**Lab 4: Preemptive Multitasking / 抢占式多任务处理**

介绍

在本次实验中，你将会在同时运行的多个用户进程（environment，下同）中实现抢占式多任务处理。

在第一部分，Part A，你将会为 JOS 系统添加多处理器支持，实现轮转调度（round-robin scheduling），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）；

第二部分，Part B，你将会实现类 Unix 的 fork() 方法，以允许用户进程创造自身的拷贝；

最终，在 Part C 你将为 JOS 提供进程间通信 (Inter-Process Communication) 支持，允许不同的用户进程显式地彼此交流和同步。你也将会实现硬件时钟中断和抢占。

Lab4 包含一些新的代码文件

kern/cpu.h 内核私有的关于多处理器支持的定义

kern/mpconfig.c 用于读取多处理器配置的代码

kern/lapic.c 驱动每个处理器 Local APIC(LAPIC) 单元的内核代码

kern/mpentry.S 非启动 CPU 的汇编入口

kern/spinlock.h 内核私有的自旋锁的定义，包括全局内核锁 (big kernel lock)

kern/spinlock.c 实现自旋锁的内核代码

kern/sched.c 将由你实现的调度方法的代码框架

# Part A: Multiprocessor Support and Cooperative Multitasking / 多处理器支持和协作式多任务

在本次实验的第一部分，你将首先拓展 JOS，使其支持在具有多个处理器的系统上运行，接下来，实现一些新的 JOS 内核系统调用以允许用户进程创建额外的新进程。你还需要实现 协作式轮转调度 (Cooperative round-robin scheduling)，允许内核在当前用户进程自愿放弃CPU或退出时切换到另一个进程。在之后的 Part C 你将实现 抢占式调度，以允许内核在一定时间后从一个用户进程中抢占CPU的控制权，即使用户进程不愿合作。

**多处理器支持**

我们将使 JOS 支持 symmetric multiprocessing (SMP)，这是一种所有 CPU 均同等地享有系统资源（例如内存和I/O总线）的多处理器模型。虽然所有的 CPU 在 SMP 模型中功能均相同，在启动过程中它们被分为两种类型：bootstrap processor (BSP) 和 application processors (APs)。前者负责初始化系统和引导操作系统，后者只有在操作系统正常运行后才被前者激活。哪一个处理器会成为 BSP 是由硬件和 BIOS 决定的。直到目前为止，你的所有 JOS 代码均运行在 BSP 上。

在 SMP 系统中，每一个 CPU 均有一个伴随的 局部APIC(LAPIC)单元（APIC，Advanced Programmable Interrupt Controller，高级可编程中断控制器）。LAPIC单元 负责在整个系统中分发中断。同时，每个 LAPIC 也为它连接的 CPU 提供一个唯一的身份标识。在本次实验中，我们会利用以下 LAPIC 单元的基本功能（在 kern/lapic.c 中）：

读取 LAIPC身份标识 (APIC ID) 以分辨我们的代码在哪个 CPU 上运行 (cpunum())

从 BSP 向 APs 发送 STARTUP处理器间中断 (Interprocesser Interrupt, IPI) 以激活其他 CPU (lapic\_startap())

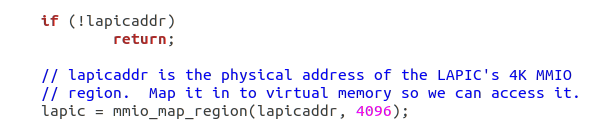
在 Part C，我们为 LAPIC 内置的计时器编程，以触发时钟中断来实现抢占式多任务处理。

处理器通过 映射于内存的输入输出 (memory-mapped I/O, MMIO) 来访问它的 LAPIC。在 MMIO 模式中，物理内存的一部分被硬连线于一些 I/O 设备的寄存器上（译注：I/O 设备的寄存器和内存被映射在物理内存的一些区域）， 所以通常用于访问内存的存取指令也可以同样用于访问设备寄存器。在之前的实验中，你已经在物理内存地址 0xA0000 的位置遇到过一个 IO hole（我们通过它写入 VGA 的显示缓冲区）。LAPIC 被连接在物理地址 0xFE000000 (距 4GB 还有 32MB 的位置) 的 I/O hole 上，这个地址对我们当前在 KERNBASE 上直接映射来说太高了， 不过 JOS 虚拟内存映射表 在 MMIOBASE 位置留了 4MB 的空隙，所以我们可以将这样的设备映射到这里。 之后的实验将引入更多的 MMIO 区域，因此，你将需要编写一个简单的函数，为这一区域分配内存，并将设备内存映射在上面。

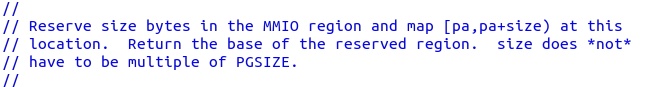
练习 1.

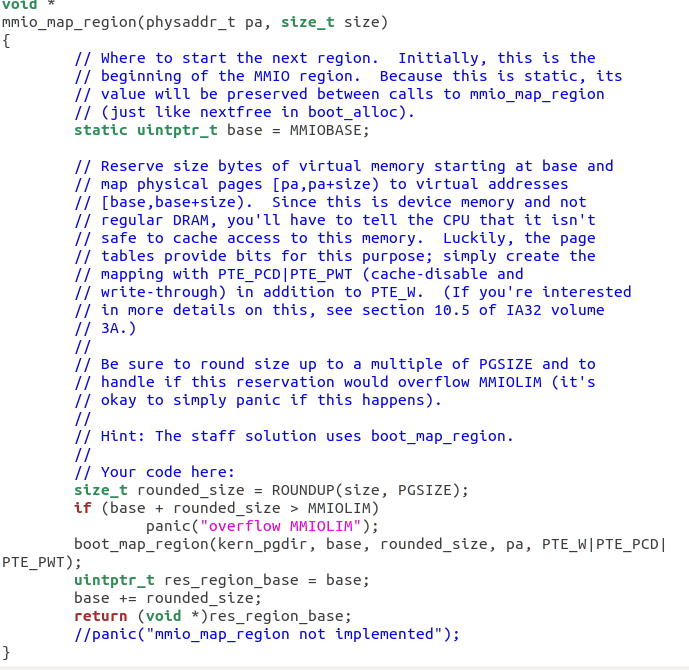
实现在 kern/pmap.c 中的 mmio\_map\_region 方法。你可以看看 kern/lapic.c 中 lapic\_init 的开头部分，了解一下它是如何被调用的。你还需要完成接下来的练习，你的 mmio\_map\_region 才能够正常运行。

Lapic\_init 的开头部分



调用函数 mmio\_map\_region()，将从 lapicaddr 开始的 4kB 物理地址映射到虚拟地址，并返回其起始地址。其中 lapicaddr 是 LAPIC 的 4K MMIO（内存映射 IO）区域的物理地址





我们首先要将大小舍入为 PGSIZE 的倍数，所以利用了 ROUNDUP 函 数。之后判断该预留空间是否会导致溢出，即大于 MMIOLIM 的限制，如果发生 那就 panic。之后就利用 boot\_map\_region 函数来建立相应空间之间的映射即可。 再更新base 值，便于下一次的分配，最后将目前的首地址返回。

**Application Processor Bootstrap**

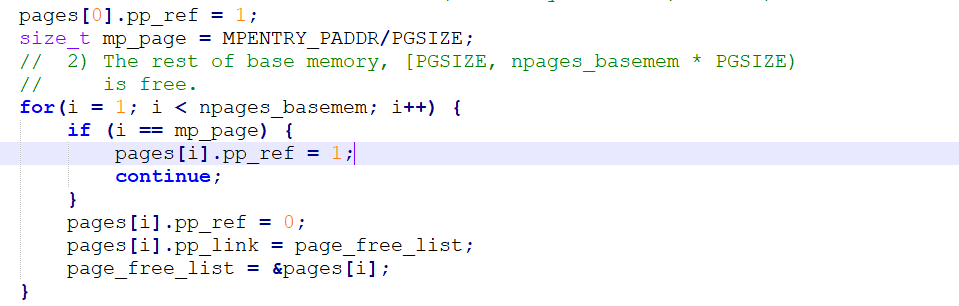
在启动 AP 之前，BSP 应当首先收集多处理器系统的信息，例如，CPU总数，他们的 APIC ID，和 LAPIC单元 的 MMIO 地址。在 kern/mpconfig.c 中的 mp\_init() 函数通过读取 BIOS 存储区域的 多处理器配置表(MP configuration table) 来获得相关信息。

在 kern/init.c 的 boot\_aps() 函数驱动 AP 的引导过程。 AP 从实模式开始启动，就像 在 boot/boot.S 中的 bootloader 一样。所以 boot\_aps() 将 AP 的入口代码 ( kern/mpentry.S ) 拷贝到一个实模式中能够访问到的内存地址。与 bootloader 不同的是，我们可以控制 AP 从哪里开始执行代码。在这里我们把入口代码拷贝到了 0x7000 (MPENTRY\_PADDR)，不过其实 640KB 以下任何一个没有使用的、按页对齐的物理内存均可使用。

而后，boot\_aps() 通过发送 STARTUP IPI （interprocesser interrupt, 处理器间中断） 并提供一个初始 CS:IP （AP 入口代码的位置，我们这里是 MPENRTY\_PADDR ） 给对应 AP 的 LAPIC 单元 ，依次激活每个 AP。 kern/mpentry.S 中的入口代码和 boot/boot.S 中的十分相似。在一些简单的处理后，它将 AP 置于保护模式，并启用页表， 接着调用 C 语言的设置例程 mp\_main() （也在 kern/init.c 中）。boot\_aps() 会等待 AP 在 它的 struct CpuInfo 中设置 cpu\_status 字段为 CPU\_STARTED 后才开始唤醒下一个 AP。

练习 2.

阅读 kern/init.c 中的 boot\_aps() 和 mp\_main() 方法，和 kern/mpentry.S 中的汇编代码。确保你已经明白了引导 AP 启动的控制流执行过程。接着，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。你的代码应当通过我们更新过的 check\_page\_free\_list() 测试，不过可能仍会在我们更新过的 check\_kern\_pgdir() 测试中失败，我们接下来将解决这个问题。



问题 1.

