

CSIS0230A Principle of Operating Systems (Class A)

Assignment 3

Real-time game under terminal

Tutor: kmcheung@csis.hku.hk, Deadline Dec 29, 2001, 11:59pm.

In this assignment, you are provided with a library (actually, header file) that supports all operations of a game that is very similar to an arcade game called “tetris”. Using this library, you should write a program which do the game interactively on a terminal emulator like kterm, gnome-terminal, xterm, rxvt or the Linux console.

This is a group assignment allowing at most 3 students in each group.

1. The game logistic

The following describes the game logistic. It assumes the reader has an idea about what is tetris. If you don't, you can first run the sample program available in the course web page, which is compiled for Redhat 6.2. Note that the sample program is not a complete implementation (e.g., high score is not shown at the beginning, reliable signal is not used, high score is not stored in the right directory, etc.)

- The game starts with a screen showing the current high scores. After the user press a specific key, the game starts.
- The size of the game board depends on the size of the terminal window. In particular, the board width is 10 less than the window width, and the board height is 1 less than the window height.
- At any time, there is a current piece to be added to the game board, and a next piece to become the current piece after the current piece is added to the game board.
- The board (which shows also the current piece) and the next piece are displayed on the terminal window.
- Initially, the current piece is placed at the center of the top row.
- There are five operations to manipulate the location and orientation of the current piece: shift left, shift right, rotate clockwise, rotate counterclockwise and down. Each of the first four are mapped to a key. If a movement failed, the program tells the user by beeping.
- The down operation will be automatically done every some time. Initially, at level 0, this will be done every second. Every increased level (explained below) will reduce this by 20%.
- If a down operation cannot be completed because it would hit something already on the board, it is added to the game board. All completely filled rows are removed. The game level increments every 5 rows removed.
- By pressing a particular key, the player can request that the down operation be repeatedly done until the current piece is added to the board.
- The game ends if, after a down operation, the new piece cannot be added to the board. The game also ends if the user changes the terminal size.
- The score obtained in a game is the total number of rows removed. The high score is stored globally in a file `/var/games/tt.hiscore`. It contains five lines, each containing a login name followed by the score obtained by the user in plain text.

2. The library

You are not required to write the core game engine: it is already done for you. You can get it from the web page of the course. It provides a class with the following operations:

```
class tetris_board {
public:
    // Empty class thrown when the window size is too small
    // during constructor, resize or restart
    class CreateBoardError {};
    // Constructor, passing width and height of board
    tetris_board(int w = 10, int h = 10);
    // Resize the board (which also restart the game)
    void resize(int w, int h);
    // Restart the game: clear board, renew the current and next piece
    void restart();
    // Get the piece down one row
    // Return 0 if new piece is created, 1 if game over
    // 2 if nothing bad happens
    // If 0 or 1 is returned, the number of rows removed is stored in nr
    int down(int &nr);
    // General movements. Return true if and only if successful.
    bool shift_left();
    bool shift_right();
    bool rot_cw();
    bool rot_ccw();
    // Get the representation of the board and current piece
    // which is represented as a vector of string,
    // each string representing a row from the top.
    const std::vector<std::string>& get_board();
    // Get the representation of the piece
    // which is represented as a structure,
    // containing a 2-D array member called piece.
    // The 2-D array is 4 by 4, row major.
    const tetris_piece& get_next_piece();
}
```

Everything else in the file is unimportant for this assignment. In the web page, you can also find a file *plain_tt.cc* which calls this library in the most primitive way.

3. Requirements and hints

- To allow cursor moving operations and clearing screens, escape sequence of the terminal must be used. You can find the escape sequences of the terminal using the ncurses library, using the `cup` and `clear` capabilities respectively (see `terminfo(5)` and `curls_terminfo(3)`).
- Use the ncurses library only for fetching and printing of escape sequences, and not to manage the keyboard or screen. In particular, only 4 functions are allowed: `setupterm()`, `tigetstr()`, `tparm()` and `putp()`. This restriction is made so that you learn the I/O primitives for terminal devices.

