**ucore实验总结**

**1.ucore实验Lab1知识总结**

**1.1Intel 80386**

ucore目前支持的硬件环境是基于Intel 80386以上的计算机系统。

Intel 80386是80x86系列中的第一种32位微处理器。80386的内部和外部数据总线都是32位，地址总线也是32位，可寻址高达4GB内存。

工作方式包括实模式、保护模式以及虚拟86模式。

**1.2Bootloader**

学习到了计算机启动是从BIOS开始，再由BIOS决定从哪个设备启动以及启动顺序，比如先从DVD启动再从硬盘启动等。计算机启动后，BIOS根据配置找到启动设备，并读取这个设备的第0个扇区，把这个扇区的内容加载到0x7c00,之后让CPU从0x7c00开始执行，这时BIOS已经交出了计算机的控制权，由被加载的扇区程序接管计算机。

第一个扇区的程序就叫Boot，它一般做一些准备工作，把操作系统内核加载进内存，并把控制权交给内核。由于Boot只能有一个扇区大小，即512字节，它所能做的工作很有限，因此它有可能不直接加载内核，而是加载一个叫Loader的程序，再由Loader加载内核。因为Loader不是BIOS直接加载的，所以它可以突破512字节的程序大小限制（在实模式下理论上可以达到1M）。如果Boot没有加载Loader而直接加载内核，我们可以把它叫做Bootloader。

Bootloader加载内核就要读取文件，在实模式下可以用BIOS的INT 13h中断。内核文件放在哪里，怎么查找读取，这里牵涉到文件系统，Bootloader要从硬盘（软盘）的文件系统中查找内核文件，因此Bootloader需要解析文件系统的能力。GRUB是一个专业的Bootloader，它对这些提供了很好的支持。

对于一个Toy操作系统来说，可以简单处理，把内核文件放到Bootloader之后，即从软盘的第1个扇区开始，这样我们可以不需要支持文件系统，直接读取扇区数据加载到内存即可。

**1.3MBR与磁盘分区**

在目前x86的系统架构中，系统硬盘位于第0号磁道：0到511KB的区块为MBR（硬盘中的每一个磁道容量为512KB），开机管理程序使用这块区域来储存第一阶段开机引导程序（stage1）。接着位于1到62号磁道作为第1.5阶段的开机引导程序（stage1.5），从第63号磁道开始才是操作系统的分区。

主引导记录（MBR，Master Boot Record）是位于磁盘最前边的一段引导（Loader）代码。它负责磁盘操作系统(DOS)对磁盘进行读写时分区合法性的判别、分区引导信息的定位，它由磁盘操作系统(DOS)在对硬盘进行初始化时产生。

**2.ucore实验Lab2知识总结**

**2.1lab2的代码结构和执行流程与lab1相比，其主要新增了以下功能：**

　　1. bootmain.S中的物理内存探测

1. 在新增的entry.S内核入口程序中开启了80386页机制

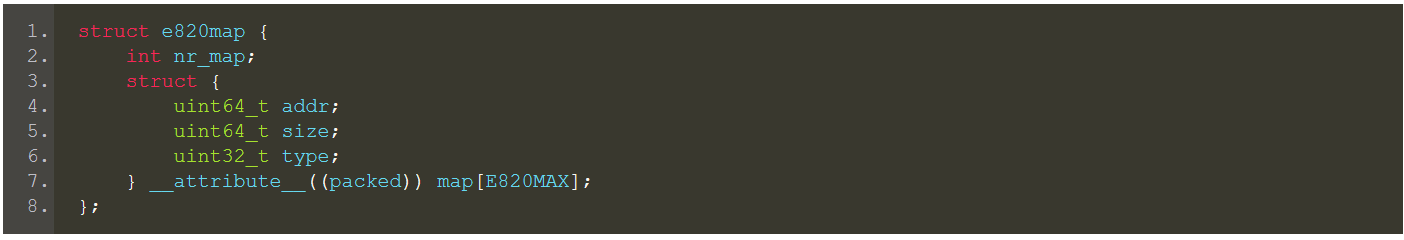
3. kern\_init内核总控函数中通过pmm\_init函数进行整个物理内存管理器的构建初始化

**2.2在本次实验主要学习到了ucore内核对物理内存的管理**

ucore被启动后，需要探测系统的物理内存布局来了解哪些物理内存空间是可用的。ucore是使用e820h中断来获取内存信息，而这个中断必须在实模式下使用，因此必须在bootloader引导进入保护模式前进行，这些收集到的数据将保存在物理地址0x8000处，通过代码中定义的e820map结构体进行映射。

