# Lab 4: 抢占式多任务

Lab 4 练习思路

在本实验中将在多个同时活跃的用户模式环境中实现抢占式多任务。

第一部分，向JOS添加多处理器支持，实现循环调度，并添加基本的环境管理系统调用（创建和销毁环境的调用以及分配/映射内存）。

第二部分，实现一个类似Unix的 `fork()` ，它允许用户模式环境创建自己的副本。

第三C部分，增加对进程间通信IPC的支持，允许不同的用户模式环境进行显式通信和同步。并且添加对硬件时钟中断和抢占的支持。

## 第一部分 多处理器支持和协同多任务

第一部分，首先扩展JOS在多处理器系统上运行，然后实施一些新的JOS内核系统调用，允许用户级环境创建其他新的环境。实施协作循环调度，当前环境退出时，允许内核从一个环境切换到另一个环境。第三部分将实施抢占式调度，即使在环境不合作的情况下，允许内核在一定时间内从环境中重新控制CPU。

### 1. 多处理器支持

我们将使JOS支持对称多处理SMP，一种多处理器模式。其中所有CPU具有对存储器和IO总线等系统资源的等效访问。虽然所有CPU在SMP中功能相同，但在引导过程中，它们可以分为两种类型：引导处理器BSP负责初始化系统和引导操作系统；应用处理器AP仅在操作系统启动并运行之后被BSP激活。哪个处理器属于BSP是由硬件和BIOS决定的。到目前为止，所有现有的JOS代码都已经在BSP上运行。

在SMP系统中，每个CPU都有一个随附的本地APIC（LAPIC）单元。LAPIC单元负责在整个系统中传递中断。LAPIC还为其连接的CPU提供唯一的标识符。在本实验中，我们利用LAPIC单元的以下基本功能（在 `kern/lapic.c` 中定义）：

- 读取LAPIC标识符以告知当前代码正在运行哪个CPU，参阅 `cpunum()`
- 将 `STARTUP` 处理器中断IPI从BSP发送到AP以启动其他CPU,参阅 `lapic_startap()`
- 在第三部分，我们将编写LAPIC的内置定时器来触发时钟中断，以支持抢占式多任务，参阅 `apic_init()`

处理器使用内存映射IO（MMIO）访问其LAPIC。在MMIO中，物理内存的一部分硬连线到某些IO设备的寄存器，因此通常用于访问存储器的访存指令可用于访问设备寄存器。您已经在物理地址0xA0000看到一个IO口（用于写入VGA显示缓冲区）。LAPIC位于一个始于物理地址0xFE000000（4GB最末端的32MB）的一个缺口中，所以我们使用我们通常在KERNBASE的直接映射访问太高了。JOS虚拟内存映射在 `MMIOBASE` 上留

下了4MB的差距，所以我们有一个映射这样的设备的地方。由于后来的实验引入了更多的MMIO区域，您将会编写一个简单的函数来分配这个区域的空间并将设备内存映射到它。

> **练习一：**
>
> 在 `kern/pmap.c` 中实现 `mmio_map_region` 。查阅 `kern/lapic.c` 中 `lapic_init` 的开头部分。测试 `mmio_map_region` 之前，还需要先完成练习二。
>
> 我的思路

## 2. 应用处理器引导

在启动AP之前，BSP应首先收集有关多处理器系统的信息，例如CPU的总数、APIC ID和LAPIC单元的MMIO地址。 `kern/mpconfig.c` 中的 `mp_init()` 函数通过读取BIOS内存区域中的MP配置表来检索此信息。

`kern/init.c` 中的 `boot_aps()` 函数驱动AP引导进程。AP以实模式开始，很像引导加载程序在 `boot/boot.s` 中的启动，所以 `boot_aps()` 将AP入口代码（ `kern/mpentry.s` ）复制到在实模式下可寻址的内存位置。与引导加载程序不同的，第一是可以控制AP在哪里开始执行代码，第二是将条目代码复制到0x7000（ `MPENTRY_PADDR` ），但低于640KB的任何未使用的，页面对齐的物理地址都可以工作。

之后 `boot_aps()` 会逐个激活AP，通过发送 `STARTUP` IPI到相应AP的LAPIC单元，以及AP应该开始运行其入口代码的初始CS: IP地址（在我们的例子中为 `MPENTRY_PADDR` ）。 `kern/mpentry.s` 中的入口代码与 `boot/boot.s` 非常相似。经过一些简单的设置，它将AP置于保护模式并启用分页，然后调用C安装程序 `mp_main()` （在 `kern/init.c` 中）。 `boot_aps()` 等待AP直到AP在其 `struct CpuInfo` 的 `cpu_status` 域中发出 `CPU_STARTED` 标志的到通知，然后唤醒下一个AP。

> **练习二：**
>
> 阅读 `kern/init.c` 中的 `boot_aps()` 和 `mp_main()` ，并在 `kern/mpentry.s` 中查看汇编代码。 确保了解AP引导过程中的控制流传输。然后在 `kern/pmap.c` 中修改 `page_init()` 的实现，以避免将 `MPENTRY_PADDR` 中的页面添加到空闲列表中，从而我们可以安全地复制并运行该物理地址的AP引导代码。您的代码应该通过更新的 `check_page_free_list()` 测试，但不能通过 `check_kern_pgdir()` 测试。
>
> 我的思路

## 3. 单CPU状态和初始化

在编写多处理器操作系统时，区分每个处理器私有的CPU的状态以及整个系统共享的全局状态是很重要的。 `kern/cpu.h` 定义了很多单CPU状态，包括存储单CPU变量的 `struct CpuInfo` 。 `cpunum()`

总是返回调用它的CPU的ID，它可以用作像 `cpus` 这样的数组的索引。或者，宏 `thiscpu` 是当前CPU的 `struct CpuInfo` 的替代。

以下是应该关注的单CPU状态：

- 单CPU内核堆栈

  因为多个CPU可以同时陷入内核，所以我们需要一个单独的内核栈，每个处理器防止它们干扰对方的执行。数组 `percpu_kstacks[NCPU][KSTKSIZE]` 为NCPU的内核堆栈保留空间。

  Lab 2将 `bootstack` 引用的物理内存映射到 `KSTACKTOP` 下方的BSP内核栈。类似地，本实验将单CPU的内核堆栈映射到该区域，其中保护页面作为它们之间的缓冲区。CPU 0的堆栈仍从 `KSTACKTOP` 下降；CPU 1的堆栈从CPU 0堆栈底部开始的 `KSTKGAP` 字节开始下降，依此类推。 `inc/memlayout.h` 中显示了这一映射布局。

- 单CPU TSS和TSS描述符

  单CPU任务状态段（TSS）也用于指定每个CPU内核栈的位置。CPU i的TSS存储在 `cpus[i].cpu_ts` 中，相应的TSS描述符在GDT入口 `gdt[(GD_TSS0 >> 3) + i]` 中定义。在 `kern/trap.c` 中定义的全局 `ts` 变量将失效。

- 单CPU当前环境指针

  因为每个CPU可以同时运行不同的用户进程，所以我们重新定义了符号 `curenv` ，用来引用 `cpus[cpunum()]` 。 `cpu_env` 或 `thiscpu->cpu_env` 指向在当前CPU（正在执行代码的CPU）上正在运行的环境。

- 单CPU系统寄存器。

  所有寄存器（包括系统寄存器）对CPU来说都是专用的。因此，初始化这些寄存器的指令（如 `lcr3()` `ltr()` `lgdt()` `lidt()` 等）都必须在每个CPU上执行一次。因此定义了 `env_init_percpu()` 和 `trap_init_percpu()` 函数。

  此外，如果在之前的Lab中添加了任何额外的单CPU状态或执行任何额外的CPU初始化（也就是说，在CPU寄存器中设置新位），就必须在每个CPU上都复制一次。

> **练习三：**
>
> 修改 `mem_init_mp()` （ `kern/pmap.c` ），映射从 `KSTACKTOP` 开始的每个CPU堆栈。每个堆栈的大小是 `KSTKSIZE` 个字节加上未映射的保护页面的 `KSTKGAP` 个字节。结果应该可以通过 `check_kern_pgdir()` 。
>
> 我的思路

> **练习四：**
>
> `trap_init_percpu()`（`kern/trap.c`）初始化BSP的TSS和TSS描述符。它在实验3中可以运行，但在其他CPU上运行时不正确。修改代码，使其可以在所有CPU上工作。
>
> 注：新代码不再使用全局 `ts` 变量。
>
> 我的思路

完成以上练习后，执行 `make qemu CPUS=4`，结果应当如下：

```
...
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
```

