CP1

ECE391: Computer Systems Engineering **Fall 2025**
Machine Problem 3

Checkpoint 1: Fri Oct 31 18:00 CDT
Checkpoint 2: Fri Nov 21 18:00 CDT
Checkpoint 3: Fri Dec 5 18:00 CDT

Illinix 391

Contents

1 Introduction **3**

2 Using the Group Repository **3**

3 The Pieces **4**

3.1 Getting started 4

3.2 Work Plan 4

3.3 Requirements 5

4 Testing **6**

5 What to Hand in **6**

5.1 Checkpoint 1: Filesystem and Drivers and Program Loading, (Oh My!) 6

5.1.1 Group Repository 6

5.1.2 VIRTIO Block Device 6

5.1.3 Read-Only Filesystem 7

5.1.4 Rambunctious RAM Disk 8

5.1.5 Locked and 8

5.1.6 . . . (ELF) Loaded 9

5.1.7 Cache Me If You Can 9

5.1.8 Existing Files 10

5.1.9 File Modifications 11

5.1.10 Running Your Code 11

5.1.11 Troubleshooting/Debugging 11
5.1.12 Checkpoint 1 Handin 11

6 Grading 12

7 Appendix A: The File System 13
7.1 File System Overview 13
7.2 File Abstractions 15
7.3 Building the File System 16
7.4 The Blob 17

8 Appendix B: Uniform I/O 18
8.1 Overview 18
8.2 Uniform I/O Operations 18
8.3 Opening Uniform I/O objects 18
8.4 Listing Objects 19
8.5 Uniform I/O Diagram 20

9 Appendix E: Troubleshooting 21
9.1 Debugging with gdb 21
9.1.1 Debugging the Kernel 21
9.1.2 Debugging User Programs 22
9.1.3 Useful gdb Commands 22

CP1

1 Introduction

Read the whole document before you begin, or you may miss points on some requirements.

In this machine problem, you collaborate to develop the core of an operating system roughly based on Unix Version 6, with modern concepts peppered in where appropriate. You'll implement interrupt logic, user threading (à la MP2), kernel and application paging, initialize some devices and a filesystem with the VIRTIO interface, and create a system call interface to support various system calls. The operating system will support running several tasks ("threads") spawned by a number of user programs; programs will interface with the kernel via system calls.

Don't worry, these aren't all "boring" programs—you'll get to run some cool games, too.

The goal for the assignment is to provide you with hands-on experience in developing the software used to interface between devices and applications, *i.e.*, operating systems. You should notice that the work here builds on concepts from the other machine problems. Many of the abstractions used here (*e.g.*, the VIRTIO interface) have been simplified to reduce the effort necessary to complete the project, but we hope that you will leave the class with the skills necessary to extend the implementation that you develop here along whatever direction you choose, by incrementally improving various aspects of your system.

2 Using the Group Repository

You should receive access to a shared repository with your group members. Your group has a two-digit group number; your group number should be released in a Piazza post, and your repository should be named something like `mp3_group_xx`, where `xx` is your number.

It is required to run the following commands on each computer that you clone the repository to, in order to make sure the line endings are set to LF (Unix style):

```
git config --global core.autocrlf input
git config --global core.eol lf
```

Another reminder, you will need to setup your git config with your name and email as stated in the group contract. Please review the contract form for instructions on how to do this. This is a requirement, failure to set this up **will result in a penalty**.

Some other tips:

As you work on MP3 with your teammates, you may find it useful to create additional branches to avoid conflicts while editing source files. Remember to `git pull` each time you sit down to work on the project and `git commit` and `git push` when you are done. Doing so will ensure that all members are working on the most current version of the sources. It is highly likely that you will benefit from proper usage of the `git stash` command, to correctly retain desired (local) changes when doing a pull.

When using Git, keep in mind that it is, in general, bad practice to commit broken sources. You should make sure that your sources compile correctly before committing them to the repository. Make sure not to commit compiled '*.o' files, besides what we have given you. You can modify your `.gitignore` in any way you want.

Finally, merge your changes into the `main` branch by each checkpoint deadline, as **this is the only branch we will use for grading**.

3 The Pieces

The basic OS is provided, but when you first get it, it won't properly do a lot of the things we expect an OS to do: perhaps most importantly [†], it can't play any cool games!

For simplicity, we will stick to text-mode graphics (for the most part), but your OS will, by the end, run the games from previous MPs as well as some new ones. We've included a few helpful pieces that will allow you to debug easier, such as `kprintf`. We **highly** encourage use of gdb to debug. Print statements can only take you so far. See **Appendix E** for details.

3.1 Getting started

In order to effectively work on this MP, you will need to develop knowledge of a lot of different and difficult concepts. The documentation on the course website and lectures will give you background information, but the best way to learn is to read and understand the code we have provided. This includes the `Makefiles`, `.ld` linker files, and `.c/.h` source files.

This MP is difficult, which is why we do not expect you to work alone. While you cannot share code or discuss details with other groups or anyone outside your group, you should work together with your team to get unstuck and build a common understanding.

3.2 Work Plan

"Work expands so as to fill the time available for its completion." - Cyril Parkinson

This project is somewhat daunting, and will require efforts from all your team members. You should partition the work accordingly to allow independent progress by all team members.

Setting up a clean testing interface will also help substantially with partitioning the work, since group members can't finish and test components before your groupmates finish the other parts (yet). The abstractions suggested should allow for some spots where a "working part" can be substituted with a functionally equivalent placeholder, so to speak—more on that later.

While splitting up the work allows for you to make more progress, it is still crucial that you spend time working together to integrate all parts. You should also be maintaining active communication between group members to make sure you all have an understanding of how all your code works. Even if you did not work on a specific section, we expect you to be familiar with how the code works and be able to explain what it and how it fits into your kernel.

