OSLAB 3 REPORT

AMAL JOY
CS11B006

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

Once JOS gets up and running, the envs pointer points to an array of Env structures representing all the environments in the system. The JOS kernel will support a maximum of NENV simultaneously active environments, Once it is allocated, the envs array will contain a single instance of the Env data structure for each of the NENV possible environments.

The kernel uses the curenv symbol to keep track of the *currently executing* environment at any given time. During boot up, before the first environment is run, curenv is initially set to NULL

An environment ID 'envid_t' has three parts:

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

The environment index ENVX(eid) equals the environment's offset in the 'envs[]' array. The uniqueifier distinguishes environments that were created at different times, but share the same environment index. All real environments are greater than 0 (so the sign bit is zero). envid_ts less than 0 signify errors. The envid_t == 0 is special, and stands for the current environment.

Env structure is defined as follows

```
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

        // Address space
        pde_t *env_pgdir;               // Kernel virtual address of page dir
};
```

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 (defined in inc/memlayout.h) so user processes can read from this array.

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

The following change is done in mem_init()

```
 envs = (struct Env *) boot_alloc(sizeof(struct Env) * NENV);
//Make 'envs' point to an array of size 'NENV' of 'struct Env
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
//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
```

The -b binary option on the linker command line causes files to be linked in as "raw" uninterpreted binary files rather than as regular .o files produced by the compiler. linker has "magically" produced a number of funny symbols with obscure names like _binary_obj_user_hello_start, _binary_obj_user_hello_end, and _binary_obj_user_hello_size. The linker generates these symbol names by mangling the file names of the binary files; the symbols provide the regular kernel code with a way to reference the embedded binary files.

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 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.

env_init()-Does the following
Mark all environments in 'envs' as free, set their env_ids to 0, and insert them into the env_free_list. Also the environments are in the free list in the same order they are in the envs array
memset((void*) envs, 0, sizeof(struct Env) * NENV);

```
  env_free_list = envs;

  for ( ;i+1 < NENV;i ++) {
    envs[i].env_id = 0;
    envs[i].env_status = ENV_FREE;
    envs[i].env_link = &envs[i+1];
  }

  envs[i].env_id = 0;
  envs[i].env_status = ENV_FREE;
  envs[i].env_link = NULL;
```

`env_setup_vm()` -Does the following
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. Returns 0 on success, < 0 on error. Errors include:
-E_NO_MEM if page directory or table could not be allocated.
pp_ref is not maintained for physical pages mapped only above UTOP, but env_pgdir is an exception -- we need to increment env_pgdir's pp_ref for env_free to work correctly

```
if (!(p = page_alloc(ALLOC_ZERO)))
    return -E_NO_MEM;
   p->pp_ref ++;
  e->env_pgdir = page2kva(p);
  memmove(e->env_pgdir, kern_pgdir, PGSIZE);
  e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
  return 0;
```

regionalloc() does the following

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.

```
region_alloc(struct Env *e, void *va, size_t len)
{
            //   'va' and 'len' values may not  be page-aligned.
            //   You should round va down, and round (va + len) up.


struct PageInfo* p = NULL;
 va = ROUNDDOWN(va, PGSIZE);
 void * va_end;
 va_end = va + ROUNDUP(len, PGSIZE);
 struct PageInfo *pp ;
 for(; va < va_end ;va += PGSIZE) {
            pp = page_alloc(0);
             if (!pp)
                        panic("region_alloc : page_alloc() out of memory.\n");
            int ret = page_insert(e->env_pgdir, pp, va, PTE_U | PTE_W);
            if (ret < 0)
                        panic("region_alloc : page_insert() %e.\n", ret);
  }
```

