Report for lab4, YiChen Mao

Part A: Multiprocessor Support and Cooperative Multitasking

Exercise 1.

Task. 实现 kern/pmap.c 中的函数 mmio_map_region。

```
1 主要任务是把物理地址 [pa,pa+size) 映射到虚拟地址 [MMIOBASE,MMIOBASE+size)
  处，这里 page 可能并非页对齐的，因此首先将 pa 和 page 进行页对齐（注意到映射
  [pa,pa+size)，因此是将 pa+size 上取对齐），由于 base 每次更新都是页对齐的因此若
  base 未对齐一定出现错误。
2 然后由于 IO 地址空间只有 1<<20 字节，判断是否出界，然后用 boot_map_region 建立映
  射，根据提示修改 权限位。
3 最后修正 base ，(如果不修改 base 那么每次分配的空间初始位置都相同无法通过
  check_page)
```

```
并返回映射的起始虚拟地址。

void *
mmio_map_region(physaddr_t pa, size_t size)
{
  static uintptr_t base = MMIOBASE;
  if (PGOFF(base))
    panic("mmio_map_region: base error!");
  size = ROUNDUP(size,PGSIZE);
  if (size > PTSIZE || base + size >= MMIOLIM)
    panic("mmio_map_region: error!");
  boot_map_region(kern_pgdir,base,size,pa,PTE_PCD|PTE_PWT|PTE_W);
  base += size;
  return (void*)(base-size);
  panic("mmio_map_region not implemented");
}
```

Exercise 2.

Task. 阅读 kern/init.c 中的函数 boot_aps() 和 mp_main()，阅读 kern/mpentry.S 中的汇编，并修改 kern/pmap.c 中 page_init 的实现，已达到不把 MPENTRY_PADDR 所对应段加入空闲链表中。

```
mp_main() 应该是获取一些硬件信息和初始化一些结构。(为后面操作做准备)
boot_aps() 是先将 mpentry.S 中汇编代码复制到 MPENTRY_PADDR(0x7000) 并启动其他处理器，其中调用了 mpentry.S 中的代码。
mpentry.S 初始化页表，建立栈等(环境已经在 mp_main 之间建好)

只需要特殊判读一下 MPENTRY_PADDR ，较为简单，具体代码见 kern/pmap.c。
```

Question.

比较 kern/mpentry.S 和 boot/boot.S，为什么 使用 MPBOOTPHYS。

答：我认为在 boot.S 中还没有页面映射，因此

Exercise 3.

Task. 修改 kern/pmap.c 中函数 mem_init_mp() (栈的映射)

```
总共 有 NCPU(8) 个处理器，对于没个处理器，分配 KSTKSIZE (8*pgsize) 作为栈，再做 KSTKGAP (8*pgsize) 作为保护页，
内核栈权限是内核读写以及用户无，因此权限位设置为 PTE_P|PTE_W
static void
mem_init_mp(void)
{
  uintptr_t address = KSTACKTOP - KSTKSIZE;
  for (int i = 0; i < NCPU; i++){

boot_map_region(kern_pgdir,address,KSTKSIZE,PADDR(percpu_kstacks[i]),PTE_W);
    address -= (KSTKSIZE + KSTKGAP);
  }
}
```

Exercise 4.

Task. 修改 kern/trap.c 中的函数 trap_init_percpu 使得它对所有处理器都有效。（提示 不能使用 ts ）

```
初始化 每个 CPU 的任务状态段。
在原先的基础上修改，保证对于每个 CPU 都是有效的。（细节比较多）
1. thiscpu 指向当前 CPU 对应结构体，从结构体可以获取 CPU 的 id （id 也可以用 cpunum() 函数获得）和 ts。
2. 将原先的 ts 替换为 thiscpu->cpu_ts 保证每个 CPU 有独立的 ts。
3. 修改 esp0 时，原先对应栈即 KSTACKTOP，现在不同 CPU 内核栈地址不同，可通过 Exercise 3 简单计算。
4. 修改 gdt 表，不同 CPU 应放入 不同表项中，用 (GD_TSS0 >> 3) + thiscpu->cpu_id 计算。
5. 加载 TSS 选择器。
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
  cprintf("%x\n",thiscpu->cpu_ts);
```

```
  thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - thiscpu->cpu_id * (KSTKSIZE + KSTKGAP);
  thiscpu->cpu_ts.ts_ss0 = GD_KD;
  thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);
  gdt[(GD_TSS0 >> 3) + thiscpu->cpu_id ] = SEG16(STS_T32A, (uint32_t) (&thiscpu->cpu_ts),
            sizeof(struct Taskstate) - 1, 0);
  gdt[(GD_TSS0 >> 3) + thiscpu->cpu_id ].sd_s = 0;
  ltr(GD_TSS0 + thiscpu->cpu_id * 8);
  lidt(&idt_pd);
}
```

