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

27 maja 2021

Abstract

Most applications with a text user interface require support for the concept of a terminal to be present in the kernel, as well as in userspace (in the form of libraries, e.g. `ncurses`). Furthermore, managing programs launched using the shell (e.g. stopping or interrupting them) requires support for job control. The thesis presents the specification and implementation of the terminal subsystem and job control in the Mimiker operating system. Both the terminal subsystem and job control implementations comply with the POSIX specification, which allows for easy porting of programs from other systems compliant with the specification.

Większość aplikacji posiadających tekstowy interfejs użytkownika wymaga wsparcia dla pojęcia terminala zarówno w jądrze, jak i w przestrzeni użytkownika (w postaci bibliotek, np. `ncurses`). Dodatkowo, zarządzanie programami uruchamianymi z powłoki (np. zatrzymywanie lub przerywanie) wymaga wsparcia dla kontroli zadań. Praca przedstawia specyfikację i implementację podsystemu terminali i kontroli zadań w systemie operacyjnym Mimiker. Zarówno podsystem terminali, jak i kontroli zadań są zgodne ze specyfikacją POSIX, co umożliwia łatwe przenoszenie programów z innych systemów zgodnych z tą specyfikacją.

Contents

Chapter 1

Introduction

Text-based interfaces are a common means of interaction with a program whenever using a graphical interface is impossible, impractical, or undesired. An interactive application with a text interface outputs characters, which are then displayed to the user. User input is typed on a keyboard, which is made available to the program in the form of characters.

In the past, hardware devices called *terminals*[1] were used for text-based interaction with a computer system. They consisted of a screen and a keyboard, and were connected to the computer using a serial link. These days, hardware terminals have been replaced with *terminal emulators* — programs which run on top of a graphical environment, and display the emulated terminal's screen in a graphical window.

The Mimiker operating system lacks a graphical environment, which leaves a text-based interface as the only sensible alternative. However, the OS was lacking common features, which limited the usability of more advanced applications, such as shells. This thesis describes the implementation of two such features: *job control* and the *terminal subsystem*. Chapter **??** describes in detail the implementation of job control, while Chapter **??** presents the terminal subsystem. We will now provide an introductory explanation of these features, and how applications make use of them.

1.1 Introduction to job control

To understand job control, we first need to understand the functionality of a shell. In essence, a shell is a program used to run other programs. The user types a command, which is then interpreted by the shell and run. Most shells can perform complex commands, which involve e.g. redirecting the output of one program to be the input to another program, or running several programs in parallel. As an example, let's break down the following command:

[1] `https://en.wikipedia.org/wiki/Computer_terminal`

```
cat /etc/passwd | grep user & echo hello
```

The command runs three programs: `cat`, `grep` and `echo`, each with a single argument, respectively: `/etc/passwd`, `user` and `hello`. The output of `cat /etc/passwd` is fed to `grep user` as input thanks to the `|` ("pipe") symbol between them.

The `cat` and `grep` programs constitute a single *job*. A job is group of programs accomplishing a single logical task. In this case, the job `cat /etc/passwd | grep user` finds the lines in the `/etc/passwd` file containing the string `user`. The `echo hello` program is launched as a job containing a single program — itself. We will discuss the meaning of the `&` symbol shortly.

Job control refers to the ability of the user and the shell to monitor and control running jobs. We will now discuss the most important use cases for job control which have been made possible on Mimiker thanks to the implementation of missing functionality by the author.

Foreground and background jobs

The `&` symbol separating the two jobs in the example command makes the first job run in the *background*, and the second one in the *foreground*. Only one job can be in the foreground at any time. Background jobs are restricted from writing output and reading input from the terminal. While the foreground job is running, the shell cannot accept new commands from the user, as it is waiting for the foreground job to complete.

Stopping and resuming jobs

The foreground job may be stopped by the user by pressing a special key on the keyboard. When a job is stopped, all processes constituting it prevented from running. A background job can be put in the foreground using the `fg` shell command, which will also resume the target job if it was stopped.

Notification of job status changes

The shell is constantly monitoring the status of its jobs. When a job terminates or stops, the shell is notified and it relays that information to the user. Also, if the foreground job becomes stopped, the shell detects it and puts itself in the foreground, awaiting commands from the user.

Jobs are purely a shell concept, but they closely map to entities within the kernel called *process groups*. When a shell runs a new job, it puts all the processes in the job in the same process group, so that they can be managed as a single unit. The discussion in subsequent chapters does not mention jobs, focusing on process groups instead.

1.2 Introduction to terminals

In this section, we shall focus on the software abstraction of a terminal as specified in POSIX [2], not on hardware terminal devices. Similarly to the previous section, we list important features which have been implemented by the author.

Input and output processing

Terminals provide a layer of processing between processes performing I/O and the raw hardware device. Processing occurs on both input and output characters, though output processing is minimal compared to input processing. Arguably the most important aspect of input processing is line editing: input is assembled into lines, which can be edited before being passed on to processes. Certain parts of input processing tie into job control, e.g. stopping the foreground job (actually the foreground *process group*) in response to a special character being received.

Pseudoterminals

A *pseudoterminal* consists of a pair of devices (in the OS, not actual hardware devices): one appears to processes as a regular terminal device, and the other one is a special device which can be used by a process to *emulate* a hardware terminal device in software. They are commonly used by terminal emulators.

Ported libraries and programs

The author has ported several programs making use of advanced terminal features:

- `atto`, a minimal `emacs`-like text editor. It makes use of random cursor movement, as well as changing the foreground colour of the terminal.
- `script`, a program which records the output of a terminal session to a file. It uses pseudoterminals to accomplish this task.
- `tetris`, a popular interactive game. Like `atto`, it makes use of random cursor movement and is capable of drawing the pieces in different colours. Unlike `atto`, it makes use of the `terminfo` library, which had to be ported as well.

1.3 Introduction to the Mimiker OS

Mimiker[2] is an educational operating system project developed at the University of Wrocław. The primary area of development is the kernel, while userspace programs and libraries are usually ported from other OSes. The kernel aims to be compliant with the POSIX specification, which makes porting of applications from other POSIX-compliant systems easier. The design of the kernel is inspired primarily by the FreeBSD and NetBSD kernels, with some components influenced by Linux.

At the time of writing, the OS supports two platforms: the MIPS Malta development board, and the Raspberry Pi 3. An important thing to note is that the OS is tested only in an emulated environment using QEMU, not on real hardware. Additionally, while the CPU on the Raspberry Pi 3 has multiple cores, the Mimiker kernel currently does not support multiprocessor architectures, and therefore uses only one core on the Pi.

This thesis contains a lot of source code listings with Mimiker source code. At the time of writing, the Mimiker project provides an interactive source code browser at `https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/`. The reader is highly encouraged to explore the source code while reading through the listings.

1.3.1 Important OS concepts

It is assumed that the reader has a basic understanding of operating systems, obtained e.g. by attending a university course. The following is a non-exhaustive list of terms used throughout this thesis, which the reader is assumed to be familiar with:

- Kernel: the portion of an OS running with special privileges.
- Process: an instance of an executing program.
- Thread: the basic unit of scheduling. Note: at the time of writing, the Mimiker kernel does not support processes with more than one thread.
- System call: the mechanism used by threads to request services from the kernel.
- Interrupt: the mechanism used by hardware devices to request services from the kernel or to notify it of some event.

1.3.2 Synchronization primitives in the Mimiker kernel

An important aspect of kernel programming is proper synchronization of accesses to shared data structures. In the Mimiker kernel, this goal is accomplished using *synchronization primitives*. The source code listings found throughout this thesis

[2] `https://mimiker.ii.uni.wroc.pl/`

contain frequent references to these primitives, hence it is useful to have a basic understanding of their usage before reading the kernel's code.

The primitives used in Mimiker include *locks* (of which there are two types) and *condition variables*. In this section, we first provide a general introduction to locks, after which their usage in the Mimiker kernel is explained in detail. Finally, we explain the purpose and usage of condition variables in the kernel.

Introduction to locking

With multiple threads executing concurrently and manipulating shared data, ensuring that the data structures are always in a consistent state becomes a crucial matter. As a toy example of what can go wrong, consider a data structure with two fields bound by an invariant: the values of the two fields must always be the same.

```
struct foo {
  /* x and y must be equal */
  int x;
  int y;
}
```

Let us now imagine two threads sharing an instance of `struct foo` called `shared_foo`: one is modifying its contents, while the other is just reading it.

Thread 1

```
void thread1(void) {
  shared_foo.x++;
  shared_foo.y++;
}
```

Thread 2

```
void thread2(void) {
  int x = shared_foo.x;
  int y = shared_foo.y;
  assert(x == y);
}
```

Listing 1: Example code of two threads modifying and reading a shared data structure.

Notice that thread 1 appears to preserve the structure's invariant: if the invariant is true prior to its execution, then it will be true after thread 1 executes. Looking at thread 2, its assertion should always pass thanks to the invariant associated with the structure. However, issues start to appear as soon as thread 1 is run concurrently with thread 2. Consider the following series of events:

```
/* Thread 2: */
int x = shared_foo.x; /* value read = 0 */
/* Thread 1: */
shared_foo.x++;
shared_foo.y++;
/* Thread 2: */
int y = shared_foo.y; /* value read = 1 */
assert(x == y);       /* Boom! */
```

As we can see, thread 1 observed a state of the `shared_foo` structure which violates the invariant, because it was modified by thread 2 in the middle of thread 1 reading it. Clearly, something must be done to prevent this.

A common solution to this problem is to *synchronize* the threads to ensure that when a thread is modifying the structure, no other threads are accessing it concurrently. Synchronization between threads is accomplished using *synchronization primitives*, with *locks* being the most commonly used category of primitives.

Arguably the simplest kind of lock is a *mutex*, which stands for *mutual exclusion*. A minimal mutex provides two operations: *acquire*/*lock* and *release*/*unlock*. Between the time a thread acquires and releases a mutex, it is said to *hold* or *own* the mutex. The crucial point about mutexes is that for any mutex, **at most one thread may hold that mutex at any given time** — this is what mutual exclusion is all about. If a thread attempts to acquire a mutex that is already held by another thread, it will *block* (i.e. stop executing) waiting for the holder of the mutex to release it.

Let's try to solve our problem with concurrent accesses to `struct foo` using a mutex. Let's call it `foo_mtx`. The function used to acquire/release a mutex is `mtx_lock()`/`mtx_unlock()`.

Thread 1

```
void thread1(void) {
  mtx_lock(&foo_mtx);
  shared_foo.x++;
  shared_foo.y++;
  mtx_unlock(&foo_mtx);
}
```

Thread 2

```
void thread2(void) {
  mtx_lock(&foo_mtx);
  int x = shared_foo.x;
  int y = shared_foo.y;
  mtx_unlock(&foo_mtx);
  assert(x == y);
}
```

Listing 2: Example code of two threads modifying and reading a shared data structure protected by a mutex.

Now it's impossible for thread 1 to modify the `shared_foo` structure between thread 2 reading `shared_foo.x` and `shared_foo.y`. Likewise, thread 2 can't access the structure while thread 1 is in the middle of modifying it. The mutex ensures that all accesses to the structure are *serialized*, i.e. they happen one after another, never concurrently. As long as all accesses to the structure are made while holding `foo_mtx`, all readers are guaranteed to see a consistent state (i.e. they won't see an intermediate state, where the structure is in the middle of being modified).

Now that we have covered the basics of locking, let us move on to how locks are used in the Mimiker kernel.

Locking in the Mimiker kernel

Locking is the primary strategy used in the Mimiker kernel to ensure that shared data structures remain in a consistent state. It is common for a data type to include a lock which protects access to at least some of its members. For instance, the `tty_t` structure (see Listing 21) includes a member `t_lock` of type `mtx_t`, which is the type of mutexes in Mimiker. Holding a terminal structure's `t_lock` is required to access any of its other fields.

In the case of data structures used across many subsystems, such as processes (`proc_t`), different fields may require different locks to be held before accessing them. For this reason, it is very important to document these requirements in the source code. Here is an abridged definition of the `proc_t` structure:

```c
/*! \brief Process structure
 *
 * Field markings and the corresponding locks:
 *  (a) all_proc_mtx
 *  (@) proc_t::p_lock
 *  (g) p_pgrp->pg_lock
 *  (!) read-only access, do not modify!
 *  ($) use only from the same process/thread
 *  (*) safe to dereference from owner process
 *  When two locks are specified (see p_pgrp), either one suffices
 *  for reading, but both must be held for writing.
 *  NOTE: You can acquire the parent's p_lock while holding the child's p_lock,
 *        but not the other way around!
 */
typedef struct proc {
  mtx_t p_lock;                 /* Process lock */
  TAILQ_ENTRY(proc) p_all;      /* (a) link on all processes list */
  /* ... */
  thread_t *p_thread;           /* (@) the only thread running in this process */
  pid_t p_pid;                  /* (!) Process ID */
  cred_t p_cred;                /* (@, *) Process credentials */
  /* ... */
  TAILQ_ENTRY(proc) p_pglist; /* (g + a) link on pg_members list */
  pgrp_t *p_pgrp;               /* (@ + a) process group */
  /* ... */
  fdtab_t *p_fdtable;           /* ($) file descriptors table */
  /* ... */
} proc_t;
```

Listing 3: `sys/include/proc.h`: abridged definition of `proc_t`.

According to the locking rules for `proc_t` listed above, in order to modify a process's `p_pgrp` field, one must hold both the process's `p_lock` and the global `all_proc_mtx` lock. For reading the field, holding just one of these locks will suffice.

As a consequence of data structure fields having locking requirements, many functions in the kernel also have locking requirements, since they often access shared data.A function's locking requirements specify which locks must be held by the caller when calling the function. In Mimiker source code, locking requirements are usually given in a comment above the function's declaration. For instance, here is an excerpt from `sys/include/tty.h`:

```
/*
 * Put a single character into the tty's input queue, provided it's not full.
 * Must be called with tty->t_lock held.
 * Returns false if there's no space in the tty's input queue, true on success.
 */
bool tty_input(tty_t *tty, uint8_t c);
```

Listing 4: `sys/include/tty.h`: declaration of `tty_input()`.

Locking requirements may also be verified at runtime using assertions at the beginning of a function. Not every function does this, but it quickly catches violations of the requirements.

While it is important to understand the principles of locking in Mimiker, they are not the focus of this thesis. Therefore, subsequent listings do not contain comments with locking requirements. The interested reader may of course read the original source code, which contains these annotations (in most cases). Listings containing the source code of functions still contain locking operations and assertions, but we won't go into detail as to *why* they are there in the first place. With that in mind, let us go into the two types of locks used in the Mimiker kernel.

Mutexes and spinlocks

There are currently two types of locks used in the Mimiker kernel: *mutexes* and *spinlocks*. Their behaviour is heavily inspired by the design of corresponding locks in the FreeBSD kernel [6]. They seem similar on the surface, in that they are both mutually exclusive locks, i.e. only one thread may hold a mutex or spinlock at a time. However, they are conventionally used for different purposes, both in Mimiker and in the FreeBSD kernel:

- Mutexes are used to synchronize threads with other threads.
- Spinlocks are used to synchronize threads with interrupt service routines (ISRs), or to synchronize ISRs with other ISRs running on different CPUs.

An ISR is a very short piece of code that is run in response to a *hardware interrupt*. Interrupts are a means for devices to request attention from the CPU. For instance, a network adapter may interrupt the CPU to notify it that a new packet has arrived. It is crucial that an ISR executes as soon as possible, and takes as short as possible to complete, in order to keep up with the device feeding the data.

Mutexes are a poor fit for ISRs for several reasons, one of which is that blocking on a mutex introduces latency between one thread releasing it and the blocked thread acquiring it. Spinlocks remedy this by using *spinning*, also called *busy waiting*, instead of blocking. When a thread spins, it keeps checking the status of a lock (i.e. whether it is held by someone else or not), and it does not voluntarily stop executing, as is the case when blocking. The drawback of this approach is that when spinning, the CPU is not doing any useful work, so spinlocks should be held for very short amounts of time. Due to this requirement, acquiring a mutex while holding a spinlock is strictly forbidden.

Another significant difference between mutexes and spinlocks is a feature called *priority propagation*. In essence, it boosts the scheduling priority of the owner of a lock if a higher-priority thread is waiting for that lock. This solves the problem of *priority inversion*. For more information about priority inversion, as well as the implementation of priority propagation in the Mimiker kernel, see [28]. Table 1.1 provides a summary of the differences between mutexes and spinlocks in the Mimiker kernel.

	Mutex	**Spinlock**
Waiting method	blocking	spinning
Disables interrupts	no	yes
Can use in ISR	no	yes
Can acquire while holding mutex	yes*	yes
Can acquire while holding spinlock	no	yes*
Priority propagation	yes	no

* Subject to lock ordering constraints, see Section 4.2.

Table 1.1: Summary of differences between mutexes and spinlocks in the Mimiker kernel.

In Mimiker source code, the `mtx_t` data type represents a mutex, while `spin_t` represents a spinlock. A mutex is acquired using the `mtx_lock()` function, which takes a pointer to `mtx_t` as an argument. To release a mutex, the `mtx_unlock()` function is used, which takes a pointer to an owned mutex. For spinlocks, the corresponding functions are `spin_lock()` and `spin_unlock()`, which take a pointer to `spin_t` as an argument.

