Exercise 1. Modify mem_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS so user processes can read from this array.

```
void mem_init(void)
{
    ……
    // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
    envs = boot_alloc(sizeof(struct Env) * NENV);
    ……
    // 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
    boot_map_region(kern_pgdir, UENVS, ROUNDUP(sizeof(struct Env) * NENV, PGSIZE), (physaddr_t)(PADDR(envs)), PTE_U | PTE_P);
    ……
}
```

需要 NENV 个 Env 结构，所以需要分配的大小为 sizeof(struct Env)*NENV)，使用 boot_alloc 分配

将这个得到的空间的头指针赋给全局变量 envs

并且用 boot_map_region 将 envs 数组映射到 NENV 处

需要注意的是，如果 envs 在 pages 之后被分配的话，需要修改 page_init，把 envs 的一部分作为已经分配的而不是加入 page_free_list

Exercise 2. In env.c, finish coding the following functions:

env_init()

Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).

env_setup_vm()

Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

region_alloc()

Allocates and maps physical memory for an environment

load_icode()

You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.

env_create()

Allocate an environment with env_alloc and call load_icode load an ELF binary into it.

env_run()

Start a given environment running in user mode.

// 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
    int i;
    // add all free env to env_free_list in the 0,1,2... order
    for (i = NENV-1; i > -1; i--) {
        envs[i].env_id = 0;
        envs[i].env_link = env_free_list;
        env_free_list = &envs[i];
    }
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

env_free_list 代表着可以分配的 env 的链表。

把 envs 数组中所有的元素加入 env_free_list

envs_free_list 是链表头为 envs[0]的正向链表

// Initialize the kernel virtual memory layout for environment e. Allocate a page directory, set e->env_pgdir accordingly, and initialize the kernel portion of the new environment's address space. Do NOT (yet) map anything into the user portion of the environment's virtual address space.

// Returns 0 on success, < 0 on error. Errors include:-E_NO_MEM if page directory or table could not be allocated.

```
static int env_setup_vm(struct Env *e)
{
   int i;
   struct Page *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.
   e->env_pgdir = page2kva(p);

   // Do NOT (yet) map anything into the user portion
   // UTOP 之下为 USER 自由管理，但 UTOP 之上是只读的
```

KERNEL 映射，或者不可读写的 KERNEL，所以从 UTOP 开始映射

```
        // NPDENTRIES number of page directory entries
#define NPDENTRIES 1024 in mmu.h
        // PDX(UTOP) page directory index of UTOP
        // env_pgdir[PDX(UTOP)] get page directory index of
UTOP, then find the content of this index in env_pgdir array
        for(i = PDX(UTOP); i < NPDENTRIES ; i++)
            e->env_pgdir[i] = kern_pgdir[i];
        p->pp_ref++;

        // 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;
}
```

首先 page_alloc 一个 page 作为 e 的页表目录（即 env_pgdir）

UTOP 之下为 USER 自由管理，全部是 0（在 page_alloc 中已经完成），但 UTOP 之上是只读的 KERNEL 映射，或者不可读写的 KERNEL，所以从 UTOP 开始映射 kern_pgdir 到 env_pgdir 中

最后 UPVT 的内容需要单独设置

// 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.

```
static void region_alloc(struct Env *e, void *va, size_t len)
{
    // You should round va down, and round (va + len) up. (Watch out for corner-cases!)

    void* i;
    for(i = ROUNDDOWN(va, PGSIZE); i < ROUNDUP(va+len, PGSIZE); i += PGSIZE)
    {
        struct Page* p = (struct Page*)page_alloc(1);
        if(p == NULL) panic("Memory out!");
        page_insert(e->env_pgdir, p, i, PTE_W | PTE_U);
    }
    e->va_start = (uint32_t)ROUNDDOWN(va, PGSIZE);
    e->va_len = (uint32_t)(ROUNDUP(va+len, PGSIZE) - ROUNDDOWN(va, PGSIZE));
}
```

利用 page_alloc 为 Env e 从虚拟地址 va 开始分配长度为 len 的空间，再将分配出的 p 利用 page_insert 函数插入到 env_pgdir 中虚拟地址为 i 的地方。以此实现为 e 分配空间的目的

va_start 与 va_len 是为了之后的 sys_sbrk 准备

// Set up the initial program binary, stack, and processor flags for a user process.

