Chapter 18
Terminal I/O

Cheng-Hsin Hsu
National Tsing Hua University
Department of Computer Science

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

Outline

- Introduction and overview
- Special input characters
- Getting and setting terminal attributes
- Terminal option flags
- stty command
- Line control functions
- Terminal identification
- Terminal modes: Canonical, non-canonical mode, cbreak, and raw
- Terminal window size
- termcap, terminfo, and curses

Introduction

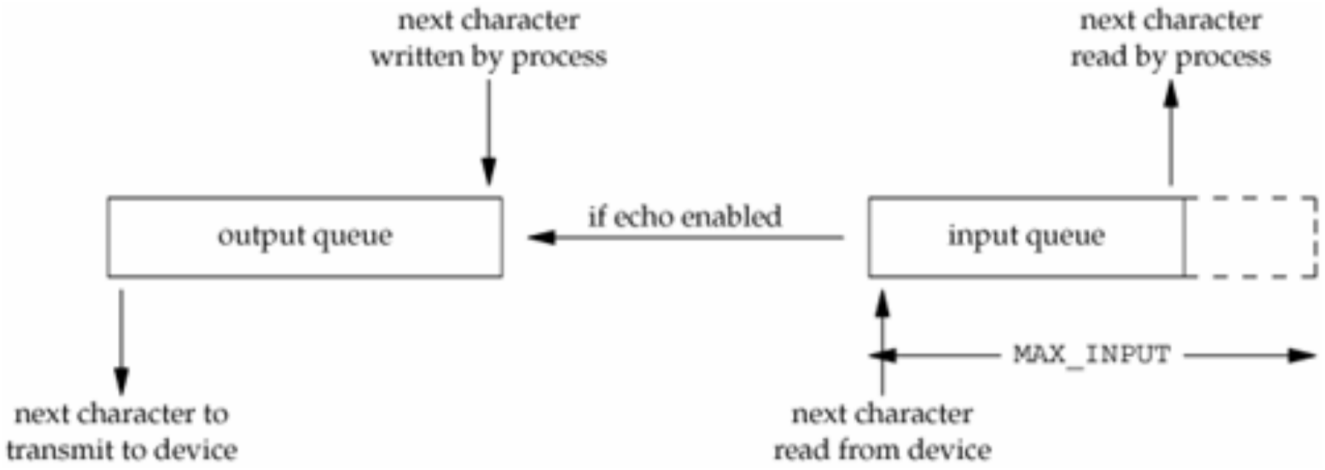
- The handling of terminal I/O is a messy area
- The manual page for terminal I/O is usually one of the longest in the programmer's manuals
- We look at all the POSIX.1 terminal functions and some of the platform-specific additions in this chapter
- Terminal I/O has two modes
- Canonical mode input processing
  - Terminal input is processed as lines
  - For example, read functions return a single line
- Non-canonical mode input processing
  - Input characters are not assembled into lines
  - For example, full screen editors like vi works in this mode

Introduction (Cont'd)

- Older BSD-style terminal drivers supported three modes for terminal input
- (a) cooked mode
  - Input is collected into lines, and the special characters are processed
- (b) raw mode
  - Input is not assembled into lines, and there is no processing of special characters
- (c) cbreak mode
  - Input is not assembled into lines, but some of the special characters are processed
- POSIX.1 defines 11 special input characters, e.g., Ctrl-D and Ctrl-Z
  - 9 of which we can change