Two convenience macros are available that can make working with locks a bit easier, although one still needs to be careful. The `WITH_MTX_LOCK()` acquires a mutex and automatically releases it upon exit from the block following the macro. (delimited by `{` and `}`). Similarly, the `SCOPED_MTX_LOCK()` macro acquires a mutex and automatically releases it upon exit from the scope immediately enclosing the macro's invocation. It is commonly used to acquire a mutex for the entire duration of a function. The `WITH_SPIN_LOCK()` and `SCOPED_SPIN_LOCK()` macros do the same thing for spinlocks. Listing 5 shows example usage of these macros. Each code sample in the listing is equivalent with respect to locking.

```c
bool maybe_do_something() {
  mtx_lock(&foo_mtx);
  if (!can_do_something()) {
    mtx_unlock(&foo_mtx);
    return false;
  }
  do_something();
  mtx_unlock(&foo_mtx);
  return true;
}
```

```c
bool maybe_do_something() {
  WITH_MTX_LOCK(&foo_mtx) {
    if (!can_do_something())
      return false;
    do_something();
  }
  return true;
}
```

```c
bool maybe_do_something() {
  SCOPED_MTX_LOCK(&foo_mtx);
  if (!can_do_something())
    return false;
  do_something();
  return true;
}
```

Listing 5: Comparison between using the `mtx_lock()`/`mtx_unlock()` functions directly, using the `WITH_MTX_LOCK()` macro, and using the `SCOPED_MTX_LOCK()` macro.

Condition variables

In the Mimiker kernel, a thread may arrive at a point where it must wait for an event that may happen at some time in the future. For instance, consider multiple producer and consumer threads sharing a buffer. The producer puts data into the buffer, while consumers take data from the buffer and do something with it. Since the buffer is a data structure shared by multiple threads, it must be protected by a lock. A mutex is a good choice, since no ISRs are involved.

Now, consider what happens when a consumer tries to take some data from the buffer, but finds out that the buffer is empty. It must somehow wait for a producer to put some data into the buffer. The consumer could just continuously keep checking whether the buffer is empty, making sure to release the mutex protecting the buffer between retries, so that the producer gets a chance to put data in. This approach is very similar to the spinning performed when waiting for a spinlock to become released, in that it wastes CPU cycles doing nothing. Clearly, a more efficient alternative is needed.

Condition variables are a very useful synchronization primitive that can be used in our example. Generally speaking, they are used whenever a thread must wait for some condition to become true, hence the name. A typical condition variable provides three operations: *wait*, *signal*, and *broadcast*. The wait operation accepts a condition variable and a lock, and it atomically releases the lock and suspends execution of the thread invoking the operation. Thanks to its atomicity, when another thread subsequently acquires the same lock, the waiting thread is guaranteed to be suspended. This prevents events known as *lost wakeups* from occurring. The signal operation is used to wake up at most one thread waiting on a condition variable, while the broadcast operation wakes up all threads waiting on a condition variable. When a waiting thread is woken up, it automatically acquires the lock passed to the `cv_wait()` function.

Let's see how condition variables could be used to solve our problem with consumers waiting for data in a buffer. Assume the buffer is protected using a mutex called `buf_mtx`, and the condition variable for consumers is called `nonempty_cv`. In Mimiker, the condition variable operations are implemented by the functions `cv_wait()`, `cv_signal()`, and `cv_broadcast()`. Listing 6 provides example source code for the consumer and producer threads.

```
  void consumer(void) {
    int data;
❶   while (true) {
❷     mtx_lock(&buf_mtx);
❸     while (buf_empty())
❹       cv_wait(&nonempty_cv, &buf_mtx);
❺     data = buf_get_data();
❻     cv_broadcast(&nonfull_cv);
❼     mtx_unlock(&buf_mtx);
❽     consume_data(data);
    }
  }
```

```
void producer(void) {
  int data;
  while (true) {
    data = produce_data();
    mtx_lock(&buf_mtx);
    while (buf_full())
      cv_wait(&nonfull_cv, &buf_mtx);
    buf_put_data(data);
    cv_broadcast(&nonempty_cv);
    mtx_unlock(&buf_mtx);
  }
}
```

Listing 6: Producer and consumer threads synchronizing using mutexes and condition variables.

Let's go step-by-step through the `consumer()` function. Each consumer thread executes an ❶ infinite loop, taking a piece of data from the buffer and consuming it. Before checking whether the buffer is empty, we must ❷ acquire `buf_mtx`, since we are accessing a shared data structure. As long as ❸ the buffer is empty, we ❹ wait on the `nonempty_cv` condition variable, releasing `buf_mtx` at the same time.

Notice that line ❸ is a `while` loop, not an `if` statement! That's because if there are multiple consumers waiting on `nonempty_cv`, we may not be the first thread that gets to read data from the buffer after a `cv_broadcast()` from the producer. The consumer before us may empty the buffer, therefore we cannot assume that the buffer is not empty after being woken up.

Once we find that the buffer is not empty, we ❺ read a piece of data from the buffer. Naturally, we need to hold `buf_mtx` for that operation. Once we have read the data, we ❻ wake up any producers waiting for space to become available in the buffer. Finally, we ❼ release the mutex and ❽ process the data read from the buffer.

This section provided an introduction to synchronization primitives found in source code listings throughout this thesis. It will hopefully make the reader less confused when reading through code making use of them.

Chapter 2

Job Control

Support for job control on POSIX-compliant systems is realised by several concepts:

- *Process groups*, which allow for grouping processes that are part of a single job, together with facilities to send a signal to every process in a process group at once.
- *Sessions*, which connect all the processes that are run by a user between logging in and logging out of the system.
- *Job control signals* like `SIGSTOP` and `SIGCONT`, which allow for stopping and continuing individual processes.
- *Background and foreground process groups*, which determine the processes that are allowed to receive user input and write output to the terminal.

We will now describe each of these concepts in detail, with the exception of background and foreground process groups, which will be described in the next chapter. For each concept, we will first bring up the relevant parts of the POSIX specification, after which we will lay out its implementation in the Mimiker operating system.

2.1 Process groups and sessions

2.1.1 POSIX process groups and sessions

POSIX process groups

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

Every process belongs to exactly one process group. When a new process is created using `fork()`[18], it joins 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()`[19] function. It is used by the shell to set the process group of all processes in a job.

There is a correspondence between process IDs and process groups IDs: every newly created process group's ID is equal to the process ID of its first inhabitant, also called the *process group leader*. The leader process is not special in any way, other than the fact that its PID is equal to the PGID of the group it is in.

Process groups are useful from a job control perspective, since certain POSIX functions like `waitpid()`[25] and `kill()`[5] can operate on entire process groups. For instance, `waitpid()` allows the caller to wait for a status change of any child process in the specified process group.

POSIX 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. Sessions are collections of process groups, much in the same way as process groups are collections of processes. A notable difference is that a process group cannot change its session during its lifetime, while a process can change its process group. Like processes and process groups, sessions have numeric identifiers called session IDs, or SIDs.

A session may have an associated terminal device. That terminal device is called the session's *controlling terminal*. They will be explained in detail in the next chapter.

Processes are not confined to their session: they can separate from it by creating their own session using the `setsid()`[12] 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 SID of the newly created session is equal to the PID of the creating process.

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, since it will be in a different session. 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`.

Figure 2.1 illustrates a typical grouping of processes into process groups and sessions. Arrows indicate parent-child relationships. It can be seen that `sh`, `sshd` and `init` are session leaders. The `sshd` process is a daemon that was started by `init`. The shell (`sh`) has two active jobs, which occupy process groups 4 and 6. The two jobs were spawned using the following shell command:

```
cat /etc/passwd | grep user & echo hello
```

This runs the pipeline `cat /etc/passwd | grep user` as a background job, and starts the job `echo hello` without waiting for the background job to finish.

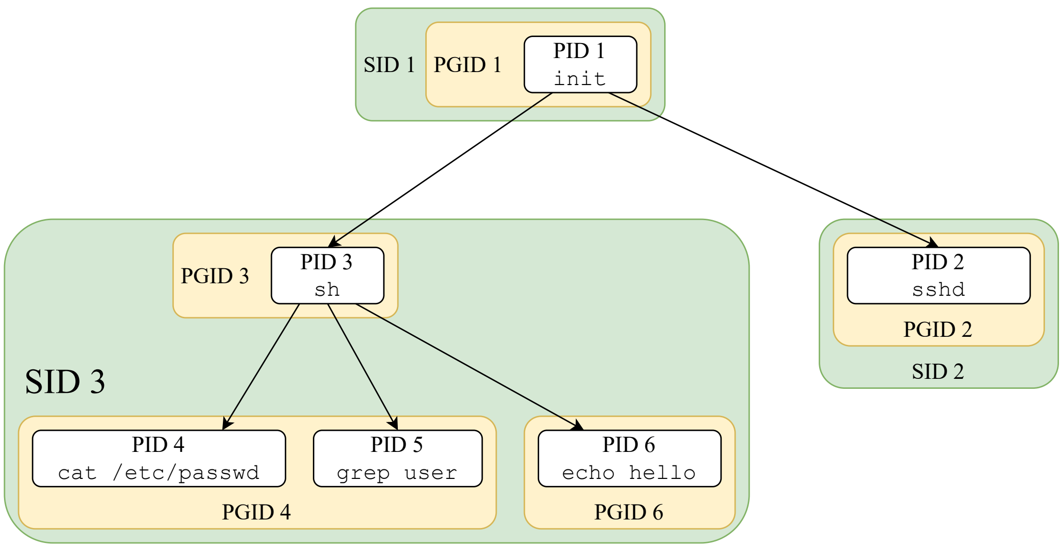

Figure 2.1: A typical process hierarchy.

Orphaned process groups

As we have just explained, processes belonging to the same shell job are put inside the same process group. The shell is responsible for managing jobs, which means keeping track of changes in their state (a job can become stopped when the user inputs a special character), as well as manipulating them according to the user's commands. However, when a shell exits, perhaps due to a bug, there might no longer be a process managing these jobs. Specifically, there might no longer be a process that can continue stopped jobs. Those jobs are doomed to being stopped forever!

This is where the concept of *orphaned process groups* comes into play. A process group is orphaned when none of the processes in the group have a parent that is in the same session, but in a different process group.

Consider the example of a shell: all jobs are in a different process group, but in the same session as the shell. Therefore, as long the shell is alive, the process groups of the jobs are not orphaned. However, when the shell process terminates, all children of the shell are *reparented*, i.e. some other process becomes their parent, usually the *init* process with PID 1. The new parent is usually in a different session, so after reparenting the process groups of the jobs are orphaned.

To solve the problem of jobs being stopped forever, when a process group becomes orphaned, if the process group contains at least one stopped process, then every process in the group is sent a `SIGHUP` signal followed by a `SIGCONT`. The `SIGCONT` signal will resume stopped processes, and the `SIGHUP` signal notifies the processes that they are now orphaned (it will also most likely kill the processes, as that is the signal's default action).

Furthermore, processes in orphaned process groups cannot be stopped by *terminal stop signals*, i.e. `SIGTSTP`, `SIGTTOU` and `SIGTTIN`. These signal are explained in Section 2.2.2.

2.1.2 Process groups in the Mimiker kernel

We will now describe the implementation of process groups in the Mimiker kernel. First, we lay out the data structures used. After that, we will go over how processes enter and exit process groups.

Data structures

`pgrp_t`

The `pgrp_t` structure represents a single process group.

```
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 7: `include/sys/proc.h`: definition of `pgrp_t`.

The `pg_lock` mutex synchronizes concurrent accesses to the list of members. The `pg_hash` field is a list entry used to link the structure into the global hashtable used to lookup process groups by PGID.

All processes that are members of the process group are on the `pg_members` list. The list allows easy access, e.g. when a signal needs to be sent to all members of the group.

The `TAILQ_ENTRY` and `TAILQ_HEAD` macros are part of Mimiker's infrastructure for creating and manipulating linked list-like structures, a *tail queue* (which is what `TAILQ` stands for) being one example. The infrastructure has been imported from FreeBSD, so the FreeBSD manual page [4] provides a good overview of its usage.

`pg_session` is a pointer to the `session_t` structure representing the session that this process group is a part of. Every process group is a part of some session.

The `pg_jobc` field is a counter that tracks how many processes *qualify the group for job control*. We say that a process qualifies the group for job control if and only if its parent is in a different process group and in the same session. This way, when `pg_jobc` drops to zero, we know that the process group has become orphaned.

The `pg_id` field is simply the numeric ID of the process group.

`proc_t::p_pgrp`

The `p_pgrp` field of the process descriptor structure is a pointer to the group that the process is a member of.

Changing process groups

Listing 8 shows kernel code implementing the POSIX `setpgid()` function. `p` is the process that is performing the system call, `target` is the PID of the process whose process group is to be changed, and `pgid` is the PGID of the process group to which the target process is to be moved.

```
int pgrp_enter(proc_t *p, pid_t target, pgid_t pgid) {
❶ SCOPED_MTX_LOCK(all_proc_mtx);
  proc_t *targetp = proc_find_raw(target);

❷ if (targetp == NULL || !proc_is_alive(targetp) ||
      (targetp != p && targetp->p_parent != p))
    return ESRCH;
❸ if (targetp == targetp->p_pgrp->pg_session->s_leader)
    return EPERM;
❹ if (targetp->p_pgrp->pg_session != p->p_pgrp->pg_session)
    return EPERM;

  pgrp_t *pg = pgrp_lookup(pgid);

  /* Create new group if one does not exist. */
❺ if (pg == NULL) {
    /* New pgrp can only be created with PGID = PID of target process. */
    if (pgid != target)
      return EPERM;
    pg = pgrp_create(pgid);
    pg->pg_session = p->p_pgrp->pg_session;
    session_hold(pg->pg_session);
❻ } else if (pg->pg_session != p->p_pgrp->pg_session) {
    /* Target process group must be in the same session
     * as the calling process. */
    return EPERM;
  }

❼ return _pgrp_enter(targetp, pg);
}
```

Listing 8: `sys/kern/proc.c`: definition of `pgrp_enter()`.

First, we ❶ acquire the `all_proc_mtx`, which synchronizes accesses to process-tree structures, such as process groups and sessions. Next, lines ❷, ❸ and ❹ respectively perform the following checks:

- The target process must exist, be alive (i.e. executing normally or stopped), and it must either be a child of the calling process, or be the calling process itself.
- The target process must not be a session leader.
- The target process must be in the same session as the calling process.

We then ❺ check whether the target process group exists. If it doesn't, it is created, but only if the requested PGID matches the PID of the target process. At ❻ we perform yet another permission check. Finally, we call `_pgrp_enter()` to perform the actual work of changing the process group of the target process.

Now, let's see how a process group switch actually happens.

```
static int _pgrp_enter(proc_t *p, pgrp_t *target) {
  pgrp_t *old_pgrp = p->p_pgrp;

❶ if (old_pgrp == target)
    return 0;

❷ pgrp_jobc_enter(p, target);
  pgrp_jobc_leave(p, old_pgrp);

❸ WITH_MTX_LOCK(&old_pgrp->pg_lock) {
    WITH_MTX_LOCK(&target->pg_lock) {
      WITH_PROC_LOCK(p) {
❹       TAILQ_REMOVE(&old_pgrp->pg_members, p, p_pglist);
        TAILQ_INSERT_HEAD(&target->pg_members, p, p_pglist);
        p->p_pgrp = target;
      }
    }
  }

❺ if (TAILQ_EMPTY(&old_pgrp->pg_members))
    pgrp_remove(old_pgrp);

  return 0;
}
```

Listing 9: `sys/kern/proc.c`: definition of `_pgrp_enter()`.

We first ❶ check whether we need to change groups at all. If we do, we ❷ adjust the `pg_jobc` counters of the old group, target group, as well as the process groups of all children of the process. We will examine these functions in a minute.

The process group switch must appear atomic. For this reason, it is necessary to hold `all_proc_mtx`, both the old group and target group's lock, and the process lock of the target process. We acquire the necessary locks at ❸. `all_proc_mtx` is not acquired, since it is already held by the caller of `_pgrp_enter()`.

When acquiring two locks of the same type (in this case process group locks), one has to be very careful not to cause a deadlock. The usual way to ensure safety from deadlocks is to establish an ordering on the locks, and whenever multiple locks need to be acquired, make sure all required locks are acquired according to that ordering. However, in this case deadlock can be avoided without specifying an order on process group locks, by enforcing the following rule:

> In order to acquire multiple process group locks, a thread must already hold the `all_proc_mtx` lock.

With this rule in force, it is impossible for two threads to concurrently attempt to acquire multiple process group locks, since `all_proc_mtx` will act as a serializer between them. The rule is a natural fit in this particular situation for two reasons:

- `_pgrp_enter()` is the only place in the whole kernel where there is a need to acquire multiple process group locks.
- The `all_proc_mtx` lock is already held everywhere `_pgrp_enter()` is called. Therefore, virtually no code changes had to be made in order to accomodate this rule.

At ❹, we are finally ready to make the switch. We remove the process from the old group's list of members, add it to the target group's list, and change the `p_pgrp` pointer. For more details about the `TAILQ_*` macros, see [4].

At the end, we ❺ check whether the process was the last remaining process in the old group. If so, the process group is removed.

Orphaned process groups

The `pg_jobc` counters need to be adjusted whenever a process leaves or enters a process group. However, it is not sufficient to adjust only the counter of the process group being entered or left. When a process leaves a process group, the children of the process may no longer qualify their process group for job control.

Figure 2.2 illustrates a scenario in which a call to `setpgid()` causes the process group of a child of the calling process to become orphaned. For this reason, it is necessary to check whether the children still qualify their process groups for job control whenever leaving or entering a process group.

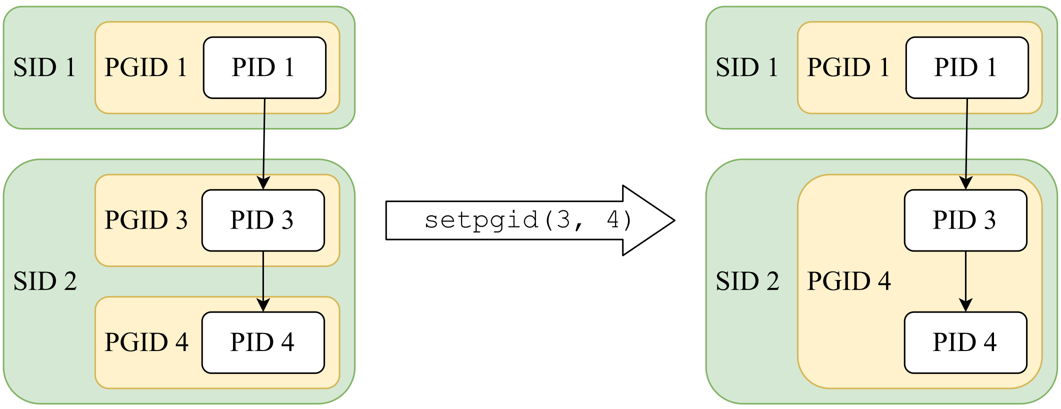

Figure 2.2: A parent's change of process group causes a child's process group to become orphaned.

The accounting associated with pg_jobc is split into two functions: pgrp_jobc_enter() and pgrp_jobc_leave().