Exercise 5.

Task. 通过 *big kernel lock* 保证所有环境至多只有一个在内核模式中运行。适当的加锁和解锁保证实现。

```
1. 在函数 i386_init()，在 BSP 唤醒其他 CPU 前加锁，该操作在 boot_aps 中执行，因此在该语句前加锁。
2. 在函数 mp_main()，在建立环境后，运行环境前加锁，同时加入运行环境语句。
3. 在函数 trap()，由用户态陷入内核态加锁。
4. 在函数 env_run()，这里应该是切换环境后，和在恢复现场前（这应该在用户模式下执行）。
```

Question. 为什么同一个时刻只有一个环境在内核模式下，还要对于不同 CPU 分配不同的内核栈 而不能共享同一个栈。

答：因为虽然用 big kernel lock 保证所有环境中只有一个在内核模式下运行，但是当中断发生的时候，在用户模式下，还未判断是否有锁，硬件部分会将部分寄存器 push 到内核栈中。如果共享栈，可能虽然没有进入内核，但仍然修改了内核栈，导致当前内核程序错误。

Exercise 6.

Task. 轮询调度。

1. 实现 kern/sched.c 中函数 sched_yield() 且满足一下要求 从上一次推出的环境开始找下一个状态为 ENV_RUNNABLE 的环境并调用。
2. 修改 syscall 实现 sys_yield()。
3. 在 mp_main 最后添加 sched_yield()

4. 建立多个环境运行 user/yield.c (注 user_yield)

```
thiscpu->cpu_env 指向当前运行环境（可能不存在，需要特殊处理）

void
sched_yield(void)
{
  struct Env *idle;
  struct Env*current_env = thiscpu->cpu_env;
  size_t id = 0;
  if (current_env != NULL)id = (ENVX(current_env->env_id) + 1) % NENV;
  for (int i = 0; i < NENV; i++,id = (id + 1) % NENV){
    if (envs[id].env_status == ENV_RUNNABLE){
      envs[id].env_cpunum = cpunum();
      env_run(&envs[id]);
      return;
    }
  }
  if (current_env != NULL && current_env->env_status == ENV_RUNNING){
    current_env->env_cpunum = cpunum();
    env_run(current_env);
    return;
  }
  // sched_halt never returns
  sched_halt();
}
```

Question.

3. 在函数 env_run() 中调用 lr3 切换页目录前后为什么 e 指针保持不变。

答：因为 e 是一个指向环境的指针，该结构维护在内核中，因此所有的环境对这段空间的映射相同。

4. 为什么在切换环境的时候要保存旧环境的寄存器等值，且在什么地方 保存。

答：因为如果要恢复环境就需要将寄存器的值恢复，如果不保存则就无法恢复。是在 trapentry.S 的 _alltraps 中，详细可见 lab3。

Exercise 7.

