Lab 3 - Part B: Page Faults, Breakpoints Exceptions, and System Calls

Sriram V - CS11B058

Exercise 5

- Modified `trap_dispatch()` in `kern/trap.c` to handle page fault exceptions.
- We check whether `tf->tf_trapno` is equal to `T_PGFLT` and call `page_fault_handler(tf)` if they are. (`lines 159 - 161`)
- `make grade` now succeeds on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests.

Exercise 6

- Modified `trap_dispatch()` in `kern/trap.c` to handle breakpoint exceptions.
- We check if `tf->tf_trapno` is `T_BRKPT` and call `monitor(tf)` if it is. (`lines 163 - 165`)
- We also change the privilege level from `0` to `3` in the IDT for the breakpoint's trap handler entry, in `trap_init()`. (`line 76`)
- `make grade` now succeeds on the `breakpoint` test.

Questions

3. **The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to `SETGATE` from `trap_init`). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?**

- If the break point entry is set up in the IDT to be run only with kernel privileges (`DPL=0`), a general protection fault is triggered when it is called from the user level due to a privilege level violation.
- However if the breakpoint entry is set with `DPL=3` (user mode) in the IDT, then it can be called from the user level.

4. **What do you think is the point of these mechanisms, particularly in light of what the `user/softint` test program does?**

- We want only a few exceptions (such as the system call and breakpoint exceptions) to be accessible by the user. The user musn't have access to all 256 exceptions for security purposes.
- Thus, we set the exceptions to be accessed by the user to `DPL=3` in the IDT, and `DPL=0` for the rest.
- In `user/softint`, the user tries to generate a page fault. He could get the kernel to run malicious code by pushing this code on to the stack and calling the page fault, so that the interrupt service routine pops the malicious code and executes it. The page fault could also be used to allocate additional physical memory (a page fault would imply that the page doesn't exist in the memory, so the kernel would allocate additional physical memory assuming it would have to load in new pages) for the process – something no user should be able to do from user mode.

Exercise 7

- To handle the system call (exception number `48`), I had to add in trap handlers (that don't take any error codes - `TRAPHANDLER_NOEC()`) for all exceptions from `20` to `48`, since I use an array for the trap handlers (Lab

3 - Part A's challenge). By adding in these extra trap handlers, I can index into the array easily, rather than remembering which index maps to which exception number.

- I then created corresponding entries for all these trap handlers in the IDT in `kern/trap.c`, by increasing my iterations from `20` (done in Part A) to `49`.
- The privilege level for the `T_SYSCALL` trap handler entry in the IDT is changed to `3` – user level. (`line 79` in `kern/trap.c`)
- `trap_dispatch()` is modified to handle `T_SYSCALL`. The `syscall()` function is called, and the `EAX`, `EDX`, `ECX`, `EBX`, `EDI` and `ESI` register values (present in the `Trapframe` structure) are passed to it.
- Its return value is stored back in the `EAX` register of the `Trapframe` structure.
- The `syscall()` function is modified in `kern/syscall.c` to handle the various system calls (the system call numbers are specified in `inc/syscall.h`).
- A `switch-case` is used to compare `syscallno` with the various system call numbers, and call the appropriate function.
- Since `sys_cputs()` doesn't return anything, we return `0` instead.
- If the system call number is invalid, an error message is printed and `-E_INVAL` is returned.
- Running `make run-hello-nox` prints `hello, world` on the console and causes a page fault in user mode.
- `make grade` now succeeds on the `testbss` test.

Exercise 8

- The `user/hello.c` program page faults because it tries to access the current environment's user id (`thisenv->env_id`). However, `thisenv` has been set to `NULL` in `lib/libmain.c`.
- We rectify this by getting the current environment's system ID using the `sys_getenvid()` system call.
- We make use of the `ENVX()` macro to get the corresponding environment entry in the `envs[]` array.
- `thisenv` is now made to point to this object.
- Booting into the kernel now prints `hello, world` as well as `i am environment 00001000`.
- Since the kernel currently supports only one user environment and `user/hello.c` calls `sys_env_destroy()`, the kernel reports that the only environment has been destroyed and drops into the kernel monitor.
- `make grade` now succeeds on the `hello` test.

Exercise 9

- The `page_fault_handler()` function in `kern/trap.c` is modified to panic if a page fault happens in kernel mode.
- We check if the low bits of `tf->tf_cs` is not equal to the user's DPL (`0x3`).
- `user_mem_check()` in `kern/pmap.c` is then implemented. The passed permissions `perm` is modified to include `PTE_P`.
- The start virtual address `va` is rounded down, and `va+len` is rounded up and set as the end address.
- If the address accessed is greater than `ULIM`, we jump to `bad`.
- Now we try to get the corresponding page entry in the environment's page table. If it doesn't exist, we once again jump to `bad`.
- If it does exist, we check if the set permissions are equal to the permissions passed to the function. If it isn't, we jump to `bad`.
- These checks are performed for all pages in the range `[va, va+len)`.
- If the user program can access this range of addresses, the function returns `0`.
- `bad` is a label below which we write the code to handle all the errors.
- We set `user_mem_check_addr` to the first erroneous virtual address. This is done by initialising this variable to `va`, and if the current address being checked is greater than `va`, we set `user_mem_check_addr` to this value (since this is the first address at which the error occured).
- We now return `-E_FAULT` as an error has occured.
- `debuginfo_eip()` is modified in `kern/kdebug.c` to call `user_mem_assert()` on `usd`, `stabs` and `stabstr`, to check whether they have `PTE_U` permissions.
- `make run-breakpoint-nox` now allows us to enter the kernel monitor and use the `backtrace` command.

- We see the backtrace traverse into lib/libmain.c before the kernel panics with a page fault.

Exercise 10

- We now run `make run-evilhello-nox`. The environment gets destroyed, as is required by this exercise.
- The output observed is:

```
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

- All the `make grade` tests pass now.