**Lab4：抢占式多任务**

介绍这一部分你会在多个活跃的用户模式环境中实施抢占式多任务

**第一部分：**

增加CPU来支持JOS，实施循环调度，然后假设基本的环境管理系统调用。（增加或者删除环境，分配映射内存）

**第二部分：**

实施Unix-like函数fork，允许用户模式环境创建自身的复制

**第三部分：**

增加内容支持IPC（进程间通讯），允许不同的用户模式环境来显式地交流和同步彼此。你也要增加内容来支持时钟中断硬件，以及优先级。

**Part 1 多cpu支持和多任务合作**

第一部分，你需要扩展JOS，使其成为多CPU系统。

然后写一些新的系统调用，来允许用户级环境来增加额外的新环境。你也要实施协作循环调度，使得核能够从一个用户环境切换到另外一个用户环境。第三部分需要写抢占式调度，使得一段时间后，即使环境不配合，核也能够重新控制CPU。

**多CPU支持：**

实施多CPU支持，使JOS成为SMP，每一个CPU能够平等的获取系统资源，比如内存和I/O总线。不过，在boot阶段，CPU们分为两类，一类是bootstrap CPU负责初始化系统，并且boot操作系统，叫做BSP。一类是应用CPU，在boot之后，由BSP激活，叫做AP。

在SMP系统中，每个CPU有一个局部APIC（LAPIC）单元。LAPIC单元负责在整个系统中，传递中断。LAPIC给其对应的CPU一个独特的识别器。这个部分，我们使用LAPIC下面的基本函数：

1. 读LAPIC ID，告诉CPU当前的代码运行在哪个CPU上。
2. 从BSP到AP发送STARTUP处理器间中断（查看lapic\_startap()）
3. 第三部分对lapic内置时钟代码修改，来支持抢占式多任务。

一个CPU使用MMIO来使用自己的LAPIC，在MMIO中，一部分物理内存硬件化为一些IO设备的寄存器，这样，同样的能够获取设备寄存器的load/store指令来获取内存。

你已经看到了，在物理地址oxA0000有一些内存洞，我们使用这些内存来写VGA显示buff，LAPIC设备在物理地址oxFE000000（32M）的内存洞，因此这个地址太高了。JOS虚拟内存在MMIOBASE留下4M内存。这样我们有地方将这样的设备进行映射。由于后面的实验会再介绍MMIO区域，你将写一个简单的函数从这个区域开始分配空间，并映射物理内存。

**练习1：**写函数kern/pmap.c中的mmio\_map\_region，为了看这个是怎样使用的，看kern/lapic.c中的lapic\_init。在测试这个函数之前，需要先完成下面的练习。

**AP bootstrap**

在boot up AP钱，BSP需要手机多CPU系统的信息，比如CPU总数量，APIC ID和LAPIC的MMIO地址。Kern/mpconfig.c的mp\_init()函数通过读取在BIOS内存上的MP配置表来获取这些信息。

Kern/init.c中的Boot\_aps()函数启动AP bootstrap进程，AOs首先在实模式下运行，就像bootloader 在boot/boot.s一样，这样boot\_aps()将APentry代码复制到在实模式可以访问的内存地址。不想bootloader，这里有一些控制AP开始执行代码的地方，复制entry code到0x7000。但是任何未使用的，低于640kb 的page-aligned物理地址都可以

之后，通过给LAPIC单元对应的AP发送STARTUP，和AP开始运行的CS:IP，boot\_aps（）一个接着一个激活AP。kern/mp entry.S中的条目代码与boot/boot.S中的条目代码非常相似。经过一些简单的设置后，它将AP置于启用分页的保护模式，然后调用C setup例程mp\_main（）（也在kern/in it.C中）。boot\_aps（）等待AP在其struct CpuInfo的CPU\_status字段中发出CPU\_STARTED标志，然后继续唤醒下一个。

