Critical Sections

- A **critical section** is a block of code modifying shared data
  - If interrupted, it may cause **incomplete** or **corrupted results**

- **Programing with signals**
  - Must identify and protect **critical sections**

- **To protect** critical sections
  - **Block** or **ignore signals** during that section
  - Prevent handlers from modifying the same data concurrently

Blocking Signals: `sigprocmask and sigsetops`

- You can block signals at the **signal-handler level** and the **process level.**

- **Blocking Signals in a Signal Handler** (p.225)

```
/* then build the list of blocked signals */
sigemptyset(&blocked);                  /* clear all bits      */
sigaddset(&blocked, SIGQUIT);           /* add SIGQUIT to list */
newhandler.sa_mask = blocked;           /* store blockmask     */
```

- **Blocking Signals for a Process**
  - A process has, at all times, a set of signals (called **signal mask**) it is blocking
  - To modify that set of blocked signals, use **`sigprocmask`**

```
sigprocmask( SIG_BLOCK, &sigs, &prevsigs);
// .. modify data structure here ..
sigprocmask( SIG_SET, &prevsigs, NULL);
```

sigprocmask		
PURPOSE	Modify current signal mask	
INCLUDE	`#include <signal.h>`	
USAGE	`int res = sigprocmask( int how, const sigset_t *sigs, sigset_t *prev);`	
ARGS	how	how to modify the signal mask
	sigs	pointer to list of signals to use
	prev	pointer to list of previous signal mask (or NULL)
RETURNS	-1	on error
	0	on success

how : SIG_BLOCK, SIG_UNBLOCK, or SIG_SET
(adding to, removing from, or replacing it with the signals in **sigs*)

If *prev* is not null, the previous signal mask is copied to **prev*.

- **Ex: Temporarily Blocking User Signals**
  - `SIGINT` and `SIGQUIT`

```
sigset_t  sigs, prevsigs;                    /* define two signal sets */

sigemptyset( &sigs );                        /* turn off all bits      */
sigaddset( &sigs, SIGINT );                  /* turn on SIGINT bit     */
sigaddset( &sigs, SIGQUIT );                 /* turn on SIGQUIT bit    */
sigprocmask( SIG_BLOCK, &sigs, &prevsigs);   /* add that to proc mask  */
// .. modify data structure here ..
sigprocmask( SIG_SET, &prevsigs, NULL);      /* restore previous mask  */
```

Building Signal Sets with `sigsetops`

- **sigset_t operations**

```
sigemptyset(sigset_t *setp)
```

Clear all signals from the list pointed to by *setp*.

```
sigfillset(sigset_t *setp)
```

Add all signals to the list pointed to by *setp*.

```
sigaddset(sigset_t *setp, int signum)
```

Add *signum* to the set pointed to by *setp*.

```
sigdelset(sigset_t *setp, int signum)
```

Remove *signum* from the set pointed to by *setp*.

7.10 kill: Sending Signals from a Process

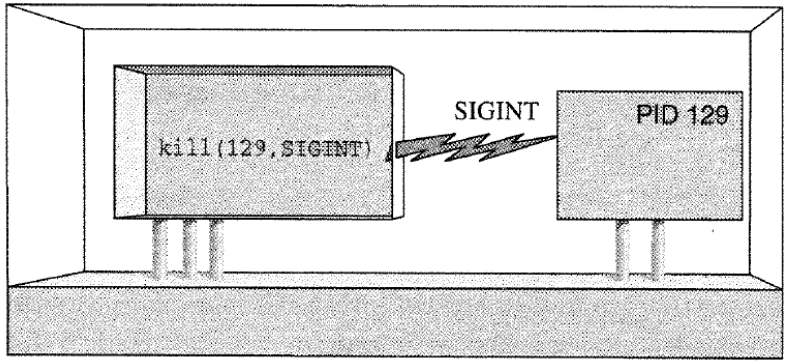


FIGURE 7.17

A process uses kill() to send a signal.

