KUDOS Documentation

Release CompSys-2016

DIKU

Jul 04, 2017

Contents

KUDOS is a skeleton operating system for exploring operating systems concepts. It is intended for:

1. teaching operating system concepts, and
2. to serve as a baseline for open-ended student projects.

CHAPTER 1

Introduction

The KUDOS operating system is heavily based upon the BUENOS operating system, and this documentation is heavily based upon the accompanying "BUENOS Roadmap".

BUENOS was originally developed at Aalto University, Finland. KUDOS is a continuation of the BUENOS effort at the Department of Computer Science at the University of Copenhagen (DIKU), Denmark. For more information about BUENOS visit the project homepage at: http://www.niksula.hut.fi/u/buenos/.

The KUDOS system supports multiple CPUs, provides threading and a wide variety of synchronization primitives. It also includes skeleton code for userland program support, partial support for a virtual memory subsystem, a trivial filesystem, and generic drivers for textual input and output.

Currently, KUDOS can run on top of YAMS, Yet Another MIPS Simulator, originally developed alongside BUENOS, or on an x86-64 simulator like QEMU; the latter will be the focus of this manual. All that you need to know, is that KUDOS is, at least in principle, a cross-platform operating system.

The main idea of KUDOS is to give you a real, working multiprocessor operating system kernel which is as small and simple as possible. KUDOS could be quite easily ported to other architectures; only device drivers and boot code need to be modified. A virtual machine environment is used because of easier development, static hardware settings and device driver simplicity, not because unrealistic assumptions are needed by the kernel.

If you are a student participating in an operating systems project course, the course staff has probably already set up a development environment for you. If they have not, you must acquire YAMS (see below for details) and compile it. You also need a MIPS32 ELF cross compiler to compile KUDOS for use with YAMS.

Expected Background Knowledge

Since the KUDOS system is written using the C programming language, you should be able to program in C. For an introduction to the C programming language, see the classical reference *[KR]*, or the more modern, and perhaps more accessible, *[ModernC]*.

We also expect that you are taking a course on operating systems or otherwise know the basics about operating systems. You can still find OS textbooks very handy when doing the exercises. We recommend that you get a hold of the book

"Operating Systems: Three Easy Pieces"*[OSTEP]*, if you are in a more classical mood, *[Stallings]* or *[Tanenbaum]*, or the more system approach found in *[BOH]*.

Since you are going to interact directly with the hardware quite a lot, you should know something about hardware. A classical introduction on this can be found in the book *[COD5e]*, while *[BOH]* again gives a complete perspective of computing systems.

Since kernel programming generally involves a lot of synchronization issues, a course on concurrent programming is recommended to be taken later on. One good book in this field is *[Andrews]*. These issues are also handled in the operating systems books cited above, but the approach is different.

How to Use This Documentation

This documentation is designed to be used both as read-through introduction and as a reference guide. To get most out of this document you should probably:

1. Read usage and autoref{sec:overview} (system overview) carefully.
2. Skim through the whole document to get a good overview.
3. Before designing and implementing your assignments, carefully read all chapters on the subject matter.
4. Use the document as a reference when designing and implementing your improvements.

Exercises

Each chapter in this document contains a set of exercises. Some of these are meant as simple thought challenges and some as much more demanding and larger programming exercises.

The thought exercises are meant for self study and they can be used to check that the contents of the chapter were understood. The programming exercises are meant to be possible assignments on operating system project courses.

The exercises look like this:

1. This is a theoretical exercise.
2. This is a programming task.

References

CHAPTER 2

Source Code Overview

The KUDOS source code is split into two main subdirectories:

1. *kudos* – operating system proper, and
2. *userland* – containing userland programs.

The KUDOS source code also contains a subdirectory called `tools`, containing useful tools for running KUDOS (see *Using KUDOS* for an overview), and `docs`, containing the source code for this documentation. You should not need to touch either of these subdirectories.

kudos

The kernel source code is split into subsystems stored in subdirectories. A subsystem typically consists of some C-files and a `subsys.mk`. To add a new subsystem, list it in the `MODULES` variables in `kudos/Makefile`. To add new C-files to a subsystem, list them in the `FILES` variable of the `subsys.mk`. A subsystem may also contain architecture-specific implementations in designated sub-subdirectories, such as `mips32` and `x86-64`.

Currently, the kernel contains the following subsystems:

kudos/init/

Kernel initialization and entry point. This directory contains the functions that KUDOS will execute to bootstrap itself. See bootstrapping-kudos for documentation of this subsystem.

kudos/kernel/

Thread handling, context switching, scheduling and synchronization. Also various core functions used in the KUDOS kernel reside here (e.g. `panic` and `kmalloc`). Documentation coming soon.

kudos/proc

Userland processes. Starting of new userland processes, loading userland binaries and handling exceptions and system calls from userland. See *System Calls* for documentation about the system call interface. Documentation about the rest of this subsystem is coming soon.

kudos/vm

Virtual memory subsystem. Managing the available physical memory and page tables. Documentation coming soon.

kudos/fs

Filesystem abstractions and the Trivial Filesystem (TFS). Documentation coming soon.

kudos/drivers

Low level device drivers and their interfaces. See *Device Drivers* and *Built-in Drivers* for documentation of this subsystem.

kudos/lib

Miscellaneous library code (`kwrite`, `kprintf`, various string-handling functions, a random number generator, etc.). Documentation coming soon.

kudos/util

Utilities for using KUDOS (e.g. `tfstool` used for writing the Trivial Filesystem disk files). See *Appendix* for more information about `tfstool`.

userland

Userland programs. These are not part of the kernel. They can be used to test the userland implementation of KUDOS by saving them to a Trivial Filesystem disk file and booting KUDOS with that. See *Using KUDOS* for information on how to do that.

CHAPTER 3

Using KUDOS

The KUDOS system requires the following software to run:

- Qemu
- GNU Binutils
- GNU GCC
- GNU Make

Note that the makefile for KUDOS sets `x86-64` standard target, but KUDOS also has a `mips32` target.

Compiling the kernel

You can compile the operating system by running `make` in the `kudos/` subdirectory of KUDOS. You can also type `make -C kudos/` from the root KUDOS source directory.

After compiling the system, you should have a binary named `kudos-x86_64` in that directory. This is your entire operating system, in one file!

Compiling the userland programs

Userland programs are compiled using the same compiler used for compiling KUDOS.

To compile userland binaries, go to the `userland/` subdirectory of KUDOS and run `make`. You can also type `make -C kudos/` from the root KUDOS source directory.

To run these programs in KUDOS, they need to be copied to a virtaul disk, where KUDOS can find them.

If you wish to add your own userland binary, list the source files in the `SOURCES` variable at the beginning of your `userland/Makefile`.

Writing to the virtual disk

KUDOS has a The Trivial Filesystem (TFS) implementation and a tool `tfstool` for managing TFS volumes. To get a summary of the arguments that `tfstool` accepts, you can run it without any arguments.

`tfstool` itself is documented in the *Appendix*.

By default, the file containing the TFS volume is named `store.file`.

Booting the system

Before KUDOS can be booted, you have to compile both KUDOS and the userland programs. When that is done, you have to create a virtual disk and copy the userland program to the virtual disk.

To boot KUDOS we need to open a terminal window, and change the directory to the KUDOS directory. When the directory is change, run `./run_qemu.sh`. A window should open and show you a boot menu. By pressing `e` you can change the kernel parameters that KUDOS uses. Currently KUDOS supports the following kernel parameters:

- `initprog`: the name of the file in the YAMS disk that the kernel starts at the first thread.
- `randomseed`: the initial seed for KUDOS' random number generator.

Example: Compile and run `halt`

In this subsection, we will go through the compilation and running of the `halt` userland program handed out together with KUDOS.

Once you have a version of KUDOS extracted on your system, build the kernel and the userland programs:

```
~/kudos$ make -C kudos
~/kudos$ make -C userland
```

Create a disk image called store.file with 16384 blocks (8 MB):

```
~/kudos$ ./kudos/util/tfstool create store.file 16384 disk
```

Then transfer the userland program `halt` onto the `store.file` virtual disk:

```
~/kudos$ ./kudos/util/tfstool write store.file userland/halt halt
```

To start qemu and boot qemu, first make `run_qemu.sh` executeable:

```
~/kudos$ chmod 755 run_qemu.sh
```

make sure to have the latest version of xorriso:

```
~/kudos$ sudo apt-get update
~/kudos$ sudo apt-get install xorriso
```

and finally you can run the halt program:

```
~/kudos$ ./run_qemu.sh halt
```

This should open a new qemu window and boot KUDOS.

Run `./kudos/util/tfstool list store.file` to list the files currently stored in the KUDOS TFS disk.

CHAPTER 4

How KUDOS Bootstraps

This section explains the bootup process of KUDOS from the first instruction ever executed by the CPU(s) to the point when userland processes can be started. This is **not** an introduction to *running* KUDOS in YAMS, for that, we refer you to *Using KUDOS*.

Modern CPUs have at least two modes of operation: a *kernel mode*, which allows for all assembly instructions, including *privileged instructions*, to be executed, and *user mode*, which *does not* allow privileged instructions to be executed.

Assembly code is both architecture-specific, and difficult to write and maintain. We therefore wish to keep the amount of assembly code in the kernel source code to the absolute minimum, running C code as soon as possible after boot.

In order to run C code, an expandable region of memory called a "stack" is required. When a function is called, depending upon the specific call conventions, function parameters are *pushed* onto the stack (or placed in registers), and space for local variables is allocated. The *return address* of the function, i.e. the address of the next instruction to be executed after the function completes execution is also pushed onto the stack.

The *stack pointer* is a CPU register that contains the memory address corresponding to the top of the stack. The *program counter / instruction pointer* registers contain the memory address of the next instruction to be executed by the CPU. This instruction is fetched from memory, and the CPU performs the corresponding operation on the values contained in the specified CPU registers. Machine instructions can read and write data from specified memory addresses into CPU registers.

Booting KUDOS/x86_64 with GRUB2

On boot, the BIOS, which is mapped to a specific location in memory, is run. The BIOS detects what hardware is present, placing this information in memory, and runs a bootloader on a specified device, e.g. a hard disk.

GRUB is a generic bootloader, which can be used by operating systems that support the multiboot specification, such as KUDOS. When the bootloader starts, the CPU is in 16-bit real mode.

GRUB loads a kernel image, and begins its execution at the *entry point*, which was defined when linking, by setting the instruction pointer to this memory address. It starts this execution in 32-bit *protected mode*. Protected mode provides memory protection, i.e. user and kernel modes, such that processes cannot interfere with one another's execution. This

allows for a multi-tasking operating system, as the kernel is protected from interference by processes running in user mode.

There is a clash of terminology here; a CPU in protected mode has the ability to switch between user and kernel mode, whereas real mode provides no such protection.

