Lab 3: User Environments

Introduction

In this lab you will implement the basic kernel facilities required to get a protected user-mode environment (i.e. "process") running. You will enhance the JOS kernel to set up the data structures to keep track of user environments, create a single user environment, load a program image into it, and start it running. You will also make the JOS kernel capable of handling any system calls the user environment makes and handling any other exceptions it causes.

Note: *In this lab, the terms environment and process are interchangeable - both refer to an abstraction that allows you to run a program.*

GCC's Inline Assembly Language Feature:

Please refer the following links as they seem to be the best:

Brennan's Guide to Inline Assembly

GCC-Inline-Assembly-HOWTO

Part A: User Environments and Exception Handling

The kernel uses the Env data structure to keep track of each user environment.

As you can see in kern/env.c, the kernel maintains three main global variables pertaining to environments:

```
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. In our design, the JOS kernel will support a maximum of NENV simultaneously active environments, although there will typically be far fewer running environments at any given time. (NENV is a constant #define'd in inc/env.h.) 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 JOS kernel keeps all of the inactive Env structures on the env_free_list. This design allows easy allocation and deallocation of environments, as they merely have to be added to or removed from the free list.

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.

Environment State

The Env structure is defined in inc/env.h as follows (although more fields will be added in future labs):

```
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
};
```

Here's what the Env fields are for:

env_tf:
This structure, defined in inc/trap.h, holds the saved register values for the environment while that environment is not running: i.e., when the kernel or a different environment is running. The kernel saves these when switching from user to kernel mode, so that the environment can later be resumed where it left off.

env_link:
This is a link to the next Env on the env_free_list. env_free_list points to the first free environment on the list.

env_id:
The kernel stores here a value that uniquely identifiers the environment currently using this Env structure (i.e., using this particular slot in the envs array). After a user environment terminates, the kernel may re-allocate the same Env structure to a different environment - but the new environment will have a different env_id from the old one even though the new environment is re-using the same slot in the envs array.

env_parent_id:
The kernel stores here the env_id of the environment that created this environment. In this way the environments can form a "family tree," which will be useful for making security decisions about which environments are allowed to do what to whom.

env_type:
This is used to distinguish special environments. For most environments, it will be ENV_TYPE_USER.

env_status:
This variable holds one of the following values:

ENV_FREE:
Indicates that the Env structure is inactive, and therefore on

the env_free_list.

ENV_RUNNABLE:

Indicates that the Env structure represents an environment that is waiting to run on the processor.

ENV_RUNNING:

Indicates that the Env structure represents the currently running environment.

ENV_NOT_RUNNABLE:

Indicates that the Env structure represents a currently active environment, but it is not currently ready to run: for example, because it is waiting for an interprocess communication (IPC) from another environment.

ENV_DYING:

Indicates that the Env structure represents a zombie environment. A zombie environment will be freed the next time it traps to the kernel.

***env_pgdir**:*

This variable holds the kernel virtual address of this environment's page directory.

Like a Unix process, a JOS environment couples the concepts of "thread" and "address space". The thread is defined primarily by the saved registers (the env_tf field), and the address space is defined by the page directory and page tables pointed to by env_pgdir. To run an environment, the kernel must set up the CPU with both the saved registers and the appropriate address space.

Our struct Env is analogous to struct proc in xv6. Both structures hold the environment's (i.e., process's) user-mode register state in a Trapframe structure. In JOS, individual environments do not have their own kernel stacks as processes do in xv6. There can be only one JOS environment active in the kernel at a time, so JOS needs only a single kernel stack.

Allocating the Environments Array

***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.

```
 // 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 | PTE_P);
```

Creating and Running Environments

Because we do not yet have a filesystem, we will set up the kernel to load a static binary image that is embedded within the kernel itself. JOS embeds this binary in the kernel as a ELF executable image.

The Lab 3 GNUmakefile generates a number of binary images in the obj/user/ directory.

```
avanatk@AVNatk:~/Study/Advanced_OS/QEMU/qemu/Labs/lab/obj/user$ ls
badsegment      breakpoint      buggyhello      buggyhello2.sym  divzero      evilhello      faultread           faultreadkernel.o   faultwrite           faultwritekernel.o   hello      softint      testbss
badsegment.asm  breakpoint.asm  buggyhello2     buggyhello.asm   divzero.asm  evilhello.asm  faultread.asm       faultreadkernel.sym faultwrite.asm       faultwritekernel.sym hello.asm  softint.asm  testbss.asm
badsegment.o    breakpoint.o    buggyhello2.asm buggyhello.o     divzero.o    evilhello.o    faultreadkernel     faultread.o         faultwritekernel     faultwrite.o         hello.o    softint.o    testbss.o
badsegment.sym  breakpoint.sym  buggyhello2.o   buggyhello.sym   divzero.sym  evilhello.sym  faultreadkernel.asm faultread.sym       faultwritekernel.asm faultwrite.sym       hello.sym  softint.sym  testbss.sym
```

If you look at kern/Makefrag, you will notice some magic that "links" these binaries directly into the kernel executable as if they were .o files.

```
# How to build the kernel itself
$(OBJDIR)/kern/kernel: $(KERN_OBJFILES) $(KERN_BINFILES) kern/kernel.ld \
       $(OBJDIR)/.vars.KERN_LDFLAGS
    @echo + ld $@
    $(V)$(LD) -o $@ $(KERN_LDFLAGS) $(KERN_OBJFILES) $(GCC_LIB) -b binary $
(KERN_BINFILES)
    $(V)$(OBJDUMP) -S $@ > $@.asm
    $(V)$(NM) -n $@ > $@.sym
```

The KERN_BINFILES is exactly our user program we have seen above.

