INTRODUCTION

Internetworking with Linux has been the most popular choice of developers. Not only in the server world where Linux has made its mark but also in the small embedded network OS market, Linux is the most popular choice. All this requires an understanding of the TCP/IP code base. Some products require implementation of firewall, and others require implementation of IPSec. There are products that require modifications in the TCP connection code for load balancing in a clustered environment. Some products require improving scalability on SMP machines. Most talked about is the embedded world, where networking is most popular. Real-time embedded products have very specific requirements and need huge modifications to the stack as far as buffer management is concerned or for performance reasons. All these require a complete understanding of stack implementation and the supporting framework.

As mentioned above, some of the embedded networking products require a minimum of the code to be compiled because of the memory requirements. This requirement involves knowledge of source code organization in the Linux source distribution. Once we know how the code is distributed, it becomes easier to find out the relevant code in which we are interested.

Mostly all the networking application work on very basic client–server technology. The server is listening on a well-known port for connection requests while the client is sending out connection request to the server. Many complex arrangements are made for security reasons or sometimes for load balancing to the client–server technology. But the basic implementation is a simple client–server program in which the client and server talk to each other. For example, telnet or...
ftp services are accessed through the inet program which hides all the details of services. There are many tunable parameters available to tune your TCP/IP connections. These can be used to best tune the connection without disturbing overall system wide tuning.

Most of the network applications are written to exchange data. Once a connection is established, either (a) the client sends data to the server or (b) data flow in the opposite direction or may flow in both directions. There are different ways to send and receive data over the connection. These different techniques may differ in the way that application blocks once the socket connection either receive or send data.

In the entire book we discuss only TCP and no other transport protocol. So, we need to understand the TCP connection process. TCP is a connection-oriented protocol that has a set process for initializing connections, and similarly it has a set process for closing connection cleanly. TCP maintains state for the connection because of handshakes during connection initiation and closure processes. We need to understand the TCP states to completely understand the TCP connection process.

In this chapter we will present an overview of how the TCP/IP protocol stack is implemented on Linux. We need to understand the Linux operating system, including the process, the threads, the system call, and the kernel synchronization mechanism. All these topics are covered though not in great detail. We also need to understand the application programming interface that uses a TCP/IP protocol stack for data transmission, which is discussed. We discuss socket options with kernel implementation. Finally, we discuss the TCP state, which covers a three-way handshake for opening connection and a four-way handshake for connection closure.

1.1 OVERVIEW OF TCP/IP STACK

Let’s see how the TCP/IP stack is implemented on Linux. First we just need to understand the network buffer that represents the packet on Linux. *sk_buff* represents the packet structure on Linux (see Fig. 1.1). *sk_buff* carries all the required information related to the packet along with a pointer to the route for the packet. *head, data, tail,* and *end* point to the start of the data block, actual start of data, end

![Figure 1.1. Network buffer, sk_buff.](image-url)
of data, and end of data block, respectively. skb_shared_info object is attached at
the end of the sk_buff header which keeps additional information about paged data
area. The actual packet is contained in the data block and is manipulated by data
& tail pointers. This buffer is used everywhere in the networking code as well as
network drivers. Details are discussed in Chapter 5.

Now we will have a look at how the stack is implemented in Linux. We will first
start with down-the-stack processing of the packet from the socket layer to the
driver layer and then move up the stack. We will take an example of sending TCP
data down the stack. In general, more or less the same stack is used for other trans-
port protocols also, but we will restrict our discussion to TCP only.

1.1.1 Moving Down the Stack

When an application wants to write data over the TCP socket, the kernel reaches
the socket through VFS (see Fig. 1.2). inode for the file of the type socket contains
a socket object, which is the starting point for the networking stack (see Section 3.2
for more details). The socket object has a pointer to a set of operations specific to
the socket type pointed to by field ops. Object proto_ops has a pointer to socket-
specific operations. In our case, the socket is of type INET, so send systemcall ends
up calling inet_sendmsg() inside kernel via VFS. The next step is to call a protocol-
specific send routine because there may be different protocols registered
under INET socket (see Section 3.1). In our case, transport later is TCP, so
inet_sendmsg() calls a protocol-specific send operation. The protocol-specific
socket is represented by a sock object pointed to by the sk field of the socket object.
A protocol-specific set of operation is maintained by a proto object pointed to
by prot field of sock object. inet_sendmsg() calls a protocol-specific send routine,
which is tcp_sendmsg().

In tcp_sendmsg(), user data are given to a TCP segmentation unit. The segmen-
tation unit breaks big chunks of user data into small blocks and copies each small
block to sk_buff. These sk_buffs are copied to the socket’s send buffer, and then
the TCP state machine is consulted to transmit data from socket send buffer. If the
TCP state machine does not allow sending new data because of any reasons, we
return. In such a case, data will be transmitted later by a TCP machine on some
event which is discussed in Section 11.3.11.

If the TCP state machine is able to transmit sk_buff, it sends a segment to the
IP layer for further processing. In the case of TCP, sk->ip->af_specific->queue_xmit
is called, which points to ip_queue_xmit(). This routine builds an IP header and
takes an IP datagram through the firewall policy. If the policy allows, an IP layer
checks if NAT/Masquerading needs to be applied to the outgoing packet. If so, a
packet is processed and is finally given to the device for final transmission by a call
to dev_queue_xmit(). Device refers to a network interface, which is represented by
net_device object. At this point, the Linux stack implements QOS. Queuing disci-
plines are implemented at the device level.

Packet (sk_buff) is queued to the device according to their priority levels and
queuing discipline. Next is to dequeue the packet from the device queue, which is
done just after queuing sk_buff. The queued packet may be transmitted here,
depending on the bandwidth for the packet’s priority. If so, the link layer header is
prepended to the packet, and the device-specific hard transmit routine is called to
transmit the frame. If we are unable to transmit the frame, the packet is requeued
Application writes data over TCP socket

Socket layer, send queue

Figure 1.2. TCP packet moving down the protocol stack.

sock->ops->sendmsg = inet_sendmsg()
socket specific processing

sk->prot->sendmsg = tcp_sendmsg()
Protocol specific processing

sk->tp->af_specific->queue_xmit = ip_queue_xmit()
Network layer processing.

dev_queue_xmit()
QoS & link layer processing

Tx Soft IRQ

Packet transmitted
on the device queue and Tx softIRQ is raised on the CPU adding device to the
CPU's transmit queue. Later on when the TX interrupt is processed, frames are
dequeued from the device queue and transmitted.

1.1.2 Moving Up the Stack

Refer to Fig. 1.3 for the flow of packet up the stack. We start with the reception of
packets at the network interface. Interrupt is generated once the packet is com-
pletely DMAed on driver's Rx ring buffer (for details see Section 18.5). In the
interrupt handler, we just remove the frame from the ring buffer and queue it on
CPU's input queue. By CPU I we mean the CPU that is interrupted. It is clear at
this point that there is per CPU input queue. Once the packet is queued on the
CPU's input queue, Rx NET softIRQ is raised for the CPU by call to netif_rx().
Once again, softIRQ's are raised and processed per CPU.

Later when Rx softIRQ is processed, packets are de-queued from CPU's receive
queue and processed one-by-one. The packet is processed completely until its des-
tination here, which means that the TCP data packet is processed until the TCP
data segment is queued on the socket's receive queue. Let's see how is this process-
ing done at various protocol layers.

netif_receive_skb() is called to process each packet in Rx softIRQ. The first step
is to determine the Internet protocol family to which a packet belongs. This is also
known as packet protocol switching. We send the packet to the raw socket in case
any raw socket is opened for the device. Once the protocol family is identified,
which in our case is IP, we call the protocol handler routine. For IP, this is the
ip_rcv() routine. ip_rcv() tries to de-NAT or de-masquerade the packet at this point,
if required. The routing decisions are made on the packet. If it needs to be delivered
locally, the packet is passed through firewall policies configured for the locally
acceptable IP packets. If everything is OK, ip_local_deliver_finish() is called to find
the next protocol layer for the packet.

ip_local_deliver_finish() implements INET protocol switching code. Once we
identify the INET protocol, its handler is called to further process the IP datagram.
The IP datagram may belong to ICMP, UDP, and TCP.

Since our discussion is limited to TCP, the protocol handler is tcp_v4_rcv().
The very first job of the TCP handler is to find out socket for the TCP packet. This
may be a new open request for the listening socket or may be another packet
for the established socket. So here, various hash tables are looked into. If the
packet belongs to the established socket, the TCP engine processes the TCP
segment. If the TCP segment contains in-sequence data, it is queued on the socket's
receive queue. If there are any data to be sent, they is sent along with the the ACK
for the data arrived here. Finally, when application issues read over the TCP socket,
the kernel processes the request by providing data from the socket's receive
queue.

The Linux stack maps to the OSI networking model (see Fig. 1.4).

