CP3.1

ECE391: Computer Systems Engineering **Fall 2024**
Machine Problem 3
Checkpoint 1: Nov 6, 2024 at 11:59pm
Checkpoint 2: Dec 2, 2024 at 5:59pm
Checkpoint 3: Dec 9, 2024 at 5:59pm

Illinix 391

Contents

CP3.1

1 Introduction

Read the whole document before you begin, or you may miss points on some requirements (*e.g.*, the bug log).

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 7 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.

While it is often unnecessary, we recommend to run the following commands to make sure the line endings are set to LF (Unix style):

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

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 files, besides what we have given you. You can modify your `.gitignore` in any way you want.

Finally, merge your changes into the `master` 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 `printf`. We **highly** encourage use of GDB to debug. Print statements can only take you so far. See **Appendix H** 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` 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

Although we are not explicitly requiring that you tell us how you plan to split up the work for this project, we do expect that you will want to do so to allow independent progress by all team members. A suggested split of the work is to have each person work on one of the subgoals of each checkpoint. For Checkpoint 1, this could be split into filesystem, `virtio`, and `elf_load`.

Setting up a clean testing interface will also help substantially with partitioning the work, since group members can 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.

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

CP3.1

4 Testing

For this project, we require you to demonstrate 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. Minimum test coverage for each checkpoint is detailed in the following sections.

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.

CP3.1

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 terminal, then stop. `trek` will run the same text-basd 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.

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 Device Driver Setup (+I/O)

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. Much of the necessary device driver code has already been written, your group must finish the implementation based on the virtio documation 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. Skip anything related to "legacy" interface.

These functions should be written in vioblk.c :

1. `void vioblk_attach(volatile struct virtio_mmio_regs * regs, int irqno)`

   Initializes the virtio block device with the necessary IO operation functions and sets the required feature bits. Also fills out the descriptors in the virtq struct. It attaches the `virtq_avail` and `virtq_used` structs using the `virtio_attach_virtq` function. Finally, the interupt service routine and device are registered.

2. `int vioblk_open(struct io_intf ** ioptr, void * aux)`

   Sets the `virtq_avail` and `virtq_used` queues such that they are available for use. (Hint, read virtio.h) Enables the interupt line for the virtio device and sets necessary flags in `vioblk_device`. Returns the IO operations to `ioptr`.

3. `void vioblk_close(struct io_intf * io)`

   Resets the `virtq_avail` and `virtq_used` queues and sets necessary flags in `vioblk_device`.

4. `long vioblk_read (struct io_intf * restrict io,`
   `void * restrict buf, unsigned long bufsz)`

   Reads `bufsz` number of bytes from the disk and writes them to `buf`. Achieves this by repeatedly setting the appropriate registers to request a block from the disk, waiting until the data has been populated in block buffer cache, and then writes that data out to `buf`. Thread sleeps while waiting for the disk to service the request. Returns the number of bytes successfully read from the disk.

5. `long vioblk_write (struct io_intf * restrict io,`
   `const void * restrict buf, unsigned long n)`

   Writes `n` number of bytes from the parameter `buf` to the disk. The size of the `virtio_device` should not change. You should only overwrite existing data. Write should also not create any new files. Achieves this by filling up the block buffer cache and then setting the appropriate registers to request the disk write the contents of the cache to the specified block location. Thread sleeps while waiting for the disk to service the request. Returns the number of bytes successfully written to the disk.

6. `vioblk_isr(int irqno, void * aux)`

   Sets the appropriate device registers and wakes the thread up after waiting for the disk to finish servicing a request.

7. `vioblk_getlen(const struct vioblk_device * dev, uint64_t * lenptr)`

   Ioctl helper function which provides the device size in bytes.

8. `vioblk_getpos(const struct vioblk_device * dev, uint64_t * posptr)`

   Ioctl helper function which gets the current position in the disk which is currently being written to or read from.

9. `vioblk_setpos(struct vioblk_device * dev, const uint64_t * posptr)`

   Ioctl helper function which sets the current position in the disk which is currently being written to or read from.

10. `vioblk_getblksz (const struct vioblk_device * dev, uint32_t * blkszptr)`

    Ioctl helper function which provides the device block size.

See **Appendix B** for more information on the `io_intf` struct.

5.1.3 Filesystem Abstractions

Broadly speaking, your filesystem abstraction should provide a comfortable interface to open, read and scan through files.

Your `kfs.c` file will need to interact with `vioblk.c` to actually interact with the “physical” (well, virtual) device, so be sure that you understand what’s going on in that file. Additionally, insofar as this interacts with virtual devices, you should be sure that you use the `io_intf` struct in the proper way.

These functions should be written in kfs.c :

1. `int fs_mount(struct io_intf* io)`

   Takes an `io_intf*` to the filesystem provider and sets up the filesystem for future `fs_open` operations. Once you complete this checkpoint, the `io` argument will be associated with the `vioblk_device`.

2. `int fs_open(const char* name, struct io_intf** io)`

   Takes the name of the file to be opened and modifies the given pointer to contain the `io_intf` of the file. This function should also associate a specific file struct with the file and mark it as in-use. The user program will use this `io_intf` to interact with the file.

3. `void fs_close(struct io_intf* io)`

   Marks the file struct associated with `io` as unused.

4. `long fs_write(struct io_intf* io, const void* buf, unsigned long n)`

   Writes `n` bytes from `buf` into the file associated with `io`. The length of the file should not change. You should only overwrite existing data. Write should also not create any new files. Updates metadata in the file struct as appropriate. The caller will first use `fs_open` to get the `io` argument.

5. `long fs_read(struct io_intf* io, void* buf, unsigned long n)`

   Reads `n` bytes from the file associated with `io` into `buf`. Updates metadata in the file struct as appropriate. The caller will first use `fs_open` to get the `io` argument.

6. `int fs_ioctl(struct io_intf*, int cmd, void* arg)`

   Performs a device-specific function based on `cmd`. Note, ioctl functions should return values by using `arg`. See `io.h` for details.

7. `int fs_getlen(file_t* fd, void* arg)`

   Helper function for `fs_ioctl`. Returns the length of the file.

