KUDOS

is the
Koebenhavns Universitet Discrete Operating System

The roadmap of the KUDOS system
Version 1.0
May 28, 2014

Juha Aatrokoski, Timo Lilja, Leena Salmela,
Teemu J. Takanen, Aleksi Virtanen and Philip Meulengracht

Contents

Chapter 1

Introduction

KUDOS is a derivative project from BUENOS which aside from supporting the old BUENOS system, now also supports the Intel x86-64 Architecture. The new operating system, KUDOS, is meant also meant as a exercise base for the operating system course, but with a more common platform in mind. Unlike its old project BUENOS, KUDOS will run on more recent real-world hardware[1] without any special architecture, but rather on your own laptop at home.

The system is kept structurally intact to BUENOS, however all platform-specific code has been split up into sub-directories, and is controlled with the makefile. KUDOS systems are ready for multi-core, however support for starting other application processors has not been implemented, while the BUENOS part has the multi-core support. Both BUENOS and KUDOS also provides skeleton code for threading, wide variety of synchronization primitives, userland support and proccesses.A simple custom filesystem and code for networking is also provided (However no network-card drivers are provided).

To keep both 32 bit code (the mips project) and 64 bit code (the x86-64 project) under the same roof, many modifications had to be made to the old BUENOS project, and thus the structure of the project has changed partially. It has also changed the procedure for starting up the new operating system, KUDOS.

Just like BUENOS, the main idea of the system is to give you a real, working multiprocessor operating system kernel which is as small and simple as possible. To boot KUDOS-x86-64 on a real computer all you would need is simply GRUB2 as a bootloader, and then make sure to add KUDOS as a boot entry into GRUB2. KUDOS now supports your everyday intel 64 bit architecture and thus can run on your everyday computer. No code modifcations would be neccessary.

1.1 How to Use This Document

This roadmap document 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 the user-guide (usage) and chapter 2 (system overview) carefully.
2. Skim through the whole document to get a good overview.
3. Before designing and implementing your assignments, read carefully all chapters on the subject matter.
4. Use the document as reference when designing and implementing your improvements.

[1] In theory.

Chapter 2

Kernel Overview

2.1 Kernel Architecture

While aiming for simplicity, the KUDOS kernel is still a quite complicated piece of software. The kernel is divided into many separate modules, each stored in a different directory as was seen above.

To understand how the kernel is built, we must first see what it actually does. The kernel works between userland processes and machine hardware to provide services for processes. It is also responsible for providing the userland processes a private sandbox in which to run. Further, the kernel also provides various high level services such as filesystems and networking which act on top of the raw device drivers.

A simplified view of the KUDOS kernel can be seen in Figure 2.1. At the top of the picture lies the userland and at the bottom is the machine hardware. Neither of these are part of the kernel, they just provide the context in which the kernel operates. The userland/kernel boundary as well as the hardware/software (hardware/kernel) interface are also marked in the picture.

On the kernel side of these boundaries lies the important interface code. At the top, we can see the system call interface, which among other userland related functionality is documented in chapter 5. The system call interface is a set of functions which can be called from userland programs[1]. These functions can then call almost any function inside the kernel to implement the required functionality. Kernel functions cannot be called directly from userland programs to protect kernel integrity and make sure that the userland sandbox doesn't leak.

On the bottom boundary are the device drivers. Device drivers are pieces of code which know how to use a particular piece of hardware. Device drivers are usually divided into two parts: the top and bottom halves. The bottom half of a device driver is an independent piece of code which is run outside the kernel threading system whenever the hardware generates an interrupt (this piece of code is called an interrupt handler). The top half of the device driver is a set of functions which can be called from within the kernel. The details of this, and description on how the device driver halves communicate with each other are documented in chapter 9.

On top of the device drivers are various services which use device drivers. Two examples can be seen in the picture: the filesystem and the networking. The filesystem (see chapter 7) is actually accessed through a module called the virtual filesystem (see section 7.3), which abstracts differences between different filesys-

[1] System calls are important part of any OS. Try reading manual pages of fork(2), wait(2), exec(2), read(2), write(2), open(2) and close(2) in any Unix system for an example of the real thing.

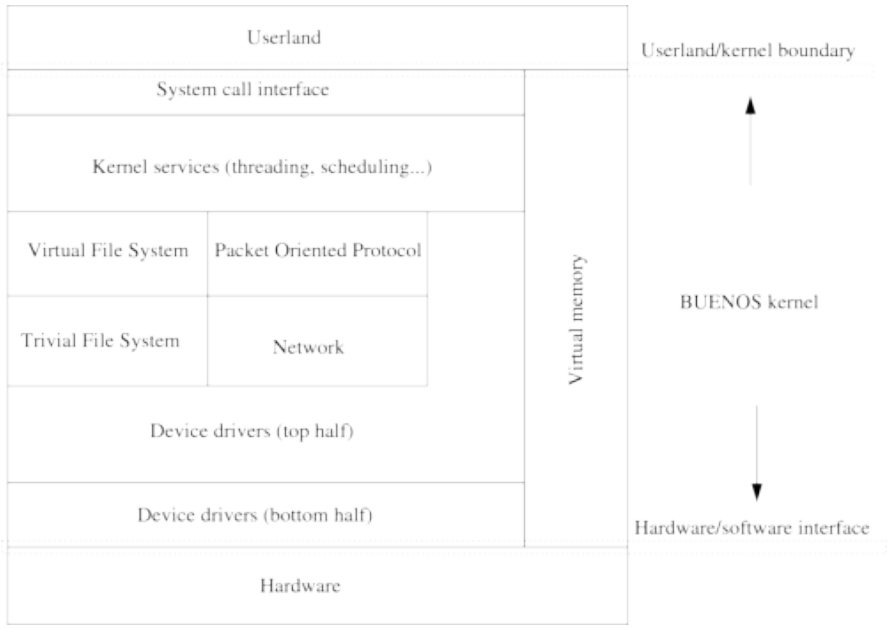

Figure 2.1: KUDOS kernel overall architecture

tems. The filesystem itself uses a disk device driver to access its permanent storage (the disk). Similarly the networking layer (see chapter 8), which uses network interface driver(s), provides tools for sending and receiving network packets. The packet oriented protocol module (POP, see section 8.2) uses the networking module to provide socket and packet port (similar to UDP ports in the Internet Protocol) functionality.

2.1.1 Threading

Now we have seen an overview of various kernel services, but we still don't have anything which can call these service functions. The core of any kernel, including BUENOS, is its threading and context switching functionality. This functionality is sometimes called a kernel by itself. Threading is provided by a threading library (see chapter 3) in BUENOS. The threading system makes it possible to execute threads, separate instances of program execution. Each thread runs independently of each other, alternating their turns on the CPU(s). The context switching system is used to switch one thread out of a CPU and to put a new one on it. Threads themselves are unaware of these switches, unless they intentionally force themselves out of execution (go to sleep).

Threads can be started by using the thread library. When starting a thread it is given a function which it executes. When the function ends, the thread dies. The thread can also commit suicide by explicitly killing itself. Threads cannot kill each other, murders are not allowed in the kernel (see exercises below). Each userland program runs inside one thread. When the actual userland code is being run, the thread cannot see the kernel memory, it can only access the system call layer.

Threads can be pre-empted at any point, both in kernel and in userland. Pre-empting means that the thread is taken out of execution in favor of some other thread. The only way to prevent pre-empting is to disable interrupts (which also disables timer interrupts used to measure thread time-slices).

Since the kernel includes many data structures and many threads are run simul-

taneously (we can have multiple CPUs), all data has to be protected from other threads. The protection can be done with exclusive access, achieved with various synchronization mechanisms documented in chapter 4.

2.1.2 Virtual Memory

In the much referenced Figure 2.1, there was one more subsystem which hasn't been explained: the virtual memory (VM) subsystem. As the figure implies, it affects the whole kernel, interacts with hardware and also with the userland.

The VM subsystem is responsible for all memory handling operations in the kernel. Its main function is to provide an illusion of private memory spaces for userland processes, but its services are also used in the kernel. Since memory can be accessed from any part of the system, VM interacts directly with all system components.

The physical memory usage in `BUENOS` can be seen in Figure 2.2. At the left side of the figure, memory addresses can be seen. At the bottom is the beginning of the system main memory (address zero) and at the top the end of the physical memory.

The kernel uses part of this physical memory for its code (kernel image), interrupt handling routines and data structures, including thread stacks. The rest of the memory is at the mercy of the VM.

As in any modern hardware, memory pages (4096 byte regions in our case) can be *mapped* in `YAMS`. The mapped addresses are also called *virtual addresses*. Mapping means that certain memory addresses do not actually refer to physical memory. Instead, they are references to a structure which *maps* these addresses to the actual addresses. This makes it possible to provide the illusion of exclusive access to userland processes. Every userland process has code at address `0x00001008`, for example. In reality this address is in the mapped address range and thus the code is actually on a private physical memory page for each process.

For more information on the virtual memory system and particularly on the various address ranges, see chapter 6.

2.1.3 Support for Multiple Processors

`BUENOS` is a multiprocessor operating system, with pre-emptive kernel threading. All kernel functions are thread-safe (re-entrant) except for those that are used only during the bootup process.

Most code explicitly concerning SMP support is found in the bootstrap code (see chapter 10). Unlike in real systems, where usually only one processor starts at boot and it is up to it to start the other processors, in `YAMS` *all* processors will start executing code simultaneously and at the same address (`0x80010000`). To handle this, the procedure described in chapter 10 is used.

Another place where the SMP support is directly evident is in the context switch code, and in the code initializing data structures used by the context switching code. Each processor must have its own stack when handling interrupts, and each processor has its own current thread. To account for these, the context switching code must know the processor on which it runs.

Finally, a warning: implementing all virtual memory exercises on a multiprocessor machine can be hard. It is suggested that for VM exercises, only one CPU is used[2].

[2]The reasons become evident when the inner details of the VM subsystem are covered later. For the curious: the problem arises from the fact of having multiple TLBs, one for each CPU. (The TLB is a piece of hardware used to map memory pages.)

Figure 2.2: `BUENOS` memory usage. Addresses are physical addresses. Note that the picture is not in scale.

Otherwise, the SMP support should be completely transparent. Of course it means that synchronization issues must be handled more carefully, but mostly everything works as it would on a single CPU system.

2.2 Kernel Programming

Kernel programming differs somewhat from programming user programs. This section explains these differences and also introduces some conventions that have been used with `BUENOS`.

2.2.1 Memory Usage

The most significant difference between kernel programming and programming of user programs is memory usage. In the MIPS32 architecture, which `YAMS` emulates, the memory is divided into segments . Kernel code can access all these segments while user programs can only access the first segment called the *user mapped segment*. In this segment the first bit of the address is 0. If the first bit is 1, the address belongs to one of the kernel segments and is not usable in userland. The most important kernel segment in `BUENOS` is the *kernel unmapped segment*, where addresses start with the bit sequence 100. These addresses point to physical memory locations. In kernel, most addresses are like this. More information about the memory segments can be found in section 6.1.

When initializing the system, a function (`kmalloc`) is provided to allocate memory in arbitrary-size chunks. This memory is permanently allocated and cannot be freed. Before initializing the virtual memory system `kmalloc` is used to allocate memory. After the initialization of the virtual memory system `kmalloc` can no longer be used. Instead, memory is allocated page by page from the virtual memory system. These pages can be freed later.

2.2.2 Stacks and Contexts

A stack is needed always when running code that is written in C. The kernel provides a valid stack for user programs so the programmer does not need to think about this. In kernel, however, nobody else provides you with a valid stack. Every kernel thread must have its own stack. In addition, every CPU must have an interrupt stack because thread stacks cannot be easily used for interrupt processing. If a kernel thread is associated with a user process, the user process must also have its own stack. `BUENOS` already sets up kernel stacks and interrupt stacks appropriately.

Because the kernel and interrupt stacks are statically allocated, their size is limited. This means that large structures and tables cannot be allocated from stack. (The variables declared inside a function are stack-allocated.) Note also that recursive functions allocate space from the stack for each recursion level. Deeply recursive functions should thus not be used.

Code can be run in several different contexts. A context consists of a stack and CPU register values. In the kernel there are two different contexts. Kernel threads are run in a normal kernel context with the thread's stack. Interrupt handling code is run in an interrupt context with the CPU's interrupt stack. These two contexts differ in a fundamental way. In the kernel context the current context can be saved and resumed later. Thus interrupts can be enabled and blocking operations can be called. In the interrupt context this is not possible so interrupts must be disabled and no blocking operations can be called. In addition, if a kernel thread is associated with a userland process, it must also have a userland context.

2.2.3 Library Functions

BUENOS provides several library functions in the directory lib/. These include functions for string processing and random number generation. These functions are needed because standard C library cannot be linked with BUENOS. The prototypes of these functions can be found in lib/libc.h.

2.2.4 Using a Console

In the kernel, reading from and writing to the console is done by using the polling TTY driver. The kprintf and kwrite functions can be used to print informational messages to the user. Debug printing should be handled with the DEBUG function. This way debug messages can be easily disabled later when they are no longer needed. Userland console access should not be handled with these functions. The interrupt driven TTY driver should be used instead. See the example in init/main.c.

2.2.5 Busy Waiting

In the kernel, special attention has to be given to synchronization issues. Busy waiting must be avoided whenever possible. The only place where busy waiting is acceptable is the spinlock implementation, which is already done for you. Because spinlocks use busy waiting, they should never be held for a long time.

2.2.6 Floating Point Numbers

YAMS does not support floating point numbers so they cannot be used in BUENOS either. If an attempt to execute a floating point instruction is made, a co-processor unusable exception will occur. (The floating point unit is co-processor 1 in MIPS32 architecture.)

2.2.7 Naming Conventions

Some special naming conventions have been used when programming BUENOS. These might help you find a function or a variable when you need it. Functions are generally named as filename_function where filename is the name of the file where the function resides and function tells what the function does. Variables are named similarly filename_variable.

2.2.8 Debug Printing

Sometimes it is usefull to be able to print debugging information from the kernel. A function which uses the polling TTY driver is provided for such printing. Because polling TTY driver is used, printing is possible from all parts of the kernel. Note that printing with the polling driver slows the system down considerably and also changes system timings which may cause trouble when debugging a SMP system.

```
void DEBUG (char *debuglevelname, char *format, ...)
```

- If debuglevelname has been given to the kernel as a boot argument, prints debug information. If not, ignores the debug printing.
- format and other arguments are given as for printf().

2.2.9 C Calling Conventions

Normally C compiler handles function calling conventions (mostly argument passing) transparently. Sometimes in kernel code the calling convention issues need to be handled manually. Manual calling convention handling is needed when calling C routines from a assembly program or when manipulating thread contexts in order to pass arguments to starting functions.

Arguments are passed to all functions in MIPS argument registers `A0`, `A1`, `A2` and `A3`. When more than 4 arguments are needed, the rest are passed in the stack. The arguments are put into the stack so that the 1st argument is in the lowest memory address.

There is one thing to note: the stack frame for arguments must always be reserved, even when all arguments are passed in the argument registers. The frame must have space for all arguments. Arguments which are passed in registers need not to be copied into this reserved space.

2.2.10 Kernel Boot Arguments

`YAMS` virtual machine provides a way to pass boot arguments from the host operating system to the booted kernel. `BUENOS` supports these arguments. Please see Appendix A for details.

Exercises

2.1. In `BUENOS`, a thread that is ready to be run will be run on whichever processor first removes it from the scheduler's ready list. This can cause the thread to bounce from processor to processor on every timeslice. This behavior is also present in real operating systems, e.g. Solaris. Why might this behavior not be a good idea?

2.2. In `BUENOS` threads cannot kill each other. There are many reasons for this, try to figure out as many as you can.

Chapter 3

Threading and Scheduling

This chapter describes the threading system implemented in BUENOS. The kernel can run multiple threads and schedule them across any number of CPUs the system happens to have.

The threading system contains three major parts: thread library, scheduler and context switching code. Each of these components is thoroughly explained below in their own sections.

The thread library contains functions for thread creation, running and finishing (dying). It also implements the system wide table of threads.

Scheduler handles the allocation of CPU time for runnable threads.

Context switch code is executed when an exception (trap or interrupt) occurs. Its purpose is to save and restore execution contexts (CPU register states, memory mappings etc.) of threads.

The context switching part is the most complicated and most hardware dependent part of the threading system. It is not necessary to understand it fully to be able to understand the whole threading system. However, it is essential to see the purpose of all these three parts.

For an introduction to these concepts, read either [Stallings] p. 108–123, 154–161, 394–407 and 438–449 or [Tanenbaum] p. 81–100 and 132–145.

3.1 Threads

BUENOS kernel supports multiple simultaneously running threads. One thread can be run on each available CPU at a given moment. Information on existing threads is stored in a fixed size table `thread_table`. The structure of the table is described in detail in section 3.1.1.

Threads and thread table are handled through a collection of library functions, that will do all necessary manipulation of the data structures. They will also take care of concurrency. Thread handling functions are described in section 3.1.2.

State diagram of BUENOS threads is presented in Figure 3.1. States in detail are described below:

- FREE indicates that this row in `thread_table` is currently unused.
- RUNNING threads are currently on CPU. In case of multiple CPUs, several threads may have this state.
- READY threads are on the scheduler's ready list and can be switched to RUNNING state.

Figure 3.1: BUENOS thread states and possible transitions

- SLEEPING threads are **not on CPU** and are in sleep queue. Sleeping threads are waiting for some resource to be freed. When access to the resource is granted, the thread is waken up and switched to READY state.
- NONREADY threads have been created, but are not yet marked to be runnable. The state is switched to READY when the function `thread_run()` is called.
- DYING threads have cleaned themselves up, but are still on CPU. The scheduler should mark them FREE when encountered.

3.1.1 Thread Table

Thread table contains all necessary information about threads. This information consists of:

- context of the thread, when it was running.
- state of the thread. The state is used mostly by the scheduler, when deciding which thread will be run next.
- pagetable of the thread. Each thread will have its own virtual memory mappings, so also own pagetables are needed.

All records and datatypes of thread table are described in Table 3.1.

Thread table is a fixed size (compile time option) structure, which has one line for each thread. Threads are referenced by thread IDs (`TID_t`), which corresponds to index to the thread table. The size of the table is defined in `kernel/config.h` by definition `CONFIG_MAX_THREADS`.

The thread table is protected by a single spinlock (`thread_table_slock`). The lock must be a spinlock, because it is used in contexts where threads cannot be switched for waiting (eg. in scheduler).

The thread table is initialized by calling `thread_table_init()` function, which will set all thread states to `FREE`.

3.1.2 Thread Library

Thread library provides functions for thread handling.

Thread Creation Functions

Threads can be manipulated by following functions implemented in `kernel/thread.c`:

Type	Name	Explanation
context_t *	context	Space for saving thread context. Context consists of all CPU registers, including the program counter (PC) and the stack pointer (SP). This pointer always refers to the thread's stack area.
context_t *	user_context	Pointer to this thread's context in userland. Field is NULL for kernel only threads.
thread_state_t	state	The current state of the thread. Valid values are: `FREE`, `RUNNING`, `READY`, `SLEEPING`, `NONREADY` and `DYING`.
uint32_t	sleeps_on	If nonzero, tells which resource the thread is sleeping on (waiting for). Nonzero value also indicates that the thread is in some list in sleep queue. Note that the thread might still be running and in middle of the process to go sleeping (in which case its state is `RUNNING`.)
pagetable_t *	pagetable	Pointer to the virtual memory mapping for this thread. This entry is `NULL` if the thread does not have a page table.
process_id_t	process_id	Index to the process entry. This field is currently unused, but thread creation sets this to a negative value.
TID_t	next	Pointer to next thread in this list. Used for forming lists of threads (ready to run list, sleep queue). If this is the last thread of a list, the value is negative.
uint32_t	dummy_alignment_fill[9]	This is needed because `thread_table` entries are expected to be 64 bytes long (by context_switch code). If new fields are added or old ones are removed this alignment should also be corrected in a proper way.