```
static void pgrp_jobc_enter(proc_t *p, pgrp_t *pg) {
  assert(mtx_owned(all_proc_mtx));

❶ if (same_session_p(p->p_parent->p_pgrp, pg))
    pg->pg_jobc++;

  proc_t *child;
❷ TAILQ_FOREACH (child, CHILDREN(p), p_child)
    if (same_session_p(child->p_pgrp, pg))
      child->p_pgrp->pg_jobc++;
}
```

Listing 10: sys/kern/proc.c: definition of pgrp_jobc_enter().

p is the process entering the process group pg. The same_session_p() function returns true if and only if the two groups are different, but in the same session.

At ❶, we check whether p will qualify the process group pg for job control after entering it. If so, the pg_jobc counter is incremented.

Next, at ❷ we do the same for the children of p: we check whether they will qualify their process groups after p enters pg. If so, the group's pg_jobc is incremented.

The pgrp_jobc_leave() function has a very similar structure, except now we call pgrp_maybe_orphan() instead of incrementing pg_jobc. The pgrp_maybe_orphan() function decrements the process group's pg_jobc, and if it reaches zero, sends the appropriate signals.

```
static void pgrp_jobc_leave(proc_t *p, pgrp_t *pg) {
  assert(mtx_owned(all_proc_mtx));

  if (same_session_p(p->p_parent->p_pgrp, pg))
    pgrp_maybe_orphan(pg);

  proc_t *child;
  TAILQ_FOREACH (child, CHILDREN(p), p_child)
    if (same_session_p(child->p_pgrp, pg))
      pgrp_maybe_orphan(child->p_pgrp);
}
```

Listing 11: sys/kern/proc.c: definition of pgrp_jobc_leave().

2.1.3 Sessions in the Mimiker kernel

We will now examine the implementation of sessions in the Mimiker kernel. After looking at the session data structure, we will see how new sessions are created.

Data structures

The `session_t` data structure represents a single session. Listing 12 presents its complete definition.

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

Listing 12: `include/sys/proc.h`: definition of `session_t`.

The `s_hash` field is used to link the structure into the hashtable used to look up session strucutures by their session ID (SID).

The `s_leader` field points to the process that is the session leader, i.e. the process that created the session by calling the `setsid()` function. Once the session leader terminates, the `s_leader` field of the session is set to `NULL`.

The `s_count` field counts the number of process groups belonging to the session. Whenever a new process group joins the session, the `session_hold()` function increments the `s_count` field (see Listing 8 for an example of its usage). When a process group is removed, the `pgrp_remove()` function, which can be seen used in Listing 11 decrements the session's `s_count` field. Once the value of the field reaches zero, the session is removed from the kernel and the memory occupied by the structure is reclaimed.

The `s_sid` field holds the numeric ID of the session. It is equal to the PID of the process that created the session.

The `s_tty` field is a pointer to the session's controlling terminal. For interactive sessions (i.e. sessions created by a user logging into the system and launching a shell), the controlling terminal is the device that provides input to the shell and spawned jobs, and receives their output. Not every session has a controlling terminal: daemon processes are usually placed in sessions without a controlling terminal. If a session has no controlling terminal, the value of its `s_tty` field is set to `NULL`.

Creating a session

User processes can create new sessions using the `setsid()` function from the C library. That function directly calls the `setsid()` system call, which is implemented by the `sys_setsid()` function in the kernel. That function, in turn, calls `session_enter()` to do the work. Listing 13 presents the source code of `session_enter()`.

```
int session_enter(proc_t *p) {
❶  SCOPED_MTX_LOCK(&all_proc_mtx);

   pgid_t pgid = p->p_pid;
   pgrp_t *pg = pgrp_lookup(pgid);

❷  if (pg)
     return EPERM;

❸  pg = pgrp_create(pgid);
❹  pg->pg_session = session_create(p);

❺  return _pgrp_enter(p, pg);
}
```

Listing 13: `sys/kern/proc.c`: definition of `session_enter()`.

First, we ❶ acquire the `all_proc_mtx` lock, which is required to perform things such as process group lookups and creating new sessions and process groups.

At ❷, we ensure that there doesn't already exist a process group with the same PGID as the PID of the calling process. The process group that is created as a result of creating a new session has a PGID equal to the PID of the calling process, and if a group with such a PGID already exists, we can't create the group we need, since PGIDs must be unique.

Once we have ensured that we can create the group, we do so at ❸. Then, we ❹ create a new session with `p` as the session leader and link it to the newly created group (the session created by `session_create()` has a `s_count` of 1, so there's no need to adjust it). Finally, we ❺ change the process group of the calling process to the one we just created.

2.2 Job Control Signals

2.2.1 POSIX signals

Signals are used to notify processes of various events. These events can occur *synchronously* or *asynchronously* with respect to the process receiving the signal. POSIX signal semantics are (intentionally) very similar to those found in the original UNIX operating system [26, Section 7.2].

A *synchronous signal* is sent as a direct consequence of some (usually erroneous) action being performed by the receiving process. For instance, a process is sent a `SIGSEGV` signal upon trying to access an invalid memory location.

An *asynchronous signal* is sent independently of the actions of the receiving process, and may be received at any time. For example, whenever a process terminates, the system sends a `SIGCHLD` signal to its parent.

Signals can be sent by the operating system in response to certain events (e.g. process termination), or by other processes, using the `kill()` function [5]. A process can send a signal to itself using the `raise()` function. As a shorthand, we shall say that a signal is sent to a process group when it's sent to every process that is a member of the group.

The type of signal (`SIGSEGV`, `SIGCHLD`, etc.) is determined by the *signal number*. In fact, `SIGSEGV` and others are *C preprocessor macros* that expand to unique signal numbers.

Processes can take different actions in response to signals with different numbers. Every process has its own set of *signal actions* associated with every signal number. The action associated with a signal is also called its *disposition*. There are three possible actions that can be taken in response to a signal:

1. Take the default action for that signal number.

   Every signal number has an associated default action. For instance, the default action of the `SIGSEGV` signal is to immediately terminate the receiving process.

2. Ignore the signal.

3. Invoke a *signal handler routine*.

   The handler routine is run in the context of the process that receiving process. After the handler finishes execution, the process is resumed.

The mapping of signal numbers to signal actions is controlled using the `sigaction()` function [13]. Not every signal can have its action modified: `SIGKILL` and `SIGSTOP` cannot have their actions changed from the default one, which is to terminate or stop the receiving process, respectively.

The *delivery* of a signal occurs when the target (i.e. receiving) process takes the action associated with the signal. A signal may be delivered long after it is initially sent, or it may not be delivered at all, even if it isn't ignored. This is because a process may *block* a set of signals from being delivered. A blocked signal cannot be delivered until it is unblocked. Contrary to signals that are ignored, blocked signals await delivery instead of being discarded.

The *signal mask* determines the set of blocked signals for a thread. It can be examined and modified using the `sigprocmask()` function [14]. Unsurprisingly, the `SIGKILL` and `SIGSTOP` signals cannot be blocked from being delivered.

2.2.2 Signals used for job control

The following signals are most commonly used to control jobs on POSIX-compliant operating systems:

- `SIGINT`

  Sent by the operating system in response to the special character `VINTR` (which commonly corresponds to `^C`, i.e. Control-C) being received on the terminal. Its purpose is to signal interruption by the user. A process that receives this signal is usually expected to terminate shortly. The default action associated with this signal is to terminate the receiving process.

- `SIGQUIT`

  Similar to `SIGINT`, except it is sent in response to the `VQUIT` character (usually `^\`), and the default action additionally generates a core dump of the receiving process.

- `SIGTTOU`

  Sent by the operating system whenever a background job attempts to write to the terminal, provided the `TOSTOP` terminal flag is set (we will say more about terminal flags later). The default action associated with this signal is to stop the receiving process.

- `SIGTTIN`

  Sent by the operating system whenever a background job attempts to read from the terminal. Background jobs may not read from the terminal, regardless of the terminal settings. The default action associated with this signal is to stop the receiving process.

- `SIGTSTP`

  Sent by the operating system in response to the special character `VSUSP` (usually `^Z`) being received on the terminal. Its purpose is to stop the foreground job. Well-behaving processes should not ignore this signal. Many programs need to do some cleanup before stopping: in that case, they register a handler that does the necessary cleanup, after which it performs `raise(SIGSTOP)`. The default action associated with this signal is to stop the receiving process.

- `SIGSTOP`

  This signal is not sent by the operating system. It unconditionally stops the receiving process. It cannot be blocked or ignored.

- SIGCONT

  This is the only signal that can resume a stopped process (apart from SIGKILL, which resumes it only to immediately terminate it). It is usually sent by the shell, e.g. when a background job that was stopped by a SIGTTIN signal is brought into the foreground.

- SIGCHLD

  Sent by the operating system in response to the termination of a process. The signal is sent to the parent of the terminating process. Shells use this signal to update their data on currently running jobs.

2.2.3 Signals in the Mimiker kernel

In this subsection we describe the implementation of signals in the Mimiker kernel. First, we lay out the data structures used, after which we go through how exactly signals are sent and delivered in the kernel. Lastly, we examine how processes are stopped in response to the delivery of a stop signal.

Signal data structures

We will now describe the various data types and structures that are critical to the implementation of signals in the Mimiker kernel.

`sigaction_t`

The `sigaction_t` data type describes the disposition of a signal.

```c
typedef void (*sig_t)(int); /* type of signal function */

typedef struct sigaction {
  union {
    sig_t sa_handler;
    void (*sa_sigaction)(int, siginfo_t *, void *);
  };
  sigset_t sa_mask;
  int sa_flags;
} sigaction_t;
```

Listing 14: `include/sys/signal.h`: definition of `sigaction_t`.

The `sa_handler` and `sa_sigaction` fields are simply pointers to a signal handler function. Processes can register either a handler of type `sig_t`, which takes only the signal number as an argument, or a handler that takes two additional arguments:

- A pointer to a structure of type `siginfo_t`, which contains more information about the signal (e.g. in the case of `SIGCHLD`, the PID of the child process).

- A pointer that can be cast to a pointer to a structure of type `ucontext_t`, which holds the processor context of the thread that received the signal at the time of the signal's delivery.

The `sa_handler` field can also have the special value `SIG_DFL` or `SIG_IGN`, which respectively mean that the signal has the default disposition or is ignored.

The `sa_mask` field is the set of signals that are blocked during the execution of the handler function. After the handler finishes, the signal mask is restored to its previous state.

The `sa_flags` field is a set of flags that modify the behaviour of the signal in various ways. At the time of writing, the only flag supported in the Mimiker kernel is `SA_RESTART`, which causes system calls that are interrupted by a signal handler to be automatically restarted, transparently to the process that issued the system call.

`proc_t::p_sigactions`

The `p_sigactions` field of the `proc_t` (i.e. process descriptor) structure is an array of structures of type `sigaction_t`. For each signal number `signo`, the disposition of that signal for the process is stored in `p_sigactions[signo]`.

`thread_t::td_sigmask`

As opposed to the signal disposition, which is shared by all the threads of a process, every thread has its own set of blocked signals. This set is stored in the `td_sigmask` field of the `thread_t` structure, which describes a single thread of execution. The field is just a bit vector, with each bit corresponding to a signal number. If a bit is set, signals with the corresponding number are blocked from being delivered.

`thread_t::td_sigpend`

The `td_sigpend` field of the `thread_t` structure represents pending signals, i.e. signals that have been sent to the thread and are waiting to be delivered. It is a structure of type `sigpend_t`, which consists of a bit vector of pending signal numbers, as well as a list of `ksiginfo_t` structures which carry additional information about signals.

Sending a signal

We will now walk through the code that does the actual work of sending a signal to a process. Sending signals in the Mimiker kernel is accomplished using the `sig_kill()` function. Listing 15 contains slightly simplified code of the function.

```
void sig_kill(proc_t *p, ksiginfo_t *ksi) {
❶ assert(mtx_owned(&p->p_lock));

  signo_t sig = ksi->ksi_signo;
  thread_t *td = p->p_thread;
  bool ignored = sig_ignored(p->p_sigactions, sig);

❷ if ((ignored && !sigprop_cont(sig)) || sig_ignore_ttystop(p, sig))
    return;

  /* If sending a stop or continue signal,
   * remove pending signals with the opposite effect. */
❸ if (defact_stop(sig)) {
    sigpend_get(&td->td_sigpend, SIGCONT, NULL);
  } else if (sigprop_cont(sig)) {
    sigpend_delete_set(&td->td_sigpend, &stopmask);
❹   if (p->p_state == PS_STOPPED)
      proc_continue(p);
    if (ignored)
      return;
  }

❺ sigpend_put(&td->td_sigpend, ksiginfo_copy(ksi));

❻ if (__sigismember(&td->td_sigmask, sig))
    return;

  WITH_SPIN_LOCK (td->td_lock) {
❼   td->td_flags |= TDF_NEEDSIGCHK;
    if (td_is_interruptible(td)) {
      spin_unlock(td->td_lock);
      sleepq_abort(td); /* Locks & unlocks td_lock */
      spin_lock(td->td_lock);
    }
  }
}
```

Listing 15: `sys/kern/signal.c`: definition of `sig_kill()`.

The function accepts two parameters. The first is a pointer to a `proc_t` structure, which represents the process that the signal will be sent to. The second is a pointer to a `ksiginfo_t` structure, which describes the signal to be sent. Most importantly, it contains the signal number.

The assertion at ❶ ensures that the function's caller has acquired the necessary lock. Processes' and threads' signal data structures are globally shared (i.e. many threads of execution can access them), so access to them must be synchronized using locks.

We then ❷ quickly filter out ignored signals, with an exception for signals that can continue stopped processes, as some processing needs to be done even if they are ignored. Furthermore, terminal stop signals (`SIGTSTP`, `SIGTTOU` and `SIGTTIN`) cannot stop a process which is in an orphaned process group: this case is detected by the `sig_ignore_ttystop()` function.

In ❸, we check whether the signal being sent is a stop or continue signal. If it is, we remove all signals that are already pending which have the opposite effect. Additionally, for continue signals ❹ we check whether the target process is stopped. If it is, we wake it up and notify its parent.

Once control reaches ❺, we are certain that the signal isn't ignored, so it should be queued for delivery to the target process. The `sigpend_put` function inserts the signal into the set of pending signals for the target process's thread.

The final step is notifying the thread that it needs to process its pending signals. This step is skipped if ❻ the signal is blocked from being delivered. The `TDF_NEEDSIGCHK` flag lets the thread know that it should check for pending signals at the nearest opportunity. If the thread is *sleeping interruptibly* (e.g. blocked inside a `read()` call on a terminal device, awaiting user input), it is awakened so that it can receive the signal.

Signal delivery

After a signal is sent, it remains in the pending set, waiting to be delivered. Signals are delivered to a process whenever control transfers from the kernel to that process, i.e. on transitions from the kernel to userspace. Every time before returning to userspace, the `on_user_exc_leave()` function is called. This function handles signal delivery and system call restarting.

```
void on_user_exc_leave(mcontext_t *ctx, syscall_result_t *result) {
  thread_t *td = thread_self();
  proc_t *p = td->td_proc;
  int sig = 0;
  ksiginfo_t ksi;

❶ if (td->td_flags & TDF_NEEDSIGCHK) {
    WITH_PROC_LOCK(p) {
❷     sig = sig_check(td, &ksi);
    }
  }

  if (result)
❸   set_syscall_retval(ctx, result, sig);

❹ while (sig) {
    WITH_PROC_LOCK(p) {
❺     sig_post(&ksi);
❻     sig = sig_check(td, &ksi);
    }
  }
}
```

Listing 16: `sys/kern/exception.c`: definition of `on_user_exc_leave()`.

Signal delivery can be divided into two steps: checking for a pending signal, and *posting* a signal found to be pending. Posting a signal means setting up the execution context of the process, so that the signal handler is the first thing that is executed after returning control to the process.

The first step is therefore to check for pending signals. The `TDF_NEEDSIGCHK` flag indicates that a signal is pending. It is set in `sig_kill()`, see Listing 15. We check the flag at ❶ to avoid unnecessarily acquiring the process lock and checking the set of pending signals. If the flag is set, we ❷ call `sig_check()`, which returns the signal number of a signal that needs to be posted, or `0` if no signal needs posting. Note that signals with special effects (i.e. stopping or killing a process) are not posted, but are instead handled inside `sig_check()`, as part of checking for a pending signal. This is an implementation choice that simplifies handling of certain edge cases.

Before posting pending signals, we ❸ set the return value of the system call we're returning from using `set_syscall_retval()` (provided we're returning from a system call at all). Note that `on_user_exc_leave()` is also called when returning to userspace from an interrupt, in which case `result` is set `NULL` to indicate that we're not returning from a system call. The `set_syscall_retval()` function also restarts the system call if needed. More details about system call restarting are given in Section 4.1.

The last step is to post all pending signals. We ❹ loop as long as there is a signal that needs posting. Inside the loop, the currently selected signal is ❺ posted and ❻ another signal is selected. We will not describe how signals are posted, as most of the details are architecture-specific, and we are primarily concerned with job control signals, which are not posted at all (unless they have registered handlers, in which case they don't behave like job control signals).

The control flow inside `on_user_exc_leave()` may seem confusing, but there is a reason for it. The main issue is that `set_syscall_retval()` must know whether a signal will be posted (note `sig` being passed as an argument to it), but it must also be called *before* `sig_post()`. The reason is that `set_syscall_retval()` modifies the context of the userspace thread that invoked the system call, and `sig_post()` also modifies the context to arrange for the signal handler to be called. The modifications made by `set_syscall_retval()` must precede the ones made by `sig_post()`, hence the somewhat convoluted control flow.

Let us now focus on the `sig_check()` function, which checks for pending signals and handles job control signals. Listing 17 contains the source code of the function.