1.2 SOURCE CODE ORGANIZATION FOR LINUX 2.4.20

Figure 1.5 shows the kernel source tree.
INTRODUCTION

Figure 1.3. TCP packet moving up the stack.

Packet received

Interrupt handler
removes packet from
DMA ring buffer

netif_rx(), Rx SoftIRQ

netif_receive_skb(), Protocol switch

ip_rcv(), IP layer processing.

ip_local_deliver_finish(),
INET protocol switcher

tcp_v4_rcv(), TCP entry point

protocol specific processing

Socket layer receive queue

Application reads data from receive queue

Figure 1.3. TCP packet moving up the stack.
1.2.1 Source Code Organization for Networking Code

Figure 1.6 shows the kernel networking source tree.

1.3 TCP/IP STACK AND KERNEL CONTROL PATHS

In this section we will see how TCP data are being processed by the Linux kernel. In totality, we will see different kernel control paths and processor context that are involved in packet processing through the kernel. When the process writes data over the TCP socket, it issues write/send system calls (see Fig. 1.7). The system call takes the process from the user land to the kernel, and now the kernel executes on behalf of the process as shown by the solid gray line. Let’s determine the different points in the kernel where the kernel thread sending TCP data on behalf of the process preempts itself.

*Kernel Control Path 1.* In this kernel control path, the kernel thread processes TCP data through the complete TCP/IP stack and returns only after transmitting data from the physical interface.

*Kernel Control Path 2.* This kernel control path processes data through TCP/IP stack but fails to transmit data because the device lock could not be obtained. In
All driver code goes here. Some of these drivers can be compiled as part of kernel and others as modules. Keeping minimum of drivers as part of kernel makes it much smaller in size.

Network specific code goes here. Protocol specific files are ipv4, ipv6, bluetooth, appletalk... socket.c has generic socket code, sched contains code specific to IP TOS and generic packet scheduling, netlink contains netlink socket source files.

Filesystem related code goes here. This directory contains generic VFS code, incode, devfs, pipe, file locks, etc are covered in this directory. File system specific code is contained here which can be directly compiled in the kernel or as module.

Core kernel generic code goes here, core kernel contains scheduler, process management module support, timers, signal, softIRQ, resource management etc.

Kernel memory management source is contained in this directory. Swap, paging, memory mapping, memory locking, high memory etc.

Inter process communication code goes here. These are shared mem, semaphore, message queues.

Figure 1.5. Kernel source tree.
Figure 1.6. Kernel networking source tree.
this case, the kernel thread returns after raising Tx softIRQ. SoftIRQ processing is
deferred to some later point of time which will transmit data queued up on the
device. See Section 17.1 for details on softIRQ processing.

**Kernel Control Path 3.** This kernel control path processes data through the
TCP layer but is not able to take it further because the QOS policy is not allowing
further transmission of data. It may happen that either someone else is processing
the queue on which packet is queued or the quota for queue is over. In the later
case, a timer is installed which will process the queue later.

**Kernel Control Path 4.** This kernel control path processes data through the
TCP layer but cannot proceed any further and returns from here. The reason may
be that the TCP state machine or congestion algorithm does not allow further
transmission of data. These data will be processed later by the TCP state machine
on generation of some TCP event.

**Kernel Control Path 5.** This kernel control path may execute in interrupt
context or kernel context. Kernel context may come from softIRQ daemon, which
runs as kernel thread and has no user context. Kernel context may also come from
kernel thread corresponding to user process which enables softIRQ on the CPU by
call to *spin_unlock_bh()*. See Section 17.6 for more detail. This kernel control path
processes all the data queued by control path 2.

**Kernel Control Path 6.** This kernel control path executes as a high-priority
tasklet that is part of softIRQ. This may also be executed in interrupt context or
kernel context as discussed above. This processes data queued by control path 3.

**Kernel Control Path 7.** This kernel control path executes as softIRQ when
incoming TCP packet is being processed. When a packet is received, it is processed
by Rx softIRQ. When a TCP packet is processed in softIRQ, it may generate an event causing transmission of pending data in the send queue. This kernel control path transmits data that are queued by control path 4.

On the reception side, the packet is processed in two steps (see Fig. 1.8). An interrupt handler plucks a packet from the DMA ring buffer and queues it on the CPU-specific input queue and raises Rx softIRQ. Rx softIRQ is processed at some later point of time in interrupt context or by softIRQ daemon. The TCP data packet is processed completely by Rx softIRQ until it is queued on the socket’s receive queue or is eaten up by the application. The TCP ACK packet is processed by a TCP state machine, and softIRQ returns only after action is taken on the events generated by the incoming ACK.

1.4 LINUX KERNEL UNTIL VERSION 2.4 IS NON-PREEMPTIBLE

Let’s define the term preemptive first and then we will move ahead with its effect on the Linux kernel. Preemption in general means that the current execution context can be forced to give away CPU for some other execution context under certain conditions. Now we will say that what is so great about it is that it is happening on any multitasking OS. On a multitasking OS, many userland processes run on the CPU one at a time. These processes are assigned quota and continue to occupy CPU until they have exhausted their quota. Once the quota for the currently running process is over, it is replaced by some other runnable process on the CPU even if the former was already executing by the kernel scheduler. So, we can say that the process was preempted here. Very true, the userland process is preempted to fairly give other processes a chance to run on the CPU. We are not discussing scheduling with respect to real-time processes and are discussing only normal priority processes that are scheduled based on a round-robin scheduling policy. This way kernel preempts the userland process.

What we would like to know in this section is very different from what has been discussed so far. We want to know how a kernel can be preemptive. Let’s suppose

![Figure 1.8. Packet reception and different kernel control paths.](image-url)

Figure 1.8. Packet reception and different kernel control paths.
that some kernel control path is being executed on the CPU and it is looping into infinite loop by mistake. Can a kernel preempt itself to get out of the infinite loop and give a CPU to some other runnable process. (Note: I’m taking an example of infinite loop inside the kernel just to explain the term preemption, but the intent here is very different. Normally, a kernel code does not end up in this situation). Kernel control path gives away CPU to other burnable process by calling scheduler.

We must first know what event causes a running process to preempt. This is done by the timer interrupt which is raised on the CPU at some definite time interval and is nonmaskable. This interrupt does all the necessary calculation determine the duration of the current execution context on the CPU. If it has expired its quota, it sets a ‘scheduling needed’ flag for the process. While returning from the interrupt, this flag is checked but only if we were interrupted in the user mode (which essentially means that the CPU was executing user land code when the timer interrupt occurred).

Control is passed to an assembly code at line 256 in cs 1.1 when we are returning from the interrupt. Line 257 first gets the pointer to a current process (kernel thread corresponding to the user land process) in ebx%. At line 259, we get EFLAGS for the current process from the stack pointer (%esp) and save this to eax%. At line 260, we get a code segment byte from the stack pointer and save it as a byte in eax%. At line 261, we check if the execution mode was within the kernel or user land at the time when the CPU was interrupted. This can be verified from the code segment that is copied to eax% at line 260. If the CPU was executing in the kernel, we jump to restore_all at line 263. restore_all will switch to the execution context within the kernel by loading register values saved at the stack and will start executing from where it was interrupted. If we were interrupted in the user land, control is passed to ret_from_sys_call. re_from_sys_call does lots of checks; for example, if there is a pending signal for the current process, reschedule is needed, and so on, and takes appropriate action. If the current process has not consumed its time slice, it will continue to execute in the user land; otherwise, some other runnable process will be given the CPU.
As shown in Fig. 1.9a, we switch to kernel mode to handle interrupts. We have shown timer interrupt in particular, but it may also happen that some other interrupt may also cause the current user process to give away CPU to some other process. For example, network interrupt may cause some process to wake up that is waiting for data over the connection. Since I/O intensive processes always have a higher priority over the CPU intensive processes, network interrupt carrying data may cause current process to give CPU to the process waiting for I/O over this connection. In the case where the current process has not consumed its time slice, it will continue to run on the CPU in case it has not received any kill signal.

Figure 1.9b shows that when a timer interrupt happens with CPU executing in the kernel, control is passed to the interrupted kernel path that was being executed at the time of interrupt. This allows the kernel to complete its execution before it can return to the user space. This design makes sure that the kernel will continue to run unless it kernel gives away CPU (by calling schedule()). Nothing can force kernel to give way CPU for any thing else other than interrupts/exceptions. The simple reason for this is data consistency, and this causes the Linux kernel to be non-preemptible. For example, if by mistake any buggy driver causes a kernel to execute an infinite loop, the single CPU system will be frozen forever.

In short, the Linux kernel 2.4 and below are not designed for real-time requirements as there may be huge latencies introduced because of a non-preemptive
kernel. An attempt is made to make Linux kernel 2.6 onwards preemptible, though not completely. We will see this in the next revision of the book.