The entry point is `kudos/init/x86_64/_boot.S`, which sets up long mode (64 bit mode), and a stack for the execution of C code, before jumping to the architecture-specific `init` function, located in `kudos/init/x86_64/main.c`.

Starting Subsystems

To provide operating system service, various subsystems are required, in order to coordinate resource usage. These are started in the respective, architecture-specific `init` functions in `init/$ARCH/main.c`.

`stalloc` Provides permanent, static kernel memory allocation.

`polltty` Allows input from the keyboard, and text output to the display. This is a *polling* device, so getting input/putting output requires repeatedly checking the status of the keyboard/display in a busy-wait loop.

`interrupt` Allows devices and user processes to *interrupt* the CPU when they have an event that must be handled. This can be used to prevent the CPU from having to poll. The same subsystem handles *exceptions*.

`thread_table` Allows the operating system to have mutiple threads of execution. It is initialised, by creating a *thread table* to keep track of the threads. A *thread* of execution is (the state of) some running code, e.g. registers such as the instruction pointer.

`sleep_queue` A syncronization mechanism, allowing threads to wait on resources currently in use by other threads.

`semaphore` Another, high-level, inter-thread syncronization mechanism.

`device` Stores information about I/O devices, such as the TTY device, in a *device table*. In KUDOS, devices implement generic interfaces, minimizing code duplication.

`vfs` The Virtual File-System (VFS) is a generic file-system abstraction over more specific file-systems, such as the KUDOS Trivial File System (TFS).

`sheduler` Performs the actual switching of threads on and off CPU cores, saving the context of the thread, such as its registers.

`vm` Allows the operating system to place pages non-contiguously in memory, by mapping addresses from *physical addresses* to *virtual addresses*. Once this subsystem has been initialised, the `stalloc` subsystem is disabled.

Finally, a new thread is created and run on the CPU instead of the current thread. On `kudos-mips32` the other CPUs are now released from the wait-loop. The new thread executes the architecture-independent function `init_startup_thread` (defined in `kudos/init/common.c`) which sets up the last two things:

1. All filesystems implementing the VFS interface are made available, or mounted, by `vfs_mount_all`.
2. The program corresponding to the `initprog` argument given at boot is loaded into memory, and execution continues at the address of the first instruction in this program. If no `initprog` argument was given, the function `init_startup_fallback` is called instead.

CHAPTER 5

System Calls

System calls are an interface through which userland programs can call kernel functions, mainly those that are I/O-related, and thus require kernel mode privileges. Userland code cannot of course call kernel functions directly, since this would imply access to kernel memory, which would break the userland sandbox and userland programs could corrupt the kernel at their whim. This means that the system call handlers in the kernel should be written very carefully. A userland program should not be able to affect normal kernel functionality *no matter what arguments it passes to the system call* (this is called *bullet proofing* the system calls).

How System Calls Work

A system call is made by first placing the arguments for the system call and the *system call function number* in predefined registers. In KUDOS, the standard MIPS32 argument registers `a0`, `a1`, `a3`, and `a3` are used for this purpose. The system call number is placed in `a0`, and its three arguments in `a1`, `a2` and `a3`. If there is a need to pass more arguments for a system call, this can be easily achieved by making one of the arguments a memory pointer which points to a structure containing rest of the arguments. The return value of the system call is placed in a predefined register by the system call handler. In KUDOS the standard return value register `v0` is used.

After the arguments are in place, the special machine instruction `syscall` is executed. The `syscall` instruction, in one, atomic instruction, does the following:

1. sets the CPU core into **kernel mode**,
2. **disables interrupts**, and
3. causes a system call exception.

When an instruction causes an exception, while the CPU core is in user mode, control is transfered to the user exception handler (defined in `kudos/proc/$ARCH/exception.c`).

The system call exception is then handled as follows (note that not all details are mentioned here):

1. The context is saved as with any exception or interrupt.
2. As we notice that the cause of the exception was a system call, interrupts are enabled and the system call handler is called. Enabling interrupts (and also clearing the EXL bit) results in the thread running as a normal thread rather than an exception handler.

3. The system call handler gets a pointer to the user context as its argument. The system call number and arguments are read from the registers saved in the user context, and an appropriate handler function is called for each system call number. The return value is then written to the `v0` register saved in the user context.
4. The program counter in the saved user context is incremented by one instruction, since it points to the `syscall` instruction which generated this exception.
5. Interrupts are disabled (and EXL bit set), and the thread is again running as an exception handler.
6. The context is restored, which also restores the thread to user mode.

The last step above uses a "return from exception" instruction, `eret`, which in one, atomic instruction, does the following:

1. clears the exception flag,
2. **enables interrupts**,
3. sets the CPU core into **user mode**, and finally,
4. jumps to the address in the `EPC` register on MIPS co-processor 0.

Note: You cannot directly change thread/process (i.e. call scheduler) when in syscall or other exception handlers, since it will mess up the stack. All thread changes should be done through (software) interrupts (e.g. calling `thread_switch`).

System Calls in KUDOS

KUDOS userland has a wrapper function for the `syscall` instruction, so there is no need for the user to write code in assembly. In addition, some syscall wrappers, with proper handling of syscall argumets are implemented in `userland/lib.c`. These wrappers, or rather, library functions, are described below.

When implementing the system calls, the interface must remain *binary compatible* with the unaltered KUDOS. This means that the already existing system call function numbers must not be changed and that return value and argument semantics are exactly as described below. When adding system calls not mentioned below the arguments and return value semantics can of course be defined as desired.

Halting the Operating System

`void syscall_halt(void)`

> This is the only system call already implemented in KUDOS. It will unmount all mounted filesystems and then power off the machine (YAMS will terminate). This system call is the *only* method for userland processes to cause the machine to halt.

Process Related

`int syscall_spawn(char const *filename, char const **argv)`

- Create a new (child) user process which loads the file identified by `filename` and executes the code in it.
- `argv` specifies the arguments passed to the `main()` function of the new process, *not* including the name of the process itself.
- Returns the process ID of the new process, or a negative number on error.

`void syscall_exit(int retval)`

- Terminate the running process with the exit code `retval`.

- This function halts the process and never returns.
- `retval` must be positive, as a negative value indicates a system call error in `syscall_join()` (see next).

`int syscall_join(int pid)`

- Wait until the child process identified by `pid` is finished (i.e. calls `process_exit()`).
- Returns the exit code of the child, i.e., the value the child passed to `syscall_exit()` (or returned from `main()`).
- Returns a negative value on error.

File-System Related

`int syscall_read(int filehandle, void *buffer, int length)`

- Read at most `length` bytes from the file identified by `filehandle` into `buffer`.
- The read starts at the current file position, and the file position is advanced by the number of bytes actually read.
- Returns the number of bytes actually read (e.g. `0` if the file position is at the end of file), or a negative value on error.
- If the `filehandle` is 0, the read is done from `stdin` (the console), which is always considered to be an open file.
- Filehandles 1 and 2 cannot be read from, and attempt to do so will always return an error code.

`int syscall_write(int filehandle, const void *buffer, int length)`

- Write `length` bytes from `buffer` to the open file identified by `filehandle`.
- Writing starts at the current file position, and the file position is advanced by the number of bytes actually written.
- Returns the number of bytes actually written, or a negative value on error. (If the return value is less than `length` but 0, it means that some error occured but that the file was still partially written).
- If the `filehandle` is 1, the write is done to `stdout` (the console), which is always considered to be an open file.
- If the `filehandle` is 2, the write is done to `stderr` (typically, also the console), which is always considered to be an open file.
- Filehandle 0 cannot be written to and attempt to do so will always return an error code.

`int syscall_open(const char *pathname)`

- Open the file addressed by `pathname` for reading and writing.
- Returns the file handle of the opened file (non-negative), or a negative value on error.
- Never returns values 0, 1 or 2, because they are reserved for `stdin`, `stdout` and `stderr`.

`int syscall_close(int filehandle)`

- Close the open file identified by `filehandle`.
- `filehandle` is no longer a valid file handle after this call.
- Returns zero on success, other numbers indicate failure (e.g. `filehandle` is not open so it can't be closed).

`int syscall_create(const char *pathname, int size)`

- Create a file addressed by `pathname` with an initial size of `size`.
- The initial size means that at least size bytes, starting from the beginning of the file, can be written to the file at any point in the future (as long as it is not deleted), i.e. the file is initially allocated `size` bytes of disk space.
- Returns 0 on success, or a negative value on error.

`int syscall_delete(const char *pathname)`

- Remove the file addressed by `pathname` from the filesystem it resides on.
- Returns 0 on success, or a negative value on error.
- Note that it is impossible to implement a clean solution for the delete interaction with open files at the system call level. You are not expected to do that at this time (filesystem chapter has a separate exercise for this particular issue).

`int syscall_seek(int filehandle, int offset)`

- Set the file position of the open file identified by `filehandle` to `offset`.
- Returns 0 on success, or a negative value on error.

`int syscall_filecount(const char *pathname)`

- Get the number of files in a directory.
- Returns 0 on success, or a negative value on error.

`int syscall_file(const char *pathname, int nth, char *buffer)`

- Put the name of the `nth` file in the directory addressed by `pathname` into `buffer`.
- Returns 0 on success, or a negative value on error.

Exercises

1. Implement a new system call `syscall_hello` in KUDOS with the system call number `0xAAA`. As a result of issuing the system call, KUDOS should print "Hello, World!" to the terminal and return to the user.

   You will need to define this system call number in `kudos/proc/syscall.h`, handle it in `kudos/proc/syscall.c`, define a wrapper for it in `userland/lib.h`, and write the wrapper itself in `userland/lib.c`. Last, but not least, write a userland program `userland/hello.c` (similar to `userland/halt.c`) to test it.

   You can use either the polling TTY, or the interrupt-driven TTY *device driver*.

CHAPTER 6

Kernel Threads

A *thread of execution* is the execution of a sequence of instructions. The *context* of a thread is the contents of the CPU registers at a given point of execution, including things like the *program counter*, *stack pointer*, and *co-processor* registers. A thread may be *interrupted*, or *pre-empted*, its context stored in memory, only to be restored, and thread *re-entered* at a later point in time. A thread may be pre-empted to let another thread re-enter and do some other useful work. A *scheduler* decides which thread gets to go next.

On a *uniprocessor* system threads offer the illusion of having multiple, co-operating CPUs, while offering truly *concurrent* execution on a multiprocessor system.

Code for a *multithreaded* system must be written in a *re-entrant* fashion, i.e. such that execution may be interrupted and re-entered at any given point, except within otherwise demarcated *critical regions* of the code. Code for a *multiprocessor* system must furthermore ensure *exclusive access* to data structures shared across multiple threads of execution.

KUDOS is designed to be a multithreaded system. It achieves this goal in two fundamental ways: It has a *pre-emptive*, *round-robin* scheduler, and is designed to be a *symmetric multiprocessing (SMP)* system, which means that it supports multiple CPU cores, running each either own thread, while sharing the same physical memory.