```
# Binary program images to embed within the kernel.
# Binary files for LAB3
KERN_BINFILES :=    user/hello \
                user/buggyhello \
                user/buggyhello2 \
                user/evilhello \
                user/testbss \
```

```
		user/divzero \
		user/breakpoint \
		user/softint \
		user/badsegment \
		user/faultread \
		user/faultreadkernel \
		user/faultwrite \
		user/faultwritekernel
```

The b binary option on the linker command line causes these files to be linked in as "raw" uninterpreted binary files rather than as regular .o files produced by the compiler.

If you look at obj/kern/kernel.sym after building the kernel, you will notice that the 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.

In i386_init() in kern/init.c you'll see code to run one of these binary images in an environment.

ENV_CREATE(user_hello, ENV_TYPE_USER);

The macro ENV_CREATE has been defined in kern/env.h as follows:

```
// Without this extra macro, we couldn't pass macros like TEST to
// ENV_CREATE because of the C pre-processor's argument prescan rule.
#define ENV_PASTE3(x, y, z) x ## y ## z

#define ENV_CREATE(x, type)                             \
	do {                                            \
		extern uint8_t ENV_PASTE3(_binary_obj_, x, _start)[];  \
		env_create(ENV_PASTE3(_binary_obj_, x, _start),        \
			   type);                       \
	} while (0)
```

For more on C pre-processor's argument prescan rule please refer the folowing link
https://gcc.gnu.org/onlinedocs/cpp/Argument-Prescan.html

IN this context, because macros like TEST are passed to the ENV_CREATE macro in kern/init.c, the ENV_PASTE macro has been defined to ensure that macro acrguments are concatenated.

The macro ENV_CREATE calls the env_create function passing the

"_binary_obj_user_hello_start" as a parameter which corresponds to the funny symbols generated by the linker. The ENV_PASTE macro generates the concatenated name.

The env_create function will initially call env_alloc function to allocate the user environment and then call the function load_icode to load the ELF binary passed as a parameter to the env_create function into memory. With kernel already in memory, courtesy of the bootloader, the load_icode function will read the binary (ELF) from memory and move it into the virtual addresses stored in ELF binary i.e. it will load all movable sections of the ELF from kernel memory into the user memory of the environment, starting at virtual address which will be mentioned in the ELF file itself i.e. in ELF file's program headers.
The data pertaining to the binary to be loaded into the memory will be contained in the struct Elf and struct Proghdr for the binary file defined in inc/elf.h.

Loading the eip register (i.e. its corresponding entry in the struct Env for the environment) with the value mentioned in e_start filed of the struct Elf for the binary, the load_icode function will also call the region_alloc function to allocate physical memory for the environment and map it to virtual address (also mentioned in the Program headers for the loadable sections of the binary which are denoted by p_type field of struct Proghdr.) in the environment's address space. Finally allocating space for the stack of the environment at USTACKTOP -PGSIZE, the function env_run will be called to run the environment.
NOTE: Only the ENV_CREATE macro, env_init and env_run functions will be called from i386_init function in kern/init.c. The rest of the functions from kern/env.c will call themselves internally within the file at appropriate locations.

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 to 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".

Ans:

```
// 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]).
/*
The env_init function is simply a linked list operation in which we mark all user environments as free (1024 environments, which is maximum that can be allocated in JOS) and insert them into the env_free_list in the same order as they appear in the envs array. An important thing to note here is the use of “.” operator to access structure elements even though envs is a pointer. The reason for this  is the use of the index ([]) operator.
*/
void
env_init(void)
{
     // Set up envs array
     // LAB 3: Your code here.

     env_free_list = &envs[0];
     envs[0].env_id = 0;

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

     envs [NENV - 1].env_link = NULL;
     // Per-CPU part of the initialization
     env_init_percpu();
}


// 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.
/*
With page_alloc returning apointer to the struct PageInfo of page allocated, ,we
use the page2kva macro to obatin the virtual address of page and make a pointer
point to the address. Copying the entire contents of kernel page directory at the
memory location pointed to by the pointer, we, finally, associate it with the
environment passed a parameter as its page directory. The mappings in the
directory however, will only be for VA's above UTOP.
*/
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.
	pde_t* e_pgdir = page2kva (p);
	memcpy (e_pgdir, kern_pgdir, PGSIZE);
	p -> pp_ref ++;
	e -> env_pgdir = e_pgdir;

	// 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;
}
```

```
// 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.
/*
      Getting the number of pages to be allocated through va and len arguments, we use page_alloc function to allocate the pages. The pointers to the struct PageInfo's of the allocated pages are then being used in the page_insert method along with addr, which has been calculated to establish mappings in terms of pages, to establish mappings and make entries for the mappings in environment's page directory through pge_insert method. The page_insert method will establish mapping between virtual address va and the physical address of the page pointed to by the struct PageInfo instance.
*/
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!)

     if (len == 0)
     return;

     uintptr_t h_addr = ROUNDUP ((uintptr_t) va + len, PGSIZE);
     uintptr_t l_addr = ROUNDDOWN ((uintptr_t) va, PGSIZE);
     uintptr_t page_count = (h_addr - l_addr) / PGSIZE;

     for (int i = 0; i < page_count; i ++)
     {
          struct PageInfo* new_page = page_alloc(ALLOC_ZERO);
         assert(new_page);

          void* addr = (void *) (l_addr + (i * PGSIZE));
          if ((page_insert(e -> env_pgdir, new_page, addr, PTE_U | PTE_W)) < 0)
               panic ("Page Insert Failed \n");
     }
}

// 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 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.
//
// All this is very similar to what our boot loader does, except the boot
// loader also needs to read the code from disk.  Take a look at
// boot/main.c to get ideas.
//
// Finally, this function maps one page for the program's initial stack.
//
// load_icode panics if it encounters problems.
//  - How might load_icode fail?  What might be wrong with the given input?

/*
The pointer argument of the function i.e. binary is a pointer to the actual
pointer to the binary which is to loaded in user address space. This function will
only run once when kernel wants to transition to the user mode. The struct Elf and
struct Proghdr defined in inc/elf.h are of great help in this function.
Instantiating a pointer instance of the struct Elf, we cast binary into the same
and check for the magic number of the ELF file to be loaded into appropriate user
memory to determine the elf file is valid.
IN the event of the elf file being valid, we load the environment's page directory
into the CR3 register since mapping will have to be established for the user
environment to be loaded and the kernel's page directory does not have any
mappings for VA's below UTOP which is where user environment is.