loadicode() Does the following

Set up the initial program binary, stack, and processor flags for a user process.
This function is ONLY called during kernel initialization, before running the first user-mode environment.This function loads all loadable segments from the ELF binary imageinto 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.
Load each program segment into virtual memory at the address specified in the ELF section header.We should only load segments with ph->p_type == ELF_PROG_LOAD.
Each segment's virtual address can be found in ph->p_va and its size in memory can be found in ph->p_memsz.The ph->p_filesz bytes from the ELF binary, starting at 'binary + ph->p_offset', should be copied to virtual address ph->p_va. Any remaining memory bytes should be cleared to zero.

```
    struct Proghdr *ph, *eph;
    struct Elf *elfhdr;
    struct PageInfo *pp;

    elfhdr = (struct Elf *) binary;
    if (elfhdr->e_magic != ELF_MAGIC)
        panic("load_icode : not an valid ELF.\n");

    ph = (struct Proghdr *) (binary + elfhdr->e_phoff);
    eph = ph + elfhdr->e_phnum;
  lcr3(PADDR(e->env_pgdir));

    for (; ph < eph; ph++) {
        if (ph->p_type != ELF_PROG_LOAD)
            continue;
        region_alloc(e, (void*)ph->p_va, ph->p_memsz);
        memset((void *)ROUNDDOWN((uintptr_t)ph->p_va, PGSIZE), 0 ,
                    ROUNDUP(ph->p_memsz, PGSIZE));
        memmove((void*)ph->p_va, binary+ph->p_offset, ph->p_filesz);
    }
  e->env_tf.tf_eip = elfhdr->e_entry;
    region_alloc(e, (void*)(USTACKTOP-PGSIZE), PGSIZE);
```

env_create() does the following
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.

```
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    /*
     * 1.env_alloc
     * 2.load the binay by load_icode
     * 3.envt_type setup
     * 4.parent id is zero.
     */
    struct Env *newenv;
    int r;
    if((r = env_alloc(&newenv, 0)) < 0 )
        panic("env_create : env_alloc %e.\n", r);

    load_icode(newenv, binary, size);
    newenv->env_type = type;
}
```

After all this the CPU discovers that it is not set up to handle this system call interrupt, it will generate a general protection exception, find that it can't handle that, generate a double fault exception, find that it can't handle that either, and finally give up with what's known as a "triple fault"

Exceptions and interrupts are both "protected control transfers," which cause the processor to switch from user to kernel mode (CPL=0) without giving the user-mode code any opportunity to interfere with the functioning of the kernel or other environments. In Intel's terminology, an ***interrupt*** is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity. An ***exception***, in contrast, is a protected control transfer caused synchronously by the currently running code, for example due to a divide by zero or an invalid memory access.

The Interrupt Descriptor Table. The processor ensures that interrupts and exceptions can only cause the kernel to be entered at a few specific, well-defined entry-points *determined by the kernel itself*, and not by the code running when the interrupt or exception is taken.

The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 255. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's *interrupt descriptor table* (IDT), which the kernel sets up in kernel-private memory, much like the GDT. From the appropriate entry in this table the processor loads:

- the value to load into the instruction pointer (EIP) register, pointing to the kernel code designated to handle that type of exception.

  The Task State Segment. The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of EIP and CS before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel.

For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) SS, ESP, EFLAGS, CS, EIP, and an optional error code. Then it loads the CS and EIP from the interrupt descriptor, and sets the ESP and SS to refer to the new stack.

All of the synchronous exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31. For example, a page fault always causes an exception through vector 14. Interrupt vectors greater than 31 are only used by *software interrupts*, which can be generated by the `int` instruction, or asynchronous *hardware interrupts*, caused by external devices when they need attention.

The processor can take exceptions and interrupts both from kernel and user mode. It is only when entering the kernel from user mode, however, that the x86 processor automatically switches stacks before pushing its old register state onto the stack and invoking the appropriate exception handler through the IDT. If the processor is *already* in kernel mode when the interrupt or exception occurs , then the CPU just pushes more values on the same kernel stack. In this way, the kernel can gracefully handle *nested exceptions* caused by code within the kernel itself.

There is one important caveat to the processor's nested exception capability. If the processor takes an exception while already in kernel mode, and *cannot push its old state onto the kernel stack* for any reason such as lack of stack space, then there is nothing the processor can do to recover, so it simply resets itself.

Each exception or interrupt has its own handler in `trapentry.S` and `trap_init()` should initialize the IDT with the addresses of these handlers. Each of the handlers should build a `struct Trapframe` on the stack and call `trap()` with a pointer to the Trapframe. `trap()` then handles the exception/interrupt or dispatches to a specific handler function.