Throughout the first part of this semester and in most/all of your previous classes, MPs were more structured. You were given a set of functions to write, and you only modified those functions. One of the goals of this

[†]Some may disagree that this is the most important thing, but who wants no games?

class is to make you into a more confident and thoughtful programmer by having you practice software design. We have deliberately left the implementation of certain sections open-ended. It is up to you and your group to find a way to meet the requirements - as with the real world, there is no "perfect" solution. The only requirement that we impose is that you follow the function interfaces that we have already specified - feel free create your own helper functions or creative implementations.

3.3 Requirements

In order to be compliant with our autograder, we will enforce the following requirements:

1. You may not add any new files to the repo other than the files from your MP2 that we specify
2. Any helper functions you write must be decared as `static`
3. You may not modify any provided function headers

Additionally, you should only modify the following files for CP1 ():

- `cache.c`
- `cache.h`
- `elf.c`
- `ktfs.c`
- `ktfs.h`
- `main.c`
- `dev/ramdisk.c`
- `dev/ramdisk.h`
- `dev/vioblk.c`

4 Testing

For this project, we strongly recommend that you write and run unit tests with adequate coverage of your source code.

As your operating system components are dependent on one another, you may find it useful to unit test each component individually to isolate any design or coding bugs.

You should create a `main_tests.c` file and add it to your `Makefile`. This file should create a kernel image (*e.g.*, `test.elf`) that you can load into QEMU and run tests with. As you add more components to your operating system, we encourage you to add corresponding tests that verify the functionality of each component at the interface level.

Keep in mind that passing all of your unit tests does not guarantee bug free code. However, the test suite provides a convenient means to run your tests frequently without having to re-debug older components as you add new functionality.

5 What to Hand in

5.1 Checkpoint 1: Filesystem and Drivers and Program Loading, (Oh My!)

The primary motivation of this checkpoint is to get 2 test programs, which we've given you, to run. `hello` will print some text to the UART screen, then stop. `trek` will run the same text-based Star Trek game that you know and love.

Rather than do this in a "hacky" way, we want to set up some key infrastructure now which will pay dividends as the project continues on.

Note: Please ensure that all the functions that we specify do not induce a kernel panic on an invalid input. You should return an error code (which one is up to you) if possible, otherwise do nothing.

For the checkpoint, you must have the following accomplished:

5.1.1 Group Repository

You must have your code in the shared group repository, and each group member should be able to demonstrate that they can read and change the source code in the repository.

5.1.2 VIRTIO Block Device

An operating system in general must communicate with external devices. One such device is obviously the real drive/disk (virtual, in this case) which contains programs and other files you want your operating system to have access to.

In order to set up this device (and any others down the line), we will need to set up the necessary framework for the VIRTIO block device. Your group must finish the implementation based on the VIRTIO documentation linked on the course website.

You may find it especially helpful to read sections 2.1-2.7, 3.1, 4.2, and 5.2 and recall your implementation of `viorng.c`. Skip anything related to ”legacy” interface.

More information about the functions that you have to write can be found on the Doxygen site linked on the course website.

1. `void vioblk_attach(volatile struct virtio_mmio_regs * regs, int irqno)`
2. `int vioblk_storage_open(struct storage * sto)`
3. `void vioblk_storage_close(struct storage * sto)`
4. `long vioblk_storage_fetch(struct storage * sto, unsigned long long pos, void * buf, unsigned long bytecnt)`
5. `long vioblk_storage_store(struct storage * sto, unsigned long long pos, const void * buf, unsigned long bytecnt)`
6. `vioblk_storage_cntl(struct storage * sto, int op, void * arg)`
7. `vioblk_isr(int irqno, void * aux)`

Also check the doxygen and the provided code for more information on the `storage` device struct. It is a little different than your previously implemented `serial` devices.

5.1.3 Read-Only Filesystem

Broadly speaking, your filesystem driver should provide a comfortable interface to open, read, and scan through files. In later checkpoints, you will add additional functionality.

Your `ktfs.c` file will need to interact with its backing device with some intermediate cache in `cache.c` to actually interact with the “physical” (well, virtual) device, so be sure that you understand what’s going on in files related to both the cache and the backing device.

More information about the functions that you have to write can be found on the Doxygen site linked on the course website.

These functions should be written in `ktfs.c`:

1. `int mount_ktfs(char * name, struct * cache)`
2. `int ktfs_open(struct filesystem * fs, const char * name, struct uio ** uioptr)`
3. `void ktfs_close(struct uio * uio)`
4. `long ktfs_fetch(struct uio* uio, void * buf, unsigned long len)`
5. `int ktfs_cntl(struct uio * uio, int cmd, void * arg)`
6. `void ktfs_flush(struct filesystem * fs)`

Note the use of a new `uio` struct here. This is how your kernel will interact with ”file-like” objects (such as files!). See **Appendix B** for more information on the `uio` struct.

In order to use the filesystem, we have provided a `mkfs_ktfs` function (see **Appendix A**) that generates a filesystem image for you. Be sure your implementation works for very large image sizes. This filesystem image is mounted by QEMU as a drive (using the `Makefile` we provide) and is accessible through VIRTIO.

For CP1, you will need to implement “fcntl”: `FCNTL_GETEND`, `FCNTL_GETPOS`, and `FCNTL_SETPOS` (Note: we are not requiring `FCNTL_SETEND` for this checkpoint since currently the FS is read only). Each opened file will need to keep track of its `pos`, and calling read with `bufsz` bytes will return `[pos, pos + bufsz - 1]` bytes as well as advance `pos` to `pos + bufsz`.