1.4.1 **System Call on Linux**

In this section we will learn implementation of system call on Linux system running on Intel X86 architecture. Any Unix system implements a system call so that user-level application programs can request kernel services. Let’s take the simple example of an open system call. When an application wants to open a file for read and write, the very first step is to issue an open system call. Just like regular files, Pipe, fifo, socket, device, and so on, are also treated as special files on the Unix systems and will use an open system call for further I/O.

Why do we need kernel services to open a file? This is required because file-system-specific information is maintained in the kernel. File-system-specific data structures are maintained in the kernel and is accessed only in the processor privileged mode; the reason for this is consistency and uninterrupted execution. Every care is taken inside the kernel to maintain data consistency by very careful programming where an execution of code can be made uninterrupted by blocking maskable interrupts. Also, kernel is non-preemptive. So we are assured that even if the kernel is interrupted by some high-priority interrupt, the processor returns its control to the point in the kernel where it left. The kernel control path can itself give away
CPU, and no one can force it to preempt. One of the most important reasons for a file system to be inside the kernel is that it is not an independent subsystem. The file system code has to interact with other subsystems such as virtual memory, network, device controllers, paging, and scheduling; all these subsystems cannot afford to run in the user land because of the reason mentioned above.

So, for execution of the system, a call takes place inside the kernel (see Fig. 1.10). The processor has to switch from user mode to privileged mode to access kernel code and data structure. This is done by software interrupt 0x80, which is generated by the open library routine. The system call number is loaded in eax, and arguments are loaded on ebx, ecx, edx, registers. The processor determines kernel stack for the process from by loading ss and eps registers. The user context is saved on the stack by the processor control unit. Once this is done, control is passed to the system call handler.

The system call handler looks into the system call table sys_call_table, which indexes system call handling routine vectors based on system call number. Control
INTRODUCTION

is passed to the system-call-specific routine; and after execution of system call, the return value is stored in eax.

1.4.2 Adding New System Call

Let’s see how we can add a new system call to the system. To add a new system call, a new number is associated with the system call, and the system-call-specific handler should register with the system. System call numbers are listed in include/asm-i386/unistd.h file as macro __NR_sys, where sys is the name of the system call (see Fig. 1.11). In this file we need to add one more line for the new system call.

The next step is to write system call routine in appropriate file in the available in kernel source tree. For example if the system call is specific to scheduling, it should be added to kernel/sys.c. Conventionally, the name of the routine should start with sys_. Once a system call number and system-call-specific routine are added to a kernel source, we need to add the system call routine to the system call table by using macro SYMBOL_NAME(). A new line should be added to file arch/i386/kernel/entry.S (see Fig. 1.12). The line for the new system call should be added exactly to the sys_call_table at the line number matching the system call number. So, it is always better that a system call number for the new system call should be the next available number, and the entry for this system call should come at the end of the sys_call_table table. The kernel is compiled and a new kernel is placed in the correct location.

How do we access the new system call from application program. So, we can use syscall() or syscall*() system calls to invoke our system call. To syscall(), we
need to pass the system call number corresponding to the new system call registered. If we use syscall() interface, we can’t pass any arguments to our system call. If our system call takes one argument, we can use syscall1(), for two arguments we can use syscall2(), and so on; we can pass four arguments using these interfaces.

Let’s see how syscall1 is implemented (see Fig. 1.13). This is implemented as a macro in /usr/include/asm/unistd.h. It can take one argument arg1. The macro breaks into an inline assembly code that generates software interrupt int 0x80 at line 293. Line 294 indicates that the result needs to be stored in eax%. There are two inputs: eax% contains a system call number that is combined as (__NR_##name) at line 294, and ebx% contains the value of the first argument for the systemcall.

1.5 LINUX PROCESS AND THREAD

Each user land process has an associated task_struct object associated with it in the kernel. The process has two modes, user and kernel. The user land context is different from the kernel context, where each one has different code, data, and stack segment registers. Each process has user mode and kernel mode stack. The kernel mode stack is an 8K memory block, which has task_struct object at the end of the stack (see Fig. 1.14). The application runs in user mode and uses a user mode stack until it makes a system call when it switches from user mode to kernel mode where it starts using kernel mode. See Section 1.4.1 for more details.

Each process has a unique process ID by which it is identified in the system. task_struct object contains the entire information about the process, including hardware context. Some of this process-specific information is file system information, file table, signal handling, memory management, and so on. Each process has a kernel level thread associated with it which is seen by the scheduler as scheduling entity. This thread is represented by task_struct object. The kernel maintains a doubly linked link list of task_object corresponding to all runnable processes in the system.

1.5.1 fork()

New processes can be created by calling fork(). It inherits all the property of the parent process and shares VM, open files, and so on. Initially, user stacks for child and parent are shared; but as the stack grows for the child, it gets its own copy of
the stack via a COW (copy-on-write) mechanism. Child created by fork has separate `task_struct` object and different kernel mode stack. Fork internally uses a clone to create a new process. The exec*() family of system calls is used to replace an existing process with a new process.

1.5.2 Thread

A thread on Linux can be user level or kernel level. User level threads are ones that are scheduled in the user land by libraries. The kernel has no idea about these threads, and there is only one kernel thread for all the threads which corresponds to the process which has created these threads. Kernel level threads are much like Linux processes. These are also called lightweight processes (LWPs). Each thread created by the process has a corresponding kernel level thread and is treated as a scheduling identity by the kernel (see Fig. 1.15). Each thread is scheduled irrespective of every other thread for the process. So, there is much better control as far as a blocking system call is concerned. The only thing that differentiates it from a normal process is its lightweight.

Threads share virtual memory, signals, and open files with its parent. But each of them has separate process IDs. A clone system call can be used to create LWPs for the process. Clone flags to create LWPs are

- CLONE_VM
- CLONE_FS
- CLONE_FILES
- CLONE_SIGHAND
- CLONE_THREAD

The pthread library creates kernel threads for the process. LWPs created by using a clone systemcall with the above flags have separate process IDs. The option
The `ps` command can show all the threads corresponding to the process. In one example, I creates a program to spawn kernel level threads using `pthread_create()`. The `ps` command is used to display all the threads for the process as shown in Fig. 1.16.

### 1.5.3 Kernel Threads

In this section we will discuss the threads that are created inside the kernel and not by user land processes. Kernel threads are the same as the one created by the user land applications in the way they both use a clone kernel interface and both have a separate kernel mode stack. Kernel threads are created by making a call to `kernel_thread()`. Kernel threads have no user context because they are not associated with any user process. A kernel thread executes in a user kernel address space and does not have an address space of its own, unlike a user process. A kernel thread is not interrupted by any one once it starts executing. It can yield CPU by itself by going to sleep. These threads are very much visible using a `ps` command and can be recognized by the name because they start with a `k`—for example, `ksoftirqd`, `kflushd`, and so on. These threads either wake up on expiry of the timer by
themselves or are woken up by some other thread inside the kernel and are scheduled by the kernel as usual.

Let’s take an example of ksoftirqd kernel thread to illustrate kernel threads. Soft IRQ are also processed by kernel daemons in case there is a lot to be processed by softIRQs; this is mostly true in the case of network packet processing. Softirq daemons are created per CPU in routine spawn_ksoftirqd() (see cs 1.2).

kernel_thread() is called in a loop 402–410 to create one kernel thread per CPU. The routine that needs to be executed as a kernel thread is passed as a first argument to kernel_thread(); that is, ksoftirqd and second argument is CPU ID. Let’s see why we pass CPU ID when we are creating a kernel thread. The name of the kernel thread is stored in current→comm. Since softirq daemons are created per CPU, the name of each daemon contains a CPU number (see cs 1.3, line 375). This name of

kernel/softirq.c

398 static __init int spawn_ksoftirqd(void)
399 {
400    int cpu;
401
402    for (cpu = 0; cpu < smp_num_cpus; cpu++) {
403        if (kernel_thread(ksoftirqd, (void *) (long) cpu,
404            CLONE_FS | CLONE_FILES | CLONE_SIGNAL) < 0)
405            printk("spawn_ksoftirqd() failed for cpu \%d\n", cpu);
406        else {
407            while (!ksoftirqd_task(cpu_logical_map(cpu)))
408                yield();
409        }
410    }
411
412    return 0;
413 }

---

Figure 1.16. ps output showing process and associated threads (LWPs) created using a clone interface.
kernel softirq daemon appears with the name *ksoftirqd_CPU0* on running *ps* command as shown in Fig. 1.17.

