Lab 4: Preemptive Multitasking

December 24, 2019

摘要

本次实验取自 MIT 6.828 Lab4，实现在同时运行的多个用户进程中实现抢占式多任务处理，实验共分为三个部分。

在第一部分，为 JOS 系统添加多处理器支持，实现轮转调度，并在系统调用中添加一些基础进程管理方法（例如，创建、销毁进程，以及分配和映射内存）。

在第二部分，实现类 Unix 的 fork() 方法，以允许用户进程创造自身的拷贝。

在第三部分，为 JOS 提供进程间通信支持，允许不同的用户进程显式地彼此交流和同步，并实现硬件时钟中断和抢占。

关键词：JOS、抢占式多任务处理、轮转调度、系统调用、进程间通信

1 作业一

修改你在 kern/pmap.c 中实现过的 page_init() 以避免将 MPENTRY_PADDR 加入到 free list 中，以使得我们可以安全地将 AP 的引导代码拷贝于这个物理地址并运行。

```
1 pages[0].pp_ref = 1;
2 size_t size_ioh = 96;
3 size_t size_ext = ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE
    ;
4 size_t i, j = npages_basemem + size_ioh + size_ext, mpp =
    MPENTRY_PADDR / PGSIZE;
```

```
for (i = 1; i < npages; i++) {
    if (i == mpp || (i >= npages_basemem && i < j)) {
        //在不可用区域里加上MPENTRY_PADDR所在处
        pages[i].pp_ref = 1;//标记为不可用
        continue;
    }
    pages[i].pp_ref = 0;//标记为可用
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

2 问题一

逐行比较 kern/mpentry.S 和 boot/boot.S。牢记 kern/mpentry.S 和其他内核代码一样也是被编译和链接在 KERNBASE 之上运行的。那么，MPBOOTPHYS 这个宏定义的目的是什么呢？为什么它在 kern/mpentry.S 中是必要的，但在 boot/boot.S 却不用？

因为 mpentry.S 的代码都在 kernbase 上，所以实模式是没办法直接寻址的，这时候需要 MPBOOTPHYS 起一个地址转换的作用，而 boot.S 就被加载在实模式可寻址的低地址，所以不需要地址转换。

3 作业二

修改位于 kern/pmap.c 中的 mem_init_mp()，将每个 CPU 堆栈映射在 KSTACKTOP 开始的区域，就像 inc/memlayout.h 中描述的那样。每个堆栈的大小都是 KSTKSIZE 字节，加上 KSTKGAP 字节没有被映射的守护页。

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

```
    //参数：目录，虚拟起始地址，大小，物理地址，权限
}
```

对于 NCPU 个栈区域，从 KSTACKTOP 开始向低地址延伸，每个区域大小为 KSTLSIZE，栈区域之间有 KSTKGAP 大小的间隔，依次标记好位置后初始化。用于遍历的临时变量从栈顶开始向下挪移，每次挪移堆栈大小 + 堆栈间隔）的长度。每次挪移对经过的堆栈进行映射。

4 作业三

需要做的工作是修改 trap_init_percpu() 函数使得它可以完成对每个 CPU 的初始化，具体来说就是将全局变量改成每个 CPU 的变量，新旧代码对比如下：

```
void
trap_init_percpu(void) {
    int cpu_id = thiscpu->cpu_id;
    struct Taskstate *this_ts = &thiscpu->cpu_ts;

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

    /*ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;*/

    this_ts->ts_esp0 = KSTACKTOP - cpu_id * (KSTKSIZE + KSTKGAP
        );
    this_ts->ts_ss0 = GD_KD;

    // Initialize the TSS slot of the gdt.

    /*gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
```

```
        sizeof(struct Taskstate) - 1, 0);
18      gdt[GD_TSS0 >> 3].sd_s = 0;*/
19
20      gdt[(GD_TSS0 >> 3) + cpu_id] = SEG16(STS_T32A, (uint32_t) (
            this_ts),sizeof(struct Taskstate) - 1, 0);
21      gdt[(GD_TSS0 >> 3) + cpu_id].sd_s = 0;
22
23      // Load the TSS selector (like other segment selectors, the
24      // bottom three bits are special; we leave them 0)
25
26      //ltr(GD_TSS0);
27      ltr(GD_TSS0 + (cpu_id << 3));
28
29      // Load the IDT
30      lidt(&idt_pd);
31  }
```

5 作业四

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

大内核锁的实现：

```
1  void
2  spin_lock(struct spinlock* lk)
3  {
4  #ifdef DEBUG_SPINLOCK
5      if (holding(lk))
6          panic("CPU %d cannot acquire %s: already holding",
               cpunum(), lk->name);
7  #endif
```

```
8
9       // The xchg is atomic.
10      // It also serializes, so that reads after acquire are not
11      // reordered before it.
12      // 关键代码，体现了循环等待的思想
13      while (xchg(&lk->locked, 1) != 0)
14          asm volatile ("pause");
15
16      // Record info about lock acquisition for debugging.
17  #ifdef DEBUG_SPINLOCK
18      lk->cpu = thiscpu;
19      get_caller_pcs(lk->pcs);
20  #endif
21  }
```

其中，在 inc/x86.h 中可以找到 xchg() 函数的实现，使用它而不是用简单的 if 和赋值是因为它是一个原子性的操作。

