Terminal I/O (Chapter 18)

- POSIX termios
- Two terminal I/O modes
- **termios** structure
- Getting and setting terminal attributes
- Terminal window size: struct winsize

Overview

- What is a tty?
  - An abbreviation for “terminal”, created decades ago for serial TeleTYpewriter. A terminal is typically interactive and can be run locally over a network, or over a serial line
  - A tty device has two ends
    - In case of real terminals, one end is connected to hardware, such as monitor, serial ports, and the other end is connected to a program, such as a shell,
    - In case of pseudo terminal, (ptys), both ends are connected to software, with one end simulates the hardware, telnet, ssh,
- The tty interface:
  - BSD sgtty
  - System V termio
  - POSIX **termios**: supersedes both sgtty and termio
- Complexity of terminal system: Terminal i/o for many things
  - Real terminals
  - Hardwired lines between computers, modems, printers, etc.

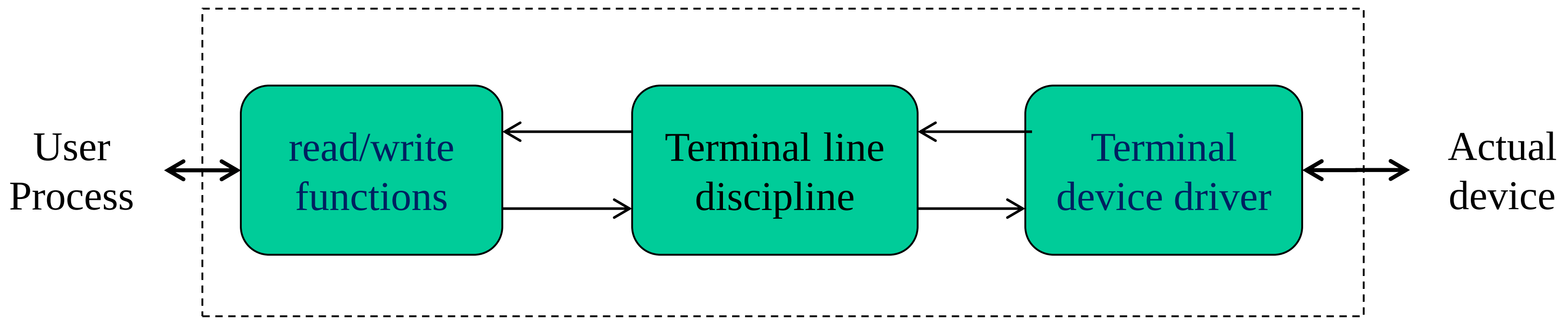
What is a Terminal?

- A terminal consists of a screen and keyboard that one uses to communicate remotely with a host computer. The program executes on the host computer, and the results display on the screen
- Text terminal
  - Text terminals are also called dumb terminals (think terminal)
  - For a text terminal, a 2-way flow of information between the computer and the terminal takes place over the cable that connects them together.
  - This flow is in bytes (such as ASCII) where each byte usually represents a printable character. Bytes typed at the keyboard go to the computer and most bytes from the computer are displayed on the terminal screen
  - Mid 1970's and mid 1980's, we used text terminals to communicate with the super computers. The cable connected the terminal to the computer. It was called terminal because it was located at the terminal end of the cable.
- Monitor+keyboard is not a terminal
  - A text terminal is often connected to a serial port via long cable, no mice, has built in not so good graphic card
  - A monitor is often right next to the computer,

Two Modes for Terminal I/O Input Processing

- Canonical mode (cooked mode)
  - the default terminal I/O mode
  - Provides a limited line editor inside the device driver and sends edited input to the application one line at a time
  - Most UNIX systems implement all the canonical processing in a module called the **Terminal Line Discipline**
  - i.e: Standard input and output
- Non-canonical mode (raw mode)
  - The input characters are not assembled into lines, the data is passed to the application as it is received with no changes made
  - i.e: vi editor, such special characters will not be processed by the terminal driver

Terminal I/O Input Processing



Terminal Operations

- All terminal device characteristics are contained in struct termios defined in <termios.h>

```
struct termios {
   tcflag_t c_iflag;   // input mode flags
   tcflag_t c_oflag;   // output mode flags
   tcflag_t c_cflag;   // control mode flags (device)
   tcflag_t c_lfalg;   // local mode flags
   cc_t c_cc[NCCS];    // control characters
}
```