**练习2。**阅读kern/init.c中的boot\_aps（）和mp\_main（），以及kern/mpentry.S中的程序集代码。确保理解在aps引导期间的控制流传输。然后在kern/pmap.c中修改page\_init（）的实现，以避免将MPENTRY\_PADDR处的页面添加到空闲列表中，这样我们就可以在该物理地址安全地复制和运行AP引导代码。您的代码应该通过更新后的check\_page\_free\_list（）测试（但可能无法通过更新后的check\_kern\_pgdir（）测试，我们将很快修复该测试）。

**问题**

将kern/mpentry.S与boot/boot.S并排进行比较。记住kern/mpentry.S与内核中的其他内容一样，都是编译并链接到KERNBASE之上运行的，宏MPBOOTPHYS的用途是什么？为什么在kern/mpentry.S中而不是在boot/boot.S中有必要呢？换句话说，如果在kern/mpentry.S中省略它，会出现什么问题？

提示：回想一下我们在实验室1中讨论过的链接地址和加载地址之间的区别。

**每个CPU状态和初始化**

在写多处理器OS时，需要区分每个处理器的多处理器状态，以及一个整个系统共享的状态。Kern/cpu.h定义了大部分CPU状态，包括struct cpuinfo，存储每个cpu的变量。Cpunum()总是返回cpu的ID，从而可以用来索引cpus()函数。另外，thiscpu宏是cpuinfo的别名。

下面是每个cpu的状态

**每个cpu的核栈**

由于多个CPU可以同时进入核，需要给每个cpu一个核栈，避免他们影响彼此的执行。percpu\_kstacks[NCPU][KSTKSIZE]数组为NCPU的内核堆栈预留空间。

在lab2中，你学会了把bootstack的物理地址映射成BSP的核栈，在kstacktop下面。类似地，在这个实验室中，您将把每个CPU的内核堆栈映射到这个区域，保护页充当它们之间的缓冲区。CPU0的核栈任然从kstacktop开始往下降，CPU1的核栈从CPU0的核栈往下增加kstkgap bytes。Inc/memlayout.h显示了layout

**每个CPU的TSS,TSS描述器**

每个CPU的TSS来声明CPU的核栈存在的地方。Cpui的TSS存在cpus[i].cpu\_ts中，对应的TSS描述器在GDT entry定义着（gdt[(GD\_TSS0 >> 3) + i]），全局ts在kern/trap.c里面定义着。

**每个cpu当前环境指示器**

每个CPU可以同时与运行不同的用户进程，我们定义curenv来参考cpus[cpunum()].cpu\_env，这个数组指向运行在当前CPU上的用户环境。

**每个CPU系统寄存器**

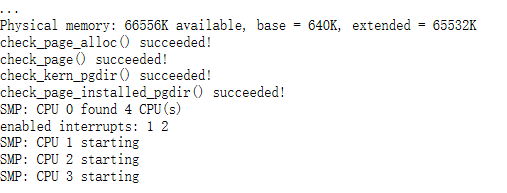
所有的寄存器，包括系统寄存器，都是CPU私有的。因此，初始化这些寄存器的指令，如lcr3(), ltr(), lgdt(), lidt()等，必须在每个CPU上执行一次。函数env\_init\_percpu()和trap\_init\_percpu()就是完成这项任务

除了这些，如果你增加了任何其他per-CPU状态，或者其他CPU-spacific初始化(比如给CPU寄存器增加新的位)。确保你在每个CPU上都进行了复制。

**练习3**，修改mem\_init\_mp()来将每个CPU的栈从kstacktop开始映射，如inc/memlayout.h里面的一样。每个stack的大小是kstksize bytes加上 kstkgap bytes的guard page.

