Lab4

Chen Zhikai 516370910008

Part A

在这一部分里，我们要完成SMP中多个处理器的启动工作。主要的步骤是先启动bootstrap processor,然后带动AP启动。 第一个exercise里，我们要实现mmio_map_region，来让我们能够把device通过memory map映射到memory上然后就可以操作 device了。 这个练习本身并不难，但是由于我在这里犯了一个很隐蔽的错误(由糟糕的变量命名导致)，后面的测试中单核是对的，但是 一旦启动多核make就会卡住或者无限死循环，经历了漫长的debug我才发现是这个函数写错了，可见这个函数的重要性。

```
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:
        size = (size_t)ROUNDUP(size + pa, PGSIZE);
        pa = ROUNDDOWN(pa, PGSIZE);
        if (base + size > MMIOLIM){
                panic("Overflow MMIOLIM region\n");
        }
        boot_map_region(kern_pgdir, base, size-pa, pa, PTE_W | PTE_PCD | PTE_PWT);
        uintptr_t ret_addr = base;
        base += (size - pa);
        return (void *)ret_addr;

}
```

主要就是用我们的boot_map_region。在这个lab里，我发现之前的4M页boot_map_region_large()会造成一些错误，但不知道背后的原因，所以我把之前4M映射的那片区域 改回了4K映射。

Exercise 2

```
uint32_t i;
        page_free_list = NULL;
        for (; i < npages_basemem; i++) {
        if (i == MPENTRY_PADDR / PGSIZE) {
                pages[i].pp_ref = 1;
                pages[i].pp_link = NULL;
                continue;
        }
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
        }

        for (; i < (EXTPHYSMEM / PGSIZE); i++) {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
        }
        for (; i < PGNUM(PADDR(boot_alloc(0))); i++) {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
        }
        for (; i < npages; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
        }
}
```

主要新加了对MPENTRY_PADDR的判断，这里这有一个page的大小

Question 1

根据代码注释中的提示，在原来的boot过程中，这些具体的地址由linker给出，在Lab1中我们也看到过对应的配置文件；现在在这里AP的boot过程中，我们使用MPBOOTPHYS来计算一个符号在gdt中的具体位置， 符号的具体位置(注意这时候AP还在实模式下，所以只能使用物理地址)。为了把它加载到我们想要的地址，我们就要使用这种方法,而不是让linker指定地址。

Exercise 3

```
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) {
                boot_map_region(kern_pgdir,
                        KSTACKTOP - KSTKSIZE - i * (KSTKSIZE + KSTKGAP),
                        KSTKSIZE,
                        PADDR(percpu_kstacks[i]),
                        PTE_W);
        }

}
```

这里也是做一下boot_map_region(),注意这里的KSTKGAP

Exercise 4

```
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.
        //
        // LAB 4: Your code here:
        // Setup a TSS so that we get the right stack
        // when we trap to the kernel.
        int index = thiscpu->cpu_id;
        thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - index * (KSTKSIZE + KSTKGAP);
        thiscpu->cpu_ts.ts_ss0 = GD_KD;
        thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);


        // extern void sysenter_handler();
        // wrmsr(0x174, GD_KT, 0);                   /* SYSENTER_CS_MSR */
        // wrmsr(0x175, thiscpu->cpu_ts.ts_esp0 , 0);/* SYSENTER_ESP_MSR */
        // wrmsr(0x176, sysenter_handler, 0);        /* SYSENTER_EIP_MSR */
        // // Initialize the TSS slot of the gdt.
        // int GD_TSS_index = GD_TSS0 + (index << 3);
        gdt[(GD_TSS0 >> 3) + index ] = SEG16(STS_T32A, (uint32_t) (&(thiscpu-
>cpu_ts)),
                                sizeof(struct Taskstate), 0);
        gdt[(GD_TSS0 >> 3) + index].sd_s = 0;

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

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

这里要把原来的ts改为thiscpu.后面的TSS设置好像在lab3里有涉及过，基本上是按照原来的代码框架来写，比如这个GD_TSS0 >> 3,然后ts_iomb在这里需要配置一下。

Exercise 5

这里就是按照题目要求在上述的文件位置加上锁，然后在env_run()里解锁。注意这里的题目要求应该是没有考虑到上次设计的sysenter,如果system call没有走Interrupt而是走的sysenter,那么这里的锁会有问题。为了简单起见，我先注释掉了sysenter的功能。

Question 2

当一个interrupt发生的时候，会越过big kernel lock把trapframe相关的寄存器信息push到kernel stack上，这时候没有per-cpu stack就会出错了。

Exercise 6

