

---

现在还是很初始的草稿版本，需要进一步修订、改进和增加内容。如发现内容有问题或有好的建议请电邮告诉我们（[chyyuu@gmail.com](mailto:chyyuu@gmail.com)），谢谢！

# 操作系统设计与实现

## The Design & Implementation of Operating System

陈渝 向勇

清华大学计算机系

2011 年 6 月

---

# 第1章 操作系统简介

[要点 (OSP): OS 定义: 位于 1.2.1]

[要点 (OSP): OS 抽象: 位于 1.2.2]

对于在校的学生和已经参加工作的工程师而言,能否以较小的时间和精力比较全面地了解操作系统呢?陆游老夫子说过“纸上得来终觉浅,绝知此事要躬行”,也许在了解基本的操作系统概念和原理基础上,通过实际动手来一步一步分析、设计和实现一个微型化的操作系统,会发现操作系统原来如此,概念原理和实际实现之间有紧密的联系和巨大的差异。

一方面现在的操作系统课本越来越庞大和抽象,反而忽视了实践。要知道,操作系统的设计实现是在没有教科书的情况下完成的。换句话说,现有操作系统的实现,再有操作系统的课本。而另一方面实际的操作系统相当庞大,如 Linux、Windows 等,都是上百万行的源代码规模,实现这些软件的目的是给人用的,不是给人学的。能否在这两方面找到一个平衡?

早期的 UNIX 操作系统和 MIT 教授 Frans Kaashoek 博士等基于 UNIX v6 设计的 xv6 操作系统给了我们启发:对一个计算机专业的本科生而言,设计实现一个操作系统有挑战但是可行!本书想进行这样的教学尝试,以设计实现一个微型但全面的“麻雀”操作系统—ucore 为基本目标,以增量式地完成各种基于 ucore 操作系统的实验为实践过程,以在此过程中逐步介绍的操作系统的概念和原理为实践指导,做到有“理”可循和有“码”可查,最终让读者了解和掌握操作系统的原理、设计与实现。

## 1.1 应具备的背景知识和学习环境

设计实现操作系统其实就是设计实现一个软件,所以本书的例子和描述需要读者学习过计算机原理课程、程序设计课程,掌握 C 语言编程(了解指针等的编程),对基于 Intel 80386 处理器体系结构有一定的了解,大致了解基于 Intel 80386 的汇编语言。本书涉及的例子和实验可在 Windows 环境和 Linux 环境下采用命令行(CLI)方式和集成开发环境(IDE)方式进行编译和运行,所以,最好能够有一台 PC 计算机用于进行操作系统实验。

## 1.2 操作系统的概念

### 1.2.1 操作系统的定义

操作系统是计算机系统机构中的一个系统软件,它的职能主要有两个:对下面(也就是计算机硬件),有效地组织和管理计算机系统硬件资源(包括处理器、内存、硬盘、显示器、键盘、鼠标等各种外设);对上面(应用程序或用户),提供简洁的服务功能接口,屏蔽硬件管理带来的差异性和复杂性,使得应用程序和用户能够灵活、方便、有效地使用计算机。为了完成这两个职能,操作系统需要在其内部实现中合理地组织计算机中的软硬件资源的使用分配和处理流程,使整个计算机系统能高效地运行。

---

## 1.2.2 操作系统的抽象

操作系统为了能够更好地管理计算机系统并对应用程序提供便捷的服务，在操作系统的发展过程中，计算机科学家提出了如下四个抽象概念，奠定了操作系统的基础。操作系统原理中的其他概念基本上都可基于上述这四个操作系统抽象。

### 中断 (Interrupt)

简单地说，中断是处理器在执行过程中的突变，用来响应处理器状态中的特殊变化。比如当应用程序正在执行时，产生了时钟外设中断，导致操作系统打断当前应用程序的执行，转而去处理时钟外设中断，处理完毕后，再回到应用程序被打断的地方继续执行。在操作系统中，有三类中断：外设中断 (Device Interrupt)、陷阱中断 (Trap Interrupt) 和故障中断 (Fault Interrupt，也称为 exception，异常)。外设中断由外部设备引起的外部 I/O 事件如时钟中断、控制台中断等。外设中断是异步产生的，与处理器的执行无关。故障中断是在处理器执行指令期间检测到不正常的或非法的内部事件（如除零错、地址访问越界）。陷阱中断是在程序中使用请求操作系统服务的系统调用而引发的有意事件。在后面的叙述中，如果没有特别指出，我们将用简称中断、陷阱、故障来区分这三种特殊的中断事件，在不需要区分的地方，统一用中断表示。

### 进程 (Process)

简单地说，进程是一个正在运行的程序。在计算机系统中，我们可以“同时”运行多个程序，这个“同时”，其实是操作系统给用户造成的一个“幻觉”。大家知道，处理器是计算机系统中的硬件资源。为了提高处理器的利用率，操作系统采用了多道程序技术。如果一个程序因某个事件而不能运行下去时，就把处理器占用权转交给另一个可运行程序。为了刻画多道程序的并发执行的过程，就要引入进程的概念。从操作系统原理上看，一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。操作系统中的进程管理需要协调多道程序之间的关系，解决对处理器分配调度策略、分配实施和回收等问题，从而使处理器资源得到最充分的利用。

### 虚存 (Virtual Memory)

简单地说，虚存就是操作系统通过处理器中的 MMU 硬件的支持而给应用程序和用户提供一个大的（超过计算机中的内存条容量）、一致的（连续的地址空间）、私有的（其他应用程序无法破坏）的存储空间。这需要操作系统将内存和硬盘结合起来管理，为用户提供一个容量比实际内存大得多的虚拟存储器，并且需要操作系统为应用程序分配内存空间，使用户存放在内存中的程序和数据彼此隔离、互不侵扰。操作系统中的虚存管理与处理器的 MMU 密切相关。

### 文件 (File)

简单地说，文件就是存放在永久存储介质（比如硬盘、光盘、U 盘等）上，方便应用程序和用户读写的数据。当处理器需要访问文件中的数据时，可通过操作系统把它们装入内存。放在硬盘上的程序也是一种文件。文件管理的任务是有效地支持文件的存储、检索和修改等操作。

## 1.3 “麻雀”操作系统--ucore

写一个操作系统难吗？别被现在上百万行的 Linux 和 Windows 操作系统吓倒。当年 Thompson 乘他老婆带着小孩度假留他一人在家时，写了 UNIX；当年 Linus 还是一个 21 岁大

学生时完成了 Linux 雏形。站在这些巨人的肩膀上，我们能否也尝试一下做“巨人”的滋味呢？

MIT 的 Frans Kaashoek 等在 2006 年参考 PDP-11 上的 UNIX Version 6 写了一个可在 X86 上跑的操作系统 xv6（基于 MIT License），用于学生学习操作系统。我们可以站在他们的肩膀上，基于 xv6 的设计，尝试着一步一步完成一个从“空空如也”到“五脏俱全”的“麻雀”操作系统—ucore，此“麻雀”包含虚存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要内核功能，总的内核代码量（C+asm）不会超过 5K 行。充分体现了“小而全”的指导思想。