```
1   static inline uint32_t
2   xchg(volatile uint32_t* addr, uint32_t newval)
3   {
4       uint32_t result;
5
6       // The + in "+m" denotes a read-modify-write operand.
7       asm volatile("lock; xchgl %0, %1"
8           : "+m" (*addr), "=a" (result) // 输出
9           : "1" (newval)                // 输入
10          : "cc");
11      return result;
12  }
```

这是一段内联汇编。lock 确保了操作的原子性，其意义是将 addr 存储的值与 newval 交换，并返回 addr 中原本的值。于是，如果最初 locked = 0，即未加锁，就能跳出这个 while 循环。否则就会利用 pause 命令自旋等待。确保了当一个 CPU 获得了 BKL，其他 CPU 如果要获得就只能自旋等待。

在这几处加大内核锁为了避免多个 CPU 同时运行内核代码，保证独立性。由于分页机制的存在，内核以及每个用户进程都有自己的独立空间。而多进程并发的时候，如果两个进程同时陷入内核态，就无法保证独立性了。例如内核中有某个全局变量 A，cpu1 让 A=1，而后 cpu2 却让 A=2，显然会互相影响。为了使系统尽快支持 SMP，直接在内核入口加大内核锁锁，保证其独立性。

其流程大致为：

BPS 启动 AP 前，获取内核锁，所以 AP 会在 mp_main 执行调度之前阻塞，在启动完 AP 后，BPS 执行调度，运行第一个进程，env_run() 函数中会释放内核锁，这样一来，其中一个 AP 就可以开始执行调度，运行其他进程。

```
// i386_init()
// Your code here:
lock_kernel();
boot_aps();
```

在唤醒其他 CPU 前需要 lock ，防止唤醒的 CPU 启动进程。

```
// mp_main()
// Your code here:
lock_kernel();
sched_yield();
```

初始化 AP 后，在调度之前需要 lock，防止其他 CPU 干扰进程的选择。

```
// trap()
// LAB 4: Your code here.
lock_kernel();
```

```
4 assert(curenv);
```

用户态引发中断陷入内核态时，需要 lock。

```
1 // env_run()
2 lcr3(PADDR(e->env_pgdir));
3 unlock_kernel();
4 env_pop_tf(&(e->env_tf));
```

离开内核态之前，需要 unlock 。

BPS 启动 AP 前，获取内核锁，所以 AP 会在 mp_main 执行调度之前阻塞，在启动完 AP 后，BPS 执行调度，运行第一个进程，之后释放内核锁，这样一来，其中一个 AP 就可以开始执行调度，若有的话运行进程。

6 问题二

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.

如果内核栈中留下不同 CPU 之后需要使用的数据，可能会造成混乱。

例如在某进程即将陷入内核态的时候（尚未获得锁），系统在 trap() 函数之前已经在 trapentry.S 中对内核栈进行了操作，压入了寄存器信息。如果共用一个内核栈，那显然会导致信息错误。

7 作业五

按照题目的要求在 sched_yield() 函数实现轮转调度。具体实现方法是如果存在上一个 running environment，就从它开始，否则从头开始搜索 envs 数组，找到一个 runnable 的 environment 并进入。如果没有其他 environment 而之前有一个正在运行的则继续执行它。其他情况执行 cpuhalt 代码如下：

```
void
sched_yield(void) {
    struct Env *idle;
    int i, k, envidx;

    if (curenv)
        envidx = ENVX(curenv->env_id);
    else
        envidx = 0;

    for (i = 0; i < NENV; ++i) {
        k = (envidx + i) % NENV;
        if (envs[k].env_status == ENV_RUNNABLE)
            env_run(&envs[k]);
    }
    if (curenv && curenv->env_status == ENV_RUNNING)
        env_run(curenv);

    // sched_halt never returns
    sched_halt();
}
```

还有，在 syscall.c 里补充相应的系统调用：

```
static void
sys_yield(void) {
   sched_yield();
}

int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t
   a3, uint32_t a4, uint32_t a5) {
```

```
8       ......
9       case SYS_yield:
10          sys_yield();
11          return 0;
12      ......
13  }
```

8 问题三

在你实现的 env_run() 中你应当调用了 lcr3()。在调用 lcr3() 之前和之后，你的代码应当都在引用变量 e，就是 env_run() 所需要的参数。在装载%cr3 寄存器之后，MMU 使用的地址上下文立刻发生改变，但是处在之前地址上下文的虚拟地址（比如说 e）却还能够正常工作，为什么 e 在地址切换前后都可以被正确地解引用呢?

在 env.c 的 env_settup_vm() 中:

```
1  e->env_pgdir = (pde_t *)page2kva(p);
2  p->pp_ref += 1;
3  memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
4
5  // UVPT maps the env's own page table read-only.
6  // Permissions: kernel R, user R
7  e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
8
9  return 0;
```

在 env_setup_vm() 中，注释说所有环境的虚拟地址空间从 UTOP 到 UVPT 都是相同的，内核的地址空间也是相同的。无论地址空间是什么，e 的虚拟地址总是相同的。例如上述代码直接以内核的页目录作为模版进行复制，两个页目录的 e 地址映射到同一物理地址，e 在地址切换前后都可以被正确地解引用。

9 作业六

作业 6 要求在 kern/syscall.c 中实现用于创建进程的系统调用，使得用户进程也可以创建和启动其他新的用户进程。其中系统调用包括 sys_exofork、sys_env_set_status、sys_page_alloc()、sys_page_map() 和 sys_page_unmap()。具体实现过程如下：

user/dumbfork.c 中提供的一种 Unix 样式的 fork()，其 duppage() 函数如下：

