JOS Lab3

Jentse Huang, 1500017836

Exercise 1

We need to create a list of environment and map it to user's space. In `pamp.c`, function `mem_init()`:

```
envs = (struct Env *)boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));
boot_map_region(kern_pgdir, UENVS, NENV * sizeof(struct Env), PADDR(envs), PTE_U | PTE_P);
```

After `make qemu`, we can find that:

```
check_kern_pgdir() succeeded!
```

which means it works.

Exercise 2

- `env_init()`

Initiate the `envs`, remember to make `env_free_list` points to `envs[0]`:

```
for (int i = NENV - 1; i >= 0; -- i) {
    envs[i].env_status = ENV_FREE;
    envs[i].env_link = env_free_list;
    env_free_list = &envs[i];
}
```

I do not set `env_ids` to 0 because when I allocate `envs`, I use `memset()` to set them to 0.

- `env_setup_vm()`

Set user's page directory. We can use kernel's page directory as a template.

```
e->env_pgdir = page2kva(p);
// Increase env_pgdir's pp_ref
p->pp_ref ++;
// Use kern_pgdir as a template
memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
```

- `region_alloc()`

Allocate and map `len` bytes of physical memory. Use function `page_alloc()` to allocate and `page_insert()` to map.

We only need to round down `va`. No need for rounding up `va + len`.

```
struct PageInfo *p;
void *begin = ROUNDDOWN(va, PGSIZE), *end = ROUNDUP(va + len, PGSIZE);
for (; begin < end; begin += PGSIZE) {
    // Allocate
    if (!(p = page_alloc(ALLOC_ZERO))) {
        panic("Allocation failed!");
    }
    // Map
    // Pages should be writable by user and kernel
    if (page_insert(e->env_pgdir, p, begin, PTE_W | PTE_U) != 0) {
        panic("Mapping failed!");
    }
}
```

- `load_icode()`

This is the key function in exercise 2.

Before filling in this function, we need to read `elf.h` first. the `uint32_t *binary` is a pointer points to a `struct Elf`. And we can find where the `struct Proghdr` is and how many headers are there in the `struct Elf` through `binary + e_phoff` and `e_phnum`.

Then, for one `struct Ptoghdr`, we first check whether `p_type == ELF_PROG_LOAD`. Then we allocate a memory of size of `p_memsz` at `p_va` using function `region_alloc()` and copy the memory from `binary + p_offset` to `p_va` of a size of `p_filesz`.

Why are two sizes of memory different? It is because that the size of loadable part is `p_filesz`, but there may be some other parts in actual memory. For example `.bss` part. So we need to set the rest to 0.

The function is invoked by the kernel, but we need to modify the virtual address of user. So we need to use `lcr3()` to switch address space when dealing with virtual address. **Remember to switch back and remember that page directory uses physical address, so the `e->env_pgdir` and `kern_pgdir` must be convert to physical address before passing in function `lcr3()`.**

We can find that our system starts the first environment by an instruction `iret`. In function `env_pop_tf()`, the `struct Trapframe` is a stack, and the address of `tf.tf_eip` is the address which will pop to real register `eip`. So we need to put the value `e_entry` to `e->env_tf.tf_eip`.

At last we allocate and map a page for user stack.

```
struct Proghdr *ph;
struct Elf *elf_hdr = (struct Elf *)binary;

// Check if it is a valid ELF
if (elf_hdr->e_magic != ELF_MAGIC) {
    panic("Wrong ELF!");
}

// Switch to user address space
lcr3(PADDR(e->env_pgdir));
// Load each program segment
```

```
    ph = (struct Proghdr *) ((uint8_t *)elf_hdr + elf_hdr->e_phoff);
    for (int i = 0; i < elf_hdr->e_phnum; ++ i) {
        if (ph[i].p_type == ELF_PROG_LOAD) {
            region_alloc(e, (void *)ph[i].p_va, ph[i].p_memsz);
            // Set memory of loadable part
            memcpy((void *)ph[i].p_va, (void *)binary + ph[i].p_offset, ph[i].p_filesz);
            // Set memory of other parts to zero
            memset((void *)ph[i].p_va + ph[i].p_filesz, 0, ph[i].p_memsz - ph[i].p_filesz);
        }
    }
    // Switch back to kernel address space
    lcr3(PADDR(kern_pgdir));

    // Set the entry of codes
    e->env_tf.tf_eip = elf_hdr->e_entry;

    region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
```

