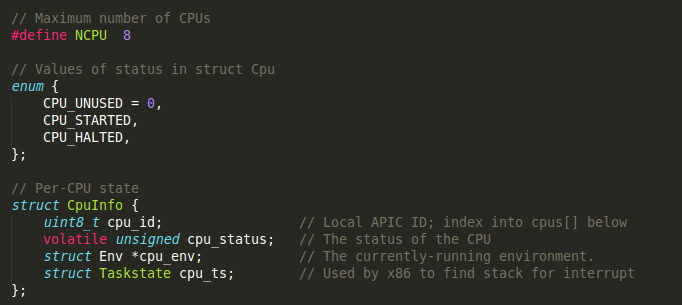
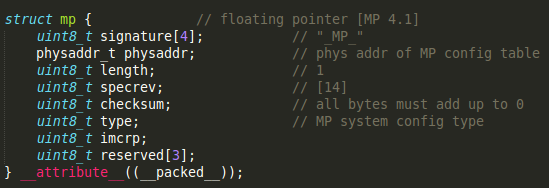
# Lab4.Preemptive Multitasking(抢占式多任务)

## Part A: Multiprocessor Support and Cooperative Multitasking

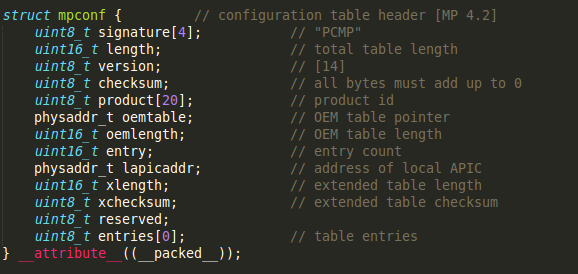
对于多核CPU, JOS对其进行了抽象. 具体在./kern/cpu.h, 我们可以看到CPU最大数目是支持8个,具体的，对CPU利用结构体进行抽象－－struct CpuInfo.



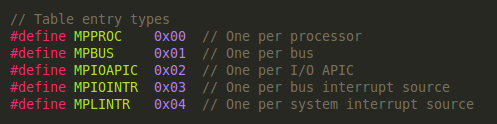
而多核处理器我们用 struct mp来进行描述



其中的成员 mp.physaddr指向结构体 struct mpconfig



值得注意的是，这里最后一个成员是一个占位的0数组，下面的宏定义用于这个entries的索引



主要是mp\_init()函数。

先通过调用mpconfig()找到struct mpconf然后根据这个结构体内的entries信息对各个CPU结构体进行配置．如果proc->flag是MPPROC\_BOOT，说明这个入口对应的处理器是用于启动的处理器，我们把结构体数组cpus[ncpu]地址赋值给bootcpu指针．注意这里ncpu是个全局变量，那么这里实质上就是把cpus数组的第一个元素的地址给了bootcpu.

那个ismp是个全局变量，默认的初始值为0, 但是我们进行mp\_init()的时候，就把这个全局变量置为1了，如果出现任何entries匹配错误(switch找不到对应项，跳进default)，这个时候我们多可处理器的初始化就失败了，不能用多核处理器进行机器的运行，于是ismp置为0

### Multiprocessor Support

"symmetric multiprocessing" (SMP),

bootstrap processor (BSP)

is responsible for initializing the system and for booting the operating system;

application processors (APs)

are activated by the BSP only after the operating system is up and running

每个CPU对应一个LAPIC unit，传递中断，通过APIC ID识别正在使用的CPU

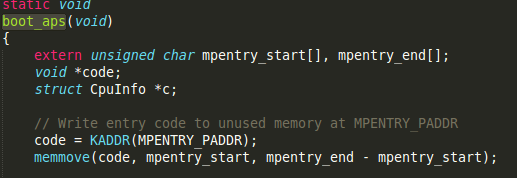
#### Exercise 1.

