JOS 设计文档（lab4范围内）
丁卓成 5120379064

一、概述

本文档为Lab1-3提交的设计文档之续，继续分析JOS的架构，本次Lab的主要内容是实现多核支持、进程调度、fork系统调用、外部中断处理和进程间通信。

二、多核支持的引入

（一）、多核系统的架构及技术

1.多核系统的架构

本次Lab的一个重大改变就是支持SMP(Symmetric Multiprocessing)，即支持了一种多核架构。常见的另一种多核架构是NUMA(Non-Uniform Memory Access)，有时为强调其支持Cache Coherence也称为ccNUMA(cache coherent NUMA)。这两者的“多核”都是紧密连接的，各个核心共用Memory和I/O，通过总线交互，与此相对，另一种架构称为MPP(Massive Parallel Processing)或Cluster，是采用大量独立的processor通过高速网络进行通信和协调，来实现一个系统的完整功能，例如课上提到的Google的Cluster即是此类。

另一个与SMP相对的概念是AMP/ASMP(Asymmetric Multiprocessing)，顾名思义，AMP指的是系统中各个processor职责不对称，例如某个CPU专门负责运行OS，其他CPU运行User态程序；SMP指的则是系统中各个processor的职责是对称的，尽管某一给定时刻它们负责的任务不同，但其任何一个processor都可以在任何时刻执行任何任务，其能力是等同的。NUMA的设计延续SMP的理念，在对称性上是一致的，主要区别在于NUMA为了提升Scalability，将Memory和CPU划分成区域，每组CPU有自己的Local Memory，各组CPU的Local Memory之间通过高速的硬件总线连接，这样就导致Memory Access对于Local Memory和Remote Memory的Latency不同，故得名Non-Uniform Memory Access。

家用电脑中最常见的x86-64处理器属于SMP，但x86-64处理器的服务器版，例如Intel Xeon和AMD Opteron采用的是NUMA架构，这是因为服务器通常要有40-60个核，而SMP的Scalability较差，一般只能支持2-15个核（根据Intel文档所称）。在本Lab中，出于教学与训练的目的，只选择了较为简单且常见的SMP进行支持，旨在让学生对多核开发、分布式系统开发的复杂性有一个感性认识。

2.多核系统中的Cache Coherence

从单核到多核系统，会产生许多问题，其中一个问题就是多核系统中每个核都有自己的cache，假若两个核各自朝同一个内存地址写数据，分别写入了各自的cache并且采用write back的方式，它们间对于该内存地址就发生了不一致。注意，即使对多个核间共享的变量采用锁保护起来，在write back的cache下仍可能产生不一致。为了解决这个问题，现代的多核处理器一般都保证一称为Cache Coherence的性质，即对于同一个内存位置，各CPU所见的状态必须互相一致。这个特性是在硬件层面上实现的，软件并不参与其中，因此不存在需要软件手动flush cache这样的问题，网上存在一些关于这一点的错误言论，不可轻信。

尽管硬件已经实现好了相关机制，但软件开发人员，尤其是开发操作系统之类的软件的底层程序员，还是应该了解相关知识，这有助于理解多核环境下non-scalable lock的问题。对于Cache的操作，是以Cache Line为单位的，因此相关的协议(Cache Coherene Protocol)也建立在此基础上。在单核的架构中，Cache Line有两个状态，Valid和Invalid，扩展到多核后，最简单的一种协议就是MSI协议，分为三种状态Modified,Shared和Invalid，M和S即对应写者和读者，任一时刻对任一内存位置，或者有一个写者，或者有若干读者，或者没有在任何CPU的Cache内。

现实中的处理器一般采用基于MSI协议的一些变种，例如为了减少写回的次数，增加Owner状态（负责write back的读者状态），为了应对频繁的Read-Modify-Write操作，减少与其他核交互，增加Exclusive状态（独占资源的读者状态）。真实的处理器的协议往往较此处简要说明要复杂的多，可能多达几十种状态，但复杂性的主要来源是split-transaction bus，即Cache之间互相沟通的bus不是由某Cache独占而导致阻塞的，而是可以并发处理多个Cache发出的控制信息和数据流（为提高throughput），从而导致协议为处理竞争增加若干状态，其从本质上仍遵循上述简单的MSI协议的基本原则。