### 4. 锁定

在 `mp_main()` 中初始化AP后，当前的代码将自动转换。在让AP进一步进行之前，当多个CPU同时运行内核代码时，我们需要首先解决竞争条件。实现这一点的最简单的方法是使用一个大的内核锁。内核锁是一个单一的全局锁，每当环境进入内核模式时都会保持该锁，并在环境返回到用户模式时释放。在这种模式下，用户模式下的环境可以在任何可用的CPU上同时运行，但是不能多于一个环境可以在内核模式下运行，任何其他尝试进入内核模式的环境都被迫等待。

`kern/spinlock.h` 声明了这一大内核锁，即 `kernel_lock`。它还提供了 `lock_kernel()` 和 `unlock_kernel()`，以快速获取和释放锁。有四个位置应该使用大内核锁：

- 在 `i386_init()` 中，在BSP唤醒其他CPU之前获取锁
- 在 `mp_main()` 中，初始化AP后获取锁，然后调用 `sched_yield()` 开始在此AP上运行环境
- 在 `trap()` 中，在从用户模式陷入时获取锁。要确定在用户模式还是内核模式下发生陷入，请检查 `tf_cs` 的低位
- 在 `env_run()` 中，在切换到用户模式之前释放锁。不要太早或太晚，否则可能导致死锁

> **练习五：**
>
> 通过在合适的位置调用 `lock_kernel()` 和 `unlock_kernel()` 来实现上述内核锁。但是到下一

个练习完成后才可以测试内核锁的实现是否正确。

我的思路

### 5. 循环调度

下一个任务是更改JOS内核，使它可以以“循环”的方式在多个环境之间交替使用。JOS中的循环调度工作如下：

- `kern/sched.c` 中的函数 `sched_yield()` 负责选择要运行的新环境。它以循环方式依次搜索 `envs[]` 数组，从刚刚运行的环境开始（如果没有以前运行的环境，则是数组的开头）选择第一个状态为 `ENV_RUNNABLE` 的环境（见 `inc/env.h` ），并调用 `env_run()` 来跳转到该环境中。
- `sched_yield()` 不能同时在两个CPU上运行相同的环境。它可以说明一个环境当前在某些CPU（可能是当前的CPU）上运行，因为该环境的状态将是 `ENV_RUNNING`
- 我们已经为您实现了一个新的系统调用 `sys_yield()` ，用户环境可以调用它来调用内核的 `sched_yield()` 函数，从而自愿将CPU放弃到不同的环境中。

**练习六：**

实现 `sched_yield()` 中的循环调度。不要忘记修改 `syscall()` 来调度 `sys_yield()` 。

确保在 `mp_main` 中调用 `sched_yield()` 。

修改 `kern/init.c` ，创建三个都运行程序 `user/yield.c` 的环境。

运行 `make qemu` ，应该可以看到环境之间来回切换五次。结果如下：

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

`yield` 程序退出后，系统中不再有可运行的环境，调度程序应调用JOS内核监视器。

如果现在使用 `CPUS = 1` ，所有环境都应该成功运行。由于未处理的定时器中断（我们将在后边修复），此时设置大于1的CPU可能会导致一般保护故障，内核页错误或其他意外中断。

我的思路

## 6. 用于创建环境的系统调用

虽然内核现在能够在多个用户级环境之间运行和切换，但仍仅限于内核初始设置好的运行环境。现在实现JOS系统调用，以允许用户环境创建和启动其他新的用户环境。

Unix提供 `fork()` 系统调用作为其进程创建原语。Unix下 `fork()` 复制调用进程的整个地址空间（父进程），以创建一个新进程（子进程）。从用户空间可观察到的两者之间的唯一差异是它们的进程ID和父进程ID（由 `getpid` 和 `getppid` 返回）。在父进程中， `fork()` 返回子进程的ID，而在子进程中， `fork()` 返回0。默认情况下，每个进程都拥有自己的专用地址空间，而对进程的内存修改都不可见。

你将提供不同的、更原始的一组JOS系统调用来创建新的用户模式环境。通过这些系统调用，除了可以创建其他风格的环境之外，也可以在用户空间中实现一个类Unix的 `fork()` 。需要实现的新的系统调用如下：

- `sys_exofork`

  该系统调用创建一个几乎为空白的新环境：在其地址空间的用户部分中没有映射，且不可运行。在 `sys_exofork` 调用时，新环境将具有与父环境相同的寄存器状态。在父级中， `sys_exofork` 将返回新创建环境的 `envid_t` （如果环境分配失败，则返回负错误代码）。在子级中，它将返回0。由于孩子开始标记为不可运行，所以 `sys_exofork` 实际上不会返回到孩子，直到父级显式地标记子项可运行

- `sys_env_set_status`

  将指定环境的状态设置为 `ENV_RUNNABLE` 或 `ENV_NOT_RUNNABLE` 。一旦它的地址空间和寄存器状态已被完全初始化，这个系统调用通常用于标记准备运行的新环境。

- `sys_page_alloc`

  分配物理内存页面，并将其映射到给定环境的地址空间中的给定虚拟地址。

- `sys_page_map`

  将页面的映射（而不是页面的内容）从一个环境的地址空间复制到另一个环境的地址空间，留下一个内存共享安排，以便新的和旧的映射都指向同一页的物理内存。

- `sys_page_unmap`

  取消映射给定环境中给定虚拟地址的页面。

对于上面接受环境ID的所有系统调用，JOS内核约定ID值为0表示“当前环境”。这个约定是由 `envid2env()` 在 `kern/env.c` 中实现的。

我们在测试程序 `user/dumbfork.c` 中提供了一个非常原始的类似Unix `fork()` 的实现。此测试程序使用上述系统调用创建并运行具有自己的地址空间的副本的子环境。然后两个环境使用 `sys_yield` 来回切

换，如前一个练习。父环境在10次迭代后退出，而孩子在20次迭代后退出。

> **练习七：**
>
> 在 `kern/syscall.c` 中实现上述系统调用。需要使用 `kern/pmap.c` 和 `kern/env.c` 中的各种功能，特别是 `envid2env()` 。目前每当调用 `envid2env()` 时，在 `checkperm` 参数中都传递1。确保您检查任何无效的系统调用参数，在这种情况下返回 `-E_INVAL` 。
>
> 使用 `user/dumbfork` 测试JOS内核。
>
> 我的思路

# 第二部分 写时复制fork

Unix系统将 `fork()` 系统调用作为其主进程创建原语， `fork()` 通过复制父进程的地址空间创建一个子进程。

xv6 Unix通过将父进程页面中的所有数据复制到为子进程分配的新页面中来实现 `fork()` ，这与 `dumbfork()` 所采用的方法基本相同。复制过程中，将父节点的地址空间复制到子节点是 `fork()` 操作中开销最大的部分。

但是，在子进程中调用 `fork()` 函数之后一般都会立即调用 `exec()` ，这样就可以用一个新的程序替换子进程的内存。在这种情况下，复制父地址空间的时间大部分是浪费的，因为在调用 `exec()` 之前，子进程将使用很少的内存。

为此，Unix的更高版本利用了虚拟内存硬件，允许父子进程共享映射到它们各自地址空间的内存，直到某一个进程实际修改它。这种技术被称为写时复制。为了做到这一点， `fork()` 从父进程复制到子进程的，是地址空间映射而不是映射页的内容，同时标记现有共享页面为只读。当两个进程之一尝试写入其中一个共享页面时，该进程将出现页面错误。这时，Unix内核意识到该页面是一个“虚拟”的“写时复制”副本，然后可以为故障过程创建一个新的、私有的、可写的页面副本。这样，单个页面的内容在实际写入之前实际上并不被复制。这一优化使得一个 `fork()` 后面跟着一个 `exec()` 在子进程中开销减少：在调用 `exec()` 之前，子进程可能只需要复制一个页面（其堆栈的当前页面）。

本实验的下一部分将实现一个“适当的”类似Unix的写时复制 `fork()` 作为用户空间库例程。在用户空间中实现 `fork()` 和写时复制的好处是内核简单得多、不易出错。它还允许单个用户模式程序为 `fork()` 定义自己的语义。一个程序想要一个略有不同的实现（例如，像 `dumbfork()` 这样开销比较大的版本，或者是父子进程之后实际确实共享内存的程序）也可以轻易实现。

### 1. 用户级页面故障处理

用户级的写时复制 `fork()` 需要知道写保护页面上的页面错误，所以是首先需要实现的。写时复制只是用

户级页面故障处理的用途之一。

为便于页面错误指示何时需要执行某些操作，通常会设置一个地址空间。例如，大多数Unix内核最初只映射一个新进程的堆栈区域中的单个页面，然后随着堆栈消耗增加而分配和映射额外的堆栈页面，并导致未映射的堆栈地址的页面错误。典型的Unix内核必须跟踪在进程空间的每个区域中出现页面错误时要执行的操作。例如，堆栈区域中的故障通常会分配和映射物理内存的新页面；程序BSS区域的故障通常会分配一个新的页面，填入0，然后进行映射；在具有需求分页可执行文件的系统中，文本区域的故障将从磁盘上读取二进制文件的相应页面，然后将其映射。