Threads are a fundamental kernel construct, which allows to implement more nuanced userland threads of execution, such as userland threads and userland processes.

Kernel Threads API

The kernel threads API, defined in `kudos/kernel/thread.h` and implemented in `kudos/kernel/thread.c`, provides functions for setting up kernel threads in the kernel thread table, and for kernel threads to interact with the scheduler.

`void thread_table_init(void)` Initialisation of thread table and idle thread for threading subsystem.

The following two functions are used by a thread to create a new thread, and mark it as ready to run:

`TID_t thread_create(void (*func)(uint32_t), uint32_t arg)` Finds the first free entry in the thread table, and sets up a thread for the given function. This function does not cause the thread to be run, and the thread's resultant state is `NONREADY`.

void thread_run(TID_t t) Causes the thread's state in the thread table to be updated to READY, allowing the scheduler to allocate the thread to a CPU core.

The following function can be used by a kernel thread to manipulate itself:

void thread_switch(void) Perform voluntary context switch. An interrupt is invoked, causing the scheduler to reschedule. Interrupts must be enabled when this function is called, and the interrupt state is restored prior to the function returning.

void thread_yield(void) Macro pointing to `thread_switch`. The name "switch" is used when, for instance, the thread goes to sleep, whereas the name yielding implies no actual effect.

void thread_finish(void) Called automatically when the thread's function terminates or voluntarily by the thread to "commit suicide". Tidies up after thread.

TID t thread_get_current_thread(void) Returns the TID of the calling thread.

Controlling Kernel Threads

To keep track of threads, a thread table is used. This is a fixed size array of elements of type `thread_table_t`, each entry being a structure describing one thread. The size of this array, or maximum possible number of threads, is defined by `CONFIG_MAX_THREADS` in `kudos/kernel/config.h`. The thread table itself is `thread_table`, defined in `kudos/kernel/thread.c`. The index into `thread_table_t` corresponds to the kernel *thread id* (TID).

A `thread_table_t` has (among others) the following fields:

context_t *context Thread context, i.e. all CPU registers, including the program counter (PC) and the stack pointer (SP), which always refers to the thread's stack area.

context_t *user_context Pointer to thread's userland context; `NULL` for kernel-only threads.

thread_state_t state The current state of the thread. Possible values are: `FREE`, `RUNNING`, `READY`, `SLEEPING`, `NONREADY` and `DYING`.

pagetable_t *pagetable Pointer to the virtual memory mapping for this thread; `NULL` if the thread does not have a page table.

pid_t pid Process identifier for corresponding userland process. Thread creation sets this to a negative value.

_kthread_t thread_data Padding to 64 bytes for context switch code. If structure is modified, the architecture-specific paddings in `_thread.h` must be updated.

A thread may correspond to a userland process, and the thread table then stores the process' context, page table, and PID. For cross-architecture compatibility, architecture-specific padding is defined in the architecture-specific `_thread.h`.

As the thread table is shared between all threads, of which several may be executing concurrently, it must be protected from concurrent updates, using a kernel lock.

CHAPTER 7

Low-Level Synchronization

KUDOS is designed to support multiple threads of execution. To avoid threads getting in the way of one another, KUDOS provides a number of low-level synchronization primitives. These can be used to demarcate a critical region, and ensure exclusive access to a kernel resource.

The low-level primitives can then be used in *more advanced synchronization techniques*.

Interrupts

KUDOS has a pre-emptive, round-robin scheduler. To prevent a thread from being pre-empted it is sufficient to disable interrupts. On a *uniprocessor* system, it is therefore sufficient to demarcate a critical section as follows:

1. Include `kudos/kernel/interrupt.h` at the top of your file.
2. Declare a variable of type `interrupt_status_t`.
3. Set this variable to the return value of `_interrupt_disable`.
4. Have your, preferably short-lived, critical section.
5. Call `_interrupt_set_state` with the state stored above, i.e. restore the interrupt state.

Spinlocks

KUDOS is designed to be a *symmetric multiprocessing (SMP)* system, which means that it supports multiple CPU cores, having each their own thread of execution, while sharing the same physical memory. This can lead to a whole new class of race conditions, where multiple concurrent threads access the same memory location at (what appears to be) exactly the same time.

A spinlock is the most basic, low-level synchronization primitive for multiprocessor systems. A spinlock is acquired by repeatedly checking the lock's value until it is available (busy-waiting or "spinning"), and then setting the value to taken. This requires an atomic *test-and-set* or *read-modify-write* mechanism; the architecture specific details are covered below.

In KUDOS, a spinlock is implemented as a signed integer, having the following meanings:

Value	Meaning
0	Free
Positive	Taken
Negative	Reserved for future use

To achieve low-level interprocessor synchronization, interrupts must be disabled *and* a spinlock must be acquired. This method **must** be used in interrupt handlers, otherwise **other code may run before the interrupt is fully handled**.

Attempting to acquire a spinlock with interrupts disabled completely ties up the processor (the processor enters a busy-wait loop). To ensure fair service, other processors should release their spinlocks, and do so as quickly as possible. Code regions protected by spinlocks should be kept as short as possible.

Spinlocks should not be moved around in memory, i.e. they cannot be memory managed by the virtual memory subsystem, and so must be statically allocated in kernel memory. This should not be a problem, as spinlocks are purely a **kernel-level synchronization primitive**.

x86_64 Exchange and test

The x86_64 architecture provides a instruction called `xchg` (exchange), which exchanges two values, either from a register to a register or from a register to a memory location. The instruction is always atomic, which means no other process/processor can do the same with the same memory location, at the same time.

This mechanism can be used to implement a spinlock. The first thing to do is to load the value 1 into a register, say `rax`. We can now use the instruction `xchg` to exchange the value of the spinlock with the value of the register `rax`. Remember the spinlock is free if it equals 0 and busy if it equals to 1. Either way the spinlock know have the value 1, since it just exchanged its value with a register that was 1. If the new value of the register `rax` equals 0, it means that the spinlock was free before and the spinlock is now acquired. If on the other it equals 1, then it means that it was busy, and we have to jump back and try again.

For an implementation of a spinlock for the x86_64 architecture, see `kudos/kernel/x86_64/spinlock.S`

Spinlock API

The low-level assembly routines implement the architecture-independent interface specified in `kudos/kernel/spinlock.h`. Recall that **interrupts must always be disabled when a spinlock is held**, otherwise Bad Things™ will happen.

`void spinlock_reset(spinlock_t *slock)` Initializes the specified spinlock to be free. Should be done before any processor attempts to acquire the spinlock. This is really an alias to `spinlock_release`.

`void spinlock_acquire(spinlock_t *slock)` Acquire specified spinlock; while waiting for lock to be free, spin.

x86_64

1. Set `rax` to 1.
2. `xchg` the register `rax` and `slock`.
3. Test if the value of `rax` is 0.
4. If step 3 is False, go to step 2.

`void spinlock_release(spinlock_t *slock)` Free the specified spinlock. This does not check whether the spinlock is actually held by the processor. In general, there is no way to check this, and so requires a strict programming practice: Spinlocks should only be "released" if acquired.

x86_64
1. Write 0 to `slock`.

Exercises

1. Do we need spinlocks on a uniprocessor system?
2. Why must interrupts be disabled when acquiring and holding a spinlock? Consider the requirement that spinlocks should be held only for a very short time. Is the problem purely efficiency or will something actually break if a spinlock is held with interrupts enabled?

CHAPTER 8

Advanced Synchronization

The *low-level synchronization primitives*, such as disabling interrupts and spinlocks, can be used to implement more advanced synchronization techniques. Kernel-level KUDOS supports *sleep queues* and *semaphores*.

Sleep Queue

As a spinlock busy-waits until the resource is free, it wastes clock-cycles. Another synchronization method, a sleep queue, allows a thread to register itself as waiting on a specific resource held by another thread, and then voluntarily give up the CPU to other threads until the resource is available. The resource is identified by an address in kernel memory, so that the same address is used by both the thread holding the resource, and the thread waiting on the resource. When the resource is released by the thread holding it, the state of the thread waiting on it is updated, so that it will be eligible to run again.

API

The sleep queue API is defined in *kudos/kernel/sleepq.h*, but **should not** be used frivolously. See the following section for notes on how to *correctly use the sleep queue API*.

`void sleepq_init(void)`

Initializes the internal sleep queue data structured. Must be called on boot.

`void sleepq_add(void *resource)`

Adds the currently running thread to the sleep queue, sleeping on the given resource. To maximize multithreading, threads should sleep on exactly the resource that they need (e.g. array element instead of en entire array).

The thread does not go to sleep when calling this function; its `state` is simply set to `SLEEPING`. An explicit call to `thread_switch` is needed. See the *following section* for details.

void sleepq_wake(void *resource)

Wakes the first thread waiting for the given `resource` from the queue. If no threads are waiting for the given `resource`, do nothing.

void sleepq_wake_all(void *resource)

As `sleepq_wake`, but wakes up all threads which are waiting on the given `resource`.

Using the Sleep Queue Correctly

There is much more to using the sleep queue than just using the sleep queue API (defined in *kudos/kernel/sleepq.h*). To wait on a resource, you will need a dedicated resource spinlock. For details about why all of the below steps are necessary, see the sleep queue implementation notes below.

Sleeping on a Resource

```
Disable interrupts
Acquire the resource spinlock
While we want to sleep :
  sleepq_add ( resource ) // Thread state set to SLEEPING
  Release the resource spinlock // Spinlock cannot be held when not on CPU
  thread_switch () // Voluntarily yield CPU to other threads
  Acquire the resource spinlock // Thread state must now be RUNNING
End While
Use the resource
Release the resource spinlock
Restore the interrupt mask
```

Disabling interrupts and acquiring the resource spinlock ensures that the thread will be in the sleep queue before another thread attempts to wake it. If another thread was run, which called `sleepq_wake` before the `sleepq_add` call was complete, then the resource may be free, but the first thread would still be waiting on it!

Awaking Threads Sleeping on a Resource

```
Disable interrupts
Acquire the resource spinlock
Use the resource
If wishing to wake up something
  sleepq_wake ( resource ) or sleepq_wake_all ( resource )
End If
Release the resource spinlock
Restore the interrupt mask
```

Implementation

In KUDOS, the sleep queue is implemented as a statically-sized hashtable, `sleepq_hashtable`. This hashtable is intimately connected with the `sleeps_on` and `next` fields of the `thread_table_t`, which form part of the hashtable data structure. The hashtable itself is protected from concurrent access by multiple threads with a spinlock `sleepq_slock`. The implementation will acquire and release the `thread_table_slock` as it becomes necessary.

Interrupts must be disabled, and the `sleepq_slock` must be held, before any sleep queue operations are carried out. The user however, should merely disable interrupts, or better yet, follow the recipes given above.