```
1  void
2  duppage(envid_t dstenv, void *addr) {
3  int r;
4
5  // This is NOT what you should do in your fork.
6  if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0)
7      panic("sys_page_alloc: %e", r);
8
9  if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W
       )) < 0)
10     panic("sys_page_map: %e", r);
11
12 memmove(UTEMP, addr, PGSIZE);
13 if ((r = sys_page_unmap(0, UTEMP)) < 0)
14     panic("sys_page_unmap: %e", r);
15 }
```

dumbfork() 首先通过 sys_exofork() 系统调用创建一个新的空白进程。

然后通过 duppage 拷贝父进程的地址空间到子进程中。用户进程地址空间开始位置是 UTEXT（0x00800000），结束位置是 end。其中 duppage 是一页页拷贝的，它将父进程的 addr 开始的一页物理内存内容拷贝到子进程 dstenv 的对应的页中。

最后完成父进程到子进程内存数据的拷贝。先通过 sys_page_alloc 为子进程 addr 开始的一页内容分配一个物理页并完成映射，此时，分配的物理页还是空的，没有数据。然后通过 sys_page_map 将子进程 va 开始的这

分配好的物理页映射到父进程的 UTEMP 地址处（0x00400000），使得在父进程中可以访问到子进程新分配的物理页。接下来，通过 memmove 函数将父进程 addr 处的一页数据拷贝到了 UTEMP 中，即将父进程的 addr 处的一页内存数据拷贝到子进程的 addr 对应的那页内存完成数据的复制（UTEMP 已经映射到了子进程的那页内存）。最后通过 sys_page_unmap 取消父进程在 UTEMP 的映射以下次使用，同时预防父进程误操作到子进程的内存数据。

9.1 sys_exofork()

sys_exofork() 创建一个用户地址空间没有内存映射，也不可以运行，几乎完全空白的新进程。

使用 env_alloc() 创建新环境。将状态设置为 ENV_NOT_RUNNABLE，从当前环境中复制寄存器集，进行调整后使 sys_exofork 看起来返回 0。如果没有可用的空闲环境，返回-E_NO_FREE_ENV，内存耗尽返回-E_NO_MEM。

```
1  static envid_t
2  sys_exofork(void) {
3      // LAB 4: Your code here.
4      // panic("sys_exofork not implemented");
5      struct Env *e;
6      int err = env_alloc(&e, curenv->env_id);
7      if (err < 0)
8          return err;
9
10     e->env_status = ENV_NOT_RUNNABLE;
11     e->env_tf = curenv->env_tf;
12     e->env_tf.tf_regs.reg_eax = 0;
13
14     return e->env_id;
15 }
```

9.2 sys_env_set_status()

sys_env_set_status() 可以将一个进程的状态设置为 ENV_RUNNABLE 或 ENV_NOT_RUNNABLE。通常用来在新创建的进程的地址空间和寄存器状态已经初始化完毕后将它标记为就绪状态。

使用 kern/ Env. c 中的'envid2env' 函数将一个 envid 转换成一个结构 Env。并将 envid2env 的第三个参数设置为 1，用于检查当前环境是否具有设置 envid 状态的权限。成功返回 0，错误返回 < 0。-E_BAD_ENV 代表环境 envid 当前不存在，或者调用者没有更改 envid 的权限返回。-E_INVAL 代表状态不是环境的有效状态。

```
1  static int
2  sys_env_set_status(envid_t envid, int status) {
3      // LAB 4: Your code here.
4      // panic("sys_env_set_status not implemented");
5      struct Env *e;
6      int err;
7      if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
8          return -E_INVAL;
9      if ((err=envid2env(envid, &e, 1) )!=0)
10         return err;
11
12     e->env_status = status;
13     return 0;
14 }
```

9.3 sys_page_alloc()

sys_page_alloc 用于分配一个物理内存页面，并将它映射在给定进程虚拟地址空间的给定虚拟地址上。

首先在'envid' 的地址空间中，使用'perm' 权限在'va' 上分配内存页并将其映射到'va', 页面的内容被设置为 0。如果一个页面已经在'va' 被映射，那么该页面将作为一个副作用被取消映射。必须设置 PTE_U | PTE_P, 而

PTE_AVAIL | PTE_W 可以设置也可以不设置，但是不能设置其他的位。在分配物理页之后通过 page_insert() 将分配的物理页映射到虚拟地址 va, 映射失败时要对分配的页面进行释放。返回信息中，-E_BAD_ENV 代表环境 envid 当前不存在，或者调用者没有更改 envid 的权限。-E_INVAL 代表 va >= UTOP、va 没有页面对齐或 perm 不合适。-E_NO_MEM 如果没有内存来分配新页或分配任何必要的页表。

