5CCS2OSC Operating Systems & Concurrency
InfOS coursework
Stephen Kell
updated 25$^{\text{th}}$ March 2024 (adding additional guidance; tasks are unchanged)

Overview

This is a structured coursework which will occupy you:

- during your lab sessions while the Operating Systems lectures are running (weeks 5 and 7–11 of the semester); and also,
- a little time beyond that—how much depends on which mini-project you choose (see below).

It is marked out of 15, of which 8 marks are for the 'introductory' exercises from the first three lab sessions. It is therefore possible to get a mark of $53\frac{1}{3}\%$, i.e. a comfortable second-class* result, with just these.

The remaining marks are for a mini-project which builds on these initial exercises, and will occupy your final three lab sessions. There are three alternative mini-projects on offer, of varying difficulty. The 'easy' variant is capped at 3 marks (total 11), the 'medium' at 5 marks (total 13), and the 'hard' at the maximum 7 marks (total 15). See §8 for more details on the marking.

All of these exercises/projects are using the InfOS teaching operating system which has been introduced during the live large-group session.

* For avoidance of doubt: the coursework does not have a qualifying mark, so only your combined mark (coursework + exam) determines whether you pass the module.

It is *required* that you submit your work from the shared machine, using the `submit` program. You will make multiple submissions: one for each of the first three lab sessions, and then one for your mini-project. Keep reading for more about this, or see §9 for a summary of how the marking works.

1 First week: introductory tasks

Your first task is to complete the introductory steps as detailed in the week 5 lab materials and accompanying screencast. You'll recall that you had to add a new system call to InfOS which is number 42 in the system call table and always returns your k-number as an integer.

I've pushed some updates to help you, so you should first pull and merge those, e.g. like so.

```
$ cd ~/infos
$ git pull /shared/5CCS2OSC/infos
$ cd ~/infos-user
$ git pull /shared/5CCS2OSC/infos-user
```

You need to submit project 1, repository `infos`, e.g. as follows.

```
$ /shared/5CCS2OSC/submit 1 ~/infos
```

This submits your working directory (here `~/infos`) as diff relative to the base revision in `/shared/5CCS2OSC/infos`. If you add any new source files, be sure to `git add` them!

In the Reading Week section on Keats there is, or will shortly appear, a screencast demonstrating the submission system, although this is also covered in full right here (see §9).

For this and subsequent tasks, it is **strongly recommended** that you work as follows.

- Use the `5CCS2OSC.nms` machine to build and run InfOS, since it already has installed the tools you will need: a C++ compiler, `make`, and QEMU. It also has the submission system. Building InfOS locally on your machine is probably not worth doing (unless you really want to, just for fun). **The submission system will not run on your machine.**

- Keep your files in your Faculty storage space. This is your home directory on `5CCS2OSC.nms` but also it can be mounted locally on your machine via SMB (if you are on campus or the King's VPN) or SFTP (otherwise). Please see the following link.

  `https://apps.nms.kcl.ac.uk/wiki/doku.php?id=computingsupport:services:storage:accessing`

  If using Windows, you might want to use some third-party software to mount (or in Windows-speak 'map') your Faculty storage as a network drive via SFTP, which works even without using the VPN (unlike SMB). To help you with this, there are some additional documents in the Reading Week section of the Keats page. Note that this isn't officially supported by King's, but students have reported it to work. Also note that if you use VSCode it has features that will let you sidestep mounting your storage (see below).

- For working with the code, the choice is yours. You *can* do everything from the terminal, using an editor such as `emacs` or `nano`. If you'd prefer an editor on your own machine, such as VSCode, you can mount your storage space as covered above, or just using the `ssh` integration that VSCode offers, which can make your remote files appear in VSCode. Again there are some tips for doing this on the Reading Week section of the Keats page. Remember that it's your responsibility to make it work if you choose to go this way.

Whatever your set-up, you will still want to keep a terminal logged in to `5CCS2OSC.nms` for building and running InfOS and running the submission system.

2 Second week: page permissions created at load time

Do a `git pull` before you continue working this week, in order to pull a tiny bug fix (mea culpa). This is only necessary if your last pull was before 4pm on Friday 23rd February.

The second week of lectures have taught you about *page tables*. Your subtasks are these.

