CS3510: Operating Systems Lab
Assignment 4

Due on 28th October 2013

Dr. V.Kamakoti

Abhinav Garlapati
CS11B030

Exercise 1 (Mapping Memory Mapped IO Region)

In this question, we are asked to write a fucntion which will help the OS to map a contigious region of virtual address to an given virtual address.

```
        size = ROUNDUP(size, PGSIZE);
        if( size > (MMIOLIM - MMIOBASE))
                panic("Errorrr!!");
        boot_map_region(kern_pgdir, base, size, pa, PTE_PCD | PTE_PWT | PTE_W);
        base += size;
        return (base-size);
```

So first we round up the size to be a multiple of PGSIZE. Then we use boot map region to map this region.

Note that we use **PTE_PCD** and **PTE_PWT** which mean that these pages are not cached, and any change is directly written through. Thus enabling us to use it as I/O interface.

Exercise 2 (Pages for MP init)

Here we are asked to modify the page_init function which we have written in pmap., in such a way that the page corresponding to MPENTRY_PADDR is not added to the page_free_list.

```
int mp_index = PGNUM(MPENTRY_PADDR);
for (i = 1; i < npages_basemem && i != mp_index; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
}
```

Question 1 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?

boot.S starts before the first CPU boots, it is used to initialize the first CPU. This works in real mode, with the code also being in lower address space. The code has been loaded there by the boot loader. It starts at 0x7c00. The code in mpentry.S is being loaded and used by a CPU which has just booted. But the code has been linked to start at KERN_BASE. This is not the case in case of the boot.S where we explictly mention the linking address as 0x7c00. Thus when the new CPU tries to execute, all its variables according the linker are linked starting from mp_start. But the cpu iteself is in real mode. and the physical address of the code is 0x7000(= MPENTRY_PADDR).

Consider a label whose physical address is x, then the linker would link that address to be
x-MPENTRY_PADDR + mp_start.

The linker itself would have linked mp_start to some virtual address offseted from KERN_BASE. Thus by using this macro the CPU gets the actual physical address of the code.

Exercise 3 (Kernel Stack Allocation)

Here we map, each processor's kernel stack, using the given data structures. This is pretty straight forward.

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

Exercise 4 (Per CPU Trap Frame Setting)

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.

```
        (thiscpu->cpu_ts).ts_esp0 = (KSTACKTOP-(thiscpu->cpu_id)*(KSTKSIZE+KSTKGAP));
        (thiscpu->cpu_ts).ts_ss0 = GD_KD;

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

        // Load the TSS selector
        ltr(GD_TSS0 + ((thiscpu->cpu_id)<<3));

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

Exercise 5 (Locking the Kernel)

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

Here we have to place 3 locks and one unlock kernel. This is to ensure only one CPU will enter the kernel at one time.

Question 2

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.

If a CPU is servicing an interrupt, and then a non maskable interrupt is raised. If a common stack is used, then for the context switch the common stack is used which screws the current running stack, inspite of it waiting on the lock acquired after trap is called.

Exercise 6 (Scheduling!)

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

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

```
        int start = -1;
        if(curenv != NULL)
                start = ENVX(curenv->env_id);
        for(i = start+1; i < (NENV+start); i++)
                if(envs[i%NENV].env_status == ENV_RUNNABLE)
                        env_run(&envs[i%NENV]);
        if(curenv != NULL && curenv->env_status == ENV_RUNNING)
                env_run(curenv);
```

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 same kernel mapping is used across all the cpus. This ensures that there are no problems when we access a kernel data strcutre using the derefernced location.

Question 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?

The context is saved on the kernel status, made by the processor when there is a context switch.

Exercise 7 (User System Calls)

Implement the system calls described above in kern/syscall.c. 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.

1. **sys_exofork**

```
        retval = env_alloc(&e, curenv->env_id);
        if(retval < 0) {
                panic("Cup param selection");
                return retval;
        }

