1 Programming the User Interface

1.1 Terminal Control the Hard Way

- Controlling terminals under Linux is covered. It will show you the low-level APIs for controlling terminals and for controlling screen output in Linux applications.
- The notion of terminal control is a holdover from computing's earliest days, when users interacted with the CPU from dumb terminals.
- *The Terminal Interface*;
  - The terminal, or *tty*, interface derives from the days when users sat in front of a typewriter attached to a printer.
  - The *tty* interface is based on a hardware model that assumes a keyboard and printer combination is connected to a remote computer system using a serial port. This model is a distant relative of the current client-server computing architecture.
  - The model is sufficiently general that almost every situation in which a program needs to interact with some sort of input or output device, such as a printer, the console, xterms, or network logins, can be described as a subset of the general case.
  - As a result, the model actually simplifies the programmer's task because it provides a consistent programming interface that can be applied in a wide variety of situations.
  - Consider all of the different keyboard models, mice, joysticks, and other devices used to transmit user input. Add to that set all of the different kinds of output devices, such as modems, printers, plotters, serial devices, video cards, and monitors. The terminal interface has to accommodate all of these devices.

1.1.1 Controlling Terminals

- POSIX defines a standard interface for querying and manipulating terminals. This interface is called *termios* and is defined in the system header file $< termios.h >$.
- *termios* provides finely grained control over how Linux receives and processes input.

- From a programmer's perspective, *termios* is a data structure and a set of functions that manipulate it. (see man 3 termios)
- Terminals operate in one of two modes,
  - canonical (or cooked) mode, in which the terminal device driver processes special characters and feeds input to a program one line at a time, (the **shell** is an example of an application that uses canonical mode)
  - non-canonical (or raw) mode, in which most keyboard input is unprocessed and unbuffered. (the screen editor **vi**, uses non-canonical mode; **vi** receives input as it is typed and processes most special characters itself (D̂, for example, moves to the end of a file in **vi**, but signals EOF to the shell))
- The termios interface includes functions
  - Attribute Control Functions; the interface includes functions for controlling terminal characteristics.
  - Speed Control Functions; the first four functions set the input and output speed of a terminal device.
  - Line Control Functions; the line control functions query and set various properties concerned with how, when, and if data flows to the terminal device.
  - Process Control Functions; the process control functions the termios interface defines enable you to get information about the processes (programs) running on a given terminal.
- The Fig. 1 depicts the relationship between the hardware model and the *termios* data structures.

1.1.2 Using the Terminal Interface

- Of the functions that manipulate *termios* structures, *tcgetattr()* and *tcsetattr()* are the most frequently used.
- As their names suggest, *tcgetattr()* queries a terminal's state and *tcsetattr()* changes it. Both accept a file descriptor *fd* that corresponds to the process's controlling terminal and a pointer *tp* to a *termios struct*.
- The program in Fig. 2 illustrates using termios to turn off character echo when entering a password.

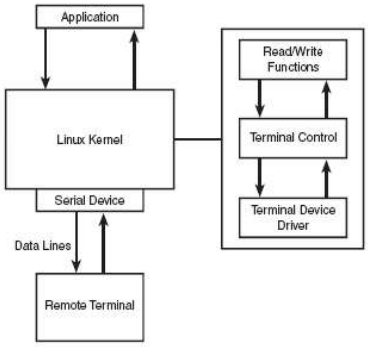

Figure 1: How the hardware model maps to the termios structure.

1.1.3 Changing Terminal Modes

- The program in Fig. 2 did manipulate some terminal attributes, but remained in canonical mode. The program in Fig. 3 puts the terminal in raw mode and performs its own processing on special characters and signals.