- Examine the InfOS code that creates the page tables when a new process is started. This will include:
  - `VMA::insert_mapping` in `arch/x86/mm.cpp`
  - `VMA::allocate_virt` in the same file
  - `ElfLoader`'s methods in `fs/exec/elf-loader.cpp`
  - `Process::Process` in `kernel/process.cpp`
- Add a new system call named `readpte()` (or `sys_readpte()`), numbered 41, that takes a 64-bit virtual address as its first and only argument, and returns either zero (if nothing is mapped at that virtual address) or the 64-bit *page table entry* covering that address (if something is). The page table entry should be returned as a `uint64_t` exactly matching the entry that is stored in physical memory.[1] You do not need to handle 'huge pages' (don't worry if you don't know what that is). **To reiterate: your new system call must be system call number 41.** Its name does not actually matter.
- Modify the code so that when a program is loaded, *write permission* is not added in regions of the program memory that do not need it. You will need to study `ElfLoader::ElfLoader` and observe that an ELF program header of kind `PT_LOAD` has the bit-flag `PF_W` set for segments that need to be writable. If this bit is not set, write permission is not needed. However, remember that you can only protect whole pages; some segments are smaller than this.
- (Your modifications won't need to modify all of the files named above, so don't worry if some of them are unchanged. They were mentioned because it is worth your while looking at them, to understand what is going on.)

Various comments have been added and cleanups done to stock InfOS to make this task easier. The program `full-page-text`, which I have added to `infos-user`, could be a useful test case: it is a program that just exits immediately, but it contains a full page of instructions, so definitely has at least one whole page that should be write-protected.

As last time, you should still write one or more of your own user programs that test your changes. You can also make InfOS generate useful debugging printout, using `Log::messagef()`, to help you understand what your code is doing.

As before, exactly what you add to `infos-user` is up to you; you will submit only `infos`.

You need to submit project 2, repository `infos`, e.g. as follows.

```
$ /shared/5CCS2OSC/submit 2 ~/infos
```

[1] Your code does not need to understand this format, but if you're curious, it is easily found by web searching, and is described in comments in the InfOS code. A good diagrammatic reference is here: `https://wiki.osdev.org/File:64-bit_page_tables2.png`.

3 Third week: deferred copy-in (basic demand paging)

The third week of lectures have taught you about *demand paging*, where pages of a process's (virtual) memory image are moved to and from disk, in order to lessen its requirements on physical memory. You will implement a *very limited* form of this technique in InfOS, by *deferred copy-in* of program text and data.

Your solution will be an extension of what you did last week in the ELF loader, when you changed how mappings were created. Now, instead of making some parts of the address space non-writable, you can make them *not initially present*, and copy the program contents *later*, only if needed. You will know that they are needed when a page fault occurs.

The technique you will use is described in this week's lectures (slides 156–157, or roughly 11:45 in the second video). To help you out, the necessary page fault handler has been written for you already. in commit `d19ab4e`. A good way to proceed with this task is roughly as follows.

1. Understand from the lectures the idea of 'storing a cookie' in the page table entry.
2. View the `git` commit `6366c97` (use `git show 6366c97`) that adds the cookie API.
3. Complete the implementation of the cookie API calls (as seen in that commit).
4. Review the ELF loader: notice how cases are broken out in a series of `if`–`else` blocks.
5. Review the page fault handler (`d19ab4e`) to figure out what cookie value to use.
6. Modify the ELF loader to use the cookie API in the relevant case.
7. Fix up the mapping permissions in the page fault handler (see the 'FIXME' comment).

The last two steps are the trickiest. To make the ELF loader defer allocation in the relevant case(s), these must skip the `allocate_virt()` and instead do `reserve_virt_unbacked()`, followed by setting the cookie to the right value. To fix up the mapping permissions you must fill in the small gap labelled 'FIXME' in the page fault handler. The code will still function without it—but effectively undoing your hard work of last week, on write-protecting certain pages! Write protection must now also be deferred to the page fault handler.

Some test programs are provided in `infos-user` (and week 3's auto-`feedback`), as follows.

- `full-page-text` (seen last week): one page of text should get the deferred-copy behaviour
- `easy-page-data`: one page of data should get the deferred-copy behaviour
- `boundary-page-data`: one page of data, in the middle of the data segment, should get the deferred-copy behaviour. This is a 'boundary condition' case. Since the segment starts on a non-page-aligned address, its beginning and end should be copied eagerly: they don't fill their containing page. The `if`–`else` blocks in the ELF loader are there to split out the cases where the segments of an ELF file do not fill a whole number of pages. If either or both of `offset` and `filesz` are not a multiple of the page size, the deferred approach will not work and you should fall back on the old way for these, copying at load time.