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.

Your _alltraps should:

1. **push values to make the stack look like a struct Trapframe**
2. **load GD_KD into %ds and %es**
3. **pushl %esp to pass a pointer to the Trapframe as an argument to trap()**
4. **call trap (can trap ever return?)**

void

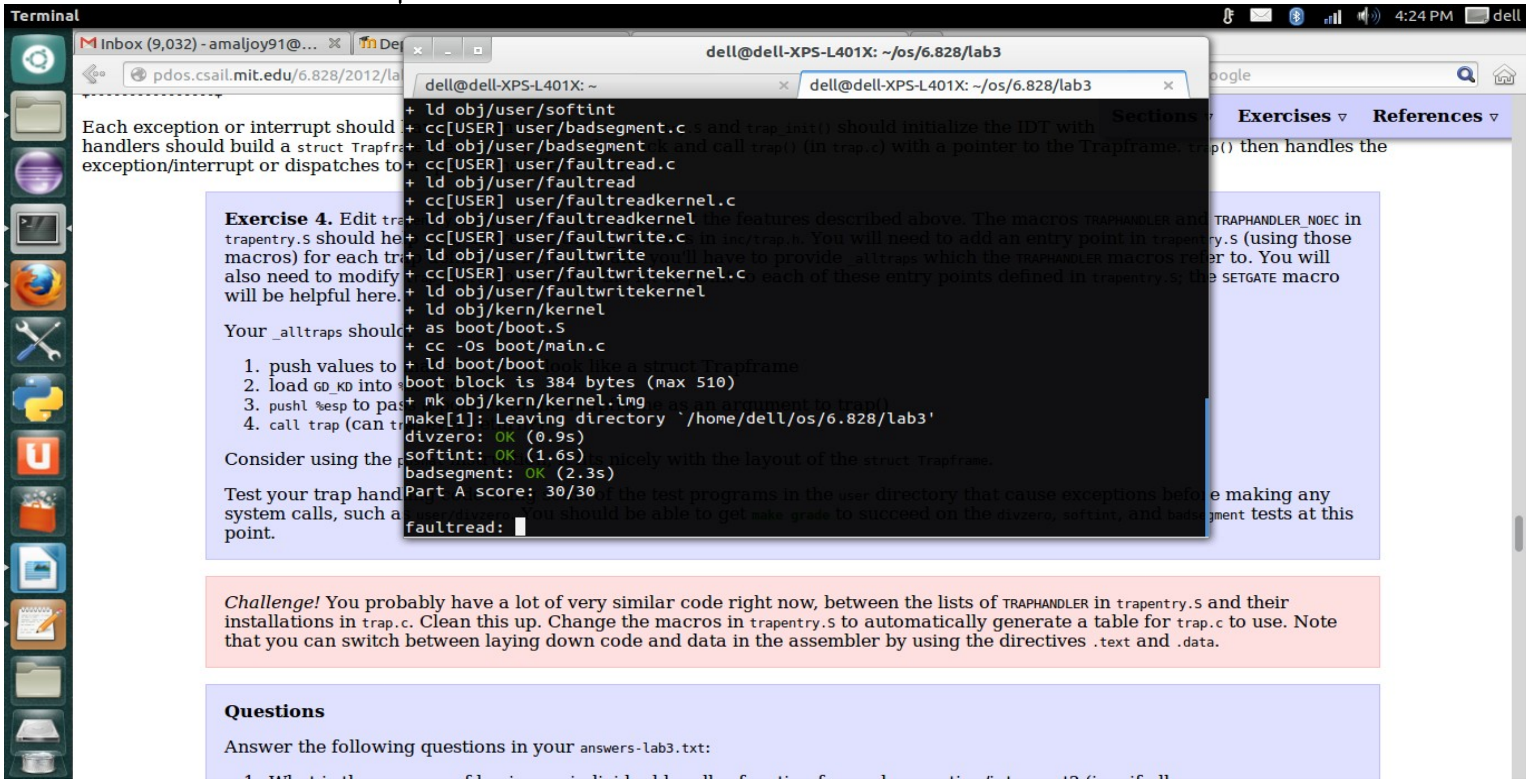
```
trap_init(void)
{
	extern struct Segdesc gdt[];

	// LAB 3: Your code here.



	extern unsigned int vectors[];
	extern void T_DEFAULT_HANDLER(), T_SYSCALL_HANDLER(), T_BRKPT_HANDLER(); // trap handlers
defined in kern/trapentry.S
	int i;
	unsigned int *vector = vectors;

	// iterate through all possible 256 vectors, defining an IDT entry for each one
	// at each step, vector points to the next defined trap number that has yet to be set in the IDT
	for(i = 0; i < 256; i++) {
		if (*vector == i) {
			/*
			 * in this case, vector points to the trap number we must now set in the IDT,
			 * and vector+1 points to the address of the trap handler for this trap
			 */
			if (i >= IRQ_OFFSET && i < IRQ_OFFSET + 16) {
				SETGATE(idt[i], 0, GD_KT, *(vector+1), 3);
			}
			else {
				SETGATE(idt[i], 0, GD_KT, *(vector+1), 0);
			}
			vector += 2;	// make vector point to the next trap
		}
		else {
			/*
			 * in this case, there is no defined trap for this vector
			 * set the IDT entry to point to the default trap handler
			 */
			SETGATE(idt[i], 0, GD_KT, &T_DEFAULT_HANDLER, 0);
		}
	}
	// these are special, since they are traps we allow the user to call
```

