
**Report for lab3, Kexing Zhou,
1900013008**

Contents

Environment Configuration	2
Test Compiler Toolchain	2
QEMU Emulator	2
Memory Management	3
Exercise 1	3
Exercise 2	3
env_init	3
env_setup_vm	3
region_alloc	3
load_icode	4
env_create	4
env_run	4
Exercise 3	4
Exercise 4 & Challenge 1	5
Questoin 1	6
Exercise 5 & Exercise 6 & Exercise 7	6
Challenge 2	6
Questions 2	7
Exercise 8	7
Exercise 9	8
Exercise 10	9

Environment Configuration

```
1 Hardware Environment:
2 Memory:             16GB
3 Processor:          Intel Core i7-8550U CPU @ 1.66GHz 8
4 GPU:                NVIDIA GeForce RTX 2070
5 OS Type:            64 bit
6 Disk:              924GB
7
8 Software Environment:
9 OS:                 Arch Linux
10 Gcc:                Gcc 11.1.0
11 Make:              GNU Make 4.3
12 Gdb:               GNU gdb 11.1
```

Test Compiler Toolchain

```
1 $ objdump -i # the 5th line say elf32-i386
2 $ gcc -m32 -print-libgcc-file-name
3 /usr/lib/gcc/x86_64-pc-linux-gnu/11.1.0/32/libgcc.a
```

QEMU Emulator

```
1 $ sudo pacman -S riscv64-linux-gnu-binutils \
2     riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb qemu-arch-extra
```

Memory Management

Exercise 1

The setup code in kern/pmap.c, Line 196.

```
1 // allocating the pages array
2 envs = boot_alloc(NENV * sizeof(*envs));
3 memset(envs, 0, NENV * sizeof(*envs));
4 .....
5 // map envs to UENVs with permission user readonly
6 boot_map_region(kern_pgdir, UENVs, PTSIZE, PADDR(envs), PTE_P | PTE_U);
```

Exercise 2

env_init

```
1 void
2 env_init(void) {
3     // Set up envs array
4     // make sure the first free env is env 0
5     env_free_list = &envs[0];
6     for(size_t i = 0; i + 1 < NENV; i++) {
7         envs[i].env_link = &envs[i + 1];
8     }
9     // Per-CPU part of the initialization
10    env_init_percpu();
11 }
```

env_setup_vm

```
1 static int
2 env_setup_vm(struct Env *e) {
3     int i;
4     struct PageInfo *p = NULL;
5
6     // Allocate a page for the page directory
7     if (!(p = page_alloc(ALLOC_ZERO)))
8         return -E_NO_MEM;
9
10    // use kern_pgdir as a template to initialize env
11    memcpy(page2kva(p), kern_pgdir, PGSIZE);
12    p->pp_ref++;
13    e->env_pgdir = page2kva(p);
14
15    // UVPT maps the env's own page table read-only.
16    // Permissions: kernel R, user R
17    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
18
19    return 0;
20 }
```

region_alloc

```
1 static void
2 region_alloc(struct Env *e, void *va, size_t len) {
3     uintptr_t start = ROUNDDOWN((uintptr_t)va, PGSIZE);
4     uintptr_t end = ROUNDUP((uintptr_t)va + len, PGSIZE);
5     int errno = 0;
6     for(uintptr_t i = start; i != end; i += PGSIZE) {
7         struct PageInfo *pp = page_alloc(0);
8         if(pp == NULL)
9             panic("page_alloc failed: %e", -E_NO_MEM);
10        if((errno = page_insert(e->env_pgdir, pp, (void*)i, PTE_P | PTE_W | PTE_U)) < 0)
11            panic("page_insert failed: %e", errno);
12    }
13 }
```

load_icode

```
1 static void
2 region_copy(struct Env *e, void *dst, void *src, size_t len) {
3     uint32_t cr3 = rcr3();
4     // load the env's pgdir to copy page
5     lcr3(PADDR(e->env_pgdir));
6     if(src) memcpy(dst, src, len);
7     else memset(dst, 0, len);
8     lcr3(cr3);
9 }
10
11 static void
12 load_icode(struct Env *e, uint8_t *binary) {
13     struct Elf *eh = (struct Elf *) (binary);
14     assert(eh->e_magic == ELF_MAGIC);
15
16     struct Proghdr *ph_start = (struct Proghdr *) (binary + eh->e_phoff);
17     for(size_t i = 0; i < eh->e_phnum; i++) {
18         struct Proghdr *ph = ph_start + i;
19         if(ph->p_type != ELF_PROG_LOAD) continue;
20         void *va = (void *) ph->p_va;
21         region_alloc(e, va, ph->p_memsz);
22         region_copy(e, va, binary + ph->p_offset, ph->p_filesz);
23         if(ph->p_filesz < ph->p_memsz) { // fill rest memory with zero
24             region_copy(e, va + ph->p_filesz, 0, ph->p_memsz - ph->p_filesz);
25         }
26     }
27     e->env_tf.tf_eip = eh->e_entry;
28
29     // Now map one page for the program's initial stack
30     // at virtual address USTACKTOP - PGSIZE.
31     region_alloc(e, (void *) (USTACKTOP - PGSIZE), PGSIZE);
32 }
```