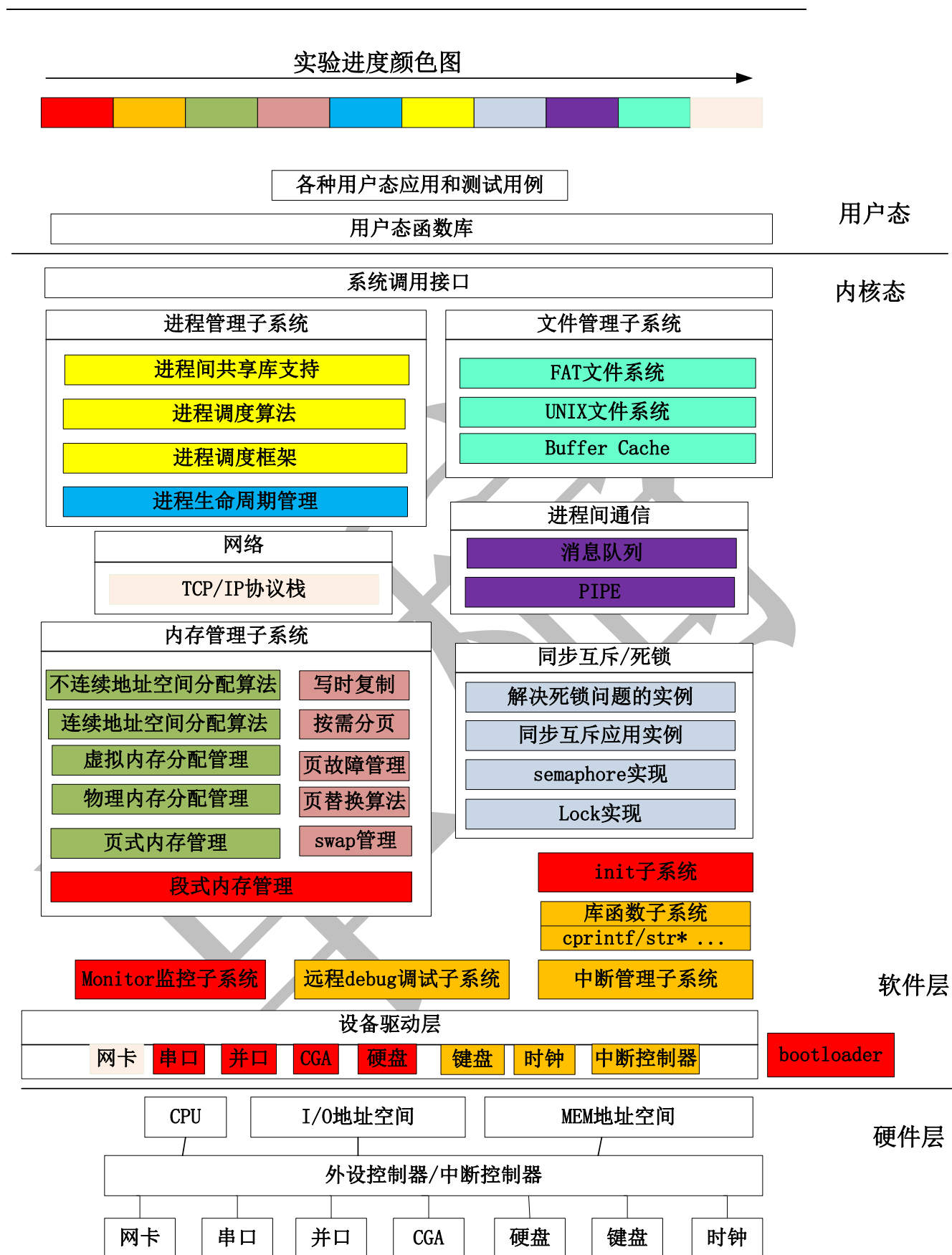
ucore 的运行环境可以是真实的 X86 计算机，不过考虑到调试和开发的方便，我们可采用 X86 模拟器，比如 QEMU、BOCHS 等，或 X86 虚拟运行环境，比如 VirtualBox、VMware Player 等。ucore 的开发环境主要是 GCC 中的 gcc、gas、ld 和 MAKE 等工具，也可采用集成了这些工具的 IDE 开发环境 Eclipse-CDT。运行环境和开发环境既可以在 Linux 或 Windows 中使用。

那我们准备如何一步一步实现 ucore 呢？安装一个操作系统的开发过程，我们可以有如下的开发步骤：

- 1) 启动操作系统的 bootloader，用于了解操作系统启动前的状态和要做的准备工作，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断--“外设中断”，“陷阱中断”，内核态和用户态的区别；
- 2) 内存管理子系统，用于理解 x86 分段/分页模式，了解操作系统如何管理物理内存和虚存、页表管理、一类中断-“故障中断”、缺页故障处理、基于页的内存替换；
- 3) 进程管理子系统，用于了解进程创建、执行、切换和结束的动态管理过程，了解在用户态通过系统调用得到内核态的内核服务的过程；
- 4) 处理器调度子系统，用于理解操作系统的调度过程和调度算法；
- 5) 同步互斥子系统，用于了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁；
- 6) 进程间通信子系统：用于了解进程间如何进行信息交换和共享；
- 7) 文件系统，了解文件系统的具体实现，与进程管理等关系，了解缓存对操作系统 IO 访问的性能改进，了解虚拟文件系统（VFS）、buffer cache 和 disk driver 之间的关系。
- 8) 网络协议栈（选做）：了解网卡驱动、TCP/IP 协议栈实现和 Web 应用。

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如下图所示（要标注处各个 proj 的特点，可在最后写）：





---

## 1.4 处理器硬件基础简介

要想深入理解 ucore，就需要了解支撑 ucore 运行的硬件环境，即了解处理器体系结构（了解硬件对 ucore 带来影响）和机器指令集（读懂 ucore 的汇编）。ucore 目前支持的硬件环境是基于 Intel 80386 以上的计算机系统。更多的硬件相关内容（比如保护模式等）将随着实现 ucore 的过程逐渐展开介绍。

### 1.4.1 Intel 80386 运行模式

80386 有四种运行模式：实模式、保护模式、SMM 模式和虚拟 8086 模式。这里对涉及 ucore 的实模式、保护模式做一个简要介绍。

**实模式：**80386 加电启动后处于实模式运行状态，在这种状态下软件可访问的物理内存空间不能超过 1MB，且无法发挥 Intel 80386 以上级别的 32 位 CPU 的 4GB 内存管理能力。实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。

**实模式：**实模式是为了和 8086 处理器兼容而设置的。在实模式下，80386 处理器就相当于一个快速的 8086 处理器。80386 处理器被复位或加电的时候以实模式启动。这时候处理器中的各寄存器以实模式的初始化值工作。80386 处理器在实模式下的存储器寻址方式和 8086 是一样的，由段寄存器的内容乘以 16 当做基地址，加上段内的偏移地址形成最终的物理地址，这时候它的 32 位地址线只使用了低 20 位，即可访问 1MB 的物理地址空间。在实模式下，80386 处理器不能对内存进行页机制管理，所以指令寻址的地址就是内存中实际的物理地址。在实模式下，所有的段都是可以读、写和执行的。实模式下 80386 不支持优先级，所有的指令相当于工作在特权级（即优先级 0），所以它可以执行所有特权指令，包括读写控制寄存器 CR0 等。实模式下不支持硬件上的多任务切换。实模式下的中断处理方式和 8086 处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和 8086 处理器一样，每 4 个字节组成一个中断向量，其中包括两个字节的段地址和两个字节的偏移地址。

**保护模式：**实际上，80386 就是通过在实模式下初始化控制寄存器，GDTR, LDTR, IDTR 与 TR 等管理寄存器以及页表，然后再通过加载 CR0 使其中的保护模式使能位置位而进入保护模式的。当 80386 工作在保护模式下时，其所有的 32 根地址线都可供寻址，物理寻址空间高达 4GB。在保护模式下，支持内存分页机制，提供了对虚拟内存的良好支持。保护模式下 80386 支持多任务，还支持优先级机制，不同的程序可以运行在不同的优先级上。优先级一共分 0~3 4 个级别，操作系统运行在最高的优先级 0 上，应用程序则运行在比较低的级别上；配合良好的检查机制后，既可以在任务间实现数据的安全共享也可以很好地隔离各个任务。

### 1.4.2 Intel 80386 内存架构

80386 是 32 位的处理器，即可以寻址的物理内存地址空间为  $2^{32}=4G$  字节。在理解操作系统的过程中，需要用到三个地址空间的概念。地址是访问地址空间的索引。**物理内存地址空间**是处理器提交到总线上用于访问计算机系统中的内存和外设的最终地址。一个计算机系

统中只有一个物理地址空间。**线性地址空间**是每个运行的应用程序看到的地址空间，在操作系统的虚存管理之下，每个运行的应用程序都认为自己独享整个计算机系统的地址空间，这样可让多个运行的应用程序之间相互隔离。处理器负责把线性地址转换成物理地址。一个计算机系统中可以有多个线性地址空间（比如一个运行的程序就可以有一个私有的线性地址空间）。线性地址空间的大小与物理地址空间的大小没有必然的连续。**逻辑地址空间**是应用程序直接使用的地址空间。这是由于 80386 中无法禁用段机制，使得逻辑地址一直存在。比如如下 C 代码片段：

```
int boo=1;
int *foo=&a;
```

这里的 boo 是一个整型变量，foo 变量是一个指向 boo 地址的整型指针变量，foo 中储存的内容就是 boo 的逻辑地址。逻辑地址由一个 16 位的段寄存器和一个 32 位的偏移量构成。Foo 中放的就是 32 位的偏移量，而对应的段信息位于段寄存器中。

上述三种地址的关系如下：

- 分段机制启动、分页机制未启动：逻辑地址->**段机制处理**->线性地址=物理地址
- 分段机制和分页机制都启动：逻辑地址->**段机制处理**->线性地址->**页机制处理**->物理地址

### 1.4.3 Intel 80386 寄存器

80386 的寄存器可以分为 8 组：通用寄存器，段寄存器，指令指针寄存器，标志寄存器，系统地址寄存器，控制寄存器，调试寄存器，测试寄存器，它们的宽度都是 32 位。一般程序员看到的寄存器包括通用寄存器，段寄存器，指令指针寄存器，标志寄存器。

**General Register(通用寄存器)：**EAX/EBX/ECX/EDX/ESI/EDI/ESP/EBP 这些寄存器的低 16 位就是 8086 的 AX/BX/CX/DX/SI/DI/SP/BP，对于 AX,BX,CX,DX 这四个寄存器来讲,可以单独存取它们的高 8 位和低 8 位 (AH,AL,BH,BL,CH,CL,DH,DL)。它们的含义如下：

EAX: 累加器  
EBX: 基址寄存器  
ECX: 计数器  
EDX: 数据寄存器  
ESI: 源地址指针寄存器  
EDI: 目的地址指针寄存器  
EBP: 基址指针寄存器  
ESP: 堆栈指针寄存器

## 通用寄存器

|     |    |    |   |    |
|-----|----|----|---|----|
| 31  | 23 | 15 | 7 | 0  |
| EAX |    |    |   | AL |
| EDX |    |    |   | DL |
| ECX |    |    |   | CL |
| EBX |    |    |   | BL |
| EBP |    |    |   | BP |
| ESI |    |    |   | SI |
| EDI |    |    |   | DI |
| ESP |    |    |   | SP |