This part is harder than the ones before, so do persevere. Use all the available tools to test what you're doing. Use your PTE print-out function from the previous week: you can use it to check that your page table changes in this task are taking effect. The command `readelf -l` can show you these headers in an ELF file. The `offset` and `filesz` fields of the program header tell you where in the file a segment starts and how many bytes it is.

You need to submit project 3, repository `infos`, e.g. as follows.

```
$ /shared/5CCS2OSC/submit 3 ~/infos
```

4 Easy mini-project

If you choose the easy mini-project, your job is to address the problem commented in the `kernel/syscall.cpp` InfOS source file, about validating pointers that are passed in to the kernel.

```
// TODO: Validate 'buffer' etc...
```

These comments are referring to how system calls like `read()` accept pointers from user code, but do not check that they point into the user's memory area! They might point into the kernel mapping. This could potentially wreak havoc if the kernel blithely copies file data into them as the user process requests with a `read()`. It could potentially leak confidential information, if the kernel blithely copies kernel data into a file as requested by `write()`.

Not all such problems in the file are commented. Your job is to fix the problem for **all** system calls that suffer it, whether or not there is a comment. So you will have to understand what the other system calls are doing.

If a process attempts to read or write outside its user-accessible memory mappings, it should be terminated in the same way as if a page fault occurred. You can see how to do this by reading the page fault handler from the previous week's task.

As in earlier weeks, try to work in a test-driven way. You could write a `scribble` program that uses the unchecked system calls to read or write a new global variable of known value, that you have added to the kernel. To test whether scribbling has worked, you could write a fresh system call that leaks this value.

You should submit project 4, repository `infos`, e.g. as follows.

```
$ /shared/5CCS2OSC/submit 4 ~/infos
```

5 Medium mini-project

If you choose the medium-difficulty mini-project, your job is to complete the easy mini-project and extend it to generate a more informative error report. You will have noticed that on a page fault, when the process is terminated, InfOS prints out the current program counter (`rip`). Your job is to add a *stack trace* to this message, showing in more detail what the user process was doing when it crashed. InfOS includes a `dump_stack()` method that is currently unimplemented and unused. You should implement it and call it from the termination path, so that:

- it iterates over the stack, and *for every active call on the user stack*, calls the helper method `dump_one_user_frame()` (this helper is provided for you);
- to each of these calls you must pass the user frame's program counter, stack pointer and the name of the *function symbol* that the program counter points within;
- these calls must happen in order starting with the most recent call (the one which caused the error) and ending with the earliest active call on the stack.

In `gdb`, the `bt` command prints a stack trace. To have InfOS do the same, it will have to understand the layout of a stack that uses *frame pointers*. A frame pointer is a pointer to the logical base of a stack frame. From the base of a frame, it is possible to discover the base of the next frame, and so on. The `rbp` register holds the current frame pointer. It is initialized from the stack pointer at the start of each function, after saving its old value.

```
push   %rbp
mov    %rsp,%rbp
```

Since the `call` instruction itself has just pushed the caller's program counter (`rip`), we know that the old frame pointer and program counter are stored just *above* the current `rbp` (the stack grows downwards). This will be covered in the fourth week of lecture material.

To get the function name, you will also have to understand *symbol tables* in ELF files. As you discovered during the second week's task, the ELF file contains much useful information that the kernel can use. You will have to discover the ELF symbol table, its associated *string table*, and how to discover the name of a given function (an offset in the string table).

This is a rather unrealistic assignment because real operating systems avoid kernel code that runs complex algorithms on data structures in user memory—especially the user's stack! To be secure, any routine that works on user data has to be written very defensively (as the Easy project also shows). For example, you should take care that your stack walker cannot get stuck in an endless loop. A naïve implementation might loop endlessly if it is fed a corrupt or malicious stack (e.g. imagine a cycle of frame pointers!). Yours should be robust to this.

This is project 5. If you are attempting this project, you only need to submit project 5, not project 4. However, you may also submit project 4 if you like. In either case, you will get the maximum of the two marks (since project 5 includes the code for project 4).

To submit project 5, use a command like the following.

```
$ /shared/5CCS2OSC/submit 5 ~/infos
```

6 Hard mini-project

This mini-project addresses the unrealistic aspect of the Medium task, by implementing a very limited form of *signal handling*. Rather than printing a stack trace and then terminating the process, you will give the user program the possibility to recover from the fault.

