Lab 4: Preemptive Multitasking

Introduction

In this lab you will implement preemptive multitasking among multiple simultaneously active user-mode environments.

In part A you will add multiprocessor support to JOS, implement round-robin scheduling, and add basic environment management system calls (calls that create and destroy environments, and allocate/map memory).

In part B, you will implement a Unix-like fork(), which allows a user-mode environment to create copies of itself.

Finally, in part C you will add support for inter-process communication (IPC), allowing different user-mode environments to communicate and synchronize with each other explicitly. You will also add support for hardware clock interrupts and preemption.

Part A: Multiprocessor Support and Cooperative Multitasking

In the first part of this lab, you will first extend JOS to run on a multiprocessor system, and then implement some new JOS kernel system calls to allow user-level environments to create additional new environments. You will also implement cooperative round-robin scheduling, allowing the kernel to switch from one environment to another when the current environment voluntarily relinquishes the CPU (or exits). Later in part C you will implement preemptive scheduling, which allows the kernel to re-take control of the CPU from an environment after a certain time has passed even if the environment does not cooperate.

Multiprocessor Support

We are going to make JOS support "symmetric multiprocessing" (SMP), a multiprocessor model in which all CPUs have equivalent access to system resources such as memory and I/O buses. While all CPUs are functionally identical in SMP, during the boot process they can be classified into two types: the bootstrap processor (BSP) is responsible for initializing the system and for booting the operating system; and the application processors (APs) are activated by the BSP only after the operating system is up and running. Which processor is the BSP is determined by the hardware and the BIOS. Up to this point, all your existing JOS code has been running on the BSP.

In an SMP system, each CPU has an accompanying local APIC (LAPIC) unit. The LAPIC units are responsible for delivering interrupts throughout the system. The LAPIC also provides its connected CPU with a unique identifier. In this lab, we make use of the following basic functionality of the LAPIC unit (in kern/lapic.c):

- *Reading the LAPIC identifier (APIC ID) to tell which CPU our code is currently running on (see cpunum()).*

- *Sending the STARTUP interprocessor interrupt (IPI) from the BSP to the APs to bring up other CPUs (see lapic_startap()).*
- *In part C, we program LAPIC's built-in timer to trigger clock interrupts to support preemptive multitasking (see apic_init()).*

A processor accesses its LAPIC using memory-mapped I/O (MMIO). In MMIO, a portion of physical memory is hardwired to the registers of some I/O devices, so the same load/store instructions typically used to access memory can be used to access device registers. You've already seen one IO hole at physical address 0xA0000 (we use this to write to the VGA display buffer). The LAPIC lives in a hole starting at physical address 0xFE000000 (32MB short of 4GB), so it's too high for us to access using our usual direct map at KERNBASE. The JOS virtual memory map leaves a 4MB gap at MMIOBASE so we have a place to map devices like this.

Exercise 1. Implement mmio_map_region in kern/pmap.c. To see how this is used, look at the beginning of lapic_init in kern/lapic.c. You'll have to do the next exercise, too, before the tests for mmio_map_region will run.

```
// Reserve size bytes in the MMIO region and map [pa,pa+size) at this
// location.  Return the base of the reserved region.  size does *not*
// have to be multiple of PGSIZE.
/*
function is self explanatory.
*/
void *
mmio_map_region(physaddr_t pa, size_t size)
{
	// Where to start the next region.  Initially, this is the
	// beginning of the MMIO region.  Because this is static, its
	// value will be preserved between calls to mmio_map_region
	// (just like nextfree in boot_alloc).
	static uintptr_t base = MMIOBASE;

	// Reserve size bytes of virtual memory starting at base and
	// map physical pages [pa,pa+size) to virtual addresses
	// [base,base+size).  Since this is device memory and not
	// regular DRAM, you'll have to tell the CPU that it isn't
	// safe to cache access to this memory.  Luckily, the page
	// tables provide bits for this purpose; simply create the
```

```
	// mapping with PTE_PCD|PTE_PWT (cache-disable and
	// write-through) in addition to PTE_W.  (If you're interested
	// in more details on this, see section 10.5 of IA32 volume
	// 3A.)
	//
	// Be sure to round size up to a multiple of PGSIZE and to
	// handle if this reservation would overflow MMIOLIM (it's
	// okay to simply panic if this happens).
	//
	// Hint: The staff solution uses boot_map_region.
	//
	// Your code here:

	assert (pa % PGSIZE == 0);
	size = ROUNDUP (size, PGSIZE);
	if ((base + size) > MMIOLIM)
		panic ("Memory to be allocated greater than MMIOLIM. Out of memory \n");

	  boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT |
PTE_P);
	void* return_base = (void*) base;
	base += size;
	return return_base;

	//panic("mmio_map_region not implemented");
}
```

Application Processor Bootstrap

Before booting up APs, the BSP should first collect information about the multiprocessor system, such as the total number of CPUs, their APIC IDs and the MMIO address of the LAPIC unit. The mp_init() function in kern/mpconfig.c retrieves this information by reading the MP configuration table that resides in the BIOS's region of memory.

The boot_aps() function (in kern/init.c) drives the AP bootstrap process. APs start in real mode, much like how the bootloader started in boot/boot.S, so boot_aps() copies the AP entry code (kern/mpentry.S) to a memory location that is addressable in the real mode. Unlike with the bootloader, we have some control over where the AP will start executing code; we copy the entry code to 0x7000 (MPENTRY_PADDR), but any unused, page-aligned physical address below 640KB would work.

After that, boot_aps() activates APs one after another, by sending STARTUP IPIs to the LAPIC unit of the corresponding AP, along with an initial CS:IP address at which the AP should start running its entry code (MPENTRY_PADDR in our case). The entry code in kern/mpentry.S is quite similar to that of boot/boot.S. After some brief setup, it puts the AP into protected mode with paging enabled, and then calls the C setup routine mp_main() (also in kern/init.c). boot_aps() waits for the AP to signal a CPU_STARTED flag in cpu_status field of its struct CpuInfo before going on to wake up the next one.

The boot_aps method in kern/init.c method will be used by the bootstrap processor (BSP) to start Application Processors (AP). The symbols / labels mpentry_start [] and mpentry_end are defined in file kern/mpentry.S signifying the beginning and end of the AP's startup code. The BSP, initially, will copy the startup AP code to the predtermined memory location i.e. MPENTRY_PADDR i.e. 0x7000 using memmove as the same supports memory overlap, if any, unlike memcpy. Then, for each AP, it stores address of the pre-allocated per-core stack in mpentry_kstack i.e. the kernel stack of each CPU which is set by the mem_init_mp function in kern/pmap.c, sends the STARTUP IPI, and waits for this code to acknowledge that it has started (which happens in mp_main in init.c). The AP will state that it started by setting its cpu_status filed of struct CpuInfo to CPU_STARTED. The boot of AP will be undertaken by the lapic_startap function defnined in kern/laipc.c. The function takes two parameters which are the local APIC ID provided by the cpu_id field of struct CpuInfo and the physical address from which the entry code will be executed.

Startig up the AP with IPI command which will be sent twice, according to Intel's algorithm, the AP will begin to execute the code at MPENTRY_PADDR i.e. 0x7000. CS and IP values from which it can begin execution of the entry code will also be provided to the AP. Setting up the AP, it will then be put into the 32-bit protected mode and CS : IP registers will be set with values to execute 32-bit code. A32 will not be enabled for the AP for reason that somewhat beats me as of now. Because the EIP will be loaded with values that are still lower addresses, the AP will load the entry_pgidr into it is CR3 register, turn on paging and load the value of the base of the stack allocated to it by the BSP into its esp register i.e. mpentry_kstack's value which has been set in boot_aps. IT shall then call the mp_main method in kern/init.c
The first thing the mp_main function will do is load the kernel's page directory i.e. kern_pgdir into the CR3 rergister of the AP. Calling various initialization functions in the aftermath mainly to set up its local APIC and memory map it into appropriate memory address above MMIOBASE, set up its GDT for user environments to be executed and set up its TSS and IDT. Finally, it will set the cpu_status field of its struct CpupInfo to CPU_STARTED to signal to boot_aps to begin the startup

procedure for the next AP until all Ap's on the system, 4 in case of JOS have been started.

```
/*
T*The changes to the function are as shown below. Importing the symbols defined in
kern/mpentry.S file, we check if the sartup code of AP fits in a page. On
assertion success, we get the virtual address of the struct PageInfo for the page
in which MPENTRY_PADDR address resides. Then we set the the pp_ref field of the
PageInfor struct for the page to be 1. Additionally, we add an if condition to the
for loop which sets rest of the base memory other than page 0 as free to check for
pp_ref field of the page's struct PageInfo. If the pp_ref field is 1, it plainly
goes on to its next iteration, thus avoiding the page containing the address
MPENTRY_PADDR from being again set as free while being the base memory i.e. the
first 640 kB of memory which is where the startup code for AP's resides.
*/
// LAB 4:
        // Change your code to mark the physical page at MPENTRY_PADDR
        // as in use
        extern unsigned char mpentry_start [], mpentry_end [];
        assert((uintptr_t)(mpentry_end - mpentry_start) <= PGSIZE);
        struct PageInfo* start_ap = pa2page(MPENTRY_PADDR);
        start_ap -> pp_ref = 1;

        // The example code here marks all physical pages as free.
        // However this is not truly the case.  What memory is free?
        //  1) Mark physical page 0 as in use.
        //     This way we preserve the real-mode IDT and BIOS structures
        //     in case we ever need them.  (Currently we don't, but...)
        pages[0].pp_ref = 1;
        pages [0].pp_link = NULL;
        //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
        //     is free.
        for (int i = 1; i < npages_basemem; i++)
        {
                if (pages[i].pp_ref ==1)
                        continue;
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages [i];
        }
```

Question

1. *Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?*

Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

Because entry.S is linked to have addresses above KERNBASE, it contains the macro #define RELOC(x) ((x) - KERNBASE) in order to translate the addresses generated by the linker (which are meant to be virtual addresses) to physical addresses (load addresses), where the data is actually stored. For example movl $(RELOC(entry_pgdir)), %eax is required to use the RELOC macro because we don't want to load the "virtual" address of entry_pgdir into %eax but rather the physical (loaded) address. So this RELOC macro is useful for "calculating" the physical addresses of the data defined inside the file itself. A similar concept exists in in mpentry.S where instead we need to use MPBOOTPHYS in order to refer to addresses inside mpentry.S itself. Notice that we'd still use RELOC inside mpentry.S for things that are defined outside of mpentry.S such as entry_pgdir. If we were to omit it, we'd get virtual addresses while the AP CPU has still not enabled paging. This would cause things to fail.