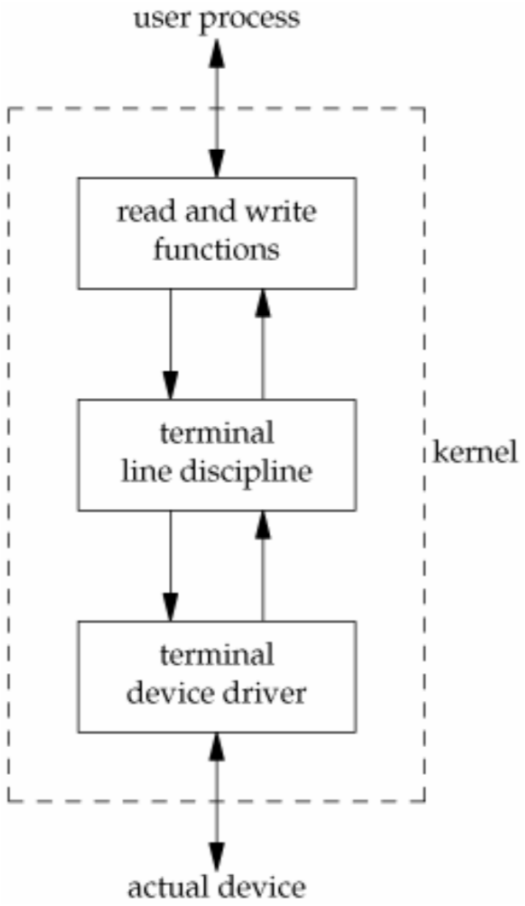
Terminal Device and Terminal Driver

- If echoing is enabled, there is an implied link between the input queue and the output queue
- The size of the input queue is limited (MAX_INPUT)
- When the input queue is full, many UNIX systems echo the bell character



Terminal Line Discipline

- Most UNIX systems implement all the canonical processing in a module called the terminal line discipline
- This module is a box that sits between the kernel's generic read and write functions and the actual device driver



The termios Structure

- All the terminal device characteristics that we can examine and change are contained in a termios structure (in <termios.h>)
  - Input modes: Handle input chars, e.g., strip 8th bit, parity check, …
  - Output modes: Handle output chars, e.g., map newline to CR/LF
  - Control modes: Control RS-232 serial lines
  - Local modes: Interfaces between the driver and the user, e.g., echo off, visually erase charaters, terminal-generated signals, …
  - cc_t array: Hold each special character

```
struct termios {
    tcflag_t c_iflag;      /* input modes */
    tcflag_t c_oflag;      /* output modes */
    tcflag_t c_cflag;      /* control modes */
    tcflag_t c_lflag;      /* local modes */
    cc_t     c_cc[NCCS];   /* special characters */
}
```

The termios Structure (Cont'd)

- Versions of System V that predated the POSIX standard had a header named <termio.h> and a structure named termio
- POSIX.1 added an s to the names, i.e., termios.h and termios, to differentiate them from their predecessors
- There a lot of flags can be used to affect the characteristic of a terminal device
- Too much items!
  - We list them in the slide, but the text size may be too small
  - Please see Figure 18.3—18.6 for complete lists

termios: c_cflag Terminal Flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
CBAUDEXT	Extended baud rate					•
CCAR_OFLPOW	DCD flow control of output		•		•	
CCTS_OFLOW	CTS flow control of output		•		•	•
CDSR_OFLOW	DSR flow control of output		•		•	
CDTR_IFLOW	DTR flow control of input		•		•	
CIBAUDEXT	Extended input baud rate					•
CIGNORE	Ignore control flags		•		•	
CLOCAL	Ignore modem status lines	•	•	•	•	•
CMSPAR	Mark or space parity			•		
CREAD	Enable receiver	•	•	•	•	•
CRTSCTS	Enable hardware flow control		•	•	•	•
CRTS_IFLOW	RTS flow control of input		•		•	•
CRTSXOFF	Enable input hardware flow control					•
CSIZE	Character size mask	•	•	•	•	•
CSTOPB	Send two stop bits, else one	•	•	•	•	•
HUPCL	Hang up on last close	•	•	•	•	•
MDMBUF	Same as CCAR_OFLOW		•		•	
PARENB	Parity enab;e	•	•	•	•	•
PAREXT	Mark of space parity					•
PARODD	Odd parity, else even	•	•	•	•	•

termios: i_cflag Terminal Flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
BRKINT	Generate SIGINT on BREAK	•	•	•	•	•
ICRNL	Map CR to NL on input	•	•	•	•	•
IGNBRK	Ignore BREAK condition	•	•	•	•	•
IGNCR	Ignore CR	•	•	•	•	•
IGNPAR	Ignore characters with parity errors	•	•	•	•	•
IMAXBEL	Ring bell on input queue full		•	•	•	•
INLCR	Map NL to CR on input	•	•	•	•	•
INPCK	Enable input parity checking	•	•	•	•	•
ISTRIP	Strip eighth bit off input characters	•	•	•	•	•
IUCLC	Map uppercase to lowercase on input			•		•
IUTF8	Input is UTF-8			•	•	
IXANY	Enable any characters to restart output	•	•	•	•	•
IXOFF	Enable start/stop input flow control	•	•	•	•	•
IXON	Enable start/stop output flow control	•	•	•	•	•
PARMRK	Mark parity errors	•	•	•	•	•