内核需要跟踪的信息有很多。与传统的Unix方法不同，如何处理用户空间中的每个页面错误将由你决定。该设计具有额外的优点：允许程序在定义其内存区域方面具有很大的灵活性。另外，你之后还将使用用户级页面故障处理来映射和访问基于磁盘的文件系统上的文件。

### 2. 设置页面故障处理程序

为了处理页面错误，用户环境需要向JOS内核注册一个页面错误处理程序入口点。 用户环境通过新的 `sys_env_set_pgfault_upcall` 系统调用注册其页面错误入口点。我们已经向 `Env` 结构添加了一个新成员 `env_pgfault_upcall` 来记录这些信息。

> **练习八：**
>
> 实现 `sys_env_set_pgfault_upcall` 系统调用。 在查找目标环境的环境ID时，请确保启用权限检查。
>
> 我的思路

### 3. 用户环境中的正常和异常堆栈

在正常执行期间，JOS中的用户环境将在普通用户堆栈上运行：其ESP寄存器开始指向 `USTACKTOP` ，其推入的堆栈数据位于 `USTACKTOP - PGSIZE` 和 `USTACKTOP - 1` 之间的页面上。然而，当用户模式发生页面错误时，内核将重新启动一个，在不同堆栈上运行指定用户级页面错误处理程序的用户环境，即用户异常堆栈。实质上，我们将使JOS内核实现代表用户环境自动的“堆栈切换”，其方式与从用户模式转移到内核模式时，x86处理器代表JOS进行堆栈切换的方式基本一致。

JOS用户异常堆栈也占有一个页大小，其顶部被定义为虚拟地址 `UXSTACKTOP` ，因此用户异常堆栈的有效字节在 `UXSTACKTOP - PGSIZE` 到 `UXSTACKTOP - 1` 范围内。在此异常堆栈上运行时，用户级页面错误处理程序可以使用JOS的常规系统调用来映射新页面或调整映射，以便修复最初导致页面错误的任何问题。然后，用户级页面故障处理程序通过汇编语言存根返回到原始堆栈上的故障代码。

想支持用户级页面故障处理的每个用户环境都需要使用第一部分中引入的 `sys_page_alloc()` 系统调用为自己的异常堆栈分配内存。

### 4. 调用用户页面故障处理程序

现在，你需要更改 `kern/trap.c` 中的页面故障处理代码，以从用户模式处理页面错误。在故障时，我们将用户环境的状态称作trap-time状态。

如果没有注册页面错误处理程序，则JOS内核会像以前一样使用消息来破坏用户环境。如果有，那么内核会在异常堆栈上设置一个trap frame，看起来像 `inc/trap.h` 的 `struct UTrapframe` ：

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

然后，内核安排用户环境恢复执行，同时页面错误处理程序使用此栈帧在异常堆栈上运行；你必须搞清如何使这种情况发生。 `fault_va` 是导致页面错误的虚拟地址。

如果异常发生时用户环境已经在用户异常堆栈上运行，则页面错误处理程序本身有故障。在这种情况下，您应该在当前 `tf->tf_esp` 而不是 `UXSTACKTOP` 下启动新的堆栈框架。你应该首先向栈中压入一个空的32位字，然后压入一个 `struct UTrapframe` 。

要测试 `tf->tf_esp` 是否已经在用户异常堆栈上，请检查它是否在 `UXSTACKTOP - PGSIZE` 和 `UXSTACKTOP - 1` 之间的范围内（包括端点）。

> **练习九：**
>
> 实现 `kern/trap.c` 中 `page_fault_handler` 部分代码，以将页面错误分派到用户模式处理程序。写入异常堆栈时，请务必采取适当的预防措施（如果异常堆栈上的用户环境空间不足，会发生什么情况？）。
>
> 我的思路

### 5. 用户模式页面故障入口点

接下来，你需要实现汇编程序，该程序将负责调用C页面错误处理程序并在原始故障指令下继续执行。该汇编程序集就是通过 `sys_env_set_pgfault_upcall()` 向内核注册的处理程序。

> **练习十：**
>
> 在 `lib/pfentry.s` 中实现 `_pgfault_upcall` 。有趣的部分是返回引起页面错误的用户代码中的原始点。 你将直接返回那里，而不需要通过内核。比较困难的部分是同时切换堆栈并重新加载EIP。
>
> 我的思路

最后，您需要实现用户级页面故障处理机制的C用户库部分。

> **练习十一：**
>
> 完成 `lib/pgfault.c` 中的 `set_pgfault_handler()` 。
>
> 我的思路

### 6. 实现写时复制fork

现在已经有充分的条件实现用户空间中的写时复制 `fork()` 了。

我们在 `lib/fork.c` 中为 `fork()` 提供了一个大致框架。和 `dumbfork()` 一样， `fork()` 应该创建一个新的环境，然后扫描父环境的整个地址空间，并在孩子中设置相应的页面映射。关键的区别在于， `dumbfork()` 复制所有页面； `fork()` 最初只会复制页面映射，只有当一个环境尝试写入它时， `fork()` 才会复制每个页面。

`fork()` 的基本控制流程如下：

1. 父环境使用上面实现的 `set_pgfault_handler()` 函数安装 `pgfault()` 作为C级页面错误处理程序。
2. 父环境调用 `sys_exofork()` 创建一个子环境。
3. 对于在UTOP以下地址空间中的每个可写或写时复制页面，父级调用 `duppage` 将写时复制页面映射的到子环境的地址空间中，然后再将写（注意这里的顺序：先在子环境中将页面标记为COW然后在父环境中标记）。 `duppage` 设置两个PTE，使页面不可写，并且在avail字段中包含 `PTE_COW` 以区分写时复制页面和真实的只读页面。

   但是，这种异常堆栈不会重新映射。你需要为子环境分配一个新的页面用于异常堆栈。由于页面错误处理程序将执行实际的复制，并且页面错误处理程序在异常堆栈上运行，异常堆栈不能写时复制：谁将复制它?

   `fork()` 还需要处理存在但不能写入或写时复制的页面。
4. 父环境将子环境的用户页面错误入口设置为自己的页面。

5. 自环境已经准备好运行了，所以父环境将其标记为可运行状态。

每当一个环境写入一个尚未写入的写时复制页面时，会出现页面错误。以下是用户页面错误处理程序的控制流程：

1. 内核将页面错误传播到 `_pgfault_upcall` ，它调用 `fork()` 的 `pgfault()` 处理程序。
2. `pgfault()` 检查故障是否是一个写入错误（检查错误代码中的 `FEC_WR` ），并且 该页面的PTE标记为 `PTE_COW` 。如果不是，发出警告。
3. `pgfault()` 分配一个映射到临时位置的新页面，并将错误页面的内容 复制进去。然后故障处理程序将新页面映射到具有读写权限的适当地址，而不是之前的只读映射。

用户级别的 `lib/fork.c` 代码必须参考环境的页表以完成上述操作（例如，页面的PTE标记为 `PTE_COW` ）。内核将环境的页表映射到 `UVPT` ，正是为此目的。它用了一个映射技巧，从而易于查找用户代码的PTE。 `lib/entry.s` 设置 `uvpt` 和 `uvpd` ，以便于查找 `lib/fork.c` 中的页表信息。

> **练习十二：**
>
> 实现 `lib/fork.c` 中的 `fork` `duppage` 和 `pgfault` 。 用 `forktree` 程序测试。它应该生成以下信息，包括 `new env` `free env` 和 `exiting gracefully` 。可能不会以此顺序显示，环境ID也可能不同。
>
> ```
> 1000: I am ''
> 1001: I am '0'
> 2000: I am '00'
> 1001: I am '000'
> 1002: I am '1'
> 3000: I am '11'
> 3001: I am '10'
> 4000: I am '100'
> 1003: I am '01'
> 5000: I am '010'
> 4001: I am '011'
> 2002: I am '110'
> 1004: I am '001'
> 1005: I am '111'
> 1006: I am '101'
> ```
>
> 我的思路

# 第三部分 抢占式多任务和进程间通信（IPC）

这一部分将修改内核以抢占不合作的环境，并允许环境明确传递消息。

## 1. 时钟中断和抢占

运行 `user/spin` 测试程序。这个测试程序解除一个子环境，一旦接收到CPU的控制，它就会一直循环。父环境和内核都不会重新获得CPU。这显然不是在用户模式环境中保护系统免受错误或恶意代码攻击的理想情况，因为任何用户模式环境都可以使整个系统停止，只需进入无限循环，永不放弃中央处理器。为了允许内核抢占一个正在运行的环境，强制重新控制CPU，我们必须扩展JOS内核以支持来自时钟硬件的外部硬件中断。