8. `int fs_getpos(file_t* fd, void* arg)`

   Helper function for `fs_ioctl`. Returns the current position in the file.

9. `int fs_setpos(file_t* fd, void* arg)`

   Helper function for `fs_ioctl`. Sets the current position in the file.

10. `int fs_getblksz(file_t* fd, void* arg)`

    Helper function for `fs_ioctl`. Returns the block size of the filesystem.

In order to use the filesystem, we have provided a `mkfs` function (see **Appendix A**) that generates a filesystem image for you. This filesystem image is mounted by QEMU as a drive (using the `Makefile` we provide) and is accessible through virtio.

For every instance of `io_intf`, you will need to implement ioctls `IOCTL_GETLEN`, `IOCTL_GETPOS`, and `IOCTL_SETPOS`. `IOCTL_GETBLKSZ` is optional, but may be helpful when implementing the filesystem `io_intf` functions.

You will also need to implement `io_lit`, which treats a buffer in memory as a device and provides an `io_intf` to interact with it. Defined in `io.c/.h`, `io_lit` requires specialized `io_intf` operations that are used to interact with the corresponding memory buffers. You will need to implement these operations.

See **Appendix A** and **Appendix B** for additional details.

CP3.1

5.1.4 Program Loading

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

We've given you a few user-level programs, 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 `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.

Your task here is to work on the `elf_load` function, which ingests an `io_intf` to read the executable and do some validation on it (to make sure it's *actually* an executable). It is then the job of `elf_load` to read the ELF header and process the program headers. The loader should only load program header entries of type `PT_LOAD`. You must create an `elf.c` file that contains `elf_load`, alongside any relevant structs and magic numbers (use `#define`!). You can find the layouts of structs and values of the magic numbers in the ELF Linux man page. You are allowed to look at `.../uapi/linux/elf.h` in the Linux source code.

Notice that since `elf_load` should support any compliant I/O interface, that we can in general load an ELF from "any source" as long as `ioread` and `ioseek` are implemented in the given `io_intf` (see **Appendix B**).

Your `elf_load` should verify that all of the sections of the program are loaded between `x80100000` and `0x81000000`.

5.1.5 Companion file

There is a script called `mkcomp.sh` in the `kern` directory. This script expects one argument, the relative path to a file.

Running this creates an otherwise-empty file called `companion.o`, placing in that object a section containing the raw data of that file. Then, when you link the kernel together (by calling a `make` task) and run it, the kernel will have access to the data in that file, if it uses the corresponding labels in `kernel.ld`.

There is also a task in the Kernel makefile which will call the script on the user `trek` program (if it has been built), and create the corresponding `companion.o` object.

5.1.6 Existing Files

During MP2, you worked with the PLIC, UART, and the timer. You will be re-using that code for MP3. You should add the following files into `kern` from MP2. They must be fully functional and have the same behavior as in MP2.

- `plic.c/.h`
- `timer.c/.h`
- `thread.c/.h`

- `thrasm.s`

5.1.7 Troubleshooting/Debugging

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

5.1.8 Test Coverage

At minimum, your tests should cover:

- Full I/O operations on `vioblk`
- Full I/O operations on `io_lit`
- Full I/O operations on the filesystem.

This list is subject to change. Keep an eye out for the rubric. You are encouraged to add more unit tests than those specified above.

5.1.9 Checkpoint 1 Handin

For handin, it's expected that you will demonstrate proper functionality by writing your own unit tests. For this checkpoint, you should be able to show that all the core functionality required here works. Namely:

- A bug log listing out the major bugs encountered, the file(s) associated with the bug, and how they were fixed.
- Be able to compile all the files associated with your kernel and user programs.
- Show that you can perform full I/O on the `vioblk` device.
- Show that you can perform full I/O on the `io_lit` device.
- Show that you can mount the filesystem, open a file, and perform full I/O on the file.
- Show that you can load a user program into memory.
- Be able to execute user programs (`hello` and `trek`) from the shell.

CP3.1

5.2 Checkpoint 2: Virtual Memory and Process Abstraction

5.2.1 Given Code

Linked on the course website are some extra files which are needed for the Checkpoint 2 implementation. Add them to your repo before starting Checkpoint 2 development. Some files need to replace existing files with the same name. It is recommended to store the old files in a different directory under a new name instead of completely removing them.

Files to add to the `kern` directory:

1. `config.h`, contains new memory locations
2. `csr.h`, should replace your existing file
3. `error.h`, should replace your existing file. Use this for error return values
4. `excp.c`
5. `ezheap.c`, should replace your existing file
6. `halt.c`, should replace your existing file
7. `intr.c`, should replace your existing file
8. `intr.h`, should replace your existing file
9. `kernel.ld`, should replace your existing file
10. `main.c`, this is the new starting point after `start.s` runs
11. `Makefile`, you still need to add object files you added
12. `memory.c`
13. `memory.h`
14. `process.c`
15. `process.h`
16. `scnum.h`
17. `start.s` should replace your existing file
18. `thrasm.s` should replace your existing file
19. `thread.c` should replace your existing file
20. `thread.h` should replace your existing file
21. `trap.h`, should replace your existing file
22. `trapasm.s` should replace your existing file

Files to add to the `user` directory:

1. `error.h`, should replace your existing file
2. `init0.c`, the most basic test program
3. `init1.c`, a slightly more complex test program
4. `init2.c`, a test program that uses the `_exec` syscall
5. `Makefile`, you still need to add any test binaries you've made
6. `scnum.h`, should be the same as in `kern`
7. `start.s`
8. `syscall.h`
9. `syscall.S`
10. `user.ld`, should replace your existing file
11. `trek`, new binary file which uses system calls

5.2.2 Vicious Virtual Memory

In order to begin working with virtual memory, we need to implement supervisor (S) mode. S mode uses a different register set than machine (M) mode, which is what we've been using in MP2 and MP3 CP1. The kernel boots up in `start.s` in M mode before switching to S mode after the `mret` instruction. Ensure that S mode is implemented properly by using the appropriate S mode registers where M mode registers were previously used. The helper functions in `csr.h` should help with this task. `plic.c` will need to be modified. `intr.c/h` were provided to switch from M mode to S mode. **Be careful** to make sure that you have actually modified every instance! You may need to refer back to the PLIC documentation.

