

THE I/O SUBSYSTEM:

The I/O subsystem allows a process to communicate with peripheral devices such as disks, tape drives, terminals, printers, and networks, and the kernel modules that control devices are known as device drivers. There is usually a one-to-one correspondence between device drivers and device types.

Systems may contain one disk driver to control all disk drives, one terminal driver to control all terminals, and one tape driver to control all tape drives. Installations that have devices from more than one manufacturer, for example, two brands of tape drives – may treat the devices as two different device types and have two separate drivers, because such devices may require different command sequences to operate properly.

A device driver controls many physical devices of a given type. For example, one terminal driver may control all terminals connected to the system. The driver distinguishes among the many devices it controls: Output intended for one terminal must not be sent to another.

DRIVER INTERFACES:

The UNIX system contains two types of devices, block devices and raw or character devices. Block devices, such as disks and tapes, look like random access storage devices to the rest of the system; character devices include all other devices such as terminals and network media. Block devices may have a character device interface, too.

The user interface to devices goes through the file system. Every device has a name that looks like a file name and is accessed like a file. The device special file has an inode and occupies a node in the directory hierarchy of the file system. The device file is distinguished from other files by the file type stored in its inode, either "block" or "character special," corresponding to the device it represents.

If a device has both a block and character interface, it is represented by two device files: its block device special file and its character device special file. System calls for regular files, such as open, close, read, and write, have an appropriate meaning for devices.

The ***ioctl*** system call provides an interface that allows processes to control character devices, but it is not applicable to regular files. However, each device driver need not support every system call interface.

For example, the trace driver allows users to read records written by other drivers, but it does not allow users to write it.

System Configuration:

System configuration is the procedure by which administrators specify parameters that are installation dependent. Some parameters specify the sizes of kernel tables, such as the process table, inode table, and file table, and the number of buffers to be allocated for the buffer pool. Other parameters specify device configuration, telling the kernel which devices are included in the installation and their "address." *There are three stages at which device configuration can be specified:*

First: administrators can hard-code configuration data into files that are compiled and linked when building the kernel code. The configuration data is typically specified in a simple format, and a configuration program converts it into a file suitable for compilation.

Second: administrators can supply configuration information after the system is already running; the kernel updates internal configuration tables dynamically.

Third: self-identifying devices permit the kernel to recognize which devices are installed. The kernel reads hardware switches to configure itself.

The kernel to driver interface is described by the block device switch table and the character device switch table (Figure 8.1). Each device type has entries in the table that direct the kernel to the appropriate driver interfaces for the system calls.

The hardware to driver interface consists of machine-dependent control registers or I/O instructions for manipulating devices and interrupt vector: When a device interrupt occurs, the system identifies the interrupting device and calls the appropriate interrupt handler.

Administrators set up device special files with the `mknod` command, supplying file type (block or character) and major and minor numbers. The `mknod` command invokes the `mknod` system call to create the device file. For example, in the command line

```
mknod /dev/tty13 c 2 13
```

`"/dev/tty13"` is the file name of the device, `c` specifies that it is a character special file (`b` specifies a block special file), `2` is the major number, and `13` is the minor number. The major number indicates a device type that corresponds to the appropriate entry in the block or character device switch tables, and the minor number indicates a unit of the device.

System Calls and Driver Interface:

It describes the interface between the kernel and device drivers. For system calls that use file descriptors, the kernel follows pointers from the user file descriptor to the kernel file

table and inode, where it examines the file type and accesses the block or character device switch table, as appropriate.

It extracts the major and minor numbers from the inode, uses the major number as an index into the appropriate table, and calls the driver function according to the system call being made, passing the minor number as a parameter.

An important difference between system calls for devices and regular files is that the inode of a special file is not locked while the kernel executes the driver. Drivers frequently sleep, waiting for hardware connections or for the arrival of data, so the kernel cannot determine how long a process will sleep. If the inode was locked, other processes that access the inode (via the stat system call, for example) would sleep indefinitely because another process is asleep in the driver.

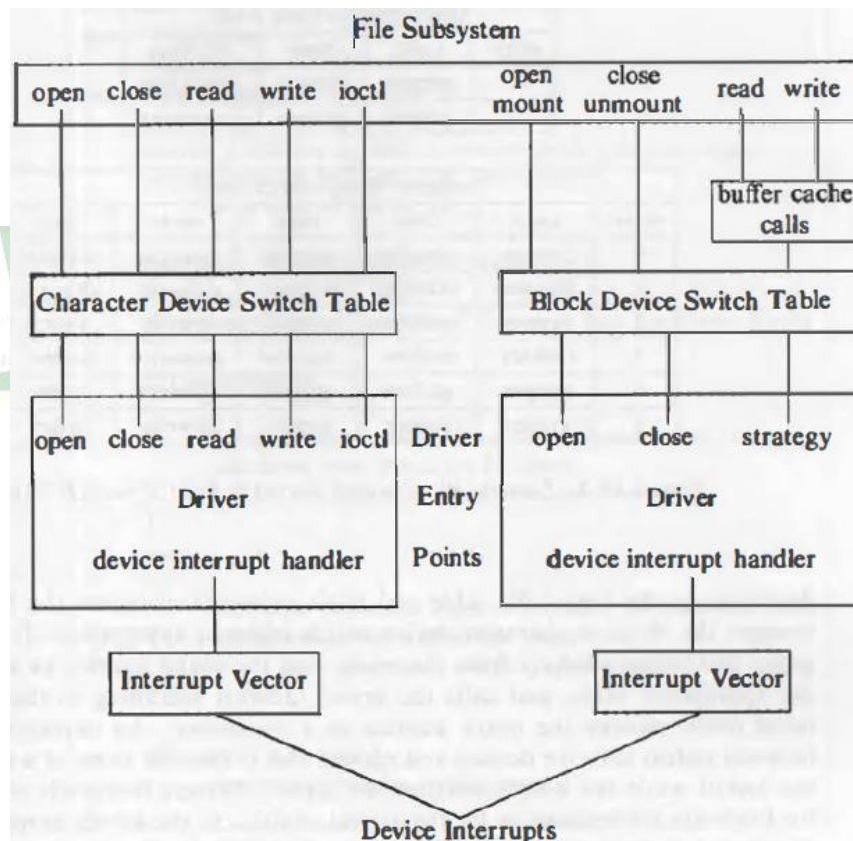


Figure 8.1: Driver Entry Points

The device driver interprets the parameters of the system call as appropriate for the device. A driver maintains data structures that describe the state of each unit that it controls; driver functions and interrupt handlers execute according to the state of the driver and the action being done.

Open:

The kernel follows the same procedure for opening a device as it does for opening regular files allocating an in-core inode, incrementing its reference count, and assigning a file table entry and user file descriptor. The kernel eventually returns the user file descriptor to the calling process, so that opening a device looks like opening a regular file. However, it invokes the device-specific open procedure before returning to user mode (Figure 8.2).

```
algorithm open          /* for device drivers */
input:  pathname
       openmode
output: file descriptor
{
    convert pathname to inode, increment inode reference count,
    allocate entry in file table, user file descriptor,
    as in open of regular file;

    get major, minor number from inode;

    save context (algorithm setjmp) in case of long jump from driver;

    if (block device)
    {
        use major number as index to block device switch table;
        call driver open procedure for index;
        pass minor number, open modes;
    }
    else
    {
        use major number as index to character device switch table;
        call driver open procedure for index;
        pass minor number, open modes;
    }

    if (open fails in driver)
        decrement file table, inode counts;
}
```

Figure 8.2: Algorithm for Opening a Device

For a block device, it invokes the open procedure encoded in the block device switch table, and for a character device, it invokes the open procedure in the character device switch table. If a device is both a block and a character device, the kernel will invoke the appropriate open procedure depending on the particular device file the user opened: The two open procedures may even be identical, depending on the driver.

The device-specific open procedure establishes a connection between the calling process and the opened device and initializes private driver data structures.

Close:

A process severs its connection to an open device by closing it. However, the kernel invokes the device-specific close procedure only for the last close of the device that is, only if no other processes have the device open, because the device close procedure terminates hardware connections; clearly this must wait until no processes are accessing the device. Because the kernel invokes the device open procedure during every open system call but invokes the device close procedure only once, the device driver is never sure how many processes are still using the device.

```
algorithm close          /* for devices */
input: file descriptor
output: none
{
    do regular close algorithm (chapter 5xxx);
    if (file table reference count not 0)
        goto finish;
    if (there is another open file and its major, minor numbers
        are same as device being closed)
        goto finish;          /* not last close after all */
    if (character device)
    {
        use major number to index into character device switch table;
        call driver close routine: parameter minor number;
    }
    if (block device)
    {
        if (device mounted)
            goto finish;
        write device blocks in buffer cache to device;
        use major number to index into block device switch table;
        call driver close routine: parameter minor number;
        invalidate device blocks still in buffer cache;
    }
    finish:
        release inode;
}
```

Figure 8.3: Algorithm for Closing a Device

The algorithm for closing a device is similar to the algorithm for closing a regular file (Figure 8.3). However, before the kernel releases the inode it does operations specific to device files.