The purpose of MPBOOTPHYS is to change the virtual address we have to a physical address by getting the offset and placing that from the MPENTRY_PADDR.
It is necessary in kern/mpentry.S but not in boot/boot.S because in boot/boot.S is loaded by the BIOS at page 0. Here the kernel is linked at KERNBASE and loaded at 0. But both of these are mapped to the same location in the page table.
If omitted in mpentry.S it will not allow us to load the code correctly (which is below KERNBASE) from the position in the physical memory, the load position.

Per-CPU State and Initialization

When writing a multiprocessor OS, it is important to distinguish between per-CPU state that is private to each processor, and global state that the whole system shares. kern/cpu.h defines most of the per-CPU state, including struct CpuInfo, which stores per-CPU variables. cpunum() always returns the ID of the CPU that calls it, which can be used as an index into arrays like cpus. Alternatively, the macro thiscpu is shorthand for the current CPU's struct CpuInfo.

Here is the per-CPU state you should be aware of:

- ***Per-CPU kernel stack.***
  Because multiple CPUs can trap into the kernel simultaneously, we need a separate kernel stack for each processor to prevent them from interfering with each other's execution. The array percpu_kstacks[NCPU][KSTKSIZE] reserves space for NCPU's worth of kernel stacks.

  In Lab 2, you mapped the physical memory that bootstack refers to as the BSP's kernel stack just below KSTACKTOP. Similarly, in this lab, you will map each CPU's kernel stack into this region with guard pages acting as a buffer between them. CPU 0's stack will still grow down from KSTACKTOP; CPU 1's stack will start KSTKGAP bytes below the bottom of CPU 0's stack, and so on. inc/memlayout.h shows the mapping layout.

- ***Per-CPU TSS and TSS descriptor.***
  A per-CPU task state segment (TSS) is also needed in order to specify where each CPU's kernel stack lives. The TSS for CPU i is stored in cpus[i].cpu_ts, and the corresponding TSS descriptor is defined in the GDT entry gdt[(GD_TSS0 >> 3) + i]. The global ts variable defined in kern/trap.c will no longer be useful.

- ***Per-CPU current environment pointer.***
  Since each CPU can run different user process simultaneously, we redefined the symbol curenv to refer to cpus[cpunum()].cpu_env (or thiscpu->cpu_env), which points to the environment currently executing on the current CPU (the CPU on which the code is running).

- ***Per-CPU system registers.***
  All registers, including system registers, are private to a CPU. Therefore, instructions that initialize these registers, such as lcr3(), ltr(), lgdt(), lidt(), etc., must be executed once on each CPU. Functions env_init_percpu() and trap_init_percpu() are defined for this purpose.

Exercise 3. *Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in inc/memlayout.h. The size of each stack is KSTKSIZE bytes plus KSTKGAP bytes of unmapped guard pages. Your code should pass the new check in check_kern_pgdir().*

```
// Modify mappings in kern_pgdir to support SMP
//   - Map the per-CPU stacks in the region [KSTACKTOP-PTSIZE, KSTACKTOP)
/*
Pretty Straight Forward. Simply follow the hints.
*/
        static void
mem_init_mp(void)
{
        // Map per-CPU stacks starting at KSTACKTOP, for up to 'NCPU' CPUs.
        //
        // For CPU i, use the physical memory that 'percpu_kstacks[i]' refers
        // to as its kernel stack. CPU i's kernel stack grows down from virtual
        // address kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP), and is
        // divided into two pieces, just like the single stack you set up in
        // mem_init:
        //     * [kstacktop_i - KSTKSIZE, kstacktop_i)
        //          -- backed by physical memory
        //     * [kstacktop_i - (KSTKSIZE + KSTKGAP), kstacktop_i - KSTKSIZE)
        //          -- not backed; so if the kernel overflows its stack,
        //             it will fault rather than overwrite another CPU's stack.
        //             Known as a "guard page".
        //     Permissions: kernel RW, user NONE
        //
```

```
        // LAB 4: Your code here:

        for (int i = 0; i < NCPU; i ++)
        {
          uintptr_t stack_start = KSTACKTOP - i*(KSTKSIZE + KSTKGAP);
                boot_map_region (kern_pgdir, stack_start - KSTKSIZE, KSTKSIZE,
PADDR(percpu_kstacks[i]), PTE_W | PTE_P);
        }
}
```

Exercise 4. *The code in trap_init_percpu() (kern/trap.c) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global ts variable any more.)*

```
// Initialize and load the per-CPU TSS and IDT
        void
trap_init_percpu(void)
{

        // The example code here sets up the Task State Segment (TSS) and
        // the TSS descriptor for CPU 0. But it is incorrect if we are
        // running on other CPUs because each CPU has its own kernel stack.
        // Fix the code so that it works for all CPUs.
        //
        // Hints:
        //   - The macro "thiscpu" always refers to the current CPU's
        //     struct CpuInfo;
        //   - The ID of the current CPU is given by cpunum() or
        //     thiscpu->cpu_id;
        //   - Use "thiscpu->cpu_ts" as the TSS for the current CPU,
        //     rather than the global "ts" variable;
        //   - Use gdt[(GD_TSS0 >> 3) + i] for CPU i's TSS descriptor;
        //   - You mapped the per-CPU kernel stacks in mem_init_mp()
        //   - Initialize cpu_ts.ts_iomb to prevent unauthorized environments
        //     from doing IO (0 is not the correct value!)
        //
        // ltr sets a 'busy' flag in the TSS selector, so if you
        // accidentally load the same TSS on more than one CPU, you'll
        // get a triple fault.  If you set up an individual CPU's TSS
        // wrong, you may not get a fault until you try to return from
        // user space on that CPU.
        /*
```

Somewhat difficult function to implement on your own. Thankfully, we have been given the procedure to follow to set up TSS and TSS descriptor for a single processor i.e. BSP. The hints are also helpful. Declaring an instance of struct Taskstate, set it to be equal to the task state of current CPU which is given in hints. We then set the stack pointer, stack segments and iomb fields of the descriptor in accordance with the procedure followed for a single processor.

Setting the GDT descriptor follows the same procedure. We replaced the global variable ts with the newly defined instance and make changes as suggested in the hints.
Initially declaring an instance of the struct Taskstate, we set it to be task state of the currently running CPU. Setting the esp0 and ss0 fields of Taskstate with values of CPU's private stack and its associated segment selector, we then set the I/O map base address.
Indexing into the GDT's entry for the CPU's TSS, which has been set to NULL in kern/env.c, we set the GDT entry for the TSS of he CPU using the SEG16 macro defined in inc/mmu.h. The macro fills in the appropriate fields of the struct Segdesc also ddefined in inc/mmu.h, of which gdt is an array of instances defined in kern/env.c. We then set the sd_s field of the struct Segdesc to 0 indicating the segment is a system segment and not an application segment.
The ltr field used wil set a busy flag on the TSS pushed into the GDT.
ltr INSTRUCTION

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

Finally, ,we load the address and limit of the IDT into the IDTR register.
The idt_pd loaded into the IDTR has its limit field set to sizeof (idt) -1 and base address field set to the address of the idt which is plainly an array of instances of struct Gatedesc, again in inc/mmu.h, which defines the look if each descriptor in IDT much like how Segdesc describes the look of each descriptor in GDT.
**/*

```
        // LAB 4: Your code here:

        // Setup a TSS so that we get the right stack
        // when we trap to the kernel.

        struct Taskstate* ts_cpu = &thiscpu -> cpu_ts;
        ts_cpu -> ts_esp0 = KSTACKTOP - thiscpu -> cpu_id * (KSTKSIZE + KSTKGAP);
        ts_cpu -> ts_ss0 = GD_KD;
```

```
        ts_cpu -> ts_iomb = sizeof (struct Taskstate);

        // Initialize the TSS slot of the gdt.
           gdt[(GD_TSS0 >> 3) + thiscpu -> cpu_id] = SEG16(STS_T32A, (uint32_t)
(ts_cpu),
                        sizeof(struct Taskstate) - 1, 0);
        gdt[(GD_TSS0 >> 3) + thiscpu -> cpu_id].sd_s = 0;

        // Load the TSS selector (like other segment selectors, the
        // bottom three bits are special; we leave them 0)
        ltr(GD_TSS0 + (thiscpu -> cpu_id << 3));

        // Load the IDT
        lidt(&idt_pd);


        /*                ts.ts_esp0 = KSTACKTOP;
                          ts.ts_ss0 = GD_KD;
                          ts.ts_iomb = sizeof(struct Taskstate);

        // Initialize the TSS slot of the gdt.
        gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
        sizeof(struct Taskstate) - 1, 0);
        gdt[GD_TSS0 >> 3].sd_s = 0;

 // Load the TSS selector (like other segment selectors, the
        // bottom three bits are special; we leave them 0)
        ltr(GD_TSS0);

        // Load the IDT
        lidt(&idt_pd);*/
```

Locking

Our current code spins after initializing the AP in mp_main(). Before letting the AP get any further, we need to first address race conditions when multiple CPUs run kernel code simultaneously. The simplest way to achieve this is to use a big kernel lock. The big kernel lock is a single global lock that is held whenever an environment enters kernel mode, and is released when the environment returns to user mode. In this model, environments in user mode can run concurrently on any available CPUs, but no more than one environment can run in kernel mode; any other environments that try to enter kernel mode are forced to wait.

kern/spinlock.h declares the big kernel lock, namely kernel_lock. It also provides lock_kernel() and unlock_kernel(), shortcuts to acquire and release the lock. You should apply the big kernel lock at four locations:

- *In i386_init(), acquire the lock before the BSP wakes up the other CPUs.*

- *In mp_main(), acquire the lock after initializing the AP, and then call sched_yield() to start running environments on this AP.*
- *In trap(), acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the tf_cs.*
- *In env_run(), release the lock right before switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.*

Exercise 5. *Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.*

The i386_init function is called by BSP in which the boot_aps function will be called to boot up the rest of the AP's. As such, we acquire the kernel lock before calling the boot_aps method to ensure that the the BSP maintains control over the kernel and its data structure and the booted up AP's don't begin to execute kernel code while BSP is still booting up the other AP's one after the another.

In mp_main function, we now make the AP's gain the kernel lock. This is essential as the lock was earlier by the kernel (prior to boot_aps) and we don't want the AP's to race against each other and the BSP inside the kernel.

With the processor going into the kernel mode on trap occuring in the user mode, , we need the processor to initially acquire the kernel lock as we might eventually need to modify some of kernel's data structures pertaining to pages or user environments.

As we want to enable the processors to run simulatously when they are executing the user mode code, we release the kernel lock.

Thus, we

Acquire the kernel lock when we switch to kernel mode from user mode. the only way this can happen is through an interrupt/trap.

Release the lock when we leave the kernel mode.

Some special care is taken during initialization to ensure that the AP's and the BSP don't step on each other's feet.

The big kernel lock defined in kern/spinlock.c and kern/spinlock.h is basically the struct spinlock in which the unsigned locked variable is set ot 1 when the lock is acquired and set to 0 when the lock is released. The rest of the structure elements is used to record information about the CPU holding the lock.