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？换句话说，如果我们忽略掉 kern/mpentry.S 哪里会出现问题呢？ 提示：回忆一下我们在 Lab 1 讨论的链接地址和装载地址的不同之处

MPBOOTPHYS 这个宏定义的目的：将高地址（虚拟地址）转换为低地址（物理地址）

之所以在 kern/mpentry.S 中是必要的而在 boot.S 中是不必要的，是因 为 boot.S 本身就是被加载在实模式可以寻址的地方。而且此时还未开启分页， 可以直接指定程序开始执行的地址，所以不需要这个宏定义来帮助寻址。而在调 用 mpentry.S 时，主 CPU 已经处于保护模式之下了，所以没办法直接指定物理地址，但是在AP的保护模式打开之前，是没有办法使用高地址的，所以，必须使用该宏定义来将线性地址映射到物理地址中。

**CPU 私有状态和初始化**

在编写一个支持多处理器的系统时，将 每个 CPU 各自私有的状态 和 与整个系统共享的公共状态 区别开来是很重要的。kern/cpu.h 定义了大多数 CPU 私有的状态，包括 struct CpuInfo，它存储着 CPU 私有的变量。cpunum() 总是返回调用它的 CPU 的ID, 可以用它来作为数组索引访问诸如 cpus 这样的数组。另外，宏定义 thiscpu 是访问当前 CPU 的 struct CpuInfo 结构的简写。

下面是你应当知道的每个 CPU 私有的状态：

每个CPU的内核堆栈(kernel stack) 因为多个CPU可以同时陷入内核，我们需要为每个 CPU 分别提供内核堆栈以防止它们互相干扰彼此的运行。 percpu\_kstacks[NCPU][KSTKSIZE] 数组为 NCPU 的内核堆栈预留了空间。 在 Lab 2 中，你把 bootstack 指向的内存映射到了紧邻 KSTACKTOP 的下面。相似地，在本次实验中，你会把每个 CPU 的内核堆栈映射到这里，同时，在每个内核堆栈之间会留有一段 守护页 作为它们之间的缓冲区。CPU 0 的堆栈仍然会从 KSTACKTOP 向下生长， CPU 1 的堆栈会在 CPU0 栈底的 KSTKGAP 以下开始向下生长，以此类推。 inc/memlayout.h 展示了内存应当如何映射。

每个CPU的任务状态段(task state segment, TSS)和任务段描述符(TSS descriptor) 每个 CPU 也需要各自的 TSS 以指定 CPU 的内核堆栈在何处。 CPU i 的 TSS 存储在 cpus[i].cpu\_ts 中，相应的 TSS descriptor 在 GDT 入口 gdt[(GD\_TSS0 >> 3) + i] 中被定义。在 kern/trap.c 中定义的全局变量 ts 此时将不再有用。

每个CPU的当前进程指针 因为每一个 CPU 可以同时运行不同的用户进程，我们重新定义了宏 curenv 来指代 cpus[cpunum()].cpu\_env (或者 thiscpu->cpuenv)，指向 当前 运行在当前 CPU 的进程。

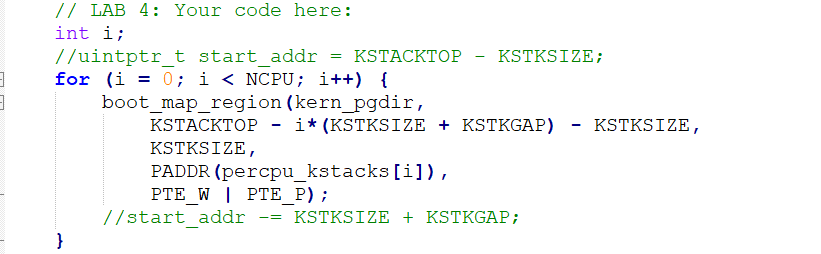
每个CPU的系统寄存器 所有寄存器，包括系统寄存器，都是CPU私有的，因此，初始化这些寄存器的指令，例如，lcr3()，ltr()，lgdt()，lidt() 等，都应当在每个 CPU 中执行一次。 函数 env\_init\_percpu() 和 trap\_init\_percpu() 就是为了这一目的而定义的。

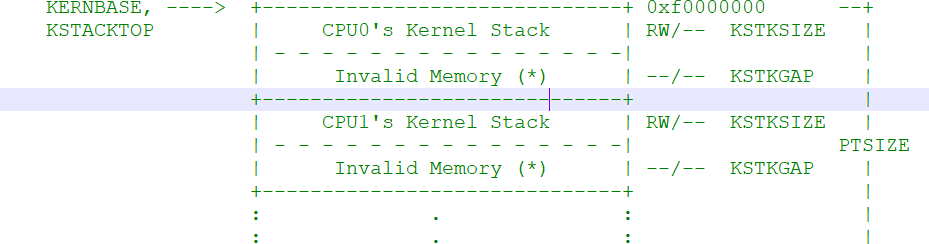
练习 3.

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 。现在，你的代码应当能够通过我们新的 check\_kern\_pgdir() 测试了。

修改mem\_init\_mp()，将内核栈线性地址映射到percpu\_kstacks处的物理地址处。

解决：按照inc/memlayout.h中的结构进行映射即可。

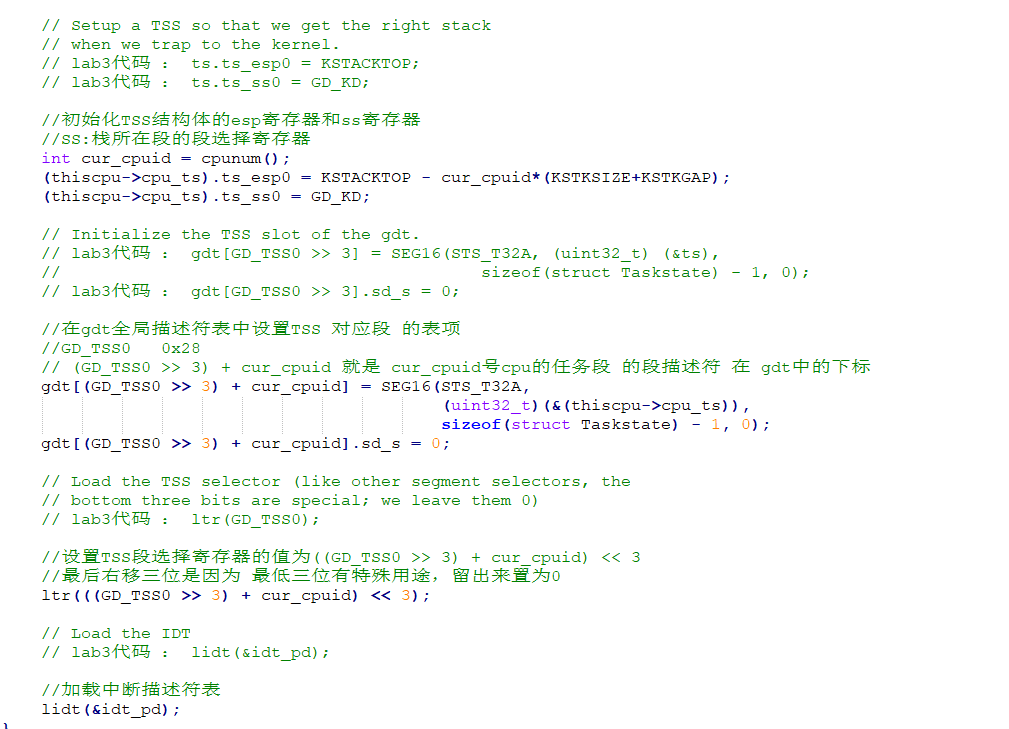




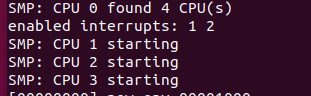
练习 4.

位于 kern/trap.c 中的 trap\_init\_percpu() 为 BSP 初始化了 TSS 和 TSS描述符，它在 Lab 3 中可以工作，但是在其他 CPU 上运行时，它是不正确的。修改这段代码使得它能够在所有 CPU 上正确执行。（注意：你的代码不应该再使用全局变量 ts。）

此处的示例代码为 CPU 0 设置任务状态段（TSS）和 TSS 描述符。 但是如果我们在其他 CPU 上运行它是不正确的，因为每个 CPU 都有自己的内核 堆栈。修复代码，使其适用于所有 CPU。 提示： - 宏“thiscpu”总是指当前 CPU 的 struct CpuInfo; - 当前 CPU 的 ID 由 cpunum（）或 thiscpu-> cpu\_id 给出; - 使用“thiscpu-> cpu\_ts”作为当前 CPU 的 TSS，而不是全局“ts”变量; - 对于 CPU i 的 TSS 描述符使用 gdt [（GD\_TSS0 >> 3）+ i]; - 您在 mem\_init\_mp（）中映射了每 CPU 内核堆栈 ltr 在 TSS 选择器中设置了 一个'busy'标志， 因此如果您不小心在多个 CPU 上加载了相同的 TSS，则会出现三重故障。如 果您将单个 CPU 的 TSS 设置为错误，则在尝试从该 CPU 上的用户空间返回之 前，可能不会出现故障。



因为目的是将原先的只能在 CPU0 上运行修改为可以在其他 CPU 上运行，为 他们初始化 TSS 和 TSS 描述符。所以我们需要先获取当前的 CPU 的编号，也就 是代码中的 cur\_cpuid 这个变量。利用的函数是在 cpu.h 的 cpunum()，用于获取 当前的 CPU 的 id 号。而下面需要设置的是 ts\_esp0 和 ts\_ss0，其中 ts\_esp0 指的 是当前 CPU 的栈顶，根据上个练习我们可以知道栈顶为 KSTACKTOP - i \* （KSTKSIZE + KSTKGAP）（其中 i 是 id）。，而下面要设置的 ts\_ss0 是栈所在段的 段寄存器，此处值不需要修改。下面按照原格式在 gdt 全局描述符表中设置 TSS 对应段的表项，只需要将 ts 替换即可。按照注释的提示，需要注意的是最后利用 ltr()函数时需要将 cpu\_id 左移三位，需要将底部这特殊的三位置 0



**加锁**

目前我们的代码在 mp\_main() 初始化完 AP 就不再继续执行了。在允许 AP 继续运行之前，我们需要首先提到当多个 CPU 同时运行内核代码时造成的 竞争状态 (race condition) ，为了解决它，最简单的办法是使用一个 全局内核锁 (big kernel lock)。这个 big kernel lock 是唯一的一个全局锁，每当有进程进入内核模式的时候，应当首先获得它，当进程回到用户模式的时候，释放它。在这一模式中，用户模式的进程可以并发地运行在任何可用的 CPU 中，但是最多只有一个进程可以运行在内核模式下。其他试图进入内核模式的进程必须等待。

kern/spinlock.h 的 kernel lock 声明了这个全局内核锁，并提供了 lock\_kernel() 和 unlock\_kernel() 两个方法来方便获得和释放锁。你应当在以下 4 个位置使用全局内核锁：

i386\_init() 中，在 BSP 唤醒其他 CPU 之前获得内核锁

mp\_main() 中，在初始化完 AP 后获得内核锁，接着调用 sched\_yield() 来开始在这个 AP 上运行进程。

trap() 中，从用户模式陷入(trap into)内核模式之前获得锁。你可以通过检查 tf\_cs 的低位判断这一 trap 发生在用户模式还是内核模式（译注：Lab 3 中曾经使用过这一检查）

env\_run() 中，恰好在 回到用户进程之前 释放内核锁。不要太早或太晚做这件事，否则可能会出现竞争或死锁。

练习 5.