```
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.
        int i, cur=0;
        struct Env* running_env = NULL;
        if (curenv) cur=ENVX(curenv->env_id);
        else cur = 0;
        for (i = 0; i < NENV; ++i) {
                int j = (cur+i) % NENV;
                if (envs[j].env_status == ENV_RUNNABLE && (!running_env ||
envs[j].priority > running_env->priority)){
                        running_env = &envs[j];
                }
        }
        if (curenv && curenv->env_status == ENV_RUNNING && ((running_env == NULL)
|| (running_env->priority > curenv->priority)))
                env_run(curenv);
        if (running_env){
                env_run(running_env);
        }
        // sched_halt never returns
        sched_halt();
}
```

这里实现了最早Linux采用的调度算法，也是找到第一个runnable的process就行，由于后面我的challenge做的是调度，所以这里有些考虑priority的逻辑。写完 sched_yield()后，需要在几个位置调用一下，然后去init里初始化三个进程进行后面的测试.

Question 3

这是由于每一个environment在的kernel space都是相同的，自然这个e也是相同的

Question 4

trap.c文件中的trap()函数有一行curenv->env_tf = *tf完成了上述的任务

Challenge

选择Challenge scheduling policy 在env.h的env struct里添加一个新的项priority 然后添加一个新的systemcall sys_change_priority 同时新建一个priority_fork() 能够修改进程的priority 具体代码部分如下 envid_t priority_fork(int pr) { // LAB 4: Your code here. extern void _pgfault_upcall (void); set_pgfault_handler(pgfault);

```
envid_t childenv;
uintptr_t addr;
int res;
childenv = sys_exofork();
if (childenv < 0){
        panic("Error when creating child env");
} else if (childenv == 0){
        thisenv = &envs[ENVX(sys_getenvid())];
        sys_change_priority(pr);
        return 0;
}
for(addr = UTEXT; addr < USTACKTOP;addr += PGSIZE){
        if((uvpd[PDX(addr)] & PTE_P) && ((uvpt[PGNUM(addr)] & (PTE_P | PTE_U)) ==
(PTE_P | PTE_U))){
                duppage(childenv, PGNUM(addr));
        }
}
res = sys_page_alloc(childenv, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W |
PTE_P);
if(res < 0){
        panic("Error to allocate exception stack\n");
}
res = sys_env_set_pgfault_upcall(childenv, _pgfault_upcall);
if(res < 0){
        panic("Error in setting pgfault upcall\n");
}
res = sys_env_set_status(childenv, ENV_RUNNABLE);
if(res < 0){
        panic("Error in setting child process state");
}
return childenv;
```

}

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.
int i, cur=0;
struct Env* running_env = NULL;
if (curenv) cur=ENVX(curenv->env_id);
else cur = 0;
for (i = 0; i < NENV; ++i) {
        int j = (cur+i) % NENV;
        if (envs[j].env_status == ENV_RUNNABLE && (!running_env ||
envs[j].priority > running_env->priority)){
                running_env = &envs[j];
        }
}
if (curenv && curenv->env_status == ENV_RUNNING && ((running_env == NULL) ||
(running_env->priority > curenv->priority)))
        env_run(curenv);
if (running_env){
        env_run(running_env);
}
// sched_halt never returns
sched_halt();
```

