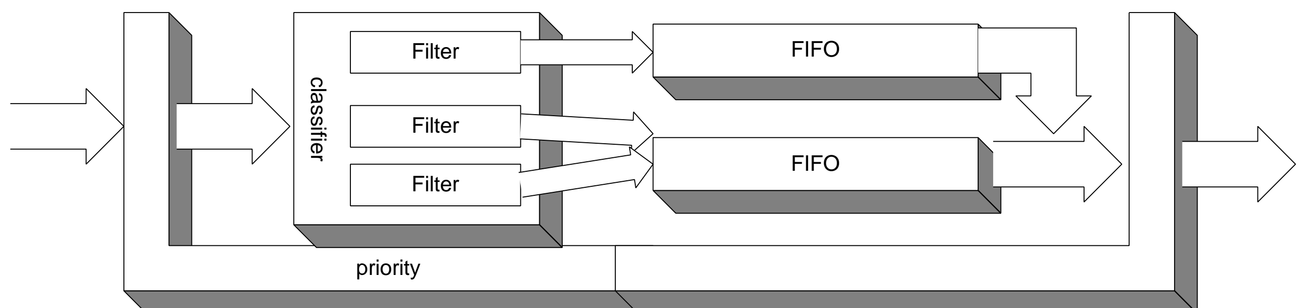


Figure 7.1: Sending a packet through a queuing discipline.

Transmission of a packet starts with the function *dev_queue_xmit* which invokes the *enqueue* of the device's queuing discipline. The above diagram shows the queuing discipline in this example.

At the beginning, *dev->enqueue* will trigger the function *prio_enqueue*. This function subsequently invokes *prio_classify* to find out the child queue to which the packet should enqueue.

prio_classify returns the class associated with the given packet buffer. It first checks if the shortcut for classification can be applied to this packet; in other words, it checks if `skb->priority` contains the class ID of the current queuing discipline. If this is the case, that class is used and no further classification will be attempted. Otherwise, it will check if the filter in the queuing discipline (`q->filter_list`) is empty or not. If it is not empty, it will attempt to perform further classification (`tc_classify`).

tc_classify scans the filter chain attached to the queuing discipline (`q->filter_list`) and finds if there is a filter whose protocol is the same as that defined in the packet buffer (`sk_buff`). If this is the case, then the `classify` function for that filter will be executed (*more information about filter will be given).*

The result of *tc_classify*, (`struct tcf_result`), indicates the bands or child queue into which the packet should put. If there is any error in the classification, the "priority to band" map will be used. For example, if the result of *tc_classify* is out of the band range, the packet will be put into the default queue, which is the first entry of the map, `prio2band[0]`; however, if *tc_classify* cannot even return successfully, then *prio_classify* will uses the packet buffer's priority and the map to decide which child queue to use (`prio2band[band&TC_PRIO_MAX]`).

After *prio_classify* has chosen the child queue to receive the packet, *prio_enqueue* will invoke the *enqueue* function of the corresponding child queue. If the child queue happens to be another priority queuing discipline, then the kernel will follow the above steps all over again. In this example, we will assume that the child queue is a FIFO queue. Therefore, the *enqueue* for FIFO queuing discipline (*pfifo_enqueue*) is called.

pfifo_enqueue is a very simple function because it simply invokes __skb_queue_tail to enqueue the packet to its packet buffer queue (struct sk_buff_head q) and then returns. If the FIFO queue was full, *pfifo_enqueue* will return 0 to indicate an error.

After *pfifo_enqueue* returns, *prio_enqueue* will immediately return. When the enqueuing part of the transmission is done, then the dequeuing part follows. First, *dev_queue_xmit* calls *qdisc_walk* which immediately calls *qdisc_restart*. *qdisc_restart* is responsible to obtain a packet fro the queuing discipline of the device. So, it call the device's *dequeue* function and if the *dequeue* function, *qdisc_restart* will invokes the device's *hard_start_xmit* to do actual sending. The device's *dequeue* is pointing to *prio_dequeue* in our example. As mentioned earlier, *prio_dequeue* dequeues the next packet by traversing the child queues from the base priority to its highest utilized priority. For each queue, it call the *dequeue* of the child queuing discipline until one of them returns successfully.

The *dequeue* of the child queue is *pfifo_dequeue*, which simply __skb_dequeue to dequeue the packet at the head of the queue if there is any.

```
static __inline__ unsigned prio_classify(struct sk_buff *skb, struct Qdisc *sch)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	struct tcf_result res;
	u32 band;

	band = skb->priority;
	if (TC_H_MAJ(skb->priority) != sch->handle) {
```

```
            if (!q->filter_list || tc_classify(skb, q->filter_list, &res)) {
                  if (TC_H_MAJ(band))
                        band = 0;
                  return q->prio2band[band&TC_PRIO_MAX];
            }
            band = res.classid;
      }
      band = TC_H_MIN(band) - 1;
      return band < q->bands ? band : q->prio2band[0];
}
```

```
/* Main classifier routine: scans classifier chain attached
   to this qdisc, (optionally) tests for protocol and asks
   specific classifiers.
 */

extern __inline__ int tc_classify(struct sk_buff *skb, struct tcf_proto *tp, struct
tcf_result *res)
{
      int err = 0;
      u32 protocol = skb->protocol;

      for ( ; tp; tp = tp->next) {
            if ((tp->protocol == protocol ||
                 tp->protocol == __constant_htons(ETH_P_ALL)) &&
                (err = tp->classify(skb, tp, res)) >= 0)
                  return err;
      }
      return -1;
}
```

```
static int
pfifo_enqueue(struct sk_buff *skb, struct Qdisc* sch)
{
      struct fifo_sched_data *q = (struct fifo_sched_data *)sch->data;

      if (sch->q.qlen <= q->limit) {
            __skb_queue_tail(&sch->q, skb);
            sch->stats.bytes += skb->len;
            sch->stats.packets++;
            return 1;
      }
      sch->stats.drops++;
#ifdef CONFIG_NET_CLS_POLICE
      if (sch->reshape_fail==NULL || sch->reshape_fail(skb, sch))
#endif
            kfree_skb(skb);
      return 0;
}
```

We now consider another scenerio in which we are going to create a queuing discipline.

The Linux traffic control uses the function *qdisc_create* to create an instance of queuing discipline. *qdisc_create* creates a new queuing discipline and enqueue the new queuing discipline at the head of the device queuing list (dev->qdisc_list). When *qdisc_create* is invoked, an option argument (tca) is supplied and one of its entries (tca[TCA_KIND-1]) provides information about the kind of queuing discipline that is to be created (Qdisc_ops).

```
struct rtattr *kind = tca[TCA_KIND-1];
struct Qdisc_ops *ops;
ops = qdisc_lookup_ops(kind);
```

Given kind, qdisc_lookup_ops compares it with the queuing discipline operation name/ID (see table 3) and if any matches, qdisc_lookup_ops will return the queuing discipline operation immediately.

After getting the queuing discipline operation, it sets up the parameters inside the queuing discipline such as its device, handle, etc. Afterward, it calls the init function of the new queuing discipline (in this example, the function that will be called is *prio_init*) to set up its data component.

prio_init is used to initialize the queuing discipline's data structure. The TCA_OPTIONS field of the argument tca(tca[TCA_OPTION-1]) specifies how data is initialized.

If the option inside tca[TCA_OPTION-1] is not given by the caller function (*qdisc_create*), *prio_init* will set these parameters by default: the bands will be set to 3, the queuing discipline will have three child queuing discipline and they will be type of pfifo_qdisc_ops and will be set up by the function *qdisc_create_dflt*.

```
if (opt == NULL) {
    q->bands = 3;
    memcpy(q->prio2band, prio2band, sizeof(prio2band));
    for (i=0; i<3; i++) {
        struct Qdisc *child;
        child = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops);
        if (child)
          q->queues[i] = child;
    }
}
```

On the other hands, if the option is provided by the caller function, then `prio_init` calls `prio_tune` to set up the child queuing discipline according to the option.

`prio_tune` gets the information from `RTA_DATA(tca[OPTIONS-1])` which returns a data structure `tc_prio_qopt`. This data structure contains information about the new map between priority and bands as well as new number of bands.

```
struct tc_prio_qopt *qopt = RTA_DATA(opt);
```

`prio_tune` first sets up all the child queuing discipline from the new band number to the maximum priority as `noop_qdisc`. Then, it assigns to the new map and the new band number to the queuing discipline's map and band number. Afterward, it checks if the new mapping has any priority map to a `noop_qdisc`. If there is one or more of such case, it calls `qdisc_create_dflt` to replace the `noop_qdisc` with `pfifo` queuing discipline.

```
/*
   Allocate and initialize new qdisc.

   Parameters are passed via opt.
 */

static struct Qdisc *
qdisc_create(struct device *dev, u32 handle, struct rtattr **tca, int *errp)
{
      int err;
      struct rtattr *kind = tca[TCA_KIND-1];
      struct Qdisc *sch = NULL;
      struct Qdisc_ops *ops;
      int size;
```

```
	ops = qdisc_lookup_ops(kind);
#ifdef CONFIG_KMOD
	if (ops == NULL && tca[TCA_KIND-1] != NULL) {
		char module_name[4 + IFNAMSIZ + 1];

		if (RTA_PAYLOAD(kind) <= IFNAMSIZ) {
			sprintf(module_name, "sch_%s", (char*)RTA_DATA(kind));
			request_module (module_name);
			ops = qdisc_lookup_ops(kind);
		}
	}
#endif

	err = -EINVAL;
	if (ops == NULL)
		goto err_out;

	size = sizeof(*sch) + ops->priv_size;

	sch = kmalloc(size, GFP_KERNEL);
	err = -ENOBUFS;
	if (!sch)
		goto err_out;

	/* Grrr... Resolve race condition with module unload */

	err = -EINVAL;
	if (ops != qdisc_lookup_ops(kind))
		goto err_out;

	memset(sch, 0, size);

	skb_queue_head_init(&sch->q);
	sch->ops = ops;
	sch->enqueue = ops->enqueue; // prio_enqueue
	sch->dequeue = ops->dequeue; // prio_dequeue
	sch->dev = dev;
	atomic_set(&sch->refcnt, 1);
	if (handle == 0) {
		handle = qdisc_alloc_handle(dev);
		err = -ENOMEM;
		if (handle == 0)
			goto err_out;
	}
	sch->handle = handle;

	if (!ops->init || (err = ops->init(sch, tca[TCA_OPTIONS-1])) == 0) { //
prio_init
		sch->next = dev->qdisc_list;
		dev->qdisc_list = sch;
#ifdef CONFIG_NET_ESTIMATOR
		if (tca[TCA_RATE-1])
			qdisc_new_estimator(&sch->stats, tca[TCA_RATE-1]);
#endif
		return sch;
	}
```

```
err_out:
	*errp = err;
	if (sch)
		kfree(sch);
	return NULL;
}
```

```
static int prio_init(struct Qdisc *sch, struct rtattr *opt)
{
	static const u8 prio2band[TC_PRIO_MAX+1] =
	{ 1, 2, 2, 2, 1, 2, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 };
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	int i;

	for (i=0; i<TCQ_PRIO_BANDS; i++)
		q->queues[I] = &noop_qdisc;

	if (opt == NULL) {
		q->bands = 3;
		memcpy(q->prio2band, prio2band, sizeof(prio2band));
		for (i=0; I<3; i++) {
			struct Qdisc *child;
			child = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops);
			if (child)
				q->queues[i] = child;
		}
	} else {
		int err;

		if ((err= prio_tune(sch, opt)) != 0)
			return err;
	}
	MOD_INC_USE_COUNT;
	return 0;
}
```

```
static int prio_tune(struct Qdisc *sch, struct rtattr *opt)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	struct tc_prio_qopt *qopt = RTA_DATA(opt);
	int i;

	if (opt->rta_len < RTA_LENGTH(sizeof(*qopt)))
		return -EINVAL;
	if (qopt->bands > TCQ_PRIO_BANDS || qopt->bands < 2) // # of bands must be at 
least 3.
		return -EINVAL;

	for (I=0; i<=TC_PRIO_MAX; I++) {
		if (qopt->priomap[i] >= qopt->bands)
			return -EINVAL;
	}

	start_bh_atomic();
```

```
        q->bands = qopt->bands;
        memcpy(q->prio2band, qopt->priomap, TC_PRIO_MAX+1);

        for (I= q->bands; i<TCQ_PRIO_BANDS; i++) {
                struct Qdisc *child = xchg(&q->queues[i], &noop_qdisc);
                if (child != &noop_qdisc)
                        qdisc_destroy(child);
        }
        end_bh_atomic();

        for (I=0; i<=TC_PRIO_MAX; i++) {
                int band = q->prio2band[i];
                if (q->queues[band] == &noop_qdisc) {
                        struct Qdisc *child;
                        child = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops);
                        if (child) {
                                child = xchg(&q->queues[band], child);
                                synchronize_bh();

                                if (child != &noop_qdisc)
                                        qdisc_destroy(child);
                        }
                }
        }
        return 0;
}
```