env_create

```
1 void
2 env_create(uint8_t *binary, enum EnvType type)
3 {
4     struct Env *e;
5     int errno;
6     if((errno = env_alloc(&e, 0)) < 0)
7         panic("env_alloc failed: %e", errno);
8     e->env_type = type;
9     load_icode(e, binary);
10 }
```

env_run

```
1 void
2 env_run(struct Env *e) {
3     // change the state of curenv
4     if(curenv && curenv->env_status == ENV_RUNNING) {
5         curenv->env_status = ENV_RUNNABLE;
6     }
7     curenv = e;
8     curenv->env_status = ENV_RUNNING;
9     curenv->env_runs++;
10
11     // Use lcr3() to switch to its address space
12     lcr3(PADDR(curenv->env_pgdir));
13
14     // switch to environment
15     env_pop_tf(&(curenv->env_tf));
16 }
```

Exercise 3

nothing to report.

Exercise 4 & Challenge 1

I modified the `PLACEHANDLER` macro to place trap message in `.data` segmeng. The message contains functoin name, trap number, privilege level.

```

1 // The Privilege Level
2 #define PL_KERNEL 0
3 #define PL_DEVDRI1 1
4 #define PL_DEVDRI2 2
5 #define PL_USER 3
6
7 #define TRAPHANDLER(name, num, dpl) \
8     .globl name; /* define global symbol for 'name' */ \
9     .type name, @function; /* symbol type is function */ \
10    .align 2; /* align function definition */ \
11    .text; \
12    name: /* function starts here */ \
13    pushl $(num); \
14    jmp _alltraps; \
15    .data; .int name; .int num; .int dpl;
16
17 #define TRAPHANDLER_NOEC(name, num, dpl) \
18     .globl name; \
19     .type name, @function; \
20     .align 2; \
21     .text; \
22     name: \
23     pushl $0; \
24     pushl $(num); \
25     jmp _alltraps; \
26     .data; .int name; .int num; .int dpl;

```

The table entry is at a very beginning, followed by the trap handler.

```

1 .data
2 .global trapentry_table
3 trapentry_table:
4
5 .text
6 TRAPHANDLER_NOEC( trap_handler_DIVIDE , T_DIVIDE , PL_KERNEL )
7 TRAPHANDLER_NOEC( trap_handler_DEBUG , T_DEBUG , PL_KERNEL )
8 TRAPHANDLER_NOEC( trap_handler_NMI , T_NMI , PL_KERNEL )
9 TRAPHANDLER_NOEC( trap_handler_BRKPT , T_BRKPT , PL_USER )
10 TRAPHANDLER_NOEC( trap_handler_OFLOW , T_OFLOW , PL_KERNEL )
11 TRAPHANDLER_NOEC( trap_handler_BOUND , T_BOUND , PL_KERNEL )
12 TRAPHANDLER_NOEC( trap_handler_ILLOP , T_ILLOP , PL_KERNEL )
13 TRAPHANDLER_NOEC( trap_handler_DEVICE , T_DEVICE , PL_KERNEL )
14 TRAPHANDLER ( trap_handler_DBLFLT , T_DBLFLT , PL_KERNEL )
15 TRAPHANDLER ( trap_handler_TSS , T_TSS , PL_KERNEL )
16 TRAPHANDLER ( trap_handler_SEGNP , T_SEGNP , PL_KERNEL )
17 TRAPHANDLER ( trap_handler_STACK , T_STACK , PL_KERNEL )
18 TRAPHANDLER ( trap_handler_GPFLT , T_GPFLT , PL_KERNEL )
19 TRAPHANDLER ( trap_handler_PGFLT , T_PGFLT , PL_KERNEL )
20 TRAPHANDLER_NOEC( trap_handler_FPERR , T_FPERR , PL_KERNEL )
21 TRAPHANDLER_NOEC( trap_handler_ALIGN , T_ALIGN , PL_KERNEL )
22 TRAPHANDLER_NOEC( trap_handler_MCHK , T_MCHK , PL_KERNEL )
23 TRAPHANDLER_NOEC( trap_handler_SIMDERR , T_SIMDERR , PL_KERNEL )
24
25 TRAPHANDLER_NOEC( trap_handler_SYSCALL , T_SYSCALL , PL_USER )
26
27 .data
28 .int 0; .int 0; .int 0;