**Segment Register(段寄存器, 也称 Segment Selector, 段选择符, 段选择子):** 除了 8086 的 4 个段外(CS,DS,ES,SS), 80386 还增加了两个段 FS, GS, 这些段寄存器都是 16 位的, 它们的含义如下:

CS: 代码段(Code Segment)

DS: 数据段(Data Segment)

ES: 附加数据段(Extra Segment)

SS: 堆栈段(Stack Segment)

FS: 附加段

GS 附加段

## 段寄存器

|                    |   |   |
|--------------------|---|---|
| 15                 | 7 | 0 |
| CS <CODE SEGMENT>  |   |   |
| SS <STACK SEGMENT> |   |   |
| DS <DATA SEGMENT>  |   |   |
| ES <DATA SEGMENT>  |   |   |
| FS <DATA SEGMENT>  |   |   |
| GS <DATA SEGMENT>  |   |   |

**Instruction Pointer(指令指针寄存器):** EIP 的低 16 位就是 8086 的 IP, 它存储的是下一条要执行指令的内存地址, 在分段地址转换中, 表示指令的段内偏移地址。

## 状态寄存器和指令寄存器

|                           |    |    |   |   |
|---------------------------|----|----|---|---|
| 31                        | 23 | 15 | 7 | 0 |
| EFLAGS                    |    |    |   |   |
| EIP <INSTRUCTION POINTER> |    |    |   |   |



- proj3.1: 内置监控自身运行状态的 ucore
- proj4: 可管理中断和处理基于中断的键盘/时钟的 ucore
- proj4.1.1: 支持通过中断方式的内核态/用户态切换的 ucore

可以掌握如下知识:

- 与操作系统原理相关
  - I/O 设备管理: 涉及程序循环检测方式和中断启动方式、I/O 地址空间
  - 内存管理: 基于分段机制的内存管理
  - 异常处理: 涉及中断、故障和陷阱
  - 特权级: 内核态和用户态
- 操作系统原理之外
  - PC 从加电到加载操作系统内核的整个过程
  - OS 内核在内存中的布局
  - 并口访问、串口访问、CGA 字符显示、硬盘数据访问、时钟访问
  - ELF 执行文件格式
  - 栈的实现并实现函数调用栈跟踪函数
  - 调试操作系统

其实这一章的内容与操作系统原理相关的部分较少, 与计算机体系结构(特别是 x86)的细节相关的部分较多。但这些内容对写一个操作系统关系较大, 要知道操作系统是直接和硬件打交道的软件, 所以它需要“知道”需要硬件细节, 才能更好地控制硬件。另一方面, 部分内容涉及到操作系统的重要抽象--中断类异常, 能够充分理解中断类异常为以后进一步了解进程切换、上下文切换等概念会很有帮助。

本章的实验内容涉及的是写一个 bootloader 能够启动一个操作系统--ucore。在完成 bootloader 的过程中, 逐渐增加 bootloader 和 ucore 的能力, 涉及 x86 处理器的保护模式切换、解析 ELF 执行文件格式等, 这对于理解操作系统的加载过程以及在操作系统在内存中的位置、内存管理、用户态与内核态的区别等有帮助。而相关 project 中 bootloader 和操作系统本身的字符显示的 I/O 处理、读硬盘数据的 I/O 处理、键盘/时钟的中断处理等内容, 则是操作系统原理中一般在靠后位置提到的设备管理的实际体现。纵观操作系统的发展史, 从早期到现在的操作系统主要功能之一就是完成繁琐的 I/O 处理, 给上层应用提供比较简洁的 I/O 服务, 屏蔽硬件处理的复杂性。这也是操作系统的虚拟机功能的体现。另外, 本章还介绍了对硬件模拟器的使用, 对操作系统的 panic 处理和远程 debug 功能的支持, 这样有助于读者能够方便地分析操作系统中的错误和调试操作系统。由于本章涉及的硬件知识较多, 无疑增大了读者的阅读难度, 需要读者在结合阅读本章并实际动手实验来进行深入理解。

## 3.1 可切换到保护模式并显示字符的 bootloader

### 3.1.1 实验目标

操作系统是一个软件, 也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件--bootloader 来完成这些工作。为此, 我们需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader, 为将来启动操作系统做准备。proj1 提供了一个非常小的 bootloader, 整个 bootloader 的大小小于 512 个字节, 这样才能放到硬盘的主引导扇区中。通过分析和实现这个 bootloader, 读者可以了解到:

- 基于分段机制的存储管理

- 设备管理的基本概念
- PC 启动 `bootloader` 的过程
- `bootloader` 的文件组成
- 编译运行 `bootloader` 的过程
- 调试 `bootloader` 的方法
- 在汇编级了解栈的结构和处理过程
- 保护模式切换到保护模式的方法
- 通过串口/并口/CGA 输出字符的方法

### 3.1.2 proj1 概述

## 1. 实现描述

`proj1` 实现了一个简单的 `bootloader`，主要完成的功能是初始化寄存器内容，完成实模式到保护模式的转换，在保护模式下通过 `PIO` 方式控制串口、并口和 `CGA` 等进行字符串输出。

## 2. 项目组成

[要点（非 OSP）：bootloader 的编译生成过程]

`lab1` 中包含的第一个工程小例子是 `proj1`：一个可以切换到保护模式并显示字符串的 `bootloader`。`proj1` 的整体目录结构如下所示：

```
proj1 /
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- libs
|   |-- types.h
|   `-- x86.h
|-- Makefile
`-- tools
    |-- function.mk
    |-- gdbinit
    `-- sign.c
```

3 directories, 9 files

其中一些比较重要的文件说明如下：

- `bootasm.S`：定义并实现了 `bootloader` 最先执行的函数 `start`，此函数进行了一定的初始化，完成了从实模式到保护模式的转换，并调用 `bootmain.c` 中的 `bootmain` 函数。
- `bootmain.c`：定义并实现了 `bootmain` 函数实现了通过屏幕、串口和并口显示字符



串。

- **asm.h**: 是 **bootasm.S** 汇编文件所需要的头文件, 主要是一些与 X86 保护模式的段访问方式相关的宏定义。
- **types.h**: 包含一些无符号整型的缩写定义。
- **x86.h**: 一些用 GNU C 嵌入式汇编实现的 C 函数 (由于使用了 **inline** 关键字, 所以可以理解为宏)。
- **Makefile** 和 **function.mk**: 指导 **make** 完成整个软件项目的编译, 清除等工作。
- **sign.c**: 一个 C 语言小程序, 是辅助工具, 用于生成一个符合规范的硬盘主引导扇区。
- **gdbinit**: 用于 **gdb** 远程调试的初始命令脚本

从中, 我们可以看出 **bootloader** 主要由 **bootasm.S** 和 **bootmain.c** 组成, 当你完成编译后, 你会发现这个 **bootloader** 只有区区的 3 百多字节。下面是编译运行 **bootloader** 的过程。

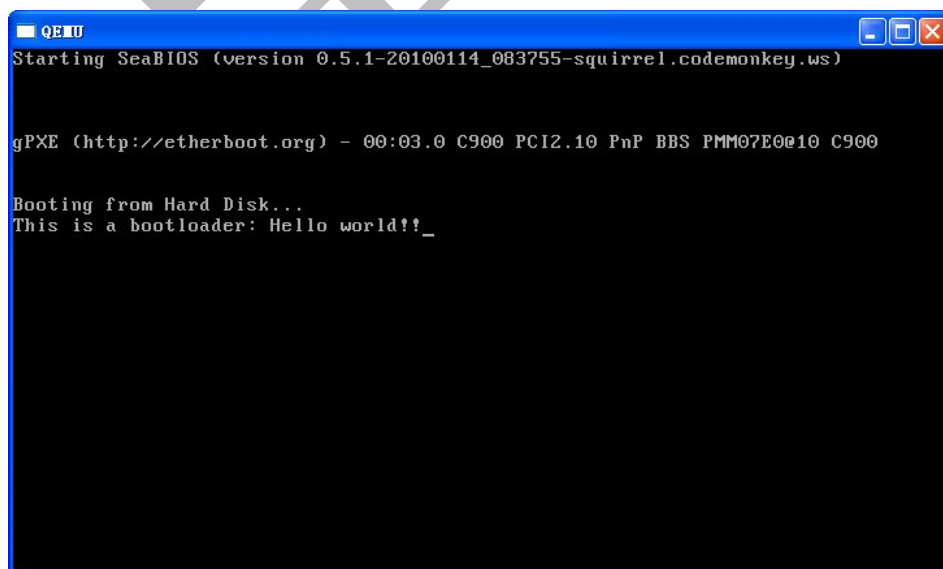
【提示】**bootloader** 是一个超小的系统软件, 在功能上与我们一般的应用软件不同, 主要用于硬件简单初始化和加载运行操作系统。在编写 **bootloader** 的时候, 需要了解它所处的硬件环境 (比如它在内存中的起始地址, 它的储存空间的位置和大小限制等)。而这些都是编写应用软件不太需要了解的, 因为操作系统和编译器帮助应用软件考虑了这些问题。