The lock is held and released with the xchg instruction as its atomic and also serializes, so that reads after acquire are not reordered before it.

The code to be written is nothing just add the methods lock_kernel and unlock_kernel at appropriate places,

Question

> *It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.*

Consider the following scenario: a processor is changing from user mode to kernel mode and the hardware pushes information on the kernel stack. Then, before the processor got a chance to do something with that information, another processor also switches from user mode and kernel mode and overrides the information the first processor put on the kernel stack. Notice that the kernel lock doesn't prevent the hardware from overriding the kernel stack.

Round-Robin Scheduling

Your next task in this lab is to change the JOS kernel so that it can alternate between multiple environments in "round-robin" fashion. Round-robin scheduling in JOS works as follows:

- *The function sched_yield() in the new kern/sched.c is responsible for selecting a new environment to run. It searches sequentially through the envs[] array in circular fashion, starting just after the previously running environment (or at the beginning of the array if there was no previously running environment), picks the first environment it finds with a status of ENV_RUNNABLE (see inc/env.h), and calls env_run() to jump into that environment.*
- *sched_yield() must never run the same environment on two CPUs at the same time. It can tell that an environment is currently running on some CPU (possibly the current CPU) because that environment's status will be ENV_RUNNING.*
- *We have implemented a new system call for you, sys_yield(), which user environments can call to invoke the kernel's sched_yield() function and thereby voluntarily give up the CPU to a different environment.*

Exercise 6. *Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().*

Make sure to invoke sched_yield() in mp_main.

Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.

*Run **make qemu**. You should see the environments switch back and forth between each other five times before terminating, like below.*

*Test also with several CPUS: **make qemu CPUS=2**.*

```
...
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

After the yield programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

```
// Choose a user environment to run and run it.
```

```
/*
```

The function is quite simple to implement. Getting the index of the next environment in the envs array to the currently running environment with the macro ENVX(eid), we use that very index in the for loop to check for the status of the environment. If the status is ENV_RUNNABLE, we break from the for loop and call function env_run through a boolean variable. If no environment is found in a runnable state we simply call sched_halt. If no env is found and the current envirnment is still running we let that environment run./

```
void

sched_yield(void)

{

//   struct Env *idle;


     // Implement simple round-robin scheduling.

     //

     // Search through 'envs' for an ENV_RUNNABLE environment in

     // circular fashion starting just after the env this CPU was

     // last running.  Switch to the first such environment found.

     //

     // If no envs are runnable, but the environment previously

     // running on this CPU is still ENV_RUNNING, it's okay to

     // choose that environment.

     //
```

```
	// Never choose an environment that's currently running on
	// another CPU (env_status == ENV_RUNNING). If there are
	// no runnable environments, simply drop through to the code
	// below to halt the cpu.

	// LAB 4: Your code here.

	uint32_t begin = curenv ? ENVX(curenv -> env_id) + 1 : 0;
	uint32_t index = 0;
	bool found = false;

	for (int i = 0; i < NENV; i++)
	{
		index = (begin + i) % NENV;
		if (envs[index].env_status == ENV_RUNNABLE)
		 {
			found = true;
			break;
		 }
	}

	if (found)
	{
		env_run (&envs [index]);
	} else if (curenv && curenv -> env_status == ENV_RUNNING)
	{
		env_run (curenv);
	} else
	{
	// sched_halt never returns
	sched_halt();
	}
}
```

In kern/init.c,

In the function mp_main ()

```
// Now that we have finished some basic setup, call sched_yield()
     // to start running processes on this CPU.  But make sure that
     // only one CPU can enter the scheduler at a time!
     //
     // Your code here:

     lock_kernel();
     // Remove this after you finish Exercise 6
     sched_yield ();
```

In function i386_init ()

```
#if defined(TEST)
     // Don't touch -- used by grading script!
     ENV_CREATE(TEST, ENV_TYPE_USER);
#else
     // Touch all you want.
     ENV_CREATE(user_yield, ENV_TYPE_USER);
     ENV_CREATE(user_yield, ENV_TYPE_USER);
     ENV_CREATE(user_yield, ENV_TYPE_USER);
```

IN the syscall function in kern/syscall.c

```
case SYS_yield:
                        sys_yield();
                        return 0;
                default:
                        return -E_INVAL;
```

Question

3.*In your implementation of env_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e, the argument to env_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the*

address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

The variable e is a pointer instance of the struct Env which is mapped above KERNBASE. Since all envirnment's page directories have same virtual memory mappings above KERNBASE as kern_pgdir, the kernel's page directory, the variable e can be referenced even after the page directory in CR3 register is changed.

4.Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

It happens in 2 steps. The first step is that _alltraps pushes into the kernel stack the registers that the hardware doesn't push automatically (such as the general purpose registers), then inside trap, we see the following code:

```
// Copy trap frame (which is currently on the stack)
// into 'curenv->env_tf', so that running the environment
// will restart at the trap point.
curenv->env_tf = *tf;
```

This basically copies over all the registers into the environment's structure so it can be later restored. This is important because when we switch environemnts, we don't want the new environment to override the registers which were used and relied by the old environment. This way we can have full isolation between the environments. The way an environment's registers are restored is happening inside env_pop_tf which is called by env_run whenever we choose to run an environment. env_pop_tf. The way this function works is that it first switches the stack pointer to point to the trap frame, and then manually pops all the registers that were pushed into the trap frame, and executes the iret instruction which pops the values that were pushed by the hardware and also returns the execution to the user process.

System Calls for Environment Creation

Although your kernel is now capable of running and switching between multiple user-level environments, it is still limited to running environments that the kernel initially set up. You will now implement the necessary JOS system calls to allow user environments to create and start other new user environments.

Unix provides the fork() system call as its process creation primitive. Unix fork() copies the entire address space of calling process (the parent) to create a new process (the child). The only differences between the two observable from user space are their process IDs and parent process IDs (as returned by getpid and getppid). In the parent, fork() returns the child's process ID, while in the child, fork() returns 0. By default, each process gets its own private address space, and neither process's modifications to memory are visible to the other.

You will provide a different, more primitive set of JOS system calls for creating

new user-mode environments. With these system calls you will be able to implement a Unix-like fork() entirely in user space, in addition to other styles of environment creation. The new system calls you will write for JOS are as follows:

sys_exofork:
This system call creates a new environment with an almost blank slate: nothing is mapped in the user portion of its address space, and it is not runnable. The new environment will have the same register state as the parent environment at the time of the sys_exofork call. In the parent, sys_exofork will return the envid_t of the newly created environment (or a negative error code if the environment allocation failed). In the child, however, it will return 0. (Since the child starts out marked as not runnable, sys_exofork will not actually return in the child until the parent has explicitly allowed this by marking the child runnable using....)
sys_env_set_status:
Sets the status of a specified environment to ENV_RUNNABLE or ENV_NOT_RUNNABLE. This system call is typically used to mark a new environment ready to run, once its address space and register state has been fully initialized.
sys_page_alloc:
Allocates a page of physical memory and maps it at a given virtual address in a given environment's address space.
sys_page_map:
Copy a page mapping (not the contents of a page!) from one environment's address space to another, leaving a memory sharing arrangement in place so that the new and the old mappings both refer to the same page of physical memory.
sys_page_unmap:
Unmap a page mapped at a given virtual address in a given environment.

For all of the system calls above that accept environment IDs, the JOS kernel supports the convention that a value of 0 means "the current environment." This convention is implemented by envid2env() in kern/env.c.

We have provided a very primitive implementation of a Unix-like fork() in the test program user/dumbfork.c. This test program uses the above system calls to create and run a child environment with a copy of its own address space. The two environments then switch back and forth using sys_yield as in the previous exercise. The parent exits after 10 iterations, whereas the child exits after 20.

Exercise 7. *Implement the system calls described above in kern/syscall.c and make sure syscall() calls them. You will need to use various functions in kern/pmap.c and kern/env.c, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.*

```
// Allocate a new environment.

// Returns envid of new environment, or < 0 on error.  Errors are:

//   -E_NO_FREE_ENV if no free environment is available.

//   -E_NO_MEM on memory exhaustion.
```

```
/*
Creating an instance of struct Env we pass it as a parameter to env_alloc method
which will create a new environment and store it in that very instacne of struct
Env. We then, set the env_status env_tf fields of the struct Env as has been
instructed and then set the register eax to 0 so as to make the exofork method
return 0 when called from child.
*/
        static envid_t
sys_exofork(void)
{
        // Create the new environment with env_alloc(), from kern/env.c.
        // It should be left as env_alloc created it, except that
        // status is set to ENV_NOT_RUNNABLE, and the register set is copied
        // from the current environment -- but tweaked so sys_exofork
        // will appear to return 0.

        // LAB 4: Your code here.

        struct Env* new_env = NULL;
        int a = env_alloc (&new_env, curenv -> env_id);
        if (a < 0)
                return a;

        new_env -> env_status = ENV_NOT_RUNNABLE;
        new_env -> env_tf = curenv -> env_tf;
        new_env -> env_tf.tf_regs.reg_eax = 0;

        return new_env -> env_id;

        //panic("sys_exofork not implemented");
}

// Set envid's env_status to status, which must be ENV_RUNNABLE
```

```
// or ENV_NOT_RUNNABLE.
//
// Returns 0 on success, < 0 on error.  Errors are:
//   -E_BAD_ENV if environment envid doesn't currently exist,
//        or the caller doesn't have permission to change envid.
//   -E_INVAL if status is not a valid status for an environment.
/*
Using the envid2env function, we get the struct Env corresponding to the envid provided while checking if the environment has the appropriate permissions and that the envid's environment is actually present. Finally, we set the correspinding env_status to status while status is one from the two provided in the description of the function.
*/
        static int
sys_env_set_status(envid_t envid, int status)
{
        // Hint: Use the 'envid2env' function from kern/env.c to translate an
        // envid to a struct Env.
        // You should set envid2env's third argument to 1, which will
        // check whether the current environment has permission to set
        // envid's status.