We then head over to the program header table of the ELF binary specified by the
e_phoff field of struct ELF and get a pointer there. We then obtain the number of
entries in the program header table from the e_phnum field and run a for loop to
load those sections in memory which have p_type field of struct Proghdr as
ELF_PROG_LOAD.
As required, we check for the file size of the section to be less than its memory
size from their respective fields in struct Proghdr.
region_alloc function stated above is then used so as to allocate memory for each
section according to its size specified in p_memsz field. The virtual address from
where to begin allocation will be provided through the p_va field of Proghdr.
region_alloc function will also insert the mappings established through page_alloc
and page_insert functions into environment's page directory.
Setting the memory allocated to 0, we head over to the start address of each
section provided through p_offset field and copy the data to the VA specified by
p_va  field. The amount of data copied will be equal tot p_filesz field. The
remainder of the memory will then be set to 0.
We then set the entry point for the environment i.e. load the address EIP will
begin execution of the environment from into the struct TrapFrame's field of
tf_eip from the the e_entry field of struct Elf. region_alloc is then used again
```

```
to establish mappings in environment’s page directory and allocate a page for its
stack. Finally, we load the CR3 register again with the kernel’s page directory
since we are still in the kernel mode.
*/
static void
load_icode(struct Env *e, uint8_t *binary)
{
     // Hints:
     //  Load each program segment into virtual memory
     //  at the address specified in the ELF segment header.
     //  You 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.
     //  (The ELF header should have ph->p_filesz <= ph->p_memsz.)
     //  Use functions from the previous lab to allocate and map pages.
     //
     //  All page protection bits should be user read/write for now.
     //  ELF segments are not necessarily page-aligned, but you can
     //  assume for this function that no two segments will touch
     //  the same virtual page.
     //
     //  You may find a function like region_alloc useful.
     //
     //  Loading the segments is much simpler if you can move data
     //  directly into the virtual addresses stored in the ELF binary.
     //  So which page directory should be in force during
     //  this function?
     //
     //  You must also do something with the program's entry point,
     //  to make sure that the environment starts executing there.
     //  What?  (See env_run() and env_pop_tf() below.)

// LAB 3: Your code here.

     struct Elf* p_binary = (struct Elf*) binary;

     if (p_binary -> e_magic != ELF_MAGIC)
          panic ("Invalid ELF File \n");

     lcr3 (PADDR(e -> env_pgdir));

     struct Proghdr* ph_browse = (struct Proghdr*) (binary + p_binary -> e_phoff);
     struct Proghdr* ph_entries = ph_browse + p_binary -> e_phnum;

     for (; ph_browse < ph_entries ; ph_browse ++)
     {
```

```
          if (ph_browse -> p_type != ELF_PROG_LOAD)
               continue;

          if (ph_browse -> p_filesz > ph_browse -> p_memsz)
               panic("Error in ElF File \n");

          region_alloc (e, (void*)ph_browse -> p_va, ph_browse -> p_memsz);
          memset ((void*)ph_browse -> p_va ,0 , ph_browse -> p_memsz);
          void* seg_offset = (void*) (binary + ph_browse -> p_offset);
          memcpy ((void*)ph_browse -> p_va, seg_offset, ph_browse -> p_filesz);
           memset ((void*)ph_browse -> p_va + ph_browse -> p_filesz , 0, (ph_browse
-> p_memsz - ph_browse -> p_filesz));
     }

     e -> env_tf.tf_eip = p_binary -> 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);
     memset ((void*) (USTACKTOP - PGSIZE), 0, PGSIZE);

     lcr3(PADDR(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, enum EnvType type)
{
     // LAB 3: Your code here.
     struct Env* new_env = NULL;
     if ((env_alloc(&new_env, 0)) < 0)
          panic ("Environment Allocation Failed \n");

     load_icode (new_env, binary);
     new_env -> env_type = type;
}

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

```
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 != NULL)
		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);
	panic ("env_pop_tf somehow returned !!! \n");

//	panic("env_run not yet implemented");
}

//
// Restores the register values in the Trapframe with the 'iret' instruction.
// This exits the kernel and starts executing some environment's code.
//
// This function does not return.
//
/*
With Trapframe structure being the place where entire data pertaining to a particular environment will be stored, this function restores the values inside of struct Trapframe's fields and begins execution of the program through the iret instruction.
iret: the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS
```

registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution.

```
*/
void
env_pop_tf(struct Trapframe *tf)
{
	asm volatile(
		"\tmovl %0,%%esp\n"
		"\tpopal\n"
		"\tpopl %%es\n"
		"\tpopl %%ds\n"
		"\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
		"\tiret\n"
		: : "g" (tf) : "memory");
	panic("iret failed");  /* mostly to placate the compiler */
}
```

An important thing to notice is the following piece of code in env_alloc function.

```
// Clear out all the saved register state,
	// to prevent the register values
	// of a prior environment inhabiting this Env structure
	// from "leaking" into our new environment.
	memset(&e->env_tf, 0, sizeof(e->env_tf));

	// Set up appropriate initial values for the segment registers.
	// GD_UD is the user data segment selector in the GDT, and
	// GD_UT is the user text segment selector (see inc/memlayout.h).
	// The low 2 bits of each segment register contains the
	// Requestor Privilege Level (RPL); 3 means user mode.  When
	// we switch privilege levels, the hardware does various
	// checks involving the RPL and the Descriptor Privilege Level
	// (DPL) stored in the descriptors themselves.
	e->env_tf.tf_ds = GD_UD | 3;
	e->env_tf.tf_es = GD_UD | 3;
	e->env_tf.tf_ss = GD_UD | 3;
	e->env_tf.tf_esp = USTACKTOP;
	e->env_tf.tf_cs = GD_UT | 3;
	// You will set e->env_tf.tf_eip later.
```

The eip has been set in the load_icode method for the environment. The value of 3 added to each segment specifies that the Requested Privilege Level (RPL) for the environment is User Mode.
We don't need to set general purpose register when we allocate an enviroment, it's ok to be 0. When we iret from the kernel mode, we return to user mode, thus we pops the stack pointer(esp) and SS from the stack. In this way, we now use user stack(USTACKTOP)

Once you are done you should compile your kernel and run it under QEMU. If all

goes well, your system should enter user space and execute the hello binary until it makes a system call with the int instruction. At that point there will be trouble, since JOS has not set up the hardware to allow any kind of transition from user space into the kernel. When 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". Usually, you would then see the CPU reset and the system reboot.

As of now, the code is entering the user mode. The faulting instruction, its address:
0x800a3b   int $0x30

Handling Interrupts and Exceptions

At this point, the first int $0x30 system call instruction in user space is a dead end: once the processor gets into user mode, there is no way to get back out. You will now need to implement basic exception and system call handling, so that it is possible for the kernel to recover control of the processor from user-mode code.

