Implementation of Pipes for the ULIX Operating System

Thomas Cyron

Bachelor Thesis in Computer Science
at the Technische Hochschule Nürnberg

June 2015

Supervised by:
Prof. Dr. rer. pol. Ralf-Ulrich Kern
Dipl.-Math. Dipl.-Inform. Hans-Georg Eßer

Abstract

ULIX is a Unix-like operating system specially designed for educational purposes. In Unix, pipes are used for inter-process communication. While anonymous pipes only reside in memory, named pipes are represented as special files in the file system. The thesis covers the implementation of both named and anonymous pipes for the ULIX OS. A virtual file system is created and exported to user land to provide an interface to debug and introspect pipes. The Literate Programming paradigm is used to describe in detail how pipes can be implemented in a Unix-like operating system.

Contents

Chapter 1

Introduction

1.1 The Ulix Operating System

Ulix is a Unix-like operating system developed by Hans-Georg Eßer and Felix Freiling at the University of Erlangen-Nuremberg. It has been created specially for educational purposes and is written using Literate Programming, a programming paradigm in which the explanation of the program becomes the key part, and the actual source code is embedded into the explanation. An introduction to Literate Programming is given in section 1.2.

Ulix itself is implemented in the C programming language and falls back to assembler code when necessary. It targets the i386 CPU architecture. Although Ulix is not meant to be used as a general-purpose operating system, it still has features found in modern operating systems, including virtual memory, paging, virtual file systems, processes and threads, and multi-user support.

The source code as well as the book *The Design and Implementation of the Ulix Operating System* [EF15] will be available at `http://ulixos.org/`. (No public version was released at the time of writing.)

1.2 Literate Programming

The Literate Programming paradigm was first described by Donald E. Knuth. In his paper from 1984, he writes: “Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.” [Knu84]

When programming in the traditional way, the source code of a program is annotated with comments which describe what a code block does, how it is implemented, or why a particular solution was chosen. Because the source code is designed to be interpreted by a compiler, it is subject to certain constraints. In C, for example, a function can only be called when it has been declared or implemented beforehand, global structures can only be defined in the top-level scope, and once a structure has been defined, it is impossible to alter the structure to add another element to it. These constraints make it hard for a human to read and understand the program when reading it from top to bottom, so jumping inside the source code document becomes inevitable. This is distracting and interrupts the flow of reading.

Literate Programming solves this problem. In a literate program, the documentation and explanation is the main component of the program. A literate program is written in a natural language and is supposed to be read like a book, without the need to jump between pages. The actual source code is embedded in the text. To avoid jumps, code chunks are used, which are pieces of code that have a name. Throughout the document, a code chunk can be referred to by its name, and a code chunk can be embedded into another code chunk. It works like a macro in the preprocessor of the C language. Whenever a code chunk is included in another chunk, the inclusion statement is replaced with the actual content of the included code chunk. Because it is possible to append code to an already existing code chunk, a literate program does not have to follow the order that is required by a compiler. For example, a `struct` definition may contain a code chunk named `struct elements`, and later on in the program, a new element can be added to the structure by appending code to that code chunk.

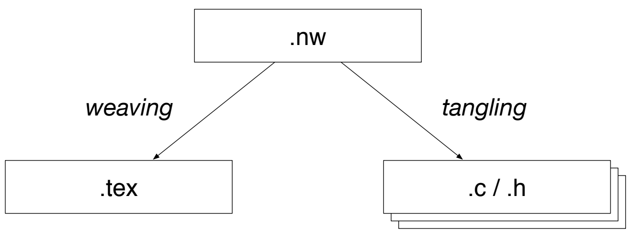

Figure 1.1: *Weaving* and *tangling* the literate program to generate documentation and source code files

Obviously, a compiler does not know how to interpret a literate program where the code is contained in code chunks. The literate program must first be translated into source code the compiler understands. Knuth calls this process *tangling*. A tool reads the literate program, extracts and resolves the code chunks, and finally outputs a source code file for the compiler. Analogously, there is a *weaving* process which takes a literate program and generates a documentation document out of it, mostly LaTeX or HTML. The same literate program is used to extract both the source code and the documentation. Figure 1.1 illustrates this process.

1.3 Pipes

In Unix-like systems, pipes are a mean of inter-process communication. Processes use pipes to communicate with each other. One process writes data into a pipe, another process reads that data from the pipe. A command pipeline in a Unix shell is a high-level usage of pipes. The pipeline

```
dmesg | grep eth0 | wc -l
```

consists of three commands. The shell starts all of them at the same time and connects them using pipes. A pipe is set up to redirect the standard output of the `dmesg` command to the standard input of the `grep` command, which results in `grep` reading the output of `dmesg`. The same is done between `grep` and `wc`. Because `wc` is last in the pipeline, its output is printed by the shell. In the end, this pipeline prints the number of kernel messages containing the term `eth0`.

There are two types of pipes: anonymous pipes and named pipes.

An anonymous pipe is created by a process using the `pipe` function. This function returns two file descriptors—the first one connected to the read end, the second one connected to the write end of the pipe. Because file descriptors are used to specify a pipe, a process can only communicate with processes it shares these file descriptors with—namely with its ancestor processes, its child processes, and with itself. Thus, anonymous pipes cannot be used to allow two arbitrary processes to communicate since they do not share file descriptors.

A named pipe, or a FIFO (first in, first out), appears as a special file in the file system. Any process with sufficient access privileges can open that file for reading or writing. After opening, a named pipe has the same semantics as an anonymous pipe. Even though it is present in the file system, data written into a named pipe is not persisted on the file system. The main purpose of the file system entry is to address the named pipe.

1.4 Scope of the Thesis

In the thesis the Ulix operating system is extended by adding support for both anonymous and named pipes to the kernel. The implementation is based on Ulix version 0.12. First, in chapter 2, a summary of the semantics of a pipe according to the POSIX specification is given. The design and architecture of the implementation is presented in chapter 3. The main part of the thesis is dedicated to the implementation in chapter 4, which, like Ulix itself, is written using Literate Programming. The final chapter 5 gives a summary of the work done in the thesis.

Chapter 2

Semantics of a Pipe

This chapter describes the semantics of a pipe. The description is based on the POSIX.1-2008 specification. [IG08]

2.1 Data Flow

A pipe is a unidirectional, or one-way, communication channel between processes with a read end and a write end. Data written into the write end of the pipe is read at the read end. If two processes A and B want to communicate with each other in a bidirectional manner, they have to use two pipes: one to send data from A to B, and another one to send data from B to A.

Data sent through a pipe is buffered by the kernel, it is never directly copied from the writing process' memory to the reading process' memory. A pipe buffer must be at least 512 bytes in size. In the Linux kernel, for example, the size of a pipe buffer can be up to 64 kilobytes.[1]

A pipe has FIFO (first in, first out) characteristics. Data is always read in the same order it is written into a pipe.

2.2 Creating and Opening a Pipe

An anonymous pipe is created by issuing the `pipe` function.

5 ⟨*POSIX:* `pipe` *function prototype* 5⟩≡

```
  int pipe(int fildes[2]);
```

[1]See `PIPE_BUF` defined in `include/uapi/linux/limits.h` and `PIPE_DEF_BUFFERS` defined in `include/linux/pipe_fs_i.h` of the Linux 3.18 source code

The function expects an `int` array of size two as its only argument. Upon returning, the first element of that field contains the file descriptor for the read end of the pipe, and the second element contains the file descriptor for the write end of the pipe. A successful call returns 0, on error, -1 is returned. There is no need to explicitly open a pipe for reading or writing, the returned file descriptors are ready to be used.

To create a named pipe, the `mkfifo` function is used.

6 ⟨*POSIX:* `mkfifo` *function prototype* 6⟩≡

```
int mkfifo(const char *path, mode_t mode);
```

As named pipes have a name on the file system, the first argument `path` of the function specifies the path on the file system where that name shall be created. The file entry is created with the mode given in the `mode` argument. On success, 0 is returned, an error is indicated by a return value of -1.

In contrast to the `pipe` function to create and open an anonymous pipe at the same time, `mkfifo` just creates a named pipe. A named pipe must explicitly be opened by calling the `open` function. The second argument of `open` is a flag parameter, which is used to tell the kernel whether to open the read end or the write end of the pipe. The flag `O_RDONLY` opens the read end, while `O_WRONLY` opens the write end of the named pipe. Trying to open a named pipe for both reading and writing by setting the `O_RDWR` flag returns an error.

The call to `open` may block. When opening a pipe for reading and there is no other process which has opened the pipe for writing (i.e. there is no file descriptor connected to the write end), `open` blocks until a process requests to open the pipe for writing. Analogously, when a process wants to open a pipe for writing and there are no readers, `open` blocks, too.

Theoretically, there is no limit on the number of processes which can open a single named pipe for either reading or writing. It is also valid for a process to open a named pipe for reading and writing by calling `open` twice. Once opened, an anonymous pipe and a named pipe behave identically, there is no difference between them. The functions for reading, writing, and closing a pipe all use the file descriptor returned by either `pipe` or `open` to refer to the pipe.

2.3 Reading from a Pipe

Reading from a pipe is done using the standard `read` function.

7a ⟨*POSIX:* `read` *function prototype* 7a⟩≡
```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Reading from a pipe is different from reading a regular file. If there is no data in the pipe buffer, the `read` call blocks until there is data available. If the buffer is empty and all writers closed their pipe file descriptor, `read` returns 0 to indicate EOF (end of file), otherwise the amount of bytes read is returned. The function may read less bytes than requested in the `nbyte` argument if there is not sufficient data available for immediate reading.

Because reading drains the pipe buffer, the number of free bytes in the buffer increases, so the kernel unblocks all processes which are blocked at a write call because there was not enough free space in the buffer to complete their write operation. These processes will then attempt to continue their write the next time the scheduler activates them.

2.4 Writing into a Pipe

To write into a pipe, the `write` function is used.

7b ⟨*POSIX:* `write` *function prototype* 7b⟩≡
```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

If there is not enough free space in the pipe buffer to write `nbyte` bytes, the call to `write` blocks until there is sufficient space available. Eventually, `write` returns `nbyte`, the amount of bytes written. If during writing all reading processes close their file descriptor, the writing process receives a `SIGPIPE` signal, and `write` returns with an error if the process chooses to ignore that signal.