```
static int
sys_page_alloc(envid_t envid, void *va, int perm) {
    // LAB 4: Your code here.
    // panic("sys_page_alloc not implemented");
    struct Env *e;
    struct PageInfo *pginfo;
    int err ;

    if ((~perm & (PTE_U|PTE_P)) != 0)
        return -E_INVAL;
    if ((perm & (~(PTE_U|PTE_P|PTE_AVAIL|PTE_W))) != 0)
        return -E_INVAL;
    if ((uintptr_t)va >= UTOP || PGOFF(va) != 0)
        return -E_INVAL;

    pginfo = page_alloc(ALLOC_ZERO);
    if (!pginfo)
        return -E_NO_MEM;

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

    err = page_insert(e->env_pgdir, pginfo, va, perm);
    if (err < 0) {
        page_free(pginfo);
```

```
27          return -E_NO_MEM;
28      }
29      return 0;
30  }
```

9.4 sys_page_map()

sys_page_map() 从一个进程的地址空间拷贝一个页的映射到另一个进程的地址空间，即建立跨进程的映射。

将 “srcva” 的内存页映射到 dstenvid 在 dstenvid 的地址空间的 “dstva” 的地址空间并设置 “perm”。Perm 与 sys_page_alloc 有相同的限制，但它也不能授予写入只读页的访问使用 dstenvid 访问 page_lookup() 来检查页面上的当前权限。

```
1   static int
2   sys_page_map(envid_t srcenvid, void *srcva,
3       envid_t dstenvid, void *dstva, int perm) {
4       struct Env *srcenv, *dstenv;
5       struct PageInfo *pp;
6       pte_t *pte;
7       int err;
8
9       if ((uint32_t)srcva >= UTOP || PGOFF(srcva) != 0)
10          return -E_INVAL;
11      if ((uint32_t)dstva >= UTOP || PGOFF(dstva) != 0)
12          return -E_INVAL;
13      if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P))
14          return -E_INVAL;
15      if ((perm & ~(PTE_SYSCALL)) != 0)
16          return -E_INVAL;
17
18      if ((err = envid2env(srcenvid, &srcenv, 1)) != 0)
```

```
19          return err;
20      if ((err = envid2env(dstenvid, &dstenv, 1)) != 0)
21          return err;
22
23      if ((pp = page_lookup(srcenv->env_pgdir, srcva, &pte)) ==
            NULL)
24          return -E_INVAL;
25      if ((*pte & PTE_W) == 0 && (perm & PTE_W) == PTE_W)
26          return -E_INVAL;
27
28      if ((err = page_insert(dstenv->env_pgdir, pp, dstva, perm))
            != 0)
29          return err;
30
31      return 0;
32  }
```

9.5 sys_page_unmap()

sys_page_unmap() 取消给定进程在给定虚拟地址的页映射。

通过进程 ID 得到进程 env 对象可以通过函数 kern/env.c 中的 envidenv() 实现。在'envid' 的地址空间中取消'va' 处内存页的映射。如果没有映射任何页面，默认返回 0。

```
1  static int
2  sys_page_unmap(envid_t envid, void *va) {
3      // LAB 4: Your code here.
4      // panic("sys_page_unmap not implemented");
5      struct Env *e;
6
7      if ((uintptr_t)va >= UTOP || PGOFF(va) != 0)
8          return -E_INVAL;
```

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

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

在 kern/syscall.c 中添加新的系统调用类型如下：

```
    ...
case SYS_exofork:
    return sys_exofork();
case SYS_env_set_status:
    return sys_env_set_status(a1, a2);
case SYS_page_alloc:
    return sys_page_alloc(a1, (void *)a2, a3);
case SYS_page_map:
    return sys_page_map(a1, (void *)a2, a3, (void *)a4, a5);
case SYS_page_unmap:
        return sys_page_unmap(a1, (void *)a2);
    ...
```

10 作业七

作业七要求实现 sys_env_set_pgfault_upcall 系统调用，用于为缺页处理函数入口点注册它的缺页处理入口

通过修改相应的 struct Env 的'env_pgfault_upcall' 字段，为'envid' 设置页面错误 upcall。当'envid' 导致页面错误时，内核会将错误记录推送到异常堆栈，然后转移到'func'。成功时返回 0，错误时即如果环境 envid 当前不存在，或者调用者没有更改 envid 的权限返回 <0, 即-E_BAD_ENV。

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

    e->env_pgfault_upcall=func;
    return 0;
}
```

之后在 syscall.c 中添加新的系统调用类型

```
case SYS_env_set_pgfault_upcall:
    return sys_env_set_pgfault_upcall(a1, (void *)a2);
```

11 作业八

变量 faultva 是导致这个缺页错误的虚拟地址。如果用户程序在发生缺页错误时已经运行在异常堆栈上，那么可以知道缺页处理函数发生了缺页错误。在这样的情况下，就需要在当前的 tf- >tfesp 之下而不是 UXSTACKTOP 这里设置新的堆栈。需要首先压入一个 32 位空字，然后是一个 UTrapframe 结构。要检验 tf- >tf_sep 是否已经在用户的异常堆栈上，检查它是否是在 UXSTACKTOP-PGSIZE 到 UXSTACKTOP-1 之间的区域即可。

```
if (curenv->env_pgfault_upcall != NULL) {
    uintptr_t esp;

