Report [Lab 3 User Environments]

I have completed the challenge to continue execution from the current location and to perform single stepping.

Exercise 1

The way to allocate space or map a region becomes quite familiar after lab 2

```
    envs = (struct Env *) boot_alloc(sizeof(struct Env) * NENV);
    // ...
    boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U | PTE_P);
```

I also update `page_init` since `envs` becomes the last data structure allocated by `boot_alloc` now.

```
        if (i >= EXTPHYSMEM / PGSIZE && (uintptr_t)KADDR(i * PGSIZE) < (uintptr_t)(envs + NENV))
            continue;
```

Exercise 2

This exercise asks us to finish several functions in `env.c`

`env_init()`

This function initializes the `env` array, setting `status` and `id` for each elements. And `env_free_list` is also set with the same order as in the array.

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    env_free_list = &envs[0];
    for (int i = 0; i < NENV; i++)
    {
        envs[i].env_status = ENV_FREE;
        envs[i].env_id = 0;
        if (i) envs[i - 1].env_link = &envs[i];
    }

    // Per-CPU part of the initialization
    env_init_percpu();
}
```

`env_setup_vm()`

This function allocates a page directory for a new environment and set up the kernel portion of its virtual memory.

Since kernel portion of all environments is identical, we use `kern_pgdir` as a template to do this.

```
    // LAB 3: Your code here.
    e->env_pgdir = page2kva(p), p->pp_ref++;
    for (int i = PDX(UTOP); i < NPDENTRIES; i++)
    {
        e->env_pgdir[i] = kern_pgdir[i];
    }
```

`region_alloc()`

This is a helper function to allocate a bulk of physic space for some contiguous virtual space. To make it more convenient we allow the start position and size not to align with `PGSIZE`. But this introduce some corner cases that if the function is call with two adjacent region of virtual address and they're not aligned, we must make sure that we won't allocate another page frame for the same page twice!

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    // LAB 3: Your code here.
    // (But only if you need it for load_icode.)
    //
    // Hint: It is easier to use region_alloc if the caller can pass
    //   'va' and 'len' values that are not page-aligned.
    //   You should round va down, and round (va + len) up.
    //   (Watch out for corner-cases!)
    int retval;
    uintptr_t L = (uintptr_t)ROUNDDOWN(va, PGSIZE);
    uintptr_t R = (uintptr_t)ROUNDUP(va + len, PGSIZE);
    for (uintptr_t i = L; i < R; i += PGSIZE)
    {
        pte_t* pte = pgdir_walk(e->env_pgdir, (void *)i, false);
        if (pte != NULL && (*pte & PTE_P))
        {
            continue;
        }
        struct PageInfo *p = page_alloc(0);
        if (p == NULL)
        {
            panic("region_alloc: page_alloc failed");
        }
        retval = page_insert(e->env_pgdir, p, (void *)i, PTE_U | PTE_W);
        if (retval)
        {
            panic("region_alloc: %e", retval);
        }
    }
}
```

`load_icode()`

This function parses an ELF binary image and loads it into the user address space, just like what we do at booting. `region_alloc` helps a lot on allocating space for each section of the binary program and we load the user's page directory into `%cr3` so we can use `memcpy` and `memset` to initialize contents conveniently.

Different from booting where we directly jump to the entry point of the kernel, we store the entry point in the `TrapFrame` and allocate a page for the user's stack.

```
    // LAB 3: Your code here.
    lcr3(PADDR(e->env_pgdir));
    if (((struct Elf *)binary)->e_magic != ELF_MAGIC)
    {
        panic("unrecoginized binary format");
    }
    struct Proghdr *ph, *eph;
    ph = (struct Proghdr *) (binary + ((struct Elf *)binary)->e_phoff);
    eph = ph + ((struct Elf *)binary)->e_phnum;
    for (; ph < eph; ph++) if (ph->p_type == ELF_PROG_LOAD)
    {
        region_alloc(e, (void *)ph->p_va, ph->p_memsz);
        memcpy((void *)(ph->p_va), binary + ph->p_offset, ph->p_filesz);
        memset((void *)(ph->p_va + ph->p_filesz), 0, ph->p_memsz - ph-
>p_filesz);
    }

    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.

    // LAB 3: Your code here.
    region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
    e->env_tf.tf_eip = ((struct Elf *)binary)->e_entry;
    lcr3(PADDR(kern_pgdir));
```

`env_create()`

This function is used to create the first user environment using `env_alloc` and `load_icode`.

```
void
env_create(uint8_t *binary, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *e;
    int retval = env_alloc(&e, 0);
    if (retval)
    {
        panic("env_create: %e", retval);
    }
    load_icode(e, binary);
    e->env_type = type;
}
```

`env_run()`

This function is used to switch between tasks. We have to alter the status of current environment and the environment to run appropriately and restore the new environment's content by loading its page directory and register values.

```
    // LAB 3: Your code here.

    if (curenv != NULL)
    {
        if (curenv->env_status == ENV_RUNNING)
        {
            curenv->env_status = ENV_RUNNABLE;
        }
        // seems we don't have to save the content yet.
    }
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));
    env_pop_tf(&e->env_tf);
```

Exercise 3

This exercise and following text in lab description give us some knowledge about the way 80386 use to handle interrupts. The main data structure is IDT (interrupt descriptor table) so that each interrupt can use its interrupt vector assigned as an index to find the function to handle it.