## 2. 中断规则

外部中断（即器件中断）被称为IRQ。有16个可能的IRQ，编号为0到15。从IRQ到IDT条目的映射不固定。 `picirq.c` 里 `pic_init` 将0～15的IRQ映射到IDT条目 `IRQ_OFFSET` 到 `IRQ_OFFSET + 15` 。

在 `inc/trap.h` 中， `IRQ_OFFSET` 被定义为十进制数32。因此IDT条目32～47就对应IRQ号0～15。例如，时钟中断是IRQ 0。因此， `IDT[IRQ_OFFSET + 0]` （即 `IDT[32]` ）包含内核中时钟中断处理程序的地址。选择这个 `IRQ_OFFSET` ，使得设备中断与处理器异常不重叠，避免导致混乱的可能。（在运行MS-DOS的早期阶段， `IRQ_OFFSET` 是0，导致处理硬件中断和处理器异常之间的巨大混乱）

在JOS中，与xv6 Unix相比，我们做了一个关键的简化。在内核中，外部设备中断总是被禁用（和xv6一样，在用户空间中启用）。外部中断由 `%eflags` 寄存器的 `FL_IF` 标志位控制（参见 `inc/mmu.h` ）。当该位置1时，外部中断被启用。虽然这个位可以通过几种方式进行修改，但由于我们的简化，我们将通过在进入和离开用户模式时保存和恢复 `%eflags` 寄存器来处理它。

你必须确保在运行用户环境时设置 `FL_IF` 标志，以便当中断到达时，它将被传递到处理器并由中断代码处理。否则，中断将被屏蔽或忽略，直到重新启用中断。我们用引导程序的第一个指令屏蔽了中断，到目前为止，我们从来没有重新启用它们。

> **练习十三：**
>
> 修改 `kern/trapentry.S` 和 `kern/trap.c` 来初始化IDT中的相应条目，并为IRQ 0～15提供处理程序。然后在 `kern/env.c` 中修改 `env_alloc()` 中的代码，以确保用户环境始终在启用中断的情况下运行。
>
> 当调用硬件中断处理程序时，处理器从不推送错误代码。
>
> 完成此练习后，如果您运行的任何测试程序运行的时间异常（例如 `spin` ），您应该看到硬件中断的内核打印陷阱帧。当处理器中的中断被启用时，JOS还没有处理它们，所以你应该看到它对当前正在运行的用户环境的每个中断都是错误的，并将其破坏。最终，它将耗尽环境、销毁并进入内核监视器。
>
> 我的思路

## 3. 处理时钟中断

`user/spin.c` 程序中，在子环境第一次运行之后，它只是在一个循环中运行，内核重新获取控制权。我们需要对硬件进行编程以定期产生时钟中断，这将强制控制回到内核，我们可以将控制切换到不同的用户环境。

已经编写的对 `lapic_init` 和 `pic_init` （ `init.c` 中的 `i386_init` ）的调用设置了时钟和中断控制器以产生中断。您现在需要编写代码来处理这些中断。

> **练习十四：**
>
> 修改内核的 `trap_dispatch()` 函数，以便在发生时钟中断时调用 `sched_yield()` 来查找和运行不同的环境。
>
> 现在应该能够使 `user/spin` 测试工作：父环境应该将子程序 `sys_yield()` 分配给它几次，但是在每种情况下都会在一段时间后重新获得CPU的控制，最后销毁子环境环境，正常终止。
>
> 我的思路

### 4. 流程间通信（IPC）

准确来说，JOS属于环境间通信或IEC，但是所有人都称之为IPC，所以我们还是使用标准术语IPC。

我们一直专注于操作系统的隔离方面，它提供了每个程序都拥有一台机器的错觉。操作系统的另一个重要服务是允许程序在必要时进行通信。让程序与其他程序进行交互可能相当强大。Unix管道模型是典型的例子。

有很多进程间通信的模型。即使在今天，仍然有关于哪些型号最好的争论。在这一部分，我们将实现一个简单的IPC机制。

### 5. JOS中的IPC

你将实现一些额外的JOS内核系统调用，它们共同提供了一个简单的进程间通信机制。你需要实现 `sys_ipc_recv` 和 `sys_ipc_try_send` 两个系统调用。然后你需要实现两个库包装器 `ipc_recv` 和 `ipc_send` 。

通信机制中的“消息”由两个组件组成：单个32位值和可选的单页映射。允许环境传递消息中的页面映射提供了一种有效的方法来传输更多的数据，而不仅仅是单个32位整数，并且还允许环境容易地设置共享内存布置。

### 6. 发送和接收消息

要接收消息，环境调用 `sys_ipc_recv` 。此系统调用会调度当前环境，并且在收到消息之前不再运行它。当环境等待接收消息时，任何其他环境都可以向其发送消息，不仅仅是特定的环境，也不仅仅是具有接收环境的父/子安排的环境。换句话说，第一部分中实现的权限检查将不适用于IPC，因为IPC系统调用是设计好以便安全的：环境不能仅通过发送消息来导致另一个环境故障（除非目标环境也是有故障的）。

为了尝试发送一个值，环境调用 `sys_ipc_try_send` ，同时发送接收方的环境ID和要发送的值。如果对象环境正在接收信息（已经调用了 `sys_ipc_recv` 但还没有获得值），那么send将传递消息并返回0。否则，返回 `-E_IPC_NOT_RECV` 来指示目标环境当前不希望接收值。

用户空间中的库函数 `ipc_recv` 将负责调用 `sys_ipc_recv` ，然后在当前环境的 `struct Env` 中查找有关接收到的值的信息。

类似地，库函数 `ipc_send` 将重新调用 `sys_ipc_try_send` 直到发送成功。

### 7. 传输页面

当环境使用有效的 `dstva` 参数（低于 `UTOP` ）调用 `sys_ipc_recv` 时，环境表示它愿意接收页面映射。如果发送方发送一个页面，那么该页面应该在接收方地址空间中的 `dstva` 处映射。如果接收方已经在 `dstva` 上映射了一个页面，那么之前的页将被取消映射。

当环境使用有效的 `srcva` 参数（低于 `UTOP` ）调用 `sys_ipc_try_send` 时，这意味着发送方想要将当前映射到 `srcva` 的页面发送给接收方，并且具有权限 `perm` 。成功进行IPC之后，发送方将 `srcva` 的页面原始映射保存在其地址空间中，但是接收方也可以在接收者的地址空间中由接收方最初指定的 `dstva` 处获取同一物理页面的映射。因此，该页面在发送者和接收者之间共享。

如果发送者或接收者没有指示一个页面应该被传送，那么没有页面被传送。在任何IPC之后，内核将接收者的 `Env` 中的新字段 `env_ipc_perm` 设置为接收到的页面的权限，如果没有接收到页面，则为零。

### 8. 实现IPC

> **练习十五：**
>
> 在 `kern/syscall.c` 中实现 `sys_ipc_recv` 和 `sys_ipc_try_send` 。当在这些例程中调用 `envid2env` 时，应该将 `checkperm` 标志设置为0，这意味着允许任何环境将IPC消息发送到任何其他环境，除了验证目标 `envid` 是否有效之外，内核没有特殊的权限检查。
>
> 然后在 `lib/ipc.c` 中实现 `ipc_recv` 和 `ipc_send` 函数。
>
> 使用 `user/pingpong` 和 `user/primes` 函数测试您的IPC机制。 `user/primes` 将为每个素数生成一个新的环境，直到JOS耗尽环境。
>
> 我的思路

# 练习思路

## 练习一：

源代码中的注释提示得已经很明确了，就是将 [ `pa` , `pa + size` ) 范围的物理地址映射到 [ `base` , `base + size` ) 范围的虚拟地址，使用的函数就是提示中给的 `boot_map_region()` 。以及函数外的注释提示 `size` 不一定是 `PGSIZE` 的倍数，所以要先向上取整。添加代码如下：

```
660     // Your code here:
661
662     void *temp = base;
663     pg_size = ROUNDUP(size, PGSIZE);
664
665     if (base + pg_size > MMIOLIM)
666         panic("mmio_map_region: reservation overflow\n");
667
668     boot_map_region(kern_pgdir, base, pg_size, pa, PTE_PCD | PTE_PWT | PTE_W);
669     base += pg_size;
670
671     return temp;
```

参考了下网络上的代码，不太清楚为什么还要考虑 `base + pgsize < base` 的情况，不知道是不是还要考虑 `size` 是负的。

## 练习二：

练习二首先要求阅读源码，搞清楚AP引导过程中的控制流传输。我一般对阅读源码的地方都是以搞清大意为目的，暂时不去细看的原因是有些函数、操作可能还会附带其他部分的知识，如果直接详细研究的话可能会比较头疼。我一般是在函数的粒度上去读源码，如果遇到搞不懂的就先顺着思路猜测，然后后期可能就会遇到解释，或者学了另一部分之后就可以理解。我个人觉得效果不错。