Figure 1.17. *ps* output shows kernel thread as *ksoftirqd_CPU0*.

c) kernel/softirq.c

```
361 static int ksoftirqd(void * __bind_cpu)
362 {
363    int bind_cpu = (int) (long) __bind_cpu;
364        ....
365    sprintf(current->comm, "ksoftirqd_CPU%d", bind_cpu);
366    ....
367 }
```

cs 1.3. *ksoftirqd()*.

include/linux/irq_cpustat.h

```
33 #define ksoftirqd_task(cpu) __IRQ_STAT((cpu), _ksoftirqd_task)
```

cs 1.4. *ksoftirqd_task()*.

kernel/softirq.c

```
53 static inline void wakeup_softirqd(unsigned cpu)
54 {
55    struct task_struct * tsk = ksoftirqd_task(cpu);
56    if (tsk && tsk->state != TASK_RUNNING)
57    wake_up_process(tsk);
58 }
```

cs 1.5. *wakeup_softirqd()*.

kernel softirq daemon appears with the name *ksoftirqd_CPU0* on running *ps* command as shown in Fig. 1.17.

softIRQ daemon is awakened by using interface *wakeup_softirqd()* . This routine gets access to softIRQ thread for the CPU by calling *ksoftirqd_task()* at line 55. *ksoftirqd_task()* is a macro that accesses thread information from CPU-specific structure by using another macro __IRQ_STAT (see cs 1.4).

Once *ksoftirqd_task()* gets softIRQ thread for the CPU, it checks if it is not already in running state (cs 1.5, line 57). If not already scheduled, it is woken up by a call to *wake_up_process()* at line 58. This routine changes the state to TASK_RUNNING and puts the thread on the kernel run queue.
1.6 KERNEL SYNCHRONIZATION MECHANISM

The Linux kernel implements many synchronization mechanisms that are applicable in different situations on different kernel control paths. Some of these synchronization mechanisms are:

- Semaphore
- Atomic operations
- Disabling interrupts locally or globally
- Spin locks

The above synchronization mechanisms work on different principles, but the aim is to synchronize access to kernel global data structures across different kernel control paths and also across CPUs. Different kernel control paths are discussed in Section 1.3, but let us summarize here:

- Kernel path executing system call on behalf of process
- Kernel path executing interrupt routine
- Kernel path executing softIRQ.

Let’s see what synchronization mechanism could be best used for different kernel control paths. Spin lock is the most commonly used synchronization mechanism in different flavors. We will discuss this in more detail in shortly. Let’s see how semaphore is implemented, and let’s discuss its usage.

1.6.1 Semaphore

A semaphore is used to synchronize access to global data structure in an asynchronous way. When many kernel control paths want to acquire a kernel resource, only one gets the lock and the rest are put to sleep until the lock is released by the one that is acquired. `down()` and `up()` are the two routines that manipulate semaphores. When the kernel control path wants to acquire a semaphore, it calls `down()`. If we are the first one to acquire semaphore, we change the state of the semaphore and get access to the shared resource. If somebody has already acquired the semaphore, the caller has to wait on a semaphore wait queue until it is woken up by the control path that has acquired it. `up()` routine is called by the kernel control path to release the semaphore, and it also wakes up all the processes waiting on a semaphore wait queue.

The best example that explains the usage of a semaphore is page fault. Process address space may be shared by many threads (LWPs) or a child process. It may happen that page fault occurs while executing for the code area or stack area. In this case, a page fault handling routine takes a semaphore for its kernel address space (`current→mm→mmap_sem`). Then it starts to find the cause of fault and tries to get the missing page and map it to the process page table. In the meantime, some other thread which is sharing the address space of the process which is already in the process of finding page for the faulting address also faults. In this case, the thread that has faulted later will go to sleep on `mm→mmap_sem` and will be woken up once the page fault handler returns for the process that faulted first.
1.6.2 Atomic Operations

This is mainly used to synchronously access a memory region when two or more kernel control paths are trying to access them simultaneously. There are instructions that may require us to test and modify a bit atomically (without being interrupted by interrupts) on the CPU. On SMP machines, such instructions appear to be non-atomic as both the CPU’s read the same value in a given memory location in two simultaneous read cycles. If the 0 value in the memory location means acquire the lock, both will acquire the lock and will wait for the big blast. On an SMP machine, these instructions should be preceded by lock instruction to lock the memory bus by any CPU until atomic instruction is executed completely.

1.6.3 Spin Lock

The third and most commonly used synchronization technique used everywhere inside the kernel is spin locks. It is used to synchronize data access when kernel control paths on two or more CPUs try to access the same memory region simultaneously. It differs from a semaphore in the way that the semaphore freezes the process that wants to acquire the semaphore when it is already acquired. Spin lock, on the other hand, does not put the process to sleep that wants to acquire the spin lock when it is already acquired. Instead, it executes a tight loop spinning around the lock each time atomically testing the lock, also called busy-wait loop. If it finds that the lock is released, it tries to acquire it atomically. Spin lock makes use of atomic instructions. Whichever CPU succeeds in acquiring the lock first gets it, and others continue to move in a tight loop and this continues.

Spin locks have an edge over semaphores because we save a lot of time in context switching when the process trying to acquire a lock is put to sleep by the semaphore. Critical section in the kernel is referred to code that modifies/accesses global data-structures accessed from a different kernel control path. Critical sections should be protected by locks. Locks held for a longer time cause other kernel control paths to paths to wait for a longer time causing a performance hit. A critical section of the kernel code is executed for a much shorter period of time. If the time required in context switching is much more than the time spent in executing a critical region, semaphores penalize the performance extensively. In such cases, waiting on a busy loop to acquire the lock gives a much better performance. Not only this, there are other reasons to use spin lock on SMP machine instead of semaphores for serialized access of global data. For example, data that are shared between a kernel control path and an interrupt cannot be protected by a semaphore because it could freeze the system by calling a schedule in interrupt routine (hypothetical case). In the same way, a spin lock cannot be used for serialized access of data shared between interrupt and kernel control path on a single CPU machine. This would cause the machine to freeze because the tight loop in the interrupt routine would never let us come out of it when a spin lock is already acquired by the other kernel control path. For this reason, we acquire a spin lock with local interrupts disabled when data are shared between kernel control path and the interrupt routine. This doesn’t stop interrupts from occurring on other CPUs, which is OK because they will wait in a tight loop until we release the lock. Maskable interrupts are disabled locally by using the macro \texttt{local_irq_disable()} and are enabled by using \texttt{local_irq_enable()}. 
A spin lock can also be used to serialize data shared between the kernel control path, softIRQ also. In such cases, two macros can be used to disable and enable soft IRQ; these are `local_bh_disable` and `local_bh_enable`, respectively. Check Section 17.2 for details.

Different flavors of spin locks are shown in Figs. 1.18 and 1.19. In some cases we need to store EFLAGS for the CPU before disabling interrupts locally to restore it once we enable interrupts once again as interrupts are handled in nested fashion. Nested interrupt handling means that an interrupt is raised when another low-priority interrupt is already being handled on the CPU. We do this because we are not sure whether interrupts were enabled at the time we disabled them. This means that IRQs may already have been disabled by an upper layer before we are going to disable them.

In such cases, `spin_lock_irqsave()` and `spin_unlock_irqrestore()` are used to serialize data access between kernel control path and interrupt. `spin_lock_irq()` and `spin_unlock_irq()` are used simply when we want to serialize access of data shared between kernel and interrupt. `spin_lock_bh()` and `spin_unlock_bh` are used to serialize access of data shared between kernel and softIRQ.

Similarly, we have the same flavors of spin locks for reader and writer locks, which we won’t discuss here in much detail. Read spin lock allows multiple readers to get access to the shared data, whereas writer lock exclusively allows only a single writer to access the resource. When writer lock is acquired, no one including the reader is allowed access to the resource.

1.7 APPLICATION INTERFACES FOR TCP/IP PROGRAMMING

In this section we will see various interfaces that are provided to the user application to write a client–server program. All networking applications are based on client–server technology other than multicasting and broadcasting applications. There may be variants to the outlook of these applications, but basically the underlying functionality remains the same. Normally, a server is a program that provides...
a known service to the client program. The example is telnet, FTP, http, and so on. Client and server are in some kind of understanding with each other for all such services. But there is one thing in common in all the programs: client–server technology. In all the cases, a server has established its identity, which is known to the client. The client sends out a request to the server for the service, which in turn offers its services once they are connected to each other. We first discuss simple server application and then client application and see how they use TCP protocol over IP to communicate with each other.

### 1.7.1 Server Application

A server program has to provide its identity to the client programs by way of listening on a specific port. Port is a unique number that identifies a connection or specific services on a given host. When we say identifying specific connection on specific port it means that the server application needs to register its service with the kernel by way of port number. When we request a kernel to register our service, a unique port number is provided by server application to the kernel to associate its services with this number.

This port number should be known to the client application so that it can send its request to the host machine running this service. Let’s see what all interfaces are providing to hook its services with specific port number and register its service with the kernel.

We want to start service using TCP transport protocol (see Fig. 1.20). The first step is to make a `socket()` system call at line 25. The socket is a framework to communicate with the network protocol within the kernel. This call opens a socket in the kernel. The arguments to the socket call are AF_INET and SOCK_STREAM. This means that we want to open an internet family socket of type STREAM referring to TCP. The socket initializes INET socket-specific data structures and also TCP protocol-specific data structures and a set of operations. It links the socket with the VFS, which is then associated with the file descriptor and returned to the application. Now using this file descriptor, the server can request to kernel any operation on the socket.