1.1.4 Using terminfo

- Where *termios* gives you very low-level control over input processing, *terminfo* gives you a similar level of control over output processing.
- *terminfo* provides a portable, lowlevel interface to operations such as clearing a terminal screen, positioning the cursor, or deleting lines or characters (see man 5 terminfo).
- *terminfo Capabilities*; For each possible terminal type, such as VT100 or xterm, terminfo maintains a list of that terminal's capabilities and features, called a capname, or CAPability NAME. Capnames fall into one of the following categories:
  - boolean,
  - numeric,
  - string.
- The program in Fig. 4 shows how to query and print a few (and some additional) of the current terminal's capabilities.

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#define PASS_LEN 8
void err_quit(char *msg, struct termios flags);
int main()
{
  struct termios old_flags, new_flags;
  char password[PASS_LEN + 1];
  int retval;
  /* Get the current terminal settings */
  tcgetattr(fileno(stdin), &old_flags);
  new_flags = old_flags;
  /* Turn off local echo, but pass the newlines through */
  new_flags.c_lflag &= ~ECHO;
  new_flags.c_lflag |= ECHONL;
  /* Did it work? */
  retval = tcsetattr(fileno(stdin), TCSAFLUSH, &new_flags);
  if(retval != 0) err_quit("Failed to set attributes", old_flags);
  /* Did the settings change? */
  tcgetattr(fileno(stdin), &new_flags);
  if(new_flags.c_lflag & ECHO)
    err_quit("Failed to turn off ECHO", old_flags);
  if(!new_flags.c_lflag & ECHONL)
    err_quit("Failed to turn on ECHONL", old_flags);
  fprintf(stdout, "Enter password: ");
  fgets(password, PASS_LEN + 1, stdin);
  fprintf(stdout, "You typed: %s", password);
  /* Restore the old termios settings */
  tcsetattr(fileno(stdin), TCSANOW, &old_flags);
  exit(EXIT_SUCCESS);
}
void err_quit(char *msg, struct termios flags)
{
  fprintf(stderr, "%s \n", msg);
  tcsetattr(fileno(stdin), TCSANOW, &flags);
  exit(EXIT_FAILURE);
}
```

Figure 2: Using termios to turn off character echo.

```
#include <termios.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
void err_quit(char *msg);
void err_reset(char *msg, struct termios *flags);
static void sig_caught(int signum);
int main(void)
{
  struct termios new_flags, old_flags;
  int i, fd;
  char c;
  /* Set up a signal handler */
  if(signal(SIGINT, sig_caught) == SIG_ERR)
    err_quit("Failed to set up SIGINT handler");
  if(signal(SIGQUIT, sig_caught) == SIG_ERR)
    err_quit("Failed to set up SIGQUIT handler");
  if(signal(SIGTERM, sig_caught) == SIG_ERR)
    err_quit("Failed to set up SIGTERM handler");
  fd = fileno(stdin);
  /* Set up raw/non-canonical mode */
  tcgetattr(fd, &old_flags);
  new_flags = old_flags;
  new_flags.c_lflag &= ~(ECHO | ICANON | ISIG);
  new_flags.c_iflag &= ~(BRKINT | ICRNL);
  new_flags.c_oflag &= ~OPOST;
  new_flags.c_cc[VTIME] = 0;
  new_flags.c_cc[VMIN] = 1;
  if(tcsetattr(fd, TCSAFLUSH, &new_flags) < 0)
    err_reset("Failed to change attributes", &old_flags);
  /* Process keystrokes until DELETE key is pressed */
  fprintf(stdout, "In RAW mode. Press DELETE key to exit\n");
  while((i = read(fd, &c, 1)) == 1) {
    if((c &= 255) == 0177)
      break;
    printf("%o\n", c);
  }
  /* Restore original terminal attributes */
  tcsetattr(fd, TCSANOW, &old_flags);
  exit(0);
}
void sig_caught(int signum)
{  fprintf(stdout, "signal caught: %d\n", signum);}

void err_quit(char *msg)
{  fprintf(stderr, "%s\n", msg);
  exit(EXIT_FAILURE); }
void err_reset(char *msg, struct termios *flags)
{  fprintf(stderr, "%s\n", msg);
  tcsetattr(fileno(stdin), TCSANOW, flags);
  exit(EXIT_FAILURE);}
```

Figure 3: Puts the terminal in raw mode.