termios: c_lflag Terminal Flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
ALTWERASE	Use alternate WERASE algorithm		•		•	•
ECHO	Enable echo	•	•	•	•	•
ECHOCTL	Enable control char as ^(Char)		•	•	•	•
ECHOE	Visually erase chars	•	•	•	•	•
ECHOK	Echo kill	•	•	•	•	•
ECHOKE	Visual erase for kill		•	•	•	•
ECHONL	Echo NL	•	•	•	•	•
ECHOPRT	Visual erase mode for hard copy		•	•	•	•
EXTPROC	External character processing		•	•	•	
FLUSHO	Output being flushed		•	•	•	•
ICANON	Canonical input	•	•	•	•	•
IEXTEN	Enable extended input char processing	•	•	•	•	•
ISIG	Enable terminal-generated signals	•	•	•	•	•
NOFLSH	Disable flush after interrupt or quit	•	•	•	•	•
NOKERNINFO	No kernel output from STATUS		•		•	
PENDIN	Retype pending input		•	•	•	•
TOSTOP	Send SIGTTOU for background output	•	•	•	•	•
XCASE	Canonical upper/lower presentation			•		•

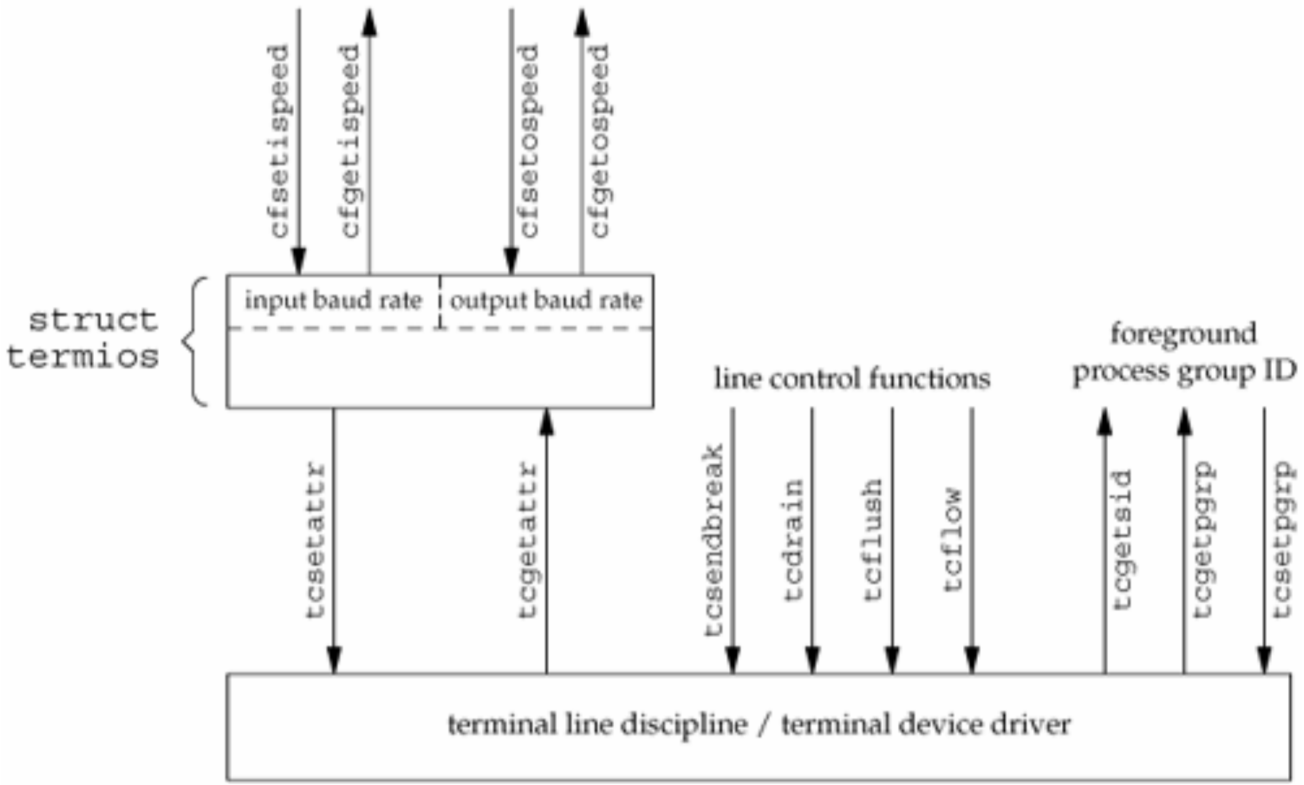
termios: c_oflag Terminal Flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
BSDLY	Backspace delay mask	XSI		•		•
CRDLY	CR delay mask	XSI		•		•
FFDLY	Form feed delay mask	XSI		•		•
NLDLY	NL delay mask	XSI		•		•
OCRNL	Map CR to NL on output	XSI	•	•		•
OFDEL	Fill is DEL, else NUL	XSI		•		•
OFILL	Use fill character for delay	XSI		•		•
OLCUT	Map lowercase to uppercase on output			•		•
ONLCR	Map NL to CR-NL	XSI	•	•	•	•
ONLRET	NL performs CR function	XSI	•	•		•
ONOCR	No CR output at column 0	XSI	•	•		•
ONOEOT	Discard EOTs (^D) on output		•		•	
OPOST	Perform output processing	•	•	•	•	•
OXTABS	Expand tabs to spaces		•		•	
TABDLY	Horizontal tab delay mask	XSI	•	•		•
VTDLY	Vertical tab delay mask	XSI		•		•