为了进行物理内存的管理，操作系统必须先探测出当前硬件环境下内存的布局，了解具体哪些物理内存空间是可用的。

**e820map结构：**



**bootasm.S内存布局探测：**



**2.3启动分页机制**

ucore在lab2中开启了80386的分页机制，实现了基于平坦段模型的段页式内存管理，为后续虚拟内存的实现做好了准备。

2.3.1虚拟地址的概念

　　需要注意的是，在80386分页机制工作原理的许多资料中，开启了页机制后由指令(段选择子+段内偏移)所构成的地址被称为逻辑地址；而逻辑地址通过GDT或LDT等段表转换之后得到的地址被称为线性地址；如果开启了页机制，得到线性地址后还需要查找页表来得到最终的物理地址。

整个的转换过程大致为：逻辑地址->线性地址->物理地址。

程序中使用的地址都是逻辑地址，逻辑地址/虚拟地址通过GDT/LDT可以转换为线性地址，线性地址可以通过页表来转化为物理地址。

2.3.2ucore开启分页机制的细节

lab2以及往后的实验中，在ucore的虚拟空间设计中，开启了页机制后的内核是位于高位地址空间的，而低位内存空间则让出来交给用户应用程序使用。(详细见lab2 重要函数总结)

2.3.3开启分页机制前后的区别

ucore内核被bootloader指定加载的物理地址基址相对lab1而言是不变的。但在开启分页机制的前后，CPU翻译逻辑地址的方式也立即发生了变化。

开启分页机制前，内核程序的指令指针是指向低位内存的，而开启了页机制后，我们希望能够正确、无损的令内核的指令指针指向高位地址空间，但保证其最终访问的物理地址不变，依然能够正确的执行。

2.3.4一些寄存器的标识及功能

CR开头的寄存器为控制寄存器。

CR0寄存器包含系统控制标志，这些标志控制着处理器的运行模式和状态。分页(CR0 的第 31 位)。置 1 启用分页，置 0 不启用分页。当禁用分页 时，所有的线性地址都当作物理地址对待。

CR3寄存器包含页目录表的物理基地址和二个标志(PCD和PWT)。该寄存器也被称为页目录基地址寄存器(PDBR)。ucore中用boot\_cr3(mm/pmm.c)来记录这个值。

**2.4 ucore是如何实现物理内存管理功能的**

　　1.初始化物理内存管理器的入口位于总控函数的pmm\_init函数。(详细见lab2 重要函数总结)

2.物理内存管理器pmm\_manager初始化

pmm\_init在得到了内核页目录表的物理地址后(boot\_cr3)，便通过init\_pmm\_manager函数初始化了物理内存管理器框架。该框架(全局变量pmm\_manager)是一个被抽象出来的，用于表达物理内存管理行为的函数指针集合，内核启动时会对这一函数指针集合进行赋值。

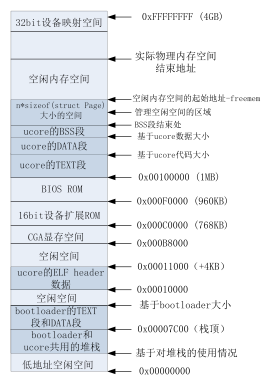
3.通过探测出的内存布局设置空闲物理映射空间

　　ucore使用一个通用的Page结构，来映射每个被管理的物理页面。

　　其中调用的init\_memmap函数，会通过pmm\_manage框架的init\_memmap，由指定的算法来初始化其内部结构。在ucore lab2的参考答案中，默认使用的是default\_pmm\_manager是一种第一次适配算法(first fit)。

通过page\_init函数可以利用之前在bootasm.S中探测到的e820map布局结构，初始化空闲物理内存空间。

**初始化完毕后ucore物理内存布局示意图：**

****

　　其中**page\_init**中的**end**指向了BSS段结束处，**freemem**指向空闲内存空间的起始地址。**pages(内核页表)**位于"管理空闲空间的区域"这一内存块中。**实际可用于分配/释放的空闲物理页**位于内存空间起始地址~实际物理内存空间结束地址之间。

**3.ucore实验Lab3知识总结**

**3.1lab3在lab2的基础上，主要新增了以下功能：**

　　1. kern\_init总控函数中新增了vmm\_init、ide\_init、swap\_init函数入口，分别完成了虚拟内存管理器、ide硬盘交互以及虚拟内存磁盘置换器的初始化。

　　2. 在trap.c的中断处理分发函数trap\_dispatch中新增了对14号中断(页访问异常)的处理逻辑do\_pgfault。

**3.2. ucore lab3实验细节分析**

lab3在虚拟内存功能的实现中，ucore需要借助lab1中建立的中断机制来处理缺页异常，同时也依赖lab2中实现的物理内存管理功能。

**3.3ucore虚拟内存管理框架**

ucore在lab3中新增了vma\_struct结构(kern/mm/vmm.h)来描述合法的连续虚拟内存空间块，一个进程合法的虚拟地址空间段将以vma集合的方式表示。

