Part A

Exercise 1

In kern/pmap.c, mem_init()

```
//////////////////////////////////////////////////////////////////////
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
// LAB 3: Your code here.
envs = (struct Env *)boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));

//////////////////////////////////////////////////////////////////////
// Map the 'envs' array read-only by the user at linear address UENVS
// (ie. perm = PTE_U | PTE_P).
// Permissions:
//    - the new image at UENVS  -- kernel R, user R
//    - envs itself -- kernel RW, user NONE
// LAB 3: Your code here.
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
```

Exercise 2

In kern/env.c

```
// Mark all environments in 'envs' as free, set their env_ids to 0,
// and insert them into the env_free_list.
// Make sure the environments are in the free list in the same order
// they are in the envs array (i.e., so that the first call to
// env_alloc() returns envs[0]).
//
void
env_init(void)
{
	// Set up envs array
	// LAB 3: Your code here.
	for (int i = NENV - 1; i >= 0; i--)
	{
		envs[i].env_id = 0;
		envs[i].env_status = ENV_FREE;
		envs[i].env_link = env_free_list;
		env_free_list = &envs[i];
	}

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

```c
static int
env_setup_vm(struct Env *e)
{
	int i;
	struct PageInfo *p = NULL;

	// Allocate a page for the page directory
	if (!(p = page_alloc(ALLOC_ZERO)))
		return -E_NO_MEM;

	// Now, set e->env_pgdir and initialize the page directory.
	//
	// Hint:
	//    - The VA space of all envs is identical above UTOP
	//      (except at UVPT, which we've set below).
	//      See inc/memlayout.h for permissions and layout.
	//      Can you use kern_pgdir as a template?  Hint: Yes.
	//      (Make sure you got the permissions right in Lab 2.)
	//    - The initial VA below UTOP is empty.
	//    - You do not need to make any more calls to page_alloc.
	//    - Note: In general, pp_ref is not maintained for
	//      physical pages mapped only above UTOP, but env_pgdir
	//      is an exception -- you need to increment env_pgdir's
	//      pp_ref for env_free to work correctly.
	//    - The functions in kern/pmap.h are handy.

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

	// UVPT maps the env's own page table read-only.
	// Permissions: kernel R, user R
	e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;

	return 0;
}
```

```c
//
// Allocate len bytes of physical memory for environment env,
// and map it at virtual address va in the environment's address space.
// Does not zero or otherwise initialize the mapped pages in any way.
// Pages should be writable by user and kernel.
// Panic if any allocation attempt fails.
//
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!)
	uintptr_t begin = (uintptr_t)ROUNDDOWN(va, PGSIZE);
	uintptr_t end = (uintptr_t)ROUNDUP(va + len, PGSIZE);
	for (uintptr_t i = begin; i < end; i += PGSIZE)
	{
		struct PageInfo *pp = page_alloc(ALLOC_ZERO);
		if (!pp)
		{
			panic("In region_alloc: failed to allocate page to user environment");
		}
		if (page_insert(e->env_pgdir, pp, (void *)i, PTE_U | PTE_W) != 0)
		{
			panic("In region_alloc: failed to insert page into user environment");
		}
	}
}
```

```c
static void
load_icode(struct Env *e, uint8_t *binary)
// LAB 3: Your code here.
struct Elf *elf = (struct Elf *)binary;
if (elf->e_magic != ELF_MAGIC)
{
	panic("In load_icode: invalid ELF");
}

struct Proghdr *ph, *eph;
ph = (struct Proghdr *)(binary + elf->e_phoff);
eph = ph + elf->e_phnum;
for (; ph < eph; ph++)
{
	if (ph->p_type == ELF_PROG_LOAD)
	{
		region_alloc(e, (void *)ph->p_va, ph->p_memsz);
		memset((void *)ph->p_va, 0, ph->p_memsz);
		memmove((void *)ph->p_va, binary+ph->p_offset, ph->p_filesz);
	}
}

// set the entry point
e->env_tf.tf_eip = elf->e_entry;

// 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_brk = (uintptr_t)ROUNDDOWN(USTACKTOP - PGSIZE, PGSIZE);  // update current program's break
```

```c
//
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, enum EnvType type)
{
	// LAB 3: Your code here.
	struct Env *e;
	if (env_alloc(&e, 0) != 0)
	{
		panic("In env_create: failed to allocate environment");
	}
	e->env_type = type;

	// switch page table when change environment
	lcr3(PADDR(e->env_pgdir));
	load_icode(e, binary);
	lcr3(PADDR(kern_pgdir));
}
```

```c
void
env_run(struct Env *e)
{
	// Step 1: If this is a context switch (a new environment is running):
	//	   1. Set the current environment (if any) back to
	//	      ENV_RUNNABLE if it is ENV_RUNNING (think about
	//	      what other states it can be in),
	//	   2. Set 'curenv' to the new environment,
	//	   3. Set its status to ENV_RUNNING,
	//	   4. Update its 'env_runs' counter,
	//	   5. Use lcr3() to switch to its address space.
	// Step 2: Use env_pop_tf() to restore the environment's
	//	   registers and drop into user mode in the
	//	   environment.