MSI协议的工作原理大致为，每当一个CPU要成为写者对它的Cache进行写入时，要通知其他保有相应Cache Line的Cache，将它们的状态设置为Invalid，每当一个CPU要成为读者对它的Cache进行读取时，要通知保有相应Cache Line的写者Cache（若存在），将它的状态设置为Invalid，并将其值直接通过Cache间通信发送给读者。具体在每种情形下对其他Cache中的Cache Line进行什么操作我们并不关心，重点在于，这个Cache之间的沟通过程通常是较慢的。在SMP系统中，通过Cache Snooping的方式进行交互，每当一个Cache想Invalid别的Cache中的某个Line，它就要在Bus上广播该信息，别的Cache监听(snoop)该信息，并相应地（若有必要）做出反应，这种机制导致了其Cache子系统的Scalability比较差，难以支撑较多的核心数。在NUMA系统中，采用Directory Based的协议，本质上还是MSI Based的协议，但每组CPU有一个Home Directory，掌握该组CPU中Cache的信息，并知道哪些Cache Line同时还被远端的Cache共享，Cache之间的交互通过Directory作为中介进行，避免了广播所造成的不存在目标Cache Line的Cache也要处理广播信息的浪费问题，使Cache子系统层次化组织从而易于扩展，但从根本上来说，假如Cache Line被本地和远端同时共享，其性能可想而知是低的，必须要在维持局部性的情况下才不会引起性能的下降。

现在知道了Cache Coherence的实现细节，就不难理解为何有些锁被称为non-scalable lock，因为各个核心要争抢同一个锁变量，即在同一个Cache Line上发生竞争，每次抢占耗时都是较大的，尤其在核心数目较多、用于服务器领域的ccNUMA上，只要核心数超过一个CPU模组中具备的数量（一般最多十多个），就会马上遇到scalabity瓶颈。

在本次Lab中默认提供最简单的spin lock代码，并要求实现ticket lock，遗憾的是这两种锁都是non-scalable的，问题在于它们都会在同一个变量上发生抢占，这在此前CSE课程的Recitation中有所提及。

3.Reorder与Memory Model

现在我们有了Cache Coherence保证多核对内存的访问是一致的，但多核系统表现得仍和单核不同，要形成一个完整的Memory Model，另一片拼图是Reorder。我们知道，现代的编译器会对程序的源代码进行重排序以优化性能，处理器也会对指令进行乱序执行，不过最终的目标是运行起来好像并未对源程序中的顺序进行过改动。但是，有些重排序的优化对于单核来说是可以的，进入多核的环境就会导致问题。

现在设想有这样的汇编代码：

```
CPU0                    CPU1
movl $1, (%x)           movl $1, (%y)
movl (%y), %r1          movl (%x), %r2
```

初始时%x和%y指向的内存存放的均为0，这类似于Dekker's Algorithm中的情形，一个熟悉单核并发的人一定会期望%r1 = 0和%r2 = 0不可能同时出现，这样算法才能正常工作。然而，由于(%x)和(%y)是两个不同的内存位置，处理器可能会将Load指令提到Store指令之前，即movl (%y), %r1和movl (%x), %r2先分别读取到了0，接着相应的内存位置才被写入1，此时获得的结果正是%r1 = %r2

= 0。

注意CPU即使在retire阶段按照原来机器码中指令顺序retire，遵循此种较强的顺序性（至少此时两条Store指令即使在流水线中交换了顺序，最终写入内存时还是按顺序的，如果乱序retire，则写入内存的顺序也会发生变化），仍无法避免此类reorder的问题，除非它放弃对Load和Store指令的reorder。注意到即使CPU完全不乱序，编译器重排序也禁止，程序仍有可能由于data race产生bug，从而需要加锁。因此放弃reorder的好处相比reorder带来的性能提升来说过于微不足道，现在的处理器倾向于提供更弱的order限制，以发掘性能上的潜能，减少CPU速度与Cache访问速度的差距造成的延迟。现在除了x86还提供非常强的Memory Order（通常用TSO(Total Store Order)称呼，该术语源于SPARC处理器的TSO模式）外，其他常见的处理器如ARM，Power PC都采用了非常宽松的Order限制。为了提供加锁以外的同步机制，允许程序员为提高性能编写Lock Free的代码，通常这些处理器都提供了一类称为Memory Barrier的指令，使得该指令前后的Memory指令不能跨越该指令重排序。Memory Barrier一般分为Load,Store和Full三种，分别起防止Load指令,Store指令和所有Memory指令跨越Barrier的作用，其他还有一些变种可参看具体处理器的手册。