        // Set the new environment to be not runnable, and to return 0 if
        //  the environment allocation was successful.
        if(retval == 0) {
                //Setting the child status to not runnable
                e->env_status = ENV_NOT_RUNNABLE;
                //Setting the trapframe as of the parent.
                e->env_tf = curenv->env_tf;
                //Changing the return value to 0.
                e->env_tf.tf_regs.reg_eax = 0;
                //Returning the pid of the child in the parent process.
                retval = e->env_id;
        }

        return retval;
```

2. **sys_env_set_status**

```
        struct Env *e;
        if (status != ENV_RUNNABLE
                 && status != ENV_NOT_RUNNABLE)
                        return -E_INVAL;

        if (envid2env(envid, &e, 1) < 0)
                return -E_BAD_ENV;
                //Setting Status of the parent process.
        e->env_status = status;
        return 0;
```

3. **sys_page_alloc**

```
                struct Env *e;
        struct PageInfo *page;
        if ((uint32_t) va >= UTOP
         || ((uint32_t) va % PGSIZE)!=0)
                return -E_INVAL;

        if ((perm & PTE_U) == 0 ||
         (perm & PTE_P) == 0)
                return -E_INVAL;

        if (envid2env(envid, &e, 1) < 0)
                return -E_BAD_ENV;

        if ((page = page_alloc(ALLOC_ZERO)) == NULL)
                return -E_NO_MEM;`
        if (page_insert(e->env_pgdir, page, va, perm) != 0 ) {
                page_free(page);
                return -E_NO_MEM;
        }
        return 0;
```

4. **sys_page_map**

```
        struct Env *se, *de;
        pte_t *septe;
        struct PageInfo *page;

        if (envid2env(srcenvid, &se, 1) < 0
                || envid2env(dstenvid, &de, 1))
                return -E_BAD_ENV;
        // case 1
        if ((uint32_t) srcva >= UTOP
                || (uint32_t) dstva >= UTOP
                 || (uint32_t) srcva % PGSIZE != 0
                 || (uint32_t) dstva % PGSIZE != 0) {
                cprintf("sys_page_map: invalid boundary or page-aligned\n");
                return -E_INVAL;
        }
        // case 2
        if ( (perm & PTE_U) == 0
                || (perm & PTE_P) == 0
                || (perm & ~PTE_SYSCALL) != 0) {
                cprintf("sys_page_map: invalid perm\n");
                return -E_INVAL;
        }

        if (!(page = page_lookup(se->env_pgdir, srcva, &septe))) {
                cprintf("sys_page_map: page not found\n");
                return -E_INVAL;
        }

        if ((perm & PTE_W) != 0
                && (*septe & PTE_W) == 0) {
                cprintf("sys_page_map: invalid PTE_W\n");
                return -E_INVAL;
        }

        if (page_insert(de->env_pgdir, page, dstva, perm)<0)
                return -E_NO_MEM;



        return 0;
```

5. **sys_page_unmap**

```
        struct Env *e;
        if ((uint32_t) va >= UTOP
                || (uint32_t) va % PGSIZE != 0)
                return -E_INVAL;
        if (envid2env(envid, &e, 1) < 0)
                return -E_BAD_ENV;

        page_remove(e->env_pgdir, va);
        return 0;
```

Exercise 8 continued on next page. . .

Exercise 8 (Page Fault Handler Entry Point *sys_env_set_pgfault_upcall*)

This system call will register a page fault handler for a user process.

```
        struct Env *e;
        if(envid2env(envid, &e, 1) < 0)
                return -E_BAD_ENV;
        e->env_pgfault_upcall = func;
        return 0;
```

Exercise 9 (The page fault Handler *page_fault_handler*)

Here we set up the trap frame for the user process on a special user expcetion stack.

```
        uint32_t esp = UXSTACKTOP;

        if(curenv->env_pgfault_upcall == NULL) {
                goto destroy;
        }
        if (tf->tf_esp < UXSTACKTOP && tf->tf_esp >= UXSTACKTOP-PGSIZE)
                esp = tf->tf_esp - 4;

        user_mem_assert(curenv, curenv->env_pgfault_upcall, PGSIZE, PTE_U | PTE_P );
        user_mem_assert(curenv,(void *)(UXSTACKTOP -4), 4, 0);
        struct UTrapframe *utf = (struct UTrapframe *)(esp - (uint32_t) sizeof(struct
            UTrapframe));

        utf->utf_fault_va = fault_va;
        utf->utf_fault_va = fault_va;
        utf->utf_err = tf->tf_err;
        utf->utf_regs = tf->tf_regs;
        utf->utf_eip = tf->tf_eip;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_esp = tf->tf_esp;

        tf->tf_esp = (uintptr_t) utf;
        tf->tf_eip = (uintptr_t) curenv->env_pgfault_upcall;

        env_run(curenv);

