Terminal Handling Principles

- Terminal access to mainframe and mini computers used hardwired terminals typically connected through a RS232 serial line.
- Today the normal command line interface to FreeBSD use a pseudo terminal.
- A pseudo terminal is built from a device pair termed the *master* and *slave* devices.
- The master side is named /dev/ptyXX and the slave side is named /dev/ttyXX.
- The slave device provides to a process an interface identical to that historically provided by a hardware device.
- Anything written on a master device appears as input on the slave device and anything written on a slave appears as input on the master device.
- Pseudo terminals are used by the terminal emulator, **xterm**, and by remote login programs such as **ssh**.
- **Xterm** opens the master side of the pseudo terminal and directs the keystrokes to its output and input from the pseudo terminal is directed to the window.
- Xterm forks a child process that opens the slave side of the pseudo terminal and execs a user shell.

Terminal Modes

The terminal processing can be set to two different modes, *canonical mode (cocked mode)* and *noncanonical mode (raw mode*).

Canonical mode properties:

- Characters are echoed by the operating system as they are typed but are buffered internally until a newline character ,NL, is typed.
- Only after a NL character is received the entire line is made available to the reading process.
- If the process tries to read before a NL character is received it is put to sleep until an NL character arrives.
- Typing errors can be corrected by entering special *erase* and *kill* characters.
- Simple output processing is performed - normally converting a NL character to NL + CR (carriage return).
- When a process has filled the terminal output queue it will be put to sleep.

Raw mode properties:

- The system makes each typed character available to the reading process as soon as it is received.
- No line editing or other processing is performed.

In reality many combinations of these modes can be specified.

Line Disciplines

- Most of the character processing done for terminal interfaces is independent of whether it is associated with a pseudo-terminal or a hardware device.
- This hardware independent processing is performed by the *line discipline*.
- The entry points of the *line discipline* is called through the *linesw* switch in the same way as the character device drivers are called through the *device* switch.
- The entry points of the line discipline are listed in table 10.1.
- Several of the routines in the serial terminal drivers (read, write, ioctl) directly transfer control to the *line discipline* when called.
- The *l_rint* (receiver interrupt) is called for each character received on a line.
- The corresponding entry for transmit-complete interrupts is *l_start*.
- The system includes several other types of line disciplines for example for the ppp protocol.

User interface

- The standard programming interface for control of the terminal line discipline is the *ioctl* system call.
- The *ioctl* system call is used to change line discipline, set and get values for special processing characters and modes and to set and get values for hardware serial line parameters.
- There have been several different specifications for the serial line ioctl commands in UNIX.
- The current interface in FreeBSD (and Linux) is based on a POSIX standard and uses the *termios* data structure.
- Some of the ioctl system calls related to the standard terminal line discipline is described at page 417 in the course book.
- Other line disciplines usually use other ioctl commands.

The tty Structure

- For every pseudo-terminal or hardware terminal the kernel allocates a *tty struct* to keep all data related to the terminal.
- The tty struct is shared by the terminal driver and the line discipline.
- The calls to the line discipline all requires a *tty struct* as a parameter.
- The *tty struct* is initialized by the terminal driver's open routine and by the line discipline open routine.
- The content of tty struct is illustrated by Table 10.2.

Process Groups and Terminal Control

- When a process creates a new session, that session has no associated terminal.
- To acquire a terminal, the *session leader* must make an TIOCSCTTY *ioctl* call.
- When the call succeeds the *session leader* becomes a *controlling process*.
- The *tty struct* will contain a pointer to the session struct and a pointer to the process group of the session leader.
- This process group pointer identifies the process group in control of the terminal - the *foreground* process group.
- Other process groups may run in the *background*.
- The *foreground* process group may be changed by making a TIOCSPGRP *ioctl* call.
- When a session leader exits the controlling terminal is revoked with the *revoke system call* and that invalidates any open descriptors in the system for that terminal.

C-lists

The terminal I/O system deals with blocks of widely varying sizes.

FreeBSD still use the data structures originally designed for terminal drivers - the *C-block* and *C-list*.

C-block A fixed size (128 bytes) buffer with space for buffered characters and a linkage pointer.

C-list Describes a queue of input or output characters. Contains pointers to the first and last characters in the C-blocks that build up the queue and a character count (fig. 10.1).