- `env_create()`

Just follow the instruction, this function is easy to implement.

```
    int r;
    struct Env *e;

    if ((r = env_alloc(&e, 0)) != 0) {
        return r;
    }
    load_icode(e, binary);
    e->env_type = type;
```

- `env_run()`

Follow the instruction and it's easy to implement.

```
    if (curenv != NULL && curenv->env_status == ENV_RUNNING) {
        curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs ++;
    lcr3(PADDR(curenv->env_pgdir));

    env_pop_tf(&(curenv->env_tf));
```

After `make qemu`, we can see the 'Triple Fault' message.

Exercise 3

I have learned exceptions and interrupts in Introduction to Computer Science and Operating System class, so I will skip this exercise.

Exercise 4

- `trapentry.S`

First, we fill in the codes in `trapentry.S`. This file is to set all the entry points of interrupts and the common routines of all traps. Two macros `TRAPHANDLER(name, num)` and `TRAPHANDLER_NOEC(name, num)` will help to build entry points. We need to pass the interrupt vector to `num` and set a function name passing to `name` and we can use `void name();` in `trap.c` to declare them.

The table below shows whether an error code should be pushed.

Name	Vector nr.	Type	Mnemonic	Error code?
Divide-by-zero Error	0 (0x0)	Fault	#DE	No
Debug	1 (0x1)	Fault/Trap	#DB	No
Non-maskable Interrupt	2 (0x2)	Interrupt	-	No
Breakpoint	3 (0x3)	Trap	#BP	No
Overflow	4 (0x4)	Trap	#OF	No
Bound Range Exceeded	5 (0x5)	Fault	#BR	No
Invalid Opcode	6 (0x6)	Fault	#UD	No
Device Not Available	7 (0x7)	Fault	#NM	No
Double Fault	8 (0x8)	Abort	#DF	Yes (Zero)
~~Coprocessor Segment Overrun~~	9 (0x9)	Fault	-	No
Invalid TSS	10 (0xA)	Fault	#TS	Yes
Segment Not Present	11 (0xB)	Fault	#NP	Yes
Stack-Segment Fault	12 (0xC)	Fault	#SS	Yes
General Protection Fault	13 (0xD)	Fault	#GP	Yes
Page Fault	14 (0xE)	Fault	#PF	Yes
Reserved	15 (0xF)	-	-	No
x87 Floating-Point Exception	16 (0x10)	Fault	#MF	No
Alignment Check	17 (0x11)	Fault	#AC	Yes
Machine Check	18 (0x12)	Abort	#MC	No
SIMD Floating-Point Exception	19 (0x13)	Fault	#XM/#XF	No
Virtualization Exception	20 (0x14)	Fault	#VE	No
Reserved	21-29 (0x15-0x1D)	-	-	No
Security Exception	30 (0x1E)	-	#SX	Yes
Reserved	31 (0x1F)	-	-	No
Triple Fault	-	-	-	No
~~FPU Error Interrupt~~	IRQ 13	Interrupt	#FERR	No