You should complete the Medium mini-project first, which will give you a good understanding of what is happening on the stack.

Next, you must add system call number 40 with the following signature.

```
void *register_fault_handler(void (*handler)(void *addr, uint64_t status));
```

This must remember the address of a handler routine in the user process. (The return value of this call is not that important, but one sensible thing to return would be the previous registered fault handler, or `NULL` if there wasn't one.) When a handler is registered, a page fault or out-of-bounds access should not immediately terminate the process. Instead, the kernel will manipulate the thread's stack and register context such that when the thread next runs, execution continues in the registered handler. The handler receives `addr` as the faulting address and a further `status` argument (optional; you may or may not want to use this for anything, but in a realistic scenario it'd probably be a special value meaning 'page fault', to allow for handling other kinds of event). When the handler returns, it must make a system call back into the kernel. For this you will have to add one further system call, number 39.

```
void exit_fault_handler(uint64_t handled);
```

What happens next depends on the value of `handled`. If zero is passed ('not handled'), the process will terminate. If non-zero is passed ('handled'), the original execution will continue.

To really understand this task, ask yourself: *from where* will it continue? By default, it will continue from the faulting instruction. In most cases, this would create an infinite loop of faults. Therefore, it is essential that **the handler may modify the saved context** to influence where the program resumes and what its register values will be. The normal way to do this is to push the saved thread context onto the stack before running the handler. The handler can then get a pointer to it (a fixed offset from the stack pointer on entry) and manipulate the saved registers. When the handler returns, this state is copied back, replacing the original saved context for the faulting thread. For example, the handler might advance the program counter to move beyond the faulting instruction, after emulating whatever the instruction was trying to do. This updated program counter will be what is restored when the faulting thread is next scheduled.

A possible test case for this project could to port the stack trace (as created for the Medium project) in user space, e.g. as a routine in the library (`infos-user/lib`) which could be called from a handler (which then terminates the process).

This is project 6. If you are attempting this project, you only need to submit project 6, not projects 4 or 5. However, you may also submit the earlier projects 4 or 5 if you like. In any case, you will get the maximum of the three marks (since your project 6 should behave like project 5 if a handler is not registered, and project 5 includes an implementation of project 4).

To submit project 6, use a command like the following.

```
$ /shared/5CCS2OSC/submit 6 ~/infos
```

7 How to do it (Updated!)

Read the code, search it, understand it. Expect to spend relatively little time writing code, and much longer reading and understanding it (including by active means, e.g. trying stuff out). Except the Medium and Hard mini-projects, for each task your solution will consist of only a few lines of code (probably between 4 and 30) but getting those few lines right will take understanding! Unlike many coding tasks, you will not make much progress by Googling 'how do I do X using technology Y?'. StackOverflow contains little about InfOS! (It can help you with C++, though.) Use `grep`, which is an essential tool If you see an identifer and want to see where it's used or where it's defined, you can `grep` the source code for it.

Actually run InfOS—don't just rely on the `feedback` program! When you run InfOS you're in control of what it is (or should be) doing, can see the debugging messages, and so on. (You can still get these logs when using `feedback`, as the log is stored in the temporary directory, as the messages explain. However, overall it's still a bit harder to see what's happening.)

Read error messages, selectively revert changes. If you see InfOS print out an error message because your changes sent it into a bad state, use `grep` to find where in the code that message came from. Overall, `grep` is your single biggest tool for navigating this large codebase. When you write code, make liberal use of `sys.messagef()` to add to the debugging print-out. Be systematic: if your changes break things, try selectively reverting them (use `git diff`).

Be active, be disciplined. Write notes, write comments in your code; keep your code understandable. Write tests. Try things out. Be deductive; don't just guess!

Learn to read page fault messages. InfOS's debug log prints when a page fault occurs. You should read and understand these messages! It helps to know that *user virtual addresses* are small-ish numbers (e.g. `401000`), while *kernel virtual addresses* are big numbers (i.e. beginning `ffff...`). (Note that *physical addresses* are rarely seen or used!) Let's see some examples.

```
warning: *** PAGE FAULT @ vaddr=0x0 rip=0xffffffff80123456 proc=kern
```

This means your kernel code has a null pointer bug. How did I know? Because the fault is an access to address zero (null) and `rip` is up in the `ffff` range (so it's kernel code that caused the fault; `proc=kern` is also a giveaway).

```
warning: *** PAGE FAULT @ vaddr=0x401012 rip=0x401012 proc=user
```