1. It searches the file table to make sure that no other processes still have the device open.

2. For a character device, the kernel invokes the device close procedure and returns to user mode. For a block device, the kernel searches the mount table to make sure that the device does not contain a mounted file system. If there is a mounted file system from the block device, the kernel cannot invoke the device close procedure, because it is not the last close of the device.
3. The kernel releases the inode of the device file.

Read and Write:

The kernel algorithms for read and write of a device are similar to those for a regular file. If the process is reading or writing a character device, kernel invokes the device drivers read or write procedure.

ioctl:

The `ioctl` system call is a generalization of the terminal-specific `stty` (set terminal settings) and `gtty` (get terminal settings) system calls available in earlier versions of the UNIX system. It provides a general, catch-all entry point for device specific commands, allowing a process to set hardware options associated with a device and software options associated with the driver.

The specific actions specified by the `ioctl` call vary per device and are defined by the device driver. Programs that use `ioctl` must know what type of file they are dealing with, because they are device-specific. This is an exception to the general rule that the system does not differentiate between different file types.

The syntax of the system call is

ioctl (fd, command, arg);

Where `fd` is the file descriptor returned by a prior `open` system call, `command` is a request of the driver to do a particular action, and `arg` is a parameter (possibly a pointer to a structure) for the command.

Commands are driver specific; hence, each driver interprets commands according to internal specifications, and the format of the data structure `arg` depends on the command. Drivers can read the data structure `arg` from user space according to predefined formats, or they can write device settings into user address space at `arg`.

Other File System Related Calls:

File system calls such as `stat` and `chmod` work for devices as they do for regular files; they manipulate the inode without accessing the driver.

Even the lseek system call works for devices. For example, if a process seeks to a particular byte offset on a tape, the kernel updates the file table offset but does no driver-specific operations. When the process later reads or writes, the kernel moves the file table offset to the u area, as is done for regular files. and the device physically seeks to the correct offset indicated in the u area.

DISK DRIVERS:

The disk driver translates a file system address, consisting of a logical device number and block number, to a particular sector on the disk. The driver gets the address in one of two ways: Either the strategy procedure uses a buffer from the buffer pool and the buffer header contains the device and block number, or the read and write procedures are passed the logical (minor) device number as a parameter; they convert the byte offset saved in the u area to the appropriate block address.

The disk driver uses the device number to identify the physical drive and particular section to be used, maintaining internal tables to find the sector that marks the beginning of a disk section. Finally, it adds the block number of the file system to the start sector number to identify the sector used for the I/O transmission.

Historically, the sizes and lengths of disk sections have been fixed according to the disk type. For instance, the DEC RP07 disk is partitioned into the sections shown in Figure 8.4. Suppose the files `"/dev/dsk0"`, `"/dev/dsk1"`, `"/dev/dsk2"` and `"/dev/dsk3"` correspond to sections 0 through 3 of an RP07 disk and have minor numbers 0 through 3.

Section	Start Block	Length in Blocks
Size of block = 512 bytes		
0	0	64000
1	64000	944000
2	168000	840000
3	336000	672000
4	504000	504000
5	672000	336000
6	840000	168000
7	0	1008000

Figure 8.4: Disk Sections for RP07 Disk

Assume the size of a logical file system block is the same as that of a disk block. If the kernel attempts to access block 940 in the file system contained in `"/dev/dsk3"`, the disk driver converts the request to access block 336940 (section 3 starts at block 336000; $336000 + 940 = 336940$) on the disk.

The sizes of disk sections vary, and administrators configure file systems in sections of the appropriate size: Large file systems go into large sections, and so on. Sections may overlap on disk. For example, Sections 0 and 1 in the RP07 disk are disjoint, but together they cover blocks 0 to 1008000, the entire disk. Section 7 also covers the entire disk.

The overlap of sections does not matter, provided that the file systems contained in the sections are configured such that they do not overlap. It is advantageous to have one section include the entire disk, since the entire volume can thus be quickly copied.

The use of fixed sections restricts the flexibility of disk configuration. The hard-coded knowledge of disk sections should not be put into the disk driver but should be placed in a configurable volume table of contents on the disk. However, it is difficult to find a generic position on all disks for the volume table of contents and retain compatibility with previous versions of the system.

Current implementations of System V expect the boot block of the first file system on a disk to occupy the first sector of the volume, although that is the most logical place for a volume table of contents. Nevertheless, the disk driver could contain hard-coded information on where the volume table of contents is stored for that particular disk, allowing variable sized disk sections.

Because of the high level of disk traffic typical of UNIX systems, the disk driver must maximize data throughput to get the best system performance. Most modem disk controllers take care of disk job scheduling, positioning the disk arm, and transferring data between the disk and the CPU; otherwise, the disk driver must do these tasks.

TERMINAL DRIVERS:

Terminal drivers have the same function as other drivers: to control the transmission of data to and from terminals. However, terminals are special, because they are the user's interface to the system. To accommodate interactive use of the UNIX system, terminal drivers contain an internal interface to line discipline modules, which interpret input and output.

In canonical mode, the line discipline converts raw data sequences typed at the keyboard to a canonical form (what the user really meant) before sending the data to a receiving process; the line discipline also converts raw output sequences written by a process to a format that the user expects. In raw mode, the line discipline passes data between processes and the terminal without such conversions.

For example, programmers are notoriously fast but error-prone typists. Terminals provide an "erase" key (or such a key can be so designated) such that the user can logically erase part of the typed sequence and enter corrections.

The terminal sends the entire sequence to the machine, including the erase characters. In canonical mode, the line discipline buffers the data into lines (the sequence of characters until a carriage-return character) and processes erase characters internally before sending the revised sequence to the reading process.

The functions of a line discipline are

- To parse input strings into lines
- To process erase characters
- To process a "kill" character that invalidates all characters typed so far on the current line
- To echo (write) received characters to the terminal
- To expand output such as tab characters to a sequence of blank spaces
- To generate signals to processes for terminal hang-ups, line breaks, or in response to a user hitting the delete key
- To allow a raw mode that does not interpret special characters such as erase, kill or carriage return.

The support of raw mode implies the use of an asynchronous terminal, because processes can read characters as they are typed instead of waiting until a user hits a carriage return or "enter" key.

Ritchie notes that the original terminal line disciplines used during system development in the early 1970s were in the shell and editor programs, not in the kernel. However, because their function is needed by many programs, their proper place is in the kernel.

Although the line discipline performs a function that places it logically between the terminal driver and the rest of the kernel, the kernel does not invoke the line discipline directly but only through the terminal driver.

Figure 8.5 shows the logical flow of data through the terminal driver and line discipline and the corresponding flow of control through the terminal driver.

Users can specify what line discipline should be used via an *ioctl* system call, but it is difficult to implement a scheme such that one device uses several line disciplines simultaneously, where each line discipline module successively calls the next module to process the data in turn.

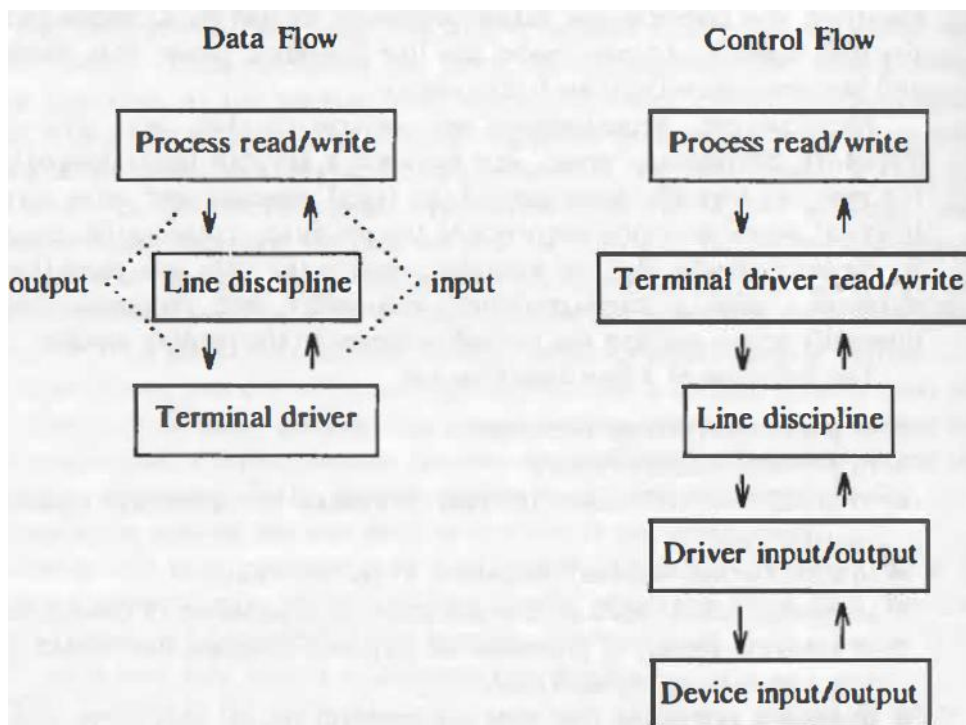


Figure 8.5: Call Sequence and Data Flow through Line Discipline

Clists:

Line disciplines manipulate data on clists. A clist, or character list, is a variable length linked list of cblocks with a count of the number of characters on the list. A cblock contains a pointer to the next cblock on the linked list, a small character array to contain data, and a set of offsets indicating the position of the valid data in the cblock (Figure 8.6).