Similar to reading, the kernel unblocks all processes blocked at a read call, as writing fills the buffer and processes may now be able to read from the pipe.

The POSIX standard demands writes of less than or equal to the size of the pipe buffer to be atomic (see section 2.1 for the pipe buffer size). In this case, a write call must append the data to the buffer as a single contiguous part. It is not allowed to interleave the write with writes of other processes.

For operations larger than the pipe buffer size, interleaving is legal behavior. The kernel may split the operation into multiple partial writes, and between them, other processes may read from and write into the pipe.

2.5 Closing a Pipe

Since an open pipe is referenced by a file descriptor, the standard `close` function is used to close it.

8 ⟨*POSIX:* `close` *function prototype* 8⟩≡
```
  int close(int fildes);
```

If there are no other readers and writers connected to the pipe, the final `close` destroys the pipe. Thus, all buffered data that was written into a pipe, but has not been read yet, is lost. For a named pipe, the file system entry created with the `mkfifo` function is not deleted. That entry can be unlinked (removed) with the `unlink` function.

2.6 Pipe Status

Like a regular file, a pipe also has status information. This information can be queried with the `stat` and `fstat` functions. As the `stat` function requires a path to the file for which to return status information, it is used for a named pipe. The `fstat` function takes a file descriptor in favor of a file path and thus is used to receive status information for an anonymous pipe or an opened named pipe.

When data is read from a pipe, its *access time*, or *atime* for short, is set to the current time. When data is written into a pipe, its *modification time* (*mtime*) and *change time* (*ctime*) are updated.

2.7 Unsupported File Operations

Not all functions which operate on file descriptors can be used with a pipe. In ULIX, there are three functions which return an error when called with a file descriptor associated with a pipe.

The `lseek` function sets the read/write position of an opened file. Since a pipe has FIFO semantics, the read position is always the beginning of the pipe buffer and the write position is always the end of the buffer's content. Both cannot be changed.

The `truncate` function and its file descriptor companion `ftruncate` truncate a file to a given length. The POSIX standard does not specify the behavior of these functions when called with a path to a named pipe or a pipe file descriptor, but the Linux, FreeBSD, and Mac OS X operating systems all return with an error in that case. When considering that these functions modify the pipe's content and violate the first-in first-out semantic, this is a sane decision.

Chapter 3

Implementation Design

3.1 Unifying Anonymous and Named Pipes

As described in chapter 2, anonymous and named pipes behave similarly once they are created or opened. To simplify the implementation of both in the operating system, it makes sense to unify them as much as possible. Ideally, the implementation code should not have to determine the type of the pipe and run different code paths depending on whether it is an anonymous or named pipe.

The main difference between a named and an anonymous pipe is that a named pipe has an entry in the file system. In the Minix file system, the one used in ULIX as the root file system, the file system entry is an inode. Using the inode to address a named pipe is not its only purpose, it is also used to store status information about the pipe (see section 2.6 Pipe Status). But it is not that only a named pipe has status information, an anonymous pipe has it, too. Because having two different locations to keep status information for both pipe types does not help unifying them, we also associate an anonymous pipe with an inode. This way, we make an anonymous pipe to look like a named pipe, with the only difference that the inodes are stored in different locations. Most implementation code does not have to check the type of the pipe and run different code. This simplifies the implementation and makes sure both kinds of pipes behave identically.

3.2 The Pipe File System

For now, we do not have a location where we store the inodes for an anonymous pipe. For a named pipe, the location of the inode is obvious since the user specifies a path when running the `mkfifo` function or command line program, but for anonymous pipes we have to choose a location. We could store the inodes in a directory on the root file system, `/var/run/pipes` for example, but doing so has two problems. First, the root file system is persistent, which means that the pipe inodes would be persisted, too. Since anonymous pipes are memory-only, they should not survive system reboots. Second, and most important, we would have to prevent users from reading, writing, and deleting these inodes. Because the Unix file access permissions would not be sufficient here since the root user could always change the permissions, we would have to modify the file system code in the kernel in order to prevent these actions.

The solution to this is to create a separate file system—the pipe file system—which only resides in memory and is used to store inodes of anonymous pipes. Since it is memory-only, its content does not survive reboots, and because it is a new file system, we can decide how reading, writing, and all other file operations are implemented. It is similar to the `/proc` file system on Linux system in the way that its files do not exist on disk and that it allows to retrieve runtime information. [TB15]

Alongside being the place where we store the inodes for anonymous pipes, the pipe file system also acts as a bridge between user mode and the kernel-internal pipe data structures. Programs in user land refer to pipes using file descriptors. The pipe file system provides these file descriptors and maps them to the underlying internal pipe data structure. Its job is also to manage the lifecycle of a pipe. It creates new pipes, allocates inodes for anonymous pipes, and when it determines that a pipe is no longer used, it takes care of the pipe's disposal.

The pipe file system is a real file system in the sense that it has a directory tree and is mounted into the root file system of the OS. Each anonymous pipe created is represented by a file entry in the pipe file system. Using these entries, it is possible to debug and introspect anonymous pipes from outside the process which created them. Figure 3.1 shows a possible listing of the root directory of the pipe file system with entries for two anonymous pipes.

The file mode `f` (in contrast to `p` used in other operating systems) indicates a named pipe (FIFO) and the size represents the number of bytes inside the pipe buffer.

```
1 dr-xr-x---  1    0    0        0  1 Jan  1970 .
2 dr-xr-x---  1    0    0        0  1 Jan  1970 ..
3 frw-------  1 1000  100        5 19 Mar  2015 3
4 frw-------  1 1000  100       16 19 Mar  2015 4
```

Figure 3.1: Possible listing of the root directory of the pipe file system

The idea to back the implementation of pipes by a file system is not new. The Linux kernel uses a special file system called *pipefs* to implement pipes, but in contrast to our pipe file system, the one in Linux is not accessible in user land as it is only mounted in kernel space. In fact, the Linux kernel has many more special file systems, for example it also has a *sockfs* for sockets. Considering the Unix philosophy *Everything is a file*, it makes sense to manage them with file systems. [BC02]

3.3 The Generic Pipe Data Structure

The fundamental data structure in our implementation is a generic pipe. Since we unified anonymous and named pipes, the generic pipe is used for both types. The generic pipe implements the semantics described in chapter 2: it provides the buffer for the data, it provides an interface for reading and writing, taking into account whether it is an atomic or non-atomic operation, and synchronizes access to the pipe data structure including blocking and unblocking of threads. The generic pipe is a kernel-only data structure, it is not exported to user land and user mode programs cannot use it. The higher-level interfaces for interacting with pipes use the generic pipe data structure and functions.

3.4 Architecture Overview

The implementation of pipes in Ulix can be divided into three layers. The highest layer, the layer users interact with, is the virtual file system (VFS). It is already present in Ulix and is responsible for dispatching the generic file operation system calls like `read`, `write`, `open`, and `close` to their respective lower-level implementation. In case of a pipe, when `read` is called with a pipe file descriptor, the VFS delegates to the `pipefs_read` function of the pipe file system, which forms the intermediate layer. The pipe file system, in turn, calls the `pipe_read` function of the underlying generic pipe. Figure 3.2 illustrates the call path.

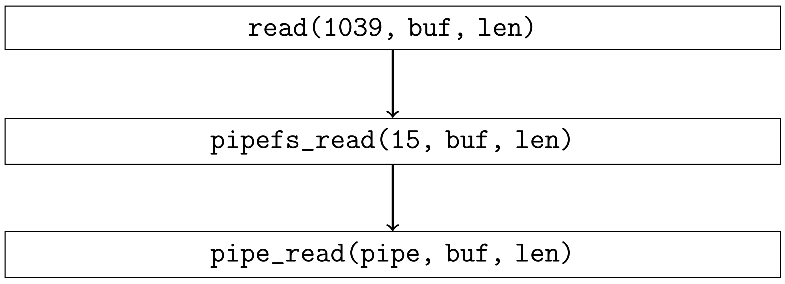

Figure 3.2: Call path when reading from a pipe

The system calls for creating pipes (`pipe` and `mkfifo`) do not belong to the VFS, but are still considered to be part of the highest layer since they provide an interface from user mode to the kernel.

Chapter 4

Implementation

4.1 Generic Pipe

4.1.1 The `pipe` Data Structure

The `pipe` structure describes a generic pipe.

14a ⟨*type definitions* 14a⟩≡
```
struct pipe {
  ⟨pipe structure elements 14b⟩
};
```

In section 3.1 we unified anonymous and named pipes by associating both with an inode. We store a reference to that inode. As we design the pipe file system to look like a Minix file system, we reuse the internal data structure for a Minix inode already present in Ulix.

14b ⟨`pipe` *structure elements* 14b⟩≡
```
struct int_minix2_inode *inode;
```

A pipe is a buffered communication channel, so we need a pointer to memory used for the buffer. We keep track of the number of bytes inside the buffer.

14c ⟨`pipe` *structure elements* 14b⟩+≡
```
void *buf;
size_t len;
```

Multiple processes may open a single pipe for reading or writing, but only one process is allowed to access the pipe at the same time. We use a mutex to synchronize access. Further, as opening a named pipe and I/O operations may block, we need several queues to keep track of processes blocked at those operations. To be able to tell when a pipe is no longer used and when opening is allowed, we count the number of reading and writing processes as well as the number of processes wanting to open the pipe.

15a ⟨`pipe` *structure elements* 14b⟩+≡
```
unsigned int nreaders, nwriters, nopenread, nopenwrite;
lock lock;
blocked_queue read_bq, write_bq, open_bq;
```

Some elements need to be initialized before they can be used. The `pipe_init` function initializes them.

