Implementation of the Terminal Subsystem and Job Control in the Mimiker Operating System

(Implementacja podsystemu terminali i kontroli zadań
w systemie operacyjnym Mimiker)

Jakub Piecuch

Praca magisterska

Promotorzy: Krystian Bacławski
Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

23 listopada 2020

Abstract

English Abstract

Polish Abstract

Contents

Chapter 1

Introduction

XXX nie jestem jeszcze pewien jak powinno wyglądać wprowadzenie. Zabiorę się za nie jako ostatnie. To co jest teraz napisane prawdopodobnie wyrzucę, ale zostawiam do wglądu.

1.1 What is a terminal?

1.1.1 Early-day terminals

In the early days of computing, a computer had a single operator. Users would submit their programs to the operator, usually on punched cards. When the user's turn came, the operator would load the program into the machine, and collect the output of the program in the form of punched cards. The whole process usually took hours. If the program contained an error, the user would know about it no sooner than when the operator loads it into the machine. Clearly, a more interactive way of submitting programs for execution was needed.

Time-sharing systems were the solution to this problem. A time-sharing system allows users to interact with the machine using a terminal. It gives the user an illusion of exclusive posession of the machine's resources. In reality, the operating system provides a fraction of the machine's computing resources to each user. Multics[10] is an early example of a time-sharing operating system.

In general, a terminal device is any device that allows its user to interact with a computing system, such as a mainframe. It allows the user to input commands, data, or even whole programs, and to examine the system's output. A terminal could communicate with the system remotely, or be directly attached to it.

In the beginning, terminals were electromechanical teletypewriters (teletypes, or TTYs for short), initially used in telegraphy. A teletype sends data typed on the keyboard to the computer, and prints the response.

In the 1970, video terminals came into widespread use. A video terminal displays output on a screen instead of printing it. Most models provide a set of special *escape codes*: sequences of characters that, when sent to the terminal, cause some action, e.g. clearing the screen. One of the most common features is a cursor that can be controlled using escape codes. This made tasks such as editing text much more enjoyable. The DEC VT100 series of video terminals was extremely successful, and is the basis of a de facto standard for terminal escape codes.

1.1.2 Terminals in a modern context

Nowadays, most people use their computer via a graphical user interface, or GUI. There is no dedicated device that handles interaction between the system and the user: input can be provided and output can be displayed by multiple devices. For instance, user input can be provided by a combination of a keyboard and mouse, and output can be displayed on a pair of monitors. Consequently, input and output handling in GUI programs is considerably more complex compared to programs written for character-based terminals. In addition, some people simply prefer to use a text-based, non-graphical interface due to its efficiency. For these reasons, it is useful to provide a means to run programs that use a terminal-based interface. This is accomplished using a *terminal emulator*.

A terminal emulator is a GUI program that emulates a terminal device, usually one that is compatible with the VT100 series of video terminals. The display of the emulated terminal is presented in the emulator's GUI window. The emulator translates the user's keystrokes into characters and feeds them as input to the application running inside the emulator. Output from the application is displayed in the GUI window in the same way it would be displayed on a real video terminal's screen. `xterm` is an example of a terminal emulator.

Another area where terminal-based interfaces are still used is communication over limited bandwidth connections, such as a Universal Asynchronous Receiver/-Transmitter (UART). Many development boards use a serial connection as the primary way of communication with the user.

To summarize, even though 1970s-style video terminals are no longer used, character-based computer interfaces that are compatible with those terminals remain an important part, or even the basis of interaction between a user and a computer system.

1.2 Job Control

The primary way of controlling a computer over a terminal connection is using a *shell*. A shell is a program that executes commands typed by the user. The purpose

of most commands is to run a specified program with a given set of arguments. For example, the command `ls -l` instructs the shell to find a program named `ls` and run it with a single argument `-l`. The executed program will then run to completion. It may accept further input from the user and write output. The shell waits for the program to finish, after which it is ready to accept more commands from the user.