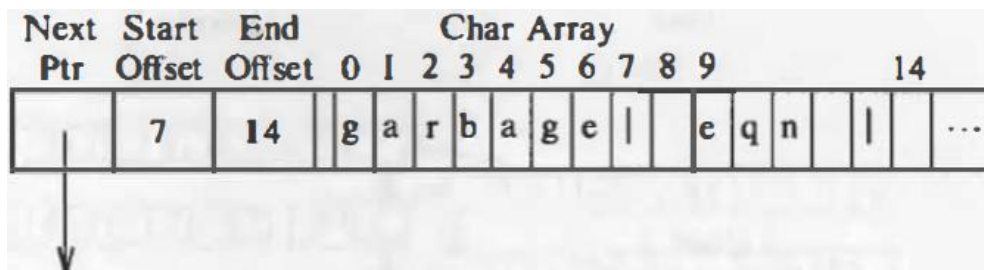


Figure 8.6: A Cblock

The start offset indicates the first location of valid data in the array, and the end offset indicates the first location of non-valid data. The kernel maintains a linked list of free cblocks and has six operations on clists and cblocks.

1. It has an operation to assign a cblock from the free list to a driver.

2. It also has an operation to return a cblock to the free list.
3. The kernel can retrieve the first character from a clist: It removes the first character from the first cblock on the clist and adjusts the clist character count and the indices into the cblock so that subsequent operations will not retrieve the same character. If a retrieval operation consumes the last character of a cblock, the kernel places the empty cblock on the free list and adjusts the clist pointers. If a clist contains no characters when a retrieval operation is done, the kernel returns the null character.
4. The kernel can place a character onto the end of a clist by finding the last cblock on the clist, putting the character onto it, and adjusting the offset values. If the cblock is full, the kernel allocates a new cblock, links it onto the end of the clist, and places the character into the new cblock.
5. The kernel can remove a group of characters from the beginning of a clist one cblock at a time, the operation being equivalent to removing all the characters in the cblock one at a time.
6. The kernel can place a cblock of characters onto the end of a clist.

Clists provide a simple buffer mechanism, useful for the small volume of data transmission typical of slow devices such as terminals. They allow manipulation of data one character at a time or in groups of cblocks.

The Terminal Driver in Canonical Mode:

The data structures for terminal drivers have three clists associated with them: a clist to store data for output to the terminal, a clist to store "raw" input data provided by the terminal interrupt handler as the user typed it in, and a clist to store "cooked" input data, after the line discipline converts special characters in the raw clist, such as the erase and kill characters.

When a process writes a terminal (Figure 8.7), the terminal driver invokes the line discipline. The line discipline loops, reading output characters from user address space and placing them onto the output clist, until it exhausts the data.

The line discipline processes output characters, expanding tab characters to a series of space characters, for example. If the number of characters on the output clist becomes greater than a high-water mark, the line discipline calls driver procedures to transmit the data on the output clist to the terminal and puts the writing process to sleep.

When the amount of data on the output clist drops below a low-water mark, the interrupt handler awakens all processes asleep on the event the terminal can accept more data.

The line discipline finishes its loop, having copied all the output data from user space to the output clist, and calls driver procedures to transmit the data to the terminal.

```
algorithm terminal_write
{
    while (more data to be copied from user space)
    {
        if (tty flooded with output data)
        {
            start write operation to hardware with data
                on output clist;
            sleep (event: tty can accept more data);
            continue; /* back to while loop */
        }
        copy cblock size of data from user space to output clist;
        line discipline converts tab characters, etc;
    }

    start write operation to hardware with data on output clist;
}
```

Figure 8.7: Algorithm for Writing Data to a Terminal

If multiple processes write to a terminal, they follow the given procedure independently. The output could be garbled; that is, data written by the processes may be interleaved on the terminal. This could happen because a process may write the terminal using several write system calls.

The kernel could switch context while the process is in user mode between successive write system calls, and newly scheduled processes could write the terminal while the original process sleeps.

Output data could also be garbled at a terminal because a writing process may sleep in the middle of a write system call while waiting for previous output data to drain from the system.

The kernel could schedule other processes that write the terminal before the original process is rescheduled. Because of this case, the kernel does not guarantee that the contents of the data buffer to be output by a write system call appear contiguously on the terminal.

Reading data from a terminal in canonical mode is a more complex operation. The read system call specifies the number of bytes the process wants to read, but the line discipline satisfies the read on receipt of a carriage return even though the character count is not satisfied.

This is practical, since it is impossible for a process to predict how many characters the user will enter at the keyboard, and it does not make sense to wait for the user to type a large number of characters. Figure 8.8 shows the algorithm for reading a terminal.

```

algorithm terminal_read
{
    if (no data on canonical clist)
    {
        while (no data on raw clist)
        {
            if (tty opened with no delay option)
                return;
            if (tty in raw mode based on timer and timer not active)
                arrange for timer wakeup (callout table);
            sleep (event: data arrives from terminal);
        }

        /* there is data on raw clist */
        if (tty in raw mode)
            copy all data from raw clist to canonical clist;
        else /* tty is in canonical mode */
        {
            while (characters on raw clist)
            {
                copy one character at a time from raw clist
                to canonical clist:
                do erase, kill processing;
                if (char is carriage return or end-of-file)
                    break; /* out of while loop */
            }
        }
    }

    while (characters on canonical list and read count not satisfied)
        copy from cblocks on canonical list to user address space;
}

```

Figure 8.8: Algorithm for Reading a Terminal

When the user types an "end of file" character (ASCII control-d), the line discipline satisfies terminal reads of the input string up to, but not including, the end of file character. It returns no data (return value 0) for the read system call that encounters only the end of file on the clists; the calling process is responsible for recognizing that it has read the end of file and that it should no longer read the terminal.

The Terminal Driver in Raw Mode:

Users set terminal parameters such as erase and kill characters and retrieve the values of current settings with the `ioctl` system call. Similarly, they control whether the terminal echoes its input, set the terminal baud rate (the rate of bit transfers), flush input and output character queues, or manually start up or stop character output.

The terminal driver data structure saves various control settings, and the line discipline receives the parameters of the `ioctl` call and sets or gets the relevant fields in the terminal data structure. When a process sets terminal parameters, it does so for all processes using the terminal. The terminal settings are not automatically reset when the process that changed the settings exits.

Processes can also put the terminal into raw mode, where the line discipline transmits characters exactly as the user typed them: No input processing is done at all. Still, the kernel must know when to satisfy user read calls, since the carriage return is treated as an ordinary input character. It satisfies read system calls after a minimum number of characters are input at the terminal, or after waiting a fixed time from the receipt of any characters from the terminal.

Terminal Polling:

It is sometimes convenient to poll a device, that is, to read it if there is data present but to continue regular processing otherwise. The BSD system has a `select` system call that allows device polling. The syntax of the call is

select(nfds, rfds, wfds, efds, timeout)

Where `nfds` gives the number of file descriptors being selected, and `rfds`, `wfds` and `efds` point to bit masks that "select" open file descriptors. That is, the bit $1 \ll fd$ (1 shifted left by the value of the file descriptor) is set if a user wants to select that file descriptor. Timeout indicates how long `select` should sleep, waiting for data to arrive, for example; if data arrives for any file descriptors and the timeout value has not expired, `select` returns, indicating in the bit masks which file descriptors were selected.

Establishment of a Control Terminal:

The control terminal is the terminal on which a user logs into the system, and it controls processes that the user initiates from the terminal. When a process opens a terminal, the terminal driver opens the line discipline.