在上述提到的位置使用内核锁，加锁时使用 lock\_kernel()， 释放锁时使用 unlock\_kernel()。

首先，在 i386\_init()函数中，在 BSP 唤醒 AP 之前，利用 lock\_kernel()上锁； 之后在 mp\_main()函数中初始化完 AP 后，利用 lock\_kernel()上锁，调用 sched\_yield()函数在 AP 上运行进程；之后再 trap()函数中，如果用户态发生 trap 时，获得锁，当时在 lab3 中用到一句 if 来判断 trap 是发生在用户态还是内核态， 此时在判断为用户态时调用 lock\_kernel()；在 env\_run()中，我们会切换到用户模 式（在 env.c 文件中的 env\_pop\_if()函数的 iret 指令之后），此时在切换前利用 unlock\_kernel()，释放锁防止竞争或死锁。

问题 2.

看起来使用全局内核锁能够保证同一时段内只有一个 CPU 能够运行内核代码。既然这样，我们为什么还需要为每个 CPU 分配不同的内核堆栈呢？请描述一个即使我们使用了全局内核锁，共享内核堆栈仍会导致错误的情形。

在某进程即将陷入内核态的时候（尚未获得锁），其实在 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，就可能会导致信息错误。

**轮转调度算法**

你的下一个任务是修改 JOS 内核以使其能够以 轮转 的方式在多个进程中切换。JOS 的轮转调度算法像这样工作：

kern/sched.c 中的 sched\_yied() 函数负责挑选一个进程运行。它从刚刚在运行的进程开始，按顺序循环搜索 envs[] 数组（如果从来没有运行过进程，那么就从数组起点开始搜索），选择它遇到的第一个处于 ENV\_RUNNABLE（参考 inc/env.h）状态的进程，并调用 env\_run() 来运行它。

sched\_yield() 绝不应当在两个CPU上同时运行同一进程。它可以分辨出一个进程正在其他CPU（或者就在当前CPU）上运行，因为这样的进程处于 ENV\_RUNNING 状态。

我们已经为你实现了新的系统调用 sys\_yield()，用户进程可以调用它来触发内核的 sched\_yield() 方法，自愿放弃 CPU，给其他进程运行。

练习 6.

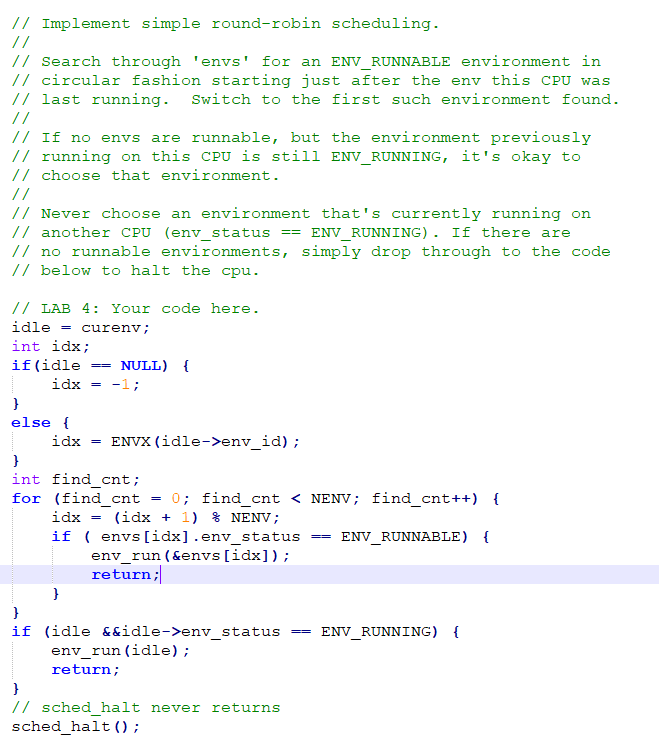
按照以上描述，实现 sched\_yield() 轮转算法。不要忘记修改你的 syscall() 将相应的系统调用分发至 sys\_yield()（译注：以后还要添加新的系统调用，同样不要忘记修改 sys\_yield()）。

确保你在 mp\_main 中调用了 sched\_yield()。

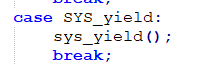
修改你的 kern/init.c 创建三个或更多进程，运行 user/yield.c。 运行 make qemu。 你应当看到进程在退出之前会在彼此之间来回切换 5 次。

轮转算法

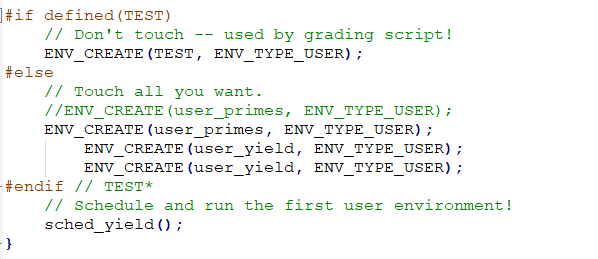
首先判断当前的进程是否是 NULL，如果是那么就将 idx 设为-1，如果非空就 将 idx 设为当前进程的 id。之后在 env 中进行查找，从上次运行的 env 之后开始 对 envs 数组进行查找（上次 env 运行的位置就是 idx）如果查找到有状态是 ENV\_RUNNABLE 的，那就调用 env\_run 执行找到的第一个；如果没有就寻找先 前在此 CPU 上运行的状态为 UNV\_RUNNING 的进程，并且运行它。如果都没有， 就运行 sched\_halt()。

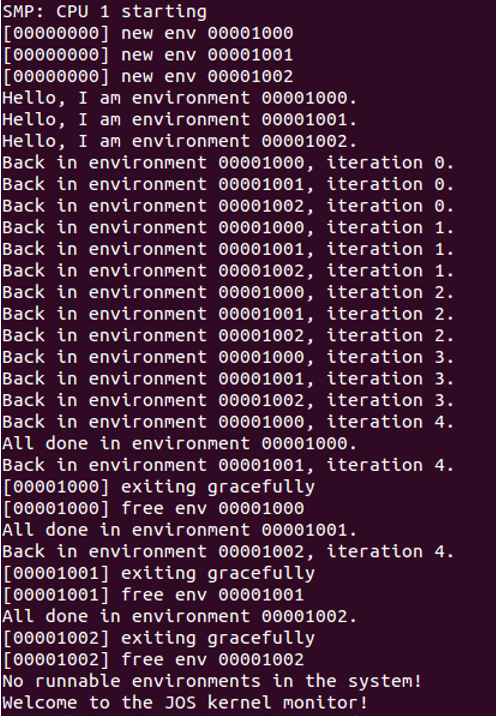


在kern/syscall.c中的syscall（）中添加新的系统调用



在kern/init.c创建更多进程

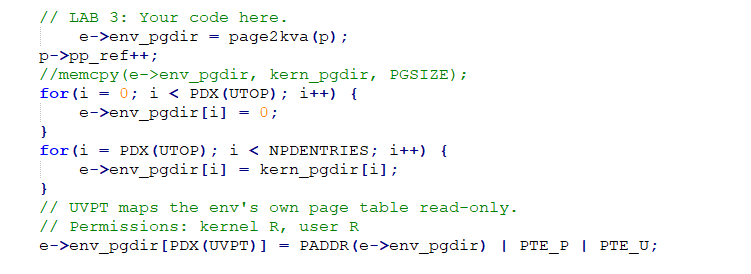




问题 3.

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？

因为之前在 lab3 中利用函数 env\_setup\_vm()初始化了环境 e 的内核虚拟内 存分布，分配了页面目录，相应地设置 e-> env\_pgdir，并初始化新环境的地址 空间的内核部分。因此即使地址上下文发生了变化，前后两个页目录的 e 的地址 实际上是映射到同一个物理地址的。所以，e 还是可以被正确的解引用。



问题 4.

无论何时，内核在从一个进程切换到另一个进程时，它应当确保旧的寄存器被保存，以使得以后能够恢复。为什么？在哪里实现的呢？

因为在进程陷入内核时，会保存当前的运行信息，这些信息都保存在内核栈 上。而当从内核态回到用户态时，会恢复之前保存的运行信息。 具体的实现在上面已经涉及到了，保存信息在 kern/trapentry.S 中，恢复信息 在 kern/env.c 中。

**用于创建进程的系统调用**

尽管你的内核目前能够运行多个用户进程并在其中切换，但仍受限于只能运行由内核创建的进程。现在，你将实现必要的系统调用，使得用户进程也可以创建和启动其他新的用户进程。

UNIX 提供了 fork() 系统调用作为创建进程的原型，UNIX 的 fork() 拷贝整个调用进程（父进程）的地址空间来创建新的进程（子进程），在用户空间唯一可观察到的区别是它们的 进程ID(process ID) 和 父进程ID(parent process ID)（分别是调用 getpid 和 getppid 返回的）。在父进程中, fork() 返回子进程 ID，但在子进程中，fork() 返回 0。默认情况下，每个进程的地址空间是私有的，内存修改对另一方不可见。

你将提供一系列不同的、更原始的系统调用来创建新的用户进程。通过这些系统调用，你将能够完全在用户空间实现类似 Unix 的 fork() 作为其他创建进程方式的补充。你将会为 JOS 实现的新的系统调用包括：

sys\_exofork:

该系统调用创建一个几乎完全空白的新进程：它的用户地址空间没有内存映射，也不可以运行。这个新的进程拥有和创建它的父进程（调用这一方法的进程）一样的寄存器状态。在父进程中，sys\_exofork 会返回刚刚创建的新进程的 envid\_t（或者一个负的错误代码，如果进程分配失败）。在子进程中，它应当返回0。（因为子进程开始时被标记为不可运行，sys\_exofork 并不会真的返回到子进程，除非父进程显式地将其标记为可以运行以允许子进程运行。

sys\_env\_set\_status:

将一个进程的状态设置为 ENV\_RUNNABLE 或 ENV\_NOT\_RUNNABLE。这个系统调用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。

sys\_page\_alloc:

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。

sys\_page\_map:

从一个进程的地址空间拷贝一个页的映射 (不是 页的内容) 到另一个进程的地址空间，新进程和旧进程的映射应当指向同一个物理内存区域，使两个进程得以共享内存。

sys\_page\_unmap:

取消给定进程在给定虚拟地址的页映射。

对于所有以上提到的接受 Environment ID 作为参数的系统调用，JOS 内核支持用 0 指代当前进程的惯例。这一惯例在 kern/env.c 的 envid2env() 函数中被实现。

我们在测试程序 user/dumbfork.c 中提供了一种非常原始的 Unix 样式的 fork()。它使用上述系统调用来创建并运行一个子进程，子进程的地址空间就是父进程的拷贝。接着，这两个进程将会通过上一个练习中实现的系统调用 sys\_yield 来回切换。 父进程在切换 10 次后退出，子进程切换 20 次

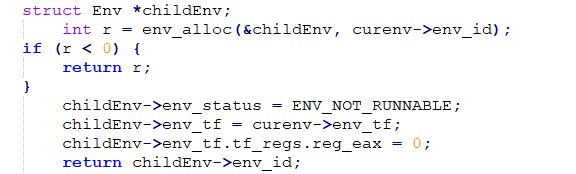
练习 7.