```

```
4       if (tf->tf_esp > UXSTACKTOP - PGSIZE && tf->tf_esp <
            UXSTACKTOP) {
5           esp = tf->tf_esp - 4 - sizeof(struct UTrapframe);
6       } else {
7           esp = UXSTACKTOP - sizeof(struct UTrapframe);
8       }
9
10      user_mem_assert(curenv, (void *) esp, sizeof(struct
            UTrapframe), PTE_W | PTE_U | PTE_P);
11
12      struct UTrapframe *utf = (struct UTrapframe *) (esp);
13      utf->utf_fault_va = fault_va;
14      utf->utf_err = tf->tf_err;
15      utf->utf_regs = tf->tf_regs;
16      utf->utf_eip = tf->tf_eip;
17      utf->utf_eflags = tf->tf_eflags;
18      utf->utf_esp = tf->tf_esp;
19
20      tf->tf_esp = esp;
21      tf->tf_eip = (uintptr_t) curenv->env_pgfault_upcall;
22      env_run(curenv);
23  }
```

向用户异常栈中压入 UTrapframe，需要判断可能发生多次异常，这种情况下需要在之前的栈顶后先留下一个空位，再压入 UTrapframe，之后会用到这个空位，然后设置 esp 和 eip 并调用 env_run() 。

12 作业九

Finish set_pgfault_handler() in lib/pgfault.c.

```
1   void
```

```
2  set_pgfault_handler(void (*handler)(struct UTrapframe *utf)) {
3      int r;
4
5      if (_pgfault_handler == 0) {
6          if (sys_page_alloc(0, (void*)(UXSTACKTOP-PGSIZE), PTE_W
               |PTE_U|PTE_P) < 0)
7              panic("set_pgfault_handler:sys_page_alloc failed")
                   ;;
8      }
9      // Save handler pointer for assembly to call.
10     _pgfault_handler = handler;
11     if (sys_env_set_pgfault_upcall(0, _pgfault_upcall) < 0)
12         panic("set_pgfault_handler:sys_env_set_pgfault_upcall
               failed");
13 }
```

13 作业十

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.

在 pgfault 函数中先判断是否页错误是由写时拷贝造成的，如果不是则 panic。借用了一个一定不会被用到的位置 PFTEMP，专门用来发生 page fault 的时候拷贝内容用的。先解除 addr 原先的页映射关系，然后将 addr 映射到 PFTEMP 映射的页，最后解除 PFTEMP 的页映射关系。

首先是 pgfault 处理 page fault 时的写时拷贝。

在 pgfault 函数中先判断是否页错误是由写时拷贝造成的，如果不是则 panic。借用了一个一定不会被用到的位置 PFTEMP，专门用来发生 page fault 的时候拷贝内容用的。先解除 addr 原先的页映射关系，然后将 addr 映射到 PFTEMP 映射的页，最后解除 PFTEMP 的页映射关系。

```
1  static void
2  pgfault(struct UTrapframe *utf) {
3      void *addr = (void *) utf->utf_fault_va;
4      uint32_t err = utf->utf_err;
5      int r;
6
7      // Check that the faulting access was (1) a write, and (2)
           to a
8      // copy-on-write page.  If not, panic.
9      // Hint:
10     //   Use the read-only page table mappings at uvpt
11     //   (see <inc/memlayout.h>).
12
13     // LAB 4: Your code here.
14
15     if (!(
16             (err & FEC_WR) && (uvpd[PDX(addr)] & PTE_P) &&
17             (uvpt[PGNUM(addr)] & PTE_P) && (uvpt[PGNUM(addr)] &
                   PTE_COW)
18         ))
19         panic("pgfault: faulting access is either not a write
               or not to a COW page");
20
21     // Allocate a new page, map it at a temporary location (
           PFTEMP),
22     // copy the data from the old page to the new page, then
           move the new
23     // page to the old page's address.
24     // Hint:
25     //   You should make three system calls.
26
27     addr = ROUNDDOWN(addr, PGSIZE);
```

```
28      if ((r = sys_page_alloc(0, PFTEMP, PTE_P | PTE_U | PTE_W))
            < 0)
29          panic("pgfault: %e", r);
30      memcpy(PFTEMP, addr, PGSIZE);
31      if ((r = sys_page_map(0, PFTEMP, 0, addr, PTE_P | PTE_U |
            PTE_W)) < 0)
32          panic("pgfault: %e", r);
33      if ((r = sys_page_unmap(0, PFTEMP)) < 0)
34          panic("pgfault: %e", r);
35
36      return;
37
38      panic("pgfault not implemented");
39  }
```

紧接着，接下来是 duppage 函数，负责进行 COW 方式的页复制，将当前进程的第 pn 页对应的物理页的映射到 envid 的第 pn 页上去，同时将这一页都标记为 COW。

```
1   static int
2   duppage(envid_t envid, unsigned pn) {
3       int r;
4       // LAB 4: Your code here.
5       void *addr = (void *) (pn*PGSIZE);
6
7       if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)) {
8           if ((r = sys_page_map(0, addr, envid, addr, PTE_P |
                PTE_U | PTE_COW)) < 0)
9               panic("duppage: %e", r);
10          if ((r = sys_page_map(0, addr, 0, addr, PTE_P | PTE_U |
                PTE_COW)) < 0)
11              panic("duppage: %e", r);
```

```
12      } else if ((r = sys_page_map(0, addr, envid, addr, PTE_P |
            PTE_U)) < 0)
13          panic("duppage: %e", r);
14
15      // panic("duppage not implemented");
16      return 0;
17  }
```