        // LAB 4: Your code here.

        struct Env* new_env = NULL;
        int a = envid2env (envid, &new_env, 1);

        if (a < 0)
                return a;

        if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
                -E_INVAL;

        new_env -> env_status = status;
```

```
        return 0;
        //panic("sys_env_set_status not implemented");
}


// Allocate a page of memory and map it at 'va' with permission
// 'perm' in the address space of 'envid'.
// The page's contents are set to 0.
// If a page is already mapped at 'va', that page is unmapped as a
// side effect.
//
// perm -- PTE_U | PTE_P must be set, PTE_AVAIL | PTE_W may or may not be set,
//         but no other bits may be set.  See PTE_SYSCALL in inc/mmu.h.
//
// Return 0 on success, < 0 on error.  Errors are:
//   -E_BAD_ENV if environment envid doesn't currently exist,
//        or the caller doesn't have permission to change envid.
//   -E_INVAL if va >= UTOP, or va is not page-aligned.
//   -E_INVAL if perm is inappropriate (see above).
//   -E_NO_MEM if there's no memory to allocate the new page,
/        or to allocate any necessary page tables.
/*
Using the envid2env function, we get the struct Env corresponding to the envid
provided while checking if the environment has the appropriate permissions and
that the envid's environment is actually present. We then perform checks for the
parameters provided as has been asked and then call the page_alloc function which
will allocate a physical page and return its corresponding struct Env.  That very
instance of struct Env is used by the page_insert method to map the allocated page
to the va provided through the parameters with the permissions as stated. If
necessary page_insert, will create a new page table for the page mapped at va and
insert it into the env's page directory. IT will also remove any existing page at
the va and invalidate the TLB.
*/
        static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
```

```
// Hint: This function is a wrapper around page_alloc() and
//   page_insert() from kern/pmap.c.
//   Most of the new code you write should be to check the
//   parameters for correctness.
//   If page_insert() fails, remember to free the page you
//   allocated!

// LAB 4: Your code here.

struct Env* new_env = NULL;
int a = envid2env (envid, &new_env, 1);

if (a < 0)
        return a;
if ((uintptr_t)va >= UTOP || (uintptr_t) va % PGSIZE != 0)
        return -E_INVAL;

if ((perm & ~PTE_SYSCALL) != 0)
        return -E_INVAL;

struct PageInfo* np = page_alloc(ALLOC_ZERO);
if (!np)
        return -E_NO_MEM;

a = page_insert (new_env -> env_pgdir, np, va, perm | PTE_SYSCALL);
if (a < 0)
        page_free (np);
return a;

return 0;

//panic("sys_page_alloc not implemented");
```

```
}
// Map the page of memory at 'srcva' in srcenvid's address space
// at 'dstva' in dstenvid's address space with permission 'perm'.
// Perm has the same restrictions as in sys_page_alloc, except
// that it also must not grant write access to a read-only
// page.
//
// Return 0 on success, < 0 on error.  Errors are:
//   -E_BAD_ENV if srcenvid and/or dstenvid doesn't currently exist,
//        or the caller doesn't have permission to change one of them.
//   -E_INVAL if srcva >= UTOP or srcva is not page-aligned,
//        or dstva >= UTOP or dstva is not page-aligned.
//   -E_INVAL is srcva is not mapped in srcenvid's address space.
//   -E_INVAL if perm is inappropriate (see sys_page_alloc).
//   -E_INVAL if (perm & PTE_W), but srcva is read-only in srcenvid's
//        address space.
//   -E_NO_MEM if there's no memory to allocate any necessary page tables.
        static int
/*
Initially, we perform the various tests as has been stated in the description. Then, using the envid2env function, we get the struct Env's corresponding to the envids provided while checking if the environments has the appropriate permissions and that the envid's environments are actually present. Checks are performed for both source and destination environments. We then call the page lookup method which will return the page i.e. strurct Env of the page mapped at the the srcva provided as parameter. We then check if the destination environment has requested write permission for the page and if the permissions of the source environement for the page read-only. If this the case, we return -E_INVAL. If not, we call the page_insert method which will establish the mapping between the page whose struct Env has been returned by page_lookup and the va with which mapping is to be established, namely dstva, with the given permissions.
*/
sys_page_map(envid_t srcenvid, void *srcva,
                envid_t dstenvid, void *dstva, int perm)
{
        // Hint: This function is a wrapper around page_lookup() and
```

```
	//   page_insert() from kern/pmap.c.
	//   Again, most of the new code you write should be to check the
	//   parameters for correctness.
	//   Use the third argument to page_lookup() to
	//   check the current permissions on the page.

	// LAB 4: Your code here.

	struct Env* source_env = NULL;
	struct Env* dest_env = NULL;
	uint32_t a = 0;
	pte_t* pte;

	if ((perm & ~PTE_SYSCALL) != 0)
		return -E_INVAL;

	uintptr_t sa = (uintptr_t) srcva;
	uintptr_t da = (uintptr_t) dstva;

	if (sa >= UTOP || (sa % PGSIZE) != 0 || da >= UTOP || (da % PGSIZE) != 0)
		return -E_INVAL;
	a = envid2env (srcenvid, &source_env, 1);
	if (a < 0)
		return a;
	a = envid2env (dstenvid, &dest_env, 1);
	if (a < 0)
		return a;
struct PageInfo* np = page_lookup (source_env -> env_pgdir, srcva, &pte);
	if (!np)
		return -E_INVAL;

	if ((perm & PTE_W) && !(*pte & PTE_W))
```

```
                return -E_INVAL;

        a = page_insert (dest_env -> env_pgdir, np, dstva, perm);
        if (a < 0)
                return a;

        return 0;

        //panic("sys_page_map not implemented");
}
// Unmap the page of memory at 'va' in the address space of 'envid'.
// If no page is mapped, the function silently succeeds.
//
// Return 0 on success, < 0 on error.  Errors are:
//   -E_BAD_ENV if environment envid doesn't currently exist,
//        or the caller doesn't have permission to change envid.
//   -E_INVAL if va >= UTOP, or va is not page-aligned.
/*
Initially, we perform the various tests as has been stated in the description.
Then, using the envid2env function, we get the struct Env corresponding to the
envid provided while checking if the environment has the appropriate permissions
and that the envid's environment are actually present. We then call the
page_remove method which shall unmap the physical page mapped to the evirtual
address va in the environment.
*/
        static int
sys_page_unmap(envid_t envid, void *va)
{
        // Hint: This function is a wrapper around page_remove().

        // LAB 4: Your code here.

        struct Env* e_env = NULL;
        uint32_t a = 0;
```

```
        if ((uintptr_t) va % PGSIZE != 0 || (uintptr_t) va >= UTOP)
                return -E_INVAL;

        a = envid2env (envid, &e_env, 1);
        if (a < 0)
                return a;

        page_remove (e_env -> env_pgdir, va);

        return 0;



        //panic("sys_page_unmap not implemented");
}
/*
Finally, we add the following to the switch of the syscall method in kern/syscall.c
*/
case SYS_exofork:
                        return sys_exofork();
                case SYS_env_set_status:
                        return sys_env_set_status((envid_t) a1, (int) a2);
                case SYS_page_alloc:
                        return sys_page_alloc((envid_t) a1, (void *)a2, (int) a3);
                case SYS_page_map:
                          return sys_page_map((envid_t) a1, (void *) a2, (envid_t) a3, (void *) a4, (int) a5);
                case SYS_page_unmap:
                        return sys_page_unmap((envid_t) a1, (void *) a2);
```

Part B: Copy-on-Write Fork

As mentioned earlier, Unix provides the fork() system call as its primary process creation primitive. The fork() system call copies the address space of the calling process (the parent) to create a new process (the child).

xv6 Unix implements fork() by copying all data from the parent's pages into new pages allocated for the child. This is essentially the same approach that dumbfork() takes. The copying of the parent's address space into the child is the most expensive part of the fork() operation.

However, a call to fork() is frequently followed almost immediately by a call to exec() in the child process, which replaces the child's memory with a new program. This is what the the shell typically does, for example. In this case, the time spent copying the parent's address space is largely wasted, because the child process will use very little of its memory before calling exec().

For this reason, later versions of Unix took advantage of virtual memory hardware to allow the parent and child to share the memory mapped into their respective address spaces until one of the processes actually modifies it. This technique is known as copy-on-write. To do this, on fork() the kernel would copy the address space mappings from the parent to the child instead of the contents of the mapped pages, and at the same time mark the now-shared pages read-only. When one of the two processes tries to write to one of these shared pages, the process takes a page fault. At this point, the Unix kernel realizes that the page was really a "virtual" or "copy-on-write" copy, and so it makes a new, private, writable copy of the page for the faulting process. In this way, the contents of individual pages aren't actually copied until they are actually written to. This optimization makes a fork() followed by an exec() in the child much cheaper: the child will probably only need to copy one page (the current page of its stack) before it calls exec().

In the next piece of this lab, you will implement a "proper" Unix-like fork() with copy-on-write, as a user space library routine. Implementing fork() and copy-on-write support in user space has the benefit that the kernel remains much simpler and thus more likely to be correct. It also lets individual user-mode programs define their own semantics for fork(). A program that wants a slightly different implementation (for example, the expensive always-copy version like dumbfork(), or one in which the parent and child actually share memory afterward) can easily provide its own.

User-level page fault handling

A user-level copy-on-write fork() needs to know about page faults on write-protected pages, so that's what you'll implement first. Copy-on-write is only one of many possible uses for user-level page fault handling.

It's common to set up an address space so that page faults indicate when some action needs to take place. For example, most Unix kernels initially map only a single page in a new process's stack region, and allocate and map additional stack pages later "on demand" as the process's stack consumption increases and causes page faults on stack addresses that are not yet mapped. A typical Unix kernel must keep track of what action to take when a page fault occurs in each region of a

process's space. For example, a fault in the stack region will typically allocate and map new page of physical memory. A fault in the program's BSS region will typically allocate a new page, fill it with zeroes, and map it. In systems with demand-paged executables, a fault in the text region will read the corresponding page of the binary off of disk and then map it.

This is a lot of information for the kernel to keep track of. Instead of taking the traditional Unix approach, you will decide what to do about each page fault in user space, where bugs are less damaging. This design has the added benefit of allowing programs great flexibility in defining their memory regions; you'll use user-level page fault handling later for mapping and accessing files on a disk-based file system.

Setting the Page Fault Handler

In order to handle its own page faults, a user environment will need to register a page fault handler entrypoint with the JOS kernel. The user environment registers its page fault entrypoint via the new sys_env_set_pgfault_upcall system call. We have added a new member to the Env structure, env_pgfault_upcall, to record this information.

Exercise 8. *Implement the sys_env_set_pgfault_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.*

```
// Set the page fault upcall for 'envid' by modifying the corresponding struct

// Env's 'env_pgfault_upcall' field.  When 'envid' causes a page fault, the

// kernel will push a fault record onto the exception stack, then branch to 'func'.

// Returns 0 on success, < 0 on error.  Errors are:

//   -E_BAD_ENV if environment envid doesn't currently exist,

//        or the caller doesn't have permission to change envid..

/*

Being given a envid, we again use the envid2env function to obtain the corresponding struct Env. The permissions checking asked for in the question will be undertaken by the function envid2env by setting the third parameter to be 1. Getting the corresponding struct Env for the given envid, we set the env_pgfault_upcall field to be equal to func which is the entry point for the envrionment’s page fault handler.