**练习4**，trap\_init\_percpu()中的代码初始化BSP的TSS和TSS描述器。在lab3中，可以正常工作，但是运行在其他CPU上时会出问题。修改代码使其能够在所有cpu上运行。提示，你的新代码不应再使用全局ts了。

修改完之后，当你运行make qemu cpus=4，应该看到下面的内容：



锁

在mp\_main（）中初始化AP后，当前代码旋转。在使AP更进一步的之前，需要首先解决多CPU同时运行时的竞争条件。最简单的方法是使用一个大的核锁，即使得同一时间只有一个CPU进入内核，并在cpu返回用户模式时释放。在这个模型下，用户模式的环境可以共现地运行，但是只有一个cpu进入核。Kern/spinlock.h声明这个大核锁，即kernel\_lock.他也提供lock\_kernel()和unlock\_kernel()。你需要在下列四个部分声明核锁：

1）I386\_init()，在BSP唤醒其他CPU之前获取锁。

2）mp\_main()，在初始化AP之后获取锁

3）trap()，在用户模式trap的时候获取锁（系统调用等），使用tf\_cs位来确定一个trap发生在用户模式还是核模式

4）在env\_run()，在返回用户模式之前释放锁。

**练习5**，在合适的地方调用lock\_kernel()和unlock\_kernel()

问题：使用大锁能确保一个时间点只有一个CPU，我们为什么还需要给每个CPU一个核栈？描述一个场景，当使用共享的核栈会导致错误。

怎样测试锁是否正确？目前还不行，实施了scheduler之后才可以

**循环调度：**

在JOS中循环调动的原理如下：

1. 函数 在kern/sched.c中的sched\_yield()函数负责选择一个新的用户环境运行。他在envs[]数组里面循环运行。在之前运行的环境之后开始运行，选择找到的第一个标记为ENV\_RUNABLE的进程开始运行，然后调用env\_run()跳到这个环境。
2. Sched\_yield()不能同时运行在两个CPU里面，我们可以通过进程状态来判断当前运行sched\_yield()的CPU，因为这个进程的状态是ENV\_RUNNING
3. 我们已经为你运行了一个新的系统调用，sys\_yield()

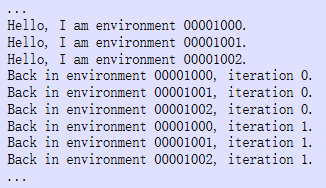
**练习6**，如上面所述，编写sched\_yield()函数的代码。记得修改syscall()来调用sys\_yield()

确保在mp\_main（）里面添加了sched\_yield()

修改kern/init.c来创建三个或者更多运行user/yield.c的用户环境，

运行make qemu.你应该看到用户环境在终止前彼此间切换了5次。

测试 多CPU情况：make qemu cpus=2



Yield程序退出之后，在系统中没有可运行的环境，scheduler应该打开JOS核监视器。如果上面中有一项没发生，必须继续修改你的代码

**问题：**

3）你的env\_run()函数中，你应该调用lcr3()。在调用前和调用后，你的代码需要声明一些变量e和参数env\_run的参考。加载%cr3寄存器后，MMU使用的寻址上下文将立即更改。但虚拟地址（即e）相对于给定地址上下文有意义——地址上下文指定虚拟地址映射到的物理地址。为什么在寻址开关之前和之后都可以解引用指针e?

4）每当内核从一个环境切换到另一个环境时，它必须确保保存旧环境的寄存器，以便以后可以正确地恢复它们。为什么？这是在哪里发生的？

**创建环境的系统调用**

尽管目前内核可以运行，并在多个用户环境之间进行切换，它的运行任然限制在核初始化后设置的环境内。你现在需要写需要的系统调用，来允许用户环境创建和开始一个新的用户环境。

Unix提供fork（）系统调用作为其进程创建原语。Fork()通过复制调用fork函数的整个环境来创建新的进程。仅有的不同是进程ID。在父进程中，fork返回自进程ID，在子进程中，fork返回0。默认情况下，每个进程有自己的私有地址空间，对本进程的内存的修改是彼此不可见的。

您将为创建新的用户模式环境提供一组不同的、更原始的JOS系统调用。通过这些系统调用，您将能够完全在用户空间中实现类似Unix的fork（），以及其他类型的环境创建。您将为JOS编写的新系统调用如下：