Task. 补充 syscall.c 使完成 fork 操作，通过 user/dumbfork。

```
sys_exofork:创建一个行新的环境，如果成功返回新环境的 ID，失败返回 error_code。
这里处理两种 error_code
E_NO_FREE_ENV 即没有可用的新的环境，在 env_alloc 函数中若无法分配则会返回该 error_code。
E_NO_MEM 通用在 env_alloc 函数中若无物理页分配时返回该 error_code。

对于新建的环境，需要以下部分，
1. env_status 修改为 ENV_NOT_RUNNABLE
2. 寄存器 继承 当前运行环境。
3. 新的环境中返回值为 0，即将寄存器 eax 修改为 0。
static envid_t
sys_exofork(void)
{
  struct Env *e;
  int err = env_alloc(&e,curenv->env_id);
  if (err < 0)return err;
  e->env_status = ENV_NOT_RUNNABLE;
  e->env_tf = curenv->env_tf;
  e->env_tf.tf_regs.reg_eax = 0;
  return e->env_id;
  panic("sys_exofork not implemented");
}
```

```
sys_env_set_status:修改当前环境状态，且只能修改为 ENV_RUNNABLE 或 ENV_NOT_RUNNABLE。
函数 envid2env 将 envid 转换为对应 env 结构体。
需要处理 error_code
E_BAD_ENV
1. 在 envid2env 中若时非法的 env 会返回该 error_code
2. 权限不足，在调用 envid2env 中将第三个参数设置为 1 即会检查权限。
E_INVAL 要求 status 为 ENV_RUNNABLE 或 ENV_NOT_RUNNABLE

修改 env 对应状态即可。
static int
sys_env_set_status(envid_t envid, int status)
{
  if (status != ENV_RUNNABLE && status != ENV_RUNNABLE)
    return -E_INVAL;
```

```
  struct Env *e;
  int err = envid2env(envid,&e,1);
  if (err < 0)return err;
  e->env_status = status;
  return 0;
  panic("sys_env_set_status not implemented");
}
```

为环境分配分配页面
参数限定:
perm 中 PTE_U 和 PTE_P 位为 1 ，且 PTE_AVAIL 和 PTE_W 可以为 0 或 1，其他位必须为 0 。
错误处理:
E_BAD_ENV:判读环境是否合法
E_INVAL:判读虚拟地址 1. 对齐，2.不能超过UTOP 3.权限位合理
E_NO_MEM:能够建立映射（先用 page_alloc 从系统分配物理页，再用 page_insert 建立虚拟地址和物理页之间映射）

```
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 *e;
  int err = envid2env(envid,&e,1);
  if (err < 0)return err;
  if ((uint32_t)va >= UTOP || PGOFF(va))return -E_INVAL;
  if ((perm & ~(PTE_AVAIL|PTE_W))^(PTE_U|PTE_P))
    return -E_INVAL;
  struct PageInfo *page = page_alloc(ALLOC_ZERO);
  if (page == NULL) return -E_NO_MEM;
  err = page_insert(e->env_pgdir,page,va,perm);
  if (err<0){
    page_free(page);
    return -E_NO_MEM;
  }
```

```
  return 0;
  panic("sys_page_alloc not implemented");
}
```

建立环境之间的页面映射关系。
参数限定：
perm ：首先和 page_alloc 要求相同，在基础上还要求对于 只读页 不能加上 可写的参数。

错误处理：
E_BAD_ENV：环境错误，这里有源环境和目的环境，要求两个都合法。
E_INVAL：
1. 源地址和目的地址 对齐且不超过 UTOP
2. 源环境中有对应的地址映射。
3. 权限位正确，包括 （要求对于 只读页 不能加上 可写的参数）。
4. 能够在目的环境中分配物理地址并建立映射。

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
       envid_t dstenvid, void *dstva, int perm)
{
  struct Env*esrc,*edst;
  int errsrc = envid2env(srcenvid,&esrc,1),errdst = envid2env(dstenvid,&edst,1);
  if (errsrc < 0 || errdst < 0)return -E_BAD_ENV;
  if ((uint32_t)srcva >= UTOP || PGOFF(srcva) || (uint32_t)dstva >= UTOP || PGOFF(dstva))return -E_INVAL;
  pte_t* pte;
  struct PageInfo* page = page_lookup(esrc->env_pgdir, srcva, &pte);
  if (page == NULL) return -E_INVAL;
  if ((perm & ~(PTE_AVAIL|PTE_W))^(PTE_U|PTE_P))
    return -E_INVAL;
  if ((perm & PTE_W)&&!(*pte & PTE_W))return -E_INVAL;
  struct PageInfo* pagedst = page_alloc(ALLOC_ZERO);
  if (page == NULL) return -E_NO_MEM;
  int err = page_insert(edst->env_pgdir,page,dstva,perm);
  if (err < 0){
    page_free(pagedst);
    return -E_NO_MEM;
  }
  return 0;
  panic("sys_page_map not implemented");
}
```