*/

        static int

sys_env_set_pgfault_upcall(envid_t envid, void *func)

{
```

```
        // LAB 4: Your code here.

        struct Env* new_env = NULL;
        int a = envid2env (envid, &new_env, 1);
        if (a < 0)
                return a;

        new_env -> env_pgfault_upcall = func;
        return 0;
        //panic("sys_env_set_pgfault_upcall not implemented");
}
```

Normal and Exception Stacks in User Environments

During normal execution, a user environment in JOS will run on the normal user stack: its ESP register starts out pointing at USTACKTOP, and the stack data it pushes resides on the page between USTACKTOP-PGSIZE and USTACKTOP-1 inclusive. When a page fault occurs in user mode, however, the kernel will restart the user environment running a designated user-level page fault handler on a different stack, namely the user exception stack. In essence, we will make the JOS kernel implement automatic "stack switching" on behalf of the user environment, in much the same way that the x86 processor already implements stack switching on behalf of JOS when transferring from user mode to kernel mode!

The JOS user exception stack is also one page in size, and its top is defined to be at virtual address UXSTACKTOP, so the valid bytes of the user exception stack are from UXSTACKTOP-PGSIZE through UXSTACKTOP-1 inclusive. While running on this exception stack, the user-level page fault handler can use JOS's regular system calls to map new pages or adjust mappings so as to fix whatever problem originally caused the page fault. Then the user-level page fault handler returns, via an assembly language stub, to the faulting code on the original stack.

Each user environment that wants to support user-level page fault handling will need to allocate memory for its own exception stack, using the sys_page_alloc() system call

Invoking the User Page Fault Handler

You will now need to change the page fault handling code in kern/trap.c to handle page faults from user mode as follows. We will call the state of the user environment at the time of the fault the trap-time state.

If there is no page fault handler registered, the JOS kernel destroys the user environment with a message as before. Otherwise, the kernel sets up a trap frame on the exception stack that looks like a struct UTrapframe from inc/trap.h:

```
                    <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax       start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

The kernel then arranges for the user environment to resume execution with the page fault handler running on the exception stack with this stack frame; you must figure out how to make this happen. The fault_va is the virtual address that caused the page fault.

If the user environment is already running on the user exception stack when an exception occurs, then the page fault handler itself has faulted. In this case, you should start the new stack frame just under the current tf→tf_esp rather than at UXSTACKTOP. You should first push an empty 32-bit word, then a struct UTrapframe.

To test whether tf→tf_esp is already on the user exception stack, check whether it is in the range between UXSTACKTOP-PGSIZE and UXSTACKTOP-1, inclusive.

Exercise 9. *Implement the code in page_fault_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)*

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



        // Call the environment's page fault upcall, if one exists.  Set up a
        // page fault stack frame on the user exception stack (below
        // UXSTACKTOP), then branch to curenv->env_pgfault_upcall.
        //
        // The page fault upcall might cause another page fault, in which case
        // we branch to the page fault upcall recursively, pushing another
```

```
	// page fault stack frame on top of the user exception stack.
	//
	// It is convenient for our code which returns from a page fault
	// (lib/pfentry.S) to have one word of scratch space at the top of the
	// trap-time stack; it allows us to more easily restore the eip/esp. In
	// the non-recursive case, we don't have to worry about this because
	// the top of the regular user stack is free.  In the recursive case,
	// this means we have to leave an extra word between the current top of
	// the exception stack and the new stack frame because the exception
	// stack _is_ the trap-time stack.
	//
	// If there's no page fault upcall, the environment didn't allocate a
	// page for its exception stack or can't write to it, or the exception
	// stack overflows, then destroy the environment that caused the fault.
	// Note that the grade script assumes you will first check for the page
	// fault upcall and print the "user fault va" message below if there is
	// none.  The remaining three checks can be combined into a single test.
	//
	// Hints:
	//   user_mem_assert() and env_run() are useful here.
	//   To change what the user environment runs, modify 'curenv->env_tf'
	//   (the 'tf' variable points at 'curenv->env_tf').


	// LAB 4: Your code here.
/*
```

With the task being to call the user environment’s page fault upcall, we check if one ha been defined through the if statement. If not, we will skip the code in the if statement and destroy the environment, printing out envid and faulting va before it.

In case it does exist, we initially check if the call to page fault upcall is a recursive one due to the page fault handler page faulting or its a fresh call through the if statement as in case of recursive call, the esp i.e. stack pointer of the environment will already be on the user exception stack. In that case, we push a 32-bit empty word on to the exception stack, thus making the exception stack of the recursive call to begin 4 bytes below the current value of esp. As to why we do that is being described below. Next, we set up a trap frame on the

```
exception stack that looks like a struct Utrapframe from inc/trap.h after
verifying that the environment can write on to the exception stack. We then set
the eip of the user environment to point to the user page fault handler and the
esp to point to the exception stack to ensure that the user environment resumes
execution with the page fault handler running on the exception stack with the
stack frame we created.

Finally, we call env_run to begin execution of the user environment.

*/

if (curenv -> env_pgfault_upcall)

        {

                uintptr_t stack_top = UXSTACKTOP;

                uintptr_t stack_bottom = UXSTACKTOP - PGSIZE;


                if (tf -> tf_esp < UXSTACKTOP && tf -> tf_esp >= stack_bottom)

                        stack_top = tf -> tf_esp - 4;


                  struct UTrapframe* utf_addr = (struct UTrapframe *) (stack_top -
sizeof (struct UTrapframe));


                     user_mem_assert (curenv, (void *) utf_addr, sizeof (struct
UTrapframe), PTE_U | PTE_W | PTE_P);


                utf_addr -> utf_fault_va = fault_va;

                utf_addr -> utf_err = tf -> tf_err;

                utf_addr -> utf_regs = tf -> tf_regs;

                utf_addr -> utf_eip = tf -> tf_eip;

                utf_addr -> utf_eflags = tf -> tf_eflags;

                utf_addr -> utf_esp = tf -> tf_esp;


                                curenv -> env_tf.tf_eip = (uintptr_t)(curenv ->
env_pgfault_upcall);

                curenv -> env_tf.tf_esp = (uintptr_t) utf_addr;


                env_run (curenv);

        }
```

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

If we run out of space in the user exception stack, the kernel will destroy the environment (if we don't take this precution, then the kernel itself will page fault, which will halt the system).

User-mode Page Fault Entry point

Next, you need to implement the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction. This assembly routine is the handler that will be registered with the kernel using sys_env_set_pgfault_upcall().

Exercise 10. *Implement the _pgfault_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.*

```
 // Now the C page fault handler has returned and you must return
     // to the trap time state.
     // Push trap-time %eip onto the trap-time stack.
     //
     // Explanation:
     //   We must prepare the trap-time stack for our eventual return to
     //   re-execute the instruction that faulted.
     //   Unfortunately, we can't return directly from the exception stack:
     //   We can't call 'jmp', since that requires that we load the address
     //   into a register, and all registers must have their trap-time
     //   values after the return.
     //   We can't call 'ret' from the exception stack either, since if we
     //   did, %esp would have the wrong value.
     //   So instead, we push the trap-time %eip onto the *trap-time* stack!
     //   Below we'll switch to that stack and call 'ret', which will
```

```
//   restore %eip to its pre-fault value.
//
//   In the case of a recursive fault on the exception stack,
//   note that the word we're pushing now will fit in the
//   blank word that the kernel reserved for us.
//
// Throughout the remaining code, think carefully about what
// registers are available for intermediate calculations.  You
// may find that you have to rearrange your code in non-obvious
// ways as registers become unavailable as scratch space.
//
// LAB 4: Your code here.
movl 40(%esp), %ebx //grab trap-time eip
movl 48(%esp), %edx // grab trap-time esp
 subl $4, %edx        //Reserve slot for eip to be pushed on to the stack of either the environement that has faulted, if call to this handler is not recursive, or this handler itself, if call is recursive.
movl %edx, 48(%esp) //adjust trap-time esp so ret will pop the eip
  mov %ebx, (%edx)     //push eip on to the trap time stack, in case of recursive call, or on the stack of the faulting environment in case of non-recursive call.
 addl $8, %esp       //skip the fault_va and error code since we have no use of them