- **`Isatty:`** to see if a file descriptor is a tty

  `int isatty(int fd);` Return 1 if true, 0 on false

- **`ttyname:`** get the name of terminal associated w/ the **`fd`**

  `char* ttyname(int fd);`

  - Return NULL on error or the file descriptor is not associated with a `tty`

- **`ctermid:`** get controlling terminal name
  - `char* name = ctermid(NULL); or`
  - `char s[L_ctermid]; ctermid(s);`

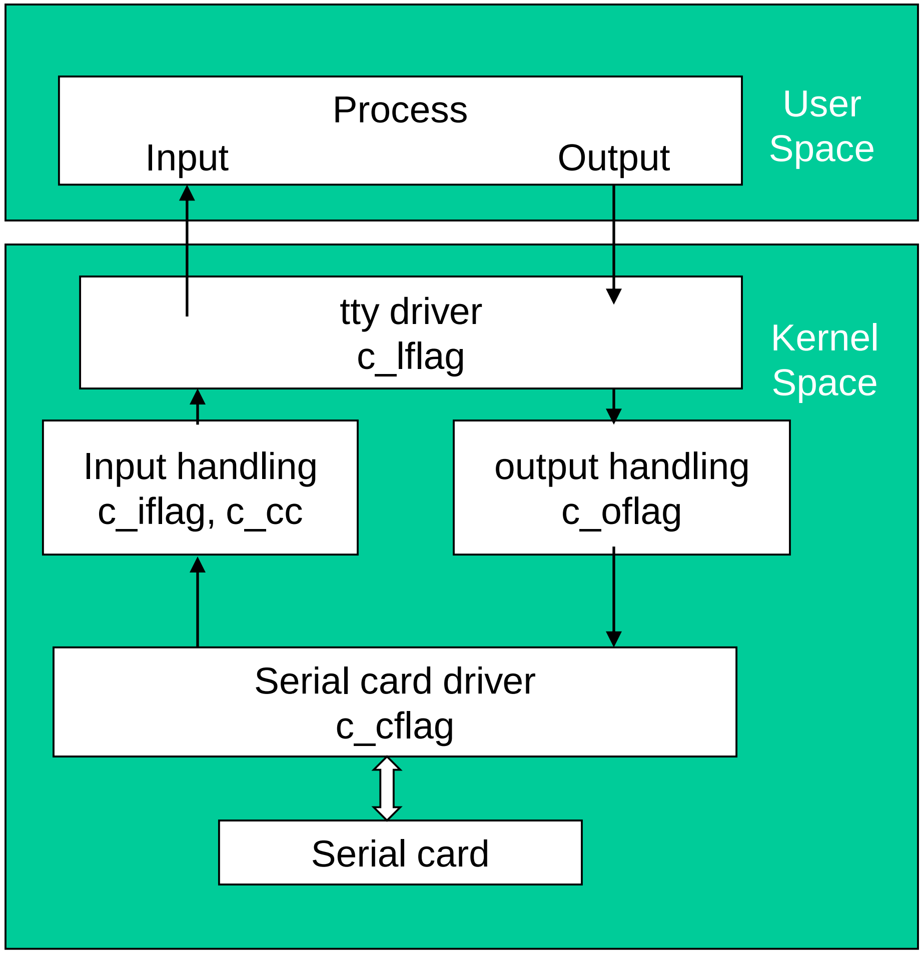


Figure 18.3~18.6 on page 635 lists different flags

- Input flags: **c_iflag**

  Control the input of characters by the terminal device driver

  - strip 8th bit (parity bit) parity checking (the 8th bit is for parity)

- Output flags: **c_oflag**

  Control the driver output , such as map newline (\n) to CR/LF, etc.

- Control flags: **c_cflag**

  Affects RS-232 serial line, stop bits, etc.

- Local flags: **c_lfag**

  Affect the interface b/w the driver and the user, such as

  - echo on or off,
  - enable terminal-generated signals,
  - job control

- **c_cc** array

  - Array contains all the special characters that can be changed, typically 15~20
  - c_cc is large enough to hold each special character and is typically an unsigned char

- Figure 18.7 on p677: List of functions for these flags, here are a few:

`tcgetattr`:	fetch attributes
`tcsetattr`:	set attributes
`tcdrain`:	wait for all output to be transmitted
`tcflush`:	flush pending input/output

