به نام خدا

گزارش Lab3 سیستم عامل پیشرفته

استاد:

دکتر جوادی

دانشجویان:

سید علیرضا حسینی – ۴۰۱۱۳۱۰۰۵

امیررضا زارع – ۴۰۱۱۳۱۰۰۸

فهرست مطالب

مقدمه

در این آزمایش، امکانات اصلی هسته مورد نیاز برای اجرای یک محیط حالت کاربر محافظت شده را پیاده‌سازی خواهیم کرد و هسته JOS را برای راه‌اندازی ساختارهای داده برای پیگیری محیط‌های کاربر، ایجاد یک محیط کاربری واحد، بارگذاری یک ایمیج از برنامه در آن و شروع اجرای آن تقویت خواهیم کرد. همچنین می‌توانیم هسته JOS را قادر کنیم به مدیریت هر فراخوانی سیستمی که محیط کاربر ایجاد می‌کند و هرگونه استثنای دیگری را که ایجاد می‌کند مدیریت کند. در این آزمایش قصد داریم یک محیط محافظت شده‌ی سطح کاربر را پیاده‌سازی کنیم و ببینیم چگونه فرایندها و همچنین وقفه‌ها و فراخوانی‌های سیستمی در سطح کاربر را به چه شکل مدیریت و محافظت می‌کند.

Lab3 شامل یک سری فایل‌های جدید است که در تصویر زیر قابل مشاهده است:

inc/	env.h	Public definitions for user-mode environments
	trap.h	Public definitions for trap handling
	syscall.h	Public definitions for system calls from user environments to the kernel
	lib.h	Public definitions for the user-mode support library
kern/	env.h	Kernel-private definitions for user-mode environments
	env.c	Kernel code implementing user-mode environments
	trap.h	Kernel-private trap handling definitions
	trap.c	Trap handling code
	trapentry.S	Assembly-language trap handler entry-points
	syscall.h	Kernel-private definitions for system call handling
	syscall.c	System call implementation code
lib/	Makefrag	Makefile fragment to build user-mode library, obj/lib/libuser.a
	entry.S	Assembly-language entry-point for user environments
	libmain.c	User-mode library setup code called from entry.S
	syscall.c	User-mode system call stub functions
	console.c	User-mode implementations of putchar and getchar, providing console I/O
	exit.c	User-mode implementation of exit
	panic.c	User-mode implementation of panic
user/	*	Various test programs to check kernel lab 3 code

جاس برای این آزمایش سه تا متغیر گلوبال را معرفی کرده است:

```
struct Env *envs = NULL;                /* All environments */
struct Env *curenv = NULL;              /* the current env */
static struct Env_list env_free_list;   /* Free list */
```

جاس از یک env_free_list برای ردیابی environmentها استفاده می‌کند که این امکان را به ما می‌دهد که اختصاص دادن و پس گرفتن را به صورت اضافه کردن و حذف کردن از یک لیست باشد.

ساختار Env که در فایل inc/env.h قابل مشاهده است به صورت زیر است:

```
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 systemenvironments
        unsigned env_status;            // Status of the environment
        uint32_t env_runs;              // Number of times environment hasrun

        // Address space
        pml4e_t *env_pml4e;
        // Kernel virtual address of page map level-4
};
```

تمرین ۱:

Exercise 1. Modify `x64_vm_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` (defined in `inc/memlayout.h`) so user processes can read from this array.

You should run your code and make sure `check_boot_pml4e()` succeeds.

در فایل kern/pmap.c وارد تابع x64_vm_init() شدیم این خط را اضافه کردیم:

```
// Your code goes here:

boot_map_region(boot_pml4e, UPAGES, page_size, PADDR(pages), PTE_U);
```

تمرین ۲:

Exercise 2. In the file `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 map level four (pml4e) for a new environment and initialize the kernel portion of the new environment's address space. Note that in the previous lab, we set up our memory mapping such that everything above UTOP is in the second entry of the PML4. Everything above UTOP needs to be mapped into every environment's virtual space. This involves simple copying of this entry form kernel's pml4 to the environment's pml4.

`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.

As you write these functions, you might find the new cprintf verb `%e` useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

در این تمرین در فایل env.c توابع زیر را کامل کردیم.

Env_init()

Env_setup_vm()

Region_alloc()

Load_icode()

Env_create()

Env_run()

در فانکشن env_init() تمام ساختارهای Env را در آرایه envs مقداردهی کنید و آن‌ها را به list_free_env اضافه کنید. همچنین percpu_init_env را فراخوانی می‌کند که سخت‌افزار تقسیم‌بندی را با بخش‌های جداگانه برای سطح صفر یا هسته و سطح ۳ یا همان کاربر پیکربندی می‌کنیم. هر env فضای آدرس مستقل خودش را دارد و هدف ما در جاس با ایجاد جداول صفحه مختلف pml4 محقق می‌شود.