And for this lab, we can add all the traps defined in `trap.h` :

```
TRAPHANDLER_NOEC(T_DIVIDE_H, T_DIVIDE)
TRAPHANDLER_NOEC(T_DEBUG_H, T_DEBUG)
TRAPHANDLER_NOEC(T_NMI_H, T_NMI)
TRAPHANDLER_NOEC(T_BRKPT_H, T_BRKPT)
TRAPHANDLER_NOEC(T_OFLOW_H, T_OFLOW)
TRAPHANDLER_NOEC(T_BOUND_Hr, T_BOUND)
TRAPHANDLER_NOEC(T_ILLOP_H, T_ILLOP)
TRAPHANDLER_NOEC(T_DEVICE_H, T_DEVICE)
TRAPHANDLER(T_DBLFLT_H, T_DBLFLT)
TRAPHANDLER(T_TSS_H, T_TSS)
TRAPHANDLER(T_SEGNP_H, T_SEGNP)
TRAPHANDLER(T_STACK_H, T_STACK)
TRAPHANDLER(T_GPFLT_H, T_GPFLT)
TRAPHANDLER(T_PGFLT_H, T_PGFLT)
TRAPHANDLER_NOEC(T_FPERR_H, T_FPERR)
TRAPHANDLER(T_ALIGN_H, T_ALIGN)
TRAPHANDLER_NOEC(T_MCHK_H, T_MCHK)
TRAPHANDLER_NOEC(T_SIMDERR_H, T_SIMDERR)
TRAPHANDLER_NOEC(T_SYSCALL_H, T_SYSCALL)
TRAPHANDLER_NOEC(T_DEFAULT_H, T_DEFAULT)
```

For `_alltraps`, we need to read `struct Trapframe` in `trap.h`:

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

As said before, CPU has pushed some registers in the stack:

```
        +--------------------+ KSTACKTOP
        | 0x00000 | old SS   |     " - 4
        |      old ESP       |     " - 8
        |     old EFLAGS     |     " - 12
        | 0x00000 | old CS   |     " - 16
        |      old EIP       |     " - 20
        |     error code     |     " - 24 <---- ESP
        +--------------------+
```

(`ss, cs, ds, es` need paddings)

We can find that in the two macros, `pushl $(num)` also push trap number. So we only need to push `%ds` and `%es`.

```
_alltraps:
  # push values to make the stack look like a struct Trapframe
  pushl %ds
  pushl %es
  pushal
  # load GD_KD into %ds and %es
  movw $GD_KD, %ax
  movw %ax, %ds
  movw %ax, %es
  # pushl %esp to pass a pointer to the Trapframe as an argument to trap()
  pushl %esp
  # call trap (can trap ever return?)
  call trap
```

- `trap_init()` in `trap.c`

In this function we need to declare the entry point of each interrupt and set a gate descriptor.

`trap.c` has defined an array `idt`, we need to use macro `SETGATE` to fill in this array.

All of these interrupts reset `IF`, so we set `istrap = 0`.

We can find kernel code segment in `env.c`, in `struct Segdesc gdt[]`:

```
// 0x8 - kernel code segment
[GD_KT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 0),
```

So the `sel` argument of `SETGATE()` is `GD_KT`.

And for now, there won't be anything wrong if we set all `dpl = 0`.

```
extern void T_DIVIDE_H();
extern void T_DEBUG_H();
extern void T_NMI_H();
extern void T_BRKPT_H();
extern void T_OFLOW_H();

extern void T_BOUND_H();
```

```
extern void T_ILLOP_H();
extern void T_DEVICE_H();
extern void T_DBLFLT_H();
extern void T_TSS_H();
extern void T_SEGNP_H();
extern void T_STACK_H();
extern void T_GPFLT_H();
extern void T_PGFLT_H();
extern void T_FPERR_H();
extern void T_ALIGN_H();
extern void T_MCHK_H();
extern void T_SIMDERR_H();
extern void T_SYSCALL_H();
extern void T_DEFAULT_H();
SETGATE(idt[T_DIVIDE], 0, GD_KT, T_DIVIDE_H, 0);
SETGATE(idt[T_DEBUG], 0, GD_KT, T_DEBUG_H, 0);
SETGATE(idt[T_NMI], 0, GD_KT, T_NMI_H, 0);
SETGATE(idt[T_BRKPT], 0, GD_KT, T_BRKPT_H, 0);
SETGATE(idt[T_OFLOW], 0, GD_KT, T_OFLOW_H, 0);
SETGATE(idt[T_BOUND], 0, GD_KT, T_BOUND_H, 0);
SETGATE(idt[T_ILLOP], 0, GD_KT, T_ILLOP_H, 0);
SETGATE(idt[T_DEVICE], 0, GD_KT, T_DEVICE_H, 0);
SETGATE(idt[T_DBLFLT], 0, GD_KT, T_DBLFLT_H, 0);
SETGATE(idt[T_TSS], 0, GD_KT, T_TSS_H, 0);
SETGATE(idt[T_SEGNP], 0, GD_KT, T_SEGNP_H, 0);
SETGATE(idt[T_STACK], 0, GD_KT, T_STACK_H, 0);
SETGATE(idt[T_GPFLT], 0, GD_KT, T_GPFLT_H, 0);
SETGATE(idt[T_PGFLT], 0, GD_KT, T_PGFLT_H, 0);
SETGATE(idt[T_FPERR], 0, GD_KT, T_FPERR_H, 0);
SETGATE(idt[T_ALIGN], 0, GD_KT, T_ALIGN_H, 0);
SETGATE(idt[T_MCHK], 0, GD_KT, T_MCHK_H, 0);
SETGATE(idt[T_SIMDERR], 0, GD_KT, T_SIMDERR_H, 0);
SETGATE(idt[T_SYSCALL], 0, GD_KT, T_SYSCALL_H, 0);
SETGATE(idt[T_DEFAULT], 0, GD_KT, T_DEFAULT_H, 0);
```