在ucore中，以vma\_struct虚地址空间的大小顺序可以组成一个双向循环链表，与lab2类似，vma\_struct反向包裹list\_link链表节点属性，利用le2vma宏可以使用page\_link节点找到所关联的vma\_struct。（代码见总结）

**3.4虚拟内存关联物理页换入换出分析**

3.4.1进行换出

虚拟内存页的换出分为主动与被动两种换出策略，在lab3中，ucore只实现了基于被动的换出策略。在ucore中，当申请分配物理页面时如果发现可用的物理内存不足时，会进行可置换物理页的换出。通过某种置换算法将选中的物理页暂时置换到磁盘的交换分区中，以腾出空闲的物理页以供分配。

分配物理内存的代码位于pmm.c中的alloc\_pages函数，alloc\_pages是在lab2中已有的。

3.4.2进行换入

当CPU进行内存访问时，发现所得到的线性地址对应的二级页表项的P位为0(不存在)，便产生页异常中断(页异常还可能在其它情况下出现)，ucore会接受异常并进入页异常中断服务例程。

在lab1中，硬件在一些中断发生时会将错误号压入栈中，ucore建立的中断机制可以通过中断栈帧trap\_frame中的tr\_err获取到一个32位的错误号。在页异常中断中，这个错误号的不同bit位标识了发生异常时的各种信息，同时80386的cr2页异常地址寄存器中也会保留最后一次页异常发生时所访问的32位线性地址。

**3.5如何在置换时从swap磁盘分区中找到二级页表项对应的物理页数据**？

　　要想在换入时准确的找到二级页表项所映射物理内存页内容在磁盘交换分区中的位置，需要建立某种映射关系或是映射表。在ucore中，没有单独的另外构建一张新的映射表，而是利用二级页表项。

　　在80386CPU中，内存寻址时线性地址对应二级页表项的Present存在位是至关重要的。当P位为1时，代表所映射的物理页存在，访问正常；而P位为0时，代表不存在，则整个二级页表项的其它位都没有意义了。ucore利用了这一特性，在进行物理页虚拟内存的置换后，将P位设置为0的同时，还将二级页表项pte的高24位作为在磁盘交换扇区中的偏移索引，并为此单独定义了swap\_entry\_t这一32位结构表示处于交换区的虚拟内存映射页的状态。

pte在lab3中有了三种不同的状态：

　　1. 全为0，代表未建立对应物理页的映射

　　2. P位为1，代表已建立对应物理页的映射

　　3. P位为0，但高24位不为0。代表所映射的物理页存在，只是被交换到了磁盘交换区中。

在本实验中只存在一个页表(内核页表)，为了简单起见就直接令swap\_entry\_t中的高24位偏移 \* 8 = 对应存储磁盘交换起始扇区号。

**4.ucore实验Lab4知识总结**

Lab4主要让我学习到了进程和线程的相关知识：

进程概念：

操作系统需要协调管理各个程序对CPU资源的使用，为此抽象出了进程(Process)的概念。进程能够随时在执行与挂起中切换，且每次恢复运行时都能够接着上次被打断挂起的地方接着执行。

线程概念：

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

**4.1lab4相对于lab3的主要改进**

　　1. 实现了进程/线程的创建、初始化、退出以及控制线程的运行状态等功能。

　　2. 实现了线程的上下文切换功能。

　　3. 实现了forkrets，用于do\_forks创建子线程后调用的返回处理。

4. 实现了一个最基本的FIFO的线程CPU调度算法。

**4.2ucore lab4实验细节分析**

Lab4是在lab2、lab3建立起了较为完善的物理、虚拟内存管理机制的基础上，建立起内存空间独立的进程机制，以及执行流独立的线程功能。

在ucore中，并不显式的区分进程与线程，都使用同样的数据结构proc\_struct进程/线程管理块进行管理。当不同的线程控制块对应的页表(cr3)相同时，ucore认为是同一进程下的不同线程。

4.2.1 线程的创建与初始化

　　ucore在lab4中建立了进程/线程的机制，在总控函数kern\_init中，通过pmm\_init创建了常驻内核的第0号线程idle\_proc和第1号线程init\_proc。

整个ucore内核可以被视为一个进程(内核进程)，而上述两个线程的cr3指向内核页表boot\_cr3，且其代码段、数据段选择子特权级都处于内核态，属于内核线程。