تابع env_init() را در تصویر زیر مشاهده می‌کنید.

```
void
env_init(void)
{
    int temp;

    // Set up envs array
    for (temp = NENV - 1; temp >= 0; temp--) {
        envs[temp].env_status = ENV_FREE;
        envs[temp].env_id = 0;
        envs[temp].env_link = env_free_list;
        env_free_list = &envs[temp];
    }

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

در فانکشن env_setup_vm() ، یک pml4e را برای یک محیط جدید اختصاص می‌دهیم و بخش هسته فضای آدرس محیط جدید را مقداردهی اولیه می‌کنیم. در آزمایش قبلی، نگاشت حافظه خود را به گونه‌ای تنظیم کردیم که همه چیز بالای UTOP در ورودی دوم PML4 باشد. همه چیز بالای UTOP باید در فضای مجازی هر محیطی نگاشت شود که شامل کپی ساده pml4 هسته این فرم ورودی به pml4 محیط است. کد این فانکشن را در تصویر زیر مشاهده می‌کنید.

```
static int env_setup_vm(struct Env *e)
{
    int temp;
    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_pml4e 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.
    //  Hint: Figure out which entry in the pml4e maps addresses
    //        above UTOP.
    //  (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_pml4e
    //  is an exception -- you need to increment env_pml4e'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_pml4e = page2kva(p);
    e->env_cr3 = page2pa(p);
```

```
    for (temp = PML4(UTOP); temp < NPMLENTRIES; temp++)
        e->env_pml4e[temp] = boot_pml4e[temp];

    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R
    e->env_pml4e[PML4(UVPT)] = e->env_cr3 | PTE_P | PTE_U;

    return 0;
}
```

فانکشن Region_alloc() به منظور تسهیل در اختصاص آدرس فیزیکی به آدرس مجازی پیاده سازی شده که حافظه فیزیکی را برای یک محیط تخصیص داده و map کردیم. کد این فانکشن را در تصویر زیر مشاهده می‌کنید.

```
//
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!)
    void *start = ROUNDDOWN(va, PGSIZE);
    void *end = ROUNDUP(va + len, PGSIZE);
    for(; start < end; start += PGSIZE) {
        struct PageInfo *pp = page_alloc(0);
        if (pp) {
            pp->pp_ref++;
            int ret = page_insert(e->env_pml4e, pp, start, PTE_W | PTE_U);
            if (ret < 0) {
                panic("region_alloc: %e \n", ret);
            }
        } else {
            panic("region_alloc: failed to allocate a page \n");
        }
    }
}
```

چون لب۳ از فایل سیستم پشتیبانی نمی‌کند، صحت با اجرای مستقیم برنامه مشخص شده پس از تغییر به حالت کاربر تأیید می‌شود. باید برنامه ELF را که جاس از قبل آماده کرده است را بخوانیم و سپس اطلاعات مربوطه را در ELF درenv_ft قرار دهیم. در فانکشن Load_icode() یک ELF binary image را parse کردیم، دقیقاً مانند آنچه که بوت لودر قبلا انجام میشد و محتویات آن را در فضای آدرس کاربر یک محیط جدید بارگذاری کردیم. کد این فانکشن را در تصویر زیر مشاهده می‌کنید.

```
void
load_icode(struct Env *e, uint8_t *binary)
{
    // LAB 3: Your code here.
    e->elf = binary;

    struct Proghdr *program_header, *end_program_header;
    struct Elf *elf = (struct Elf *) binary;

    program_header = (struct Proghdr *) (binary + elf->e_phoff);
    end_program_header = program_header + elf->e_phnum;

    lcr3(e->env_cr3);
    region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);

    for (; program_header < end_program_header; program_header++) {
        if (program_header->p_type == ELF_PROG_LOAD) {
            region_alloc(e, (void *) program_header->p_va, program_header->p_memsz);
            memmove((void *) program_header->p_va, (void *)binary + program_header->p_offset, program_header->p_filesz);
            memset((void *)program_header->p_va + program_header->p_filesz, 0, program_header->p_memsz - program_header->p_filesz);
        }
    }
    e->env_tf.tf_rip = elf->e_entry;
}
```

آخرین مرحله ایجاد یک Env جدید و اجرای فرآیند تغییر از حالت هسته Env به حالت کاربر Env است. در فانکشن ()Env_create یک محیط را با env_alloc اختصاص دادیم و load_icode را فراخوانی کردیم و یک باینری ELF را در آن بارگذاری کردیم. کد این فانکشن را در تصویر زیر مشاهده می‌کنید.

```
//
// 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* envStruct;
    int temp;

    if ((temp = env_alloc(&envStruct, 0)) != 0) {
        panic("env_create: %e", temp);
    }
    load_icode(envStruct, binary);
    envStruct->env_type = type;
}
```

هنگام اجرای فرآیند سوئیچینگ Env لازم است تا وضعیت Env را تایید و تنظیم کنیم در فانکشن ()Env_run یک محیط معین را در حالت کاربر شروع کردیم. کد این فانکشن را در تصویر زیر مشاهده می‌کنید.