15b ⟨*function prototypes* 15b⟩≡
```
int pipe_init(struct pipe *);
```

15c ⟨*function implementations* 15c⟩≡
```
int pipe_init(struct pipe *p) {
  ⟨initialize pipe structure 15d⟩
  return 0;
}
```

At the beginning, a pipe has no reading and no writing processes, no processes wanting to open the pipe and no data buffered.

15d ⟨*initialize* `pipe` *structure* 15d⟩≡
```
p->nreaders = p->nwriters = 0;
p->nopenread = p->nopenwrite = 0;
p->len = 0;
```

For the buffer we request a new memory page from the kernel. We define the size of the pipe buffer to be the size of a memory page, so allocating a single page is all we have to do.

15e ⟨*constants* 15e⟩≡
```
#define PIPE_BUF PAGE_SIZE
```

15f ⟨*initialize* `pipe` *structure* 15d⟩+≡
```
p->buf = request_new_page();
if (p->buf == NULL) return -1;
p->len = 0;
```

The block queues have to be initialized, too.

15g ⟨*initialize* `pipe` *structure* 15d⟩+≡
```
initialize_blocked_queue(&p->read_bq);
```

```
  initialize_blocked_queue(&p->write_bq);
  initialize_blocked_queue(&p->open_bq);
```

ULIX already provides a locking mechanism. We do not request a new lock from the kernel every time a pipe is initialized, we do it once when the operating system initializes because the lock is already needed before `pipe_init` is called.

16a ⟨*initialize kernel global variables* 16a⟩≡

```
  for (int i = 0; i < MAX_PIPES; i++) {
    if ((pipes[i].lock = get_new_lock("pipe lock")) == NULL) {
      printf("error: cannot get lock for pipe %d\n", i);
    }
  }
```

4.1.2 Creating a New Pipe

The kernel manages a fixed number of pipe structures.

16b ⟨*global variables* 16b⟩≡

```
  struct pipe pipes[MAX_PIPES] = {{ 0 }};
```

16c ⟨*constants* 15e⟩+≡

```
  #define MAX_PIPES 256
```

When creating a new pipe we iterate through the pipes and pick the first one that is not used. To tell whether a pipe is in use or not we add a `used` element to the `pipe` structure. If there is no unused pipe, we return `NULL`. In case a free pipe is found, we mark it as used, initialize it, and return a pointer to the structure.

16d ⟨`pipe` *structure elements* 14b⟩+≡

```
  int used;
```

16e ⟨*function prototypes* 15b⟩+≡

```
  struct pipe *pipe_new(struct int_minix2_inode *inode);
```

16f ⟨*function implementations* 15c⟩+≡

```
  struct pipe *pipe_new(struct int_minix2_inode *inode) {
    for (int i = 0; i < MAX_PIPES; i++) {
      struct pipe *p = &pipes[i];
      mutex_lock(p->lock);
      if (!p->used) {
        if (pipe_init(p) == -1) {
          mutex_unlock(p->lock);
          return NULL;
        }
        p->used = 1;
        p->inode = inode;
        inode->pipe = p;
        mutex_unlock(p->lock);
        return p;
      }
      mutex_unlock(p->lock);
    }
    return NULL;
  }
```

The new pipes stores a reference to the inode that is used for the pipe, but the inode references the pipe as well. The `int_minix2_inode` structure in ULIX does not have a `pipe` member, we have to add it. Unfortunately, there is no code chunk we can use to add a new member to the structure. Therefore, we replace code chunk 423a defined on page 423 in the ULIX book with the following one:

17a ⟨*code chunk 423a replacement* 17a⟩≡
```
  struct int_minix2_inode {
    ⟨external minix2 inode⟩
    int ino;
    unsigned int refcount;
    unsigned short clean;
    short device;
    ⟨int_minix2_inode structure elements 17b⟩
  };
```

By including a code chunk we can now introduce new structure elements easily.

17b ⟨`int_minix2_inode` *structure elements* 17b⟩≡
```
  struct pipe *pipe;
```

4.1.3 Reading and Writing

To read data from a pipe we add a `pipe_read` function.

17c ⟨*function prototypes* 15b⟩+≡
```
  int pipe_read(struct pipe *p, void *buf, int nbyte);
```

17d ⟨*function implementations* 15c⟩+≡

```
int pipe_read(struct pipe *p, void *buf, int nbyte) {
  ⟨pipe_read implementation 18a⟩
}
```

First, we validate the arguments and return an error if they are invalid. Reading zero bytes always succeeds. We acquire the lock to ensure we are the only process using the pipe right now.

18a ⟨pipe_read *implementation* 18a⟩≡
```
  if (nbyte < 0) { set_errno(EINVAL); return -1; }
  if (nbyte == 0) return 0;
  mutex_lock(p->lock);
```

The read function reads what is inside the pipe buffer, but it never reads more than `nbyte` bytes. If, for example, 120 bytes are requested, but there are only 100 bytes in the buffer, 100 bytes are read. The function returns immediately and does not block and wait for the remaining 20 bytes. It returns the number of bytes read. A special case is when there is no data in the buffer. In that case, the read function blocks until data is available, but only if there are any processes which have opened the pipe for writing or want to open the pipe for writing. If there is no data and there are no writers, the function leaves the reading loop and thus returns 0, indicating EOF (end of file).

18b ⟨pipe_read *implementation* 18a⟩+≡
```
  int nread = 0;
  for (;;) {
    if (p->len > 0) {
      int toread = MIN(p->len, nbyte);
      ⟨pipe_read: copy toread bytes from pipe 19a⟩
      nread = toread;
      ⟨pipe_read: unblock writers and update time 19b⟩
      break;
    } else if (p->nwriters > 0 || p->nopenwrite > 0) {
      ⟨pipe_read: block process reading from pipe 19c⟩
    } else {
      break;
    }
  }
```

We use `memcpy` to copy the pipe buffer data to user memory and to adjust the pipe buffer by moving the remaining data to the front. Strictly speaking, we should use the `memmove` function to adjust the buffer because source and destination memory may overlap, but in ULIX, `memcpy` is implemented in a way that works correctly with overlapping memory, so using it here is legitimate.

19a ⟨`pipe_read`*: copy* `toread` *bytes from pipe* 19a⟩≡
```
  memcpy(buf + nread, p->buf, toread);
  memcpy(p->buf, p->buf + toread, p->len - toread);
  p->len -= toread;
  p->inode->i_size -= toread;
```

After reading we unblock all processes which are in the pipe write block queue so they can continue their write operation since there is now space available in the buffer. We also set the inode's access time to the current system time.

19b ⟨`pipe_read`*: unblock writers and update time* 19b⟩≡
```
  while (p->write_bq.next != 0) {
    deblock(p->write_bq.next, &p->write_bq);
  }
  p->inode->i_atime = system_time;
```

If the process needs to be blocked, we release the pipe mutex before appending it to the read block queue. Once the process is unblocked and resumes, we re-acquire the lock. The ⟨*resign*⟩ code chunk is part of the ULIX scheduler.

19c ⟨`pipe_read`*: block process reading from pipe* 19c⟩≡
```
  mutex_unlock(p->lock);
  block(&p->read_bq, TSTATE_WAITIO);
  ⟨resign⟩
  mutex_lock(p->lock);
```

Finally we release the mutex and return the number of bytes read.

19d ⟨`pipe_read` *implementation* 18a⟩+≡
```
  mutex_unlock(p->lock);
  return nread;
```

Writing data into a pipe using the `pipe_write` function is basically the same as reading, we just put data inside the buffer instead of removing data. Additionally, we have to keep atomicity in mind.

19e ⟨*function prototypes* 15b⟩+≡
```
  int pipe_write(struct pipe *p, void *buf, int nbyte);
```

19f ⟨*function implementations* 15c⟩+≡

```
int pipe_write(struct pipe *p, void *buf, int nbyte) {
  ⟨pipe_write implementation 20a⟩
}
```

First, we check the arguments. Reading zero bytes always succeeds.

20a ⟨pipe_write *implementation* 20a⟩≡
```
  if (nbyte < 0) {
    set_errno(EINVAL);
    return -1;
  }
  if (nbyte == 0) return 0;
```

We also have to check whether there is at least one process reading from the pipe at the time of writing. If there is not, the `SIGPIPE` signal is sent to the writing process and -1 is returned with error `EPIPE`. We use a separate code chunk for the checking code as the code is embedded multiple times later on. We cannot just check it once at the beginning because the number of reading processes can change during writing when the writing process is blocked.

20b ⟨pipe_write*: check reading processes* 20b⟩≡
```
  if (p->nreaders == 0 && p->nopenread == 0) {
    mutex_unlock(p->lock);
    u_kill(thread_table[current_task].pid, SIGPIPE);
    set_errno(EPIPE);
    return -1;
  }
```

We use individual code paths to handle atomic and non-atomic write operations.

20c ⟨pipe_write *implementation* 20a⟩+≡
```
  mutex_lock(p->lock);
  int nwritten = 0;
  if (nbyte <= PIPE_BUF) {
    ⟨pipe_write: write atomically 20d⟩
  } else {
    ⟨pipe_write: write non-atomically 21d⟩
  }
```

If there is enough space in the buffer to write the given data as a contiguous sequence, we append the data to the buffer.

20d ⟨pipe_write*: write atomically* 20d⟩≡
```
  for (;;) {
    ⟨pipe_write: check reading processes 20b⟩
    if (PIPE_BUF - p->len >= nbyte) {
      int towrite = nbyte;
      ⟨pipe_write: append towrite bytes 21a⟩
      nwritten = nbyte;
      ⟨pipe_write: unblock readers and update times 21b⟩
      break;
    }
    ⟨pipe_write: block process 21c⟩
  }
```

21a ⟨pipe_write*: append* towrite *bytes* 21a⟩≡

```
  memcpy(p->buf + p->len, buf + nwritten, towrite);
  p->len += towrite;
  p->inode->i_size += towrite;
```

We unblock all processes blocked at reading and update the inode modification and change time.