Note : This checkpoint will only contain a single memory space. You will not need to create new memory spaces for this checkpoint besides the "main" memory space

We've given you a file, `memory.c`, where you will implement all of the functions in `memory.h`. You will need to write the following functions in order to get paging up-and-running:

1. `void memory_init(void)`

   Sets up page tables and performs virtual-to-physical 1:1 mapping of the kernel megapage. Enables Sv39 paging. Initializes the heap memory manager. Puts free pages on the free pages list. Allows S mode access of U mode memory. We have provided most of this function for you.

2. `void memory_space_reclaim(void)`

   Switch the active memory space to the main memory space and reclaims the memory space that was active on entry. All physical pages mapped by the memory space that are not part of the global mapping are reclaimed.

3. `void *memory_alloc_page(void)`

   Allocate a physical page of memory using the free pages list. Returns the virtual address of the direct-mapped page as a `void*`. Panics if there are no free pages available. `ezheap.c` will call this function when the heap is full.

4. `void memory_free_page(void *pp)`

   Return a physical memory page to the physical page allocator (free pages list). The page must have been previously allocated by `memory_alloc_page`.

5. `void *memory_alloc_and_map_page(uint64_t vma, uint8_t rwxug_flags)`

   Allocate a single physical page and maps a virtual address to it with provided flags. Returns the mapped virtual memory address.

6. `void *memory_alloc_and_map_range(uint64_t vma, size_t size, uint8_t rwxug_flags)`

   Allocates the range of memory and maps a virtual address with the provided flags. Returns the mapped virtual memory address.

7. `void memory_set_page_flags(const void *vp, uint8_t rwxug_flags)`

   Sets the flags of the PTE associated with `vp`. Only works with 4 kB pages.

8. `void memory_set_range_flags(const void *vp, size_t size, uint8_t rwxug_flags)`

   Modify `flags` of all PTE within the specified virtual memory range.

9. `void memory_unmap_and_free_user(void)`

   Unmaps and frees ALL user space pages (that is, all pages with the `U` flag asserted) in the current memory space.

10. `int memory_validate_vptr_len(const void *vp, size_t len, uint8_t rwxug_flags)`

    (Extra Credit) Ensure that the virtual pointer provided (`vp`) points to a mapped region of size `len` and has at least the specified flags.

11. `int memory_validate_vstr(const char *vs, uint8_t ug_flags)`

    (Extra Credit) Ensure that the virtual pointer provided (`vs`) contains a NUL-terminated string.

12. `void memory_handle_page_fault(const void *vptr)`

    Handle a page fault at a virtual address. May choose to panic or to allocate a new page, depending on if the address is within the user region. You must call this function when a store page fault is triggered by a user program.

Some of these functions may build off of others. Some functions will also be called by functions used to implement processes (§5.2.3) Your code must meet the functionality requirements outlined in the rubric.

We've also provided many helper functions that we found useful in implementing virtual memory. You do not need to use them, but they may make your job easier. There is also a more complex helper function that we have not provided but is described below:

```
struct pte* walk_pt(struct pte* root, uintptr_t vma, int create)
```

This function takes a pointer to your active root page table and a virtual memory address. It walks down the page table structure using the VPN fields of `vma`, and if `create` is non-zero, it will create the appropriate page tables to walk to the leaf page table (”level 0”). It returns a pointer to the page table entry that represents the 4 kB page containing `vma`. This function will only walk to a 4 kB page, not a megapage or gigapage.

Consult **Appendix D** for more information about virtual memory.

5.2.3 Pretty Processes

The process abstraction is one of the key abstractions of an operating system. A process can be defined informally as just a ”running user program”. A user often wants to run multiple processes at once which requires common resources like processing power, devices, and memory to be managed.

An instance of a process structure contains everything that a process owns and uses internally. Each user process is actually just a wrapper around a kernel thread. What this means is whenever a user process is created, a process struct will have to be initialized to contain the information below:

1. An identifier for the current process
2. An identifier for the kernel thread related to this process
3. An identifier for the memory space of the process
4. A list of I/O interfaces for this process ; remember that an I/O interface can currently represent
    - A terminal device
    - An open file
    - A block device
    - An in-memory buffer

In this checkpoint, all user processes will share the same memory space, dubbed the ”main” memory space. The execution lifecycle of a process will be as follows

1. The kernel launches in S-mode.
2. `procmgr_init` is called. Creating a process struct around the main thread
3. `process_exec` jumps to user mode and starts executing a user program. Effectively turning the main kernel thread into a user process
4. When the user process exits, the kernel exits

Our kernel space process API is made up of the functions below. These should be written in `process.c`:

1. `void procmgr_init(void)`

   Initializes processes globally by initializing a process structure for the main user process (init). The init process should always be assigned process ID (PID) 0.

2. `int process_exec(struct io_intf *exeio)`

   Executes a program referred to by the I/O interface passed in as an argument. We only require a maximum of 16 concurrent processes.

   Executing a loaded program with `process_exec` has 4 main requirements:

   (a) First any virtual memory mappings belonging to other user processes should be unmapped.

   (b) Then a fresh 2nd level (root) page table should be created and initialized with the default mappings for a user process. (This is not required for Checkpoint 2, as in Checkpoint 2 any user process will live in the "main" memory space.)

   (c) Next the executable should be loaded from the I/O interface provided as an argument into the mapped pages. (Hint: `elf_load`)

   (d) Finally, the thread associated with the process needs to be started in user-mode. (Hint: An assembly function in `thrasm.s` would be useful here)

   Context switching was relatively trivial when both contexts were at the same privilege level (i.e. machine-mode to machine-mode switching or supervisor-mode to supervisor-mode switching), but now we need to switch from a more privileged mode (supervisor-mode) to less privileged mode (user-mode).

   Doing so requires using clever tricks with supervisor-mode CSRs and supervisor-mode instructions. Here are some tips to consider while implementing a context switch from supervisor-mode to user-mode

   i. Consider instructions that can transition between lower-privilege modes and higher privilege modes. Can you repurpose them for context switching purposes?

   ii. If you did repurpose them for context switching purposes, what CSRs would you need to edit so that the transition would start the thread's start function in user-mode?