The next step is to bind the socket with a specific port number by making the `bind()` system call at line 33. This is the way we are requesting a kernel to allocate a specific port number to its service. Here comes the concept of socket address whose C equivalent is `sockaddr_in`. This has two fields: port number and IP address. If the host machine has more than one interface, an application can request a kernel to bind the socket with a given interface or with all the available interfaces. This means that application may want to accept connection requests from only one interface or from all the available interfaces. In the former case, the `sin_addr` field of the socket address is initialized to the specific IP address and the same field needs to be initialized to INADDR_ANY in the latter case, line 31. Since this is INET address family, the `sin_family` field of the socket address is initialized to AF_INET. The port number to which we want to glue the services is initialized at line 32. The socket address is now ready for registration as object `sockaddr_in`.

The socket address is passed to `bind()` call. If the return value is less than zero, the socket could not be bound to the given port number because there may be any reason, including the fact that a port number may already be allocated to some other services. Otherwise, we got the port number that was requested.
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

---

// Figure 1.20. Server program.

```
Next is to request the kernel to start accepting the connection, which is done by making a call to `listen()` at line 37. A listen call will actually start the services for the server application. Now the kernel will start accepting connection requests for the socket. A second argument to `listen()` call is to accept a queue length for the listening socket. All the established connections for the socket sit in this queue to be accepted. Connection requests can come faster than they can be accepted by the application. For this reason we need a queuing mechanism to buffer a pending connection on the busy server.

The final step is a call to `accept()` system call at line 40. `accept()` call is made in an infinite loop. This call blocks until a new connection is available from the accept queue. As soon as a new connection is available, application is awakened and new connection is returned to the application associated with the file descriptor associated with the new socket connection.

The returned value of the accept call is associated with a new connection and can be used for communication between two ends. This opens a new channel between the two ends and is differentiated from all other connections for the same service using a remote port and an IP address. For each connection, a remote port number or a remote IP address will be unique.

Our serve program forks a new process for the newly accepted connection by a call to `fork()` at line 43. `fork()` syscall returns with value zero in the child process. In the parent process, it returns child's PID. This way we start services in the child thread in while loop 47–61. We are blocked to read data over the socket by a call to `read()` at line 53. Once it has read data over the socket, it writes received data back to the sender at line 56 by a call to `write()`. A child thread closes a listening socket at line 48 because additional reference was held on the listening socket when we were waiting on accept in parent. Parent thread closes a new socket at line 62. In the next section we will see what the client program does.

### 1.7.2 Client Application

A client program has to be sure of the server it needs to contact. To contact the server, it has to know two things about the server:

- Port number of the server at which it is listening
- IP address of the host machine where this server is running

Refer to Fig. 1.21 for a client program. The socket address consisting of these two information C equivalent of socket address is `struct sockaddr_in`, as discussed in Section 4.2. First we make `socket()` call at line 27 to open TCP socket. `sin_addr` field is initialized to the IP address of the server and `sin_port` field is initialized to port number of the listening server at lines 39 and 42, respectively. Next we make a call to `connect()` at line 43, to which we pass the socket address of the server. We pass the socket descriptor to the `connect()` on which the connection is to be established. The kernel finds route for the destination (server) and then initializes the connection process. Once the connection is established, the connect returns.

Once `connect()` returns, we are ready to communicate with the server using `read` & `write` calls using a socket descriptor. In the while loop 47–56, we are reading one line from the standard input (keyboard) at line 49 and writing it over the socket by a call to write at line 51. Just after writing data over the socket, we are waiting to
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#define READ_BUFFER 50000

int main(int argc, char *argv[]) {
  int sockfd, portno, n;
  struct sockaddr_in serv_addr;
  in_addr_t addr;
  struct hostent *server;
  char buffer[READ_BUFFER];
  if (argc < 3) {
    fprintf(stderr, "usage %s hostname port\n", argv[0]);
    exit(0);
  }
  portno = atoi(argv[2]);
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) {
    perror("ERROR opening socket");
    exit(2);
  }
  server = gethostbyname(argv[1]);
  if (server == NULL) {
    fprintf(stderr, "ERROR, no such host\n");
    exit(0);
  }
  bzero((char *)&serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy(&addr, (char *)&serv_addr.sin_addr.s_addr, sizeof(addr));
  serv_addr.sin_port = htons(portno);
  if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    perror("ERROR connecting");
    exit(3);
  }
  while(1) {
    printf("Please enter the message: ");
    fgets(buffer,READ_BUFFER,stdin);
    bzero(buffer,READ_BUFFER);
    n = write(sockfd,buffer,READ_BUFFER);
    if (n < 0) {
      perror("ERROR writing to socket");
      n = read(sockfd,buffer,READ_BUFFER);
      if (n < 0) {
        perror("ERROR reading from socket");
      } else {
        buffer[n] = '\0';
        printf("%s", buffer);
      }
    }
  }
  return 0;
}
```

**Figure 1.21.** Client program.
read data over the socket by a call to read at line 54. Data received are printed at line 59. The server returns whatever it has read over the socket, which is read by the client and displayed at standard output. This makes an echo server.

1.7.3 Socket Options

Sockets can be tuned as per the requirements by an application. This facility can save us from tuning the entire system where different applications have different requirements. For example, telnet connection requires setting a KEEP_ALIVE timer for the TCP connection between telnet server and client. This facility is required because telnet connection can be open for months without any activity. With KEEP_ALIVE socket option, the server can probe client to find out if it is alive. On the other hand, FTP doesn’t need this option.

`setsockopt()`. There are many socket options that can be used to tune different TCP connections. `setsockopt()` is an interface that is provided to the application to set socket options for a given connection without disturbing global settings (see Fig. 1.22). Arguments to the system call are as follows:

- `s`: This is the socket descriptor as returned by the socket.
- `optname`: This is the name of the socket option that needs to be tuned.
- `optval`: This is the value of the socket option to be set.
- `optlen`: This is the length of the optional value that is passed to the kernel to mark the end of option length. The reason is that `optlen` is a pointer to void.

`getsockopt()`. `getsockopt()` is an interface provided to get the value of socket option (see Fig. 1.23). The arguments are the same as they are for `setsockopt()`, with the difference being that they are used to fetch the value of the socket options.

1.7.4 Option Values

**SO_DEBUG.** This turns on debugging at various protocol layers. This may be useful when we want to track allocation of buffers, traversal of packets on the stack, behavior of TCP algorithms, and so on. If the socket debug option is enabled, the `SOCK_DEBUG` macro prints messages on reception of bogus ACK for the byte that is not yet sent (line 1908, cs 1.6).

```c
int setsockopt(int s, int level, int optname, const void *optval, int optlen);
```

**Figure 1.22.** `setsockopt()`.

```c
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
```

**Figure 1.23.** `getsockopt()`.
The SOCK_DEBUG macro uses the kernel printk() interface to write debug messages. These messages can be seen through dmsg command or from file /var/log/messages. We can see that SOCK_DEBUG first checks if debug option is on for the socket (sk->debug) at line 468 (cs 1.7). sk->debug is set by the application using setsockopt() interface.

**SO_BROADCAST**. This enables sending of broadcast messages, if this is supported by the protocol. Broadcast is not supported by TCP. Only UDP and raw socket support broadcast. In udp_sendmsg(), if the route is of type broadcast (RTCF_BROADCAST), it can send broadcast messages only if socket option enables (sk->broadcast) is set (line 525, cs 1.8).
SO_REUSEADDR. Whenever any server application wants to bind to a port which is already in use by some other application on the same machine, this option may allow us to use the same port number under certain conditions. This option sets the `reuse` field of the `sock` object.

`tcp_v4_get_port()` is called inside the kernel through a bind path when application wants to bind to a specific port. We traverse through the bind hash list; and if we find port already occupied and `sk->reuse` is set more than 1 (line 250, cs 1.9), we can directly use the port. Otherwise, if the value of `sk->reuse` is set to 1 (line 252, cs 1.9), it has to go through some additional checks before getting the port.

SO_KEEPALIVE. This option enables a heartbeat mechanism for TCP connection. An application like telnet may be active for months, where one end never knows about the other end when connections are ideal. It may happen that the one end has gone down, in which case the other end will never know. Half-connection will unnecessarily be open, thereby occupying resources. This option keeps sending messages to the other end once connection is idle for some time. In return, the sending end expects acknowledgment. If acknowledgments are not received, the connection is closed after a certain number of retries.

When the option is enabled, `tcp_set_keepalive()` is called to set the keepalive timer for TCP, and `sk->keepopen` is set to 1. `tcp_set_keepalive()` resets the keepalive timer in case it is not already set; this is done by calling `tcp_reset_keepalive_timer()` (see cs 1.10, line 568).