### 3. 编译运行

#### 【实验 编译运行 **bootloader**】

在 **proj1** 下执行 **make**, 在 **proj1/bin** 目录下可生成一个 **ucore.img**。**ucore.img** 是一个包含了 **bootloader** 或 **OS** 的硬盘镜像, 通过执行如下命令可在硬件虚拟环境 **qemu** 中运行 **bootloader** 或 **OS**:

```
make                //生成 bootloader 和对应的主引导扇区
make qemu           //通过 qemu 硬件模拟器来运行 bootloader
```



```
make clean          //清除生成的临时文件, bootloader 和对应的主引导扇区
```

---

我们除了需要了解 bootloader 的功能外，还需要进一步了解 bootloader 的编译链接和最终执行码的生成过程，从而能够清楚生成的代码是否是我们所期望的。proj1 中的 Makefile 是一个配置脚本，make 软件工具能够通过 Makefile 完成管理 bootloader 的 C/ASM 代码生成执行码的整个过程。Makefile 的内容比较复杂，不过读者在前期只需会执行 make [参数]来生成代码和清除代码即可。对于本实验的 make 的执行过程如下所示：

---

1. gcc -O2 -o tools/sign tools/sign.c
  2. i386-elf-gcc -fno-builtin -Wall -MD -ggdb -m32 -fno-stack-protector -O -nostdinc -Iinclude -Iinclude/x86 -c bootloader/bootmain.c -o obj/bootmain.o
  3. i386-elf-gcc -fno-builtin -Wall -MD -ggdb -m32 -fno-stack-protector -nostdinc -Iinclude -Iinclude/x86 -c bootloader/bootasm.S -o obj/bootasm.o
  4. i386-elf-ld -N -e start -Ttext 0x7C00 -o obj/bootblock.o obj/bootasm.o obj/bootmain.o
  5. i386-elf-objdump -S obj/bootblock.o > obj/bootblock.asm
  6. i386-elf-objcopy -S -O binary obj/bootblock.o obj/bootblock.out
  7. sign.exe obj/bootblock.out obj/bootblock  
obj/bootblock.out size: 380 bytes  
build 512 bytes boot sector: obj/bootblock success!
  8. dd if=/dev/zero of=obj/ucore.img count=10000  
10000+0 records in  
10000+0 records out  
5120000 bytes (5.1 MB) copied, 0.509 s, 10.1 MB/s
  9. dd if=obj/bootblock of=obj/ucore.img conv=notrunc  
1+0 records in  
1+0 records out  
512 bytes (512 B) copied, 0.011 s, 46.5 kB/s
- 

这 9 步的含义是：

1. 编译生成 sign 执行程序，用于生成一个符合规范的硬盘主引导扇区；
2. 用 gcc 编译器编译 bootmain.c，生成 ELF 格式的目标文件 bootmain.o；
3. 用 gas 汇编器(gcc 只是一个包装)编译 bootasm.S,生成 ELF 格式的目标文件 bootasm.o；
4. 用 ld 链接器把 bootmain.o 和 bootasm.o 链接在一起，形成生成 ELF 格式的执行文件 bootblock.o；
5. 目标文件信息导出工具 objdump 反汇编 bootblock.o，生成 bootblock.asm，通过查看 bootblock.asm 内容，可以了解 bootloader 的实际执行代码；

6. 文件格式转换和拷贝工具 `objcopy` 把 ELF 格式的执行文件 `bootblock.o` 转换成 binary 格式的执行文件 `bootblock.out`;
7. 通过 `sign` 执行程序, 把 `bootblock.out` (本身大小需要小于 510 字节) 扩展到 512 字节, 形成一个符合规范的硬盘主引导扇区 `bootblock`;
8. 设备级转换与拷贝工具 `dd` 生成一个内容都为“0”的磁盘文件 `ucore.img`;
9. 设备级转换与拷贝工具 `dd` 进一步把 `bootblock` 覆盖到 `ucore.img` 的前 512 个字节空间中, 这样就可以把 `ucore.img` 作为一个可启动的硬盘被硬件模拟器 `qemu` 使用。

如果需要了解 Makefile 中的内容, 需要进一步看看如下补充材料。

### 3.1.3 【背景】Intel 80386 加电后启动过程

【要点 (非 OSP): 80836 物理内存地址空间】

【要点 (非 OSP): 80836 加电后的第一条指令位】

大家一般都知道 `bootloader` 负责启动操作系统, 但 `bootloader` 自身是被谁加载并启动的呢? 为了追根溯源, 我们需要了解当计算机加电启动后, 到底发生了什么事情。

对于绝大多数计算机系统而言, 操作系统和应用软件是存放在磁盘 (硬盘/软盘)、光盘、EPROM、ROM、Flash 等可在掉电后继续保存数据的存储介质上。当计算机加电后, 一般不直接执行操作系统, 而是一开始会到一个特定的地址开始执行指令, 这个特定的地址存放了系统初始化软件, 通过执行系统初始化软件 (可固化在 ROM 或 Flash 中, 也称 `firmware`, 固件) 完成基本 I/O 初始化和引导加载操作系统的功能。简单地说, 系统初始化软件就是在操作系统内核运行之前运行的一段小软件。通过这段小软件的基本 I/O 初始化部分, 我们可以初始化硬件设备、建立系统的内存空间映射图, 从而将系统的软硬件环境带到一个合适的状态, 以便为最终调用操作系统内核准备好正确的环境。最终系统初始化软件的引导加载部分把操作系统内核映像加载到 RAM 中, 并将系统控制权传递给它。

对于基于 Intel 80386 的计算机而言, 其中的系统初始化软件由 BIOS (Basic Input Output System, 即基本输入/输出系统, 其本质是一个固化在主板 Flash/CMOS 上的软件) 和位于软盘/硬盘引导扇区中的 OS Boot Loader (在 `ucore` 中的 `bootasm.S` 和 `bootmain.c`) 一起组成。BIOS 实际上是被固化在计算机 ROM (只读存储器) 芯片上的一个特殊的软件, 为上层软件提供最底层的、最直接的硬件控制与支持。

以基于 Intel 80386 的计算机为例, 计算机加电后, 整个物理地址空间如下图所示:

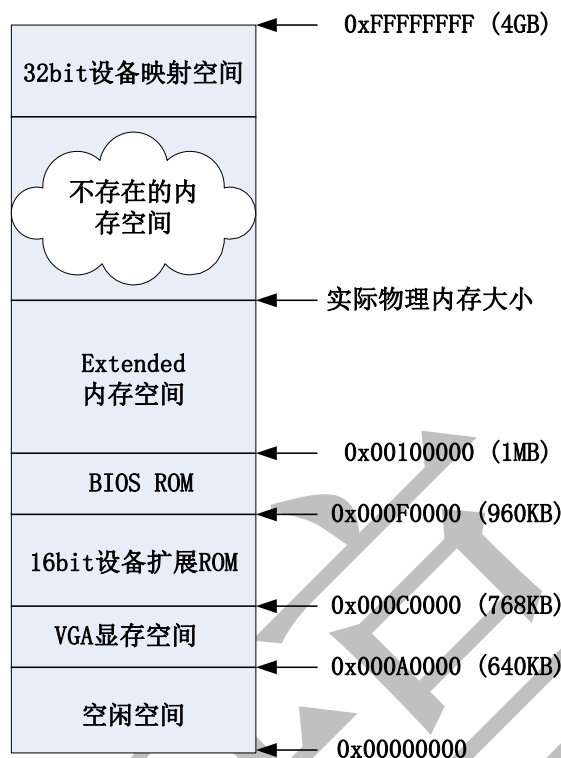
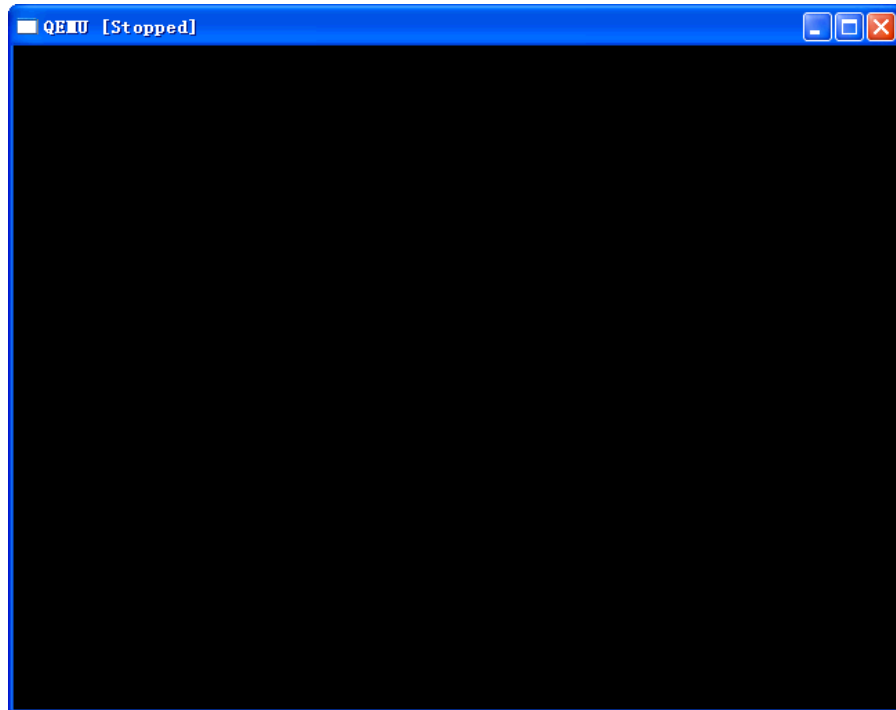