- Q: 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?)

  A: Then we will have some trouble with protection.

- Q: 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)?

  A: Because page fault's `dpl = 0`, which means kernel does not allow user to invoke this interrupt. Nothing will happen when we set its `dpl = 3`, but we do not want user to interfere with memory management.

After `make grade`, we can see:

```
divzero: OK (0.8s)
softint: OK (1.1s)
badsegment: OK (0.8s)
Part A score: 30/30
```

Exercise 5

After reading `trap()` in `trap.c`, we will find that it uses `trap_dispatch()` to deliver different traps to different handlers. So we need to invoke different handlers by `tf->tf_trapno`:

```
switch (tf->tf_trapno) {
case T_PGFLT:
    page_fault_handler(tf);
    return;
default:
    break;
}
```

After `make grade`, we can find that:

```
faultread: OK (0.8s)
    (Old jos.out.faultread failure log removed)
faultreadkernel: OK (1.1s)
    (Old jos.out.faultreadkernel failure log removed)
faultwrite: OK (0.9s)
    (Old jos.out.faultwrite failure log removed)
faultwritekernel: OK (1.0s)
    (Old jos.out.faultwritekernel failure log removed)
```

Exercise 6

We need to add another case in `trap_dispatch()` in `trap.c`, and we can use `monitor()` in `monitor.c` to invoke kernel monitor.

Add this code to the `switch` of exercise 5:

```
case T_BRKPT:
    monitor(tf);
    return;
```

However, we will not pass the test of breakpoints. We need to set `dpl = 3` in `trap_init()`:

```
SETGATE(idt[T_BRKPT], 0, GD_KT, T_BRKPT_H, 3);
```

After `make qemu`, we can see:

```
breakpoint: OK (2.0s)
  (Old jos.out.breakpoint failure log removed)
```

Challenge 2

After `int $3` and we get into kernel monitor and will not return. But there is still way to let the original environment run. The `struct Trapframe *tf` is passed into monitor functions, and we can recover the environment by this argument.

First, include `kern/env.h` in `monitor.c`:

```
#include <kern/env.h>
```

Then, add `commands[]`, we can use either `continue` or `c` (and `stepi` or `si`):

```
    { "continue", "Continue execution the environment in tf", mon_continue },
    { "c", "Continue execution the environment in tf", mon_continue },
    { "stepi", "Execution one instruction of the environment in tf", mon_stepi },
    { "si", "Execution one instruction of the environment in tf", mon_stepi },
```

Then add these functions into `monitor.h`:

```
int mon_continue(int argc, char **argv, struct Trapframe *tf);
int mon_stepi(int argc, char **argv, struct Trapframe *tf);
```

We need to check whether it is in `backtrace`, which means we need to check whether `tf == NULL`. And I use `env_run()` to recover context from `tf`.