在 kern/syscall.c 中实现上面描述的系统调用。你将需要用到在 kern/pmap.c 和 kern/env.c 中定义的多个函数，尤其是 envid2env()。此时，无论何时你调用 envid2env()，都应该传递 1 给 checkperm 参数。确定你检查了每个系统调用参数均合法，否则返回 -E\_INVAL。 用 user/dumbfork 来测试你的 JOS 内核，在继续前确定它正常的工作。（make run-dumbfork）

用 kern / env.c 中的'envid2env'函数将 envid 转换为 struct Env。

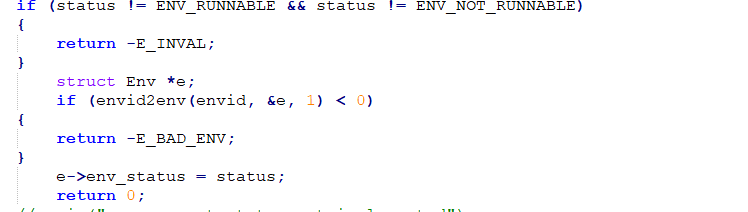
sys\_exofork:

在父进程中, fork() 返回子进程 ID，但在子进程中，fork() 返回 0。



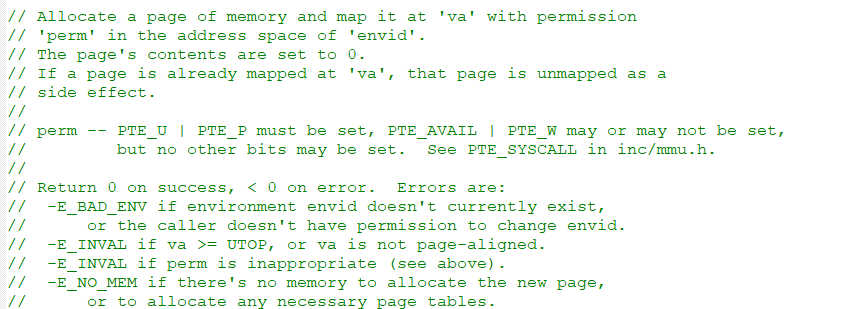
sys\_env\_set\_status:

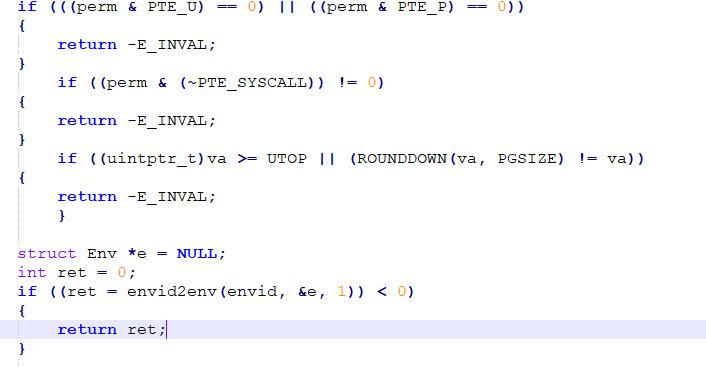
将一个进程的状态设置为 ENV\_RUNNABLE 或 ENV\_NOT\_RUNNABLE。这个系统调用通常用来在新创建的进程的地址空间和寄存器状态已经初始化完毕后将它标记为就绪状态

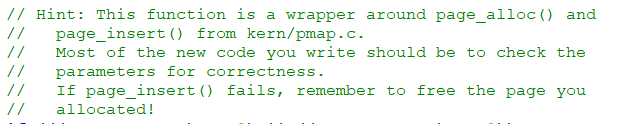


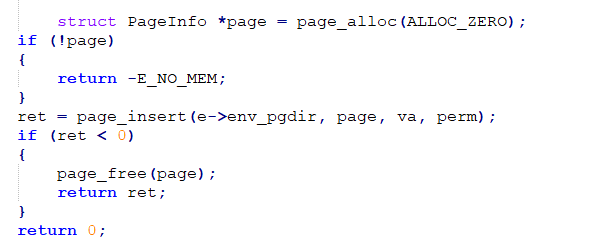
sys\_page\_alloc:

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。





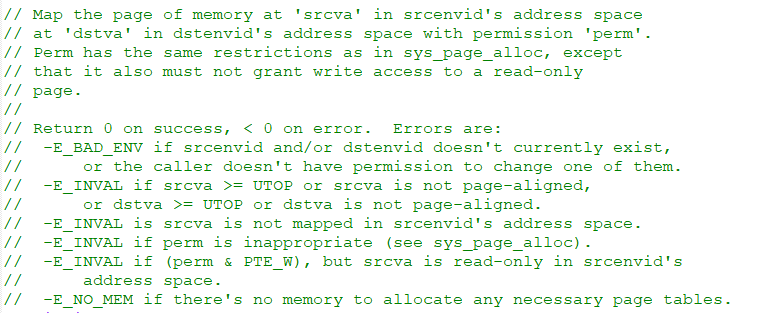


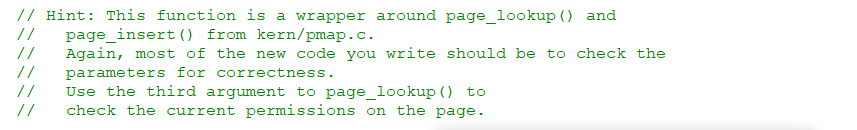


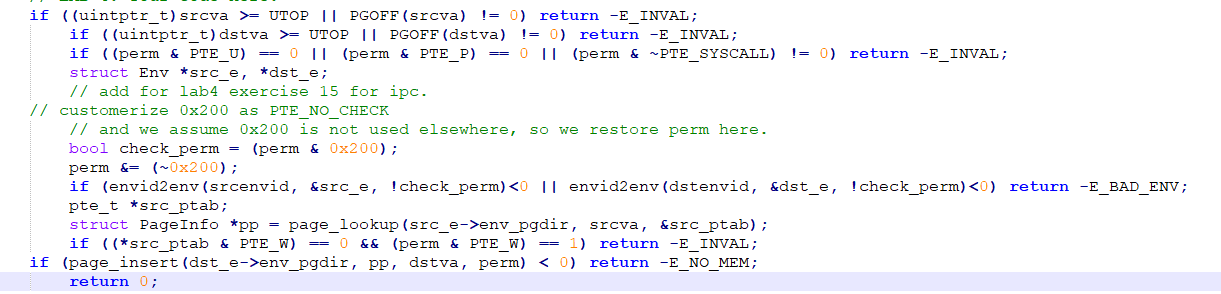
sys\_page\_map:

从一个进程的地址空间拷贝一个页的映射 (不是 页的内容) 到另一个进程的地址空间，新进程和旧进程的映射应当指向同一个物理内存区域，使两个进程得以共享内存。

函数开始是按照注释部分的要求进行的一系列检查，分别是检查地址、检 查权限位、检查进程 id 是否对应一个进程。 在主体部分，首先是利用 page\_lookup()返回虚拟地址对应的物理页，之后 建立一个父进程对应的物理内存到子进程对应的虚拟地址的映射，从而使得两 个进程可以共享内存。

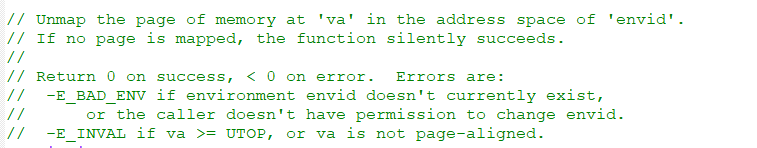


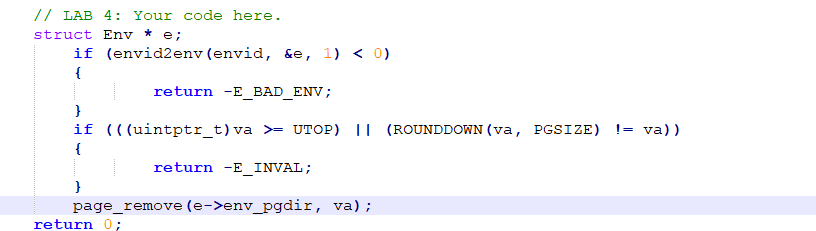




sys\_page\_unmap:

取消给定进程在给定虚拟地址的页映射。





该函数作用是取消进程在给定虚拟地址的页映射，代码很简单，前 面部分是按照注释进行的映射页面判断，判断当前环境是否存在或者权限是否 正确，最后调用之前的 page\_remove()函数即可，将映射取消

测试：

父进程在切换10次后退出，子进程在切换20次后退出



# Part B: Copy-on-Write Fork / 写时复制的派生

如之前所述，Unix提供 fork() 系统调用作为它主要的进程创建的基础操作。fork() 系统调用拷贝父进程的地址空间，来创建一个新进程（子进程）。

xv6 Unix 的 fork() 实现方式是从父进程的页面中将全部数据拷贝给子进程。这个基本上与我们的 dumbfork() 方式相同。将父进程的地址空间拷贝给子进程是这种 fork() 操作最昂贵的部分。

然而，多数情况下，调用 fork() 后紧接着就会在子进程中调用 exec()，这一方法会将子进程的内存完全替换成一个新的进程。比如， shell 通常就这么做。在这种情况下，拷贝父进程的地址空间所花费的大部分时间都被浪费了，因为子进程在调用 exec() 之前只需要一点点它的内存。