This means a user program is faulting during instruction fetch, in a way that may or may not be intended. How did I know? Because the access address is small (`401012`) and the program counter `rip` is the same value, i.e. the fault happens as the CPU tries to fetch the instruction. The fault might be intended in week 3, if the load of that user page was deferred. Check the handler is reading at the right offset in the ELF file! Later debug messages show this.

```
warning: *** PAGE FAULT @ vaddr=0x402008 rip=0x401012 proc=user
```

This means a user program is faulting at an instruction that reads or writes data. The instruction was fetched fine; it's the data access that's the prob;lem. How did I know? Again the addresses are small, but the two addresses are different. Again this may or may not be intended: either because loading of the data was deferred (week 3 again) or because you're running a test case that writes some read-only data and so should be trapped and killed (weeks 2 or 3).

Common pitfalls (New!)

Weeks 1 or 2: forgetting to plumb in your system call. If your system call seems to be returning -1 (`0xffffffffffffffff` as a 64-bit unsigned integer), it might be that you forgot to register it in the system call table. You will need to do this in both weeks 1 (system call number 42) and week 2 (system call number 41).

Week 2: confusing a page table entry with a physical address. The page table entry contains part of the physical address (frame number) but it also contains other stuff (flags). It's the page table entry that we need.

Week 2: namespace problems. To get the raw 64-bit value of the page table entry, you could do something like `pte->bits()`, but from `syscall.cpp` you'll get an 'incomplete type' compiler error unless you add an `#include` of `<arch/x86/vma.h>` at the top of the file. Alternatively you can skip the `bits()` call and use the nasty C-style cast method (`*(uint64_t *)ptr`).

Week 3: over-thinking `reserve_virt_unbacked()`. If you followed the screencast to get started, the `reserve_virt_unbacked()` doesn't have to do very much: it can just call your refactored helper that ensures a page table entry exists. Since `reserve_virt_unbacked()` takes an argument for the number of pages, you should really use a loop to do this the right number of times. You could even skip that since `nr_pages` is currently always 1.

Week 3: missing cases in the ELF loader. The ELF loader has already been refactored for you so that it walks page-by-page over the segment, splitting out the important cases. It's best to work with this structure. Beware the `else` block that does a `continue` to skip the copying. This ensures that after `filesz` bytes, the segment is implicitly zeroed up to its total `memsz`. *Be sure that you are still doing an `allocate_virt()` along this path.* Otherwise programs trying to access these zeroes will fault. Finding no cookie, the handler will kill the process or hang.

Week 3: getting confused by week 2 test failures. Right now, beware that the week 2 `feedback` auto-test cases no longer pass with a completed week 3. (I should fix this but it's tricky!) If you submitted your week 2 before starting week 3, there's no problem. Otherwise, use `git` to `stash` or `commit` your week 3, then revert to your week 2 and submit that.

Week 3: failing lookup of a non-present page table entry. Accessing a cookie relies on getting at a non-'present' page table entry. So check that the functions you're using to get a pointer to the page table entry actually do correctly return a pointer even to a non-present entry. If you use `get_mapping()` or the helper that the week 2 screencast suggests refactoring out of it (called `find_pte()`), this does *not* support returning a non-present PTE (it returns null). It's easy to change so that it does, but check that you don't break `get_mapping()`: it should always return false if there's no *present* PTE. For setting the cookie, you may also have to create page tables. You could do what the week 3 screencast suggests (from 20:30): a similar refactoring to week 2's but on `insert_mapping()`. This creates the page tables, and also happily lets you get a pointer to the raw page table entry (see `pte` in the code visible right at the end of the video). It should return non-null even when the entry it points is not 'present'.

Week 3: passing ELF flags where page table flags are needed. The page fault handler has a comment telling you how it expects the offset and flags to have been combined into a single 32-bit number. Beware that ELF files have different flags from page tables! The ELF bit-flag meaning 'memory is writable' (`PF_W`) need not be the same as the x86 page tables' bit-flag (`PTE_WRITABLE`) so you should explicitly test and translate. (It turns out you *can* get away without worrying about this, but it makes for confusing code.)

Slightly more advanced how-to hints (New / updated!)