Basics of Protected Control Transfer

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.

In order to ensure that these protected control transfers are actually protected, the processor's interrupt/exception mechanism is designed so that the code currently running when the interrupt or exception occurs does not get to choose arbitrarily where the kernel is entered or how. Instead, the processor ensures that the kernel can be entered only under carefully controlled conditions. On the x86, two mechanisms work together to provide this protection:

1. ***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 value to load into the code segment (CS) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. (In JOS, all exceptions are handled in kernel mode, privilege level 0.)*

2. ***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.

Although the TSS is large and can potentially serve a variety of purposes, JOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in JOS is privilege level 0 on the x86, the processor uses the ESP0 and SS0 fields of the TSS to define the kernel stack when entering kernel mode. JOS doesn't use any other TSS fields.

Types of Exceptions and Interrupts

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.

An Example

Let's put these pieces together and trace through an example. Let's say the processor is executing code in a user environment and encounters a divide instruction that attempts to divide by zero.

1. *The processor switches to the stack defined by the SS0 and ESP0 fields of the TSS, which in JOS will hold the values GD_KD and KSTACKTOP, respectively.*

2.*The processor pushes the exception parameters on the kernel stack, starting at address KSTACKTOP:*

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

3.*Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets CS:EIP to point to the handler function described by the entry.*
4.The handler function takes control and handles the exception, for example by terminating the user environment.

For certain types of x86 exceptions, in addition to the "standard" five words above, the processor pushes onto the stack another word containing an error code. The page fault exception, number 14, is an important example. See the 80386 manual to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the exception handler when coming in from user mode:

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

Nested Exceptions and Interrupts

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 (the low 2 bits of the CS register are already zero), 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. This capability is an important tool in implementing protection

If the processor is already in kernel mode and takes a nested exception, since it does not need to switch stacks, it does not save the old SS or ESP registers. For

exception types that do not push an error code, the kernel stack therefore looks like the following on entry to the exception handler:

```
                    +--------------------+ <---- old ESP
                    |     old EFLAGS     |     " - 4
                    | 0x00000 | old CS   |     " - 8
                    |      old EIP       |     " - 12
                    +--------------------+
```

For exception types that push an error code, the processor pushes the error code immediately after the old EIP, as before.

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.

Setting Up the IDT

The header files inc/trap.h and kern/trap.h contain important definitions related to interrupts and exceptions that you will need to become familiar with. The file kern/trap.h contains definitions that are strictly private to the kernel, while inc/trap.h contains definitions that may also be useful to user-level programs and libraries.

The overall flow of control that you should achieve is depicted below:

```
      IDT                   trapentry.S         trap.c

+----------------+
|   &handler1    |---------> handler1:          trap (struct Trapframe *tf)
|                |             // do stuff      {
|                      |                   call  trap              // handle  the
exception/interrupt
|                |             // ...           }
+----------------+
|   &handler2    |--------> handler2:
|                |            // do stuff
|                |            call trap
|                |            // ...
+----------------+
       .
       .
       .
+----------------+
|   &handlerX    |--------> handlerX:
|                |             // do stuff
|                |             call trap
|                |             // ...
+----------------+
```

Each exception or interrupt should have 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 (see inc/trap.h) on the stack and call trap() (in trap.c) 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?)*

Consider using the pushal instruction; it fits nicely with the layout of the struct Trapframe.

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.*

Initially, we set the handlers in kern/trapentry.S for the descriptors to be setup in the IDT in kern/trap.c. In order to determine which exceptions / interrupts push an error code on to the stack so as to use the appropriate TRAPHANDLER macro, refer to the following link.

https://wiki.osdev.org/Exceptions

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */


TRAPHANDLER_NOEC (divide_exception, T_DIVIDE);
TRAPHANDLER_NOEC (debug_exception, T_DEBUG);
TRAPHANDLER_NOEC (nmi_interupt, T_NMI);
TRAPHANDLER_NOEC (breakpoint_exception, T_BRKPT);
TRAPHANDLER_NOEC (overflow_exception, T_OFLOW);
TRAPHANDLER_NOEC (bounds_check_exception, T_BOUND);
```

```
TRAPHANDLER_NOEC (illegal_opcode_exception, T_ILLOP);
TRAPHANDLER_NOEC (coprocessor_exception, T_DEVICE);
TRAPHANDLER (double_fault_exception, T_DBLFLT);
TRAPHANDLER (tss_exception, T_TSS);
TRAPHANDLER (segment_np_exception, T_SEGNP);
TRAPHANDLER (stack_np_excecption, T_STACK);
TRAPHANDLER (general_protection_fault, T_GPFLT);
TRAPHANDLER (page_fault_exception, T_PGFLT);
TRAPHANDLER_NOEC (fp_err_exception, T_FPERR);
TRAPHANDLER (alignment_exception, T_ALIGN);
TRAPHANDLER_NOEC (machine_exception, T_MCHK);
TRAPHANDLER_NOEC  (SIMDerr_exception, T_SIMDERR);


/*
 * Lab 3: Your code here for _alltraps
 */
/*
The TRAPHANDLER macro above will push an error code on to the stack and jump to the _alltraps code below. _alltraps will push the existing ds and es onto the stack along with all the register state i.e. it will push will the enirre existing processor state on to the stack to resume once the exception / interrupt has been resolved.
A brief sumary of pushal instruction is as follows:
Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the the first register.
We then load the address of kernel data segment in %ds and %es registers and so as to provide a pointer to the values pushed over the stack i.e. for the entire struct Trapframe which will be given as a pointer to the trap function in kern/trap.c, we again the push the latest value of esp on the stack and call the trap function as has been asked for in the question.
*/
```

```
_alltraps:
pushl %ds
pushl %es
pushal
movw $GD_KD, %ax
movw %ax, %ds
movw %ax, %es
pushl %esp
call trap
```

FILE NAME: kern/trap.h

```
/*
Exception and Interrupt handler function declarations for the respective functions to be invoked.
While the description of TRAPHANDLER explicitly states these functions shouldn't be called in C code, we declare them none the less mainly for the purpose of providing function pointers to the descriptors in IDT during its setup.
*/

void divide_exception ();
void debug_exception ();
void nmi_interupt ();
void breakpoint_exception ();
void overflow_exception ();
void bounds_check_exception ();
void illegal_opcode_exception ();
void coprocessor_exception ();
void double_fault_exception () ;
void tss_exception ();
void segment_np_exception ();
void stack_np_excecption ();
void general_protection_fault ();
```

```
void page_fault_exception ();
void fp_err_exception ();
void alignment_exception ();
void machine_exception ();
void SIMDerr_exception ();


/*
Again use the link provided above to determine which of the descriptors should be
set as traps and which as interrupts. The SETGATE macro is used to set up normal
interrupt / trap descriptor.
*/
trap_init(void)
{
        extern struct Segdesc gdt[];


        // LAB 3: Your code here.