// Restore the trap-time registers.  After you do this, you
// can no longer modify any general-purpose registers.
// LAB 4: Your code here.
popal
// Restore eflags from the stack.  After you do this, you can
// no longer use arithmetic operations or anything else that
// modifies eflags.
// LAB 4: Your code here.
addl $4, %esp       //We skip the trap time ei since it is no use to us.
popfl
```

// Switch back to the adjusted trap-time stack.

// LAB 4: Your code here.

popl %esp //restore the value of trap-time esp i.e. esp of either the faulting envornment or the handler itself (if the call is recursive)

// Return to re-execute the instruction that faulted.

// LAB 4: Your code here.

ret //return to either the faulting environment or the handler in case of recursive call.

why do we need to push a an empty 32-bit word between the stack frames?

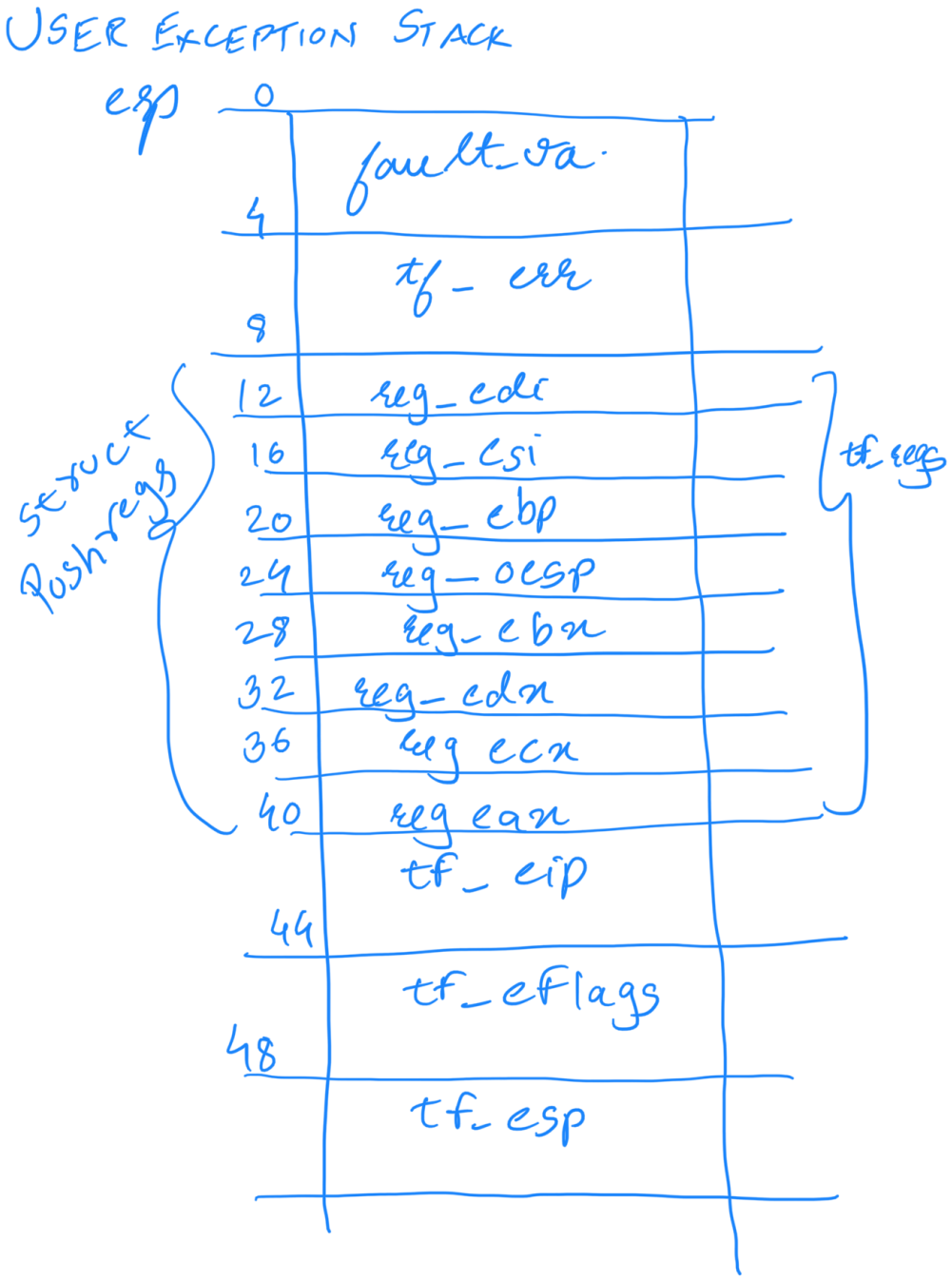
Let's first think about how we return the execution back to where it page faulted: We need to restore all general purpose registers, the eflags, the stack pointer (%esp), and the instruction pointer (%eip).

popl %esp is the instruction that switches to the destination stack (restores its previus value before the page fault). The value of %esp will be the user environment that faulted if this the only time the handler has been called. In the event that the handler itself has faulted, it will be called recursively ad thus will return to the same function i.e. the handler function will return to the handler function and therein lies out problem.
How do we restore the %eip? This can't be done directly cause of the reason mentioned in the hints section. instead, we will use the stack to reload it. For example, if we have %eip on the stack and we execute the ret instruction, the processor will load the %eip with whatever is on the stack (immediately next to the value of %esp at offset of 0 and will pop off that value). Since we ret after we already restored the stack, we need our %eip to exist right next to where the restored %esp is pointing towards!. in other words, it needs to be in the 4 bytes underneath the original target %esp. Thus, in case of handler being called only once, the %ei needs to be on the stack of the user environment that faulted. In case of recursive call however, it must be on the exception stack. When the handler faults, the user exception stack of the very handler will be passed as a parameter i.e. struct Trapframe to its recursive call through the page_fault_handler function in kern/trap.c. This trap-time state of the handler will be stored over the user exception stack only below the stack of the handler that faulted. Now, as mentioned for this recursive call of handler function to return, it must push the value of %eip on to the stack of the stack of the function that called the handler, in this the case the handler itself as the call is recursive. And therefore, we push an empty word onto the exception stack so that we can push this %eip value on the stack without corrupting the user exception stack.*

In order words, for the plan to work, we MUST guarantee that we'll be able to store something 4 bytes underneath the target %esp. Usually this is not an issue if our target esp is not in the exception stack, but if it is, then our trap frame

(from which we are returning) is exactly below the destination stack (to which we are returning). Therefore, if we don't allocate the 4 bytes as described, writing underneath the target %esp will corrupt the top of our trap frame!

To understand the above assembly, please refer the image below..



The image above is of the struct Utrapframe as will be pushed on to the stack by the page_fault_handler function in kern/trap.c we implemented in question 9. The offsets have been added for reference.
The first assembly instruction will store the trap time of either the user environment that faulted or the handler itself in case that the latter has

faulted.
The next instruction will store the trap-time %esp.
We then subtract 4 from the value of trap-time esp, store the new value of %esp on to the exception stack and store the value of %eip at the memory location pointed to by the trap-time %esp i.e. we store the %eip value either over the stack of the user environment that faulted or on the user exception stack itself in case call to handler is recursive. In latter case, we store the %eip into the 4 byte empty space we left on the exception stack before calling the handler recursively.
We then add 8 to the value %esp since to skip fault_va and error code and restore the values of trap-time registers i.e. regs in struct Pushregs.
Adding further 4 to %esp to skip the %eip value which is no use, we restore the e %eflags registers. Finally we replace the value in %esp with the one n stack i.e. we put trap-time esp into the %esp register and call ret function which will return from the handler function after setting %eip to the address that is held in the next 4 bytes from the address of %esp.

Finally, you need to implement the C user library side of the user-level page fault handling mechanism.

Exercise 11. *Finish set_pgfault_handler() in lib/pgfault.c.*

```
//
// Set the page fault handler function.
// If there isn't one yet, _pgfault_handler will be 0.
// The first time we register a handler, we need to
// allocate an exception stack (one page of memory with its top
// at UXSTACKTOP), and tell the kernel to call the assembly-language
// _pgfault_upcall routine when a page fault occurs.
/*
      Simple function. Call sys_page_alloc to allocate page for exception stack. Envid i.e. first parameter is 0 as it is first time we are registering the handler. Later use sys_env_set_pgfault_upcall to register the handler entry point with the kernel. i.e. after registering the handler entry point, user environments on page fau;ts will be directed to the lib/pfentry.S function which in turn will call this C page fault handler function.
*/
        void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
        int r;
  if (_pgfault_handler == 0) {
                // First time through!
                // LAB 4: Your code here.
                 int a = sys_page_alloc (0, (void *) (UXSTACKTOP -PGSIZE), PTE_U | PTE_W);
                if (a < 0)
                        panic ("sys_page_alloc Failed. %e", a);
                //panic("set_pgfault_handler not implemented");
        }
```

```
sys_env_set_pgfault_upcall (0,  _pgfault_upcall);

// Save handler pointer for assembly to call.
_pgfault_handler = handler;
```

OVERVIEW OF USER-MODE PAGE FAULT HANDLING

User environment or any user program that wishes to implement user-mode page fault handling will call the function set_pgfault_handler () defined in lib/pgfault.c within its main (umain) function. The set_pgfault_handler () function upon being called the first time will allocate an exception stack of 1 page for the environment at UXSTACKTOP and register the page fault entry point as _pgfault_upcall defined in lib/pfentry.S through the sys_set_env_pgfault_upcall system call. Finally, it will set the global function pointer, _pgfault_handler, to be equal to the actual page fault handler function which has been passed to set_pgfault_handler () function through parameter as a function pointer. The actual ccustom page fault hander for the purpose of copy on write is pgfault in lib/fork.c. _pgfault_handler will then be used by the assembly code in pfentry.S to call the page fault handler function. Thus, what we are doing it is passing the address of pfentry.S to the kernel which in turn will evoke the actual page fault handler.

When a user envelopment page faults, its execution is suspended and control passes to the kernel which upon knowing the fault is a page fault through trap_dispatch () method calls the page_fault_handler function. The page_fault_handler function initially checks if the environment has its pgfault_upcall field (curenv->env_pgfault_upcall)in struct Env set and that it has allocated a user exception stack. The kernel then copies all the values of environment's trap frame into the exception stack in the form of Utrapframe. Finally, it makes amends necessary for the user environment to resume execution of the pagefault handler function by setting the %eip of the user environment through its trapframe to the pgfault_upcall which is the entrypoint for handler function. The stack pointer is made to point to the user exception stack. The execution is then in the assembly code, which calls our custom page fault handler and then later cleans up after itself.

why do user/faultalloc and user/faultallocbad behave differently?

faultalloc

faultalloc calls cprintf("%s\n", (char)0xDeadBeef); and this causes the following:*

- *the first character of the string located in 0xDeadBeef is attempted to be read by the environment.*

- *a page fault occurs becase 0xDeadBeef is unmapped*
- *the page fault handler that we have registered runs. During the run it maps the missing page and also populates the string (using the snprintf function)*
- *execution returns back from where it faulted, and cprintf can resume printing to the screen because the memory is now mapped and even contains some data.*

faultallocbad

faultallocbad calls sys_cputs((char)0xDEADBEEF, 4); and this causes the following:*

- *the memory in 0xDEADBEEF is not accessed by the environment, but is instead passed directly to the kernel through a system call. (notice that there was no page fault here and thus the handler has never ran).*
- *During the handling of this system call, the kernel detects that the memory is unmapped and kills the environment. Therefore, nothing is printed.*

Implementing Copy-on-Write Fork

You now have the kernel facilities to implement copy-on-write fork() entirely in user space.

We have provided a skeleton for your fork() in lib/fork.c. Like dumbfork(), fork() should create a new environment, then scan through the parent environment's entire address space and set up corresponding page mappings in the child. The key difference is that, while dumbfork() copied pages, fork() will initially only copy page mappings. fork() will copy each page only when one of the environments tries to write it.

The basic control flow for fork() is as follows:

1. *The parent installs pgfault() as the C-level page fault handler, using the set_pgfault_handler() function you implemented above.*
2. *The parent calls sys_exofork() to create a child environment.*
3. *For each writable or copy-on-write page in its address space below UTOP, the parent calls duppage, which should map the page copy-on-write into the address space of the child and then remap the page copy-on-write in its own address space. [Note: The ordering here (i.e., marking a page as COW in the child before marking it in the parent) actually matters! Can you see why? Try to think of a specific case where reversing the order could cause trouble.] duppage sets both PTEs so that the page is not writeable, and to contain PTE_COW in the "avail" field to distinguish copy-on-write pages from genuine read-only pages.*

The exception stack is not remapped this way, however. Instead you need to allocate a fresh page in the child for the exception stack. Since the page fault handler will be doing the actual copying and the page fault handler runs on the exception stack, the exception stack cannot be made copy-on-write: who would copy it?

fork() also needs to handle pages that are present, but not writable or copy-on-write.

4. *The parent sets the user page fault entrypoint for the child to look like its own.*
5. *The child is now ready to run, so the parent marks it runnable.*

Each time one of the environments writes a copy-on-write page that it hasn't yet written, it will take a page fault. Here's the control flow for the user page fault handler:

1. *The kernel propagates the page fault to _pgfault_upcall, which calls fork()'s pgfault() handler.*
2. *pgfault() checks that the fault is a write (check for FEC_WR in the error code) and that the PTE for the page is marked PTE_COW. If not, panic.*
3. *pgfault() allocates a new page mapped at a temporary location and copies the contents of the faulting page into it. Then the fault handler maps the new page at the appropriate address with read/write permissions, in place of the old read-only mapping.*

The user-level lib/fork.c code must consult the environment's page tables for several of the operations above (e.g., that the PTE for a page is marked PTE_COW). The kernel maps the environment's page tables at UVPT exactly for this purpose. It uses a clever mapping trick to make it to make it easy to lookup PTEs for user code. lib/entry.S sets up uvpt and uvpd so that you can easily lookup page-table information in lib/fork.c.

ON closer examination for the trick, I found the following lines in the file inc/memlayout.h