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

Exercise 10 (*_pgfault_upcall*)

This is an assembly code, which takes care of the actual switch between the kernel to the user exception stack and also to back to the line where the exception occurred.

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
        // Call the C page fault handler.
        pushl %esp                      // function argument: pointer to UTF
        movl _pgfault_handler, %eax
        call *%eax
        addl $4, %esp                   // pop function argument

        // 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 %esp, %ebx
        movl 40(%esp), %eax
        movl 48(%esp), %esp
        pushl %eax

        // Restore the trap-time registers.  After you do this, you
        // can no longer modify any general-purpose registers.
        // LAB 4: Your code here.

        movl %ebx, %esp
        subl $4, 48(%esp)
        popl %eax
        popl %eax
        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.
        add $4, %esp
        popfl
```

```
	// Switch back to the adjusted trap-time stack.
	// LAB 4: Your code here.
	popl %esp

	// Return to re-execute the instruction that faulted.
	// LAB 4: Your code here.
	ret
```

Exercise 11 (*set_pgfault_handler*)

This system call is to register a page fault handler for a user process. In this routine, we get the current environment id, and then allocate the stack space for exception handeling. Then we set the upcall, and the there is code for setting the pgfault handler.

```
                envid_t envid = sys_getenvid();
                if(sys_page_alloc(envid, (void*)(UXSTACKTOP-PGSIZE), PTE_P  | PTE_U | PTE_W)
                    )
                        panic("No memory for pgfault exception!");
                if(sys_env_set_pgfault_upcall(envid, _pgfault_upcall) < 0)
                        panic("Can't set exception handler");
```

Exercise 12 (*fork*, *duppage* and *pgfault*)

Fork

Fork when called first sets the page fault handler to pgfault. Then we call the sys_exofork() which we have defined in one of our previous exercises. Then we look through all the page table entries and duplicate them in the child process using the id obtained from exofork(). Then we run the child process.

fork()

```
        set_pgfault_handler(pgfault);
        envid_t childid = sys_exofork();
        if(childid < 0)
                panic("Fork Failed\n");
        if(childid == 0) {
                thisenv = &envs[ENVX(sys_getenvid())];
                return 0;
        }

        int i, j;
        for(i = 0; i < PDX(UTOP); i++) {
                if (!(uvpd[i] & PTE_P)) continue;
                for(j = 0; (j < 1024) && (i*NPDENTRIES + j < PGNUM(UXSTACKTOP - PGSIZE)); j
                    ++ ) {
                        if(!uvpt[i*NPDENTRIES + j] & PTE_P) continue;
                        if(duppage(childid, i*NPDENTRIES + j) < 0)
                                panic("dup page failed");
                }
        }
        if ((sys_page_alloc(childid, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_P | PTE_W))
            < 0) {
          panic("Allocation of page for Exception stack cups!\n");
        }

        if ((sys_env_set_pgfault_upcall(childid, thisenv->env_pgfault_upcall)) < 0) {
          panic("Unable to set child process' upcall");
        }

         // Copy own uxstack to temp page
        memmove((void *)(UXSTACKTOP - PGSIZE), PFTEMP, PGSIZE);
        int r;
        // Unmap temp page
        if (sys_page_unmap(sys_getenvid(), PFTEMP) < 0) {
                return -1;
        }

        if ((r = sys_env_set_pgfault_upcall(childid, thisenv->env_pgfault_upcall)) < 0)
                panic("sys_env_set_pgfault_upcall: error %e\n", r);

        if ((r = sys_env_set_status(childid, ENV_RUNNABLE)) < 0) {
                cprintf("sys_env_set_status: error %e\n", r);
                return -1;
        }


        return childid;
```

duppage

This is a straight forward function to map a given page in the child process.