```
#include <stdlib.h>
#include <stdio.h>
#include <term.h>
#include <curses.h>
#define NUMCAPS 3
void clrscr(void);
void mv_cursor(int, int);
int main(void)
{
  char *boolcaps[NUMCAPS] = { "am", "bce", "km" };
  char *numcaps[NUMCAPS] = { "cols", "lines", "colors" };
  char *strcaps[NUMCAPS] = { "cup", "flash", "hpa" };
  char *buf;
  int retval, i;
  if(setupterm(NULL, fileno(stdout), NULL) != OK) {
    perror("setupterm()");
    exit(EXIT_FAILURE);  }
  clrscr();
  for(i = 0; i < NUMCAPS; ++i) {
    /* position the cursor */
    mv_cursor(i, 10);
    retval = tigetflag(boolcaps[i]);
    if(retval == FALSE)
      printf("'%s' unsupported\n", boolcaps[i]);
    else
      printf("'%s' supported\n", boolcaps[i]);  }
  sleep(3);
  clrscr();
 for(i = 0; i < NUMCAPS; ++i) {
    mv_cursor(i, 10);
    retval = tigetnum(numcaps[i]);
    if(retval == ERR)
      printf("'%s' unsupported\n", numcaps[i]);
    else
      printf("'%s' is %d\n", numcaps[i], retval);  }
  sleep(3);

  clrscr();
  for(i = 0; i < NUMCAPS; ++i) {
    mv_cursor(i, 10);
    buf = tigetstr(strcaps[i]);
    if(buf == NULL)
      printf("'%s' unsupported\n", strcaps[i]);
    else
      printf("'%s' is \\E%s\n", strcaps[i], &buf[1]);  }
  sleep(3);
  exit(0);
}
// Clear the screen
void clrscr(void)
{  char *buf = tigetstr("clear");
  putp(buf);}
/* Move the cursor to the specified row row and column col*/
void mv_cursor(int row, int col)
{  char *cap = tigetstr("cup");
  putp(tparm(cap, row, col));}
```

Figure 4: Puts the terminal in raw mode.

1.2 Screen Manipulation with ncurses

- A much easier way to use a set of libraries for manipulating terminals. *ncurses* (new curses) provides a simple, high-level interface for screen control and manipulation.

- *ncurses*, the free implementation of the classic UNIX screenhandling library, curses (cursor optimization).

- Using *termios* or, even worse, the *tty* interface, to manipulate the screen's appearance is code-intensive. In addition, it is also terminal-specific.

- The System V UNIX releases continued curses' march along the feature trail, adding support for forms, menus, and panels.

  - Forms enable the programmer to create easy-touse data entry and display windows, simplifying what is usually a difficult and application- specific coding task.
  - Panels extend curses' ability to deal with overlapping and stacked windows.
  - Menus provide, well, menus, again with a simpler, generalized interface.

- compile with