Table 3.1: Fields in thread table record

`TID_t thread_create (void (*func)(uint32_t), uint32_t arg)`

- Creates a new thread by allocating a slot from `thread_table`. PC in this thread's context is set to the beginning of the `func` and parameters are saved to the proper registers in context. The context is saved in the stack area of the newly created thread. When the scheduler decides to run this thread, context is restored and it looks like function `func` would have been called. The return address of the context is set to beginning of the function `thread_finish`.
- Returns the thread ID of the created thread. If the return value is negative, thread could not be created. The possible reasons for failure are: full thread table and virtual memory shortage.
- The argument `arg` is passed to the `func` which is called when the new thread starts after a call to `thread_run()`.

`void thread_run (TID_t t)`

- Calls `scheduler_add_ready(t)`, which sets the thread state to `READY` and adds the thread to the ready-to-run list.

Self Manipulation Functions

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

`void thread_switch (void)`

- Perform voluntary context switch. Scheduler will later add the thread to ready to run list if the thread is not sleeping on something (`sleeps_on` is zero). Context switch is performed by causing the software interrupt 0 which is handled the same way as the context switch. Interrupts are enabled before raising the software interrupt, since otherwise the switch might not happen. The interrupt state is restored before returning from this function.
- Note that there is also a macro called `thread_yield` which points to this function. Since yielding is mechanically equivalent to switching, the implementation is the same. The name yielding is used when the yield has no actual effect, switching is used when something actually happens (thread goes to sleep).

`void thread_finish (void)`

- Commit suicide. The thread calling this function will terminate itself and free its resources (stack and thread table entry). The thread marks its state to be `DYING`. The row in thread table is later freed in the scheduler.
- If a pagetable has been reserved for this thread it must be freed before calling `thread_finish`.

`TID_t thread_get_current_thread (void)`

- Returns the TID of the calling thread.

`kernel/thread.c`, `kernel/thread.h`	Thread library
`kernel/_interrupt.s`, `kernel/interrupt.h`	Interrupt mask setting functions

3.2 Scheduler

Scheduler is a piece of code that allocates CPU time for threads. The basic BUENOS scheduler is pre-emptive and allocates CPU time in a round robin manner. Threads do not have priorities. Even threads currently in kernel can be interrupted when their time slice has been spent. This can be prevented by disabling interrupts.

The timeslice allocated for a thread is defined in `kernel/config.h` and the name of the configuration variable is `CONFIG_SCHEDULER_TIMESLICE`. The value defines how many CPU cycles a thread can use before it will be interrupted and next thread will be selected for running. Timeslice includes the time spent in context restoring, so it must be at least 250 cycles to guarantee that the thread will get at least some real processing done. The actual timeslice length is determined randomly and is at least the configured number of ticks, see Appendix A.

Scheduler works by maintaining a global `scheduler_current_thread` table of current threads (one per CPU). It also has a list of ready threads, maintained in the local list variable `scheduler_ready_to_run`. The actual implementation of the ready list is two indexes. One points to the beginning of the list in thread table and the other to the end. A negative value in both head and tail indicates an empty list.

The whole scheduler is locked by one spinlock to prevent multiple CPUs entering the scheduler at the same time. Interrupts are always disabled when scheduler is running, because it is called only from interrupt and exception handlers. The spinlock used is `thread_table_slock` and it also controls the access to the thread table.

Time ticks are handled by the CPU co-processor 0 counter mechanism. A timer interrupt is generated when the counter meets the compare value (time slice is over). The master interrupt handler will call the scheduler when a timer interrupt has occured. Scheduler will also be called if software interrupt 0 occured (thread gave up its timeslice) or when any interrupt occurs and idle thread is currently running on the current CPU. A new timer interrupt is scheduled after the scheduler has selected a new running thread.

When the scheduler is entered, the current thread is checked. If the current thread's state is marked as `DYING` or `RUNNING` and sleeping on something (`sleeps_on` is nonzero, see section 4.2) the current thread is not placed on the ready-to-run list, but its state is updated. For `DYING` threads the state is changed to `FREE` and for `RUNNING` (and sleeping) threads to `SLEEPING`. In all other cases the thread is placed at the end of the ready-to-run list and its state is updated to `READY`.

`void scheduler_schedule (void)`

- Selects the next thread to run. Updates `scheduler_current_thread` for current CPU. This **must not** be called from any thread, only from the interrupt handler.
- Implementation:
  1. Lock the thread table by `thread_table_slock` spinlock (interrupts **must** be off when calling this function, so they are not explicitly disabled).
  2. If the current thread state is `DYING`, mark it `FREE`. This releases the thread table entry for reuse.
  3. Else, if the thread is sleeping on something, just mark the state as *Sleeping*. The thread has placed itself on sleep queue before explicitly switching to scheduler.

4. Else, add the current thread to the end of `scheduler_ready_to_run` and mark it `READY`. Idle thread (thread 0) is never added to this list.
5. Remove the first thread from `scheduler_ready_to_run`. This might be the same thread placed on the list in the previous step. The function that will return the removed thread will return 0 (idle thread) if the ready to run list was empty.
6. Mark the removed thread as `RUNNING`.
7. Release the thread table spinlock.
8. Set the removed thread as the current thread for this CPU.
9. Set the hardware timer to generate an interrupt after configured number of ticks.

Threads can be added to the scheduler's ready list by calling the following function. This function is called *only* from the thread library function `thread_run` and from the synchronization library.

```
void scheduler_add_ready (TID_t t)
```

- Adds the thread `t` to the end of the ready-to-run list.
- Implementation:
  1. Lock the thread table (interrupts off, take thread table spinlock).
  2. Add `t` to the end of the list `scheduler_ready_to_run`.
  3. Release the thread table spinlock, restore interrupt status.

3.2.1 Idle thread

The idle thread, TID 0 (or `IDLE_THREAD_TID`), is a special case of a thread. Its context is not saved (and *must not* be saved on a SMP machine) and it can be running simultaneously on many CPUs. When restoring its context, only PC needs to be restored. The idle thread will enter a neverending waiting loop whenever run. Note that since the thread is used simultaneously on all CPUs, the code cannot do anything useful!

`kernel/scheduler.c`, `kernel/scheduler.h`	Scheduler

3.3 Context Switch

Context switching is traditionally the most bizarre piece of code in most operating system implementations. There are many reasons for this. One of them is that the context switch code must be written in assembler and not in any high level language. Another reason is the fact that it might be hard to follow the execution when the context of execution changes. Unfortunately context switching is also the hardest to understand of all parts of `BUENOS`. Luckily, it is not necessary to fully comprehend it to be able to understand the whole system.

Before going into details we must define what is actually meant by a context or context switching. In the scope of the threading system, a context means some particular computation process (note that this is not the same thing as userland process). This piece of code is mostly unaware that any other code is being run on

the same CPU. It is the responsibility of the threading system to provide an illusion for other pieces of code that they run in an isolated environment.

Thus when the need arises to give CPU time to some other part of the system the currently running code (thread) is interrupted. This might happen for three distinct reasons. An exception might have occured in kernel mode and the cause of the exception needs to be examined. An exception can also have occured in user mode in which case the thread wishes to switch from its user context to kernel context. An interrupt might have occured and CPU time needs to be given to the interrupt handler. This case covers also the special case of a timer interrupt. The timer interrupt is served in an interrupt handler and after the handler returns a new piece of code (thread) is running and the old is waiting for its turn to get the CPU. To be able to do all this transparently, the system needs to save state information on the interrupted thread. This state information is the context of that thread and in `BUENOS` this information is saved in the kernel stack area of the thread.

The exact details of the contents of thread contexts are described later, but the most important part of the data is the contents of the CPU registers. The values of the registers are saved and those of the new thread are put into the CPU registers. Since the registers contain the program counter and the stack pointer, both threads can be unaware of each other. The process of saving the state of one thread and restoring the state of some other thread is called context switching.

It should be noted that threads are not the only entities having execution contexts. Interrupt handler(s) needs to have its own private context which can be used at any time when an interrupt occurs. All context switching and interrupt handling are done in the context of interrupt handling, by using a separate stack area reserved for serving interrupts. The high level interrupt handlers are described in detail in section 9.1.

3.3.1 Interrupt Vectors

First thing to do in order to have proper interrupt/exception handling is to set up the MIPS interrupt/exception handler vectors. This is done during the boot up. Also, in boot, interrupt handler stack `kernel_interrupt_stacks` must be allocated for each CPU present in the `YAMS` simulator.

A few words on the difference of interrupts and exceptions; interrupt is a coordinated interruption of execution caused by raise of either hardware or software interrupt line. Interrupts can be blocked by setting an appropriate interrupt mask. Exceptions and traps are caused by CPU instructions either on purpose (traps to syscalls), as a side effect (TLB miss) or by accident (divide by zero). Exceptions cannot be blocked.

All interrupts and exceptions transfer control to three special Interrupt Vector Areas. These areas are located in memory addresses 0x80000000, 0x80000180 and 0x80000200. The maximum size of these areas is 32 bytes, so each of them can fit only 8 instructions.

It is obvious that the real interrupt handling code cannot be written to area of size of 8 instructions. Therefore, these interrupt vector areas contain only a jump to an assembler routine labeled `cswitch_switch` and a delay slot instruction. This code is labeled as `cswitch_vector_code`. The label is needed so that the code can be injected into the interrupt handler vector area. The size of this code is 8 words (instructions) or 32 bytes[1].

[1]The size of interrupt vector area is mandated by the location of the next interrupt vector. The vector size is cleverly chosen by hardware designers to be long enough to contain TLB refilling code. We avoid that (good, efficient and realistic) solution to make it possible to handle TLB misses in C.

The assembly code in the interrupt vector is just a jump to `_cswitch_switch`-function.

Now, the problem is, how to inject the above assembly code to its proper place in the interrupt handler vector. This is done by finding the interrupt handler code address from label `cswitch_vector_code` and copying two words from there to the memory areas 0x80000000, 0x80000180 and 0x80000200. This code is written in C and is a part of `interrupt_init()` function in `kernel/interrupt.c`.

3.3.2 Context Switching Code

Actual context switch related functions are performed in the `_cswitch_switch` code. This code is written in assembly language because the interrupt handler stack is not yet usable and therefore we cannot use C-functions. We also must be careful that we don't use any registers which are not saved.

The general processing of a context switch is the same for all three causes (kernel mode exception, user mode exception and interrupt) for entering the context switch code. It consists of the following actions (in this order):

Save current context. Data is saved from processor to the `context_t` data structure in the kernel stack area of the currently running thread. The structure of the current thread is pointed by global variable `scheduler_current_thread` (see section 3.2). The current thread is found from `scheduler_current_thread`-table, indexed by CPU number. The following things are saved:

- Co-processor 0 EPC register contains Program Counter value before jump to interrupt handler.
- All CPU registers including hi and lo except k0 and k1.
- Status register (Co-processor 0) fields UM (bit 4), IM0–IM7 (bits 8–15), IE (bit 0). This saves the interrupt mask of the current thread and remembers whether we came from userland (UM bit enabled) or from kernel (UM bit is zero).
- Link to the `context_t` saved to the thread structure. Thus when nested exceptions or interrupts occur, we can unfold this list one reference per context switch and finally come back to the actual running context of the thread.

Initialize stack. A stack is needed to be able to call C functions. If we are going to handle interrupts and/or reschedule threads, we set up stack in the interrupt stack area. In other cases we use thread's kernel stack.

Call the appropriate function to handle the exception/interrupt This is a C function which will take care of the interrupt/exception processing.

Restore new context. After the interrupt/exception is handled, context is restored from `scheduler_current_thread`. Note that in `interrupt_handle` the scheduler might have changed the currently running thread to something else than the one we just saved. Therefore we might start running a new thread at this point.

Return with ERET PC is restored from EPC by this special machine instruction. The EXL bit preventing interrupts is also cleared by the CPU.

In the case of an interrupt, the stack that is initialized is the interrupt context stack. Interrupt stack pointers are defined in the table `kernel_interrupt_stacks`. Table is indexed by CPU number. The stack pointer is set to point to the interrupt

stack reserved for this CPU. Since we don't have nested interrupts, only one stack area per CPU is sufficient. Then the function `interrupt_handle` is called. This C-function will call all registered interrupt handlers and the scheduler, when appropriate. The function is implemented in `kernel/interrupt.c`. Last the context is restored from current thread's `context`.

We use interrupt stacks also for scheduler running, because we cannot continue to run in a stack of some other thread after the context has been switched. If we used the kernel stacks of threads, some other CPU might have picked up our previous thread and run it and thus mess up our stack.

If an exception has occured in kernel mode, it is handled mostly the same way as an interrupt except that instead of calling the function `interrupt_handle`, the function `kernel_exception_handle` is called. The only other difference is that we use the kernel stack area of the current thread instead of the interrupt stack area. The handling function is implemented in `kernel/exception.c`.

When an exception occurs in user mode, the thread wishes to switch from its user context to kernel context. The stack is initialized to the current position of the kernel stack of this thread. The stack information is dug from previous context information, usually from the initial context of the thread.

Because the thread is switching from user mode to kernel mode the base processing mode of the processor (indicated by the UM bit in Status register) is changed to kernel mode. The user mode exceptions are handled by the function `user_exception_handle`, which is implemented in `proc/exception.c`. This function will enable interrupts by setting the EXL bit in the Status register and handle the user mode exception. After returning from this function the context is restored normally from saved context information.

The basic structure of the `cswitch_switch` is thus the following[2]:

```
    .set noreorder
    .set nomacro

cswitch_switch:
    <figure out the appropriate context_t structure>
    j cswitch_context_save
    nop
    <init stack>      # After this we can call C-functions
    <change base mode if appropriate>
    <set up arguments to *_hanlde>
    <call *_handle>
    <figure out the appropriate context_t structure>
    j cswitch_context_restore
    nop
    eret

    .set reorder
    .set macro
```

Note that before the context is saved, we can only use the registers k0 and k1, which are reserved for the kernel by MIPS calling convention.

3.3.3 Thread Contexts

The context of a thread is saved in the `context_t` structure, which is usually referenced by a pointer in `thread_t` in thread table (see section 3.1.1). Contexts are

[2]We need to disable GNU Assembler instruction reordering and macro instruction usage because their interpretation needs some special registers that are not yet saved.

always stored in the stack of the corresponding thread. It has the following fields:

Type	Name	Explanation
uint32_t[29]	cpu_regs	CPU registers except zero, k0 and k1. That makes 29 registers.
uint32_t	hi	The hi register.
uint32_t	lo	The lo register.
uint32_t	pc	PC register which can be obtained from the CP0 register EPC.
uint32_t	status	The saved bits of the CP0 status register.
void *	prev_context	Link to previous saved context. This field links saved contexts up to the point when the thread was initially started.

`kernel/cswitch.S`	`_cswitch_vector_code`, `_cswitch_switch`, `_cswitch_context_save`, `_cswitch_context_restore`
`kernel/interrupt.c`	`interrupt_handle`
`kernel/cswitch.h`	`context_t`

3.4 Exception Processing in Kernel Mode

When an exception occurs in kernel mode, the function `kernel_exception_handle` is called. The cause of an exception in kernel might be a TLB miss or there might be a bug in the kernel code.

`void kernel_exception_handle (int exception)`

- This function is called when an exception has occured in kernel mode. Handles the given `exception`.
- If kernel uses mapped addresses, this function should handle TLB exceptions. Other exceptions indicate that there is some kind of bug in the kernel code.
- Implementation:
  1. Panic with a message telling which exception has occured.

`kernel/exception.h`, `kernel/exception.c`	`kernel_exception_handle`

Exercises

3.1. The context switching code is written wholly in assembler. Why can it not be implemented in C? The code uses CPU registers `k0` and `k1`, but it doesn't touch other registers before the thread context has been saved. Why `k0` and `k1` can be used in the code?

3.2. The current exception system in `BUENOS` doesn't allow interrupts to occur when an interrupt handler is running. What modifications to the system are needed to implement a hierarchical interrupt scheme where higher priority interrupts can occur while lower priority ones are being served?

3.3. The current `BUENOS` scheduler doesn't have any priority handling for threads. Implement a priority scheduler in which all threads can be given a priority value. Higher priority threads will get more processor time than lower priority threads. Your solution should guarantee that no thread will starve (get no processor time at all).

3.4. After you have implemented your priority scheduler, you might have noticed an effect known as *Priority inversion*. This effect is caused by a situation where a high priority thread will block and wait for a resource currently held by a low priority thread. Since there might also be other threads in the system which have higher priority than the thread holding the resource, it may not get any CPU time. Therefore also the high priority thread is hindered as if it had a low priority. How can this problem be prevented?

Chapter 4

Synchronization Mechanisms

The BUENOS kernel has many synchronization primitives which can be used to protect data integrity. These mechanisms are interrupt disabling, spinlocks, the sleep queue and semaphores. Locks and condition variables are left as an exercise.

For an introduction on synchronization concepts, read either [Tanenbaum] p. 100–132 and 159–164 or [Stallings] p. 198–253 and 266–274.

4.1 Spinlocks

A spinlock is the most basic, low-level synchronization primitive for multiprocessor systems. For a uniprocessor system, it is sufficient to disable interrupts to achieve low-level sychronization (a nonpre-emptible code region). When there are multiple processors, this is obviously not enough, since the other processors may still interfere. To achieve low-level interprocessor synchronization, interrupts must be disabled *and* a spinlock must be acquired.

Spinlock acquisition process is very simple: it will repeatedly check the lock value until it is free ("spin"), then set the value to taken. This will of course completely tie up the processor (since interrupts are disabled), so code regions protected by a spinlock should be as short as possible.

Disabling interrupts and spinlock acquiring can (and must) be used in interrupt handlers since they must never cause a sleeping block.

In BUENOS, the spinlock data type is a signed integer containing the value of the lock. Zero indicates that the lock is currently free. Positive values mean that the lock is reserved. The exact value can be anything, as long as it is positive. The value must never be negative (reserved for future extensions).

Due to the nature of the spinlock implementation, spinlocks should never be moved around in memory. In practice this means that they must reside on the kernel unmapped segment which is not part of the virtual memory page pool. This should not be a problem, since spinlocks are purely a kernel synchronization primitive.

4.1.1 LL and SC Instructions

To achieve safe synchronization for spinlock implementation in a multiprocessor system, a version of a *test-and-set* machine instruction is needed. On a MIPS architecture, this is the LL/SC instruction pair. The LL (load linked word) instruction loads a word from the specified memory address. This marks the beginning of a RMW (read-modify-write) sequence for that processor. The RMW sequence is "broken" if a memory write to the LL address is performed *by any processor*. If the RMW sequence was not broken, the SC (store conditional word) instruction will

store a register value to the address given to it (the LL address) and set the register to 1. If the RMW sequence was broken, SC will not write to memory and sets the register to 0.

4.1.2 Spinlock Implementation

The following functions are available to utilize spinlocks. Note that *interrupts must always be disabled when a spinlock is held*, otherwise Bad Things will happen (see exercises below).

