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

9 grudnia 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.

POMYSŁ NA WPROWADZENIE:

- Przedstawienie kontroli zadań w powłoce (motywacja).
- Jakie rzeczy są potrzebne żeby to wszystko działało? Sygnały, grupy procesów itp.
- W Mimikerze to nie działało, a teraz działa, i o tym jest ta praca.

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

This chapter lays out the implementation of POSIX terminals and job control in the Mimiker operating system. The discussion focuses on the kernel, as that is where the most work had to be done. The relevant kernel data structures and functions are described.

3.1 Job Control

3.1.1 Jobs

Jobs are purely a userspace concept, and are implemented by the shell. The Mimiker operating system uses `ksh`, which is an existing implementation of a POSIX compatible shell.

3.1.2 Process groups

A process group is represented in the kernel as a structure of type `pgrp_t`.

```
typedef struct pgrp {
  mtx_t pg_lock;
  TAILQ_ENTRY(pgrp) pg_hash;
  TAILQ_HEAD(, proc) pg_members;
  session_t *pg_session;
  int pg_jobc;
  pgid_t pg_id;
} pgrp_t;
```

Listing 1: `include/sys/proc.h`: definition of `pgrp_t`.

The `pg_lock` mutex synchronizes concurrent accesses to `pg_members`. `pg_hash` is used to link the structure into the global hash table mapping process group IDs to process group structures. `pg_members` is a list of processes belonging to this process group. `pg_session` points to the session that this process group belongs to. `pg_jobc` is a counter used to detect orphaned process groups. `pg_id` holds the process group ID.

Orphaned process groups

Let us say that a process *qualifies its process group for job control* when its parent process is in a different process group, but in the same session. According to this notion, a process group is orphaned when it has no processes that qualify it for job control.

The `pg_jobc` field keeps track of the number of processes that qualify the process group for job control. When it drops to 0, the process group becomes orphaned, which results in `SIGHUP` and `SIGCONT` being sent to every stopped process in the group.

The field's value requires adjustment whenever a process from the group or its parent changes process groups. The `pgrp_jobc_enter()` and `pgrp_jobc_leave()` functions perform the necessary adjustments when a process enters or leaves a process group. When `pgrp_jobc_leave()` notices a process group's `pg_jobc` drop to 0, it orphans the process group, sending the necessary signals.

Process group lifecycle

A new process group may be created only in `sys_setpgid()`, which is the kernel's implementation of the POSIX `setpgid()` function. Processes may enter and leave the process group. When a process terminates and becomes a zombie process, it remains on its process group's `pg_members` list. When a process is reaped, it is removed from the list.

Once the list is empty, the process group's lifetime ends. During deallocation, a check is performed to see if the controlling terminal's foreground process group is equal to the group being deallocated, in order to prevent a dangling reference. If it is, the foreground process group is set to `NULL`.

TODO maybe some code walkthrough, e.g. `setpgid()`?

3.1.3 Sessions

The `session_t` structure describes a session in the kernel.

```
typedef struct session {
  TAILQ_ENTRY(session) s_hash;
  proc_t *s_leader;
  int s_count;
  sid_t s_sid;
  tty_t *s_tty;
} session_t;
```

Listing 2: `include/sys/proc.h`: definition of `session_t`.

The `s_hash` field links the structure into the global hash table mapping session IDs to session structures. `s_leader` points to the session leader if one exists. `s_count` is the number of process groups in the session. `s_sid` holds the session ID. `s_tty` points to the session's controlling terminal, if any.

3.2 Terminals

The `tty_t` structure encapsulates state associated with a single terminal device.

```
typedef enum {
  TF_WAIT_OUT_LOWAT = 0x1,
  TF_WAIT_DRAIN_OUT = 0x2,
  TF_OUT_BUSY = 0x4,
} tty_flags_t;

typedef struct tty {
  mtx_t t_lock;
  tty_flags_t t_flags;
  ringbuf_t t_inq;
  condvar_t t_incv;
  ringbuf_t t_outq;
  condvar_t t_outcv;
  linebuf_t t_line;
  size_t t_column;
  size_t t_rocol, t_rocount;
  condvar_t t_serialize_cv;
  ttyops_t t_ops;
  struct termios t_termios;
  pgrp_t *t_pgrp;
  session_t *t_session;
  vnode_t *t_vnode;
  void *t_data;
} tty_t;
```

Listing 3: `include/sys/tty.h`: definition of `tty_t`.