```
$ gcc -o curses_prog curses_prog.c -lcurses
```

1.2.1 About Windows

- *Screen*; Screen refers to the physical terminal screen in character or console mode. Under the X Window system, *screen* means a terminal emulator window.

- *Window*; Window is used to refer to an independent rectangular area displayed on a screen. It may or may not be the same size as the screen.

- *stdscr*; This is an ncurses data structure, a (WINDOW *), that represents what you currently see on the screen. It might be one window or a set of windows, but it fills the entire screen. You can think of it as a palette on which you paint using ncurses routines.

- *curscr*; Another pointer to a WINDOW data structure, *curscr* contains ncurses' idea of what the screen currently looks like. Like *stdscr*, its size is the width and height of the screen. Differences between *curscr* and *stdscr* are the changes that appear on the screen.

- *Refresh*; This word refers both to an ncurses function call and a logical process. The *refresh()* function compares curscr, ncurses' notion of what the screen currently looks like, to *stdscr*, updates any changes to *curscr*, and then displays those changes to the screen. Refresh is also used to denote the process of updating the screen.

- *Cursor*; This term, like *refresh*, has two similar meanings, but always refers to the location where the next character will be displayed. On a screen (the physical screen), *cursor* refers to the location of the physical cursor. On a window (an ncurses window), it refers to the logical location where the next character will be displayed. Generally, in this chapter, the second meaning applies. ncurses uses a (y,x) ordered pair to locate the cursor on a window.

- *ncurses* defines window layout sanely and predictably. Windows are arranged such that the upper-left corner has the coordinates (0,0) and the lower-right corner has the coordinates (LINES, COLUMNS). Rather than using these global variables, however, use the function call *getmaxyx()* to get the size of the window with which you are currently working.

- *ncurses' Function Naming Conventions*;

  - the *move(y,x)* call, which moves the cursor to the coordinates specified by y and x on stdscr,
  - but *wmove(win, y, x)*, which moves the cursor to the specified location in the window *win*.
  - That is, *move(y, x)*; is equivalent to *wmove(stdscr, y, x)*.

1.2.2 Illustrating ncurses Initialization and Termination

The program and also other program in Fig. 5 illustrate the usage of the ncurses routines. The first program shows the standard ncurses initialization and termination idioms, using *initscr()* and *endwin()*, while the second one demonstrates the proper use of *newterm()* and *delscreen()*.

```
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
int main(void)
{
  if((initscr()) == NULL) {
    perror("initscr");
    exit(EXIT_FAILURE);  }
  printw("This is a curses window\n");
  refresh();
  sleep(3);
  printw("Going bye-bye now\n");
  refresh();
  sleep(3);
  endwin();
  exit(0);
}
```

```
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
int main(void)
{
  SCREEN *scr;
  if((scr = newterm(NULL, stdout, stdin)) == NULL) {
    perror("newterm");
    exit(EXIT_FAILURE);
  }
  if(set_term(scr) == NULL) {
    perror("set_term");
    endwin();
    delscreen(scr);
    exit(EXIT_FAILURE);  }
  printw("This curses window created with newterm()\n");
  refresh();
  sleep(3);
  printw("Going bye-bye now\n");
  refresh();
  sleep(3);
  endwin();
  delscreen(scr);
  exit(0);
}
```

Figure 5: UPPER: Shows the initialization and terminations. LOWER: Demonstrates the usage of newterm and delscreen.

1.2.3 Input and Output

- *ncurses* has many functions for sending output to and receiving input from screens and windows.
- It is important to understand that C's standard input and output routines do not work with ncurses' windows. Fortunately, ncurses' I/O routines behave very similarly to the standard I/O (< *stdio.h* >) routines, so the learning curve is tolerably shallow.
- Output Routines,
  - character routines,
  - string routines,
  - miscellaneous routines.
- The program in Fig. 6 illustrates ncurses' character output functions. Compile with

```
$ gcc -c utilfcns.c
$ gcc -o curschar curschar.c -lcurses utilfcns.o
```

- The program in Fig. 7 demonstrates using the string output functions.
- The program in Fig. 8 illustrates using line graphics characters and also the *box()* and *wborder()* calls.
- The program and also other program in Fig. 9 illustrates the usage of the input routines getch, getstr, and scanw.
- *Color Routines*;
  - we have already seen that ncurses supports various highlighting modes. Interestingly, it also supports color in the same fashion.
  - before you use ncurses' color capabilities, you have to make sure that the current terminal supports color. The *has_colors()* call returns TRUE or FALSE depending on whether or not the current terminal has color capabilities.
  - The program in Fig. 10 illustrates basic color usage. It must be run on a terminal emulator that supports color, such as a color xterm.

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"
int main(void)
{
  app_init();
  addch('X');
  addch('Y' | A_REVERSE);
  mvaddch(2, 1, 'Z' | A_BOLD);
  refresh();
  sleep(3);
  clear();
  waddch(stdscr, 'X');
  waddch(stdscr, 'Y' | A_REVERSE);
  mvwaddch(stdscr, 2, 1, 'Z' | A_BOLD);
  refresh();
  sleep(3);
  app_exit();
}
```

Figure 6: Illustrates ncurses' character output functions.