Learn to read the output of '`readelf`'. For weeks 2, 3 and 5, you will need to know a little bit about the ELF file format. For weeks 2 and 3, you will find yourself reading the ELF loader code and how it processes a `LOAD` (segment) program header. For week 5, some additional notes are provided in a new appendix to this document (§B). In both cases, the `readelf` utility (on the shared machine, not within InfOS) is useful. You can see the program headers of a file using `readelf -l` (or, on a wide enough terminal, `readelf -Wl` may be more readable), adding the name of the file you want to inspect.

Try the debugger `gdb`. If you do '`run-infos serial-gdb`' you will get instructions for attaching `gdb`. Some useful `gdb` commands are `continue`, `break`, `next`, `step`, `print`, the use of Ctrl+C to interrupt execution, `backtrace` (or just `bt`), and `frame` (or `up` and `down`), `detach` and `quit`. Use `tui enable` to turn on a friendlier UI. It still does not look *so* friendly, but its online help is not bad. Within `gdb` try '`help <command>`', or if you're not sure which command you need, '`apropos <keyword>`'. You can debug both the kernel and the user applications it runs; use `add-symbol-file` to tell `gdb` about the user program binary.

If you modify a `Makefile`, beware build skew. You do not need to modify a Makefile, but some students do this. (It's not recommended.) Although `make` is pretty good (if used correctly) at figuring out when a file needs to be recompiled, it is fooled if you edit the rules in the Makefile itself: it won't rebuild the files automatically using these new rules. Doing '`make clean; make`' is a crude but sometimes useful way to force a rebuild. If you want to see in full what commands are being run during the build process, use the following command ('q' is for 'quiet'; here it is redefined to be empty so that quietening does not take effect).

```
make q=''
```

Understand the kernel's virtual address space. Kernel code mostly deals with kernel virtual addresses (often abbreviated `kva` in helper functions). Some low-level code dealing with page tables works with physical addresses (abbreviated `pa`), but there is little you can do with a physical address directly, because address translation is always enabled in the CPU. Instead, a region of the kernel's virtual address space contains a mapping of the first 4GB of physical memory, so it can be addressed virtually (these are abbreviated `vpa`). There is also an array of 'frame descriptor' records, one per frame of physical memory, and these are abbreviated `pfdescr`. Finally, a quirk is that *the kernel's own virtual address mappings are also present in every user process*, but are not accessible by user code because their page table entries lack the 'user-accessible' bit. This is an optimisation meaning that when handling a system call, it is not necessary to switch address spaces. A comment in `arch/x86/mm.cpp` explains the memory layout in more detail.

The most important bit

Get help, ask your questions. If you're stuck, don't just beat your head against the desk (proverbially, or really). Use the Discussion forum, use your TAs, use my office hours. It's what they're there for. Just remember: when you ask for help, be specific! Articulate what you're stuck on and what your uncertainties are.

8 How it's marked

This coursework will account for 15% of your module mark.

For both the introductory exercises and the mini-project, your work will be subjected to a series of automated tests, and these results will account for most of the mark. Your code will also be eyeballed briefly (but not tested by hand in any detail).

The introductory tasks are worth (in order) 2, 3 and 3 marks. **Partial credit** will be given for a meaningful attempt at the task, provided that it passes one or more functional tests. For example, in week 3, if you do not correctly handle the 'boundary condition' as mentioned, but your code otherwise behaves correctly, you might get 2 marks out of 3. In week 2, if you write-protect some pages but not others that should be protected (or do protect some that shouldn't be!) you might well get 2 marks out of 3. Partial credit will be given at no finer granularity than half a mark. Exactly how much functionality must work for a given level of credit is a matter for my academic judgement.

You must make a separate submission for each of the first three weeks.

The remaining marks are for the mini-project. As mentioned before, the mini-projects are worth respectively 3, 5 and 7 marks. Partial credit will be given on the same basis as before, except that for the medium and hard projects, one mark of the credit is reserved for a solution that demonstrates initiative and good taste beyond simply passing functional tests. This might be comments, test cases, appropriate console messages, error handling, or extensions that make the OS's behaviour more useful in practice.

You must write your own code. It's good – encouraged! – to discuss your task with peers, to help each other debug problems, and so on. TAs will also help you if you ask, and you should ask on the discussion forum. However, (repeating myself) **you must write your own code**. Your code will be subjected to plagiarism detection. Attempts at disguising plagiarism, such as by making superficial changes to code you didn't write, will still be detected and are an especially serious offence under the College's plagiarism regulations.