```
取消映射
错误处理:
E_BAD_ENV 环境错误。
E_INVAL 地址对齐和 不超过 UTOP

用 page_remove 删除。
static int
sys_page_unmap(envid_t envid, void *va)
{
  struct Env *e;
  int err = envid2env(envid,&e,1);
  if (err < 0)return err;
  pte_t*pte;
  struct PageInfo* page = page_lookup(e->env_pgdir,va,&pte);
  if (pte == NULL || !(*pte & PTE_W))return -E_BAD_ENV;
  if ((uint32_t)va >= UTOP || PGOFF(va)) return -E_INVAL;
  page_remove(e->env_pgdir,va);
  return 0;
  panic("sys_page_unmap not implemented");
}
```

Part B: Copy-on-Write Fork

Exercise 8.

Task. 不同段的 page fault 处理过程并不相同，因此需要用 *page fault handler entrypoint* 记录。实现 sys_env_set_pgfault_upcall 函数。

```
修改 对应的 env_pgfault_upcall 项即可。
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
  struct Env *e;
  int err = envid2env(envid,&e,1);
  if (err < 0)return err;
  e->env_pgfault_upcall = func;
  return 0;
}
```

Exercise 9.

Task. 实现 kern/trap.c 中 page_fault_handler 对于用户程序的 page fault 的 处理。

```
这里 处理 在用户环境中 发生的page_fault
主要流程包括以下几个步骤
1. 找对应处理函数。
2. 在异常栈中分配空间保存现场。
3. 修改环境 eip 和 esp 并运行处理函数。

第一步中，在环境中查找 page fault upcall ，如果有会在 sys_env_set_pgfault_upcall 建立映射。
第二步中，在用户异常栈中分配栈帧用于保存现场，
1. 首先要分配从外部跳入到 page_fault 还是 从 page_fault 处理过程中 再次跳入 page_fault。这里检查原现场的 esp 即可
（应为递归调用是 esp 一定在用户异常栈之间）。
如果从外部跳入，则从异常栈头开始，否则 从上一次 -1 word 开始，（具体为什么要多分配 4 字节空间，是因为在回退的时候需要保存地址，在第 Exercise 10 中就可发现）
2.用 user_mem_assert 去检查分配的栈帧 是否是 用户可读可写的,user_mem_check 主要检查越界和权限两方面。
3.保存现场，这里仿照lab 中的图赋值即可。
第三步中，将指令跳转到处理程序中，栈切换位用户异常栈，并运行。
Hint 中 tf 即当前环境的 env_tf

void
page_fault_handler(struct Trapframe *tf)
{
  uint32_t fault_va;
  fault_va = rcr2();
  if ( (tf->tf_cs&1)!=1 )
    panic("page_fault_handler: kernel page fault!\n");

  // LAB 4: Your code here.
  if (curenv->env_pgfault_upcall!=NULL){
    // cprintf("%x\n",tf->tf_esp);
    struct UTrapframe *utf = (tf->tf_esp >= UXSTACKTOP || tf->tf_esp < UXSTACKTOP - PGSIZE) ?
    (struct UTrapframe *)(UXSTACKTOP -  sizeof(struct UTrapframe)): (struct UTrapframe *)(tf->tf_esp - 4 - sizeof(struct UTrapframe));
    // cprintf("find %x\n",utf);
    user_mem_assert(curenv,(const void*)utf,sizeof(struct UTrapframe),PTE_U|PTE_W|PTE_P);
```

```
    // cprintf("find2\n");
    utf->utf_esp = tf->tf_esp;
    utf->utf_eflags = tf->tf_eflags;
    utf->utf_eip = tf->tf_eip;
    utf->utf_regs = tf->tf_regs;
    utf->utf_err = tf->tf_trapno;
    utf->utf_fault_va = fault_va;

    tf->tf_eip = (uintptr_t)curenv->env_pgfault_upcall;
    // cprintf("eip %x\n",tf->tf_eip);
    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);
}

```

Exercise 10.

Task. 用汇编实现 page_fault handler 回掉程序。