```
int sig_check(thread_t *td, ksiginfo_t *out) {
  proc_t *p = td->td_proc;
  signo_t sig;

❶ assert(mtx_owned(&p->p_lock));

❷ while ((sig = sig_pending(td))) {
❸   sigpend_get(&td->td_sigpend, sig, out);

❹   if (sig_should_stop(p->p_sigactions, sig)) {
      if (!sig_ignore_ttystop(p, sig))
        proc_stop(sig);
      continue;
    }

❺   if (sig_should_kill(p->p_sigactions, sig))
      sig_exit(td, sig);

❻   return sig;
  }

  WITH_SPIN_LOCK (td->td_lock)
❼   td->td_flags &= ~TDF_NEEDSIGCHK;
❽ return 0;
}
```

Listing 17: `sys/kern/signal.c`: definition of `sig_check()`.

This function manipulates signal state, which is protected by the process's lock, hence the assertion at ❶.

The function first ❷ extracts a pending signal that is not currently blocked using the `sig_pending()` function. It returns just a signal number, so in the next step ❸ we remove the signal from the set of pending signals.

We then check ❹ if the signal should stop the receiving process. Notice that after the process is stopped using `proc_stop()` and subsequently resumed by another process, control goes back to ❷.

Next, we ❺ handle signals that should kill the process. The `sig_exit()` function terminates the calling process and never returns.

If the pending signal is neither a stop nor a kill signal, ❻ the signal number is returned. Additional information about the signal is passed to the caller via the `out` output parameter. The signal is then passed to `sig_post()` to arrange for the handler to be called.

If no signal needs to be posted, the function ❼ clears the thread's `TDF_NEEDSIGCHK` flag, since there is no need to check for pending signals anymore. A return value of 0 ❽ indicates to the caller that no signal needs to be posted.

Stopping and continuing processes

One of the job control features that were implemented from scratch as part of the implementation effort described in this thesis is support for stopping and continuing processes by means of the `SIGSTOP` and `SIGCONT` signals.

A process is always in one of several states, denoted by the value of the `p_state` field in the process descriptor. When a process is executing normally, its state is `PS_NORMAL`. When a process is stopped, its state is `PS_STOPPED`. All the threads of a stopped process are also stopped and unable to run until the process is continued.

The `proc_stop()` function stops the current process in response to a stop signal. Its source code is listed in Listing 18.

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

❶ assert(mtx_owned(&p->p_lock));
  assert(p->p_state == PS_NORMAL);

❷ p->p_state = PS_STOPPED;
❸ p->p_stopsig = sig;
  p->p_flags |= PF_STATE_CHANGED;
  WITH_PROC_LOCK(p->p_parent) {
❹   proc_wakeup_parent(p->p_parent);
    sig_child(p, CLD_STOPPED);
  }
❺ 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);
❻ if (td->td_flags & TDF_STOPPING) {
    td->td_flags &= ~TDF_STOPPING;
    td->td_state = TDS_STOPPED;
❼   sched_switch(); /* Releases td_lock. */
  } else {
    spin_unlock(td->td_lock);
  }
  proc_lock(p);
  return;
}
```

Listing 18: `sys/kern/proc.c`: definition of `proc_stop()`.

This function manipulates process state, hence the assertion at ❶. The next assertion simply makes sure that the process is in the expected state. Next, at ❷ the process state is set to `PS_STOPPED`.

At ❸, we set up information that is used by the `wait4()` system call. It is used by a process to wait for one of its children to change state. As the reader might have already guessed, stopping counts as a state change. The call to `proc_wakeup_parent()` at ❹ notifies the parent process of the status change. In the next line, a `SIGCHLD` signal is sent to the parent.

The rest of the function attempts to stop the process's thread. This task is fairly simple, due to the fact that all processes in the Mimiker kernel are single-threaded. Still, it is not as simple as one might like, due to some technicalities around locking.

Specifically, we are not allowed to hold the thread's spinlock (`td->td_lock`) while releasing a mutex. Furthermore, we must release the process's lock before stopping the thread in `sched_switch()`, and the thread's spinlock must be held when calling sched_switch(). These constraints require us to briefly hold no locks at all. During this window of time, another process might continue the process we are trying to stop, in which case we should not stop the thread.

The solution is to add the TDF_STOPPING thread flag, which signals that the thread is about to stop, but hasn't stopped yet. It is set ❺ while still holding the process's lock. If the process is continued before the thread is stopped, the TDF_STOPPING flag is cleared. The stopping thread examines ❻ the flag before changing its state. Thanks to this, the thread will stop only if the process is still stopped.

After setting the thread state to TDS_STOPPED, the call ❼ to sched_switch() hands over control to the scheduler, which will select another thread to run. The stopped thread will not be selected by the scheduler to run until it is continued.

Let us now see how stopped processes are woken up. Continuing a process is simpler than stopping one, as can be seen by looking at Listing 19.

```
  void proc_continue(proc_t *p) {
    thread_t *td = p->p_thread;

❶   assert(mtx_owned(&p->p_lock));
    assert(p->p_state == PS_STOPPED);

❷   p->p_state = PS_NORMAL;
❸   p->p_flags |= PF_STATE_CHANGED;
    WITH_PROC_LOCK(p->p_parent) {
      proc_wakeup_parent(p->p_parent);
    }
❹   WITH_SPIN_LOCK (td->td_lock) { thread_continue(td); }
  }
```

Listing 19: sys/kern/proc.c: definition of proc_continue().

The assertion at ❶ should come as no surprise at this point, as we are modifying the process's state. The next assertion ensures that we only attempt to continue processes that are actually stopped.

We then ❷ restore the process's p_state to PS_NORMAL. Next, we ❸ notify the parent of the state change. Note that, in contrast to proc_stop(), we don't send a SIGCHLD signal to the parent process. This is in line with the POSIX specification.

Lastly, we ❹ wake up the thread of the process we are continuing. The thread_continue() function is very simple, and is presented in Listing 20.

```
void thread_continue(thread_t *td) {
❶ if (td->td_flags & TDF_STOPPING) {
    td->td_flags &= ~TDF_STOPPING;
  } else {
❷   assert(td_is_stopped(td));
❸   sched_wakeup(td, 0);
  }
}
```

Listing 20: `sys/kern/thread.c`: definition of `thread_continue()`.

An important thing to notice is that even if a process's state indicates that it is stopped (i.e. `p_state == PS_STOPPED`), its thread is not guaranteed to also be stopped (i.e. `td_state == TDS_STOPPED`). The call to `proc_continue()` can occur at the time in `proc_stop()` where we are not holding any locks.

For this reason, we first ❶ check the `TDF_STOPPING` flag. If it is set, the thread has not stopped yet, and all we need to do is clear the flag. If the flag is not set, then we know that the thread is stopped, hence the assertion at ❷. We then ❸ call `sched_wakeup()` to make the thread runnable again.

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 3

The Terminal Subsystem

The implementation of the terminal subsystem in the Mimiker kernel consists of three primary components:

- The device-independent terminal layer.
- The UART terminal driver.
- The *pseudoterminal* subsystem, which includes another terminal driver.

In this chapter, we will describe each component in detail. Before describing the implementation, we will present relevant parts of the POSIX specification.

This chapter uses concepts such as *terminals*, *terminal devices*, and *terminal drivers*. Their meanings are as follows:

- A *terminal* does **not** refer to a hardware device like a teletype. Instead, it is an abstract representation of such a device in the kernel, not tied to a specific hardware model. In fact, it may not even be "backed" by any actual hardware device, thanks to pseudoterminals.
- A *terminal device* is the hardware device that can be represented by an abstract terminal. A UART is an example of a terminal device.
- A *terminal driver* acts as the "glue" between the terminal layer (which is device-independent) and a specific hardware device.

3.1 The terminal layer

The terminal layer provides a layer of processing between processes and the terminal device. This processing might seem unnecessary at first, but its usefulness very quickly becomes apparent.

For example, on UNIX-like systems a long-running command can be interrupted by pressing the Control-C combination of keys on the keyboard. If there was no processing done on incoming characters by the operating system, the process in question would simply receive the character as input, and it would be up to the process to respond to it. The process would need to be prepared to receive such characters at any time. This is clearly an inconvenience, especially for simple programs.

With the processing done by the operating system, the behavior is uniform and requires less effort from writers of userspace programs. The interruption is delivered as a signal, for which a handler can be easily installed.

The preceding example illustrates just one useful feature of terminals. The POSIX specification includes the concept of terminals, and describes how incoming and outgoing characters should be processed by the terminal subsystem. We will now present the most important aspects of this specification.

A very detailed description of the terminal interfaces available in several UNIX-like operating systems, which goes well beyond POSIX, is presented in [29, Chapter 18]. For a Linux-specific discussion, see [27, Chapter 62].

3.1.1 POSIX terminals

On the surface, a terminal is no different than a standard character device. To interact with it, a process can open the corresponding device file using `open()` and then perform input and output using `read()` and `write()` respectively. The terminal interface specified by POSIX is much richer than that. Terminals play a significant role in job control, and therefore are connected to the concepts of process groups and sessions. We will now examine the most important aspects of the terminal specification [2] found in POSIX and see how terminals tie into job control.

Terminal input and output

Data coming in from the hardware terminal device is processed according to the terminal mode and flags, which will be described in a subsequent section. After processing, it is buffered in the *input queue*. Processes calling `read()` on a terminal file descriptor receive data from the input queue.

When a process writes data to a terminal file descriptor using `write()`, after processing, the data may be written directly to the underlying terminal device, or it may be buffered by the OS in the terminal's *output queue* and output asynchronously relative to the `write()` call. Most operating systems choose the latter approach, and the same choice is made in the Mimiker kernel.

Processes performing output are not the only source of characters for the terminal's output queue: if the `ECHO` terminal flag is set, characters arriving from the hardware device are automatically copied (i.e. "echoed") to the output queue. This is very useful, e.g. in the case of a shell, where the user should see the command they are typing. The high-level flow of data between processes and terminal devices is shown in Figure 3.1.

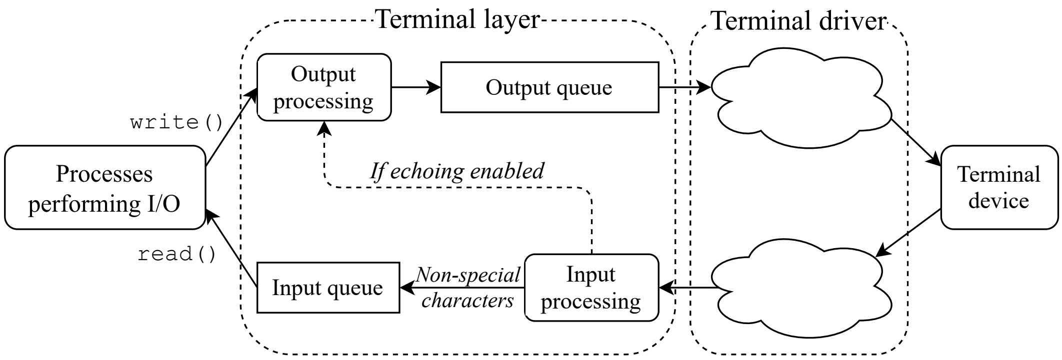

Figure 3.1: A high-level overview of terminal data flow. The clouds in the terminal driver symbolize two things: first, the terminal layer is not concerned with how the driver processes the characters it receives from the hardware or from the the terminal layer. Second, different drivers may process characters differently.

Controlling terminals

A terminal may be designated as the controlling terminal of a session. A session may have at most one controlling terminal, and a terminal 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. We define the controlling terminal of a process to be simply the controlling terminal of that process's session.

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 can, but don't need to, have their access to the controlling terminal revoked (see [2, Section 11.1.3]). If a controlling terminal disappears from the system (e.g. due to the terminal device no longer being available), the leader of the associated session is sent a `SIGHUP` signal. Note that the signal is not necessarily fatal, as it can be caught or ignored by any process.

Foreground process groups

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

A process group may 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()`[16] 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, Section 11.1.4].

As a concrete example, 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` shell 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 new foreground job's termination.

Figure 3.2 presents the process hierarchy from Figure 2.1, but now it includes the concepts of controlling terminals and foreground process groups.

Terminal modes and flags

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

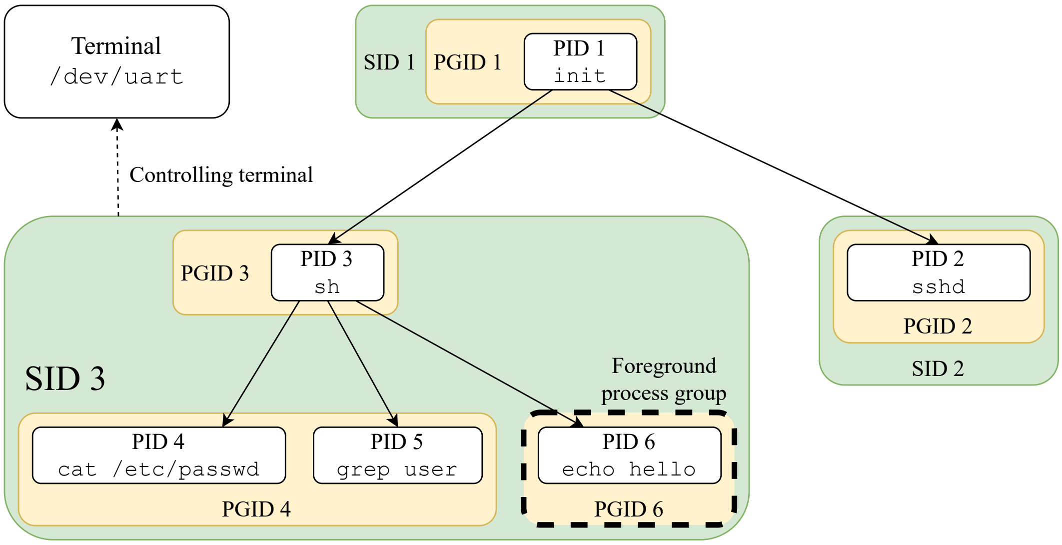

Figure 3.2: Proces hierarchy from figure 2.1 with controlling terminals and foreground process groups included. Note that sessions 1 and 2 do not have a controlling terminal. The terminal `/dev/uart` is the controlling terminal of session 3. Process group 6 is the controlling terminal's foreground process group. Once process 6 terminates, the shell (PID 3) will set the foreground process group to its own process group (PGID 3).

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

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

A text editor like `vi` 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 more complex applications, line-oriented input is usually accomplished using a library such as GNU Readline [23]. 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*, the former providing basic line editing. The terminal mode has no influence over other aspects of input processing, such as sending signals in response to certain control characters.

In canonical mode, typed characters are appended to the *line buffer*. Characters may be erased from the line buffer by typing the special characters `VERASE` (which erases the character at the end of the buffer) or `VKILL` (which erases the whole buffer). On most systems, the `VERASE` character corresponds to the Backspace key on the keyboard, while `VKILL` corresponds to Control-U. These special characters are processed by the operating system and are not forwarded to userspace. Once the user types a *line-terminating* character (e.g. `'\n'`, bound to the Return key on most keyboards), the contents of the line buffer are appended to the input queue and become available as input to userspace programs.

In non-canonical mode there is no line buffer, and every non-special character typed by the user is immediately available to userspace programs. The `VERASE`, `VKILL`, and line-terminating characters are no longer processed in a special way by the OS and appear as input on the terminal.

The terminal mode controls only part of the processing that is performed on characters coming in. There are many *terminal flags* that can enable additional character processing on input, as well as output. We will now examine the data structure that houses almost all terminal settings that can be changed by userspace programs.

The `termios` structure

For each terminal, almost all of its settings, including the terminal mode, are stored in a `termios` structure associated with that terminal. The only exception is the terminal window size, which isn't a member of the `termios` structure and is stored separately. Application code may examine and modify the contents of this structure using the `tcgetattr()`[20] and `tcsetattr()`[21] functions respectively.

The `termios` structure must contain the following fields:

- `c_iflag`: contains flags that control basic terminal input handling. Some notable flags are:
  - `ICRNL`: map the *carriage return* (CR, `'\r'`) character to the *new line* (NL, `'\n'`) 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 flags that control terminal output processing. Some notable flags 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 flags that control terminal hardware parameters. Some notable flags are:
  - `CREAD`: enable hardware receiver.
  - `CSIZE`: number of bits transmitted or received per byte (from 5 to 8).

- `c_lflag`: contains flags that control various additional functions. Some notable flags are:
  - `ECHO`: enable character echoing. This flag is cleared e.g. when the user is typing their password.
  - `ICANON`: enable canonical mode.
  - `ISIG`: send signals in response to certain control characters being received.
  - `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 the end of input to a program.
  - `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.

The current values of terminal flags and control characters can be examined from the shell using the `stty` program [15], available on most UNIX-like operating systems, including Linux. To list the current values of all terminal flags and control characters for the shell's controlling terminal, use the command:

```
stty -a
```

For the full specification of the `termios` structure, see [22].

3.1.2 Terminals in the Mimiker kernel

We will now describe the implementation of the terminal layer in the Mimiker kernel.