对于这个练习中的两个C语言函数，注释基本上做了很大提示。我的理解就是先把代码段写入 `MPENTRY_PADDR` ，然后逐个引导AP，然后引导的时候还会调用 `mp_main` 进行一些基本设置，包括初始化LAPIC、环境、中断异常等，最后把AP的状态调换成 `CPU_STARTED` ，进行下一个AP的引导。

对于文件 `kern/mpentry.s` ，大部分都不大清楚是做什么的，所以就看了一下网上的学习笔记，我理解的大致意思就是和之前的 `boot.s` 作用差不多（当时 `boot.s` 啃不动也是看的学习笔记），主要就是开启分页，转到内核栈上去，最后跳转到 `mp_main` 函数中去。

修改 `page_init()` 非常简单，只要把 `MPENTRY_PADDR` 页面标为已用就行了，代码如下：

```
330     int mpentry = MPENTRY_PADDR / PGSIZE;
331
332     for (i = 0; i < npages; i++) {
333         if (i == bsmem_head ||
334             (i >= iohole_start && i < iohole_end) ||
335             (i >= extmem_used_start && i < extmem_used_end)) {
336             pages[i].pp_ref = 1;
337             pages[i].pp_link = NULL;
338         } else if (i == mpentry) {
339             continue;
340         } else {
341             pages[i].pp_ref = 0;
342             pages[i].pp_link = page_free_list;
343             page_free_list = &pages[i];
344         }
345     }
```

## 练习三：

这部分和之前 `mem_init()` 里映射堆栈的过程是一样的，只是堆栈栈底和栈的大小有所变化，权限等级注释中也说得很清楚是内核可读写、用户不可访问。比较简单，代码如下：

```
282     for (int i = 0; i < NCPU; i++) {
283         boot_map_region(kern_pgdir,
284                         KSTACKTOP - i * (KSTKSIZE + KSTKGAP) - KSTKSIZE,
285                         ROUNDUP(KSTKSIZE, PGSIZE),
286                         PADDR(&percpu_kstacks[i]),
287                         PTE_P | PTE_W);
288     }
289 }
```

## 练习四：

这个练习就是把 `trap_init_percpu()` 里所有全局的TSS换成对应的当前单CPU的TSS，注释里也有相应替换的方法。代码如下：

```
144     // LAB 4: Your code here:
145
146     // Setup a TSS so that we get the right stack
147     // when we trap to the kernel.
148     thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpunum() * (KSTKGAP + KSTKSIZE);
149     thiscpu->cpu_ts.ts_ss0 = GD_KD;
150
151     // Initialize the TSS slot of the gdt.
152     gdt[(GD_TSS0 >> 3) + thiscpu->cpu_id] = SEG16(STS_T32A, (uint32_t) (&(thiscpu->cpu_ts)),
153                                 sizeof(struct Taskstate) - 1, 0);
154     gdt[(GD_TSS0 >> 3) + thiscpu->cpu_id].sd_s = 0;
155
156     // Load the TSS selector (like other segment selectors, the
157     // bottom three bits are special; we leave them 0)
158     ltr(GD_TSS0 + sizeof(struct Segdesc) * cpunum());
159
160     // Load the IDT
161     lidt(&idt_pd);
```

不过和网上笔记的代码对比了下，我没太明白158行里边为什么要这样替换，先留个疑问。

## 练习五：

这个练习只要找到调用 `lock_kernel()` 和 `unlock_kernel()` 的合适位置就好了。在练习四题目之前的笔记中已经有详细说明需要在以下四个位置调用：

- 在 `i386_init()` 中，在BSP唤醒其他CPU之前获取锁。注释很详细，代码如下：

```
51    // Acquire the big kernel lock before waking up APs
52    // Your code here:
53    lock_kernel();
54
55    // Starting non-boot CPUs
56    boot_aps();
```

- 在 `mp_main()` 中，初始化AP后获取锁，然后调用 `sched_yield()` 开始在此AP上运行环境。 `sched_yield()` 尚未实现，代码如下：

```
115    // Now that we have finished some basic setup, call sched_yield()
116    // to start running processes on this CPU.  But make sure that
117    // only one CPU can enter the scheduler at a time!
118    //
119    // Your code here:
120    lock_kernel();
121    sched_yield();
```

- 在 `trap()` 中，在从用户模式陷入时获取锁。要确定在用户模式还是内核模式下发生陷入，请检查 `tf_cs` 的低位。检查已经给出，注释很详细，代码如下：

```
285    if ((tf->tf_cs & 3) == 3) {
286        // Trapped from user mode.
287        // Acquire the big kernel lock before doing any
288        // serious kernel work.
289        // LAB 4: Your code here.
290        lock_kernel();
291        assert(curenv);
```

- 在 `env_run()` 中，在切换到用户模式之前释放锁。不要太早或太晚，否则可能导致死锁。代码中注释有提示， `env_pop_tf()` 的作用之一就是切换到用户模式，在其之前解锁即可，代码如下：

```
587    unlock_kernel();
588    env_pop_tf(&curenv->env_tf);
```

## 练习六：

首先要按照注释的提示写好循环，注意开头是当前环境的ID或者0，结尾是循环了一遍回到起始点，循环体里判断环境状态是否为 `ENV_RUNNABLE` ，最后还要判断当前环境状态是不是 `ENV_RUNNING` 。我刚开始写完适中会引发中断异常，对比了一下网上的代码，发现网上的代码是从 `start` 开始判断的，但是注释里说starting just after the env this CPU was last running，我不太清楚原因，但还是为了不报异常修改了代码。最终代码如下：

```
31      // LAB 4: Your code here.
32      idle = thiscpu->cpu_env;
33      uint32_t start = ((idle != NULL) ? ENVX(idle->env_id) : 0);
34      uint32_t i = start;
35
36      while (true) {
37          if (envs[i].env_status == ENV_RUNNABLE){
38              env_run(&envs[i]);
39              return;
40          }
41          i = (i == NENV ? 0 : i + 1);
42          if (i == start)
43              break;
44      }
45
46      if (idle != NULL && idle->env_status == ENV_RUNNING) {
47          env_run(idle);
48          return;
49      }
50
51      // sched_halt never returns
52      sched_halt();
```

然后要修改 `kern/syscall.c` 里的 `syscall()`，多加一种 `case`， `case` 的名称和调用的函数分别参照 `inc/syscall.h` 和 `kern/syscall.c` 就可以了，代码如下：

```
287         case SYS_yield:
288             sys_yield();
289             return 0;
```

接着要在 `mp_main` 里调用 `sched_yield()`，这一步在上一个练习添加内核锁的时候已经顺便加上了。

最后要在 `kern/init.c` 里创建三个运行 `user/yield.c` 的环境，我当时想的是 `env_create()` 函数，然后看到 `init.c` 里边用的是 `ENV_CREATE()`，是在 `env.h` 里定义的一个宏，具体操作原理没有细看，但是效果和 `env_create()` 差不多，所以就直接仿照源代码给的形式修改了一下，代码如下：

```
61      // Touch all you want.
62      //ENV_CREATE(user_primes, ENV_TYPE_USER);
63      ENV_CREATE(user_yield, ENV_TYPE_USER);
64      ENV_CREATE(user_yield, ENV_TYPE_USER);
65      ENV_CREATE(user_yield, ENV_TYPE_USER);
```

然后按照题目要求进行测试即可。

## 练习七：

这个练习我个人觉得不太难，尤其是注释里已经给的很清楚，但是还是比较耗时间的，最后还是稍微参考了一下网上的代码，终于磕磕绊绊完成了。

和注释中说得完全一样，需要实现的五个系统调用函数中最主要的部分就是错误代码的判断，内部需要调用的函数在注释中给的很清楚，在 `pmap.c` 和 `env.c` 找到对应的函数读一下前边的注释和参数的含义就可以了，比较头疼的是错误情况的判断，有些情况我能看懂但是不知道怎么转换成代码。以下对上述系统调用作一一详述：

- `sys_exofork`

  这个函数比较简单，错误代码只可能有两个 `-E_NO_FREE_ENV` 和 `-E_NO_MEM` ，而且恰好就是函数内部调用的 `env_alloc()` 的两个错误返回值，所以只需要判断 `env_alloc()` 返回得对不对就可以了。代码如下：

```
// 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.
    int ret;
    struct Env *newenv_store;

    ret = env_alloc(&newenv_store, curenv->env_id);
    if (ret != -E_NO_FREE_ENV && ret != -E_NO_MEM) {
        newenv_store->env_status = ENV_NOT_RUNNABLE;
        newenv_store->env_tf = curenv->env_tf;
        newenv_store->env_tf.tf_regs.reg_eax = 0;
        ret = newenv_store->env_id;
    }

    return ret;
}
```