```
/*
 * The page directory entry corresponding to the virtual address range
 * [UVPT, UVPT + PTSIZE) points to the page directory itself.  Thus, the page
 * directory is treated as a page table as well as a page directory.
 *
 * One result of treating the page directory as a page table is that all PTEs
 * can be accessed through a "virtual page table" at virtual address UVPT (to
 * which uvpt is set in lib/entry.S).  The PTE for page number N is stored in
 * uvpt[N].  (It's worth drawing a diagram of this!)
 *
 *     (UVPT + (UVPT >> PGSHIFT)), to
 * which uvpd is set in lib/entry.S.
 */
extern volatile pte_t uvpt[];     // VA of "virtual page table"
extern volatile pde_t uvpd[];     // VA of current page directory
```

Exercise 12. *Implement fork, duppage and pgfault in lib/fork.c.*

Test your code with the forktree program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

```
//
// Custom page fault handler - if faulting page is copy-on-write,
// map in our own private writable copy.
//
/*
      Simple function to implement. Getting the page table index for faulting address through offset obtained from the address and using it in uvpt which is the page table as an index, check if the error is caused by a write operation by checking for FEC_WR in the error code in the trap frame. We also check if the permissions for the page in which the faulting address resides is  PTE_COW> If neither is the case, we panic. Else, using the system call sys_page_alloc, we allocate a new page at PFTEMP, copy the contents of the faulting page to the new page and then map the new page at the address of the old page. Finally, we unmap the newly allocated page from PFTEMP.
*/
        static void
pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
        uint32_t err = utf->utf_err;
        int r;

        // Check that the faulting access was (1) a write, and (2) to a
        // copy-on-write page.  If not, panic.
        // Hint:
        //   Use the read-only page table mappings at uvpt
```

```
        //   (see <inc/memlayout.h>).

        // LAB 4: Your code here.

        pte_t pte = uvpt [(uintptr_t)addr >> PTXSHIFT];

        if (!(err & FEC_WR))
        {
                panic ("Page fault Not caused by a write. %x \n", err);
        } else if (!(pte & PTE_COW))
                panic ("Page fault Not caused by copy on write %x \n", err);

        // Allocate a new page, map it at a temporary location (PFTEMP),
        // copy the data from the old page to the new page, then move the new
        // page to the old page's address.
        // Hint:
        //   You should make three system calls.

        // LAB 4: Your code here.

        int perm = PTE_P | PTE_W | PTE_U;
        r = sys_page_alloc (0, (void *) PFTEMP, perm);
        if (r < 0)
                panic ("Sys_page_alloc failed %e", r);

        memmove (PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);

          if ((r = sys_page_map (0, (void *)PFTEMP, 0, ROUNDDOWN (addr, PGSIZE),
perm)) < 0)
                panic ("sys_page_map Failed. %e", r);


        if ((r = sys_page_unmap (0, (void*) PFTEMP)) < 0)
                panic ("sys_page_unmap Failed \n. %e", r);

        //panic("pgfault not implemented");

}

//
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
```

```
//
/*
Getting the address for the page provided, we check if the pte for for the page is
marked as PTE_W or PTE_COW i.e. if the page is writable or copy on write. If it
is, we use sys_page_map system call to map it into the address space of the
destination environment provided as a function parameter as copy on write. We then
set the page to be copy on write in the parent environment. If the page is not
writable or copy on write, we plainly copy the page mapping into the destination
environment with PTE_U and PTE_P.
*/
        static int
duppage(envid_t envid, unsigned pn)
{
        int r;

        // LAB 4: Your code here.
        void* address = (void*) (pn*PGSIZE);

        if ((uvpt [pn] & PTE_W) || (uvpt[pn] & PTE_COW))
        {
                if ((r = sys_page_map (0, address, envid, address, PTE_COW | PTE_P
| PTE_U)) < 0)
                        return r;

                 if ((r = sys_page_map (0, address, 0, address, PTE_COW | PTE_P |
PTE_U)) < 0)
                        return r;
        } else
        {
                if ((r = sys_page_map (0, address, envid, address, PTE_U | PTE_P))
< 0)
                        return r;
        }

        //        panic("duppage not implemented");
        return 0;
}

//
// User-level fork with copy-on-write.
// Set up our page fault handler appropriately.
// Create a child.
// Copy our address space and page fault handler setup to the child.
// Then mark the child as runnable and return.
//
// Returns: child's envid to the parent, 0 to the child, < 0 on error.
// It is also OK to panic on error.
//
// Hint:
```

```
//   Use uvpd, uvpt, and duppage.
//   Remember to fix "thisenv" in the child process.
//   Neither user exception stack should ever be marked copy-on-write,
//   so you must allocate a new page for the child's user exception stack.
//
/*
As described in lab writeup, we initially set the pgfault to the C Level page
fault handler and call sys_exofork to create the child. Since all address space
mappings of parent are to copied to the child using duppage function, we browse
through the page directory of the environment, skipping to the next entry if
present bit not set for the entry. In the event it is set, we enter the
corresponding page table and get the page number for each page in the page table.
We then implement further checks. First of which is if the page if of user
exception stack. Since its mappings are not to be copied we skip to the next entry
in the page table. If the page table entry for a page has its present bit set, we
call duppage to copy its mappings t the child.
Setting the child’s page fault handler entry point to be the _pgfault_upcall in
lib/pfentry.S. we allocated a page for user exception in the child through
sys_page_alloc, mark the child as runnable and return the child environment id.
In the event of the fork being called in the child, we set the thisenv pointer to
point to the child and return 0.
*/
        envid_t
fork(void)
{
        // LAB 4: Your code here.

        set_pgfault_handler(pgfault);

        envid_t child_envid ;
        child_envid = sys_exofork();
        if (child_envid < 0)
        {
                panic ("sys_exofork failed \n. %e", child_envid);
        } else if (child_envid == 0)
        {
                set_pgfault_handler(pgfault);
                thisenv = &envs [ENVX(sys_getenvid())];
                return child_envid;
        }

        bool is_below_UTOP = true;

        for (int  i = 0; is_below_UTOP && i < NPDENTRIES; i ++)
        {
                 if (!(uvpd [i] & PTE_P))     //execute the statement only if the
conndition is false.
                        continue;
```

```
                for (int j = 0; is_below_UTOP && j < NPTENTRIES; j ++)
                {
                        uint32_t page_number = i * NPTENTRIES + j;

                        if (page_number == (UXSTACKTOP - PGSIZE) >> PGSHIFT)
                        {
                                continue;
                        } else if (page_number >= (UTOP >> PGSHIFT))
                        {
                                is_below_UTOP = false;
                        } else if (uvpt [page_number] & PTE_P)
                        {
                                duppage (child_envid, page_number);
                        }
                }
        }

        extern void _pgfault_upcall ();

        sys_env_set_pgfault_upcall(child_envid, _pgfault_upcall);
        sys_page_alloc (child_envid, (void*) (UXSTACKTOP - PGSIZE), PTE_SYSCALL);

        sys_env_set_status (child_envid, ENV_RUNNABLE);

        return child_envid;
}
```

Part C: Preemptive Multitasking and Inter-Process communication (IPC)

In the final part of lab 4 you will modify the kernel to preempt uncooperative environments and to allow environments to pass messages to each other explicitly.

Clock Interrupts and Preemption

Run the user/spin test program. This test program forks off a child environment, which simply spins forever in a tight loop once it receives control of the CPU. Neither the parent environment nor the kernel ever regains the CPU. This is obviously not an ideal situation in terms of protecting the system from bugs or malicious code in user-mode environments, because any user-mode environment can bring the whole system to a halt simply by getting into an infinite loop and never giving back the CPU. In order to allow the kernel to preempt a running environment, forcefully retaking control of the CPU from it, we must extend the JOS kernel to support external hardware interrupts from the clock hardware.

Interrupt discipline

External interrupts (i.e., device interrupts) are referred to as IRQs. There are 16 possible IRQs, numbered 0 through 15. The mapping from IRQ number to IDT entry is not fixed. pic_init in picirq.c maps IRQs 0-15 to IDT entries IRQ_OFFSET through IRQ_OFFSET+15.

In inc/trap.h, IRQ_OFFSET is defined to be decimal 32. Thus the IDT entries 32-47 correspond to the IRQs 0-15. For example, the clock interrupt is IRQ 0. Thus, IDT[IRQ_OFFSET+0] (i.e., IDT[32]) contains the address of the clock's interrupt handler routine in the kernel. This IRQ_OFFSET is chosen so that the device interrupts do not overlap with the processor exceptions, which could obviously cause confusion.

In JOS, we make a key simplification compared to xv6 Unix. External device interrupts are always disabled when in the kernel (and, like xv6, enabled when in user space). External interrupts are controlled by the FL_IF flag bit of the %eflags register (see inc/mmu.h). When this bit is set, external interrupts are enabled. While the bit can be modified in several ways, because of our simplification, we will handle it solely through the process of saving and restoring %eflags register as we enter and leave user mode.

You will have to ensure that the FL_IF flag is set in user environments when they run so that when an interrupt arrives, it gets passed through to the processor and handled by your interrupt code. Otherwise, interrupts are masked, or ignored until interrupts are re-enabled. We masked interrupts with the very first instruction of the bootloader, and so far we have never gotten around to re-enabling them.

***Exercise 13.** Modify kern/trapentry.S and kern/trap.c to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in env_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.*

Also uncomment the sti instruction in sched_halt() so that idle CPUs unmask interrupts.

The processor never pushes an error code when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., spin), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor.