```

Three `.int 0` is put at the ending, to tell `trap_init` where the table ends. The `trap_init` function is very simple:

```

1 void
2 trap_init(void) {
3     extern struct Segdesc gdt[];
4
5     extern uint32_t trapentry_table[];
6     for(size_t i = 0; trapentry_table[i]; i += 3) {
7         // extract functoin name, trap number, privilege level
8         uintptr_t func_addr = trapentry_table[i];
9         int trap_no = trapentry_table[i + 1];
10        int dpl = trapentry_table[i + 2];
11        SETGATE(idt[trap_no], 1, GD_KT, func_addr, dpl);
12    }
13
14    // Per-CPU setup
15    trap_init_percpu();
16 }

```

Question 1

What is the purpose of having an individual handler function for each exception/interrupt?

Some traps may push an extra `errcode` into stack frame. We implement individual handler to organise the different stack frames into a uniform `Trapframe`, then switch to C code.

if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?

If don't do this, the handler doesn't know whether the code in stack top is an `errcode` or saved registers. He will fail to get the execution context in the trap.

Did you have to do anything to make the user/softint program behave correctly? When user want to use `int` command to make a software interrupt, his privilege level must be equal to or less than the level of that trap. Among all the traps, only `BRKPT` and `SYSCALL` can be induced by user, so their privilege level is 3, which equals to the user's privilege level.

Exercise 5 & Exercise 6 & Exercise 7

The trap dispatch function:

```
1 static void
2 trap_dispatch(struct Trapframe *tf) {
3     switch (tf->tf_trapno) {
4         case T_DEBUG: monitor(tf); break;
5         case T_PGFLT: page_fault_handler(tf); break;
6         case T_BRKPT: monitor(tf); break;
7         case T_SYSCALL:
8             // The system call number will go in %eax,
9             // and the arguments (up to five of them) will go in %edx, %ecx, %ebx, %edi, and %esi, respectively.
10            // The kernel passes the return value back in %eax.
11            tf->tf_regs.reg_eax = syscall(
12                tf->tf_regs.reg_eax,
13                tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx, tf->tf_regs.reg_ebx,
14                tf->tf_regs.reg_edi, tf->tf_regs.reg_esi
15            );
16            env_run(curenv);
17            break;
18        default: break;
19    }
20
21    // Unexpected trap: The user process or the kernel has a bug.
22    print_trapframe(tf);
23    if (tf->tf_cs == GD_KT)
24        panic("unhandled trap in kernel");
25    else {
26        env_destroy(curenv);
27        return;
28    }
29 }
```

Challenge 2

The the `TF` flag in `eflags` register is set to 1, the processor goes into Trap Mode. In Trap Mode, after each assembly code is executed, the processor will cause a `DEBUG` interrupt.

So I set the flag to 1, to enable step debug. And clear it to 0, to continue the program.

```
1 int
2 mon_debug(int argc, char **argv, struct Trapframe * tf) {
3     if (argc > 1) {
4         if (tf->tf_trapno != T_BRKPT) {
5             cprintf("Trap is not a breakpoint, continuing.\n");
6         }
7         char * cmd = argv[1];
8         if (strcmp(cmd, "si") == 0) {
9             tf->tf_eflags |= FL_TF; // step one code
```

```
10     env_run(curenv);
11 }
12 else if(strcmp(cmd, "c") == 0) {
13     tf->tf_eflags &= ~FL_TF; // continuing
14     env_run(curenv);
15 }
16 }
17 cprintf("Usage: debug <si|c>\n");
18 return 0;
19 }
```

And the code in `syscall`

```
1 static void
2 sys_cputs(const char *s, size_t len) {
3     if(user_mem_check(curenv, s, len, PTE_U) < 0) {
4         env_destroy(curenv);
5     }
6     else {
7         cprintf("%.s", len, s);
8     }
9 }
10
11 int32_t
12 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
13 {
14     int32_t ret = 0;
15     switch (syscallno) {
16         case SYS_cgetc:      ret = sys_cgetc(); break;
17         case SYS_cputs:      sys_cputs((const char *)a1, a2); break;
18         case SYS_env_destroy: ret = sys_env_destroy(a1); break;
19         case SYS_getenvid:   ret = sys_getenvid(); break;
20         default:             ret = -E_INVALID; break;
21     }
22     return ret;
23 }
```

Questions 2

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?

explained in Question 1.