21b ⟨pipe_write*: unblock readers and update times* 21b⟩≡

```
  while (p->read_bq.next != 0) {
    deblock(p->read_bq.next, &p->read_bq);
  }
  p->inode->i_mtime = p->inode->i_ctime = system_time;
```

If we cannot write atomically right now, we block the process.

21c ⟨pipe_write*: block process* 21c⟩≡

```
  mutex_unlock(p->lock);
  block(&p->write_bq, TSTATE_WAITIO);
  ⟨resign⟩
  mutex_lock(p->lock);
```

Writing non-atomically is simpler, we just append data to the buffer as there is free space and return once all data has been appended.

21d ⟨pipe_write*: write non-atomically* 21d⟩≡

```
  while (nwritten < nbyte) {
    ⟨pipe_write: check reading processes 20b⟩
    if (p->len < PIPE_BUF) {
      int towrite = MIN(PIPE_BUF - p->len, nbyte - nwritten);
      ⟨pipe_write: append towrite bytes 21a⟩
      nwritten += towrite;
      ⟨pipe_write: unblock readers and update times 21b⟩
    } else {
      ⟨pipe_write: block process 21c⟩
    }
  }
```

At last, we unlock the mutex, and return the number of bytes written.

22a ⟨`pipe_write` *implementation* 20a⟩+≡
```
mutex_unlock(p->lock);
return nwritten;
```

The thread state `TSTATE_WAITIO` does not exist in the default ULIX source code, so we add it.

22b ⟨*public constants* 22b⟩≡
```
#define TSTATE_WAITIO 12
```

There is no code chunk we can extend to provide a string representation for the `TSTATE_WAITIO` state. We have to replace code chunk 164b defined on page 164 in the ULIX book.

22c ⟨*code chunk 164b replacement* 22c⟩≡
```
char *state_names[13] = {
  "--", "READY", "--", "FORK", "EXIT", "WAIT4", "ZOMBY",
  "W_KEY", "W_FLP", "W_LCK", "STOPD", "W_IDE", "W_IO"
};
```

The `EPIPE` error constant is not defined yet as well.

22d ⟨*error constants* 22d⟩≡
```
#define EPIPE 32
```

4.1.4 Pipe Status

A pipe maintains several status information (see section 2.6 Pipe Status). We provide a `pipe_stat` function which fills a `stat` structure for a given pipe.

22e ⟨*function prototypes* 15b⟩+≡
```
int pipe_stat(struct pipe *pipe, struct stat *buf);
```

Almost all attributes of the `stat` structure have a corresponding element in the internal inode structure.

22f ⟨*function implementations* 15c⟩+≡
```
int pipe_stat(struct pipe *pipe, struct stat *buf) {
  struct int_minix2_inode *inode = pipe->inode;
  buf->st_dev   = inode->device;
  buf->st_rdev  = 0;
  buf->st_ino   = inode->ino;
  buf->st_mode  = inode->i_mode;
```

```
    buf->st_uid   = inode->i_uid;
    buf->st_gid   = inode->i_gid;
    buf->st_size  = inode->i_size;
    buf->st_atime = inode->i_atime;
    buf->st_ctime = inode->i_ctime;
    buf->st_mtime = inode->i_mtime;
    buf->st_nlink = inode->i_nlinks;
    return 0;
  }
```

4.1.5 Freeing a Pipe

Once a pipe has no readers, no writers, and no opening requests, we can free all the resources the pipe has allocated and put it back into the pool so they can be reused. The `pipe_free` function takes care of this.

23a ⟨*function prototypes* 15b⟩+≡

```
  int pipe_free(struct pipe *p);
```

23b ⟨*function implementations* 15c⟩+≡

```
  int pipe_free(struct pipe *p) {
    ⟨pipe_free implementation 23c⟩
  }
```

We ensure it is valid to free the pipe by checking that none of the process counting variables is greater than zero.

23c ⟨`pipe_free` *implementation* 23c⟩≡

```
  if (p->nreaders > 0 || p->nwriters > 0 ||
      p->nopenread > 0 || p->nopenwrite > 0) {
    return -1;
  }
```

After ensuring that we can actually free the pipe, we release the memory page used for the buffer and mark the pipe as unused. The `release_page` function in ULIX expects the number of the page to free as its argument. We do not have that number, we just have a pointer to the memory the page refers to, but we get it by dividing the pointer address by the size of a page. We do not release the lock itself because it is still necessary when searching for an unused pipe. There is no need to free the block queues since they are not allocated dynamically.

23d ⟨`pipe_free` *implementation* 23c⟩+≡

```
  p->used = 0;
```

```
p->inode->pipe = NULL;
p->inode->i_size = 0;
p->inode = NULL;
release_page((unsigned int)p->buf / PAGE_SIZE);
return 0;
```

Inside the `pipe_free` function it is not necessary to acquire the pipe lock because at every point the function is called, the lock has already been acquired.

4.2 Pipe File System

4.2.1 File Descriptors

While the generic pipe functions work with `struct pipe` pointers, the pipe file system works with file descriptors. We need a way to map file descriptors to their underlying generic pipe structure, and define a `pipefs_filestat` structure for this purpose. We choose the name to match with the name of the data structure `mx_filestat` in the Minix subsystem of ULIX to manage open files. Since a file descriptor for a pipe can only be used for reading or writing, not for both, we store the mode in the structure as well.

24a ⟨*type definitions* 14a⟩+≡
```
struct pipefs_filestat {
  struct pipe *pipe;
  short mode;
};
```

The operating system manages a pool of file descriptors. The array of structures is initialized with zero, so when the operating system starts, the pointer inside the structure is a null pointer and the mode is zero. We use the null pointer to determine later on whether a specific entry is used or not. A file descriptor not associated with a pipe is unused.

24b ⟨*global variables* 16b⟩+≡
```
struct pipefs_filestat pipefs_filestats[PIPEFS_MAX_FILES] = {{ 0 }};
```

24c ⟨*constants* 15e⟩+≡
```
#define PIPEFS_MAX_FILES 256
```

For now we have just defined a pool of structures, we do not have real file descriptors yet, which, in Unix-like systems, are integers. Instead of assigning each `pipefs_filestat` in the pool a unique number, we use the index of the structure in the pool array as the file descriptor. Keep in mind that these file descriptors are local to the pipe file system—the VFS layer of ULIX takes care of providing global file descriptors and translating them to local ones.

Having file descriptors in place, we are now able to provide a function which allocates a new file descriptor. We want the function to return a file descriptor for a given generic pipe and mode (either reading or writing).

25a ⟨*function prototypes* 15b⟩+≡

```
int pipefs_get_fd(struct pipe *pipe, short mode);
```

Its implementation is straightforward. We look at each entry in the `pipefs_filestats` array and identify if it is unused. Once we find a free one, we set its elements to the function's arguments and return the index which represents the file descriptor. If all entries are used, we return -1.

25b ⟨*function implementations* 15c⟩+≡

```
int pipefs_get_fd(struct pipe *pipe, short mode) {
  if (mode != O_RDONLY && mode != O_WRONLY) return -1;
  for (int i = 0; i < PIPEFS_MAX_FILES; i++) {
    struct pipefs_filestat *st = &pipefs_filestats[i];
    if (st->pipe == NULL) {
      st->pipe = pipe;
      st->mode = mode;
      ⟨pipefs_get_fd: update counters 25c⟩
      return i;
    }
  }
  return -1;
}
```

After associating a new file descriptor with a pipe, we have to update the counter of the generic pipe which track the number of readers and writers. We increment the `nreaders` or `nwriters` counter, respectively.

25c ⟨`pipefs_get_fd`*: update counters* 25c⟩≡

```
mutex_lock(pipe->lock);
if (mode == O_RDONLY) {
  pipe->nreaders++;
} else {
  pipe->nwriters++;
}
mutex_unlock(pipe->lock);
```

4.2.2 Allocating inodes for Anonymous Pipes

Before we can use file descriptors, we need a way to create pipes they can refer to. In section 4.1, where we implemented generic pipes, we defined a function `pipe_new` to get a new pipe from the pool. That function requires a pointer to an inode. When we are about to create a generic pipe to back a named pipe, we already have that inode, it is present in the file system we want that named pipe to be created on, but for anonymous pipes, we have to create an inode in the pipe file system.

Just like file descriptors, the operating system has a fixed number of inodes we can use.

26a ⟨*global variables* 16b⟩+≡
```
struct int_minix2_inode pipefs_inodes[PIPEFS_MAX_INODES] = {{ 0 }};
```

26b ⟨*constants* 15e⟩+≡
```
#define PIPEFS_MAX_INODES 256
```

The function `pipefs_get_inode` finds a free inode in the pool and returns a pointer to it. An inode is considered free when its reference counter is 0. The counter never exceeds 1. The counting variables of the generic pipe are used for detailed tracking.

26c ⟨*function prototypes* 15b⟩+≡
```
struct int_minix2_inode *pipefs_get_inode();
```

26d ⟨*function implementations* 15c⟩+≡
```
struct int_minix2_inode *pipefs_get_inode() {
  for (int i = 0; i < PIPEFS_MAX_INODES; i++) {
    struct int_minix2_inode *inode = &pipefs_inodes[i];
    if (inode->refcount == 0) {
      inode->refcount = 1;
      return inode;
    }
  }
  return NULL;
}
```

We initialize the table by setting the inode number and the number of links pointing to the inode. The first inode is used for the root directory entry (`.`) and the second is used for the parent directory entry (`..`) in the pipe file system.

27a ⟨*initialize kernel global variables* 16a⟩+≡
```
for (int i = 0; i < PIPEFS_MAX_INODES; i++) {
  pipefs_inodes[i].ino = i + 1;
  pipefs_inodes[i].i_nlinks = 1;
}

pipefs_inodes[0].refcount = 1; // .
pipefs_inodes[0].i_mode = S_IFDIR | 0550;
pipefs_inodes[1].refcount = 1; // ..
pipefs_inodes[1].i_mode = S_IFDIR | 0550;
```

4.2.3 Opening a Named Pipe

A named pipe is opened using the `open` function with the path to the named pipe and the opening mode. Because opening a file is specific to the file system, each file system provides its own implementation. In ULIX, the Minix file system is the only one we have to extend to add support for named pipes. When its `mx_open` function finds out that the inode is for a named pipe, it delegates the opening process to the `pipefs_open_named_pipe` function along with a reference to the inode and the opening mode.