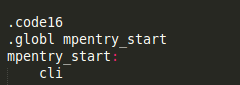
实现函数void \*mmio\_map\_region(physaddr\_t pa, size\_t size)，在文件kern/pmap.c中. 在MMIO区域映射内存到pa

### Application Processor Bootstrap

kern/mpconfig.c文件中的mp\_init()函数通过检索MP configuration table收集多处理器系统的信息。

对于SMP, 一开始系统是单核启动的(BSP), 而后完成启动后, 才会是APs.就是启动APs的入口函数---- boot\_aps()，boot\_aps()会将AP entry code (kern/mpentry.S)拷贝到0x7000 (MPENTRY\_PADDR)，并开始运行，但这个地址不固定，任何未使用的页对齐的640KB以下的物理地址都满足要求，这点与boot.S的要求不同



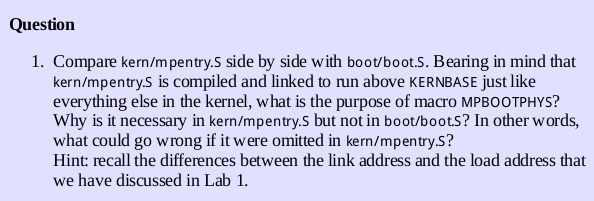
memmove会把几乎整个./kern/mpentry.S 文件从mpentry\_start 到mpentry\_end. 部分的汇编代码copy到指针code指向的地址处．

文件mpentry.S完成从实模式到保护模式。

控制流boot\_aps -> mpentry.S -> mp\_main()

#### Exercise 2.

modify your implementation of page\_init() in kern/pmap.c to avoid adding the page at MPENTRY\_PADDR to the free list, so that we can safely copy and run AP bootstrap code at that physical address



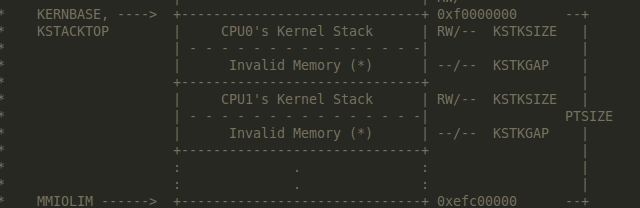
在boot.S中，由于尚没有启用分页机制，所以我们能够指定程序开始执行的地方以及程序加载的地址；但是，在mpentry.S的时候，由于主CPU已经处于保护模式下了，因此是不能直接指定物理地址的，给定线性地址，映射到相应的物理地址是允许的。

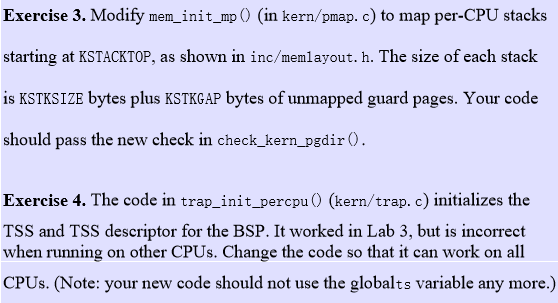
### Per-CPU State and Initialization

多核CPU的内存布局。

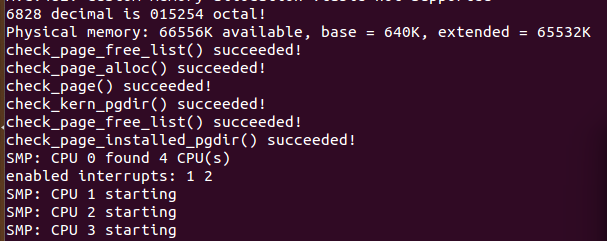
每个CPU独立的内核栈，array percpu\_kstacks[NCPU][KSTKSIZE] reserves space for NCPU's worth of kernel stacks.