图 2-1 基于 Intel 80386 的计算机物理地址空间

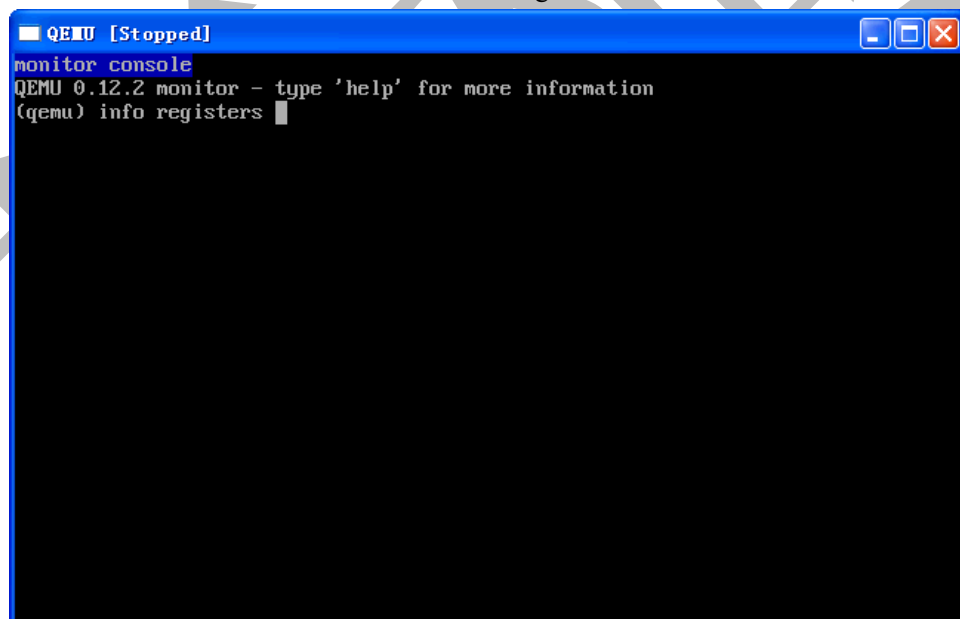
处理器处于实模式状态（在 86386 中，段机制一直存在，可进一步参考 2.1.5 【背景】理解保护模式和分段机制），从物理地址 0xFFFFFFF0 开始执行。初始化状态的 CS 和 EIP 确定了处理器的初始执行地址，此时 CS 中可见部分-选择子（selector）的值为 0xF000，而其不可见部分-基地址（base）的值为 0xFFFF0000；EIP 的值是 0xFFF0，这样实际的线性地址（由于没有启动也机制，所以线性地址就是物理地址）为 CS.base+EIP=0xFFFFFFF0。在 0xFFFFFFF0 这里只是存放了一条跳转指令，通过跳转指令跳到 BIOS 例行程序起始点。更详细的解释可以参考文献[1]的第九章的 9.1 节“INITIALIZATION OVERVIEW”。另外，我们可以通过硬件模拟器 qemu 来进一步认识上述结果。

实验 2-1：通过 qemu 了解 Intel 80386 启动后的 CS 和 EIP 值，并分析第一条指令的内容

1. 启动 qemu 并让其停到执行第一条指令前，这需要增加一个参数“-S”  
qemu -S
2. 这是 qemu 会弹出一个没有任何显示内容的图形窗口，显示如下：



3. 然后通过按”Ctrl+Alt+2”进入 qemu 的 monitor 界面，为了了解 80386 此时的寄存器内容，在 monitor 界面下输入命令 “info registers”



4. 可获得 intel 80386 启动后执行第一条指令前的寄存器内容，如下图所示

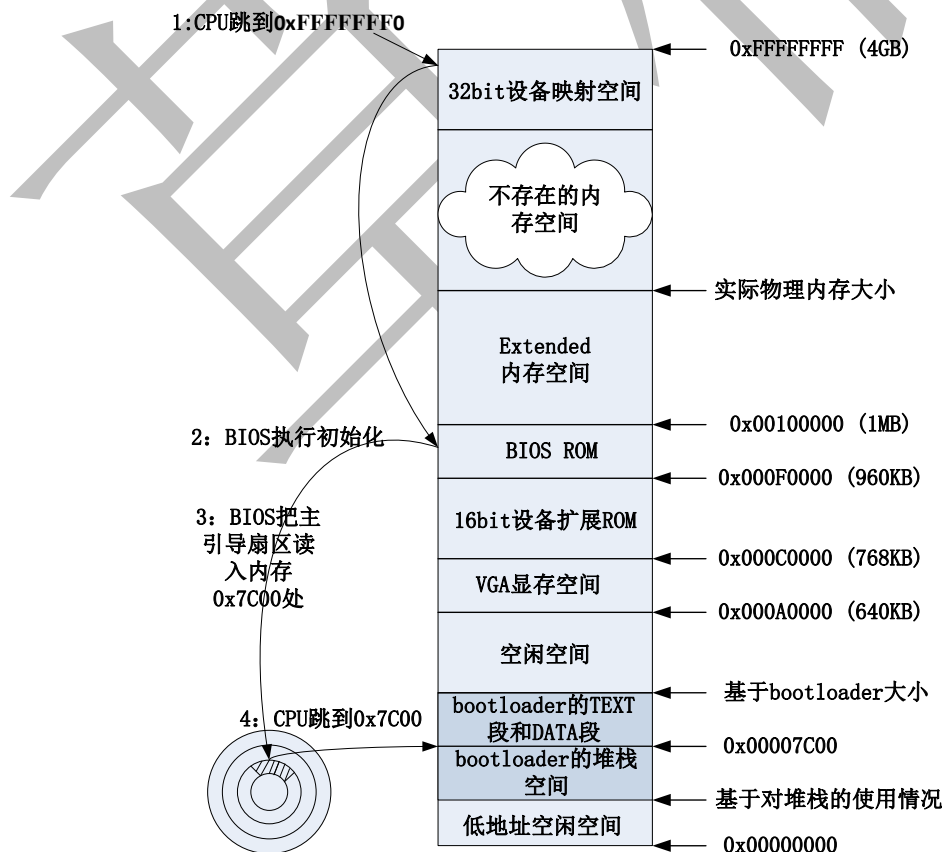
```

QEMU [Stopped]
EIP=0000ffff EFL=00000002 [-----] CPL=0  II=0  A20=1  SMM=0  HLT=0
ES =0000 00000000 0000ffff 00009300
CS =f000 ffff0000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT= 00000000 0000ffff
IDT= 00000000 0000ffff
CR0=60000010 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
FCW=037f FSW=0000 IST=01 FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu)

```

从上图，我们可以看到 EIP=0xffff0, CS 的 selector=0xf000, CS 的 base=0xffff0000。

BIOS 做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区(即主引导扇区或启动扇区)到内存一个特定的地址 0x7c00 处，然后 CPU 控制权会转移到那个地址继续执行。至此 BIOS 的初始化工作做完了，进一步的工作交给了 ucore 的 bootloader; ucore 的 bootloader 会完成处理器从实模式到保护模式的转换，并从硬盘上读取并加载 ucore。其大致流程如下图所示：



### 3.1.4 【背景】设备管理：理解设备访问机制

在本章涉及的 `bootloader` 和 `ucore` 都需要对 I/O 设备进行访问，比如通过串口、并口和 CGA 显示器显示字符串，读取硬盘数据，处理时钟中断等，已经需要读者用到操作系统的 I/O 设备管理知识了。为此，我们需要操作系统的设备管理进行一个简要描述。

在计算机系统中，操作系统需要管理各种设备，即给它们发送控制命令、捕获中断、错误处理等；为此专门设置了一个子系统：设备管理子系统来完成这些琐碎的工作。同时设备管理子系统还需要提供一个简单易用的统一接口，并尽可能地使其他内核功能组件或应用可通过这个统一接口访问所有的设备，即实现与设备的无关性。比如在 `proj1` 中，`bootloader` 提供了一个显示字符的函数接口 `cons_putc`（位于 `bootmain.c` 中），在 `proj3` 中的提供了一个显示格式化信息的函数接口 `cprintf`（位于 `printf.c` 中），这样操作系统的其他功能组件就可以直接使用这些简单易用的接口来输出信息，而不是通过繁琐的 I/O 命令与具体的设备打交道。`cprintf` 的实现相对复杂，用到 C 语言的可变列表参数等，大家只要把它的功能理解为 C 语言应用库中的 `printf` 的简化版即可，并掌握最后是如何通过调用 `cons_putc` 函数完成具体的 I/O 字符输出。

接下来，我们将从操作系统概念的角度对 I/O 设备组成、控制设备的方式进行阐述，并进一步对实验中所使用的基于 Programmed I/O（PIO）方式访问并口、CGA 和硬盘进行具体分析。