This example presented a very simple use case. The user may want to execute *jobs* that consist of a pipeline of programs, with programs in the middle of the pipeline accepting input from the previous one and feeding output to the next one. Such jobs may run for a long time, so the user should be able to run any job in the background, without making the shell wait for it to finish.

Job control is a general feature of the system that allows the user to control running jobs. A job may be suspended and resumed, terminated, a background job may be brought into the foreground and vice versa. Job control usually requires support from the operating system, and it's up to the shell to group related programs into jobs.

Chapter 2

POSIX Terminals and Job Control

POSIX[9] is a set of standards specifying an operating system interface. Its primary goal is to make it possible to write portable applications that work across different operating systems, although the application usually needs to be compiled for each operating system separately.

Among other things, it defines an Application Programming Interface (API) for programs written in the C programming language that allows them to interact with the system and use its services. Every POSIX-compliant operating system must provide an implementation of this API.

The Mimiker operating system implements the POSIX API, but the implementation is far from complete. This allows us to use existing programs using the POSIX API like shells, command line utilities and text editors.

2.1 POSIX Job Control

2.1.1 Jobs

A job consists of one or more processes. The processes are usually connected in a pipeline, with each one accepting input from the previous one in the chain, and feeding output to the next one.

Jobs are a purely a shell concept: they are implemented entirely in the shell, and the operating system has no knowledge of them. However, there is a close correspondence between shell jobs and process groups, which are implemented by the operating system. The shell puts jobs in their own process groups.

2.1.2 Process groups

Process groups are central to job control. They relate processes performing a common task, such as a shell job. Process groups are identified by their *process group ID*, or *PGID* for short.

Every process belongs to some process group. When a new process is created using `fork()`[3], it runs in the process group of its parent.

A process can change its own process group, or that of one of its children. This is done using the `setpgid()`[4] function. It is used by the shell to set the process group of all the processes in a job.

TODO orphaned process groups

2.1.3 Sessions

The concept of a session is fairly intuitive. A new session starts when a user logs into the system. Initially, the user's shell is the only process in the session. All jobs (and therefore process groups) created by the shell belong to the same session. Every process group must belong to exactly one session. Session are collections of process groups, much in the same way as process groups are collections of processes.

Processes are not confined to their session: they can separate from it by creating their own session using the `setsid()` function. It creates a new session, initially containing just the calling process. All sessions are created in this way. Creating a new session necessarily means also creating a new process group: if it didn't, we could have two processes in the same process group, but in different sessions.

The process that creates a new session is called the *session leader*. Usually, the session leader a is shell, or some other program "in charge" of running and controlling all other programs in the session. It is supposed to be the last process in its session to exit. If a session leader exits while its session contains other processes, every process in the session will receive a `SIGHUP` signal, whose default effect is to kill the receiving process. Processes can ignore this signal, so it is possible for a session to outlive its leader.

Some programs are supposed to run indefinitely and without user intervention. A good example are *daemons*: programs that run in the background, e.g. providing services to other programs. A user may launch a daemon process from the shell. If the shell process exits, the daemon should continue to run. The daemon can become independent from the shell by creating its own session. When the shell exits, the daemon will not be notified in any way. Independence from the shell should not be confused with independence from the *user*: the user may open a shell in another session and send a signal to the daemon process, e.g. `SIGKILL`.

2.2 POSIX Terminals

On the surface, a terminal device is no different than a standard characters device: a process can open it using `open()` and then perform input and output using `read()` and `write()` respectively. The terminal interface specified by POSIX is much richer than that. Terminal devices play a significant role in job control, and therefore are connected to the concepts of process groups and sessions.

2.2.1 Controlling terminals

A terminal device may be the controlling terminal of a session. A session may have at most one controlling terminal, and a terminal device may be the controlling terminal of at most one session. A process can always access its controlling terminal (if it has one) by opening the `/dev/tty` device file.