每个CPU一个TSS，cpus[i].cpu\_ts，TSS descriptor 定义在 GDT entry gdt[(GD\_TSS0 >> 3) + i]中。





完成exercise3和4，命令行输入make qemu CPUS=4，输出如下。

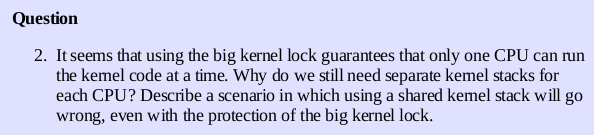


### Locking

kern/spinlock.h文件中定义了big kernel lock,，也叫作 kernel\_lock.用于防止多个cpu进入运行内核代码。lock\_kernel() 上锁操作，unlock\_kernel()用于释放锁。

#### Exercise 5

在 i386\_init() , mp\_main() ,trap() , 这几个函数中添加lock\_kernel()，在env\_run() 中添加unlock\_kernel();



答：

共享一个栈可能会让一个CPU执行的程序更改另一个CPU程序执行时栈中的数据，导致错误发生。

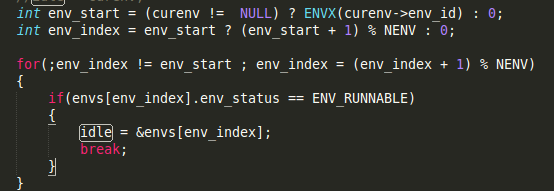
### Round-Robin Scheduling

循环轮转调度方式，从当前运行的environment开始在envs[]中循环遍历查找（如果当前没有进程在运行，从envs[]开始处查找），找到ENV\_RUNNABLE的进程，调用env\_run()

#### Exercise 6.

实现进程调度函数sched\_yield()，修改syscall()函数处理系统功能调用sys\_yield()

实验中出现的错误，期初用如下的方式实现循环轮转调度，但运行后发现会报错，后调试发现原因为该情况处理env\_index初始值为0时不能找到正确的进程，故会发生错误。

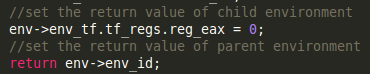




### System Calls for Environment Creation

sys\_exofork

父进程和子进程中的返回值不同，父进程返回子进程进程id，子进程返回0，通过以下语句实现。返回值存在寄存器eax中。



sys\_env\_set\_status

设置进程状态为ENV\_RUNNABLE 或 ENV\_NOT\_RUNNABLE

sys\_page\_alloc

static int sys\_page\_alloc(envid\_t envid, void \*va, int perm)，申请一个物理页，映射到进程envid的虚拟地址va处。

sys\_page\_map

static int sys\_page\_map(envid\_t srcenvid, void \*srcva, envid\_t dstenvid, void \*dstva, int perm)，将dstenvid在dstva处映射到和srcenvid的srcva对应的物理地址处，权限设为perm。

sys\_page\_unmap

解除映射。

#### Exercise 7

在文件kern/syscall.c中实现上述函数。

## Part B: Copy-on-Write Fork

Unix的fork()函数，子进程复制父进程的地址空间，并将数据一同拷贝过来，开销较大。Jos里dumnfork()函数就是用类似的方法实现。这种实现方法可能出现浪费现象。

因此最新版本的Unix采用了copy-on-write的方法，子进程先映射到父进程相同的地址空间，将共享页标记为read-only，当两个进程试图修改这些pages中的内容，产生page fault中断，此时alloc一个新的私有可写的页，复制该页的内容。

### User-level page fault handling

首先要实现用户层页错误处理函数，copy-on-write只是该函数众多用处中的一种。

### Setting the Page Fault Handler

注册页错误处理函数入口。

#### Exercise 8.

Implement the sys\_env\_set\_pgfault\_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call

### Normal and Exception Stacks in User Environments

正常运行时使用normal stack，使用USTACKTOP - PGSIZE 到 USTACKTOP-1的地址空间

用户模式下产生page fault使用的是user exception stack，使用UXSTACKTOP-PGSIZE 到UXSTACKTOP-1的地址空间

当用户模式下page fault 发生时，kernel会在user exception stack重新开始用户进程并运行page fault handler

用户层页错误处理函数user-level page fault handler能调用常规系统功能调用以修复问题，后通过一段汇编语言程序返回原错误代码和原运行栈执行。