// This function loads all loadable segments from the ELF binary image into the environment's user memory, starting at the appropriate virtual addresses indicated in the ELF program header. At the same time it clears to zero any portions of these segments that are marked in the program header as being mapped but not actually present in the ELF file - i.e., the program's bss section.

// Finally, this function maps one page for the program's initial stack.

// load_icode panics if it encounters problems.

static void load_icode(struct Env *e, uint8_t *binary, size_t size)

{

// 把 ELF 中的 code 复制到 p_va 中，并通过映射 p_va 在 env_pgdir 中映射该 code

// 即把 ELF 中的 code 映射到 env_pgdir 中

// 最后为 stack 开出内存页表

// switch cr3(base of physical address) to e->env_pgdir

lcr3(PADDR(e->env_pgdir));

```
        struct Elf* ELFHDR = (struct Elf*)binary;
        // is this a valid ELF?
        if (ELFHDR->e_magic != ELF_MAGIC)
            panic("Not a elf binary");

        // ph: start of program header
        // eph: end of program header
        // e_phnum: program header number of envioronment
        // e_phoff: start of the program header offset from elf header
        struct Proghdr *ph = (struct Proghdr *)((uint8_t *) ELFHDR + ELFHDR->e_phoff);
        struct Proghdr *eph = ph + ELFHDR->e_phnum;

        // ph->p_va: segment's virtual address
        // ph->p_memsz: segment's size in memory
        for (; ph < eph; ph++)
            // only load segments with ph->p_type == ELF_PROG_LOAD
            if(ph->p_type == ELF_PROG_LOAD)
            {
```

```
            // set p_va to env_pgdir(so we can get p_pa) so that after we can copy to p_va, it can be found in env_pgdir
            region_alloc(e, (void*)ph->p_va, ph->p_memsz);
            // copy: ph->p_va = binary + ph->p_offset, copy_size: ph->p_filesz
            int i;
            char *va = (char*)ph->p_va;
            for(i = 0; i < ph->p_filesz; i++)
                va[i] = binary[ph->p_offset+i];
            // (The ELF header should have ph->p_filesz <= ph->p_memsz)
            // set the remaining bytes to 0
            for(; i < ph->p_memsz; i++)
                va[i] = 0;
        }

    // env_tf: envrionment's saved registers
    // set tf_eip to make sure environment start here
    e->env_tf.tf_eip = ELFHDR->e_entry;

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

```
        // alloc a new page and then insert it to env_pgdir
        struct Page* p = (struct Page*)page_alloc(1);
        if(p == NULL) panic("Not enough mem for user stack!");
        page_insert(e->env_pgdir, p, (void*)(USTACKTOP-
PGSIZE), PTE_W | PTE_U);

        // switch cr3(base of physical address) again to
kern_pgdir
        lcr3(PADDR(kern_pgdir));
}
```

把 ELF 中的 code 复制到 p_va 中，并通过映射 p_va 在 env_pgdir 中映射该 code，即把 ELF 中的 code 映射到 env_pgdir 中，最后为 stack 开出内存页表

先把 cr3（页表基地址）变为 env_pgdir

ph 是在 ELFHDR + ELFHDR->e_phoff 处，共有 e_phnum 个

对于每个 program header，检查是不是 ELF_PROG_LOAD，开出 p_memsz 的空间，把从 p_va 地址开始的 p_filesz 空间用 binary+ph->p_offset 填充，其余（p_memsz-p_filesz）用 0 填充

分配一个 USTACKTOP 处的 page，并插入到 env_pgdir 中

最后把 cr3 改回 kern_pgdir

// 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, size_t size, enum EnvType type)

{

struct Env* env;

// allocate a new env of which parent id is 0

if(env_alloc(&env,0)==0)

{

// set env_type to type

env->env_type=type;

// load code from binary(elf) to env

load_icode(env, binary,size);

}

}

用 env_alloc 创建一个 parent_id 为 0 的 env，设置其 type，且利用 load_icode 读取运行内容

// Context switch from curenv to env e. If this is the first call to env_run, curenv is NULL. This function does not return.