Overall design

The terminal layer provides the abstraction of a terminal to userspace processes, as well as to the rest of the kernel. It contains code that is independent of the underlying hardware terminal device. In the subsystem hierarchy, it sits below the generic file handling layer and above terminal drivers.

The layer is responsible for character processing, as well as buffering them in the input and output queues. It includes a mechanism to notify the hardware terminal driver when new characters arrive in the output queue, as well as when space becomes available in the input queue.

Data structures

`tty_t`

The `tty_t` structure encapsulates the hardware-independent state associated with a single terminal.

```
typedef enum {
  TF_WAIT_OUT_LOWAT = 0x1,
  TF_WAIT_DRAIN_OUT = 0x2,
  TF_OUT_BUSY = 0x4,
  TF_IN_HIWAT = 0x8,
  TF_DRIVER_DETACHED = 0x10
} 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;
  struct winsize t_winsize;
  pgrp_t *t_pgrp;
  session_t *t_session;
  vnode_t *t_vnode;
  uint32_t t_opencount;
  void *t_data;
} tty_t;
```

Listing 21: `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 the same terminal.
- `TF_IN_HIWAT`: the *input high watermark* has been reached. The watermark is reached when the terminal driver attempts to insert a character into the input queue while it is already full. Once space becomes available in the input queue, the hardware terminal driver will be notified.
- `TF_DRIVER_DETACHED`: the hardware terminal driver detached from this terminal, usually because the terminal device is no longer available. Any further

I/O on this terminal should return an error.

`t_inq` is a ring buffer storing input coming from the hardware terminal driver. A ring buffer is a data structure implementing a FIFO queue with a bounded size. They are a popular in OS kernels thanks to their simplicity and efficiency. `read()` calls on the terminal file descriptor 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 characters that should be written out by the underlying driver. The characters can come from processes writing to the terminal, or from the input processing stage, where characters can be echoed. The terminal 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. The type `linebuf_t` is a structure containing a buffer containing the line's contents, the current length of the line, and the maximum capacity of the buffer. The line's contents can be edited by the user. Once the user submits the line, its contents are copied into `t_inq`. The contents of `t_inq` cannot be changed. In non-canonical 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 being typed in by the user 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 driver operations. There are currently four operations that a driver can supply, but only one is required:

```
typedef void (*t_notify_out_t)(struct tty *);
typedef void (*t_notify_in_t)(struct tty *);
typedef void (*t_notify_active_t)(struct tty *);
typedef void (*t_notify_inactive_t)(struct tty *);

typedef struct {
  t_notify_out_t t_notify_out;
  t_notify_in_t t_notify_in;
  t_notify_active_t t_notify_active;
  t_notify_inactive_t t_notify_inactive;
} ttyops_t;
```

Listing 22: `include/sys/tty.h`: definition of `ttyops_t`.

The `t_notify_out` function notifies the driver that new data has appeared in the terminal structure's output queue (i.e. `t_outq`). The driver should ensure that data from the queue is written out to the terminal device as soon as possible. It is the only required function.

The `t_notify_in` function notifies the driver that space has become available in the input queue (i.e. `t_inq`).

The `t_notify_active` function is called when the number of open file descriptors for the terminal increases from 0 to 1. Predictably, `t_notify_inactive` is called when the number of open file descriptors drops from 1 to 0.

The terminal configuration described in the previous section is stored in the `t_termios` field. The terminal window size is stored separately in the `t_winsize` field. The `t_pgrp` field points to the foreground process group, if any, while `t_session` points to the session controlled by this terminal.

`t_vnode` points to the filesystem node representing the terminal. It is used to implement the `/dev/tty` device file, which refers to different terminals for processes in different sessions. The `t_opencount` field keeps track of the number of open file descriptors for the terminal.

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

Lifecycle of a terminal

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

The `tty_makedev()` function takes care of creating the special file. When creating the file, the function supplies `tty_fileops` to the file-handling layer as the implementation of file operations. This way, `read()` and `write()` (and other) calls on the file are directed to the terminal layer, which may in turn call the terminal driver via `t_ops`.

Once a terminal is created and visible in the filesystem, processes can open and perform I/O on it.

The process of deleting a terminal begins when the terminal driver detects that the underlying device is no longer available (e.g. due to a modem disconnect). The driver then *detaches* itself from the in-kernel terminal structure. After that happens, processes are no longer able open or perform I/O on the terminal. Once the last open file descriptor referencing the terminal is closed, the terminal structure's memory is reclaimed.

Terminal input

Processes read data from a terminal by invoking `read()` on an open terminal file descriptor. The generic file-handling layer in the kernel then uses the `fo_read` function pointer in the structure representing the open file to call the appropriate function for that particular type of file. In the case of files representing terminals, the `fo_read` function pointer is set to the `tty_read()` function.

The `tty_read()` function does very little besides calling `tty_do_read()`, which does the actual work of reading characters from the input queue. Its very slightly simplified source code is presented in Listing 23.

```
static int tty_do_read(file_t *f, uio_t *uio) {
  tty_t *tty = f->f_data;
  int error = 0;
  uint8_t c;

  WITH_MTX_LOCK (&tty->t_lock) {
❶   if (tty_detached(tty))
      return ENXIO;

    while (true) {
❷     if ((error = tty_check_background(tty, SIGTTIN)))
        return error;
❸     if (tty->t_inq.count > 0)
        break;
❹     if ((error = tty_wait(tty, &tty->t_incv)))
        return error;
    }

    if (tty->t_lflag & ICANON) {
      /* In canonical mode, read as many characters as possible,
       * but no more than one line. */
      while (ringbuf_getb(&tty->t_inq, &c)) {
❺       if (CCEQ(tty->t_cc[VEOF], c))
          break;
        if ((error = uiomove(&c, 1, uio)))
          break;
❻       if (uio->uio_resid == 0)
          break;
        /* Check for end of line. */
❼       if (tty_is_break(tty, c))
          break;
      }
    } else {
❽     ringbuf_getb(&tty->t_inq, &c);
      error = uiomove(&c, 1, uio);
    }

❾   tty_check_in_lowat(tty);
  }

  return error;
}
```

Listing 23: `kern/tty.c`: simplified definition of `tty_do_read()`.

The `f` parameter points to a `file_t` structure representing an open file. Since the file represents a terminal, the `f_data` field contains a pointer to the associated terminal's `tty_t` structure.

The `uio_t` structure describes the destination or source buffer (or buffers) of an I/O operation in the kernel. The `uio_resid` field contains the number of remaining bytes to be processed. Its design in the Mimiker kernel (see file `include/sys/uio.h`) was inspired by the `struct uio` structure found in kernels from the BSD family, see e.g. [24].

After acquiring the terminal's mutex, we first ❶ check whether the driver has detached from this terminal. If it has, we cannot do any more I/O on it, so we return an error.

The next step is to check whether we are currently allowed to read from the terminal. As previously mentioned, processes in background process groups are usually not allowed to read from their controlling terminal. The ❷ `tty_check_background()` function determines whether the calling process can perform I/O on the given terminal. The `SIGTTIN` argument specifies that we are trying to read from the terminal. If the function determines that we are not allowed to continue with the operation, it will return an error code. It may also send the specified signal (`SIGTTIN` in this case) to every process in the calling process's process group.

Next, we ❸ check whether the input queue contains any characters. The `read()` system call is blocking (at least by default), so if no data is available, the calling thread is put to sleep awaiting data. This is why, if the input queue is empty, we go on to ❹ call `tty_wait()` to wait on the `t_incv` condition variable. The role of the `tty_wait()` function will be explained at the end of this section.

Once the `tty_wait()` function returns `0`, we must repeat the checks starting from ❷, since we may no longer be allowed to read from the terminal, e.g. because our process group was moved to the background. Repeating the check at ❶ is not necessary, since `tty_wait()` does it for us. The loop continues until we find that there are characters in the input queue.

After breaking out of the loop, we need to decide how to read the characters. In canonical mode, at most one line can be read in a single `read()` call. The `VEOF` character behaves like a line terminator, but it is special in that it doesn't get passed to userspace. This is why we check for it at ❺, before we copy the character read from the input queue to the user buffer. After copying the character, we ❻ check whether we have filled the user buffer or ❼ whether the character was an ordinary line terminator. In both cases, we break out of the read loop.

In non-canonical mode, ❽ exactly one character is read and returned to userspace. This behaviour is very different from that specified in POSIX (see [2, Section 11.1.7]), where the semantics of `read()` in non-canonical mode depend on the values of the `VMIN` and `VTIME` parameters stored in the `t_cc` array. The POSIX semantics are considerably more difficult to implement and provide little to no benefit, since no applications that have been ported to Mimiker actually use them.

Finally, ❾ the `tty_check_in_lowat()` function will notify the driver if the input high watermark had been previously reached and the number of characters in the input queue has dropped below a certain threshold.

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

It is the responsibility of the terminal driver to pass characters arriving at the hardware device to the terminal layer. The driver does this by invoking the `tty_input()` function on the terminal structure, supplying the new character. The function takes care of all input processing and putting characters into the input queue.

Due to the number of terminal flags affecting input processing, the source code of the `tty_input()` function is quite large. For this reason, we will only take a look at the function's control flow, with many details omitted for the sake of readability.

```
void tty_input(tty_t *tty, uint8_t c) {
  handle IGNCR, ICRNL and INLCR flags;

  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) {
      append the line buffer to the input queue;
      notify threads waiting on t_incv;
    }
  } else {
    insert c into the input queue;
    notify threads waiting on t_incv;
  }

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

Listing 24: `kern/tty.c`: simplified control flow of `tty_input()`.

The `t_incv` condition variable is used by reading threads to wait for data in the input queue. Other than that, the listing is fairly self-explanatory, so we won't go into detail describing every step.

This is a good moment to take a closer look at the `tty_wait()` function, which is used throughout the terminal layer to wait on condition variables. It performs extra checks to make sure that the terminal driver hasn't detached from the terminal while the calling thread was sleeping.

```
static int tty_wait(tty_t *tty, condvar_t *cv) {
  int error;

❶ error = cv_wait_intr(cv, &tty->t_lock);

❷ if (tty_detached(tty))
    return ENXIO;

  /* Did we receive a signal while sleeping? */
❸ if (error == EINTR)
    return ERESTARTSYS;

  return error;
}
```

Listing 25: `kern/tty.c`: definition of `tty_wait()`.

The function first ❶ sleeps on the given condition variable. Note that it passes `&tty->t_lock` as the second argument to `cv_wait_intr()`, so the lock on the `tty` terminal is released before going to sleep. After being woken up, the `cv_wait_intr()` function acquires the lock before returning. However, note that almost anything could have happened to the terminal while we were sleeping, since we were not holding the terminal's `t_lock`. Most importantly, the associated driver might have detached from the terminal, in which case we must disallow any further I/O. Therefore, immediately after waking up we ❷ check whether the driver has detached and return an error code if it has.

Instead of being woken up by another thread signalling the condition variable, it is also possible that we were woken up by receiving a signal that needs handling. In other words, our sleep was *interrupted* (that's what the `intr` in `cv_wait_intr() stands for` — *interruptible*). At ❸ we check whether our sleep was interrupted by a signal. If it was, we return `ERESTARTSYS`, which is a special return value that will cause the system call to be restarted if it turns out that no signal has been caught. This special return value is explained in more detail in Section 4.1.

Terminal output

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

```
static int tty_write(file_t *f, uio_t *uio) {
  tty_t *tty = f->f_data;
  int error = 0;

  WITH_MTX_LOCK (&tty->t_lock) {
    if (tty_detached(tty))
      return ENXIO;

    if (tty->t_lflag & TOSTOP) {
❶     if ((error = tty_check_background(tty, SIGTTOU)))
        return error;
    }

❷   while (tty->t_flags & TF_OUT_BUSY)
      if ((error = tty_wait(tty, &tty->t_serialize_cv)))
        return error;
    tty->t_flags |= TF_OUT_BUSY;

    error = tty_do_write(tty, uio);

    tty->t_flags &= ~TF_OUT_BUSY;
    cv_signal(&tty->t_serialize_cv);
  }

  return error;
}
```

Listing 26: `kern/tty.c`: simplified definition of `tty_write()`.

The function's primary role is to ❶ check whether we are allowed to perform the operation using `tty_check_background()` and to serialize calls to `tty_do_write()` by ❷ waiting until the `TF_OUT_BUSY` flag is cleared.

The `tty_do_write()` function is responsible for putting each character in the supplied buffer into the output queue and notifying the driver.

```
static int tty_do_write(tty_t *tty, uio_t *uio) {
  uint8_t c;
  int error = 0;

  while (uio->uio_resid > 0) {
❶   if ((error = uiomove(&c, 1, uio)))
      break;
❷   if ((error = tty_output_sleep(tty, c)))
      break;
    tty->t_rocount = 0;
  }
❸ tty_notify_out(tty);

  return error;
}
```

Listing 27: `kern/tty.c`: simplified definition of `tty_do_write()`.

The function ❶ reads consecutive characters from the buffer supplied by the process and ❷ puts them into the output queue one at a time using `tty_output_sleep()`. At the end, it ❸ notifies the terminal driver of new data using `tty_notify_out()`.

The `tty_output_sleep()` function puts a single character into the output queue, sleeping if there is no space available.

```
static int tty_output_sleep(tty_t *tty, uint8_t c) {
  int error;
❶ while (!tty_output(tty, c)) {
❷   tty_notify_out(tty);
    /* tty_notify_out() can synchronously write characters to the device,
     * so it may have written enough characters for us not to need to sleep. */
❸   if (tty->t_outq.count < TTY_OUT_LOW_WATER)
      continue;
❹   tty->t_flags |= TF_WAIT_OUT_LOWAT;
❺   if ((error = tty_wait(tty, &tty->t_outcv)))
      return error;
  }
  return 0;
}
```

Listing 28: `kern/tty.c`: definition of `tty_output_sleep()`.

The function repeatedly ❶ attempts to insert the character into the output queue using `tty_output()`, which will return `false` if there is no space left in the queue. If that happens, we wait until the number of characters in the output queue drops below the *output low watermark*. Before going to sleep, we ❷ notify the terminal driver to make sure that it will eventually transfer some characters from the output queue, allowing us to proceed. The driver may transfer the characters *synchronously*, i.e. directly in the `tty_notify_out()` function, which is why we must ❸ check whether the low watermark has already been reached. If it hasn't, we ❹ set the `TF_WAIT_OUT_LOWAT` flag to indicate that there is a thread waiting for space in the output queue, and finally we ❺ go to sleep on the `t_outcv` condition variable. The variable will be signalled by the driver after it makes enough space available in the output queue.

Let us now take a look at the code of the `tty_output()`, which takes care of output character processing and putting the characters into the output queue.

```
static bool tty_output(tty_t *tty, uint8_t c) {
  int oflag = tty->t_oflag;
  if (!(oflag & OPOST)) {
❶   return tty_outq_putc(tty, c);
  }

  uint8_t cb[2];
  cb[0] = c;
❷ int ccount = tty_process_out(tty, oflag, cb);
  int col = tty->t_column;

❸ if (!tty_outq_write(tty, cb, ccount))
    return false;