`void spinlock_acquire (spinlock_t *slock)`

- Acquire given spinlock. While waiting for lock to be free, spin.
  1. `LL` the address `slock`.
  2. Test if the value is zero. If not, jump to case 1.
  3. `SC` the address to one. If fails, jump to case 1.

`void spinlock_release (spinlock_t *slock)`

- Free the given spinlock.
  1. Write zero to the address `slock`.

`void spinlock_reset (spinlock_t *slock)`

- Initializes the given spinlock to be free.
- Implementation:
  1. Set spinlock value to zero. This is actually an alias to `spinlock_release`.

4.2 Sleep Queue

Thread level synchronization in kernel requires some way for threads to sleep and wait for a resource, like a semaphore, to be available. To avoid the need to implement the sleeping mechanism separately for each such resource, a general sleeping mechanism called sleep queue is implemented in `BUENOS`.

4.2.1 Using the Sleep Queue

The sleep queue enables a thread to wait for a specific resource and to be later woken up by some other thread which has released the resource. The resource, on which the thread is sleeping, is identified by an address. This address must be from the kernel unmapped segment so that the different threads agree on it.

There are three functions which threads can call to manipulate the sleep queue structure. The function `sleepq_add` is called by a thread that wishes to wait for a resource. The functions `sleepq_wake` and `sleepq_wake_all` are called by a thread that wishes to wake up another thread. When using these functions, careful thought has to be given to the synchronization issues involved. The resource on which threads wish to sleep is usually protected by a spinlock. Before calling the sleep queue functions interrupts must be disabled and the resource spinlock must be acquired. This ensures that the thread wishing to go to sleep will indeed be in the sleep queue before another thread attempts to wake it up.

```
1  Disable interrupts
2  Acquire the resource spinlock
3  While we want to sleep:
4    sleepq_add(resource)
5    Release the resource spinlock
6    thread_switch()
7    Acquire the resource spinlock
8  EndWhile
9  Do your duty with the resource
10 Release the resource spinlock
11 Restore the interrupt mask
```

Figure 4.1: Code executed by a thread wishing to go to sleep.

```
1  Disable interrupts
2  Acquire the resource spinlock
3  Do your duty with the resource
4  If wishing to wake up something
5    sleepq_wake(resource) or sleepq_wake_all(resource)
6  EndIf
7  Release the resource spinlock
8  Restore the interrupt mask
```

Figure 4.2: Code executed by a thread wishing to wake up another thread.

If the resource spinlock is not held while calling the sleep queue functions, the following scenario can happen. One thread concludes that it wishes to go to sleep and calls `sleepq_add`. Before this call is serviced, another thread ends its business with the resource and calls `sleepq_wake`. No threads are found in the sleep queue so no thread is woken up. Now the call to `sleepq_add` by the first thread is serviced and the first thread goes to sleep. Thus in the end the resource is free, but the first thread is still waiting for it.

The function `sleepq_add` does not cause the thread to actually go to sleep. It merely inserts the thread into the sleep queue. The thread needs to call `thread_switch` to release the CPU. The scheduler will then notice that the thread is waiting for something and change the state of the thread to `SLEEPING`. This mechanism is needed because the thread needs to release the resource spinlock before actually going to sleep. Because the thread calling `sleepq_add` holds a spinlock, it has also disabled interrupts. Interrupts also need to be disabled to make sure that the thread is not switch out and put to sleep before it is ready to do so. Thus the `sleepq_add` function checks that interrupts are disabled.

The following is an example of the correct usage of the sleep queue. The thread wishing to go to sleep executes the code shown in the Figure 4.1. Lines 1 and 2 ensure protection from other threads using this same resource. The while-loop on line 3 is necessary if it is possible that some other thread can also get the resource. Because we need to release the resource spinlock in the while loop body, another thread might acquire the resource first. The resource spinlock is released on line 5 because the thread cannot hold it while it is not on CPU. Line 6 will make the scheduler choose another thread to run.

The thread, or interrupt handler, wishing to wake up a thread executes the code shown in the Figure 4.2.

Figure 4.3: Linked lists in sleep queue.

4.2.2 How the Sleep Queue is Implemented

Sleep queue is a structure which contains linked lists of threads waiting for a specific resource. The actual structure is implemented as a static size hashtable `sleepq_hashtable` with separate chaining. The chains are implemented using the `thread_table_t`'s `next` field, which is also used for the linked lists (all lists with same hash value are linked in the same list, see the Figure 4.3). New threads are always added to the end of the list and threads are released from the beginning of the chain. This makes the wakeup operation run in shorter time and it is desirable to have it this way, because it is often run in device driver code. Also the first thread in the chain is not necessary the thread we want to wake up.

To protect the hashtable from concurrent access, it is protected by a spinlock `sleepq_slock` . This lock must be held and interrupts must be disabled in all sleep queue operations.

Threads are referenced in the sleep queue system by the resource they are waiting for (sleeping on). The information is stored in `thread_table_t` structure's field `sleeps_on` . Zero in this field indicates that the thread is not waiting for anything. The resource waiting is in practice done by waiting for the address of a resource (a semaphore struct, for example).

Sleep queue functions:

`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. An explicit call to `thread_switch` is needed. The thread will sleep on the given `resource`, which is identified by its 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)
```

- Exactly like `sleepq_wake`, but wakes up all threads which are waiting for the given `resource`.

The sleep queue system is initialized in the boot sequence by calling the following function:

```
void sleepq_init (void)
```

- Sets all hashtable values to -1 (free).

`kernel/sleepq.h`, `kernel/sleepq.c`	Sleep queue operations

4.3 Semaphores

Interrupt disabling, spinlocks and sleep queue provide the low level syncronization mechanisms in `BUENOS`. However, these methods have their limitations; they are cumbersome to use and thus error prone and they require uninterrupted operations when doing processing on a locked resource. Semaphores are higher level synchronization mechanisms which solve these issues.

A semaphore can be seen as a variable with an integer value. Three different operations are defined on a conceptual semaphore:

1. A semaphore may be initialized to any non-negative value.
2. The P-operation[1] decrements the value of the semaphore. If the value becomes negative, the calling thread will block (sleep) and wait until awakened by some other thread in V-operation. (`semaphore_P()`)

[1] The traditional names V and P for operations are the initials of Dutch words for test (proberen) and increment (verhogen).

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

In addition to these operations, we must be able to create and destroy semaphores. Creation can be done by calling `semaphore_create()` and a no longer used semaphore can be freed by calling `semaphore_destroy()`.

4.3.1 Semaphore Implementation

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. Spinlock `semaphore_table_slock` is used to SMP-lock the structure. A semaphore is defined by `semaphore_t`, which is a structure of three fields:

Type	Name	Description
`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`	`creator`	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.

The following functions are defined for semaphores:

`semaphore_t * semaphore_create (int value)`

- Creates a new semaphore and initializes its value to `value`.
- Implementation:
  1. Assert that the initial value is non-negative.
  2. Disable interrupts.
  3. Acquire spinlock `semaphore_table_slock`.
  4. Find free (`creator == -1`) semaphore from `semaphore_table` and set its `creator` to the current thread. If no free semaphores are available `NULL` is later returned.
  5. Release the spinlock.
  6. Restore the interrupt status.
  7. Return with `NULL` if no semaphores were available.
  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`.
- Implementation:
  1. Set the `creator` field in `sem` to -1 (free).

`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 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 `sem`'s value by one.
  4. If the value becomes negative, start sleeping on this semaphore and simultaneously release the spinlock.
  5. Else, release the spinlock.
  6. Restore the interrupt status.

`kernel/semaphore.h`, `kernel/semaphore.c`	Semaphores

Exercises

4.1. 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?

4.2. How could the spinlock acquiring and releasing be improved in efficiency when the kernel is compiled for a uniprocessor system? (Hint: read the spinlock introduction carefully.)

4.3. When waking up a thread in `sleepq_wake` the thread in sleep queue is either *Running* or *Sleeping*. Why can the thread still be *Running*? Consider the usage example of the sleep queue shown in Figure 4.1 and Figure 4.2. What happens if the thread is woken up by some other thread (running on another CPU) between lines 5 and 6 in the code in Figure 4.1?

4.4. 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?

4.5. 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()`

4.6. Locks and condition variables provide an alternative synchronization method to semaphores. Implement locks and Lampson–Redell (Mesa) style condition variables without the timeout rule. (The structure with a lock and several condition variables is also known as a monitor.)

You have to implement procedures for handling lock *acquiring*, *releasing* and condition variable *waiting*, *signaling* and *broadcasting*. You **may not** use semaphores (see section 4.3) to build the locks and condition variables. Use the primitive thread handling routines (defined in chapter 3) and synchronization mechanisms (spinlocks, interrupt disabling and sleep queue) instead. You **must** use the following interface:

For locks:

- `lock_t *lock_create(void)`
- `void lock_destroy(lock_t *lock)`
- `void lock_acquire(lock_t *lock)`
- `void lock_release(lock_t *lock)`

For condition variables:

- `cond_t *condition_create(void)`
- `void condition_destroy(cond_t *cond)`
- `void condition_wait(cond_t *cond, lock_t *condition_lock)`
- `void condition_signal(cond_t *cond, lock_t *condition_lock)`
- `void condition_broadcast(cond_t *cond, lock_t *condition_lock)`

It is up to you to define the `lock_t` and `cond_t` types and provide exact semantics for each of the functions above. Write your lock and condition variable implementation in `kernel/lock_cond.c` and `kernel/lock_cond.h`.

In Lampson–Redell style monitors *signaling* and *broadcasting* will move the thread(s) to the ready list but it is not guaranteed that the thread is the next to run. Thus, the woken thread must recheck the condition before it can continue. What is the other style to define condition variables? What is it called and how the semantics differ from Lampson–Redell? (Remember that, in this exercise, you **have to** implement Lampson–Redell semantics.)

4.7. Implement a synchronized bounded buffer. The buffer has some preset size. You have to implement two synchronized operations on this buffer: `buffer_put` and `buffer_get`. `buffer_put` puts one byte into the buffer and `buffer_get` gets one byte from the buffer. `buffer_put` must block until it has put the byte into the buffer, and `buffer_get` must block until it can return a byte (there is something to return). Use your implementation of *locks and condition variables* as synchronization primitives (No interrupt disabling, no spinlocks, no sleep queue usage, no semaphores).

Test your code by running multiple threads calling `buffer_put` (producers) and multiple threads calling `buffer_get` (consumers).

4.8. Implement a solution for the following toy problem: You have to synchronize chemical reactions needed to form water out of hydrogen and oxygen atoms. Mother nature doesn't seem to get it right because of the synchronization problems involved.

Atoms are represented by threads calling either `hydrogen` or `oxygen` functions. The function calls do not return until the atom is part of a formed water molecule. You must implement these functions as well as `makewater` function which is called by one of the atoms in the just formed new water molecule. The `makewater` function prints a text to the console when the new water molecule has been formed.

Use *semaphores* as synchronization primitives in your implementation (no busy waiting, no sleep queue, no interrupt disabling, no spinlocks).

4.9. Implement a solution for the following toy problem: Mother nature is in trouble again. The whale population in oceans does not seem to grow. The problem seems to be in the complex mating procedure followed by the whales.

Three (!) whales are needed to be present in order to make a successful mating: one male, one female and one matchmaker. The matchmaker will literally push the male and female together.

The whales are represented by threads. The threads call either `male`, `female` or `matchmaker` functions. Both genders and the matchmakers must wait until all three are present and then initiate the mating. After a successful mating, all three functions return.

Use *locks and condition variables* as synchronization primitives in your implementation (no busy waiting, no sleep queue, no interrupt disabling, no spinlocks).

Hint: Matchmaker should be treated as a third gender.

4.10. Implement a solution for the following toy problem: The guild of computer science students uses one room at the university building as a living room for their members. This room has many sofas, but only one rather small table. Many members like to play a card game called Bridge, which requires exactly

four players. The table is so small, that only one card game can be played at a time. The students queuing for their turn to play like to sleep while waiting.

The students wanting to play Bridge are represented by threads. (Those students who do not want to play are ignored.) You have to synchronize the access to the game table. Threads call `student_arrives` function when they enter the room and want to play. This function returns when four players are present at the game table. The return value of the function is the thread ID of the person (thread) on the opposite side of the table (who is called a pair, for Bridge is a team game). When the function has returned, the thread will call `play_bridge` function which should print the ID of the thread, as well as the ID of the thread's partner. After the printing, the function calls `thread_sleep` (if one is available) to simulate the time spent on playing the game. When the `play_bridge` function returns, the thread will call `leave_table` function, which will free the place at the game table for someone else.

Use *semaphores* as synchronization primitives in your implementation (no busy waiting, no sleep queue, no interrupt disabling, no spinlocks).

Note that the requirement which states that the students want to sleep while waiting their turns is implicitly fulfilled when calling `semaphore_P`, since that function forces the thread into sleep while waiting the semaphore value to raise.

4.11. Implement a mechanism which allows threads to sleep for a specified time. Create a function `thread_sleep`, which takes a number of milliseconds as an argument. When a thread calls this function, it will go to sleep. The thread will wake up when at least the given number of milliseconds has passed.

The thread may not wake up before the specified time has elapsed, even to just go back to sleep again. It may however wake up some (short) time later than the specified time (this is not a real time operating system).

Hints: You may find it helpful to use the real time clock driver (see section 9.3.6) and modify the way in which timer interrupts are scheduled in `scheduler_schedule`.

Chapter 5

Userland Processes

BUENOS has currently implemented a very simple support for processes run in userland. Basically processes differ from threads in that they have an individual virtual memory address space. Userland processes won't of course have an access to kernel code except via system calls (see section 5.4). There is currently no separate process table.

Processes are started as regular threads. During process startup in the function `process_start()`, function `thread_go_to_userland()` is called. This function will switch the thread to usermode by setting the usermode bit in the CP0 status register. After this, a context switch is done. Next time the thread is switched to running mode it will run in usermode.

Processes have their own virtual memory address space. In the case of user processes this space is limited to *user mapped* segment of the virtual memory address space. Individual virtual memory space is provided by creating a pagetable for the process. This is done by calling `vm_create_pagetable()` . Because of the limitations of the current virtual memory system, the whole pagetable must fit to the TLB at once. This limits the memory space to 16 pages ($16 * 4096$ bytes). Both the userland binary and the memory allocated for the data must fit in this limited space. More details about virtual memory is found in chapter 6.

Because processes are run in threads, the `thread_t` structure has a few fields for (userland) processes (see section 3.1 and Table 3.1). In context switches `user_context` is set to point to the saved user context of the process. The context follows the regular `context_t` data structure. The `pagetable` field is provided for the pagetable created during process startup. The `process_id` field is currently not used. It could be used for example as an index to a separate process table.

For an introduction to userland and process issues, read either [Stallings] p. 108–142, 154–168, 302–308 and 325–326 or [Tanenbaum] p. 71–80 and 202–207.

5.1 Process Startup

New processes can currently be started by calling the function `process_start`. The function needs to be modified before used to implement the `Exec` system call, but it can be used to fire up test processes.

