COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Today

- Terminal Programming
- Job Control
Why bother with text mode?

- Often only practical mode for accessing systems over the network
- Available on all development platforms
- Easier to implement than graphical interfaces

⇒ Text-mode often only/first interface for prototypes

⇒ Text-mode is great for testing: focus on problem, ignore GUI

- Learn more about good programming practice and operating systems
What is a Terminal?
What is a Terminal?

A VT-100!
What is a Terminal?

An xterm!
What is a Terminal?

• A device used for human-computer interaction

• Terminals provide text input and text output

• These days, a terminal can refer to hardware or software emulating hardware
Terminal = Keyboard + Monitor?

Not quite:

- Terminals can be programmed and support individual character IO or line-based IO
- Monitors display pixels, terminals display characters
- Keyboards report key strokes and key release, terminals report characters
- Keyboard hardware has no transmission problems; historical (networked) terminals did
Terminals vs. stdin/stdout

- **stdin** (when used with a terminal) is by default line-based
- **stdin** echos characters typed in to the terminal (by default)
- A terminal maybe shared by multiple processes!

⇒ Sharing of stdin and stdout must be controlled!
⇒ Job control
Controlling a Terminal

- Terminals are controlled using file descriptors (int)
- The same terminal maybe accessed via many different file descriptors
- Control operations affect all users of the (same) terminal
Identifying Terminals

• IO redirection may mean that stdin or stdout do not correspond to a terminal!

• int isatty(int filedes) can be used to check!
Input Processing Modes

• Canonical: input is processed in lines, the OS provides (minimal) editing facilities

• Noncanonical: granularity of input is not fixed, user can specify number of characters and timeout; no editing support by the OS
Getting and Setting Modes

- int tcgetattr(int filedes, struct termios * termios-p)
- int tcsetattr(int filedes, int when, const struct termios * termios-p)

- When: TCSANOW (immediately), TCSADRAIN (after queued output has been written), TCSAFLUSH (like DRAIN, but discard all input), TCSASOFT (read manual)
struct termios

struct termios {
    tcflag_t c_iflag;     // input
    tcflag_t c_oflag;     // output
    tcflag_t c_cflag;     // control (network)
    tcflag_t c_lflag;     // control (local)
    cc_t   c_cc[NCCS];   // special characters
    // ...
}

Setting Modes

- `struct termios` may have additional members
- Additional bits maybe defined in the future

⇒ Never set the entire structure! Instead:

1. Get the current values
2. Modify the `bit` you intend to change
3. Set the resulting structure
Example: enter non-canonical mode

void go_noncanonical(int td) {
    struct termios settings;

    if (0 != tcgetattr (td, &settings))
        return 1;
    settings.c_lflag &= ~ICANON;
    if (0 != tcsetattr (td, TCSANOW, &settings))
        return 1;
    return 0;
}

Example: enter canonical mode

```c
void go_canonical(int td) {
    struct termios settings;

    if (0 != tcgetattr (td, &settings))
        return 1;
    settings.c_lflag |= ICANON;
    if (0 != tcsetattr (td, TCSANOW, &settings))
        return 1;
    return 0;
}
```
Example: toggle echo

```c
void toggle_canonical(int td) {
    struct termios settings;

    if (0 != tcgetattr (td, &settings))
        return 1;
    settings.c_lflag ^= ECHO;
    if (0 != tcsetattr (td, TCSANOW, &settings))
        return 1;
    return 0;
}
```
Controlling the input rate

In noncanonical mode, the application has to define how soon input should be transmitted:

- `terminos.c_cc[VMIN]` specifies the minimum number of bytes that must be available in order for read to return

- `terminos.c_cc[VTIME]` specifies how long to wait for another character before returning

Details are in the GNU C library manual, section 17.4.10.
Other modes

- There are many more options for terminals
- Some are only of historic value (modems, old hardware)
- Read the manual if you need something specific!
Curses!

• Terminals interpret certain control sequences for special actions, such as deleting a character, moving the cursor or clearing the screen.

• Different (historic) terminals supported different control sequences; today, VT100’s set is the most common standard.

• The curses (and now ncurses) libraries provide a common API for virtually all terminals.
GNU readline

- Software library for line-input processing with editing
- `char * readline(const char * prompt)`
- Supports history using `add_history(const char * line)`
- Supports TAB completion
Job Control

- Concepts
- Sharing a Terminal
- SIGHUP
- Shells
Sessions

The canonical setup under UNIX is that:

- A session contains all of the processes associated with a particular terminal
- The shell is the session leader and controls the terminal
- `pid_t setsid(void)` creates a new session, making the current process the session leader
Process Groups

The canonical setup under UNIX is that:

- A process group contains all of the processes belonging to the same shell command
- `setpgid(pid_t pid, pid_t pgid)` sets the process group
- Job control operations in the shell determine which process group (currently) controls the terminal
Sharing a Terminal: Reading

- Only one process can read from the terminal at any given time

- This is the foreground job

- `int tcsetpgrp(int filedes, pid_t pgid_id)` sets the foreground job for a terminal

- All other processes trying to read will receive a SIGTTIN signal, which by default stops all of the processes in that group
Sharing a Terminal: Writing

• By default, multiple processes can write

• If TOSTOP is set in c_lflag, writing will cause a SIGTTOU signal, which again by default stops all processes in the group
What happens if the terminal is disconnected (or the session leader terminates)?

- A signal SIGHUP is sent to all processes in the group
- By default, this causes these processes to terminate
- SIGHUP can also be caught
- The `nohup` command installs a handler causing SIGHUP to be ignored
Shells

• The GNU C library manual (section 27.6) contains a skeleton

• You can use the code from the manual in your implementation

• You must acknowledge using code from the manual in comments

• You can use GNU’s libreadline for input processing
Questions