❹ for (int i = 0; i < ccount; i++) {
    adjust the col variable based on the character class of cb[i];
  }
  tty->t_column = col;
  return true;
}
```

Listing 29: `kern/tty.c`: simplified definition of `tty_output()`.

If the `OPOST` flag isn't set, we skip all output processing and ❶ write the character directly to the output queue. Otherwise, we ❷ call `tty_process_out` to handle `OCRNL`, `ONLCR` and `ONOCR` terminal flags. During processing, a single character can be converted into two characters due to the `ONLCR` flag, which is why we use the `cb` buffer to store them, and `ccount` is the number of resulting characters after processing. If the input flag `OXTABS` was implemented (currently it is not), then a single character could be converted into as many as eight characters, since if the flag is set, a tab character (`'\t'`) is converted into eight spaces.

We then ❸ attempt to write the processed characters to the output queue. The call to `tty_outq_write()` will return `false` if there isn't enough space in the output queue for the characters. If that's the case, we return `false` to indicate failure.

If we successfully wrote the characters to the output queue, then we need to adjust the value of the terminal's `t_column` field, which keeps track of the column number of the terminal cursor. Each character has a *class*, which determines how it impacts the column number. For instance, the `BACKSPACE` class (which contains only the backspace character `'\b'`) decreases the column number by one, unless the column number is already zero. We therefore ❹ loop over every character we have written to the output queue and adjust the column number based on its class.

Finally, we return `true` to indicate that we have successfully written the characters to the output queue.

After the characters have been written to the output queue and the terminal driver has been notified, it is up to the driver to consume characters from the queue and transmit them to the hardware device. We will examine how this is done in the case of the UART driver in Section 3.2.

Controlling terminals

As seen in Section 2.1.3, 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 the controlling terminal of a session automatically when that session's leader opens a terminal 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.

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 terminals. 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 does not need to have a foreground process group at all times, even if it is associated with a session.

3.2 The UART terminal driver

The terminal layer delegates the task of transmitting characters over the hardware device to a *terminal driver*. Currently, the only hardware supported by the terminal subsystem is UART, a simple serial link. In this section, we will see how the UART driver interacts with the device-independent terminal layer.

3.2.1 General design

The driver consists of two main components:

- A *worker thread* that handles data transfers between the driver and the terminal layer.
- An *interrupt service routine* (ISR) that responds to hardware events (e.g. a new character being received).

The driver has two internal buffers separate from the terminal layer's input and output queues: a *transmit* and *receive buffer*. The transmit buffer holds characters taken from the output queue, while the receive buffer holds characters waiting to be processed by the `tty_input()` function.

Actually, the driver does not need to use an intermediate transmit 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 this driver.

The worker thread transfers characters from the terminal structure's output queue to the transmit buffer, and passes characters from the receive buffer to the terminal layer using the `tty_input()` function.

The interrupt service routine takes characters from the transmit buffer and writes them to the appropriate device registers to send them over the UART link. It also receives characters directly from the hardware and puts them in the receive buffer.

A good question to ask at this point is: do we even need the worker thread? Couldn't we instead call `tty_input()` directly when receiving a character in the ISR and output characters to the UART directly (i.e. synchronously) in `t_notify_out`?

The answer is yes, we do need the worker thread. Calling `tty_input()` directly from the ISR would violate the kernel's synchronization rules, as `tty_input()` uses mutexes for synchronization, and mutexes cannot be acquired in an ISR, only spinlocks. A thread context is needed to acquire the necessary mutex before calling `tty_input()`. Therefore, even if we performed output synchronously, we would still need a worker thread to do the input processing.

The flow of data between the terminal layer's queues, the UART driver's buffers and the UART device is shown in Figure 3.3. In the figure, different "agents" are responsible for different parts of the data flow, labelled with numbers:

- The thread executing the `write()` system call copies the characters from the userspace buffer (1), processes them, and puts them in the terminal's output queue (2).
- The UART driver's worker thread transfers characters between the output queue and the UART transmit buffer (3) or writes them directly to the UART hardware, bypassing the transmit buffer (10). It also takes characters from the UART receive buffer and processes them (6). Characters echoed during input processing go through output processing (9) and into the output queue (2). After input processing, the worker thread puts the characters in the terminal's input queue (7).
- The UART driver's interrupt service routine writes characters from the UART transmit buffer to the UART hardware (4) and puts characters received from the UART hardware into the receive buffer (5).
- The thread executing the `read()` system call copies characters from the terminal's input queue to the userspace buffer (8).

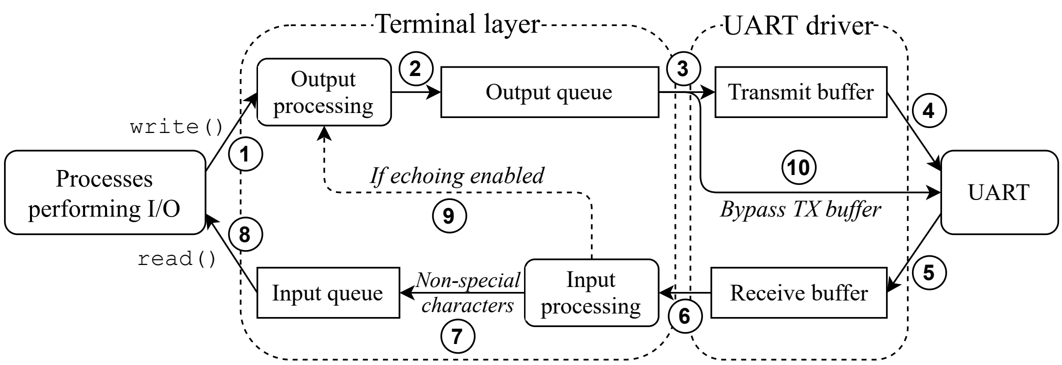

Figure 3.3: Data flow between the terminal layer's queues, the UART driver's buffers and the UART device.

3.2.2 Data structures

`tty_thread_t`

The `tty_thread_t` structure represents a worker thread.

```
typedef struct tty_thread {
  thread_t *ttd_thread;
  tty_t *ttd_tty;
  condvar_t ttd_cv;
  uint8_t ttd_flags;
} tty_thread_t;
```

Listing 30: `include/sys/uart_tty.h`: definition of `tty_thread_t`.

The `ttd_thread` field is a pointer to the `thread_t` structure representing the thread. The `ttd_tty` field is the worker thread's assigned terminal. The `ttd_cv` condition variable is used by the worker thread to wait for a signal from the ISR to do some work. The `ttd_flags` field is used to communicate what work needs to be done. It can contain the following flags:

- `TTY_THREAD_TXRDY`: The worker thread should transfer characters from the terminal layer's output queue to the transmit buffer.
- `TTY_THREAD_RXRDY`: The worker thread should transfer characters from the receive buffer to the terminal layer using the `tty_input()` function.
- `TTY_THREAD_OUTQ_NONEMPTY`: This flag is used as an optimization. It tells the ISR whether it should wake up the worker thread to fill the transmit buffer. If the flag is cleared, then the terminal layer's output queue is empty, so it's unnecessary to wake up the worker to fill the transmit buffer. At first, this flag might seem redundant: why not just check how many characters are in the terminal's output queue? Notice that holding the terminal's mutex is required in order to access its output queue. We need to know whether the queue is empty in the ISR, which is not allowed to acquire mutexes. Therefore, we use this flag as a workaround, as access to it requires holding the UART's `u_lock` spinlock, which is allowed in an ISR.

`uart_state_t`

The `uart_state_t` structure encapsulates the UART driver state associated with a single device.

```
typedef struct uart_state {
  spin_t u_lock;
  ringbuf_t u_rx_buf;
  ringbuf_t u_tx_buf;
  tty_thread_t u_ttd;
  void *u_state;
} uart_state_t;
```

Listing 31: `include/sys/uart.h`: definition of `uart_state_t`.

The `u_lock` spinlock protects all fields of the structure, including the `u_ttd` field containing the worker thread and its flags. The receive and transmit buffers are in the `u_rx_buf` and `u_tx_buf` fields respectively. The `u_state` field holds state specific to the UART hardware the driver is handling, such as the locations of device registers.

3.2.3 UART terminal driver implementation

Let us now examine the primary components of the UART terminal driver:

- The driver's implementation of the `t_notify_out` operation.
- The worker thread's main loop.
- The hardware interrupt service routine.

`uart_tty_notify_out()`

The `uart_tty_notify_out()` function is called by the terminal layer whenever new characters are inserted to the output queue. The function does very little besides calling `uart_tty_fill_txbuf()`, which transfers characters from the output queue to the UART's transmit buffer. Actual output is performed in the driver's ISR.

In theory, we do not need to call `uart_tty_fill_txbuf()` here. Instead, we could signal the worker thread to do the work for us. However, that would incur extra overhead due to synchronization and context switching. In order to avoid it, we try not to involve the worker in moving characters from the output queue to the transmit buffer.

```
  static void uart_tty_fill_txbuf(device_t *dev) {
    uart_state_t *uart = dev->state;
    tty_t *tty = uart->u_ttd.ttd_tty;
    uint8_t byte;

    while (true) {
      SCOPED_SPIN_LOCK(&uart->u_lock);
❶     uart_tty_try_bypass_txbuf(dev);
❷     if (ringbuf_full(&uart->u_tx_buf) || !ringbuf_getb(&tty->t_outq, &byte)) {
❸       if (!ringbuf_empty(&uart->u_tx_buf))
          uart_tx_enable(dev);
❹       tty_set_outq_nonempty_flag(&uart->u_ttd);
        break;
      }
❺     tty_set_outq_nonempty_flag(&uart->u_ttd);
❻     ringbuf_putb(&uart->u_tx_buf, byte);
    }
❼   tty_getc_done(tty);
  }
```

Listing 32: `sys/kern/uart_tty.c`: definition of `uart_tty_fill_txbuf()`.

The function consists of a loop that moves characters from the `tty` terminal's output queue to the transmit buffer. However, before transferring any characters, we ❶ try to bypass the transmit buffer entirely, transmitting characters directly from the output queue to the hardware. If we didn't try to bypass the transmit buffer, writing a single character to the terminal would always wake up the worker thread, which incurs extra overhead due to context switches. Such single-character writes are common, for example, when echoing characters typed by the user.

The next step is to ❷ check whether the transmit buffer is full or the output queue is empty. If neither condition is true, then we can ❻ transfer a single character. Before that, we ❺ update the `TTY_THREAD_OUTQ_NONEMPTY` flag. We then return to the beginning of the loop.

If either condition at ❷ is true, we can no longer transfer any characters. In that case, if there are characters in the transmit buffer, we ❸ tell the hardware to send an interrupt whenever it's ready to accept data. That way, the ISR will be called and it will move characters from the transmit buffer to the hardware. We then ❺ update the `TTY_THREAD_OUTQ_NONEMPTY` flag before breaking out of the loop.

The last step in the function is ❼ calling `tty_getc_done()`. In the case of terminal drivers that output characters asynchronously, the driver is responsible for waking up threads waiting for space in the terminal's output queue. This is done by calling the `tty_getc_done()` function after moving 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 the `t_outcv` condition variable.

The worker thread loop

The worker thread loop is essential to the operation of the UART driver. It is the link connecting the driver's transmit and receive buffers with the terminal layer's output and input queues.

```
static void uart_tty_thread(void *arg) {
  device_t *dev = arg;
  uart_state_t *uart = dev->state;
  tty_thread_t *ttd = &uart->u_ttd;
  tty_t *tty = ttd->ttd_tty;
  uint8_t work, byte;

  while (true) {
    WITH_SPIN_LOCK (&uart->u_lock) {
      /* Sleep until there's work for us to do. */
❶     while ((work = ttd->ttd_flags & TTY_THREAD_WORK_MASK) == 0)
        cv_wait(&ttd->ttd_cv, &uart->u_lock);
      ttd->ttd_flags &= ~TTY_THREAD_WORK_MASK;
    }
    WITH_MTX_LOCK (&tty->t_lock) {
      if (work & TTY_THREAD_RXRDY) {
❷       while (uart_getb_lock(uart, &byte))
❸         if (!tty_input(tty, byte))
            klog("dropped character %hhx", byte);
      }
      if (work & TTY_THREAD_TXRDY)
❹       uart_tty_fill_txbuf(dev);
    }
  }
}
```

Listing 33: `sys/kern/uart_tty.c`: definition of `uart_tty_thread()`.

The first step in the loop is to ❶ wait until the ISR gives us a signal to do some work by setting at least one of the flags contained in `TTY_THREAD_WORK_MASK`, i.e. `TTY_THREAD_RXRDY` or `TTY_THREAD_TXRDY`.

Once we get a signal, we check what kind of work needs to be done. If the `TTY_THREAD_RXRDY` flag is set, we ❷ read characters from the receive buffer and ❸ pass them to the terminal layer using the `tty_input()` function. If the `TTY_THREAD_TXRDY` flag is set, we ❹ move characters from the terminal's output queue to the transmit buffer using the `uart_tty_fill_txbuf()` function.

The interrupt service routine

The UART driver's interrupt service routine responds to two kinds of hardware events:

- A character has been received ("RX Ready").
- The hardware is ready to accept a new character ("TX Ready").

The ISR performs the actual transmission of characters in the transmit buffer. 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 queue into the transmit buffer.

```
intr_filter_t uart_intr(void *data) {
  device_t *dev = data;
  uart_state_t *uart = dev->state;
  tty_thread_t *ttd = &uart->u_ttd;

  WITH_SPIN_LOCK (&uart->u_lock) {
❶   if (uart_rx_ready(dev)) {
❷     (void)ringbuf_putb(&uart->u_rx_buf, uart_getc(dev));
❸     ttd->ttd_flags |= TTY_THREAD_RXRDY;
      cv_signal(&ttd->ttd_cv);
    }

❹   if (uart_tx_ready(dev)) {
      uint8_t byte;
❺     while (uart_tx_ready(dev) && ringbuf_getb(&uart->u_tx_buf, &byte))
        uart_putc(dev, byte);
❻     if (ringbuf_empty(&uart->u_tx_buf)) {
❼       if (ttd->ttd_flags & TTY_THREAD_OUTQ_NONEMPTY) {
❽         ttd->ttd_flags |= TTY_THREAD_TXRDY;
          cv_signal(&ttd->ttd_cv);
        }
❾       uart_tx_disable(dev);
      }
    }
  }
}
```

Listing 34: `sys/kern/uart.c`: definition of `uart_intr()`.

If ❶ a character has been received by the hardware, we ❷ put it into the receive buffer and ❸ signal the worker thread to pass the character to the terminal layer.

If ❹ the hardware is ready to accept a new character, we ❺ try to transmit as many characters from the transmit buffer as possible, hence the while loop. After that, if ❻ we have emptied the transmit buffer, we ❽ signal the worker thread to move some characters into the transmit buffer, but only if ❼ the terminal's output queue is not empty. If the transmit buffer is empty, we also ❾ prevent the hardware from reporting TX ready interrupts, as we can't do anything about them until the worker thread puts some data in the transmit buffer. The `uart_tty_fill_txbuf()` function takes care of enabling the interrupt, see Listing 32.

3.3 Pseudoterminals

Apart from embedded systems and debugging, character-based communication with the user over a serial link is becoming increasingly rare. Commodity desktop systems use a *graphical* shell instead of a text-based one. However, text-based interaction with the system is still possible thanks to *terminal emulators*.

A terminal emulator is a program that provides an illusion of a terminal device to processes and the user. On the user side, the emulator displays a graphical window with contents of the emulated terminal window. On the process side, it exposes a terminal file that behaves in the same way as a normal terminal file would. Processes can open it and perform I/O on it. The terminal emulator plays the role of the hardware device: it translates the user's keystrokes into characters that are then made available as input to the processes reading from the emulated terminal. It also reads the characters output by processes and displays them in the emulated terminal window.

The OS facility which allows for the emulation of a terminal device in software is called a *pseudoterminal*. A terminal emulator is far from the only use case for pseudoterminals. Programs like `sshd` (an SSH server) or `script` (which records an interactive terminal session into a file) also use pseudoterminals as a basic component. We will now take a look at how POSIX specifies their operation.

3.3.1 POSIX pseudoterminals

A pseudoterminal consists of a pair of devices: a *master device* and a *slave device*. The slave device is exposed to processes as a normal terminal. The master device is used to emulate the terminal device exposed by the slave.

The master and slave devices are linked together: everything written to the master device is processed by the terminal layer as though it came from a hardware device, and is made available as input to processes reading from the slave device. Similarly, everything written to the slave device goes through output processing in the terminal layer and appears as input to processes reading from the master device. The flow of data between master and slave devices is shown in Figure 3.4.

A new pseudoterminal can be created using the `posix_openpt()`[8] function.

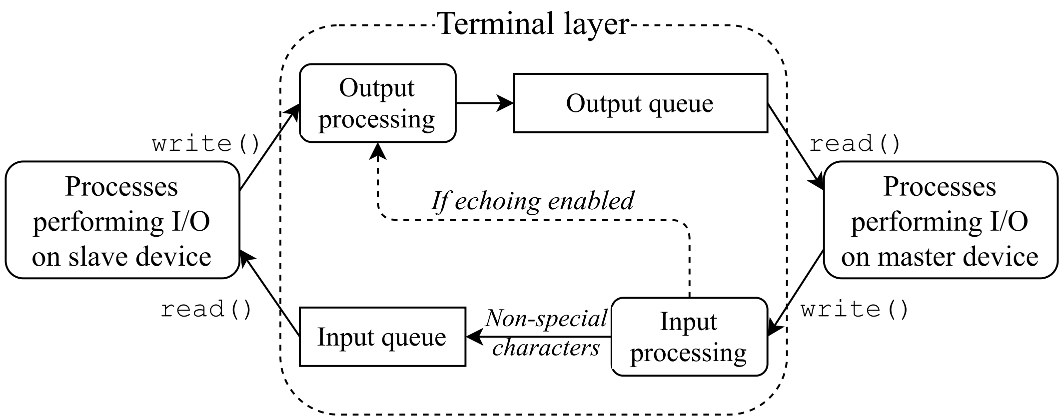

Figure 3.4: Data flow between the pseudoterminal master and slave devices, with the terminal layer's queues acting as intermediate buffers.

It returns an open file descriptor to the master device of a pseudoterminal. Note that the master device isn't exposed in the filesystem: initially only the the process that created the pseudoterminal has access to the master device (although it can duplicate and transfer the file descriptor to another process).

Once a pseudoterminal is created, its slave device is accessible via the filesystem. The slave device files are usually inside the `/dev/pts` directory. The file path of the slave device corresponding to a particular master device can be retrieved using the `ptsname()`[10] function.

The slave device is available as long as the master device file is open. Once the last master device file descriptor is closed, the slave device disappears from the filesystem, and the effect is as though the emulated terminal device was disconnected from the system. Any subsequent I/O operation on the slave device fails with an error code.

3.3.2 Pseudoterminals in the Mimiker kernel

The implementation of pseudoterminals in the Mimiker kernel is relatively simple, as most of the logic is already implemented in the terminal layer. All master device logic is implemented as a terminal driver and plugs into the terminal layer in the same way as the UART driver.

Data structures

pty_t

There is very little state associated with a pseudoterminal, as most of the state is managed by the terminal layer. The pty_t structure contains pseudoterminal-specific state.

```
typedef struct {
  atomic_int pt_number;
  condvar_t pt_incv;
  condvar_t pt_outcv;
} pty_t;
```

Listing 35: sys/kern/pty.c: definition of pty_t.

The pt_number field is a unique number identifying the pseudoterminal. The number determines the filesystem path under which the slave device is available. For instance, if pt_number is equal to 3, the slave device's path is /dev/pts/3.

The pt_incv condition variable is used by master-side readers to wait for data to become available in the slave terminal's **output** queue, while the pt_outcv condition variable is used by master-side writers to wait for space to become available in the slave terminal device's **input** queue.

Master-side input

Processes read from the master device of a pseudoterminal using the read() system call, which inside the kernel uses the fo_read function pointer to call the appropriate implementation. In the case of master device files, that function pointer points to pty_read(). Let us examine how it operates.