- Figure 18.9 on p678 lists the special input characters defined by POSIX.1
  - Among the special input characters, CR (\r) and NL(\n) can not be changed.
  - All others can be changed to whatever we like, or to be disabled
- Getting and setting terminal attributes

```
#include <termios.h>
int tcgetattr(int fd, struct termios *termptr);
int tcsetattr(int fd, int opt, const struct termios *termptr);
```

Both return 0 if OK, -1 on error

  - `opt`: specify when the new terminal attributes to take effect
    - TCSANOW: immediately
    - TCSADRAIN: changed after all the output has been transmitted
    - TCSAFLUSH: changed after all the output has been transmitted, and all the unread input is discarded

Special Characters

Character	c_cc subscript	Typical value	Description
EOL	VEOL		End of Line
EOF	VEOF	^D	End of file
NL	(can not changed)	\n	New Line
INTR	VINTR	^C	Interrupt signal
ERASE	VERASE	^H	Backspace on erase
QUIT	VQUIT	^\	Quit signal
WERASE	VWERASE	^W	Erase a word
KILL	VKILL	^U	Erase line

- We can disable some of these by setting the value of c_cc array to the value of _PC_VDISABLE. We can get this value by fpathconf

```
long vdisable = fpathconf(STDIN_FILENO,
_PC_VDISABLE)

cc[VQUIT]=vdisable
```

- Disable ctrl-c is different from ignoral INTR (in signal interrupt)

Modify Terminal Attributes

```
#include "cs590.h"  // textbook, figure18.10
#include <termios.h>
void err_quit(char* msg){ printf("%s\n", msg); return 1;}
int main(void){
    struct termios    term;
    long              vdisable;
    if (isatty(0) == 0)
        err_quit("stdin is not a terminal device");
    if ((vdisable = fpathconf(0, _PC_VDISABLE)) < 0)
        err_quit("_POSIX_VDISABLE not in effect");
    if (tcgetattr(0, &term) < 0)   /* fetch tty state */
        perror("tcgetattr error");
    term.c_cc[VINTR] = vdisable; //disable INTR character
    term.c_cc[VEOF] = 2;      // EOF is ^B Check ASCII table
    if (tcsetattr(0, TCSAFLUSH, &term) < 0)
        perror("tcsetattr error");
    exit(0);
} //  now ctrl-c will not be able to generate SIGINT
```

Terminal Option Flags (§ 18.5)

- **`c_cflag:`**
  - CSIZE: a mask that specifies the number of bits per byte for both transmission and reception. This size does not include the parity bit, it any. The values can be CS5, CS6, CS7, CS8
    - See example Fig 18.11 on page 684
  - PARENB: If set, parity generation is enabled for output char, and parity checking is performed on incoming char
  - PARODD: If set, odd parity; otherwise, even parity
- **`c_lflag:`**
  - ECHO: If set, input characters are echoed back to the terminal device
  - ECHONL: If set and ICANON is set, the NL (new line) character is echoed regardless if ECHO is set or not
  - ICANON: If set, canonical mode is in effect
- **`c_iflag`**
  - ISTRIP: If set, valid input bytes are stripped to 7 bits, otherwise, all 8 bits are processed
- ETC…

```
#include “cs590.h“ // make sure <termios.h> is there
int main(void){
    struct termios  term;
    if (tcgetattr(STDIN_FILENO, &term) < 0) perror("tcgetattr error");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte\n");     break;
    case CS6:
        printf("6 bits/byte\n");     break;
    case CS7:
        printf("7 bits/byte\n");     break;
    case CS8:
        printf("8 bits/byte\n");     break;
    default:
        printf("unknown bits/byte\n");
    }
    term.c_cflag &= ~CSIZE;     /* zero out the bits */
    term.c_cflag |= CS8;        /* set 8 bits/byte */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        perror("tcsetattr error");
    exit(0);
}
```

```
/* readpass.c - Reads a password without displaying it on the
   terminal
*/

#include “cs590.h”
int main(void) {
    struct termios ts, ots;
    char passbuf[1024];
    /* get and save current termios settings */
    tcgetattr(0, &ts);
    ots = ts;

    /* change and set new termios settings */
    ts.c_lflag &= ~ECHO;
    ts.c_lflag |= ECHONL;
    tcsetattr(0, TCSAFLUSH, &ts);
    /* check that the settings took effect */
      tcgetattr(0, &ts);
      if (ts.c_lflag & ECHO) {
           fprintf(stderr, "Failed to turn off echo\n");
           tcsetattr(0, TCSANOW, &ots);
           exit(1);
      }
    /* get and print the password */
      printf("enter password: "); fflush(stdout);
      fgets(passbuf, 1024, stdin);
      printf("read password: %s", passbuf);
      /* there was a terminating \n in passbuf */
      /* restore old termios settings */
      tcsetattr(0, TCSANOW, &ots);
      exit(0);
}
```