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

Relationships among the Terminal-related Functions



Special Input Characters

- POSIX.1 defines 11 characters that are handled specially on input
- See the table in the next page
- We can change 9 of them to almost any value that we like
  - Except for newline (\n) and carriage return (\r) characters, and perhaps STOP and START characters
  - We can modify the appropriate entry in the `c_cc` array of the termios structure
- POSIX.1 allows us to disable these characters
  - Set the value of an entry in the c_cc array to _POSIX_VDISABLE

Special Input Characters (Cont'd)

Character	Description	c_cc subscript	Enabled by		Typical value	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
			field	flag						
CR	Carriage return	--	c_lflag	ICANON	\r	•	•	•	•	•
DISCARD	Discard output	VDISCARD	c_lflag	IEXTEN	^O		•	•	•	•
DSUP	Delayed suspend (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•		•	•
EOF	End of file	VEOF	c_lflag	ICANON	^D	•	•	•	•	•
EOL	End of line	VEOL	c_lflag	ICANON		•	•	•	•	•
EOL2	Alternate end of line	VEOL2	c_lflag	ICANON			•	•	•	•
ERASE	Backspace one character	VERASE	c_lflag	ICANON	^H, ^?	•	•	•	•	•
ERASE2	Alternate backspace character	VERASE2	c_lflag	ICANON	^H, ^?		•			
INTR	Interrupt signal (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•	•	•	•	•
KILL	Erase line	VKILL	c_lflag	ICANON	^U	•	•	•	•	•
LNEXT	Literal next	VLNEXT	c_lflag	IEXTEN	^V		•	•	•	•
NL	Line feed (newline)	--	c_lflag	ICANON	\n	•	•	•	•	•
QUIT	Quit signal (SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•	•	•	•	•
REPRINT	Reprint all input	VREPRINT	c_lflag	ICANON	^R		•	•	•	•
START	Resume output	VSTART	c_lflag	IXON/IXOFF	^Q	•	•	•	•	•
STATUS	Status request	VSTATUS	c_lflag	ICANON	^T		•		•	
STOP	Stop output	VSTOP	c_lflag	IXON/IXOFF	^S	•	•	•	•	•
SUSP	Suspend signal (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•	•	•	•	•
WERASE	Backspace one word	VWERASE	c_lflag	ICANON	^W		•	•	•	•