通过使用Memory Barrier，我们可以在任何只提供较弱Memory Order的处理器上实现Sequential Consistent的Memory Model(或更正式的，Memory Consistency Model)，即并行程序的执行结果，总是可以和单核上并发执行该程序的某个结果对应，而不会产生程序员意料之外的结果，这在Java和C#中是通过对进程共享变量加volatile修饰符实现的。然而，在C和C++的标准中，仅规定了volatile修饰符是用于访问MMIO的外设设备的，在C11中提供了_Atomic修饰符，C++11中提供了std::atomic模板以实现同等效果。除了Sequential Consistency外，还有一些更强的或是更弱的模型，由于在较弱的模型中能实现的Lock Free算法有限，这些模型一般仅仅是出于研究上的兴趣和价值而被发展起来的，和软件开发者距离较远，有时比Sequential Consistency更弱的Memory Model就统称为Weak Consistency。

在本次Lab中，无需实现Lock Free算法，根据x86的手册，其原子指令(atomic instruction，即Compare and Store,Test and Set这一类指令)会锁住总线实现原子语义且不会被乱序，因此我们的任务就只是简单地使用原子指令实现一个锁即可，不用和Memory Barrier打交道。不过，由于这个概念在课堂上没有详细讲解过，在Lab迁移到多核环境后加以了解也是很有必要的，对今后Lab的编写以及以后更多编程工作有长远意义。

（二）、多核系统的启动和中断处理

下面来关注一下关于多核系统具体如何启动的细节，由于这涉及IPI，故将多核系统的中断处理一并介绍。

1.多核系统的中断处理

在多核系统中，每个CPU都具备处理外部中断的能力，并且除了一般的外部中断外，还有一类特殊的中断，称为IPI(Inter Processor Interrupt)，由某个CPU发给另一个或一组CPU，以实现中断的转发以及CPU间通信。

在x86架构中，中断的处理是由APIC(Advanced Programmable Interrupt Controller)管理的，它分为两个部分，Local APIC和I/O APIC。每个processor都具备一个Local APIC，负责接收中断并发送给该processor，此处的processor指的是逻辑核心，即如果某个核心支持超线程，则其中的多个逻辑核心每一个都有自己的一套Local APIC。每个Local APIC都有一个unique APIC ID，在发送IPI时可以通过APIC ID指定发送的目标，OS也可以将APIC ID作为CPU的ID使用。在早一些的系统中，APIC ID从0-14,15被留作广播用，新一些的系统中APIC ID从0-254,255被留作广播用。一个系统可以拥有一个或多个I/O APIC，其作用是管理外部中断的发送，当有外部中断发生时，由它决定发给一个或多个processor的Local APIC，由于我们的Lab中没有使用I/O APIC，不详细介绍。

一个Local APIC中的中断有以下几个来源：I/O APIC发送给它的中断，其它Local APIC发送给它的中断，通过CPU上的LINT0和LINT1 pin直接传入的中断，以及自身根据设置自行产生的中断（例如APIC Timer，Performance Counter，Thermal Sensor等来源）。

此外，在多核系统中还有一个8259兼容的PIC模块，以提供向前兼容性。可以通过操作MSR寄存器来（硬件）禁用Local APIC，这样该CPU将直接与PIC进行交互，注意在禁用后（除特定型号处理器外）除非重启，将无法再次开启Local APIC。与之相对，在Local APIC中有一个Spurious Interrupt Vector寄存器，通过设置其中的Enable位，可以（软件）禁用或启用Local APIC，这在禁用后是可以启用的。在软件禁用状态下，将不能接收普通的中断（如键盘产生的中断之类），只能接收如NMI,INIT,SIPI等特殊信息。