Baud rate function

- Baud rate is bits per second: B2400, B9600, B19200, B38400, etc.
- Get I/O Baud rate

```
speed_t cfgetipeed(const struct termios *termptr);
speed_t cfgetospeed(const struct termios *termptr);
```

  - Return baud rate value

- Set I/O Baud rate

```
int cfsetispeed(struct termios *termptr, speed_t spd);
int cfsetospeed(struct termios *termptr,speed_t spd);
```

  - Return 0 if OK, -1 on error

Line control functions: see p693

- `tcdrain (int fd); //`wait for all output to be transmitted
- `tcflow(int fd, int action);` // control the flow(i/o) with action
- `tcflush(int fd, int queue);`
- `tcsendbreak(int fd, int duration); //`transmit a series of zero bits for a specified duration

Figure 18.7. Summary of terminal I/O functions

Function	Description
`tcgetattr`	fetch attributes (`termios` structure)
`tcsetattr`	set attributes (`termios` structure)
`cfgetispeed`	get input speed
`cfgetospeed`	get output speed
`cfsetispeed`	set input speed
`cfsetospeed`	set output speed
`tcdrain`	wait for all output to be transmitted
`tcflow`	suspend transmit or receive
`tcflush`	flush pending input and/or output
`tcsendbreak`	send BREAK character
`tcgetpgrp`	get foreground process group ID
`tcsetpgrp`	set foreground process group ID
`tcgetsid`	get process group ID of session leader for controlling TTY (XSI extension)

Canonical mode

- Provides a limited line editor inside the device driver and sends edited input to the application one line at a time (assemble the bytes received into lines) (such as erase, etc.)
- Input is made available line by line (NL, EOL,etc.)

  We issue a read, the terminal driver returns when a line has been entered. Several conditions cause the read to return

  - When the requested number of bytes has been read
  - When the line delimiter is encountered (NL, EOF, EOL, EOL2)
  - When maximum line length is reached (4096 characters)
  - When a signal is caught, but read is not automatically restarted

Non-Canonical Mode

- Specified by turning off the ICANON flag in local flag, `c_lflag` of termios
- Input data in non-canonical mode is not assembled into lines
  - Input is available immediately, no input processing is performed
  - Some characters, ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE are not processed
- Two variable values (MIN, TIME) in the `c_cc` array are used to determine how to process the bytes received, what input to return to caller
  - **`MIN:`** specifies the minimum number of bytes before a read returns.
  - **`TIME`** specifies the number of tenths of a second to wait for data to arrive

Our terminal: `min=1, time=0`

Four Cases for “read”

	MIN>0	MIN=0
TIME>0 (10th of second)	A Read returns MIN bytes before timer expires Read returns[1,MIN) if time expires (TIME=interbyte timer)	C Read return [1, nbytes] before timer expires Read returns 0 if timer expires TIME=read timer
TIME=0	B (blocking read) read returns [MIN, nbytes] when available	D (polling read) Read returns [0, nbytes] immediately

Terminal Window Size

- Most UNIX systems provide a way to keep track of the current terminal window size and to have the kernel notify the foreground process group when the size changes

```
 struct winsize{
    unsigned short ws_row;
    unsigned short ws_col;
    unsigned short ws_xpixel  //not in use for Linux;
    unsigned short ws_ypixel  //not in use for Linux
 };
```

- We can fetch the current value of the structure using an `ioctl` with flag TIOC**G**WINSZ
- We can store a new value of structure in kernel using `ioctl` with TIOC**S**WINSZ, if the new value differs from the current value stored in the kernel, a SIGWINCH signal is sent to the foreground process group (by default, the signal is ignored)
- Interpretation of the values in the structure is the responsibility of the caller

```c
#include "cs590.h"

void pr_winsize(int fd){
    struct winsize    size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        perror("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void sig_winsz(int signo){
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
}

int main(void){
    if (isatty(STDIN_FILENO) == 0)  exit(1);
    if (signal(SIGWINCH, sig_winsz) == SIG_ERR)
        perror("signal error");
    pr_winsize(STDIN_FILENO);   /* print initial size */
    for ( ; ; )                 /* and sleep forever */
        pause();
}
```