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"
int main(void)
{
  int xmax, ymax;
  WINDOW *tmpwin;
  app_init();
  getmaxyx(stdscr, ymax, xmax);
  addstr("Using the *str() family\n");
  hline(ACS_HLINE, xmax);
  mvaddstr(3, 0, "This string appears in full\n");
  mvaddnstr(5, 0, "This string is truncated\n", 15);
  refresh();
  sleep(3);
  if((tmpwin = newwin(0, 0, 0, 0)) == NULL)
    err_quit("newwin");
  mvwaddstr(tmpwin, 1, 1, "This message should appear in a new window");
  wborder(tmpwin, 0, 0, 0, 0, 0, 0, 0, 0);
  touchwin(tmpwin);
  wrefresh(tmpwin);
  sleep(3);
  delwin(tmpwin);
  app_exit();
}
```

Figure 7: Demonstrates using the string output functions.

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"
int main(void)
{
  int ymax, xmax;
  app_init();
  getmaxyx(stdscr, ymax, xmax);
  mvaddch(0, 0, ACS_ULCORNER);
  hline(ACS_HLINE, xmax - 2);
  mvaddch(ymax - 1, 0, ACS_LLCORNER);
  hline(ACS_HLINE, xmax - 2);
  mvaddch(0, xmax - 1, ACS_URCORNER);
  vline(ACS_VLINE, ymax - 2);
  mvvline(1, xmax - 1, ACS_VLINE, ymax - 2);
  mvaddch(ymax - 1, xmax - 1, ACS_LRCORNER);
  mvprintw((ymax / 3) - 1, (xmax - 30) / 2, "border drawn the hard way");
  refresh();
  sleep(3);
  clear();
  box(stdscr, ACS_VLINE, ACS_HLINE);
  mvprintw((ymax / 3) - 1, (xmax - 30) / 2, "border drawn the easy way");
  refresh();
  sleep(3);
  clear();
  wborder(stdscr, ACS_VLINE | A_BOLD, ACS_VLINE | A_BOLD,
  ACS_HLINE | A_BOLD, ACS_HLINE | A_BOLD,
  ACS_ULCORNER | A_BOLD, ACS_URCORNER | A_BOLD, \
  ACS_LLCORNER | A_BOLD, ACS_LRCORNER | A_BOLD);
  mvprintw((ymax / 3) - 1, (xmax - 25) / 2, "border drawn with wborder");
  refresh();
  sleep(3);
  app_exit();
}
```

Figure 8: Illustrates using line graphics characters.

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"
int main(void)
{
  int c, i = 0;
  int xmax, ymax;
  char str[80];
  WINDOW *pwin;
  app_init();
  crmode();
  getmaxyx(stdscr, ymax, xmax);
  if((pwin = subwin(stdscr, 3, 40, ymax / 3, (xmax - 40) / 2 )) == NULL)
    err_quit("subwin");
  box(pwin, ACS_VLINE, ACS_HLINE);
  mvwaddstr(pwin, 1, 1, "Password: ");
  noecho();
  while((c = getch()) != '\n' && i < 80) {
    str[i++] = c;
    waddch(pwin, '*');
    wrefresh(pwin);
  }
  echo();
  str[i] = '\0';
  wrefresh(pwin);
  mvwprintw(pwin, 1, 1, "You typed: %s\n", str);
  box(pwin, ACS_VLINE, ACS_HLINE);
  wrefresh(pwin);
  sleep(3);
  delwin(pwin);
  app_exit();
}
```

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include <string.h>
#include "utilfcns.h"
int main(int argc, char *argv[])
{
  char str[20];
  char *pstr;
  app_init();
  crmode();
  printw("File to open: ");
  refresh();
  getstr(str);
  printw("You typed: %s\n", str);
  refresh();
  sleep(3);
  if((pstr = malloc(sizeof(char) * 20)) == NULL)
    err_quit("malloc");
  printw("Enter your name: ");
  refresh();
  getnstr(pstr, 20);
  printw("You entered: %s\n", pstr);
  refresh();
  sleep(3);
  free(pstr);
  app_exit();
}
```

Figure 9: Illustrates the usage of the input routines.