因此，随后版本的 Unix 利用虚拟内存硬件的优势，允许父进程和子进程 分享 它们映射在各自地址空间的内存，直到某个进程 修改 它为止。这种技术被称为 写时复制(copy-on-write)。为了实现这一点，调用 fork() 时内核只需从父进程拷贝地址空间的 映射 到子进程，而不是整个映射的页面的内容，与此同时，将复制的页映射标记为只读。当其中之一试图修改这些被分享的页面时，进程产生一个缺页（page fault），此时，Unix 内核意识到这个页面实际上是 虚拟 的拷贝，或者说是 写时复制 的拷贝，所以它为这个缺页的进程创建一个新的、私有的、可写的页面拷贝。通过这种方式，每一个页面的内容直到它们确实被修改时才真正被拷贝。这种优化使得在子进程中调用 fork() 后紧接着调用 exec() 的代价变得小得多：子进程在调用 exec() 之前也许只需要拷贝一个页面（当前进程的栈）。

在本次实验接下来的部分，你将会实现 真正的 ，采取写时复制方式的类 Unix 的 fork() 作为用户空间的调用库例程(user space library routine)。在用户空间实现 fork() 和写时复制支持的好处是，内核仍旧很简单，因此更容易保持正确。它也使得每个用户进程能够定义它们自己的 fork() 策略(semantic)。如果一个进程想要使用一些不同的实现方式（比如，我们在 dumpfork() 中引入的 总是拷贝 版本， 或者使 fork() 之后父进程和子进程完全共享空间），它们可轻易地自行实现。

# 用户模式下的缺页处理

用户模式写时复制版本的 fork() 需要知道那些由在写保护的页面上进行写操作引起的缺页，所以这是你将在这部分首先实现的内容。写时复制只是众多可能的用户缺页处理应用中的一种。

It’s common to set up an address space so that page faults indicate when some action needs to take place. / 通常在我们建立好地址空间后，发生缺页都意味着需要执行一些操作。比如，大多数 Unix 内核一开始只在新的进程的 堆栈 区域映射一个页面，之后，随着进程栈的增长，在尚未映射的内核地址造成缺页时，为其分配并映射更多的栈页面。一个通常的 Unix 内核必须追踪进程在不同内存区域缺页时应当进行什么操作，例如，当缺页发生在栈区域时通常需要分配并映射新的物理页，当发生在程序的 BSS （Block Started by Symbol，用于存储全局静态变量，应当由操作系统初始化为 0）区域时通常需要分配一个新的物理页、用 0 填充并映射它。在 demand-paging （译注：在需要时才在硬盘中读取相应的代码）的系统中，在 text（代码段） 区域发生的缺页会从硬盘中读取相应的页面并映射它。

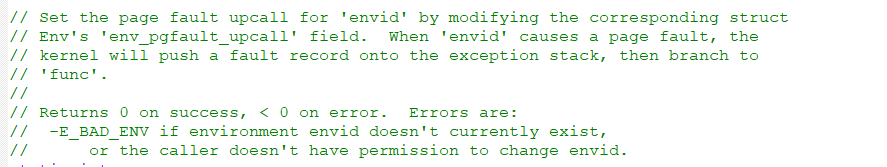
对于内核来说，需要追踪的信息太多了。此处我们不使用传统的 Unix 处理方式，你将在用户空间决定应当如何处理缺页，这样可以使 Bug 的破坏性变得小一些。这种处理方式也使得应用程序在定义其内存区域时拥有更大的灵活性。在随后的实验中你将需要用到用户模式的缺页处理机制，映射并访问以磁盘为基础的文件系统上的文件。

## 设置缺页处理函数

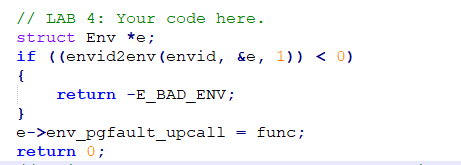
为了处理自己的缺页，用户进程需要向 JOS 内核注册一个 page fault handler entry point 缺页处理函数入口点。 用户进程通过我们新引入的 sys\_env\_set\_pgfault\_upcall 系统调用注册它的缺页处理入口。我们也在 Env 结构体中添加了一个新的成员，env\_pgfault\_upcall，来记录这一信息。

## 练习 8.

实现 sys\_env\_set\_pgfault\_upcall 系统调用。因为这是一个 “危险” 的系统调用，不要忘记在获得目标进程信息时启用权限检查。



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，错误时返回<0。 错误是：-E\_BAD\_ENV如果环境envid当前不存在，或者调用者没有更改envid的权限。



**用户进程的通常堆栈和异常堆栈**

在正常执行时，JOS上的用户进程会在 通常 用户堆栈中运行：它的 ESP 寄存器指向 USTACKTOP 的起点，压入堆栈的数据存储在 [USTACKTOP-PGSIZE, USTACKTOP-1] 这一页面中。然而，当缺页发生在用户模式时，内核会在一个不同的堆栈重新启动用户进程所指定的用户模式缺页处理函数，换句话说，这个堆栈就是 user exception stack / 用户异常栈。大体上讲，我们会让 JOS 内核代表进程实现堆栈的自动切换，这看起来很像是当用户模式向内核模式转换时，x86 处理器实现的堆栈切换那样。

JOS 用户异常堆栈大小也是一个页面，栈顶被定义在虚拟地址 UXSTACKTOP 位置，所以用户异常堆栈可用的字节是 [UXSTACKTOP-PGSIZE, UXSTACKTOP-1]。当运行在这一异常堆栈上时，用户模式的缺页处理函数可以调用 JOS 的常规系统调用来映射新的页面，或者调整映射以修复最初造成缺页的问题。接着用户模式下的缺页处理函数返回，通过一个汇编语言代码段(stub)，返回原始栈上的造成缺页的代码。

每个想要支持用户模式缺页处理的进程需要通过调用 Part A 引入的 sys\_page\_alloc() 为自己的异常堆栈分配内存。

**调用用户缺页处理函数**

你现在需要修改在 kern/trap.c 中的缺页处理代码，像下面这样处理用户模式的缺页。我们将此时缺页的用户进程的状态称为陷入时状态(trap-time state)。

如果没有缺页处理函数被注册，JOS 内核像以前一样，销毁用户进程并打印消息（译注：实际情况要比此处提及的稍复杂一些，请参考对应注释）。否则，内核在用户进程的异常堆栈中构造一个与 inc/trap.h 中的 UTrapframe 一样的 trap frame：

```

<-- 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

```

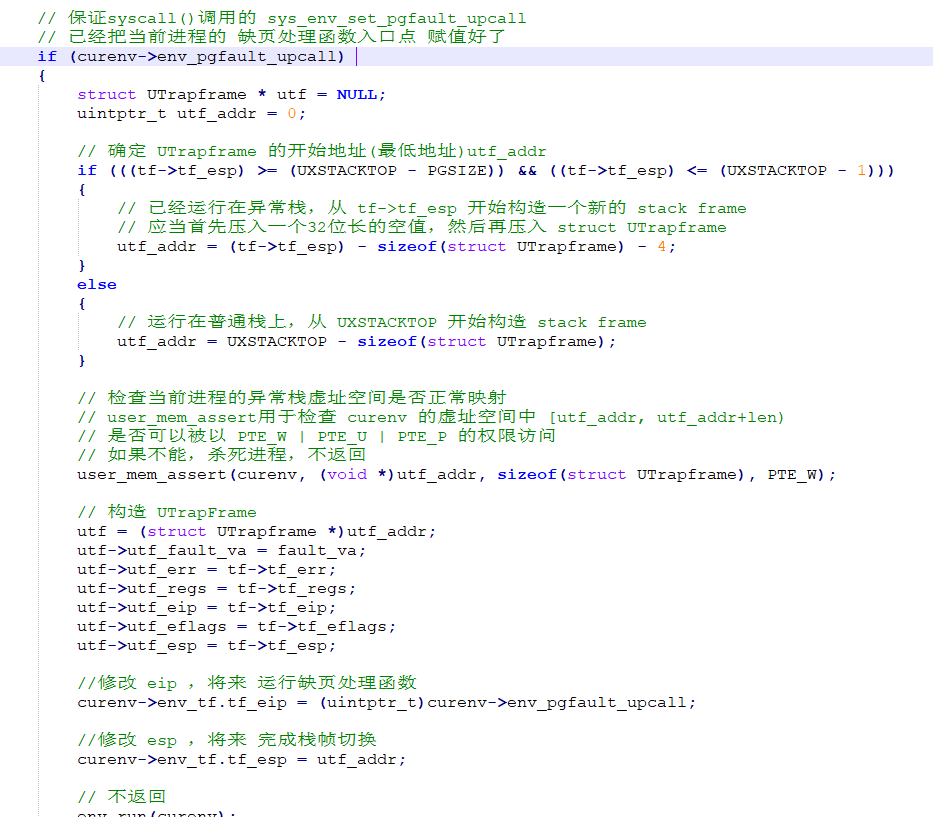
（译注：这个结构比在代码注释中提到的结构更靠谱一点）

内核接下来安排用户进程，使其在异常堆栈上运行它的缺页处理函数，异常处理函数带有一个栈帧(stack frame)作为参数；你应当清楚怎样内核是怎样做到这一点的。 fault\_va 是造成缺页的虚拟地址。

如果用户进程在缺页发生时已经运行在异常堆栈上了，那么缺页处理函数处理自己的缺页异常。在这种情况下，你应当就在当前的 tf->tf\_esp 上构造一个新的 栈帧 (stack frame) 而不是从 UXSTACKTOP 开始构造。你应当首先压入一个空的32位长的值，然后再压入 struct UTrapframe。

## 练习 9.

实现在 kern/trap.c 中的 page\_fault\_handler 方法，使其能够将缺页分发给用户模式缺页处理函数。确认你在写入异常堆栈时已经采取足够的预防措施了。（如果用户进程的异常堆栈已经没有空间了会发生什么？）

先检查处理函数的地址空间是否存在，如果不存在就应将引发错误的env摧毁掉，否则再判断env运行在用户栈还是异常栈，如果是用户栈就将当前状态压入异常栈，是异常栈就隔一段空位再压栈。之后利用 user\_mem\_assert()来判断当前进程是否有权限访问异常栈，接下来是将异常栈 中的各部分复制进去。最后修改 eip 运行 handler，修改 esp 完成栈的切换。 因为 eip 修改了，环境发生改变，env\_run()是不会返回的。 如果用户进程的异常栈没有了空间，根据 inc/memlayout.h 中的栈图可以 知道，异常栈下面是空的页面，所以会出现访问出错的情况。 

## 用户模式缺页入口点

接下来，你需要实现汇编例程(routine)，来调用 C 语言的缺页处理函数，并从异常状态返回到一开始造成缺页中断的指令继续执行。这个汇编例程 将会成为通过系统调用 sys\_env\_set\_pgfault\_upcall() 向内核注册的处理函数。

## 练习 10.

实现在 lib/pfentry.S 中的 \_pgfault\_upcall 例程。返回到一开始运行造成缺页的用户代码这一部分很有趣。你在这里将会直接返回，而不是通过内核。最难的部分是同时调整堆栈并重新装载 EIP。