   It's a useful exercise to try to figure out how such an approach could work with the CSRs and supervisor-mode instructions on your own. However, implementation on its own is a sufficient challenge and we don't require you to figure this out. You can read **Appendix C** to find out how you can carry out a context switch between user-mode to supervisor mode.

3. `void process_exit(void)`

   Cleans up after a finished process by reclaiming the resources of the process. Anything that was associated with the process at initial execution should be released. This covers:

   - Process memory space
   - Open I/O interfaces
   - Associated kernel thread

5.2.4 Suspicious Syscalls

You will need to implement a series of syscalls (system calls) for this checkpoint. The user program uses these to request actions from the kernel.

The syscalls you must implement for this checkpoint are:

1. `static int sysexit(void);`

   Exits the currently running process.

2. `static int sysmsgout(const char *msg);`

   Prints `msg` to the console.

3. `static int sysdevopen(int fd, const char *name, int instno);`

   Opens a device at the specified file descriptor and returns error code on failure.

4. `static int sysfsopen(int fd, const char *name);`

   Opens a file at the specified file descriptor and returns error code on failure.

5. `static int sysclose(int fd);`

   Closes the device at the specified file descriptor.

6. `static long sysread(int fd, void *buf, size_t bufsz);`

   Reads from the opened file descriptor and writes `bufsz` bytes into `buf`.

7. `static long syswrite(int fd, const void *buf, size_t len);`

   Reads `bufsz` bytes from `buf` and writes it to the opened file descriptor.

8. `static int sysioctl(int fd, int cmd, void *arg);`

   Performs desired ioctl based on `cmd`

9. `static int sysexec(int fd);`

   Halts currently running user program and starts new program based on opened file at file descriptor.

10. `extern void syscall_handler(struct trap_frame *tfr);`

    Called from the usermode exception handler to handle all syscalls. Jumps to a system call based on the specified system call number. (Hint: Look at `syscall.S`)

The function `syscall_handler` is used for system call linkage. Looking into the `syscall.S`, we see that system call exceptions are generating using the `ecall` instruction. The exception should be handled by `_trap_entry_from_umode` in `trapasm.S`. This assembly code should call the `umode_excp_handler` function in `excp.c`. User mode exception handling is used for implementing both system calls and page fault handling. You will need to add to this function.

(Extra Credit) System calls pass as parameters pointers in user memory. This is a vulnerability since the kernel could read or write from invalid memory. Use the virtual memory validation functions to protect the kernel from malicous user programs.

CP3.1

5.2.5 Additional Modifications

Due to the new virtual memory, process abstraction, and system calls, we are now able to run programs in User mode instead of Supervisor mode. This means that some of the code you wrote in Checkpoint 1 will no longer work. Here is a (non-exhaustive) list of modifications you'll have to make to your existing code:

1. `elf.c/h` must be modified to use virtual memory. User programs should now be loaded between the `USER_START_VMA` and `USER_END_VMA` virtual addresses. You will need to allocate and map the appropriate amount of pages, as well as set the appropriate flags in the page table. You should also modify the `entryptr` argument to `void (**entryptr)(void)`.

2. `main_shell.c` will be no longer used, since it launches programs in S mode. You will now use `main.c` (provided) as the "main" function of your kernel. `start.s` (provided) provides an assembly linkage which sets up the `mtvec` and `stvec` registers, switches to S mode, and calls `main`. `#define INIT_PROC` in `main.c` is the name of the program you want to execute first. For testing, we have given you `init0`, `init1`, and `init2` as well as a new `trek` binary. It is **very important** that you use the new `trek` binary, since the old one will not work with syscalls. `hello` will also no longer work. You should remake your filesystem image using `mkfs` with the new programs. We suggest starting with `init0` as your most basic test program, although you will have to have all 3 working for this checkpoint.

5.2.6 Checkpoint 2 Handin

For handin, it's expected that you will demonstrate proper functionality by writing your own unit tests. For this checkpoint, you should be able to show that all the core functionality required here works. Namely:

- A bug log listing out the major bugs encountered, the file(s) associated with the bug, and how they were fixed.
- Be able to compile all the files associated with your kernel and user programs.
- Working pointer dereference in valid virtual memory.
- Testing page fault during dereference outside valid memory, including near page boundaries.
- Show that you can load a user program into memory.
- Be able to execute user programs `init0`, `init1`, and `init2`.

CP3.1

5.3 Checkpoint 3: Fork, Lock, Reference Counting and Preemptive Multitasking

5.3.1 Given Code

Linked on the course website are some extra files which are needed for the Checkpoint 3 implementation. Add them to your repo before starting Checkpoint 3 development. Some files need to replace existing files with the same name. It is recommended to store the old files in a different directory under a new name instead of completely removing them.

Files to add to the `kern` directory:

1. `io.h` should replace your existing file
2. `lock.h`
3. `main.c` should replace your existing file
4. `Makefile` should replace your existing file
5. `mmode_trapasm.s` contains interrupt functions for M mode to place in your `trapasm.s` file
6. `scnum.h` should replace your existing file
7. `start.s` should replace your existing file
8. `thread_spawn.c` contains `thread_spawn` function to replace in your `thread.c` file
9. `timer.c` should replace your existing file
10. `timer.h` should replace your existing file
11. `uart.c` should replace your existing file and includes an example of reference counting

Files to add to the `user` directory:

1. `fib.c` runs a compuational intensive task to calculate Fibonacci numbers
2. `init_fib_fib.c` runs two instances of fib using fork
3. `init_fib_rule30.c` runs instance of fib and rule30 using fork
4. `init_trek_rule30.c` runs instance of trek and rule30 using fork
5. `Makefile` should replace your existing file
6. `syscall.h` should replace your existing file
7. `syscall.S` should replace your existing file

You also must update the `.equ` statement in `thrasm.s` to increase `IDLE_STACK_SIZE` to 4096 (from 1024).