SO_LINGER. The linger option is to enable a TCP socket to provide enough time to send unsent data in the send queue when a socket is closed by an application. We provide a timeout value with this option so that the kernel hangs on for this much time before closing the socket. In this time, the TCP gets enough time to flush all the data to the receiver. If timeout is not provided, the kernel waits until all the data are flushed out.

This option sets `sk->linger` to 1, and `sk->lingertime` is set to a timeout value provided by user application. When an application issues a `close()` syscall an INET socket, `inet_release()` is called. If a linger option is set, a linger timeout value is taken...
INTRODUCTION

from sk→lingertime (cs 1.11, line 463). Finally, a protocol-specific close routine is called with a linger timeout value at line 465 (see cs 1.11).

In tcp_close(), we check the timeout value passed as an argument to the routine. If set, the kernel puts the process to sleep before by calling add_wait_queue() at line 1978 (see cs 1.12). By the time we request a timeout, all data would have been flushed. Once we have performed the timeout, the socket is closed.

**SO_OOBINLINE.** This option is related to a TCP urgent byte. If the option is set, the TCP urgent byte is received inline; otherwise, it is received on different channel as out-of-band data. The option sets sk→urginline to 1. sk→urginline is discussed in much detail in Section 8.3.2.

**SO_SNDBUF.** This option sets send buffer size for the socket, sk→sndbuf. This value puts a limit on the total amount of memory allocated for the send buffer. In
case the segments get acknowledged, they stay in the send buffer and account for
the send buffer consumption.

tcp_memory_free() is called when application data are written over the TCP
socket to check if we have enough space in the send buffer for application data. If
this returns TRUE, we can queue new data to socket’s send buffer, otherwise not
(see cs 1.13).

SO_RCVBUF. The option is the same as SO_SNDBUF with the difference that
this option sets an upper limit on the receive buffer, sk→rcvbuf. In tcp_data_queue(),
we check if allocated memory for receive socket buffer is more than socket send
buffer limit at line 2571 (cs 1.14). If the condition is true, we try to squeeze some
memory from the receive queue by calling tcp_prune_queue() at line 2573.

SO_DONTROUTE. This option is mainly used by RAW sockets or UDP sockets
and sets sk→localroute to 1. If this option is enabled, the normal routing policy is
disabled for the outgoing packet. The packet will be routed only if the destination
is directly connected to the network.

SO_RCVTIMEO. This sets the timeout value for the socket that specifies the
maximum amount of time the process should be blocked for an incoming event such
as the following:

• Accept blocked for new connection on listening socket.
• Read is blocked to receive data on the connected socket.
**INTRODUCTION**

net/ipv4/tcp.c

```c
2522 static void tcp_data_queue(struct *sk, struct *sk_buff *skb)
2523 {
    ..... 
2570         if (eaten < 0 &&
2571             (atomic_read(&sk->rmem_alloc) > sk->recvbuf ||
2572             !tcp_rmem_schedule(sk, skb))) {
2573                 if (tcp_prune_queue(sk) < 0 || !tcp_rmem_schedule(sk, skb))
2574                     goto drop;
2575         }
2576 }
```

**cs 1.14. tcp_data_queue().**

include/net/sock.h

```c
1238 static inline long sock_rcvtimeo(struct *sk, int noblock)
1239 {
1240     return noblock ? 0 : sk->rcvtimeo;
1241 }
```

**cs 1.15. sock_rcvtimeo().**

include/net/sock.h

```c
1467 int tcp_recvmsg(struct *sk, struct msghdr *msg,
1468                    int len, int noblock, int flags, int *addr_len)
1469 {
    ..... 
1488     timeo = sock_rcvtimeo(sk, noblock);
    ..... 
1638     } else {
1639         timeo = tcp_data_wait(sk, timeo);
1640     }
    ..... 
1770 }
```

**cs 1.16. tcp_recvmsg().**

`sock_rcvtimeo()` returns a value of timeout for blocking sockets, (see cs 1.15). `tcp_recvmsg()` calls `sock_rcvtimeo()` at line 1488 (cs 1.16) to get a timeout value for the socket. Once requested data are not available, `tcp_data_wait()` is called at line 1639 (cs 1.16) with a timeout value returned by `sock_rcvtimeo()`. This puts the process to sleep until timeout occurs or until data are received, whichever happens first.

**SO_SN DTIMEO.** This option is similar to **SO_RCVTIMEO** except that this sets a timeout for receiving events on the socket. This sets a value of `sk->sndtimeo`. 
sock_sendtimeo() returns a timeout value as sk→sndtimeo for blocking sockets (see cs 1.17).

tcp_sendmsg() calculates records timeout value at line 1025 (cs 1.18) by call to sock_sndtimeo(). If it fails to allocate memory for copying new data into a network buffer (line 1068, cs 1.18), it has to wait for memory by calling wait_for_tcp_memory() until it times out or memory is available, whichever happens first.

### 1.8 SHUTDOWN

The client–server program may be sending and receiving data from both the ends because TCP is a fully duplex stream protocol. It may happen that one end doesn’t want to send or receive any more data because it is already done. In such a case, it will close that end of the socket. If any activity happens on that end further, the socket will throw an error saying that operation is not permitted. The shutdown() function shall cause all or part of a full-duplex connection on the socket to be shut down.

The shutdown() function takes the following arguments (Fig. 1.24):
socket. This is a file descriptor associated with the socket.

how. This specifies what action needs to be taken. The values are as follows:

SHUT_RD. This disables reading of any more data over the socket. TCP may be accepting data, but the application is not allowed to read data over the socket.

SHUT_WR. This disables writing of data over the socket. When application wants to send data over the socket after write side is shut down, the socket throws an error to the application, indicating that a pipe is broken.

SHUT_RDWR. This disables further send and receive operations.

1.8.1 Kernel Shutdown Implementation
Let’s see how shutdown is implemented in the kernel. sk→shutdown flags shutdown events. There are two flags here:

- SEND_SHUTDOWN, set to disable send events.
- RCV_SHUTDOWN, set to disable receive events.

1.8.2 Send Shutdown
When an application wants to send a message after the send side of the socket is shut down, tcp_sendmsg() handles the situation. sk→shutdown has SEND_SHUTDOWN bit set for the socket in this case. An error is initialized to E_PIPE at line 1042, cs 1.19. At line 1043 we check the shutdown flag. If the SEND_SHUTDOWN bit is set, we go to error handling at line 1202. It is rare that any data are copied to the application buffer. I mean that it is rare that shutdown is called from application when the kernel is in the process of reading data from the socket buffer. So, we move to error handling at line 1205. Here we do some cleanup operation and then return error number which is set to E_PIPE.

1.8.3 Receive Shutdown
When an application wants to receive data over a TCP socket, a kernel calls tcp_recvmsg(). Error number is initialized to ENOTCONN. We read data in do-while loop 1502–1703, cs 1.20. In the process, we check if a shutdown bit is set for the socket at line 1568. If so, we break. We do a cleanup operation and then return the value of copied, which may be a positive value if there was any data copied from a receive buffer or 0 if there was nothing copied from the receive buffer. It doesn’t return an E_PIPE error instead 0. Zero return value to the application means that nothing was there to be read from the socket.
net/ipv4/tcp.c

1009 int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)
1010 {
    ...
1042     err = -EPIPE;
1043     if (sk->err || (sk->shutdown&SEND_SHUTDOWN))
1044     goto do_error;
    ...
1202 do_error:
1203     if (copied)
1204     goto out;
1205 out_err:
1206     err = tcp_error(sk, flags, err);
1207     TCP_CHECK_TIMER(sk);
1208     release_sock(sk);
1209     return err;
1210 }

CS 1.19 tcp_sendmsg().

net/ipv4/tcp.c

1467 int tcp_recvmsg(struct sock *sk, struct msghdr *msg,
1468     int len, int nonblock, int flags, int *addr_len)
1469 {
1470     struct tcp_opt *tp = &(sk->tcp_info.af_tcp);
    ...
1484     err = -ENOTCONN;
    ...
1502     do {
    ...
1568         if (sk->shutdown & RCV_SHUTDOWN)
1569             break;
    ...
1730 } while (len > 0);
    ...
1758     TCP_CHECK_TIMER(sk);
1759     release_sock(sk);
1760     return copied;
    ...
1770 }

CS 1.20 tcp_recvmsg().
1.9 I/O

In this section we discuss different system calls on Unix systems that deal with I/O. Our discussion will be more focused on the feature that system call adds to I/O activities. These system calls can be used to receive or send normal- or high-priority data over the socket.

1.9.1 read()

This is the simplest system call to read data over the socket. We specify a socket descriptor as a first argument, address of the location where data should go as a second argument, and number of bytes to be read in the buffer as a third argument (see Fig. 1.25). The system call can a block or return immediately, depending on whether the socket is blocking or nonblocking. By default, it is blocking. If the socket is blocking, read blocks in case its request is not satisfied completely.