	// Hint: This function loads the new environment's state from
	//	e->env_tf.  Go back through the code you wrote above
	//	and make sure you have set the relevant parts of
	//	e->env_tf to sensible values.

	// LAB 3: Your code here.
	if (curenv != e)
	{
		if (curenv && curenv->env_status == ENV_RUNNING)
		{
			curenv->env_status = ENV_RUNNABLE;
		}
		curenv = e;
		e->env_status = ENV_RUNNING;
		e->env_runs++;
		lcr3(PADDR(e->env_pgdir));
	}
	env_pop_tf(&e->env_tf);
}
```

Exercise 4

In kern/trapentry.S

```
.text

/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
// 9 and 15 are reserved
TRAPHANDLER_NOEC(ENTRY_DIVIDE, T_DIVIDE)        // divide error
TRAPHANDLER_NOEC(ENTRY_DEBUG, T_DEBUG)          // debug exception
TRAPHANDLER_NOEC(ENTRY_NMI, T_NMI)              // non-maskable interrupt
TRAPHANDLER_NOEC(ENTRY_BRKPT, T_BRKPT)          // breakpoint
TRAPHANDLER_NOEC(ENTRY_OFLOW, T_OFLOW)          // overflow
TRAPHANDLER_NOEC(ENTRY_BOUND, T_BOUND)          // bounds check
TRAPHANDLER_NOEC(ENTRY_ILLOP, T_ILLOP)          // illegal opcode
TRAPHANDLER_NOEC(ENTRY_DEVICE, T_DEVICE)        // device not available
TRAPHANDLER(ENTRY_DBLFLT, T_DBLFLT)             // double fault
TRAPHANDLER(ENTRY_TSS, T_TSS)                   // invalid task switch segment
TRAPHANDLER(ENTRY_SEGNP, T_SEGNP)               // segment not present
TRAPHANDLER(ENTRY_STACK, T_STACK)               // stack exception
TRAPHANDLER(ENTRY_GPFLT, T_GPFLT)               // general protection fault
TRAPHANDLER(ENTRY_PGFLT, T_PGFLT)               // page fault
TRAPHANDLER_NOEC(ENTRY_FPERR, T_FPERR)          // floating point error
TRAPHANDLER(ENTRY_ALIGN, T_ALIGN)               // alignment check
TRAPHANDLER_NOEC(ENTRY_MCHK, T_MCHK)            // machine check
TRAPHANDLER_NOEC(ENTRY_SIMDERR, T_SIMDERR)      // SIMD floating point error
TRAPHANDLER_NOEC(ENTRY_SYSCALL, T_SYSCALL)      // system call

/*
 * Lab 3: Your code here for _alltraps
 */
.globl _alltraps;
.type _alltraps,@function;
.align 2;
_alltraps:
        pushl %ds
        pushl %es
        pushal
        movl $GD_KD, %eax
        movw %ax, %ds
        movw %ax, %es
        pushl %esp
        call trap
```

In kern/trap.c

Declare entry point functions, then use SETGATE macro to initialize IDT

```c
// entry points, 9 and 15 reserved
extern void ENTRY_DIVIDE();      // 0 divide error
extern void ENTRY_DEBUG();       // 1 debug exception
extern void ENTRY_NMI();         // 2 non-maskable interrupt
extern void ENTRY_BRKPT();       // 3 breakpoint
extern void ENTRY_OFLOW();       // 4 overflow
extern void ENTRY_BOUND();       // 5 bounds check
extern void ENTRY_ILLOP();       // 6 illegal opcode
extern void ENTRY_DEVICE();      // 7 device not available
extern void ENTRY_DBLFLT();      // 8 double fault
extern void ENTRY_TSS();         // 10 invalid task switch segment
extern void ENTRY_SEGNP();       // 11 segment not present
extern void ENTRY_STACK();       // 12 stack exception
extern void ENTRY_GPFLT();       // 13 general protection fault
extern void ENTRY_PGFLT();       // 14 page fault
extern void ENTRY_FPERR();       // 16 floating point error
extern void ENTRY_ALIGN();       // 17 aligment check
extern void ENTRY_MCHK();        // 18 machine check
extern void ENTRY_SIMDERR();     // 19 SIMD floating point error
extern void ENTRY_SYSCALL();     // 48 system call

void
trap_init(void)
{
        extern struct Segdesc gdt[];