        SETGATE(idt[T_DIVIDE], true, GD_KT, divide_exception, 0);
        SETGATE(idt[T_DEBUG], true, GD_KT, debug_exception, 0);
        SETGATE(idt[T_NMI], false, GD_KT, nmi_interupt, 0);
        SETGATE(idt[T_BRKPT], true, GD_KT, breakpoint_exception, 3);
        SETGATE(idt[T_OFLOW], true, GD_KT, overflow_exception, 0);
        SETGATE(idt[T_BOUND], true, GD_KT, bounds_check_exception, 0);
        SETGATE(idt[T_ILLOP], true, GD_KT, illegal_opcode_exception, 0);
        SETGATE(idt[T_DEVICE], true, GD_KT, coprocessor_exception, 0);
        SETGATE(idt[T_DBLFLT], false, GD_KT, double_fault_exception, 0);
        SETGATE(idt[T_TSS], true, GD_KT, tss_exception, 0);
        SETGATE(idt[T_SEGNP], true, GD_KT, segment_np_exception, 0);
        SETGATE(idt[T_STACK], true, GD_KT, stack_np_excecption, 0);
        SETGATE(idt[T_GPFLT], true, GD_KT, general_protection_fault, 0);
        SETGATE(idt[T_PGFLT], true, GD_KT, page_fault_exception,0);
        SETGATE(idt[T_FPERR], true, GD_KT, fp_err_exception, 0);
        SETGATE(idt[T_ALIGN], true, GD_KT, alignment_exception, 0);
```

```
        SETGATE(idt[T_MCHK], false, GD_KT, machine_exception, 0);
        SETGATE(idt[T_SIMDERR], true, GD_KT, SIMDerr_exception, 0);


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

Questions

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

1. *What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)*

Each exception / interrupt is handled in a unique fashion by the kernel depending on the reason that caused the same and its vector number. As such, while interrupts rest the IF in Flags register, exceptions do not. Additionally, with exceptions being of the type ABORT, execution of the program is terminated completely and not resumed as is the case with faults and traps. As such, having a single handler for all would cripple these very abilities, thus, significantly reducing protection mechanism for the kernel while also reducing its exception / interrupt handling capabilities to a bare minimum.

2. *Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int $14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?*

We did not have to do anything to make the program behave correctly.

The user-space program is not supposed to be able to trigger trap 14 (page fault). We explicitly forbid the user to trigger these CPU-generated interrupts artificially (using the int instruction) by setting dpl to 0 at the interrupt gate (in the SETGATE macro). This means that if the user space program artificially issues an interrupt that it is not authorized to, a protection fault will be generated instead (trap 13). If the kernel actually allows this interrupt to be triggered by the user space program, the page fault handler would have run. However, a potentially invalid value might be stored in cr2 (the virtual address which caused the fault), and depending on how the page fault handler is implemented, this can potentially cause allocation of pages that the user process is not otherwise authorized to do, or some other unintended kernel bugs that can compromise the security or the stability of the system.

PART A COMPLETED

OVERVIEW TILL NOW:

With the Interrupt Descriptor Table (IDT) now set up in the trap_init function with SETGATE macro in kern/trap.c. Each time an interrupt / exception is triggered asynchronously, externally or through the user code, the IDT will be referred using the vector number (multiplied by 8) as an offset. With descriptors in the IDT / gates containing pointer (function pointers) for the handler functions which have been declared in kern/trap.h, the very handler functions in kern/trapentry.S, defined through TRAPHANDLER macros, will be referenced (again based on the vector number). Pushing the vector number on the stack, the handlers will jump to the _alltraps code in the same file. The _alltraps push the %ds, %es on the stack along with all the registers i.e. it will push the current trap-frame of the environment on to the stack. Finally pushing the value of esp which can be used as a pointer to the already pushed trap-frame, it will call the trap function in kern/trap.c.

The trap function performs some important functionality which begins by checking if the Interrupt flag has been cleared i.e. whether interrupts are disabled. It will then check of the trap frame which has just been pushed on to the stack by alltraps belongs to a user environment. IN the event it does belong it will set the trapframe of the currenv (which houses the currently executing environment, now suspended due to exception / interrupt) with the trap frame existing on the kernel stack so that the environment can be restarted later on. It then calls, the function trap_dispatch with the trap frame of _alltraps, present on the kernel stack from KSTACKTOP, as a parameter. Depending on the vector number of the trap / interrupt, the trap dispatch function will take necessary actions, much of which will be implemented in PART B of the Lab. The trap_dispatch function, as of now, will check the trap number and if some action to be implemented for the trap exists, will take the action and return. If no action for trap is present, it will destroy the user environment and return. The function will panic if trap is encountered in kernel i.e. kernel's text segment, i.e. kernel's code. We now, for the future, know that if the kernel panics with unhandled trap message, that means some of the code we wrote is inappropriate. On trap_dispatch returning, the trap function will check if the environment that caused the trap or the environment that was executing when interrupt occurred still exists and if it still running (the latter is for the time whence we implement scheduling for multitasking). If yes, the trap function calls the env_run function to resume execution of the environment. As is evident, since env_run never returns, trap function too will never return.