除了直接连接PIC模块(PIC Mode)和完全启用APIC的特性进行多核中断处理(Symmetric I/O Mode)，还有一种Virtual Wire Mode，也就是将Local APIC作为Virtual Wire，将PIC模块产生的中断信号通过LINT1传到CPU（还可以配置将I/O APIC也作为Virtual Wire的一部分）。注意PIC Mode和Virtual Wire Mode都bypass了I/O APIC的中断分发机制，因此只有一个核能处理中断，其他核应该禁止外部中断。

2.多核系统的启动

多核系统中只有一个核能运行OS的启动代码，称为BSP(Bootstrap Processor)，其他核称为AP(Application Processor)，这是从软件的角度出发看的。从硬件的角度来说，不必有一个核较其他核特殊，事先固定为BSP，也就是说BSP可以在启动过程中动态地决定，这样即使某些核出现故障整个系统仍能正常启动。在启动时，首先由硬件决定哪个核是BSP，然后该核心可以执行BIOS代码，其他AP则处于等待唤醒状态。

BSP上运行的BIOS在进行一些工作后，会广播一个SIPI(Start-IPI)使所有AP启动。SIPI包括一个Interrupt Vector，一般中断的此域用作IDT索引，而对于SIPI则用于指定AP的启动地址，其启动地址为0xVV000，VV为Interrupt Vector的Hex值。根据Intel MultiProcessor Specification，在BIOS运行阶段，会建立一张MultiProcessor Configuration Table，其中储存了系统的信息，包括了所有核心的信息，在OS的启动阶段需要读取这些信息进行相应的初始化。BIOS会通过一个信号量序列化AP的初始化过程，在各AP初始化期间会将这个MP Table填充完整，此外还会填充ACPI Table（ACPI为Advanced Configuration and Power Interface，用于多核系统的配置和电源管理，此处不讨论），以及对Processor Counter加一以统计系统中processor数量。AP执行完初始化过程后，以一条cld指令接一条halt指令停止执行，再次进入等待唤醒的状态。最后，所有AP均停止后，BSP执行完BIOS代码，开始运行Boot Loader。

在OS的Boot阶段，要读取MP Table以获得系统内的CPU信息，据此可以知道系统中的其他AP（表项中存有其APIC ID，以此为Destination可以通过Local APIC对其发送IPI），通过给AP发送INIT中断，将AP的CPU内部状态重置，接着连续发送两次SIPI（这是Intel文档的规定，在Lab的lapic_startup函数的注释中提到第二个SIPI会被硬件忽略），即可将其启动，并执行OS中的初始化过程。

3.Lab中的初始化与AP启动代码分析

前三个Lab的工作都是在BSP上进行，在本次Lab中，我们需要唤醒AP，实现多核环境的支持。

3.1.Exercise1-3

Exercise1-3要求修改Lab2、Lab3中的代码，这些代码在初始化LAPIC并唤醒AP之前进行。我们只需要为各个核分配自己的栈和TSS，为Local APIC所在的物理地址提供映射，并为接下来AP要执行代码的位置预留一个物理页，即可实现多核的启动。注意在调用trap_init()函数时，Lab4中做的初始化工作还未进行，调用trap_init_percpu()时会认为thiscpu是&cpus[0]，从而为BSP的Task Register加载GD_TSS0段，假若BSP不是CPU0，在CPU0进行初始化时会再次加载该段，导致CPU0发生Triple Fault。也就是

说，它的正确性依赖于QEMU总是把CPU0选为BSP的行为。

3.2.kern/mpconfig.c

完成Lab2、Lab3中的初始化工作后，首先调用了mp_init()，负责获取MultiProcessor Configuration Table中关于Processor的信息，以确定系统内有多少核心，最多只支持8个核心（NCPU=8在kern/cpu.h定义）。mp_init()位于kern/mpconfig.c，其中定义的结构体和执行的功能均依照Intel MultiProcessor Specification所描述，在此不详细介绍每句代码的含义，具体可参考上述文档。