27b ⟨*function prototypes* 15b⟩+≡
```
int pipefs_open_named_pipe(struct int_minix2_inode *inode,
                           short mode);
```

27c ⟨*function implementations* 15c⟩+≡
```
int pipefs_open_named_pipe(struct int_minix2_inode *inode,
                           short mode) {
  ⟨pipefs_open_named_pipe implementation 27d⟩
}
```

This function returns a file descriptor for the pipe which belongs to the given inode. If the named pipe is opened for the first time, the inode does not have a generic pipe associated and we have to allocate a new one.

27d ⟨`pipefs_open_named_pipe` *implementation* 27d⟩≡
```
struct pipe *pipe = inode->pipe;
if (pipe == NULL) {
  if ((pipe = pipe_new(inode)) == NULL) return -1;
  inode->pipe = pipe;
}
```

We use separate code paths depending on whether the pipe is opened for reading or writing. Remember that opening a named pipe may block the calling process (see section 2.2 Creating and Opening a Pipe).

28a ⟨pipefs_open_named_pipe *implementation* 27d⟩+≡

```
mutex_lock(pipe->lock);
if (mode == O_RDONLY) {
  for (;;) {
    ⟨pipefs_open_named_pipe: open for reading 28b⟩
  }
} else {
  for (;;) {
    ⟨pipefs_open_named_pipe: open for writing 29b⟩
  }
}
return -1;
```

When a process wants to open a pipe for reading, one of the following conditions must be met to open it immediately without blocking: (a) There is at least one process which has opened the pipe for writing, or (b) wants to open the pipe for writing, but is blocked, or (c) there is data inside the pipe buffer.

The latter condition may sound odd, but it is necessary and valid. Let us consider the following scenario: A process wants to open a pipe for reading and is blocked because there is no writer. Another process attempts to open the pipe for writing and succeeds. As a result, the reading process is removed from the block queue, but it is not activated yet by the scheduler, the writing process is still the only active one. The writing process continues, writes data into the pipe, closes it, and exits eventually. Now the reading process is activated by the scheduler. Without the last condition, the reading process would not be able to open the pipe for reading as there are no writing processes, but it should be, it just happened that the writer exited before the scheduler assigned CPU time to the reader.

When condition (a) or (c) is met, we just return a new file descriptor. When condition (b) is met, we additionally unblock all processes currently blocked at opening.

28b ⟨pipefs_open_named_pipe: *open for reading* 28b⟩≡

```
if (pipe->nwriters > 0 || pipe->len > 0) {
  mutex_unlock(pipe->lock);
  return pipefs_get_fd(pipe, mode);
}
if (pipe->nopenwrite > 0) {
  mutex_unlock(pipe->lock);
  int fd = pipefs_get_fd(pipe, mode);
```

```
    if (fd == -1) return -1;
    while (pipe->open_bq.next != 0) {
      deblock(pipe->open_bq.next, &pipe->open_bq);
    }
    return fd;
  }
```

If no condition is met, we increase the counter of processes wanting to open the pipe for reading, put the process into blocked state, and resign the CPU.

29a ⟨pipefs_open_named_pipe: *open for reading* 28b⟩+≡

```
  pipe->nopenread++;
  block(&pipe->open_bq, TSTATE_WAITIO);
  mutex_unlock(pipe->lock);
  ⟨resign⟩
  mutex_lock(pipe->lock);
  pipe->nopenread--;
```

When the pipe is to be opened for writing, there are just two states that do not result in blocking. The first is when there is at least one processes which already has the pipe opened for reading. In this case, we just provide a new file descriptor.

29b ⟨pipefs_open_named_pipe: *open for writing* 29b⟩≡

```
  if (pipe->nreaders > 0) {
    mutex_unlock(pipe->lock);
    return pipefs_get_fd(pipe, mode);
  }
```

The second state is when there is at least one process attempting to open the pipe for reading. We also return a file descriptor then, but additionally we unblock all processes blocked at opening.

29c ⟨pipefs_open_named_pipe: *open for writing* 29b⟩+≡

```
  if (pipe->nopenread > 0) {
    mutex_unlock(pipe->lock);
    int fd = pipefs_get_fd(pipe, mode);
    if (fd == -1) return -1;
    while (pipe->open_bq.next != 0) {
      deblock(pipe->open_bq.next, &pipe->open_bq);
    }
    return fd;
  }
```

If the pipe cannot be opened for writing right now, we adjust the respective counter and block the process.

30a ⟨pipefs_open_named_pipe*: open for writing* 29b⟩+≡
```
pipe->nopenwrite++;
block(&pipe->open_bq, TSTATE_WAITIO);
mutex_unlock(pipe->lock);
⟨resign⟩
mutex_lock(pipe->lock);
pipe->nopenwrite--;
```

4.2.4 Opening an Anonymous Pipe

The pipe file system is a real file system which is mounted at `/pipe`. Each file represents an anonymous pipe, the file name is the number of the inode associated with that pipe. For example, an anonymous pipe with inode number 5 is represented by `/pipe/5`. When one of these files is opened with the `open` function, the VFS layer delegates to the file-system-specific `pipefs_open` function.

30b ⟨*function prototypes* 15b⟩+≡
```
int pipefs_open(const char *path, short mode);
```

30c ⟨*function implementations* 15c⟩+≡
```
int pipefs_open(const char *path, short mode) {
  ⟨pipefs_open implementation 30d⟩
}
```

The function returns a new file descriptor for the anonymous pipe the path refers to with the given mode. First, we check the provided mode is valid (either read-only or write-only).

30d ⟨pipefs_open *implementation* 30d⟩≡
```
if (mode != O_RDONLY && mode != O_WRONLY) return -1;
```

To parse the inode number out of the path and get the corresponding inode structure, we write a separate function we implement later. With the inode we can access the pipe associated with it. Remember that we unified anonymous and named pipes so that an anonymous pipe looks like a named pipe. In subsection 4.2.3 we implemented a function to open a named pipe, and we reuse it here.

30e ⟨pipefs_open *implementation* 30d⟩+≡
```
struct int_minix2_inode *inode = pipefs_path_to_inode(path);
if (inode == NULL) { set_errno(EINVAL); return -1; }
return pipefs_open_named_pipe(inode, mode);
```

The parsing function takes a path and returns the file descriptor number, or -1 if the path is malformed.

31a ⟨*function prototypes* 15b⟩+≡
```
struct int_minix2_inode *pipefs_path_to_inode(const char *path);
```

After sanitizing the input by removing leading slashes and ensuring the string is not empty, we also make sure the string only contains digits. If it does, we use the `atoi` function ULIX provides to convert that string into an integer. We cannot use `atoi` directly without checking the string only consists of digits first because `atoi` ignores trailing non-digit characters and we do not want a path like `/pipe/5abc` to be valid input.

31b ⟨*function implementations* 15c⟩+≡
```
struct int_minix2_inode *pipefs_path_to_inode(const char *path) {
  while (*path == '/') path++;
  if (*path == '\0') return NULL;
  for (const char *s = path; *s != '\0'; s++) {
    if (*s < '0' || *s > '9') return NULL;
  }
  int ino = atoi((char *)path);
  if (ino < 3 || ino > PIPEFS_MAX_INODES) return NULL;
  return &pipefs_inodes[ino-1];
}
```

4.2.5 Reopening a File Descriptor

When a process is forked, the child process inherits the parent's file descriptors. In ULIX, the kernel calls the `u_reopen` function for each file descriptor of the parent process to get a new file descriptor to use for the child. As reopening is not a generic operation, each file system provides its own function for reopening a file descriptor, and the pipe file system provides one as well.

31c ⟨*function prototypes* 15b⟩+≡
```
int pipefs_reopen(int fd);
```

The function takes a file descriptor and returns a new one referring to the same underlying generic pipe. We already have a function called to get a file descriptor for a pipe (`pipefs_get_fd`), all we have to do is to call that function.

31d ⟨*function implementations* 15c⟩+≡
```
int pipefs_reopen(int fd) {
    if (fd < 0 || fd >= PIPEFS_MAX_FILES) {
      set_errno(EINVAL);
      return -1;
    }
    struct pipefs_filestat *st = &pipefs_filestats[fd];
    if (st->pipe == NULL) return -1;
    return pipefs_get_fd(st->pipe, st->mode);
  }
```

4.2.6 Reading and Writing

With file descriptors and inodes in place, we can now implement the pipe file system functions for reading and writing. The read function looks like `pipe_read`, with the only difference that it takes a file descriptor instead of a pointer to a pipe.

32a ⟨*function prototypes* 15b⟩+≡

```
  int pipefs_read(int fd, void *buf, int nbyte);
```

The implementation is simple. After checking that the given file descriptor is a valid one and making sure it refers to the read end of the pipe, we delegate to the `pipe_read` function with the generic pipe associated with the file descriptor. That is all we have to do, the read function of the generic pipe does all the heavy lifting. The main task of the `pipefs_read` function is to translate the file descriptor to the generic pipe.

32b ⟨*function implementations* 15c⟩+≡

```
  int pipefs_read(int fd, void *buf, int nbyte) {
    if (fd < 0 || fd > PIPEFS_MAX_FILES) {
      set_errno(EINVAL);
      return -1;
    }
    struct pipefs_filestat *fs = &pipefs_filestats[fd];
    if (fs->pipe == NULL || fs->mode != O_RDONLY) return -1;
    return pipe_read(fs->pipe, buf, nbyte);
  }
```

Writing is implemented in the same way.