When a session is created, it has no controlling terminal. The session leader is responsible for setting and changing the controlling terminal device, although the way in which this is done is not specified in the standard.

When the leader of a session terminates, the controlling terminal (if any) is dissociated from the session. Any processes left in the session may have their access to the controlling terminal revoked. If a controlling terminal device disappears from the system, the leader of the associated session is sent a `SIGHUP` signal.

2.2.2 Foreground process groups

Access to the controlling terminal by processes can be controlled using foreground and background process groups.

At any time, a process group can be designated as the foreground process group of its session's controlling terminal. All other process groups in the session are background process groups. The foreground process group of a terminal can be set using the `tcsetpgrp()`[1] function. Any process can set the foreground process group of its controlling terminal, although some restrictions apply to processes from background process groups. A controlling terminal does not need to have a foreground process group at all times.

Processes in the foreground process group are allowed to `read()` and `write()` to their controlling terminal. When a process in a background process group attempts to `read()` or `write()` to its controlling terminal, every process in its process group is sent a `SIGTTIN` or `SIGTTOU` signal respectively. The default effect of both signals is to stop the target process. In the case of `write()`, the signal is sent only if the `TOSTOP` terminal flag is set. If it is not set, background processes can write to their controlling terminal without restrictions. The exact semantics are a bit more

nuanced, see [2].

When a shell is accepting input from the user, its process is necessarily in the foreground process group. When the shell starts a job, the job is usually put in the foreground, so that the user can interact with it. The shell waits for the foreground job to complete, and then makes its own process group the foreground process group, so that it can accept the next command.

The user may make the job run in the background by appending `&` to the command. In that case, the shell remains in the foreground process group and does not wait for the job to complete. As explained earlier, a background job will be stopped if it tries to accept input from the user.

A background job can be put in the foreground using the `fg` command. The command simply sets the controlling terminal's foreground process group to the process group of the target job, continues the target job if it was stopped, and waits for the foreground job's termination.

2.2.3 Terminal modes and flags

Command line applications can be roughly divided into two groups, depending on how they process user input:

- One line at a time (*line-oriented*);
- One character at a time (*character-oriented*).

A shell belongs to the first group. It accepts a whole command at a time. Most notably, the user may edit the line before submitting it to the program by pressing the return key (labelled "enter" on most keyboards).

A text editor belongs to the second group. Each key performs an action within the editor, such as moving the cursor or inserting text at the position of the cursor.

In serious applications, line-oriented input is usually accomplished using a library such as GNU Readline [8]. However, POSIX mandates that basic line-editing functionality is provided by the operating system itself. Applications may choose whether they want to use this functionality by choosing the *terminal mode*.

Two terminal modes are distinguished: *canonical* and *non-canonical* (often called *raw*). Canonical mode provides basic line editing:

- Characters typed by the user are "echoed" to the terminal;
- Characters may be erased from the line.

It also causes certain actions to be taken in response to special *control characters* being received.

Character echoing allows the user to see the contents of the line they are typing. This is necessary for the user to be able to spot mistakes in their input. It is sometimes useful to turn off character echoing, e.g. when the user is typing their password.

Characters can be erased from the line by sending certain control characters. For instance, the `VERASE` character erases the character at the end of the line, and the `VKILL` character causes the whole line to be erased.

Raw mode does not perform any processing on characters received from the device. It is up to the application to perform tasks like echoing or erasing characters. This is exactly what libraries such as GNU Readline do.

The `termios` structure

For each terminal device, all of its settings, including the terminal mode, are stored in a `termios` structure associated with that terminal. Application code may examine and modify the contents of this structure using the `tcgetattr()`[5] and `tcsetattr()`[6] functions respectively.