```
这里主要实现从处理函数结束恢复现场的过程。
观察栈帧结构可以知道，UTrapframe 如此设计的妙处，
即先恢复寄存器，再恢复标志寄存器，最后恢复 esp。

主要想法，处理函数的调用实际上是从一个栈跳转到了异常处理栈中并处理过程，在恢复的时候，我们直接从异常处理栈中 ret 不能保证寄存器完整，因此我们先在原栈中构建一个新的小的栈帧，再在原栈中 ret。
1.我们先把 eip 写到原栈的下面，仿佛是在原栈中进行因此内部的过程调用，并保存的 eip。
2.对齐恢复所有的常用寄存器。
3.这里先跳过 eip ，再恢复标志寄存器。
4.最后由于栈帧的最顶部保存的是原现场的 esp，因此用 pop %esp 指令可以得假装过程调用的栈地址。
5.再用 ret 恢复到真实 eip 和 esp
.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

  movl 0x28(%esp),%eax
  subl $0x4, 0x30(%esp)
  movl 0x30(%esp), %ebx
  movl %eax, (%ebx)
  addl $8,%esp
  popal

  addl $4,%esp
  popfl

  mov (%esp),%esp

  ret
```

Exercise 11.

Task. 完善 set_pgfault_handler 函数

```
第一次处理的时候要 建立异常栈 和 将指针连接到 环境处理函数中,
之后修改处理函数只需要修改函数指针即可。

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

  if (_pgfault_handler == 0) {
    envid_t eid = sys_getenvid();
    r = sys_page_alloc(eid, (void *)(UXSTACKTOP - PGSIZE),
PTE_P|PTE_U|PTE_W);
    if (r<0)panic("set_pgfault_handler: sys_page_alloc\n");
    r = sys_env_set_pgfault_upcall(eid,_pgfault_upcall);
    if (r<0)panic("set_pgfault_handler: sys_env_set_pgfault_upcall\n");
```

```
  }

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

测试到过程中还发现到一个 lab3 中的bug

```
lab3 的 bug
应先检查 pte 是否为空，否则对pte 引用会导致内核的 page_fault。
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
  ...
  for (uintptr_t address = start; address < end; address+= PGSIZE){
    if (address >= ULIM){
      user_mem_check_addr = (address>(uintptr_t)va)?address:
(uintptr_t)va;
      return -E_FAULT;
    }
    pte_t *pte = pgdir_walk(env->env_pgdir, (void *)address, 0);
    if (pte == NULL || (*pte & perm) != perm){
      user_mem_check_addr = (address>(uintptr_t)va)?address:
(uintptr_t)va;
      return -E_FAULT;
    }
  }
  return 0;
}
```

Exercise 12.