## 1. 硬件设备简介

对于硬件设备而言，操作系统所关心的并不是硬件自身的设计，而是如何来对它进行控制，即该硬件所接受的控制命令、所完成的功能，以及所返回的出错。所以在设计操作系统的设备管理子系统时，需要了解计算机系统中 I/O 总线上连接的 I/O 控制器（比如 PC 机中的 CGA 控制器、串口控制器、并口控制器、时钟控制器 8254，中断控制器 8259 等）。

I/O 控制器在物理上包含三个层次：I/O 地址空间、I/O 接口和设备控制器。每个连接到 I/O 总线上的设备都有自己的 I/O 地址空间（即 I/O 端口），这也是 CPU 可以直接访问的地址。在 PC 机中，支持基于 I/O 的 I/O 地址空间（通过 `IN/OUT` 这类的 I/O 访问指令访问），也支持基于内存的 I/O 地址空间（通过 `MOV` 等访存指令访问）。这些 I/O 访问请求通过 I/O 总线传递给 I/O 接口。

I/O 接口是处于一组 I/O 端口和对应的设备控制器之间的一种硬件电路。它将 I/O 访问请求中的特定值转换成设备所需要的命令和数据；并且检测设备的状态变化，及时将各种状态信息写回到特定 I/O 地址空间，供操作系统通过 I/O 访问指令来访问。I/O 接口包括键盘接口、图形接口、磁盘接口、总线鼠标、网络接口、括并口、串口、通用串行总线、PCMCIA 接口和 SCSI 接口等。

设备控制器并不是所有 I/O 设备所必须的，只有少数复杂的设备才需要。它负责解释从 I/O 接口接收到的高级命令，并将其以适当的方式发送到 I/O 设备；并且对 I/O 设备发送的消息进行解释并修改 I/O 端口的状态寄存器。典型的设备控制器就是磁盘控制器，它将 CPU 发送过来的读写数据指令转换成底层的磁盘操作。

---

## 2. 控制设备的方式

操作系统对硬件设备的控制方式主要有三种：程序循环检测方式(Programmed I/O, 简称 PIO)、中断驱动方式(Interrupt-driven I/O)、直接内存访问方式(DMA, Direct Memory Access)。

在本章的 proj1 实验中, bootloader 需要显示字符串, 就是采用相对简单的 PIO 方式。PIO 方式是一种通过 CPU 执行 I/O 端口指令来进行数据读写的数据交换模式, 被广泛应用于硬盘、光驱等设备的基础传输模式中。这种 I/O 访问方式使用 CPU I/O 端口指令来传送所有的命令、状态和数据, 需要处理器全程参与, 效率较低, 但编程很简单。后面讲到的中断方式和直接内存访问 (Direct Memory Access, DMA) 方式将更加高效。

对于程序循环检测方式而言, 其控制方式体现在执行过程中通过不断地检测 I/O 设备的当前状态, 来控制 I/O 操作。具体而言, 在进行 I/O 操作之前, 要循环地检测设备是否就绪; 在 I/O 操作进行之中, 要循环地检测设备是否已完成; 在 I/O 操作完成之后, 还要把输入的数据保存到内存 (输入操作)。从硬件的角度来说, 控制 I/O 的所有工作均由 CPU 来完成。所以此方式也称为繁忙等待方式 (busy waiting) 或轮询方式 (polling)。其缺点是在进行 I/O 操作时, 一直占用 CPU 时间。

中断驱动方式的基本思路是用户任务通过系统调用函数来发起 I/O 操作。执行系统调用后会阻塞该任务, 调度其他的任务使用 CPU。在 I/O 操作完成时, 设备向 CPU 发出中断, 然后在中断服务例程中做进一步的处理。在中断驱动方式下, 数据的每次读写还是通过 CPU 来完成。但是当 I/O 设备在进行数据处理时, CPU 不必等待, 可以继续执行其他的任务。采用这种方式可提供 CPU 利用率。编程方面, 要考虑异步特性, 相对麻烦一些。

使用 DMA 的控制方式, 首先需要有 DMA 控制器。该控制器可集成在设备控制器中, 也可集成在主板上。DMA 控制器可以直接去访问系统总线, 它能代替 CPU 指挥 I/O 设备与内存之间的数据传送, 在执行完毕后再通知 CPU。这种方式可大大减少 CPU 的执行开销, 适合大数据量的设备数据传送。在编程方面, 需要对 DMA 进行编程和异步中断编程, 相对更加复杂一些。

## 3. 串口 (serial port) 访问控制

串口是一个字符设备, proj1 通过串口输出需要显示的信息。考虑到简单性, 在 proj1 中没有对串口设备进行初始化, 通过串口进行输出的过程也很简单: 第一步: 执行 inb 指令读取串口的 I/O 地址 (COM1 + COM\_LSR) 的值, 如果发现读出的值代表串口忙, 则空转一小会 (0x84 是什么地址???) ; 如果发现读出的值代表串口空闲, 则执行 outb 指令把字符写到串口的 I/O 地址 (COM1 + COM\_TX), 这样就完成了一个字符的串口输出。在 proj1 的 bootmain.c 中的 serial\_putc 函数完成了串口输出字符的工作, 可参看其函数来了解大致实现。有关串口的硬件细节可参考附录 补充材料。

## 4. 并口 (parallel port) 访问控制

并口也是一个字符设备, proj1 也通过并口输出需要显示的信息。考虑到简单性, 在 proj1 中没有对并口设备进行初始化, 通过并口进行输出的过程也很简单: 第一步: 执行 inb 指令



读取并口的 I/O 地址 (LPTPORT + 1) 的值, 如果发现读出的值代表并口忙, 则空转一小会再读; 如果发现读出的值代表并口空闲, 则执行 `outb` 指令把字符写到并口的 I/O 地址 (LPTPORT), 这样就完成了一个字符的并口输出。在 `proj1` 的 `bootmain.c` 中的 `lpt_putc` 函数完成了并口输出字符的工作, 可参看其函数来了解大致实现。有关并口的硬件细节可参考附录 补充材料。

## 5. CGA 字符显示控制

彩色图形适配器 (Color Graphics Adapter, CGA) 支持 7 种彩色和文本/图形显示方式, `proj1` 也通过 CGA 进行信息显示。在 80 列×25 行的文本字符显示方式下, 有单色和 16 色两种显示方式。CGA 显示控制器标配 16KB 显示内存 (占用内存地址范围 0xb8000~0xbc000), 可以看成是一种内存块设备, 即 `bootloader` 和操作系统可以直接对显存进行内存访问, 从而完成信息显示。在 CGA 显示控制器中, 字符显示内存从线性地址 0x000B8000 开始, 在 80 列×25 行的范围内, 共 2000 字符。每个字符需要两个字节来显示: 第一个字节是想要显示的字符, 第二个字节用来确定前景色和背景色。前景色用低 4 位 (0~3 位) 来表示, 背景色用第 4 位到第 6 位来表示。最高位表示这个字符是否闪烁, 1 表示闪烁, 0 表示不闪烁。

如果要在屏幕上设置光标, 则它须通过 CGA 显示控制器的 I/O 端口开控制。显示控制索引寄存器的 I/O 端口地址为 0x3d4; 数据寄存器 I/O 端口地址为 0x3d5。CGA 显示控制器内部有一系列寄存器可以用来访问其状态。0x3d4 和 0x3d5 两个端口可以用来读写 CGA 显示控制器的内部寄存器。方法是先向 0x3d4 端口写入要访问的寄存器编号, 再通过 0x3d5 端口来读写寄存器数据。存放光标位置的寄存器编号为 14 和 15。两个寄存器合起来组成一个 16 位整数, 这个整数就是光标的位置。比如 0 表示光标在第 0 行第 0 列, 81 表示第 1 行第 1 列 (设屏幕共有 80 列)。

在 `proj1` 中没有对 CGA 显示控制器进行初始化, 通过 CGA 显示控制器进行输出的过程也很简单: 首先通过 `in/out` 指令获取当前光标位置; 然后根据得到的位置计算出显存的地址, 直接通过访存指令写内存来完成字符的输出; 最后通过 `in/out` 指令更新当前光标位置。在 `proj1` 的 `bootmain.c` 中的 `cga_putc` 函数完成了 CGA 字符方式在某位置输出字符的工作, 可参看其函数了解大致实现。

## 6. 设备管理封装

`proj1` 把上述三种设备进行了一个封装, 提供了一个 `cons_puts` 函数接口: 完成字符串的输出; 和一个 `cons_putc` 函数接口, 完成字符的输出。其他内核功能模块只需调用 `cons_puts` 或 `cons_putc` 就可完成向上述三个设备进行字符输出的功能。这也就体现了设备管理子系统提供一个简单易用的统一接口的操作系统设计思想。

### 3.1.5 【背景】内存管理: 理解保护模式和分段机制

为何要了解 Intel 80386 的保护模式和分段机制? 首先, 我们知道 Intel 80386 只有在进入保护模式后, 才能充分发挥其强大的功能, 提供更好的保护机制和更大的寻址空间, 否则仅仅是一个快速的 8086 而已。没有一定的保护机制, 任何一个应用软件都可以任意访问所有的计算机资源, 这样也就无从谈起操作系统设计了。且 Intel 80386 的分段机制一直存在,