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

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

Whenever TRAPHANDLER is called, two pieces of .long data are written to memory in the vectors data segment.
 * These are the trap number and the address of the trap handler, respectively.
 * If we have 'extern unsigned int vectors[]' and 'int i' with '!(i%2)', then vectors[i] is a trap number, and vectors[i+1] is the address of the trap handler for that trap number.
* To denote the end of the vectors data segment, T_DEFAULT is the last defined trap, since we can identify it by the fact that it is not a valid trap number.
In Trapentry.s

```
vectors:
  TRAPHANDLER_NOEC1(T_DIVIDE)
  TRAPHANDLER_NOEC1(T_DEBUG)  etc

pushl %ds /* push onto tf_ds */
  pushl %es /* push onto tf_es */
  pushal /* push onto PushRegs */
  movw $GD_KD, %ax
  movw %ax, %ds
  movw %ax, %es
  pushl %esp
  call trap
  addl $4, %esp
```

Now Part A is complete


```
+ ld obj/user/softint
+ cc[USER] user/badsegment.c
+ ld obj/user/badsegment
+ cc[USER] user/faultread.c
+ ld obj/user/faultread
+ cc[USER] user/faultreadkernel.c
+ ld obj/user/faultreadkernel
+ cc[USER] user/faultwrite.c
+ ld obj/user/faultwrite
+ cc[USER] user/faultwritekernel.c
+ ld obj/user/faultwritekernel
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 384 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/dell/os/6.828/lab3'
divzero: OK (0.9s)
softint: OK (1.6s)
badsegment: OK (2.3s)
Part A score: 30/30
faultread: 
```

Challenge! 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.

Questions

Answer the following questions in your answers-lab3.txt:

PART B

When the processor takes a page fault, it stores the linear (i.e., virtual) address that caused the fault in a special processor control register, CR2. In trap.c we have provided the beginnings of a special function, page_fault_handler(), to handle page fault exceptions.

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 (tf->tf_trapno == T_PGFLT) {
                        page_fault_handler(tf);
                        return;
          }
```

The breakpoint exception, interrupt vector 3 (T_BRKPT), is normally used to allow debuggers to insert breakpoints in a program's code . In JOS we will abuse this exception slightly by turning it into a primitive pseudo-system call that any user environment can use to invoke the JOS kernel monitor. This usage is actually somewhat appropriate if we think of the JOS kernel monitor as a primitive debugger.

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.