kill: Sending Signals from a Process

- A process can send a signal using the **kill()** system call

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

프로세스 간에 Signal를 보내기 위해 사용
프로세스 제어, 특정 이벤트를 알리기 위한 신호

kill: Sending Signals from a Process

- **The sending process of the kill()** must:
  - Have the **same user ID** as the target process, **or**
  - Be **root/superuser**
  - A process may even **send a signal to itself**

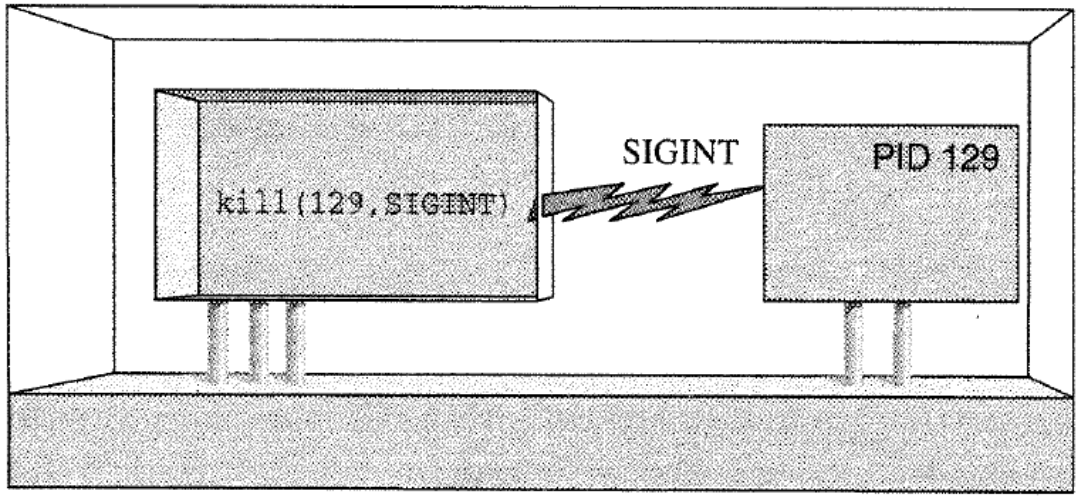


FIGURE 7.17

A process uses kill() to send a signal.

Implications for Interprocess Communication (IPC)

- **Processes can communicate and control each other** by
  - sending signals, with signal handlers enabling specific responses