```
void env_run(struct Env *e)
{
  if(curenv != NULL)
      if(curenv->env_status == ENV_RUNNING)
          curenv->env_status = ENV_RUNNABLE;
  if(curenv != e)
  {
      curenv = e;
      e->env_status = ENV_RUNNING;
      e->env_runs++;
      lcr3(PADDR(e->env_pgdir));
  }
  env_pop_tf(&e->env_tf);
}
```

如果当前 env(curenv)存在，且状态为 ENV_RUNNING（运行状态），那么将其状态改为 ENV_RUNNABLE，即可运行状态

更改当前 env 为 e，更改 e 的状态为 ENV_RUNNING，更新 env_runs 计数器，将页表基地址变为 env_pgdir

用 env_pop_tf 存储 env 的寄存器，并进入用户态

Exercise 4. Edit trapentry.S and trap.c and implement the features described above. The macros TRAPHANDLER and TRAPHANDLER_NOEC in trapentry.S should help you, as well as the T_* defines in inc/trap.h. You will need to add an entry point intrapentry.S (using those macros) for each trap defined in inc/trap.h, and you'll have to provide _alltraps which the TRAPHANDLER macros refer to. You will also need to modify trap_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the SETGATE macro will be helpful here.

```
.text
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
/*
 * Vector    Description                Type       Error code
 * 0       Divide error                Fault       No
 * 1       Debug exception                         No
 * 2       Mon-maskable interrupt      Interrupt No
 * 3       Breakpoint                  Trap        No
```

```
* 4      Overflow                    Trap          No
* 5      Bounds check                Fault         No
* 6      Illegal opcode              Fault         No
* 7      Device not available        Fault         No
* 8      Double fault                Abort         Zero
* 9      Reserved
* 10     Invalid task switch segment Fault         Yes
* 11     Segment not present         Fault         Yes
* 12     Stack exception             Fault         Yes
* 13     General protection fault    Fault         Yes
* 14     Page fault                  Fault         Yes
* 15     Reserved
* 16     Floating point error        Fault         No
* 17     Aligment check              Fault         Zero
* 18     Machine check               Abort         No
* 19     SIMD floating point error   Fault         No
* 20-31  Reserved
* 32-255 User defined interrupts     Interrupt No
* 48      System call
*/

/* TRAPHANDLER_NOEC trap handler with no error code
```

```
*/
    TRAPHANDLER_NOEC(entry0, T_DIVIDE);
    TRAPHANDLER_NOEC(entry1, T_DEBUG);
    TRAPHANDLER_NOEC(entry2, T_NMI);
    TRAPHANDLER_NOEC(entry3, T_BRKPT);
    TRAPHANDLER_NOEC(entry4, T_OFLOW);
    TRAPHANDLER_NOEC(entry5, T_BOUND);
    TRAPHANDLER_NOEC(entry6, T_ILLOP);
    TRAPHANDLER_NOEC(entry7, T_DEVICE);
    TRAPHANDLER(entry8, T_DBLFLT);
    /* TRAPHANDLER(entry9, T_COPROC ); */
    TRAPHANDLER(entry10, T_TSS);
    TRAPHANDLER(entry11, T_SEGNP);
    TRAPHANDLER(entry12, T_STACK);
    TRAPHANDLER(entry13, T_GPFLT);
    TRAPHANDLER(entry14, T_PGFLT);
    /* TRAPHANDLER(entry15, T_RES);*/
    TRAPHANDLER_NOEC(entry16, T_FPERR);
    TRAPHANDLER(entry17, T_ALIGN);
    TRAPHANDLER_NOEC(entry18, T_MCHK);
    TRAPHANDLER_NOEC(entry19, T_SIMDERR );
```

```
_alltraps:
    /* push values to make the stack look like a struct Trapframe */
    pushw $0      #uint16_t tf_padding2
    pushw %ds  #uint16_t tf_ds
    pushw $0      #uint16_t tf_padding1
    pushw %es  #uint16_t tf_es
    pushal        #struct PushRegs tf_regs

    /* load GD_KD into %ds and %es */
    movl $GD_KD, %eax
    movw %ax, %ds
    movw %ax, %es

    /* pushl %esp to pass a pointer to the Trapframe as an argument to trap() */
    pushl %esp

    /* call trap */
    call trap
```

```
void trap_init(void)
{
    extern struct Segdesc gdt[];
    // declare idt
    extern void entry0();
    extern void entry1();
    extern void entry2();
    extern void entry3();
    extern void entry4();
    extern void entry5();
    extern void entry6();
    extern void entry7();
    extern void entry8();
    // extern void entry9();
    extern void entry10();
    extern void entry11();
    extern void entry12();
    extern void entry13();
    extern void entry14();
    // extern void entry15();
    extern void entry16();
    extern void entry17();
```

```
    extern void entry18();
    extern void entry19();