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"
int main(void)
{
  int n;
  app_init();
  if(has_colors()) {
    if(start_color() == ERR)
      err_quit("start_color");
    /* Set up some simple color assignments */
    init_pair(COLOR_BLACK, COLOR_BLACK, COLOR_BLACK);
    init_pair(COLOR_GREEN, COLOR_GREEN, COLOR_BLACK);
    init_pair(COLOR_RED, COLOR_RED, COLOR_BLACK);
    init_pair(COLOR_CYAN, COLOR_CYAN, COLOR_BLACK);
    init_pair(COLOR_WHITE, COLOR_WHITE, COLOR_BLACK);
    init_pair(COLOR_MAGENTA, COLOR_MAGENTA, COLOR_BLACK);
    init_pair(COLOR_BLUE, COLOR_BLUE, COLOR_BLACK);
    init_pair(COLOR_YELLOW, COLOR_YELLOW, COLOR_BLACK);
    for(n = 1; n <= 8; n++) {
      attron(COLOR_PAIR(n));
      printw("color pair %d in NORMAL mode\n", n);
      attron(COLOR_PAIR(n) | A_STANDOUT);
      printw("color pair %d in STANDOUT mode\n", n);
      attroff(A_STANDOUT);
      refresh();
    }
    sleep(10);
  }
  else {
    printw("Terminal does not support color\n");
    refresh();
    sleep(3);
  }
  app_exit();
}
```

Figure 10: Illustrates basic color usage.