32c ⟨*function prototypes* 15b⟩+≡

```
  int pipefs_write(int fd, void *buf, int nbyte);
```

32d ⟨*function implementations* 15c⟩+≡

```
  int pipefs_write(int fd, void *buf, int nbyte) {
    if (fd < 0 || fd > PIPEFS_MAX_FILES) {
```

```
    set_errno(EINVAL);
    return -1;
  }
  struct pipefs_filestat *fs = &pipefs_filestats[fd];
  if (fs->pipe == NULL || fs->mode != O_WRONLY) return -1;
  return pipe_write(fs->pipe, buf, nbyte);
}
```

4.2.7 Closing

When a file descriptor associated with a pipe is closed with the `close` function, the virtual file system layer calls the `pipefs_close` function.

33a ⟨*function prototypes* 15b⟩+≡
```
int pipefs_close(int fd);
```

33b ⟨*function implementations* 15c⟩+≡
```
int pipefs_close(int fd) {
  ⟨pipefs_close implementation 33c⟩
}
```

33c ⟨`pipefs_close` *implementation* 33c⟩≡
```
if (fd < 0 || fd > PIPEFS_MAX_FILES) {
  set_errno(EINVAL);
  return -1;
}
struct pipefs_filestat *st = &pipefs_filestats[fd];
mutex_lock(st->pipe->lock);
```

Closing a file descriptor means there is one file descriptor less connected to the pipe, so we update the process counters accordingly.

33d ⟨`pipefs_close` *implementation* 33c⟩+≡
```
if (st->mode == O_RDONLY) {
  st->pipe->nreaders--;
} else {
  st->pipe->nwriters--;
}
```

Now that we decreased the process counters, we check whether the pipe is still in use and release it if it is not. A pipe is considered to be no longer in use if all process counters are 0.

33e ⟨`pipefs_close` *implementation* 33c⟩+≡
```
if (st->pipe->nreaders == 0 && st->pipe->nwriters == 0 &&
    st->pipe->nopenread == 0 && st->pipe->nopenwrite == 0) {
  ⟨pipefs_close: free pipe resources 34a⟩
} else {
  ⟨pipefs_close: unblock processes 34b⟩
}
```

To free the pipe's resources, we first mark the inode as unused by setting its reference count to 0. We then call the `pipe_free` function to release resources for the generic pipe. The code chunk ⟨`pipefs_close`*: call inode onfree function* 41b⟩ is defined and explained in section 4.4 and is not relevant right now.

34a ⟨`pipefs_close`*: free pipe resources* 34a⟩≡

```
  ⟨pipefs_close: call inode onfree function 41b⟩
  st->pipe->inode->refcount = 0;
  st->pipe->inode->onfree = NULL;
  pipe_free(st->pipe);
```

If the pipe is still in use, we may need to unblock processes. More specific, if the just closed file descriptor was the last reading descriptor, we need to unblock all blocked writers, and if the just closed file descriptor was the last writing descriptor, we need to unblock all blocked readers. The unblocked processes receive an end of file (readers) or `EPIPE`/`SIGPIPE` (writers) error, respectively.

34b ⟨`pipefs_close`*: unblock processes* 34b⟩≡

```
  if (st->mode == O_RDONLY && st->pipe->nwriters > 0) {
    while (st->pipe->write_bq.next != 0) {
      deblock(st->pipe->write_bq.next, &st->pipe->write_bq);
    }
  } else if (st->mode & O_WRONLY && st->pipe->nreaders > 0) {
    while (st->pipe->read_bq.next != 0) {
      deblock(st->pipe->read_bq.next, &st->pipe->read_bq);
    }
  }
```

Finally, we can mark the file descriptor as free.

34c ⟨`pipefs_close` *implementation* 33c⟩+≡

```
  mutex_unlock(st->pipe->lock);
  st->pipe = NULL;
  return 0;
```

4.2.8 File Status

There are two functions to retrieve file status information—`stat` and `fstat`. The first takes a path to a file, the latter a file descriptor.

35a ⟨*function prototypes* 15b⟩+≡
```
int pipefs_stat(const char *path, struct stat *buf);
int pipefs_fstat(int fd, struct stat *buf);
```

We begin with the implementation of `pipefs_fstat`. From the file descriptor we derive the underlying generic pipe and call the `pipe_stat` function to fill the `stat` structure.

35b ⟨*function implementations* 15c⟩+≡
```
int pipefs_fstat(int fd, struct stat *buf) {
  if (fd < 0 || fd >= PIPEFS_MAX_FILES) {
    set_errno(EINVAL);
    return -1;
  }
  struct pipefs_filestat *st = &pipefs_filestats[fd];
  if (st->pipe == NULL) return -1;
  return pipe_stat(st->pipe, buf);
}
```

In `pipefs_stat`, taking a path instead of a file descriptor, we need to handle two special paths: the root directory (`.`) and the parent directory (`..`).

35c ⟨*function implementations* 15c⟩+≡
```
int pipefs_stat(const char *path, struct stat *buf) {
  ⟨pipefs_stat implementation 35d⟩
}
```

If the given path is one of these special cases, we fill the structure manually from the inodes we reserved for these directories (see subsection 4.2.2 Allocating inodes for Anonymous Pipes).

35d ⟨`pipefs_stat` *implementation* 35d⟩≡
```
struct int_minix2_inode *inode = NULL;
if (strequal(path, "/") || strequal(path, "/.")) {
  inode = &pipefs_inodes[0];
} else if (strequal(path, "/..")) {
  inode = &pipefs_inodes[1];
}
if (inode != NULL) {
    buf->st_ino   = inode->ino;
    buf->st_dev   = inode->device;
    buf->st_rdev  = 0;
    buf->st_mode  = inode->i_mode;
    buf->st_nlink = inode->i_nlinks;
    buf->st_uid   = inode->i_uid;
    buf->st_gid   = inode->i_gid;
    buf->st_size  = inode->i_size;
    buf->st_atime = inode->i_atime;
    buf->st_ctime = inode->i_ctime;
    buf->st_mtime = inode->i_mtime;
    return 0;
}
```

Otherwise, we try to get the inode the path refers to and delegate to `pipe_stat` with the pipe the inode is assigned with.

36a ⟨`pipefs_stat` *implementation* 35d⟩+≡

```
  inode = pipefs_path_to_inode(path);
  if (inode == NULL) { set_errno(EINVAL); return -1; }
  return pipe_stat(inode->pipe, buf);
```

4.2.9 Directory Listing

In order to support listing the contents of the pipe file system root directory, we have to implement the `getdent` function, which is part of the ULIX VFS layer.

36b ⟨*function prototypes* 15b⟩+≡

```
  int pipefs_getdent(const char *path, int index,
                     struct dir_entry *buf);
```

The `ls` program in ULIX to print a directory listing calls the `getdent` function with index 0 to get the first entry in the directory, then with index 1 to get the second entry and so forth, until it returns -1 to indicate that there are no more entries in the directory.

36c ⟨*function implementations* 15c⟩+≡

```
  int pipefs_getdent(const char *path, int index,
                     struct dir_entry *buf) {
    ⟨pipefs_getdent implementation 37a⟩
  }
```

The first two entries are designated for the directory itself (`.`) and for the parent directory (`..`). Remember that in subsection 4.2.2 we reserved the first and second inode for these entries. If the index is 0 or 1, we fill the `dir_entry` structure manually.

37a ⟨pipefs_getdent *implementation* 37a⟩≡
```
if (index == 0) {
  buf->inode = 1;
  sprintf(buf->filename, ".");
  return 0;
}
if (index == 1) {
  buf->inode = 2;
  sprintf(buf->filename, "..");
  return 0;
}
```

Each anonymous pipe is represented by a file in the root directory of the pipe file system (see section 3.2 The Pipe File System), and each anonymous pipe has an inode in the pipe file system. That inode is used to fill the `dir_entry` structure. We only list inodes that are in use, i.e. that have a reference count greater than 0. The file name is set to the inode number in decimal notation.

37b ⟨pipefs_getdent *implementation* 37a⟩+≡
```
index -= 2;
for (int i = 2; i < PIPEFS_MAX_INODES; i++) {
  struct int_minix2_inode *inode = &pipefs_inodes[i];
  if (inode->refcount == 0) continue;
  if (index == 0) {
    buf->inode = inode->ino;
    sprintf(buf->filename, "%d", inode->ino);
    return 0;
  }
  index--;
}
return -1;
```

4.3 Integration into the Ulix VFS

Ulix does not provide an interface to add new file systems to the virtual file system layer at runtime, neither does the literate program provide code chunks to integrate new file systems easily. We have to modify the original code chunks to add the pipe file system.

First, we add a constant to identify the pipe file system.

37c ⟨*constants* 15e⟩+≡
```
#define FS_PIPE 4
```

To give it a name, we have to replace the code chunk 374b on page 374 with the following.

38a ⟨*code chunk 374b replacement* 38a⟩≡
```
char *fs_names[] = { "ERROR", "minix", "fat", "dev", "pipe" };
```

In ULIX, the mount table is static. There is also no code chunk to add entries to the table, so we have to replace code chunk 369b on page 369 with the table which mounts the pipe file systems at `/pipe`.

38b ⟨*code chunk 369b replacement* 38b⟩≡
```
mount_table_entry mount_table[16] = {
  { "/",      FS_MINIX, DEV_HDA,  0 },
  { "/mnt/",  FS_MINIX, DEV_FD1,  0 },
  { "/tmp/",  FS_MINIX, DEV_HDB,  0 },
  { "/dev/",  FS_DEV,   DEV_NONE, 0 },
  { "/pipe/", FS_PIPE,  DEV_NONE, 0 },
  { { 0 } }
};
short current_mounts = 5;
```

The generic `open`, `read`, `write`, `close`, and `stat` functions all have a switch statement on the file system type, and each file system supporting that function has a case where the specific function is called. Unfortunately, there is also no code chunk to append to, we have to add a case for the pipe file system to the ULIX code manually.

The `u_open` function is defined in code chunk 376c on page 376. There we add a case for the pipe file system.