        // LAB 3: Your code here.
        SETGATE(idt[T_DIVIDE ], 1, GD_KT, ENTRY_DIVIDE , 0);
        SETGATE(idt[T_DEBUG  ], 1, GD_KT, ENTRY_DEBUG  , 0);
        SETGATE(idt[T_NMI    ], 0, GD_KT, ENTRY_NMI    , 0);
        SETGATE(idt[T_BRKPT  ], 1, GD_KT, ENTRY_BRKPT  , 3);
        SETGATE(idt[T_OFLOW  ], 1, GD_KT, ENTRY_OFLOW  , 3);
        SETGATE(idt[T_BOUND  ], 1, GD_KT, ENTRY_BOUND  , 3);
        SETGATE(idt[T_ILLOP  ], 1, GD_KT, ENTRY_ILLOP  , 0);
        SETGATE(idt[T_DEVICE ], 1, GD_KT, ENTRY_DEVICE , 0);
        SETGATE(idt[T_DBLFLT ], 1, GD_KT, ENTRY_DBLFLT , 0);
        SETGATE(idt[T_TSS    ], 1, GD_KT, ENTRY_TSS    , 0);
        SETGATE(idt[T_SEGNP  ], 1, GD_KT, ENTRY_SEGNP  , 0);
        SETGATE(idt[T_STACK  ], 1, GD_KT, ENTRY_STACK  , 0);
        SETGATE(idt[T_GPFLT  ], 1, GD_KT, ENTRY_GPFLT  , 0);
        SETGATE(idt[T_PGFLT  ], 1, GD_KT, ENTRY_PGFLT  , 0);
        SETGATE(idt[T_FPERR  ], 1, GD_KT, ENTRY_FPERR  , 0);
        SETGATE(idt[T_ALIGN  ], 1, GD_KT, ENTRY_ALIGN  , 0);
        SETGATE(idt[T_MCHK   ], 1, GD_KT, ENTRY_MCHK   , 0);
        SETGATE(idt[T_SIMDERR], 1, GD_KT, ENTRY_SIMDERR, 0);
        SETGATE(idt[T_SYSCALL], 1, GD_KT, ENTRY_SYSCALL, 3);

        // Per-CPU setup
        trap_init_percpu();
}
```

Part B

Exercise 5&6

In kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
	// Handle processor exceptions.
	// LAB 3: Your code here.
	switch (tf->tf_trapno)
	{
		case T_PGFLT:
			page_fault_handler(tf);
			return;
		case T_BRKPT:
			monitor(tf);
			return;
		case T_SYSCALL:
```

Exercise 7

The entry of syscall in IDT can be seen in the answer of exercise 4.

In kern/trap.c, trap_dispatch()

```
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;
```

In kern/syscall.c

```
// Dispatches to the correct kernel function, passing the arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
	// Call the function corresponding to the 'syscallno' parameter.
	// Return any appropriate return value.
	// LAB 3: Your code here.

	switch (syscallno)
	{
	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((envid_t)a1);
	case SYS_map_kernel_page:
		return sys_map_kernel_page((void *)a1, (void *)a2);
	case SYS_sbrk:
		return sys_sbrk(a1);
	case NSYSCALLS:
	default:
		return -E_INVAL;
	}
}
```

Exercise 8

In kern/trapentry.S

Add sysenter_handler

```
.global sysenter_handler
.type sysenter_handler, @function
.align 2
sysenter_handler:
        pushl %esi
        pushl %edi
        pushl %ebx
        pushl %ecx
        pushl %edx
        pushl %eax
        call syscall
        movl %esi, %edx
        movl %ebp, %ecx
        sysexit
```

In kern/trap.c

Add declaration of sysenter handler

```
extern void sysenter_handler();
```

Trap_init()

```
wrmsr(0x174, GD_KT, 0);                // SYSENTER_CS_MSR
wrmsr(0x175, KSTACKTOP, 0);            // SYSENTER_ESP_MSR
wrmsr(0x176, sysenter_handler, 0);     // SYSENTER_EIP_MSR
```

In inc/x86.h

Add implementation of wrmsr

```
#define wrmsr(msr,val1,val2) \
        __asm__ __volatile__("wrmsr" \
        : /* no outputs */ \
        : "c" (msr), "a" (val1), "d" (val2))
```