Part B: Page Faults, Breakpoints Exceptions, and System Calls

Handling Page Faults

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 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****. For instance,* ***make run-hello-nox*** *runs the hello user program.*

```
// Handle processor exceptions.

// LAB 3: Your code here.

/*

The explanation for the code is pretty simple. Checking if the trap is page fault one, we call the page_fault handler function with appropriate trap frame i.e. trapframe pushed on the stack by _alltraps.

*/

if (tf -> tf_trapno == T_PGFLT)

{

     page_fault_handler(tf);

     return;

}
```

The Breakpoint Exception

The breakpoint exception, interrupt vector 3 (T_BRKPT), is normally used to allow debuggers to insert breakpoints in a program's code by temporarily replacing the relevant program instruction with the special 1-byte int3 software interrupt instruction. 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. The user-mode implementation of panic() in lib/panic.c, for example, performs an int3 after displaying its panic message.

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.*

```
        static void

trap_dispatch(struct Trapframe *tf)

{

        // Handle processor exceptions.

        // LAB 3: Your code here.
```

```
/*
Quite simple. Simply call the function monitor defined in kern/monitor.c.
*/
        if (tf -> tf_trapno == T_PGFLT)
        {
                page_fault_handler(tf);
                return;
        } else if (tf -> tf_trapno == T_BRKPT)
        {
                monitor (tf);
                return;
        }
```

Questions

3. *The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?*

The Descriptor Privilage field of the SETGATE macro shouold be set to 3 for breakpoint exception (T_BRKPT) for user code to invoke the breakpoint exception.

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

Settiing the DPL field to 0, will mean that only the kernel can trigger the exception. As such, in the event of instructions such as int 0x03 being executed by programs having DPL of 3 will trigger exception 13 I.e general protetion fault as the excecption that can be triggered by the kernel has been triggered by user program.

4. *What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?*

The main point of these exceptions is to protect the program from malicious / buggy code executing in user space which may trigger exceptions for kernel to invoke and execute their handlers in situations where the same is hardly necessary. Additionally, absence of these mechanism would provide too much control of the operating system to the user program which can eventually bend it to its will causing the system to crash.

System calls

User processes ask the kernel to do things for them by invoking system calls. When the user process invokes a system call, the processor enters kernel mode, the

processor and the kernel cooperate to save the user process's state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process. The exact details of how the user process gets the kernel's attention and how it specifies which call it wants to execute vary from system to system.

In the JOS kernel, we will use the int instruction, which causes a processor interrupt. In particular, we will use int $0x30 as the system call interrupt. We have defined the constant T_SYSCALL to 48 (0x30) for you. You will have to set up the interrupt descriptor to allow user processes to cause that interrupt. Note that interrupt 0x30 cannot be generated by hardware, so there is no ambiguity caused by allowing user code to generate it.

The application will pass the system call number and the system call arguments in registers. This way, the kernel won't need to grub around in the user environment's stack or instruction stream. The system call number will go in %eax, and the arguments (up to five of them) will go in %edx, %ecx, %ebx, %edi, and %esi, respectively. The kernel passes the return value back in %eax. The assembly code to invoke a system call has been written for you, in syscall() in lib/syscall.c. You should read through it and make sure you understand what is going on.

***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. Handle all the system calls listed in inc/syscall.h by invoking the corresponding kernel function for each call.*

*Run the user/hello program under your kernel (**make run-hello**). 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.*

Add handler for the system calls to be made.

In file kern/trap.c in function trap_init, we add a descriptor i.e. gate for the syscalls in the IDT. The below is necessary as system calls will be executed by issuing a software interrupt through the INT instruction.

SETGATE (idt[T_SYSCALL], false, GD_KT, syscall_interrupt, 3);

Similarly, we add the function declartion of syscall_interrupt in kern/trap.h and add the followinig line in kern/trapentry.S

TRAPHANDLER_NOEC (syscall_interrupt, T_SYSCALL);

which is the actual handler for system call.

Heading into the inc/syscall.h file, we notice the following system calls will have to be implemented in kern/syscall.c

```
enum {
     SYS_cputs = 0,
     SYS_cgetc,
     SYS_getenvid,
     SYS_env_destroy,
     NSYSCALLS
};
```

Heading into kern/syscall.c, our task is made easy as three of the four system calls are already being implemented leaving us just the task of implementing one system call and calling the system call functions at appropriate places through the switch in syscall function.

```
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
/*
The implementation is super easy as a function to do the same exists in kern/pmap.c. user_mem_assert will check if the environment passed as a parameter has the right to access the memory in the range of va + len with permissions PTE_U | PTE_P. Its declaration is as follows:
void user_mem_assert(struct Env *env, const void *va, size_t len, int perm)
The function will destroy the environment if it doent have the rights which pretty much does out task. We just check if the environment is a valid user environment and call the function.
*/
        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.
        if (curenv -> env_tf.tf_cs &3)
               user_mem_assert (curenv, (const void*)s, len, PTE_U | PTE_P);
```

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


// Dispatches to the correct kernel function, passing the arguments.
/*
Dispatches / Calls the appropriate syscall function according to the sycall made which can be, as of now, one of the four defined in inc/syscall.h. As such, we just call the handler functions for the respective system calls in the switch. Since sys_cputs returns void, we explicitly return 0 within the function.
*/
        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.

        //     panic("syscall not implemented");

        switch (syscallno) {

                case SYS_cputs:
                        sys_cputs ((const char*) a1, 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);
```

```
                default:
                        return -E_INVAL;
        }
}.
```

Finally, coming back to kern/trap.c to modify trap_dispatch to handle the system call, we put in the following code.

```
// Handle processor exceptions.
        // LAB 3: Your code here.