For `stepi`, we only need to set `TF = 1` in `tf->tf_eflags`. `TF` is the eighth bit, so we need to `|= 0x100` to set it, or `&= ~0x100` to clear it.

```
int
mon_continue(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 1) {
        cprintf("Usage: c\n       continue\n");
        return 0;
    }
    if (tf == NULL) {
        cprintf("Not in backtrace\n");
        return 0;
    }

    curenv->env_tf = *tf;
    curenv->env_tf.tf_eflags &= ~0x100;
    env_run(curenv);
    return 0;

}
```

```
int
mon_stepi(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 1) {
        cprintf("Usage: si\n       stepi\n");
        return 0;
    }
    if (tf == NULL) {
        cprintf("Not in backtrace\n");
        return 0;
    }

    curenv->env_tf = *tf;
    curenv->env_tf.tf_eflags |= 0x100;
    env_run(curenv);
    return 0;
}
```

And let's check the performance by `make run-breakpoint`:

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b5000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800037
  cs   0x----001b
  flag 0x00000082
  esp  0xeebfdfd0
  ss   0x----0023
K> c
Incoming TRAP frame at 0xefffffbc
[00001000] exiting gracefully
[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> c
```

```
Not in backtrace
```

We can see that the environment continues to run, and when the environment ends, it will not do anything if you use `c`. So do `si`:

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b5000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800037
  cs   0x----001b
  flag 0x00000082
  esp  0xeebfdfd0
  ss   0x----0023
K> si
Incoming TRAP frame at 0xefffffbc
TRAP frame at 0xf01b5000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdff0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800038
  cs   0x----001b
  flag 0x00000182
  esp  0xeebfdfd4
  ss   0x----0023
[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.
```

- Q: 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?

  A: Because of the `dpl` argument in `SETGATE()`. If it is 0, it will be a general protection fault. If it is 3, it will be a break point exception.

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

  A: For protection.

Exercise 7

We have added `T_SYSCALL` in exercise 4, but we need to change the `dpl = 3` in `trap_init()`:

```
SETGATE(idt[T_SYSCALL], 0, GD_KT, T_SYSCALL_H, 3);
```

The system call number is in `eax`, and the five arguments are in `edx, ecx, ebx, edi, esi` respectively. And the return value should be passed to `eax`. Then we can add a new case in `trap_dispatch()` in `trap.c`:

```
case T_SYSCALL:
    tf->tf_regs.reg_eax = 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);
    return;
```

Then we modify the `syscall` in `kern/syscall.c`. We deliver different system call number to different function using `switch()`. Remember to delete the `panic()`. The system calls that we need to implement are in `inc/syscall.h`.

```
case SYS_cputs:
    sys_cputs((char *)a1, (size_t)a2);
    return 0;
case SYS_cgetc:
    return sys_cgetc();
case SYS_getenvid:
    return sys_getenvid();
case SYS_env_destroy:
    return sys_env_destroy(sys_getenvid());
```

After `make qemu`, we can see:

```
testbss: OK (2.1s)
  (Old jos.out.testbss failure log removed)
```

and after `make run-hello`, we can see a "hello, world" and a page fault:

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
hello, world
Incoming TRAP frame at 0xefffffbc
[00001000] user fault va 00000048 ip 00800048
TRAP frame at 0xf01b3000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0x00000048
  err  0x00000004 [user, read, not-present]
  eip  0x00800048
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfb8
  ss   0x----0023
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

Exercise 8

`entry.S` has defined `envs` and mapped it to `UENV`. So we can use it as the address of environments array. And we can use `sys_getenvid()` to get the current environment's id. Remember the function returns an environment id, but the index is the lowest 10 bit. See `env.h`:

```
// +1+---------------21-----------------+--------10--------+
// |0|          Uniqueifier             |   Environment    |
// | |                                  |      Index       |
// +------------------------------------+------------------+
//                                       \--- ENVX(eid) --/
```

So we need to use `ENVX(sys_getenvid())` to get the index.

```
  thisenv = envs + ENVX(sys_getenvid());
```

After `make grade`, we can see:

```
hello: OK (1.9s)
    (Old jos.out.hello failure log removed)
```

Exercise 9

- `page_fault_handler()`

We need to panic the kernel if a page fault happens in kernel mode. To check if it is in kernel mode, just check the low 2 bits of `tf->tf_cs` as in `trap()`.

```
if ((tf->tf_cs & 3) == 0) {
    panic("A Page Fault in Kernel!");
}
```