7.2 Classes

Classes can be identified in two ways: (1) by the *class ID*, which is assigned by the user, and (2) by the *internal ID*, which is assigned by the queuing discipline, which has to be unique within a given queuing discipline. The class ID (classid) is of type `u32`, while the internal ID (class) is of type unsinged long. A class is referenced by its internal ID inside the kernel; only the *`get`* and *`change`* functions use the class ID.

Note that multiple class ID's may map to the same internal class ID. In this case, the class ID conveys additional information from the classifier to the queuing discipline or class.

Class ID's are structured like queuing discipline ID's, with the major number corresponding to their instance of the queuing discipline, and the minor number identifying the class within that instance.

Typically, each class "owns" one queue, but it is in principle also possible that several classes share the same queue or even that a single queue is used by all classes of the respective queuing discipline.

Queuing disciplines with classes provide the following set of functions to manipulate classes.

graft attaches a new queuing discipline to a class and returns the previously used queuing discipline.

```
static int prio_graft(struct Qdisc *sch, unsigned long arg, struct Qdisc *new,
                  struct Qdisc **old)
{
      struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
      unsigned long band = arg - 1;

      if (band >= q->bands)
            return -EINVAL;

      if (new == NULL)
            new = &noop_qdisc;

      *old = xchg(&q->queues[band], new);

      return 0;
}
```

leaf returns the queuing discipline of a class.

```
static struct Qdisc *
prio_leaf(struct Qdisc *sch, unsigned long arg)
{
      struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
      unsigned long band = arg - 1;

      if (band >= q->bands)
            return NULL;

      return q->queues[band];
}
```

get looks up a class by its class ID and returns the internal ID. If the class maintains a usage count, *get* should increment it.

```
static unsigned long prio_get(struct Qdisc *sch, u32 classid)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	unsigned long band = TC_H_MIN(classid);

	if (band - 1 >= q->bands)
		return 0;
	return band;
}
```

We think that for priority, class ID is one-to-one relationship with internal ID and therefore, one class contains only one qdisc.

`put` is invoked whenever a class that was previously referenced with `get` is de-referenced. If the class maintains a usage count, `put` should decrement it. If the usage count reaches zero, `put` may remove the class.

```
static void prio_put(struct Qdisc *q, unsigned long cl)
{
	return;
}
```

change changes the properties of a class. *change* is also used to create new classes, where applicable – some queuing discipline have a constant number of classes which are created when the queuing discipline is initialized.

```
static int prio_change(struct Qdisc *sch, u32 handle, u32 parent, struct rtattr
**tca, unsigned long *arg)
{
	unsigned long cl = *arg;
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;

	if (cl - 1 > q->bands)
		return -ENOENT;
	return 0;
}
```

`delete` handles requests to delete a class. It checks if the class is not in use, and de-activates and removes it in this case.

```
static int prio_delete(struct Qdisc *sch, unsigned long cl)
{
```

```
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	if (cl - 1 > q->bands)
		return -ENOENT;
	return 0;
}
```

Comment: Basically, priority queueing discipline does not do much for classes. So priority is a bad example when studying class.

`walk` iterates over all classes of a queuing discipline and invokes a callback function for each of them. This is used to obtain diagnostic data for all classes of a queuing discipline.

```
static void prio_walk(struct Qdisc *sch, struct qdisc_walker *arg)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
	int prio;

	if (arg->stop)
		return;

	for (prio = 0; prio < q->bands; prio++) {
		if (arg->count < arg->skip) {
			arg->count++;
			continue;
		}
		if (arg->fn(sch, prio+1, arg) < 0) { // fn can be qdisc_class_dump or 
check_loop_fn.
			arg->stop = 1;
			break;
		}
		arg->count++;
	}
}
```

```
static int qdisc_class_dump(struct Qdisc *q, unsigned long cl, struct qdisc_walker 
*arg)
{
	struct qdisc_dump_args *a = (struct qdisc_dump_args *)arg;

	return tc_fill_tclass(a->skb, q, cl, NETLINK_CB(a->cb->skb).pid,
			      a->cb->nlh->nlmsg_seq, NLM_F_MULTI, RTM_NEWTCLASS);
}
```

```
static int
check_loop_fn(struct Qdisc *q, unsigned long cl, struct qdisc_walker *w)
{
	struct Qdisc *leaf;
```

```
	struct Qdisc_class_ops *cops = q->ops->cl_ops;
	struct check_loop_arg *arg = (struct check_loop_arg *)w;

	leaf = cops->leaf(q, cl);
	if (leaf) {
		if (leaf == arg->p || arg->depth > 7)
			return -ELOOP;
		return check_loop(leaf, arg->p, arg->depth + 1);
	}
	return 0;
}
```

tcf_chain returns a pointer to the anchor of the list of filters associated with a class. This is used to manipulate the filter list.

```
static struct tcf_proto ** prio_find_tcf(struct Qdisc *sch, unsigned long cl)
{
	struct prio_sched_data *q = (struct prio_sched_data *)sch->data;

	if (cl)
		return NULL;
	return &q->filter_list;
}
```

bind_tcf binds an instance of a filter to the class. *bind_tcf* is usually identical to *get*, except when the queuing discipline needs to be able to explicitly refuse class deletion such as in CBQ.

```
static unsigned long prio_bind(struct Qdisc *sch, unsigned long parent, u32 classid)
{
	return prio_get(sch, classid);
}
```

unbind_tcf removes an instance of a filter from the class. *unbind_tcf* is usually identical to *put*.

For priority class, *prio_class_ops* defines *unbind_tcf* function pointer to use the function *prio_put*.

dump_class returns diagnostic data, like *dump* does for queuing discipline.

```
static int prio_dump_class(struct Qdisc *sch, unsigned long cl, struct sk_buff *skb,
```

```
                     struct tcmsg *tcm)
{
      struct prio_sched_data *q = (struct prio_sched_data *)sch->data;

      if (cl - 1 > q->bands)
            return -ENOENT;
      if (q->queues[cl-1])
            tcm->tcm_info = q->queues[cl-1]->handle;
      return 0;
}
```

prio_dump_class, the *dump_class* for priority, assigns the handle (class ID) of the specified child queue to the tc message's information field (*tcm->tcm_info*).

7.3 Filters

Filters are used by a qdisc to assign incoming packets to one of its classes. This happens during the enqueue operation of the queuing discipline.

Filters are kept in filter lists which can be maintained per queuing discipline or per class. Filters in a filter list are ordered by priority (u32 prio inside struct tcf_proto), in ascending order. In addition, they are distinguished by the protocols described below. These protocol numbers (u32 protocol inside struct tcf_proto) are also used in skb->protocol and they are defined in include/linux/if_ether.h. Filters under the same protocol must have different priorities on the same filter list.

```
              Some defined Ethernet protocol numbers in if_ether.h:

#define ETH_P_LOOP      0x0060            /* Ethernet Loopback packet   */
#define ETH_P_ECHO      0x0200            /* Ethernet Echo packet       */
#define ETH_P_PUP 0x0400            /* Xerox PUP packet           */
#define ETH_P_IP  0x0800            /* Internet Protocol packet   */
#define ETH_P_X25 0x0805            /* CCITT X.25                 */
#define ETH_P_ARP 0x0806            /* Address Resolution packet  */
#define ETH_P_DEC       0x6000          /* DEC Assigned proto          */
#define ETH_P_DNA_DL    0x6001          /* DEC DNA Dump/Load           */
#define ETH_P_DNA_RC    0x6002          /* DEC DNA Remote Console      */
#define ETH_P_DNA_RT    0x6003          /* DEC DNA Routing             */
#define ETH_P_LAT       0x6004          /* DEC LAT                     */
#define ETH_P_DIAG      0x6005          /* DEC Diagnostics             */
#define ETH_P_CUST      0x6006          /* DEC Customer use            */
#define ETH_P_SCA       0x6007          /* DEC Systems Comms Arch      */
#define ETH_P_RARP      0x8035            /* Reverse Addr Res packet    */
#define ETH_P_ATALK     0x809B            /* Appletalk DDP        */
#define ETH_P_AARP      0x80F3            /* Appletalk AARP       */
#define ETH_P_IPX 0x8137            /* IPX over DIX               */
#define ETH_P_IPV6      0x86DD            /* IPv6 over bluebook         */
```

A filter may control its internal elements which are referenced by handles. However, these handles are not split into major and minor numbers. A zero always refer to the filter itself. Classes are selected in *enqueue* of the qdisc by invoking *tc_classify*; it returns `tcf_result` that contains the class ID and maybe the internal ID. After selecting the class, *enqueue* of the inner qdisc is invoked to take care of the next step.

7.3.1 Types of Filters

There are two types of filters: generic (`cls_fw` and `cls_route`) and specific (`cls_u32` and `cls_rsvp`).

When using generic filters, one instance per qdisc can classify (*filter_classify*) packets for all classes. Generic filters set the class field in `tcf_result` to zero and leave the class lookup operation to the qdisc. In kernel version 2.2.5 or higher, `cls_fw` and `cls_route` can become specific filters automatically when explicitly bind classes to them.

```
Filters are controlled via functions defined in struct tcf_proto_ops:

classify: performs the classification and returns one of the TC_POLICE_XX
values.  It also returns the selected class ID.  If the internal class ID is
not omitted, it is returned and stored in struct tcf_result; otherwise the
value zero must be stored in res->class.

get: looks up a filter element by its handle and returns the internal filter
ID or minor number (zero means qdisc).
put: is invoked when a filter element previously referenced with get is no
longer used. (empty in cls_fw)
change: configure new filter or change existing filter.  It registers the
addition of a new filter or filter element to a class by calling bind_tcf.
delete: removes a filter element.
destroy: removes an entire filter.
init: initializes the filter.
walk: It traverse through all elements of a filter and invokes a callback
function (dump) for each of them.
dump: returns diagnostic data for a filter or filter element.
```

Specific filters can have one or more instances of the filter per class. In addition the *filter_classify* uses the packet content information to search through the filter list to find a matching filter element which points to the class with the same ID. Those filter elements are distinguished by an internal filter ID (it is a pointer to the class structure). Unlike classes, filters have no "filter ID". Instead they are identified by the qdisc or class for which they are registered, and their priority among the filters there. Here specific filters do the lookup operation which allows quick retrieval of class information by the queuing discipline.