Task. 完善 fork 函数。

```
从当前环境将 pn 对应页映射到 envid 环境中，
1. pn * pgsize 即 该页的虚拟地址，
2. perm 首先设置为 PTE_U 和 PTE_P，如果 是 可写或写时 复制则同时加上 PTE_COW
3. 映射到 child env 中
4. 若当前环境该页 非 写时复制 则再回映射到 parent env 中。
static int
duppage(envid_t envid, unsigned pn)
```

```
{
  int r;
  // LAB 4: Your code here.envid_t myenvid = sys_getenvid();
  void *va = (void*)(pn * PGSIZE);
  pte_t pte = uvpt[pn];
  envid_t envid_parent = sys_getenvid();
  int perm = PTE_U|PTE_P|(((pte&(PTE_W|PTE_COW))>0)?PTE_COW:0);
  int err;
  err = sys_page_map(envid_parent,va,envid,va,perm);
  if (err < 0)return err;
  if ((perm|~pte)&PTE_COW){
    err = sys_page_map(envid_parent,va,envid_parent,va,perm);
    if (err < 0)return err;
  }
  return 0;
  panic("duppage not implemented");
  return 0;
}
```

```
错误处理函数
1. 首先检查 fault is a write 和 当前页时 PTE_COW
2. 在 PFTEMP 分配一个用户可读可写的页 ,然后复制，并建立映射。
static void
pgfault(struct UTrapframe *utf)
{
  void *addr = (void *) utf->utf_fault_va;
  uint32_t err = utf->utf_err;
  int r;
  if ((err&FEC_WR)==0)
    panic("pgfault: error!\n");
  if ((uvpt[PGNUM(addr)]&PTE_COW)==0)
        panic("pgfault: error!\n");
    envid_t envid = sys_getenvid();
    int err = sys_page_alloc(envid,(void*)PFTEMP,PTE_P|PTE_U|PTE_W);
  if (err<0)panic("pgfault: error!\n");
    addr = ROUNDDOWN(addr,PGSIZE);
    memcpy(PFTEMP,addr,PGSIZE);
    err = sys_page_map(envid,
(void*)PFTEMP,envid,addr,PTE_P|PTE_U|PTE_W);
  if (err<0)panic("pgfault: error!\n");
    err = sys_page_unmap(envid, PFTEMP);
  if (err<0)panic("pgfault: error!\n");
```

```
  return;
  panic("pgfault not implemented");
}
```

```
fork 函数
函数流程
1. parent: 将 page_fault_handle 设置为之前定义的 pgfault().
2. parent: 调用 sys_exofork 建立新的 环境。
3.1 parent: 对于 UTOP 以下的页，调用之前定义的 duppage 将标记为 W 或者 COW 的页映射到子进程的地址空间，标记为 COW ，同时父进程本身也修改为 COW。（先子后父）。异常栈则是需要申请一个新的页。
3.2 child: 准备运行状态
4 parent 修改 child 状态为 runable

envid_t
fork(void)
{
  // LAB 4: Your code here.
  set_pgfault_handler(pgfault);
  envid_t envid = sys_exofork();
  if (envid<0)
    panic("fork : error!\n");
  if (envid==0){
    thisenv = envs+ENVX(sys_getenvid());
    return envid;
  }
  envid_t envid_child = envid;
  envid_t envid_parent = sys_getenvid();;
  for (uintptr_t addr=0;addr<USTACKTOP;addr+=PGSIZE)
        if ((uvpd[PDX(addr)]&PTE_P)&&(uvpt[PGNUM(addr)]&PTE_P))
            duppage(envid_child,PGNUM(addr));
    if (sys_page_alloc(envid_child,(void *)(UXSTACKTOP-PGSIZE),PTE_U|PTE_W|PTE_P)<0)panic("fork : error!\n");
    if (sys_env_set_pgfault_upcall(envid_child,(envs + ENVX(envid_parent))->env_pgfault_upcall)<0)panic("fork : error!\n");
  if (sys_env_set_status(envid_child, ENV_RUNNABLE)<0)panic("fork : error!\n");
    return envid_child;
  panic("fork not implemented");
}
```

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

Exercise 13.

Task. 修改 kern/trapentry.S 和 kern/trap.c 为 IDT 新增关于 IRQ 的表项，修改 kern/env.c 中 函数 env_alloc() 保证用户环境能够中断。

```
类似 lab3 为异常增加表项。
```

Exercise 14.

Task. 实现时钟中断时实现切换进程。

```
这里只处理 时钟中断，lapic_eoi 确认中断，并切换调度。
static void
trap_dispatch(struct Trapframe *tf)
{
  ...
  if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
    cprintf("Spurious interrupt on irq 7\n");
    print_trapframe(tf);
    return;
  }
  switch (tf->tf_trapno){
    case IRQ_OFFSET + IRQ_TIMER: {
      lapic_eoi();
      sched_yield();
      return;
    }
  }
  ...
}
```

Exercise 15.