来具体考察其代码，mp_init()首先调用了mpconfig()函数，该函数首先调用了mpsearch()函数。整个过程的第一步，就是调用mpsearch()函数来搜寻一个mp结构体，即mp floating pointer，它并不是我们要找的MP Table，只是提供了一个MP Table的起始地址以及一些基本信息。由于它可能出现在三处位置（注释中有提到，详见Intel MP Spec），OS无法实现知晓其位置，因此需要搜索。搜到后，利用mp->physaddr可以找到MP Table，mpconfig()函数进行了一些检查以确保该表格式正确无误，检查具体含义详见Intel MP Spec。如果检查结果正确，将会返回一个mpconf结构体，为MP Table的Header，由此可以知道Local APIC的寄存器映射到了物理内存中的何处(mpconf->lapicaddr)。此外还可以知道表中有多少项(mpconf->entry)，据此进行一个for循环处理每一项。可以看到，事实上我们只处理MPPROC类型的表项，即处理器信息类型的表项，并且我们实际上并没有去读取其APIC ID(mpproc->apicid)，只是把读到的第n个MPPROC表项视作CPU n-1。注意，这一点又是依赖于QEMU所使用的BIOS的行为，因为Intel MP Spec里没有规定APIC ID必须连续取值，它甚至专门举例指出可以之出现APIC ID为0,2,4的表项，即使连续取值，也没有规定它们在MP Table中是按顺序出现的。最后，我们把BSP的状态设为STARTED，假如系统支持PIC Mode，还要手动禁止使用该模式。

3.3.lapic_init()

紧随mp_init()函数，调用了lapic_init()，位于kern/lapic.c，其中定义的宏和进行的操作可以参考Intel 64 and IA-32 Manual Vol.3 Chapter 10 APIC。此函数不仅被BSP调用，还要被所有AP调用。

由于Local APIC是通过MMIO(Memory Mapped I/O)的方式访问，且我们在mp_init()中已经获得了其基址(lapic变量)，只需要通过通常的内存读写即可操作其寄存器，在此处用lapicw函数包装了一下。通过lapic变量可以直接获得APIC ID，用cpunum()函数进行了包装，此外还提供了lapic_eoi()和lapic_ipi()函数，前者在中断例程结束时必须被调用以通知APIC中断已处理完成，后者只有中断向量这一个参数，其行为是向其他所有processor广播一个IPI。

mp_init()的第一句代码设置了SVR(Spurious Vector Register)，从而启用了Local APIC（软件启用，见1.中所述），这是因为CPU在重置后Local APIC的状态是软件禁用的，所以对于所有AP都需要手动启用。当然，这句话同时也设置了Spurious Interrupt的Interrupt Vector，所谓Spurious Interrupt顾名思义是由于某种硬件上的错误状态导致的虚假中断，并不能表示任何设备的请求，对于这样的中断在本Lab中的一个处理方式仅仅是输出信息以说明发生了一个Spurious Interrupt。

随后的三句设置了APIC Timer，参数PERIODIC表示TCCR(Timer Current Count)从TICR(Timer Initial Count)开始减少，减少到零产生一个中断，并重设为TICR，不断循环，TDCR(Timer Divide Configuration Register)设置为X1表示每个时钟周期Count减一。这个Timer的设置与许多单片机上的类似，不过作用却不同，本次Lab中利用APIC Timer提供的时钟中断来实现了抢占式任务调度。

随后几句禁止了Local APIC向CPU发送LINT0,LINT1上的中断以及Performance Counter(假如有的话)引起的中断，除了BSP没有禁止LINT0。并且最后一句将TPR(Task Priority Register)设置为了0，TPR的作用是阻塞所有优先级高于它的中断，优先级取决于Interrupt Vector，粗略地说越大优先级越高，设为0则有效地禁止了I/O APIC传入的中断以及Local APIC之间的普通IPI。综上，对于AP我们仅启用了Local APIC的Timer而禁用了其他所有功能，对于BSP我们还启用了LINT0，根据注释以及Intel MP Spec，BSP将处于Virtual Wire Mode工作，PIC兼容模块将把中断通过LINT0送往BSP，在我们的Lab中也只有BSP能处理外部设备产生的中断。