Currently only `cls_rsvp`, `cls_rsvp6`, and `cls_u32` filters support policing. When *tc_classify* returns policing information, it is up to the qdisc to take appropriate action to handle them. Note that `sch_prio` ignores all policing information.

```
tc_classify() returns the following policing information to the enqueue()
function of a qdisc:

TC_POLICE_OK: no special treatment is requested; treated with high priority.
TC_POLICE_RECLASSIFY: packet exceeds certain bounds and needs
reclassification; treated with low priority.
TC_POLICE_SHOT: packet was selected by filter but violates certain bounds and
needs to be discarded.
TC_POLICE_UNSPEC: no match found; treated with low priority or discard.
```

We will only discuss `u32` specific filters here. This type of filter is stored in hash tables (`struct tc_u_hnode`) of key nodes (`struct tc_u_knode`).

```
struct tc_u_knode
{
      struct tc_u_knode *next;
      u32               handle;
      struct tc_u_hnode *ht_up;
#ifdef CONFIG_NET_CLS_POLICE
      struct tcf_police *police;
#endif
      struct tcf_result res;
      struct tc_u_hnode *ht_down;
      struct tc_u32_sel sel;
};
```

```
struct tc_u_hnode
{
      struct tc_u_hnode *next;
      u32               handle;
      struct tc_u_common      *tp_c;
      int               refcnt;
      unsigned          divisor;
      u32               hgenerator;
      struct tc_u_knode *ht[1];
};
```

Every key node has a set of key and mask (which pointed to by `sel.keys`) of type `u32` that are used in conjunction with packet content information to search through the filter list to find a matching filter for a class.

```
struct tc_u32_sel
{
      unsigned char           flags;
      unsigned char
      offshift;
      unsigned char           nkeys;
      __u16             offmask;
      __u16             off;
      short             offoff;
      short             hoff;
      __u32             hmask;
      struct tc_u32_key keys[0];
};
```

```
struct tc_u32_key
{
      __u32       mask;
      __u32       val;
      int         off;
      int         offmask;
};
```

These filters are distinguished by an internal filter ID (it is a pointer to the class structure). Unlike classes, filters have no "filter ID". Instead they are identified by the qdisc or class for which they are registered, and their priority among the filters there. Specific filters let the qdisc do the operation lookup which allows quick retrieval of class information.

`u32_classify:`

This is a classifier function for the u32 specific filter. To find a matching filter, two sets of conditions must be met: `skb->nh.raw` and `sel.flags` must both be satisfied (explained later). This function has defined a stack (with `maxdepth` of 8 elements) structure within it to hold temporarily found key nodes.

```
static int u32_classify(struct sk_buff *skb, struct tcf_proto *tp, struct
tcf_result *res)
{
      struct {
            struct tc_u_knode *knode;
            u8             *ptr;
      } stack[TC_U32_MAXDEPTH];
      struct tc_u_hnode *ht = (struct tc_u_hnode*)tp->root;
      u8 *ptr = skb->nh.raw;

      struct tc_u_knode *n;
      int sdepth = 0;
      int off2 = 0;
      int sel = 0;
      int i;
```

u32_classify first traverses through all the key nodes within a hash node to check that the first condition is satisfied with some bit-wise operations (in a blackbox) on the key value and offset. If any key is not satisfied on a key node then you go onto the next key node until you find one or that you have reached the end of a hash node.

```
next_ht:
      n = ht->ht[sel];
next_knode:
      if (n) {
            struct tc_u32_key *key = n->sel.keys;
            for (i = n->sel.nkeys; i>0; i--, key++) {
                  if ((*(u32*)(ptr+key->off+(off2&key->offmask))^key-
>val)&key->mask) {
                        n = n->next;
                        goto next_knode;
                  }
            }
```

On the one hand, if the latter case is true then you fall through the POP section and check if there is some found key node on the stack. If some key node is there then you check the second condition else you return –1.

```
/* POP */
if (sdepth--) {
     n = stack[sdepth].knode;
     ht = n->ht_up;
     ptr = stack[sdepth].ptr;
     goto check_terminal;
}
return -1;
```

On the other hand, if the first condition is met then you check the next condition. The condition is that you must have checked all the hash nodes and that the `sel.flags` must satisfied the bit-wise operation. If those are all met than you will save the result in res (`tcf_result`).

```
check_terminal:
                  if (n->sel.flags&TC_U32_TERMINAL) {
                        *res = n->res;
                              return 0;
                  }
                  n = n->next;
                  goto next_knode;
            }
```

If you found a key within a key node that satisfied the first condition and you have not check the other hash nodes, then you must go ahead and do the latter. This is done with PUSH and save them on the stack.

```
/* PUSH */
if (sdepth >= TC_U32_MAXDEPTH)
   goto deadloop;
stack[sdepth].knode = n;
stack[sdepth].ptr = ptr;
sdepth++;
```

Next, you must assign the next hash node and reset some flags. All those flags assignments are done in bit-wise operations.

```
            ht = n->ht_down;
            sel = 0;
            if (ht->divisor)
                  sel = ht->divisor&u32_hash_fold(*(u32*)(ptr+n->sel.hoff),
                        &n->sel);
            if (!(n->sel.flags&(TC_U32_VAROFFSET|TC_U32_OFFSET|TC_U32_EAT)))
                  goto next_ht;
            if (n->sel.flags&(TC_U32_EAT|TC_U32_VAROFFSET)) {
                  off2 = n->sel.off + 3;
                  if (n->sel.flags&TC_U32_VAROFFSET)
                        off2 += ntohs(n->sel.offmask & *(u16*)(ptr+n-
                                >sel.offoff)) >>n->sel.offshift;
                  off2 &= ~3;
            }
            if (n->sel.flags&TC_U32_EAT) {
                  ptr += off2;
                  off2 = 0;
            }
            if (ptr < skb->tail)
                  goto next_ht;
      }
```

After you have gone through all the hash nodes, then you will fall through to POP which will eventually pop out all those matching key nodes on the stack. The last step is obviously check that the second condition is satisfied.

`u32_init`:

u32_init is responsible to initializes a given filter and this function is called only when the kenrel *tc_ctl_tfilter* to create a filter node (tcf_proto). *tc_ctl_tfilter* is not only capable of creating a filter node but also capable of changing and removing a filter node. *u32_init* is only responsible for part of the initialization of a given filter and part of the initialization is done in *tc_ctl_tfilter*.

Instead of restricting ourselves on initialization inside *u32_init*, we will cover the initialization in both areas, that is, inside *tc_ctl_tfilter* and *u32_init*, when creating a new filter node (struct tcf_proto).

7.3.2 Creating a new filter node

Inside tc_ctl_tfilter

Before *tc_ctl_tfilter* decides to create a new filter node, it needs to concern various conditions, such as whether the caller function supplies enough information to create a new filter node and whether there has been a filter node that contains the specified information from the caller function. If the latter condition is the case, there is no need to create a new filter node; instead, we will change the existing filter node. Changing a filter node will be discussed later in this document.

The caller function supplies the information in the form of a message (struct nlmsghdr) that in turns embed another message structure (struct tcmsg). The tcmsg message provides three important types of information: the first one is the filter node's protocol (located at the minor part of the tcmsg's tcm_info); the next information is the filter node's priority (located at the major of the tcmsg's tcm_info); the last information s the parent ID (given in tcmsg's tcm_parent).

Before performing any action, *tc_ctl_tfilter* attempts to look for the filter node with the specified information (piority and protocol). It uses the interface information in the tcmsg (tcm_ifindex) to find out the device associated with this

operation, and within that device, it finds the queuing discipline by the parent ID (tcm_parent). Then, *tc_ctl_tfilter* uses the queuing discipline class operation – *tcf_chain* (*prio_find_tcf* for priority queuing discipline class operation) to obtain the queuing discipline filter list (&qdisc->data.filter_list). Note that the priority queuing discipline must have all of its filters attached to a queuing discipline and not to a class and therefore the parent ID must be only describing a queuing discipline.

After obtaining the filter list, *tc_ctl_tfilter* traverse the list to look for the filter with the same priority as the given one. The filter nodes are sorted according to priority in ascending order. If the filter is not found, the function will create a new filter node. The kind of the new filter node to be created is determined by an option argument. This option argument (struct rtattr **tca) is also passed when *tc_ctl_tfilter* is called and it contains information of how the new filter node should be initialized.

The type of the filter node is determined by its filter operation to which it is attached, and the type is specified in one of the entries in the option argument (a tcf_proto_ops that is pointed by tca[TCA_KIND-1])

Then, *tc_ctl_tfilter* begins to initialize the structure of the new filter node. It assigns the filter type to the new filter node operation pointer (ops). Its queuing discipline pointer points to the queuing discipline associated with this new filter (that is, the queuing discipline that has its filter containing this filter node). Its classifier function pointer (classifier) points to the *classify* function in its filter operation. Finally, its class ID (u32 classid) is assigned to the ID of its associated queuing discipline.

When the structure of the new filter node has been set, *tc_ctl_tfilter* calls *u32_init* to initialize the rest of the filter structure.

Inside *u32_init*

The initialization process is designed to be separated in two parts because different type of filter node has different internal structure. Therefore, *tc_ctl_tfilter* is designed so that it only initialize the generic filter node structure while the initialization function for a particular filter type(such as *u32_init* for u32 filter) is responsible for initializing its internal structure.

To begin the initialization, *u32_init* traverses through the global tc_u_common list (u32_list) to look for the tc_u_common that has its queuing discipline pointer (q) pointing to the same queuing discipline by the new filter node structure.

It then create a new structure root_ht and sets up its parameters including:

- its divisor (divisor, which is number of struct tc_u_knode *ht) is set to 0
- its reference counter (refcnt)is incremented by 1
- its handle (handle) will be assigned by a generator function (gen_new_htid) if the tc_u_common has been found in the first step. Otherwise, its handle will be assigned to 0x80000000.

If tc_u_common was not found, then *u32_init* will create a new tc_u_common and sets its queuing discipline pointer (q) to point to the same queuing discipline that the new filter node is pointing to. The new tc_u_common is then put at the head of the global tc_u_common list (u32_list). Its reference counter will be incremented by 1.

The new tc_u_hnode is then inserted to the head of the tc_u_common's hlist. Afterward, the new tc_u_hnode updates its tp_c pointer so that it points to the tc_u_common to which it is belong after the insertion.

Finally, we update the new filter node so that its data points to the tc_u_common and its root pointing to the new tc_u_hnode.