Each entry in the hashtable corresponds to the hashed value of resource addresses; the same entry may correspond to multiple resources that hash to the same key. A value is a `TID_t`, corresponding to the `thread_table_t` of the first thread waiting on a resource with this key. The `sleeps_on` field of the `thread_table_t` is used to store the (non-hashed) address of the actual resource that the thread is waiting for – it is 0 if the thread is not waiting on any resource. This `next` field of the `thread_table_t`, contains the `TID_t` of the next thread waiting on a resource with this hash, if any. New threads are added to the end of this linked list, and threads are awoken from the beginning of the chain, to avoid potentially having to run through the whole list. Note that that as multiple resources will have the same hash, the first thread in the chain isn't necessarily the one awoken.

Fig. 8.1: An illustration of the sleep queue hashtable.

`void sleepq_init(void)`

Sets all hashtable values to -1 (free).

`void sleepq_add(void *resource)`

Adds the currently running thread into the sleep queue. The thread is added to the sleep queue hashtable. The thread does not go to sleep when calling this function; its `state` is simply set to `SLEEPING`. An explicit call to `thread_switch` is needed. The thread will sleep on the given `resource` address.

Implementation:

1. Assert that interrupts are disabled. Interrupts need to be disabled because the thread holds a spinlock and because otherwise the thread can be put to sleep by the scheduler before it is actually ready to do so.
2. Set the current thread's sleeps on field to the resource.
3. Lock the sleep queue structure.
4. Add the thread to the queue's end by hashing the address of given resource.
5. Unlock the sleep queue structure.

`void sleepq_wake(void *resource)`

Wakes the first thread waiting for the given `resource` from the queue. If no threads are waiting for the given `resource`, do nothing.

Implementation:

1. Disable interrupts.
2. Lock the sleep queue structure.
3. Find the first thread waiting for the given resource by hashing the resource address and walking through the chain.
4. Remove the found thread from the sleep queue hashtable.
5. Lock the thread table.
6. Set sleeps on to zero on the found thread.
7. If the thread is sleeping, add it to the scheduler's ready list by calling scheduler add to ready list.
8. Unlock the thread table.

9. Unlock the sleep queue structure.
10. Restore the interrupt mask.

`void sleepq_wake_all(void *resource)`

As `sleepq_wake`, but wakes up all threads which are waiting on the given resource.

Semaphores

Interrupt disabling, spinlocks and sleep queue provide the low level synchronization mechanisms in KUDOS. However, these methods have their limitations; they are cumbersome to use and thus error prone and they also require uninterrupted operations when doing processing on a locked resource. Semaphores are higher level synchronization mechanisms which solve these issues, and additionally can allow multiple units of a resource to be available to be accounted for. A semaphore can be thought of as a variable with an integer value. The resource protected by a binary semaphore can either be available (1), or locked (0, or a negative value indicating number of waiters). The counting semaphores implemented in KUDOS can have any value, with positive values indicating the number of units of a resource currently available. Three different operations are defined on a conceptual semaphore:

Initialization A semaphore may be initialized to any non-negative value indicating the number of concurrent accesses that may occur/units of resource available.

The P-operation (semaphore P()) decrements the value of the semaphore. If the value becomes negative, the calling thread will block by being added to the sleep queue waiting on this semaphore, until awakened by some other thread's V-operation.

The V-operation (semaphore V()) increments the value of the semaphore. If the resulting value is not positive, one thread blocking in P-operation will be unblocked.

API

The KUDOS semaphores API is defined in `kudos/kernel/semaphore.h`.

`semaphore_t *semaphore_create(int value)`

Creates a new semaphore, by finding the first unused semaphore in semaphore table, and initializes its value to the specified value.

Implementation:

1. Assert that the given value is non-negative.
2. Disable interrupts.
3. Acquire spinlock semaphore table slock.
4. Find free semaphore in the semaphore table and set its creator to the current thread.
5. Release the spinlock.
6. Restore the interrupt status.
7. Return NULL if no semaphores were available (in step 5).
8. Set the initial value of the semaphore to *value*.
9. Reset the semaphore spinlock.

10. Return the allocated semaphore.

void semaphore_destroy(semaphore_t *sem)

Destroys the given semaphore *sem*, freeing its entry in the *semaphore_table*.

void semaphore V(semaphore_t *sem)

Increments the value of `sem` by one. If the value was originally negative (there are waiters), wakes up one waiter.

Implementation:

1. Disable interrupts.
2. Acquire `sem`'s spinlock.
3. Increment the value of `sem` by one.
4. If the value was originally negative, wake up one thread from sleep queue which is sleeping on this semaphore.
5. Release the spinlock.
6. Restore the interrupt status.

void semaphore P(semaphore t *sem)

Decreases the value of `sem` by one. If the value becomes negative, block (sleep). Conceptually the value of the semaphore is never below zero, since this call returns only after the value is non-negative.

Implementation:

1. Disable interrupts.
2. Acquire `sem`'s spinlock.
3. Decrement the value of `sem` by one.
4. If the value becomes negative, start add current thread to sleep queue, waiting on this semaphore, and simultaneously release the spinlock.
5. Else, release the spinlock.
6. Restore the interrupt status.

Implementation

KUDOS semaphores are implemented in `kudos/kernel/semaphore.h`.

Semaphores are implemented as a static array of semaphore structures with the name semaphore table. When semaphores are "created", they are actually allocated from this table. A spin- lock semaphore table slock is used to prevent concurrent access to the semaphore table. A semaphore is defined by `semaphore_t`, which is a structure with three fields:

spinlock_t slock

Spinlock which must be held when accessing the semaphore data.

int value

The current value of the semaphore. If the value is negative, it indicates that thread(s) are waiting for the semaphore to be incremented. Conceptually the value of a semaphore is never below zero since calls from semaphore P() do not return while the value is negative.

TID_t

The thread ID of the thread that created this semaphore. Negative value indicates that the semaphore is unallocated (not yet created). The creator information is useful for debugging purposes.

Exercises

1. Suppose you need to implement periodic wake-ups for threads. For example, threads can go to sleep and then they are waked up every time a timer interrupt occurs. In this case a resource spinlock is not needed to use the sleep queue. Why can the functions `sleepq_add`, `sleepq_wake` and `sleepq_wake_all` be called without holding a resource spinlock in this case?
2. Some synchronization mechanisms may be used in both threads and interrupt handlers, some cannot. Which of the following functions can be called from a interrupt handler (why or why not?):
   (a) `interrupt disable()`
   (b) `interrupt enable()`
   (c) `spinlock acquire()`
   (d) `spinlock release()`
   (e) `sleepq add()`
   (f) `sleepq wake()`
   (g) `sleepq wake all()`
   (h) `semaphore V()`
   (i) `semaphore P()`

CHAPTER 9

Device Drivers

Since KUDOS is a realistic operating system, it can use hardware devices to interact with the outside world. Hardware devices include things like consoles, disks and network interface adapters.

Device drivers provide an interface between the hardware devices and the operating system. Device drivers use two hardware-provided mechanisms intensively: they depend on hardware generated interrupts and command the hardware with memory mapped I/O.

Most hardware devices generate interrupts when they have completed the previous action or when some asynchronous event, such as user input, occurs. Device drivers implement handlers for these interrupts and react to events.

Memory mapped I/O is an interface to the hardware components. The underlying machine provides certain memory addresses which are actually ports in hardware. This makes it possible to send and receive data to and from hardware components. Certain components also support block data transfers with direct memory access (DMA). In DMA the data is copied between main memory and the device without going through the CPU. Completion of DMA transfer usually causes an interrupt.

Interrupt driven device drivers can be thought to have two halves, top and bottom. The top half is implemented as a set of functions which can be called from threads to get service from the device. The bottom half is the interrupt handler which is run asynchronously whenever an interrupt is generated by the device. It should be noted that the bottom half might be called also when the interrupt was actually generated by some other device which shares the same interrupt request channel (IRQ).

Top and bottom halves of a device driver typically share some data structures and require synchronized access to that data. The threads calling the service functions on the top half might also need to sleep and wait for the device. Resource waiting (also called blocking or sleeping) is implemented by using the sleep queue or semaphores. The syncronization on the data structures however needs to be done on a lower level since interrupt handlers cannot sleep and wait for access to the data. Thus the data structures need to be synchronized by disabling interrupts and acquiring a spinlock which protects the data. In interrupt handlers interrupts are already disabled and only spinlock acquiring is needed.

Interrupt Handlers

All device drivers include an interrupt handler. When an interrupt occurs the system needs to know which interrupt handlers need to be called. This mechanism is implemented with an interrupt handler registration scheme. When the device drivers are initialized, they will register their interrupt handler to be called whenever specified interrupts occur. When an interrupt occurs, the interrupt handling mechanism will then call all interrupt handlers which are registered with the occured interrupt. This means that the interrupt handler might be called although the device has not generated an interrupt.

The registered interrupt handlers are kept in the table interrupt handlers which holds elements of type interrupt entry t. The fields of this structure are described in the following table:

Type	Name	Explanation
`device_t`	`device`	The device for which this interrupt is registered.
`uint32_t`	`irq`	The interrupt mask. Bits 8 through 15 indicate the interrupts that this handler is registered for. The interrupt handler is called whenever at least one of these interrupts has occured.
`void (*) (device_t *)`	`handler`	The interrupt handler function called when an interrupt occurs. The argument given to this function is `device`.

Fields in structure interrupt `entry_t`.

`void interrupt_register (uint32_t irq, void (*handler)(device_t *), device_t device)`

- Registers an interrupt handler for the `device`. `irq` is an interrupt mask, which indicates the interrupts this device has registered. Bits 8 through 15 indicate the registered interrupts. handler is the interrupt handler called when at least one of the specified interrupts has occured. This function can only be called during bootup.
- Implementation:
  1. Find the first unused entry in `interrupt_handlers`.
  2. Insert the given parameters to the found table entry.

`void interrupt_handle (uint32_t cause)`

- Called when an interrupt has occured. The argument `cause` contains the Cause register. Goes through the registered interrupt handlers and calls those interrupt handlers that have registered the occured interrupt.
- Implementation:
  1. Clear software interrupts.
  2. Call the appropriate interrupt handlers.
  3. Call the scheduler if appropriate.

Device Abstraction Layers

The device driver interface in KUDOS contains several abstraction layers. All device drivers must implement standard interface functions (initialization function and possibly interrupt handler) and most will also additionally implement functions for some generic device type. Three generic device types are provided in KUDOS: generic character device (`gcd`), generic block device (`gbd`) and generic network device (`gnd`). These can be thought as "superclasses" from which the actual device drivers are inherited.

Generic character device is a device which provides uni- or bidirectional bytestream. The only such device preimplemented in KUDOS is the console. Generic block device is a device which provides random read/write access to fixed

sized blocks. The only such device implemented is the disk driver. These interfaces could also be used to implement stream based network protocol or network block device, for example. The interface for generic network device is also given.