Special Input Characters: Example – Disable Interrupt and Set EOF to ^B

```
int main(void) {
    struct termios  term;
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Relevant Functions

- Determine if a descriptor is a terminal device
  - `int isatty(int fd);`
  - Returns: 1 if fd refers to a terminal, 0 if not (errno is also set)
- Get configuration values for a file (e.g., get _POSIX_VDISABLE)
  - `long fpathconf(int fd, int name);`
  - Returns: The requested value, or -1 on error.

Relevant Functions (Cont'd)

- Set and get terminal attributes
  - `int tcgetattr(int fd, struct termios *termptr);`
  - `int tcsetattr(int fd, int opt, struct termios *termptr);`
  - Returns: zero if OK, or -1 on error
- opt can be
  - TCSANOW: The change occurs immediately
  - TCSADRAIN: The change occurs after all output has been transmitted
  - TCSAFLUSH: Similar to TCSADRAIN, and when the change takes place, all input data that has not been read is discarded (flushed).
- Notice!
  - tcsetattr returns OK if any one of the requested actions is performed, not all
  - You have to call tcgetattr again to confirm if all the requested actions were performed

The stty Command

- All the options described in the previous section can be examined and changed using the stty command
- Show the configuration for the current terminal
  - Option names preceded by a hyphen are disabled
- stty uses its standard input to get and set the terminal option flags
  - We may read configuration of other ttys
  - $ sudo stty –a < /dev/pts/7 (require root permissions)

```
$ stty -a
speed 9600 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc ixany imaxbel -
iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe -echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

Line Control Functions

- Wait for all output to be transmitted
  - `int tcdrain(int filedes);`
- Perform input and output flow control
  - `int tcflow(int filedes, int action);`
  - "action" can be one of the follows
  - `TCOOFF`: Output is suspended
  - `TCOON`: Output that was previously suspended is restarted
  - `TCIOFF`: The system transmits a STOP character, which should cause the terminal device to stop sending data
  - `TCION`: The system transmits a START character, which should cause the terminal device to resume sending data

Line Control Functions (Cont'd)

- Flush (throw away) input or output buffer
  - `int tcflush(int filedes, int queue);`
  - For input: discard received data that has not been read by the user
  - For output: discard queued data that has not been transmitted
  - "queue" can be one of the following
  - `TCIFLUSH`: The input queue is flushed
  - `TCOFLUSH`: The output queue is flushed
  - `TCIOFLUSH`: Both the input queue and the output queue are flushed
- Send a continuous stream of zero bits
  - `int tcsendbreak(int filedes, int duration);`
  - If duration == 0: transmit for 0.25 ~ 0.5 seconds
  - If duration != 0: It's implementation dependent

Terminal Identification

- Retrieve the name of the controlling terminal
  - `char *ctermid(char *ptr);`
  - The input size must be at least L_ctermid
  - Usually returns "/dev/tty"
- Determine if a descriptor is a terminal
  - `int isatty(int filedes);`
- Determine the terminal name of a descriptor
  - `char *ttyname(int filedes);`
  - How ttyname is implemented?
  - Retrieve the major/minor number of filedes
  - Search for the name major/minor number in /dev/
  - See the example: termio/ttyname.c

Terminal Identification: Example

- Print out tty name if a descriptor is a tty
- Print out a message “not a tty” if a descriptor is not a tty

```
#define PRINT_TTY(fd)   \
        printf("fd %d: %s\n", fd, \
                isatty(fd) ? ttyname(fd) : "not a tty");

int main() {
        char buf[L_ctermid];
        printf("termid %s\n", ctermid(buf));
        PRINT_TTY(0);
        PRINT_TTY(1);
        PRINT_TTY(2);
        return 0;
}
```

Terminal Identification: Example (Cont'd)

```
$ ./termid
termid /dev/tty
fd 0: /dev/pts/1
fd 1: /dev/pts/1
fd 2: /dev/pts/1


$ sudo bash -c './termid < /dev/console 2> /dev/null'
[sudo] password for username: ******************
termid /dev/tty
fd 0: /dev/console
fd 1: /dev/pts/1
fd 2: not a tty
```

Simple Implementation of ttyname

- Determine if the descriptor is a tty?
- Retrieve the major and minor number
- Recursively find files in /dev that matches the major/minor number

```
char *ttyname_x(int fd) {
        struct stat st;
        list<string> dirlist;    // STL list and string
        if(isatty(fd) == 0)
        return NULL;
        if(fstat(0, &st) < 0)
                return NULL;
        if(S_ISCHR(st.st_mode) == 0)
                return NULL;
        dirlist.push_back("/dev");
        return search_dir(dirlist,
        major(st.st_rdev), minor(st.st_rdev));
}
```