This reaches to the end of *u32_init* and *tc_ctl_tfilter* will hold the control again. After the full initialization, the new filter node is inserted to the queuing discipline filter list.

u32_change:

u32_change is capable of performing one of the tasks at a time and these three task are:

1. Changing an existing key node (tc_u_knode).
2. Creating a new hash node (tc_u_hnode).
3. Creating a new key node (tc_u_knode).

The action of *u32_change* is according to the available information in the given option argument (struct rtattr **tca). This option argument contains different category of information and *u32_change* invokes *rtattr_parse* to sort out from this information and put the information in a table that has each of its entry specific to each category of information.

The table 7.4 below indicates the categories and the kind of information associated with it.

Table 7.4:

TCA_U32_UNSPEC	a handle of an existing key node/hash node.
TCA_U32_CLASSID	a new class ID.
TCA_U32_HASH	
TCA_U32_LINK	
TCA_U32_DIVISOR	divisor value for a new hash node.
TCA_U32_SEL	
TCA_U32_POLICE	

If the caller function wants to change a key node, it will invokes *u32_change* and pass a reference to the key node to the argument (unsigned long *arg). It will also gives specific information at TCA_U32_LINK and TCA_U32_CLASSID of the option argument (rtattr). When *u32_change* begins, it checks if the argument (arg) refers to a key node. Since this is the case, it will change the information in the key node by the function *u32_set_parms*.

u32_set_parms is a general function that is responsible for changing information inside the specified key node.

If the table entry TCA_U32_LINK has been set, *u32_set_parms* will update the key node's hash node pointer, ht_down. This hash node pointer will point to the new hash node that can be found by the filter ID (handle) specified at TCA_U32_LINK entry in the table. The tc_u_hnode is found from the hash list (hlist) in the specified tc_u_common (key node->ht_up.tp_c). If it is found, the hash node has its reference counter incremented by 1 and is assigned to the key node's ht_down. The old hash node will has its reference counter decremented by 1.

If the table entry TCA_U32_CLASSID has been set, *u32_set_parms* will update the class ID and class in the tcf_result of the key node. The table entry TCA_U32_CLASSID contains the new class ID and this will be assigned to the classid of the tcf_result. The class corresponding to this ID is found by the queuing discipline class operation – bind_tcf. Let assume that the queuing discipline is priority, therefore, the bind_tcf operation is pointing to prio_bind. This operation will take the class ID and return the band index (an index to the child queue, queues[TCQ_PRIO_BANDS]). This band index is then assigned to the class of tcf_result.

After the key node has been set up, *u32_change* will immediately return to the caller function.

If the caller function wants to create a new hash node, it will invoke *u32_change* and passes the target filter node and the divisor value in the table entry TCA_U32_DIVISOR. When *u32_change* begins, it will create a new hash node. The size of the hash node memory depends on the value of the divisor because divisor indicates the number of its associated key node (ht). Also, the memory space for the hash node and its associated key node are located sequentially. As a result, when the memory for the new hash node is allocated, the kernel will allocate sufficient space for both the key node and for its key node in one single kmalloc operation. Then the hash node is set up as follows:

- The hash node has its tp_c pointer pointing to the tc_u_common that is pointed by the data pointer of the given filter node.
- Its reference counter (refcnt) is set to 0.
- Its divisor (divisor) is set to that in the table entry TCA_U32_DIVISOR.
- Its handle (handle) is set to the handle passed from the caller function.
- The hash node is equeued at the head of the tc_u_common's hash list (hlist).

When the set up is done, u32_change returns immediately and puts the address of the new hash node to the argument pointer arg so that the call function can access it later.

If the caller function wants to create a new key node, it will invoke *u32_change* and pass information in the table entries, TCA_U32_HASH and TCA_HASH_SEL. When *u32_change* begins, it gets an ID for a tc_u_hnode at the table entry TCA_U32_HASH. This ID is used to find the tc_u_hnode that we will act upon. If TCA_U32_HASH contains no information, the filter node's root will be used instead. If handle is given when *u32_change* is called. The handle value will be changed to htid |

TC_U32_NODE(handle); otherwise, if the handle is not specified, gen_new_kid will be invoked to create a new handle.

The next step is to examine the table entry TCA_U32_SEL which provides information about an existing tc_u32_sel and its associated keys.

After both table entries are examined, the memory space for the new key node is allocated. The size of the memory space depends on the number of keys specified in tc_u32_sel->nkeys. Therefore, the memory space for the key node and its keys is organized sequentially. Following to memory allocation is to assign the content at the table entry TCA_U32_SEL to the keys of the new key node.

The tc_u_hnode and the handle we mentioned earlier are assigned to ht_up and handle in the key node respectively. Next, *u32_change* invokes *u32_set_parms* to set up the new key node. The *u32_set_parms* is called differently from the call above. The above *u32_set_parms* finds tc_u_hnode from the hash list (hlist) in the specified tc_u_common (key node->ht_up.tp_c). This time, the tc_u_hnode is found from the hash list (hlist) in another tc_u_common (tc_u_hnode->tp_c). Except this, the rest of the operation stays in the same way.

After the set up, the new key node is inserted to the head in one of the hash list (hlist) of the tc_u_hnode. The hash list is chosen by the macro TC_U32_HASH and its handle.

u32_delete:

u32_delete is used to remove a filter element in a filter. The argument (unsigned long arg) supplied from the caller function may either point to a filter element/key node (tc_u_knode) or to a hash node (tc_u_hnode).

If the argument is a pointer to the key node, it will remove the key node in the filter by *u32_delete_key*. *u32_delete_key* removes the key node by removing it in the ht list of the hash node (tc_u_hnode) to which this key node belongs. This hash node can be found at the ht_up pointer of the key node and by using the macro TC_U32_HASH and the key handle, *u32_delete_key* is able to find the hash list that contains the key node. Since the hash list is known, the next step is to traverse the hash list and remove the key node. When the key node is removing from the list, *u32_delete* destroys the key node by *u32_destroy_key*.

u32_destroy_key is a simple function in which it resets the class in the tcf_result of the key node back to 0. As well, it decrements the reference counter of the hash node pointed by the key node's ht_down by 1.

Now, we have discussed what will happen if the argument for *u32_delete* is pointing to a key node. Now, we will discuss what would happen if the argument is pointing to a hash node.

If the argument is a pointer to a hash node, it will decrement that hash node's reference counter by1. If the reference counter reaches to zero, the hash node will be removed via the function call *u32_destroy_node*.

u32_destroy_node is responsible for removing a hash node (*tc_u_hnode*) from the filter. First, it removes and clears all key node inside the hash node via *u32_clear_hnode*. Afterward, it traverses the hash list inside the filter node's data (*tc_u_common*) and removes the hash node from the list.

u32_clear_hnode clears and deletes all the key node (tc_u_knode) by traversing each hash list (ht) in the hash node (tc_u_hnode) and call *u32_destroy_key* (see above) to remove each key one by one.

In the Linux kernel, only *tc_ctl_tfilter* calls *u32_delete* to delete a hash node.

u32_destroy:

u32_destroy is responsible for removing an entire filter. First, *u32_destroy* decrements the reference counters of the nodes pointed by the root and data pointers. The root pointer points to a hash node while the data pointer points to a tc_u_common node. The root pointer is set to NULL and if the reference counter of the "old" root/hash node reaches to 0, *u32_destroy* will call *u32_destroy_hnode* to destroy the hash node (see above).

On the other side, the data pointer is also set to NULL and if the reference counter of the old data/tc_u_common reaches to 0, it will be removed by the global tc_u_common list (u32_list). When tc_u_common is removed, its associated hash node in its hash list (hlist) are also remvoed via u32_clear_hnode.

In the Linux kernel, only *tc_ctl_tfilter* calls *u32_destroy* to remove an entire filer.

u32_walk:

It traverses through all the key nodes of a hash list and invokes a callback function for each of the key nodes. This callback function is assigned to *tcf_node_dump* in /net/cls_api.c; this function will eventually invoke *u32_dump* to place the diagnostic data in the tail space of the skb buffer.

```
static void u32_walk(struct tcf_proto *tp, struct tcf_walker *arg)
{
	struct tc_u_common *tp_c = tp->data;
	struct tc_u_hnode *ht;
	struct tc_u_knode *n;
	unsigned h;
```

If the `arg->stop` is set, you don't have to go any further. In addition, you must check that if the count argument is less than the skip argument because you only invoke the call back function if the count is greater than or equal to skip.

```
for (h = 0; h <= ht->divisor; h++) {
    for (n = ht->ht[h]; n; n = n->next) {
        if (arg->count < arg->skip) {
           arg->count++;
          continue;
        }
        if (arg->fn(tp, (unsigned long)n, arg) < 0) {
           arg->stop = 1;
           return;
        }
        arg->count++;
   }
}
```

`u32_dump`:

This function returns diagnostic data for a filter (hash node) or filter element (key node).

```
static int u32_dump(struct tcf_proto *tp, unsigned long fh,
                 struct sk_buff *skb, struct tcmsg *t)
{
      struct tc_u_knode *n = (struct tc_u_knode*)fh;
      unsigned char      *b = skb->tail;
      struct rtattr *rta;
```

It first check to see if one of the parameter (`unsigned long fh`) is `NULL`, if it is then return the length of the `skb`. There are various diagnostic data to pass to the upper layers and they are all done with the *`RTA_PUT`* function.

```
if (TC_U32_KEY(n->handle) == 0) {
        struct tc_u_hnode *ht = (struct tc_u_hnode*)fh;
        u32 divisor = ht->divisor+1;
        RTA_PUT(skb, TCA_U32_DIVISOR, 4, &divisor);
} else {
        RTA_PUT(skb, TCA_U32_SEL,
               sizeof(n->sel) + n->sel.nkeys*sizeof(struct tc_u32_key),
               &n->sel);
        if (n->ht_up) {
               u32 htid = n->handle & 0xFFFFF000;
               RTA_PUT(skb, TCA_U32_HASH, 4, &htid);
        }
```

```
        if (n->res.classid)
        RTA_PUT(skb, TCA_U32_CLASSID, 4, &n->res.classid);
        if (n->ht_down)
        RTA_PUT(skb, TCA_U32_LINK, 4, &n->ht_down->handle);
```

The *RTA_PUT* is defined as follows:

```
#define RTA_PUT(skb, attrtype, attrlen, data) \
({ if (skb_tailroom(skb) < (int)RTA_SPACE(attrlen)) goto rtattr_failure; \
   __rta_fill(skb, attrtype, attrlen, data); })
```

It will first check if the skb buffer has enough tail space to hold the attribute by comparing their length using some bit-wise operation. If not enough then you will goto rtattr_failure.

```
#define RTA_ALIGN(len) ( ((len)+RTA_ALIGNTO-1) & ~(RTA_ALIGNTO-1) )
#define RTA_ALIGNTO     4
#define RTA_LENGTH(len) (RTA_ALIGN(sizeof(struct rtattr)) + (len))
#define RTA_SPACE(len)  RTA_ALIGN(RTA_LENGTH(len))
```

Here, you'll call *skb_trim* to first check if the skb->len is bigger than the actual data in the skb buffer; if true then you reduce the data space and increase the tail space in the skb buffer. At the end return –1.

However, if there's room then you do a *memcpy* to place the data and attrlen in the skb->tail.

When all the necessary diagnostic data have been delivered, the last thing this function does is to return the length of the skb.