```
In file kern/trap.c
/*
Using the SETGATE macro, define idt entries for external device interrupts i.e. IRQ’s. Idt for the Timer  i.e. IRQ0 defined differently as it has an explicit handler to enable preemption of the environment.
*/
SETGATE (idt[IRQ_OFFSET + IRQ_TIMER], false, GD_KT, IRQ_INTTimer, 0);

        for (int i = 0; i < 15; i ++)
        {
```

```
                SETGATE (idt[IRQ_OFFSET + i], false, GD_KT, IRQ_INTError, 0);
        }
```

```
in file kern/trapentry.S
/*
Define handlers for IRQ's. Also define the declare the function defined here in
kern/trap.h
*/
TRAPHANDLER_NOEC (IRQ_INTTimer, IRQ_OFFSET + IRQ_TIMER)
TRAPHANDLER_NOEC (IRQ_INTError, IRQ_OFFSET + IRQ_ERROR)
```

```
In file kern/env.c
        // Enable interrupts while in user mode.
        // LAB 4: Your code here.
        e -> env_tf.tf_eflags |= FL_IF;
```

Handling Clock Interrupts

In the user/spin program, after the child environment was first run, it just spun in a loop, and the kernel never got control back. We need to program the hardware to generate clock interrupts periodically, which will force control back to the kernel where we can switch control to a different user environment.

The calls to lapic_init and pic_init (from i386_init in init.c), which we have written for you, set up the clock and the interrupt controller to generate interrupts. You now need to write the code to handle these interrupts.

Exercise 14. *Modify the kernel's trap_dispatch() function so that it calls sched_yield() to find and run a different environment whenever a clock interrupt takes place.*

You should now be able to get the user/spin test to work: the parent environment should fork off the child, sys_yield() to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

```
if (tf -> tf_trapno == IRQ_OFFSET + IRQ_TIMER)
        {
                lapic_eoi();
                sched_yield();
        }
```

Inter-Process communication (IPC)

We've been focusing on the isolation aspects of the operating system, the ways it provides the illusion that each program has a machine all to itself. Another important service of an operating system is to allow programs to communicate with each other when they want to. It can be quite powerful to let programs interact

with other programs. The Unix pipe model is the canonical example.

IPC in JOS

You will implement a few additional JOS kernel system calls that collectively provide a simple interprocess communication mechanism. You will implement two system calls, sys_ipc_recv and sys_ipc_try_send. Then you will implement two library wrappers ipc_recv and ipc_send.

The "messages" that user environments can send to each other using JOS's IPC mechanism consist of two components: a single 32-bit value, and optionally a single page mapping. Allowing environments to pass page mappings in messages provides an efficient way to transfer more data than will fit into a single 32-bit integer, and also allows environments to set up shared memory arrangements easily.

Sending and Receiving Messages

To receive a message, an environment calls sys_ipc_recv. This system call de-schedules the current environment and does not run it again until a message has been received. When an environment is waiting to receive a message, any other environment can send it a message - not just a particular environment, and not just environments that have a parent/child arrangement with the receiving environment. In other words, the permission checking that you implemented in Part A will not apply to IPC, because the IPC system calls are carefully designed so as to be "safe": an environment cannot cause another environment to malfunction simply by sending it messages (unless the target environment is also buggy).

To try to send a value, an environment calls sys_ipc_try_send with both the receiver's environment id and the value to be sent. If the named environment is actually receiving (it has called sys_ipc_recv and not gotten a value yet), then the send delivers the message and returns 0. Otherwise the send returns -E_IPC_NOT_RECV to indicate that the target environment is not currently expecting to receive a value.

A library function ipc_recv in user space will take care of calling sys_ipc_recv and then looking up the information about the received values in the current environment's struct Env.

Similarly, a library function ipc_send will take care of repeatedly calling sys_ipc_try_send until the send succeeds.

Transferring Pages

When an environment calls sys_ipc_recv with a valid dstva parameter (below UTOP), the environment is stating that it is willing to receive a page mapping. If the sender sends a page, then that page should be mapped at dstva in the receiver's address space. If the receiver already had a page mapped at dstva, then that previous page is unmapped.

When an environment calls sys_ipc_try_send with a valid srcva (below UTOP), it means the sender wants to send the page currently mapped at srcva to the receiver, with permissions perm. After a successful IPC, the sender keeps its original

mapping for the page at srcva in its address space, but the receiver also obtains a mapping for this same physical page at the dstva originally specified by the receiver, in the receiver's address space. As a result this page becomes shared between the sender and receiver.

If either the sender or the receiver does not indicate that a page should be transferred, then no page is transferred. After any IPC the kernel sets the new field env_ipc_perm in the receiver's Env structure to the permissions of the page received, or zero if no page was received.

Exercise 15. *Implement sys_ipc_recv and sys_ipc_try_send in kern/syscall.c. Read the comments on both before implementing them, since they have to work together. When you call envid2env in these routines, you should set the checkperm flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.*

Then implement the ipc_recv and ipc_send functions in lib/ipc.c.

Use the user/pingpong and user/primes functions to test your IPC mechanism. user/primes will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read user/primes.c to see all the forking and IPC going on behind the scenes.

```
// Try to send 'value' to the target env 'envid'.
// If srcva < UTOP, then also send page currently mapped at 'srcva',
// so that receiver gets a duplicate mapping of the same page.
//
// The send fails with a return value of -E_IPC_NOT_RECV if the
// target is not blocked, waiting for an IPC.
//
// The send also can fail for the other reasons listed below.
//
// Otherwise, the send succeeds, and the target's ipc fields are
// updated as follows:
//    env_ipc_recving is set to 0 to block future sends;
//    env_ipc_from is set to the sending envid;
//    env_ipc_value is set to the 'value' parameter;
//    env_ipc_perm is set to 'perm' if a page was transferred, 0 otherwise.
// The target environment is marked runnable again, returning 0
// from the paused sys_ipc_recv system call.  (Hint: does the
```

```
// sys_ipc_recv function ever actually return?)
//
// If the sender wants to send a page but the receiver isn't asking for one,
// then no page mapping is transferred, but no error occurs.
// The ipc only happens when no errors occur.
//
// Returns 0 on success, < 0 on error.
// Errors are:
//   -E_BAD_ENV if environment envid doesn't currently exist.
//        (No need to check permissions.)
//   -E_IPC_NOT_RECV if envid is not currently blocked in sys_ipc_recv,
//        or another environment managed to send first.
//   -E_INVAL if srcva < UTOP but srcva is not page-aligned.
//   -E_INVAL if srcva < UTOP and perm is inappropriate
//        (see sys_page_alloc).
//   -E_INVAL if srcva < UTOP but srcva is not mapped in the caller's
//        address space.
//   -E_INVAL if (perm & PTE_W), but srcva is read-only in the
//        current environment's address space.
//   -E_NO_MEM if there's not enough memory to map srcva in envid's
//        address space.
/*
Function is easy and self explanatory.
*/
        static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
        // LAB 4: Your code here.
        int a  = 0;
        int r_perm = 0;
        struct Env* d_env = NULL;

        uintptr_t address = (uintptr_t) srcva;
```

```
        a = envid2env (envid, &d_env, 0);
        if (a < 0)
        {
                return a;
        } else if (!(d_env -> env_ipc_recving))
        {
                return -E_IPC_NOT_RECV;
        } else if ((address < UTOP) && (address % PGSIZE) != 0)
        { return -E_INVAL;
        } else if ((address < UTOP) && (perm & ~PTE_SYSCALL) != 0)
        {
                return -E_INVAL;
        }

        if (address < UTOP)
        {
                pte_t* pte = NULL;
                  struct PageInfo* page = page_lookup (curenv -> env_pgdir, srcva,
&pte);
                if (!page)
                {
                        return -E_INVAL;
                } else if ((perm & PTE_W) && !(*pte &PTE_W))
                {
                        return -E_INVAL;
                }

                 a = page_insert (d_env -> env_pgdir, page, d_env -> env_ipc_dstva,
perm);
                if (a < 0)
                        return a;
```

```
                r_perm = perm;
        }

        d_env->env_ipc_value = value;
        d_env->env_ipc_from = curenv->env_id;
        d_env->env_ipc_perm = r_perm;
        d_env->env_ipc_recving = false;
        d_env->env_status = ENV_RUNNABLE;
        return 0;

        //        panic("sys_ipc_try_send not implemented");
}

// Block until a value is ready.  Record that you want to receive
// using the env_ipc_recving and env_ipc_dstva fields of struct Env,
// mark yourself not runnable, and then give up the CPU.
//
// If 'dstva' is < UTOP, then you are willing to receive a page of data.
// 'dstva' is the virtual address at which the sent page should be mapped.
//
// This function only returns on error, but the system call will eventually
// return 0 on success as will set register eax to 0.
// Return < 0 on error.  Errors are:
//    -E_INVAL if dstva < UTOP but dstva is not page-aligned.
/*
Function is easy and self explanatory.
*/
         static int
sys_ipc_recv(void *dstva)
{
         // LAB 4: Your code here.
         uintptr_t address = (intptr_t) dstva;
```

```
        if ((address < UTOP) && (address % PGSIZE) != 0)
        {
                return -E_INVAL;
        }

        curenv -> env_status = ENV_NOT_RUNNABLE;
        curenv -> env_ipc_recving = true;

        if (address >= UTOP)
        {
                curenv -> env_tf.tf_regs.reg_eax = 0;
                sched_yield();
        }

        curenv -> env_ipc_dstva = dstva;
        curenv -> env_tf.tf_regs.reg_eax = 0;
        sched_yield();


        //panic("sys_ipc_recv not implemented");
        return 0;
}
In file lib/ipc.c

// Receive a value via IPC and return it.
// If 'pg' is nonnull, then any page sent by the sender will be mapped at
//    that address.
// If 'from_env_store' is nonnull, then store the IPC sender's envid in
//    *from_env_store.
// If 'perm_store' is nonnull, then store the IPC sender's page permission
//    in *perm_store (this is nonzero iff a page was successfully
//    transferred to 'pg').
// If the system call fails, then store 0 in *fromenv and *perm (if
```

```
//    they're nonnull) and return the error.
// Otherwise, return the value sent by the sender
//
// Hint:
//   Use 'thisenv' to discover the value and who sent it.
//   If 'pg' is null, pass sys_ipc_recv a value that it will understand
//   as meaning "no page".  (Zero is not the right value, since that's
//   a perfectly valid place to map a page.)
/*
The value sent to the system calls as va is KERNBASE if no page is to be mapped.
Since the KERNBASE is not below UTOP, no page mapping will be established and
will set in the sorresponding perm parameter of the environment.
Function is easy and self explanatory.
*/
        int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
        int a = 0;
        if (!pg)
                pg = (void*) KERNBASE;

        a = sys_ipc_recv (pg);

        envid_t s_envid = 0;
        int s_perm = 0;

        if (a >= 0)
        {
                s_envid = thisenv -> env_ipc_from;
                s_perm = thisenv -> env_ipc_perm;
        }
```

```
        if (from_env_store)
                *from_env_store = s_envid;

        if (perm_store)
                *perm_store = s_perm;

        return (a >=0) ? thisenv -> env_ipc_value : a;

        // panic("ipc_recv not implemented");
        // return 0;
}

// Send 'val' (and 'pg' with 'perm', if 'pg' is nonnull) to 'toenv'.
// This function keeps trying until it succeeds.
// It should panic() on any error other than -E_IPC_NOT_RECV.
//
// Hint:
//   Use sys_yield() to be CPU-friendly.
//   If 'pg' is null, pass sys_ipc_try_send a value that it will understand
//   as meaning "no page".  (Zero is not the right value.)
/*
Function is easy and self explanatory.
*/
        void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
        if (!pg) {
                pg = (void *)KERNBASE;
        }
        while(true) {
                int err = sys_ipc_try_send(to_env, val, pg, perm);
                if (err == -E_IPC_NOT_RECV) {
```

```
                sys_yield();
        } else if (err == 0) {
                break;
        } else {
                panic("ipc_send failed: %e", err);
        }
    }
}
```