If the process is a process group leader as the result of a prior `setpgrp` system call and if the process does not have an associated control terminal, the line discipline makes the opened terminal the control terminal.

It stores the major and minor device number of the terminal device file in the u area, and it stores the process group number of the opening process in the terminal driver data structure.

The control terminal plays an important role in handling signals. When a user presses the delete, break, rubout (erase), or quit keys, the interrupt handler invokes the line discipline, which sends the appropriate signal to all processes in the control process group. Similarly, if the user hangs up, the terminal interrupt handler receives a hangup indication from the hardware, and the Line discipline sends a hangup signal to all processes in the process group.

Indirect Terminal Driver:

Processes frequently have a need to read or write data directly to the control terminal, even though the standard input and output may have been redirected to other files. For example, a shell script can send urgent messages directly to the terminal, although its standard output and standard error files may have been redirected elsewhere. UNIX systems provide "indirect" terminal access via the device file `"/dev/tty"`, which designates the control terminal for every process that has one. Users logged onto separate terminals can access `"/dev/tty"`, but they access different terminals.

There are two common implementations for the kernel to find the control terminal from the file name `"/dev/tty"`. First, the kernel can define a special device number for the indirect terminal file with a special entry in the character device switch table.

When invoking the indirect terminal, the driver for the indirect terminal gets the major and minor number of the control terminal from the u area and invokes the real terminal driver through the character device switch table.

The second implementation commonly used to find the control terminal from the name `"/dev/tty"` tests if the major number is that of the indirect terminal before calling the driver open routine. If so, it releases the inode for `"/dev/tty"`, allocates the inode for the control terminal, resets the file table entry to point to the control terminal inode, and calls the open routine of the terminal driver. The file descriptor returned when opening `"/dev/tty"` refers directly to the control terminal and its regular driver.

Logging In:

Process 1, `init`, executes an infinite loop, reading the file `"/etc/inittab"` for instructions about what to do when entering system states such as "single user" or "multi-user." In multi-user state, a primary responsibility of `init` is to allow users to log into terminals (Figure 8.9).

```

algorithm login          /* procedure for logging in */
{
    getty process executes:
    set process group (setpgrp system call);
    open tty line;      /* sleeps until opened */
    if (open successful)
    {
        exec login program:
        prompt for user name;
        turn off echo, prompt for password;
        if (successful) /* matches password in /etc/passwd */
        {
            put tty in canonical mode (ioctl);
            exec shell;
        }
        else
            count login attempts, try again up to a point;
    }
}

```

Figure 8.9: Algorithm for Logging In

It spawns processes called getty (for get terminal or get "tty") and keeps track of which getty process opens which terminal; each getty process resets its process group using the setpgrp system call, opens a particular terminal line, and usually sleeps in the open until the machine senses a hardware connection for the terminal.

When the open returns getty execs the login program; which requires users to identify themselves by login name and password. If the user logs in successfully, login finally execs the shell and the user starts working. This invocation of the shell is called the login shell.

The shell process has the same process ID as the original getty process, and the login shell is therefore a process group leader. If a user does not log in successfully, login exits after a suitable time limit, closing the opened terminal line, and init spawns another getty for the line.

Init pauses until it receives a death of child signal. On waking up, it finds out if the zombie process had been a login shell and, if so, spawns another getty process to open the terminal in place of the one that died.

STREAMS:

A stream is a full-duplex connection between a process and a device driver. It consists of a set of linearly linked queue pairs, one member of each pair for input and the other for output. When a process writes data to a stream, the kernel sends the data down the output queues; when a device driver receives input data, it sends the data up the input queues to a reading process.

The queues pass messages to neighboring queues according to a well-defined interface. Each queue pair is associated with an instance of a kernel module, such as a driver, line discipline, or protocol and the modules manipulate data passed through its queues.

Each queue is a data structure that contains the following elements:

- ✓ An open procedure, called during an open system call
- ✓ A close procedure, called during a close system call
- ✓ A "put" procedure, called to pass a message into the queue
- ✓ A "service" procedure, called when a queue is scheduled to execute
- ✓ A pointer to the next queue in the stream
- ✓ A pointer to a list of messages awaiting service
- ✓ A pointer to a private data structure that maintains the state of the queue
- ✓ Flags and high- and low-water marks, used for flow control, scheduling, and maintaining the queue state.

The kernel allocates queue pairs, which are adjacent in memory; hence, a queue can easily find the other member of the pair.

A device with a streams driver is a character device; it has a special field in the character device switch table that points to a streams initialization structure, containing the addresses of routines and high- and low-water marks.

When the kernel executes the open system call and discovers that the device file is character special, it examines the new field in the character device switch table. If there is no entry there, the driver is not a streams driver, and the kernel follows the usual procedure for character devices.

However, for the first open of a streams driver, the kernel allocates two pairs of queues, one for the stream-head and the other for the driver. The stream-head module is identical for all instances of open streams: It has generic put and service procedures and is the interface to higher-level kernel modules that implement the read, write, and ioctl system calls.

The kernel initializes the driver queue structure, assigning queue pointers and copying addresses of driver routines from a per-driver initialization structure, and invokes the driver open procedure. The driver open procedure does the usual initialization but also saves information to recall the queue with which it is associated.

Finally, the kernel assigns a special pointer in the in-core inode to indicate the stream-head (Figure 8.10).

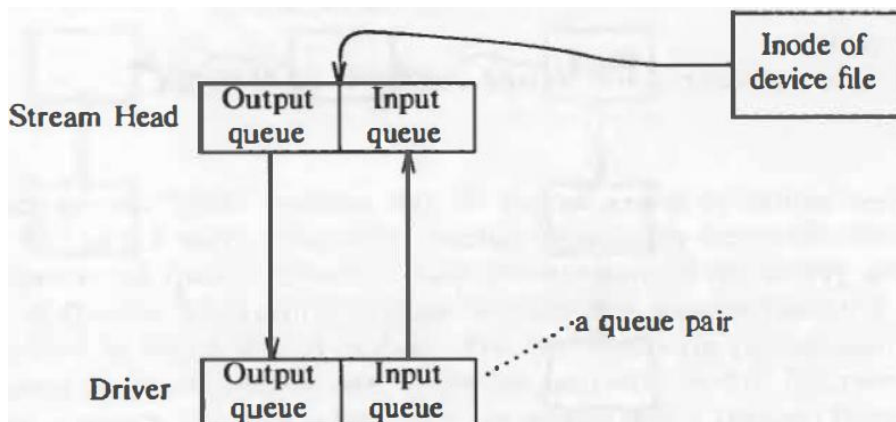


Figure 8.10: A Stream after Open

When another process opens the device, the kernel finds the previously allocated stream via the inode pointer and invokes the open procedure of all modules on the stream. Modules communicate by passing messages to neighboring modules on a stream.

A message consists of a linked list of message block headers; each block header points to the start and end location of the block's data. There are two types of messages - *control* and *data* - identified by a type indicator in the message header.

Control messages may result from ioctl system calls or from special conditions, such as a terminal hang-up, and *data messages* may result from write system calls or the arrival of data from a device.

When a process writes a stream, the kernel copies the data from user space into message blocks allocated by the stream-head. The stream-head module invokes the put procedure of the next queue module, which may process the message, pass it immediately to the next queue, or enqueue it for later processing.

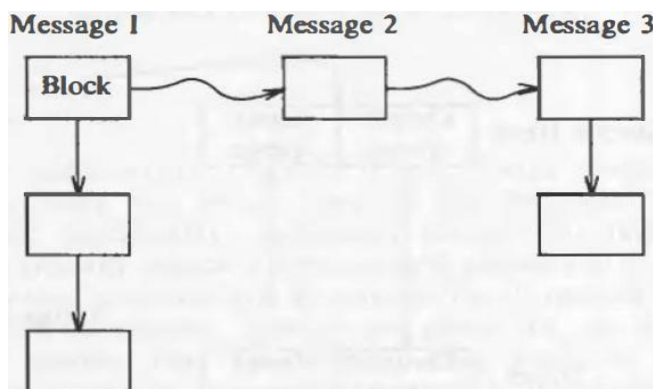


Figure 8.11: Streams Messages

In the latter case, the module links the message block headers on a linked list, forming a two way linked list (Figure 8.11). Then it sets a flag in its queue data structure to indicate that it has data to process, and schedules itself for servicing. The module places the queue on a linked list of queues requesting service and invokes a scheduling mechanism; that scheduler calls the service procedures of each queue on the list.

Analysis of Streams:

Ritchie mentions that he tried to implement streams only with put procedures or only with service procedures. However, the service procedure is necessary for flow control, since modules must sometimes enqueue data if neighboring modules cannot receive any more data temporarily. The put procedure interface is also necessary, because data must sometimes be delivered to a neighboring module right away.