Task.

```
进程间信息通信发送信息，
错误处理
E_BAD_ENV：环境错误
E_IPC_NOT_RECV：目标进程不允许通信，用 env_ipc_recving 标记
E_INVAL:对于 srcva<UTOP
首先要 页对齐 且权限位正确
发送环境中要有该页面映射
不能对只读页面进行可写权限的映射。
E_NO_MEM：缺乏物理页

否则 修改 ipc_recving，ipc_from，ipc_value，ipc_perm，status 为对应的值。

static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned
perm)
{
  // LAB 4: Your code here.
  envid_t src_envid = sys_getenvid();
    struct Env *e;
  int err;
    err = envid2env(envid,&e,0);
  // cprintf("err %x\n",err);
  if (err<0)return err;
    if (!e->env_ipc_recving)return -E_IPC_NOT_RECV;
  if ((uint32_t)srcva < UTOP && PGOFF(srcva))return -E_INVAL;
  if ((perm & ~(PTE_AVAIL|PTE_W))^(PTE_U|PTE_P))return -E_INVAL;
  pte_t *pte;
  struct PageInfo *page = page_lookup(curenv->env_pgdir,srcva,&pte);
  if ((uint32_t)srcva < UTOP && page == NULL)return -E_INVAL;
  if ((uint32_t)srcva < UTOP && (perm & PTE_W) && (~*pte &
PTE_W))return -E_INVAL;
  if ((uint32_t)srcva < UTOP){
    err = page_insert(e->env_pgdir,page,e->env_ipc_dstva,perm);
    if (err<0)return err;
  }
// cprintf("find\n");
  e->env_ipc_recving = false;
  e->env_ipc_from = src_envid;
```

```
  e->env_ipc_value = value;
  e->env_ipc_perm = ((uint32_t)srcva < UTOP)?perm:0;
  e->env_status = ENV_RUNNABLE;

  e->env_tf.tf_regs.reg_eax = 0;
  return 0;
  panic("sys_ipc_try_send not implemented");
}
```

```
进程接受信息，如果 dstva < UTOP 说明要页映射，因此要页对齐。
将 ipc_recving 修改为true，修改dstva，并将进程设置成 不能再被调用。
static int
sys_ipc_recv(void *dstva)
{
  // LAB 4: Your code here.
  if ((uintptr_t)dstva<UTOP&&PGOFF(dstva)!=0)return -E_INVAL;
  envid_t envid = sys_getenvid();
  struct Env *e;
  e->env_ipc_recving = true;
  e->env_ipc_dstva = dstva;
  e->env_status = ENV_NOT_RUNNABLE;
  sys_yield();
  return 0;
}
```

```
若 pg 为 NULL 则修改为 UTOP 表示不进行页面传递。
若接收成功则保存原环境和权限，并返回value 否则清空，返回 error_code。

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

  if (pg==NULL)pg=(void *)UTOP;
  // cprintf("now %x\n",pg);
  int err = sys_ipc_recv(pg);
  // cprintf("%x\n",err);
  if (err < 0){
    if (from_env_store != NULL)*from_env_store=0;
    if (perm_store != NULL)*perm_store=0;
```

```
    return err;
  }else {
    if (from_env_store != NULL) *from_env_store = thisenv->env_ipc_from;
    if (perm_store != NULL) *perm_store = thisenv->env_ipc_perm;
    return thisenv->env_ipc_value;
    }
  panic("ipc_recv not implemented");
  return 0;
}
```

```
一直发送，直至成功或出错。
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
  // LAB 4: Your code here.
  if (pg==NULL)pg=(void *)UTOP;
  int err;
  while (true){
    err = sys_ipc_try_send(to_env,val,pg,perm);
    // cprintf("%x\n",err);
    if (err == -E_IPC_NOT_RECV)sys_yield();
    else if (err < 0)panic("ipc send: error!\n");
    else return;
    }
  return;
  panic("ipc_send not implemented");
}
```

Challenge

Challenge! Implement a shared-memory `fork()` called `sfork()`. This version should have the parent and child *share* all their memory pages (so writes in one environment appear in the other) except for pages in the stack area, which should be treated in the usual copy-on-write manner. Modify `user/forktree.c` to use `sfork()` instead of regular `fork()`. Also, once you have finished implementing IPC in part C, use your `sfork()` to run `user/pingpongs`. You will have to find a new way to provide the functionality of the global `thisenv` pointer.