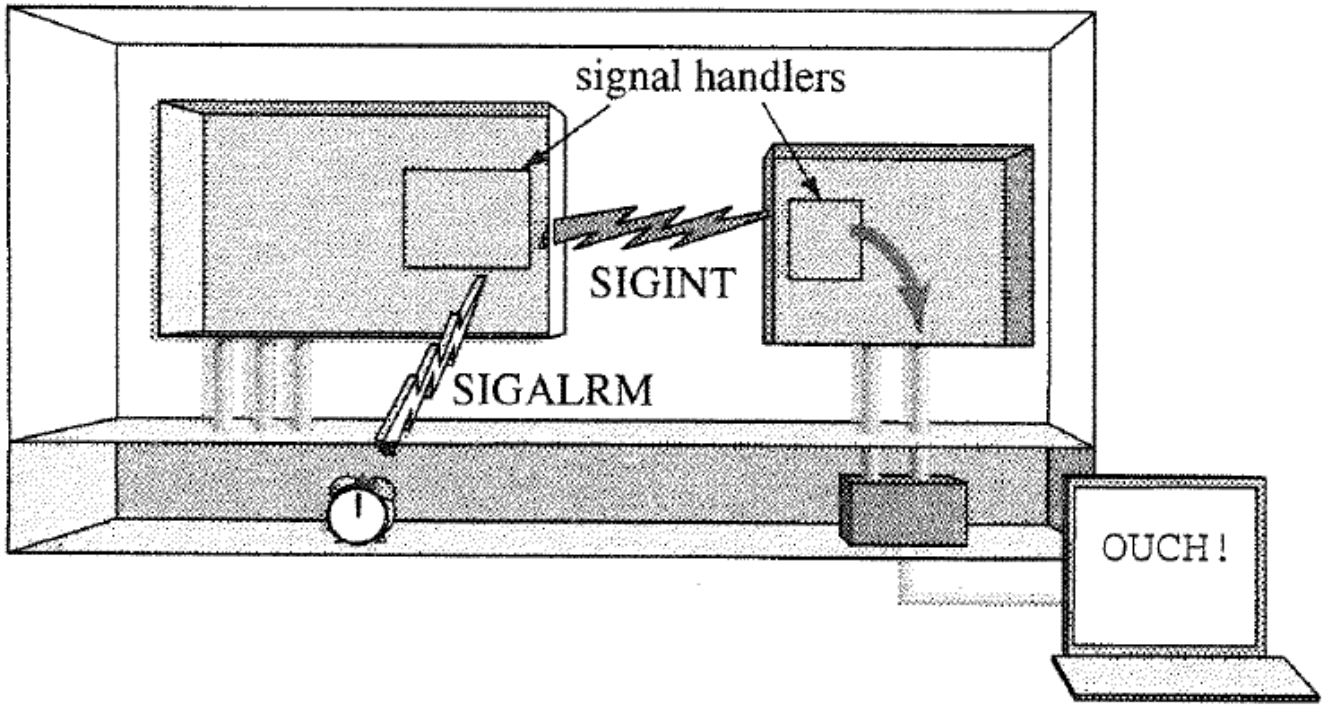


FIGURE 7.18

Complex use of signals.

Signals designed for IPC: `SIGUSR1`, `SIGUSR2`

- **Unix provides two user-defined signals:**
  - SIGUSR1
  - SIGUSR2
- **These signals:**
  - Have **no predefined meaning**
  - Are ideal for **custom interprocess communication**

```c
void signal_handler(int sig) {
    printf("Received signal %d\n", sig);
}

int main() {
    pid_t pid = getpid(); // Get the current process ID

    // Register signal handler for SIGUSR1
    if (signal(SIGUSR1, signal_handler) == SIG_ERR) {
        perror("Error registering signal handler");
        exit(1);
    }
    printf("Process ID: %d\n", pid);
    printf("Sending SIGUSR1 to itself...\n");

    // Send SIGUSR1 signal to the current process
    if (kill(pid, SIGUSR1) == -1) {
        perror("Error sending signal");
        exit(1);
    }
    pause();

    return 0;
}
```

```
2007200_40634@Ubuntu-038:~/다운로드$ ./ex
Process ID: 24932
Sending SIGUSR1 to itself...
Received signal 10
```

`bounce1d.c`: Controlled Animation on a Line

- Two main elements: ***animation*** **and** ***user input***

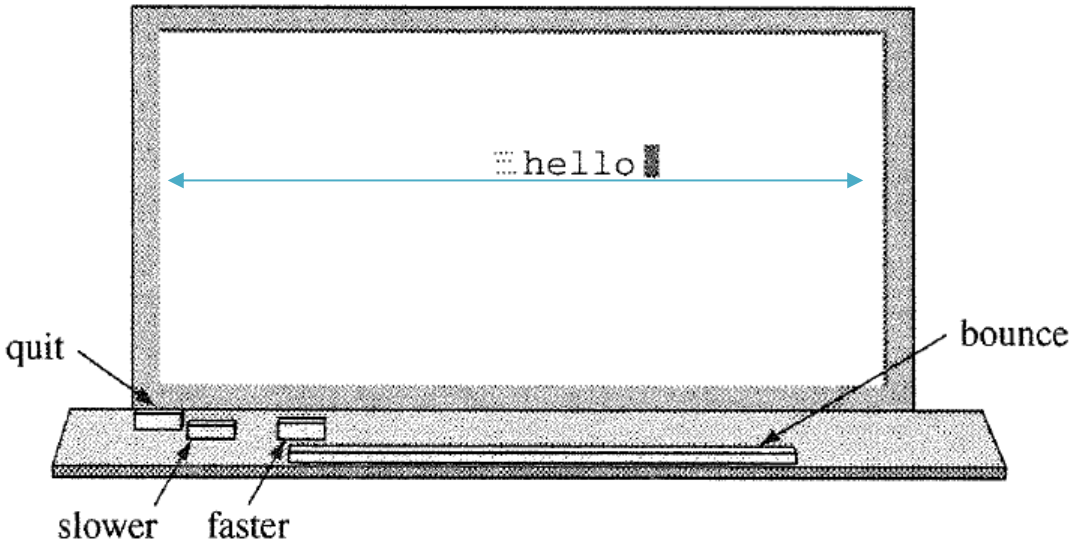


FIGURE 7.19

`bounce1d` in action: user-controlled animation.