```
void env_run(struct Env *env) {
    // 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.
    if (env != curenv && env->env_status != ENV_RUNNABLE)
        panic("the env could not run");
    if (curenv && curenv->env_status == ENV_RUNNING)
        curenv->env_status = ENV_RUNNABLE;
    curenv = env;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(curenv->env_cr3);
    env_pop_tf(&(curenv->env_tf));
}
```

تمرین ۳:

Exercise 3. Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

در این تمرین فقط لازم بود تا کمی در مورد Exception و interrupt مطالعه کنیم که انجام شد.

تمرین ۴:

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 in `trapentry.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.

Hint: your `_alltraps` should:

1. push values to make the stack look like a struct Trapframe
2. load `GD_KD` into `%ds` and `%es`
3. Pass a pointer to the Trapframe as an argument to trap() (Hint: Review the x64 calling convention from lab 1).
4. `call trap` (can `trap` ever return?)

Consider using the `PUSHA` and `POPA` macros; they fit nicely with the layout of the `struct Trapframe`.

Be sure to initialize every possible trap entry to a default handler (T_DEFAULT).

Test your trap handling code using some of the test programs in the `user` directory that cause exceptions before making any system calls, such as `user/divzero`. You should be able to get `make grade` to succeed on the `divzero`, `softint`, and `badsegment` tests at this point.

Important note: Be very careful not to declare your trap handling functions (defined by a `TRAPHANDLER` or `TRAPHANDLER_NOEC` macro) with a name already in use in the code. In your C code, these will likely be declared as `extern`, which disables the compiler checking for conflicting symbol names---the compiler instead just picks the first instance of a symbol it finds.

For instance, the function `breakpoint` is already defined in `inc/x86.h`, and the compiler will not warn you about this if you also call your breakpoint trap handler `breakpoint`; rather, the compiler will likely pick the wrong function. For this reason, one strategy might be to pick a random prefix for all traphandlers, like `XTRPX_divzero`, `XTRPX_pgfault`, etc.

سوال ۴.۱: هدف از داشتن یک تابع کنترل کننده جداگانه برای هر Exception/وقفه چیست؟ (به عنوان مثال، اگر همه استثناها/وقفه ها به یک کنترل کننده تحویل داده شوند، چه ویژگی ای که در اجرای فعلی وجود دارد نمی تواند ارائه شود؟)

پاسخ:

دو دلیل برای اینکار وجود دارد:

- برای پوش کردن کد خطای مربوطه روی پشته. این برای کدهایی استفاده می‌شود که قرار است آن را مانند trap_dispatch() مدیریت کنند تا وقفه‌ها را تشخیص دهند.
- برای ارائه کنترل یا جداسازی دسترسی‌ها یا مجوزها. برای هر کنترل‌کننده وقفه مستقل، می‌توانیم تعریف کنیم که آیا می‌تواند توسط یک برنامه کاربر راه‌اندازی شود یا نه. با قرار دادن چنین محدودیت‌هایی برای کنترل‌کننده‌های وقفه می‌توانیم اطمینان حاصل کنیم که برنامه‌های کاربر با هسته تداخل نخواهند کرد، هسته را خراب نمی‌کنند یا حتی کنترل کل رایانه را در دست نمی‌گیرند.

اگر برای همه‌ی وقفه‌ها/استثناها فقط یک هندل کننده یا کنترل کننده داشته باشیم، نمیشد فهمید که کدام وقفه رخ داده است. زیرا X86 بردار وقفه را که کنترل‌کننده را راه‌اندازی کرده است، پوش نمی‌کند.

سوال ۴.۲: آیا برای اینکه کاربر/برنامه نرم افزاری به درستی رفتار کند باید کاری انجام دهید؟ اسکریپت درجه انتظار دارد که یک خطای حفاظتی کلی ایجاد کند (تله ۱۳)، اما کد softint می گوید int $14 . چرا این باید بردار وقفه ۱۳ را تولید کند؟ اگر هسته واقعاً به دستور int $14 softint اجازه دهد تا کنترل کننده خطای صفحه هسته (که بردار وقفه ۱۴ است) را فراخوانی کند، چه اتفاقی می افتد؟

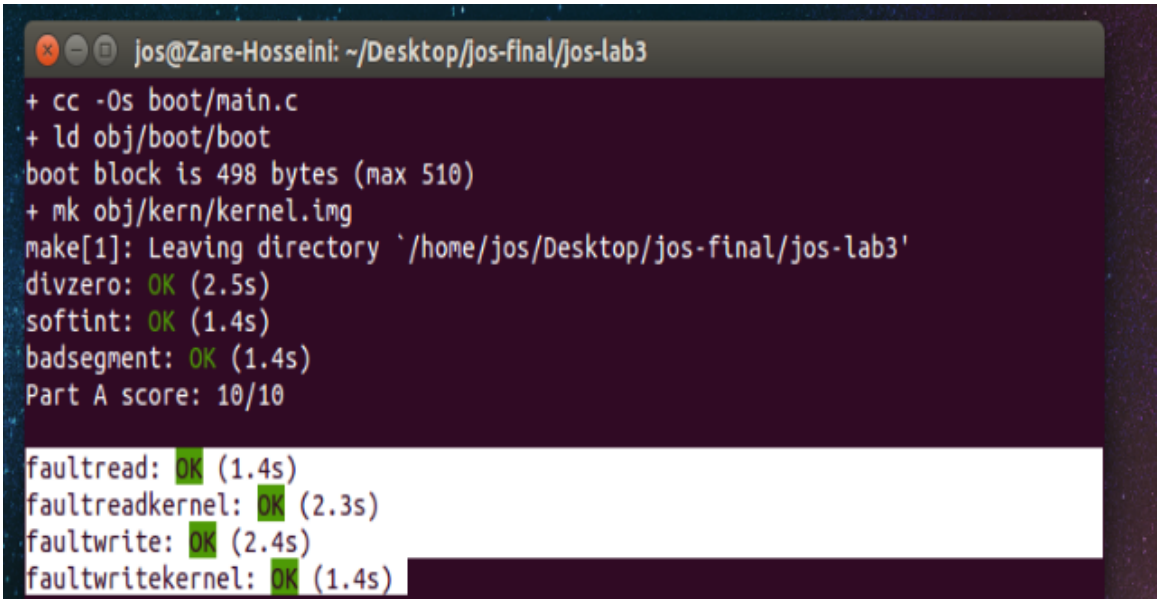
پاسخ:

برنامه فضای کاربر قرار نیست بتواند وقفه ۱۴ را راه‌اندازی کند (خطای صفحه). ما صراحتاً از کاربر منع می‌کنیم که این وقفه‌های تولید شده توسط CPU را به طور مصنوعی (با استفاده از دستورالعمل int) با تنظیم dpl روی 0 در interrupt gate ایجاد کند. این بدان معنی است که اگر برنامه فضای کاربر بطور مصنوعی وقفه‌ای صادر کند که مجاز به آن نیست، در عوض یک خطای حفاظتی ایجاد می‌شود (وقفه ۱۳). اگر هسته واقعاً اجازه دهد که این وقفه توسط برنامه فضای کاربر ایجاد شود، کنترل‌کننده خطای صفحه اجرا می‌شود. با این حال، یک مقدار بالقوه نامعتبر ممکن است در cr2 (آدرس مجازی که باعث خطا شده) ذخیره شود و بسته به نحوه پیاده‌سازی کنترل کننده خطای صفحه، این امر به طور بالقوه می‌تواند باعث تخصیص صفحاتی شود که فرآیند کاربر در غیر این صورت مجاز به انجام آن نیست. یا برخی از اشکالات هسته ناخواسته دیگر که می‌توانند امنیت یا ثبات سیستم را به خطر بیندازند. پس اگر اجازه دهیم خطای صفحه توسط یک برنامه کاربری مانند softint راه‌اندازی شود. می‌تواند حافظه مجازی را دستکاری کند و ممکن است مشکلات امنیتی جدی ایجاد کند.

تمرین ۵:

Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get `make grade` to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using `make run-x` or `make run-x-nox`.

در این بخش ما trap_dispatch() را تغییر دادیم تا خطاهای صفحه به page_fault_handler() ارسال شود. همچنین make grade گرفتیم که در تصویر زیر مشاهده می‌کنید که همه‌ی موارد خواسته شده کامل هستند و تایید شده است.



تمرین ۶:

Exercise 6. 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.

سوال ۶.۱: مورد تست نقطه شکست یا یک استثنا نقطه شکست یا یک خطای حفاظتی کلی ایجاد می کند بسته به اینکه چگونه ورودی نقطه شکست را در IDT مقداردهی اولیه کرده اید. چگونه باید آن را راه‌اندازی کنید تا استثناء نقطه شکست همانطور که در بالا مشخص شده است کار کند و چه تنظیمات نادرستی باعث ایجاد یک خطای حفاظتی عمومی می‌شود؟

پاسخ:

دروازه نقطه شکست در IDT (که توسط ماکرو SETGATE تنظیم شده است) حاوی یک فیلد dpl (سطح امتیاز Descriptor) است که از صفر تا سه مقداردهی می‌شود. این فیلد کمترین سطحی را که می‌تواند این وقفه را توسط نرم افزار فراخوانی کند، کنترل می‌کند. ما باید آن را روی ۳ تنظیم کنیم تا بتوانیم توسط کاربر فراخوانی شود. اگر آن را روی صفر تنظیم کنیم، اگر بخواهیم آن را در نرم افزار فراخوانی کنیم (با صدور دستورالعمل int 3) یک خطای حفاظت کلی ایجاد می‌کند. بنابراین، راه درست برای مقداردهی اولیه گیت به صورت زیر است:

```
SETGATE(idt[T_BRKPT], true, GD_KT,t_brkpt, 3);
```

سوال ۶.۲: به نظر شما هدف این مکانیسم‌ها، به‌ویژه با توجه به کاری که برنامه تست کاربر softint/انجام می‌دهد چیست؟

پاسخ:

این مکانیسم‌ها کاری را که برنامه‌های کاربر می‌توانند انجام دهند محدود می‌کنند، بنابراین، هسته را از تداخل یا خراب شدن محافظت می‌کنند. آن‌ها همچنین هسته را از برنامه‌های کاربر جدا می‌کنند و این باعث افزایش استحکام سیستم می‌شود. در واقع هدف این مکانیسم این است که کاربر نتواند کنترل کننده‌های وقفه‌ای که قرار است فقط برای موقعیت‌های استثنایی اجرا شوند را اجرا کند. همچنین این مکانیسم می‌تواند هسته را در برابر برنامه‌های مخرب محافظت کند.

تمرین ۷:

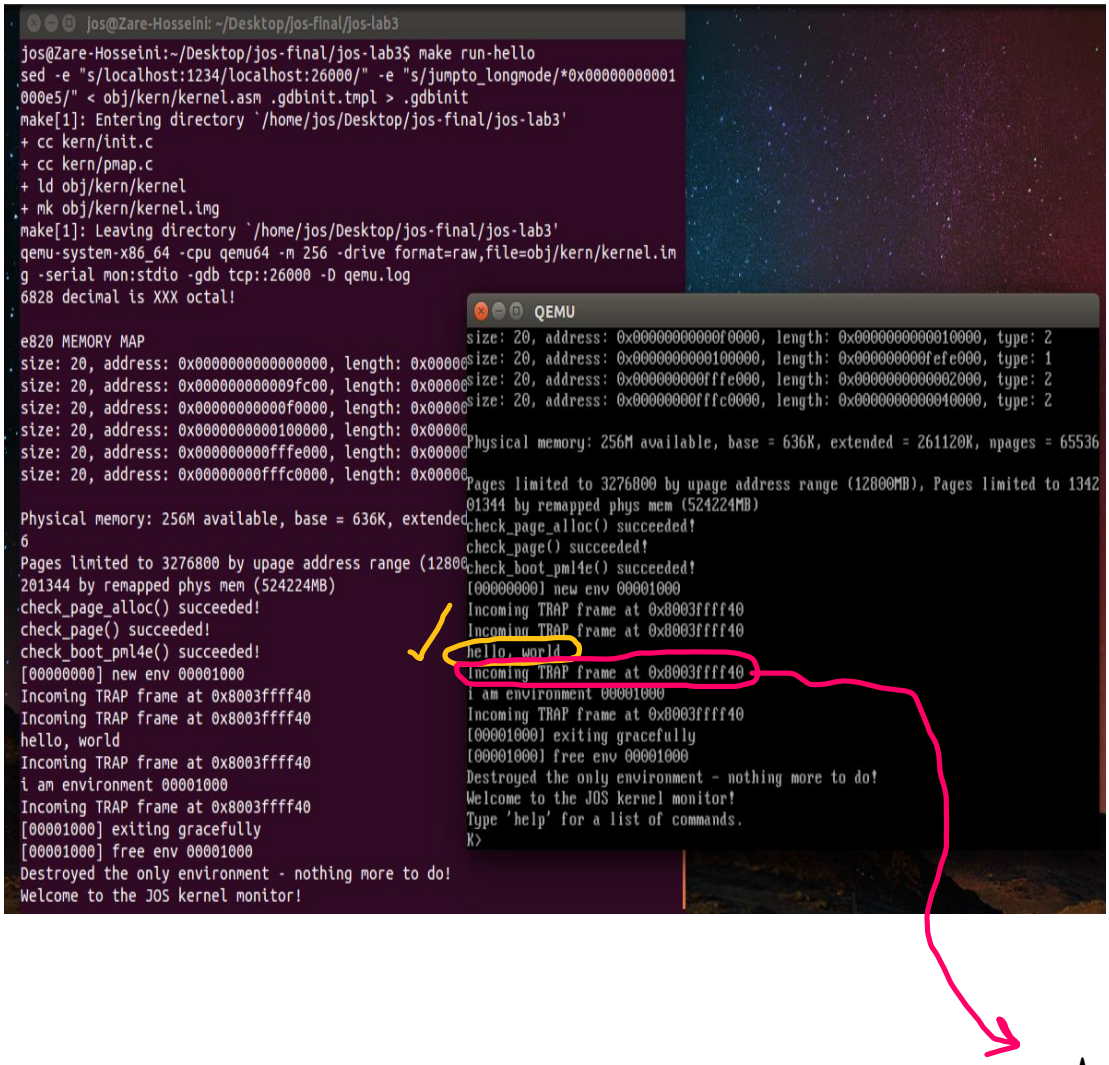
Exercise 7. Add a handler in the kernel for interrupt vector `T_SYSCALL`. You will have to edit `kern/trapentry.S` and `kern/trap.c`'s `trap_init()`. You also need to change `trap_dispatch()` to handle the system call interrupt by calling `syscall()` (defined in `kern/syscall.c`) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall()` in `kern/syscall.c`. Make sure `syscall()` returns `-E_INVAL` if the system call number is invalid. You should read and understand `lib/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. You may also find it helpful to read `inc/syscall.h`.