9 How to submit

You make a submission by running the `/shared/5CCS2OSC/submit` command, giving two arguments: a project number, and a directory. Do this for each of the first three weeks, and for your chosen mini-project.

If you followed the instructions given for week 1, the directory name will be `~/infos`. The submitted directory must contain a complete `git` checkout of `infos`, descended from the base revision in `/shared/5CCS2OSC/infos`. Note there are two distinct `git` repositories: `infos` (the kernel) and `infos-user` (the user-land). It is only `infos` that you submit.

If you did the development in the obvious way, then all the above should be the case without your needing to do anything. It is a good idea to commit your changes using `git commit`.

If you submit a project more than once, your most recent submission will be the one marked. You can submit each project as many times as you like. The deadline is the same for all submissions: 4pm on (**updated**) Wednesday 3rd April.

You must not publish your code to GitHub or in any other public place, nor share it with anyone. If you do, you might be held responsible for academic misconduct when *others'* submissions are found to be overly similar to yours.

Note that use of `git` does not imply use of GitHub! If you follow the instructions above, you will find yourself staying within the rules.

(Private GitHub repositories are fine, although not a lot is gained by using one. You can push/pull directly to/from a repository on your Faculty storage space, using the `ssh://` syntax with `git`.)

A Stephen's changes

This section lists the files that I changed in my solutions for each of the first three weeks. As you can see, not many files are involved and they are all ones that have been called out in the brief and/or the screencasts. So this is just to reassure you that there is nothing sneaky going on and you will not have to go roving to unexplored corners of the codebase.

Week 1

Files changed:

```
include/infos/kernel/syscall.h
kernel/syscall.cpp
```

Week 2

Files changed:

```
arch/x86/mm.cpp
fs/exec/elf-loader.cpp
include/infos/mm/vma.h
kernel/syscall.cpp
```

Week 3

Files changed:

```
arch/x86/mm.cpp
fs/exec/elf-loader.cpp
include/infos/mm/vma.h
```

B ELF files and their symbol tables

In weeks 2 and 3 you have had to inspect the program headers of an ELF file to know whether some memory needed to be mapped read-only or read-write. In the Medium mini-project you will need to dig more into ELF files in order to find the *symbol* information, which can (usually!) tell you the name of a function or global variable.

The ELF format is documented publicly[2] but this section gives you a summary of the necessary parts.

[2]e.g. here: `https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.intro.html`.

