Lab 3 - Part A : User Environments and Exception Handling

Sriram V - CS11B058

Exercise 1

- Modified `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array.
- The added code is very similar to the allocation and mapping of `pages` in the previous lab. This is because although the previous lab dealt with the management of pages and the current one with user environments, both need to be kept track of by the kernel, and this book-keeping for both is done by different `structs`.
- `NENV` instances (pointed to by `envs`) of the `Env` structure are initialised with the help of `boot_alloc()` at `line 162`.
- `envs` is also mapped to `UENVS` (read-only), which allows user processes to access this data without modifying it. This is done by means of the `boot_map_region()` function, which allocates the necessary pages and adds entries in the Page Directory. (`line 197` in `kern/pmap.c`)
- The kernel however has read-write access because it accesses `envs` at a higher virtual address that has RW permissions for the kernel, but cannot be accessed by the user. (Done by the mapping at `line 222`).
- Running `make qemu` after this gives a `check_kern_pgdir()` success message.

Exercise 2

- Now that the code to create `envs` is done, we fill in the code to actually create and run a user environment. Code to update `envs` (to help the kernel keep track of user environments) is also added in. All this is written in the `kern/env.c` file.
- The `cprintf` verb `%e` is used in the `panic()` calls to print out descriptions corresponding to error codes.

1. `env_init()`:
   - This function initialises the `Env` structures in the `envs` array and adds it to the `env_free_list` linked list. (`lines 114 - 134`)
   - `env_free_list` is initially set to `NULL`.
   - Since the environments in the free list are to be in the same order as they are in the `envs` array, `envs` is looped over in reverse order, so that when an element of `envs` is being added to the list, its `env_link` can be made to point to the element that is next in the free list array.
   - The status is set to `ENV_FREE` for each of the elements, and their runs are initialised to `0`.
   - Finally `env_init_percpu()` is called, which configures the segmentation hardware with separate segments for the kernel(PL 0) and user(PL 3).
2. `env_setup_vm()`:
   - In this function, the kernel virtual memory layout for an environment is initialised. (`lines 167 - 206`)
   - A page directory is allocated using `page_alloc()` (initialised with all bytes equal to `\0`).
   - The kernel's page directory is copied into the environment's page directory. This is because the virtual address space of all environments is identical above `UTOP`, except at `UVPT`. This virual axdress space contains the kernel. Hence, the kernel's virtual address mappings are copied in.
   - The reference counter for the page directory is incremented as well, since it is inserted into itself, at `UVPT`.
3. `region_alloc()`:
   - This function allocates `len` bytes of physical memory for an environment, and also mapes it at a virtual address that is in the environment's address space. (`lines 278 - 297`)

- As required by the implementation of the function, the passed `va` and `va+len` are rounded down and rounded up respectively, so that the usage of this function becomes simpler.
- Physical memory for this environment is then allocated in the form of pages (using `page_alloc()`) and inserting them into the `env_pgdir` (using `page_insert()`) with the relevant permissions (user and kernel RW).

4. `load_icode():`

   - Since we don't have a filesystem as yet, the kernel is set up to load static binary images that are embedded within the kernel itself. These binaries are in the `ELF` format.
   - The `load_icode()` function loads a passed binary into the passed environment, and sets up its stack and processor flags
   - A good place for reference is the `boot/main.c` file, since the loading of a binary into an environment is very similar to the loading of the kernel into the memory by the bootloader.
   - The binary is first tested to be of a valid `ELF` format, with the help of the `ELF_MAGIC` value.
   - `ph` is then set to point to the start of the program header structures in the binary.
   - The page directory is also set to the passed environment's (using `lcr3()`) page directory (even though the environment is not running, but is simply being loaded into), so that the virtual address mappings for this environment are set up, thus making the loading of the program segments at the relevant virtual addresses much simpler.
   - For each loadable program segment (specified by a `p_type` of `ELF_PROG_LOAD`), a region of physical memory is allocated, and the binary is moved into this physical space using `memmove()`.
   - The remaining memory is set to `0` using `memset()`.
   - The file size must be less than the memory allocated for a particular loadable program segment, otherwise an error must be thrown.
   - The environment's `EIP` is set to the entry point for the code, mentioned in the binary's `e_entry` variable. The EIP is stored in the Trapframe associated with an environment.
   - Finally, physical memory for the stack (for the loaded binary) is allocated with `region_alloc()`, of size `PGSIZE`.
   - The page directory to be used is also forced back to the kernel's page directory, using the `lcr3()` function.

5. `env_create():`

   - Used to create a new environment, load in a passed binary, and set the environment type to the passed type.
   - The new environment is created using the `env_alloc()` function. The parent ID of the new environment is set to `0`.
   - Failure to allocate a new environment results in a panic.
   - Otherwise, the binary is loaded into the newly created environment using `load_icode()`
   - The `env_type` variable for the environment is set to the passed variable, `type`.

6. `env_run():`

   - This function enables a context switch from the current environment to the passed environment.
   - If a current environment (`curenv`) exists (not the first call to this function), then set its status from `ENV_RUNNING` to `ENV_RUNNABLE`.
   - `curenv` is set to the passed environment, and its status is set to `ENV_RUNNING`.
   - The number of runs for this environment is incremented, and `lcr3()` is used to switch to its address space, by forcing the use of its page directory.
   - A call to `env_pop_tf()` is made with the passed environment's Trapframe as an argument.
   - This restores the register values in the Trapframe with the `iret` instruction, exits the kernel, and starts executing the environment's code.