} 然后修改了客户端的hello文件进行测试 目测有三个进程进行调度 void umain(int argc, char **argv) { int i; for (i = 1; i <= 3; ++i) { int pid = priority_fork(i); if (pid == 0) { cprintf("child %x\n with priority %x\n", i, i); int j; for (j = 0; j < 3; ++j) { cprintf("child %x\n with priority %x\n", i, i); sys_yield(); } break; } } }

Exercise 7

```
// 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.
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* newEnv;
        int res = env_alloc(&newEnv, curenv->env_id);
```

```
        if(res < 0){
                panic("Fail to establish a new environment");
                return res;
        }
        newEnv->env_tf = curenv->env_tf;
        newEnv->env_status = ENV_NOT_RUNNABLE;
        newEnv->env_tf.tf_regs.reg_eax = 0;
        return newEnv->env_id;

}

// 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.
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* newEnv;
        int res = envid2env(envid, &newEnv, 1);
        if (res < 0){
                panic("No corresponding environment for this environment id");
                return -E_BAD_ENV;
        }
        if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE){
                panic("Invalid status");
                return -E_INVAL;
        }
        newEnv->env_status = status;
        return 0;
}

// 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.
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        // LAB 4: Your code here.
```

```
        struct Env* newEnv;
        int r = envid2env(envid, &newEnv, 1);
        if (r < 0){
                return -E_BAD_ENV;
        }
        newEnv->env_pgfault_upcall = func;
        return 0;
}

// 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.
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 PageInfo* newpage = page_alloc(ALLOC_ZERO);
        if (((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) || (uintptr_t)va >= UTOP
|| PGOFF(va) || (perm & (~PTE_SYSCALL))){
                return -E_INVAL;
        }
        if(!newpage){
                panic("Out of memory");
                return -E_NO_MEM;
        }
        struct Env* newEnv;
        int res = envid2env(envid, &newEnv, 1);
        if(res < 0){
                panic("No corresponding envid");
                return -E_BAD_ENV;
        }
        int r;
        if((r = page_insert(newEnv->env_pgdir, newpage, va, perm)) < 0){
                panic("Error inserting");
```

```
                page_free(newpage);
        }
        return 0;
}

// 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
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 *env1;
        struct Env *env2;
        pte_t* pte;
        int res1 = envid2env(srcenvid, &env1, 1);
        int res2 = envid2env(dstenvid, &env2, 1);
        if (res1 < 0 || res2 < 0){
                return -E_BAD_ENV;
        }
        if (((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) || (uintptr_t)srcva >=
UTOP || PGOFF(srcva) || PGOFF(dstva) || (uintptr_t)dstva >= UTOP || (perm &
(~PTE_SYSCALL))){
                return -E_INVAL;
        }
        struct PageInfo* lookupPage;
        if (!(lookupPage = page_lookup(env1->env_pgdir, srcva, &pte))){
                panic("No mapped physical page for srcva");
                return -E_INVAL;
        }
        if ((perm & PTE_W) && !(*pte & PTE_W)){
                return -E_INVAL;
        }
```

```
        if (page_insert(env2->env_pgdir, lookupPage, dstva, perm)){
                return -E_NO_MEM;
        }
        return 0;

}

// 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.
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* newenv;
        int result = envid2env(envid, &newenv, 1);
        if(result < 0){
                return -E_BAD_ENV;
        }
        if ((uintptr_t)va >= UTOP || PGOFF(va)){
                return -E_INVAL;
        }
        page_remove(newenv->env_pgdir, va);
        return 0;
}
```

这里我们实现了一堆的system call.这些函数的流程都是一样的，通过envid2env获得具体地env,然后进行各种address和permession的check,如果有问题就返回对应的错误码。

Exercise 8

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        // LAB 4: Your code here.
        struct Env* newEnv;
        int r = envid2env(envid, &newEnv, 1);
        if (r < 0){
                return -E_BAD_ENV;
        }
        newEnv->env_pgfault_upcall = func;
        return 0;
}
```

设定好pgfault_upcall,为后面的COW fork做准备。

Exercise 9