        if (tf -> tf_trapno == T_PGFLT)
        {
                page_fault_handler(tf);
                return;
        } else if (tf -> tf_trapno == T_BRKPT)
        {
                monitor (tf);
                return;
        } else if (tf -> tf_trapno == T_SYSCALL)
        {
                cprintf("SYSCALL Initiated \n");
                    int32_t return_value = 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);
                tf -> tf_regs.reg_eax = return_value;
                return;
        }
```

As has been mentioned, the user environment on making system call will pass its number, i.e. the number associated with one of the four system calls in inc/syscall.h in register eax and argument for the system call functions in registers %edx, %ecx, %ebx, %edi, and %esi, respectively, we pass the very register values stored by the environment to the syscall function as its six parameters and store the return value back in register eax of the environment so that the return value can be passed to the user process.

The overall process happening is as follows:

User process on making int $0x30 instruction, the kernel will refer the IDT and know it is a system call.

Invoking its appropriate handler from kern/trapentry.S, it will execute alltraps which wil l push envirnment state on to kernel stack and call trap function. Checking that the trap has occurred from user mode, the trap function will save the environment's trap frame struct and call trapdispatch.

The trap_dispatch function will check the trap number and call the syscall function from kern/syscall.c providing the register values of the environment as arguments of the function.

Syscall function will then check the first parameter ie. value provided from register eax, for the system call number i.e. which system call has been invoked and pass control over to the appropriate system call handler function.

The return value of the function if any will be returned by syscall and saved in eax register of the user process to make it available to the process when the process eventually resumes execution when trap_dispatch returns, the trap function verifies that the environment is active and still running and calls the env_run function.

Testbss test of make grade succeeds at this point.

User-mode startup

A user program starts running at the top of lib/entry.S. After some setup, this code calls libmain(), in lib/libmain.c. You should modify libmain() to initialize the global pointer thisenv to point at this environment's struct Env in the envs[] array. (Note that lib/entry.S has already defined envs to point at the UENVS mapping you set up in Part A.) Hint: look in inc/env.h and use sys_getenvid.

libmain() then calls umain, which, in the case of the hello program, is in user/hello.c. Note that after printing "hello, world", it tries to access thisenv->env_id. This is why it faulted earlier. Now that you've initialized thisenv properly, it should not fault.

Investigating the reason for the user program to srart running at top of entry.S in lib directory, I found out the following piece of code in user/Makefrag.

$(OBJDIR)/user/%: $(OBJDIR)/user/%.o $(OBJDIR)/lib/entry.o $(USERLIBS:%=$(OBJDIR)/lib/lib%.a) user/user.ld

@echo + ld $@

(V)(LD) -o $@ $(ULDFLAGS) $(LDFLAGS) -nostdlib $(OBJDIR)/lib/entry.o $@.o -L$(OBJDIR)/lib $(USERLIBS:%=-l%) $(GCC_LIB)

(V)(OBJDUMP) -S $@ > $@.asm

(V)(NM) -n $@ > $@.sym

which wires for the user rograms to be run above lib/entry.S i.e. entry.S will initially run when switches to user mode and then check for arguments on stack. If

no arguments found, it would just drop down into libmain in lib/libmain.c and call which ever user program is to be executed.

As for the specific address of each user program, examining the user/user.ld file, the following can be found.

```
SECTIONS
{
     /* Load programs at this address: "." means the current address */
     . = 0x800020;


     .text : {
          *(.text .stub .text.* .gnu.linkonce.t.*)
     }


     PROVIDE(etext = .); /* Define the 'etext' symbol to this value */
```

As is evident, the linker links the user program that is to be executed at memory address 0x800020. This is the reason why all user programs, currently in /user directory begin executing at 0x800020.

In current context, this means that we link the entry point of hello to 0x800020. when we drop to user mode by iret, we go to 0x800020. And this address stores hello compiled by entry.o and hello.o

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.*

```
  // set thisenv to point at our Env structure in envs[].
     // LAB 3: Your code here.
/*
sys_envid will return the environment ID of current environment. The environment index which will be given by ENVX macro i.e. last 10 bits of envid will be used as offset in the envs array to get the current environment. We use the & sign as we have been asked to make thisenv point towads the Env Structure which would not be possible without the & symbol.
*/
     envid_t id = sys_getenvid();
```

```
thisenv = &envs [ENVX(id)];
```

Page faults and memory protection

Memory protection is a crucial feature of an operating system, ensuring that bugs in one program cannot corrupt other programs or corrupt the operating system itself.

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.

As an example of a fixable fault, consider an automatically extended stack. In many systems the kernel initially allocates a single stack page, and then if a program faults accessing pages further down the stack, the kernel will allocate those pages automatically and let the program continue. By doing this, the kernel only allocates as much stack memory as the program needs, but the program can work under the illusion that it has an arbitrarily large stack.

System calls present an interesting problem for memory protection. Most system call interfaces let user programs pass pointers to the kernel. These pointers point at user buffers to be read or written. The kernel then dereferences these pointers while carrying out the system call. There are two problems with this:

1. *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.*
2. *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.*

For both of these reasons the kernel must be extremely careful when handling pointers presented by user programs.

You will now solve these two problems with a single mechanism that scrutinizes all pointers passed from userspace into the kernel. When a program passes the kernel a pointer, the kernel will check that the address is in the user part of the address space, and that the page table would allow the memory operation.

Thus, the kernel will never suffer a page fault due to dereferencing a user-supplied pointer. If the kernel does page fault, it should panic and terminate.

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. What causes this page fault? You don't need to fix it, but you should understand why it happens.*

In the function page_fault_handler in the file kern/trap.c we add the following code