space bar : the message reverses direction
s and f : make the message move slower and faster
Q : quits the game

```c
/* bounce1d.c
 *      purpose animation with user controlled speed and direction
 *      note    the handler does the animation
 *              the main program reads keyboard input
 *      compile cc bounce1d.c set_ticker.c -lcurses -o bounce1d
 */
#include        <stdio.h>
#include        <curses.h>
#include        <signal.h>
#include        <string.h>


/* some global settings main and the handler use */

#define MESSAGE "hello"
#define BLANK   "      "

int     row;    /* current row          */
int     col;    /* current column       */
int     dir;    /* where we are going   */

int set_ticker(int);
```

state variables

```
int main()
{
        int     delay;          /* bigger => slower     */
        int     ndelay;         /* new delay            */
        int     c;              /* user input           */
        void    move_msg(int);  /* handler for timer    */

        initscr();
        crmode();
        noecho();
        clear();

        row   = 10;             /* start here           */
        col   = 0;
        dir   = 1;              /* add 1 to row number  */
        delay = 200;            /* 200ms = 0.2 seconds  */

        move(row,col);          /* get into position    */
        addstr(MESSAGE);        /* draw message         */
        signal(SIGALRM, move_msg );
        set_ticker( delay );

        while(1)
        {
                ndelay = 0;
                c = getch();
                if ( c == 'Q' ) break;
                if ( c == ' ' ) dir = -dir;
                if ( c == 'f' && delay > 2 ) ndelay = delay/2;
                if ( c == 's' ) ndelay = delay * 2 ;
                if ( ndelay > 0 )
                        set_ticker( delay = ndelay );
        }
        endwin();
        return 0;
}
```

cbreak()

change the ***direction*** variable or ***speed*** variable.

change
the ***position***

```
void move_msg(int signum)
{
        signal(SIGALRM, move_msg);      /* reset, just in case  */
        move( row, col );
        addstr( BLANK );
        col += dir;                     /* move to new column   */
        move( row, col );               /* then set cursor      */
        addstr( MESSAGE );              /* redo message         */
        refresh();                      /* and show it          */
        /*
         * now handle borders
         */
        if ( dir == -1 && col <= 0 )
                dir = 1;
        else if ( dir == 1 && col+strlen(MESSAGE) >= COLS )
                dir = -1;
}
```

