# MIT JOS LAB4

**1713429 曾志韬 计算机科学与技术**

# 进程切换+缺页处理+进程之间的通信

**1 mmio\_map\_region //映射**

**2 page\_init // MPENTRY\_PADDR不要加入到free list**

**3 KERNBASE(0xf0000000）之上，实模式无法寻址**

**mpentry.S加载到了0x7000处，所以需要通过MPBOOTPHYS来寻址。**

**boot.S加载的位置本身就是实模式可寻址的低地址，所以不用额外转换。**

**4 mem\_init\_mp()将每个CPU堆栈映射在 KSTACKTOP 开始的区域，**

**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**

**5 trap\_init\_percpu**

**完成每个CPU的TSS初始化。设置ts\_esp0和ts\_ss0**

**设置全局描述符的时候加上cpu\_id作为索引值**

**6 加锁和解锁**

**加锁位置1 i386\_init()的 boot\_aps()函数前**

**加锁位置2 mp\_main()的函数末尾，这里还要加上 sched\_yield()**

**加锁位置3 trap()里面**

**释放锁 env\_run**

**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**

**env\_run() 中，恰好在 回到用户进程之前 释放内核锁**

**7 轮转调度进程**

**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。**

**8 Q3 lcr3地址空间的使用，切换之后仍然可以使用正确的地址**

**进程env\_pgdir的高地址的映射跟kern\_pgdir的是一样的**

**9 Q4保存寄存器状态的代码是 trap.c 中的 curenv->env\_tf = \*tf;**

**sys\_exofork:**

**10 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。**

**11** **sys\_env\_set\_pgfault\_upcall 系统调用**

**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，**

**12 page fault 处理**

**在page\_fault\_handler完成用户页面错误的处理，**

**主要是切换堆栈到异常栈，并设置异常栈内容，**

**最后设置EIP为页面错误处理函数的地址，切回用户态执行页面错误处理函数。**

**13 set\_pgfault\_handler() 其中调用缺页的syscall**

**14 缺页之后 置换页面**

**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。**

**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。**

**15添加中断处理函数**

**16 IPC功能，在syscall里面分发加新增的两个系统调用**

**调用 envid2env 时，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**