You can get rid of `io.c/.h` from the `user` directory, since programs no longer receive pointers into the kernel. We also suggest creating things called symbolic links (symlinks), so that certain files will be synchronized between your `user` and `kern` directories. This will prevent you from having to copy and paste things back and forth. The files you should symbolic-link are probably: `scnum.h, string.c, string.h`. A mini-tutorial has been provided in **Appendix F**.

5.3.2 Friendly Forking

“Fork this, I’m out” -Bobby

In Checkpoint 3 we need to support ”forking” processes so we can have multiple processes running in our system. We’re starting off with the `fork` syscall which has the following function signature.

- `static int sysfork(const struct trap_frame * tfr);`

The `fork` system call duplicates the currently running process and creates a child process which starts at the same point in the original or parent process. `fork` returns the pid of the child process to the parent process. It returns 0 to the child process. You will also need to connect this system call to your syscall handler.

It starts by allocating a new process and copying all the `iotab` pointers from the parent to the child process. It also initializes all the reference counts as described in §5.3.4. Fork then completes its task by calling the following helper functions with the specified signature.

- `extern int thread_fork_to_user (`
  `struct process * child_proc, const struct trap_frame * parent_tfr);`

This function allocates new memory for the child process and sets up another thread struct. It also initializes a stack anchor to reclaim the thread pointer when coming back from a U mode interrupt. The child’s memory space is switched into and the thread is set to be run. Another helper function, with the following signature, should be written in assembly which performs the context switch:

- `extern void _thread_finish_fork (`
  `struct thread * child, const struct trap_frame * parent_tfr);`

This function begins be saving the currently running thread. Switches to the new child process thread and back to the U mode interrupt handler. It then restores the ”saved” trap frame which is actually the duplicated parent trap frame. Be sure to set up the return value correctly, it will be different between the child and parent process. As always, we `sret` to jump into the new user process.

- `uintptr_t memory_space_clone(uint_fast16_t asid);`

Function implemented in `memory.c` that should clone your memory space for current process and return the mtag of the new memory space. Should be used in `thread_fork_to_user` to setup the memory space for the child process.

You may be delighted to hear that the `fork` implementation here is virtually identical to how it’s done in “real” UNIX-like operating systems, yay!

5.3.3 Serene Sleeping and Whimsical Waiting

You will need to implement the system calls described in lecture: `wait`, which waits for the child to exit, and `usleep`, which sleeps for a parameter (the number of microseconds). Here are their function signatures:

1. `static int syswait(int tid);`

   Wait for certain child to exit before returning. If tid is the main thread, wait for any child of current thread to exit.

2. `static int sysusleep(unsigned long us);`

   Sleep for us number of microseconds. (Hint: Read `timer.c`)

Make sure to add these new system calls to the appropriate files in the `user` directory as well.

5.3.4 Really Cooked Reference Counting

Reference counting is a typical operation taken for memory management. In our case, when a program is forked, its current execution state is effectively "cloned".

Part of the state we are already keeping for each process is the `io_intf` array. By incrementing the reference count field we can keep track of how many active references to this particular `io_intf` exist.

In this way, we prevent closing the `io_intf` while there is still a process somewhere that depends on it being open. In the updated version of `io.h` we distributed, you will need to add this functionality to `ioclose`. You should decrement the `refcnt` of the `io_intf` each time that `ioclose` is called, and when the `refcnt` reaches 0 (all programs that opened this device have called close), you should actually close the device. As a consequence, you should be using the helper function instead of the raw function pointer, to properly respect the reference count field. Similarly, your `io_intf`s should initialize their reference count to 1 when they are created (during their respective open functions). Make sure you add this functionality to **all** functions that initialize an `io_intf`.

5.3.5 Lethargic Locking

Since we have multiple processes, we should use a lock to decide who gets access to shared data. You only need a sleep lock for this MP. We have provided an almost-finished implementation of a sleep lock in `lock.h` (part of the distributed files on the course site), all you have to do is write the lock acquisition function `lock_acquire`. Nice!

You must use this lock in your `kfs` and `vioblk` drivers. This involves:

1. Declaring a lock within your driver file
2. Initializing the lock using `lock_init` during your "setup"
3. Acquiring (and later releasing!) the lock when you do read/write operations.

5.3.6 Perniciously Meticulous Preemptive Multitasking

In this checkpoint you will be implementing preemptive multitasking for programs running in user mode. Preemptive multitasking is very similar to the cooperative multitasking from MP2 but instead of calling `thread_yield` synchronously through the program itself during execution, it is called asynchronously via the interrupt handler. This will preempt the currently running thread by causing it to suspend and hand over control to the next waiting thread. Remember we will only preempt the thread if we are running programs in the user mode.

5.3.7 Checkpoint 3 Handin

For this checkpoint, you should be able to show that all the core functionality required here works. Namely:

- A bug log listing out the major bugs encountered, the file(s) associated with the bug, and how they were fixed.
- Be able to compile all the files associated with your kernel and user programs.
- Be able to execute programs `init_trek_rule30`, `init_fib_fib`, `init_fib_rule30`, and `fib`.
- Any tests showing working functionality for previous checkpoints.

5.4 Extra Credit

We will hold a design competition at the end of the semester. In order to be eligible for this competition, your operating system **must implement all required functionality correctly**. The competition will then be decided based on the amount and difficulty of additional functionality that your team has incorporated into your operating system.

You may also be able to earn some extra credit (or make up for other lost points) by choosing to include some of the features in this section. Extra credit earned in this way will be limited to 10 points of the baseline grade.

Details about the design competition will be found on Piazza as the date approaches.

5.4.1 Signals

Add support for delivery of five different signals to a task. We are relatively flexible on how you choose to implement this (for Fall 2024 only), but try to make it comply to the general specification of how signal delivery works.

You may want to look into the POSIX spec (search “opengroup”) to find details.

We also expect this to be a helpful resource if you choose to pursue this extra credit: link to gnu.org

5.4.2 Pipes

You may opt to add support for UNIX-style pipes in your operating system.

We recommend creating an I/O interface provider, which you might consider (but have no obligation) naming `io_pipe`. This interface would function similarly to your existing I/O interfaces and will include the logic required for sending data between different processes.