Simple Implementation of ttyname (Cont'd)

```
char
*search_dir(list<string>& dirlist, unsigned major_n, unsigned minor_n) {
  DIR *dir;
  struct dirent *d;
  struct stat st;
  char fullname[8192];
  //
  while(dirlist.size() > 0) {
    string name = dirlist.front();
    dirlist.pop_front();
    if((dir = opendir(name.c_str())) == NULL)
      continue;
    while((d = readdir(dir)) != NULL) {
      if(strcmp(d->d_name, ".") == 0)
        continue;
      if(strcmp(d->d_name, "..") == 0)
        continue;
      snprintf(fullname, sizeof(fullname),
        "%s/%s", name.c_str(), d->d_name);
      if(strcmp(fullname, "/dev/fd") == 0)
        continue;
      ...
      if(lstat(fullname, &st) != 0)
        continue;
      if(S_ISDIR(st.st_mode)) {
        dirlist.push_back(fullname);
        continue;
      }
      if(S_ISCHR(st.st_mode) == 0)
        continue;
      if(major(st.st_rdev) == major_n
      && minor(st.st_rdev) == minor_n) {
        closedir(dir);
        return strdup(fullname);
      }
    }  // readdir(dir) != NULL
    closedir(dir);
  }    // dirlist.size() > 0
  return NULL;
}
```

Terminal Modes

- Canonical mode
- Non-canonical mode
- cbreak mode
- Raw mode

Canonical Mode

- The read function returns when the terminal driver receives a line
- Conditions that cause the read to return:
- When the requested number of bytes have been read
  - The provided buffer size is less than the length of a line
  - The next read starts where the previous read stopped
- When a line delimiter is encountered
  - NL, EOL, EOL2, and EOF
  - CR, if ICRNL is set and IGNCR is not set
  - The delimiter will be the last character in the read buffer, except EOF
- When a signal is received and read is not automatically restarted

Non-canonical Mode

- Turn off the ICANON flag in the `c_lflag` field of the `termios` structure
- Input data is not assembled into lines
- The following special characters are not processed
  - ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE
- Timing to return from a read function – determined by MIN and TIME
  - MIN: The minimum bytes that has been received
  - TIME: The number of tenths of a second to wait for data to arrive
- Case A: MIN > 0 and TIME > 0
  - If MIN bytes are received, read returns
  - The timer only starts when at least one byte is received
  - If at least one byte is received, read returns when the timer expires

Non-canonical Mode (Cont'd)

- Case B: MIN > 0 and TIME == 0
  - The read does not return until MIN bytes have been received
  - read may be blocked infinitely
- Case C: MIN == 0 and TIME > 0
  - The timer starts when read is called
  - The read returns when a single byte is received or when the timer expires
  - If the timer expires, read returns 0
- Case D: MIN == 0 and TIME == 0
  - If some data is available, read returns up to the number of bytes requested
  - If no data is available, read returns 0 immediately

cbreak and Raw Mode

- cbreak mode
  - Non-canonical mode
  - Signals are allowed, but should be caught and restore the terminal mode
  - ECHO OFF
  - One byte at a time input: Set MIN = 1 and TIME = 0
- Raw mode
  - Non-canonical mode
  - Disable signal generating chars (ISIG) and extended input char processing (IEXTEN), disable break char (turn off BRKINT)
  - ECHO OFF
  - Disable CR-to-NL mapping (ICRNL), input parity detection (INPCK), the stripping of 8^{th} bit on input (ISTRIP) and output flow control (IXON)
  - Enable 8-bit chars (CS8)
  - Disable parity checking (PARENB), and all output processing (OPOST)
  - One byte at a time input: Set MIN = 1 and TIME = 0

Set to cbreak Mode

```
int tty_cbreak(int fd)  { /* put terminal into a cbreak mode */
  int    err;
  struct termios  buf;

  if (tcgetattr(fd, &buf) < 0)
     return(-1);
  buf.c_lflag &= ~(ECHO | ICANON);    // Echo off, canonical mode off
  buf.c_cc[VMIN] = 1;         // MIN = 1
  buf.c_cc[VTIME] = 0;        // TIME = 0
  if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
     return(-1);
  if (tcgetattr(fd, &buf) < 0)   // verify the configuration
     return(-1);
  if ((buf.c_lflag & (ECHO | ICANON))
  ||   buf.c_cc[VMIN] != 1 || buf.c_cc[VTIME] != 0)
     return -1;

  return(0);
}
```