The `t_lock` mutex synchronizes access to the structure. `t_flags` may contain the following flags:

- `TF_WAIT_OUT_LOWAT`: a thread is waiting for space to become available in the output queue;
- `TF_WAIT_DRAIN_OUT`: a thread is waiting for the output queue to become empty;
- `TF_OUT_BUSY`: a thread is currently writing to this terminal. This flag is used to serialize `write()` calls on a terminal device.

`t_inq` is a ring buffer storing input coming from the terminal device. `read()` calls on the terminal device file receive characters from this buffer. `t_incv` is a condition variable used to wait for input to become available in `t_inq`.

`t_outq` is a ring buffer storing output from processes. The terminal device driver reads characters from this buffer and takes care of transmitting them. `t_outcv` is a

condition variable used to wait for space to become available in the output queue or for the queue to become empty.

`t_line` stores the contents of the line the user is typing in canonical mode. Its contents can be edited by the user. Once the user submits the line, its contents are copied into `t_inq`. The contents if `t_inq` cannot be changed. In raw mode, `t_line` is not used.

`t_column` keeps track of the position of the terminal cursor. It is needed to support the `ONOCR` terminal flag, which forbids sending the carriage return character when the cursor is at column 0.

The `t_rocol` and `t_rocount` fields are needed due to the fact that the echoed line can be interleaved with output from processes, and we only want to let the user erase the characters they typed, not ones output by processes. The two fields keep track of the longest prefix of the line that is not interrupted by output from processes.

The `t_serialize_cv` condition variable is used by processes calling `write()` to wait for their turn to access the terminal.

`t_ops` is a structure containing implementations of terminal device driver operations. Currently, only one operation is provided by device drivers:

```
typedef void (*t_notify_out_t)(struct tty *);

typedef struct {
  t_notify_out_t t_notify_out;
} ttyops_t;
```

Listing 4: `include/sys/tty.h`: definition of `ttyops_t`.

The `t_notify_out` operation notifies the device driver that new data has appeared in the terminal structure's output buffer. The driver should ensure that data from the buffer is written out to the device as soon as possible.

`t_termios` stores the terminal configuration described in the previous chapter.

`t_pgrp` points to the foreground process group, if any.

`t_session` points to the session controlled by this terminal.

`t_vnode` points to the filesystem node representing the terminal device. It is used to implement the `/dev/tty` device file, which refers different devices for processes in different sessions.

`t_data` is an opaque pointer for the device driver to store its private data.

3.2.1 Creating a terminal device

When a serial device driver attaches to a device, it is responsible for creating its corresponding terminal device. This involves allocating a new `tty` structure, setting device-specific fields in that structure (e.g. `t_ops`) and creating a terminal device node in the `/dev` directory of the filesystem.

When creating the file, the driver supplies `tty_fileops` as the implementation of file operations. This way, `read()` and `write()` (and other) calls on the file are directed to the TTY subsystem, which may in turn call the serial device driver via `t_ops`.

In the current implementation, which aims for simplicity, terminal devices live forever. Once a terminal file is created, it never goes away.

3.2.2 Terminal input

Processes read data from a terminal by invoking `read()` on an open terminal file descriptor. The terminal subsystem implements this operation in the `tty_read()` function.

As mandated by POSIX, the function first checks whether the calling process is in the foreground process of its controlling terminal, if it has one. If it is, the process may proceed. Otherwise, the process group of the calling process is sent a `SIGTTIN` signal, which will stop the process until it is resumed by a `SIGCONT` signal, most likely sent by the shell. The read will then be attempted again.

After passing the check, the process checks whether there are characters available in the terminal's input buffer. If it is empty, the process goes to sleep on the `t_incv` condition variable. The process will be woken up once there is data in the buffer.

If the input buffer contains characters, the process copies them into the userspace buffer supplied as an argument to `read()`. In canonical mode, care must be taken not to read more than a single line. If a line-terminating character is read from the buffer, no further characters are copied, but the line-terminating character is (with the exception of the special character `VEOF`).

We have covered how characters find their way from the terminal input buffer to the buffer supplied by the reading user process. The question that remains is: How are characters written into the input buffer?

A character's journey through the kernel begins with its reception by the serial device. An interrupt-driven device (e.g. a UART) signals the reception of a character by asserting its interrupt line, which will eventually lead to interrupting the CPU. The CPU will then jump to an *interrupt service routine*, or ISR.

The ISR will execute the serial device driver's code that extracts the character from the device registers and inserts it into the serial device's *receive buffer*. This buffer is different from the associated terminal's input buffer. The driver will then wake up a worker thread that copies characters from the receive buffer into the terminal's input buffer.