```
if (tf->tf_trapno == T_BRKPT) {
                        monitor(tf); // breakpoint exceptions invoke the kernel monitor
                        return;
          }
```

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.

TRAPHANDLER_NOEC1(T_SYSCALL) in trapentry.s

SETGATE(idt[T_SYSCALL], 0, GD_KT, &T_SYSCALL_HANDLER, 3); in trap_init in trap.c

```
if (tf->tf_trapno == 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 trap_dispatch  in trap.c
```

syscall is implemented as follows

```
// Call the function corresponding to the 'syscallno' parameter.
// Return any appropriate return value.
// LAB 3: Your code here.

        static void *syscalls[] = {
                [SYS_cputs] &sys_cputs,
                [SYS_cgetc] &sys_cgetc,
                [SYS_getenvid] &sys_getenvid,
                [SYS_env_destroy] &sys_env_destroy,

        };
        uint32_t ret = 0, esp = 0, ebp = 0;

        // return -E_INVAL if the system call number is invalid  if it is not the case that 0 <= syscallno <
NSYSCALLS
        if (!(0 <= syscallno && syscallno < NSYSCALLS)) {
                return -E_INVAL;
        }
```

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.

```
libmain(int argc, char **argv)
{
// set thisenv to point at our Env structure in envs[].
 // LAB 3: Your code here.
 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);

}
```

Operating systems usually rely on hardware support to implement memory protection. The OS keeps the hardware informed about which virtual addresses are valid and which are not. When a program tries to access an invalid address or one for which it has no permissions, the processor stops the program at the instruction causing the fault and then traps into the kernel with information about the attempted operation. If the fault is fixable, the kernel can fix it and let the program continue running. If the fault is not fixable, then the program cannot continue, since it will never get past the instruction causing the fault.

A page fault in the kernel is potentially a lot more serious than a page fault in a user program. If the kernel page-faults while manipulating its own data structures, that's a kernel bug, and the fault handler should panic the kernel (and hence the whole system). But when the kernel is dereferencing pointers given to it by the user program, it needs a way to remember that any page faults these dereferences cause are actually on behalf of the user program.

The kernel typically has more memory permissions than the user program. The user program might pass a pointer to a system call that points to memory that the kernel can read or write but that the program cannot. The kernel must be careful not to be tricked into dereferencing such a pointer, since that might reveal private information or destroy the integrity of the kernel.

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.

Boot your kernel, running user/buggyhello. 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!
```

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 run backtrace from the kernel monitor and see the backtrace traverse into lib/libmain.c before the kernel panics with a page fault.

In page fault handler

```
if ((tf->tf_cs & 3 ) == 0){
                panic("page_fault_halder : page fault in kernel mode.\n");
                return ;
        }
//We've already handled kernel-mode exceptions, so if we get here, the page fault happened in user mode.
print_trapframe(tf);
env_destroy(curenv);
```

user_mem_check does the following

Check that an environment is allowed to access the range of memory [va, va+len) with permissions 'perm | PTE_P'. 'va' and 'len' need not be page-aligned; we must test every page that contains any of that range. A user program can access a virtual address if (1) the address is below ULIM, and (2) the page table gives it permission. These are exactly
the tests you should implement here.

If there is an error, set the 'user_mem_check_addr' variable to the first
erroneous virtual address.

Returns 0 if the user program can access this range of addresses,
and -E_FAULT otherwise.

```
        int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
        // LAB 3: Your code here.

        void * va_end = (void*)ROUNDUP(va + len, PGSIZE);
        pte_t * pte;

        for ( ; va < va_end ; va = ROUNDUP(va + PGSIZE, PGSIZE)) {
                pte = pgdir_walk(env->env_pgdir, va, 0);
                if (!pte || ( (*pte & (perm | PTE_P)) != (perm | PTE_P) )) {
                        user_mem_check_addr = (uintptr_t)va;
                        return  -E_FAULT;
                }
        }

        return 0;
}
```

In kdebug.c,the following changes are done in debug_info_eip

The user-application linker script, user/user.ld, puts information about the application's stabs (equivalent to __STAB_BEGIN__, __STAB_END__, __STABSTR_BEGIN__, and __STABSTR_END__) in a structure located at virtual address USTABDATA.