This implementation facilitates one-way communication, allowing messages to be sent from one program to another, kind of like a mailbox. Mature operating systems will often have pipes actually be bidirectional (where the backing interface is different if you “choose”) to make your pipe bidirectional. If you want, you can implement this functionality, but bidirectional pipes are also okay.

Consult typical sources for UNIX documentation to learn more about the details of pipes. You do not need a feature-complete implementation, but more functionality is good (this ties back in to the note above about extra credit quality evaluations).

Another consideration: if you are interested in exploring more complex extra credit features, the pipe mechanism will likely be quite important for getting those to work, in the absence of some other important syscalls (e.g. in certain situations, it can take the place of the absent `select` or `poll` calls).

5.4.3 . . . And everything else

If you have an impressive feature in mind for extra credit, you are free to pursue that. If your heart burns with desire to win the design competition or many extra credit points, you may want to check in with course staff or professor(s) to see if your idea is reasonable / how “impressive” we would consider it.

For instance, we are unlikely to grant much (if any) extra credit points for dynamic memory allocation—there are numerous implementations all over the internet / you’ve likely done this in other classes.

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

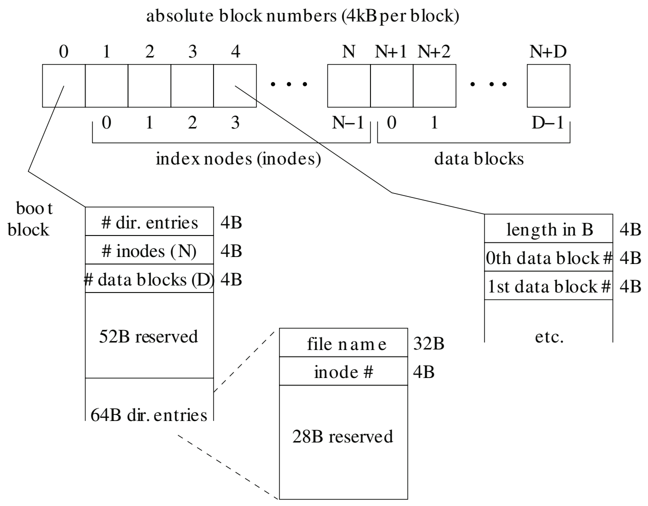
Teamwork is an important part of this class and will be used in the grading of this MP. We expect that you will work to operate effectively as a team, leveraging each member’s strengths and making sure that everyone understands how the system operates to the extent that they can explain it. In the final 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.**

We will also ask each of you to apportion credit for the overall MP to each of the other team members (not including yourself) in order to gauge how contributions were balanced amongst your team. This information will be used to adjust your final score for the MP. Teams that operate smoothly together are free to opt for simply marking the form as equal, which is fine, but the information that you provide about relative effort is treated as confidential. Note that you cannot affect your own grade through any choice of credit assignment, only those of your teammates (and vice-versa).

7 Appendix A: The File System

7.1 File System Overview

The figure below shows the structure and contents of the file system. The file system memory is divided into 4 kB blocks. The first block is called the boot block, and holds both file system statistics and the directory entries (dentries). Both the statistics and each directory entry occupy 64B, so the file system can hold up to 63 files.



Each directory entry gives a name (up to 32 characters, zero-padded, but *not necessarily including a terminal EOS or 0-byte*) and an index node number for the file.

Each regular file is described by an index node (inode) that specifies the file's size in bytes and the data blocks that make up the file. Each block contains 4 kB; only those 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 inode to access the correct data in the correct order.

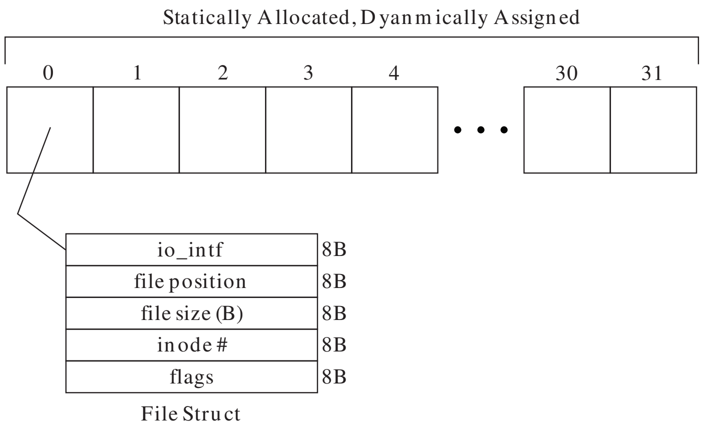
7.2 File System Abstractions

Each task can have up to 32 open files. While the data associated with each file is stored on the device that stores the filesystem, the metadata that our filesystem interface needs to properly read/write/etc. on that file is stored in an array of **file structs**.

This array should store structures containing:

1. The `io_intf` structure associated with each file. The `ops` field in this structure should be a pointer to an `io_ops` struct that contains the correct `close, read, write, ctl` function pointers to interface with the file.

2. A "file position" member that keeps track of where the user is currently reading from/writing to in the file. This should be updated whenever the user reads, writes, or uses `ioseek`.
3. A "file size" member that stores the length of the file in bytes.
4. The inode number for this file.
5. A "flags" member for, among other things, marking this file struct as "in-use."



When `fs_open` is called, the kernel should find the next available file struct in the array and set up the metadata associated with the file.

7.3 Building the File System

The raw data file that will contain our filesystem (*i.e.*, 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 a file `mkfs.c` within the `util` directory. `mkfs` is a customized version of the Linux `mkfs` function that generates a filesystem image according to our MP3 spec detailed above (*i.e.*, modified `ext2`).

In order to use `mkfs`, you must build it using the `Makefile` in `util`. It's usage is as follows:

```
./mkfs [filesystem image output] [file1] [file2] ...
```

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

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

The files in your filesystem image are also given as arguments to `mkfs`. You can (and should) provide multiple files in order to create a full filesystem image. `mkfs` will only recognize file paths that are of these 3 formats: `../user/bin/[file]`, `user/bin/[file]`, or `[file]`. These files will have appropriate dictionary entries made in the boot block, be assigned an inode, and have their data stored in data blocks. All of these blocks will be put in order according to the spec above and stored in your filesystem image (*i.e.*, kfs.raw). This means that the filesystem image is already in the MP3 order: boot block, then inodes, then data blocks.