最后是 fork 函数，将页映射拷贝过去，这里需要考虑的地址范围就是从 UTEXT 到 UXSTACKTOP 为止，而在此之上的范围因为都是相同的，在 env_alloc 的时候已经设置好了。

首先需要为父进程设定错误处理例程。这里调用 set_pgfault_handler 函数是因为当前并不知道父进程是否已经建立了异常栈，没有的话就会建立一个，而 sys_env_set_pgfault_upcall 则不会建立异常栈。

调用 sys_exofork 准备出一个和父进程状态相同的子进程，状态暂时设置为 ENV_NOT_RUNNABLE。然后进行拷贝映射的部分，在当前进程的页表中所有标记为 PTE_P 的页的映射都需要拷贝到子进程空间中去。但是有一个例外，是必须要新申请一页来拷贝内容的，就是用户异常栈。因为 copy-on-write 就是依靠用户异常栈实现的，所以说这个栈要在 fork 完成的时候每个进程都有一个，要硬拷贝过来。

主要流程就是：

1、申请新的物理页，映射到子进程的 (UXSTACKTOP-PGSIZE) 位置上去。

2、父进程的 PFTEMP 位置也映射到子进程新申请的物理页上去，这样父进程也可以访问这一页。

3、在父进程空间中，将用户错误栈全部拷贝到子进程的错误栈上去，也就是刚刚申请的那一页。

4、然后父进程解除对 PFTEMP 的映射。

5、最后把子进程的状态设置为可运行。

```
1  envid_t
2  fork(void) {
```

```
3       int r;
4       envid_t envid;
5       uintptr_t addr;
6
7       set_pgfault_handler(pgfault);
8
9       if ((envid = sys_exofork()) == 0) {
10          // child
11          thisenv = &envs[ENVX(sys_getenvid())];
12          return 0;
13      }
14
15      for (addr = 0; addr < USTACKTOP; addr += PGSIZE) {
16          if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] &
                PTE_P)
17              && (uvpt[PGNUM(addr)] & PTE_U))
18              duppage(envid, PGNUM(addr));
19      }
20
21      if ((r = sys_page_alloc(envid, (void *) (UXSTACKTOP -
            PGSIZE),
22          PTE_P | PTE_U | PTE_W)) < 0)
23          panic("fork: %e", r);
24
25      extern void _pgfault_upcall();
26      sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
27
28      if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
29          panic("fork: %e", r);
30
31      return envid;
32      panic("fork not implemented");
```

```
33 }
```

14 作业十一

修改 kern/trapenrty.S 和 kern/trap.c 来初始化一个合适的 IDT 入口，并为 IRQ 0-15 提供处理函数。接着，修改 kern/env.c 中的 env_alloc() 以确保用户进程总是在中断被打开的情况下运行。

首先，在 kern/trapenrty.S 中初始化 IDT 入口 IRQ 0-15。

```
1  TRAPHANDLER_NOEC(IRQ_0, IRQ_OFFSET + 0)
2  TRAPHANDLER_NOEC(IRQ_1, IRQ_OFFSET + 1)
3  TRAPHANDLER_NOEC(IRQ_2, IRQ_OFFSET + 2)
4  TRAPHANDLER_NOEC(IRQ_3, IRQ_OFFSET + 3)
5  TRAPHANDLER_NOEC(IRQ_4, IRQ_OFFSET + 4)
6  TRAPHANDLER_NOEC(IRQ_5, IRQ_OFFSET + 5)
7  TRAPHANDLER_NOEC(IRQ_6, IRQ_OFFSET + 6)
8  TRAPHANDLER_NOEC(IRQ_7, IRQ_OFFSET + 7)
9  TRAPHANDLER_NOEC(IRQ_8, IRQ_OFFSET + 8)
10 TRAPHANDLER_NOEC(IRQ_9, IRQ_OFFSET + 9)
11 TRAPHANDLER_NOEC(IRQ_10, IRQ_OFFSET + 10)
12 TRAPHANDLER_NOEC(IRQ_11, IRQ_OFFSET + 11)
13 TRAPHANDLER_NOEC(IRQ_12, IRQ_OFFSET + 12)
14 TRAPHANDLER_NOEC(IRQ_13, IRQ_OFFSET + 13)
15 TRAPHANDLER_NOEC(IRQ_14, IRQ_OFFSET + 14)
16 TRAPHANDLER_NOEC(IRQ_15, IRQ_OFFSET + 15)
```