Exercise 4

We have to set up IDT in this exercise which is a somehow burdensome work.

Two macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` are given in `trapentry.S` as some traps in 80386 push an error code on stack and others don't. So `TRAPHANDLER_NOEC` does much the same thing as `TRAPHANDLER` including pushing the interrupt vector into stack and jumping to `_alltraps` except pushing an extra `0` in the same position of the error code so the 2 kinds of traps finally look like the same.

So we have to read the manual to see whether each trap has an error code and use the right macro to make its entry point.

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */

TRAPHANDLER_NOEC(ENTRY_DIVIDE, T_DIVIDE)
TRAPHANDLER_NOEC(ENTRY_DEBUG, T_DEBUG)
TRAPHANDLER_NOEC(ENTRY_NMI, T_NMI)  // Exception Error Code: Not applicable.
TRAPHANDLER_NOEC(ENTRY_BRKPT, T_BRKPT)
TRAPHANDLER_NOEC(ENTRY_OFLOW, T_OFLOW)
TRAPHANDLER_NOEC(ENTRY_BOUND, T_BOUND)
TRAPHANDLER_NOEC(ENTRY_ILLOP, T_ILLOP)
TRAPHANDLER_NOEC(ENTRY_DEVICE, T_DEVICE)
TRAPHANDLER(ENTRY_DBLFLT, T_DBLFLT)
TRAPHANDLER(ENTRY_TSS, T_TSS)
TRAPHANDLER(ENTRY_SEGNP, T_SEGNP)
TRAPHANDLER(ENTRY_STACK, T_STACK)
TRAPHANDLER(ENTRY_GPFLT, T_GPFLT)
TRAPHANDLER(ENTRY_PGFLT, T_PGFLT)
TRAPHANDLER_NOEC(ENTRY_FPERR, T_FPERR)
TRAPHANDLER(ENTRY_ALIGN, T_ALIGN)
TRAPHANDLER_NOEC(ENTRY_MCHK, T_MCHK)
TRAPHANDLER_NOEC(ENTRY_SIMDERR, T_SIMDERR)
```

And at `_alltraps`, we push the rest of values onto stack to form the layout of the data structure `TrapFrame`. Then we push the address of it (`%esp`) as the argument and call `trap`

```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushw $0;
    pushw %ds;
    pushw $0;
    pushw %es;
    pushal;
    movw $GD_KD, %ax;
    movw %ax, %ds;
    movw %ax, %es;
    pushl %esp;
    call trap;
```

In `trap_init` we are about to set up the IDT (i.e. the entry points for each interrupt handler we defines in `trapentry.S`). Actually the entries in IDT are called gate descriptors in 80386's terminology and there is one bit to determine whether the exception is a trap or a fault (they differ in the returning point after handling of the exception). So we need to refer to the manual again and use `SETGATE` macro to fill in the entries accordingly.