All device drivers must have an initialization function. A pointer to this function must be placed in the `drivers_available` array in `drivers/$ARCH/drivers.c`, together with a designated name and a device typecode identifier. Device typecodes which are defined in `drivers/device.h`. The system will initialize the device drivers on bootup for each device in the system by calling these initialization functions. This initialization is done by `device_init()`, found in `drivers/$ARCH/device.c`.

Device Driver Initialization

Every device driver's initialization function must return a pointer to the device descriptor (`device_t`) for this device, described in `kudos/drivers/device.h`.

Device driver initialization code is called from `init()` on bootup. The function called is:

```
void device_init(void)
```

Finds all devices connected to the system and attempts to initialize device drivers for them.

Implementation:

1. Loop through the device descriptor area of YAMS.
2. For each found device, try to find the driver by scanning through the list of available drivers (`drivers_available` in `kudos/drivers/$ARCH/drivers.c`).
3. If a matching driver is found, call its initialization function and print the match to the console. Store the initialized driver instance to the device driver table `device_table`.
4. Otherwise print a warning about an unrecognized device.

After device drivers are initialized, we must have some mechanism to get a handle of a specific device. This can be done with the `device_get` function:

```
device_t *device_get(uint32_t typecode, uint32_t n)
```

Finds initialized device driver based on the type of the device and sequence number. Returns nth initialized driver for device with type `typecode`. The sequencing begins from zero. If device driver matching the specifield type and sequence number if not found, the function returns `NULL`.

Generic Character Device

The generic character device (GCD) is an abstraction for any character-buffered (stream based) I/O device (e.g. a terminal). A GCD specifies read and write functions for the device, which have the same syntax for every GCD. Thus, when using GCD for all character device implementations, the code which reads or writes them does not have to care whether the device is a TTY or some other character device.

The generic character device is implemented as a structure with the fields described in the `gcd_t` structure in `kudos/drivers/gcd.h`.

Generic Block Device

The generic block device (GBD) is an abstraction of a block-oriented device (e.g. a disk). GBD consists of a function interface and a `request` data structure that abstracts the blocks to be handled. All functions are implemented by the

actual device driver.

The function interface is provided by the `gbd_t` data structure in `kudos/drivers/gbd.h`. To use this interface, it is necessary to describe requests in detail; for this, the `gbd_request_t` data structure is used. This structure includes all necessary information related to the reading or writing of a block.

The GBD interface supports both synchronous and asynchronous calls (see the `gbd.h` file for the practical details).

In case of asynchronous calls, the `gbd` interface functions will return immediately. This means that the user must wait on an associated kernel semaphore before continuing. Memory reserved for the request may not be released until the semaphore is released. The thread using a GBD device must be very careful especially with reserving memory from function stacks (ie. static allocation). If the function is exited before the request is served, the memory area of the request may corrupt.

In case of synchronous calls, the `gbd` interface functions will block until the request is handled. The memory of the `request` data structure may be released when control is returned.

CHAPTER 10

Built-in Drivers

KUDOS ships with several built-in working drivers. The drivers for the MIPS target are all designed to work with YAMS hardware. The purpose of this section is to show how one can create actual drivers for for the KUDOS device driver interface.

TTY

TTY roughly stands for "teletype", and in this context refers to a class of drivers that provide a low-level interface for terminal-style user interaction. KUDOS comes with both a *polling*, and *interrupt-driven* TTY driver built-in. The former works by repeatedly polling the keyboard device, which can waste precious clock-cycles, while the latter relies on an interrupt handler to wake up the kernel thread when input from the keyboard actually becomes available.

Historically, TTY drivers also implement a line discipline, and the KUDOS TTY drivers are no exception.

Polling TTY Driver

Two separate drivers are provided for the TTY (the terminal). The first one is implemented by *polling* and the other with *interrupt handlers*.

Polling means that the kernel again and again asks the (virtual, in this case) hardware if anything new has come up. Depending on interrupt handlers means that the hardware signals the kernel when a change has occurred.

The polling driver is needed when booting up, since interrupts are disabled. It is also useful in kernel panic situations, because interrupt handlers might not be relied on in such error cases.

Perhaps the easiest way to use the polling TTY driver is using the built-in functions `kwrite` and `kprintf` (defined in `kudos/lib/libc.h`). See `kudos/drivers/polltty.h` and `kudos/drivers/$ARCH/polltty.c` for the implementation and documentation of the driver itself.

Interrupt-driven TTY Driver

The interrupt driven (i.e. the *asynchronous*) TTY driver is the terminal device driver used most of the kernel terminal I/O-routines. The terminal driver has two functions to provide output to the terminal and input to the kernel. Both of these happen asynchronously. i.e. the input handling is triggered when the user presses a key on the keyboard. The output handler is invoked when some part of the kernel requests a write. The asynchronous TTY driver is implemented in `drivers/$ARCH/tty.c` and implements the **generic character device interface**.

The following functions implement the TTY driver:

`device_t *tty_init(io_descriptor_t *desc)`

Initialize a driver for the TTY defined by `desc`. This function is called once for each TTY driver present in the YAMS virtual machine.

Implementation:

1. Allocate memory for one `device_t`.
2. Allocate memory for one `gcd_t` and set `generic_device` to point to it.
3. Set `gcd->device` to point to the allocated `device_t`, `gcd->write` to `tty_write` and `gcd->read` to `tty_read`.
4. Register the interrupt handler (`tty_interrupt_handle`).
5. Allocate a structure that has (small) read and write buffers and head and count variables for them, and a spinlock to synchronize access to the structure and `real_device` to point to it. The first tty driver's spinlock is shared with `kprintf()` (i.e. the first tty device is shared with polling TTY driver).
6. Return a pointer to the allocated `device_t`.

`void tty_interrupt_handle(device_t *device)`

Handle interrupts concerning `device`. This function is never called directly from kernel code, instead it is invoked from interrupt handler.

Implementation if WIRQ (*write interrupt request*) is set:

1. Acquire the driver spinlock.
2. Issue the WIRQD into COMMAND (inhibits write interrupts).
3. Issue the Reset WIRQ into COMMAND.
4. While WBUSY is not set and there is data in the write buffer, Reset WIRQ and write a byte from the write buffer to DATA.
5. Issue the WIRQE into COMMAND (enables write interrupts).
6. If the buffer is empty, wake up the threads sleeping on the write buffer.
7. Release the driver spinlock.

Implementation if RIRQ (*read interrupt request*) is set:

1. Acquire the driver spinlock.
2. Issue the Reset RIRQ command to COMMAND. If this caused an error, panic (*serious* hardware failure).
3. Read from DATA to the read buffer while RAVAIL is set. Read *all* available data, even if the read buffer becomes filled (because the driver expects us to do this).

4. Release the driver spinlock.
5. Wake up all threads sleeping on the read buffer.

`static int tty_write(gcd_t *gcd, void *buf, int len)`

Write `len` bytes from `buf` to the TTY specified by `gcd`.

Implementation:

1. Disable interrupts and acquire driver spinlock.
2. As long as write buffer is not empty, sleep on it (release-reacquire for the spinlock).
3. Fill the write buffer from `buf`.
4. If WBUSY is not set, write `one` byte to the DATA port. (This is needed so that the write IRQ is raised. The interrupt handler will write the rest of the buffer.)
5. If there is more than one byte of data to be written, release the spinlock and sleep on the write buffer.
6. If there is more data in `buf`, repeat from step 3.
7. Release spinlock and restore interrupt state.
8. Return the number of bytes written.

`static int tty_read(gcd_t *gcd, void *buf, int len)`

Read at least one and at most `len` bytes into `buf` from the TTY specified by `gcd`.

Implementation:

1. Disable interrupts and acquire driver spinlock.
2. While there is no data in the read buffer, sleep on it (release-reacquire for the spinlock).
3. Read `MIN(len, data-in-readbuf)` bytes into `buf` from the read buffer.
4. Release spinlock and restore interrupt state.
5. Return the number of bytes read.

Disk Driver

The disk driver implements the Generic Block Device (GBD) interface. The driver is interrupt-driven and provides both synchronous (blocking) and asynchronous (non-blocking) operating modes for request. The driver has three main parts:

- An initialization function, which is called in startup when a disk is found.
- An interrupt handler.
- Functions which implement the GBD interface (read, write and information inquiring).

The disk driver maintains a queue of pending requests. The queue insertion is handled in disk scheduler, which currently just inserts new requests at the end of the queue. This queue, as well as access to the disk device, is protected by a spinlock. The spinlock and queue are stored in driver's internal data. The internal data also contains a pointer to the currently served disk request.

The disk driver is implemented and documented in `kudos/drivers/$ARCH/disk.c`. Note how the fields modified by both the inquiring and interrupt-ready parts of the driver are marked as `volatile`, so that the compiler won't optimize access to them (store them in registers and assume that value is valid later, which would be a flawed approach because of interrupts, which can change the values of the variables asynchronously).

Timer Driver

The Timer driver allows to set timer interrupts at certain intervals. The `timer_set_ticks()` C function works as a front-end for the `_timer_set_ticks` assembler function. The C function takes anumber of processor clock cycles after the timer interrupt is wanted to happen, and it passes it to the assembler function that does all work.

A timer interrupt is caused by using `CP0` registers `Count` and `Compare`. The `Count` register contains the current cycle count, and the `Compare` register contains the cycle number where the timer interrupt is to happen. The assembler function simply adds the number of cycles to the current cycle count and writes it to the `Compare` register.

The timer driver is implemented and documented in `kudos/drivers/timer.c` and `kudos/drivers/$ARCH/_timer.S`.

Metadevice Drivers

"Metadevices" is a name for those devices documented in the YAMS documentation as non-peripheral devices (the `0x100` series). They don't really interface to any specific device but rather to the system itself (the motherboard main chipset, firmware or similar). The metadevices and their drivers are very simple, and they are as follows.

See `kudos/drivers/metadev.h` and `kudos/drivers/$ARCH/metadev.c` for the implementation and description of the following metadevices.

Meminfo

The system memory information device provides information about the amount of memory present in the system.

RTC

The Real Time Clock (RTC) device provides simulated real time data, such as system uptime and clock speed. It is a wrapper to the RTC device I/O ports.

Shutdown

The (software) shutdown device is used to either halt the system by dropping to the YAMS console (firmware console) or "poweroff" the system by exiting YAMS completely.

CPU Status

Each processor has its own status device. These devices can be used to count the number of CPUs on the system or to generate interrupts on any CPU.

Exercises

1. Both `kwrite` and `kprintf` use the polling TTY driver. Why?

CHAPTER 11

Filesystems

A "filesystem" is an organization of "files" into a system, often backed by some sort of long-term storage device. Before the kernel or userland can perform file operations, the filesystem has to be properly "mounted". Modern operating systems support the mounting of multiple filesystems, and provide a virtual filesystem layer.

KUDOS is no exception.

KUDOS supports one filesystem, called the *Trivial Filesystem*. Filesystems are managed and accessed through a layer called the *Virtual Filesystem* layer which represents a union of all mounted filesystems.