```
// Handle kernel-mode page faults.
        // LAB 3: Your code here.
/*
With the function being invoked in case of the pasge fault occuring due to some fault virtual address, we check the priviloage level of the address in its last two bits. If the privilage level is zero, we conclude, that the page fault occurred in the kernel.
*/
        if((tf -> tf_cs & 0x03) == 0)
                panic ("Page Fault in Kernel at address %x", fault_va);
```

FILE: kern/pmap.c

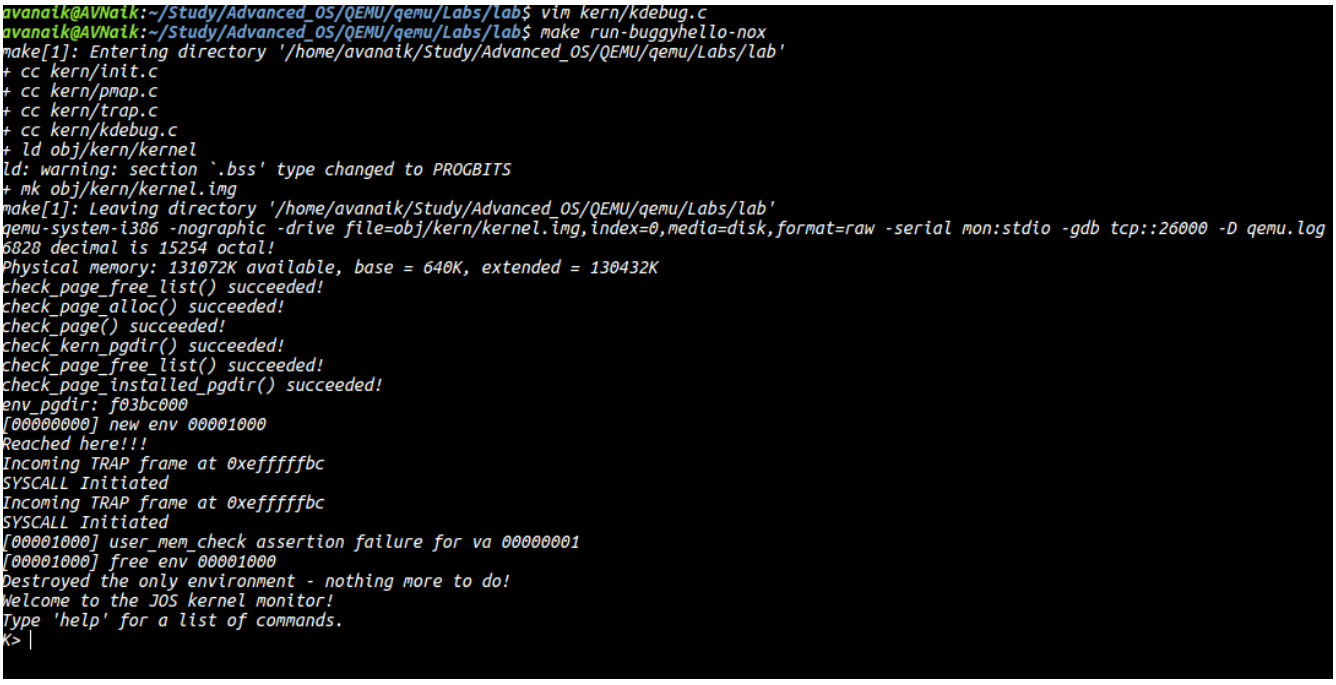
```
// Check that an environment is allowed to access the range of memory
// [va, va+len) with permissions 'perm | PTE_P'.
// Normally 'perm' will contain PTE_U at least, but this is not required.
// 'va' and 'len' need not be page-aligned; you must test every page that
// contains any of that range.  You will test either 'len/PGSIZE',
```

```
// 'len/PGSIZE + 1', or 'len/PGSIZE + 2' pages.
//
// 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_chec0_addr' variable to the first
// erroneous virtual address.
//
// Returns 0 if the user program can access this range of addresses,
// and -E_FAULT otherwise.
/*
With it being stated, that va and len don’t have to be page aligned, we take a pointer to va and increment it in steps of PGSIZE i.e. 4096, rounding it down to its lower every time until it is less than the sum of va and len parameters.
The page_lookup method used will return the address of the struct PageInfo associated with the page pointed to by pointer “a” as it is incremented by PGSIZE for each iteration of the for loop. With the third parameter of the page_lookup method containing the address of the page table entry for the page, we check if the permissions for the page containing the address in its page table match with those provided as parameters for the function and PTE_P. Lastly, we also check if the address, initially and as it is incremented, goes above ULIM. In either of the cases holding true or if the page_lookup method failing that there being no page mapped at the virtual address, we stire the erroneous address in user_mem_check_address variable and return -E_FAULT signaling an error.
*/
        int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
        // LAB 3: Your code here.


          for (char* a = (char*) va; a < (char*) va + len; a = ROUNDDOWN (a + PGSIZE, PGSIZE))
        {
                pte_t* pte = NULL;
                    struct PageInfo* mapped_page = page_lookup (env -> env_pgdir, (void*) a, &pte);
```

```
                if ((!mapped_page) || !(*pte & (perm | PTE_P)) || ((uintptr_t)a >=
ULIM))
                {
                        user_mem_check_addr = (uintptr_t) a;
                        return -E_FAULT;
                }
        }
        return 0;
}
```

Running the user program bugghello, we get the following output



Next, we insert the method user_mem_check in kern/kdebug.c's function debuginfo_eip as follows:

```
 // 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, (void*)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, (void*) stabs, (uintptr_t)(stab_end - stabs), PTE_U) < 0) || (user_mem_check (curenv, (void*) stabstr, (uintptr_t) (stabstr_end - stabstr), PTE_U) < 0))
    return -1;
```

Casting to uinptr_t is necessary above as stabs and stuff have different data types that are expected by the function.

Running breakpoint we get the following output

```
avanaik@AVNaik:~/Study/Advanced_OS/QEMU/qemu/Labs/lab$ make run-breakpoint-nox
make[1]: Entering directory '/home/avanaik/Study/Advanced_OS/QEMU/qemu/Labs/lab'
make[1]: Leaving directory '/home/avanaik/Study/Advanced_OS/QEMU/qemu/Labs/lab'
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
env_pgdir: f03bc000
[00000000] new env 00001000
Reached here!!!
Incoming TRAP frame at 0xeffffbc
SYSCALL Initiated
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b4000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800037
  cs   0x----001b
  flag 0x00000082
  esp  0xeebfdfd0
  ss   0x----0023
K> backtrace
Stack Backtrace:
ebp efffff20 eip f0100911 args  00000001  efffff38  f01b4000  00000000  f0172840
           kern/monitor.c:0: monitor+276
ebp efffff90 eip f0103788 args  f01b4000  efffffbc  00000000  00000082  00000000
           kern/trap.c:0: trap+165
ebp efffffb0 eip f01038a1 args  efffffbc  00000000  00000000  eebfdfd0  efffffdc
           kern/syscall.c:0: syscall+0
ebp eebfdfd0 eip 00800073 args  00000000  00000000  eebfdff0  00800049  00000000
           lib/libmain.c:0: libmain+58
ebp eebfdff0 eip 00800031 args  00000000  00000000 Incoming TRAP frame at 0xeffffe94
kernel panic at kern/trap.c:245: Page Fault in Kernel at address eebfe000
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

The kernel panics because of the page fault at address 0xeebfe000.

Examining the inc/memlayout.h file, we conclude that it is the address of USTACKTOP which forms the top of User Environment's stack. As such, beyond the very address no pages are mapped, as a result of which the page fault occurs and the kernel panics since the faulting address will be in kernel's address space.

The address 0xeffffe94 is associated with kernel stack, and as such will be below the address KSTACKTOP in which the trap frames for the user environment are stored.

****

This finishes the LAB.