(Hint: implement the `ramdisk` device and its related functions to test your KTFS filesystem driver as well as the `cache` independently from `vioblk`.)

See **Appendix A** for additional details.

5.1.4 Rambunctious RAM Disk

You may wonder why we need a `ramdisk` device if we already have a `vioblk` device? The `ramdisk` device is a helper backing device that allows you to test your filesystem and ELF loader without using `vioblk`. Within your linker script `kernel.ld`, there is a section from `_kimg_blob_start` to `_kimg_blob_end` that you can use to place your whole filesystem image or an ELF file to load (see **Appendix A** about the blob).

More information about the functions that you have to write can be found on the Doxygen site linked on the course website.

For better unit testing and debugging, you must implement the `ramdisk` device “driver” in the `dev` directory. The following is a list of functions you need to implement:

1. `void ramdisk_attach()`
2. `int ramdisk_open(struct storage * sto)`
3. `void ramdisk_close(struct storage * sto)`
4. `int ramdisk_cntl(struct storage * sto, int cmd, void * arg)`
5. `long ramdisk_fetch(struct storage * sto, unsigned long long pos, void * buf, long bytecnt)`

If you would like to also test your ELF loader with the `ramdisk` device, you will need to interface it with a `uio`. To do this, see **Appendix B** for details on how `uio` and the filesystem are connected.

5.1.5 Locked and ...

Locks prevent concurrency issues when multiple threads are accessing the same resource. We have already provided locking objects and functions for you to utilize as needed. However, it is on you to determine where locks will be necessary for your OS in order to prevent concurrency issues.

Hint: In order to prevent deadlocks, an exiting thread must release all held locks.

More information about the functions we have given can be found on the Doxygen site linked on the course website.

5.1.6 . . . (ELF) Loaded

One of the key roles of the operating system is to be able to run other programs.

We want to be able to run many user-level programs, but for now you can focus on `hello` and `trek`. While we've given you the pre-compiled binary of `trek`, you'll have to compile `hello` using the `usr/Makefile`. Because of this, you also have access to `hello.c`. All the binaries are in a format called ELF (Executable and Linkable Format), which has a specific layout — it is the standard for Unix and Unix-like systems, historically, which means it is still very relevant. See the **Tools, References, and Links** page on the course website for the Linux manual page on ELF. Your loader will only need to deal with the program headers, not sections, so focus on that documentation.

Notice that since `elf_load` should support any compliant UIO interface, that we can in general load an ELF from "any source" as long as the proper functions are implemented in the given `uio_intf` (see **Appendix B**).

5.1.7 Cache Me If You Can

"Software gets slower faster than hardware gets faster." -Wirth's Law

This semester, you will implement a caching system to cache blocks from a backing interface. As you've learned in this course, communicating with devices is extremely slow relative to the CPU's clock cycle. Previously, you've implemented asynchronous communication (condition variables) so that while one thread is waiting on a device response, another thread can run.

While this significantly reduces the problem of "wasting" CPU cycles, it does not eliminate the problem of latency - the original thread still must wait a long time for the device's response. A cache is a commonly used way to reduce this latency.

Once you complete this checkpoint, your backing interface will be the VIRTIO block device, but we will refer to it generically as the backing interface in this document. Rather than reading/writing a block directly from/to the backing interface, we first check whether the block exists in the cache. If the block exists in the cache, we access the block via the cache rather than sending a new request to the backing interface. If the block does not exist in the cache, we read it from the backing device into the cache. Note that you may or may not need to evict a block currently in the cache in order to bring in this new block.

Your cache may have any level of associativity and may be write-back, write-through, or some other concoction. Please note that we are intentionally leaving the specific details of the cache vague; this is intended to be a design exercise.

Some additional scenarios/specifications for the cache are as follows (these would be good test cases for you to write):

- Initially (when the OS is first started), the cache is empty.

- The total number of blocks that the cache can hold must be 64.
- If the cache is initially empty and we read 64 contiguous blocks into it (e.g. 5, 6, 7, ..., 68), all 64 of those blocks must remain in the cache. This means that a read within any of those blocks should not generate a request to the backing device.
- If `cache_get_block()` is called and block *k* is read into the cache, block *k* should remain in the cache at least until `cache_get_block()` is called again (note that block *k* may remain in the cache for longer). This is to say, if we call `cache_get_block()` once and do not call it again, the block that was cached must remain in the cache.

More information about the functions that you have to write can be found on the Doxygen site linked on the course website.

You will implement the following 4 functions in `cache.c`:

1. `int create_cache(struct storage * sto, struct cache ** cptr)`
2. `int cache_get_block(struct cache * cache, unsigned long long pos, void ** pptr)`
3. `void cache_release_block(struct cache * cache, void * pblk, int dirty)`
4. `int cache_flush(struct cache * cache)`

As a reminder, you can also create any other helper functions that you need for your cache implementation.

5.1.8 Existing Files

During MP2, you worked with the PLIC, UART, and some other devices. You will be re-using that code for this checkpoint. You should add the following files into `sys` from MP2. They must be fully functional and (besides `thread.c`) should have the same functionality as a completed MP2. You can collaborate with your MP3 groupmates to choose whose MP2 code to use.

- `sys/dev/rtc.c/.h`
- `sys/dev/uart.c/.h`
- `sys/dev/viorng.c`
- `sys/plic.c/.h`
- `sys/timer.c/.h`
- `sys/thread.c/.h`
- `sys/thrasm.s`

5.1.9 File Modifications

For this MP there are a few small changes to make to the given code:

- Replace all inclusions of `assert.h` to `misc.h`
- Add the line `#include see.h` to `thread.c`
- Add the line `#include <stddef.h>` to `thread.c`

5.1.10 Running Your Code

Finally, once you have all of your code completed, you will need to finish `sys/main.c` to run your program. We've left some comments on how to run `trek`, you may also find it helpful to refer to your MP2 CP3 `main` function implementation. You can also run `hello` or another user program that you create.