```
                unsigned va = (pn << PGSHIFT);
        if (!(uvpt[pn] & PTE_P))
                return -E_INVAL;
        if (!((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW))) {
                if ((sys_page_map(0, (void *) va, envid, (void *) va, PGOFF(uvpt[pn]) )) <
                    0) {
                        panic("Sys page map cupped!\n ");
                }
        }
        if (pn >= PGNUM(UTOP) || va >= UTOP)
                panic("page out of UTOP\n");

        if (!(uvpt[pn] & PTE_U))
                panic("page must user accessible\n");

        // change current env perm
        if ((sys_page_map(0, (void *) va, envid, (void *) va, PTE_P | PTE_U | PTE_COW)) < 0)
                panic("Sys Page Map cupped, for the current env\n");

        // use syscall to change parent perm
        if ((r = sys_page_map(0, (void *) va, 0, (void *) va,
                PTE_P | PTE_U | PTE_COW)) < 0)
                panic("sys_page_map: error %e\n", r);

        if ((uvpt[pn] & PTE_W) && (uvpt[pn] & PTE_COW))
                panic("duppage: should now set both PTE_W and PTE_COW\n");

        return 0;
```

pgfault

```
        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.
        if ((err & FEC_WR) == 0) {
                panic("Denied write 0x%x: err %x\n\n", (uint32_t) addr, err);
        }
        pte_t pte = uvpt[PGNUM((uint32_t)addr)];
        if (!(pte & PTE_COW)) {
                panic("Denied copy-on-write 0x%x, env_id 0x%x\n", (uint32_t) addr, thisenv->
                    env_id);
        }
        // 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.
        //   No need to explicitly delete the old page's mapping.

        // LAB 4: Your code here.
        void *tmpva = (void *) ROUNDDOWN((uint32_t) addr, PGSIZE);
        if ((r = sys_page_alloc(0, (void *) PFTEMP, PTE_P|PTE_W|PTE_U)) < 0) {
                panic("sys_page_alloc: error %e\n", r);
        }

        memmove((void *)PFTEMP, tmpva, PGSIZE);

        if ((r = sys_page_map(0, (void *)PFTEMP, 0, tmpva,
                PTE_P|PTE_W|PTE_U)) < 0)
                panic("sys_page_map: error %e\n", r);
```

Exercise 13 (IDT entry for IRQ interrupts)

Here we add entries in IDT and trap handlers for the IRQ interrupts form 32-47.

```
        extern void trap_32();
        SETGATE(idt[32], 0, GD_KT, trap_32, 0);
        extern void trap_33();
        SETGATE(idt[33], 0, GD_KT, trap_33, 0);
        extern void trap_34();
        SETGATE(idt[34], 0, GD_KT, trap_34, 0);
        extern void trap_35();
        SETGATE(idt[35], 0, GD_KT, trap_35, 0);
        extern void trap_36();
        SETGATE(idt[36], 0, GD_KT, trap_36, 0);
        extern void trap_37();
        SETGATE(idt[37], 0, GD_KT, trap_37, 0);
        extern void trap_38();
        SETGATE(idt[38], 0, GD_KT, trap_38, 0);
        extern void trap_39();
        SETGATE(idt[39], 0, GD_KT, trap_39, 0);
        extern void trap_40();
        SETGATE(idt[40], 0, GD_KT, trap_40, 0);
        extern void trap_41();
        SETGATE(idt[41], 0, GD_KT, trap_41, 0);
        extern void trap_42();
        SETGATE(idt[42], 0, GD_KT, trap_42, 0);
        extern void trap_43();
        SETGATE(idt[43], 0, GD_KT, trap_43, 0);
        extern void trap_44();
        SETGATE(idt[44], 0, GD_KT, trap_44, 0);
        extern void trap_45();
        SETGATE(idt[45], 0, GD_KT, trap_45, 0);
        extern void trap_46();
        SETGATE(idt[46], 0, GD_KT, trap_46, 0);
        extern void trap_47();
        SETGATE(idt[47], 0, GD_KT, trap_47, 0);
```