```
    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, &ENTRY_DIVIDE, 0);
    SETGATE(idt[T_DEBUG], 1, GD_KT, &ENTRY_DEBUG, 0);
    // debug exception is preferred to be handled by a task
    // "Instruction address breakpoint conditions are faults,
    //  while other debug conditions are traps."
    // set it to be trap here for simplicity
    SETGATE(idt[T_NMI], 0, GD_KT, &ENTRY_NMI, 0); // exception class is not applicable for NMI
    SETGATE(idt[T_BRKPT], 1, GD_KT, &ENTRY_BRKPT, 3);
    SETGATE(idt[T_OFLOW], 1, GD_KT, &ENTRY_OFLOW, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, &ENTRY_BOUND, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, &ENTRY_ILLOP, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, &ENTRY_DEVICE, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, &ENTRY_DBLFLT, 0); // abort
    SETGATE(idt[T_TSS], 0, GD_KT, &ENTRY_TSS, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, &ENTRY_SEGNP, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, &ENTRY_STACK, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, &ENTRY_GPFLT, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT, &ENTRY_PGFLT, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, &ENTRY_FPERR, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, &ENTRY_ALIGN, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, &ENTRY_MCHK, 0); // abort
    SETGATE(idt[T_SIMDERR], 0, GD_KT, &ENTRY_SIMDERR, 0);
```

Here I set the DPL field of all traps to be 0 to prevent user software to use `int` to invoke them in an unintended way except `T_BRKPT` so that user can use `int 3` instruction to debug.

Questions

1

The only difference between the entry points for each trap is that whether any extra 0 is pushed onto stack to take the position of the error code which is only generated in some kinds of traps. If there is only one kind of entry point, we won't have a uniform structure `Trapframe` to store all the information we concern.

2

The DPL field of each trap is set to 0, preventing user programs to use `int` to invoke any of the traps. So `int $14` is an invalid action for user program `softint` and therefore general protection fault is invoked as a result of privilege violation. If this action is allowed, it means that user programs have a convenient way to give some signals to kernel in an unintended way such as invoking a page fault with a fabricated virtual address and something evil might stem from this.

Exercise 5

As the work in last exercise make the control flow of all interrupts flow to `trap_dispath` with a unified `struct Trapframe` distinguished by `tf->tf_trapno`, it's simple to leverage this to let our `page_fault_handler` to handle that specific fault.

```
    // in trap_dispatch
    if (tf->tf_trapno == T_PGFLT)
    {
        page_fault_handler(tf);
        return;
    }
```

Exercise 6

In `trap_dispatch()`, we call `monitor()` with `tf` for breakpoint exceptions.

```
    if (tf->tf_trapno == T_BRKPT)
    {
        monitor(tf);
        return;
    }
```

Challenge

The content in Chapter 12 Debugging is really useful for completing this challenge.

To understand how to continue execution from the monitor, we have to understand the control flow of interrupt handling in JOS. The entry points in IDT leads all interrupts to `_alltraps` and then `trap()`. In `trap()`, `trap_dispatch()` is called to dispatch interrupts to their true handlers. If `trap_dispatch()` returns without panicking, the control flow returns to the current environment running.

So to continue the execution we simple return -1 to force monitor ends its loop to eventually return from `trap_dispatch()`

```
int
mon_continue(int argc, char **argv, struct Trapframe *tf)
{
    return -1;
}
```

To implement the function of single stepping, we have to understand the debug feature of 80386. There are two kinds of debug interrupt vectors: `T_DEBUG` and `T_BRKPT`. While we are familiar with the latter now, `T_DEBUG` is important for us to implement single stepping function. The debug exception can be triggered in various situations, defined by the debug registers `DR0` ~ `DR7` and some other bits, including the TF bit in `%eflags`. If the TF bit is set, debug exception will be triggered after every instruction completion.

So to implement single stepping, we implement `mon_si` to set the TF bit in the `eflags` image in `tf`. And modify `mon_continue` at the same time to clear the TF bit, leaving the single-step mode.

```c
int
mon_continue(int argc, char **argv, struct Trapframe *tf)
{
    tf->tf_eflags &= ~FL_TF;
    return -1;
}

int
mon_si(int argc, char **argv, struct Trapframe *tf)
{
    tf->tf_eflags |= FL_TF;
    return -1;
}
```

And in `trap_dispatch()`, we invoke the monitor for both `T_BRKPT` and `T_DEBUG`.

```c
    if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG)
    {
        monitor(tf);
        return;
    }
```

Questions

3

Since the interrupt/trap gate descriptor in 80386 has a field to determine the privilege level required to use `int` instruction to invoke it, a general protection fault will be invoked instead of breakpoint exception if we didn't set the corresponding entry in IDT with appropriate privilege level i.e. 3.

```c
    SETGATE(idt[T_BRKPT], 1, GD_KT, &ENTRY_BRKPT, 3);
```

If this was set up incorrectly, there will be a violation of protection based on privilege and therefore a general protection fault is triggered.

4

For some trap/interrupt, we only wish it to be triggered in particular situation such as page fault for the access of a page not in RAM. So these mechanisms are introduced to avoid inducing such traps/interrupts in an unintended way which will introduce ambiguity in the handling of such trap/interrupt.

Exercise 7

First we need to set up the IDT for the system call interrupt, which is without error code and returns to the instruction next to the one invoked the trap.

```
// trapentry.S
TRAPHANDLER_NOEC(ENTRY_SYSCALL, T_SYSCALL)

// trap.c, trap_init()
    SETGATE(idt[T_SYSCALL], 1, GD_KT, &ENTRY_SYSCALL, 3);
```

As `lib/syscall.c` stores the arguments in registers and expects returning value in `%eax`, we manipulate the registers saved in `Trapframe` to achieve this.