- `user_mem_check()`

This function check these things below:

- The page and page table entry of this virtual address exist.
- The permission bits of the page table entry equals to the argument `perm`
- All the addresses are below `ULIM`

And this function needs to record the address which causes an error. Remember to use `va` when the address is below `va` (because of `ROUNDDOWN`).

```
uintptr_t begin = ROUNDDOWN((uintptr_t)va, PGSIZE), end = ROUNDUP((uintptr_t)va + len, PGSIZE);
for (; begin < end; begin += PGSIZE) {
    pte_t *p = pgdir_walk(env->env_pgdir, (void *)begin, 0);
    if (p == NULL || (*p & PTE_P) == 0 || (*p & perm) != perm || begin >= ULIM) {
        user_mem_check_addr = ((begin < (uintptr_t)va) ? (uintptr_t)va : begin);
        return -E_FAULT;
    }
}
```

- `debuginfo_eip`

We need to use `user_mem_check()` to check some memory.

```
    if (user_mem_check(curenv, (void *)usd, sizeof(usd), PTE_P) < 0) {
        return -1;
    }
    if (user_mem_check(curenv, (void *)stabs, sizeof(stabs), PTE_P) < 0) {
        return -1;
    }
    if (user_mem_check(curenv, (void *)stabstr, sizeof(stabstr), PTE_P) < 0) {
        return -1;
    }
```

After `make grade`, we can see:

```
buggyhello: OK (1.2s)
  (Old jos.out.buggyhello failure log removed)
buggyhello2: OK (1.9s)
  (Old jos.out.buggyhello2 failure log removed)
evilhello: OK (1.9s)
  (Old jos.out.evilhello failure log removed)
```

After `make run-breakpoint`, we can use `backtrace` in the kernel monitor:

```
K> backtrace
Stack backtrace:
 ebp effff20  eip f0100c77  args 00000001 efffff38 f01b4000 00000000 f0172840
      kern/monitor.c:256: monitor+276
 ebp efffff90  eip f01039fd  args f01b4000 efffffbc 00000000 00000082 00000000
      kern/trap.c:192: trap+169
 ebp efffffb0  eip f0103b0e  args efffffbc 00000000 00000000 eebfdfd0 efffffdc
      kern/syscall.c:69: syscall+0
 ebp eebfdfd0  eip 00800073  args 00000000 00000000 eebfdff0 00800049 00000000
      lib/libmain.c:26: libmain+58
 ebp eebfdff0  eip 00800031  args 00000000 00000000Incoming TRAP frame at 0xeffffe94
kernel panic at kern/trap.c:267: A Page Fault in Kernel!
```

The page fault is caused because in `entry.S`, if it is kernel who starts the environment, `%esp` will equal to `USTACKTOP`, so we need to push two arguments:

```
.text
.globl _start
_start:
  // See if we were started with arguments on the stack
  cmpl $USTACKTOP, %esp
  jne args_exist

  // If not, push dummy argc/argv arguments.
  // This happens when we are loaded by the kernel,
  // because the kernel does not know about passing arguments.
  pushl $0
  pushl $0

args_exist:
  call libmain
1:  jmp 1b
```

And the implementation of `backtrace` need to output five arguments. So when it accesses the third argument, it will be a page fault.

Exercise 10

After `make run-evilhello`, we can see that there is not any panic:

```
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
Incoming TRAP frame at 0xefffffbc
[00001000] user_mem_check assertion failure for va f010000c
[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.
```

Score

```
divzero: OK (1.1s)
softint: OK (2.0s)
badsegment: OK (2.1s)
Part A score: 30/30

faultread: OK (2.0s)
faultreadkernel: OK (2.0s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.9s)
breakpoint: OK (2.0s)
testbss: OK (2.1s)
hello: OK (0.9s)
buggyhello: OK (2.0s)
buggyhello2: OK (2.0s)
evilhello: OK (1.1s)
Part B score: 50/50

Score: 80/80
```

I have fixed some small bugs of previous labs, which are not discussed in this report. So please use the latest code.