An ELF file consists of a number of *sections* and *header tables*, which are linked together using *file offsets* in a manner a bit like pointers. The *ELF header* lives at the start of the file, but everything else can live at (almost) any offset. A very simple example ELF file looks as follows. The program consists of 7 bytes of instructions and no data at all. The numbers down the left are the file offset; each row shows 16 bytes, in hexadecimal. You can get a similar listing of any file using the `hd` or `hexdump -C` command.

```
offset    data as hex bytes                                 data as ASCII      notes
--------------------------------------------------------------------------------------------------------------------------
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............| ELF header (64 bytes)
00000010  02 00 3e 00 01 00 00 00  00 00 40 00 00 00 00 00  |..>.......@.....|                   entry pt 0x400000
00000020  40 00 00 00 00 00 00 00  b8 10 00 00 00 00 00 00  |@...............| p.hdr offset 0x40, s.hdr offset 0x10b8
00000030  00 00 00 00 40 00 38 00  01 00 40 00 05 00 04 00  |....@.8...@.....| 1 phdr of size 0x38, 5 shdrs of size 0x40
00000040  01 00 00 00 05 00 00 00  00 10 00 00 00 00 00 00  |................| Program headers (here: 1x56 bytes,
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....| LOAD, vaddr and paddr 0x400000,
00000060  07 00 00 00 00 00 00 00  07 00 00 00 00 00 00 00  |................| filesz and memsz 7,
00000070  00 10 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................| offs 0x1000 |
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*                                                                              padding, all zeroes
00001000  b8 02 00 00 00 cd 81 00  00 00 00 00 00 00 00 00  |................| text! ending 'cd 81', then symbol table,
00001010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................| here 5x 24byte entries, first always zeroed
00001020  00 00 00 00 03 00 01 00  00 00 40 00 00 00 00 00  |..........@.....| name=0, value=0x400000,
00001030  00 00 00 00 00 00 00 00  01 00 00 00 10 00 01 00  |................| size=0;        name=1,
00001040  00 00 40 00 00 00 00 00  00 00 00 00 00 00 00 00  |..@.............| value=0x400000, size=0;
00001050  08 00 00 00 10 00 01 00  00 10 40 00 00 00 00 00  |..........@.....| name=8, value=0x401000,
00001060  00 00 00 00 00 00 00 00  0f 00 00 00 10 00 01 00  |................| size=0;         name=0xf,
00001070  00 10 40 00 00 00 00 00  00 00 00 00 00 00 00 00  |..@.............| value=0x401000, size=0 |
00001080  00 5f 73 74 61 72 74 00  5f 65 64 61 74 61 00 5f  |._start._edata._| Strings for symbol table: "name" above
00001090  65 6e 64 00 00 2e 73 79  6d 74 61 62 00 2e 73 74  |end...symtab..st| is an offset here. | Strings for
000010a0  72 74 61 62 00 2e 73 68  73 74 72 74 61 62 00 2e  |rtab..shstrtab..| section header table, same
000010b0  74 65 78 74 00 00 00 00  00 00 00 00 00 00 00 00  |text............| idea        |section hdrs
000010c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................| 5x 64 bytes, [0] first entry is
000010d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................| all zeroes...
000010e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000010f0  00 00 00 00 00 00 00 00  1b 00 00 00 01 00 00 00  |................|             |[1].text header
00001100  06 00 00 00 00 00 00 00  00 00 40 00 00 00 00 00  |..........@.....|
00001110  00 10 00 00 00 00 00 00  07 00 00 00 00 00 00 00  |................| file offset 0x1000, size 7
00001120  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
00001130  00 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00  |................|             |[2].symtab header
00001140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001150  08 10 00 00 00 00 00 00  78 00 00 00 00 00 00 00  |........x.......| file offset 0x1008, size 0x78
00001160  03 00 00 00 02 00 00 00  08 00 00 00 00 00 00 00  |................| 'link' section 3, 'info' section 2
00001170  18 00 00 00 00 00 00 00  09 00 00 00 03 00 00 00  |................|             |[3].strtab header
00001180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001190  80 10 00 00 00 00 00 00  14 00 00 00 00 00 00 00  |................| file offset 0x1080, size 0x14
000011a0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
000011b0  00 00 00 00 00 00 00 00  11 00 00 00 03 00 00 00  |................|             |[4].shstrtab header
000011c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000011d0  94 10 00 00 00 00 00 00  21 00 00 00 00 00 00 00  |........!.......| file offset 0x1094, size 0x21
000011e0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
```

To find the symbol table in an ELF file, you must (try it now on the above!):

1. from the ELF header, get the *section header table* offset, entry size and number of entries;
2. read the section headers (i.e. the entries of the section header table) one by one, looking for the symbol table i.e. the one with type `SHT_SYMTAB`;
3. use `sh_link` in the symbol table's section header to find the *string table*'s section header;
4. read the symbol table entries, which have type `Elf64_Sym`, from the range of the file described by the `sh_offset` and `sh_size` of the symbol table section header;
5. to get the name of a symbol in the table, use its `sh_name` as an index into the string table, whose `sh_offset` and `sh_size` are found in *its* section header.

Given an address in the program, you might want to do a linear search of the table for a symbol whose range (`st_value` is the starting address, `st_size` the length) covers that address. Be prepared for this search to fail, since many addresses are not within any symbol's range.

Getting ELF definitions into InfOS

Although InfOS contains some definitions describing the ELF file format, it does not contain definitions for the ELF symbol (`Elf64_Sym`) and section header (`Elf64_Shdr`) data structures. You could write them yourself from the ELF spec (easily found online) but an easier way is to get them from the system headers, specifically `/usr/include/elf.h`.

Normally InfOS does not use system headers: host system libraries wouldn't work in or on InfOS. However, since this header only includes type definitions, not code or data declarations, it is fine to use it.

Unfortunately you can't quite just `#include </usr/include/elf.h>` because it is not completely self-contained. So you can either copy and paste the relevant parts of `elf.h`, or from my `infos`, directory the following commands create enough dummy header files for `elf.h` to be included successfully.

```
echo -e '#define __BEGIN_DECLS\n#define __END_DECLS' > include/features.h
touch include/stdint.h
mkdir include/bits
touch include/bits/auxv.h
```

(Or you could just copy the definitions from `elf.h` into a new file, editing out unnecessary content.)

Note also that in general, copying and pasting code is not to be done lightly! It's fine to do here because the ELF definitions are part of a freely-available published standard, so are intended to be used in this way.