5.1.11 Troubleshooting/Debugging

See **Appendix E** for more information about debugging and common issues.

5.1.12 Checkpoint 1 Handin

For handin, your work must be completed and pushed to the `main` branch of your team's GitHub remote repository by the deadline.

For this checkpoint you must complete the following:

- Read-write operations on `vioblk`
- Read-only filesystem
- Read operations on `ramdisk`
- Read-write operations on the cache
- ELF Load

6 Grading

Your final MP score is a baseline score that will be adjusted for teamwork and for individual effort. Note that the "correct" behavior of certain routines includes interfaces that allow one to prevent buffer overflows (that is, the interfaces do not leave the size of a buffer as an unknown) and other such behavior. While the TAs will probably not have time to sift through all of your code in detail, they will read parts of it and look at your overall design. The rough breakdown of how points will be distributed will be periodically released on the website prior to checkpoint deadlines.

A Note on Teamwork

Teamwork is an important part of this class and will be used in the grading of this MP. We expect that you will work to operate effectively as a team, leveraging each member's strengths and making sure that everyone understands how the system operates to the extent that they can explain it. In the final demo, for example, we will ask each of the team members questions and expect that they will be able to answer at a reasonable level **without** referring to another team member. **Failure to operate as a team will significantly reduce your overall grade for this MP.**

At the end of the MP there will be a teammate evaluation which will factor into your final MP grade.

You are also expected to adhere to the MP3 contract.

CP1

7 Appendix A: The File System

7.1 File System Overview

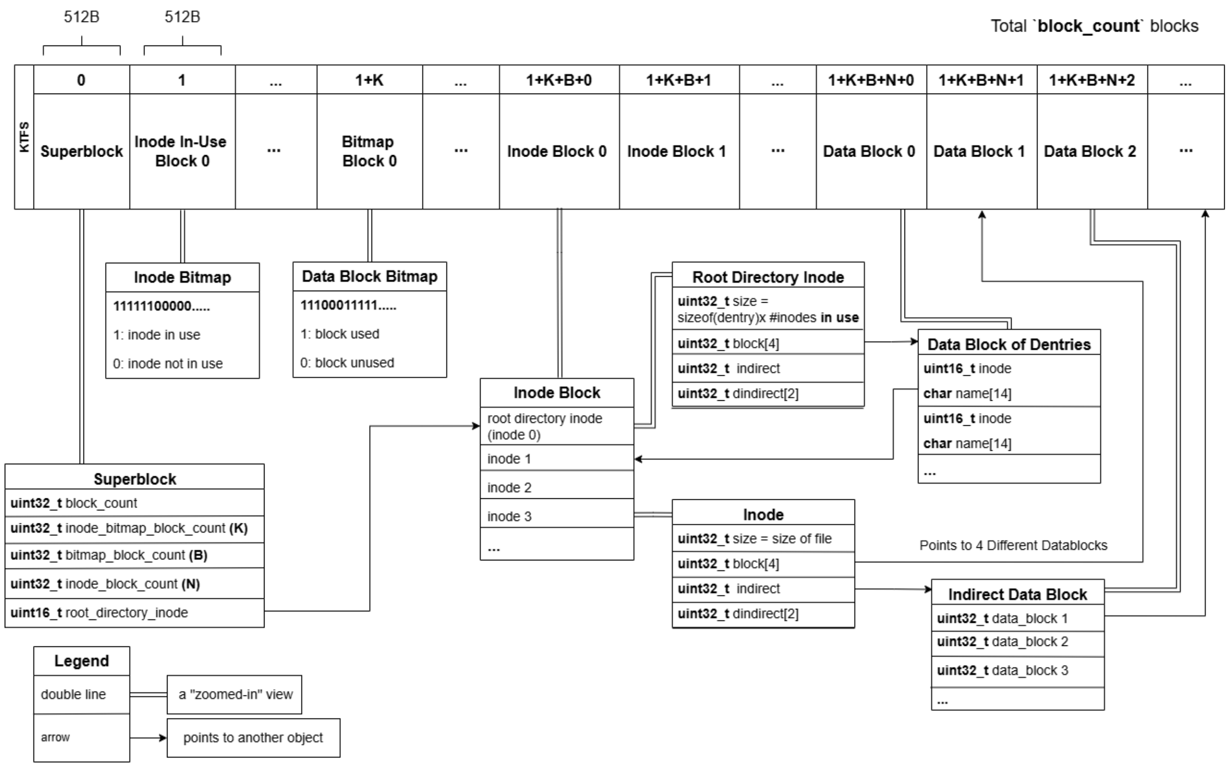
A filesystem can be considered an abstraction that allows for easy access of one or more ”files” within a large chunk of data. For this MP, you will be implementing KTFS - a custom filesystem inspired by ext2.

First, we will provide definitions of important terms.

The *filesystem image* refers to a ”blob” of data that contains everything needed to mount the filesystem and all the data associated with it.

The *backing device* for a filesystem refers to where the filesystem image is stored. In an actual computer, it’s typically stored on the hard drive. However, for this MP, the backing device will be the VIRTIO block device, which emulates a real hard drive. To test your filesystem without the VIRTIO block device, you can use a buffer in memory (see `ramdisk`).

The figure below shows the structure and contents of the filesystem image.