Exercise 3

- Read the chapter on 'Exceptions and Interrupts' in the 80386 Programmer's Manual and the IA-32 Developer's Manual, and understood the differences between exceptions and interrupts with regard to Intel.

Exercise 4

- When an interrupt/exception occurs, control is transferred to the kernel.
- However, in order to ensure that the protected control transfers are actually *protected*, the processor's interrupt/exception mechanism is designed so that the code running currently when the interrupt or exception occurs doesn't get to choose arbitrarily where and how the kernel is to be entered.
- Hence, we need to specify entry points into the kernel for the various exceptions. In this Exercise, we only handle the processor exceptions.
- For the 20 processor defined exceptions (given by `#define`s in `inc/trap.h`), we add an entry point by using either the `TRAPHANDLER()` or `TRAPHANDLER_NOEC()` macro, depending on whether the exception has an associated error code or not respectively. Whether an exception pushes an error code on the stack can be found out by looking at the Error Code Summary in the 80386 Programmer's Manual.
- For a particular exception, we can specify any name that we wish as the first parameter of the appropriate macro, and this enables us to pass the exception handling routine as a function pointer (during IDT setup), by using this name.
- Certain exceptions are reserved by Intel (9 and 15 in the first 20 exceptions), and since they will never be generated by the processor we can handle them in any way as we desire.
- The macro creates the exception handler routine for an excepion. The exception handler routine pushes a trap number and either an error code or a zero onto the stack, and jumps to `_alltraps`.
- The code for `_alltraps` in `kern/trapentry.S` pushes the DS and ES segment descriptors onto the stack, which is followed by pushing in all the 8 registers. It is done in this order because it fits in nicely with the layout of `strcut Trapframe`. (Trapframe registers on the top, followed by `ES`, `DS`, the trap number, error code etc.)
- Now the kernel's data segment descriptor (at `GD_KD`) is loaded into `DS` and `ES`.
- `ESP` is pushed onto the stack, to pass the built up Trapframe as an argument to `trap`. (Passing `ESP` means that a pointer to the Trapframe is being passed.)
- A `call` to `trap` is made.
- It doesn't look like `trap` can return, since a dispatch is made depending on what type of trap occured, which is then followed by a call to `env_run()`, which we know exits the kernel and enters the user environment.

Challenge:

- I got rid of the similar looking code in `kern/trapentry.S` and `kern/trap.c`, by making use of an array, and adding the function names to this array as and when they are created.
- I referred to the xv6 source code, which makes use of a `vectors` array for this purpose.
- In `trapentry.S`, I defined the array with the code:

```
.data
.globl thandlers
thandlers:
```

- I also modified the two macros by adding a `.text` in the beginning (to specify that the following code is part of the program's text portion) and added

```
.data;
.long name
```

  at the end (to specify that this bit of code is part of the program's data section).

- Thus, this builds an array of function pointers to interrupt handler routines.
- By specifying `extern int thandlers[];` at `line 61` in `kern/trap.c`, we can use `thandlers[i]` in a `for` loop (that has `SETGATE()` as the body), instead of having multiple lines of `SETGATE()` calls.
- Also, the one `extern` array declaration helps us to get rid of the multiple `extern void NAME();` declarations that were required earlier.

Questions

1. **What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)**

- The purpose of having an individual handler function for each exception/interrupt is so that the OS can have different values on the stack for different interrupts/exceptions.
- For those that take an error code, their error code may be pushed onto the stack. For those that don't, a zero is pushed. Apart from this, different values may also be pushed onto the stack specifically for different interrupts/exceptions.
- If all exceptions/interrupts were delivered to the same handler, then distinct trap numbers wouldn't be pushed onto the stack before calling `trap`. Thus, the `trap()` function wouldn't be able to distinguish between different exceptions/interrupts. The hardware too is incapable of distinguishing between trap errors.
- Without this distinguishing feature present in the kernel code, if we wish to handle different interrupts/exceptions, we would have to allow the user to invoke the exception/interrupt, and this is a serious security issue.

2. **Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (`trap 13`), but `softint`'s code says `int $14`. Why should this produce interrupt vector 13? What happens if the kernel actually allows `softint`'s `int $14` instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?**

- No, I didn't have to do anything to make it behave correctly. The `softint` program tries to invoke the page fault handler on its own, but we shouldn't allow users to invoke exceptions of their choice. This is ensured by setting `DPL=0` while filling up the IDT using `SETGATE()`(the last parameter) – `line 72` in `trap/entry.c`.
- Thus, when the user at a numerically higher privilege level (`PL = 3`) tries to access code at a numerically lower privilege level (`PL = 0`), a General Protection Fault (`trap 13`) is triggered instead, to handle the error of a piece of code violating privilege rules.
- If we did allow the user to invoke exceptions as and when they choose, it can result in the execution of malicious code with kernel privileges. The user can push in malicious code into the stack, and then make a call to an exception, which would pick up the malicious code off the stack and execute it with kernel privileges.