The `termios` structure must contain the following fields:

- `c_iflag`: contains bit fields that control basic terminal input handling. Some notable bit fields are:
  - `ICRNL`: map the *carriage return* (CR) character to the *new line* (NL) character on input.
  - `BRKINT`: send a `SIGINT` signal to the foreground process group upon detecting a break condition.
  - `IGNBRK`: ignore hardware break conditions.
  - `INPCK`: enable input parity checking.
  - `IGNPAR`: ignore characters with parity errors.
- `c_oflag`: contains bit fields that control terminal output processing. Some notable bit fields are:
  - `OPOST`: enable output processing.
  - `ONLCR`: map NL to CR-NL sequence on output.
  - `OCRNL`: map CR to NL on output.
  - `ONOCR`: do not output CR characters if the cursor is at column 0.
- `c_cflag`: contains bit fields that control terminal hardware parameters. Some notable bit fields are:
  - `CREAD`: enable hardware receiver.

  - `CSIZE`: number of bits transmitted or received per byte (from 5 to 8).

- `c_lflag`: contains bit fields that control various additional functions. Some notable bit fields are:
  - `ECHO`: enable character echoing. This flag is cleared when the user is typing their password.
  - `ICANON`: enable canonical mode.
  - `ISIG`: send signals in response to certain control characters.
  - `TOSTOP`: send `SIGTTOU` if a background process tries to write to the terminal.

- `c_cc`: array defining control character codes. A control character can be disabled by setting its code to `_POSIX_VDISABLE`. Important control characters include:
  - `VERASE`: erase the character at the end of the line.
  - `VKILL`: erase the whole line.
  - `VEOF`: marks end of user input.
  - `VINTR`: if the `ISIG` flag is set, send `SIGINT` to the foreground process group.
  - `VSUSP`: if the `ISIG` flag is set, send `SIGTSTP` (terminal stop signal) to the foreground process group.

For the full specification of the `termios` structure, see [7].

Chapter 3

Implementation of POSIX Terminals and Job Control in the Mimiker Operating System

3.1 Job Control

3.1.1 Jobs

3.1.2 Process groups

3.1.3 Sessions

3.2 Terminals

3.2.1 Creating a terminal device

3.2.2 Terminal input and output

3.2.3 Controlling terminals

3.2.4 Foreground process groups

Chapter 4

Implementation Challenges

4.1 Process control

4.1.1 Stopping and resuming processes with signals

4.1.2 Interruptible sleep and stop signals

4.2 Locking

Chapter 5

Conclusion

Bibliography

[1] `tcsetpgrp` - set the foreground process group ID.
`https://pubs.opengroup.org/onlinepubs/9699919799/functions/tcsetpgrp.html`.
Accessed: 17/11/2020.

[2] Terminal Access Control.
`https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap11.html#tag_11_01_04`.
Accessed: 19/11/2020.

[3] `fork` - create a new process.
`https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html`.
Accessed: 17/11/2020.

[4] `setpgid` - set process group ID for job control.
`https://pubs.opengroup.org/onlinepubs/9699919799/functions/setpgid.html`.
Accessed: 17/11/2020.

[5] `tcgetattr` - get the parameters associated with the terminal.
`https://pubs.opengroup.org/onlinepubs/9699919799/functions/tcgetattr.html`.
Accessed: 17/11/2020.

[6] `tcsetattr` - set the parameters associated with the terminal.
`https://pubs.opengroup.org/onlinepubs/9699919799/functions/tcsetattr.html`.
Accessed: 17/11/2020.

[7] `termios.h` - define values for termios.
`https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/termios.h.html`.
Accessed: 23/11/2020.

[8] The GNU Readline Library.
`https://tiswww.case.edu/php/chet/readline/rltop.html`.
Accessed: 23/11/2020.

[9] The Open Group Base Specifications Issue 7, 2018 edition.
`https://pubs.opengroup.org/onlinepubs/9699919799/`.
Accessed: 17/11/2020.

[10] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), page 185–196, New York, NY, USA, 1965. Association for Computing Machinery.