```
void process_start (char *executable)
```

- Starts one userland process. The code and data for the process is loaded from file `executable`.

- The thread calling this function will be used to run the process. A call to this function will never return.
- Implementation:
  1. Allocate one `context_t` from the stack for the new userland process. (Stack allocation is done simply by declaring the variable inside the function). Since the context switching code expects the context to be in the stack, this is the most convenient way to do that.
  2. Create a new page table for this thread by calling `vm_create_pagetable()`.
  3. Disable interrupts. (Interrupts must be disabled when manipulating thread information so that partial writes into thread entries are never used in case of an interrupt occuring during page table setup.)
  4. Set the new page table as the page table of this thread.
  5. Restore the interrupt status.
  6. Open the `executable` file.
  7. Calculate the total size of both the read-only and the read-write program segments in pages (4096 byte chunks).
  8. Allocate and map the stack for the new process.
  9. Allocate and map pages for both program segments.
  10. Put the mapped pages into the TLB. This must be done manually here before we have a proper virtual memory subsystem. Note that the TLB is filled automatically after threads are switched by the scheduler, so we could replace this force filling by calling `thread_yield()`. Interrupts are disabled during this operation to prevent scheduler's TLB filling code interference.
  11. Fill all allocated pages (including the stack) with zero.
  12. Copy segments from the `executable` into memory by using information provided by the `elf`-library (see below for details on `elf` library). We can use userland virtual addresses as target addresses, because we know for sure that the pages are mapped and are not swapped out (we have no swapping).
  13. Zero all registers in the userland context.
  14. Set the stack pointer into SP-register of the userland context.
  15. Set the program counter (PC) in the userland context.
  16. Call `thread_goto_userland()` , which will never return.

5.2 Userland Binary Format

When a new userland process is created, the code run in this process needs to be loaded from a file. This file needs to be understood by the kernel code which loads the userland binary into the memory. The userland binary format used in `BUENOS` is ELF.

The ELF binary format has sections used for linking, relocation and debugging purposes in addition to storing data and program code, as well as program segments which are the ones relevant to program loading. Each program segment includes one or more of the sections.

The MIPS32 architecture only supports two kinds of memory pages, read-only and read-write. This means that in effect there will be only two program segments

in the binary file, the read-only and read-write segments. The ELF code in `BUENOS` requires that there are indeed at most one of each kind of segments. The segments are as follows:

- `ro_segment`: contains the actual code run in the process (`.text`) as well as read-only data needed by the program (`.rodata`).
- `rw_segment`: contains initialized data needed by the program (`.data`) as well as uninitialized data (`.bss`). The uninitialized data is not stored in the binary and the file only contains the size and addressing information about it.

An ELF executable file is organized in the following way from the program loading viewpoint. The ELF header is in the beginning of the file. It includes a magic string to identify it as an ELF file, as well as the number of program segment headers and their location in the file. These program headers are the ones used when loading the executable into memory. The ELF header also contains the program entry point and information to determine if the file is of the right format (MIPS big-endian), as well as other information which is not relevant to the `BUENOS` ELF loader.

For each program segment there is a header in the ELF file containing (among others) the following relevant information:

- The type of the segment. The ones loaded into memory have a type of `PT_LOAD`.
- The flags for the segment, mainly readable, writable and executable. Only the writable flag is checked by `BUENOS`.
- The virtual address of the beginning of the segment. This is the address that the code uses to reference this segment and the address where the segment should be loaded at.
- The size of the segment stored in the file.
- The size of the segment in memory. Since uninitialized data is not stored in the file, this size may be different from the size that is stored in the file.
- The location of the initialized data (if any) or code in the file.

The current implementation of `BUENOS` contains the function `elf_parse_header` to parse the headers of an ELF file. This function reads the headers from a given file and returns the result in structure `elf_info_t`, which is described in Table 5.1.

`int elf_parse_header (elf_info_t *elf, openfile_t file)`

- Reads the ELF headers from `file` and returns the information about program segments in `elf`.
- Returns 0 on failure (ie. `file` was not a valid ELF file or no program segments were found). Other values indicate success.
- Implementation:
  1. Read the ELF header. If the read fails return 0.
  2. Check that the ELF magic, file format, version and type are correct in the ELF header. If not, return 0.
  3. Zero the `elf` structure.

Type	Name	Explanation
uint32_t	entry_point	The entry point for this program.
uint32_t	ro_location	The location of the read-only segment in the ELF file.
uint32_t	ro_size	The size of the read-only segment stored in the ELF file.
uint32_t	ro_pages	The number of memory pages needed by the read-only segment.
uint32_t	ro_vaddr	The virtual address of the start of the read-only segment.
uint32_t	rw_location	The location of the read-write segment in the ELF file.
uint32_t	rw_size	The size of the read-write segment stored in the ELF file.
uint32_t	rw_pages	The number of memory pages needed by the read-write segment.
uint32_t	rw_vaddr	The virtual address of the start of the read-write segment.

Table 5.1: The structure elf_info_t returned by function elf_parse_header.

4. For each program segment do the following:
   (a) Read the program header from `file`. If the read fails return 0.
   (b) If the program header type is `PT_NULL`, `PT_NOTE` or `PT_PHDR`, continue from the next program header (these types can safely be ignored).
   (c) If the segment type is `PT_LOAD`, check the flags for whether this is the read-only or read-write segment and fill the appropriate fields in `elf`.
   (d) If the segment type is none of the above, this is an unsupported file (not a statically linked executable). Return 0.
5. Return the boolean: *# of loadable segments* > 0

5.3 Exception Handling

When an exception occurs in user mode the context switch code switches the current thread from user context to kernel context. The thread will resume its execution in kernel mode in function `user_exception_handle`. This function will handle the TLB misses and system calls caused by the userland process.

`void user_exception_handle (int exception)`

- This function is called when an exception has occured in user mode. Handles the given `exception`.
- Implementation:
  1. Dispatch system calls to the syscall handler, PANIC on other exceptions.

`proc/exception.c`	`user_exception_handle`
`proc/elf.h`, `proc/elf.c`	`elf_parse_header()`
`proc/syscall.c`	System call handling
`proc/process.h`, `proc/process.c`	Process management

5.4 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).

5.4.1 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 `BUENOS`, the standard MIPS argument registers `a0--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.

After the arguments are in place, the special machine instruction `syscall` is executed. It generates a system call exception and thus transfers control to the kernel exception handler. The return value of the system call is placed in a predefined register by the system call handler. In `BUENOS` the standard return value register `v0` is used.

The system call exception is handled then 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.

Note: You cannot directly change thread/process (ie. 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`).

5.4.2 System Calls in `BUENOS`

`BUENOS` has a wrapper function for the `syscall` instruction, so there is no need to write code in assembler. In addition, some syscall function numbers are specified (in `proc/syscall.h`) and wrapper functions with proper argumets for these are implemented in `tests/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 `BUENOS`. 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 `BUENOS`. 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.

File System Related

`int syscall_open (const char *filename)`

- Open the file identified by *filename* 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 *filename, int size)`

- Create a file with the name *filename* and 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), ie. 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 *filename)`

- Remove the file identified by *filename* 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_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 zero, 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 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.

Process Related

`void syscall_exit (int retval)`

- Terminate the current process with the exit code *retval*.
- Note that *retval* must be non-negative since negative return values for `syscall_join` are interpreted as errors in the join call itself.
- This function never returns.

`int syscall_exec (const char *filename)`

- Create a new process (child process), load the file identified by *filename* and execute it as the created process.
- Returns the process ID (PID) of the created process, or a negative value on error.

`int syscall_join (int pid)`

- Wait until the execution of the child process identified by *pid* is finished.
- Returns the exit code of the joined process, or a negative value on error.
- This call should work correctly and return the exit code of a once started process, even if the process to be joined has already finished execution before or during this call. (These processes are usually called *zombies*.)

Extra System Calls

These are actually also process related, but since their implementation is beyond the scope of the basic system call exercise, they are listed in their own section.

`int syscall_fork (void (*func)(int), int arg)`

- Create a new thread running in the same address space as the caller.
- The new thread will start at function *func* and the thread will end when *func* returns. *arg* is passed as an argument to *func*.
- Returns 0 on success and a negative value on error.

This system call is implemented in one virtual memory exercise in chapter 6.

`void * syscall_memlimit (void *heap_end)`

- Allocate or free memory by trying to set the heap to end at the address *heap_end*.
- Returns the new end address of the heap (the last addressable byte), or NULL on error.
- If *heap_end* is NULL, the current heap end is returned.

If you implement argument passing between parent and child processes, use this version of exec instead of the standard one (see exercises below).

`int syscall_execp (const char *filename, int argc, const char **argv)`

- Creates a new process (child process), loads the file identified by *filename* and executes it as the created process.
- Passes `argc` arguments to the child process.
- The arguments are in a table of string pointers (`char *`), and there are thus `argc` rows in the table `argv` which holds the argument strings.
- Returns the process ID (PID) of the created process, or a negative value on error.

Exercises

5.1. The userland binary is divided into different segments: `text` segment, `rdata` segment, `data` segment and `bss` segment. In addition to these, the userland program has a stack, but this is not defined in the binary. What is the purpose of each of these segments?

The binary could be loaded into memory in one big chunk if these segments were not defined. Which of these could be set read only in memory and what benefits would that gain? What are the other advantages of this segmented approach?

5.2. Implement a way to transfer data safely between kernel and userland. When implementing system calls, various data blocks need to be transferred between userland process memory and kernel memory regions. It must not be possible for the userland process to fool the kernel into giving it access rights to the memory space of other processes or kernel memory areas (even one written or read byte in the wrong place is extra access).

You need to provide two types of functionality: One to move blocks of predefined size between kernel and userland and the other to safely transfer strings (C strings, the length is not known in advance but can have a reasonably big upper limit, ends when a 0 byte is encountered).

5.3. Implement process entries. You need to provide a synchronized data structure to store information on running userland processes. The entry for a process must contain at least the name of the process (the binary file name is ok, useful for debugging) and the thread(s) which belong to it. All threads associated with userland processes must also know which process they belong to.

You also need to add fields in this data structure for all process-related information needed to implement system calls properly (see next exercise).

5.4. Implement system calls. Implement all predefined system calls except `fork`, `execp` and `memlimit`. The system calls must be bulletproof so that the only way userland processes can stop the system is the `halt` system call and there is no way for any userland process to interfere with other processes.

You don't need to fix the filesystem to provide proper synchronized access to the same files, but you need to make sure that processes don't interfere with the open file handles of other processes (no filesystem or VFS modifications should be needed, but are allowed).

Note that you can add other system calls if you wish, but the predefined set must work as documented so that your operating system can run precompiled binaries built against the system call definitions.

Note also that this exercise implies that you must handle exception conditions caused by userland processes in some other sensible manner than the current `PANIC`, since the current approach gives userland processes an easy way to shut the system down without calling `halt`.

5.5. Implement a shell. A shell is a userland program which interacts with the user through the console and enables the user to start programs by typing names of programs. The shell must make it possible to start programs into the background (shell use continues) and into the foreground (the shell is not usable until the started process ends). It must be possible to exit from the shell.

The shell must print the return value of a started (foreground) process when the process finishes. Can you find a good way to inform the user when a background process has finished and print its return value?

5.6. Implement a set of userland programs to test your system call implementation. Make sure that you test all implemented system calls. The programs should do something at least remotely useful (like copy files). If you do not implement arguments for programs (see exercise below), you can hard-code the parameters into the test programs.

Remember also to test that your system calls do not do more than they are supposed to do! Note also that the shell can be used as a test program for some syscalls.

5.7. Implement the system call `fork`. Fork enables you to run multiple threads in the context of one process and thus bring the SMP threading capabilities of the `BUENOS` kernel into userland.

Remember to plan how the `exit` system call behaves when a process has multiple threads. When does the process actually end? (First to `exit`?, Last to `exit`? Original thread `exit`s?).

5.8. Add a way to pass arguments from one userland process calling `execp` to the started child process. You must use the version of `execp` presented in the section 5.4.2. Note that the system call ids for for both `exec` and `execp` are the same, so that `exec` should be backward binary compatible with your new `exec` implementation..

Arguments are defined as an arbitrary (0 to N) number of strings. You can of course set some configurable upper limit on the number of arguments and/or their size.

The newly created process should receive its arguments as arguments to the C function `main()`. Study the calling convention (section 2.2.9) before starting this assignment.

Chapter 6

Virtual Memory

By definition, virtual memory provides an illusion of unlimited sequential memory regions to threads and processes. Also the VM subsystem should isolate processes so that they cannot see or manipulate memory allocated by other processes. The current `BUENOS` implementation does not achieve these goals. Instead, it provides tools and utility functions which are useful when implementing a real and working virtual memory subsystem.

Currently the VM subsystem has primitive page tables for threads and processes, utilities to manipulate hardware TLB and a simple mechanism for allocating and freeing physical pages. There is no swapping, the pagetables are inefficient to use and hardware TLB is used in a very limited way. Kernel threads must also manipulate allocated memory directly by pages. Suggested improvements are documented as exercises at the end of this chapter.

As result of this simple approach, the system can support only 16 pages of mappings (64 kB) for each (userland) process. These 16 mappings can be fit into the TLB and are currently done so by calling `tlb_fill` after changing threads by the scheduler. The system does not handle TLB exceptions.

The current kernel implementation does not use mapped memory. It also does all its memory reservations through pagepool, which is described in section 6.3. Since kernel needs both virtual addresses for actual usage and physical address for hardware, simple mapping macros are available for easy conversion. These macros are `ADDR_PHYS_TO_KERNEL()` and `ADDR_KERNEL_TO_PHYS()` and they are defined in `vm/pagepool.h`. Note that the macros can support only kernel region addresses which are within the first 512MB of physical memory. See below for description on address regions.

6.1 Hardware Support for Virtual Memory

The hardware in `YAMS` supports virtual memory with two main mechanisms: memory segmentation and translation lookaside buffer (TLB). The system doesn't support hardware page tables. All page table operations and data structures are defined by the operating system. The page size of the hardware is 4 kB (4096 bytes). All mappings are done in page sized chunks.

Memory segmentation means that addresses of different regions of address space behave differently. The system has 32-bit address space.

If the topmost bit of an address is 0 (the first 2GB of address space), the address is valid to use even if the CPU is in user mode (not in kernel mode). This region of addresses is called `user mapped region` and it is used in userland programs and in kernel when userland memory is manipulated. This region is *mapped*. Mapping

means that the addresses do not refer to real memory addresses, but the real memory page is looked up from TLB when an address in this region is used. The TLB is described in more detail in its own section (see section 6.5).

The rest of the address space is reserved for the operating system kernel and will generate an exception if used when the CPU is in user (non-privileged) mode. This space is divided into four segments: `kernel unmapped uncached`, `kernel unmapped`, `supervisor mapped` and `kernel mapped`. Each segment is 512MB in size. The supervisor mapped region is not used in `BUENOS`. The kernel unmapped uncached region is also not used in `BUENOS` except for memory mapped I/O-devices (YAMS doesn't have caches).

The kernel mapped region behaves just like the user mapped region, except that it is usable only in kernel mode. This region can be used for mapping memory areas for kernel threads. The area is currently unused, but its usage might be needed in proper VM implementation.

The kernel unmapped region is used for static data structures in the kernel and also for the kernel binary itself. The region maps directly to the first 512MB of system memory (just strip the topmost bit in an address).

In some parts of the system a term *physical memory address* is used. Physical addresses are addresses starting from 0 and extending to the top of the machine's real memory. These are used for example in TLB to point to actual pages of memory and in device drivers when doing DMA data transfers.

6.2 Virtual memory initialization

During virtual memory initialization (function `vm_init`) page pool data structure is initialized (see section 6.3) and the ability to do arbitrary length permanent memory reservation (i.e. `kmalloc`) is disabled. `kmalloc` is disabled so that it will not mess up with dynamically reserved pages.

6.3 Page Pool

Page pool is a data structure containing the status of all physical pages. The status of a physical page is either free or reserved. This status information (of the nth page) is kept in (the nth bit of) a bitmap field `pagepool_free_pages`, zero meaning a free and one a reserved page.

A spinlock is provided to secure synchronous access to the bitmap field. It is needed to prevent two (or more) threads from reserving the same physical page. Note that when you modify the virtual memory system to support swapping, these pagepool functions must still work because they are used in device drivers, networking and filesystem code. You can reserve a certain amount of physical memory for the kernel (pagepool) and rest for userland processes (mapped access) if you wish.

`void pagepool_init ()`

- Initializes the pagepool. After this it is known which pages may be used by virtual memory system for dynamic memory reservation. Statically reserved pages are marked as reserved.
- Implementation:
  1. Find out total number of physical pages from `kmalloc`.
  2. Reserve space for `pagepool_free_pages` bitmap field. Note that this is still a permanent memory reservation.

Type	Name	Explanation
uint32_t	ASID	Address space identifier. The entries placed in TLB will be set with this ASID. Only entries in TLB with ASID matching with ASID of the currently running thread will be valid. In BUENOS we use ASID == Thread ID.
uint32_t	valid_count	Number of valid mapping entries in this pagetable.
tlb_entry_t [PAGETABLE_ENTRIES]	entries	The actual page mapping entries in the form accepted by hardware TLB. See also section 6.5.1 for description of TLB entries.

Table 6.1: Pagetable (pagetable_t) structure fields

3. Find out the number of reserved pages from kmalloc. This is the total amount of reserved memory divided by page size, rounded up.
4. Mark all reserved pages as ones in bitmap field.

Following pagepool handling functions are provided to handle page pool data structure.

`uint32_t pagepool_get_phys_page ()`

- Returns the physical address of a free page. If no free pages are available, returns zero.
- Function finds first zero bit from pagepool_free_pages and marks it to one. The address is calculated by multiplying the bit number with page size.

`void pagepool_free_phys_page (uint32_t phys_addr)`

- Frees a physical page by setting the corresponding bit to zero.
- Asserts that the freed page is *a)* reserved and *b)* is not statically reserved.

6.4 Pagetables and Memory Mapping

BUENOS uses very primitive pagetables to store memory mappings for userland programs. Each thread entry in the system has private pagetable field in its information structure. If the entry is NULL, thread is a kernel-only thread. If the entry is available, thread is used in userland.

The pagetable stores virtual address physical address mapping pairs for the process. Virtual addresses are private for the process, but physical addresses are global and refer to actual physical memory locations. The pagetable is stored in pagetable_t structure described in Table 6.1. The internal representation is the same as used by hardware TLB. See section 6.5.1 for details on TLB entries.

To use memory mapping, thread must create a pagetable by calling the function `vm_create_pagetable()` giving its thread ID as an argument. This pagetable is then stored in thread's information structure. For an example on usage, see `process_start()` in `proc/process.c`. Note that the current VM implementation cannot handle TLB dynamically, which means that TLB must be filled with proper mappings manually before running thread (userland process) which needs them. This can be achieved by calling `tlb_fill()` (see `proc/process.c: process_start()` and `kernel/interrupt.c: interrupt_handle()` for current usage).

When the thread no longer needs its memory mappings, it must destroy its pagetable by calling `vm_destroy_pagetable()`. Note that this only clears the mappings, but does not invalidate the pagetable entry in thread information structure, free the physical pages used in mappings or clear the TLB. These things must be handled by the thread wishing to free memory (eg. a dying userland process).

`pagetable_t * vm_create_pagetable (uint32_t asid)`

- Creates a new pagetable. Returns pointer to the table created.
- Argument `asid` defines the address space identifier associated with this page table. In `BUENOS` we use asids which equal to thread IDs.
- Pagetable occupies one hardware page (4096 bytes).
- Implementation:
  1. Reserve one physical memory page from pagepool. This page will contain one `pagetable_t` structure.
  2. Set the ASID field in the created structure.
  3. Set the number of valid mappings to 0.
  4. Return the created pagetable.

`void vm_destroy_pagetable (pagetable_t *pagetable)`

- Frees the given pagetable.
- Pagetable must not be used after it is freed. The freeing is done when thread is finished or userland program terminates.
- Note that this function does not invalidate any entries present on TLB on any CPU.
- Implementation:
  1. Free the page used for the pagetable by calling pagepool's freeing function.

Memory mappings can be added to pagetables by calling `vm_map()`. Note that with the current implementation threads should manipulate only their own mappings, not mappings of other threads. The current TLB implementation cannot handle more than 16 pagetable mappings correctly, a better system is left as an exercise.

Mappings can be removed one by one with `vm_unmap()`, but implementation is left as an exercise. The dirty bit of a mapping can be changed by calling `vm_set_dirty()`.

```
void vm_map (pagetable_t *pagetable, uint32_t physaddr,
     uint32_t vaddr, int dirty)
```

- Maps the given virtual address (`vaddr`) to point to the given physical address (`physaddr`) in the context of given `pagetable`. The addresses must be page aligned (4096 bytes).
- If `dirty` is true, the mapping is marked dirty (read/write mapping). If false, the mapping will be clean (read-only).
- Implementation:
  1. If the pagetable already contains the pair entry for the given virtual address (page), the pair entry is filled. Pagetables use hardware TLB's mapping definitions where even and odd pages are mapped to the same entry but can point to different physical pages.
  2. Else creates new mapping entry, fills the appropriate fields and invalidates the pairing (not yet mapped) entry.

```
void vm_unmap (pagetable_t *pagetable, uint32_t vaddr)
```

- Unmaps the given virtual address (`vaddr`) from given `pagetable`. The address must be page aligned and mapped in this `pagetable`.
- Implementation:
  1. This function is not implemented, the implementation is left as an exercise.