中间这几句目的是清空此前可能的错误状态，我们无需关心，只需知道Local APIC内部发生错误时会发送中断给处理器，并且我们为其设置了中断向量即可。

3.4.kern/picirq.c

随后执行的是pic_init()函数，在kern/picirq.c中定义，负责初始化8259A兼容的PIC模块，这个初始化操作只需在处于Virtual Wire Mode的BSP中执行，其他AP则无法获取外部中断。

该函数中的注释提供了一些解释，但仍有模糊之处，具体可以参考8259A的文档，现稍作解释。ICW指的是Initialization Control Word，OCW指的是Operation Control Word，ICW只用于初始化PIC。一开始oub(IO_PIC1+1, 0xFF)以及irq_setmask_8259A()函数中的oub(IO_PIC1+1, (char)mask)，其实在文档中称为OCW1，只要是直接向端口IO_PICx+1写入就属于OCW1，而其他操作方式均需要涉及IO_PICx端口。这个OCW1的作用就是设置Mask Bitmap，显然根据picirq.c中的代码，我们将PIC初始化成了Mask掉所有中断。

另外PIC的可配置稍差，它配置的Interrupt Vector的方式是设置一个Offset，其末尾3位必须是0，某个中断(IRQ)来到PIC后，加上Offset即得Interrupt Vector，它无法像APIC那样对每个中断源设置单独的中断向量。如代码所示，我们将其中断向量配置为了从IRQ_OFFSET开始的16项。

和APIC一样，中断例程的结尾，要发送给PIC一个EOI(End of Interrupt)以通知中断已处理完成，这要通过OCW2来实现。我们需要用到的格式为

OCW2: 0ei00irq ei: 01 = non-specific EOI, 11 = specific EOI; irq: IRQ number

注释中所说Automatic EOI mode即为CPU从总线上获取Interrupt Vector后回应PIC，PIC立即执行一个non-specific EOI，将目前ISR(In-Service Register，这是一个bitmap)中优先级最高的位清零。注意到Slave无法启用Automatic EOI Mode，所以对于Slave管理的IRQ，总是需要手动发送EOI通知它中断已处理完成。

关于其clear specific mask的语句，有必要解释其作用。在PIC中和APIC一样，中断都有优先级，默认为中断号小者优先级高，在某个中断例程正在执行而没有返回EOI时，优先级比它低的中断是被屏蔽的。假若开启了specific mask mode（或称special mask mode），只有通过OCW1显式禁止的中断会被屏蔽，其他任何中断即使是低优先级的中断也能打断正在执行的中断例程抢先执行。clear specific mask即为显式地关闭了specific mask mode。

3.5.最终的启动步骤

BSP启动AP的最后一步是调用boot_aps()函数，该函数将mpentry.S的代码从1M以上的物理地址复制到物理地址0x7000的位置，并调用lapic_startap()函数唤醒AP，AP被唤醒后就将从物理地址0x7000开始运行，从而执行mpentry.S的代码，然后跳转到mp_main()函数继续执行。当AP在mp_main()函数中完成加载页表、初始化Local APIC，初始化段寄存器、TSS和IDT后，将自己的状态改为CPU_STARTED，而boot_aps()函数则用一个循环等待一个AP初始化完毕后，再唤醒下一个AP。来看lapic_startup()函数，除了通过INIT-SPI-SPI序列向AP发送唤醒指令以外，还设置了CMOS以及warm reset vector，这是为了处理器此后每次软重启都能直接在指定的warm reset vector（此处即为0x7000）开始执行，避免重复进行自检操作。至此，整个系统已经完全启动，所有核心都已经处于开启页表的保护模式，接下来由BSP创建若干初始进程后，调用sched_yield()启动用户进程即可让系统开始运作。