In lib/syscall.c

```
  asm volatile(
        // not quite understand, aided by others
        // Store return %esp to %ebp, store return pc to %esi
        "pushl %%esp\n\t"
        "popl %%ebp\n\t"
        "leal after_sysenter_label%=, %%esi\n\t" // Use "%=" to generate a unique label number.
        "sysenter\n\t"
        "after_sysenter_label%=:\n\t"
        : "=a" (ret)
        : "a" (num),
          "d" (a1),
          "c" (a2),
          "b" (a3),
          "D" (a4),
        : "cc", "memory");
```

Exercise 9

In lib/libmain.c, libmain()

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

Exercise 10

In inc/env.h, struct Env
Add a new member to record current program's break

```
uintptr_t env_brk;              // current program's break
```

In kern/syscall.c

```
static int
sys_sbrk(uint32_t inc)
{
        // LAB3: your code here.
        // use region_alloc to allocate memory for environment
        region_alloc(curenv, (void *)(curenv->env_brk - inc), inc);
        curenv->env_brk = (uintptr_t)ROUNDDOWN(curenv->env_brk - inc, PGSIZE);
        return curenv->env_brk;         // return current break rather than previous one
}
```

Exercise 11

In kern/trap.c, page_fault_handler()
Use lowest 2 bits in tf_cs to check page fault

```
// Handle kernel-mode page faults.

// LAB 3: Your code here.
if (!(tf->tf_cs & 0x3))
{
        panic("In page_fault_handler: kernel page fault");
}
```

In kern/pmap.c

```c
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
        // LAB 3: Your code here.
        // first align start and end address so that every relevant pages can be involved
        uintptr_t begin = (uintptr_t)ROUNDDOWN(va, PGSIZE);
        uintptr_t end = (uintptr_t)ROUNDUP(va+len, PGSIZE);
        perm |= PTE_U;          // guarantee perm includes PTE_U
        for (uintptr_t i = begin; i < end; i += PGSIZE)
        {
                pte_t *pte;
                struct PageInfo *pp = page_lookup(env->env_pgdir, (void *)i, &pte);
                // every page needs to exist, be valid, and meet the requirements
                if (!pp || (*pte & PTE_P) == 0 || ((*pte & perm) == 0 && i >= ULIM))
                {
                        user_mem_check_addr = (i == begin) ? (uintptr_t)va : i;
                        return -E_FAULT;
                }
        }
        return 0;
}
```

In kern/syscall.c

```c
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
static void
sys_cputs(const char *s, size_t len)
{
        // Check that the user has permission to read memory [s, s+len).
        // Destroy the environment if not.

        // LAB 3: Your code here.
        user_mem_assert(curenv, (void *)s, len, PTE_U);

        // Print the string supplied by the user.
        cprintf("%.*s", len, s);
}
```

In kern/kdebug.c, debuginfo_eip()

```c
// 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), PTE_U))
{
        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, PTE_U) || user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U))
{
        return -1;
}
```

Exercise 13

In user/evilhello2.c

```c
// Invoke a given function pointer with ring0 privilege, then return to ring3
void ring0_call(void (*fun_ptr)(void)) {
    // Here's some hints on how to achieve this.
    // 1. Store the GDT descripter to memory (sgdt instruction)
    // 2. Map GDT in user space (sys_map_kernel_page)
    // 3. Setup a CALLGATE in GDT (SETCALLGATE macro)
    // 4. Enter ring0 (lcall instruction)
    // 5. Call the function pointer
    // 6. Recover GDT entry modified in step 3 (if any)
    // 7. Leave ring0 (lret instruction)

    // Hint : use a wrapper function to call fun_ptr. Feel free
    //        to add any functions or global variables in this
    //        file if necessary.

    // Lab3 : Your Code Here
    // step 1, store gdt in e_gdt
    struct Pseudodesc r_gdt;
    sgdt(&r_gdt);

    // step 2, map gdt to vaddr
    int t = sys_map_kernel_page((void* )r_gdt.pd_base, (void* )vaddr);
    if (t < 0)
    {
        cprintf("ring0_call: sys_map_kernel_page failed, %e\n", t);
    }

    // step 3
    // set the address of the callgate
    uint32_t base = (uint32_t)(PGNUM(vaddr) << PTXSHIFT);
    uint32_t index = GD_UD >> 3;
    uint32_t offset = PGOFF(r_gdt.pd_base);

    gdt = (struct Segdesc*)(base+offset);
    entry = gdt + index;
    old= *entry;

    // set up callgate
    // put call_fun_ptr (wrapper function) into entry, set privilege level to 3
    SETCALLGATE(*((struct Gatedesc*)entry), GD_KT, call_fun_ptr, 3);

    // step 4 and step 5
    asm volatile("lcall $0x20, $0");
}
```

```c
void call_fun_ptr()
{
    evil();

    // step 6 below
    *entry = old;

    // step 7 below
    asm volatile("leave");
    asm volatile("lret");
}
```