1.9.2 write()

This is simplest system call to send data over the socket (see Fig. 1.26). Arguments are same as that for the read; the difference is that instead of reading, this will write data. The blocking and non-blocking nature is the same as that for read.

1.9.3 recv()

This system call would receive data over the socket with some added control (Fig. 1.27). The first three arguments are the same as that for read, with an additional fourth argument as control flags. With the additional flag, we can just peek for the data or can receive TCP urgent data as out-of-band data. In the latter case, the process will never block even if the socket is blocking.

```c
ssize_t read(int fildes, void *buf, size_t count);
```

**Figure 1.25. read().**

```c
ssize_t write(int fildes, const void *buf, size_t count);
```

**Figure 1.26. write().**

```c
ssize_t recv(int s, void *buf, size_t len, int flags);
```

**Figure 1.27. recv().**
1.9.4 **send()**

This system call would send data over the socket with some added control (Fig. 1.28). This is the same as recv, with the difference being that this is used for sending data instead of receiving data. The flags argument has the same meaning as it is for recv.

1.9.5 **select()**

The select system call offers more features with added complexity (Fig. 1.29). The added feature is to do I/O multiplexing demultiplexing. With the system calls discussed so far, we can do I/O only on a single socket descriptor or file descriptor. With select, we can block on multiple events for different descriptors. The events are read, write, and exception. For each event, we have pointer to fd_set object. We can mark the bit corresponding to the file/socket descriptor in fd_set object. We do this by using macro FD_SET(). We pass pointers to fd_set for each event to select. The first argument to select is a maximum file descriptor number that will be one more than the highest number received as the file/socket descriptor for the process. We can also provide a timeout value as the fifth argument. Once select returns, the return value indicates the number of events that has occurred. We need to check each event by using macro FD_ISSET on each descriptor to check which event has occurred. For example, if there are data to be read on the socket and we want this event to be notified, select returns with bit set for read event. FD_ISSET() for readfs event will return 1 for the descriptor that received data.

1.10 **TCP STATE**

TCP is a state-oriented protocol. Each TCP session maintains a state of its own. The state of the TCP connection is a kind of marker for the protocol which decides the behavior of the protocol at any given point of time. Each state will have a pre-decided set of rules that need to be followed strictly. Specific events can change the
state of the protocol, which in turn changes the next course of action. Any diversion from the current course of action may lead to major failures caused from breaking protocol. As we see later in the discussion, there is a way in which a connection needs to be established initially between two TCP peers. If the protocol is not followed as expected, the two ends keep on exchanging the connection-specific packets forever, thereby causing a lot of damage to the system as well as to network resources.

Let’s see what these TCP states are. We divide the discussion into three different categories, depending on the stage of the TCP connection:

1. Connection initiation (active and passive)
2. Established connection
3. Connection closure (active and passive)

Connection initiation (three-way handshake) is illustrated in Fig. 1.30. We have already discussed the client-server program in Section 1.7. We take the same example and see what happens when a client is trying to send a connection request to the server.

On a time-line diagram, the connection initiation would be as shown in Fig. 1.31. Connection initiation is started by the client, which invokes connect system call. So, a client sends SYN packet to the server at time 10:07:35.210908. The server responds to the connection request by ACKing (acknowledging) the SYN. Finally, the client acknowledges the SYN/ACK by sending the final ACK. From Fig. 1.30,
it is worth noting that some information is exchanged between the peers in initial SYN and SYN/ACK packets. The information contains TCP options. Please refer to Section 2.2 for detailed information about protocol headers. Let’s see how the client and server side TCP state changes with each event.

Figure 1.32 shows the transition of TCP states at client and server when some event triggers. First look at client side states:

- Initially, the client’s TCP is in a CLOSED state when it sends out SYN packet to the server. This SYN packet is a connection request to the server from client. Here the client is supposed to be doing active open.
- After the client has sent out the SYN packet (connection request), its state changes from CLOSED to SYN_SENT.
- Now the client waits for the server to send ACK for the SYN sent. Once the client receives ACK for the connection request, its TCP state changes from SYN_SENT to ESTABLISHED.

Handling error at client end. If the client receives an RST (reset) packet in reply for the initial SYN sent, its state changes to CLOSED.

Let’s look at the server side TCP state transition:

- At the server side, we have a listening socket. So, the initial TCP state at the server side is LISTENING.
- The server receives connection request for the LISTENING socket—that is, the first SYN packet from the client. The server sends out an SYN/ACK packet in response to the client’s connection request. The server side TCP state doesn’t change because the connection request is still pending to be completed until the server receives the final ACK from the client. This
connection request remains open until the final ACK is received from the client and is queued in the SYN queue for the listening socket. No new socket is created at this point in time.

- The final ACK is received from the client. So the three-way handshake is completed here. A new socket is created for the connection request, which is in the SYN_RECV state. Before any event occurs, the socket is further processed and its state is changed to ESTABLISHED because both sides have agreed completely for this connection and negotiation is completed between client and server.

Once the connection is in an established state, both ends can exchange data until one of the ends decides to close the connection. Let’s see what happens when one of the ends does an active close. The client is 192.168.1.4 and the server is moksha. The client sends 100 bytes of data to the server and then does an active close to the connection. Figure 1.33 shows the tcpdump output of the life cycle of the TCP connection.

We have already discussed three-way handshake, so we won’t discuss packets 1, 2, and 3. Packet 4 is 100 bytes of data from a client which is ACKed (acknowledged) by a server in packet 5. Thereafter, the client closes the connection and hence sends FIN packet (packet 6) with 1 byte of data. The server acknowledges byte 101 in packet 7 and then sends out an FIN packet with 1 byte (packet 8). Finally, the client that did the active close gets a final FIN with ACK from the server. The client sends the final ACK to the server. Now we see how the state of TCP connection changes with each event during close.

Let’s see how the state transition happens at the two ends of the TCP connections. We take the same example where the client is writing data to the server; and after the write of 100 bytes is over, the client closes the connection (Fig. 1.34). From Fig. 1.35 we can see that once the client does an active close, it sends out a FIN segment to the other end and its state changes from ESTABLISHED to FIN_WAIT1. So, the FIN_WAIT1 state indicates that FIN still needs to be acknowledged. At the server side, FIN is received so it knows that that the client wants to close the connection in a normal way. On reception of FIN for the connection, the state of server side TCP changes from ESTABLISHED to CLOSE_WAIT. In response to the FIN received, the server can do two things here:

Figure 1.33. Complete life cycle of TCP connection.
1. It sends out ACK in reply to the FIN received from the client & send out FIN segment as another packet (Fig. 1.34).
2. It sends out FIN with ACK (Fig. 1.35).

In the former case, the state of the server side TCP doesn’t change after it has sent out ACK. But the client is actually waiting to receive a FIN segment from the server.
The client receives ACK from the server in response to its FIN. This event changes the client side TCP state from FIN_WAIT1 to FIN_WAIT2. So, the FIN_WAIT2 state indicates that FIN has been acknowledged but is waiting for the FIN segment from the peer. In the latter case, the FIN_WAIT2 state is skipped at the side that has done an active close. Finally, the server sends out a FIN segment to the client so that the server side TCP state changes from CLOSE_WAIT to LAST_ACK, which means that now the server is waiting for the final ACK from the client that would be acknowledgment for the server side of FIN. On reception of FIN from the server, the client sends out a final ACK to the server and the server goes to the TIME_WAIT state. The server receives the final ACK form the client and goes to the CLOSED state. Now when does the client close the connection that is in the TIME_WAIT state?

**TIME_WAIT.** The TCP side that has done an active close goes to the TIME_WAIT state finally before going to the CLOSED state. It remains in the TIME_WAIT state for some definite time which we discuss later before it goes to the CLOSED state. It is primarily because this side of the TCP connection is the last to send out the ACK segment to the peer. After sending out the final ACK, it has to wait to make sure that the final ACK is received by the peer. It might happen that the final ACK is lost and the peer retransmits the FIN once again, thinking that its FIN is lost because it has not received the final ACK. So, someone has to be there at the active close end to respond to such retransmissions. If the TIME_WAIT state does not exist and the active close end does not bother to wait any longer for the final ACK segment status, it might mess up the closing process because a response to the retransmitted final FIN from the passive close end will be an RST segment.

This is one of the reasons that we need to have the TIME_WAIT state for the TCP that did the active close.

Other reasons are more obvious which might happen rarely but nevertheless cannot be ignored. Suppose the server does an active close and does not go into the TIME_WAIT state. In the meantime, the client crashes and reboots. Immediately after reboot, the client tries to connect to the server using the same port number that it used for the previous connection. It gets the connection. The two ends start communicating with each other. The sequence number used by the client in the current connection overlaps with the previous connection by coincidence. If there is some TCP segment from the previous connection held with some router and it reaches the server (delayed segment), that this is surely to cause a mess up with the data integration. If we wait here in the TIME_WAIT state, the server refuses the connection request from the client because it finds a TCP connection for the quadruplet (local IP, local port, remote IP, and remote port) which is in the TIME_WAIT state. Make sure that no connection is established with the client using a port number for which the TCP connection exists in the TIME_WAIT state, thus avoiding any unforeseen disaster.