三、Lab具体设计

（一）、加锁策略

为了防止多核同时进入内核而发生竞争，必须要采用适当的加锁策略。在本次Lab中，我们采用了最简单的Big-lock策略，即使用同一把kernel lock，将内核的所有部分都锁住，显然这样丧失了潜在的并行性，降低了性能，但简化了加锁的策略。在实现加锁时唯一需要注意的是，system call可以通过sysenter指令和int $T_SYSCALL两种方式进入（尽管Lab 3中规定只允许使用sysenter指令，但Lab 4中sys_exofork()调用规定要使用int $T_SYSCALL，事实上两种方式都是存在的），通过sysenter指令进入的需要在syscall()函数中加锁，而通过int $T_SYSCALL进入的在trap()函数中已经加过锁，不需要在syscall()函数中加锁，因此需要为两者提供不同版本的syscall()，其中加锁的版本作为不加锁的版本的wrapper。

此外，Exercise4.1要求实现一个ticket lock，默认的版本是使用xchg指令实现的最简单的spin lock，ticket lock则是通过fetch and add指令实现的（在x86上为xadd指令）。实际上，使用test and set，fetch and add这样的指令，只能实现两个thread的同步，即只能实现互斥锁(mutex)，为了实现信号量(semaphore)，需要使用一类称为CAS（Compare And Swap）的指令，在x86架构上就是cmpxchg指令。它有三个操作数，old_v,mempos和new_v，old_v和new_v为寄存器，mempos为内存地址，其执行的操作是：比较old_v和mempos内的值，若相等，向mempos写入new_v的值，否则不进行修改，此外最终会在old_v寄存器内存入mempos在操作开始前的旧值。这样，实现信号量时只需先读出旧值，进行相应加/减，再执行CAS指令，如果失败说明信号量已被别人修改，进行循环重复至成功为止即可。在RISC的处理器上，该指令由于有三个操作数一般不被采用，它们使用一类称为LL/SC（Load-Link and Store-Conditional）的指令，将原本一条指令完成的工作分为了LL和SC两条指令，使用LL指令锁定某个内存地址，接着读取其内容并修改后再用SC指令存入，如果在此期间其内容没有被修改则指令成功，否则失败，以此来保证Read-Modify-Write语义。

（二）、进程调度机制和外部中断处理

Lab中首先要实现的是进程调度机制，即sched_yield()函数，有了这个才能正常地启动第一个程序。我们采取的策略同样很简单，为Round-Robin调度，即对所有进程按envs数组中的存放顺序调度，当前进程yield后就从当前进程在数组中的位置开始向后搜寻可调度的进程，搜到尾后从数组开头开始即可。通过这种简单的方式，不引入优先级，我们反而不会遭遇优先级反转或是进程饿死等情况，大大简化了问题。

不过，我们的一个问题在于，用户进程通过sysenter指令进入内核调用sys_yield()系统调用，然而调度是基于iret指令的，需要设置相应的Trapframe。因此，我们在sysenter_handler（位于kern/trapentry.S中的汇编代码）的开始处增加了一条pushal和一条pushfl以在栈上压入通用寄存器和EFLAGS，并在syscall()函数中利用这些信息构建Trapframe，从而解决了这一问题。

进行到这里，我们仍无法实现抢占式调度，为此需要在env_alloc()函数中将用户进程的EFLAGS寄存器中IF位初始化为1，从而启用外部中断。当然我们也要在trap.c中注册与外部中断相关的IDT表项，并处理时钟中断(IRQ_OFFSET + IRQ_TIMER)。在trap.c中，我们将所有的IDT表项均设置为了Interrupt Gate，使得当处于内核态时外部中断处于禁止状态，当然通过iret指令返回时中断会重新启用，因为被压入内核栈的EFLAGS仍保持IF位原本的状态。不过，sysenter指令会自动将中断屏蔽，从而导致此前提到用pushfl压入的EFLAGS值的TF位为0，因此在构建Trapframe时需要为读取到的EFLAGS的IF位置1。并且由于sysexit指令并不会自动启用中断，在执行sysexit指令之前需要手动执行sti指令以开启中断。至此就实现了在用户态启用外部中断而在内核态禁止，由于我们的AP禁用了所有中断，而BSP也将PIC的IRQ全都mask掉，所以目前除了时钟中断外并无其他来自程序外部的中断。