    // initialize the idt to point to each of these entry points
    // define SETGATE(gate, istrap, sel, off, dpl)
    // - sel: Code segment selector for interrupt/trap handler
    // - off: Offset in code segment for interrupt/trap handler
    // - dpl: Descriptor Privilege Level -
    // GD_KT: kernel text
    SETGATE(idt[T_DIVIDE],  0, GD_KT, entry0,  0);
    SETGATE(idt[T_DEBUG],   0, GD_KT, entry1,  0);
    SETGATE(idt[T_NMI],     0, GD_KT, entry2,  0);
    SETGATE(idt[T_BRKPT],   0, GD_KT, entry3,  3);   // user
mode ring3
    SETGATE(idt[T_OFLOW],   0, GD_KT, entry4,  0);
    SETGATE(idt[T_BOUND],   0, GD_KT, entry5,  0);
    SETGATE(idt[T_ILLOP],   0, GD_KT, entry6,  0);
    SETGATE(idt[T_DEVICE],  0, GD_KT, entry7,  0);
    SETGATE(idt[T_DBLFLT],  0, GD_KT, entry8,  0);
    // SETGATE(idt[T_COPROC],     0, GD_KT, entry9,  0);
    SETGATE(idt[T_TSS],     0, GD_KT, entry10,     0);
    SETGATE(idt[T_SEGNP],   0, GD_KT, entry11,     0);
```

```
    SETGATE(idt[T_STACK],    0, GD_KT, entry12,      0);
    SETGATE(idt[T_GPFLT],    0, GD_KT, entry13,      0);
    SETGATE(idt[T_PGFLT],    0, GD_KT, entry14,      0);
    // SETGATE(idt[T_RES],         0, GD_KT, entry15,      0);
    SETGATE(idt[T_FPERR],    0, GD_KT, entry16,      0);
    SETGATE(idt[T_ALIGN],    0, GD_KT, entry17,      0);
    SETGATE(idt[T_MCHK],     0, GD_KT, entry18,      0);
    SETGATE(idt[T_SIMDERR],     0, GD_KT, entry19,      0);

    // Per-CPU setup
    extern void sysenter_handler();
    wrmsr(0x174, GD_KT, 0);
    wrmsr(0x175, KSTACKTOP, 0);
    wrmsr(0x176, sysenter_handler, 0);
    // Per-CPU setup
    trap_init_percpu();
    cprintf("Trap init end.\n");
}
```

在 trapentry.S 中按照表（如前所述）声明 idt 中各项的处理函数（entry0—entry19），且在 trap.c 中外部引用，之后用 SETGATE 设置其 gate，是否是 trap，sel（为 GD_KT，KERNEL TEXT），offset 为处理函数位置，dpl

其中 breakpoint 的 dpl 为 3（即 user 触发），其余都是 kernel 触发，为 0（breakpoint 相关问题在 Exercise9 中得见）

_alltraps 为 entry 函数的一部分，先依次压入 Trapframe 的成员，使其像一个 Trapframe 结构，再把 GD_KT（KERNEL TEXT 断）位置读到 ds 和 es 寄存器中（SETGATE 的代码段起始）把 TrapFrame 压入栈中当做 trap 函数的参数，最后调用 trap()函数

Exercise 5. Modify trap_dispatch() to dispatch page fault exceptions topage_fault_handler().

```
static void trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    switch(tf->tf_trapno) {
        case T_DEBUG:
        case T_BRKPT:
            monitor(tf);
            break;
        case T_PGFLT:
            page_fault_handler(tf);
    }
    // Unexpected trap: The user or the kernel has a bug.
    print_trapframe(tf);
    if (tf->tf_cs == GD_KT) panic("unhandled trap in kernel");
    else {
        env_destroy(curenv);
        return;
    }
}
```

检查 tf_trapno 即 trap number 是否为 T_PGFLT(page fault)，如果是就调用 page_fault_handler

Exercise 6. Implement system calls using the sysenter and sysexit instructions.