- `sys_env_set_status`

  这个相对也比较简单，错误情况不多，一种是 `-E_BAD_ENV` 表示当前 `envid` 不存在或者当前操作无权对这个环境进行更改，另一种是 `-E_INVAL` 表示当前要修改的环境状态无效（也就是注释中说的不是 `ENV_RUNNABLE` 和 `ENV_NOT_RUNNABLE` 中的某一个）。对于 `-E_BAD_ENV` 中的无权修改，我实在不清楚怎么去判断，后来看了下网上的代码发现把这一条“省略”了，我以为是题目的问题，后来才发现 `envid2env` 本来就包含了权限不够情况的处理，所以 `-E_BAD_ENV` 整体只需要判断函数 `env2envid` 返回对不对就行了。代码如下：

```
109 static int
110 sys_env_set_status(envid_t envid, int status)
111 {
112     // Hint: Use the 'envid2env' function from kern/env.c to translate an
113     // envid to a struct Env.
114     // You should set envid2env's third argument to 1, which will
115     // check whether the current environment has permission to set
116     // envid's status.
117
118     // LAB 4: Your code here.
119     int ret;
120     struct Env *env_store;
121
122     envid2env(envid, &env_store, 1);
123                                             // **LIRONGJIA**
124     if (env_store != NULL) {                // Succeed on envid2env.
125         if (status == ENV_RUNNABLE ||       //
126             status == ENV_NOT_RUNNABLE) {   // Valid status, return 0 and
127             ret = 0;                        // set status.
128             env_store->env_status = status; //
129         }                                   //
130         else                                // Invalid status, return -E_INVAL.
131             ret = -E_INVAL;                 //
132     } else                                  // Not Succeed on envid2env,
133         ret = -E_BAD_ENV;                   // return -E_BAD_ENV.
134                                             // *************
135     return ret;
136 }
```

- `sys_page_alloc`

  这个函数就比较复杂了，错误情况很多，需要自己写代码判断。所有需要判断的情况如下：

  - `-E_BAD_ENV`：这一情况和 `sys_env_set_status` 一样，只需判断 `envid2env()` 即可，不再赘述
  - `-E_INVAL`： `va >= (void *)UTOP` 或 `PGOFF(va)`，第二种情况没太明白，大概能明白意思
  - `-E_INVAL`： `perm` 权限有误。对权限参数的要求注释中给得很详细，但是一致不太清楚这种某一位满足条件怎么去转换成代码，不太会用按位与和按位或
  - `-E_NO_MEM`：这个错误代码就是函数 `page_insert()` 返回得错误代码

  最终代码如下：

```
169 static int
170 sys_page_alloc(envid_t envid, void *va, int perm)
171 {
172     // Hint: This function is a wrapper around page_alloc() and
173     //   page_insert() from kern/pmap.c.
174     //   Most of the new code you write should be to check the
175     //   parameters for correctness.
176     //   If page_insert() fails, remember to free the page you
177     //   allocated!
178
179     // LAB 4: Your code here.
180     int ret;
181     struct Env *env_store;
182     struct PageInfo *pp;
183                                                     // **LIRONGJIA**
184     envid2env(envid, &env_store, 1);                //
185     if (env_store == NULL)                          // Fail on envid2env,
186         return -E_BAD_ENV;                          // return -E_BAD_ENV.
187                                                     //
188     if (va >= (void *)UTOP ||                       // If va >= UTOP or
189         PGOFF(va))                                  // va is not page-aligned (???),
190         return -E_INVAL;                            // return -E_INVAL.
191                                                     //
192     if ((perm & PTE_U) == 0 ||                      // PTE_U must be set.
193         (perm & PTE_P) == 0 ||                      // PTE_P must be set.
194         (perm & ~(PTE_P | PTE_U | PTE_AVAIL | PTE_W)) != 0) // no other bits may be set ???,
195         return -E_INVAL;                            // otherwise, return -E_INVAL.
196                                                     //
197     pp = page_alloc(1);                             //
198     ret = page_insert(env_store->env_pgdir,         //
199                       pp, va, perm);                //
200     if (ret != 0) {                                 // Fail on page_insert,
201         page_free(pp);                              // free the page and
202         return -E_NO_MEM;                           // return -E_NO_MEM.
203     }                                               // **************
204
205     return 0;
206 }
```

- `sys_page_map`

  这个是最复杂的一个，不过有一些情况在之前已经遇到过，如下：

  - `-E_BAD_ENV`：这个情况只需判断 `envid2env()`，但是因为这里有 `srcenvid` `dstenvid` 对应的两个环境，所以要两个都判断
  - `-E_INVAL`： `srcva >= (void *)UTOP` 或 `PGOFF(srcva)` 或 `dstva >= (void *)UTOP` 或 `PGOFF(dstva)`，道理一样，也是两个判断
  - `-E_INVAL`： `srcva` 虚拟地址不在 `srcenvid` 映射的地址空间内，这个情况注释中有提示，调用函数 `page_lookup()` 检索一下就可以
  - `-E_INVAL`：权限有误，这个，照搬就行
  - `-E_INVAL`：如果 `perm & PTE_W` 而且 `srcva` 是只读的情况。这个按照原话直接翻译成代码就可以了，后半句要把 `pte_store` 和 `PTE_W`（可写）按位与一下再取个反
  - `E_NO_MEM`：这个情况也和 `sys_page_alloc` 里边的判断一样，判断函数 `page_insert()` 返回对不对就行了

  最终代码如下：

```
224 static int
225 sys_page_map(envid_t srcenvid, void *srcva,
226          envid_t dstenvid, void *dstva, int perm)
227 {
228     // Hint: This function is a wrapper around page_lookup() and
229     //   page_insert() from kern/pmap.c.
230     //   Again, most of the new code you write should be to check the
231     //   parameters for correctness.
232     //   Use the third argument to page_lookup() to
233     //   check the current permissions on the page.
234
235     // LAB 4: Your code here.
236
237     struct Env *srcenv_store, *dstenv_store;
238     struct PageInfo *pp;
239     pte_t *pte_store;
240     int ret;
241
242     envid2env(srcenvid, &srcenv_store, 1);
243     envid2env(dstenvid, &dstenv_store, 1);
244     if (srcenv_store == NULL || dstenv_store == NULL)
245         return -E_BAD_ENV;
246
247     if (srcva >= (void *)UTOP || PGOFF(srcva) ||
248         dstva >= (void *)UTOP || PGOFF(dstva))
249         return -E_INVAL;
250
251     pp = page_lookup(srcenv_store->env_pgdir, srcva, &pte_store);
252     if (pp == NULL)
253         return -E_INVAL;
254
255     if ((perm & PTE_U) == 0 ||
256         (perm & PTE_P) == 0 ||
257         (perm & ~(PTE_P | PTE_U | PTE_AVAIL | PTE_W)) != 0)
258         return -E_INVAL;
259
260     if ((perm & PTE_W) && !(*pte_store & PTE_W))
261         return -E_INVAL;
262
263     ret = page_insert(dstenv_store->env_pgdir, pp, dstva, perm);
264     if (ret != 0)
265         return -E_NO_MEM;
266
267     return 0;
```

- `sys_page_unmap`

  最后一个函数的错误情况都是前边见过的，类似地往下搬就可以了，代码如下：

```
278 static int
279 sys_page_unmap(envid_t envid, void *va)
280 {
281     // Hint: This function is a wrapper around page_remove().
282
283     // LAB 4: Your code here.
284     struct Env *env_store;
285
286     envid2env(envid, &env_store, 1);
287     if (env_store == NULL)
288         return -E_BAD_ENV;
289
290     if (va >= (void *)UTOP || PGOFF(va))
291         return -E_INVAL;
292
293     page_remove(env_store->env_pgdir, va);
294
295     return 0;
296 }
```

最后 `make run-dumbfork` 或者 `make grade` 测试就行了。

## 练习八：

这个练习和练习七道理一样，判断错误情况，内部进行一定处理就行了，比较简单，代码如下：

```
147 sys_env_set_pgfault_upcall(envid_t envid, void *func)
148 {
149     // LAB 4: Your code here.
150     struct Env *env_store;
151 
152     envid2env(envid, env_store, 1);
153     if (env_store == NULL)
154         return -E_BAD_ENV;
155 
156     env_store->env_pgfault_upcall = func;
157     return 0;
158 }
```