```
#include <stdlib.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"
int main(void)
{
  WINDOW *win;
  FILE *fdump;
  int xmax, ymax, n = 0;
  app_init();
  if(!has_colors()) {
    printw("Terminal does not support color\n");
    refresh();
    sleep(3);
    app_exit();
  }
  if(start_color() == ERR)
    err_quit("start_color");
  init_pair(COLOR_RED, COLOR_RED, COLOR_BLACK);
  init_pair(COLOR_YELLOW, COLOR_YELLOW, COLOR_BLACK);
  init_pair(COLOR_WHITE, COLOR_WHITE, COLOR_BLACK);
  bkgd('#' | COLOR_PAIR(COLOR_RED));
  refresh();
  sleep(3);
  if((win = subwin(stdscr, 10, 10, 0, 0)) == NULL)
    err_quit("subwin");
  wbkgd(win, '@' | COLOR_PAIR(COLOR_YELLOW));
  wrefresh(win);
  sleep(1);
  getmaxyx(stdscr, ymax, xmax);
  while(n < xmax - 10) {
    mvwin(win, ((ymax - 10) / 2), n);
    refresh();
    sleep(1);
    if(n == ((xmax - 10)/ 2)) {
      /* Dump the subwindow to a file */
      fdump = fopen("dump.win", "w");
      putwin(stdscr, fdump);
      fclose(fdump);
    }
    n += 10;
  }
  fdump = fopen("dump.win", "r");
  win = getwin(fdump);
  wrefresh(win);
  sleep(3);
  clear();
  bkgd(' ' | COLOR_PAIR(COLOR_WHITE));
  mvprintw(1, 1, "ERASE character: %s\n", unctrl(erasechar()));
  mvprintw(2, 1, "KILL character : %s\n", unctrl(killchar()));
  mvprintw(3, 1, "BAUDRATE (bps) : %d\n", baudrate());
  mvprintw(4, 1, "TERMINAL type  : %s\n", termname());
  refresh();
  sleep(5);
  delwin(win);
  app_exit();
}
```

Figure 11: Illustrates the usage of the some utility functions.

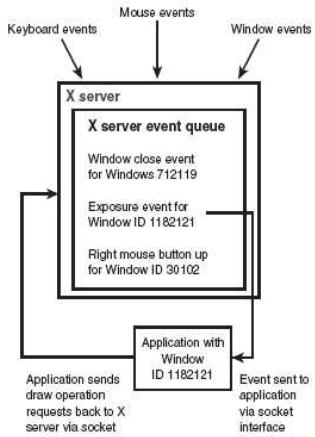

Figure 12: Interaction between events, the X server, and application programs.

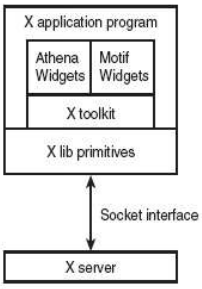

Figure 13: The X Window Programming APIs.

– The program in Fig. 11 illustrates using some of the utility functions.

ncurses is both an easier and more powerful interface for manipulating the display than termios. *ncurses* is a ubiquitous library, used in popular programs such as the mutt mail client, the Midnight Commander file manager, lynx, ncftp, and nvi.

1.3 X Window Programming

- The X Window software was written at MIT as part of the Athena project. X Windows allows a program to use the display of a computer

other than the computer the program is running on if the owner of that computer permits it.

- X Windows separates the display and event handling from application programs. Instead, an application program communicates with the X server via a socket interface.
- The X server handles keyboard input, mouse input, and the display screen. For example, when the user clicks the mouse, the X server detects where the mouse event occurred and sends the mouse event to the appropriate application program.
- When a window on the display is uncovered, the X server sends the appropriate application a window expose event.
- Window expose events occur when part or all of a window becomes visible and therefore needs to be redrawn.
- The application will usually respond by sending back draw operations to the X server to redraw the window contents.
- Figure 12 shows the interaction between actual user events, the X server event queue, and application program event queues.
- As shown in Figure 13, X Window applications can use any combination of the lowlevel Xlib API, the X toolkit (or the X intrinsics), the Athena Widget set, and the Motif Widget set.
- The Xlib library provides the interface between application programs and the (possibly remote) X server via a socket interface.

1.3.1 The Xlib API

- For most X Window applications that you'll develop, you'll probably use one of the high-level toolkits such as Motif, Athena, GTK, or Qt.
- However, in addition to providing the software interface between higher-level toolkits and the X server, the Xlib API provides useful graphics operations that you're likely to need in graphics-oriented applications.
- *XopenDisplay*; Xlib functions require a display pointer. We use XopenDisplay to connect a program to an X server. The display name has the general form hostname:displaynumber:screen-number. You might want to simply use localhost and default the display and screen numbers. The function signature is as follows:

```
Display * XOpenDisplay(char *display_name)
```

- *XcreateSimpleWindow and XcreateWindow*; The function XcreateWindow is the general-purpose function for creating new windows on an X display. The simpler XCreateSimpleWindow function creates a window using default values inherited from the parent window. The function signatures are as follows:

```
Window XcreateWindow(Display *display, Window parent,
int x, int y,
int width, int height,
unsigned int border_width,
unsigned int depth, int class,
Visual * visual,
unsigned long valuemask,
XSetWindowAttributes * attributes)

Window XCreateSimpleWindow(Display * display,
Window parent,
int x, int y,
unsigned int width,
unsigned int height,
unsigned int border_width,
unsigned long border,
unsigned long background)
```

  see man XSetWindowAttributes

- *Mapping and Unmapping Windows*; An X Window is made visible by *mapping* and *invisible* by unmapping. The function signatures for the utility functions that control window visibility are as follows:

```
XMapWindow(Display *display, Window w)
XMapSubwindows(Display * display, Window w)
XUnmapWindow(Display * display, Window w)
```

  - Mapping a window might not immediately make it visible because a window and all of its parents need to be mapped to become visible.
  - This behavior is convenient because you can make an entire tree of nested (or child) windows invisible by unmapping the topmost window.

  - When you unmap a window, the X server automatically generates a windows expose event for any windows that are uncovered.

- *Destroying Windows*; In order to free system resources, X Windows should be destroyed when they are no longer needed. The XDestroyWindow function destroys a single window, while XDestroySubWindows destroys all child windows. The signatures for these functions are as follows:

```
XDestroyWindow(Display * display, Window w)
XDestroySubwindows(Display * display, Window w)
```

- *Event Handling*; Event handling with the Xlib API is not too difficult. A program must register for the types of events that it is interested in by using *XSelectInput*, and check for events from the X server by using *XNextEvent*.

- *XSelectInput*; The function signature for XSelectInput is as follows:

```
XSelectInput(Display * display, Window w, long event_mask)
```

  The display variable defines which X server the application is using for display. Events are window-specific.

- *Initializing Graphics Context and Fonts*; An X window has a Graphics Context (GC) that specifies drawing parameters such as foreground and background colors and the current font. A GC is created with XCreateGC that has the following function signature:

```
GC XCreateGC(Display * display, Drawable d, unsigned long valuemask,
             XGCValues * values)
```

  - A *Drawable* is usually a Window object, but can also be a PixMap.
  - After defining a GC, a program might want to set the font, foreground color, and background color using the following functions:

```
XSetFont(Display *display, GC gc, Font font)
XSetForeground(Display *display, GC gc, unsigned long a_color)
XSetBackground(Display *display, GC gc, unsigned long a_color)
```

  - A color value can be obtained by using XParseColor or using one of the following macros:

```
BlackPixel(Display * display, Screen screen)
WhitePixel(Display * display, Screen screen)
```

- *Drawing in an X Window*; Usually, drawing in an X Window is performed after a program receives an expose event. The following list of drawing functions shows a sample of the graphics operations available to X Window programmers:

```
XDrawString(Display *display, Drawable d, GC gc, int x, int y,
            char * string, int string_length)
XDrawLine(Display *display, Drawable d, GC gc, int x1, int y1,
            int x2, int y2)
XDrawRectangle(Display *display, Drawable d, GC gc, int x,
                int y, unsigned int width, unsigned int height)
XDrawArc(Display *display, Drawable d, GC gc, int x, int y,
       unsigned int width, unsigned int height, int angle1, int angle2)
```

1.3.2 A Sample Xlib Program

- The sample program provides a simple example of creating a window, handling events, and performing simple graphics operations using the Xlib API.

- This simple program uses the function *draw_bifurcation* to draw the window contents. The function prototype (or signature) is as follows:

```
void draw_bifurcation(Window window, GC gc, Display *display,
                      int screen, XColor text_color)
```

- The function *main* defines the following data:

```
Display *display; For referring to X server's display
int screen; Identifies which screen on the X server you're using
Window win; Identifies the application's window
XColor blue; You want to display text using the color blue
unsigned int width=500; Specifies the initial window width
unsigned int height=231; Specifies the initial window height
XFontStruct *font_info; For using a display font
GC gc; Identifies the windows GC
Colormap cmap; Used to query the X server for the color blue
XEvent x_event; Used to fetch X events
XGCValues values; Used for the GC attributes returned by the X server
```

1.3.3 The X Toolkit API

- We have seen that the Xlib API is a low-level but very efficient library for writing X Window applications.
- The X toolkit (or intrinsics) library provides higher-level programming support for writing widgets.
- Widgets are object-oriented display objects, like data entry fields, utilities for plotting data, and so on. They are usually written in the C language, but they're object-oriented in the sense that they support inheritance and maintain private data with a public API for accessing internal widget data.
- Before an application can use the X toolkit, the following function must be called before any other toolkit functions (the type String is defined as char *, and a Cardinal is an int):

```
Widget XtInitialize(String shell_name, String application_class,
                    XrmOptionDescRec *options, Cardinal num_options,
                    int * argc, char **argv)
```