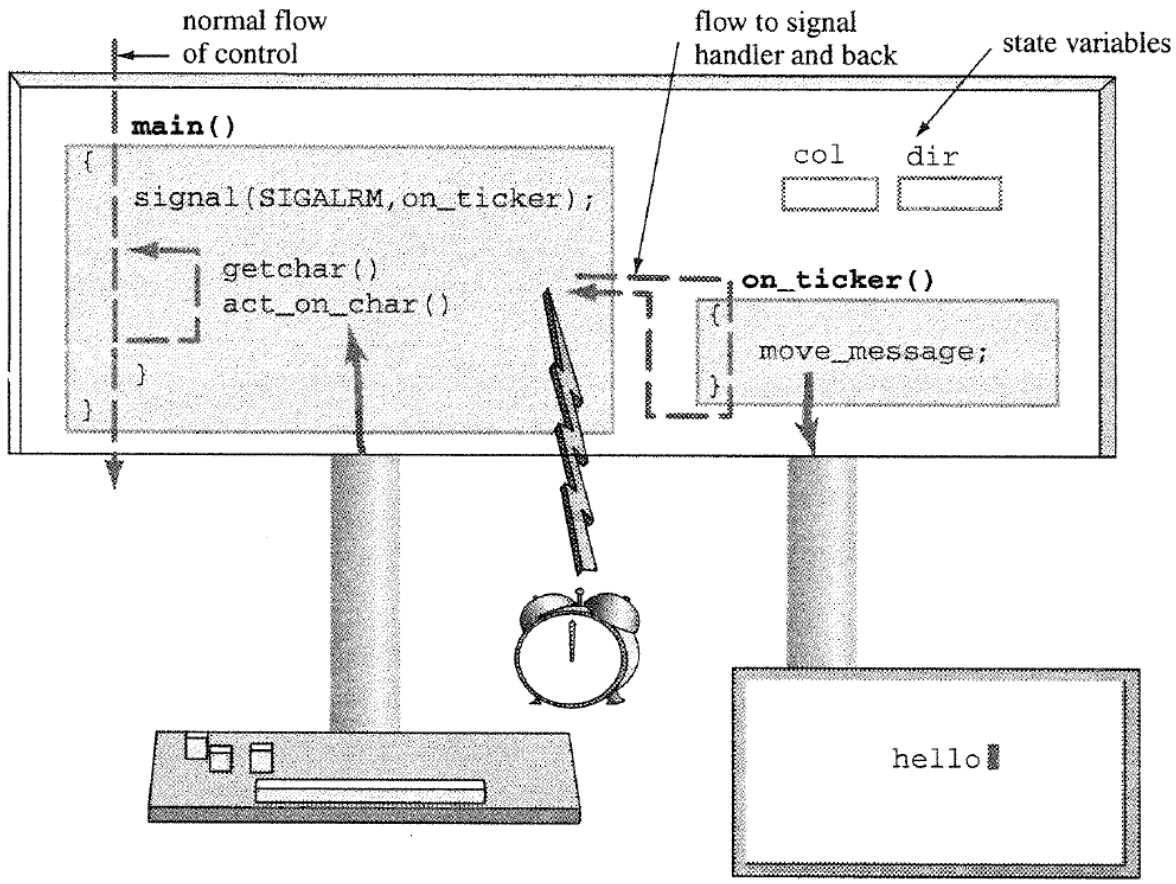


FIGURE 7.20

User input changes values. Values control action.

`bounce2d.c`: Animation in Two Dimensions

- **`bounce2d` uses *the same three-part design* of `bounce1d`.**
  - Timer Driven: …
  - Keyboard Blocked: …
  - State Variables

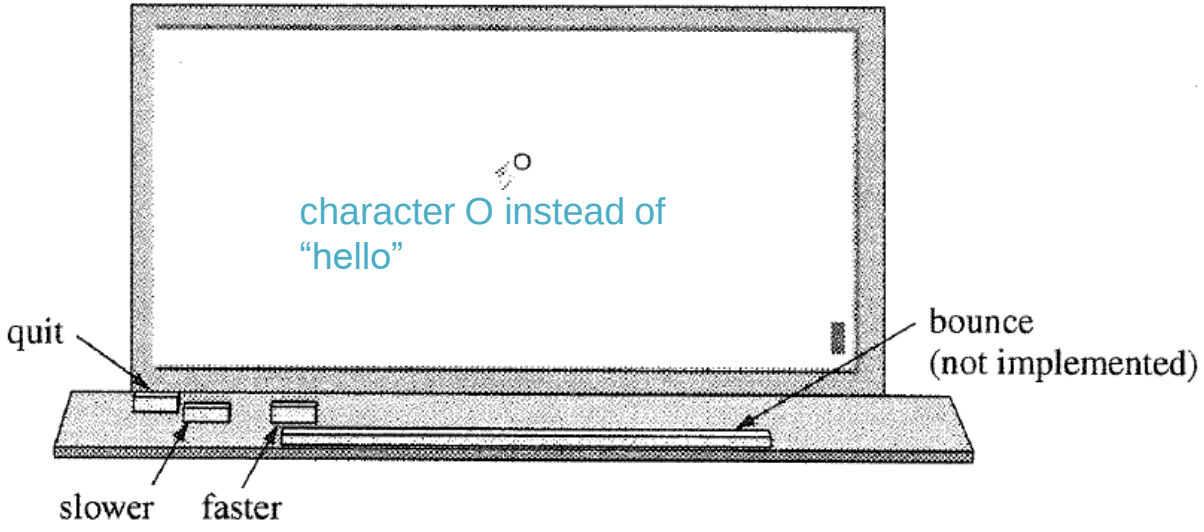
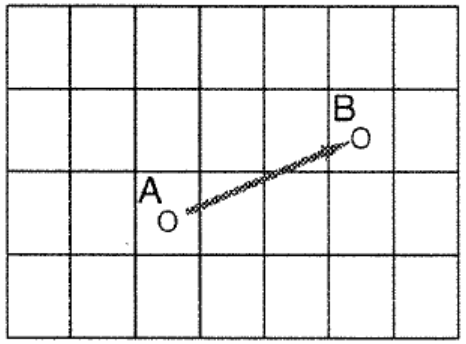