```
if ((tf->tf_cs & 0x3) == 0) {
                panic("Kernel page fault");
        }

        if (curenv->env_pgfault_upcall) {
                struct UTrapframe *utf;
                if((uint32_t)(UXSTACKTOP - tf->tf_esp <= PGSIZE) && (uint32_t)
(UXSTACKTOP - tf->tf_esp) >= 1){
                        utf = (struct UTrapframe *)(tf->tf_esp - sizeof(void *) -
sizeof(struct UTrapframe));
                } else {
                        utf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct
UTrapframe));
                }
                user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe),
PTE_W);

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

这块代码真的很难写，要先手动做一个Utrapframe然后控制curenv换栈跳转到exception stack和pgfault_upcall上，也是OS里面处理错误的惯用套路。

Exercise 10

```
movl 0x28(%esp), %edx # 0x28=40, the position of eip
        movl 0x30(%esp), %eax # 0x30=48, the position of esp
        subl $0x4, %eax
        movl %edx, (%eax)
        movl %eax, 0x30(%esp)
        // Restore the trap-time registers.  After you do this, you
        // can no longer modify any general-purpose registers.
        // LAB 4: Your code here.
```

```
        addl $0x8, %esp
        popal # pop out the push_regs struct
        // 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 $0x4, %esp
        popfl # pop out the eflags
        // 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
```

这里就是按着注释写吧，绕来绕去把栈上的东西都弹走然后跳回到发生错误前的状态。

Exercise 11

```
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
        int r;

        if (_pgfault_handler == 0) {
                // First time through!
                // LAB 4: Your code here.
                if ((r = sys_page_alloc((envid_t)0, (void *)(UXSTACKTOP - PGSIZE),
PTE_U | PTE_P | PTE_W)) < 0){
                        panic("set page fault handler");
                }
                if ((r = sys_env_set_pgfault_upcall((envid_t)0, _pgfault_upcall))
< 0){
                        panic("sys_env_set_pgfault_upcall");
                }
        }

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

刚刚那个是system call,这个是library端，给user用的。

Exercise 12

```
static void
pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
```

```
        //uint32_t pos;
        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) || !(uvpd[PDX(addr)] & PTE_P) || !((uvpt[PGNUM(addr)]
& (PTE_P | PTE_COW)) == (PTE_P | PTE_COW))) {
                panic("Incorrect permission in pgfault\n");
        }
        // 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.
        addr = (void *)ROUNDDOWN(addr, PGSIZE);
        r = sys_page_alloc(0, PFTEMP, PTE_P | PTE_W | PTE_U);
        if (r < 0){
                panic("Fail to allocate the page\n");
        }
        memcpy(PFTEMP, addr, PGSIZE);
        // LAB 4: Your code here.
        r = sys_page_map(0, PFTEMP, 0, addr, PTE_P | PTE_W |PTE_U);
        if (r < 0){
                panic("fail to map the copy-on-write page\n");
        }
        r = sys_page_unmap(0, PFTEMP);
        if (r < 0){
                panic("Fail to unmap the old page in COW\n");
        }
        return;

}

//
// 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.
//
static int
duppage(envid_t envid, unsigned pn)
{
        int r;
```

```
        if ((uvpt[pn] & (PTE_W)) || (uvpt[pn] & PTE_COW)){
        // LAB 4: Your code here.
                r = sys_page_map(0, (void *)(pn * PGSIZE), envid, (void *)(pn *
PGSIZE), PTE_P | PTE_U | PTE_COW);
                if (r < 0){
                        panic("Error in duppage\n");
                }
                r = sys_page_map(0, (void *)(pn * PGSIZE), 0, (void *)(pn*PGSIZE),
PTE_P | PTE_U | PTE_COW);
                if (r < 0){
                        panic("Error in duppage\n");
                }
        } else {
                r = sys_page_map(0, (void *)(pn * PGSIZE), envid, (void *)
(pn*PGSIZE), PTE_P | PTE_U);
        }
        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.
//
envid_t
fork(void)
{
        // LAB 4: Your code here.
        extern void _pgfault_upcall (void);
        set_pgfault_handler(pgfault);