Layout of KTFS

KTFS is a block-based filesystem. We use a block size of 512B, which coincidentally matches with the block size of the VIRTIO device. There are 5 types of blocks:

1. Superblock (stores basic statistics of the file system image)
2. Inode Bitmap blocks (keeps track of which inodes are in use)
3. Bitmap blocks (keeps track of which blocks are in-use)
4. Inode blocks (stores index nodes (inodes), which contain info about each file)
5. Data blocks (stores actual data)

The first block is called the **superblock**. The superblock has five fields:

- `uint32_t block_count` - the total number of block in the file system
- `uint32_t inode_bitmap_block_count` - the total number of inode bitmap blocks
- `uint32_t bitmap_block_count` - the number of bitmap blocks
- `uint32_t inode_block_count` - the number of inode blocks
- `uint16_t root_directory_inode` - the index of the root directory inode (see below)

These fields only take up the first 18B of the 512B, so the remaining bytes are unused. The fields in the superblock are set when creating the filesystem image (see §**Building the File System**) and should not be modified by your OS.

The **inode bitmap blocks** contain ”bits”, as the name suggests. Each bit represents whether an inode entry is in-use. This is different from an inode block being in-use. For example, the 0th bit in the 0th inode bitmap block would correspond to the first 16B inode entry in Inode Block 0. This bitmap will need to be updated when removing or adding files (CP2).

The **bitmap blocks** make up another bitmap, where each bit indicates whether a block is used: 1 means used, 0 means free. For example, the 0th bit of the 0th bitmap block corresponds to the 0th block of the filesystem (i.e. the superblock). If a block is free, then it can be used later when writing to a file or creating a new file. When the filesystem image is created, the superblock, bitmap blocks, and inode blocks are all marked as in-use. During filesytem image creation, the ”disk size” is set and fixed, which means the number of bitmap blocks is also fixed.

The **inode blocks** contain the index nodes (inodes) of the files in the filesystem. Each file is described by an inode that specifies the file’s size in bytes and the numbers of the data block (e.g. data blocks 1,5,10 ...) that make up the file. During filesystem image creation, the number of inode blocks is set and cannot change.

Each inode only provide direct access to 4 data blocks through an array within the inode. However, inodes additionally store 1 indirect data block number and 2 doubly-indirect data block numbers. An indirect data block number points to a data block that contains an array of data block numbers, instead of actual data. For example, if you want to access the 4th data block of a file, you will have to get the indirect data block number first (*e.g.*`inode.indirect = 11`), get data block 11 from backing device, and read the first `uint32_t` (first 4B) of data block 11 (*e.g.*14). Then, go to data block 14 and there will be actual data. The idea is the same for doubly-indirect blocks, except there’s one more layer of indirection: the doubly-indirect data block number points to a data block that stores an array of indirect data block numbers, which further points to indirect data blocks that store an array of ”real” data block numbers.

Each indirect/doubly-indirect data block contains 512B (just like any other data block), but only the part of the indirect/doubly-indirect data block that points to data blocks necessary to contain the specified size need be valid, so be careful not to read and make use of block numbers that lie beyond those necessary to contain the file data. The data blocks that make up a file are not necessarily contiguous or in any specific order. You must use the data block numbers in the order specified in the inode and the indirect/doubly-indirect data blocks to access the correct data in the correct order.

There's one special inode called the root directory inode. Instead of containing the actual data of a file in its specified data blocks, these data blocks contain the directory entries. KTFS does not support nested directories, so the root directory is the only directory that you are required to support. Each directory dentry (or dentries for short) contains an inode number and a string that contains the file name. The size of each dentry is 16B: 2B for the inode number and 14B for the file name. Note that because we need 1B for the null terminator, the maximum length of a file name is 13B (13 characters). The "file size" specified in the root directory inode is equal to (the number of dentries) $\times$ (the size of each dentry). Therefore, you can determine the number of files in the filesystem by checking the size field of the root directory inode. When you create or delete a file, you must add/remove the dentry for that file and update the size appropriately. For this file system, there is a 1:1:1 correspondence between files, dentries, and inodes. No two dentries should contain the same inode number or file name.

Note: Dentries **must** be contiguous to be able to use the file size properly. This is a strict requirement that filesystem images will follow and your filesystem driver must maintain. Keep in mind that "contiguous" here does not actually mean contiguous in memory, just that if you were reading the root directory inode like a file, the first `size` bytes would be the dentries.

In most cases, the **data blocks** can be seen as normal chunks of data in a file. That means when you open a file, you should be reading data from and writing data to these actual data blocks. However, because of the root directory inode and indirect data blocks, data blocks can contain different things. So, a data block can be classified into the following types depending on what data it contains:

1. "Real" data blocks (contains actual data of a file)
2. Directory data blocks (pointed to by the root directory inode, contains directory entries)
3. Indirect data blocks (contains an array of data block numbers)
4. Doubly-indirect data blocks (contain an array of indirect data block numbers)

Note: Since the root directory inode is treated similar to a "regular" inode, it can grow in size like other files with indirect and doubly-indirect data blocks. This means that the limiting factor on the number of files in a filesystem is typically the number of inodes.

7.2 File Abstractions

In order to make your filesystem driver, you should create a `struct ktfs_file`. This will be the internal representation of a file that your filesystem driver uses to keep track of the state of each file. This structure should contain *at least* the following fields (add more if you find it useful):

1. The `uio` struct associated with each file. The `intf` field in this structure should be a pointer to an `uio_intf` struct that contains the correct `close, read, write, cntl` function pointers to interface with the file.
2. A **file size** member that stores the length of the file in bytes. This should be updated whenever write is called, if the length of the file has changed.
3. The **directory entry** struct for this file. As a reminder, this contains the inode number for the file and the file name.
4. A **pos** argument that stores the current position in bytes for the file. This will need to be used for calls to `ktfs_cntl`, when `cmd` is `FNCTL_GETPOS` or `FNCTL_SETPOS`.