FIGURE 7.21

Animation in two directions.

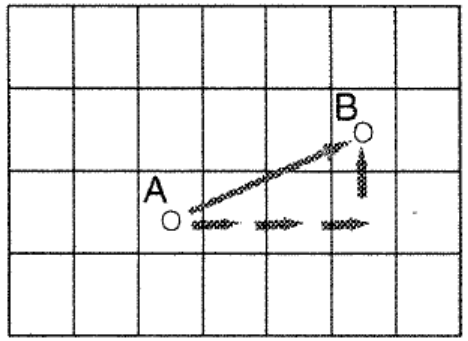
How Does the Ball Move along a Diagonal?



Question: How to move 'O' from cell A to cell B smoothly?

FIGURE 7.22

A path with a slope of $\frac{1}{3}$.



To approximate diagonal motion:

move right every two timer ticks, and move up every six timer ticks.

This technique requires two counters, one to count timer ticks for horizontal motion and one to count timer ticks for vertical motion.

FIGURE 7.23

Moving one cell at a time looks better.

- **A program only has one real-time interval ticker**
  - We need to build two timers of our own and use the interval timer to count down each of our timers.

- **The Code**
  - To produce motion in two directions at once, we create **two counters** to serve as timers.
  - Each of those counters has two parts (**value, interval**)
    - the # of ticks to go (ttg) before the next redraw
    - the # of ticks to move (ttm), that is, the interval between each redraw

```
/* bounce.h                          */

/* some settings for the game        */

#define BLANK           ' '
#define DFL_SYMBOL      'o'
#define TOP_ROW         5
#define BOT_ROW         20
#define LEFT_EDGE       10
#define RIGHT_EDGE      70

#define X_INIT          10              /* starting col        */
#define Y_INIT          10              /* starting row        */
#define TICKS_PER_SEC   50              /* affects speed       */

#define X_TTM           5
#define Y_TTM           8

/** the ping pong ball **/

struct ppball {
                int     y_pos, x_pos,
                        y_ttm, x_ttm,
                        y_ttg, x_ttg,
                        y_dir, x_dir;
                char    symbol ;
        } ;
```

※ two timers(counters) of our own

a counter for y direction

a counter for x direction

```
/*  bounce2d 1.0
 *      bounce a character (default is 'o') around the screen
 *      defined by some parameters
 *
 *      user input:     s slow down x component, S: slow y component
 *                      f speed up x component,  F: speed y component
 *                      Q quit
 *
 *      blocks on read, but timer tick sends SIGALRM caught by ball_move
 *      build:   cc bounce2d.c set_ticker.c -lcurses -o bounce2d
 */

#include        <stdio.h>
#include        <curses.h>
#include        <signal.h>
#include        <string.h>
#include        "bounce.h"


int set_ticker(int);
```

```
struct ppball the_ball ;

/**  the main loop  **/

void set_up();
void wrap_up();


int main()
{
        int     c;

        set_up();

        while ( ( c = getchar()) != 'Q' ){
                if ( c == 'f' )       the_ball.x_ttm--;
                else if ( c == 's' ) the_ball.x_ttm++;
                else if ( c == 'F' ) the_ball.y_ttm--;
                else if ( c == 'S' ) the_ball.y_ttm++;
        }
        wrap_up();
}
```