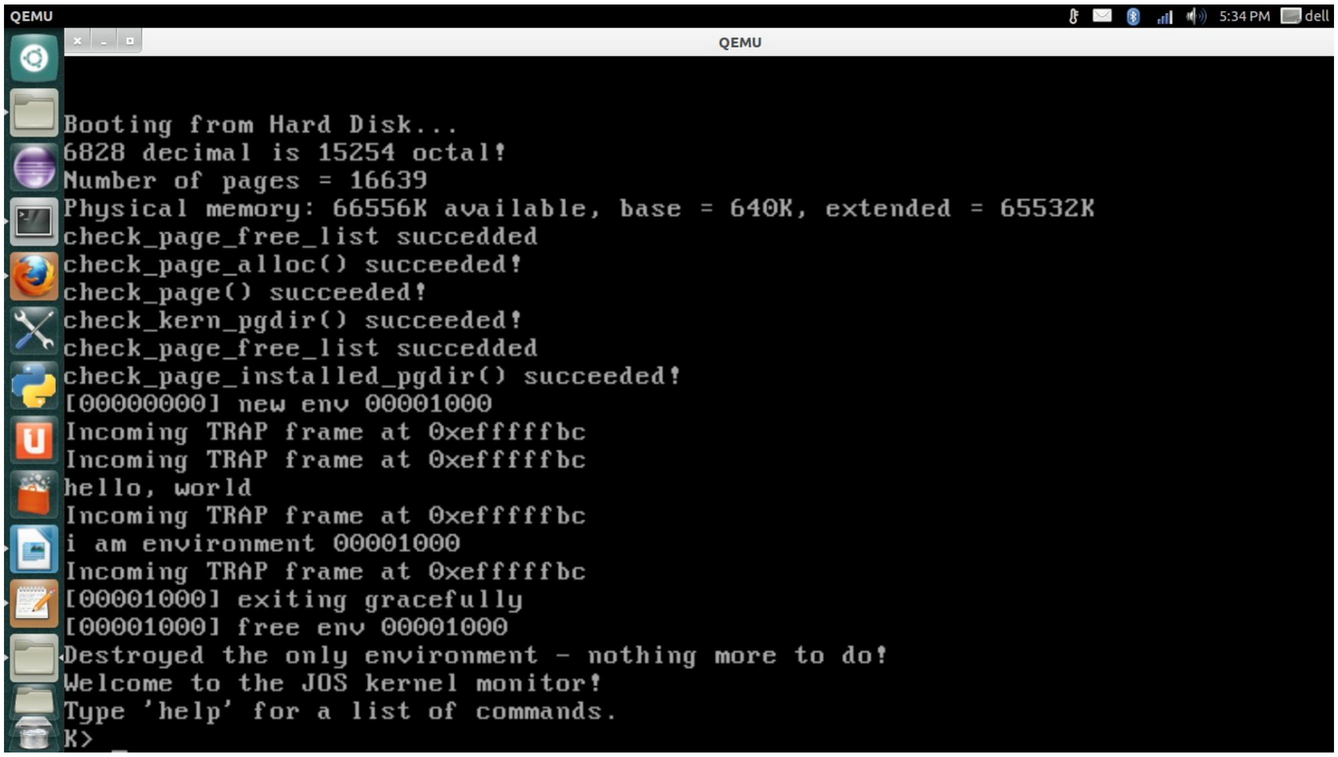
```
const struct UserStabData *usd = (const struct UserStabData *)  USTABDATA;


                // 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) < 0)
 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) < 0 ||
  user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U) < 0)
 return -1;
        }
```

Exercise 10. Boot your kernel, running user/evilhello.