```
void vm_set_dirty (pagetable_t *pagetable, uint32_t vaddr, int dirty)
```

- Sets the dirty bit to `dirty` of a given virtual address (`vaddr`) in the context of the given `pagetable`. The address must be page aligned (4096 bytes).
- If `dirty` is true (1), the mapping is marked dirty (read/write mapping). If false (0), the mapping will be clean (read-only).
- Implementation:
  1. Find the mapping of the given virtual address.
  2. Set the dirty bit if a mapping was found.
  3. If the mapping was not found, panic.

6.5 TLB

Most modern processors access virtual memory through a Translation Lookaside Buffer (TLB). It is an associative table inside the memory management unit (MMU, CP0 in MIPS32) which consists of a small number of entries similar to page table entries mapping virtual memory pages to physical pages.

When the address of a memory reference falls into a mapped memory range (`0x00000000-0x7fffffff` or `0xc0000000-0xffffffff` in MIPS) the virtual page of the address is translated into a physical page by the MMU hardware by looking it up in the TLB and the resulting physical address is used for the reference. If the virtual page has no entry in the TLB, a TLB exception occurs.

6.5.1 TLB dual entries and ASID in MIPS32 architectures

In MIPS32 architecture, one TLB entry always maps two consecutive pages, even and odd. This needs to be taken into account when implementing the TLB handling routines, as a new mapping may need to be added to an already existing TLB entry. One might think that the consecutive pages could be mapped in separate entries, leaving the other page in the entry as invalid, but this would result in duplicate TLB matches and thus cause undefined behavior.

A MIPS32 TLB entry also has an Address Space ID (ASID) field. When the CP0 is checking for a TLB match, also the *ASID* of the entry must match the current *ASID* for the processor, specified in the *EntryHi* register (or the global bit is on, see `YAMS` and MIPS32 documentation for details). Thus, when using different *ASID* for each thread, the TLB need not necessarily be invalidated when switching between threads.

`BUENOS` uses `tlb_entry_t` structure to store page mappings. The entries in this structure are compatible with the hardware TLB. The fields are described in Table 6.2.

The exception handler in `kernel/exception.c` should dispatch TLB exceptions to the following functions, implemented in `vm/tlb.c` (note that the current implementation does not dispatch TLB exceptions):

`void tlb_load_exception (void)`

- Called in case of a TLB miss exception caused by a load reference.

`void tlb_store_exception (void)`

- Called in case of a TLB miss exception caused by a store reference.

`void tlb_modified_exception (void)`

- Called in case of a TLB modified exception.

6.5.2 TLB miss exception, Load reference

The cause of this exception is a memory load operation for which either no entry was found in the TLB (TLB refill) or the entry found was invalid (TLB invalid). These cases can be distinguished by probing the TLB for the failing page number. The exception code is *EXCEPTION_TLBL*.

6.5.3 TLB miss exception, Store reference

This exception is the same as the previous except that the operation which caused it was a memory store. The exception code is *EXCEPTION_TLBS*.

6.5.4 TLB modified exception

This exception occurs if an entry was found for a memory *store* reference but the entry's D bit is zero, indicating the page is not writable. The D bit can be used both for write protection and pagetable coherence when swapping is enabled (dirty/not dirty). The exception code is *EXCEPTION_TLBM*.

6.5.5 TLB wrapper functions in `BUENOS`

The following wrapper functions to CP0 TLB operations, implemented in `vm/_tlb.S`, are provided so that writing assembler code is not required.

Type	Name	Explanation
unsigned int:19	VPN2	Virtual page pair number. These are the upper 19 bits of a virtual address. VPN2 describes which consecutive 2 page (8192 bytes) region of virtual address space this entry maps.
unsigned int:5	dummy1	Unused
unsigned int:8	ASID	Address space identifier. When ASID matches CP0 setted ASID this entry is valid. In BUENOS, we use mapping ASID = Thread ID.
unsigned int:6	dummy2	Unused
unsigned int:20	PFN0	Physical page number for even page mapping (VPN2 + 0 bit).
unsigned int:3	C0	Cache settings. Not used.
unsigned int:1	D0	Dirty bit for even page. If this is 0, page is write protected. If 1 page can be written.
unsigned int:1	V0	Valid bit for even page. If this bit is 1, this entry is valid.
unsigned int:1	G0	Global bit for even page. Cannot be used without the global bit of odd page.
unsigned int:6	dummy3	Unused
unsigned int:20	PFN1	Physical page number for odd page mapping (VPN2 + 1 bit).
unsigned int:3	C1	Cache settings. Not used.
unsigned int:1	D1	Dirty bit for odd page. If this is 0, page is write protected. If 1 page can be written.
unsigned int:1	V1	Valid bit for odd page. If this bit is 1, this entry is valid.
unsigned int:1	G1	Global bit for odd page. Cannot be used without the global bit of even page. If both bits are 1, the mapping is global (ignores ASID), otherwise mapping is local (checks ASID).

Table 6.2: TLB entry (tlb_entry_t structure fields)

`void _tlb_get_exception_state (tlb_exception_state_t *state)`

- Get the state parameters for a TLB exception and place them in `state`.
- This is usually the first function called by all TLB exception handlers.
- Implementation:
  1. Copy the *BadVaddr* register to `state->badvaddr`.
  2. Copy the *VPN2* field of the *EntryHi* register to `state->badvpn2`.
  3. Copy the *ASID* field of the *EntryHi* register to `state->asid`.

The structure `tlb_exception_state_t` has the following fields:

Type	**Name**	**Explanation**
`uint32_t`	`badvaddr`	Contains the failing virtual address.
`uint32_t`	`badvpn2`	Contains the VPN2 (bits 31..13) of the failing virtual address.
`uint32_t`	`asid`	Contains the ASID of the reference that caused the failure. Only the lowest 8 bits are used.

`void _tlb_set_asid (uint32_t asid)`

- Sets the current ASID for the CP0 (in *EntryHi* register).
- Used to set the current address space ID after operations that modified the *EntryHi* register.
- Implementation:
  1. Copy `asid` to the *EntryHi* register.

`uint32_t _tlb_get_maxindex (void)`

- Returns the index of the last entry in the TLB. This is one less than the number of entries in the TLB.
- Implementation:
  1. Return the *MMU size* field of the *Conf1* register.

`int _tlb_probe (tlb_entry_t *entry)`

- Probes the TLB for an entry defined by the *VPN2*, *dummy1* and *ASID* fields of `entry`.
- Returns an index to the TLB, or a negative value if a matching entry was not found.
- Implementation:
  1. Load the *EntryHi* register with VPN2 and ASID.
  2. Execute the *TLBP* instruction.
  3. Return the value in the *Index* register.

`int _tlb_read (tlb_entry_t *entries, uint32_t index, uint32_t num)`

- Reads `num` entries from the TLB, starting from the entry indexed by `index`. The entries are placed in the table addressed by `entries`.
- Only `MIN(TLBSIZE-index, num)` entries will be read.
- Returns the number of entries actually read, or a negative value on error.
- Implementation:
  1. Load the *Index* register with `index`.
  2. Execute the *TLBR* instruction.
  3. Move the contents of the *EntryHi*, *EntryLo0* and *EntryLo1* registers to corresponding fields in `entries`.
  4. Advance `index` and `entries`, and continue from step 1 until enough entries are read.
  5. Return the number of entries read.

`int _tlb_write (tlb_entry_t *entries, uint32_t index, uint32_t num)`

- Writes `num` entries to the TLB, starting from the entry indexed by `index`. The entries are read from the table addressed by `entries`.
- Only `MIN(TLBSIZE-index, num)` entries will be written.
- Returns the number of entries actually written, or a negative value on error.
- Implementation:
  1. Load the *Index* register with `index`.
  2. Fill the *EntryHi*, *EntryLo0* and *EntryLo1* registers from `entries`.
  3. Execute the *TLBWI* instruction.
  4. Advance `index` and `entries`, and continue from step 1 until enough entries are written.
  5. Return the number of entries written.

`void _tlb_write_random (tlb_entry_t *entry)`

- Writes the `entry` to a "random" entry in the TLB. The entry is read from `entry`.
- Note that if this function is called more than once, it is *not* guaranteed that the newest write will not overwrite the previous, although this is usually the case. This function should only be called to write a single entry.
- Implementation:
  1. Fill the *EntryHi*, *EntryLo0* and *EntryLo1* registers from `entry`.
  2. Execute the *TLBWR* instruction.

The following function should be used only until a proper VM implementation is done:

`void tlb_fill (pagetable_t *pagetable)`

- Fills the TLB of the current CPU with entries from given `pagetable`. Supports only 16 mappings and cannot be used if `pagetable` might contain more mappings.
- If the `pagetable` is `NULL`, the TLB is not touched.
- Implementation:
  1. Return if `pagetable` is `NULL`.
  2. Assert that there are no more mappings than TLB can handle.
  3. Write entries to TLB.
  4. Set ASID in CP0 to match ASID of the pagetable (equals to thread ID in `BUENOS`).

`vm/vm.h, vm/vm.c`	Virtual Memory core, pagetable handling, memory mapping
`vm/pagepool.h, vm/pagepool.c`	Pagepool implementation, address mapping macros
`vm/pagetable.h`	Pagetable definitions
`vm/tlb.h, vm/tlb.c, vm_tlb.S`	TLB manipulation

Exercises

6.1. Implement software management for the TLB. The current implementation in `BUENOS` simply fills the TLB with page mappings after each scheduler run. This is not sufficient, because only 16 pages can be mapped this way. The approach is also slow, because many unneeded pages are also mapped. Write handlers for TLB exceptions and make it possible to use any page mapped for this purpose even if there are more than 16 mappings.

Note that you need handlers for both userland and kernel exceptions.

6.2. Implement better page tables. The current `BUENOS` page tables are limited to 340 page mappings. Implement a solution which makes it possible to efficiently map any number of available pages in a pagetable. Your solution must:

- Make it possible to map any sensible number of pages in a pagetable.
- Implement an efficient way to find a mapping for a given virtual page from a page table (linear search is not efficient).
- Support page unmapping (write the implementation for `vm_unmap` function).

6.3. Implement paging. Write a solution which allows the system to extend physical memory to disk and run larger programs than the system memory can hold. It is sufficient to make paging possible only for memory used by userland processes.

Hints: You can add a new disk to the system to represent a "swap partition" if you wish. Keep the pagepool (see section 6.3) functional, it is used in many

places in the kernel code (including disk handling). You can reserve a part of the system memory for the pagepool and the rest for user programs if you want to. You can decrease the amount of available memory in `YAMS` for easier testing.

6.4. Refine your paging implemented in the previous assignment. Implement on-demand loading for userland programs. In on-demand loading, pages are filled only when they are used the first time. Text segments (code) and initialized data will be read from the binary and un-initialized data will be filled with zeroes when used for the first time. Avoid writing any such page to swap which could be read from the binary when needed.

6.5. Make it possible for kernel threads to allocate mapped memory. Implement new memory allocation routines, which allocate memory from the page pool and map it to kernel thread's pagetable. Threads should be able to reserve and free a memory chunk of any size (within the limits of available memory and possible swap). Remember to make it possible for threads to free the allocated memory properly without causing too much memory fragmentation.

6.6. Evaluate the performance of your virtual memory system. Cache misses (in our case TLB misses and page faults) can be divided into three different categories:

(a) Compulsory misses happen when a page is referenced for the first time. There is no way to avoid a compulsory miss.

(b) Capacity misses occur when the cache size is too small and a page must be replaced by another page. However, a miss is only counted as a capacity miss if the replacement could not be avoided with an optimal replacement policy.

(c) Conflict misses occur when the replacement policy has performed sub-optimally and the miss could have been avoided if correct choices would have been made in the replacement algorithm.

Instrument the kernel to count the number of different misses for both TLB misses and page faults (swap ins). Print all six numbers when the kernel shuts down with the halt system call.

Write a set of userland programs which stress the virtual memory system in different ways (produce large amounts of different kind of misses).

Hint: decrease the available memory in `YAMS` to introduce more swapping.

6.7. Implement a memory allocation library for the userland. Extend the userland libc to contain `malloc` and `free` functions, which behave as normally in C. The interfaces for the functions must be the following:

- `void *malloc(int size)`
- `void free(void *ptr)`

To be able to implement these functions, you must also implement the system call *memlimit*, defined in the section 5.4.2.

Chapter 7

Filesystem

Filesystem is a collection of files which can be read and usually also written. `BUENOS` can support multiple filesystems at the same time, thus you can attach (mount) several different filesystems on different mount-points at any time.

`BUENOS` has one implemented filesystem, which is called Trivial Filesystem (see section 7.4). Filesystems are managed and accessed through a layer called Virtual Filesystem which represents a union of all available filesystems (see section 7.3).

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.

For an introduction on filesystem concepts, read either [Stallings] p. 483–493, 515–518 and 526–550 or [Tanenbaum] p. 300–302, 315–322 and 379–428.

7.1 Filesystem Conventions

Files on filesystems are referenced with filenames. In `BUENOS` 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 as well as possible directory and the name of the file.

An example of a valid filename is `shell`. A full absolute path to a shell might be `[root]shell` or `[root]bins/shell`. Here `shell` is the name of a file, `root` is a volumename (you could also call it disk, filesystem or mount-point). If directories are used `bins` is a name of a directory. Directories have the same restrictions on filenames as files do[1]. Directories are separated by slashes.

7.2 Filesystem Layers

Typically a filesystem is located on a disk (but it can also be a network filesystem or even totally virtual[2]). Disks are accessed through Generic Block Devices (`gbd`, see section 9.2.4). 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.

[1] This should be logical, especially when we consider that usually directories are implemented as files.

[2] Totally virtual filesystems do not have any real files. The contents are created on the fly by the kernel. An example of this is the `/proc`-filesystem in Unix which has one virtual directory for each process in the system and these directories contain virtual files which tell the process name, memory footprint size, etc.

Virtual filesystem is a super-filesystem which contains all attached (mounted) filesystems. The same access functions are used to access local, networked and fully virtual filesystems. The actual filesystem driver is recognized from the volume name part of a full absolute pathname provided to the access functions.

7.3 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 attached filesystems are referenced with names, which are called mount-points or volumes.

VFS provides a set of file access functions (see section 7.3.5) and a set of filesystem access functions (see section 7.3.6). 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 attach (mount) filesystems into VFS, detach filesystems and get information on mounted filesystems (free space on volume). A mechanism for forceful unmounting of all filesystems is also provided. This mechanism is needed when the system performs shutdown and to prevent filesystem corruption.

To be able to provide these services, VFS keeps track of attached (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 (but see exercises below).

7.3.1 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 defined return values are negative. The full list of values is:

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.)

Type	Name	Explanation
`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.

Table 7.1: Mounted filesystem information structure (`vfs_entry_t`)

Type	Name	Explanation
`semaphore_t *`	`sem`	A binary semaphore used to lock access to this table.
`vfs_entry_t [CONFIG_MAX_ FILESYSTEMS]`	`filesystems`	Table of mounted filesystems.

Table 7.2: Table of mounted filesystems (`vfs_table`)

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.

7.3.2 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 (`VFS_NAME_LENGTH == 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 (`VFS_PATH_LENGTH == 256`).

Type	Name	Explanation
`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.

Table 7.3: VFS information on open file (`openfile_entry_t`)

Type	Name	Explanation
semaphore_t *	sem	A binary semaphore used to lock access to this table.
openfile_entry_t [CONFIG_MAX_ OPEN_FILES]	files	Table of open files.

Table 7.4: Table of open files in VFS (openfile_table)

7.3.3 Internal Data Structures

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

The table of all filesystems, vfs_table, is described in Table 7.1 and Table 7.2. 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 described in Table 7.3 and Table 7.4. This table is also protected by a semaphore (openfile_table.sem). Whenever the table is altered, this semaphore must be held.

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. 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.

7.3.4 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 unusable.
  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 operation on VFS. Operation is any such action which touches a filesystem.
- Returns `VFS_OK` if the operations can continue, 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`.

7.3.5 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 described by `pathname`. The name must include both the full pathname and the filename. (e.g. `[root]shell`)
- Returns an openfile identifier. Openfile 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 mountpoint, etc.), call `vfs_end_op` and return with error code `VFS_ERROR`.
  4. Acquire locks to the filesystem table and the openfile table by calling `semaphore_P` on `vfs_table.sem` and `openfile_table.sem`.
  5. Find a free entry in the openfile table. If no free entry is found (the table is full), free locks by calling `semaphore_V` on `openfile_table.sem` and `vfs_table.sem`, 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` from the filesystem table. If the volume is not found, return with the same procedure as for full openfile table except that the error code is `VFS_NO_SUCH_FS`.
  7. Allocate the found free openfile entry by setting its filesystem field.
  8. Free the locks by calling `semaphore_V` on `openfile_table.sem` and `vfs_table.sem`.
  9. Call filesystem's `open` function. If the return value indicates error, lock the openfile table by calling `semaphore_P` on `openfile_table.sem`, mark the entry free and free the lock with `semaphore_V`. Call `vfs_end_op` and return the error given by the filesystem.
  10. Save the fileid returned by the filesystem in the openfile 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 openfile table as the openfile 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 openfile table by calling `semaphore_P` on `openfile_table.sem`.
  3. Verify that the given `file` is really open (kernel panics if it is not).
  4. Call `close` on the actual filesystem for the `file`.
  5. Mark the entry in the openfile table free.
  6. Free the openfile table by calling `semaphore_V` on `openfile_table.sem`.
  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 behaviour 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 error is returned by it, return immediately with error code `VFS_UNUSABLE`.
  2. Locks the openfile table by calling `semaphore_P` on `openfile_table.sem`.
  3. Verify that the `file` is really open (panic if not).
  4. Set the new seek position in openfile table.
  5. Free the openfile table by calling `semaphore_V` on `openfile_table.sem`.
  6. Call `vfs_end_op`.
  7. Return `VFS_OK`.

Files are read and written by the following two functions:

```
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 `file` is really open (panic if not).
  3. Call the `read` function of the filesystem.

4. Lock the openfile table by calling `semaphore_P` on `openfile_table.sem`.
5. Update the seek position in the openfile table.
6. Free the openfile table by calling `semaphore_V` on `openfile_table.sem`
7. Call `vfs_end_op`.
8. Return the value returned by 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 `file` is really open (panic if not).
  3. Call the `write` function of the filesystem.
  4. Lock the openfile table by calling `semaphore_P` on `openfile_table.sem`.
  5. Update the seek position in the openfile table.
  6. Free the openfile table by calling `semaphore_V` on `openfile_table.sem`
  7. Call `vfs_end_op`.
  8. Return the value returned by 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`, 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 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.)
  5. Find the filesystem from the filesystem table. If it is not found, free the table by calling `semaphore_V` on `vfs_table.sem`, call `vfs_end_op` and return with the error code `VFS_NO_SUCH_FS`.

6. Call filesystem's `create`.
7. Free the filesystem table by calling `semaphore_V` on `vfs_table.sem`.
8. Call `vfs_end_op`.
9. Return with the return code of filesystem's `create` function.

`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`, 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 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.)
  5. Find the filesystem from the filesystem table. If it is not found, free the table by calling `semaphore_V` on `vfs_table.sem`, call `vfs_end_op` and return with the error code `VFS_NO_SUCH_FS`.
  6. Call filesystem's `remove`.
  7. Free the filesystem table by calling `semaphore_V` on `vfs_table.sem`.
  8. Call `vfs_end_op`.
  9. Return with the return code of filesystem's `remove` function.

7.3.6 Filesystem Operations

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

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

`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 all the following steps:
  2. Get the device entry for the disk by calling `device_get`.

3. Dig the generic block device entry from the device descriptor.
4. Attempt to mount the filesystem on the disk by calling `vfs_mount_fs` with `NULL` 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`). The mount is performed by trying out all available filesystem drivers in `fileystems.c`. The first match 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 mount-point.
- Returns `VFS_OK` (zero) on success, negative on error (no matching filesystem driver or too many mounted filesystems).
- Implementation:
  1. Try out `init` functions of all available filesystems in `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 invalid, 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.

7.3.7 Filesystem Driver Interface

Filesystem drivers are implemented in `BUENOS` by creating a set of functions which map into the function pointers required to fill the filesystem driver information structure `fs_t`. This structure is described in Table 7.5.

In addition to these functions, an initialization function returning `fs_t` pointer and taking a generic block device (disk) as an argument must be implemented. When this function is called it determines whether the given disk contains the filesystem supported by this driver. If not, it returns `NULL`. If the filesystem matches, a filled `fs_t` is returned. All values must be valid (not `NULL`) in the returned structure pointer.

When the filesystem driver functions, specified by the function pointers in the `fs_t` structure are called, the `fs_t` pointer from which they are called is also given as an argument (treat like `this` in object oriented languages).

Type	Name	Explanation
`void *`	`internal`	Internal data of the filesystem driver.
`char [16]`	`volume_name`	Advisory mount-point name filled by the filesystem driver.
`int (*)(struct fs_struct *fs)`	`unmount`	A function pointer to a function which unmounts this driver instance from the disk. A call to this function also invalidates the `fs_t` pointer to this struct. The filesystem on the disk must be in a stable state when this function returns.
`int (*)(struct fs_struct *fs, char *filename)`	`open`	A function pointer to a function which opens a file on the filesystem. The name of the file without the mount-point part is given as an argument. Returns a non-negative file id, negative values are errors.
`int (*)(struct fs_struct *fs, int fileid)`	`close`	A function pointer to a function which closes an open file specified by the argument `fileid`. Returns `VFS_OK` (zero) on success, negative on failure.
`int (*)(struct fs_struct *fs, int fileid, void *buffer, int bufsize, int offset)`	`read`	A function pointer to a function which reads at most `bufsize` bytes from the file specified by `fileid` into the `buffer` starting from `offset`. The number of bytes read is returned. Zero is returned only at the end of the file (nothing read). Negative values are errors. The filesystem drivers should always attempt to read the requested number of bytes if possible.