Consider another case where a client does an active close and does not go into the TIME_WAIT state. In this case, it might reuse the same port as used by the previous connection to connect to the server. This may again cause the same problem. This problem may be curbed if the client has entered the TIME_WAIT state. Some of the implementations may allow reuse of the port that is already in use by a TCP that has entered TIME_WAIT state by deciding on the sequence
number for the new connection. Here we need to make sure that the new connection gets the sequence that will never overlap with the sequence number from the previous connection. So, in case the new sequence number obtained is overlapping with the previous connection that has gone into the TIME_WAIT state, we add a number to the current selected sequence number that makes it greater than the maximum sequence used by the previous connection and reuse the port (RFC 1185). This makes the connection unique, and delayed segment if any from the previous connection can be taken care of. Please refer to Section 4.6.7 for implementation of the logic in Linux.

Now we should be wondering for how long the connection should go into the TIME_WAIT state? RFC 793 states some of the fixed values for the TIME_WAIT state duration. Any fixed values for this may cause overestimating or underestimating the values. For example, if we are in a local subnet and we go into the TIME_WAIT state for a fixed duration of 1 minute, this causes an unnecessary wait period because any delayed segment from the last connection will not get held up for so long. On the other hand, if we keep the TIME_WAIT duration on the lower side (few seconds), and the destinations are many routers away (say internet), we might end up waiting for the disaster to happen. So, we need to decide upon TIME_WAIT duration dynamically for each connection, depending on how many routers a packet has to pass to reach to the destination. This is decided by the number of hops. So, msl (maximum segment lifetime) is the correct parameter to decide upon the TIME_WAIT duration. msl is the maximum lifetime of the segment in the internet after which it should be discarded. So, this is updated at equal intervals and averaged out each time because for the same destination, routes may differ at different times. The msl for the packet is a function of the hops field in the IP header. For more details refer to Section 2.11.

1.10.1 Partial Close

Until now we have seen the case where data flow is in one direction and the end that is sending data initiates the close when it has sent all the required data. Now we will look at the case where the connected TCP ends are sending data whereby each end can notify its peer that the data transfer is over from their side. This means that application can do partial close from its end when it thinks that it is done with sending all the data it had and we will see how the other end is notified in such case.

We take an example where both client and server are sending data to each other. The TCP end that is done first with sending all its data will close the write end of the socket. It means that it won’t send any more data to its peer. At the same time it can still continue to receive data from its peer until the peer closes its write side. We take client and server programs that will use shutdown.

A client issues a connect to the server; and after getting connected, it enters a loop where it issues three writes of 1024 block of data over the TCP connection to the server and then does a partial close to close its write end. At the same time it continues to receive data from the server until the server is done. Finally, the client doesn’t issue any close on the socket. The client does close the write end of its side by issuing shutdown() with the SHUT_WR option.

The server accepts the connection request from the client by issuing accept() and gets a new socket for this connection. It then enters a loop for five iterations
of data transfer. At each iteration it reads data; and if the read returns 0, it knows that the client will send no more data. So, it doesn’t issue any additional reads. At the same time it continues to send data in a block of 1024 bytes. After issuing 5 writes of 1024 bytes each, the server issues a close from its side, which is an indication for the client that the server is done with sending data. After this close, both ends are done and finally the sockets at both client and sever close the connection fully.

Let’s study the whole phenomenon of data transfer and TCP signaling with the help of the tcpdump output when the client and the server are transacting data. Figure 1.37 is the tcpdump output for the entire transaction until both the ends are finally closed. The client is 192.168.1.4 and the server is moksha. The first three packets are nothing but a three-way handshake when the connection is initiated. Packets 4 and 5 are a first write of 1024 bytes issued by client and acknowledgment for this write from server. Packets 6 and 7 are a repeat of packets 4 and 5; but this time, write is issued from the server side, and this write is acknowledged by the client. This continues to happen from both the ends until the client and server have issued three writes and received acknowledgment for all the writes (until packet 12). Packet 13 can be seen as a client sending FIN to the server. This means that after the third write is over, the client has closed its write end by issuing shutdown. This shutdown generates FIN from the client’s side TCP. Packets 14 and 15, each consisting of a 1024-byte block, are writes issued by the server. After these two writes, the server decides to close the connection. So, FIN is combined with the final TCP data segment; that’s why FIN appears in packet 15. The client acknowledges the FIN segment, and the connection is closed at both ends.

Let’s map the transaction to the time-line diagram (Fig. 1.36).

![Figure 1.36. Time-line diagram for client that issues shutdown on write.](image-url)
## 1.10.2 tcpdump Output for Partial Close

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Time</th>
<th>Source IP</th>
<th>Source Port</th>
<th>Destination IP</th>
<th>Destination Port</th>
<th>Sequence Number</th>
<th>Acknowledgment</th>
<th>Window Size</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11:00:21.622198</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>960507178</td>
<td>960507178(0)</td>
<td>win 49640 &gt; mss1460, nop, wscale 0, nop, nop, sack OK &gt; (DF)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11:00:21.622255</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>S 1884652429:1884652429(0)</td>
<td>ack 960507179 win 5840 &lt; mss 1460, nop, nop, sack OK, nop, wscale 0 &gt; (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11:00:21.622448</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>ack 1 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11:00:21.623359</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>P 1:1025(1024) ack 1 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11:00:21.623414</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>ack 1025 win 8192 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11:00:21.623443</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>P 1:1025(1024) ack 1025 win 8192 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11:00:21.624478</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>ack 1025 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11:00:21.625369</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>P 1025:2049(1024) ack 1025 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11:00:21.625390</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>P 1025:2049(1024) ack 2049 win 11264 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11:00:21.626389</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>ack 2049 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11:00:21.627284</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>P 2049:3073(1024) ack win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11:00:21.628420</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>P 2049:3073(1024) ack 3073 win 14336 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>11:00:21.629451</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>F 3073:3073(0) ack 3073 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>11:00:21.630857</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>P 3073:4097(1024) ack 3074 win 14336 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>11:00:21.630925</td>
<td>moksha.5000</td>
<td>192.168.1.4.34289</td>
<td>FP 4097:5121(1024) ack 3074 win 14336 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11:00:21.632744</td>
<td>192.168.1.4.34289</td>
<td>moksha.5000</td>
<td>ack 5122 win 49640 (DF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.37.** tcpdump output to illustrate TCP shutdown process.
1.11 SUMMARY

When an application sends out TCP data, the application’s associated kernel thread may return after transmitting data completely. TCP data may be queued at different levels such as socket’s send queue, device queue (TOS), and CPU output queue. This data are transmitted asynchronously by kernel timers or Tx softIRQ.

TCP data are processed in two steps: The packet is queued to CPU’s input queue and is processed completely later on by Rx softIRQ. SoftIRQ may execute in interrupt context or may also be executed by a kernel thread.

A network-specific kernel code can be found under net directory of the kernel source tree. An IPv4-specific code can be found under ipv4 subdirectory of net. A packet-scheduling-specific code can be found under sched subdirectory of net directory.

Linux kernel 2.4 and below are non-preemptive kernels; as a result, they are not suitable for real-time applications that require low latencies and timeliness for execution.

A system call is implemented by raising soft interrupt int 0x80. This interrupt switches from user to kernel mode and switches processor privilege to super-user mode where kernel code and data structure can be accessed on behalf of application. A kernel searches sys_call_table to execute syscall. sys_call_table maps a system call number to syscall callback routines.

Each Linux process has a kernel thread and kernel mode stack. A processor switches to kernel mode stack when the process enters a kernel via syscall. The kernel thread is a scheduling entity for the kernel. The pthread library on Linux creates an LWP for the process. These LWPs share resources with the parent process including process address space. All the lightweight processes (LWP) as scheduling entities inside the kernel.

Threads created in the kernel cannot be preempted unless they yield on their own. Kernel threads can be seen with ps command and usually start with the letter k, like kflushd.

Linux implements atomic operations, semaphores, and spin locks as a synchronization mechanism. Spin locks are the most extensively used synchronization mechanism to synchronize data access between two CPUs, kernel control path and softIRQs, kernels, and interrupts and have a performance edge over semaphores.

Applications communicate over the TCP/IP protocol by way of client–server technique. These programs use a socket interface to open connection and communicate over the socket using different I/O interfaces provided to the application programs.

TCP is a connection-oriented protocol that maintains state. To start a connection, TCP completes a three-way handshake and attains an established state. TCP closes connection cleanly by way of a four-way handshake. It maintains state at each step of connection initiation and connection closure stages and defines action for each state.