```
static int pty_read(file_t *f, uio_t *uio) {
  tty_t *tty = f->f_data;
  pty_t *pty = tty->t_data;
  int error;

  SCOPED_MTX_LOCK(&tty->t_lock);

  /* Wait until there is at least one byte of data. */
❶ while (ringbuf_empty(&tty->t_outq)) {
❷   if (!tty_opened(tty))
      return 0;
❸   if (cv_wait_intr(&pty->pt_incv, &tty->t_lock))
      return ERESTARTSYS;
  }

❹ error = ringbuf_read(&tty->t_outq, uio);
❺ tty_getc_done(tty);

  return error;
}
```

Listing 36: `sys/kern/pty.c`: definition of `pty_read()`.

The function first ❶ loops until there is some data in the corresponding slave terminal's output queue. If ❷ the slave terminal isn't opened by any processes, we return without reading anything. Otherwise, we ❸ wait on the pseudoterminal's `pt_incv` condition variable. If our sleep got interrupted by a signal, we return an error code of `ERESTARTSYS`.

The `pt_incv` condition variable is signaled when new data arrives in the slave terminal's output queue. Recall from Section 3.1.2 that the `t_notify_out` function is called when new characters are added to the output queue. This function's implementation is supplied by the driver, so in our case, the function simply wakes up all threads waiting on the `pt_incv` condition variable:

```
static void pty_notify_out(tty_t *tty) {
  pty_t *pty = tty->t_data;
  /* Notify PTY readers: input is available. */
  cv_broadcast(&pty->pt_incv);
}
```

Listing 37: `sys/kern/pty.c`: definition of `pty_notify_out()`.

Once we get past the loop, we know that there is data available in the output queue, so we ❹ read the data into the buffer supplied by the reading process. The interface of the terminal layer requires us to call `tty_getc_done()` after consuming characters from the output queue, so at ❺ we do just that.

Master-side output

Writing to the master device is accomplished in the kernel using the `pty_write()` function, which hooks into the `fo_write` operation in the generic file-handling layer.

```
static int pty_write(file_t *f, uio_t *uio) {
  tty_t *tty = f->f_data;
  pty_t *pty = tty->t_data;
  int error = 0;
  uint8_t c;
  uiostate_t save;

  SCOPED_MTX_LOCK(&tty->t_lock);

❶ while (uio->uio_resid > 0) {
❷   uio_save(uio, &save);
❸   if ((error = uiomove(&c, 1, uio)))
      break;
❹   if ((error = pty_putc_sleep(tty, pty, c))) {
❺     uio_restore(uio, &save);
      break;
    }
  }

  return error;
}
```

Listing 38: `sys/kern/pty.c`: definition of `pty_write()`.

The function ❶ loops as long as there are character left to write in the buffer supplied by the calling process. Before reading the actual character, ❷ the current status of the I/O operation is saved. We then ❸ read the character to be written from the userspace buffer and ❹ write it to the slave terminal's input queue using the `pty_putc_sleep()` function, which sleeps on the `pt_outcv` condition variable if there's no space in the input queue, and also returns an error if the slave terminal isn't opened by any processes. Once there's space in the input queue, the `t_notify_in` function pointer is invoked, which in our case points to `pty_notify_in`:

```
static void pty_notify_in(tty_t *tty) {
  pty_t *pty = tty->t_data;
  /* Notify PTY writers: there is space in the slave TTY's input queue. */
  cv_broadcast(&pty->pt_outcv);
}
```

Listing 39: `sys/kern/pty.c`: definition of `pty_notify_in()`.

If the `pty_putc_sleep()` function fails, we must roll back the read of the character at ❸, since we failed to properly consume it. To do that, we ❺ restore the previously saved state of the I/O operation. After rolling back, we break out of the loop and return with an error code.

Chapter 4

Implementation Challenges

In order to properly implement terminal and job control support in the Mimiker 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.

This first issue concerns the way stop signals (e.g. `SIGSTOP`) interact with threads that are sleeping interruptibly. The second one arose when trying to reconcile the existing locking rules with the terminal subsystem's requirements. The last challenge described involves working around Mimiker's lack of `poll()`[7] and `select()`[9] functionality when porting the `tetris` program.

4.1 Interruptible sleep and stop signals

When a process performs certain I/O operations (e.g. reading from a terminal), if the data isn't immediately available, the calling thread will *sleep* (i.e. suspend execution in the kernel) awaiting the data. Two kinds of sleep are usually distinguished: *interruptible* and *uninterruptible*. Interruptible sleep is used in case of I/O that may take an arbitrarily long time to complete, as is the case when reading from a terminal — there is no guarantee that data will eventually arrive. Uninterruptible sleep is used when the event being awaited is expected to occur within a short time, for instance during disk I/O.

When a thread sleeping interruptibly receives a signal for which it has registered a handler, it is resumed so that it can handle the signal: the thread's sleep has been *interrupted*.

There are two things that can happen after a thread has its sleep interrupted by a signal, depending on the value of the handler's `SA_RESTART` flag, whose value is set when installing a signal handler via `sigaction()`[13]:

- If the flag is set, the interrupted system call is restarted.
- If the flag is not set, the interrupted system call returns to userspace, usually with an error code of `EINTR`.

When a thread receives a stop signal (e.g. `SIGSTOP`) with a default disposition[1], its execution should be immediately stopped. Once the thread is resumed using `SIGCONT`, assuming the `SIGCONT` signal had no registered handler, according to POSIX [1, Section 2.4.4], the thread should resume at the point it was stopped. This means that the reception of a `SIGSTOP` signal followed by `SIGCONT` should usually be imperceptible to userspace processes.

In most kernels, signals are usually handled whenever a thread returns from the kernel to userspace. The set of pending signals for the current thread is checked, and if a signal which should be delivered is found, it is delivered either by killing or stopping the thread, or arranging for the signal handler to be called. Having a single point where signals are handled is beneficial from an architectural standpoint, as it introduces the smallest possible amount of complexity, making it easier to understand.

However, what should happen when a stop signal is being sent to a thread that is sleeping interruptibly? Should it be stopped *while* it is sleeping and remain sleeping when it is resumed? Or should its sleep be interrupted, making the interrupted thread continue execution to the userspace boundary, where it will handle the stop signal itself? Note that in the second case, the system call that was interrupted must be restarted in order to maintain the illusion that execution has been resumed at the point it was stopped.

The first approach is implemented in the FreeBSD kernel. It has the advantage of being somewhat more true to the specification, as the stopped thread will continue at exactly the point it was stopped. However, there are some drawbacks to this approach:

- It introduces additional logic to handle sending a stop signal to a thread that is sleeping interruptibly.
- Signals are no longer always processed at the kernel-userspace boundary: stop signals can also be processed when a thread is going to sleep.
- A thread sleeping interruptibly can be indefinitely prevented from waking up by sending it a stop signal. This could potentially be exploited to perform denial of service attacks on certain components of the system. For instance, a thread writing to a terminal may sleep (interruptibly) on a condition variable while having exclusive access to that terminal. If a stop signal is sent to that thread while it is sleeping, it won't relinquish its exclusive access until it is

[1] Note that `SIGSTOP`'s disposition cannot be changed from the default one, see Section 2.2.1.

resumed[2]. A malicious program could use this technique to prevent all other processes from writing to a terminal.

The curious reader can read the relevant FreeBSD (version 13.0) source code: the `sig_suspend_threads()` function found in `sys/kern/kern_sig.c`[3] is responsible for stopping the threads of a process. It is called when sending the process a stop signal.

The second approach, used in the Linux kernel, avoids the drawbacks of the first approach:

- Sending a stop signal to a thread that is sleeping interruptibly is done in the same way as sending any other signal: if the signal isn't blocked or ignored, the receiving thread's sleep is interrupted and the signal is handled at the userspace boundary.
- Signals are processed in one spot: if a thread detects a pending stop signal when it is about to go to sleep, it won't go to sleep, and instead report to the caller that a signal needs to be handled.
- Since threads can stop only at the userspace boundary, they can't hold any exclusive kernel resource (such as exclusive access to a terminal) when stopping.

When implementing stop signals in the Mimiker kernel, a choice had to be made about which approach to choose. The second approach seems conceptually simpler and avoids the pitfalls of the first one, which is why it was chosen. Let us now get into more details about how it works.

In the following discussion, a signal is considered *caught* only if its handler is executed. Therefore, if a thread is stopped using `SIGSTOP` and resumed using `SIGCONT`, assuming `SIGCONT` had no associated handler, then no signal was caught. According to this definition, a signal with a default disposition cannot be caught.

When a thread that is about to go to sleep inside a system call detects a pending signal, it returns a special error code that specifies under what circumstances the system call should be restarted:

- `ERESTARTSYS`: restart the system call if and only if no signal was caught or the caught signal's handler has the `SA_RESTART` flag set.
- `ERESTARTNOHAND`: restart the system call if and only if no signal was caught.

Consequently, if a system call returns `ERESTARTSYS` or `ERESTARTNOHAND`, and the only signals handled at the userspace boundary are `SIGSTOP` followed by `SIGCONT`

[2] This was an actual bug in the FreeBSD kernel found by the author, see `https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=255816`

[3] The FreeBSD source code is available at `https://github.com/freebsd/freebsd-src`.

(without a handler), then the system call will be restarted in the same way it would be if there were no signals to handle at all.

For an example of how the `ERESTARTSYS` error code is used in the Linux kernel, see line 2238 of file `drivers/tty/n_tty.c`[4] in Linux 5.12.4. Before sleeping, the `n_tty_read()` function checks for pending signals and returns `-ERESTARTSYS` if a signal is pending.

Linux defines two more error codes like this, but at the time of writing they are not needed to implement any system call in the Mimiker kernel, so they have not been included:

- `ERESTARTNOINTR`: always restart the system call.
- `ERESTART_RESTARTBLOCK`: like `ERESTARTSYS`, except when restarting, the interrupted system call is replaced with `restart_syscall()`[11], which adjusts time-related parameters (e.g. timeout duration in the case of `poll()`) before restarting the original system call.

These error codes may be imported into Mimiker in the future, as more functionality finds its way into the kernel. The definition of all four special error codes can be found in the Linux kernel source code (version 5.12.4) in `include/linux/errno.h`.

To illustrate exactly how the Linux approach works when adapted for the Mimiker kernel, let's trace what happens in Mimiker when a thread sleeping inside the `tty_wait()` function is sent a `SIGSTOP` signal:

1. In `sig_kill()`, the thread sending the signal calls `sleepq_abort()` on the sleeping thread, interrupting its sleep (see Listing 15).
2. The `cv_wait_sig()` function called by the sleeping thread returns `EINTR`, indicating the sleep was interrupted.
3. The `tty_wait()` function converts the `EINTR` error code to `ERESTARTSYS` (see Listing 25).
4. The `ERESTARTSYS` error code is propagated through the call stack. If the interrupted thread had previously gained exclusive access to the terminal in `tty_write()`, it relinquishes it during this step (see Listing 29).
5. Before returning to userspace, in `on_user_exc_leave()` (see Listing 16), the `sig_check()` function checks for pending signals (see Listing 17). Signals that should stop or kill the receiving process are handled immediately inside `sig_check()`. In our case the only pending signal is `SIGSTOP`, so this is where the thread is stopped.

[4] The Linux kernel source code is available at `https://elixir.bootlin.com/linux/v5.12.4/source`.

6. After being resumed via SIGCONT, sig_check() checks for pending signals once again. The return value from sig_check() indicates whether there are any pending signals with a registered handler. Let's assume that no new signals apart from SIGCONT arrived while the thread was stopped, and that the SIGCONT has no registered handler, so the return value from sig_check() indicates that no signals need to be caught.

7. The return value from sig_check(), along with the system call return value (ERESTARTSYS in our case), is passed to set_syscall_retval(). This function handles the special return codes and arranges for the system call to be restarted if needed. According to the semantics of ERESTARTSYS, since no signal is caught, the system call is arranged to be restarted by appropriately modifying the userspace thread's context.

8. Upon return from the kernel to userspace, the write() system call is immediately restarted.

In conclusion, the Linux approach to handling stop signals and interruptible sleep is a good fit for the Mimiker kernel due to its simplicity. While its implementation required introducing support for restarting system calls, that turned out not to be a big problem.

4.2 Refinement of locking rules

As explained in Section 1.3.2, locking primitives such as mutexes are used to synchronize concurrent accesses to members of data structures that are shared between threads. When using these locking primitives, care must be taken to prevent *deadlocks*. A deadlock occurs when a group of two or more threads is stuck, with each thread in the group waiting for a resource that is held by another thread in the group.

The simplest example of a deadlock is when two threads each attempt to acquire two mutexes: mtx_a and mtx_b, with thread 1 acquiring the mutexes in a different order than thread 2. Listing 40 shows example code for the threads.

Consider the following sequence of events:

1. Thread 1 acquires mtx_a;

2. Thread 2 acquires mtx_b;

3. Thread 1 attempts to acquire mtx_b and blocks, since it is already held by thread 2;

4. Thread 2 attempts to acquire mtx_a and blocks, since it is already held by thread 1.

Thread 1

```c
void thread1(void) {
  mtx_lock(mtx_a);
  mtx_lock(mtx_b);
  /* ... */
  mtx_unlock(mtx_b);
  mtx_unlock(mtx_a);
}
```

Thread 2