You need to have some way to keep track of the `ktfs_file` structs that correspond to currently opened files (array, linked list, etc.), so that future UIO operations can be performed on them.

7.3 Building the File System

The filesystem image needs to be made in order for QEMU to read it and make it accessible to you (via VIRTIO). To do this, we have provided you with 2 binary executables within the `util/fs` directory.

- `mkfs_ktfs` is a customized version of Linux `mkfs` that generates a filesystem image according to the KTFS spec detailed above. This is how we will generate filesystem images to test and grade your filesystem driver.
- `mkfs_ktfs_inorder` is a modified version of `mkfs_ktfs` that does not randomize the order of data blocks associated with each file. **This is provided for debugging purposes only** and will not be used to grade your filesystem driver. You must be able to interface with files that use non-contiguous data blocks.

We have also provided you with a 3rd binary executable, `unmkfs_ktfs`. This program "unwraps" the files contained with a KTFS filesystem image to allow you to parse them on your own. You may find this useful when implementing the write/writeat functions in your filesystem driver, since these functions will modify your actual filesystem image and persist between QEMU shutdowns.

`mkfs_ktfs` usage is as follows:

```
./mkfs_ktfs [filesystem_image]
            [disk_size in K, M, or G]
            [inode_count]
            [file1] [file2] ...
```

In order for your filesystem image to work with our existing `Makefile`, you should name the image `ktfs.raw` and put it in the `sys` directory.

An example:

```
./mkfs_ktfs ../../sys/ktfs.raw 8M 16 ../../usr/bin/hello ../../usr/bin/trek
```

This would create a `ktfs.raw` file in the `sys` directory with a total size of 8MiB and 16 inodes (*i.e.*, 1 inode block). `hello` and `trek` would be the only 2 files in the filesystem image, with the rest of the inodes free.

Since we cannot have fractional data blocks, your disk size should always be a multiple of 512 (KTFS block size). The number of inode blocks is also rounded up, since we cannot have fractional inode blocks. When interfacing with the filesystem, you can consider any block marked as an "inode block" as able to be used to store inodes. In other words, the total number of possible inodes in your file system will always be a multiple of 16. However, bitmap blocks may be "partially" used. Be careful to not read/write past the size of the filesystem image.

Note: Make sure you are running your `mkfs_ktfs` binary and not the Linux `mkfs` function.

The files in your filesystem image are also given as arguments to `mkfs_ktfs`. You can (and should) provide multiple files in order to create a full filesystem image. All the metadata related to these files will be prepared for you, including the superblock and other structures that you've read about in **Appendix A** already. All of these blocks will be put in order according to the spec above and stored in your filesystem image (*i.e.*, `ktfs.raw`). This means that the filesystem image is already compliant with the KTFS spec.

To load a user program into the filesystem image, you need to compile it to an executable binary. This can be done by using the `Makefile` in `usr`. If you want to change a file in the filesystem or add/remove files (without using your filesystem driver), you **must** remake the filesystem image each time.

7.4 The Blob

The blob will be created as an object file and is not part of the regular filesystem. It is a tool for debugging and testing. If `blob.raw` exists in your `sys` directory, the `Makefile` will input the file contents of `blob.raw` to a read-only data section starting from `_kimg_blob_start` to `_kimg_blob_end`. These addresses can be obtained via `extern`. If you want to use the blob, simply rename the relevant file to `blob.raw` in the `sys` directory and add the object file when you build your kernel image ELF file as well as `extern` both addresses in your code to access the blob.

8 Appendix B: Uniform I/O

8.1 Overview

Due to the wide variety of different types of objects in our kernel, it can be very helpful to unify these under a common interface. You have already begun to see this with `serial` devices, which unifies many different types of devices into a single type of struct. Now, we will go even further, unifying all “file” like objects under a single Uniform I/O interface. We define file-like objects to be any object that can be read from and/or written to, and can be “closed”. This primarily applies to devices and files, but will go on to include listings, pipes, and generally any other form I/O.

8.2 Uniform I/O Operations

The kernel will interact with Uniform I/O using the struct `uio`. Each struct `uio` contains a pointer to a struct `uio_intf`, which in turn contains pointers to `close`, `read`, `write`, and `cntl`, functions specific to each endpoint. Note that not all these functions need to be implemented - if the device driver does not implement the function, the pointer can be NULL. Under the hood, these callback functions are endpoint specific and performs the relevant operation.

The `cntl` function allows control over endpoint specific operations that don't map nicely into `read` and `write`. For example, objects with an internal position, such as storage devices and files, need some way for users to determine where to read from and write to. Thus these endpoint's `cntl` functions accept `getpos` and `setpos` operations, defined as `FCNTL_GETPOS` and `FCNTL_SETPOS` in `uio.h`. Other enpoint specific operations are defined there as well.

8.3 Opening Uniform I/O objects

For object with similar endpoints, we group them together in the kernel via “mountpoints”. In our kernel, we will have the `dev` mountpoint (devfs), which will contain all devices, and the `c` mountpoint (ktfs), which will contain all files. We have a higher level wrapper around these mountpoints within `filesys.c`, which is responsible for mapping mountpoint names to their respective filesystems. Therefore, to open any file or device, you simply call `open_file` with the filename and its respective mountpoint name. When we open a file like this, we first iterate through our list of mountpoints to find the matching name, then call the open function attached to that filesystem. The filesystem itself is then responsible for finding the filename, opening it, and returning the uioptr with that file as its endpoint.