无法屏蔽或避免。其次,在我们的 bootloader 设计中,涉及到了从实模式到保护模式的处理,我们的操作系统功能(比如分页机制)是建立在 Intel 80386 的保护模式上来设计的。如果我们不了解保护模式和分段机制,则我们面向 Intel 80386 体系结构的操作系统设计实际上是建立在一个空中楼阁之上。

## 1. 实模式

80386 的实模式是为了与 8086 处理器兼容而设置的。在实模式下,80386 处理器就相当于一个快速的 8086 处理器。80386 处理器被复位或加电的时候以实模式启动。这时候处理器中的各寄存器以实模式的初始化值工作。80386 处理器在实模式下的存储器寻址方式和 8086 基本一致,由段寄存器的内容乘以 16 作为基地址,加上段内的偏移地址形成最终的物理地址,这时候它的 32 位地址线只使用了低 20 位,即可访问 1MB 的物理地址空间。在实模式下,80386 处理器不能对内存进行分页机制的管理,所以指令寻址的地址就是内存中实际的物理地址。在实模式下,所有的段都是可以读、写和执行的。实模式下 80386 不支持优先级,所有的指令相当于工作在特权级(即优先级 0),所以它可以执行所有特权指令,包括读写控制寄存器 CR0 等。这实际上使得在实模式下不太可能设计一个有保护能力的操作系统。实模式下不支持硬件上的多任务切换。实模式下的中断处理方式和 8086 处理器相同,也用中断向量表来定位中断服务程序地址。中断向量表的结构也和 8086 处理器一样,每 4 个字节组成一个中断向量,其中包括两个字节的段地址和两个字节的偏移地址。应用程序可以任意修改中断向量表的内容,使得计算机系统容易受到病毒、木马等的攻击,整个计算机系统的安全性无法得到保证。

---

### 【历史: 寻址空间: A20 地址线与处理器向下兼容】

Intel 早期的 8086 CPU 提供了 20 根地址线,可寻址空间范围即  $0 \sim 2^{20}(00000H \sim FFFFFH)$  的 1MB 内存空间。但 8086 的数据处理位为 16 位,无法直接寻址 1MB 内存空间,所以 8086 提供了段地址加偏移地址的地址转换机制,就是我们常见的“段地址(16 位):偏移地址(16 位或有效地址)”,实际的计算方法为:“段地址 $\times 0x10H$ +偏移地址”,作为段地址的数据是放在段寄存器中的(16 位),而作为位偏移地址的数据则是通过 8086 提供的寻址方式来计算而来的(16 位)。而“段值: 偏移”这种表示法能够表示的最大内存为  $0x10FFEEH$ (即  $0xFFFF0H + 0xFFFFH$ ),所以当寻址到超过 1MB 的内存时,会发生“回卷”(不会发生异常)。但下一代的基于 Intel 80286 CPU 的 PC AT 计算机系统提供了 24 根地址线,这样 CPU 的寻址范围变为  $2^{24}=16M$ ,同时也提供了保护模式,可以访问到 1MB 以上的内存了,此时如果遇到“寻址超过 1MB”的情况,系统不会再“回卷”了,这就造成了向下不兼容。为了保持完全的向下兼容性,IBM 决定在 PC AT 计算机系统上加个硬件逻辑,来模仿以上的回绕特征。他们的方法就是把 A20 地址线控制和键盘控制器的一个输出进行 AND 操作,这样来控制 A20 地址线的打开(使能)和关闭(屏蔽\禁止)。一开始时 A20 地址线控制是被屏蔽的(总为 0),直到系统软件通过一定的 I/O 操作去打开它(参看 bootloder 的 bootasm.S 文件)。

当 A20 地址线控制禁止时,则程序就像在 8086 中运行,1MB 以上的地是不可访问的。在保护模式下 A20 地址线控制是要打开的。为了使能所有地址位的寻址能力,必须向键盘控制器 8042 发送一个命令。键盘控制器 8042 将会将它的的某个输出引脚的输出置高电平,作为 A20 地址线控制的输入。一旦设置成功之后,内存将不会再被绕回(memory wrapping),这样我们就可以寻址 intel 80286 CPU 支持的 16M 内存空间,或者是寻址 intel 80386 以上级

---

别 CPU 支持的所有 4G 内存空间了。

8042 键盘控制器的 I/O 端口是 0x60~0x6f, 实际上 IBM PC/AT 使用的只有 0x60 和 0x64 两个端口 (0x61、0x62 和 0x63 用于与 XT 兼容目的)。8042 通过这些端口给键盘控制器或键盘发送命令或读取状态。输出端口 P2 用于特定目的。位 0 (P20 引脚) 用于实现 CPU 复位操作, 位 1 (P21 引脚) 用户控制 A20 信号线的开启与否。系统向输入缓冲 (端口 0x64) 写入一个字节, 即发送一个键盘控制器命令。可以带一个参数。参数是通过 0x60 端口发送的。命令的返回值也从端口 0x60 去读。

在 proj1 的 bootasm.S 中, “seta20.1”标号和“seta20.2”标号后的汇编代码即是用来完成 A20 地址线控制打开工作的。

---

## 2. 保护模式概述

简单地说, 通过保护模式, 可以把虚拟地址空间映射到不同的物理地址空间, 且在超出预设的空间范围会报错 (一种保护机制的体现), 且可以保证处于低特权级的代码无法访问搞特权级的数据 (另外一种保护机制的体现)。

只有在保护模式下, 80386 的全部 32 位地址才能有效, 可寻址高达 4G 字节的线性地址空间和物理地址空间, 可访问 64TB (有  $2^{14}$  个段, 每个段最大空间为  $2^{32}$  字节) 的虚拟地址空间, 可采用分段存储管理机制和分页存储管理机制。这不仅为存储共享和保护提供了硬件支持, 而且为实现虚拟存储提供了硬件支持。通过提供 4 个特权级和完善的特权检查机制, 既能实现资源共享又能保证代码数据的安全及任务隔离。

在保护模式下, 特权级总共有 4 个, 编号从 0 (最高特权) 到 3 (最低特权)。有 3 种主要的资源受到保护: 内存, I/O 地址空间以及执行特殊机器指令的能力。在任一时刻, intel 80386 CPU 都是在一个特定的特权级下运行的, 从而决定了代码可以做什么, 不可以做什么。这些特权级经常被称为保护环 (protection ring), 最内的环 (ring 0) 对应于最高特权 0, 最外面的环 (ring 3) 一般给应用程序使用, 对应最低特权 3。在 ucore 中, CPU 只用到其中的 2 个特权级: 0 (内核态) 和 3 (用户态)。在保护模式下, 我们可以通过查看 CS 寄存器的最低两位来了解当前正在运行的处理器是处于哪个特权级。

## 3. 分段机制的地址转换

intel 80386 CPU 提供了分段机制和分页机制两种内存管理方式, 在当前计算机系统中是否需要这两种机制共存没有一个明确的答案, 二者有它们各自独特的功能。在 intel 80386 CPU 中, 只要进入保护模式, 必然需要启动分段机制, 且一直存在下去 (分页不一定要一直存在), 所以我们需要了解分段机制的原理。分段机制体现了内存中不同地址的一种转换/映射方式, 即程序员编程所使用的地址 (逻辑地址) 和实际计算机中的物理地址需要通过分段机制来建立映射关系。分段机制将内存划分成以起始地址和长度限制这两个参数表示的内存块, 这些内存块就称之为段 (Segment)。编译器把源程序编译成执行程序时用到的代码段、数据段、堆和栈等概念在这里可以与段联系起来, 二者在含义上是一致的。

从操作系统原理上看, 编译器实际上采用了基于分段的虚存管理方式来生成执行程序的, 即应用程序员看到的逻辑地址和位于计算机上的物理地址之间有映射关系, 二者可以是不同

的。当然，后续章节中，我们还将介绍分页机制，即另一种使用更加广泛的地址转换/映射方式，这是操作系统实现虚存管理的重要基础。

简单地说，当 CPU 执行一条访存指令时（一个具体的指令），基于分段模式的具体硬件操作过程如下：

1. 根据指令的内容确定应该使用的段寄存器，比如取内存指令的内存地址所对应的数据段寄存器为 DS；
2. 根据段寄存器 DS 的值作为选择子，以此选择子值为索引，在段描述符表（可理解为一个大数据组）找到索引指向的段描述符（可理解为数组中的元素）；
3. 在段描述符中取出基地址域（段的起始地址）和地址范围域（段的长度）的值；
4. 将指令内容确定的地址偏移，与地址范围域的值比较，确保地址偏移小于地址范围，这样是为了确保地址范围不会跨出段的范围；（第一层保护）
5. 根据指令的性质（当前指令的 CS 值的低两位）确定当前指令的特权级，需要高于当前指令访问的数据段的特权级；（第二层保护）；
6. 根据指令的性质（指令是做读还是写操作），需要当前指令访问的数据段可读或可写；（第三层保护）
7. 将 DS 指向的段描述符中基地址域的值加上指令内容中指定的访存地址段内偏移值，形成实际的物理地址（实现地址转换），发到数据地址总线上，到物理内存中寻址，并取回该地址对应的数据内容。