然后在 `syscall()` 里添加相应的 `case` 即可。

## 练习九：

这个练习大致上怎么写注释里已经有所提示了，但是有些细节的地方自己把握得还不好。按照注释的提示，首先入如果当前环境存在页面故障处理函数，那么就要设置 `UTrapframe` 和一系列操作。但在设置 `UTrapframe` 前还要判断当前的环境是不是本身已经是异常的了，如果是就得先向堆栈里压一个空32位，也就是4个字。怎么判断当前的环境在不在异常堆栈题目前也有说明。然后就可以设置 `UTrapframe` 里的各字段了，最后恢复执行就行了。

大致流程没问题，但是各种测试都不对，而且注释最后的提示我没搞懂哪里有用，最后还是参考了下网上的代码。自己的代码有以下两个问题：

- 没有任何预防措施，异常情况没有考虑到，导致部分测试程序通不过
- 没有对用户环境进行修改（提示中的要求）

经过修正，最终代码如下：

```
410     // LAB 4: Your code here.
411     if (curenv->env_pgfault_upcall) {
412
413         struct UTrapframe *utf;
414         uintptr_t utfstart;
415
416         if (tf->tf_esp <= UXSTACKTOP - 1 &&
417             tf->tf_esp >= UXSTACKTOP - PGSIZE)
418             utfstart = tf->tf_esp - 32 / 8;
419         else
420             utfstart = UXSTACKTOP;
421
422         utf = (struct UTrapframe *)(utfstart - sizeof(struct UTrapframe));
423
424         user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe), PTE_U | PTE_W);
425
426         utf->utf_fault_va   = fault_va;
427         utf->utf_err        = tf->tf_trapno;
428         utf->utf_regs       = tf->tf_regs;
429         utf->utf_eip        = tf->tf_eip;
430         utf->utf_eflags     = tf->tf_eflags;
431         utf->utf_esp        = tf->tf_esp;
432
433         tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
434         tf->tf_esp = (uint32_t)utf;
435
436         env_run(curenv);
437
438     }
439
440     // Destroy the environment that caused the fault.
441     cprintf("[%08x] user fault va %08x ip %08x\n",
442         curenv->env_id, fault_va, tf->tf_eip);
443     print_trapframe(tf);
444     env_destroy(curenv);
445 }
```

## 练习十：

汇编语言没什么基础，感觉注释里说得挺清楚但还是一头雾水，只能直接看代码了。目前能看懂每一步在干什么，代码注释也有说明，但是不太清楚目的是什么，只能先照搬代码，后期有想法再回头看了。代码如下：

```
67      // LAB 4: Your code here.
68      movl 48(%esp), %ebp
69      subl $4, %ebp
70      movl %ebp, 48(%esp)
71      movl 40(%esp), %eax
72      movl %eax, (%ebp)
73
74      // Restore the trap-time registers.  After you do this, you
75      // can no longer modify any general-purpose registers.
76      // LAB 4: Your code here.
77      addl $8, %esp
78      popal
79
80      // Restore eflags from the stack.  After you do this, you can
81      // no longer use arithmetic operations or anything else that
82      // modifies eflags.
83      // LAB 4: Your code here.
84      addl $4, %esp
85      popfl
86
87      // Switch back to the adjusted trap-time stack.
88      // LAB 4: Your code here.
89      popl %esp
90
91      // Return to re-execute the instruction that faulted.
92      // LAB 4: Your code here.
93      ret
```

## 练习十一：

这个练习调用之前已经写好的函数就行了。注释说得很清楚：如果 `_pgfault_handler` 等于0表示还没有处理函数，先分配一个页面，然后把 `pfentry.s` 里的 `_pgfault_upcall` 设成处理函数就行了。分配页面调用 `sys_page_alloc()` ，设置处理函数调用 `sys_env_set_pgfault_upcall()` 。代码如下：

```
24 void
25 set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
26 {
27      int r;
28
29      if (_pgfault_handler == 0) {
30          // First time through!
31          // LAB 4: Your code here.
32          void *va = (void *) (UXSTACKTOP - PGSIZE);
33          envid_t envid = sys_getenvid();
34
35          sys_page_alloc(envid, va, PTE_P | PTE_U | PTE_W);
36          sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
37      }
38
39      // Save handler pointer for assembly to call.
40      _pgfault_handler = handler;
41 }
```

## 练习十二：

练习之前步骤的讲解我觉得比较明确，感觉自己对流程比较清楚，但是具体写的时候还是一直跑不对，不得

已还是看了网上的代码，和之前的问题一样，还是一些细节上的毛病。比如：

- 权限位只顾 `PTE_COW` 和 `PTE_W` 了，把最基本的 `PTE_P` 和 `PTE_U` 给忘了
- `fork.c` 里不能直接使用 `curenv` 和 `thiscpu` ，所以要靠 `sys_getenvid()` 获取当前环境ID
- 权限位获取可以通过 `upvt` 和 `upvd` 获取

注意上述问题，然后根据提示依次调用之前已经写好的系统调用，除了一些地方目前不太明白（比如子环境ID是0的时候为什么要那样处理）。以下依次为 `pgfault` `duppage` 和 `fork` 函数的代码：

```
14 static void
15 pgfault(struct UTrapframe *utf)
16 {
17     void *addr = (void *) utf->utf_fault_va;
18     uint32_t err = utf->utf_err;
19     int r;
20     envid_t curenvid = sys_getenvid();
21
22     // Check that the faulting access was (1) a write, and (2) to a
23     // copy-on-write page.  If not, panic.
24     // Hint:
25     //   Use the read-only page table mappings at uvpt
26     //   (see <inc/memlayout.h>).
27
28     // LAB 4: Your code here.
29     if (!(err & FEC_WR) && !(uvpt[PGNUM(addr)] & PTE_COW))
30         panic("pgfault error: wrong faulting access");
31
32     // Allocate a new page, map it at a temporary location (PFTEMP),
33     // copy the data from the old page to the new page, then move the new
34     // page to the old page's address.
35     // Hint:
36     //   You should make three system calls.
37
38     // LAB 4: Your code here.
39     sys_page_alloc(curenvid, PFTEMP, PTE_W | PTE_U | PTE_P);
40
41     addr = (void *)(PGNUM(addr) * PGSIZE);
42     memmove(PFTEMP, addr, PGSIZE);
43
44     sys_page_unmap(curenvid, addr);
45     sys_page_map(curenvid, PFTEMP, curenvid, addr, PTE_W | PTE_U | PTE_P);
46     sys_page_unmap(curenvid, PFTEMP);
47 }
```

```
60 static int
61 duppage(envid_t envid, unsigned pn)
62 {
63     int r;
64     int perm;
65     envid_t curenvid = sys_getenvid();
66
67     // LAB 4: Your code here.
68     if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW))
69         perm = PTE_COW;
70
71     r = sys_page_map (curenvid, (void *)(pn * PGSIZE),
72                 envid, (void *)(pn * PGSIZE), perm | PTE_U | PTE_P);
73     if (r < 0) {
74         panic("duppage error: %e", r);
75         return r;
76     }
77
78     r = sys_page_map (curenvid, (void *)(pn * PGSIZE),
79                 curenvid, (void *)(pn * PGSIZE), perm | PTE_U | PTE_P);
80     if (r < 0) {
81         panic("duppage error: %e", r);
82         return r;
83     }
84
85     return 0;
86 }
```

```
104 envid_t
105 fork(void)
106 {
107     // LAB 4: Your code here.
108     extern void _pgfault_upcall();
109     envid_t new_envid, curenvid;
110
111     curenvid = sys_getenvid();
112     set_pgfault_handler(pgfault);
113     new_envid = sys_exofork();
114
115     if (new_envid < 0)
116         panic("fork error: %e", new_envid);
117     else if (new_envid == 0)
118         thisenv= &envs[ENVX(curenvid)];
119     else {
120
121         for(uint32_t addr = UTEXT; addr < USTACKTOP; addr += PGSIZE){
122             if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
123                 duppage(new_envid, PGNUM(addr));
124         }
125
126         sys_page_alloc(new_envid, (void *)UXSTACKTOP - PGSIZE, PTE_P|PTE_U|PTE_W);
127         sys_env_set_pgfault_upcall(new_envid, _pgfault_upcall);
128         sys_env_set_status(new_envid, ENV_RUNNABLE);
129
130     }
131
132     return new_envid;
133     //panic("fork not implemented");
134 }
```

## 练习十三：

这个练习比较简单，由两个人任务组成，一是初始化为IRQ 0～15提供处理程序，二是设置运行环境的 `FL_IF` 标志。

第一个任务和之前给各种中断异常添加处理程序一样，在 `trapentry.s` 和 `trap.c` 中作添加即可，一方面注意没有错误代码所以都用 `TRAPHANDLER_NOEC` ，另一方面因为是用户进程间的通信，所以 `SETGATE` 权限等级应该是0。