```c
void thread2(void) {
  mtx_lock(mtx_b);
  mtx_lock(mtx_a);
  /* ... */
  mtx_unlock(mtx_a);
  mtx_unlock(mtx_b);
}
```

Listing 40: Example code of two threads acquiring mutexes in different orders.

Clearly, neither thread can proceed, since each thread is waiting for a lock that is held by the other thread. Hence, we have a deadlock.

One of the most common ways to prevent such deadlocks is to impose an *ordering* on the locks and require that locks are always acquired according to that ordering. For instance, if we imposed an ordering that placed `mtx_a` before `mtx_b`, it would be illegal to acquire `mtx_a` while holding `mtx_b`. If a thread wanted to acquire both mutexes, it would need to acquire `mtx_a` first and then `mtx_b`. Thus, the two threads from the example would acquire the two mutexes in the same ordering, preventing a deadlock from happening.

The Mimiker kernel tries to avoid deadlocks using this method. Whenever multiple locks are acquired, they are acquired according to an informally defined ordering. During the integration of the terminal subsystem with other subsystems of the kernel (e.g. signal handling), the per-terminal `t_lock`[5] embedded in the `tty_t` structure had to be placed somewhere in the lock ordering. This posed some problems, as the following example illustrates.

Consider what happens when a session leader terminates. If the session has a controlling terminal, and the terminal has a foreground process group, then every process in the foreground process group should receive a `SIGHUP` signal. In order to access a process's session structure and its controlling terminal, the `all_proc_mtx`[6] mutex must be held. In order to examine a terminal's foreground process group, the terminal's `t_lock` mutex must be held. Therefore, `all_proc_mtx` must be acquired first in order to look up the controlling terminal, and then `t_lock` must be acquired to examine the foreground process group. This places `all_proc_mtx` before `t_lock` in the lock ordering.

After looking up the foreground process group, we might need to send a signal to each process in the group using the `sig_pgkill()` function. This function requires

[5] The per-terminal `t_lock` mutex synchronizes accesses to all mutable members of the `tty_t` structure, see e.g. Listing 27.

[6] The global `all_proc_mtx` mutex synchronizes accesses to process-tree structures such as process groups and sessions, see e.g. Listing 8.

both all_proc_mtx and the target process group's pg_lock[7] to be held by the caller. This implies that pg_lock comes after t_lock and all_proc_mtx in the ordering.

Now that we have established that all_proc_mtx < t_lock < pg_lock, let's look at what needs to happen during terminal input processing. If the ISIG terminal flag is set, we must send a signal to every process in the foreground process group in response to certain control characters. Understandably, character processing happens while holding t_lock, so that other threads can't change the terminal settings in the middle of processing. In order to send a signal to the foreground process group, we must acquire all_proc_mtx and the group's pg_lock. However, note that acquiring all_proc_mtx while holding t_lock would violate the ordering!

Several bad solutions quickly come to mind:

- Require callers of tty_input() to hold all_proc_mtx.
  This is bad, because all_proc_mtx is only needed when sending a signal, which happens rarely during terminal input processing. Processing of normal characters should not bear the cost of synchronization that is needed only in exceptional cases.

- When a signal needs to be sent, drop t_lock, and then acquire all_proc_mtx, t_lock and the foreground process group's pg_lock.
  This might work, but requires some extra precautions. First of all, the terminal device might disappear while we are not holding t_lock. Secondly, dropping t_lock would lead to headaches for callers of tty_input(), which might rely on holding t_lock for their own synchronization purposes.

A semi-reasonable solution is to delegate sending the signal to another thread. This is the approach used in the NetBSD kernel. This approach is actually *necessary* in the case of NetBSD, as the terminal layer there uses spinlocks for synchronization, not mutexes, and sending a signal requires holding a mutex. Since it is illegal to acquire a mutex while holding a spinlock, delegating the job to a thread is a necessity.

We chose not to follow NetBSD's approach and instead follow the approach found in the FreeBSD kernel, where it is possible to send a signal to a process group while holding a terminal's mutex. That's because the locking requirements of the function used to send signals in FreeBSD are less strict compared to the Mimiker kernel. We therefore relaxed the locking requirements of the sig_kill() function, so that it no longer requires all_proc_mtx to be held by the caller. This was not a trivial task. To understand why, we need to understand why sig_kill() required all_proc_mtx in the first place.

A parent may be waiting for a child process's state to change using the waitpid() function. Resuming a process using the SIGCONT signal counts as a state change.

[7] The per-process-group pg_lock mutex synchronizes accesses to the group's list of member processes, see e.g. Listing 11.

Therefore, the parent of a stopped process receiving a `SIGCONT` signal must be notified of the state change. This notification happens inside the `sig_kill()` function. The code of the `wait4()` system call (which implements the `waitpid()` POSIX function) used `all_proc_mtx` to synchronize with `sig_kill()`, so if we were to remove `all_proc_mtx` from `sig_kill()`, we would also need to adjust the `wait4()` system call's code.

Another complication is that in order to notify the parent, we need to access the `p_parent` field of the process receiving the signal. Since this field is mutable (when a process terminates, it changes its children's `p_parent` pointer to another process), accesses to were synchronized using `all_proc_mtx`. This requirement also needed to be changed.

The first issue was resolved by using the parent process's `p_lock` to synchronize the parent thread with the thread sending the signal. The `PF_CHILD_STATE_CHANGED` flag is set in the parent process to tell it that a child process's state has changed.

The second issue was resolved by changing the locking rules around the `p_parent` field of `proc_t`. The field is now protected by *two* locks: the process's `p_lock` and `all_proc_mtx`. In order to read the field's value, holding either lock is sufficient. However, in order to change the value, *both* locks must be held. Since in `sig_kill()` we're only reading the value of `p_parent`, holding the target process's `p_lock` is enough.

While the two issues are now resolved, another problem arises as a result: in order to notify the parent of the target process's state change, we must acquire the parent's `p_lock` *while* holding its child's `p_lock`. This used to be forbidden by the locking rules. However, note that we can safely acquire the parent's mutex if we enforce the following rule:

> The `p_lock` of the parent of a process `p` may be acquired while holding the `p_lock` of `p`, but not the other way around.

Since the process hierarchy forms a tree, there are no cycles, so it is impossible for a deadlock to occur. We could extend this rule to arbitrary ancestors instead of just the parent, but currently it does not bring any additional benefits.

With these adjustments in place, the `sig_kill()` function no longer requires `all_proc_mtx` to be held by the caller, and it's possible to send a signal to the foreground process group directly from within `tty_input()`.

4.3 Porting `tetris` to Mimiker

With the terminal subsystem in place, Mimiker needed an application that would showcase its capabilities, as well as test them. The well-known game of Tetris seemed to be a good fit. The NetBSD operating system includes the game in its sources. It is written in a very simple and portable way, making it a good candidate for porting to the Mimiker OS. However, there were several challenge along the way, which will be described in this section.

4.3.1 The `terminfo` library

NetBSD's `tetris` program uses advanced terminal features, such as moving the cursor to arbitrary positions on the screen and coloured output. These features are implemented using *escape codes*: special character sequences which are interpreted by the terminal device or terminal emulator. There is a standard set of ANSI escape codes[8]. For instance, in order to clear the entire screen on a terminal device supporting the ANSI escape codes, a program would output the sequence `^[[2J`, where `^[` stands for the *escape* character, whose ASCII code is `0x1B`.

Some terminal devices and terminal emulators do not strictly conform to the ANSI escape code standard. They may implement extra functionality using custom escape codes, or use different escape codes for functionality that is part of the standard. This is clearly a problem when trying to write programs that use advanced terminal features, as supporting all the different escape codes quickly becomes a pain. This is where the `terminfo`[17] library, or simply `libterminfo`, comes in to save the day.

The `terminfo` library provides a portable way of accessing terminal-related information, such as escape codes. To this end, the library makes use of a database of information about various specific terminal device models or terminal emulators. This database is often called the `terminfo` database. For instance, on the author's Linux system, the database is located at `/usr/lib/terminfo`. When a program using `libterminfo` wants to e.g. clear the screen, it would call a library function to get the proper escape code for the specific terminal device being used. The `TERM` environment specifies the model of the terminal device being used.

Since the `tetris` program uses the `terminfo` library to handle escape codes, the library needed to be ported to Mimiker, which was not especially difficult. Below is a summary of the steps taken to port `libterminfo` to Mimiker:

- Import missing `libc` functionality from NetBSD (e.g. `cdb` database handling functions).
- Modify the library's build process to use GNU `gperf` instead of NetBSD's `nbperf` to generate perfect hashing functions.
- Add terminal descriptions for several terminal models: `xterm`, `xterm-256color`, `xterm-direct`, `vt100`, and `vt220`.

[8] `https://en.wikipedia.org/wiki/ANSI_escape_code`

- Set the TERM environment variable to a sensible default value (i.e. xterm) in the shell's startup script. xterm is a good default choice, since it is an old terminal emulator, and most modern emulators are compatible with it.

4.3.2 Working around lack of select()/poll()

The biggest issue with porting tetris to Mimiker was how the game handles user input. Naturally, tetris is an interactive game, so whenever a character is typed by the user, the game reacts immediately, updating its state and redrawing the screen. Besides that, the piece that is falling from the top is periodically lowered, regardless of user input. Therefore, when reading input from the terminal, some sort of timeout must be used, so that the game doesn't block indefinitely if the user doesn't press any keys.

In its original form found in NetBSD sources, the poll()[7] system call is used to wait for user input with a timeout. Unfortunately, at the time of writing the Mimiker kernel does not implement the poll() system call or any other system calls with similar functionality, such as select()[9]. To make the game work, some workaround had to be put in place.

Thankfully, there is a viable workaround that uses the SIGALRM signal and the setitimer()[3] function. The setitimer() function sets up a *timer* for the calling process. When the timer expires, the process receives a SIGALRM signal. We can use the timer and the signal as a timeout mechanism.

Suppose we want to read input from the terminal with a timeout of one second. First, we set up a handler for the SIGALRM signal that doesn't do anything. Next, we set a timer to expire one second in the future using the setitmer() function, and call the read() function to read input from the terminal. The read() function will sleep awaiting input if none is available. However, thanks to the timer, it will sleep for at most one second, since after one second a SIGALRM signal will interrupt the sleep. If the sleep is interrupted by a signal, the read() function will return an error code of EINTR, which lets us know that the timeout has expired. If the read() function returns without an error, which means that input arrived before the timeout, then we disable the timer.

It is worth repeating that this is a workaround, not a solution. There are some edge cases that can pose problems. For example, if the timer expires before calling read(), the read() function will not be interrupted. This can be remedied by making the timer periodic.

Funnily enough, this workaround (which works around missing functionality in Mimiker) itself required functionality that was missing, namely the setitimer() and getitimer() functions. The functions' corresponding system calls with the same names had to be implemented in the kernel from scratch. This required some modifications to the kernel's *callout* facility, which is used to arrange for functions to be called at a specified point in time. Overall, the changes required to implement timers amounted to about 300 lines of code.

With the missing functionality in place, the workaround worked without issues, and `tetris` is fully playable on the Mimiker OS. Figure 4.1 is a screenshot of `tetris` running under Mimiker on an emulated MIPS Malta board.

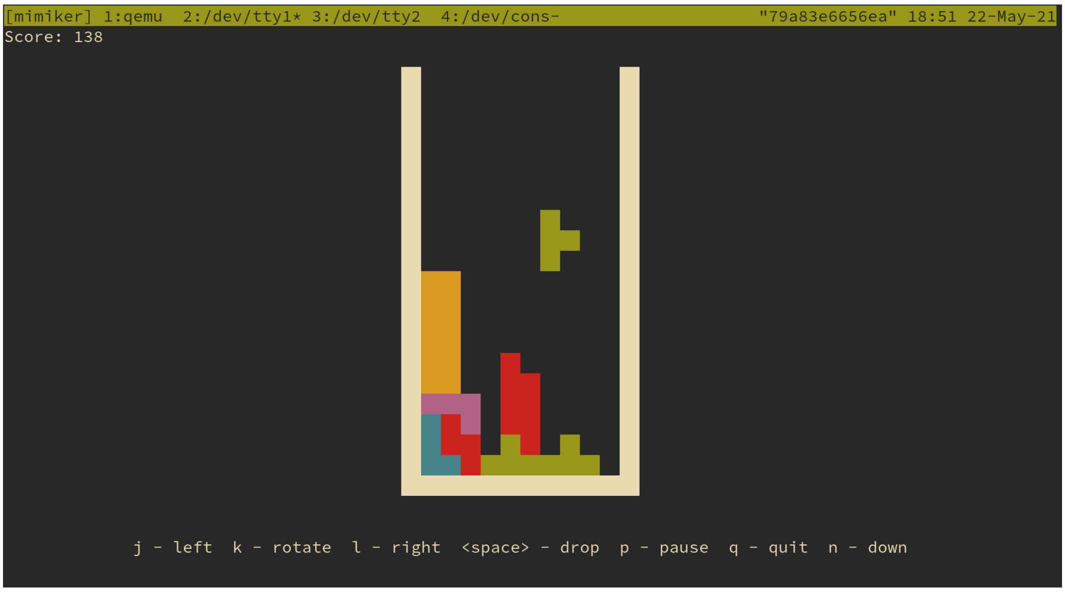

Figure 4.1: Screenshot of `tetris` running under Mimiker.

Chapter 5

Conclusion

This chapter lays out instructions on how to compile and run the Mimiker OS, as well as test some use cases enabled by proper support for job control and the terminal subsystem. Finally, the author's contributions to Mimiker and other projects are summarized.

5.1 Compilation and usage instructions

This section presents step-by-step instructions on how to compile and run Mimiker, as well as some shell commands that showcase the functionality implemented by the author. The `git` and `docker` programs are assumed to be installed on the reader's system.

Compiling Mimiker

1. Get the source code.
   The following command will download the Mimiker sources into the `mimiker` subdirectory of the current working directory:
   `git clone --depth 1 https://github.com/cahirwpz/mimiker`

2. Download the development environment.
   Most Mimiker developers use the Mimiker server[1] to compile and run their code. However, access to the server requires an account, which the reader is assumed to not have. While the project provides instructions[2] on how to install the software required to compile Mimiker, the procedure is quite cumbersome. For this reason, the author has prepared a Docker container with all the software needed to compile and run Mimiker on an emulated machine. The following command downloads the container: `docker pull jpiecuch96/mimiker-dev`

[1] `mimiker.ii.uni.wroc.pl`

[2] `https://github.com/cahirwpz/mimiker/blob/master/README.md`

3. Run a shell inside the development environment.
   ```
   docker run --rm -it -u 1000 -v $PWD/mimiker:/mimiker -w /mimiker \
     jpiecuch96/mimiker-dev bash
   ```
   Note: this command should be run in the directory containing the `mimiker` subdirectory. Also, the reader's user ID is assumed to be 1000 (which is the case on most systems).

4. Compile Mimiker.
   Run the following command in the shell inside the development environment:
   `make -j$(nproc)`
   If the command fails for whatever reason, try running it again without the `-j` option.

Running Mimiker

To run a shell inside Mimiker, run the following command in the shell inside the development environment:
`./launch init=/bin/ksh`
This should open a `tmux` session with the `ksh` shell running in the active window. We can now run various commands inside Mimiker.

Terminal commands

This section presents some terminal commands that make use of Mimiker's job control capabilities or the terminal subsystem.

Terminal access control based on background process groups

1. List current terminal settings:
   `stty -a`
   The output contains `-tostop`, which indicates that the `TOSTOP` flag is not set. This means that a process in a background process group should be able to write to the terminal.

2. Check that both background and foreground processes can write to the terminal:
   `echo background & echo foreground`
   Both `background` and `foreground` will be output.

3. Check that background processes are not allowed to read from the terminal:
   `cat &`

The `cat` program tries to read from the terminal, but since it is in a background process group, the attempt results in it being stopped by a `SIGTTIN` signal. This can be verified by running the `jobs` command, which will show the stopped process. It can be brought into the foreground using the `fg` command, after which it can be terminated by typing `^C` (Control-C).

4. Set the `TOSTOP` terminal flag:
   `stty tostop`

5. Check that background processes are not allowed to write to the terminal:
   `echo background & echo foreground`
   This time, only `foreground` will be output. Running the `jobs` command will reveal that the background process was stopped upon trying to write to the terminal. If we bring it into the foreground using `fg`, `background` will be output.

Recording a terminal session with `script`

The `script` program uses pseudoterminals to record the output of all programs in a terminal session. It can record the output to a plaintext file, or it can also record the timestamp of each output event, making it possible to play back the session exactly as it was recorded.

1. Change into the `/tmp` directory:
   `cd /tmp`

2. To record into a plaintext file, just use the command:
   `script`

3. Now type some commands, for instance:
   `echo one && echo two`

4. Type `^D` (Control-D) to end the recorded terminal session.

5. The plaintext file with the recording is at `/tmp/typescript`:
   `cat typescript`

6. To record a session with timestamps, use the command:
   `script -r`
   To play it back, use:
   `script -pq typescript`

5.2 Summary of contributions

Mimiker source code

Below is a summary of the author's code contributions to the Mimiker OS made as part of the overall implementation effort described in this thesis:

Job control

- Implementation of process stopping and continuing via signals.
  Most important functions:
  - `sys/kern/proc.c`: `proc_stop()`, `proc_continue()`
  - `sys/kern/signal.c`: `sig_kill()`, `sig_check()`
- Adding support to the `wait4()` system call for waiting for child processes to be stopped or continued.
  Most important functions:
  - `sys/kern/proc.c`: `do_waitpid()`, `proc_wakeup_parent()`
- Implementation of POSIX sessions and the `setsid()` system call.
  Most important functions:
  - `sys/kern/proc.c`: `session_enter()`, `session_leave()`
- Expansion of the `setpgid()` system call's functionality.
  Most important functions:
  - `sys/kern/proc.c`: `pgrp_enter()`
- Implementation of orphaned process group detection.
  Most important functions:
  - `sys/kern/proc.c`: `pgrp_jobc_leave()`, `pgrp_jobc_enter()`

Terminal subsystem

- Terminal layer: `sys/kern/tty.c`
- UART terminal driver: `sys/kern/uart_tty.c`, `sys/drv/uart.c`
  Note: the version originally written by the author only supported the UART found on the MIPS Malta development board. The code was later adapted by Paweł Jasiak to be independent of any specific UART device.
- Pseudoterminals: `sys/kern/pty.c`

Signal handling

- Implementation of the `sigprocmask()` and `sigsuspend()` system calls.
  Most important functions:
  - `sys/kern/signal.c`: `do_sigprocmask()`, `do_sigsuspend()`
- Adding support for the `sa_mask` field in the `sigaction()` system call.
  Most important functions:
  - `sys/kern/signal.c`: `sig_post()`, `sig_return()`
- Adding support for the `SA_RESTART` flag in the `sigaction()` system call and implementation of system call restarting.
  Most important functions:
  - `sys/kern/exception.c`: `on_user_exc_leave()`, `set_syscall_retval()`

Ported programs and libraries

- `atto`: minimal `emacs`-like text editor.
- `script`: records terminal input and output into a file.
- `libterminfo`: terminal information library.
- `tetris`

Miscellaneous

- Management of PIDs, PGIDs and SIDs using hash tables.
- Extension of the `devfs` filesystem to allow for removing device nodes.
- Implementation of `setitimer()` and `getitimer()` system calls.

Other contributions

When reading FreeBSD source code, the author found and reported 3 bugs in the FreeBSD kernel:

- Bug 250701 - "Race condition between `tty_wait_background()` and `doenterpgrp()`"
  `https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=250701`
- Bug 251915 - "TOCTOU race between `tty_signal_sessleader()` and `killjobc()`"
  `https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=251915`

- Bug 255816 - "tty: Potential DoS on terminal writes by stopping thread inside `ttydisc_write()`"
  `https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=255816`

All the bugs have been fixed, with the last fix being committed to the `main` branch on the 18th of May, 2021.

Bibliography

[1] General Information.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html.
Accessed: 28/04/2021.

[2] General Terminal Interface.
https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap11.html.
Accessed: 13/04/2021.

[3] getitimer, setitimer - get and set value of interval timer.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/select.html.
Accessed: 11/05/2021.

[4] Implementations of singly-linked lists, singly-linked tail queues, lists and tail queues.
https://www.freebsd.org/cgi/man.cgi?query=TAILQ_HEAD&apropos=0&sektion=0&manpath=FreeBSD+13.0-RELEASE+and+Ports&arch=default&format=html#TAIL_QUEUES.
Accessed: 26/05/2021.

[5] kill - send a signal to a process or a group of processes.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/kill.html.
Accessed: 29/01/2021.

[6] locking - kernel synchronization primitives.
https://www.freebsd.org/cgi/man.cgi?query=locking&manpath=FreeBSD+12.2-RELEASE+and+Ports.
Accessed: 19/05/2021.

[7] poll - input/output multiplexing.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/poll.html.
Accessed: 11/05/2021.

[8] `posix_openpt` - open a pseudo-terminal device.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_openpt.html.
Accessed: 23/04/2021.

[9] `pselect, select` - synchronous I/O multiplexing.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/select.html.
Accessed: 11/05/2021.

[10] `ptsname` - get name of the slave pseudo-terminal device.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/ptsname.html.
Accessed: 23/04/2021.

[11] `restart_syscall` - restart a system call after interruption by a stop signal.
https://man7.org/linux/man-pages/man2/restart_syscall.2.html.
Accessed: 11/05/2021.

[12] `setsid` - create session and set process group ID.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/setsid.html.
Accessed: 25/02/2021.

[13] `sigaction` - examine and change a signal action.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/sigaction.html.
Accessed: 29/01/2021.

[14] `sigprocmask` - examine and change blocked signals.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/sigprocmask.html.
Accessed: 29/01/2021.

[15] `stty` - set the options for a terminal.
https://pubs.opengroup.org/onlinepubs/9699919799/utilities/stty.html.
Accessed: 07/05/2021.

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

[17] Terminal Information Library.
https://man.netbsd.org/terminfo.3.
Accessed: 20/05/2021.

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

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

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

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

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

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

[24] `uio`, `uiomove`, `uiomove_frombuf`, `uiomove_nofault` - device driver I/O routines.
https://www.freebsd.org/cgi/man.cgi?query=uio&manpath=FreeBSD+12.2-RELEASE+and+Ports.
Accessed: 07/05/2021.

[25] `wait, waitpid` - wait for a child process to stop or terminate.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/waitpid.html.
Accessed: 25/02/2021.

[26] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., USA, 1986.

[27] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* No Starch Press, USA, 1st edition, 2010.

[28] Julian Pszczołowski. An overview of synchronization methods in Unix-like kernels and an implementation of turnstiles in Mimiker OS. Bachelor's thesis, University of Wrocław, 2018. Written in Polish.

[29] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 3rd edition, 2013.