It would also have been preferable to implement each module as a separate process, but use of a large number of modules could cause the process table to overflow. They are implemented with a special scheduling mechanism – software interrupt - independent of the normal process scheduler. Therefore, modules cannot go to sleep, because they would be putting an arbitrary process to sleep (the one that was interrupted).

Several anomalies exist in the implementation of streams.

- Process accounting is difficult under streams, because modules do not necessarily run in the context of the process that is using the stream.
- Users can put a terminal driver into raw mode, such that read calls return after a short time if no data is available. It is difficult to implement this feature with streams, unless special-case code is introduced at the stream-head level.
- Streams are linear connections and do not easily allow multiplexing in the kernel.

In spite of these anomalies, streams hold great promise for improving the design of driver modules.

INTERPROCESS COMMUNICATION:

Inter process communication mechanisms allow arbitrary processes to exchange data and synchronize execution. We have several forms of inter process communication, such as pipes, named pipes, and signals.

Pipes (unnamed) suffer from the drawback that they are known only to processes which are descendants of the process that invoked the pipe system call: Unrelated processes cannot communicate via pipes.

Although named pipes allow unrelated processes to communicate, they cannot generally be used across a network nor do they readily lend themselves to setting up multiple communications paths for different sets of communicating processes: it is impossible to multiplex a named pipe to provide private channels for pairs of communicating processes.

Arbitrary processes can also communicate by sending signals via the kill system call, but the “message” consists only of the signal number.

PROCESS TRACING:

The UNIX system provides a primitive form of inter process communication for tracing processes, useful for debugging. A debugger process, such as sdb, spawns a process to be traced and controls its execution with the ptrace system call, setting and clearing break points, and reading and writing data in its virtual address space. The syntax for ptrace system call:

ptrace(cmd, pid, addr, data);

Process tracing thus consists of synchronization of the debugger process and the traced process and controlling the execution of the traced process. The pseudo-code in Figure 8.12 shows the typical structure of a debugger program. The debugger spawns a child process, which invokes the ptrace system call and, as a result, the kernel sets a trace bit in the child process table entry; the child now execs the program being traced.

```

11 if ((pid = fork()) == 0)
{
    /* child - traced process */
    ptrace(0, 0, 0, 0);
    exec("name of traced process here");
}
/* debugger process continues here */
for (;;)
{
    wait((int *) 0);
    read(input for tracing instructions)
    ptrace(cmd, pid, ...);
    if (quitting trace)
        break;
}

```

Figure 8.12: Structure of Debugging Process

Where cmd specifies various commands such as reading data, writing data, resuming execution and so on, pid is the process ID of the traced process, addr is the virtual address to be read or written in the child process, and data is an integer value to be written.

When executing the ptrace system call, the kernel verifies that the debugger has a child whose ID is pid and that the child is in the traced state and then uses a global trace data structure to transfer data between the two processes.

It locks the trace data structure to prevent other tracing processes from overwriting it, copies cmd, addr, and data into the data structure, wakes up the child process and puts it into the "ready-to-run" state, then sleeps until the child responds.

When the child resumes execution (in kernel mode), it does the appropriate trace command, writes its reply into the trace data structure, then awakens the debugger. Depending on the command type, the child may reenter the trace state and wait for a new command or return from handling signals and resume execution.

When the debugger resumes execution, the kernel saves the "return value" supplied by the traced process, unlocks the trace data structure, and returns to the user. If the debugger process is not sleeping in the wait system call when the child enters the trace state, it will not discover its traced child until it calls wait, at which time it returns immediately and proceeds.

The use of ptrace for process tracing is primitive and suffers several drawbacks:

- The kernel must do four context switches to transfer a word of data between a debugger and a traced process. The overhead is necessary, because a debugger has no other way to gain access to the virtual address space of a traced process, but process tracing is consequently slow.
 - ✓ The kernel switches context in the debugger in the ptrace call until the traced process replies to a query.
 - ✓ The Kernel switches context to and from the traced process.
 - ✓ And the kernel switches context back to the debugger process with the answer to the ptrace call.
- A debugger process can trace several child processes simultaneously, although this feature is rarely used in practice. More critically, a debugger can only trace child processes: If a traced child forks, the debugger has no control over the grandchild, a severe handicap when debugging sophisticated programs.
- A debugger cannot trace a process that is already executing if the debugged process had not called ptrace to let the kernel know that it consents to be traced. This is inconvenient, because a process that needs debugging must be killed and restarted in trace mode.

- It is impossible to trace setuid pr{}grams, because users could violate security by writing their address space via ptrace and doing illegal operations.

SYSTEM V IPC:

The UNIX System V IPC package consists of three mechanisms:

1. Messages allow processes to send formatted data streams to arbitrary processes,
2. Shared memory allows processes to share parts of their virtual address space.
3. Semaphores allow processes to synchronize execution.

Implemented as a unit, they share common properties.

- Each mechanism contains a table whose entries describe all instances of the mechanism.
- Each entry contains a numeric key, which is its user-chosen name.
- Each mechanism contains a "get" system call to create a new entry or to retrieve an existing one, and the parameters to the calls include a key and flags. The kernel searches the proper table for an entry named by the key.
 - ✓ Processes can call the "get" system calls with the key IPC_PRIVATE to assure the return of an unused entry.
 - ✓ They can set the IPC_CREAT bit in the flag field to create a new entry if one by the given key does not already exist.
 - ✓ And they can force an error notification by setting the IPC_EXCL and IPC_CREAT flags, if an entry already exists for the key.
 - ✓ The "get" system calls return a kernel-chosen descriptor for use in the other system calls and are thus analogous to the file system creat and open calls.
- For each IPC mechanism, the kernel uses the following formula to find the index into the table of data structures from the descriptor: ***index - descriptor modulo (number of entries in table)***
- Each IPC entry has a permissions structure that includes the user ID and group ID of the process that created the entry, a user and group ID set by the "control" system call (below), and a set of read-write-execute permissions for user, group, and others, similar to the file permission modes.

- Each entry contains other status information, such as the process ID of the last process to update the entry (send a message, receive a message, attach shared memory, and so on), and the time of last access or update.
- Each mechanism contains a "control" system call to query status of an entry, to set status information, or to remove the entry from the system. When a process queries the status of an entry, the kernel verifies that the process has read permission and then copies data from the table entry to the user address.

Messages:

There are four system calls for messages: `msgget` returns (and possibly creates) a message descriptor that designates a message queue for use in other system calls, `msgctl` has options to set and return parameters associated with a message descriptor and an option to remove descriptors, `msgsnd` sends a message, and `msgrcv` receives a message.

The syntax of the `msgget` system call is: **`msgqid = msgget(key, flag);`** where `msgqid` is the descriptor returned by the call.

The kernel stores messages on a linked list (queue) per descriptor, and it uses `msgqid` as an index into an array of message queue headers. In addition to the general IPC permissions field the queue structure contains the following fields:

- Pointers to the first and last messages on a linked list;
- The number of messages and the total number of data bytes on the linked list;
- The maximum number of bytes of data that can be on the linked list;
- The process IDs of the last processes to send and receive messages;
- Time stamps of the last `msgsnd`, `msgrcv`, and `msgctl` operations.

When a user calls `msgget` to create a new descriptor, the kernel searches the array of message queues to see if one exists with the given key. If there is no entry for the specified key, the kernel allocates a new queue structure, initializes it, and returns an identifier to the user. Otherwise, it checks permissions and returns.

A process uses the `msgsnd` system call to send a message: **`msgsnd(msgqid, msg, count, flag);`** where `msgqid` is the descriptor of a message queue typically returned by a `msgget` call, `msg` is a pointer to a structure consisting of a user-chosen integer type and a character array, `count` gives the size of the data array, and `flag` specifies the action the kernel should take if it runs out of internal buffer space.

The kernel checks (Figure 8.13) that the sending process has write permission for the message descriptor, that the message length does not exceed the system limit, that the message queue does not contain too many bytes, and that the message type is a positive integer. If all tests succeed, the kernel allocates space for the message from a message map and copies the data from user space.

The kernel allocates a message header and puts it on the end of the linked list of message headers for the message queue. It records the message type and size in the message header, sets the message header to point to the message data, and updates various statistics fields (number of messages and bytes on queue, time stamps and process ID of sender) in the queue header. The kernel then awakens processes that were asleep, waiting for messages to arrive on the queue.

```

algorithm msgsnd          /* send a message */
input: (1) message queue descriptor
          (2) address of message structure
          (3) size of message
          (4) flags
output: number of bytes sent
{
    check legality of descriptor, permissions;
    while (not enough space to store message)
    {
        if (flags specify not to wait)
            return;
        sleep(until event enough space is available);
    }
    get message header;
    read message text from user space to kernel;
    adjust data structures: enqueue message header,
                           message header points to data,
                           counts, time stamps, process ID;
    wakeup all processes waiting to read message from queue;
}

```

Figure 8.13: Algorithm for Msgsnd

If the number of bytes on the queue exceeds the queue's limit, the process sleeps until other messages are removed from the queue. If the process specified not to wait (flag `IPC_NOWAIT`), however, it returns immediately with an error indication.