Task. 实现 sfork 使得对于所有用户空间 除栈外 共享。

难点，thisenv 因为 thisenv 位于用户空间中，父子进程共享用户空间，因此父子进程的 thisenv 会相同（即如果再子进程修改 thisenv，那么父进程也会被修改导致错误。）
考虑到用户栈是不会被共享的，因此将 thisenv 保存在用户栈中。

原先 thisenv 为全局变量指向当前 env 的指针，现修改为 pthisenv 为指向 当前 env 的指针的指针。
在函数局部定义 local_thisenv 为指向当前 env 的指针，由于在局部定义，因此位于用户栈中。
将 pthisenv 指向 local_thisenv，并将 thisenv define 为 (*pthisenv) （该操作只是为了保证最小化修改，只需要在库文件中加入该宏定义，否则对于原代码中使用到 thisenv 的地方修改也可）

```
#include <inc/lib.h>
extern void umain(int argc, char **argv);
const char *binaryname = "<unknown>";
#define thisenv (*pthisenv)
const volatile struct Env **pthisenv;

void
libmain(int argc, char **argv)
{
  const volatile struct Env *local_thisenv = (envs +
ENVX(sys_getenvid()));
  pthisenv = &local_thisenv;
  if (argc > 0)
    binaryname = argv[0];
  umain(argc, argv);
  exit();
}
```

类似于 fork，在映射过程中有所修改，对于除栈外空间用 sys_page_map 从父进程映射到子进程，对于 用户栈 还是用 COW 方式。

```
int
sfork(void)
{
  set_pgfault_handler(pgfault);
  envid_t envid = sys_exofork();
  cprintf("envid :%x\n",envid);
  if (envid<0)
    panic("sfork : error!\n");
```

```
  if (envid==0){
    thisenv = envs+ENVX(sys_getenvid());
    return envid;
  }
  envid_t envid_child = envid;
  envid_t envid_parent = sys_getenvid();;
  for (uintptr_t addr=UTEXT;addr<USTACKTOP - PGSIZE;addr+=PGSIZE)
    if ((uvpd[PDX(addr)]&PTE_P)&&(uvpt[PGNUM(addr)]&PTE_P)){
        // if ((uvpd[PDX(addr)]&PTE_P)&&(uvpt[PGNUM(addr)]&PTE_P))
        //    duppage(envid_child,PGNUM(addr));
      sys_page_map(envid_parent,(void*)addr,envid_child,
(void*)addr,PTE_U|PTE_P|PTE_W);
    }
  duppage(envid_child,PGNUM(USTACKTOP - PGSIZE));
   if (sys_page_alloc(envid_child,(void *)(UXSTACKTOP-
PGSIZE),PTE_U|PTE_W|PTE_P)<0)panic("sfork : error!\n");
   if (sys_env_set_pgfault_upcall(envid_child,(envs +
ENVX(envid_parent))->env_pgfault_upcall)<0)panic("sfork : error!\n");
  if (sys_env_set_status(envid_child, ENV_RUNNABLE)<0)panic("sfork :
error!\n");
    return envid_child;

  panic("sfork not implemented");
  return -E_INVAL;
}
```

测试：

```
0
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
eebfdfe4
[00001000] new env 00001001
envid :1001
i am 00001000; thisenv is 0xeec00000
envid :0
send 0 from 1000 to 1001
1001 got 0 from 1000 (thisenv is 0xeec0007c 1001)
1000 got 1 from 1001 (thisenv is 0xeec00000 1000)
1001 got 2 from 1000 (thisenv is 0xeec0007c 1001)
1000 got 3 from 1001 (thisenv is 0xeec00000 1000)
1001 got 4 from 1000 (thisenv is 0xeec0007c 1001)
1000 got 5 from 1001 (thisenv is 0xeec00000 1000)
1001 got 6 from 1000 (thisenv is 0xeec0007c 1001)
1000 got 7 from 1001 (thisenv is 0xeec00000 1000)
1001 got 8 from 1000 (thisenv is 0xeec0007c 1001)
1000 got 9 from 1001 (thisenv is 0xeec00000 1000)
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000 (thisenv is 0xeec0007c 1001)
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
```

运行 pingpongs.c 的结果，可以发现 val 在父进程和子进程之间得到共享。