A set of utility routines are defined to manipulate C-lists:

getc() Removes the next character from a C-list.

putc() Adds a character to the end of a C-list.

b_to_q() Add several characters to a C-list.

q_to_b() Read several characters from a C-list.

unputc() Remove the last character from a C-list.

nextc() Examine the next character on a C-list.

catq() Concatenate two C-lists.

Terminal I/O Implementation

Open

- Each time the master side of a pseudo-terminal is opened, the pseudo-terminal driver's open routine is called.
- The driver routine initiates the tty struct and calls the line discipline routine *l_open*.
- The setup of the terminal completes when the slave side of the pseudo-terminal is opened.

Terminal I/O Implementation

Write

- After a terminal have been opened, writes to the resulting file descriptor results in a call to the corresponding character device driver routine *d_write*.
- d_write will call the line discipline write routine, *l_write*, with a tty struct and an uio struct as parameters.
- The *l_write* routine for the default terminal line discipline is actually named *ttwrite()*.
- The ttwrite() routine performs most of the work involved in outputting characters to a terminal.
- The helper subroutine *ttyoutput()* is called by *ttwrite()* to do output processing of special characters.

ttwrite() Implementation

The ttwrite() routine loops until all data is processed performing the following:

- Check that the terminal is allowed to do output. If the terminal is controlling terminal for a process group, output is normally allowed only if the process group is in the foreground.
- When write is allowed, copy characters from the calling process into the kernel. Check if output processing is needed and move data to the output queue.
- As soon as data are placed on the output queue of the tty, *ttstart()* is called to start output.
- ttstart() will call the start routine specified in the t_oproc field in *tty struct.*
  - → For a pseudo tty, the start routine will wake the process sleeping on the master side.
- The size of the output queue is limited by the *high watermark*.
- If *high watermark* is exceeded the process is put to sleep (Fig. 10.2).
  - → When sleeping the TS_SO_OLOWAT flag is set in the *tty struct* to request that the process is awakened when the queue size falls below the *low watermark* that is half of the *high watermark*.

ttwrite() Implementation cont.

The output processing done by ttwrite() is dependent on the terminal mode:

Canonical mode:

- Groups of characters that do not need special processing is located by scanning through the output string and at the same time marking characters that might need translation.
- Each group of characters that do not need translation is moved to the output queue using *b_to_q()*.
- The characters that may need translation is processed by a call to *ttyoutput().*
- The *ttyoutput()* routine may perform the following translations:
  - → Newline characters may be replaced by newline plus carriage return.
  - → Tabs may be expanded to spaces.

Noncanonical mode:

- The entire buffer is copied to the output queue without processing.

Terminal Input

- Terminal input arrives asynchronously when the terminal line receives characters from the local keyboard or from a communications line in the case of remote login.
- Thus most of the input processing is done at interrupt time.
- When a character arrives over a network, the locally running remote-login daemon writes it into the master side of the pseudo-terminal.
- The pseudo-tty driver will pass the characters to the line discipline receiver interrupt routine (*l_rint*) at the slave side.
- The receiver interrupt routine for the normal terminal line discipline is *ttyinput()*.

ttyinput() implementation

- Check that the input queue is not too large.
  - → For pseudo terminals, when input queue becomes full the kernel stops reading characters from the master side.
- Echo characters if desired.

In canonical mode do:

- If normal character, put it on the raw input queue.
- If character with special meaning, take requested actions.
  - → If erase character, modify raw input queue accordingly.
  - → If newline character, concatenate raw queue to canonical queue and call *ttwakeup()* to wake up the process waiting for data.

In noncanonical mode do:

- Put the character in the raw input queue without processing and call *ttwakeup()*.

Terminal input

- When a read system call is made on a file descriptor for a terminal device, the device driver's *d_read* routine is called.
- The *d_read* routine will call the line disciplines *l_read* routine, which is called *ttread()* for the normal terminal line discipline.
- ttread() checks that the process belongs to the session and is in the foreground process group.
  - → If the process is in the background it is sent a SIGTTIN signal.
- Finally ttread() checks for data in the canonical queue if in canonical mode and in the raw queue if in noncanonical mode.
  - → If no data is present ttread() returns the error EAGAIN if the terminal uses nonblocking I/O otherwise it sleeps on the address of the raw queue.
- When characters are present, they are removed from the queue one at a time with the *getc()* command and copied out to the user's buffer with *ureadc()*.