A process receives messages by

count = msgrcv(id, msg, maxcount, type, flag);

Where `id` is the message descriptor, `msg` is the address of a user structure to contain the received message, `maxcount` is the size of the data array in `msg`, `type` specifies the message type the user wants to read, and `flag` specifies what the kernel should do if no messages are on the queue. The return value, `count`, is the number of bytes returned to the user.

The kernel checks (figure 8.14) that the user has the necessary access rights to the message queue. If the requested message type is 0, the kernel finds the first message on the linked list.

```

algorithm msgrcv      /* receive message */
input: (1) message descriptor
       (2) address of data array for incoming message
       (3) size of data array
       (4) requested message type
       (5) flags
output: number of bytes in returned message
{
    check permissions;
loop:
    check legality of message descriptor;
    /* find message to return to user */
    if (requested message type == 0)
        consider first message on queue;
    else if (requested message type > 0)
        consider first message on queue with given type;
    else /* requested message type < 0 */
        consider first of the lowest typed messages on queue,
            such that its type is <= absolute value of
            requested type;
    if (there is a message)
    {
        adjust message size or return error if user size too small;
        copy message type, text from kernel space to user space;
        unlink message from queue;
        return;
    }
    /* no message */
    if (flags specify not to sleep)
        return with error;
    sleep (event message arrives on queue);
    goto loop;
}

```

Figure 8.14: Algorithm for Receiving a Message

If its size is less than or equal to the size requested by the user, the kernel copies the message data to the user data structure and adjusts its internal structures appropriately: it decrements the count of messages on the queue and the number of data bytes on the queue, sets the receive time and receiving process ID, adjusts the linked list, and frees the kernel space that had stored the message data.

If the process ignores size constraints, however (bit MSG_NOERROR is set in flag), the kernel truncates the message, returns the requested number of bytes, and removes the entire message from the list.

A process can receive messages of a particular type by setting the type parameter appropriately. If it is a positive integer, the kernel returns the first message of the given type. If it is negative, the kernel finds the lowest type of all messages on the queue, provided it is less than or equal to the absolute value of type, and returns the first message of that type.

A process can query the status of a message descriptor, set its status, and remove a message descriptor with the `msgctl` system call. The syntax of the call is **`msgctl(id, cmd, mstatbuf)`**; where `id` identifies the message descriptor, `cmd` specifies the type of command, and `mstatbuf` is the address of a user data structure that will contain control parameters or the results of a query.

Shared Memory:

Processes can communicate directly with each other by sharing parts of their virtual address space and then reading and writing the data stored in the shared memory. The system calls for manipulating shared memory are similar to the system calls for messages.

The **`shmget`** system call creates a new region of shared memory or returns an existing one, the **`shmat`** system call logically attaches a region to the virtual address space of a process, the **`shmdt`** system call detaches a region from the virtual address space of a process, and the **`shmctl`** system call manipulates various parameters associated with the shared memory.

Processes read and write shared memory using the same machine instructions they use to read and write regular memory. After attaching shared memory, it becomes part of the virtual address space of a process, accessible in the same way other virtual addresses are; no system calls are needed to access data in shared memory.

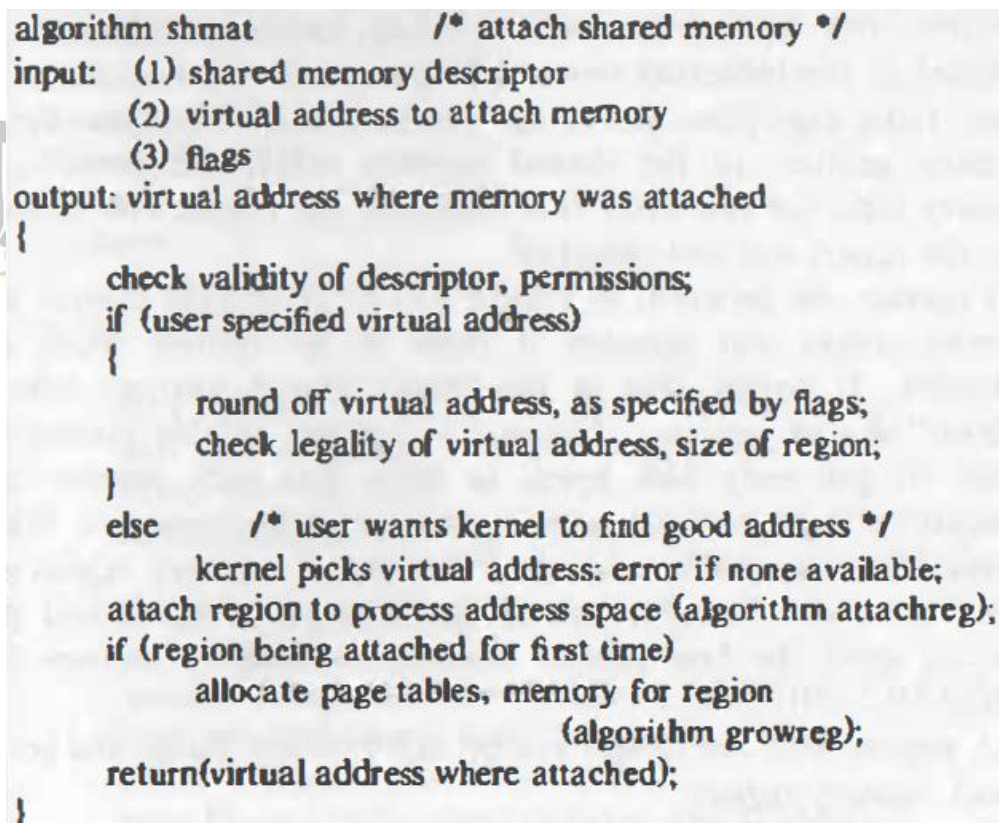
The syntax of the `shmget` system call is **`shmid = shmget(key, size, flag)`**; where `size` is the number of bytes in the region. The kernel searches the shared memory table for the given key: if it finds an entry and the permission modes are acceptable, it returns the descriptor for the entry.

If it does not find an entry and the user had set the `IPC_CREAT` flag to create a new region, the kernel verifies that the size is between system-wide minimum and maximum values and then allocates a region data structure using algorithm `allocreg`.

A process attaches a shared memory region to its virtual address space with the `shmat` system call: `virtaddr = shmat(id, addr, flags)`; where `id`, returned by a previous `shmget` system call, identifies the shared memory region, `addr` is the virtual address where the user wants to attach the shared memory, and `flags` specify whether the region is read-only and whether the kernel should round off the user-specified address.

The return value, `virtaddr`, is the virtual address where the kernel attached the region, not necessarily the value requested by the process.

When executing the `shmat` system call, the kernel verifies that the process has the necessary permissions to access the region (Figure 8.15). It examines the address the user specifies: If 0, the kernel chooses a convenient virtual address.



```

algorithm shmat          /* attach shared memory */
input:  (1) shared memory descriptor
        (2) virtual address to attach memory
        (3) flags
output: virtual address where memory was attached

    check validity of descriptor, permissions;
    if (user specified virtual address)
    {
        round off virtual address, as specified by flags;
        check legality of virtual address, size of region;
    }
    else /* user wants kernel to find good address */
        kernel picks virtual address: error if none available;
        attach region to process address space (algorithm attachreg);
        if (region being attached for first time)
            allocate page tables, memory for region
                (algorithm growreg);
        return(virtual address where attached);
}

```

Figure 8.15: Algorithm for Attaching Shared Memory

The shared memory must not overlap other regions in the process virtual address space; hence it must be chosen judiciously (meaning: with care) so that other regions do not grow into the shared memory.

A process detaches a shared memory region from its virtual address space by ***shmdt(addr)***; where *addr* is the virtual address returned by a prior *shmat* call.

A process uses the *shmctl* system call to query status and set parameters for the shared memory region: ***shmctl(id, cmd, shmstatbuf)***; where *id* identifies the shared memory table entry, *cmd* specifies the type of operation, and *shmstatbuf* is the address of a user-level data structure that contains the status information of the shared memory table entry when querying or setting its status.