第二个任务根据题目中的提示和题目之前的讲解，在 `env.c` 的 `env_alloc()` 修改 `env` 的 `%eflag` 寄存器即可，怎么访问可以查阅 `struct Env` 定义，不过应该比较熟悉能猜出来了。 `trapentry.s` `trap.c` 和 `env.c` 中添加代码如下：

- `trapentry.s`

```
73 TRAPHANDLER_NOEC(t_irq0, IRQ_OFFSET + 0);
74 TRAPHANDLER_NOEC(t_irq1, IRQ_OFFSET + 1);
75 TRAPHANDLER_NOEC(t_irq2, IRQ_OFFSET + 2);
76 TRAPHANDLER_NOEC(t_irq3, IRQ_OFFSET + 3);
77 TRAPHANDLER_NOEC(t_irq4, IRQ_OFFSET + 4);
78 TRAPHANDLER_NOEC(t_irq5, IRQ_OFFSET + 5);
79 TRAPHANDLER_NOEC(t_irq6, IRQ_OFFSET + 6);
80 TRAPHANDLER_NOEC(t_irq7, IRQ_OFFSET + 7);
81 TRAPHANDLER_NOEC(t_irq8, IRQ_OFFSET + 8);
82 TRAPHANDLER_NOEC(t_irq9, IRQ_OFFSET + 9);
83 TRAPHANDLER_NOEC(t_irq10, IRQ_OFFSET + 10);
84 TRAPHANDLER_NOEC(t_irq11, IRQ_OFFSET + 11);
85 TRAPHANDLER_NOEC(t_irq12, IRQ_OFFSET + 12);
86 TRAPHANDLER_NOEC(t_irq13, IRQ_OFFSET + 13);
87 TRAPHANDLER_NOEC(t_irq14, IRQ_OFFSET + 14);
88 TRAPHANDLER_NOEC(t_irq15, IRQ_OFFSET + 15);
```

- `trap.c`

```
38 void t_irq0();
39 void t_irq1();
40 void t_irq2();
41 void t_irq3();
42 void t_irq4();
43 void t_irq5();
44 void t_irq6();
45 void t_irq7();
46 void t_irq8();
47 void t_irq9();
48 void t_irq10();
49 void t_irq11();
50 void t_irq12();
51 void t_irq13();
52 void t_irq14();
53 void t_irq15();
```

```
132     SETGATE(idt[IRQ_OFFSET + 0],    0, GD_KT, t_irq0,   0);
133     SETGATE(idt[IRQ_OFFSET + 1],    0, GD_KT, t_irq1,   0);
134     SETGATE(idt[IRQ_OFFSET + 2],    0, GD_KT, t_irq2,   0);
135     SETGATE(idt[IRQ_OFFSET + 3],    0, GD_KT, t_irq3,   0);
136     SETGATE(idt[IRQ_OFFSET + 4],    0, GD_KT, t_irq4,   0);
137     SETGATE(idt[IRQ_OFFSET + 5],    0, GD_KT, t_irq5,   0);
138     SETGATE(idt[IRQ_OFFSET + 6],    0, GD_KT, t_irq6,   0);
139     SETGATE(idt[IRQ_OFFSET + 7],    0, GD_KT, t_irq7,   0);
140     SETGATE(idt[IRQ_OFFSET + 8],    0, GD_KT, t_irq8,   0);
141     SETGATE(idt[IRQ_OFFSET + 9],    0, GD_KT, t_irq9,   0);
142     SETGATE(idt[IRQ_OFFSET + 10],   0, GD_KT, t_irq10,  0);
143     SETGATE(idt[IRQ_OFFSET + 11],   0, GD_KT, t_irq11,  0);
144     SETGATE(idt[IRQ_OFFSET + 12],   0, GD_KT, t_irq12,  0);
145     SETGATE(idt[IRQ_OFFSET + 13],   0, GD_KT, t_irq13,  0);
146     SETGATE(idt[IRQ_OFFSET + 14],   0, GD_KT, t_irq14,  0);
147     SETGATE(idt[IRQ_OFFSET + 15],   0, GD_KT, t_irq15,  0);
```

- `env.c`

```
285     // Enable interrupts while in user mode.
286     // LAB 4: Your code here.
287     e->env_tf.tf_eflags |= FL_IF;
```

## 练习十四：

这个练习是在练习十三的基础上调用 `sched_yield()` 处理时钟中断，在 `trap_dispatch()` 里添加一种情况判断就好了。

不过刚开始我把新增的时钟中断的情况直接当作和 `T_SYSCALL` 和 `T_BRKPT` 等同都放到了 `switch-case` 语句里边。但是后来看到处理时钟中断的判断实际上是安排在 `switch-case` 后边的，这个判断的位置虽然并不影响代码执行测试，但是仔细读一下注释可以发现 `switch-case` 语句中处理的都是处理器中断异常的部分，而时钟中断属于硬件中断，这样安排我觉得对于代码的重写和阅读什么的还是比较有益的。

增加部分代码如下：

```
275     // Handle clock interrupts. Don't forget to acknowledge the
276     // interrupt using lapic_eoi() before calling the scheduler!
277     // LAB 4: Your code here.
278     if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
279         lapic_eoi();
280         sched_yield();
281         return;
282     }
```

## 练习十五：

这个练习的代码部分在注释中有非常详细的提示，按照步骤写即可完成，比较简单。

首先是两个新增的系统调用 `sys_ipc_try_send()` 和 `sys_ipc_recv()` ，代码如下：

```
342 sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
343 {
344     // LAB 4: Your code here.
345     struct Env *dstenv;
346     struct PageInfo *pp;
347     pte_t *pte_store;
348     int ret;
349     unsigned to_perm = 0;
350 
351     envid2env(envid, &dstenv, 0);
352     if (dstenv == NULL)
353         return -E_BAD_ENV;
354 
355     if (!dstenv->env_ipc_recving)
356         return -E_IPC_NOT_RECV;
357 
358     if (srcva < (void *)UTOP) {
359         if (PGOFF(srcva))
360             return -E_INVAL;
361 
362         if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0 ||
363             (perm & ~(PTE_P | PTE_U | PTE_AVAIL | PTE_W)) != 0)
364             return -E_INVAL;
365 
366         pp = page_lookup(curenv->env_pgdir, srcva, &pte_store);
367         if (pp == NULL)
368             return -E_INVAL;
369 
370         ret = page_insert(dstenv->env_pgdir, pp, dstenv->env_ipc_dstva, perm);
371         if (ret != 0)
372             return -E_NO_MEM;
373         to_perm = perm;
374     }
375 
376     if ((perm & PTE_W) && !(*pte_store & PTE_W))
377         return -E_INVAL;
378 
379     dstenv->env_ipc_recving = false;
380     dstenv->env_ipc_from = curenv->env_id;
381     dstenv->env_ipc_value = value;
382     dstenv->env_ipc_perm = to_perm;
383     dstenv->env_status = ENV_RUNNABLE;
384     dstenv->env_tf.tf_regs.reg_eax = 0;
385     return 0;
386 }

399 static int
400 sys_ipc_recv(void *dstva)
401 {
402     // LAB 4: Your code here.
403     if (dstva < (void *)UTOP) {
404         if (PGOFF(dstva) != 0)
405             return -E_INVAL;
406         curenv->env_ipc_dstva = dstva;
407     }
408 
409     curenv->env_ipc_recving = true;
410     curenv->env_status = ENV_NOT_RUNNABLE;
411     curenv->env_ipc_from = curenv->env_id;
412 
413     return 0;
414 }
```

然后是两个分别调用上述系统调用的库函数 `ipc_send()` 和 `ipc_recv()`，注释中也有详细的完成步骤和必要的提示，代码如下：

```
59 void
60 ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
61 {
62     // LAB 4: Your code here.
63     int r;
64
65     if (pg == NULL)
66         pg = (void *)-1;
67
68     while (true) {
69         r = sys_ipc_try_send(to_env, val, pg, perm);
70         if (r >= 0)
71             return;
72         else if (r != -E_IPC_NOT_RECV)
73             panic("ipc_send error: %e", r);
74
75         sys_yield();
76     }
77     //panic("ipc_send not implemented");
78 }


22 int32_t
23 ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
24 {
25     // LAB 4: Your code here.
26     int r;
27
28     if (pg != NULL)
29         r = sys_ipc_recv(pg);
30     else
31         r = sys_ipc_recv((void *)-1);
32
33     if (r < 0) {
34
35         if (from_env_store != NULL)
36             *from_env_store = 0;
37         if (perm_store != NULL)
38             *perm_store = 0;
39         return r;
40
41     } else {
42
43         if (from_env_store != NULL)
44             *from_env_store = thisenv->env_ipc_from;
45         if (perm_store != NULL)
46             *perm_store = thisenv->env_ipc_perm;
47         return thisenv->env_ipc_value;
48     }
49 }
```