Continued on the next page

Table 7.5: Filesystem driver information structure (`fs_t`)

Continued from the previous page

Type	Name	Explanation
`int (*)(struct fs_struct *fs, int fileid, void *buffer, int datasize, int offset)`	`write`	A function pointer to a function which writes at most `datasize` bytes from the `buffer` into the file specified by `fileid` starting write from file location `offset`. The number of bytes written is returned. Any return value other than `datasize` is an error, negative values are specific errors and positive values partial writes (can occur when the filesystem fills up, for example).
`int (*)(struct fs_struct *fs, char *filename, int size)`	`create`	A function pointer to a function which creates a new file with the given `filename` and `size`. Returns the standard VFS return codes.
`int (*)(struct fs_struct *fs, char *filename)`	`remove`	A function pointer to a function which removes the file with the given `filename`. Returns the standard VFS return codes.
`int (*)(struct fs_struct *fs)`	`getfree`	A function pointer to a function which returns the number of free bytes on the filesystem. Negative values are errors.

Table 7.5: Filesystem driver information structure (`fs_t`)

The newly implemented driver must be added to the filesystem information structure `filesystems` in `fs/filesystems.c` with the name of the filesystem driver and the init function. This table is used by the following function (called from within VFS) to probe possible filesystems on disks:

```
fs_t * filesystems_try_all (gbd_t *disk)
```

- Tries to mount the given `disk` with all available filesystem drivers in `filesystems`.
- Return initialized filesystem driver (return value from its `init` function), or `NULL` if no match was found.
- Implementation:
  1. Loop through all known filesystem drivers and call `init` on each. If a match is found (non-`NULL` return value), return the filesystem driver instance.

Figure 7.1: Illustration of disk blocks on a TFS volume

2. If no match is found, return `NULL`.

7.4 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) file system and a tool (see **??**) for moving data between TFS and the native filesystem of the platform on which `YAMS` is run.

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 `BUENOS` 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), it just means writing programs for other platforms than `BUENOS`.)

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. Figure 7.1 illustrates the structure of the TFS.

Note that all multibyte data in TFS is *big-endian*. This is not a problem in `BUENOS`, since `YAMS` is big-endian also, but must be taken into consideration if you want to write e.g. TFS debugging tools (native to the simulator platform).

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) for TFS volumes.
0x04	char [TFS_VOLUMENAME_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 content of the file is stored to the allocated blocks in the order they appear in the block list (the first BLOCKSIZE 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.
0x04	uint32_t [TFS_BLOCKS_MAX]	block	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).

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 BUENOS 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.

7.4.1 TFS Driver Module

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

```
fs_t * tfs_init (gbd_t *disk)
```

- Attempts to initialize a TFS on the given disk (a generic block device, actually). 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 ($\geq$ 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 ($\geq$ 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.

`fs/vfs.h, fs/vfs.c`	Virtual Filesystem implementation
`fs/filesystems.h, fs/filesystems.c`	Available filesystems
`fs/tfs.h, fs/tfs.c`	TFS implementation

Exercises

Note that your filesystem, and other exercises in this chapter, must use a new disk. First, create a new disk device with a blocksize of 128 bytes by adding the entry defined in section C.1 into your `yams.conf`.

Generic hints: Do not modify TFS or `tfstool`, make copies and name them for example SFS (student filesystem) and `sfstool`. This way you can still use `tfstool` to transfer files to the system's TFS volume and you only need to support filesystem creation with your own tool. Note that you must compile SFS and `sfstool` with the 128 byte blocksize (which is a configuration definition in the filesystem header file). Remember to include correct headers in your own tool (`sfs.h`, not `tfs.h`).

You don't have to make TFS comply to constraints given in this or any other exercise, it is enough that your filesystem and VFS are correct. You can still use the old disk for the TFS.

7.1. Improve the concurrency of the filesystem. Modify the filesystem so that:

- The same file may be read or written concurrently by any number of processes.
- All filesystem operations must be atomic and serializable. This means that all reads and writes must look like atomic operations, although in reality they are done concurrently. Thus for example when a thread writes to a file all readers see either the whole write or none of it.
- File deletion must work even when the file is open by one or more threads. If the file is deleted and it is currently open in some thread, only new opens on that file must fail and all already opened file handles must work until they are closed. The blocks on a disk are released only after the file is not open with any thread[3].

7.2. Create a filesystem with large files. Your filesystem must support files up to the size of the disk and disks of any size up to 1 megabyte. Note that for future extensibility, do not make the block pointer types any smaller than in TFS (let them be 32-bit wide). Note also that retrieving a block from a disk takes quite a lot of time. Make sure that your design is fast enough to be feasible.

7.3. Make files extensible. If a file is written beyond its end, the file is extended so much that the write is possible. This also makes it possible to create files with length 0 and expand them as needed.

7.4. Implement directories. Directories can contain files and other directories, forming a hierarchical namespace together with mount-point identifiers. For example, a full pathname to shell could be `[root]bins/sys/shell`.

[3] Typical way to use temporary files is to first create the file, then open it and finally delete it. The removal will then be handled automatically when the process exits.

Hints: Handle directories as files internally. Plan carefully how concurrent access to directories is handled.

7.5. Implement a kernel information filesystem. The filesystem should be a virtual filesystem, which is not on any disk. It can be mounted normally to any mountpoint. The filesystem should contain one file per each process in the system and each file describes the current status of the corresponding process. Status information should include at least process ID, name of executable, memory usage and current thread state (running, sleeping, ready for running). If you have implemented userland threads, replace the thread state with the number of threads the process currently runs.

7.6. Improve the performance of your filesystem in the case of many concurrent users. The typical ways to improve filesystem performance are:

(a) Implement a mechanism to use a part of system main memory as a cache for disk blocks. Three main styles for doing this are:

- A fixed sized (but configurable) chunk of memory is used for caching.
- All otherwise unused memory pages are used for caching.
- The virtual memory system will treat cache pages and pages used by programs equally. Note that cache page might be clean (read cache) or dirty (write cache).

Evaluate these three alternatives and implement the one you consider to be the best. You can of course use your own scheme if you find it superior to all of these.

(b) Implement an I/O-scheduler for disk access. The current method of handling disk read and write requests is a strict FIFO. Implement a better `disksched_schedule()` function which will improve system performance by:

- Taking into account the disk block number where the requests in the queue are made on. The seeks of the disk read/write head take a lot of time and much speed can be gained by considering its movement when ordering requests.
- Make sure that no request can ever completely starve.
- If you have implemented a priority scheduler, consider also using thread priorities as a parameter when ordering disk requests (note that the disk scheduler is currently run in the context of the thread which made the request).

(c) Tune the filesystem so that it will try to place blocks that are usually used sequentially close to each other (like blocks of one file). Together with a good disk scheduler, this should also improve overall performance.

Write also a set of test programs which demonstrate the performance improvements gained. Analyze the performance gains. You might need to instrument the operating system to get measurable results out of your test program runs.

Chapter 8

Networking

The implementation of `BUENOS` networking is organized in layers. Each layer adds some more functionality to the lower layers. The device driver implementing the Generic Network Device (GND) interface can be thought of as the bottommost layer of the network stack. This layer issues commands to the network device and handles interrupts generated by the device. The implementation of this layer was left as an exercise to the students. The GND interface is documented in section 9.2.5. Some hints about implementing the device driver are given in section 9.3.3.

Above the device driver layer resides the network frame layer discussed in section 8.1. This layer abstracts the possibly multiple GNDs found in the system. Packets are received from all GNDs and forwarded further to the correct upper layer packet handler.

Packet Oriented Protocol (POP) implements an abstraction similar to the unix sockets. It allows packets sent by different entities in the same machine to be distinguished. The implementation of POP is further explained in section 8.2.

A stream oriented protocol is left as an exercise to the students. This layer should add connections and reliability to the services provided by POP. Some hints about the implementation of a stream oriented protocol are given in section 8.3.

For more information on advanced networking topics, see [Stallings] p. 699–707, 586–590 and 608–615.

8.1 Network Services

Frame layer transfers frames through (possible) multiple Network Interface Cards (NIC) abstracted by GNDs. There is a receive service thread for each GND and when a frame is received it is forwared to the appropriate upper level frame handler. Frames given to be sent are sent through the appropriate GND.

Network addresses in `YAMS` are one word long (32 bits). There are two kinds of special addresses:

- *Addresses containing all zeros* are loopback addresses. While sending they are pushed immediately to the upper level frame handlers.
- *Addresses containing all ones* are broadcast addresses. `broadcast address, network` These frames are sent through all GNDs.

The frame consists of header and payload as described in Table 8.1. Frame size is limited by page size of the virtual memory system. This is because there is no way of reserving two consecutive memory pages and device drivers handle *physical addresses*. Payload size is therefore page size minus header size.

Type	Name	Explanation
network_frame_header_t	header	Header of the frame.
uint8_t []	payload	Payload to be transferred.

Table 8.1: Fields in structure network_frame_t

Type	Name	Explanation
network_address_t	destination	Destination address of the frame.
network_address_t	source	Source address of the frame.
uint32_t	protocol_id	The higher level protocol id for this frame.

Table 8.2: Fields in structure network_frame_header_t

The frame header consists of source and destination addresses and the protocol identification for payload. Source and destination addresses belong to the frame header of YAMS network devices, but the protocol identification field is considered as payload by YAMS. The header is described in Table 8.2.

Upper Level Protocols

All upper level protocols are defined in a static table network_protocols. Table entry, defined as network_protocols_t, contains the following information:

Type	Name	Explanation
uint32_t	protocol_id	Typecode of the protocol.
frame_handler_t	frame_handler	Pointer to the function that will handle the payload of the frame.
void (*)(void)	init	Initialization function of the protocol.

frame_handler_t is a function type which behaves like this:

```
int frame_handler (network_address_t source,
      network_address_t destination, uint32_t protocol_id,
      void *payload)
```

- Upper level handler for the frame (payload). Takes as parameters source and destination addresses of the frame, protocol identification of the frame and payload of the frame.

An initialization function protocols_init() is provided. Function calls initialization function for each protocol in network_protocols table.

Initialization

In network initialization all network devices are searched and network_interfaces table is set up. Also socket and protocol initialization functions are called here. For each GND found a receive service thread is started.

The following network initialization function is provided.

`void network_init (void)`

- Initializes networking code. Also calls initialization functions for sockets and protocols. Starts a receive thread for each GND found.
- Implementation:
  1. Mark all entries as unused (gnd == NULL) in the network interface table.
  2. Find all network interfaces by `device_get`, get their GNDs and store GNDs, addresses and MTUs in the table. For each MTU, assert that it is smaller than page size (the page size is 4096 bytes).
  3. Create and run a thread for each GND. All the threads will run `network_receive_thread` with a pointer to the GND as the argument.

Receive Service Thread

For each network interface found a receive service thread is started. The main job is to allocate memory for frames to be received and when a frame is recieved call the `network_receive()` function. Each thread has one interface attached to it (given as parameter) and frames are received through it.

The receive service thread is implemented as follows:

`void network_receive_thread (uint32_t interface)`

- Receives frames from the given network device ad infinitum. Calls the function `network_receive_frame()` which will call upper level frame handler.
- Index to the `network_interfaces` table is given as parameter.
- Implementation:
  1. Allocate a page for frame receiving. Assert errors.
  2. Call `GND->receive`.
  3. If a frame is succesfully received, call `network_receive_frame()`.
  4. Go back to step one.

`static int network_receive_frame (network_frame_t *frame)`

- Finds a frame handler for the appropriate upper level protocol and calls it.
- Implementation
  1. Find the frame handler by calling `protocols_get_frame_handler()`. The protocol ID found from frame is given as parameter.
  2. If found call it and return its value.
  3. Else return zero as failure.

Upper level frame handlers must free the page reserved for the frame by calling `network_free_frame()`.

Service API

Following functions are provided as service API. Upper level protocols may be implemented on top of these.

`network_address_t network_get_source_address (int n)`

- Get the local address of the `n`th network interface. Returns 0 if no such interface exists.
- Implementation:
  1. Check that n $\geq$ 0 and smaller than `CONFIG_MAX_GNDS`. If not, return 0.
  2. Get the `n`th entry from the table. If GND is not NULL, return the address, otherwise return 0.

`network_address_t network_get_broadcast_address (void)`

- Gets the global broadcast address.
- Implementation:
  1. Return `0xffffffff`.

`network_address_t network_get_loopback_address (void)`

- Gets the loopback address.
- Implementation:
  1. Return 0.

`int network_get_mtu (network_address_t local_address)`

- Gets the MTU of a GND. The frame header (12 bytes) is decremented from the size of the frame. If the broadcast address is given, minimum of all GND's MTU is returned.
- Implementation:
  1. If broadcast address: go trough all GNDs and find the minimun MTU.
  2. Else: find `local_address` from the GND table and get the MTU.
  3. If not found, return 0.
  4. Return MTU - 12.

```
int network_send (network_address_t source,
     network_address_t destination, uint32_t protocol_id, int length,
     void *buffer)
```

- Sends one packet to network. Blocks.
- If the `source` is broadcast address, the frame is broadcast on all network interfaces (with the interface's address as source, of course).
- Returns positive value on success and negative on failure.
- Implementation:
  1. ASSERT that the `length` is smaller or equal to 4084 (page size - 12)
  2. Allocates page for the packet with `pagepool_get_physical_page`. If page allocation fails, return NET_ERROR.

3. If `destination` is loopback, push frame to upper levels by calling `network_receive_frame()` immediately and return.
4. If `source` is broadcast: for each interface, do the following steps using interface address as source. If the `source` is not broadcast do the following steps only once.
5. Find `source` address from GND table (local address).
6. If not found, return `NET_DOESNT_EXIST`.
7. Call `network_send_interface()`.
8. Return success or failure. Negative values indicate failure and zero or positive values success.

```
static int network_send_interface (int interface,
     network_address_t destination, network_frame_t *frame)
```

- Sends a frame through the given interface. This is a helper function to ease handling in more complex functions.
- Implementation
  1. Get `gnd` from `network_interfaces` table.
  2. Call `gnd->send()`.

```
void network_free_frame (void *frame)
```

- Frees the given frame. Called from protocol-specific frame handler after the frame is handled.
- Implementation:
  1. Call `vm_free_page()`.

8.2 Packet Oriented Transport Protocol

Packet Oriented Protocol (POP) is very similar to UDP. Port numbers are used to identify different entities on the same machine. POP offers unreliable delivery from one entity on one machine to another entity on another machine.

The port numbers are implemented by a socket abstraction which is very similar to the sockets found in UNIX like operating systems. A socket is bound to a port number which can be given explicitly when creating the socket or it may be chosen randomly by `BUENOS`. The implementation of POP includes functions to open and close sockets and to send and receive a packet through an opened socket. The socket implementation is further discussed in section 8.2.1.

POP also needs some structures and functions that are not essentially a part of the socket abstraction. These include the format of a POP packet and a queue for incoming packets. A thread which places the incoming packets to correct receive buffers is also needed. These issues and the implementation of the protocol specific functions of the socket abstraction are discussed in section 8.2.2.

Type	Name	Explanation
uint16_t	port	The port that this socket is bound to.
uint8_t	protocol	The protocol id of the protocol used with this socket.
void *	rbuf	This is the address of the receive buffer if some thread is currently waiting for input from this socket, NULL if there is no waiting thread.
uint32_t	bufsize	Size of the receive buffer. No more than this number of bytes are copied to the buffer.
network_address_t *	sender	When a packet is received the sender's address is stored here.
int *	copied	When a packet is received, the number of copied bytes is stored here.
uint16_t *	sport	When a packet is received, the port number of the sender is stored here.

Table 8.3: Fields in structure socket_descriptor_t

8.2.1 Sockets

Open sockets are stored in a static size table called open_sockets. This table contains entries of the form socket_descriptor_t which is described in Table 8.3. The size of this table is determined by CONFIG_MAX_OPEN_SOCKETS. The access to this table is synchronized by a semaphore, open_sockets_sem.

The socket implementation has three functions, socket_init, socket_open and socket_close. In addition to these, the POP implementation includes functions socket_sendto and socket_recvfrom. The stream-oriented transport protocol is left as an exercise to the students. The implementation of this protocol should include functions like socket_connect, socket_read , socket_listen and socket_write.

```
void socket_init (void)
```

- Initializes the structures needed to implement the socket abstraction.
- Implementation:
  1. Ensure that this function is called only once.
  2. Initialize open_sockets_sem to 1 (free) and assert that the initialization succeeds.
  3. Initialize all open sockets to empty (protocol 0).

```
sock_t socket_open (uint8_t protocol, uint16_t port)
```

- This function will create a socket and bind it to the given port. A handle for the socket is returned.
- `protocol` is 0x01 for POP.
- `port` is the port to bind to. If set to 0, `BUENOS` will select a free port.
- Implementation:
  1. Check that `protocol` is one of the supported ones, return error (-1) if not.
  2. Call `semaphore_P` on `open_sockets_sem`.
  3. Find a free socket descriptor from the table. If the table was full, call `semaphore_V` on `open_sockets_sem` and return error.
  4. If `port` is 0, find the first unused port by looking through all open sockets in the table.
  5. Otherwise check that the port is unused. If the port is in use call `semaphore_V` on `open_sockets_sem` and return error.
  6. Save `protocol` and `port` into the table entry and initialize other fields to 0 or NULL.
  7. Call `semaphore_V` on `open_sockets_sem`.
  8. Return the index to the socket table.

```
void socket_close (sock_t socket)
```

- This function unbinds the `socket` in question. The socket can no longer be used after this.
- Implementation:
  1. Check that `socket` has a valid value.
  2. Call `semaphore_P` on `open_sockets_sem`.
  3. Mark the entry in the socket table as unused by setting the protocol to 0 and also zero all other fields.
  4. Call `semaphore_V` on `open_sockets_sem`.

8.2.2 POP-Specific Structures and Functions

POP defines its own packet format which is described in Table 8.4. The header includes the port values which are used to distinguish different entities in a machine. The source and destination network addresses are found in the lower level network headers and are therefore not included in the POP header. POP allows an application to send packets of length less or equal to the network MTU (including all headers). To know where the data ends POP header thus contains the SIZE field, which tells the payload data length in bytes.

The main functionality of POP is to send and receive packets. Sending in POP is done synchronously. That is, the sending thread is used to send the packet so that no packet queueing is needed. Receiving, however, needs to be done asynchronously so POP contains a queue for received packets. The structure of entries in the queue for received POP packets is presented in Table 8.5. The queue is of static length defined by `CONFIG_POP_QUEUE_SIZE`. Access to the POP queue is protected by a semaphore, `pop_queue_sem`.

Offset	Name	Size	Description
0	SPORT	2	Source port; the sender is bound to this port.
2	DPORT	2	Destination port; the receiver listens to (is bound to) this port.
4	SIZE	4	The size of the payload data in bytes.
8	DATA	variable	The payload data, of length SIZE.

Table 8.4: POP packet format

Type	Name	Explanation
`void *`	`frame`	The received packet.
`sock_t`	`socket`	The socket that will receive this packet.
`uint32_t`	`timestamp`	The time when this packet was put to the queue.
`network_address_t`	`from`	The address of the sender.
`int`	`busy`	1 if this entry is in use, 0 otherwise.

Table 8.5: Structure for entries in the pop queue.

When a frame arrives, the `BUENOS` network layer examines the protocol number in the frame header and calls the appropriate frame handler. The frame handler for POP is `pop_push_frame`. This function will place the arrived packet to the POP queue. When POP is initialized, a service thread is created. This thread continually scans the POP queue and delivers packets to applications if they are ready to receive packets.