```
.globl sysenter_handler;
.type sysenter_handler, @function;
.align 2;
sysenter_handler:
    pushl $GD_UD|3
    pushl %ebp
    pushfl
    pushl $GD_UT|3
    pushl %esi
    pushl $0
    pushl $0
    pushl %ds
    pushl %es
    pushal
    movw $GD_KD, %ax
    movw %ax, %ds
    movw %ax, %es
    pushl %esp
    call my_syscall
```

```
    popl %esp
    popal
    popl %es
    popl %ds
    movl %ebp, %ecx
    movl %esi, %edx
    sysexit

// 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.
    switch (syscallno){
        case SYS_getenvid:
            return sys_getenvid();
        case SYS_cputs:
            sys_cputs((const char*) a1,a2);
            return 0;
        case SYS_cgetc:
```

```
            return sys_cgetc();
        case SYS_env_destroy:
            return sys_env_destroy(a1);
        case SYS_map_kernel_page:
            return sys_map_kernel_page((void *)a1, (void *)a2);
        case SYS_sbrk:
            return sys_sbrk(a1);
        default:
            return -E_INVAL;
    }
}

void my_syscall(struct Trapframe *tf){
    curenv->env_tf = *tf;
    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,0);
    return;
}
```

```
static inline int32_t
syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3,
uint32_t a4, uint32_t a5)
{
    int32_t ret;
    asm volatile("pushl %%ecx\n\t"
        "pushl %%edx\n\t"
            "pushl %%ebx\n\t"
        "pushl %%esp\n\t"
        "pushl %%ebp\n\t"
        "pushl %%esi\n\t"
        "pushl %%edi\n\t"

                "movl %%esp,%%ebp\n\t"
                "leal 1f, %%esi\n\t"
                "sysenter\n\t"
                "1:\n\t"

                "popl %%edi\n\t"
                "popl %%esi\n\t"
                "popl %%ebp\n\t"
                "popl %%esp\n\t"
```

```
                "popl %%ebx\n\t"
                "popl %%edx\n\t"
                "popl %%ecx\n\t"

                : "=a" (ret)
                : "a" (num),
                  "d" (a1),
                  "c" (a2),
                  "b" (a3),
                  "D" (a4)
                : "cc", "memory");


    if(check && ret > 0)
        panic("syscall %d returned %d (> 0)", num, ret);

    return ret;
}
```

```
#define wrmsr(msr,val1,val2) \
    __asm__ __volatile__("wrmsr" \
    : \
    : "c" (msr), "a" (val1), "d" (val2))

void trap_init(void)
{
    ......
    extern void sysenter_handler();
    wrmsr(0x174, GD_KT, 0);
    wrmsr(0x175, KSTACKTOP, 0);
    wrmsr(0x176, sysenter_handler, 0);
    ......
}
```

sysenter_handler 与之前注册的 entry 类似，只是最后不是调用 trap 而是调用 my_syscall，调用完以后 pop，并 sysexit

my_syscall 修改当前 env 的 env_tf，并调用 syscall，把返回值放进 reg_eax 中

wsmr 修改寄存器的值

lib/syscall.c 中的 syscall 将各寄存器的值赋给各个参数

kern/syscall.c 中的 syscall 根据想要的 syscallno 调用相应的函数

Exercise 7. Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys_env_destroy().

```
void libmain(int argc, char **argv)
{
    // set thisenv to point at our Env structure in envs[].
    // The environment index ENVX(eid) equals the environment's offset in the 'envs[]' array.
    thisenv = envs+ENVX(sys_getenvid());
    // save the name of the program so that panic() can use it
    if (argc > 0)
        binaryname = argv[0];

    // call user main routine
    umain(argc, argv);

    // exit gracefully
    exit();
}
```

利用 sys_getenvid()获取当前 env 的 env_id 并利用 ENVX 找出其在 envs 中的偏移量，并赋值给 thisenv，使得 thisenv 不为空

Exercise 8. You need to write syscall sbrk. The sbrk(), as described in the manual page (man sbrk), extends the size of a process's data segment (heap). It dynamically allocates memory for a program. Actually, the famous malloc allocates memory in the heap using this syscall.

As int sys_sbrk(uint32_t increment);shows, sbrk() increase current program' data space by increment bytes. On success, sbrk() returns the current program 's break after being increased. NOTE: it is different from the standard behavior of sbrk().

For the implementation, you just need to allocate multiple pages and insert them into the correct positions in page table, growing the heap higher. The load_icode() may act as a hint. You also might need to modify struct Env to record the current program's break and update them accordingly in sbrk().