The kernel treats the commands for querying status and changing owner and permissions similar to the implementation for messages. When removing a shared memory region, the kernel frees the entry and looks at the region table entry: If no process has the region attached to its virtual address space, it frees the region table entry and all its resources, using algorithm *freereg*.

If the region is still attached to some processes (its reference count is greater than 0), the kernel just clears the flag that indicates the region should not be freed when the last process detaches the region. Processes that are using the shared memory may continue doing so, but no new processes can attach it.

Semaphores:

The semaphore system calls allow processes to synchronize execution by doing a set of operations atomically on a set of semaphores. Before the implementation of semaphores, a process would create a lock file with the *creat* system call if it wanted to lock a resource: The *creat* fails if the file already exists, and the process would assume that another process had the resource locked.

The major disadvantages of this approach are that the process does not know when to try again, and lock files may inadvertently be left behind when the system crashes or is rebooted.

Dijkstra published the Dekker algorithm that describes an implementation of semaphores, integer-valued objects that have two atomic operations defined for them: P and V.

The P operation decrements the value of a semaphore if its value is greater than 0, and the V operation increments its value; because the operations are atomic, at most one P or V operation can succeed on a semaphore at any time.

The semaphore system calls in System V are a generalization of Dijkstra's P and V operations, in that several operations can be done simultaneously and the increment and decrement operations can be by values greater than 1.

The kernel does all the operations atomically; no other processes adjust the semaphore values until all operations are done. If the kernel cannot do all the operations, it does not do any; the process sleeps until it can do all the operations.

A semaphore in UNIX System V consists of the following elements:

- The value of the semaphore,
- The process ID of the last process to manipulate the semaphore,
- The number of processes waiting for the semaphore value to increase,
- The number of processes waiting for the semaphore value to equal 0.

The semaphore system calls are **semget** to create and gain access to a set of semaphores, **semctl** to do various control operations on the set, and **semop** to manipulate the values of semaphores.

The semget system call creates an array of semaphores:

```
id = semget(key, count, flag);
```

Where key, flag and id are similar to those parameters for messages and shared memory. The kernel allocates an entry those points to an array of semaphore structures with count elements (Figure 8.16).

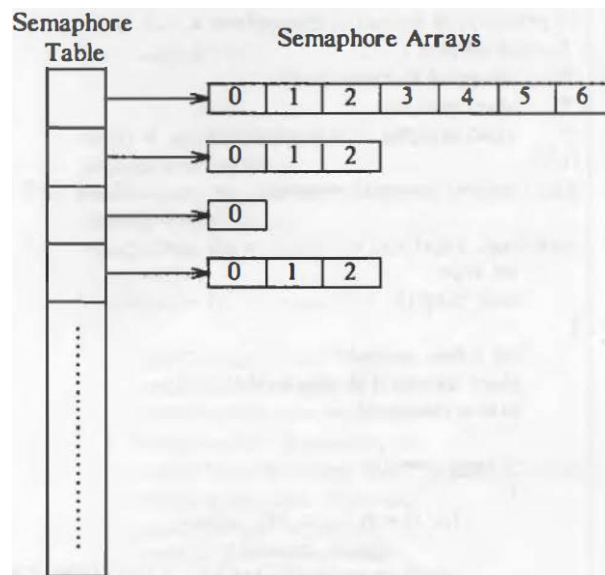


Figure 8.16: Data Structures for Semaphores

The entry also specifies the number of semaphores in the array, the time of the last semop call, and the time of the last semctl call.

Processes manipulate semaphores with the semop system call: ***oldval = semop(id, oplist, count)***; where *id* is the descriptor returned by *semget*, *oplist* is a pointer to an array of semaphore operations, and *count* is the size of the array. The return value, *oldval*, is the value of the last semaphore operated on in the set before the operation was done. The format of each element of *oplist* is:

- The semaphore number identifying the semaphore array entry being operated on
- The operation
- Flags

The kernel reads the array of semaphore operations, *oplist*, from the user address space and verifies that the semaphore numbers are legal and that the process has the necessary permissions to read or change the semaphores (Figure 8.17).

```

algorithm semop      /* semaphore operations */
inputs: (1) semaphore descriptor
        (2) array of semaphore operations
        (3) number of elements in array
output: start value of last semaphore operated on
{
    check legality of semaphore descriptor;
start: read array of semaphore operations from user to kernel space;
    check permissions for all semaphore operations;

    for (each semaphore operation in array)
    {
        if (semaphore operation is positive)
        {
            add "operation" to semaphore value;
            if (UNDO flag set on semaphore operation)
                update process undo structure;
            wakeup all processes sleeping (event semaphore value increases);
        }
        else if (semaphore operation is negative )
        {
            if ("operation" + semaphore value >= 0)
            {
                add "operation" to semaphore value;
                if (UNDO flag set)
                    update process undo structure;
                if (semaphore value 0)
                    /* continued next page */
            }
        }
    }
}

```

Figure 8.17: Algorithm for Semaphore Operation

The kernel changes the value of a semaphore according to the value of the operation.

- If positive, it increments the value of the semaphore and awakens all processes that are waiting for the value of the semaphore to increase.
- If the semaphore operation is 0, the kernel checks the semaphore value: If 0, it continues with the other operations in the array; otherwise, it increments the number of processes asleep, waiting for the semaphore value to be 0, and goes to sleep.
- If the semaphore operation is negative and its absolute value is less than or equal to the value of the semaphore, the kernel adds the operation value (a negative number) to the semaphore value.
- If the result is 0, the kernel awakens all processes asleep, waiting for the semaphore value to be 0. If the value of the semaphore is less than the absolute value of the semaphore operation, the kernel puts the process to sleep on the event that the value of the semaphore increases.

```

                                wakeup all processes sleeping (event
                                semaphore value becomes 0);
                                continue;
                                }
                                reverse all semaphore operations already done
                                this system call (previous iterations);
                                if (flags specify not to sleep)
                                    return with error;
                                sleep (event semaphore value increases);
                                goto start;    /* start loop from beginning */
                                }
                                else    /* semaphore operation is zero */
                                {
                                    if (semaphore value non 0)
                                    {
                                        reverse all semaphore operations done
                                        this system call;
                                        if (flags specify not to sleep)
                                            return with error;
                                        sleep (event semaphore value == 0);
                                        goto start;    /* restart loop */
                                    }
                                }
                                } /* for loop ends here */
                                /* semaphore operations all succeeded */
                                update time stamps, process ID's;
                                return value of last semaphore operated on before call succeeded;
                                }

```

Figure 8.17: Algorithm for Semaphore Operation (continued)

Whenever a process sleeps in the middle of a semaphore operation, it sleeps at an interruptible priority; hence, it wakes up on receipt of a signal.

NETWORK COMMUNICATIONS:

Programs such as mail, remote file transfer, and remote login that wish to communicate with other machines have historically used ad hoc methods to establish connections and to exchange data. For example, standard mail programs save the text of a user's mail messages in a particular file, such as `"/usr/mail/mjb"` for user `"mjb"`.

When a person sends mail to another user on the same machine, the mail program appends the mail to the addressee's file, using lock files and temporary files to preserve consistency. When a person reads mail, the mail program opens the person's mail file and reads the messages.

To send mail to a user on another machine, the mail program must ultimately find the appropriate mail file on the other machine. Since it cannot manipulate files there directly, a process on the other machine must act as an agent for the local mail process; hence the local process needs a way to communicate with its remote agent across machine boundaries. The local process is called the client of the remote server process.

Because the UNIX system creates new processes via the fork system call, the server process must exist before the client process attempts to establish a connection. It would be inconsistent with the design of the system if the remote kernel were to create a new process when a connection request comes across the network.

Instead, some process, usually *init*, creates a server process that reads a communications channel until it receives a request for service and then follows some protocol to complete the setup of the connection. Client and server programs typically choose the network media and protocols according to information in application data bases, or the data may be hard-coded into the programs.

Network communications have posed a problem for UNIX systems, because messages must frequently include data and control portions. The control portion may contain addressing information to specify the destination of a message. Addressing information is structured according to the type of network and protocol being used.

Hence, processes need to know what type of network they are talking to, going against the principle that users do not have to be aware of a file type, because all devices look like files. Traditional methods for implementing network communications consequently rely heavily on the `ioctl` system call to specify control information, but usage is not uniform across network types.

This has the unfortunate side effect that programs designed for one network may not be able to work for other networks. There has been considerable effort to improve network interfaces for UNIX systems. The streams implementation in the latest releases of System V provides an elegant mechanism for network support, because protocol modules can be combined flexibly by pushing them onto open streams and their use is consistent at user level.