### Invoking the User Page Fault Handler

在文件kern/trap.c中实现页错误处理函数page\_fault\_handler。

如果没有page\_fault\_handler被注册，按照先前的方式先打印异常信息，再销毁错误environment。否则将在用户栈中的异常信息拷贝到exception stack，并重新开始执行进程的page\_fault\_handler。因此这一步要设置进程的eip。

如果出错进程原本就是在exception stack上运行的，将原先的esp再减32bit（push an

empty 32-bit word），在此基础上拷贝栈帧信息。

#### Exercise 9.

Implement the code in page\_fault\_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. 问：(What happens if the user environment runs out of space on the exception stack?)

答：user\_mem\_assert会报错

### User-mode Page Fault Entrypoint

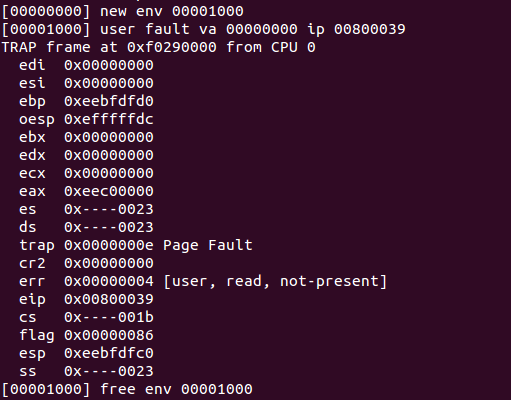
在文件lib/pfentry.S中实现一段汇编程序，该程序对页错误进行处理，并返回原始错误指令处运行，该程序就是内核函数static int sys\_env\_set\_pgfault\_upcall(envid\_t envid, void \*func)中的func

故顺序为set\_pgfault\_handler调用sys\_env\_set\_pgfault\_upcall注册\_pgfault\_handler，并且设置入口env\_pgfault\_upcall，出错时进入该函数，调用\_pgfault\_handler（真正的处理函数），后返回源代码运行。

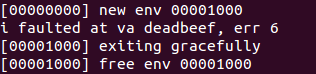
#### Exercise 10.

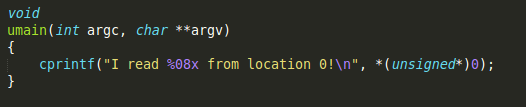
Implement the \_pgfault\_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

Run user/faultread



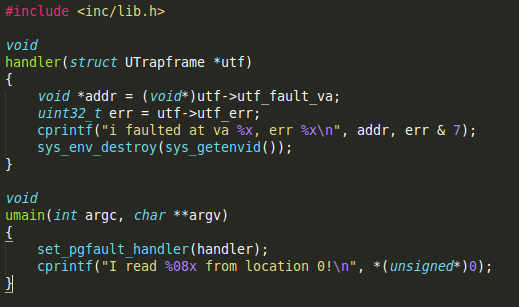
Run user/faultdie

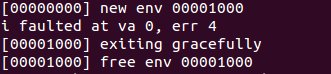




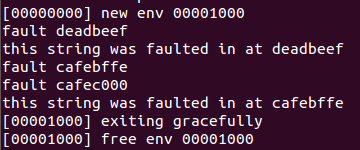
Faultread由于地址0x00000000不在主存中，故有页错误，且未设置页错误处理函数，故kernel不知道如何处理该错误，输出TrapFrame信息。

将faultdie中改为和faultread一样的操作，另外添加set\_pgfault\_handler函数设置处理函数，则能够正常处理该错误





Run user/faultalloc



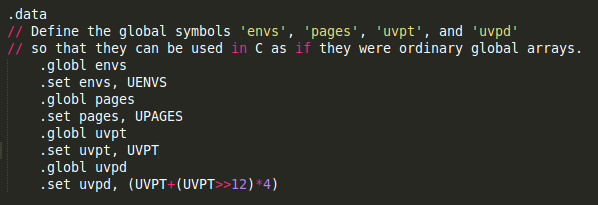
Run user/faultallocbad



文件faultalloc与faultallocbad输出不同是因为前者采用cprintf后者采用sys\_cputs

### Implementing Copy-on-Write Fork

The user-level lib/fork.c code must consult the environment's page tables for several of the operations above (e.g., that the PTE for a page is marked PTE\_COW). The kernel maps the environment's page tables at UVPT exactly for this purpose. It uses a clever mapping trick to make it to make it easy to lookup PTEs for user code. lib/entry.S sets up uvpt and uvpd so that you can easily lookup page-table information in lib/fork.c.