```
static int sys_sbrk(uint32_t inc)
{
    sbrk_alloc(curenv, (void*)(curenv->va_start + curenv->va_len), inc);
    return curenv->va_start;
}
```

```
void sbrk_alloc(struct Env *e, void *va, size_t len)
{
    region_alloc(e,va,len);
}

struct Env {
    struct Trapframe env_tf;   // Saved registers
    struct Env *env_link;        // Next free Env
    envid_t env_id;            // Unique environment identifier
    envid_t env_parent_id;          // env_id of this env's parent
    enum EnvType env_type;     // Indicates special system
environments
    unsigned env_status;       // Status of the environment
    uint32_t env_runs;      // Number of times environment has
run
    uint32_t va_start;
    uint32_t va_len;
    // Address space
    pde_t *env_pgdir;       // Kernel virtual address of page dir
};
```

region_alloc 为 static 函数，不能外部调用，所以用 sbrk_alloc 包装，并外部引用调用

利用 va_start 和 va_len 记录上一次 region_alloc 时分配起始地址和分配长度，使得在调用 sys_sbrk 时可以确定分配的起始位置为 va_start+va_len，且最后返回这次分配完以后的 va_start

Exercise 9. Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test. After that , you should modify the JOS kernel monitor to support GDB-style debugging commands c, si and x.

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "time","Display time the function need", mon_time },
    { "c", "Continue to run", mon_c},
    { "si", "Step one", mon_si},
    { "x", "", mon_x}
};

int mon_c(int argc, char **argv, struct Trapframe *tf)
{
    if (tf == NULL) {
        cprintf("No trapped environment\n");
        return 1;
    }
```

```
    // make trap frame's eflag be not traped
    tf->tf_eflags = tf->tf_eflags & (~FL_TF);
    env_run(curenv);
    return 0;
}

int mon_si(int argc, char **argv, struct Trapframe *tf)
{
    if (tf == NULL) {
        cprintf("No trapped environment\n");
        return 1;
    }

    // make trap frame's eflag be traped
    // FL_TF Eflags register trap flag
    tf->tf_eflags = tf->tf_eflags | FL_TF;

    cprintf("tf_eip=0x%x\n", tf->tf_eip);
    env_run(curenv);
    return 0;
}
```

```
int mon_x(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 2) cprintf("please enter x addr");

    // read argv[1] as 16-base
    uint32_t addr = strtol(argv[1], NULL, 16);
    // get the value in address addr
    uint32_t val = *(int *)addr;
    cprintf("%d\n", val);
    return 0;
}
```

trap_dispatch 中加入是否是 T_DEBUG 或 T_BRKPT 的判断，如果是，则进入 monitor

mon_c 与 mon_si 类似 ,只是 mon_c 将 eflag 的 trap_flag 置为 ~FL_TF ,也就是不被 trap ,而 mon_si 将其置为 FL_TF ,也就是仍然被 trap 住，然后 run_env

mon_x 以 16 进制读取参数变为指针，并取出指针指向的值打印

Exercise 10,11. Change kern/trap.c to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the tf_cs.

Read user_mem_assert in kern/pmap.c and implement user_mem_check in that same file.

Change kern/syscall.c to sanity check arguments to system calls.

Finally, change debuginfo_eip in kern/kdebug.c to call user_mem_check on usd, stabs, and stabstr. If you now run user/breakpoint, you should be able to runbacktrace from the kernel monitor and see the backtrace traverse into lib/libmain.cbefore the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens.

```
void page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // Trapped from kernel mode, panic
    // CPL 是 CS 的最低两位
    if((tf->tf_cs & 3) != 3) panic("Kernel page fault!");

    // We've already handled kernel-mode exceptions, so if we get here, the page fault happened in user mode.

    // Destroy the environment that caused the fault.
    cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}
```

```
int user_mem_check(struct Env *env, const void *va, size_t len,
int perm)
{
        // with permissions 'perm | PTE_P', and normally 'perm'
will contain PTE_U at least
    perm = perm | PTE_U | PTE_P;
    uintptr_t i = (uintptr_t)ROUNDDOWN(va,PGSIZE);

    for (; i < (uintptr_t)ROUNDUP(va + len, PGSIZE); i +=
PGSIZE )
    {
        // check if the address is below ULIM
        if ((uint32_t)i >= ULIM)
        {
            // for we need to know the first virtual address cause
fail, then if the fist is not aligned, since we have used
ROUWNDDOWN, we have to change it back to the first
            if(i < (uintptr_t)va) i = (uintptr_t)va;
            // set the 'user_mem_check_addr' variable to the first
erroneous virtual address
            user_mem_check_addr = (uintptr_t)i;
            return -E_FAULT;
```

```
        }
        // find the page of va
        pte_t *pte = pgdir_walk (env->env_pgdir, (void*)i, 0);
        // check if the page table gives it permission
        if (pte == NULL || (*pte & perm) != perm)
        {
            if(i < (uintptr_t)va) i = (uintptr_t)va;
            user_mem_check_addr = (uintptr_t)i;
            return -E_FAULT;
        }
    }
    // the user program can access this range of addresses
    return 0;
}
```

```
// 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.
    user_mem_assert(curenv, (void*)s, len, PTE_U | PTE_P);
    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