　\_pgfault\_upcall是所有用户页错误处理程序的入口，在这里调用用户自定义的处理程序，并在处理完成后，从错误栈中保存的UTrapframe中恢复相应信息，然后跳回到发生错误之前的指令，恢复原来的进程运行。

<-- 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

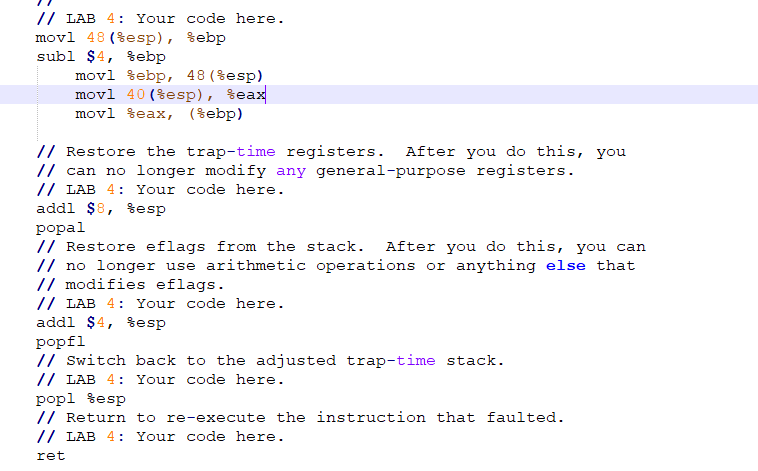
trap-time ebp

trap-time esi

trap-time edi end of struct PushRegs

tf\_err (error code)

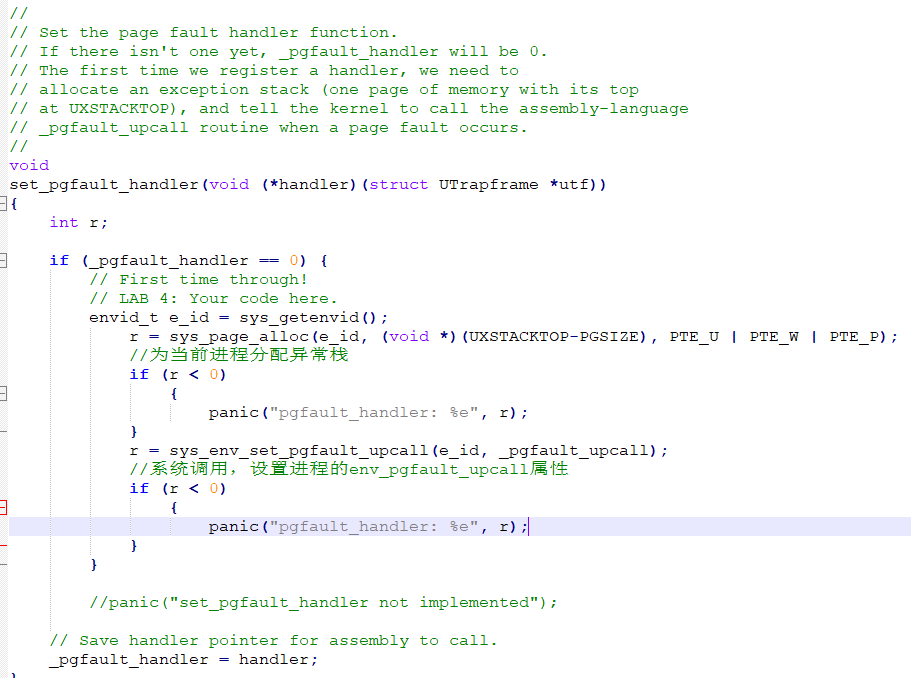
fault\_va <-- %esp when handler is run



在 trap-time esp 上存在一个 4 字节的保留空间，是作为中断递归用的。 即上一次异常的栈顶下面间隔 4 字节，就是一个新的异常。因为一共有 13 项， 所以为了到达顶端的 esp 位置需要移动 48 位，也就是第一句代码所做的，先将 栈顶指针 ebp 指向 utf\_esp 加上保留空间的位置。下面将取出的栈顶减四再存 回栈中，那么如果是中断递归的情况，esp-4 就是保留的 4 个字节地址，如果 不是中断递归的情况，那么就是用户运行栈的栈顶。之后将 eip 指令寄存器取 出，放入上面空出来的保留字节中，以便于后面的恢复运行，原因在注释中提 到了。接下来恢复通用寄存器和 EFLAG 标志寄存器。最后恢复栈顶，利用 pop esp 切换为原来出错程序的运行栈，利用 ret 返回出错程序。

## 练习 11.

完成在 lib/pgfault.c 中的 set\_pgfault\_handler() 。



确保你清楚为什么 user/faultalloc 和 user/faultallocbad 的表现不同。

user/faultalloc 和 user/faultallocbad，两个的 page fault handler 是 一样的，但是两个输出的方式不同，一个是利用系统调用里的 sys\_cput() 输出，另外一个是利用 cprintf()输出。它在调用 sys\_cputs() 之前，首先在 用户态执行了 vprintfmt() 将要输出的字符串存入结构体 b 中。在此过程中 试图访问 0xdeadbeef 地址，触发并处理了页错误（其处理方式是在错误位置 处分配一个字符串，内容是 "this string was faulted in at ..."），因此 在继续调用 sys\_cputs() 时不会出现 panic。

缺页处理小结：

引发缺页中断，执行内核函数链：trap()->trap\_dispatch()->page\_fault\_handler()

page\_fault\_handler()切换栈到用户异常栈，并且压入UTrapframe结构，然后调用curenv->env\_pgfault\_upcall（系统调用sys\_env\_set\_pgfault\_upcall()设置）处代码。又重新回到用户态。

进入\_pgfault\_upcall处的代码执行，调用\_pgfault\_handler（库函数set\_pgfault\_handler()设置）处的代码，最后返回到缺页中断发生时的那条指令重新执行。

# 实现写时复制的 Fork

至此，你已经让内核为 能够在用户空间实现写时复制的 fork() 提供了足够的基本方法。

我们为你在 lib/fork.c 提供了 fork() 方法的骨架。与 dumbfork() 相似，fork() 也应当创建一个新的进程，接下来，扫描整个父进程的地址空间，并在子进程中建立相应的页面映射。最关键的不同之处在于，dumbfork() 拷贝物理页，fork() 最初只拷贝 映射 。 fork() 只有在其中一个进程试图修改页面时才复制它。

fork() 最基本的控制流如下：

父进程安装 pgfault() 作为 C 语言的缺页处理函数。这一步需要使用你在上面实现的 set\_pgfault\_handler()。

父进程调用 sys\_exofork() 创建一个子进程。

对于其地址空间 UTOP 以下每一个可写的或者写时复制的页面，父进程调用一次 duppage，这个函数应当将这些页面在子进程的地址空间映射为写时复制的，同时还要在它自己的地址空间中重新映射为写时复制的。duppage 修改两个进程的 PTE 使得这个页面不再可写，并在 avail 段（译注：PTE的后12位中的某些位）包含 PTE\_COW 以将写时复制的页面与真正的只读页面区分开来。

然而，异常堆栈并不能像这样被重新映射。与此不同，你需要重新在子进程中分配一个新的页面作为其异常堆栈。因为缺页处理函数执行这个复制工作，而缺页处理函数运行在异常堆栈上。如果它被标记成了写时复制，那谁来复制它呢？（译注：这是个反问）

fork()也需要处理那些存在，却不可写或者不是写时拷贝的页面。

父进程设置子进程的缺页处理入口，就像设置自己的一样。

现在子进程已经准备好运行了，所以父进程将其标记为可以运行。

每次进程写入写时复制的页面时，会造成一次缺页。这时缺页中断控制流：

内核将缺页分发给 \_pgfault\_upcall，它会调用 fork() 的 pgfault()。

pgfault() 检查 ①这个缺页是不是写操作（在 error code 中检查 FEC\_WR），②PTE 是否被标记为了 PTE\_COW。如果不是，panic（译注：在这里 panic 说明你的代码在其他地方存在问题，也有可能是缺页处理函数本身的其他地方）。

pgfault() 在临时位置分配一个新的页面，并将造成缺页的页面内容拷贝给这个新的页面。接下来，将新的页面在恰当的地址映射为可读/写，替换原有的只读映射。

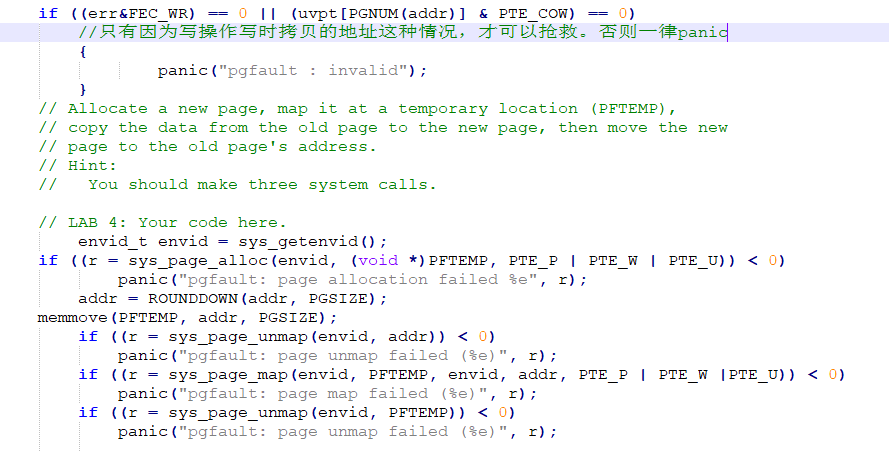
用户模式的 lib/fork.c 代码必须查询进程的页表来执行上面提到的一些操作（例如，得知一个页面的 PTE 是不是被标记为了 PTE\_COW ）。内核将进程的页表映射在了 UVPT 就是为了这一目的。它使用了一种 聪明的映射技巧 使得用户代码查询 PTE 变得更简单。 lib/entry.S 设置了 uvpt 和 uvpd 所以你可以很容易地在 lib/fork.c 中找到页表的信息。