Run the `user/hello` program under your kernel. It should print "`hello, world`" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get `make grade` to succeed on the `testbss` test.

در این بخش یک کنترل کننده در هسته برای بردار وقفه T_SYSCALL اضافه کردیم. kern/trapentry.S و trap_init () kern/trap.c را ویرایش کردیم. همچنین trap_dispatch() را تغییر دادیم. در نهایت syscall() را در kern/sycall.c پیاده‌سازی کردیم. مطمئن شوید که اگر شماره سیستم کال نامعتبر باشد، syscall() عبارت E_INVAL را برمی‌گرداند.

همچنین برنامه user/hello را اجرا کردیم. عبارت "hello, world" روی کنسول چاپ شد و سپس باعث خطای صفحه در حالت کاربر شد. اگر این اتفاق نیفتد، احتمالاً به این معنی است که کنترل کننده تماس سیستم کاملاً درست نیست. همچنین در testbss هم نمره کامل را گرفتیم که در تصویر زیر قابل مشاهده است.

تصویر اجرای دستور make run-hello و چاپ "hello, world" و سپس خطای صفحه را در تصویر زیر مشاهده می‌کنید.



تمرین ۸:

Exercise 8. 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()` (see `lib/libmain.c` and `lib/exit.c`). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get `make grade` to succeed on the `hello` test.

این تمرین را هم انجام دادیم و در تصویر سوال قبل قابل مشاهده است.

تمرین ۹:

Exercise 9. 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`.

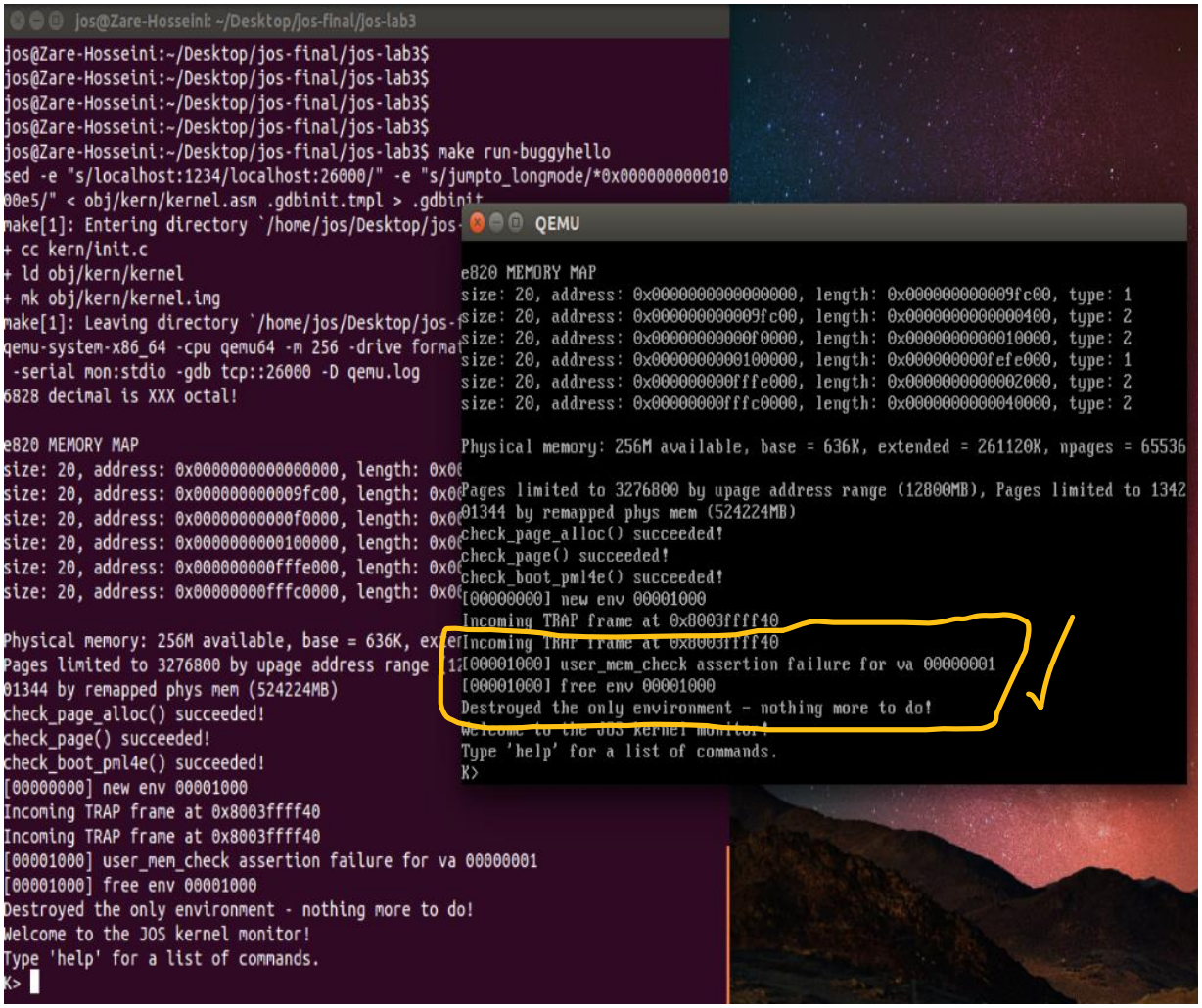
Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

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

Change `kern/init.c` to run `user/buggyhello` instead of `user/hello`. Compile your kernel and boot it. The environment should be destroyed, and the kernel should *not* panic. You should see:

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

موارد خواسته شده را انجام دادیم و با اجرای دستور make run-buggyhello خروجی خواسته شده را دریافت کردیم که در تصویر زیر نشان داده شده است.

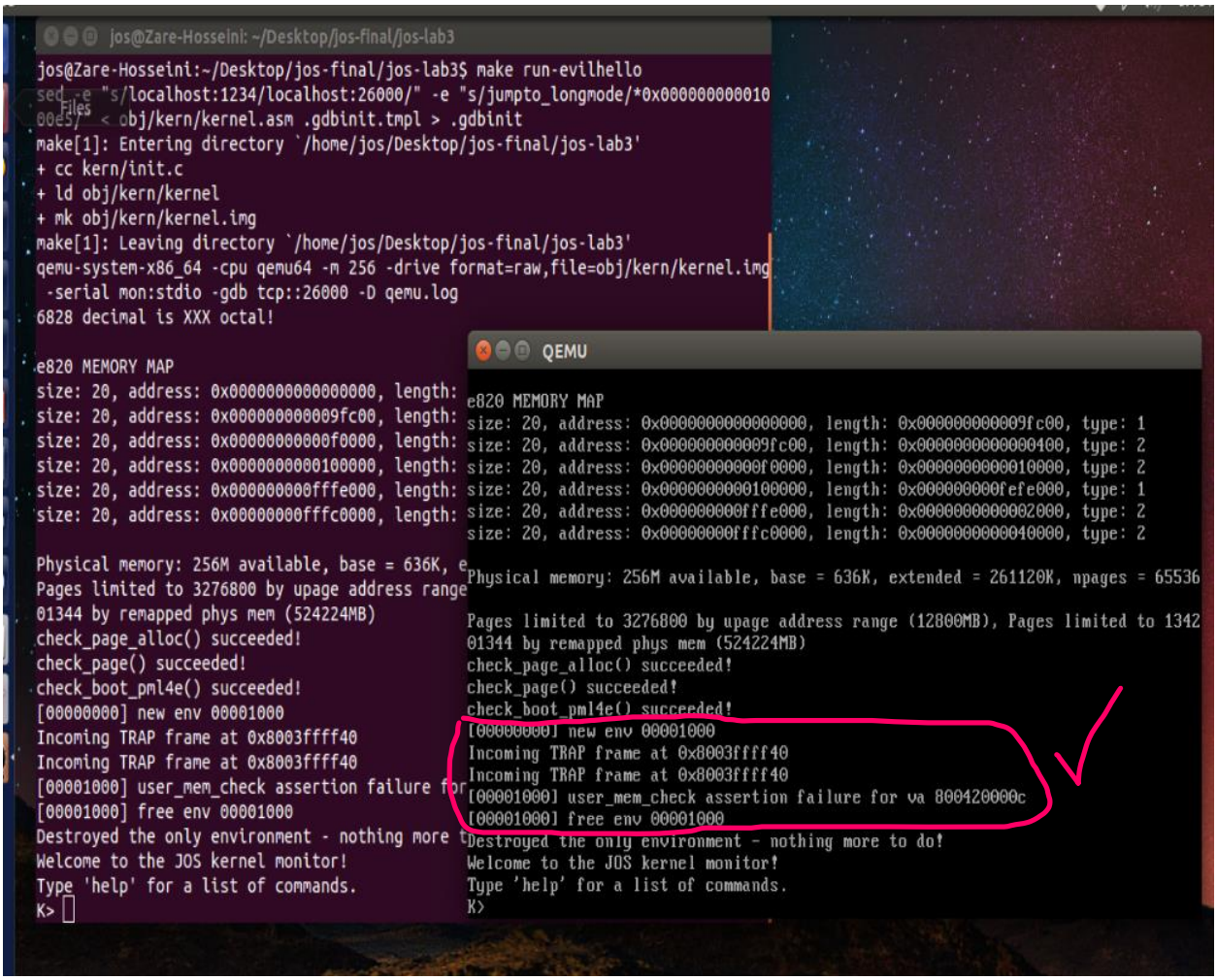


تمرین ۱۰:

Exercise 10. Change `kern/init.c` to run `user/evilhello`. Compile your kernel and boot it. The environment should be destroyed, and the kernel should not panic. You should see:

```
        [00000000] new env 00001000
        [00001000] user_mem_check assertion failure for va f010000c
        [00001000] free env 00001000
```

با اجرای دستور make run-evilhello به خروجی مد نظر تمرین رسیدیم که در تصویر زیر مشاهده می‌کنید.



چالش ۱:

Challenge 1! (2 bonus points) You probably have a lot of very similar code right now, between the lists of `TRAPHANDLER` in `trapentry.S` and their installations in `trap.c`. Clean this up. Change the macros in `trapentry.S` to automatically generate a table for `trap.c` to use. Note that you can switch between laying down code and data in the assembler by using the directives `.text` and `.data`.

یک راه حل برای این چالش پیدا کردیم. در این راه حل، یک آرایه trap_init.c ایجاد می‌کند تا از آن استفاده کند. از دو برچسب اسمبلی text. و data. برای جابجایی بین کد Layouting و داده استفاده می‌کند. به عنوان مثال، TRAPHANDLER به صورت زیر بازنویسی می‌شود.

```
#define TRAPHANDLER(name, num)                                          \
        .text;                                                          \
        .globl name;            /* define global symbol for 'name' */   \
        .type name, @function;  /* symbol type is function */           \
        .align 2;               /* align function definition */         \
        name:                   /* function starts here */              \
        pushl $(num);                                                   \
        jmp _alltraps;                                                  \
        .data;                                                          \
        .long name;
```

این راه حل ابتدا از برچسب text. برای تولید کد و سپس از برچسب data. برای بردارهای متغیرگلوبال استفاده می‌کند. در زیر، از برچسب data. استفاده می‌کند و بردارهای متغیر گلوبال را تولید می‌کند. سپس می‌توان جدول کنترل کننده بردارها را مانند این تولید کرد.