7.4 tc, rtnetlink, netlink, and kernel implementation overview

tc is a user-space program that allows its user to manipulate individual traffic control elements inside the kernel by means of sending control messages. This sending of control message requires the use of *rtnetlink* which based on *netlink*, serves as a communication channel between traffic control elements in user-space and in the kernel.

The following provides an overview of the important data structures in the transmission of this control messages.

nl_table is an array of INET socket linked list (shown in figure 7.2). Its size is set statically to `MAX_LINKS`. Each linked list in this array corresponds to a specific protocol such as `NETLINK_ROUTE`, and each socket stored in this linked list may contains one or more control messages that are waiting to be processed.

rtnetlink_links is an array of pointers to `rtnetlink_link` data structure.[14] Each **rtnetlink_link** data structure corresponds to a `rtnetlink` command such as `RTM_NEWTFILTER` which is a command for adding a new filter to traffic control. Each `rtnetlink_link` contains two internal elements: one has a variable name `doit` that points to a function that will be carried out whenever a particular command (embedded in the control message) is encountered. The other

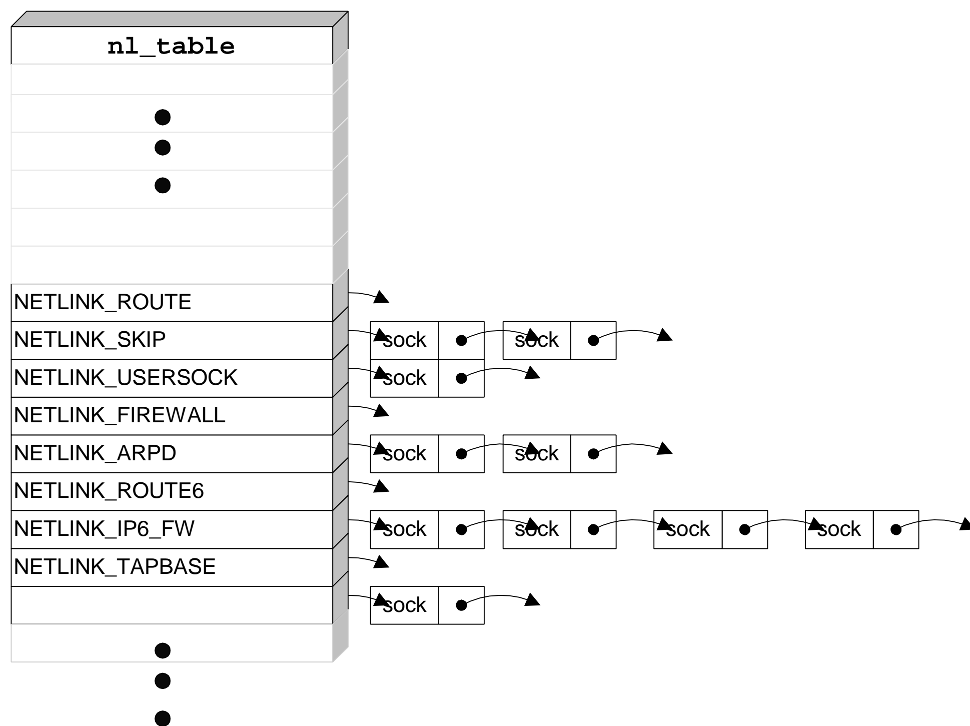


Figure 7.2: `nl_table` data structure

[14] Notice that one has a 's' (i.e. `rtnetlink_links`) and one does not (i.e. `rtnetlink_link`).

element is a variable called `dumpit` that points to a function necessary to clear up data after completion of a command or error.

Each entry in `rtnetlink_links`, in turn, corresponds to a particular family, such as `AF_NETLINK`. `rtnetlink_links` can be viewed as a two-dimensional matrix, so that, its rows corresponds to the family and each column on a row corresponds to command in that family. Figure 2 illustrates the structure of `rtneltink_links and rtnetlink_link`.

7.4.1 Adding a new queuing discipline using `tc`

To understand better the interaction between the various parts of the network implementation inside `tc` and `netlink`, we shall follow the path of the control message which are from `tc` deep into the kernel when the `tc` user adds a new queuing discipline. The use of this particular example does not limit our viewpoint, because other commands (such as removing a filter) follow the same execution path and use the same data

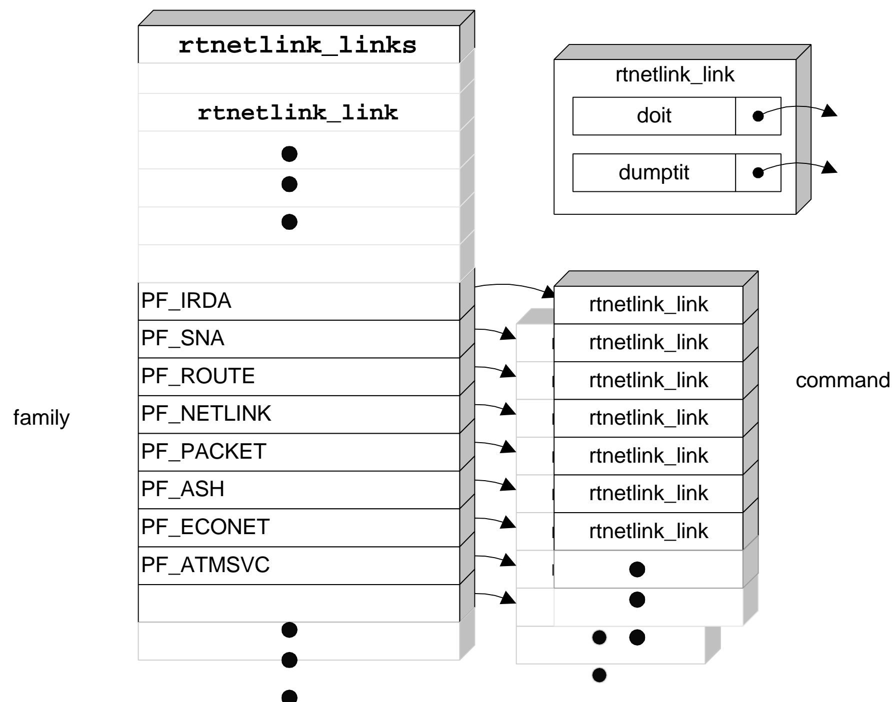


Figure 7.3: `rtnetlink_links` and `rtnetlink_link` data structures

structures.

To change the Linux traffic control configuration, the `tc` user will type the command `tc` followed by its arguments in the Linux shell prompt. The arguments are different each time depending on the action that the user wants. The `tc` command will triggers the execution of *main* in the `tc` program. *main* is responsible for matching the first argument with four different keywords: `qdisc`, `class`, `filter`, and `help`. If the first argument is

`qdisc`, then the function *do_qdisc* will be invoked;
`class`, then the function *do_class* will be invoked;
`filter`, then the function *do_filter* will be invoked;
`help`, then the function *usage* will be invoked to print out help messages.

In this case, *do_qdisc* will be triggered to add a queuing discipline. *do_qdisc* will try to match the next argument with the next set of keywords: `add`, `change`, `replace`, `link`, `delete`, `lst`, `list`, and `help`.

If the argument is either `add`, `change`, `replace`, `delete` or `link`, then the function *tc_qdisc_modify* will be invoked. Otherwise, if the argument is `lst` or `list`, then *do_qdisc* will call *tc_qdisc_list*. Lastly, `help` will call *usage* to print out help messages.

Some reader may be curious about how the tasks at a later time could distinguish `add` or `delete` if both argument (or command) use the same function *tc_qdisc_modify*. The answer is, although they share the same function but they differ in the argument passed to it. For example, the first argument passed to the function specifies the command, such as `RTM_NEWQDISC` for adding a new queuing discipline, and `RTM_DELQDISC` for removing a queuing discipline. In this case, *do_qdisc* will call

tc_qdisc_modify with RTM_NEWQDISC to indicate the action is to add a new queuing discipline.

Once inside *tc_qdisc_modify*, a new data structure req is allocated and initialized. The structure inside req is illustrated in figure 7.4. This data structure is responsible for holding important information in the request message that will be passed into the kernel. The initialization of the data structure include setting

- req.n.nlmsg_len to the size of the tcmsg data structure.
- req.n.nlmsg_flags to logical of NLM_F_REQUEST with the flags that are supplied as argument by the tc user.
- req.n.nlmsg_type to the previous command variable (i.e. in this case, req.n.nlmsg_type is set to RTM_NEWQDISC).
- REQ.T.TCM_FAMILY to AF_UNSPEC.

After initialization has been done, the executing function checks and processes the arguments that follow. Then, it calls the function *rtnl_open* to create an rtnetlink socket. It creates the socket by calling *socket* and passes the arguments AF_NETLINK as its family, SOCK_RAW as its type, and NETLINK_ROUTE as its protocol. The new file descriptor is stored in the fd field of the new data structure named rtnl_handle (see figure 7.4). The next step is to set up a local address structure, struct sockaddr_nl local (inside rtnl_handle), so that the socket can be bind to this address structure. After creating and binding the socket, a request message is sent by calling *rtnl_talk*.

rtnl_talk allocates a message structure named struct msghdr msg which encapsulates the request message req that was setup above. The structure of this message structure is shown in figure 7.3. One of the elements in the message structure is an address structure named nladdr which holds the addressing information; another is a linked list called msg_iov that can hold one or more request message. The request message set up earlier is queued in this linked list. This message structure msg will be

sent via the function call *sendmsg*. *sendmsg* will trigger a system call sys_sendmsg( ) in the kernel space.

sys_sendmsg is responsible for copying the data structures in the user space to the kernel space. Such data structures include the request message req and the message structure msg. The first step to copy msg is to create a new data structure that has the same type with the original msg and name it msg_sys. Then *copy_from_user* will be used to copy each elements inside the data structure from the user space to kernel space. These steps will also applied to the data structures that are required to be duplicated. After copying has been done, the executing function will call *sock_sendmsg* and msg_sys will be passed as an argument.

sock_sendmsg creates a data structure scm_cookie named scm which stands for Socket level Control Message. It then calls *scm_send* to set up the parameters inside scm appropriately. After *scm_send*, *sock_sendmsg* will invoke the current socket's function pointer named sendmsg. In this case, the operation pointer points to the function set, netlink_ops, and its sendmsg operation is *netlink_sendmsg*.

netlink_sendmsg is the point where the information of the message is encapsulated into the sk buffer (also called skbuff). The new skbuff is allocated and the function call *memcpy_from_iovec* copies the message buffer in msg into the skbuff's data area. After setting up of the skbuff, *netlink_sendmsg* will either call *netlink_broadcast* or *netlink_unicast* depending on the value of dstgroups that was set in the upper layer.

netlink_unicast uses the socket's protocol as an index to pick a the corresponding linked list in the global table nl_table, and searches through that linked list for the a sock with the same pid. If blocked mode was defined when the socket is created, then *add_wait_queue* will be called to put the current process into the socket's wait queue and set the process's state to TASK_INTERRUPTIBLE. Then, the executing

function performs some overload checking and if there is no overload, it retrieves the current process back into `TASK_RUNNING` state. Finally, it enqueues the skbuff to the socket's receive queue and call the function pointed by the socket's `data_ready` function pointer. All `rtnelink` socket has its `data_ready` pointed to *rtnetlink_rcv;* so in this case, *rtnetlink_rcv* will be invoked.

rtnetlink_rcv dequeues and processes each skbuff at a time until all the skbuff inside the socket's receive queue has been processed. For each skbuff, the executing function calls *rtnetlink_rcv_skb* to retrieve the data inside the skbuff and put it into another data structure named `nlh`. Then, it passes `nlh` to the function call *rtnetlink_rcv_msg*.

rtnetlink_rcv_msg checks if the incoming message is a request message and if it is not, this function will discard it and return an error message. Then, the function calls `doit` that is stored in the `rtnetlink_link` in the table `rtnetlink_links` at `family` row and `type` column entries. `family` and `type` can be found in skbuff and were set up in the `tc` earlier. `doit` is a function pointer to a handler and in this case, the handler is *tc_modify_qdisc*; this handler is responsible for modifying the queuing discipline.