To load a user program into the filesystem image, you need to compile it to an executable binary. This is done by using the Makefile in user. If you want to change a file in the filesystem or add/remove files, you **must** remake the filesystem image.

CP3.1

8 Appendix B: I/O Interfaces

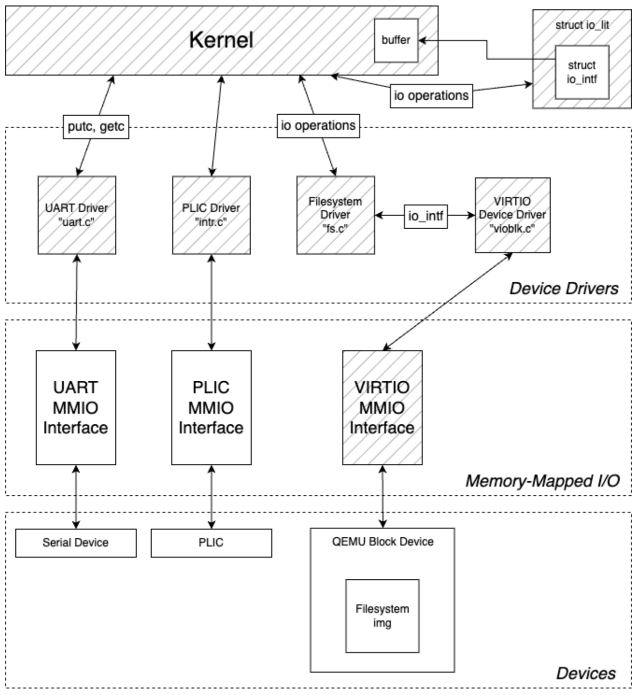
8.1 Overview

For Checkpoint 1, you must support three devices, the UART, the PLIC, and the filesystem block device. When interacting with devices, abstraction is necessary. Historically, companies have come together in consortiums related to networking and inter-device communication (see PCI-E, USB, CCIX). These interconnects enable fast, standardized communication between devices.

QEMU is simulating devices and their interconnects. Your filesystem image, for example, (see Appendix A) will be managed by the QEMU block device.

8.2 I/O Operations

The kernel will interact with devices using the struct `io_intf`. Each struct `io_intf` contains a pointer to a struct `io_ops`, which in turn contains pointers to `close`, `read`, `write`, and `ctl` functions specific to each device driver. Under the hood, these callback functions to the device drivers will use the memory-mapped I/O regions to communicate with devices.



The shaded blocks are the parts that you will modify.

9 Appendix C: The Process Abstraction

9.1 History

The process abstraction in Unix like systems is a critical part of understanding an operating system's inner workings. Historically, processes were the first abstraction over a user program rather than threads. User programs were seen as having a single flow of instruction and resided in separate memory spaces. As computer hardware advanced and provided stronger computing, it became desirable to have multiple streams of instructions (threads) running within a single address space. For Checkpoint 2 we will only consider processes that contain only a single thread

9.2 Overview

The Linux kernel's smallest abstraction over a stream of instructions is a thread. For a UNIX-like system (like ours), each process by default will be spawned around a main thread. This main thread is what is scheduled by the operating system.

A process being scheduled by the kernel to run is just the process's thread being put on the ready list and a process running is just the thread associated with the process running (as the current thread).

On top of owning a thread, each process also owns a virtual memory space. This virtual memory space is represented by a 3-level page table. You can read more about the paging and virtual memory in the appropriate appendix and chapter

In addition, a process should also keep track of files and devices that it's interacting with. Note that we have a common interface for files and devices (io_intf)

The process abstraction provides 3 key guarantees to a user process

- **Protection :** Assures that a malicious process cannot corrupt, fault or deny other user processes
- **Virtualization :** Creates the illusion to a process that resources are boundless and only used by that process i.e. virtual memory, illusion of concurrency with scheduling, etc.
- **Abstraction :** Abstracts away implementation details and provides a common interface to user processes e.g. a user program opening a file is the same code regardless of the file system being backed by an io_lit struct or a vioblk device. They are covered by the same system call

9.3 Context Switching between User-Mode and Supervisor-Mode

Context switching between two privilege levels fundamentally requires two innstructions with two unique properties

1. An instruction to go from a lower privilege level to a higher privilege level
2. An instruction to go from a higher privilege level to a lower privilege level

The ECALL Instruction :
The environment call (`ecall`) instruction will be our way of going from a lower privilege level (user-mode) to a higher privilege level (supervisor-mode). It is meant to be used by user-mode software to make a service request to a higher privilege context

Reading the description for the `ecall` instruction in the RISCV Privileged ISA manual we can see that it causes the following effects

1. `ecall` will trigger an exception which should be delegated to the supervisor-mode trap handler. This exception being triggered from user-mode will have the following effects
   (a) The `sepc` register is loaded with the virtual address of the instruction that triggered the exception (i.e. the `ecall` instruction itself)
   (b) The `scause` register contains the cause code for the exception (the cause code for an environment call from user-mode)
   (c) The `sstatus` register's Supervisor Previous Privilege bits (SPP) will be set to 0 to indicate that it is a trap from user-mode
   (d) The `sstatus` register's Supervisor Previous Interrupt Enable bits (SPIE) will indicate whether or not supervisor-mode interrupts were enabled prior to trapping into supervisor-mode.
   (e) The `sstatus` register's Supervisor Interrupt Enable bits (SIE) will be cleared to disable supervisor-mode interrupts while in supervisor-mode

The SRET Instruction :
The supervisor return (`sret`) instruction is what we'll use to transition from a higher-privilege level (supervisor-mode) to a lower privilege level (user-mode). It is meant to return to user-mode once a user request has been completed

Reading the description for the `sret` instruction in the RISCV Privileged ISA manual we can see that it causes the following effects

1. `sret` will return to a lower privileged mode (user-mode) with the following effects
   (a) The `sstatus` register's SIE bit will be set to SPIE
   (b) The `sstatus` register's SPIE bit will be set to 1
   (c) The `sstatus` register's SPP bits will determine the execution privilege mode after `sret`
   (d) The `sstatus` register's SPP bits will be set to the least-privileged supportd mode (user-mode)
   (e) The `pc` register is set with the value of `sepc`