        envid_t childenv;
        uintptr_t addr;
        int res;
        childenv = sys_exofork();
        if (childenv < 0){
                panic("Error when creating child env");
        } else if (childenv == 0){
                thisenv = &envs[ENVX(sys_getenvid())];
                return 0;
        }
        for(addr = UTEXT; addr < USTACKTOP;addr += PGSIZE){
                if((uvpd[PDX(addr)] & PTE_P) && ((uvpt[PGNUM(addr)] & (PTE_P |
```

```
PTE_U)) == (PTE_P | PTE_U))){
                        duppage(childenv, PGNUM(addr));
                }
        }
        res = sys_page_alloc(childenv, (void *)(UXSTACKTOP - PGSIZE), PTE_U |
PTE_W | PTE_P);
        if(res < 0){
                panic("Error to allocate exception stack\n");
        }
        res = sys_env_set_pgfault_upcall(childenv, _pgfault_upcall);
        if(res < 0){
                panic("Error in setting pgfault upcall\n");
        }
        res = sys_env_set_status(childenv, ENV_RUNNABLE);
        if(res < 0){
                panic("Error in setting child process state");
        }
        return childenv;

}
```

又是非常长的一个exercise,完成具体的COW fork.这里的pgfault就是一个具体地fault handler,后面我们在用户态也会设定更多自定义的handler.

Exercise 13

```
TRAPHANDLER_NOEC(ENTRY_IRQ_TIMER, IRQ_OFFSET+IRQ_TIMER);
TRAPHANDLER_NOEC(ENTRY_IRQ_KBD, IRQ_OFFSET+IRQ_KBD);
TRAPHANDLER_NOEC(ENTRY_IRQ_2, IRQ_OFFSET+2);
TRAPHANDLER_NOEC(ENTRY_IRQ_3, IRQ_OFFSET+3);
TRAPHANDLER_NOEC(ENTRY_IRQ_SERIAL, IRQ_OFFSET+IRQ_SERIAL);
TRAPHANDLER_NOEC(ENTRY_IRQ_5, IRQ_OFFSET + 5);
TRAPHANDLER_NOEC(ENTRY_IRQ_6, IRQ_OFFSET+6);
TRAPHANDLER_NOEC(ENTRY_IRQ_SPURIOUS, IRQ_SPURIOUS);
TRAPHANDLER_NOEC(ENTRY_IRQ_8, IRQ_OFFSET+8);
TRAPHANDLER_NOEC(ENTRY_IRQ_9, IRQ_OFFSET+9);
TRAPHANDLER_NOEC(ENTRY_IRQ_10, IRQ_OFFSET+10);
TRAPHANDLER_NOEC(ENTRY_IRQ_11, IRQ_OFFSET+11);
TRAPHANDLER_NOEC(ENTRY_IRQ_12, IRQ_OFFSET+12);
TRAPHANDLER_NOEC(ENTRY_IRQ_13, IRQ_OFFSET+13);
TRAPHANDLER_NOEC(ENTRY_IRQ_IDE, IRQ_OFFSET+IRQ_IDE);
TRAPHANDLER_NOEC(ENTRY_IRQ_15, IRQ_OFFSET+15);
TRAPHANDLER_NOEC(ENTRY_IRQ_ERROR, IRQ_OFFSET+IRQ_ERROR);
```

这里又是考验眼力的练习了，手动填写一大堆的entry,没什么技术含量，当然不能忘了IRQ_OFFSET,这在注释里也是提到过的。

Exercise 14

只需要在syscall里添加一个的处理项即可。

Exercise 15

最后要实现的是IPC,第一次接触感觉设计挺奇怪的，又是传值又是传mapping的。 实现的话就是system call端加上user端，各设计一对负责send和recv的函数,代码太长就不放了。

至此make grade应该都能通过了。