```
.data
.globl vectors
vectors:
TRAPHANDLER_NOEC(vector0, T_DIVIDE);
TRAPHANDLER_NOEC(vector1, T_DEBUG);
TRAPHANDLER_NOEC(vector2, T_NMI);
```

نمادهای long. تعریف شده در ماکرو به این بردارهای گلوبال اضافه خواهند شد. سپس در trap_init() داریم:

```
extern void (*vectors[])();
int i;

for (i = 0; i < 20; i++)
  SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
SETGATE(idt[1], 1, GD_KT, vectors[1], 0);
SETGATE(idt[3], 1, GD_KT, vectors[3], 3);
SETGATE(idt[4], 1, GD_KT, vectors[4], 0);
// interrupt handler for syscall
SETGATE(idt[48], 1, GD_KT, vectors[20], 3);
```

برای این چالش از این سایت کمک گرفته شده است.

چالش ۲:

Challenge 2! (5 bonus points; mega-bragging rights for a good disassembler) Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the `int3`, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the `EFLAGS` register in order to implement single-stepping.

Optional: If you're feeling really adventurous, find some x86 disassembler source code - e.g., by ripping it out of QEMU, or out of GNU binutils, or just write it yourself - and extend the JOS kernel monitor to be able to disassemble and display instructions as you are stepping through them. Combined with the symbol table loading from lab 2, this is the stuff of which real kernel debuggers are made.

باید از env_run(curenv) برای اجرای محیط متوقف شده استفاده کنیم. باید یک دستور مانیتور mon_continue را در monitor.c اضافه کنیم. ابتدا مطمئن شوید که شماره وقفه T_BRKPT یا T_DEBUG باشد. سپس از env_run(curenv) برای ادامه اجرا استفاده کنید.

برای پیاده سازی single stepping یا تک گامه ، می‌دانیم که اگر بیت شماره ۸ در EFLAGS یعنی trap flag را روشن کنیم، پردازنده وارد حالت تک گامه می‌شود. پردازنده در هر مرحله متوقف می‌شود و یک وقفه T_DEBUG ایجاد می‌کند. یک دستور مانیتور mon_stepi را در monitor.c اضافه می‌کنیم. ابتدا مطمئن می‌شود trapno T_BRKPT یا T_DEBUG باشد. سپس بیت شماره ۸ tf->tf_flags را روی ۱ قرار می‌دهد و اجرای محیط فعلی را ادامه می‌دهد. همچنین اگر کاربر بخواهد پس از تک گامه شدن ادامه دهد، باید فلگ وقفه را در mon_continue ریست کنیم. کد mon_continue و mon_stepi را در تصویر زیر مشاهده می‌کنید.

```
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;
}
```

برای این چالش از این سایت کمک گرفته شده است.

خروجی make grade **و نمره نهایی تمرین**