```
        TRAPHANDLER_NOEC(trap_32, 32)
        TRAPHANDLER_NOEC(trap_33, 33)
        TRAPHANDLER_NOEC(trap_34, 34)
        TRAPHANDLER_NOEC(trap_35, 35)
        TRAPHANDLER_NOEC(trap_36, 36)
        TRAPHANDLER_NOEC(trap_37, 37)
        TRAPHANDLER_NOEC(trap_38, 38)
        TRAPHANDLER_NOEC(trap_39, 39)
        TRAPHANDLER_NOEC(trap_40, 40)
        TRAPHANDLER_NOEC(trap_41, 41)
        TRAPHANDLER_NOEC(trap_42, 42)
        TRAPHANDLER_NOEC(trap_43, 43)
        TRAPHANDLER_NOEC(trap_44, 44)
        TRAPHANDLER_NOEC(trap_45, 45)
        TRAPHANDLER_NOEC(trap_46, 46)
        TRAPHANDLER_NOEC(trap_47, 47)
```

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.)

```
        case IRQ_OFFSET+0:
                lapic_eoi();
                sched_yield();
```

Exercise 15 (Inter Process Communication protocol)

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.

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
        // LAB 4: Your code here.
        struct Env *e;
        struct PageInfo *page;
        pte_t *pte;

        if (envid2env(envid, &e, 0) < 0)
                return -E_BAD_ENV;
        if (e->env_ipc_recving == 0)
                return -E_IPC_NOT_RECV;


        if ((uint32_t) e->env_ipc_dstva < UTOP
            && (uint32_t) srcva < UTOP) {

                page = page_lookup(curenv->env_pgdir, srcva, &pte);
                if (!page)
                        return -E_INVAL;

                if ((perm & PTE_W) && (*pte & PTE_W) == 0)
                        return -E_INVAL;

                if ((page_insert(e->env_pgdir, page, e->env_ipc_dstva, perm)) < 0) {
                        return -E_NO_MEM;
                }

        }

    //cprintf("sys_ipc_try_send: from 0x%x\n", e->env_ipc_from);

        e->env_ipc_recving = 0;
        e->env_ipc_dstva = (void *) UTOP; // invalid dstva

        e->env_ipc_from = curenv->env_id;
        e->env_ipc_value = value;
        e->env_ipc_perm = perm;

        e->env_status = ENV_RUNNABLE;
        return 0;
        panic("sys_ipc_try_send not implemented");
}

static int
sys_ipc_recv(void *dstva)
{

        // LAB 4: Your code here.
        if (((uint32_t) dstva < UTOP)
                && ((uint32_t) dstva % PGSIZE != 0)) {
                        return -E_INVAL;
        }
        // cprintf("sys_ipc_recv: nva 0x%x\n", (uint32_t) dstva);
        curenv->env_ipc_dstva = dstva;
        curenv->env_ipc_recving = 1;

        curenv->env_ipc_value = 0;
        curenv->env_ipc_from = 0;
        curenv->env_ipc_perm = 0;

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

        // should not call schedule_yield because of trap call by timer
        while(curenv->env_ipc_recving)
                sched_yield();
        return 0;
}
```

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
        int32_t val;
        envid_t sender;
        int perm;

        if (!pg) {
                pg = (void *) UTOP; // invalid dstva
        }

        if ((val = sys_ipc_recv(pg)) < 0) {
                sender = 0;
                perm = 0;
        } else {
                sender = thisenv->env_ipc_from;
                perm = thisenv->env_ipc_perm;
                val = thisenv->env_ipc_value;
        }

        if (from_env_store) {
                *from_env_store = sender;
        }

        if (perm_store) {
                *perm_store = perm;
        }

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

void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
        int r;
        if (!pg) {
                pg = (void *) UTOP; // invalid srcva
                perm = 0;
        }
        while (1) {
                r = sys_ipc_try_send(to_env, val, pg ? pg : (void *) UTOP, perm);
                if (!r) {
                        break;
                } else if (r != -E_IPC_NOT_RECV) {
                        panic("ipc_send: error %e\n", r);
                }

                sys_yield();
        }
        //panic("ipc_send not implemented");
}
```