Characters being copied from the receive buffer must be processed in order to take appropriate actions in response to certain control characters being received. The degree of processing depends on the state of flags in the `termios` structure. For instance, if the `ISIG` flag is set, the characters `VINTR`, `VSUSP` and `VQUIT` respectively cause a `SIGINT`, `SIGTSTP` or `SIGQUIT` signal to be sent to the foreground process group of the receiving terminal.

The terminal subsystem provides the `tty_input()` function for inserting characters into the terminal input buffer. The function accepts a character and performs processing on it before appending it to the line buffer or the input buffer. Its control flow can be summarized by the following pseudocode:

```
void tty_input(tty_t *tty, uint8_t c) {
  if (ISIG flag is set) {
    /* Signal processing */
    if (c is VINTR or VSUSP or VQUIT) {
      send the appropriate signal to the foreground process group;
      return;
    }
  }

  if (ICANON flag is set) {
    /* Line editing */
    if (c is VERASE or VKILL) {
      erase the approriate characters from the line;
      return;
    }

    insert c into the line buffer;

    if (c is a line terminator) {
      copy the line buffer to the input buffer;
    }
  } else {
    insert c into the input buffer;
  }

  if (ECHO flag is set) {
    echo c;
  }
}
```

Listing 5: `kern/tty.c`: simplified control flow of `tty_input()`.

The `tty_input()` function cannot be called from an ISR, as the function uses blocking synchronization primitives, such as mutexes. This is why a worker thread is used instead of calling `tty_input()` directly from an ISR.

3.2.3 Terminal output

Processes write data to a terminal by invoking `write()` on an open terminal file descriptor. The kernel in turn calls `tty_write()` to carry out the operation.

If the `TOSTOP` terminal flag is set, a background check similar to the one in `tty_read()` is performed, with the difference that the signal sent to the offending process group is `SIGTTOU` instead of `SIGTTIN`.

After passing the check, the calling process writes the data into the terminal's output buffer. If at any point the buffer becomes full, the process goes to sleep on the `t_outcv` condition variable, waiting for the serial device driver to make space in the buffer.

After writing data to the output buffer or before going to sleep on `t_outcv`, the serial device driver is notified of new data. The `tty_notify_out()` function calls the `t_notify_out` function supplied by the driver. After that point, it is the responsibility of the driver to write the characters in `t_outq` to the serial device.

Unlike in the case of terminal input, the driver does not need to use an intermediate buffer between the terminal's output buffer and the hardware. It may simply write all data from the buffer in a single invocation of `t_notify_out`. In other words, the driver may output the characters *synchronously*. However, that would most likely require busy-waiting for the hardware to become ready to accept data, unnecessarily consuming CPU time. For this reason, an *asynchronous* approach is taken in the UART driver found in the Mimiker kernel.

The driver's implementation of `t_notify_out` copies characters from the terminal's output buffer to the *transmit buffer* associated with the device. Actual output is performed in the driver's ISR: the hardware signals that is ready to accept a character via an interrupt. The ISR then takes one character from the transmit buffer and writes it to a hardware register. When the transmit buffer runs out of characters, the ISR wakes up a worker thread that copies characters from the terminal's output buffer into the transmit buffer. The actual implementation doesn't perform output exclusively from the ISR, but that is just an optimization.

In the case of asynchronous drivers, the driver is also responsible for waking up writers waiting for space in the terminal's output buffer. This is done by calling the `tty_getc_done()` function after copying characters from the output queue. This function simply checks whether the number of characters in the output buffer has fallen below some threshold. If it has, it wakes up all waiters waiting on `t_outcv`. Similarly to `tty_input()`, this function cannot be called from an ISR, as it uses blocking synchronization primitives.

3.2.4 Controlling terminals

As seen in section 3.1.3, the controlling terminal for a session is stored in the `s_tty` field of `session_t`. When a new session is created, its `s_tty` field is set to `NULL`.

A terminal becomes associated with a session automatically when a session leader opens a terminal device file. The association happens only when the session doesn't already have a controlling terminal, and the terminal being opened isn't associated with any session.

When the session leader process exits, the session loses its controlling terminal.

This is the only way a terminal may become dissociated from its session.

3.2.5 Foreground process groups

The `t_pgrp` field in the `tty_t` structure points to the foreground process group associated with the terminal. Only controlling terminals (i.e. ones associated with a session) may have a foreground process group. When a terminal first becomes associated with a session, its foreground process group is set to the process group of the session leader.