然后参照其他 IDT 部分代码，在 kern/trap.c 中建立 interrupt/trap gate descriptor。

```
1 extern void IRQ_0();
2 extern void IRQ_1();
```

```
3  extern void IRQ_2();
4  extern void IRQ_3();
5  extern void IRQ_4();
6  extern void IRQ_5();
7  extern void IRQ_6();
8  extern void IRQ_7();
9  extern void IRQ_8();
10 extern void IRQ_9();
11 extern void IRQ_10();
12 extern void IRQ_11();
13 extern void IRQ_12();
14 extern void IRQ_13();
15 extern void IRQ_14();
16 extern void IRQ_15();
17
18 SETGATE(idt[IRQ_OFFSET + 0], 0, GD_KT, IRQ_0, 0);
19 SETGATE(idt[IRQ_OFFSET + 1], 0, GD_KT, IRQ_1, 0);
20 SETGATE(idt[IRQ_OFFSET + 2], 0, GD_KT, IRQ_2, 0);
21 SETGATE(idt[IRQ_OFFSET + 3], 0, GD_KT, IRQ_3, 0);
22 SETGATE(idt[IRQ_OFFSET + 4], 0, GD_KT, IRQ_4, 0);
23 SETGATE(idt[IRQ_OFFSET + 5], 0, GD_KT, IRQ_5, 0);
24 SETGATE(idt[IRQ_OFFSET + 6], 0, GD_KT, IRQ_6, 0);
25 SETGATE(idt[IRQ_OFFSET + 7], 0, GD_KT, IRQ_7, 0);
26 SETGATE(idt[IRQ_OFFSET + 8], 0, GD_KT, IRQ_8, 0);
27 SETGATE(idt[IRQ_OFFSET + 9], 0, GD_KT, IRQ_9, 0);
28 SETGATE(idt[IRQ_OFFSET + 10], 0, GD_KT, IRQ_10, 0);
29 SETGATE(idt[IRQ_OFFSET + 11], 0, GD_KT, IRQ_11, 0);
30 SETGATE(idt[IRQ_OFFSET + 12], 0, GD_KT, IRQ_12, 0);
31 SETGATE(idt[IRQ_OFFSET + 13], 0, GD_KT, IRQ_13, 0);
32 SETGATE(idt[IRQ_OFFSET + 14], 0, GD_KT, IRQ_14, 0);
33 SETGATE(idt[IRQ_OFFSET + 15], 0, GD_KT, IRQ_15, 0);
```

外部中断被处在%eflags 的 FL_IF 标志位控制。当这一位被置位时，外部中断被打开。这个标志位可以有多种方式被修改，但为了简化，我们仅仅需要在保存和恢复%eflags 的时候，即，进入或退出用户模式时，修改。

修改 kern/env.c 中的 env_alloc() 以确保用户进程总是在中断被打开的情况下运行。

```
1 e->env_tf.tf_eflags |= FL_IF;
```

此时，当我们运行任何一个运行时间较长的测试程序时（比如 make run-spin），可以看到内核打印硬件中断的 trap frame。因为到目前为止，虽然处理器的硬件中断已经被打开了，但 JOS 还没有处理它，它以为这个中断发生在正在运行的用户进程，并将其销毁。

15 作业十二

修改内核的 trap_dispatch() 函数，使得其每当收到时钟中断的时候，它会调用 sched_yield() 寻找另一个进程并运行。

根据注释，在调用 scheduler 之前，先调用 lapic_eoi() 承认中断。

```
1 if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
2     lapic_eoi();
3     sched_yield();
4     return;
5 }
```

16 作业十三

实现 kern/syscall.c 中的 sys_ipc_recv 和 sys_ipc_try_send。在实现它们前，你应当读读两边的注释，因为它们需要协同工作。当你在这些例程中调用 envid2env 时，你应当将 checkperm 设置为 0，这意味着进程可以与任何其他进程通信，内核除了确保目标进程 ID 有效之外，不会做其他任何检查。

接下来在 lib/ipc.c 中实现 ipc_recv 和 ipc_send。

用 user/pingpong 和 user/primes 来测试你的 IPC 机制。user/primes 会为每一个素数生成一个新的进程，直到 JOS 已经没有新的进程页可以分配了。

我们首先在 syscall() 中增加对这两个函数的调用：

```
case SYS_ipc_recv:
    return sys_ipc_recv((void *)a1);
case SYS_ipc_try_send:
    return sys_ipc_try_send((envid_t)a1, (uint32_t) a2, (void
        *)a3, (unsigned) a4);
```

进程调用 sys_ipc_recv 来接受一个消息。系统调用将其移出运行队列，直到收到消息前都不再运行。当一个进程在等待接受消息状态时，任何一个进程都可以向它发送消息。

进程调用 sys_ipc_recv 时如果带有一个有效的 dstva 参数（在 UTOP 之下），它即表明自己希望收到一个页映射。如果发送者发送了一个页面，这个页应当被映射在接收者地址空间的 dstva 位置。如果接收者在 dstva 位置已经映射了一个页面，之前的页面将被取消映射。

在任何一个进程间通信发生后，内核应当将接收者的 struct Env 中新的字段 env_ipc_perm 设置为接收到的页面权限，如果没有收到页面，应当设置为 0。

```
static int
sys_ipc_recv(void *dstva) {
    // LAB 4: Your code here.
    if((uintptr_t)dstva < UTOP && (ROUNDDOWN((uintptr_t)dstva,
        PGSIZE) != (uintptr_t)dstva))
        // -E_INVAL if status is not a valid status for an
            environment.
        return -E_INVAL;

    // Only when dstva is below UTOP, record it in struct Env
    if((uintptr_t)dstva < UTOP)
        curenv->env_ipc_dstva = dstva;
```

```
11      curenv->env_ipc_recving = true;
12
13      // mark yourself as not runnable, give up CPU
14      curenv->env_status = ENV_NOT_RUNNABLE;
15      sched_yield();
16
17      // This sentence will never be executed
18      return 0;
19  }
```