The POP implementation includes the following functions:

`void pop_init (void)`

- Initialize the POP layer by emptying the entries in the `pop_queue` and starting the POP service thread.
- Implementation:
  1. Ensure that this function is executed only once.
  2. Assert that POP header has the correct lenght.
  3. Allocate a page for the send buffer and assert that this succeeds.
  4. Create the three needed semaphores (`pop_send_buffer_sem`, `pop_queue_sem` and `pop_service_thread_sem`) and assert that this succeeds.
  5. Initialize all entries in the `pop_queue` to empty.
  6. Start the service thread.

```
int pop_push_frame (network_address_t fromaddr,
     network_address_t toaddr, uint32_t protocol_id, void *frame)
```

- Place the frame into the POP frame queue and wake up the POP service thread. If there is no space in the queue, return 0. `frame` points to the

beginning of the page containing the frame, and the frame will include the from and to addresses of the frame layer (ie. it is in the full frame format).

- `frame` is a page allocated by the caller (frame layer). When the POP layer has no more need for the page it will call `network_free_frame(frame)`.
- Returns 1 if the frame was accepted (placed in the queue) and 0 if not. In case of return value 0, the caller may free or reuse the frame immediately.
- Implementation:
  1. Check that the `protocol_id` is POP.
  2. Call `semaphore_P` on `pop_queue_sem`.
  3. Search the queue for an empty slot. If no empty slot was found, find the oldest *nonbusy* entry. If the oldest entry is younger than `CONFIG_POP_QUEUE_MIN_AGE`, call V on `pop_queue_sem` and return 0.
  4. For the selected entry, set the frame field to `frame`, socket field to -1, `from` to `fromaddr`, timestamp to `rtc_get_msec` and busy to 0.
  5. Call `semaphore_V` on `pop_queue_sem`.
  6. Call `semaphore_V` on `pop_service_thread_sem` to signal the POP service thread.
  7. Return 1 to indicate that the frame was accepted.

`void pop_service_thread (uint32_t dummy)`

- This function runs in its own thread delivering incoming POP packets to right receive buffers and discarding packets whose destination port is not listened. When there is nothing to do, the service thread will wait on the `service_thread_sem`.
- Implementation: repeat the following ad infinitum:
  1. Call `semaphore_P` on `open_sockets_sem`.
  2. Call `semaphore_P` on `pop_queue_sem`.
  3. Find the first nonempty entry in the pop queue.
  4. If its destination port is not listened, mark the queue entry as empty and call `network_free_frame`. The call must be postponed after the semaphore release because many semaphores are held.
  5. If the destination port is listened but no one is waiting for a packet for that socket (receive buffer is NULL), find the next nonempty frame and repeat from the previous step.
  6. If the destination port is listened and someone is waiting for a packet, mark the queue entry as busy and mark the frame (function internal) to be transferred.
  7. Call `semaphore_V` on `pop_queue_sem`.
  8. Call `semaphore_V` on `open_sockets_sem`.
  9. If a frame was marked to be discarded, call `network_free_frame` and mark the row in the queue as unused.
  10. If a frame was marked to be transferred, do the following:

(a) Transfer the proper amount of POP payload bytes to the receive buffer of the socket and set the `sender`, `sport` and `copied` fields to corresponding values (sockets need not be synchronized since no one should touch our socket when it is in waiting state).
(b) Mark the receive buffer for the socket as NULL.
(c) Mark the queue entry as empty (no synchronization is needed here either, since no one *else* will touch busy entries).
(d) Call `network_free_frame` for the frame.
(e) Wake the thread waiting for the transfer to complete..

11. If any frames were processed (transferred or freed), repeat from step 1.
12. Call `semaphore_P` on `pop_service_thread_sem`.

The following functions are actually part of the socket interface but they are implemented by POP.

```
int socket_sendto (sock_t s, network_address_t addr, uint16_t dport,
     void *buf, int size)
```

- Send `size` bytes from buffer `buf` to address `addr`, port `dport`, using socket `s`.
- Implementation:
  1. Check that `s`, `size` and `buf` are sane, return error (-1) if not.
  2. Limit `size` so that the whole frame will fit into one page.
  3. Call `semaphore_P` on `open_sockets_sem`.
  4. Check that the given socket is a POP socket.
  5. Copy the entry indexed by `s` to a local variable.
  6. Call `semaphore_V` on `open_sockets_sem`.
  7. Call `semaphore_P` on `pop_send_buffer_sem`.
  8. Fill the POP header located at the start of `pop_send_buffer` with PRID=0x01, RSRVD=0x00, SPORT=port from the socket entry, and DPORT=dport.
  9. Move `size` bytes from `buf` to the data area in the POP packet.
  10. Call `network_send` using broadcast address as source address so that the packet will be sent through all network interfaces.
  11. Call `semaphore_V` on `pop_send_buffer_sem`.
  12. Return the number of payload bytes sent or error if `network_send` returned error.

```
int socket_recvfrom (sock_t s, network_address_t *addr,
     uint16_t *sport, void *buf, int maxlength, int *length)
```

- Receive at most `maxlength` bytes from network using socket `s`, storing the received data into buffer `buf`. The sender's address is stored in `*addr`. The number of actually received bytes is stored in `*length`.
- Implementation:
  1. Check that the parameters are sane.
  2. Call `semaphore_P` on `open_sockets_sem`.

3. If the `rbuf` field of the socket is not NULL, release the semaphore and return -1 (someone else is waiting for a packet for the same socket, this is not supported). Also check that this is a POP socket.
4. Set the fields `rbuf`, `bufsize`, `sender`, `sport` and `copied` for `s` from the arguments.
5. Call `semaphore_V` on `open_sockets_sem`.
6. Wake up the POP service thread by calling `semaphore_V` on the semaphore `pop_service_thread_sem`.
7. Wait until the packet has been transfered by calling `semaphore_P` on `receive_complete` semaphore in the socket structure.
8. Return the number of bytes received.

8.3 Stream Oriented Protocol API

The existing network implementation doesn't support connection oriented reliable sockets. This kind of sockets provide reliable communication on unreliable network and can transfer arbitrary number of bytes on single connection. The interface (for non-exisisting protocol) to stream sockets is following (see also `net/sop.h`):

```
int socket_connect (sock_t s, network_address_t addr, int port)
```

- Connects to remote machine (address `addr`) at port `port`. The connection remains open until explicitly closed by call to `socket_close()` or connection is lost.
- Return 0 on success, 1 on failure.

```
void socket_listen (sock_t s)
```

- Waits until given socket `s` has been connected by someone (listen on server socket).

```
int socket_read (sock_t s, void *buf, int length)
```

- Reads at most `length` bytes from given socket `s`. The data read is written to buffer `buf`.
- Returns the number of bytes read. Zero indicates end of stream and negative values are returned on errors.

```
int socket_write (sock_t s, void *buf, int length)
```

- Writes `length` bytes to given socket `s`. The data is read from buffer `buf`.
- Returns the number of bytes successfully delivered to the destination. If the return value is not equal to `length`, an unrecoverable error has occured and the socket connection is lost.

`net/network.h`, `net/network.c`	Network frame layer
`net/protocols.h`, `net/protocols.c`	List of available network protocols
`net/socket.h`, `net/socket.c`	Socket library
`net/pop.h, net/pop.c`	Packet oriented unreliable networking protocol
`net/sop.h`	Stream oriented reliable networking protocol API (no implementation available)

Exercises

8.1. Implement a reliable stream oriented network protocol. The interface to the protocol is described in section 8.3.

8.2. Implement a network filesystem. The filesystem should be mountable to the standard VFS interface (see section 7.3). The server side implementation must support multiple simultaneous clients on the same filesystem at the same time. Userland programs must be able to use network filesystem just like a local filesystem.

8.3. Implement process migration through network. Any userland process must be able to call new system call (you define it) and give an address of a target machine. The process is then migrated into that new machine. All already open files must work normally after the migration, but console prints will go to the console of the new host machine. The process can re-migrate at any time it wishes.

Chapter 9

Device Drivers

Since `BUENOS` runs on a complete simulated machine, it needs to be able to access the simulated devices in `YAMS`. These hardware devices include system consoles, disks and network interface adapters. 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 arrival of a network frame, occurs. Device drivers implement handlers for these interrupts and react to the event.

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 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.

For an introduction on device drivers and hardware, read either [Tanenbaum] p. 269–300 and 327–341 or [Stallings] p. 474–486.

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`.

Table 9.1: Fields in structure `interrupt_entry_t`

9.1 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 Table 9.1.

```
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 booting.
- 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:

Figure 9.1: BUENOS device abstraction layers.

1. Clear software interrupts.
2. Call the appropriate interrupt handlers.
3. Call the scheduler if appropriate.

kernel/interrupt.h, kernel/interrupt.c	interrupt_entry_t, interrupt_register, interrupt_handle

9.2 Device Abstraction Layers

The device driver interface in BUENOS 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 BUENOS: generic character device, generic block device and generic network device. These can be thought as "superclasses" from which the actual device drivers are inherited. The hierarchy of device driver abstractions is shown in Figure 9.1.

Generic character device is a device which provides uni- or bidirectional bytestream. The only such device preimplemented in BUENOS 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. However there is no device driver implementing this interface since the network device driver is left as an exercise.

All device drivers must have an initialization function. Pointer to this function is placed in a structure drivers_available in drivers/drivers.c together with a device typecode identifier. The system will initialize the device drivers in bootup for each device in the system by calling these initialization functions. This initialization is done in device_init().

9.2.1 Device Driver Implementor's Checklist

When implementing a new device driver for BUENOS at least the following things must be done:

1. Place new driver in drivers/.
2. Implement functions which provide interface to the device for threads. If possible, use generic device abstractions.

Type	Name	Explanation
`uint32_t`	`typecode`	The typecode of the device this driver is intended for.
`const char *`	`name`	The name of this driver. Printed to console before the driver is initialized.
`device_t * (*)(io_descriptor_t *descriptor)`	`initfunc`	A pointer to the initialization function for the driver. Starts the driver for the hardware device described by `descriptor` and return pointer to the device driver instance.

Table 9.2: Fields in structure `drivers_available_t`

3. Implement interrupt handler for the device.

4. Implement initialization function which will allocate and initialize device structure and register the interrupt handler.

5. Put the device driver's initialization function in `drivers_available` table in `drivers/drivers.c`.

6. Use `volatile` keyword in the variable declarations that can be changed during the execution of a thread (e.g., when the process is sleeping, interrupted, ...). (The `volatile` keyword tells the compiler that the variable in question can be changed without any action taken by the code nearby the variable.)

9.2.2 Device Driver Interface

Device driver initialization functions are placed in table `drivers_available`. The structure of an entry in that table is shown in Table 9.2.

Every device driver's initialization function must return a pointer to device descriptor (`device_t`) for this device. The descriptor structure is explained in Table 9.3.

The device entry has a field of type `io_descriptor_t *`. This refers to device descriptor record provided by the hardware (`YAMS`). This structure is thus not allocated, but just referenced from hardware device descriptor area in memory. The fields are documented in detail in `YAMS`'s manual, but are also shown in Table 9.4.

In system boot-up, device driver initialization code is called from `init()`. 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 drivers/drivers.c`).

Type	Name	Explanation
`void *`	`real_device`	Pointer to the device driver's internal data structures.
`void *`	`generic_device`	Pointer to a generic device handle (generic character device, generic network device or generic block device). Will be NULL if the device driver does not implement any generic device interface.
`io_descriptor_t *`	`descriptor`	Pointer to the device descriptor for the hardware device in device descriptor area provided by `YAMS`
`uint32_t`	`io_address`	Start address of the memory-mapped I/O-area of the device.
`uint32_t`	`type`	The typecode of this device. Typecodes are listed in `drivers/yams.h`

Table 9.3: Fields in structure `device_t`

Type	Name	Explanation
`uint32_t`	`type`	Typecode of the device.
`uint32_t`	`io_area_base`	Start address of the device's memory mapped I/O area.
`uint32_t`	`io_area_len`	Lenght of the device's memory mapped I/O area in bytes.
`uint32_t`	`irq`	The interrupt request line used by this device. `0xffffffff` if the device doesn't use interrupts.
`char`	`vendor_string`	Vendor string of the device. Note that the string is not 0-terminated.
`uint32_t[2]`	`resv`	Reserved for future extensions.

Table 9.4: Fields in `YAMS` device descriptor structure `io_descriptor_t`.

Type	Name	Explanation
device_t *	device	Pointer to the "real" device.
int (*)(gcd_t * gcd, const void * buf, int len)	write	Pointer to a function which writes `len` bytes from `buf` to the device. The function returns the number of bytes successfully written.
int (*)(gcd_t * gcd, void * buf, int len)	read	Pointer to a function which reads at most `len` bytes to `buf` from the device. The function returns the number of bytes successfully read.

Table 9.5: Generic Character Device (`gcd_t`)

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. Else 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[1]:

```
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`.

9.2.3 Generic Character Device

A 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 e.g. a TTY or some other character device.

The generic character device is implemented as a structure with the fields described in the Table 9.5.

9.2.4 Generic Block Device

Generic block device (GBD) is an abstraction of a block-oriented device (e.g. disk). GBD consists of function interface and a *request* data structure that abstracts the blocks to be handled. All functions are implemented by the actual device driver. Function interface is provided as the `gbd_t` (see Table 9.6) data structure.

Blocks to be handled are abstracted by the `gbd_request_t` data structure (Table 9.7). Structure includes all necessary information related to the reading or writing of a block.

The `gbd_operation_t` is an enumeration of following values: `GBD_OPERATION_READ` and `GBD_OPERATION_WRITE`.

[1] If you are familiar with Unix device driver interface, it may help to think of the `typecode` as major device number and `n` as minor device number.

Type	Name	Explanation
`device_t *`	`device`	Pointer to the actual device.
`int (*)(gbd_t * gbd, gbd_request_t *request`	`read_block`	A pointer to a function which reads a `request->block` from the device `gbd` to the buffer `request->buf`. Before calling, fill the fields `block`, `buf` and `sem` in `request`. The call of this function is synchronous if `sem` is NULL. The call of this function is asynchronous otherwise. When the asynchronous read is done the semaphore `sem` is signaled. In synchronous mode the return value 1 indicates success and 0 failure. In asynchronous mode 1 is returned when the work is submitted to the lower layer, 0 indicates failure in submission.
`int (*)(gbd_t *gbd, gbd_request_t *request`	`write_block`	A pointer to a function which writes a `request->block` to the device `gbd` from the buffer `request->buf`. Before calling, fill the fields `block`, `buf` and `sem` in `request`. The call of this function is synchronous if `sem` is NULL. The call of this function is asynchronous otherwise. When the asynchronous write is done the semaphore `sem` is signaled. In synchronous mode the return value 1 indicates success and 0 failure. In asynchronous mode 1 is returned when the work is submitted to the lower layer, 0 indicates failure in submission.
`uint32_t (*)(gbd_t * gbd)`	`block_size`	Returns the block size of the device in bytes.
`uint32_t (*)(gbd_t * gbd)`	`total_blocks`	Returns the total number of blocks on the device.

Table 9.6: Fields in the structure `gbd_t`.

Type	Name	Explanation
`gbd_operation_t`	`operation`	Read or write. Set when write or read is called, preset values are ignored.
`uint32_t`	`block`	Block number to operate on.
`uint32_t`	`buf`	Non mapped address (physical memory address) to a buffer of size equal to blocksize of the device. Address must be a physical memory address, because physical devices will handle only those.
`sem_t *`	`sem`	Semaphore which will be incremented when the request is done. Can be `NULL`. If `NULL`, the request will be handled synchronously (will block).
`void *`	`internal`	Driver internal information, ignored when using this structure.
`gbd_request_t *`	`next`	Pointer to the next request in the chain. Ignore when using, driver will use this in the I/O-scheduler.
`int`	`return_value`	Return status of this request. Set when request is handled. This is 0 if the request was successful.

Table 9.7: Fields in the structure `gbd_request_t`.

In case of asynchronous calls *gbd*-interface functions will return immediately and waiting is left for the caller. This means creating a semaphore before submitting the request and the waiting it to be released. Memory reserved for the request may not be released until the *request* is really served by the interrupt handler (ie. semaphore is released). The thread using a GBD device must be very careful especially with reserving memory from function stacks (ie. static allocation). If function is exited before the *request* is served, memory area of the request may corrupt.

In case of synchronous calls *gbd*-inerface functions will block until the request is handled. The memory of the *request* data structure may be released when returned from *gbd*-interface functions.

9.2.5 Generic Network Device

A generic network device (GND) is an abstraction of any network device. The GND interface defines functions for receiving and sending data as well as finding the maximum transfer unit (MTU) or the network address of the interface. GND is a generic interface which allows the code that uses the network device to be unaware of the actual implementation of the network device driver. The GND structure is described in Table 9.8.

`drivers/device.h`, `drivers/device.c`	Device driver interface
`drivers/drivers.h`, `drivers/drivers.c`	List of available device drivers
`drivers/yams.h`	Constants derived from the `YAMS` hardware
`drivers/gcd.h`	Generic character device
`drivers/gbd.h`	Generic block device
`drivers/gnd.h`	Generic network device

9.3 Drivers

9.3.1 Polling TTY driver

Two separate drivers are provided for TTY. The first one is implemented by polling and the other with interrupt handlers. The polling driver is needed in boot up sequence when interrupts are disabled. It is also useful in kernel panic situations, because interrupt handlers might not be relied on in error cases.

`void polltty_init (void)`

- Initializes the polling TTY driver. Finds the first console device in YAMS and attaches to that. Other `polltty`-functions must not be called before `polltty_init()` has been called.

`int polltty_getchar (void)`

- Gets one character from TTY device. Blocks (busyloop) until a character has been successfully read. Returns 0 on error (no TTY device).
- Returns the character read.
- Note that the polling TTY driver is unreliable on reads: characters may be lost if input buffer overflows in the hardware (buffer is 1 character in size).

Type	Name	Explanation
`device_t *`	`device`	Pointer to the real device.
`int (*)(struct gnd_struct *gnd, void *frame, network_address_t addr)`	`send`	Pointer to a function which sends one network frame to the given address. The network frame must be in the format defined by the media. (For `YAMS` this means that the first 8 octets are filled by the network layer and the rest is data.) The call of this function blocks until the frame is sent. Note that the pointer to the frame is a physical address, not a segmented one and the frame must have the size returned by the `frame_size` function. The return value 0 means success. Other values indicate failure.
`int (*)(struct gnd_struct *gnd, void *frame)`	`recv`	Pointer to a function which receives one network frame. The network frame returned will be in the format defined by the media. (For `YAMS` this means that the first 8 octets specify the source and destination addresses and the rest is data.) Note that the pointer to the frame is a physical address, not a segmented one and the frame must have the size returned by the `frame_size` function. The call of this function will block until a frame is received. Otherwise the call will return error when no frame is available. The return value 0 means success. Other values indicate failure.
`uint32_t (*)(struct gnd_struct *gnd)`	`frame_size`	Pointer to a function which returns the frame size of the media in octets.
`network_address_t (*)(struct gnd_struct *gnd)`	`hwaddr`	Pointer to a function which returns the network address (MAC) of this interface.

Table 9.8: Fields in the structure `gnd_t`.

`void polltty_putchar (char c)`

- Writes character `c` to TTY. If TTY is not initialized or found, ignores the write.

`drivers/polltty.c`, `drivers/polltty.h`	Polling TTY driver implementation
`lib/libc.c`, `lib/libc.h`	`kwrite()` and `kread()`

9.3.2 Interrupt driven TTY driver

The interrupt driven or 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/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 sets `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 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 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.