（三）、fork调用的实现

我们的fork将作为一个用户态的调用实现，而非内核态，内核只提供一些基本的服务，如创建进程、映射虚拟地址等。第一步是实现sys_exofork,sys_env_set_status,sys_page_alloc,sys_page_map,sys_page_unmap这五个系统调用，以提供对自己以及自己的子进程的基本控制。总体上这些功能已经在Lab 2,Lab 3中实现过，此处只要做一个wrapper即可，当然还需要做一些错误检查，唯一值得注意的是sys_exofork是通过int $T_SYSCALL调用的，需要先传到trap()函数再传到syscall()函数。

第二步，我们需要实现一个用户态的page fault处理机制，从内核态返回用户态的handler并不难，只需在struct Env中记录handler的位置即可，注意内核应当负责在User的Exception Stack上构建一个UTrapframe就能完成。而用户态运行在Exception Stack上的page fault handler如何返回发生page fault的位置则需要一些技巧，必须在pfentry.S的汇编代码（即用户态page fault handler的入口点）中小心地设计回复寄存器的顺序，并且通过在栈上伪造一个返回地址，先换栈，再利用这个返回地址使用ret指令跳转回发生page fault时的eip。

第三步，借助用户态的page fault handler，可以实现一个copy-on-write的fork()调用。一旦由于copy on write的页被写而发生page fault，page fault handler可以检测到并将它复制到一个新的物理页上去，将权限设置为可写，其具体的细节和Lab中默认提供的user程序dumbfork有些类似，可以作为参考，在此不做赘述。

（四）、进程间通信(IPC)的实现

IPC从理论上来说应该较为复杂，不过我们在此处所实现的是一个极其简单的版本。总的来说，我们实现的IPC只有两个功能，一个是传递一个32位整数作为Message，另一个是，假如sender希望提供一个页进行共享，且reciever也希望在某个虚拟地址建立映射与他人共享，则可以建立一个共享的页，双方可通过这个页来交互。在我们的实现中，IPC仅由两个系统调用sys_ipc_try_send()和sys_ipc_recv()以及两个用户态调用ipc_send()和ipc_recv()构成，其中sys_ipc_try_send()是一个非阻塞的操作，假若目标暂时无法接收(此时它并不处于调用sys_ipc_recv()的过程中)，立即返回，报告一个E_IPC_NOT_RECV错误；sys_ipc_recv()是一个阻塞的操作，设定好相应信息后将自己设置为ENV_NOT_RUNNABLE，随后调用sched_yield()调度到别的进程，直到有一个sender向其发送数据（可能同时映射页表），并将其状态设为ENV_RUNNABLE，才能返回。在struct Env中Lab的初始代码已经为我们提供了记录所要传递的Message所需的域，我们只需正确使用即可顺利实现该功能。

完成该功能后，我在此部分还完成了一个Challenge。该Challenge描述，由于ipc_send()底层的系统调用sys_ipc_try_send()是非阻塞的，在ipc_send()中要进行一个while循环以保证在用户态的调用是阻塞的，如何去除。我的解决方案是，将底层的系统调用做成阻塞的，为此在struct Env中又加入了若干域（和Lab默认提供的类似），记录阻塞的sender的状态，receiver调用sys_ipc_recv时先检查有没有正阻塞的sender且目标正是自己，如果有，接收其发送的信息并将其设置为ENV_RUNNABLE并直接返回，如果没有，设置自己的状态并yield。

为了测试Challenge完成的正确性，我为Lab增加了一个user程序blocksend.c，其功能是一个父进程创建若干子进程，由子进程不断通过IPC向父进程发送信息，由于同时存在较多子进程，会有许多sender同时发向一个receiver从而导致sender阻塞，经测试功能正常实现。

四、结论

本次Lab由于引入了多核，导致有许多概念需要辨析，多核系统的启动过程也较为复杂，为了理解清楚具体的原理和启动过程我查阅了不少资料，不过这些都是值得的。经过这次Lab，我对多核系统的理解更为深入，这个Lab也逐渐设计得更接近一个真实的操作系统，至少操

作系统中最重要的抽象—进程及其相关操作已经实现完毕，到下一个Lab应该会继续扩展功能，实现一个简单的文件系统，届时再行论述。