Note that for devices, the instance number is part of the filename. So, to open a UART device with an instance number of 1, you would call `open_file` on mountpoint name `dev` and file name `uart1`. You will have to do this in CP1 in order to pass in the UART device to trek, which will take in a `uio` object as an argument.

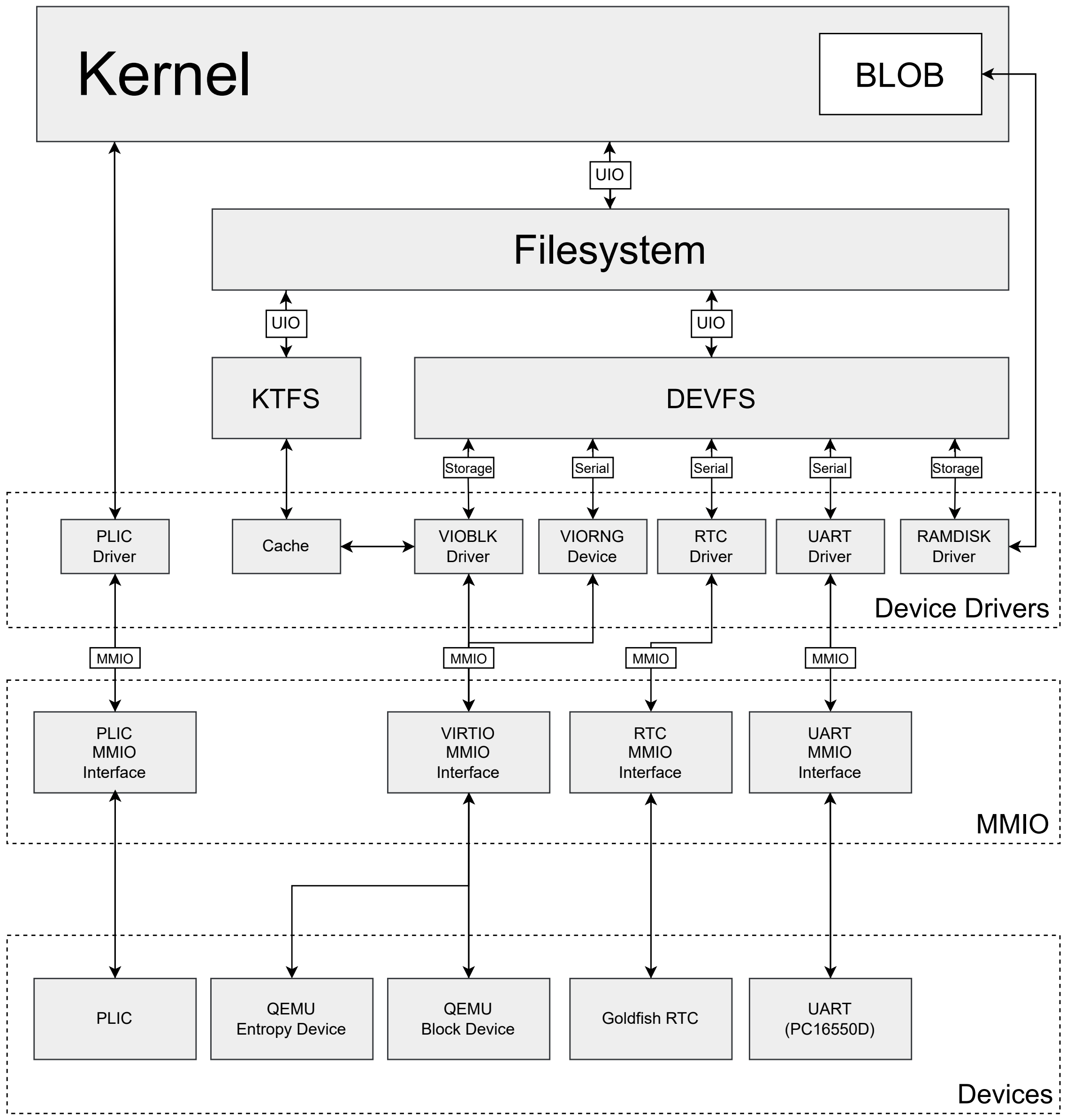
Please read through `filesys.h/c` and the `devfs` functions in `device.c` to gain a better understanding of how the system works.

8.4 Listing Objects

Listing objects are a special type of uio that are specific to mountpoints. It captures all files that are in the mountpoint when the listing object was opened. When passed in a `NULL` or empty string as a filename, this listing object of the mountpoint is returned instead. Calling `read` on the listing object will iterate through the filesystem files, returning a new filename on each call. This is what allows us to pass filesystem information to the user and eventually implement programs like `ls`.

There are three listing objects in your kernel. One is for devfs and is provided in release. Another is for ktfs, which you will have to implement in CP3. The final one is for `filesys.c`, which is returned when `open` is called and the mountpoint name is `NULL` or the empty string. This listing captures the list of mountpoints

8.5 Uniform I/O Diagram



Overall structure of the 391 Kernel.

9 Appendix E: Troubleshooting

9.1 Debugging with gdb

"If debugging is the process of removing bugs, then programming must be the process of putting them in." -Edsger W. Djikstra

One of the most important skills you can have as a software engineer is being able to debug. While print statements are useful and easy to implement, using a real debugger like gdb will allow you to find and solve your problems much more quickly. If you invest time into learning it at the start of this MP, it will pay dividends for the rest of the assignment and future classes. Many ECE 391 alumni (including course staff) have said that the most important thing they learned from the course was not how to make an operating system, but how to debug effecitvely on their own. While course staff are there to help you, they are not there to handhold you and it is expected that you are able to debug on your own, especially with gdb.

9.1.1 Debugging the Kernel

This section describes how to set up your kernel to work with gdb. This will allow you to step through kernel and user program code if it is compiled with debugging symbols (`-g`).