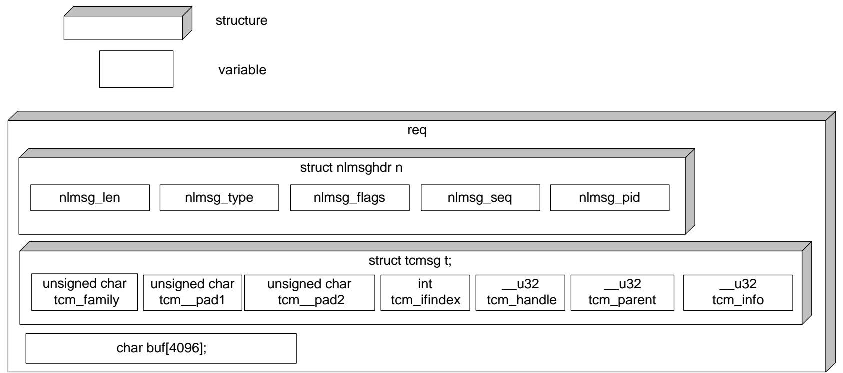
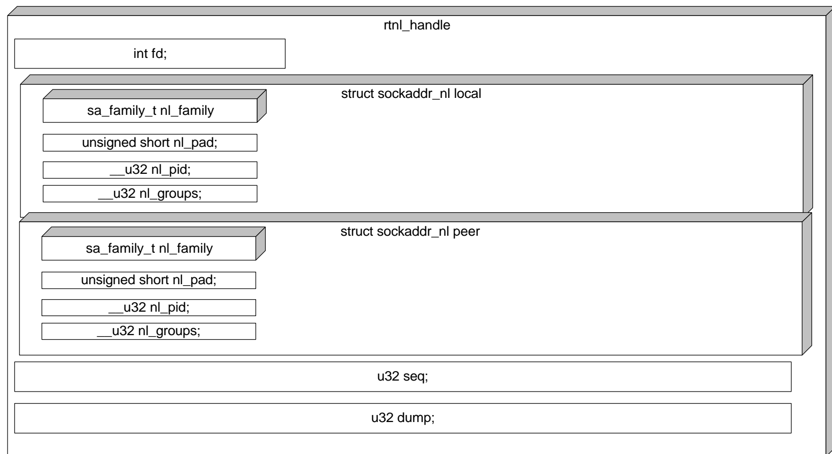
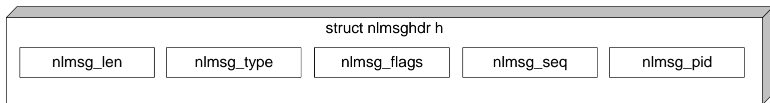
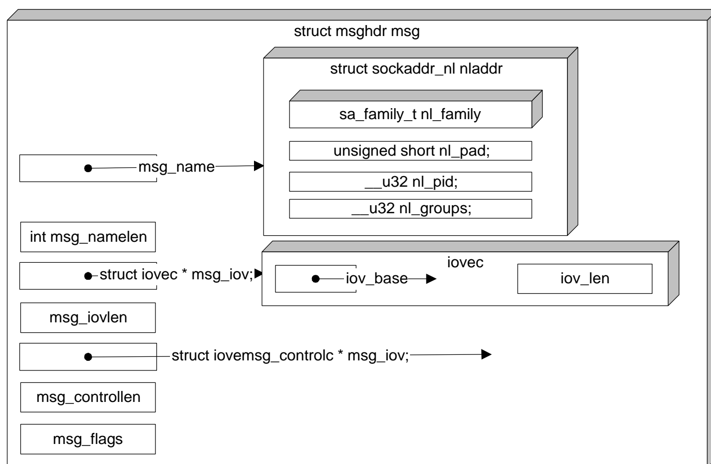
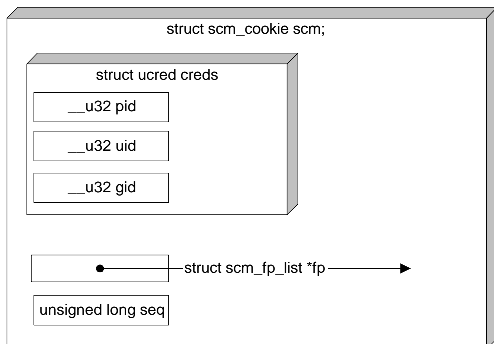


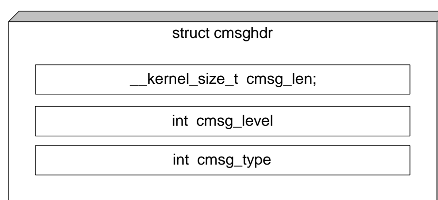
Figure 7.4: important data structures.











7.5 *Modifying queuing discipline inside the kernel*

When a Linux user wishes to modify the traffic control within the kernel, he/she does that by entering an `tc` with arguments that describes the desired result. This `tc` command triggers a system call through which a request message (consists of a netlink message header `nlmsghdr` and a `tc` message `tcmsg`) is created and is passed with the use of `netlink` and `rtnetlink`. During the processing, *`rtnetlink_rcv`* will be called and subsequently, it will call either the following functions:

- *`tc_modify_qdisc`* to create/change `qdisc`.
- *`tc_ctl_tfilter`* to create/change `tfilter`.
- *`tc_get_qdisc`* to delete/get `qdisc`.

`tc_modify_qdisc`

`tc_modify_qdisc` first calls *`dev_get_by_index`* with `tcm->tcm_ifindex` as an argument to find out which device's interface index (`ifindex`) has the same value with the one specified in `tcm_ifindex`. If the qdisc parent (`tcm->tcm_parent`) was specified at `tc` and if its value is not equal to `TC_H_ROOT`, it calls *`qdisc_lookup`* and *`qdisc_leaf`* to find out the parent qdisc `p` and the specific band qdisc `q` that is described by the `tcm_ifindex`. (Basically, what it does is to look at the minor number in the `tcm_ifindex` and then use it as an index to look up the band qdisc array by `p->data.queues[tcm_ifindex]`.) On the other hand, if `tcm->tcm_parent` is equal to `TC_H_ROOT`, then the band qdisc points to the device's `qdisc_sleeping`.

Then the function checks if there is any error, such as, if the previous attempt of finding `q` fails or `tcm->tcm_handle` is empty. In the case that `tcm->tcm_handle` is not empty, then it calls *`qdisc_lookup`* to search the band qdisc `q`, with `dev` and `tcm->tcm_handle` as the arguments,. If it cannot be found, then it jumps to the label `create_n_graft` (see below). Otherwise, it jumps to `graft` (see below).

In the case that tcm->tcm_handle is empty, it checks if q is also empty; if q is also empty, then it jumps to create_n_graft (discussed later). On the other hand, if q is not empty, it jumps to create_n_graft under the following conditions:

1. We are allowed to create/graft only if CREATE and REPLACE flags are set.
2. If EXCL is set, requestor wanted to say, that qdisc tcm_handle is not expected to exist, so that we choose create/graft too.
3. The last case is when no flags are set. Alas, it is sort of hole in API, we cannot decide what to do unambiguously. For now we select create/graft, if user gave KIND, which does not match existing.

The above discussion is about the response of *tc_modify_qdisc* if tcm->tcm_parent was specified at tc by user. However, if tcm->tcm_parent was not specified, then *tc_modfigy_qdisc* does thing differently; it checks whether tcm->tcm_handle is empty or not; if tcm_handle is not empty, it calls *qdisc_lookup* with the arguments dev and tcm_handle, to get the target queuing discipline q.

When either case finishes, it calls *qdisc_change*(q, tca) to change the queuing discipline. *qdisc_change* performs some error checking and then calls *prio_tune*. See below for more discussion for *prio_tune*.

create_n_graft

At create_n_graft, the kernel checks if the netlink flag word (nlmsghdr->nlmsg_flags) has its NLM_F_CREATE bit set to 1; if it is set to 1, it calls *qdisc_create* and passes dev, tcm->tcm_handle, tca, and &err, where dev is the device that we concern, tcm->tcm_handle is the handle identifier (i.e. *major*:*minor*), tca is the argument, and &err is the error return value.

qdisc_create

qdisc_create first looks for the kind of qdisc by looking at the TCA_KIND-1 entry in the argument tca. Then, *qdisc_create* will use this kind to get the qdisc operation (Qdisc_ops). Then, it allocates memory space for queuing discipline and calls *skb_queue_head_init* to initialize the skb_head inside the qdisc. It then sets up the queuing discipline operation, such as enqueue, dequeue, etc. After that, it checks if handle is equal to 0; if it is 0, this means the user has not specifies an ID for the new queuing discipline and therefore, it allocates an unique handle from the space managed by the kernel. Then it assigns the handle value to the queuing discipline's handle. After all of these have been done, it calls ops->init. (In this case, it points to *prio_init*.) If *prio_init* operation returns successfully, then it enqueues this new queuing discipline to the device's qdisc_list.