The break point interrupt is a software interrupt -- `int 3`. When user want to use `int` command to make a software interrupt, his privilege level must be equal to or less than the level of that trap.

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?

Set the privilege level to 3. Which is user privilege level in JOS.

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

I think this facility prevents user generating some hardware interrupt (such as Timer, BIOS, Security Chips...), to protect the system.

Exercise 8

The code in `libmain`

```
1 void
2 libmain(int argc, char **argv) {
3     // set thisenv to point at our Env structure in envs[]
4     thisenv = envs + ENVX(sys_getenvid());
5
6     // save the name of the program so that panic() can use it
```

```
7     if (argc > 0)
8         binaryname = argv[0];
9
10    // call user main routine
11    umain(argc, argv);
12
13    // exit gracefully
14    exit();
15 }
```

Exercise 9

```
1 void
2 page_fault_handler(struct Trapframe *tf)
3 {
4     uint32_t fault_va;
5
6     // Read processor's CR2 register to find the faulting address
7     fault_va = rcr2();
8
9     // Handle kernel-mode page faults.
10
11    // LAB 3: Your code here.
12    if(tf->tf_es == GD_KD && tf->tf_ds == GD_KD) {
13        cprintf("kernel page fault va %08x ip %08x\n", fault_va, tf->tf_eip);
14        print_trapframe(tf);
15        panic("kernel page fault va %08x ip %08x\n", fault_va, tf->tf_eip);
16    }
17
18    // We've already handled kernel-mode exceptions, so if we get here,
19    // the page fault happened in user mode.
20
21    // Destroy the environment that caused the fault.
22    cprintf("[%08x] user fault va %08x ip %08x\n",
23        curenv->env_id, fault_va, tf->tf_eip);
24    print_trapframe(tf);
25    env_destroy(curenv);
26 }
```

In my implementations, I found the assembly code of function `umain` is mysterious:

```
1 00800033 <umain>:
2 void
3 umain(int argc, char **argv)
4 {
5     asm volatile("int $3");
6 800033: cc                int3
7 }
8 800034: c3                ret
```

There isn't `push %ebp` and `mov %esp,%ebp`, so the `backtrace` won't work correctly. I made a fix by adding

```
1 void
2 umain(int argc, char **argv)
3 {
4     asm volatile("push %ebp");
5     asm volatile("mov %esp, %ebp");
6     //..... the following code
7 }
```