Whenever a change is made to your kernel, you should rebuild your kernel using `make`. For the purposes of this appendix, we'll be describing debugging with the `kernel.elf` kernel image, which we've given you. If you're using a different kernel image, use that instead. You can create different kernel images by writing an appropriate `main_<prog>.c` file and adding it to the `Makefile`.

To launch QEMU and have it wait for remote debugging, you must include the `-S` flag, which we have done for the command `make debug`. This will launch your `kernel.elf` kernel image as normal, but no instructions will execute until you connect to it with gdb and run `continue`.

Once you've launched your kernel, you need to open a second terminal and run

```
riscv64-unknown-elf-gdb [kernel image]
```

This will run the correct version of gdb and load the debugging symbols from your kernel image, allowing you to set breakpoints and step through the code. We have included a `.gdbinit` file in the `sys` directory, which executes a few commands on gdb startup — this should cause it to automatically connect to your QEMU instance. To use the `.gdbinit` file, simply launch gdb from the `sys` directory.

Note: The first time you use a `.gdbinit` file, you might get a message that looks like this

```
warning:  File "/path/to/src/kern/.gdbinit" auto-loading has been declined by
your ‘auto-load safe-path’.
To enable execution of this file add
add-auto-load-safe-path /path/to/src/kern/.gdbinit
line to your configuration file "~/.config/gdb/gdbinit".
```

In order to resolve this, you should simply follow the instructions that gdb gives you and add the path. You may have to use `mkdir` to make the `~/.config/gdb` directory before using your favorite text editor to add the `add-auto-load-safe-path /path/to/src/kern/.gdbinit` line into `~/.config/gdb/gdbinit`. You only need to perform this setup process once per `.gdbinit` file, even if you modify it in the future. It is **highly recommended** that you use the `.gdbinit` file to save you the trouble of having to connect to the QEMU instance every time.

Note: You may need to modify the `target remote 127.0.0.1:1234` line in your `.gdbinit` file to match the port your QEMU instance uses. The `-s` flag provided to QEMU allows it to use the default port (1234), but if you are having trouble connecting with that port, you can explicitly specify which port to use in the QEMU command. Then, modify your `.gdbinit` file to match.

9.1.2 Debugging User Programs

If you want to set breakpoints or step through a user program, things are slightly more complicated. Since `kernel.elf` (or whatever your kernel image is) does not contain the user program, we need to add the debugging symbols for it separately into gdb.

First, you need to compile your user program with debugging symbols enabled. In the `Makefile` we have provided, we have already enabled debugging symbols (`-ggdb3`). If you write your own user program, you can add it to the `Makefile`. To find the address of the program itself, run

```
readelf -Wl [program binary]
```

This gives you information on where different sections of your program are loaded. Find the executable section (`E` flag) and note down the `VirtAddr`. For checkpoint 1, it should be located around `0x80100000`. For later checkpoints, once you have virtual memory enabled, it should be located between `USER_START_VMA` and `USER_END_VMA`. If using `UMODE` and virtual memory, be sure to enable the `UMODE` flag within `usr/Makefile`. You may need to manually tell gdb that this is where the program starts.

To do this, open gdb as normal with `make debug`, then use the `add-symbol-file` command to add your user program. An example would be

```
add-symbol-file ../usr/bin/hello
```

If there is a prompt, hit ”Y” to accept. Now, you should be able to set breakpoints and step through the user program. You will need to run this command every time that you re-open gdb — if you find yourself using it constantly, consider adding it to your `.gdbinit`.

9.1.3 Useful gdb Commands

The best way to get good at using gdb is to actually use it. To get you started, here are some pages with gdb commands we found useful. There are a lot more out there, and it is highly encouraged that you spend time looking at the gdb man page (`man gdb`) and any other online resources for gdb.

Note: Your code may run noticably slower when using gdb, especially if you are performing a lot of comparisons.

- Stanford CCRMA gdb Guide
- gdb Wiki

Here are some additional useful gdb tips and tricks:

1. Manipulating Local Variables

   You can use arithmetic operations (+, -, *, /, etc.) as well as pointer arithmetic (*, &, [], etc.) on variables in gdb. You can also cast variables or numbers, just like in C. To access the fields of a struct, use `.` or `->` appropriately. This is most useful with `p[rint]`.

2. Watchpoints

   You can use watchpoints to ”watch” the value of a variable and trigger a breakpoint if it changes. Use `watch <var>` to set a watchpoint on a variable.

3. Conditional Breakpoints and Watchpoints

   You can use the format `break WHERE if CONDITION` to set a conditional breakpoint, where it will only happen if a certain condition is met. For example, `watch i if i == 100` would break if `i == 100`. If your variable is constantly being modified, this can cause a major slowdown in your program execution.

4. Printing Registers

   Using `i[nfo] r[registers]` can be useful to see all (or any specific) registers, but all registers can also be referenced with the `$<register name>` variable. For example, `p/x $sp` prints out the value of `sp` in hexadecimal. However, for CSRs, it is recommended you use `i[nfo] r[egisters]` in order to get ”pretty printing”.

5. gdb History

   By adding `set history save on` to your `.gdbinit` file, you can access the history of commands from previous gdb sessions. You can also further customize the file location and history depth.

6. `until` and `finish`

   When using gdb, stepping through long for loops can be somewhat annoying. Typically, you will set a breakpoint after the loop and use `c[ontinue]` to skip over it. However, you can also use the `until <line number>` command to skip without having to set a breakpoint.

   `fin[ish]` works in a similar way. It runs until the current function exits, then reports the return value of the function. You may find this useful if you accidentally `s[tep]` instead of `n[ext]`.