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

1 marca 2021

Abstract

English Abstract

Polish Abstract

Contents

Chapter 1

Introduction

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

POMYSŁ NA WPROWADZENIE:

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

1.1 What is a terminal?

1.1.1 Early-day terminals

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

Time-sharing systems were the solution to this problem. A time-sharing system allows users to interact with the machine using a terminal. It gives the user an illusion of exclusive posession of the machine's resources. In reality, the operating system provides a fraction of the machine's computing resources to each user. Multics[16] is an early example of a time-sharing operating system.

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

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

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

1.1.2 Terminals in a modern context

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

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

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

To summarize, even though 1970s-style video terminals are no longer used, character-based computer interfaces that are compatible with those terminals remain

an important part, or even the basis of interaction between a user and a computer system.

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 their *process group ID*, or *PGID* for short.

Every process belongs to exactly one process group. When a new process is created using `fork()`[7], it runs in the process group of its parent. A process can change its own process group, or that of one of its children. This is done using the `setpgid()`[8] 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()`[14] and `kill()`[1] 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. Session 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()`[2] 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.

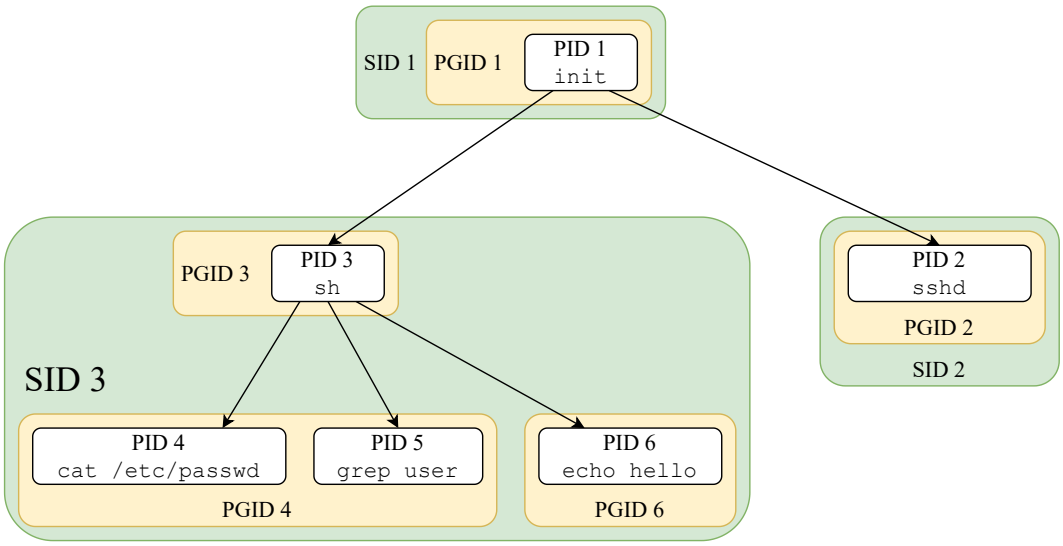

Figure 2.1: A typical process hierarchy.

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` signal followed by `SIGHUP`. 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 1: `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.

`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 2 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 2: `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 mtx_lock_pair(&old_pgrp->pg_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;
  }

  mtx_unlock_pair(&old_pgrp->pg_lock, &target->pg_lock);

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

  return 0;
}
```

Listing 3: `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 both the old group and target group's lock. However, the order in which we acquire the two locks must be consistent across the whole kernel, in order to avoid deadlocks. The ❸ `mtx_lock_pair()` function acquires two mutex locks in a consistent order, based on the memory addresses of the locks.

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.

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.

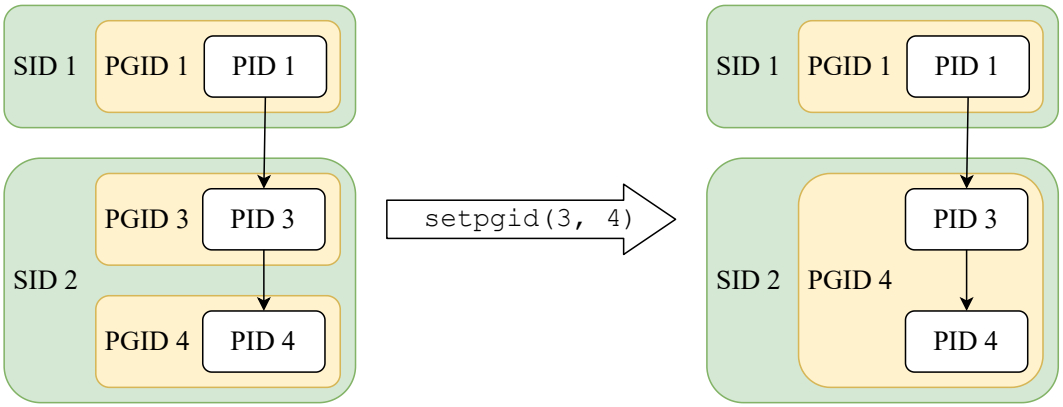

Figure 2.2: A parent's change of process group causes a child's process group to become orphaned.

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 4: `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 5: `sys/kern/proc.c`: definition of `pgrp_jobc_leave()`.

2.1.3 Sessions in the Mimiker kernel

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 [15, 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 [1]. 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. 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 [3]. 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 [4]. 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` 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, 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 (more on 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` 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.