**sys\_exofork:**

这个系统调用创建了一个几乎是空白的新环境：地址空间的用户部分没有映射任何内容，并且无法运行。新环境将在sys\_exofork调用时具有与父环境相同的注册状态。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）。但是，在子项中，它将返回0。（由于子项开始被标记为不可运行，sys\_exofork实际上不会返回子项，除非父项已通过使用…..标记子项可运行来显式允许此操作）

**sys\_env\_set\_status:**

将指定环境的状态设置为ENV\_RUNNABLE或ENV\_NOT\_RUNNABLE。此系统调用通常用于在地址空间和寄存器状态完全初始化后，将新环境标记为准备运行。

**sys\_page\_alloc:**

分配物理内存页并将其映射到给定环境地址空间中的给定虚拟地址。

**sys\_page\_map:**

复制页映射（不是页的内容！）从一个环境的地址空间到另一个环境的地址空间，保留内存共享安排，以便新映射和旧映射都引用同一页物理内存。

**sys\_page\_unmap:**

取消映射在给定环境中给定虚拟地址处映射的页。

对于上面接受环境id的所有系统调用，JOS内核支持值为0表示“当前环境”的约定。该约定由kern/env.c中的envid2env（）实现。

我们在测试程序user/dumbfork.c中提供了一个非常原始的类Unix的fork（）实现。这个测试程序使用上面的系统调用创建并运行一个带有自己地址空间副本的子环境。然后，这两个环境使用sys\_yield来回切换，如前一个练习中所述。父级在10次迭代后退出，子级在20次迭代后退出。

**练习7**，实现kern/syscall.c中描述的系统调用，并确保syscall（）调用它们。您需要在kern/pmap.c和kern/env.c中使用各种函数，特别是envid2env（）。现在，无论何时调用envid2env（），都要在checkperm参数中传递1。请确保检查是否有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，并确保它在继续之前工作。

这就完成了实验室的A部分；当您运行make grade时，确保它通过了所有的A部分测试，并像往常一样使用make hand in将其交上来。如果您试图找出特定测试用例失败的原因，请运行./grade-lab4-v，它将显示每个测试的内核构建和QEMU运行的输出，直到测试失败。当测试失败时，脚本将停止，然后您可以检查jos.out以查看内核实际打印的内容。

**B部分：书面复制fork**

如前所述，Unix提供fork（）系统调用作为其主进程创建原语。fork（）系统调用复制调用进程（父进程）的地址空间以创建新进程（子进程）。

xv6 Unix通过将父页中的所有数据复制到为子页分配的新页中来实现fork（）。这与dumbfork（）采用的方法基本相同。将父地址空间复制到子地址空间是fork（）操作最昂贵的部分。

但是，在调用fork（）之后，子进程中几乎立即会调用exec（），从而用新程序替换子进程的内存。例如，这就是shell的典型功能。在这种情况下，复制父进程的地址空间所花费的时间很大程度上是浪费的，因为在调用exec（）之前，子进程将使用很少的内存。

因此，较新版本的Unix利用虚拟内存硬件，允许父级和子级共享映射到各自地址空间的内存，直到其中一个进程实际修改它为止。这种技术被称为写时拷贝。为此，在fork（）上，内核将把地址空间映射从父对象复制到子对象，而不是映射页的内容，同时将现在的共享页标记为只读。当两个进程中的一个试图写入其中一个共享页面时，该进程将出现页面错误。这时，Unix内核意识到页面实际上是一个“虚拟”或“写时拷贝”拷贝，因此它为出现错误的进程生成了一个新的、私有的、可写的页面拷贝。这样，在实际写入单个页面之前，不会实际复制这些页面的内容。这种优化使得fork（）后面跟着exec（）的子进程要便宜得多：子进程在调用exec（）之前可能只需要复制一个页面（堆栈的当前页面）。

在本实验室的下一部分中，您将实现一个“适当的”类Unix的fork（），并将其作为一个用户空间库例程进行写时拷贝。在用户空间中实现fork（）和copy-on-write支持的好处是内核仍然简单得多，因此更有可能是正确的。它还允许单个用户模式程序为fork（）定义自己的语义。一个需要稍微不同的实现的程序