It works well:


```

jos : make — Konsole
文件(F)  编辑(E)  视图(V)  书签(B)  设置(S)  帮助(H)

es  0x---0023
ds  0x---0023
trap 0x00000003 Breakpoint
err  0x00000000
eip  0x00000037
cs  0x---001b
flag 0x00000082
esp  0xeebdfc0
ss  0x---0023

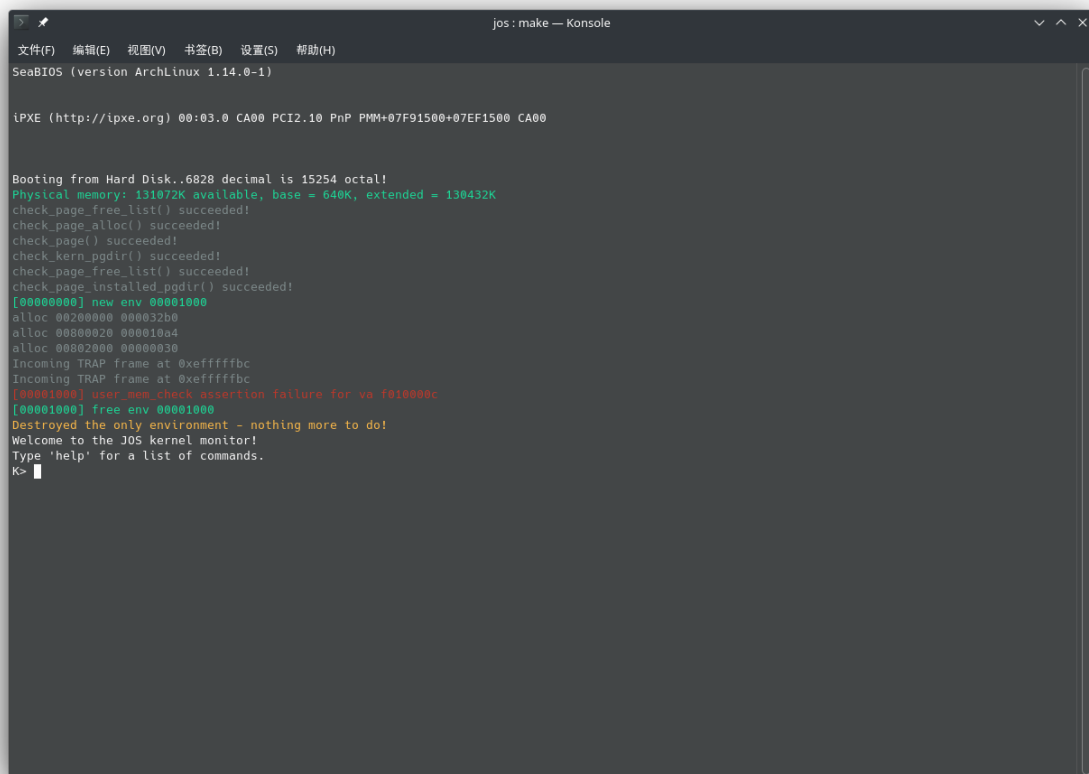
K> backtrace
Stack backtrace:
ebp 0xfffff00 eip f0101144
ebp 0xfffff00 eip f0101144 args 00000001 effffff28 f01c9000 effffff5c
    kern/monitor.c:187: monitor+353
ebp 0xfffff80 eip f0105a18
ebp 0xfffff80 eip f0105a18 args f01c9000 00000000 f014f40c 00000082
    kern/trap.c:159: trap+326
ebp 0xfffffb0 eip f0105ae5
ebp 0xfffffb0 eip f0105ae5 args effffffbc 00000000 00000000 eebdfc0
    kern/trapentry.S:95: <unknown>+4
ebp eebdfc0 eip 00000084
ebp eebdfc0 eip 00000084 args 00000000 00000000 eebdff0 00000057
    lib/libmain.c:23: libmain+76
ebp eebdff0 eip 00000031
Incoming TRAP frame at 0xeffffe6c
kernel page fault va eebfe004 ip f0100ee6
TRAP frame at 0xeffffe6c
edi  0xeebdfdf0
esi  0x00000007
ebp  0xefffff00
oesp 0xeffffe8c
ebx  0xf0184c58
edx  0x000003d5
ecx  0x000003d4
eax  0x00000000
es   0x---0010
ds   0x---0010
trap 0x0000000e Page Fault
cr2  0xeebfe004
err  0x00000000 [kernel, read, not-present]
eip  0xf0100ee6
cs   0x---0008
flag 0x00000046
kernel panic at kern/trap.c:237: kernel page fault va eebfe004 ip f0100ee6
Welcome to the JOS kernel monitor!

```

There is a `pagefault` in the figure, that is because the `backtrace` will look up to 4 parameters above the stack. But when it traces at `lib/entry.S`, i.e. the user environment init code whose stack is `USTACKTOP`, he will watch 4 parameters above `USTACKTOP`. These page is not mapped, so a `pagefault` is generated.

Exercise 10

It works perfectly:



The image shows a terminal window titled "jos: make — Konsole". The terminal displays the boot log of the JOS kernel. The log starts with the iPXE boot loader, followed by booting from a hard disk. It then shows memory statistics and several system checks (page free list, page alloc, page free, kern pgdir) that all succeed. A new environment is created, and memory is allocated. Two trap frames are shown. A user memory check assertion failure is reported for address f01000c. The environment is then freed, and the kernel monitor is reached, displaying the message "Welcome to the JOS kernel monitor!" and "Type 'help' for a list of commands." The prompt "K>" is visible at the bottom.

```
jos: make — Konsole
文件(F)  编辑(E)  视图(V)  书签(B)  设置(S)  帮助(H)
SeaBIOS (version ArchLinux 1.14.0-1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F91500+07EF1500 CA00

Booting from Hard Disk..6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
alloc 00200000 000032b0
alloc 00000020 000010a4
alloc 00002000 00000030
Incoming TRAP frame at 0xeffffbfc
Incoming TRAP frame at 0xeffffbfc
[00001000] user mem check assertion failure for va f01000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```