```
int debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info)
{
    ......
    const struct UserStabData *usd = (const struct UserStabData *) USTABDATA;

    // Make sure this memory is valid.
    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.
    if (!user_mem_check(curenv, stabs, stab_end-stabs, PTE_U)) return -1;
    if (!user_mem_check(curenv, stabstr, stabstr_end-stabstr, PTE_U)) return -1;
    ......
}
```

page_fault_handler 中检查 tf_cs 的后二位是不是 3（user mode），不是说明 kernel 调用的 page_fault_handler，则 panic

user_mem_check 从 va 开始检查 len 长度的正确性，检查其是否超过 ULIM，且是否具有 perm（即检查 pgdir_walk 得到的 page 中权限是否相等）

由于传入的 va 不一定是对齐的，所以从 va 所在页表的第一位开始（ROUNDDOWN），但是需要知道第一个出问题的 va，如果是在该页查出问题，那么需要将其换回 va（否则就是需要检查的内存之外的内存出问题了）

Exercise 12. evilhello2.c want to perform some privileged operations in functionevil(). Function ring0_call() takes an function pointer as argument. It calls the provided function pointer with ring0 privilege and then return to ring3. There's few ways to achieve it. You should follow the instructions in the comments to enter ring0.

sgdt is an unprivileged instruction in x86 architecture. It stores the GDT descripter into a provided memory location. After mapping the page contains GDT into user space, we could setup an callgate in GDT. Callgate is one of the cross-privilege level control transfer mechanisms in x86 architecture. After setting up the call gate. Applications may use lcall (far call) instruction to call into the segment specified in callgate entry (For example, kernel code segment). After that, lret instruction could be used to return to the original segment. For more information on Callgate. Please refer to intel documents.

Finish ring0_call() and run evilhello2.c, you should see IN RING0!!! followed by a page fault. (the function evil() is called twice, one in ring0 and one in ring3).

To make your life easier, some utility macros and data structure are provided inmmu.h. (SETCALLGATE, SEG, struct Pseudodesc, struct Gatedesc ...) You could use them to manage GDT.

```
char vaddr[PGSIZE];
struct Segdesc old, *gdt, *entry;

void call_fun_ptr()
{
    // 5. Call the function pointer
    evil();
    // 6. Recover GDT entry modified in step 3 (if any)
    *entry = old;
    // 7. Leave ring0 (lret instruction)
    asm volatile("popl %ebp");
    asm volatile("lret");
}

// Invoke a given function pointer with ring0 privilege, then return to ring3
void ring0_call(void (*fun_ptr)(void)) {
     struct Pseudodesc gdtd;
     // 1. Store the GDT descripter to memory (sgdt instruction)
    sgdt(&gdtd);

    // 2. Map GDT in user space (sys_map_kernel_page)
```

```
    // User applications in JOS could map any kernel page into userspase using sys_map_kernel_page
    int t = sys_map_kernel_page((void* )gdtd.pd_base, (void* )vaddr);
    if (t < 0) cprintf("ring0_call: sys_map_kernel_page failed, %e\n", t);

    uint32_t base = (uint32_t)(PGNUM(vaddr) << PTXSHIFT);
    uint32_t index = GD_UD >> 3;
    uint32_t offset = PGOFF(gdtd.pd_base);

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

    // 3. Setup a CALLGATE in GDT (SETCALLGATE macro)
    SETCALLGATE(*((struct Gatedesc*)entry), GD_KT, call_fun_ptr, 3);
    // 4. Enter ring0 (lcall instruction)
    asm volatile("lcall $0x20, $0");
}
```

根据提示一步一步走，先取出 GDT 中的值，将其映射到 user 可访问处，设置 CALLGATE，将权限改为 3（user mode 可访问），进入 ring0，调用 evil 函数，再把 GDB 中的 entry 改回，并返回 ring3