38c ⟨`u_open`*: add pipe file system* 38c⟩≡
```
case FS_PIPE: return pipefs_open(localpath, oflag);
```

`u_read` is defined in code chunk 378b on page 378.

38d ⟨`u_read`*: add pipe file system* 38d⟩≡
```
case FS_PIPE: return pipefs_read(localfd, buf, nbyte);
```

`u_write` is defined in code chunk 379a on page 379.

38e ⟨`u_write`*: add pipe file system* 38e⟩≡
```
case FS_PIPE: return pipefs_write(localfd, buf, nbyte);
```

`u_close` is defined in code chunk 382a on page 382.

38f ⟨`u_close`*: add pipe file system* 38f⟩≡
```
case FS_PIPE: return pipefs_close(localfd);
```

`u_stat` is defined in code chunk 385d on page 385.

39a ⟨`u_stat`*: add pipe file system* 39a⟩≡
```
case FS_PIPE: return pipefs_stat(localpath, buf);
```

4.4 Minix File System Support for Named Pipes

The Minix file system is the only file system in ULIX for which we have to add support for named pipes. We make use of the layered abstraction we have created, so we only have to extend one function of the Minix subsystem to integrate support for named pipes. The function we have to update is `mx_open`, which is called by the virtual file system layer when a file on a Minix file system is opened. The function looks up the corresponding inode for the given file path, opens the file, and returns a file descriptor.

In order to support opening named pipes, we have to check the inode's mode attribute. If it has the `S_IFIFO` flag set, the inode specifies a named pipe, and we delegate the opening to the `pipefs_open_named_pipe` function. Thus, the `mx_open` function does not return a file descriptor for the Minix file system, but a file descriptor for the pipe file system. All further file operations like `read` and `write` are handled by the pipe file system directly.

The `mx_open` implementation in ULIX is not designed for this purpose. At the point where we could check the inode's mode, the implementation would have already allocated a Minix file descriptor, which we would have to deallocate if the inode happened to be a named pipe. For this reason, we replace the existing function with an implementation that does the same, but has the code reordered to only allocate resources when they are actually needed.

We replace code chunk 428b on page 428 of the ULIX book with the following one:

39b ⟨*code chunk 428b replacement* 39b⟩≡
```
int mx_open(int device, const char *path, int oflag) {
  ⟨modified mx_open implementation 50⟩
}
```

The ⟨*modified* `mx_open` *implementation* 50⟩ code chunk is defined in Appendix A and includes the ⟨`mx_open`*: check for named pipe* 40a⟩ chunk at the point where the inode is available and we can check if it represents a named pipe. The code is thoroughly described in the ULIX book [EF15].

We determine whether the inode is for a named pipe by inspecting the inode's mode attribute having the `S_IFIFO` flag set. If it is, we call the `pipefs_open_named_pipe` function. `mx_open` is supposed to return a file descriptor for the Minix file system, but the `pipefs_open_named_pipe` function returns a file descriptor for the pipe file system. To make sure it is not treated as a Minix file descriptor, we turn it into a global file descriptor. The code chunk ⟨`mx_open`*: set onfree function* 41c⟩ is defined later in this section.

40a ⟨`mx_open`*: check for named pipe* 40a⟩≡
```
if (inode->i_mode & S_IFIFO) {
  int pipefd = pipefs_open_named_pipe(inode, oflag);
  if (pipefd == -1) {
    inode->refcount = 0;
    return -1;
  }
  ⟨mx_open: set onfree function 41c⟩
  return (FS_PIPE << 8) + pipefd;
}
```

Normally, the conversion into a global file descriptor is done in the `u_open` function after the file-system-specific open function returned. We have to update `u_open` so that it does not turn it into a global file descriptor if the returned one is already global (which is the case if `mx_open` is called for a named pipe inode). We replace the `FS_MINIX` case with the following:

40b ⟨`u_open`*: Minix file system case* 40b⟩≡
```
case FS_MINIX:
  fd = mx_open(device, localpath, oflag);
  if (fd == -1) return -1;
  if (fd >> 8 == 0) fd |= fs << 8;
  return fd;
```

There is one last task we have to do to finish the named pipe support implementation. Let us consider a opened named pipe where the last file descriptor is being closed. Because it is the last file descriptor referring to the pipe, we can free the pipe resources and mark the pipe inode as not in use. Before we can do that, we have to sync the inode (which only resides in memory) back to disk. We only have to synchronize inodes of named pipes, but at the time of closing, the pipe file system does not know whether it is a named or anonymous pipe. We cannot add the synchronization code in the `pipefs_close` function directly.

To solve this, we allow each inode to be associated with a function which is called when its reference count drops to 0. We add an element to the `int_minix2_inode` structure to save a pointer to that function.

41a ⟨`int_minix2_inode` *structure elements* 17b⟩+≡
```
  void (*onfree)(struct int_minix2_inode *);
```

In subsection 4.2.7 where we implemented the `pipefs_close` function, we used the code chunk ⟨`pipefs_close`*: call inode onfree function* 41b⟩ we have not defined yet. In that chunk we just call the onfree function if the inode has one.

41b ⟨`pipefs_close`*: call inode onfree function* 41b⟩≡
```
  if (st->pipe->inode->onfree != NULL) {
    st->pipe->inode->onfree(st->pipe->inode);
  }
```

When a named pipe is opened, we set the onfree function to function `mx_sync_inode_on_free`.

41c ⟨`mx_open`*: set onfree function* 41c⟩≡
```
  inode->onfree = mx_sync_inode_on_free;
```

41d ⟨*function prototypes* 15b⟩+≡
```
  void mx_sync_inode_on_free(struct int_minix2_inode *inode);
```

The `mx_sync_inode_on_free` function, which receives a pointer to the internal inode structure in its arguments, writes the inode to disk.

41e ⟨*function implementations* 15c⟩+≡
```
  void mx_sync_inode_on_free(struct int_minix2_inode *inode) {
    mx_write_inode(inode->device, inode->ino,
                   (struct minix2_inode *)inode);
  }
```

4.5 System Calls and Library Functions

4.5.1 The `pipe` System Call

An anonymous pipe is created with the `pipe` system call. The POSIX specification defines its prototype as:

41f ⟨*POSIX* `pipe` *system call prototype* 41f⟩≡
```
  int pipe(int fildes[2]);
```

The system call takes a two-element integer field as its only argument, which basically is just an `int` pointer, and places a file descriptor connected to the read end of a newly created pipe into the first element of the field and a file descriptor connected to the write end into the second element.

We define a syscall handler and install it in the operating system. The handler itself extracts the function argument and dispatches to the `u_pipe` function. This follows the convention of implementing system calls in ULIX, which also defines the `__NR_pipe` syscall number.

42a ⟨*syscall prototypes* 42a⟩≡
```
void syscall_pipe(context_t *r);
```

42b ⟨*syscall functions* 42b⟩≡
```
void syscall_pipe(context_t *r) {
  r->eax = u_pipe((int *)r->ebx);
}
```

42c ⟨*linux system calls* 42c⟩≡
```
#define __NR_pipe 42
```

42d ⟨*initialize syscalls* 42d⟩≡
```
install_syscall_handler(__NR_pipe, syscall_pipe);
```

The actual implementation is done in the `u_pipe` function.

42e ⟨*function prototypes* 15b⟩+≡
```
int u_pipe(int fds[2]);
```

42f ⟨*function implementations* 15c⟩+≡
```
int u_pipe(int fds[2]) {
  ⟨pipe system call implementation 42g⟩
}
```

Before we can create a new generic pipe, we have to allocate an inode we associate the anonymous pipe with. We defined the `pipefs_get_inode` function for this purpose.

42g ⟨`pipe` *system call implementation* 42g⟩≡
```
struct int_minix2_inode *inode = pipefs_get_inode();
if (inode == NULL) return -1;
```

Now we are able to actually get a new generic pipe. In the case it fails, we release the previously allocated inode by setting its reference counter back to zero and return. On success, we let the inode know which pipe it represents, set ownership to the current user, the file type to a FIFO, and access permissions so only the user can open, read, and write it.

43a ⟨pipe *system call implementation* 42g⟩+≡
```
  struct pipe *pipe = pipe_new(inode);
  if (pipe == NULL) {
    inode->refcount = 0;
    return -1;
  }
  inode->pipe   = pipe;
  inode->i_size = 0;
  inode->i_uid  = thread_table[current_task].uid;
  inode->i_gid  = thread_table[current_task].gid;
  inode->i_mode = S_IFIFO | S_IRUSR | S_IWUSR;
```

We call the `pipefs_get_fd` function twice to get two file descriptors—the first for reading, the second for writing—and place them directly into the user-supplied field. If one of the file descriptor allocations fails, we undo previous work and return an error.

43b ⟨pipe *system call implementation* 42g⟩+≡
```
  int rfd = pipefs_get_fd(pipe, O_RDONLY);
  if (rfd == -1) {
    ⟨pipe system call: cleanup on error 43c⟩
  }

  int wfd = pipefs_get_fd(pipe, O_WRONLY);
  if (wfd == -1) {
    pipefs_close(rfd);
    ⟨pipe system call: cleanup on error 43c⟩
  }

  fds[0] = gfd2pfd((FS_PIPE << 8) + rfd);
  fds[1] = gfd2pfd((FS_PIPE << 8) + wfd);
  return 0;
```

Cleaning up on error consists of marking the inode as not used and freeing the generic pipe.

43c ⟨pipe *system call: cleanup on error* 43c⟩≡
```
  inode->refcount = 0;
  inode->pipe = NULL;
  mutex_lock(pipe->lock);
  pipe_free(pipe);
  mutex_unlock(pipe->lock);
```

We extend the Ulix standard library with a `pipe` helper function so user mode programs have a convenient way to create an anonymous pipe.