> *prio_init* receives two parameters, a pointer to the new queuing discipline sch and an argument opt. At first, it sets the children of the new qdisc to noop_qdisc. If the argument opt is checked to be NULL, it then sets the number of bands in the queuing discipline to 3 and sets these three child queuing disciplines to default by calling *qdisc_create_dflt*. (By default, the type of qdisc is type pfifo). If the argument opt is not NULL, then it calls prio_tune.

graft

graft calls *qdisc_graft* function with dev, p, clid, q, and &old_q as its arguments, where p is the parent queuing discipline, clid is the class ID, q is band queuing discipline, and old_q is the old queuing and is set to NULL. graft is to graft qdisc "new" to class "classid" of qdisc "parent" or to device "dev". First it checks if the supplied parent queuing discipline p is empty; if so, call *dev_graft_qdisc*(dev, new) and the return value is assigned to *old. Otherwise, the function calls get from the parent queuing discipline's class operation set. and supplies p, and clid; the return

value is assigned to a type long variable named cl. If the return value is not 0, it calls graft and put of the parent queuing discipline's class operations

```
            if (cops) {
                  unsigned long cl = cops->get(parent, classid);
                  if (cl) {
                        err = cops->graft(parent, cl, new, old);
                        cops->put(parent, cl);
                  }
            }
```

Upon the return of qdisc_graft, it checks if the return value (err) is greater than 0; if the err is greater than 0, this indicates error occurred and if q is not NULL, call *qdisc_destroy* to destroy the q. On the other hands, if there is no error, graft calls qdisc_notify and passes skb, n, clid, old_q, and q. Then if the old_q is not NULL, call *qdisc_destroy*.

dev_graft_qdisc first deactivate the device by calling *dev_deactivate*. Then, put the old qdisc_sleeping to oqdisc variable and checks it the supplied new queuing discipline is empty or not. If it is empty, set the new queuing discipline to noop_qdisc. Then, the function sets the dev's qdisc_sleeping to the new queuing discipline and dev->qdisc to noop_qdisc. Reactivate the device at the end and return the old queuing discipline oqdisc.

prio_get is to get the *minor* class ID from the class id supplied.

prio_graft receives arguments including parent, cl, new, and old, where parent is the parent queuing discipline, cl is the *minor* class ID, new is the previous q queuing discipline, and old is the past queuing discipline that the device was using. This function takes the value of the *minor* class ID as an index to which band the new queuing discipline is to be added.

```
*old = xchg(&q->queues[band], new);
```

qdisc_change

The *qdisc_change* directly calls sch->ops->change, where sch is the queuing discipline. The change function pointer points to *prio_tune* function.

prio_tune

The opt supplied by the caller function (*qdisc_change*) contains many important information such as the number of bands (bands). This number (bands) is assigned to the band inside the qdisc (q->bands). It then checks if the bands ranged from bands to TCQ_PRIO_BANDS is pointing to noop_qdisc. If not, it will destroy whatever it is pointing to and set it to point to noop_qdisc. As a result, these bands are assigned to point to noop_qdisc. Next, it updates the child queuing discipline by checking the bands associated with each priority in the prio2band array. prio2band is an array with its index corresponds to the priority and the content of each element corresponds to the band associated with each priority. For example, prio2band[2] = abc would means that whenever an skb with its priority equal to 2 is encountered, put it in the abc band. To update the child queuing discipline, it checks each child queuing discipline if it is equal to noop_qdisc (which means, it is not initialized.) If so, the function creates a child queuing discipline by the function *qdisc_create_dflt* (pfifo_qdisc_ops as its second argument). This *qdisc_create_dflt* sets up the child qdisc and sets its operator to pfifo_qdisc_ops so it will behave as a FIFO queue.

Chapter 8

Linux Management Information Base (MIB)

The MIB is used by the SNMP manager to manage a network device. The SNMP manager is able to discover all devices in the network and then to browse the contents of each MIB device. The MIB allows the network manager to set up and manage devices remotely.

Although linux does not implement SNMP, it maintains several MIB structures inside the kernel to keep track of network and several protocol statistics. There are two different "types" of MIBs inside the kernel. One is solely for accounting purpose where you can watch these activities in the PROC file system. And the other is used for control purpose which allow you to manage the network adaptor.

There is no special technique to manipulate these MIB structures, they are rather "simple" compared to the analogous FIB structure. The kernel simply updates them through simple assignment statements. Though, the "control" type MIB structure is a little bit more complicated because it has to decipher the meaningful data or set the data definitions according to the ANSI standard.

8.1 Statistics Structures

`struct ip_mib` (it is usually declared as `ip_statistics`) is the MIB structure used to keep track of the SNMP management statistics of a network device. *`snmp_get_info`* (described below) is called from the PROC file system module to output all the protocol statistics used in SNMP to `/proc/net/snmp`.

```
struct ip_mib
```

```
{
      unsigned long     IpForwarding;  /* by default set to No */
      unsigned long     IpDefaultTTL;  /* time to live; set to 64 */
      unsigned long     IpInReceives;
      unsigned long     IpInHdrErrors;
      unsigned long     IpInAddrErrors;
      unsigned long     IpForwDatagrams;
      unsigned long     IpInUnknownProtos;
      unsigned long     IpInDiscards;
      unsigned long     IpInDelivers;
      unsigned long     IpOutRequests;
      unsigned long     IpOutDiscards;
      unsigned long     IpOutNoRoutes;
      unsigned long     IpReasmTimeout;
      unsigned long     IpReasmReqds;
      unsigned long     IpReasmOKs;
      unsigned long     IpReasmFails;
      unsigned long     IpFragOKs;
      unsigned long     IpFragFails;
      unsigned long     IpFragCreates;
};
```

`struct linux_mib` (usually declared as `net_statistics`) is the MIB struture used to hold the network activity information about a network device. *netstat_get_info* is called to output these network statistics to `/proc/net/netstat`.

```
struct linux_mib
{
      unsigned long     SyncookiesSent;
      unsigned long     SyncookiesRecv;
      unsigned long     SyncookiesFailed;
      unsigned long     EmbryonicRsts;
      unsigned long     PruneCalled;
      unsigned long     RcvPruned;
      unsigned long     OfoPruned;
      unsigned long     OutOfWindowIcmps;
      unsigned long     LockDroppedIcmps;
};
```

In order to use *snmp_get_info* and *netstat_get_info* the kernel must register these entries with the inode operation list. For example, to register the snmp inode operation the following function must be called: *proc_net_register*(&proc_net_snmp).

```
static struct proc_dir_entry proc_net_snmp = {
```

```
        PROC_NET_SNMP, 4, "snmp",
        S_IFREG | S_IRUGO, 1, 0, 0,
        0, &proc_net_inode_operations,
        snmp_get_info
};

static struct proc_dir_entry proc_net_netstat = {
        PROC_NET_NETSTAT, 7, "netstat",
        S_IFREG | S_IRUGO, 1, 0, 0,
        0, &proc_net_inode_operations,
        netstat_get_info
};
```

8.2 Usage

When you create an INET socket using `inet_create` inside `af_inet.c`, it does all the socket initialization as necessary and it also assigns the default "time to live" to the socket buffer as follows:

```
sk->ip_ttl=ip_statistics.IpDefaultTTL.
```

As you can see, manipulating the "statistics type" MIB structure inside the linux kernel is very intuitive. there are several functions that manipulates the structure fields by simply increasing or decreasing different fields. Below are a few examples:

```
ip_rcv() changes the statistics of the number of received ip packets as follows:
    ip_statistics.IpInReceives++;

ip_output() requests output as follows:
    ip_statistics.IpOutRequests++;
```

When a user wants to see all the protocol statistics used in SNMP, he simply check the `/proc/net/snmp` directory. This is all done through calling `snmp_get_info` from the PROC file system module. `snmp_get_info` prints out all the SNMP statistics and return the length of the printed buffer.

```
int snmp_get_info(char *buffer, char **start, off_t offset, int length,
int dummy)
{
        extern struct tcp_mib tcp_statistics;
        extern struct udp_mib udp_statistics;
        int len;
```

```
        len = sprintf (buffer,
              "Ip: Forwarding DefaultTTL InReceives InHdrErrors
                  ip_statistics.IpForwarding, ip_statistics.IpDefaultTTL,
                  ip_statistics.IpInReceives,
```

The returned length could be an integer or zero if the length is less than the offset. Then the `start` pointer is changed to point to the beginning of the new buffer section.

```
if (offset >= len)
      {
         *start = buffer;
         return 0;
      }
      *start = buffer + offset;
      len -= offset;
      if (len > length)
         len = length;
      if (len < 0)
         len = 0;
      return len;
}
```

Similarly if a user wants to print network statistics instead *netstat_get_info* is called. It performs the exact routine, so we will not discuss it here.

8.3 Control Structures

Now we will talk about the structures and some functions used to extract the commands sent from other hosts to control the network adaptor. *dfx_ctl_get_stats* gets current MIB objects from a network adapter, then returns FDDI (Fiber Distributed Data Interface) statistics structure as defined in `if_fddi.h`. It takes `dev` (a pointer to the device structure) as the argument.

Since the FDDI statistics structure is still new and the device structure does not have an FDDI-specific get statistics handler, we'll return the FDDI statistics structure as a pointer to an Ethernet statistics structure.

```
struct net_device_stats *dfx_ctl_get_stats()
```

That way, at least the first part of the statistics structure can be decoded properly, and it allows other applications to perform a second cast to decode the FDDI-specific statistics. We'll have to pay attention to this routine as the device structure becomes more mature and LAN media independent.

There are two types of DMA command structures used to maintain the hardware statistics. One always preceded by SNMP_* and the other SMT_*. Both the *_SetRequest and *_SetResponse are the same in SNMP_* and SMT_*, however, SMT_* command structures are actually used to control the network adaptor.

SNMP_Set Request

```
typedef struct
      {
      PI_UINT32          cmd_type;
      struct
      {
            PI_UINT32   item_code;
            PI_UINT32   value;
            PI_UINT32   item_index;
            } item[PI_CMD_SNMP_SET_K_ITEMS_MAX];
      } PI_CMD_SNMP_SET_REQ;
```

SMT_MIB_Set Request

```
typedef struct
      {
      PI_UINT32   cmd_type;
      struct
            {
            PI_UINT32   item_code;
            PI_UINT32   value;
            PI_UINT32   item_index;
            } item[PI_CMD_SMT_MIB_SET_K_ITEMS_MAX];
      } PI_CMD_SMT_MIB_SET_REQ;
```

SNMP_Set Response

```
typedef struct
      {
      PI_RSP_HEADER   header;
      } PI_CMD_SNMP_SET_RSP;
```

SMT_MIB_Set Response

```
typedef struct
      {
      PI_RSP_HEADER   header;
      } PI_CMD_SMT_MIB_SET_RSP;
```

SMT_MIB_Get Request

```
typedef struct
        {
        PI_UINT32  cmd_type;
        } PI_CMD_SMT_MIB_GET_REQ;
```

The above command structures contain some high-level information of what is needed to be done, although the structures below have more meaningful data regarding the control and statistics about the hardware.