Set to Raw Mode

```
int tty_raw(int fd)  { /* put terminal into a raw mode */
  ...
  buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  buf.c_cflag &= ~(CSIZE | PARENB);
  buf.c_cflag |= CS8;
  buf.c_oflag &= ~(OPOST);
  buf.c_cc[VMIN] = 1;       // MIN = 1
  buf.c_cc[VTIME] = 0;      // TIME = 0
  ...
  if ((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
      (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
      (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
      (buf.c_oflag & OPOST) ||
      buf.c_cc[VMIN] != 1 || buf.c_cc[VTIME] != 0)
     return -1;

  return 0;
}
```

Terminal Modes: Backup and Restore

```
staitc int save_fd;
static struct termios save_termios;

int tty_backup(int fd, struct termios *t) {
  if (fd < 0)                return -1;
  if (tcgetattr(fd, t) < 0) return -1;
  return fd;
}

int tty_restore(int fd, struct termios *t) {
  if (fd < 0)                            return -1;
  if (tcsetattr(fd, TCSAFLUSH, t) < 0) return -1;
  return fd;
}

void tty_atexit(void) {
  tty_restore(save_fd, &save_termios);
}

... save_fd = tty_backup(fd, &save_termios);
    atexit(tty_exit); ...
```

tty Modes: Example

```
int main() {
  int i, c;
  save_fd = tty_backup(0, &save_termios);
  atexit(tty_atexit);
  // REGISTER handler for SIGINT, QUIT, TERM (omitted)
  if (tty_raw(0) < 0) err_sys("tty_raw error");
  printf("Enter raw mode characters, terminate with DELETE\n");
  while ((i = read(0, &c, 1)) == 1) {
    if ((c &= 255) == 0177 /* DELETE */)  break;
    printf("%x\n", c);
  }
  if (tty_restore(0, &save_termios) < 0)  err_sys("tty_reset error");
  if (i <= 0)                             err_sys("read error");
  if (tty_cbreak(0) < 0)                  err_sys("tty_cbreak error");
  printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
  while ((i = read(0, &c, 1)) == 1)
    printf("%x\n", c & 255);
  if (tty_restore(0, &save_termios) < 0)  err_sys("tty_reset error");
  if (i <= 0)                             err_sys("read error");
  return 0;
}
```

tty Modes: Example (Cont'd)

- Raw mode: Ctrl-D (x4), F7, ESC (x1b), [(x5b), 1 (x31), 8 (x38), and ~ (x7e)
- cbreak mode: Ctrl-A, backspace, Ctrl-C

```
Enter raw mode characters, terminate with DELETE
4
1b
5b
31
38
7e
1b
5b
31
38
7e

Enter cbreak mode characters, terminate with SIGINT
1
8
signal caught
```

The results may be different depending on the terminal client

Terminal Window Size

- Most UNIX systems provide a way to keep track of the current terminal window size
- The kernel notifies the foreground process group when the size changes
  - We can fetch the size using an ioctl of TIOCGWINSZ
  - We can store a new value using an ioctl of TIOCSWINSZ
  - The kernel sends a SIGWINCH to the foreground process when the content of winsize structure changes

```
struct winsize {
    unsigned short ws_row;    /* rows, in characters */
    unsigned short ws_col;    /* columns, in characters */
    unsigned short ws_xpixel; /* horizontal size, pixels (unused) */
    unsigned short ws_ypixel; /* vertical size, pixels (unused) */
};
```

Windows Size Example

```
static void pr_winsize(int fd) {
     struct winsize size;
     if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
          err_sys("TIOCGWINSZ error");
     printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

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

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

Windows Size Example (Cont'd)

```
$ ./fig18.22-winch
41 rows, 141 columns
SIGWINCH received
40 rows, 136 columns
SIGWINCH received
23 rows, 80 columns
^C
```

termcap

- termcap stands for "terminal capability," and it refers to the text file /etc/termcap and a set of routines to read this file
- The termcap file contains descriptions of various terminals
  - What features the terminal supports, e.g., how many lines and rows and whether the terminal support backspace
  - How to make the terminal perform certain operations, e.g., clear the screen and move the cursor to a given location

termcap helps the implementation of full-screen application, e.g., the vi editor

termcap is a text-based file – we have to scan through the entire file to find the entry we want

On BSD workstation, you may have a look at the /etc/termcap file

TERM environment variable: the name of current terminal capability

termcap Example

- /etc/termcap from FreeBSD
- A number of capabilities are described for a given terminal, see termcap(5) for more details
- For example, the xterm-256color terminal
  - Co - maximum numbers of colors on screen
  - pa - maximum number of color-pairs on the screen
  - AB - Set ANSI background color
  - AF - Set ANSI foreground color
  - tc – Use capabilities inherited from another similar terminal

termcap Example (Cont'd)

Remember the F7 key we have in the RAW mode example?