44a ⟨*ulixlib.h* 44a⟩≡
```
#define __NR_pipe 42
int pipe(int fds[2]);
```

44b ⟨*ulixlib.c* 44b⟩≡
```
int pipe(int fds[2]) {
  return syscall2(__NR_pipe, (unsigned int)fds);
}
```

4.5.2 The `mknod` System Call and `mkfifo` Function

Creating a named pipe involves creating an inode whose `i_mode` has the flag `S_IFIFO` set. Unix-like operating systems typically provide a function to create an inode at a given path with given mode and other settings called `mknod`, for example to create device files in `/dev`. POSIX defines its prototype as:

44c ⟨*POSIX* `mknod` *prototype* 44c⟩≡
```
int mknod(const char *path, mode_t mode, dev_t dev);
```

Ulix does not have a `mknod` function, so we implement it and use it later to create a new named pipe. The system call number is the same as on Linux machines. The actual implementation is done in the `u_mknod` function.

44d ⟨*linux system calls* 42c⟩+≡
```
#define __NR_mknod 14
```

44e ⟨*syscall prototypes* 42a⟩+≡
```
void syscall_mknod(context_t *r);
```

44f ⟨*syscall functions* 42b⟩+≡
```
void syscall_mknod(context_t *r) {
  r->eax = u_mknod((char *)r->ebx, (short)r->ecx, (int)r->edx);
}
```

44g ⟨*initialize syscalls* 42d⟩+≡
```
install_syscall_handler(__NR_mknod, syscall_mknod);
```

44h ⟨*function prototypes* 15b⟩+≡
```
int u_mknod(char *path, short mode, int dev);
```

45a ⟨*function implementations* 15c⟩+≡

```
int u_mknod(char *path, short mode, int dev) {
  ⟨u_mknod implementation 45b⟩
}
```

First of all we check whether there is already an inode at the given path by calling `u_stat`, giving us status information on that inode. If the call succeeds, there is already an inode with that path and we cannot create a new one, so we return an error.

45b ⟨`u_mknod` *implementation* 45b⟩≡

```
struct stat st;
if (u_stat(path, &st) != -1) return -1;
```

The path in the function's first argument may be a relative one. When the path does not start with `/`, it is relative, and we use the `relpath_to_abspath` function already present in ULIX to resolve it into an absolute path.

45c ⟨`u_mknod` *implementation* 45b⟩+≡

```
char abspath[256] = { 0 };
if (*path != '/') {
  relpath_to_abspath(path, abspath);
} else {
  strncpy(abspath, path, sizeof(abspath) - 1);
}
```

Having the absolute path, the `get_dev_and_path` function is now able to determine the device and file system the path points to by inspecting the mount table.

45d ⟨`u_mknod` *implementation* 45b⟩+≡

```
short device, fs;
char localpath[256] = { 0 };
get_dev_and_path(abspath, &device, &fs, localpath);
localpath[sizeof(abspath) - 1] = 0;
```

The inode creation implementation depends on the file system. If the file system the path points to is Minix, we delegate to the `mx_mknod` function, all other file systems do not provide `mknod`.

45e ⟨`u_mknod` *implementation* 45b⟩+≡

```
switch (fs) {
case FS_MINIX: return mx_mknod(device, localpath, mode, dev);
default: return -1;
}
```

46a ⟨*function prototypes* 15b⟩+≡
```
int mx_mknod(int device, char *path, short mode, int dev);
```

46b ⟨*function implementations* 15c⟩+≡
```
int mx_mknod(int device, char *path, short mode, int dev) {
  ⟨mx_mknod implementation 46c⟩
}
```

The Minix `mknod` implementation first tries to get the number of the inode at the given path to ensure there is really no inode with that path and then requests a new inode. On failure, or when an inode does exist, it returns with an error.

46c ⟨`mx_mknod` *implementation* 46c⟩≡
```
int ino = mx_pathname_to_ino(device, path);
if (ino != -1) return -1;

ino = mx_request_inode(device);
if (ino == -1) return -1;
```

We initialize the newly requested inode, especially set its mode to the given one. The `dev` argument is ignored as ULIX does not support device special files.

46d ⟨`mx_mknod` *implementation* 46c⟩+≡
```
struct minix2_inode inode = { 0 };
inode.i_size   = 0;
inode.i_atime  = inode.i_ctime = inode.i_mtime = system_time;
inode.i_uid    = thread_table[current_task].uid;
inode.i_gid    = thread_table[current_task].gid;
inode.i_nlinks = 0;
inode.i_mode   = mode;
```

Finally, we write the new inode and link it in the directory.

46e ⟨`mx_mknod` *implementation* 46c⟩+≡
```
mx_write_inode(device, ino, &inode);
mx_write_link(device, ino, path);
return 0;
```

Like for every system call, we define a user mode function with the same name for convenience.

46f ⟨*ulixlib.h* 44a⟩+≡
```
#define __NR_mknod 14
int mknod(char *path, short mode, int dev);
```

47a ⟨*ulixlib.c* 44b⟩+≡
```
int mknod(char *path, short mode, int dev) {
  return syscall4(__NR_mknod, (unsigned int)path, mode, dev);
}
```

The implementation of the `mkfifo` function now does not consist of more than calling `mknod` with the `S_IFIFO` mode flag set.

47b ⟨*ulixlib.h* 44a⟩+≡
```
int mkfifo(char *path, short mode);
```

47c ⟨*ulixlib.c* 44b⟩+≡
```
int mkfifo(char *path, short mode) {
  return mknod(path, S_IFIFO | mode, 0);
}
```

We also provide a command line program called `mkfifo` to create a named pipe.

47d ⟨*mkfifo.c* 47d⟩≡
```
#include "../ulixlib.h"

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("usage: mkfifo file\n");
    exit(2);
  }
  if (mkfifo(argv[1], 0644) == -1) {
    printf("mkfifo: failed with errno %d\n", errno);
    exit(1);
  }
  exit(0);
}
```

Chapter 5

Conclusion

5.1 Testing

Because anonymous and named pipes both use the generic pipe underneath and the read and write operations are implemented at the generic pipe level, it is not necessary to test both types thoroughly.

The basic read, write, and blocking functionality is tested by creating a named pipe and two processes. One process reads from the named pipe and the other writes into the pipe. By first starting the reading process, followed by the writing process, the blocking behavior when opening a pipe is tested. The data written and read by the processes must be identical.

Synchronization and concurrent access is tested with a parent and several child processes. The parent process creates an anonymous pipe and spawns N_R reading and N_W writing processes. These child processes read from and write into the pipe inside an endless loop, respectively. A writing process writes the string *writer N* into the pipe and a reading process prints the data it reads. A reading process must always read data of the form *writer N* when synchronization and atomic writes work correctly. The test is run with different values for N_R and N_W.

To test the access of an anonymous pipe via the pipe file system, a process is started which creates an anonymous pipe and writes the string *test*. On another console, the pipe content is read and printed with `cat /pipe/3` (3 is the pipe's inode number). The output must be identical to the input string.

5.2 Summary

We have extended the Ulix kernel to support anonymous and named pipes. By implementing the pipe semantics in a generic pipe data structure—which is the basis of both pipe types—duplicate work has been avoided. The pipe file system acts as a bridge between user mode and kernel. Further, the pipe file system provides an interface to debug and introspect pipes.

The implementation has been written and documented using Literate Programming. In this way, the thesis has contributed to the Ulix operating system project to provide a Unix-like operating system written as a literate program to be used in education and teaching.

5.3 Future Work

The Ulix kernel now has support for pipes, but they are not used yet. An obvious program which would benefit from using pipes is the command line shell. The current shell in Ulix does not support command pipelines. In order to actually make use of the advantages of pipes, the kernel has to provide the `dup` and `dup2` system calls for duplicating file descriptors. These functions are needed in a pipeline to connect output and input of two programs.

Another system call not existing in Ulix yet is `fstat`. The pipe file system implementation already provides a `pipefs_fstat` function, but it is not available in user mode due to the missing system call.

In order to simplify the integration of new file systems into the kernel and to get rid of code chunk replacements and `case` statements in the VFS functions, it would be convenient to be able to register new file systems at runtime, similar to how system call and fault handlers are added.

Appendix A

Modified `mx_open` Implementation

50 ⟨*modified* `mx_open` *implementation* 50⟩≡

```
int ext_ino = mx_pathname_to_ino(device, path);
if (ext_ino == -1) {
  if ((oflag & O_CREAT) != 0) {
    ext_ino = mx_creat_empty_file(device, path, 0644);
  }
  return -1;
}

short file_already_open = false;
int i, int_ino = -1;

if (count_open_files == 0) {
  int_ino = 0;
} else {
  for (i = 0; i < MAX_INT_INODES; i++) {
    if (mx_inodes[i].ino == ext_ino && mx_inodes[i].device == device) {
      file_already_open = true;
      int_ino = i;
      break;
    }
  }
  if (int_ino == -1) int_ino = mx_get_free_inodes_entry();
}

if (int_ino == -1) return -1;
struct int_minix2_inode *inode = &mx_inodes[int_ino];

if (!file_already_open) {
  mx_read_inode(device, ext_ino, (struct minix2_inode*) inode);
  inode->ino = ext_ino;
  inode->device = device;
  inode->clean = true;
}
inode->refcount++;

⟨mx_open: check for named pipe 40a⟩

int mfd = mx_get_free_status_entry();
mx_status[mfd].int_inode = inode;
mx_status[mfd].pos       = 0;
mx_status[mfd].mode      = oflag;

if ((oflag & O_APPEND) != 0) mx_status[mfd].pos = inode->i_size;
count_open_files++;
if (!file_already_open) count_int_inodes++;
return mfd;
```

Bibliography

[BC02] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, 2nd edition.* O'Reilly & Associates, Inc., 2002.

[EF15] Hans-Georg Eßer and Felix C. Freiling. *The Design and Implementation of the ULIX Operating System.* 2015.

[IG08] The IEEE and The Open Group. POSIX.1-2008, IEEE Std 1003.1-2008, The Open Group Technical Standard Base Specifications, Issue 7, 2008.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, 1984.

[TB15] A.S. Tanenbaum and H. Bos. *Modern Operating Systems: Global Edition.* Pearson Education Limited, 2015.