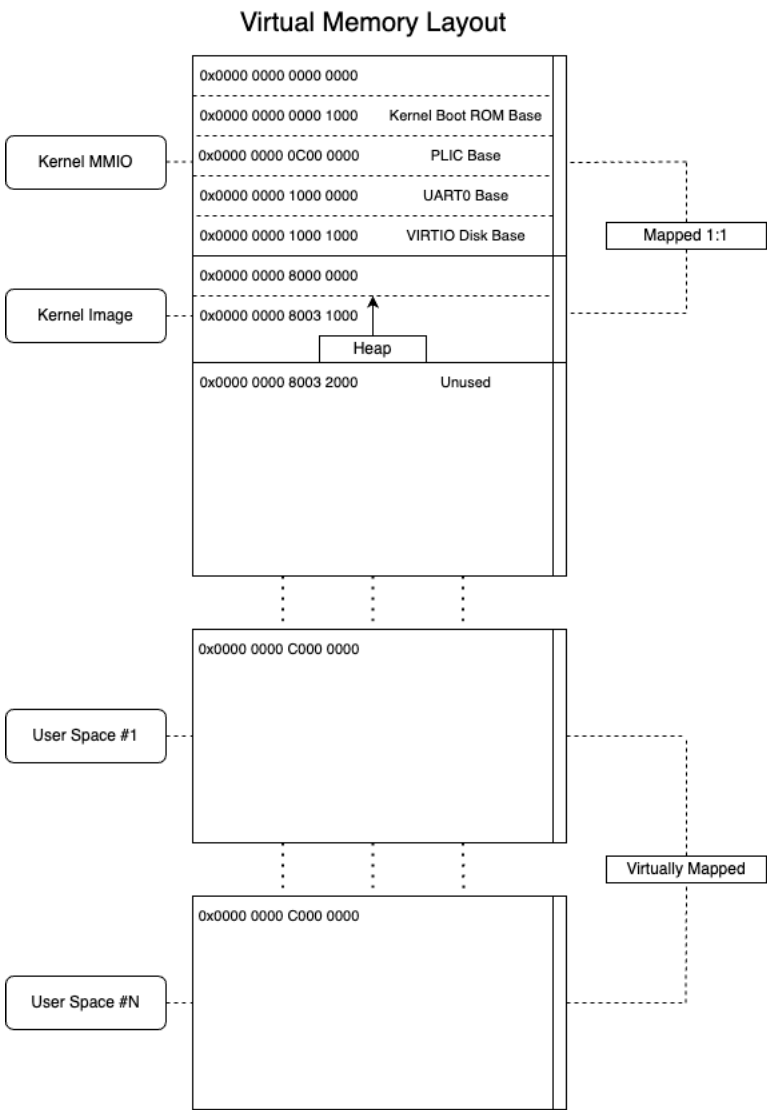
Now that we know the effects of `ecall` and `sret`, we can reason about how to use these instructions during context switches. There are two specific cases to consider.

- Switching to supervisor-mode from user-mode : This type of switch is the most straightforward transition and does not require any preparation. You can just use the `ecall` instruction

- Switching to user-mode from supervisor-mode : Since our kernel initially boots into supervisor-mode, it can be a bit tricky to figure out how to use `sret` to transition into user-mode. This confusion may come up because at first glance `sret` requires a prior user program to have trapped into supervisor-mode (so that the SPP and SPIE bits will be pre-populated with values). However, if this is the first time a user program is being executed there can't be any prior user-mode trap frame and SPP and SPIE cannot be filled in. It is up to the process_exec function to fill in the proper values for SPP and SPIE so that `sret` can properly jump to a userspace function

10 Appendix D: RISC-V Paging

10.1 Illinix Mappings



10.2 Sv39

RISC-V offers a few different paging schemes, each of which offers a different virtual memory size. We are using Sv39 for this MP. Each `struct pte_t` (defined in `memory.c`) describes a page table entry specific to Sv39. You will need to look at the RISC-V privileged docs to learn more about each field: here.

hrefVirtual memory address translation, also called `paging`, is a hardware-level tool that supports process virtualization. When paging is enabled and the machine is in umode, the hardware will translate all user-program memory accesses from virtual to physical addresses.

Multiple programs may be staged to execute at the same address. Memory virtualization allows each process to execute with the idea that they have the entire memory space at their disposal, and ensures that the operating system can manage many programs executing in parallel.

To use the QEMU console (useful in debugging) you can add the following line in `src/kern/Makefile` where you set your `QEMUOPTS`

```
QEMUOPTS += -monitor pty
```

When running `make run-kernel`, your will see another `/dev/pts/N` that you will need to connect to in order to interact with your QEMU console. To view the virtual memory mappings of your system, you can run `info mem` in your qemu console.

11 Appendix F: Symbolic Links

This is a mini-tutorial on how to use `ln` in UNIX. If you want more information, try `man ln`.

Create a symbolic link to a file or directory:

```
ln -s /path/to/file_or_directory path/to/symlink
```

Overwrite an existing symbolic link to point to a different file:

```
ln -sf /path/to/new_file path/to/symlink
```

12 Appendix H: Troubleshooting

12.1 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) hae 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.

12.1.1 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 `shell.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-shell`. This will launch your `shell.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 `kern` 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 `kern` 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 127.0.0.1:26000` line in your `.gdbinit` file to match the port your QEMU instance uses. We've added a line to the `Makefile` that finds an open port, and when you launch QEMU you should see a line that says `-gdb tcp::<port>`. This port is what your QEMU instance exposes for gdb to connect to, and you should modify your `.gdbinit` file to that port (since it tends to stay the same per machine).

12.1.2 Debugging User Programs

If you want to set breakpoints or step through a user program, things are slightly more complicated. Since `shell.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 (`-ggdb`). 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`. It should be located between `USER_START_VMA` and `USER_END_VMA`. You will need to manually tell gdb that this is where the program starts.

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

```
add-symbol-file ../user/bin/hello 0x0000000080100000
```

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`.

12.1.3 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.