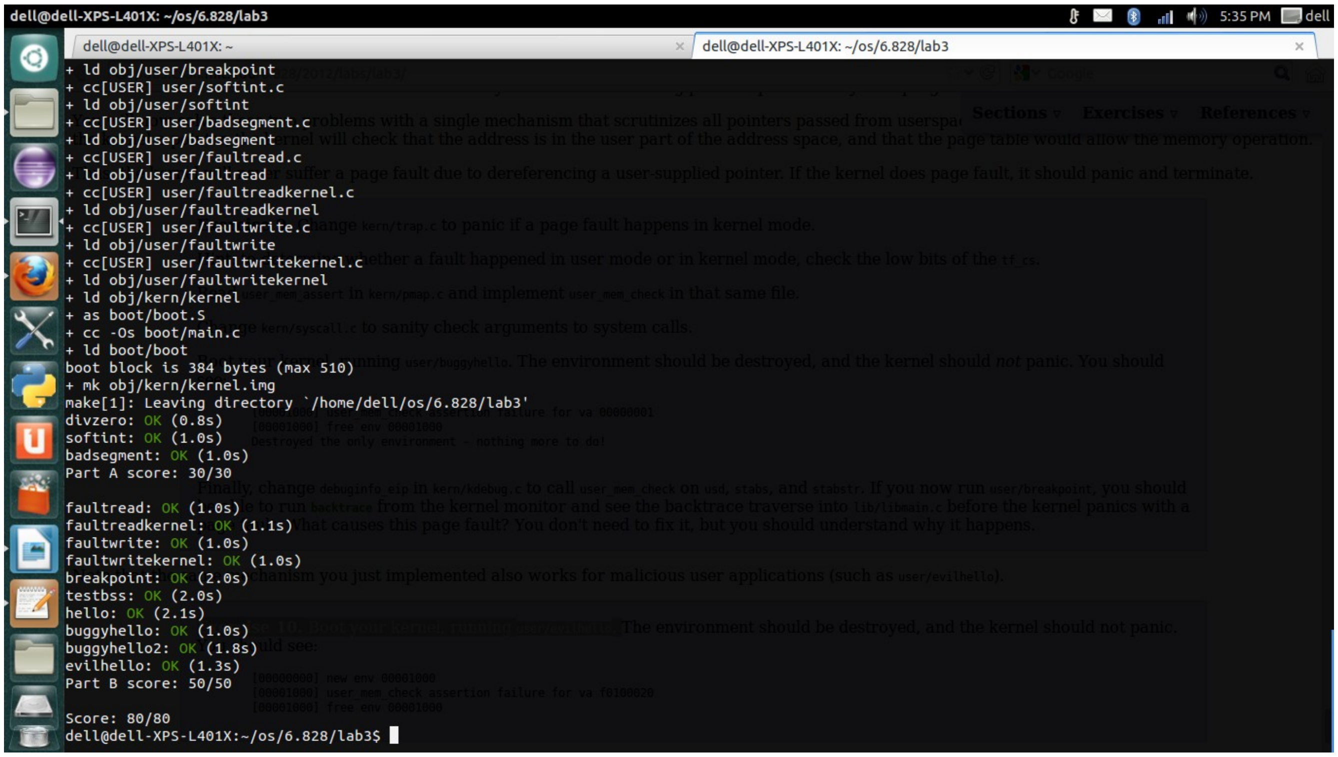
Same as 10

FOR MAKE QEMU



```
Booting from Hard Disk...
6828 decimal is 15254 octal!
Number of pages = 16639
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_free_list succedded
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list succedded
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
hello, world
Incoming TRAP frame at 0xefffffbc
i am environment 00001000
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> _
```

FOR MAKE GRADE



dell@dell-XPS-L401X: ~/os/6.828/lab3

dell@dell-XPS-L401X: ~ | dell@dell-XPS-L401X: ~/os/6.828/lab3

```
+ ld obj/user/breakpoint
+ cc[USER] user/softint.c
+ ld obj/user/softint
+ cc[USER] user/badsegment.c
+ ld obj/user/badsegment
+ cc[USER] user/faultread.c
+ ld obj/user/faultread
+ cc[USER] user/faultreadkernel.c
+ ld obj/user/faultreadkernel
+ cc[USER] user/faultwrite.c
+ ld obj/user/faultwrite
+ cc[USER] user/faultwritekernel.c
+ ld obj/user/faultwritekernel
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 384 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/dell/os/6.828/lab3'
divzero: OK (0.8s)
softint: OK (1.0s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (1.1s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.0s)
breakpoint: OK (2.0s)
testbss: OK (2.0s)
hello: OK (2.1s)
buggyhello: OK (1.0s)
buggyhello2: OK (1.8s)
evilhello: OK (1.3s)
Part B score: 50/50

Score: 80/80
dell@dell-XPS-L401X:~/os/6.828/lab3$
```