　在proc\_init函数可以看到，ucore中要创建一个新的内核线程(init\_proc)，是通过kernel\_thread实现的。创建内核线程时，新线程相当于是current当前线程fork出的一个子线程。

　　调用kernel\_thread函数时，需要指定线程的执行入口(例如：init\_main)，入口函数的参数(例如："Hello world!")，以及指定是否需要采取写时复制的机制进行fork时父子进程的内存映射。

4.2.2 ucore线程调度时线程上下文的切换

　　由于在proc\_init中，令全局变量current指向了idle\_proc，代表当前占用CPU的是线程idel\_proc，设置idel\_proc的need\_resched为1。proc\_init函数返回，总控函数kern\_init完成了一系列初始化工作后，最终执行了cpu\_idle函数。

　　cpu\_idle函数可以视为idle\_proc的执行流，在其中进行了一个while(1)的无限循环，当发现自己需要被调度时，调用schedule函数进行一次线程的调度。

4.2.3 线程的上下文

　　在proc\_run中，调用了switch\_to函数。switch\_to是汇编实现的函数(子过程)，其参数是两个struct context结构体的指针。

　　第一个参数from代表着当前线程的上下文，第二个参数to代表着新线程的上下文，switch\_to的功能就是保留current线程的上下文至from上下文结构中，并将to上下文结构中的内容加载到CPU的各个寄存器中，恢复新线程的执行流上下文现场。

**由于函数调用时是先调用后返回的，整个执行流程体现出一种先进后出的结构，因此普遍采用栈来实现函数调用，且不同执行流之间的栈是互相隔离的。ucore中，线程的上下文除了各个通用寄存器、段寄存器、指令指针寄存器等寄存器上下文之外，还需要额外的维护各自的栈结构。当然如果发生了进程间的切换，还需要切换页表。**

1. **关于ucore实验的一些优化**

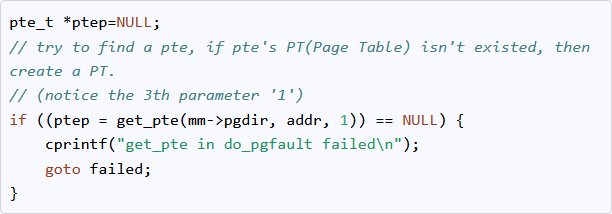
对照答案对代码进行优化。

1.do\_pgfault调用get\_pte时没有检查返回值。

个人代码：

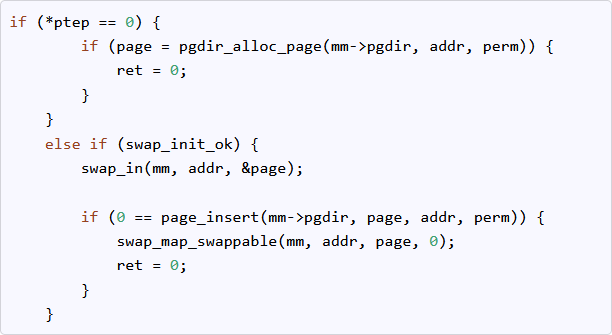


答案的代码：

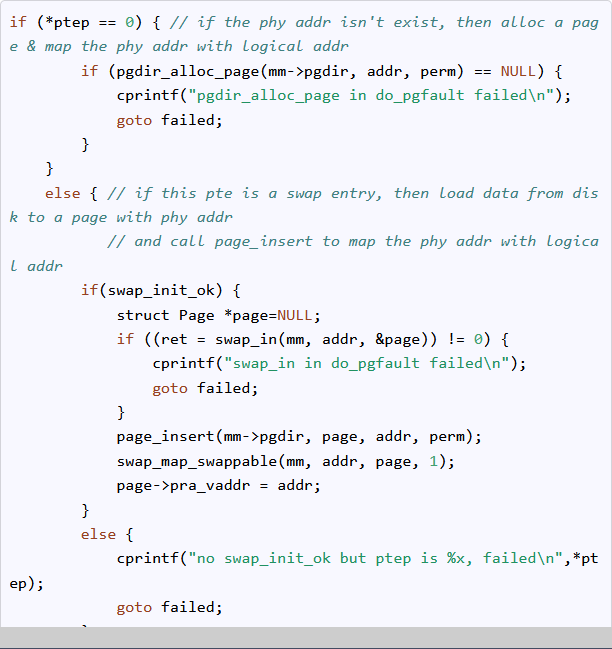


2.do\_pgfault调用pgdir\_alloc\_page和swap\_in失败后没打印错误信息以方便定位。

个人代码：



答案的代码：



**[ucore 拓展之——用 gdb 调试程序从用户态到内核态再回到用户态的整个流程](https://zhuanlan.zhihu.com/p/88210724)**