- The window size can be obtained by an `ioctl` call (`TIOCGWINSZ`) of the standard output descriptor (i.e., 1). See `ioctl(2)` to find the description of the `ioctl` system call. See `ioctl_list(2)` to find the parameters involved.
- You will need to configure the input to non-canonical mode (`ICANON`). The input mode should be fetched and set using `ioctl` calls (`TCGETS`, `TCSETS`) to the standard input descriptor (i.e., 0). See `termios(3)` for information.
- When the size of the terminal window changes, a signal (`SIGWINCH`) is sent to your process. Use this signal to determine that the current game should be abandoned.
- Your program should not have race condition when dealing with signals. In particular, there should not be circumstances when a signal raised or captured is lost. In practice, this means your whole program must be completely driven by signals, with the following main loop:

```
Block all signals (see sigprocmask)
Setup all handlers (see sigaction)
Setup timers (see setitimer)
Setup standard input (see fcntl about F_SETFL, O_NONBLOCK and O_ASYNC)
done = false;
while (!done) {
    Unblock, wait and block signals (see sigsuspend)
    if (key-pressed) { // SIGIO
        // Process key
    } else if (alarm occurred) { // SIGALRM
        // Call the down method of tetris_board
    } else if (window size changed) { // SIGWINCH
        done = true
    }
}
Unblock all signals
Reset timers and standard input
Reset all signal handlers
```

Note: there is actually another possibility using the `pselect()` system call. Read its man page and find the end of the DESCRIPTION section to see how to use it to support reliable signals.

- The high score file should be readable and writable only by a separate group, and should be installed by the system administrator and never be created by the program. Need-to-know principle must be followed: most of the time, your program have the special gid only as `SGID`, not `UGID` or `EGID`.
- File locking must be done to protect race condition when two person play the game on the machine at the same time and access the high score file simultaneously. Read `flock(2)` to find the details.
- Since we will need to perform low-level configurations, we recommend that you avoid the C++ library to deal with input and output.

4. Other useful manpage

- C library input and output: `getchar(3)`, `putchar(3)`, `fflush(3)`, `puts(3)`,

```
printf(3).
```

- Dealing with permissions: `getgid(2)`, `getegid(2)`, `setgid(2)`, `setegid(2)`.
- Handling files: `open(2)`, `close(2)`, `ftruncate(2)`, `lseek(2)`.
- Dealing with signal sets for signal masks: `sigsetops(2)`.

5. Appendix: Some example code to get you started

Terminfo: The ncurses library can be used to fetch and print escape sequence applicable for the current terminal. For example, from `terminfo(5)`, you can see a capability called `erase_chars` with terminfo code `ech`, which says to “erase #1 characters”. So the capability needs one argument. To use it, you should have something similar to the following in your code:

```
setupterm(0, 1, 0); // At the beginning of program, initialize terminfo
...
char *erase_esc_template = tigetstr("ech");
if (erase_esc_template == 0 || erase_esc_template == (char *)(-1)) {
    // Error and exit
}
char *erase_esc = tparm(erase_esc_template, 5); // Erase 5 characters
putp(erase_esc); // Print out the escape sequence
```

Both character arrays returned belong to ncurses, and should not be deallocated by your program. If a capability needs no argument, you don’t need to call `tparm()`.

Reliable signal: Early signal implementations suffer from a race condition that may cause signal to be lost. For example, suppose you have setup a timer, and want to wait for the SIGALRM signal in some piece of your code. This should be done by `pause()`.

```
// Setup the timer
... // do other things
pause(); // Wait for signal
... // Process SIGALRM
```

But it can happen that the signal occurs during the “do other things” section, before `pause()` function executes. In this case, `pause()` can never get the signal it waits for. The signal is said to be lost. To combat this possibility, the `sigsuspend()` system call is designed. The idea is to block the signal until we are ready to wait for it, and then unblock and wait for it atomically. It is used as follows.

```
sigset_t newmask, oldmask, zeromask;
sigemptyset(&zeromask); // Empty set of signals
sigemptyset(&newmask);
sigaddset(&newmask, SIGALRM); // Set containing just SIGALRM
sigprocmask(SIG_BLOCK, &newmask, &oldmask); // Block SIGALRM
// Setup the timer
... // do other things, with SIGALRM blocked
sigsuspend(&zero_mask); // Unblock SIGALRM, wait, and block SIGALRM again
... // Process SIGALRM
sigprocmask(SIG_SETMASK, &oldmask, 0); // Unblock SIGALRM
```