```
xterm-basic|modern xterm common:\
        :am:bs:km:mi:ms:ut:xn:AX:\
        :Co#8:co#80:kn#12:li#24:pa#64:\
        :AB=\E[4%dm:AF=\E[3%dm:AL=\E[%dL:DC=\E[%dP:DL=\E[%dM:\
        :DO=\E[%dB:LE=\E[%dD:RI=\E[%dC:UP=\E[%dA:ae=\E(B:al=\E[L:\
        :as=\E(0:bl=^G:cd=\E[J:ce=\E[K:cl=\E[H\E[2J:\
        :cm=\E[%i%d;%dH:cs=\E[%i%d;%dr:ct=\E[3g:dc=\E[P:dl=\E[M:\
        :ei=\E[4l:ho=\E[H:im=\E[4h:is=\E[!p\E[?3;4l\E[4l\E>:\
        :kD=\E[3~:kb=^H:ke=\E[?1l\E>:ks=\E[?1h\E=:kB=\E[Z:le=^H:md=\E[1m:\
        :me=\E[m:ml=\El:mr=\E[7m:mu=\Em:nd=\E[C:op=\E[39;49m:\
        :rc=\E8:rs=\E[!p\E[?3;4l\E[4l\E>:sc=\E7:se=\E[27m:sf=^J:\
        :so=\E[7m:sr=\EM:st=\EH:\
        :ue=\E[24m:up=\E[A:us=\E[4m:ve=\E[?12l\E[?25h:vi=\E[?25l:vs=\E[?12;25h:

xterm-new|modern xterm:\
        :@7=\EOF:@8=\EOM:F1=\E[23~:F2=\E[24~:K2=\EOE:Km=\E[M:\
        :k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:k5=\E[15~:k6=\E[17~:\
        :k7=\E[18~:k8=\E[19~:k9=\E[20~:k;=\E[21~:kI=\E[2~:\
        :kN=\E[6~:kP=\E[5~:kd=\EOB:kh=\EOH:kl=\EOD:kr=\EOC:ku=\EOA:\
        :tc=xterm-basic:

xterm-256color|xterm alias 3:\
        :Co#256:pa#32767:\
        :AB=\E[48;5;%dm:AF=\E[38;5;%dm:tc=xterm-new:
```

terminfo

- terminfo is a terminal capability data base
  - xterm-color: /lib/terminfo/x/xterm-color, or /usr/share/terminfo/x/xterm-color
- You may consider terminfo as a “compiled” termcap
  - You can read capabilities from a database instead of parsing a text file
- Historically, we have BSD-derived systems used termcap and System-V based systems (e.g., Linux) used terminfo
  - Modern systems may support both termcap and terminfo
  - However, Mac OS X only supports terminfo
- Usually we will not work termcap nor terminfo directly
- Instead, we work with curses and ncurses library to simply the access to terminals

curses and ncurses

- We already know that different terminals will have different setups
- For example, the function key F7 for vt100 and xterm-256color
  - vt100: k7=\EOv
  - xterm-256color: k7=\E[18~
  - With TERM variable and termcap/terminfo, we would be able to interpret the key sequences
- There are even more differences for the terminals
- So … How to handle these differences in a *portable* manner?
- The curses / ncurses library
  - The CRT screen handling and optimization package
  - Give the user a terminal-independent method of updating character screens with reasonable optimization

Simple curses Example (see termio/curses.c)

```
static WINDOW *w = NULL;

int main() {
        if((w = initscr()) == NULL)  return -1;

        cbreak();           // cbreak mode
        noecho();           // no echo on key press
        nonl();             // no translation for new-line char
        intrflush(w, FALSE);     // no flush on interrupt
        keypad(w, TRUE);         // enable key pad support
        curs_set(0);        // hide cursor

        start_color();      // enable color
        init_pair(1, COLOR_RED, COLOR_BLACK);    // create color #1
        init_pair(2, COLOR_YELLOW, COLOR_BLACK); // create color #2

        signal(SIGWINCH, handler);   // handler for SIGWINCH
        // DO WHAT YOU WANT HERE
        endwin();           // reset everything

        return 0;
}
```

Q&A