```
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 6: `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 7 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 7: `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. For instance, pending signals are delivered before returning from a system call.

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.

Signals with special effects (i.e. stopping or killing a process) are not posted, but are instead handled as part of checking for a pending signal. This is an implementation choice that simplifies handling of certain edge cases.

Pending signals are not checked at all if the current thread does not have the `TDF_NEEDSIGCHK` flag set. This flag is set in `sig_kill()`, see listing 7. This saves us from unnecessarily checking for pending signals every time we return to userspace.

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). Instead, we will focus on the `sig_check()` function, which

checks for pending signals and handles job control signals. Listing 8 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 8: `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 stopping the process with `proc_stop()`, control goes back to ❷.

Next, we ❺ handle signals that should kill the process. The `sig_exit()` function 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 ❸ 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 9.

```
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 9: `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 10.

```
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 10: `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 11.

```
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 11: `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

TODO Zrobić coś z tym i POSIX Job Control

3.1 What is POSIX?

POSIX [13] is a set of standards specifying an operating system interface. Its primary goal is to make it easy to write portable libraries and applications that work across different operating systems, although they usually need to be compiled for each operating system separately.

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

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

3.2 POSIX Job Control

3.2.1 Jobs

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

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

the operating system. The shell puts different jobs in different process groups.

3.3 POSIX Terminals

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

3.3.1 Controlling terminals

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

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

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

3.3.2 Foreground process groups

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

At any time, a process group can be designated as the foreground process group of its session's controlling terminal. All other process groups in the session are background process groups. The foreground process group of a terminal can be set using the `tcsetpgrp()`[5] 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 [6].

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

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

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

3.3.3 Terminal modes and flags

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

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

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

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

In serious applications, line-oriented input is usually accomplished using a library such as GNU Readline [12]. However, POSIX mandates that basic line-editing functionality is provided by the operating system itself. Applications may choose whether they want to use this functionality by choosing the *terminal mode*.

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

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

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

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

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

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

The `termios` structure

For each terminal device, all of its settings, including the terminal mode, are stored in a `termios` structure associated with that terminal. Application code may examine and modify the contents of this structure using the `tcgetattr()`[9] and `tcsetattr()`[10] functions respectively.

The `termios` structure must contain the following fields:

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

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

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

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

For the full specification of the `termios` structure, see [11].

Chapter 4

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

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

4.1 Job Control

4.1.1 Jobs

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

4.1.2 Process groups

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

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

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

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

Orphaned process groups

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

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

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

Process group lifecycle

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

Once the list is empty, the process group's lifetime ends. During deallocation,

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

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

4.1.3 Sessions

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

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

Listing 13: `include/sys/proc.h`: definition of `session_t`.

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

4.2 Terminals

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

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

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

Listing 14: `include/sys/tty.h`: definition of `tty_t`.

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

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

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

`t_outq` is a ring buffer storing output from processes. The terminal device driver reads characters from this buffer and takes care of transmitting them. `t_outcv` is a condition variable used to wait for space to become available in the output queue or for the queue to become empty.

`t_line` stores the contents of the line the user is typing in canonical mode. Its contents can be edited by the user. Once the user submits the line, its contents are

copied into `t_inq`. The contents if `t_inq` cannot be changed. In raw mode, `t_line` is not used.

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

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

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

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

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

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

Listing 15: `include/sys/tty.h`: definition of `ttyops_t`.

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

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

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

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

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

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

4.2.1 Creating a terminal device

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

node in the `/dev` directory of the filesystem.

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

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

4.2.2 Terminal input

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

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

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

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

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

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

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

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

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

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

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

    insert c into the line buffer;

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

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

Listing 16: `kern/tty.c`: simplified control flow of `tty_input()`.

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

4.2.3 Terminal output

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

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

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

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

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

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

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

4.2.4 Controlling terminals

As seen in section 4.1.3, the controlling terminal for a session is stored in the `s_tty` field of `session_t`. When a new session is created, its `s_tty` field is set to `NULL`.

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

When the session leader process exits, the session loses its controlling terminal. This is the only way a terminal may become dissociated from its session.

4.2.5 Foreground process groups

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

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

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

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

Chapter 5

Implementation Challenges

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

5.0.1 Interruptible sleep and stop signals

5.1 Locking

Chapter 6

Conclusion

Bibliography

[1] `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.

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

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

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

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

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

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

[8] `setpgid` - set process group ID for job control.
`https://pubs.opengroup.org/onlinepubs/9699919799/functions/setpg`

id.html.
Accessed: 17/11/2020.

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

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

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

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

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

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

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

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