The `tcgetpgrp()` and `tcsetpgrp()` functions are provided by the standard C library. Their implementations use the `ioctl()` system call to perform the required operation.

`ioctl()` is a very general, catch-all system call that is most commonly used to provide functionality that is too simple to merit a separate system call. It takes as arguments a file descriptor, an *opcode* (short for *operation code*), and a generic argument. The opcode is simply a numeric value that tells the kernel what operation to perform. The argument's interpretation depends on the specific opcode. Most opcodes are only valid for file descriptors of a certain type, e.g. ones representing open terminal device files. The opcodes used for getting and setting the foreground process group are `TIOCGPGRP` and `TIOCSPGRP` respectively.

When the foreground process group becomes empty, the terminal's `t_pgrp` field is set to `NULL`. Thus, a terminal may not have a foreground process group at all times, even if it is associated with a session.

Chapter 4

Implementation Challenges

In order to properly implement terminal and job control support in the kernel, many seemingly unrelated issues needed to be resolved. Some of them were unimplemented features, while others concerned more fundamental design choices in the kernel. This chapter describes a number of interesting issues that arose, and how they have been resolved.

4.1 Process control

Certain signals used by the terminal subsystem, such as `SIGTTOU` and `SIGTTIN`, may *stop* the receiving process. A stopped process cannot run or respond to signals until it is sent a `SIGCONT` signal.

There was no prior support for process stopping in the Mimiker kernel. In this section, we describe the implementation of this feature and some design questions that needed to be answered.

4.1.1 Stopping and resuming processes with signals

There are four signals that may stop the receiving process: `SIGSTOP`, `SIGTTOU`, `SIGTTIN`, and `SIGTSTP`. The first one is special in that it cannot be ignored or blocked from being delivered: if it is sent, it will always stop the target process.

The other three signals are *terminal stop signals*: they are usually sent by the terminal system in response to certain events, e.g. upon receiving the `VSTOP` control character. They may be ignored and blocked from being delivered. A terminal stop signal cannot stop a process that is a member of an orphaned process group. The reason for this is that an orphaned process group most likely doesn't have a supervising process such as a shell, and therefore it is unlikely that a stopped process in such a group would ever be resumed by a `SIGCONT` signal.

In the kernel, stop signals are sent in much the same way as any other signal, using the `sig_kill()` function. The function inserts the signal into the target thread's pending signal set and wakes it up from sleep if it is sleeping interruptibly. In addition, it removes any pending `SIGCONT` signals to avoid having contradictory signals pending at the same time.

All threads handle pending signals just before returning to userspace, in the `on_user_exc_leave()` function. The `sig_check()` function checks for pending signals that have installed handlers. If it finds a special signals that doesn't have a handler installed, such as `SIGKILL` or `SIGSTOP`, it takes the appropriate action. In the case of stop signals, it calls `proc_stop()`.

Process stopping

Stopping a process is not as simple as setting its state to `PS_STOPPED`. The parent of the stopping process must be notified of a child's status change by a `SIGCHLD` signal. At the end, the thread of the stopping process is stopped by settings its state to `TDS_STOPPED` and calling `sched_switch()`, which will perform a context switch to another thread.

Due to locking nuances, stopping a thread is a two-step operation. We first set the `TDF_STOPPING` flag in the thread to signal our intent to stop. We then release all locks held and acquire the thread spinlock, which must be the only lock held when performing a context switch. The context switch is then performed provided the `TDF_STOPPING` flag is still set. The code in listing shows the exact steps.

```
void proc_stop() {
  thread_t *td = thread_self();
  proc_t *p = td->td_proc;

  assert(mtx_owned(&p->p_lock));

  p->p_state = PS_STOPPED;
  notify parent of child status change;
  WITH_SPIN_LOCK (td->td_lock) { td->td_flags |= TDF_STOPPING; }
  proc_unlock(p);
  /* We're holding no locks here, so our process can be continued
   * before we actually stop the thread.
   * This is why we need the TDF_STOPPING flag. */
  spin_lock(td->td_lock);
  if (td->td_flags & TDF_STOPPING) {
    td->td_flags &= ~TDF_STOPPING;
    td->td_state = TDS_STOPPED;
    sched_switch(); /* Releases td_lock. */
  } else {
    spin_unlock(td->td_lock);
  }
  proc_lock(p);
  return;
}
```

Listing 6: `sys/proc.c`: simplified version of `proc_stop()`.

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.