The Trivial Filesystem supports only the most primitive filesystem operations and does not enable concurrent access to the filesystem. Only one request (read, write, create, open, close, etc.) is allowed to be in action at any given time. TFS enforces this restriction internally.

Filesystem Conventions

Files on filesystems are addressed by filenames. In KUDOS, filenames can have at most 15 alphanumeric characters. The full path to a file is called an absolute pathname and it must contain the volume (mount-point or filesystem) on which the file is, possibly a directory path, and finally, the name of the file within that directory.

An example of a valid filename is shell. A full absolute path to a shell might be `[disk]shell` or `[disk]bin/shell`. Here `shell` is the name of a file, `disk` is a volume name (you could also call it a disk, filesystem or mount-point). If directories are used, `bin` is a name of a directory. Directories have the same restrictions on filenames as files do (directory are really just files). Directory names in a path are separated by `/`.

Filesystem Layers

Typically a filesystem is located on a disk (but it can also be a network filesystem or even totally virtual). Disks are accessed through Generic lock Devices (see *Device Drivers*). At boot time, the system will try to mount all available filesystem drivers on all available disks through their GBDs. The mounting is done into a virtual filesystem.

Virtual Filesystem is a super-filesystem which contains all attached (mounted) filesystems. The same access functions are used to access disk, networked and fully virtual filesystems. An example of a "fully virtual filesystem" is the `proc` filesystem on Linux, which makes a range of process-related information available from under the `/proc` directory. The actual filesystem driver is recognized from the volume name part of a full absolute pathname provided to the access functions.

Files	Purpose
`vfs.[hc]`	*Virtual Filesystem* implementation
`filesystems.[hc]`	Available filesystems
`tfs.[hc]`	*Trivial Filesystem* implementation

CHAPTER 12

Virtual Filesystem

Virtual Filesystem (VFS) is a subsystem which unifies all available filesystems into one big virtual filesystem. All filesystem operations are done through it. Different mounted filesystems are referenced with names, which are called mount-points or volumes.

VFS provides a set of file access functions (see *File Operations*) and a set of filesystem access functions (see *Filesystem Operations*). The file access functions can be used to open files on any filesystem, close open files, read and write open files, create new files and delete existing files.

The filesystem manipulation functions are used to mount filesystems into VFS, unmount filesystems, and get information about mounted filesystems (e.g. the amount of free space on a volume). A mechanism for forceful unmounting of all filesystems is also provided. This mechanism is needed when the system performs shutdown, to prevent filesystem corruption.

To be able to provide these services, VFS keeps track of mounted filesystems and open files. VFS is thread-safe and synchronizes all its own operations and data structures. However TFS, which is accessed through VFS, does not provide proper concurrent access, it simply allows only one operation at a time.

Return Values

All VFS operations return non-negative values as an indication of successful operation and negative values as failures. The return value `VFS_OK` is defined to be zero, and indicates success. The rest of the pre-defined return values are negative. The full list of is as follows:

`VFS_OK` The operation succeeded.

`VFS_NOT_SUPPORTED` The requested operation is not supported and thus failed.

`VFS_INVALID_PARAMS` The parameters given to the called function were invalid and the operation failed.

`VFS_NOT_OPEN` The operation was attempted on a file which was not open and thus failed.

`VFS_NOT_FOUND` The requested file or directory does not exist.

`VFS_NO_SUCH_FS` The referenced filesystem or mount-point does not exist.

VFS_LIMIT The operation failed because some internal limit was hit. Typically this limit is the maximum number of open files or the maximum number of mounted filesystems.

VFS_IN_USE The operation couldn't be performed because the resource was busy. (Filesystem unmounting was attempted when filesystem has open files, for example.)

VFS_ERROR Generic error, might be hardware related.

VFS_UNUSABLE The VFS is not in use, probably because a forceful unmount has been requested by the system shutdown code.

Limits

VFS limits the length of strings in filesystem operations. Filesystem implementations and VFS file and filesystem access users must make sure to use these limits when interacting with VFS.

The maximum length of a filename is defined to be 15 characters plus one character for the end of string marker, i.e. `VFS_NAME_LENGTH` is set to 16.

The maximum path length, including the volume name (mount-point), possible absolute directory path and filename is defined to be 255 plus one character for the end of string marker, i.e. `VFS_PATH_LENGTH` is set to 256.

Internal Data Structures

VFS has two primary data structures: the table of all mounted filesystems and the table of open files.

The table of all filesystems, `vfs_table`, is structured as follows:

Type	Name	Description
`sempahore_t *`	`sem`	A binary semaphore used for exclusive access to the filesystems table.
`vfs_entry_t[CONFIG_MAX_FILESYSTEMS]`	`filesystems`	The filesystems table itself.

A `vfs_entry_t` itself has the following fields:

Type	Name	Description
`fs_t *`	`filesystem`	The filesystem driver for this mount-point. If `NULL`, this entry is unused.
`char[VFS_NAME_LENGTH]`	`mountpoint`	The name of this mount-point.

The table is initialized to contain only `NULL` filesystems. All access to this table must be protected by acquiring the semaphore used to lock the table (`vfs_table.sem`). New filesystems can be added to this table whenever there are free rows, but only filesystems with no open files can be removed from the table.

The table of open files (`openfile_table`) is structured as follows:

Type	Name	Description
`semaphore_t *`	`sem`	A binary semaphore used for exclusive access to this table.
`openfile_entry_t[CONFIG_MAX_OPEN_FILES]`	`files`	Table of open files.

The open files table is also protected by a semaphore (`openfile_table.sem`). Whenever the table is altered, this semaphore must be held.

An `openfile_entry_t` itself has the following fields:

Type	Name	Description
fs_t *	filesystem	The filesystem in which this open file is located. If NULL, this is a free entry.
int	fileid	A filesystem defined id for this open file. Every file in a filesystem must have a unique id. Ids do not need to be globally unique.
int	seek_position	The current seek position in the file.

If access to both tables is needed, the semaphore for vfs_table must be held before the openfile_table semaphore can be lowered. This convention is used to prevent deadlocks.

In addition to these, VFS uses two semaphores and two integer variables to track active filesystem operations. The first semaphore is vfs_op_sem, which is used as a lock to synchronize access to the three other variables. The second semaphore, vfs_unmount_sem, is used to signal pending unmount operations when the VFS becomes idle.

The initial value of vfs_op_sem is one and vfs_unmount_sem is initially zero. The integer vfs_ops is a zero initialized counter which indicates the number of active filesystem operations on any given moment. Finally, the boolean vfs_usable indicates whether VFS subsystem is in use. VFS is out of use before it has been initialized and it is turned out of use when a forceful unmount is started by the shutdown process.

VFS Operations

The virtual filesystem is initialized at the system bootup by calling the following function:

void vfs_init(void)

- Initializes the virtual filesystem. This function is called before virtual memory is initialized.
- Implementation:
  1. Create the semaphore vfs_table.sem (initial value 1) and the semaphore openfile_table.sem (initial value 1).
  2. Set all entries in both vfs_table and openfile_table to free.
  3. Create the semaphore vfs_op_sem (initial value 1) and the semaphore vfs_unmount_sem (initial value 0).
  4. Set the number of active operations (vfs_ops) to zero.
  5. Set the VFS usable flag (vfs_usable).

When the system is being shut down, the following function is called to unmount all filesystems:

void vfs_deinit(void)

- Force unmounts on all filesystems. This function must be used only at system shutdown.
- Sets VFS into unusable state and waits until all active filesystem operations have been completed. After that, unmounts all filesystems.
- Implementation:
  1. Call semaphore_P on vfs_op_sem.
  2. Set VFS usable flag to false.
  3. If there are active operations (vfs_ops > 0): call semaphore_V on vfs_op_sem, wait for operations to complete by calling semaphore_P on vfs_unmount_sem, re-acquire the vfs_op_sem with a call to semaphore_P.
  4. Lock both data tables by calling semaphore_P on both vfs_table.sem and openfile_table.sem.

5. Loop through all filesystems and unmount them.
6. Release semaphores by calling `semaphore_V` on `openfile_table.sem`, `vfs_table.sem` and `vfs_op_sem`.

To maintain count on active filesystem operations and to wake up pending forceful unmount, the following two internal functions are used. The first one is always called before any filesystem operation is started and the latter when the operation has finished.

`static int vfs_start_op(void)`

- Start a new VFS operation. A VFS operation is anything that touches a filesystem.
- Returns `VFS_OK` if the operation can continue, or error (negative value) if the operation cannot be started (VFS is unusable). If the operation cannot continue, it should not later call `vfs_end_op`.
- Implementation:
  1. Call `semaphore_P` on `vfs_op_sem`.
  2. If VFS is usable, increment `vfs_ops` by one.
  3. Call `semaphore_V` on `vfs_op_sem`.
  4. If VFS was usable, return `VFS_OK`, else return `VFS_UNUSABLE`.

`static void vfs_end_op(void)`

- End a started VFS operation.
- Implementation:
  1. Call `semaphore_P` on `vfs_op_sem`.
  2. Decrement `vfs_ops` by one.
  3. If VFS is not usable and the number of active operations is zero, wake up pending forceful unmount by calling `semaphore_V` on `vfs_unmount sem`.
  4. Call `semaphore_V` on `vfs_op_sem`.

File Operations

The primary function of the virtual filesystem is to provide unified access to all mounted filesystems. The filesystems are accessed through file operation functions.

Before a file can be read or written it must be opened by calling `vfs_open`:

`openfile_t vfs_open(char *pathname)`

- Opens the file addressed by `pathname`. The name must include both the full pathname and the filename. (e.g. `[root]shell.mips32`)
- Returns an open file identifier. Open file identifiers are non-negative integers. On error, negative value is returned.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Parse `pathname` into volume name and filename parts.
  3. If filename is not valid (too long, no mount point, etc.), call `vfs_end_op` and return with error code `VFS_ERROR`.

4. Acquire locks to the filesystem table and the open file table.
5. Find a free entry in the open file table. If no free entry is found (the table is full), free the locks, call `vfs_end_op`, and return with the error code `VFS_LIMIT`.
6. Find the filesystem specified by the volume name part of the pathname in the filesystem table. If the volume is not found, return with the same procedure as for a full open file table, except that the error code is `VFS_NO_SUCH_FS`.
7. Allocate the found free open file entry by setting its filesystem field.
8. Free the filesystem and the open file table locks.
9. Call the filesystem's internal open function. If the return value indicates an error, lock the open file table, mark the entry free and free the lock. Call `vfs_end_op` and return the error given by the filesystem.
10. Save the file identifier returned by the filesystem in the open file table.
11. Set file's seek position to zero (beginning of the file).
12. Call `vfs_end_op`.
13. Return the row number in the open file table as the open file identifier.