#### Exercise 12

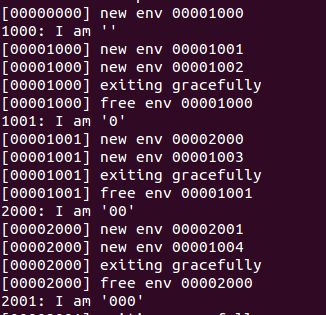
在文件lib/fork.c中实现fork, duppage and pgfault三个函数。

函数fork流程：

1. 调用sys\_pgfault\_handler，设置page fault handler为pgfault函数
2. 调用sys\_exofork，创建子进程
3. 调用duppage复制父进程的地址空间
4. 创建子进程自己的exception stack
5. 拷贝给子进程page fault handler
6. 设置子进程为ENV\_RUNNABLE状态

函数duppage，复制父进程的一个页给子进程，对于原本只读的空间，直接复制，对于原本可读可写的页，设置为PTE\_COW。

函数pgfault当对权限为PTE\_COW的页进行更改时发生的页错误进行处理，先申请新的一个页，临时映射到虚拟地址PFTEMP，复制内容到该页，后重新映射到原先出错的虚拟地址处，并解除和PFTEMP的映射。



## Part C: Preemptive Multitasking and Inter-Process communication (IPC)

### Clock Interrupts and Preemption

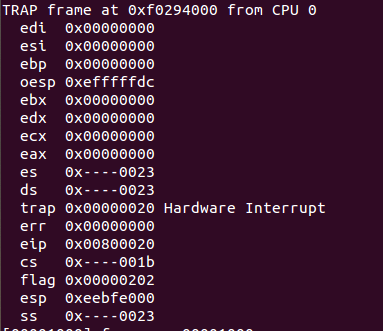
运行user/spin测试程序，出现死循环，无论是父进程还是kernel都不能重新获得CPU使用权。为防止出现死循环，扩展JOS实现来自外部硬件的硬件中断。

### Interrupt discipline

#### Exercise 13

修改kern/trapentry.S 和kern/trap.c 初始化IDT 中IRQ的入口，并添加定义中断处理函数 修改文件kern/env.ce 中的函数env\_alloc()使得进程运行时中断打开。

通过修改%eflags的FL\_IF实现

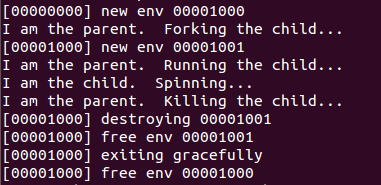


### Handling Clock Interrupts

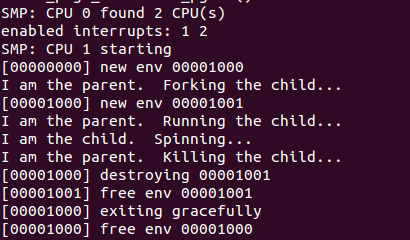
#### Exercise 14.

修改内核的trap\_dispatch() 函数，能够处理时钟中断。

Make run-spin



Make CPUS=2 run-spin



### Inter-Process communication (IPC)

我们已经实现的功能使得每个进程好像自己独立拥有一台机器。

另一个机制是进程间通信，管道模型是一个典型的例子

### IPC in JOS

### Sending and Receiving Messages

#### Exercise 15

实现文件kern/syscall.c中的函数sys\_ipc\_recv和函数sys\_ipc\_send函数，文件lib/ipc.c中的ipc\_recv和ipc\_send函数。