## 练习 12.

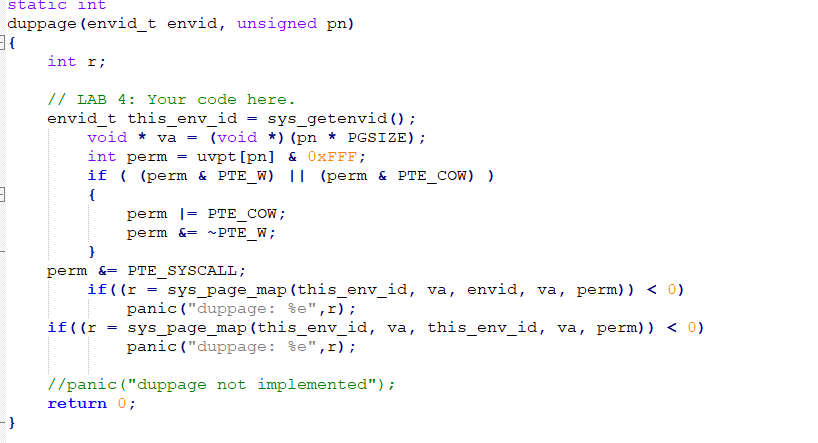
实现在 lib/fork.c 中的 fork，duppage 和 pgfault。 用 forktree 程序来测试你的代码(make run-forktree)。

pgfault() 检查 ①这个缺页是不是写操作（在 error code 中检查 FEC\_WR），②PTE 是否被标记为了 PTE\_COW。如果不是，panic。

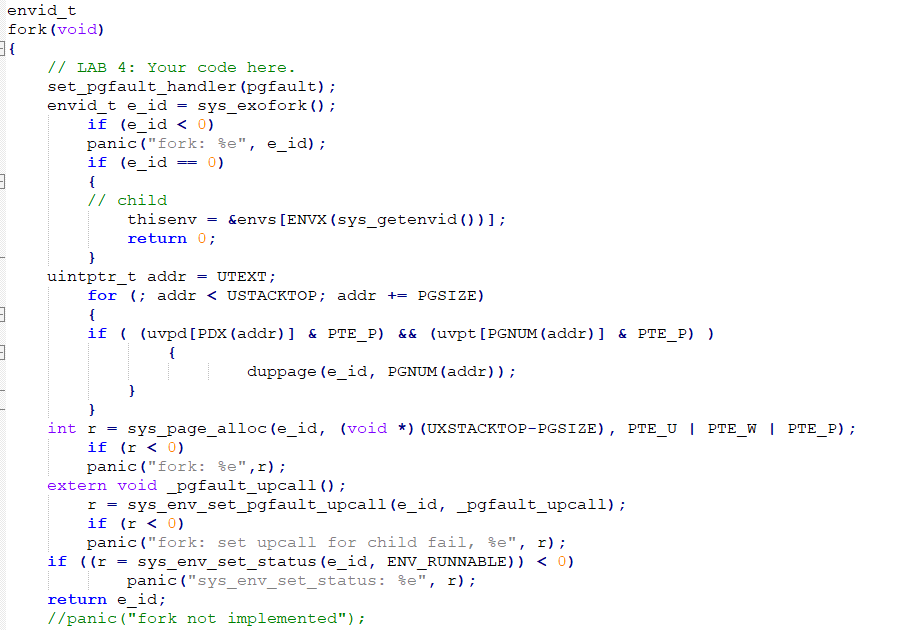
pgfault() 在临时位置分配一个新的页面，并将造成缺页的页面内容拷贝给这个新的页面。接下来，将新的页面在恰当的地址映射为可读/写，替换原有的只读映射。



该函数用于处理 page fault 的写是拷贝。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。之后检查虚拟地址对应的页目 录项是否是 COW 页。再按照注释使用临时位置 PFTEMP，给它先分配一个物理 页，然后通过 memmove() 函数将发生异常的虚拟地址 addr 所在页面拷贝至 PFTEMP，此时有两个物理页保存了同样的内容。之后先解除 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。 使得只有 addr 指向新分配的物理页.



该函数负责进行 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。利用 prem 获取到 父进程页项对于该物理页的特权级后,先判断是否是可写或写时复制的,将其标 志为 COW 页,只读。然后要将权限修改为系统调用的权限再进行系统调用操作。



　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。为子进程设置 page fault upcall，因为 sys\_exofork() 并不会复制父进程的 e->env\_pgfault\_upcall 给子进程。 设置状态

# Part C: Preemptive Multitasking and Inter-Process communication / 抢占式多任务与进程间通信(IPC)

在实验的最后一部分你将修改内核来抢占不合作进程，并允许进程间显式地进行通信。

## 时钟中断和抢占

运行测试程序 user/spin 。这个测试程序创建一个子进程，一旦子进程占据 CPU ，它将永远循环下去。无论是父进程还是内核都无法再重新占据 CPU 。对于保护系统不受 Bug 或者用户模式的恶意进程影响来说，这显然不是一个理想的情况，因为任何一个用户进程，只要无限循环，不再让出 CPU，就会让整个系统宕机。为了允许内核能够抢占 (preemption)一个运行中的进程，强制从其上取得 CPU 的控制权，我们必须拓展我们的 JOS 内核以支持时钟硬件发出的外部硬件中断。

中断原理

外部中断（或者说，设备中断）被称为 IRQ（interrupt request） ，有 16 个可能的 IRQ，编号是从 0 到 15。 将 IRQ 映射到 IDT 入口的方法不是固定的。picirq.c 中的 pic\_init 方法将 0-15 号 IRQ 映射到了 IDT 入口的 IRQ\_OFFSET 到 IRQ\_OFFSET + 15 的位置。

在 inc/trap.h，IRQ\_OFFSET 被定义为 32， 因此 IDT 入口的 32-47 就相应的对应 IRQ 的 0-15。例如，时钟中断是 IRQ 0，所以内核中的 IDT[IRQ\_OFFSET + 0]（即，IDT[32] )包含时钟中断的中断处理函数的地址（译注：这是你在接下来的练习中应当实现的内容）。选择这个 IRQ\_OFFSET 的原因，是因为设备中断不会与处理器中断有所重叠，否则显然会造成困扰。（事实上，在早些日子，个人计算机运行 MS-DOS 时，IRQ\_OFFSET 取值就是 0，这当然为处理设备中断和处理器中断造成了大量困扰。）

在 JOS 中，我们和 xv6 Unix 相比做了关键的简化。在内核运行时，外部设备的中断总是被禁止（和 xv6 一样，在用户空间时启用。）外部中断被处在 %eflags 的 FL\_IF 标志位控制（参考 inc/mmu.h）。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。

你需要确保 FL\_IF 标志位在用户进程运行时总是被设置的，以使得每当中断到达的时候，它会被传入处理器并被你的中断处理代码处理。否则，我们说中断被屏蔽(mask)了，或者说，被忽略了，直到中断被重新打开。我们已经在 bootloader 的一开始屏蔽了中断，到目前为止我们还从未重新打开它。

## 练习 13.

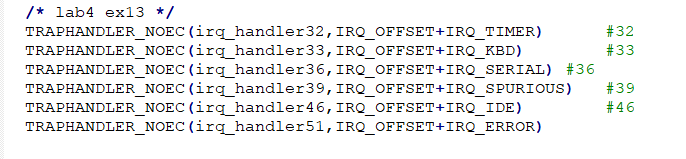
修改 kern/trapenrty.S 和 kern/trap.c 来初始化一个合适的 IDT 入口，并为 IRQ 0-15 提供处理函数。接着，修改 kern/env.c 中的env\_alloc() 以确保用户进程总是在中断被打开的情况下运行。

当调用用户中断处理函数时，处理器从来不会将 error code 压栈，也不会检查IDT 入口的描述符特权等级 (Descriptor Privilege Level, DPL) 。此时你可能需要重新阅读一下 80386 手册 中 9.2 这一部分，或者 IA-32 Intel Architecture Software Developer’s Manual, Volume 3 的 5.8 章节。

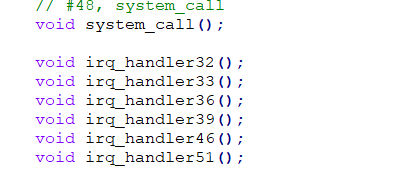
完成这个练习后，当你运行任何一个运行时间较长的测试程序时（比如 make run-spin），你应当看见内核打印硬件中断的 trap frame。因为到目前为止，虽然处理器的硬件中断已经被打开了，但 JOS 还没有处理它，所以你应该注意到，它以为这个中断发生在正在运行的用户进程，并将其销毁。最终当没有进程可以销毁的时候，JOS 会陷入内核监视器。

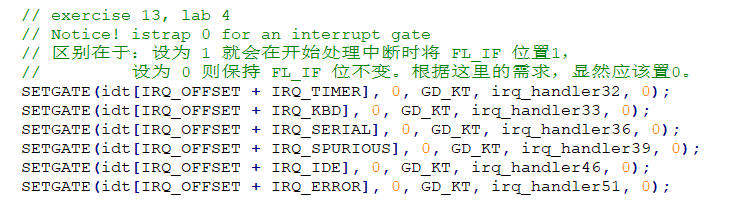
在kern/trapentry.S中加入：

当调用用户中断处理函数时，处理器从来不会将 error code 压栈，也不会检查IDT 入口的描述符特权等级，所以在trapentry.S中使用TRAPHANDLER\_NOEC（）,当某个中断发生时，根据偏移量将对应的中断号压栈，然后开始执行相应的中断处理函数（call trap）：

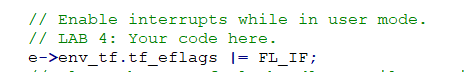


首先声明处理函数，之后使用SETEGATE设置表项。





确保用户进程总是在中断被打开的情况下运行，保证 FL\_IF 被置位，如果这个进程运行时出现中断，中断就可以到达处理器并被相应的中断处理代码所处理。所以在kern/env.c的env\_alloc()中加入：



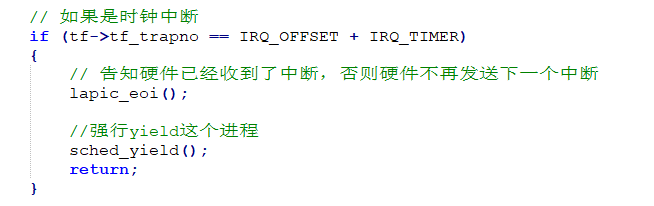
## 处理时钟中断

在测试程序 user/spin 中，子进程一旦运行，它就会不断地循环，内核则不再能重新取得控制权。我们需要为硬件编程，使其定期产生时钟中断，当收到中断时内核将会夺回控制权，从而我们可以切换到不同的用户进程。

我们为你写好的 lapic\_init 和 pic\_init （位于 init.c 的 i386\_init） 函数中设置了时钟和中断控制器来产生中断。你现在需要完成处理这些中断的代码。

## 练习 14.

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。



首先使用 lapic\_eoi()确认中断，之后利用 sched\_yield()来寻找另一个进 程并运行。

你现在应当能让 user/spin 测试程序正常工作了（译注：这里有一个文档中没有提到的细节。如果你发现时钟中断只发生一次就再也不会发生了，你应当再去看看 kern/lapic.c）：父进程会创建子进程，sys\_yield() 会切换到子进程几次，但在时间片过后父进程会重新占据 CPU，并最终杀死子进程并正常退出。