Open files must be properly closed. If a filesystem has open files, the filesystem cannot be unmounted except on shutdown where unmount is forced. The closing is done by calling `vfs_close`:

`int vfs_close(openfile_t file)`

- Closes an open file `file`.
- Returns `VFS_OK` (zero) on success, negative on error.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Lock the open file table.
  3. Verify that the given file is really open, otherwise, free the open file table lock and return `VFS_INVALID_PARAMS`.
  4. Call close on the actual filesystem for the file.
  5. Mark the entry in the open file table free.
  6. Free the open file table lock.
  7. Call `vfs_end_op`.
  8. Return the return value given by the filesystem when close was called.

The seek position within the file can be changed by calling:

`int vfs_seek(openfile_t file, int seek position)`

- Seek the given open file to the given seek position.
- The position is not verified to be within the file's size and behavior on exceeding the current size of the file is filesystem dependent.
- Returns `VFS_OK` on success, negative on error.
- Implementation:

1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
2. Lock the open file table.
3. Verify that the given file is really open, otherwise, free the open file table lock and return `VFS_INVALID_PARAMS`.
4. Set the new seek position in open file table.
5. Free the open file table.
6. Call `vfs_end_op`.
7. Return `VFS_OK`.

`int vfs_read(openfile_t file, void *buffer, int bufsize)`

- Reads at most `bufsize` bytes from the given file into the buffer. The read is started from the current seek position and the seek position is updated to match the new position in the file after the read.
- Returns the number of bytes actually read. On most filesystems, the requested number of bytes is always read when available, but this behaviour is not guaranteed. At least one byte is always read, unless the end of file or error is encountered. Zero indicates the end of file and negative values are errors.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Verify that the given file is really open, otherwise return `VFS_INVALID_PARAMS`.
  3. Call the internal `read` function of the filesystem.
  4. Lock the open file table.
  5. Update the seek position in the open file table.
  6. Free the open file table.
  7. Call `vfs_end_op`.
  8. Return the value returned by the filesystem's `read`.

`int vfs_write(openfile_t file, void *buffer, int datasize)`

- Writes at most `datasize` bytes from the given `buffer` into the open `file`.
- The write is started from the current seek position and the seek position is updated to match the new place in the file.
- Returns the number of bytes written. All bytes are always written unless an unrecoverable error occurs (filesystem full, for example). Negative values are error conditions on which nothing was written.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Verify that the given file is really open, otherwise return `VFS_INVALID_PARAMS`.
  3. Call the internal `write` function of the filesystem.
  4. Lock the open file table.
  5. Update the seek position in the open file table.
  6. Free the open file table.

7. Call `vfs_end_op`.
8. Return the value returned by the filesystem's `write`.

Files can be created and removed by the following two functions:

`int vfs_create(char *pathname, int size)`

- Creates a new file with given `pathname`. The size of the file will be `size`. The `pathname` must include the mount-point (full name would be `[root]shell.mips32`, for example).
- Returns `VFS_OK` on success, negative on error.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Parse the `pathname` into volume name and file name parts.
  3. If the `pathname` was badly formatted or too long, call `vfs_end_op` and return with the error code `VFS_ERROR`.
  4. Lock the filesystem table. (This is to prevent unmounting of the filesystem during the operation. Unlike read or write, we do not have an open file to guarantee that unmount does not happen.)
  5. Find the filesystem from the filesystem table. If it is not found, free the table, call `vfs_end_op` and return with the error code `VFS_NO_SUCH_FS`.
  6. Call the internal `create` function of the filesystem.
  7. Free the filesystem table.
  8. Call `vfs_end_op`.
  9. Return the value returned by the filesystem's `create`.

`int vfs_remove(char *pathname)`

- Removes the file with the given `pathname`. The pathname must include the mount-point (a full name would be `[root]shell.mips32`, for example).
- Returns `VFS_OK` on success, negative on failure.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Parse the pathname into the volume name and file name parts.
  3. If the `pathname` was badly formatted or too long, call `vfs_end_op` and return with the error code `VFS_ERROR`.
  4. Lock the filesystem table. (This is to prevent unmounting of the filesystem during the operation. Unlike read or write, we do not have an open file to guarantee that unmount does not happen.)
  5. Find the filesystem from the filesystem table. If it is not found, free the filesystem table, call vfs end op and return with the error code `VFS_NO_SUCH_FS`.
  6. Call the internal `remove` function of the filesystem.
  7. Free the filesystem table by calling semaphore V on vfs table.sem.
  8. Call `vfs_end_op`.
  9. Return the value returned by the filesystem's `remove`.

Filesystem Operations

In addition to providing an unified access to all filesystems, VFS also provides functions to mount and unmount filesystems. Filesystems are automatically mounted at boot time with the function `vfs_mount_all`, which is described below.

The file `kudos/fs/filesystems.c` contains a table of all available filesystem drivers. When an automatic mount is attempted, this table is traversed by the `filesystems_try_all` function to find a driver that matches the filesystem on the disk, if any.

`void vfs_mount_all(void)`

- Mounts all filesystems found on all disks attached to the system. Tries all known filesystems until a match is found. If no match is found, prints a warning and ignores the disk in question.
- Called in the system boot up sequence.
- Implementation:
  1. For each disk in the system do the following steps:
  1. Get the device entry for the disk by calling `device_get`.
  2. Dig the generic block device entry from the device descriptor.
  3. Attempt to mount the filesystem on the disk by calling `vfs_mount_fs` with `NULL` as the volumename (see below).

To attach a filesystem manually either of the following two functions can be used. The first one probes all available filesystem drivers to initialize one on the given disk and the latter requires the filesystem driver to be pre-initialized.

`int vfs_mount_fs(gbd_t *disk, char *volumename)`

- Mounts the given disk to the given mountpoint (`volumename`). `volumename` must be non-empty. The mount is performed by trying out all available filesystem drivers listed in the `filesystems` array in `kudos/fs/fileystems.c`. The first match (if any) is used as the filesystem driver for the disk.
- If `NULL` is given as the `volumename`, the name returned by the filesystem driver is used as the mountpoint.
- Returns `VFS_OK` (zero) on success, negative on error (no matching filesystem driver or too many mounted filesystems).
- Implementation:
  1. Try the `init` functions of all available filesystems in `kudos/fs/filesystems.c` by calling `filesystems_try_all`.
  2. If no matching filesystem driver was found, print warning and return the error code `VFS_NO_SUCH_FS`.
  3. If the `volumename` is `NULL`, use the name stored into `fs_t->volume` name by the filesystem driver.
  4. If the `volumename` is an empty string, unmount the filesystem driver from the disk and return `VFS_INVALID_PARAMS`.
  5. Call `vfs_mount` (see below) with the filesystem driver instance and `volumename`.
  6. If `vfs_mount` returned an error, unmount the filesystem driver from the disk and return the error code given by it.
  7. Return with `VFS_OK`.

`int vfs_mount(fs_t *fs, char *name)`

- Mounts an initialized filesystem driver `fs` into the VFS mount-point `name`.
- Returns `VFS_OK` on success, negative on error. Typical errors are `VFS_LIMIT` (too many mounted filesystems) and `VFS-ERROR` (mount-point was already in use).
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Lock the filesystem table by calling `semaphore_P` on `vfs_table.sem`.
  3. Find a free entry on the filesystem table.
  4. If the table was full, free it by calling `semaphore_V` on `vfs_table.sem`, call `vfs_end_op` and return the error code `VFS_LIMIT`.
  5. Verify that the mount-point name is not in use. If it is, free the filesystem table by calling `semaphore_V` on `vfs_table.sem`, call `vfs_end_op` and return the error code `VFS_ERROR`.
  6. Set the `mountpoint` and `fs` fields in the filesystem table to match this mount.
  7. Free the filesystem table by calling `semaphore_V` on `vfs_table.sem`.
  8. Call `vfs_end_op`.
  9. Return `VFS_OK`.

To find out the amount of free space on given filesystem volume, the following function can be used:

`int vfs_getfree (char *filesystem)`

- Finds out the number of free bytes on the given filesystem, identified by its mount-point name.
- Returns the number of free bytes, negative values are errors.
- Implementation:
  1. Call `vfs_start_op`. If an error is returned by it, return immediately with the error code `VFS_UNUSABLE`.
  2. Lock the filesystem table by calling `semaphore_P` on `vfs_table.sem`. (This is to prevent unmounting of the filesystem during the operation. Unlike read or write, we do not have an open file to guarantee that unmount does not happen.)
  3. Find the filesystem by its mount-point name `filesystem`.
  4. If the filesystem is not found, free the filesystem table by calling `semaphore_V` on `vfs_table.sem`, call `vfs_end_op` and return the error code `VFS_NO_SUCH_FS`.
  5. Call filesystem's `getfree` function.
  6. Free the filesystem table by calling `semaphore_V` on `vfs_table.sem`
  7. Call `vfs_end_op`.
  8. Return the value returned by filesystem's `getfree` function.

CHAPTER 13

Trivial Filesystem

Trivial File System (TFS) is, as its name implies, a very simple file system. All operations are implemented in a straightforward manner without much consideration for efficiency, there is only simple synchronization and no bookkeeping for open files, etc. The purpose of the TFS is to give students a working (although not thread-safe) filesystem and a tool (see *Using KUDOS*) for moving data between TFS and the native filesystem of the platform on which KUDOS is being developed.

When students implement their own filesystem, the idea is that files can be moved from the native filesystem to the TFS using the TFS tool, and then they can be moved to the student filesystem using KUDOS itself. This way students don't necessarily need to write their own tool(s) for the simulator platform. It is, of course, perfectly acceptable to write your own tool(s).

Trivial filesystem uses the native block size of a drive (must be predefined). Each filesystem contains a volume header block (block number 0 on disk). After header block comes block allocation table (BAT, block number 1), which uses one block. After that comes the master directory block (MD, block number 2), also using one block. The rest of the disk is reserved for file header (inode) and data blocks. The following figure illustrates the structure of a TFS volume:

Fig. 13.1: An illustration of the disk blocks on a TFS volume.

Note that all multibyte data in TFS is *big-endian*. This is not a problem in the MIPS32 version of KUDOS, since YAMS is big-endian also, but in the x86-64 version of KUDOS this is a problem, since x86-64 is little-endian. This means that we need to go through the function `from_big_endian32` (defined in `kudos/lib/libc.h`) when dealing with TFS. For x86-64 this function translates the value into little-endian, for MIPS32 this function does nothing.

The volume header block has the following structure. Other data may be present after these fields, but it is ignored by TFS.

Offset	Type	Name	Description
`0x00`	`uint32_t`	`magic`	Magic number, must be 3745 (`0x0EA1`) TFS volumes.
`0x04`	`char[TFS_VOLNAME_MAX]`	`volname`	Name of the volume, including the terminating zero

The block allocation table is a bitmap which records the free and reserved blocks on the disk, one bit per block, 0 meaning free and 1 reserved. For a 512-byte block size, the allocation table can hold 4096 bits, resulting in a 2MB disk. Note that the allocation table includes also the three first blocks, which are always reserved.