SMT_MIB_Get Response

This command structure retrieves definitions from the appropriate SMT_MIB_SET commands sent by other hosts. These item and group code definitions are taken from the ANSI FDDI SMT Rev. 7.3.

```
typedef struct
        {
        PI_RSP_HEADER  header;

        /* SMT GROUP */
        PI_STATION_ID      smt_station_id;
        PI_UINT32          smt_op_version_id;
        PI_UINT32          smt_hi_version_id;
        PI_UINT32          smt_lo_version_id;
        PI_UINT32          smt_user_data[8];
        PI_UINT32          smt_mib_version_id;
        PI_UINT32          smt_mac_ct;
        PI_UINT32          smt_non_master_ct;
        PI_UINT32          smt_master_ct;
        PI_UINT32          smt_available_paths;
        PI_UINT32          smt_config_capabilities;
        PI_UINT32          smt_config_policy;
        PI_UINT32          smt_connection_policy;
        PI_UINT32          smt_t_notify;
        PI_UINT32          smt_stat_rpt_policy;
        PI_UINT32          smt_trace_max_expiration;
        PI_UINT32          smt_bypass_present;
        PI_UINT32          smt_ecm_state;
        PI_UINT32          smt_cf_state;
        PI_UINT32          smt_remote_disconnect_flag;
        PI_UINT32          smt_station_status;
        PI_UINT32          smt_peer_wrap_flag;
        PI_CNTR            smt_msg_time_stamp;
        PI_CNTR            smt_transition_time_stamp;

        /* MAC GROUP */
        PI_UINT32          mac_frame_status_functions;
        PI_UINT32          mac_t_max_capability;
        PI_UINT32          mac_tvx_capability;
        PI_UINT32          mac_available_paths;
        PI_UINT32          mac_current_path;
        PI_LAN_ADDR        mac_upstream_nbr;
        PI_LAN_ADDR        mac_downstream_nbr;
```

```
PI_LAN_ADDR         mac_old_upstream_nbr;
PI_LAN_ADDR         mac_old_downstream_nbr;
PI_UINT32           mac_dup_address_test;
PI_UINT32           mac_requested_paths;
PI_UINT32           mac_downstream_port_type;
PI_LAN_ADDR         mac_smt_address;
PI_UINT32           mac_t_req;
PI_UINT32           mac_t_neg;
PI_UINT32           mac_t_max;
PI_UINT32           mac_tvx_value;
PI_UINT32           mac_frame_error_threshold;
PI_UINT32           mac_frame_error_ratio;
PI_UINT32           mac_rmt_state;
PI_UINT32           mac_da_flag;
PI_UINT32           mac_unda_flag;
PI_UINT32           mac_frame_error_flag;
PI_UINT32           mac_ma_unitdata_available;
PI_UINT32           mac_hardware_present;
PI_UINT32           mac_ma_unitdata_enable;

/* PATH GROUP */
PI_UINT32           path_configuration[8];
PI_UINT32           path_tvx_lower_bound;
PI_UINT32           path_t_max_lower_bound;
PI_UINT32           path_max_t_req;

/* PORT GROUP */
PI_UINT32           port_my_type[PI_PHY_K_MAX];
PI_UINT32           port_neighbor_type[PI_PHY_K_MAX];
PI_UINT32           port_connection_policies[PI_PHY_K_MAX];
PI_UINT32           port_mac_indicated[PI_PHY_K_MAX];
PI_UINT32           port_current_path[PI_PHY_K_MAX];
PI_UINT32           port_requested_paths[PI_PHY_K_MAX];
PI_UINT32           port_mac_placement[PI_PHY_K_MAX];
PI_UINT32           port_available_paths[PI_PHY_K_MAX];
PI_UINT32           port_pmd_class[PI_PHY_K_MAX];
PI_UINT32           port_connection_capabilities[PI_PHY_K_MAX];
PI_UINT32           port_bs_flag[PI_PHY_K_MAX];
PI_UINT32           port_ler_estimate[PI_PHY_K_MAX];
PI_UINT32           port_ler_cutoff[PI_PHY_K_MAX];
PI_UINT32           port_ler_alarm[PI_PHY_K_MAX];
PI_UINT32           port_connect_state[PI_PHY_K_MAX];
PI_UINT32           port_pcm_state[PI_PHY_K_MAX];
PI_UINT32           port_pc_withhold[PI_PHY_K_MAX];
PI_UINT32           port_ler_flag[PI_PHY_K_MAX];
PI_UINT32           port_hardware_present[PI_PHY_K_MAX];

/* GROUP for things that were added later, so must be at the end. */
PI_CNTR                   path_ring_latency;

} PI_CMD_SMT_MIB_GET_RSP;
```

Some notes about traffic control in linux:

All the traffic control objects have 32bit identifiers called "handles". They consist of two fields: major and minor numbers. For example, qdisc handles always have minor number equal to zero, classes have major equal to parent qdisc major, and minor uniquely identifying class inside qdisc.

There are three new traffic control elements added to the 2.2.x kernel:

1. `sch_dsmark` – queuing discipline to extract and to set the DSCP

- if `set_tc_index` is set, it retrieves the content of the DS field and stores it in `skb->tc_index`.
- it invokes a classifier and stores the class ID returned in `skb->tc_index` else `default_index` is used instead.

2. `cls_tcindex` – the classifier
3. `sch_gred` – a queuing discipline which supports mulitple drop priorities and buffer sharing

CBQ(class based queuing) is an alternative to the combination of FIFO+priority+TBF.

Appendix A

This section discusses the four mechanisms Zebra daemon can use to communicate with the Linux kernel.

1. ioctl

`ioctl` is a very traditional way for reading or writing kernel information. It can be used for looking up interface and get or set interface address, flags, MTU and other information. Also `ioctl` can insert and delete kernel routing table information. It will be available on most platform which zebra supports but it is a little bit ugly so if more better method is supported by the kernel zebra uses that.

2. sysctl

`sysctl` can lookup kernel information by MIB (Management Information Base) syntax. Normally it only provides a way of getting information from the kernel. So if one wants to change kernel information then `ioctl` is required.

3. proc filesystem

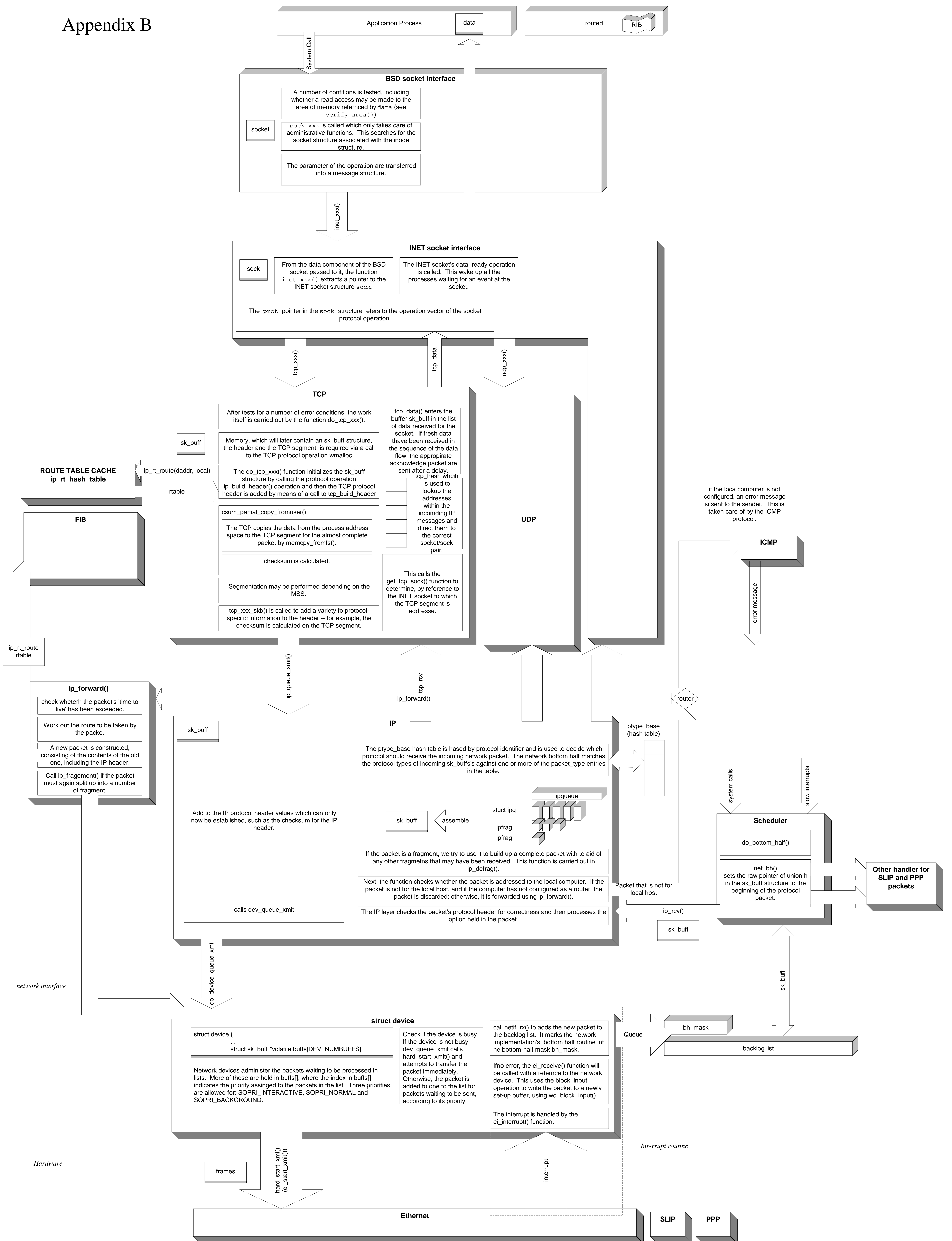
`proc` filesystem provides an easy way of getting kernel information. It can only display useful information about the kernel such as currently Running processes.

4. Netlink

On recent Linux kernel (2.0.x and 2.2.x), there is a kernel/user communication support which called `netlink`. It can make it possible to have asynchronous communication between the kernel and Zebra such as routing socket over BSD systems.

Before you use this feature, you must select kernel support option Kernel/User network link driver and Routing messages. Today, `/dev/route` special device file is obsolete. Netlink communication is done by read/write over netlink socket. You can use netlink as a dynamic routing update channel between Zebra and kernel.

Appendix B



tc

main

argv?

qdisc

arg1

do_qdisc

call tc_qdisc_list() if there is no arg2.

Call tc_qdisc_modify() if arg2 equal to add, change, replace, link, and delete.

Call tc_qdisc_get() if arg2 is equal to get. <not implemented in tc>

Call tc_qdisc_list() if arg2 is either equal to list, show, or lst.

Call usage() if arg2 is eql to help.

Print error message.

class

filter

help

qdisc_get()

qdisc_get()

tc_qdisc_modify()

set up the tc req message

rtnl_open(), create a netlink socket and bind it to the address structure.

Find the device interface.

rtnl_talk()

rtnl_talk

allocates a message structure, `msg`.

`msg`

sendmsg()

application

kernel

sys_sendmsg

copy all the necessary data structure from user space to kernel space by copy_from_user and verify_iovec

call sock_sendmsg()

`sys_msg`

enscapulating

sock_sendmsg

create a socket level control message (scm) and calls scm_send to set up its parameter.

call netlink_sendmsg

netlink_sendmsg

create a sk_buff and use memcpy_from_iovec to copy the msg's iovec to the data area of sk_buff.

`sk_buff`

dstgroups

netlink_unicast

netlink_broadcast

netlink_unicast

Search for a sock that has the same pid with the current process in the nl_table.

enqueue the sk_buff into the socket's receive queue.

call rtnetlink_rcv

receive queue

rtnetlink_rcv

dequeue each sk_buff in the receive queue until the queue is empty.

`sk_buff`

extract family and type from the sk_buff and check the doit function from the rtnetlink_links table

tc_modify_qdisc

Create/change qdisc

tc_ctl_tfilter

Create/change tfilter

tc_get_qdisc

Delete/get qdisc

References

[1] David A Rusling. *The Linux Kernel version 0.8-3.*

[2] M Beck, H Bohme, M Dziadzka, U Kunitz, R Magnus, D Verworner. *Linux Kernel Internals Second Edition*. Addison-Wsley, ISBN 0-201-33143-8.

[3] Alessandro Rubini. *Linux Device Drivers*. O'Reilly, ISBN 1-56592-292-1.

[4] GNU Zebra (routing software): http://www.zebra.org

[5] GateDaemon: http://www.gated.org/

[6] Werner Almesberger, Jamal Hadi Salim, Alexey Kuznetsov. *Differentiated Service on Linux*. June 25, 1999.

[7] Werner Almesberger, *Linux Network Traffic Control – Implementation Overview*. April 23, 1999.

[8] Linux source code.

[9] Steve Waldbusser. *CMU-SNMP-Linux Readme*. Carregie-Mellon University

[10] Daniel Ridruejo, *The Linux Networking Overview HOWTO*, http://www.linuxdoc.org/HOWTO/Networking-overview-HTOWTO.html.