```
    if (tf->tf_trapno == T_SYSCALL)
    {
        int retval = syscall(tf->tf_regs.reg_eax,
                             tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx, tf->tf_regs.reg_ebx,
                             tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
        if (retval < 0)
        {
            panic("trap_dispatch: %e", retval);
        }
        tf->tf_regs.reg_eax = retval;
        return;
    }
```

Finally, we call the corresponding kernel functions to handle the system call according to `syscallno` in `syscall()`

```
    switch (syscallno) {
    case SYS_cputs:
        sys_cputs((char *)a1, a2);
        return 0;
    case SYS_cgetc:
        return sys_cgetc();
    case SYS_getenvid:
        return sys_getenvid();
    case SYS_env_destroy:
        return sys_env_destroy(a1);
    default:
        return -E_INVAL;
    }
```

Exercise 8

In this exercise we have to set `thisenv` to point to corresponding element in `envs` array during user environment setup. `lib/entry/S` has defined `envs` to point to `UENVS` we set up in the virtual address space so we can use it to access the kernel's `envs` array. And the index is able to be required by the system call `sys_getenvid()` and `ENVX` macro defined at `inc/env.h`

```
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = &envs[ENVX(sys_getenvid())];
```

Exercise 9 & Exercise 10

We treat page fault in the kernel as a kernel BUG, so we panic at such situation

```
    // LAB 3: Your code here.
    if (!(tf->tf_cs & 3))
    {
        panic("page fault in the kernel");
    }
```

Since `user_mem_check` require to fill in `user_mem_check_addr` the first virtual address invalid, I implement a help function called `user_mem_check_page` to check whether one single virtual address is valid by acquiring corresponding PTE by `pgdir_walk`

```
static bool user_mem_check_page(struct Env *env, uintptr_t va, int perm)
{
    if (va >= ULIM) return false; // kernel space
    pte_t *pte = pgdir_walk(env->env_pgdir, (void *)va, false);
    if (pte == NULL || ~(*pte) & PTE_P)
    {
        return false; // PTE not present
    }
    return ((*pte) & perm) == perm;
}
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    uintptr_t L = (uintptr_t)va;
    uintptr_t R = L + len;

    // check the first page
    if (!user_mem_check_page(env, L, perm))
    {
        user_mem_check_addr = L;
        return -E_FAULT;
    }

    // check the rest pages
    uintptr_t i = L % PGSIZE == 0 ? L + PGSIZE : ROUNDUP(L, PGSIZE);
    while (i < R)
    {
        if (!user_mem_check_page(env, i, perm))
        {
            user_mem_check_addr = i;
            return -E_FAULT;
        }
        i += PGSIZE;
    }

    return 0;
}
```

Now we can check whether the pointer arguments passed to system calls are valid by calling `user_mem_assert`

```
    // LAB 3: Your code here.
    user_mem_assert(curenv, s, len, 0);
```

Do the same check in `debuginfo_eip()`. Just be careful for the pointer stuff.

```
        // Make sure this memory is valid.
        // Return -1 if it is not.  Hint: Call user_mem_check.
        // LAB 3: Your code here.
        if (user_mem_check(curenv, usd, sizeof(struct UserStabData), 0))
        {
            return -1;
        }

        stabs = usd->stabs;
        stab_end = usd->stab_end;
        stabstr = usd->stabstr;
        stabstr_end = usd->stabstr_end;

        // Make sure the STABS and string table memory is valid.
        // LAB 3: Your code here.
        if (user_mem_check(curenv, stabs, (stab_end - stabs) * sizeof(struct Stab), 0))
        {
            return -1;
        }
        if (user_mem_check(curenv, stabstr, stabstr_end - stabstr, 0))
        {
            return -1;
        }
```

If now we run `user/breakpoint` and call `backtrace` command in the monitor, a page fault will happen in the kernel at printing the arguments for `libmain`.

```
Stack backtrace:
  ebp efffff00  eip f0100bf5  args 00000001 efffff28 f01c5000 f0106ab7 f0106876
         kern/monitor.c:149: monitor+353
  ebp efffff80  eip f0104557  args f01c5000 efffffbc f014b178 00000092 f011bfd8
         kern/trap.c:196: trap+312
  ebp efffffb0  eip f010463f  args efffffbc 00000000 00000000 eebfdff0 efffffdc
         kern/syscall.c:69: syscall+0
  ebp eebfdff0  eip 00800031  args 00000000 00000000Incoming TRAP frame at 0xeffffe74
kernel panic at kern/trap.c:277: page fault in the kernel
```

This is because the stack of `libmain` is at the top of user stack with only two arguments for `user/breakpoint` so when `backtrace` tries to print 5 words above `%eip` it accesses the invalid page above `UTOP` which causes a page fault in the kernel.