SOCKETS:

Network Communications shows how processes on different machines can communicate, but the methods by which they establish communications are likely to differ, depending on protocols and media. Furthermore, the methods may not allow processes to communicate with other processes on the same machine, because they assume the existence of a server process that sleeps in a driver open or read system call. To provide common methods for inter process communication and to allow use of sophisticated network protocols, the BSD system provides a mechanism known as *sockets*.

Now we will see some user-level aspects of sockets.

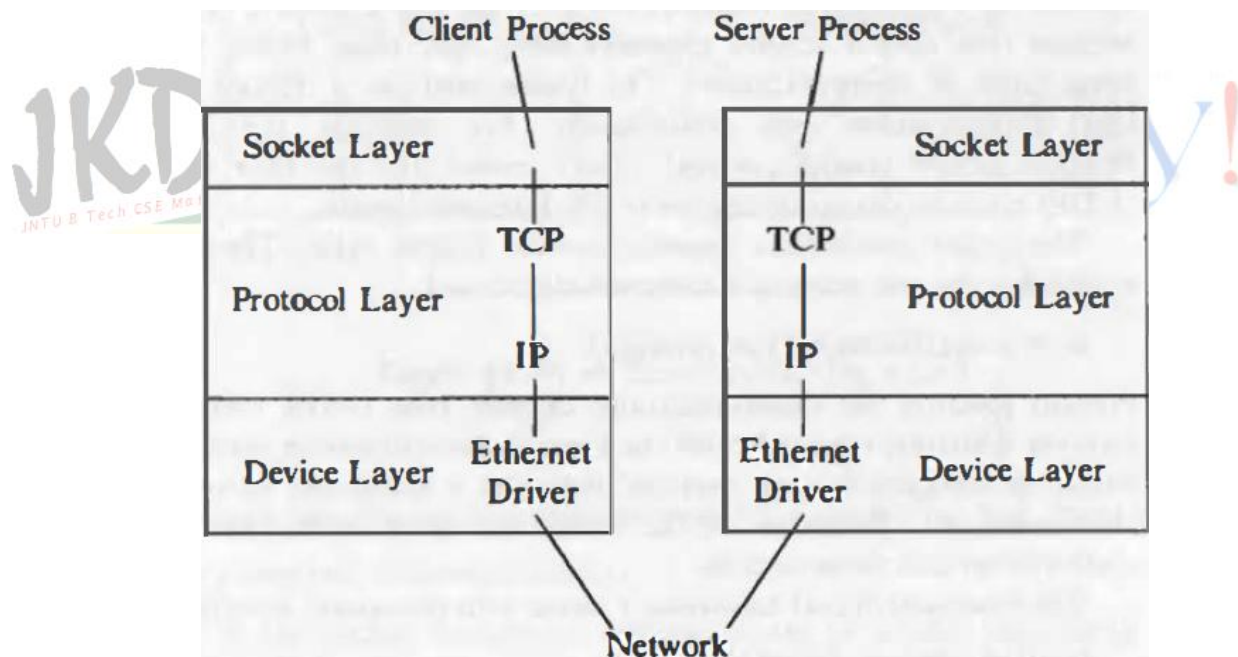


Figure 8.18: Sockets Model

The kernel structure consists of three parts: the socket layer, the protocol layer, and the device layer (Figure 8.18). The socket layer provides the interface between the system calls and the lower layers, the protocol layer contains the protocol modules used for communication (TCP and IP in the figure), and the device layer contains the device drivers that control the network devices.

Legal combinations of protocols and drivers are specified when configuring the system, a method that is not as flexible as pushing streams modules. Processes communicate using the client-server model: a server process listens to a socket, one end point of a two-way communications path, and client processes communicate to the server process over another socket, the other end point of the communications path, which may be on another machine. The kernel maintains internal connections and routes data from client to server.

Sockets that share common communications properties, such as naming conventions and protocol address formats, are grouped into domains. The 4.2 BSD system supports the "UNIX system domain" for processes communicating on one machine and the "Internet domain" for processes communicating across a network using the DARPA (Defense Advanced Research Project Agency) communications protocols.

Each socket has a type - a virtual circuit (stream socket in the Berkeley terminology) or datagram. A virtual circuit allows sequenced, reliable delivery of data. Datagram's do not guarantee sequenced, reliable, or unduplicated delivery, but they are less expensive than virtual circuit, because they do not require expensive setup operations; hence, they are useful for some types of communication.

The socket mechanism contains several system calls. The socket system call establishes the end point of a communications link.

sd = socket(format, type, protocol);

Format specifies the communications domain (the UNIX system domain or the Internet domain), type indicates the type of communication over the socket (virtual circuit or datagram), and protocol indicates a particular protocol to control the communication. Processes use the socket descriptor *sd* in other system calls. The close system call closes sockets.

The bind system call associates a name with the socket descriptor:

bind(sd, address, length);

sd is the socket descriptor, and *address* points to a structure that specifies an identifier specific to the communications domain and protocol specified in the socket system call. *Length* is the length of the address structure; without this parameter, the kernel would not know how long the address is because it can vary across domains and protocols.

The connect system call requests that the kernel make a connection to an existing socket: ***connect(sd, address, length);*** where the semantics of the parameters are the same as for bind, but *address* is the address of the target socket that will form the other end of the communications line.

Both sockets must use the same communications domain and protocol, and the kernel arranges that the communications links are set up correctly. If the type of the socket is a datagram, the connect call informs the kernel of the address to be used on subsequent send calls over the socket; no connections are made at the time of the call.

When a server process arranges to accept connections over a virtual circuit, the kernel must queue incoming requests until it can service them. The listen system call specifies the maximum queue length:

listen(sd, qlength)

where *sd* is the socket descriptor and *qlength* is the maximum number of outstanding requests.

The accept call receives incoming requests for a connection to a server process:

nsd = accept (sd, address, addrlen);

Where *sd* is the socket descriptor, *address* points to a user data array that the kernel fills with the return address of the connecting client, and *addrlen* indicates the size of the user array. When *accept* returns, the kernel overwrites the contents of *addrlen* with a number that indicates the amount of space taken up by the address.

Accept returns a new socket descriptor *nsd*, different from the socket descriptor *sd*. A server can continue listening to the advertised socket while communicating with a client process over a separate communications channel (Figure 8.19).

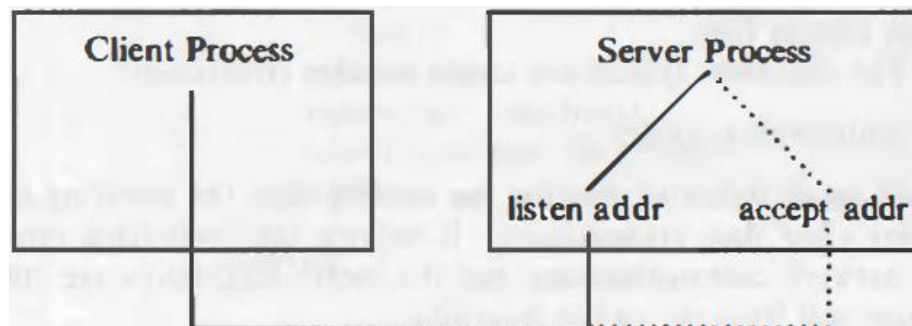


Figure 8.19: A Server Accepting a Call

The send and recv system calls transmit data over a connected socket:

count = send(sd, msg, length, flags);

Where *sd* is the socket descriptor, *msg* is a pointer to the data being sent, *length* is its length, and *count* is the number of bytes actually sent.

The flags parameter may be set to the value `SOF_OOB` to send data "out-of-band," meaning that data being sent is not considered part of the regular sequence of data exchange between the communicating processes.

A "remote login" program, for instance, may send an "out of band" message to simulate a user hitting the delete key at a terminal.

The syntax of the `recv` system calls is:

count = recv(sd, buf, length, flags) ;

Where `buf` is the data array for incoming data, `length` is the expected length, and `count` is the number of bytes copied to the user program. Flags can be set to "peek" at an incoming message and examine its contents without removing it from the queue, or to receive "out of band" data. The datagram versions of these system calls, ***sendto*** and ***recvfrom***, have additional parameters for addresses.

Processes can use read and write system calls on stream sockets instead of send and recv after the connection is set up. Thus, servers can take care of network-specific protocol negotiation and spawn processes that use read and write calls only, as if they are using regular files.

The shutdown system call closes a socket connection:

shutdown(sd, mode)

Where `mode` indicates whether the sending side, the receiving side, or both sides no longer allow data transmission; it informs the underlying protocols to close down the network communications, but the socket descriptors are still intact. The close system call frees the socket descriptor.

The `getsockname` system call gets the name of a socket bound by a previous bind call:

getsockname(sd, name, length);

The ***getsockopt*** and ***setsockopt*** calls retrieve and set various options associated with the socket, according to the communications domain and protocol of the socket.