`drivers/tty.c`	The interrupt driven TTY implementation
`drivers/tty.h`	

9.3.3 Network driver

YAMS includes a simulated network interface card (NIC). The driver for this device is not included in BUENOS because it was left as an exercise for the students. The YAMS NIC is very similar to the other YAMS DMA devices. The network card has a memory mapped I/O-area which has ports for reading data and a command port for giving commands. The YAMS NIC will signal completion of tasks by raising interrupts. See the YAMS manual for further details.

When implementing the network driver you need to provide implementations for the interface functions specified by the general network device, which are explained in section 9.2.5. In addition to this at least an initialization function and an interrupt handler is needed. See also the device driver implementor's checklist in section 9.2.1.

9.3.4 Disk driver

The disk driver implements the Generic Block Device (GBD) interface (see section 9.2.4). The driver is interrupt driven and provides both synchronous (blocking) and asynchronous (non-blocking) operating modes for request. The driver has three main parts:

- Initialization function, which is called in startup when a disk is found.
- 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 (see Table 9.9). The internal data also contains a pointer to the currently served disk request.

Note how the fields modified by both top- and bottom-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 obviously be a flawed approach because of interrupts).

The implementation contains the following functions:

`device_t disk_init (io_descriptor_t *desc)`

- Initializes the disk driver for the disk pointed by `desc`.
- Implementation:

  1. Allocate memory for device record (`device_t`), generic block device record (`gbd_t`) and internal data (`disk_real_device_t`, see Table 9.9).
  2. Initialize the device record entries.
  3. Set GBD function pointers to point to disk's implementation.
  4. Initialize internal data, including the spinlock used for synchronization for this device.
  5. Register the interrupt handler (`disk_interrupt_handle`).

Type	Name	Explanation
`spinlock_t`	`slock`	Spinlock which must be held when operating the device (disk), or manipulating driver's internal data structures.
`volatile gbd_request_t *`	`request_queue`	The head of a linked list containing all the pending requests for this disk.
`volatile gbd_request_t *`	`request_served`	Pointer to the request which the disk is currently processing (request sent to the hardware and waiting for its interrupt). The same request is never in this variable and in request queue at the same time.

Table 9.9: Fields in disk driver's internal data structure (`disk_real_device_t`)

`static void disk_interrupt_handle (device_t *device)`

- Handle an interrupt on an interrupt line for which this handler for device driver has been registered.
- Note that this function may be called at any time, even on all CPUs at once and even for nothing (in case of shared IRQs).
- The handler will check whether a request has ended and if so, start a new request if one is available. New requests are taken from the beginning of the request queue.
- Implementation:

  1. Acquire the device spinlock (interrupts are disabled by default).
  2. Check whether our disk has pending interrupts. If not, release the spinlock and return. (This interrupt was actually for some other device on the same IRQ line).
  3. Reset pending IRQs on the device.
  4. Assert that we have a reference to the served request in device's internal data (`request_served`). This is the request that should now be complete, because the device generated an IRQ.
  5. Set return value to 0 (Success) in the served request.
  6. Call `semaphore_V` for served request's semaphore, so that the waiter (caller or internal routine) will know that the request is ready.
  7. Call `disk_next_request`. That function will start new request on the disk if one is available in the queue of pending requests.
  8. Release the device spinlock.

`static void disk_next_request (gbd_t *gbd)`

- Start new operation on an idle disk device if queued requests are available.
- This function assumes that the device spinlock is already held and that interrupts are disabled.
- Implementation:
  1. Assert that the disk is not busy.
  2. Assert that no request is marked as the currently served request.
  3. If there are no requests in the queue, return.
  4. Remove the first request from the queue of pending requests and set it as the served request.
  5. Write the sector value to the disk's sector-port.
  6. Write the address of the request's buffer to the disk's address-port (note that this must be a physical address, not a segmented address).
  7. Write the read or write command to disk's command-port.

`static int disk_read_block (gbd_t *gbd, gbd_request_t *request)`

- Takes in a new read request. This function implements the read-interface on Generic Block Device (GBD).
- Returns 1 on success, 0 otherwise.
- Implementation:
  1. Mark the request as read-request.
  2. Submit the request to the driver with `disk_submit_request`.

`static int disk_write_block (gbd_t *gbd, gbd_request_t *request)`

- Takes in a new write request. This function implements the write-interface on Generic Block Device.
- Returns 1 on success, 0 otherwise.
- Implementation:
  1. Mark the request as write-request.
  2. Submit the request to the driver with `disk_submit_request`.

`static int disk_submit_request (gbd_t *gbd, gbd_request_t *request)`

- Submits a new request into the disk's request queue. If the disk is currently idle, puts the request to the disk device.
- Implementation:
  1. Check whether the semaphore in the `request` is `NULL`. If it is, set `sem_null` to true, else set it to false.
  2. If `sem_null` = true, create new semaphore and set it as the semaphore for this request.
  3. Disable interrupts.

4. Acquire the device spinlock.
5. Call `disksched_schedule` to place the new request in the queue of pending requests.
6. If the disk is idle (no served request), call `disk_next_request` to start a new request on the device.
7. Release the device spinlock.
8. Restore the interrupt status.
9. If `sem_null` = true (we created the semaphore for this request) call `semaphore_P` on the created semaphore. Thus if this was a blocking call, wait until the request is complete. After the semaphore lowering returns, destroy the semaphore and set it back to `NULL` in the request structure.
10. Return with success (1) or error (0).

`static uint32_t disk_block_size (gbd_t *gbd)`

- Returns the blocksize of the disk in bytes. Implements the getblocksize-interface in the Generic Block Device (GBD).
- Implementation:

1. Disable interrupts.
2. Acquire the device spinlock.
3. Write the blocksize request command into the disk's command port.
4. Read the blocksize from the disk's data-port.
5. Release the device spinlock.
6. Restore the interrupt status.
7. Return the blocksize in bytes.

`static uint32_t disk_total_blocks (gbd_t *gbd)`

- Returns the total number of blocks on this device.
- Implementation:

1. Disable interrupts.
2. Acquire the device spinlock.
3. Write the block number request command to the disk's command-port.
4. Read the number of blocks from the disk's data-port.
5. Release the device spinlock.
6. Restore the interrupt status.
7. Return the total number of blocks.

Disk Scheduler

```
void disksched_schedule (volatile gbd_request_t **queue,
     gbd_request_t *request)
```

- Adds given `request` to `queue`. The placement location depends on the disk scheduling policy. The current policy is strict FIFO (first in, first out). Thus we always add new requests to the end of request queue.
- The first argument is marked `volatile`, because the function is often called from places where queues are `volatile` and thus extra casting is avoided at the calling side.
- Implementation:
  1. Add the request to the end of linked list `queue`.

`drivers/disk.h`, `drivers/disk.c`	Disk driver
`drivers/disksched.h`, `drivers/disksched.c`	Disk scheduler
`drivers/gbd.h`	Generic Block Device

9.3.5 Timer driver

Timer driver allows to set timer interrupts (harware interrupt 5) at certain intervals. C-function `timer_set_ticks()` works as a front-end for the assembler function `_timer_set_ticks`. C-function takes number 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*. *Count* register contains the current cycle count and *Compare* register the cycle number that the timer interrupt happens. The assembler function simply adds the number of cycles to the current cycle count and writes it to the *Compare* register.

```
void timer_set_ticks (uint32_t ticks)
```

- Passes the argument to the assembler function that sets a timer interrupt to happen after `ticks` clock cycles.

```
_timer_set_ticks (A0)
```

- Sets a timer interrupt to happen by adding the contents of a0 to the current value of *Count* register and writing it to the *Compare* register.

`drivers/timer.c`, `drivers/timer.h`, `drivers/_timer.S`	Timer driver implementation

9.3.6 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.

Meminfo

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

The meminfo device driver is a wrapper to the meminfo device I/O ports, and consists of the following functions:

```
device_t * meminfo_init (io_descriptor_t *desc)
```

- Initializes the system meminfo device.

```
uint32_t meminfo_get_pages (void)
```

- Get the number of physical memory pages (4096 bytes/page) in the machine from the system meminfo device. Reads the *PAGES* port of the `YAMS` meminfo device.

RTC

The Real Time Clock device provides simulated real time data, such as system uptime and clock speed.

The RTC device driver is a wrapper to the RTC device I/O ports, and consists of the following functions:

```
device_t * rtc_init (io_descriptor_t *desc)
```

- Initializes the system RTC device.

```
uint32_t rtc_get_msec (void)
```

- Get the number of milliseconds elapsed since system startup from the system RTC. Reads the *MSEC* port of the `YAMS` RTC device.

```
uint32_t rtc_get_clockspeed (void)
```

- Get the machine (virtual/simulated) clock speed in Hz from the system RTC. Reads the *CLKSPD* port of the `YAMS` RTC device.

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.

The shutdown device driver consists of the following functions:

```
device_t * shutdown_init (io_descriptor_t *desc)
```

- Initializes the system shutdown device.

`void shutdown (uint32_t magic)`

- Shutdown the system with the given magic word. Writes the magic word to the *SHUTDN* port of the `YAMS` shutdown device.
- The magic word should be either *DEFAULT_SHUTDOWN_MAGIC* or *POWEROFF_SHUTDOWN_MAGIC*.
- *Can* be called even though the shutdown device is not initialized (kernel should always be able to panic).

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. The driver implements the following functions:

`device_t * cpustatus_init (io_descriptor_t *desc)`

- Initializes the CPU status device.

`int cpustatus_count ()`

- Returns the number of CPUs in the system.

`void cpustatus_generate_irq (device_t *dev)`

- Generates an interrupt on the CPU described by `dev`.

`void cpustatus_interrupt_handle (device_t *dev)`

- Clears the interrupt generated by `dev`.

`drivers/metadev.c`, `drivers/metadev.h`	Metadevice driver implementation

Exercises

9.1. Why does the TTY driver have small input and output buffers? What are they used for and what are the benefits and drawbacks (if any) of having these kinds of buffers?

9.2. Why doesn't `tty_write` write `*buf` by itself? Can you trace the control of the kernel during writing, say a five character buffer, to the terminal?

9.3. Interrupt handlers cannot print anything in `BUENOS`, because they cannot access the interrupt driven TTY driver by proper syncronization (why?). Can the polling TTY driver be used to print in an interrupt handler? Why or why not?

9.4. Implement a device driver for the network interface. The hardware is documented in `YAMS` manual. The driver is the low level (link layer) interface to the network card and it will be used to access the card when implementing a network protocol stack.

You might find it helpful to take a look at the disk device driver before designing your own driver.

The driver should implement the Generic Network Device interface (specified in `drivers/gnd.h`, see section 9.2.5), and in addition of course have an initialization function and an interrupt handler.

Hint: take a look at section 9.3.3.

Chapter 10

Booting and Initializing Hardware

This chapter explains the bootup process of the BUENOS system from the first instruction ever executed by the CPU to the point when userland processes can be started.

10.1 In the Beginning There was _boot.S ...

When YAMS is powered up, its program counter is set to value 0x80010000 for all processors. This is the address where the BUENOS binary image is also loaded. Code in _boot.S is the very first code that YAMS will execute. Because no initializations are done (ie. there is no stack), _boot.S is written in assembly.

The first thing that the _boot.S code will do is processor separation. The processor number is detected and all processors except number 0 will enter a wait loop waiting for the kernel initialization to be finished. Later, when the kernel initialization (in main.c) is finished, processor 0 will signal the other processors to continue.

So that further initialization code could be written in a high-level language, we need a stack. A temporary stack will be located at address 0x8000fffc, just below the starting point of the BUENOS binary image. The stack will grow downward. Setting up the base address of the stack is done after processor separation in _boot.S. Later, after initialization code, every kernel thread will have own stack areas.

After this we have a stack and high-level language routines may be used. On the next line of _boot.S, we'll jump to the high-level initialization code init() located in main.c.

10.2 Hardware and Kernel Initialization

The first thing the init() function does is set up the polling TTY driver (see section 9.3.1). The polling driver is needed in bootup, because interrupts cannot be enabled before hardware is properly set up and system interrupt handlers are initialized. Polling TTY is accessed through kwrite(), kread() and kprintf() functions.

Next, the kernel will set up the memory allocation system (kmalloc), which can be used during the boot process. Memory allocated at this stage is never released. After the memory allocation setup, the kernel reads boot arguments from YAMS (see

Appendix A) and seeds the random number generation system based on the boot arguments.

Further, the kernel will initialize interrupt handling system (interrupts still disabled, but handlers can now be installed), sets up the threading system and high level synchronization primitives (the sleep queue and semaphores).

The next step is to detect all supported hardware in the system, which is done by calling `device_init()`. After the call, drivers for all supported devices have been installed. After device drivers, the virtual filesystem is initialized.

Now we are in a state where we can initialize the virtual memory subsystem which also disables kernel memory allocation system.

A thread is created (note that the bootup doesn't run in any thread) and it will be started (since interrupts are disabled it doesn't actually run). This new thread will later run system startup sequence in function `init_startup_thread()`.

Finally, other CPUs in the system are released from waiting loop and interrupts are enabled. Explicit software interrupt is generated and the startup thread is forced to run.

10.3 System Start-up

Now we are running inside a real thread in function `init_startup_thread`. The system is actually already running, but now we do all the initializations that can be done on a running system.

First, all filesystems are mounted into the VFS. Then networking subsystem is also initialized.

Finally, if `initprog` boot argument was given to the kernel, we will start the specified userland program. This ends the kernel bootup sequence. If an initial program was not given, the init thread will fall back to function `init_startup_fallback()` which can be used to run test code.

`init/_boot.S`	Kernel entry point after boot
`init/main.c`	Kernel bootup code

Appendix A

Kernel Boot Arguments

YAMS virtual machine provides a way to pass boot arguments from the host operating system to the booted kernel. BUENOS supports these arguments. Typically arguments are given like this:

```
yams buenos randomseed=123 'initprog=[root]shell' debuginit
```

In the example above, we give three arguments to the kernel. Two of the arguments have values, one has only name. Note the quotation used to protect the second argument string from host shell. The arguments without a value are equal to arguments with a value of an empty string (not NULL).

Boot arguments can be accessed in BUENOS with the following function:

```
char * bootargs_get (char *key)
```

- Gets the boot argument specified by `key`.
- Returns the value of the `key`. Returns `NULL` if the argument was not given on kernel command line. Valueless parameters return a pointer to an empty string.

The DEBUG printing system uses boot arguments to decide whether the particular debug string should be printed or not. `main.c` contains example on DEBUG usage and uses `debuginit`-argument. The console test in `main.c` also uses boot argument (`testconsole`).

The following boot arguments have predefined meaning:

initprog Defines the process to start after the system has been booted. Example: "`initprog=[root]halt`".

randomseed Specifies the seed with which to initialize the (pseudo)random number generator. If this argument is not present, the random number generator is seeded with 0. Example: "`randomseed=123`" seeds the generator with 123. The random number generator is currently used only to introduce some variance to the length of the time slice. It can of course be used in any place where there is need for (pseudo)random numbers.

`drivers/bootargs.h`, `drivers/bootargs.c`	Boot argument handling
`lib/debug.h`, `lib/debug.c`	Debug printing

Appendix B

Kernel Configuration Settings

Many static constants defining limits of BUENOS kernel can be tuned by editing the kernel configuration file kernel/config.h. All configuration options are defined as C preprocessor macros starting with prefix CONFIG_.

Every parameter can be changed in the limits defined in the comment just above the corresponding configuration parameter. Many limits are arbitrary, but some values really have to be within the limits in order to get a working system.

The current implementation restricts the number of threads to 256 which is the maximum number of address space identifiers in MIPS32 CPU. The kernel stack size should not be increased much, since the space is statically allocated and multiplies by the number of possible running threads. The system can handle more than 32 CPUs, but YAMS will start to run out of device descriptors (it has 128) if more than this amount is defined.

Here is a list of current configuration parameters:

CONFIG_MAX_THREADS

- **Purpose:** Defines the size of the thread table and thus the maximum number of threads supported by the kernel
- **Value range:** from 2 (idle + init) to 256 (max. ASID)

CONFIG_THREAD_STACKSIZE

- **Purpose:** Sets the size of the private kernel stack area of each thread.
- **Value range:** from 2048 (must hold contexts) to any size, but settings over 4096 are not recommended.

CONFIG_MAX_CPUS

- **Purpose:** Sets the maximum number of CPUs supported by the kernel.
- **Value range:** 1 – 32

CONFIG_SCHEDULER_TIMESLICE

- **Purpose:** Defines the length of the scheduling interval (timeslice) in processor cycles.
- **Value range:** from 200 (can get out of context switch) to any higher value.

`CONFIG_BOOTARGS_MAX`

- **Purpose:** Sets the maximum number of boot arguments the kernel will accept.
- **Value range:** 1 – 1024

`CONFIG_MAX_SEMAPHORES`

- **Purpose:** Defines the total number of semaphores in the system.
- **Value range:** 16 – 1024

`CONFIG_MAX_DEVICES`

- **Purpose:** Defines the maximum number of hardware devices supported by the kernel.
- **Value range:** 16 – 128 (YAMS maximum)

`CONFIG_MAX_FILESYSTEMS`

- **Purpose:** Defines the maximum number of filesystems.
- **Value range:** 1 – 128

`CONFIG_MAX_OPEN_FILES`

- **Purpose:** Defines the maximum number of open files.
- **Value range:** 16 – 65536

`CONFIG_MAX_OPEN_SOCKETS`

- **Purpose:** Defines the maximum number of network sockets the kernel will support.
- **Value range:** 4 – 512

`CONFIG_POP_QUEUE_SIZE`

- **Purpose:** Defines the the size of receive queue of packet oriented prototol (POP).
- **Value range:** 4 – 512

`CONFIG_POP_QUEUE_MIN_AGE`

- **Purpose:** Defines the minumum time in milliseconds that POP packets stay in the input queue if nobody is interested in receiving them.
- **Value range:** 0 – 10000

`CONFIG_MAX_GNDS`

- **Purpose:** Defines the maximum number of network interfaces the kernel will support.
- **Value range:** 1 – 64

`CONFIG_USERLAND_STACK_SIZE`

- **Purpose:** Defines the number of stack pages the userland process has.
- **Value range:** 1 – 1000

`kernel/config.h`	Configurable kernel parameters

Appendix C

Example YAMS Configurations

C.1 Disk

A good example disk for filesystem implementation which do not cause too large store files to be created on the host operating system could be (note that if pointed here by an exercise, you must use this entry as it is):

```
Section "disk"
  vendor               "128k"
  irq                  3
  sector-size          128
  cylinders            256
  sectors              1024
  rotation-time        25            # milliseconds
  seek-time            200           # milliseconds, full seek
  filename             "store.file"
EndSection
```

Bibliography

[Andrews] Andrews, G. R., *Foundations of multithreaded, parallel and distributed programming*, ISBN 0-201-35752-6, Addison-Wesley Longman, 2000

[Patterson] Patterson, D. A., *Computer organization and design: the hardware/software interface*, ISBN 1-55860-491-X, Morgan Kaufmann Publishers, 1998

[Stallings] Stallings, W., *Operating Systems: Internals and Design Principles*, 4th edition, ISBN 0-13-032986-X, Prentice-Hall, 2001

[K&R] Kernighan B. W., Ritchie D. M., *The C Programming Language*, 2nd Edition, ISBN 0-13-110362-8, Prentice-Hall, 1988

[Tanenbaum] Tanenbaum, A. S., *Modern Operating Systems*, 2nd edition, ISBN 0-13-031358-0, Prentice-Hall, 2001

[Miller] Miller, Peter, *Recursive Make Considered Harmful*, http://www.tip.net.au/~Emillerp/rmch/recu-make-cons-harm.html

Index