The master directory consists of a single disk block, containing a table of the following 20-byte entries. This means that a disk with a 512-byte block size can have at most 25 files (512/20 = 25.6).

Offset	Type	Name	Description
0x00	uint32_t	inode	Number of the disk block containing the file header (inode) of this file.
0x04	char[TFS_FILENAME_MAX]	name	Name of the file, including the terminating zero.

This means that the maximum file name length is actually TFS_FILENAME_MAX-1.

A file header block ("inode") describes the location of the file on the disk and its actual size. The contents of the file is stored to the allocated blocks in the order they appear in the block list (the first TFS_BLOCK_SIZE bytes are stored to the first block in the list etc.). A file header block has the following structure:

Offset	Type	Name	Description
0x00	uint32_t	filesize	Size of the file in bytes. When compiling for x86-64 this field is big-endian so the conversion function is used on this field.
0x04	uint32_t[TFS_BLOCKS_MAX]	blocks	Blocks allocated for this file. Unused blocks are marked as 0 as a precaution (since block 0 can never be a part of any file). When compiling for x86-64 this field is big-endian so the conversion function is used on this field.

With a 512-byte block size, the maximum size of a file is limited to 127 blocks (512/4 1) or 65024 bytes.

Note that this specification does not restrict the block size of the device on which a TFS can reside. However, the KUDOS TFS implementation and the TFS tool do not support block sizes other than 512 bytes. Note also that even though the TFS filesystem size is limited to 2MB, the device (disk image) on which it resides can be larger, the remaining part is just not used by the TFS.

TFS Driver Module

The KUDOS TFS module implements the Virtual File System interface with the following functions.

fs_t * tfs_init(gbd t *disk, uint32_t sector)

- Attempts to initialize a TFS on the given disk (a generic block device, actually) at the given sector sector. If the initialization succeeds, a pointer to the initialized filesystem structure is returned. If not (e.g. the header block does not contain the right magic number or the block size is wrong), NULL is returned.
- Implementation:
  1. Check that the block size of the disk is supported by TFS.
  2. Allocate semaphore for filesystem locking (tfs->lock).
  3. Allocate a memory page for TFS internal buffers and data and the filesystem structure (fs_t).
  4. Read the first block of the disk and check the magic number.
  5. Initialize the TFS internal data structures.
  6. Store disk and the filesystem locking semaphore to the internal data structure.
  7. Copy the volume name from the read block into fs_t.
  8. Set fs_t function pointers to TFS functions.
  9. Return a pointer to the fs_t.

int tfs_unmount (fs_t *fs)

- Unmounts the filesystem. Ensures that the filesystem is in a "clean" state upon exit, and that future operations will fail with `VFS_NO_SUCH_FS`.
- Implementation:
  1. Wait for active operation to finish by calling `semaphore_P` on `tfs->lock`.
  2. Deallocate the filesystem semaphore `tfs->lock`.
  3. Free the memory page allocated by `tfs_init`.

`int tfs_open(fs_t *fs, char *filename)`

- Opens a file for reading and writing. TFS does not keep any status regarding open files, the returned file handle is simply the inode block number of the file.
- Implementation:
  1. Lock the filesystem by calling `semaphore_P` on `tfs->lock`.
  2. Read the MD block.
  3. Search the MD for filename.
  4. Free the filesystem by calling `semaphore_V` on `tfs->lock`.
  5. If filename was found the MD, return its inode block number, otherwise return `VFS_NOT_FOUND`.

`int tfs_close(fs_t *fs, int fileid)` Does nothing, since TFS does not keep status for open files.

`int tfs_create(fs_t *fs, char *filename, int size)`

- Creates a file with the given name and size (TFS files cannot be resized after creation).
- The file will contain all zeros after creation.
- Implementation:
  1. Lock the filesystem by calling `semaphore_P` on `tfs->lock`.
  2. Check that the size of the file is not larger than the maximum file size that TFS can handle.
  3. Read the MD block.
  4. Check that the MD does not contain `filename`.
  5. Find an empty slot in the MD, return error if the directory is full.
  6. Add a new entry to the MD.
  7. Read the BAT block.
  8. Allocate the inode and file blocks from BAT, and write the block numbers and the filesize to the inode in memory.
  9. Write the BAT to disk.
  10. Write the MD to disk.
  11. Write the inode to the disk.
  12. Zero the content blocks of the file on disk.
  13. Free the filesystem by calling `semaphore_V` on `tfs->lock`.
  14. Return `VFS_OK`.

`int tfs_remove(fs_t *fs, char *filename)`

- Removes the given file from the directory and frees the blocks allocated for it.

- Implementation:
  1. Lock the filesystem by calling `semaphore_P` on `tfs->lock`.
  2. Read the MD block.
  3. Search the MD for `filename`, return error if not found.
  4. Read the BAT block.
  5. Read inode block.
  6. Free inode block and all blocks listed in the inode from the BAT.
  7. Clear the MD entry (set inode to 0 and name to an empty string).
  8. Write the BAT to the disk.
  9. Write the MD to disk.
  10. Free the filesystem by calling `semaphore_V` on `tfs->lock`.
  11. Return `VFS_OK`.

`int tfs_read (fs_t *fs, int fileid, void *buffer, int bufsize, int offset)`

- Reads at most `bufsize` bytes from the given file into the given buffer. The number of bytes read is returned, or a negative value on error. The data is read starting from given offset. If the offset equals the file size, the return value will be zero.
- Implementation:
  1. Lock the filesystem by calling `semaphore_P` on `tfs->lock`.
  2. Check that `fileid` is sane (3 and not beyond the end of the device/filesystem).
  3. Read the inode block (which is `fileid`).
  4. Check that the `offset` is valid (not beyond end of file).
  5. For each needed block do the following:
     (a) Read the block.
     (b) Copy the appropriate part of the block into the right place in buffer.
  6. Free the filesystem by calling `semaphore_V` on `tfs->lock`.
  7. Return the number of bytes *actually* read.

`int tfs_write(fs_t *fs, int fileid, void *buffer, int datasize, int offset)`

- Writes (at most) `datasize` bytes to the given file. The number of bytes actually written is returned. Since TFS does not support file resizing, it may often be the case that not all bytes are written (which should actually be treated as an error condition). The data is written starting from the given offset.
- Implementation:
  1. Lock the filesystem by calling `semaphore_P` on `tfs->lock`.
  2. Check that `fileid` is sane (3 and not beyond the end of the device/filesystem).
  3. Read the inode block (which is `fileid`).
  4. Check that the `offset` is valid (not beyond end of file).
  5. For each needed block do the following:
     (a) If only part of the block will be written, read the block.

   (b) Copy the appropriate part of the block from the right place in buffer.

   (c) Write the block.

6. Free the filesystem by calling `semaphore_V` on `tfs->lock`.
7. Return the number of bytes *actually* written.

`int tfs_getfree(fs_t *fs)`

- Returns the number of free bytes on the filesystem volume.
- Implementation:
  1. Lock the filesystem by calling `semaphore_P` on `tfs->lock`.
  2. Read the BAT block.
  3. Count the number of zeroes in the bitmap. If the disk is smaller than the maximum supported by TFS, only the first appropriate number of bits are examined (of course).
  4. Get number of free bytes by multiplying the number of free blocks by block size.
  5. Free the filesystem by calling `semaphore_V` on `tfs->lock`.
  6. Return the number of free bytes.

CHAPTER 14

Appendix

More about the `tfstool`

The utility program, `tfstool`, which is shipped with KUDOS, provides a way to transfer files to a filesystem KUDOS understands. `tfstool` can be used to create a Trivial Filesystem (TFS, documentation about TFS coming soon) to a given file, to examine the contents of a file containing a TFS, and to transfer files to the TFS. KUDOS implementation of TFS does not include a way to initialize the filesystem, so using `tfstool` is the only way to create a new TFS. `tfstool` is also used to move userland binaries to TFS. When you write your own filesystem to KUDOS, you might find it helpful to leave TFS intact. This way you can still use `tfstool` to transfer files to the KUDOS system without writing another utility program for your own filesystem.

The implementation of the `tfstool` is provided in the `kudos/util/` directory. The KUDOS `Makefile` can be used to compile it to the executable `kudos/util/tfstool`. Note that `tfstool` is compiled with the native compiler, not the cross-compiler used to compile KUDOS. The implementation takes care of byte-order conversion (big-endian, little-endian) if needed.

To get a summary of the arguments that `tfstool` accepts you may run it without arguments:

```
$ kudos/util/tfstool
KUDOS Trivial Filesystem (TFS) Tool
...
```

The accepted commands are also explained below:

`create <filename> <size> <volume-name>`

> Create a new TFS volume and write it to file `<filename>`. The total size of the file system will be `<size>` 512-byte blocks. Note that the three first blocks are needed for the TFS header, the TFS master directory, and the TFS block allocation table. `<size>` must therefore be at least 3. The created volume will have the name `<volume-name>`.
>
> Note that the number of blocks must be the same as the setting in `yams.conf`.

`list <filename>`

> List the files found in the TFS volume residing in `<filename>`.

`write <filename> <local-filename> [<TFS-filename>]`

Write a file from the local system (`<local-filename>`) to the TFS volume residing in the file `<filename>`. The optional fourth argument specifies the filename to use for the file inside the TFS volume. If not given, `<local-filename>` will be used.

Note that you probably want to give a `<TFS-filename>`, since otherwise you end up with a TFS volume with files named like `userland/halt.mips32`, which can cause confusion since **TFS does not support directories**.

`read <filename> <TFS-filename> [<local-filename>]`

Read a file (`<TFS-filename>`) from TFS volume residing in the file filename to the lo- cal system. The optional fourth argument specifies the filename in the local system. If not given, the `<TFS-filename>>` will be used.

`delete <filename> <TFS-filename>`

Delete the file with name `<TFS-filename>` from the TFS volume residing in the file `<filename>`.

Bibliography

[KR] Brian Kernighan and Dennis Ritchie. *The C Programming Language*, 2nd Edition. Prentice-Hall, 1988.

[ModernC] Jens Gustedt. *Modern C*. Unpublished, 2015. Available for free from http://icube-icps.unistra.fr/index.php/File:ModernC.pdf.

[OSTEP] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015.Available for free from http://pages.cs.wisc.edu/~remzi/OSTEP/.

[Stallings] William Stallings. *Operating Systems: Internals and Design Principles*, 4th edition. Prentice-Hall, 2001.

[Tanenbaum] Andrew Tanenbaum. *Modern Operating Systems*, 2nd edition. Prentice-Hall, 2001.

[COD5e] David A. Patterson and John L. Hennessy. *Computer Organization and Design*, 5th edition. Elsevier, 2014.

[Andrews] Gregory R. Andrews., *Foundations of multithreaded, parallel and distributed programming*. Addison-Wesley Longman, 2000.

[BOH] Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Pearson, 2016.