※ ‘ ‘ is not implemented.

```
void set_up()
/*
 *      init structure and other stuff
 */
{
        void    ball_move(int);

        the_ball.y_pos = Y_INIT;
        the_ball.x_pos = X_INIT;
        the_ball.y_ttg = the_ball.y_ttm = Y_TTM ;
        the_ball.x_ttg = the_ball.x_ttm = X_TTM ;
        the_ball.y_dir = 1  ;
        the_ball.x_dir = 1  ;
        the_ball.symbol = DFL_SYMBOL ;
        initscr();
        noecho();
        crmode();

        signal( SIGINT , SIG_IGN );
        mvaddch( the_ball.y_pos, the_ball.x_pos, the_ball.symbol  );
        refresh();

        signal( SIGALRM, ball_move );
        set_ticker( 1000 / TICKS_PER_SEC );  /* send millisecs per tick */
}

void wrap_up()
{

        set_ticker( 0 );
        endwin();               /* put back to normal   */
}
```

※ `mvaddch()` : add a character to a curses window, then advance the cursor. It is analogous to `putchar` in `stdio`.

```
void ball_move(int signum)
{
        int     y_cur, x_cur, moved;

        signal( SIGALRM , SIG_IGN );            /* dont get caught now  */
        y_cur = the_ball.y_pos ;                /* old spot             */
        x_cur = the_ball.x_pos ;
        moved = 0 ;

        if ( the_ball.y_ttm > 0 && the_ball.y_ttg-- == 1 ){
                the_ball.y_pos += the_ball.y_dir ;      /* move */
                the_ball.y_ttg = the_ball.y_ttm  ;      /* reset*/
                moved = 1;
        }

        if ( the_ball.x_ttm > 0 && the_ball.x_ttg- == 1 ){
                the_ball.x_pos += the_ball.x_dir ;      /* move */
                the_ball.x_ttg = the_ball.x_ttm  ;      /* reset*/
                moved = 1;
        }

        if ( moved ){
                mvaddch( y_cur, x_cur, BLANK );
                mvaddch( y_cur, x_cur, BLANK );
                mvaddch( the_ball.y_pos, the_ball.x_pos, the_ball.symbol );
                bounce_or_lose( &the_ball );
                move(LINES-1,COLS-1);
                refresh();
        }
        signal( SIGALRM, ball_move);            /* for unreliable systems */
}
```

```
int bounce_or_lose(struct ppball *bp)
{
        int     return_val = 0 ;

        if ( bp->y_pos == TOP_ROW ){
                bp->y_dir = 1 ;
                return_val = 1 ;
        } else if ( bp->y_pos == BOT_ROW ){
                bp->y_dir = -1 ;
                return_val = 1;
        }
        if ( bp->x_pos == LEFT_EDGE ){
                bp->x_dir = 1 ;
                return_val = 1 ;
        } else if ( bp->x_pos == RIGHT_EDGE ){
                bp->x_dir = -1;
                return_val = 1;
        }
        return return_val;
}
```

```
/* [from set_ticker.c]
 * set_ticker( number_of_milliseconds )
 *      arranges for interval timer to issue SIGALRM's at regular intervals
 *      returns -1 on error, 0 for ok
 *      arg in milliseconds, converted into whole seconds and microseconds
 *      note: set_ticker(0) turns off ticker
 */

int set_ticker( int n_msecs )
{
        struct itimerval new_timeset;
        long    n_sec, n_usecs;

        n_sec = n_msecs / 1000 ;                    /* int part      */
        n_usecs = ( n_msecs % 1000 ) * 1000L ;      /* remainder     */

        new_timeset.it_interval.tv_sec  = n_sec;    /* set reload */
        new_timeset.it_interval.tv_usec = n_usecs;  /* new ticker value */
        new_timeset.it_value.tv_sec     = n_sec  ;  /* store this       */
        new_timeset.it_value.tv_usec    = n_usecs;  /* and this         */

        return setitimer(ITIMER_REAL, &new_timeset, NULL);
}
```

millisecond

seconds

microseconds

Event-Driven Programming