现在，是时候做一些回溯检查 (regression testing)了。确保你启用中断后没有让曾经能够正常运行的部分（比如, forktree）被破坏掉。也应当试试多个 CPU，make CPUS=2 target（译注：使用多个 CPU 有概率导致 user/spin 无法通过测试 可以想一下为什么？）。现在你应该能够通过 stresssched 这个测试程序了。试试运行 make grade 看看是不是这样。你应该能够获得这个实验的 65/75 分了（译注：应该是 65/80）。

## 进程间通信 (IPC)

（技术上来讲在 JOS 中应该是 “环境间通信” 或者说 “IEC”，但是既然所有其他系统都叫它 IPC，我们也会用标准术语。）（译注：为了翻译方便，上文中的进程实际上均为 environment，我猜，也许你已经在 Lab 3 中习惯了把 环境 翻译成 进程了 >.< ）

我们一直着眼于操作系统的独立层面，即认为每一个进程都独立地享有机器的所有资源。但操作系统另一个重要服务是允许程序在它们想要通信的时候互相通信。允许程序与其他程序互动可以非常强大。Unix 的管道模型就是非常古典的例子。

有许多进程间通信的模型，即使直到今天对于哪种模型最好这一问题的仍有很多争议。我们不会与他们争执，而是实现一种简单的 IPC 策略并尝试它。

JOS 的进程间通信

你现在需要实现一些额外的 JOS 系统调用，它们共同提供了一种简单的进程间通信方式。你将会实现两个系统调用 sys\_ipc\_recv 和 sys\_ipc\_try\_send。接下来你会实现两个库封装 (library wrapper) 函数 ipc\_recv 和 ipc\_send。

使用 JOS 的进程间通信策略，用户进程可以互相发送的消息有两种；一个 32 位整数，一个可选的页面映射。允许用户进程在通信时传递页面映射与传递一个 32 位整数相比是一个非常有效的方式，同时也允许了用户进程能够很容易地安排共享内存区域。

发送和接受消息

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，而不是只有特定的进程可以，也不仅限于它的父进程/子进程。换句话说，在 Part A 中你使用过的权限检查在 IPC 过程中就不再有用了，因为 IPC 系统调用经过仔细的设计以保证它是安全的：一个用户进程不会因为发送消息而导致另一个进程错误运行，除非另一个进程也同样存在 Bug。

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。

用户空间的库函数 ipc\_recv 会处理对 sys\_ipc\_recv的调用，并在当前进程的 struct Env 中查找有关收到的消息的一些信息。

与之相似，库函数 ipc\_send 会处理对 sys\_ipc\_try\_send 的重复调用直到信息发送成功。

传递页面

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。

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。这样做的结果是，这个物理页在发送者和接收者之间得以共享。

如果发送者或接收者之一没有提到应当传递页面，那么页面就不会传递。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。

实现进程间通信

## 练习 15.

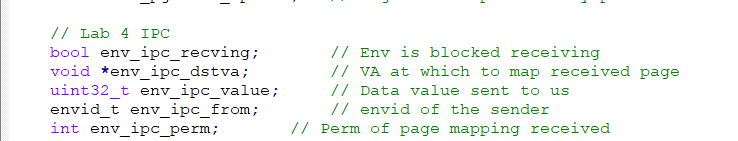
实现 kern/syscall.c 中的 sys\_ipc\_recv 和 sys\_ipc\_try\_send。在实现它们前，你应当读读两边的注释，因为它们需要协同工作。当你在这些例程中调用 envid2env 时，你应当将 checkperm 设置为 0，这意味着进程可以与任何其他进程通信，内核除了确保目标进程 ID 有效之外，不会做其他任何检查。

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。

user/primes.c 用来创建子进程和通信的代码读起来可能很有趣。（译注：可能因为 user/primes 的输出过多，有时无法从 QEMU 输出串口读取全部的输出，测试脚本可能判定程序运行错误。多运行几次试试看？）

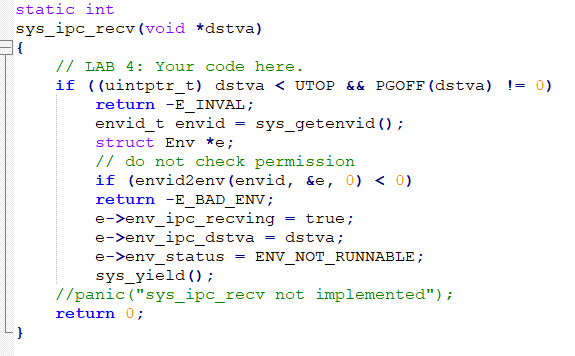
在 JOS 的 IPC 实现中，Env 结构体中有以下的一些修改



env\_ipc\_recving： 当进程使用 env\_ipc\_recv 函数等待信息时，会将这个成员设置为 1，然后 堵塞等待；当一个进程向它发消息解除堵塞后，发送进程将此成员修改为 0。 env\_ipc\_dstva： 如果进程要接受消息并且是传送页，保存页映射的地址，且该地址 <=UTOP。

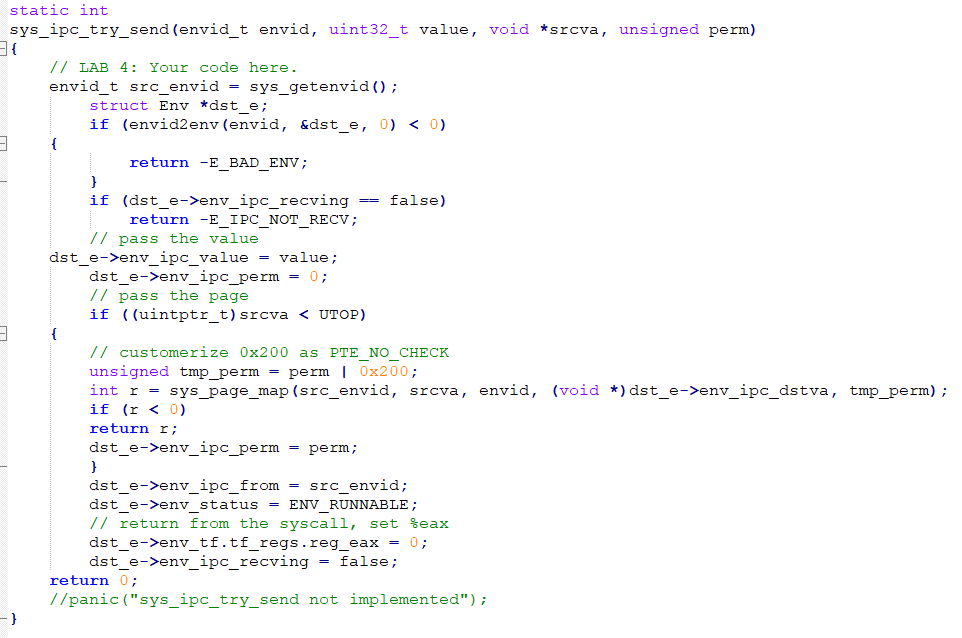
env\_ipc\_value： 若等待消息的进程接收到消息，发送方将接收方此成员设置为消息值。 env\_ipc\_from： 发送方负责设置该成员为自己的 envid 号。

env\_ipc\_perm： 如果进程要接收消息并且传送页，那么发送方发送页之后将传送的页权限 赋给这个成员。



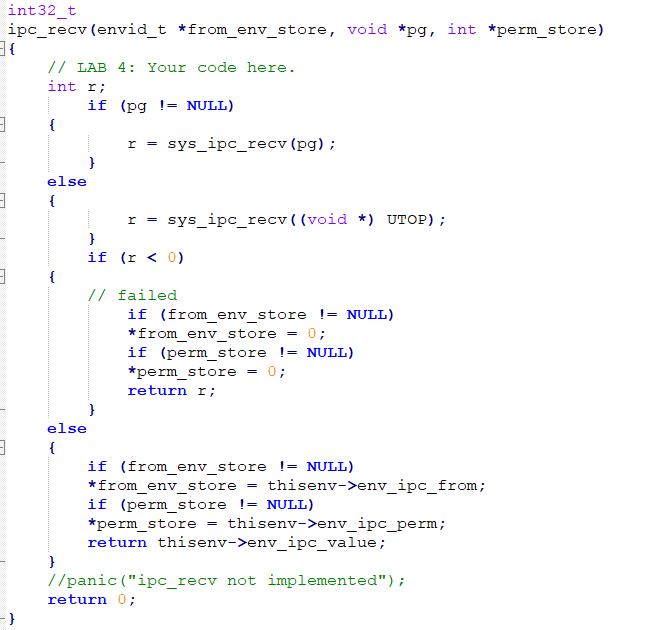
功能：将当前进程挂起，放弃CPU，准备接受其他进程发来的消息

实现方式：设置env\_ipc\_recving、env\_ipc\_dstva、env\_status，执行系统调用sys\_yield()



功能：试着发送消息，目标进程的id不存在、目标进程未开始接受、传递的页映射有问题 等原因都会导致发送失败。若发送成功，则更新目标进程，使其变为就绪态，不再接受消息。

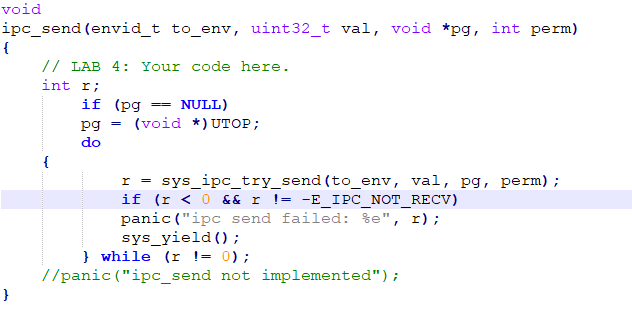
实现方式：先判断发送的条件是否全满足，若都满足，更新目标进程的env\_ipc\_recving、enc\_ipc\_from、env\_ipc\_value等



功能：接受发送者发来的value并返回它，获取发送进程的id，页权限

实现方式：执行系统调用，挂起当前进程，如果接收者想要共享页（pg非空），则将接收方共享页的虚拟地址设为pg，准备接收，修改from\_env\_store、perm\_store

（如果发送者发来了共享页，perm\_store和发送者的相同;发送者未发送页，则perm\_store=0，权限为内核）



功能：向目标进程发送value和内存页（如果pg非空）

实现方式：循环调用sys\_ipc\_try\_send(),直到发送成功，同时调用sys\_yield()，避免一直占用CPU，像spin一样被kill。