进程调用 sys_ipc_try_send 来发送一个值。这个函数带有两个参数接收者的进程 ID 和想要发送的值。如果目标进程正处于接收消息的状态（即，已经调用了 sys_ipc_recv 但还没有收到一个消息），这个函数将发送消息并返回 0。否则函数返回 -E_IPC_NOT_RECV 来指示目标进程并不希望收到一个值。

进程调用 sys_ipc_try_send 时如果带有一个有效的 srcva 参数（在 UTOP 之下），这意味着发送者希望发送一个目前映射在 srcva 的页面给接收者，权限是 perm。进程间通信成功后，发送者地址空间在 srcva 的原有页面保持不变，接收者在 dstva 获得一份同一个物理页的拷贝。

```
1   static int
2   sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva,
        unsigned perm) {
3       // LAB 4: Your code here.
4       struct Env* recv = NULL;
5       int error_code = 0;
6       if((error_code = envid2env(envid, &recv, 0)) < 0)
7           return error_code;
8       if(!recv->env_ipc_recving)
9           return -E_IPC_NOT_RECV;
10      recv->env_ipc_perm = 0;
11      recv->env_ipc_from = curenv->env_id;
```

```
12      recv->env_ipc_value = value;
13
14      // when to do the following check
15      if((uintptr_t)srcva < UTOP && (uintptr_t)(recv->
            env_ipc_dstva) < UTOP) {
16          if((uintptr_t)srcva != ROUNDDOWN((uintptr_t)srcva,
                PGSIZE))
17              return -E_INVAL;
18
19          // check perm, is PTE_U and PTE_P already set?
20          if(((perm & PTE_U) == 0) || ((perm & PTE_P) == 0) )
21              return -E_INVAL;
22
23          // is perm set with other perms that should never be
                set?
24          // bit-and ~PTE_SYSCALL clear the four bits
25          if((perm & ~PTE_SYSCALL) != 0)
26              return -E_INVAL;
27          pte_t* pte_addr = NULL;
28          struct PageInfo* page = NULL;
29          page = page_lookup(curenv->env_pgdir, srcva, &pte_addr)
                ;
30
31          // srcva is not mapped
32          if(page == NULL)
33              return -E_INVAL;
34
35          // the page is read-only, but perm contains write
36          if((perm & PTE_W) && !((*pte_addr) & PTE_W))
37              return -E_INVAL;
38
39          // Now start to do the real stuff
```

```
40          if((error_code = page_insert(recv->env_pgdir, page,
                recv->env_ipc_dstva, perm)) < 0)
41              return error_code;
42          recv->env_ipc_perm = perm;
43      }
44
45      // unblock and make it running
46      recv->env_ipc_recving = 0;
47      recv->env_tf.tf_regs.reg_eax = 0;
48      recv->env_status = ENV_RUNNABLE;
49      return 0;
50  }
```

// Receive a value via IPC and return it. ipc_recv() 通过 IPC 接受一个值并返回它。

如果 pg 非空,将发送者发送的任何页面都映射到其中;如果 from_env_store 非空，将发送者的 envid 记录到其上；如果 perm_store 非空，将发送者的页面的权限记录到其上；最后，如果系统调用失败，将 0 存到 fromenv 和 perm 中并返回错误。

```
1  // Hint:
2  //   Use 'thisenv' to discover the value and who sent it.
3  //   If 'pg' is null, pass sys_ipc_recv a value that it will
       understand
4  //   as meaning "no page".  (Zero is not the right value, since
       that's
5  //   a perfectly valid place to map a page.)
6  int32_t
7  ipc_recv(envid_t *from_env_store, void *pg, int *perm_store) {
8      // LAB 4: Your code here.
9      if (from_env_store != NULL) {
10         *from_env_store = 0;
```

```
11      }
12      if (perm_store != NULL) {
13          *perm_store = 0;
14      }
15      if (pg == NULL) {
16          // Set a value > UTOP
17          pg = (void *)-1;
18      }
19      int r = sys_ipc_recv(pg);
20      if (r < 0) {
21          // Error
22          return r;
23      }
24      if (from_env_store != NULL) {
25          *from_env_store = thisenv->env_ipc_from;
26      }
27      if (perm_store != NULL) {
28          *perm_store = thisenv->env_ipc_perm;
29      }
30      return thisenv->env_ipc_value;
31  }
```

ipc_send() 将 val 发送给 toenv，对任何错误都应当 panic。

```
1  // Hint:
2  //   Use sys_yield() to be CPU-friendly.
3  //   If 'pg' is null, pass sys_ipc_try_send a value that it
       will understand
4  //   as meaning "no page".  (Zero is not the right value.)
5  void
6  ipc_send(envid_t to_env, uint32_t val, void *pg, int perm) {
7      // LAB 4: Your code here.
```

```
8       if (pg == NULL) {
9           // Set a value > UTOP
10          pg = (void *)-1;
11      }
12      if (sys_ipc_try_send(to_env, val, pg, perm) < 0) {
13          panic("ipc_send failed!");
14      }
15  }
```