分段机制涉及 4 个关键内容：逻辑地址（Logical Address, 应用程序员看到的地址，在操作系统原理上称为虚拟地址，以后提到虚拟地址就是指逻辑地址）、物理地址（Physical Address, 实际的物理内存地址）、段描述符表（包含多个段描述符的“数组”）、段描述符（描述段的属性，及段描述符表这个“数组”中的“数组元素”）、段选择子（即段寄存器中的值，用于定位段描述符表中段描述符表项的索引）。

虚拟地址到物理地址的转换主要分以下两步：

1. 分段地址转换：CPU 把虚拟地址（由段选择子 selector 和段偏移 offset 组成）中的段选择子值作为段描述符表的索引，找到表中对应的段描述符，然后把段描述符中保存的段基址加上段偏移值，形成线性地址（Linear Address，在操作系统原理上没有直接对应的描述，在没有启动分页机制的情况下，可认为就是物理地址；如果启动了分页机制，则可理解为第二级虚拟地址）。如果不启动分页存储管理机制，则线性地址等于物理地址。
2. 分页地址转换，这一步中把线性地址转换为物理地址。（注意：这一步是可选的，由操作系统决定是否需要。在后续试验中会涉及。）

上述转换过程对于应用程序员来说是不可见的。**线性地址空间**由一维的线性地址构成，在分段机制下的线性地址空间和物理地址空间对等。线性地址 32 位长，线性地址空间容量为 4G 字节。分段机制中虚拟地址到线性地址转换转换的基本过程如下图所示。

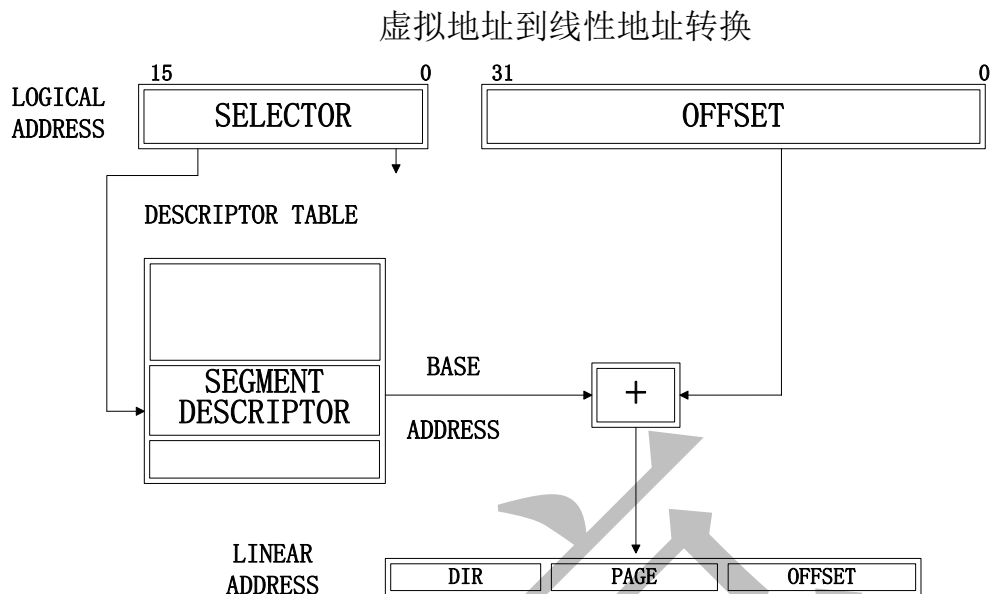


图 1 分段机制中虚拟地址到线性地址转换基本过程

分段存储管理机制需要在启动保护模式的前提下建立。从上图可以看出，为了使得分段存储管理机制正常运行，需要在启动保护模式前建立好段描述符和段描述符表（参看 `bootasm.S` 中的“`lgdt gdt_desc`”语句和 `gdt` 标号/`gdt_desc` 标号下的数据结构）。

### 段选择子

**段选择子**是用来选择哪个描述符表和在该表中索引哪一个描述符的。选择子可以做为指针变量的一部分，从而对应用程序员是可见的，但是一般是由编译器（`gcc`）和链接工具（`ld`）来设置的。段选择子的内容一般放在段寄存器中。选择子的格式如下图所示：



TI - 0：全局描述符表；1：局部描述符表

RPL - 请求特权级

图 2 段选择子结构

- **索引（Index）**：在描述符表中从 8192 个描述符中选择一个描述符。处理器自动将这个索引值乘以 8（描述符的长度），再加上描述符表的基址来索引描述符表，从而选出一个合适的描述符。
- **表指示位（Table Indicator, TI）**：选择应该访问哪一个描述符表。0 代表应该访问全局描述符表（GDT）；1 代表应该访问局部描述符表（LDT）。LDT 在实验中没有涉及。
- **请求特权级（Requested Privilege Level, RPL）**：用于段级的保护机制，比如，段选择子是 CS，则这两位表示当前执行指令的处理器所处的特权级的值，从而你可以了解到当前处理器是处于用户态（Ring 3）还是内核态（Ring 0）。在后续试验中会进一步讲解。

### 段描述符

在分段存储管理机制的保护模式下，每个段由如下三个参数进行定义：段基地址(Base Address)、段界限(Limit)和段属性(Attributes)。

- 段基地址：即线性地址空间中段的起始地址。在 80386 保护模式下，段基地址长 32 位。因为基地址长度与寻址地址的长度相同，所以任何一个段都可以从 32 位线性地址空间中的任何一个字节开始，而不象实方式下规定的边界必须被 16 整除。在实验中，一般都简化了段机制的使用，把所有段的段基地址设置为 0。
- 段界限：规定段的大小。在 80386 保护模式下，段界限用 20 位表示，而且段界限可以是以单字节为最小单位或以 4K 字节为最小单位。在实验中，一般都简化了段机制的使用，把所有段的段界限设置为 0xFFFFF，以 4K 字节为最小单位，即段的界限为 4GB；
- 段属性：确定段的各种性质，如段所对应的内存空间是否可读/写/执行等。
  - ◆ 段属性中的粒度位 (Granularity)，用符号 G 标记。G=0 表示段界限以单字节为最小单位，20 位的界限可表示的范围是 1 字节至 1M 字节，增量为 1 字节；G=1 表示段界限以 4K 字节为最小单位，于是 20 位的界限可表示的范围是 4K 字节至 4G 字节，增量为 4K 字节。在实验中，一般都简化了段机制的使用，把 G 设置为 1。
  - ◆ 类型 (TYPE)：用于区别不同类型的描述符。可表示所描述的段是代码段还是数据段，所描述的段是否可读/写/执行，段的扩展方向等。
  - ◆ 描述符特权级 (Descriptor Privilege Level) (DPL)：用来实现保护机制。
  - ◆ 段存在位 (Segment-Present bit)：如果这一位为 0，则此描述符为非法的，不能被用来实现地址转换。如果一个非法描述符被加载进一个段寄存器，处理器会立即产生异常。图 2 显示了当存在位为 0 时，描述符的格式。操作系统可以任意的使用被标识为可用 (AVAILABLE) 的位。
  - ◆ 已访问位 (Accessed bit)：当处理器访问该段（当一个指向该段描述符的选择子被加载进一个段寄存器）时，将自动设置访问位。操作系统可清除该位。

上述表示段的属性的参数通过段描述符(Segment Descriptor)来表示，一个段描述符占 8 字节。段描述符的结构如下图所示：

The diagram illustrates the structure of the 32-bit Segment Register. It is divided into two main sections: the upper 16 bits (bits 31-16) and the lower 16 bits (bits 15-0). The upper section contains the following fields:

- BASE 31..24**: 8 bits (bits 31-24)
- G**: 1 bit (bit 23)
- X**: 1 bit (bit 22)
- O**: 1 bit (bit 21)
- AVL**: 3 bits (bits 20-19)
- LIMIT 19..16**: 4 bits (bits 19-16)
- P**: 1 bit (bit 15)
- DPL**: 1 bit (bit 14)
- 1**: 1 bit (bit 13)
- TYPE**: 2 bits (bits 12-11)
- A**: 1 bit (bit 10)
- BASE 23..16**: 8 bits (bits 23-16)

The lower section contains the following fields:

- SEGMENT BASE 15..0**: 16 bits (bits 15-0)
- SEGMENT LIMIT 15..0**: 16 bits (bits 15-0)

|                    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |                     |   |   |             |                 |  |  |  |   |     |  |   |      |  |             |  |   |  |   |
|--------------------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|---------------------|---|---|-------------|-----------------|--|--|--|---|-----|--|---|------|--|-------------|--|---|--|---|
| 31                 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 23                  |   |   |             | 15              |  |  |  | 7 |     |  |   | 0    |  |             |  |   |  |   |
| BASE 31..24        |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | G                   | X | O | A<br>V<br>L | LIMIT<br>19..16 |  |  |  | P | DPL |  | 0 | TYPE |  | BASE 23..16 |  |   |  | 4 |
| SEGMENT BASE 15..0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | SEGMENT LIMIT 15..0 |   |   |             |                 |  |  |  |   |     |  |   |      |  |             |  | 0 |  |   |

- A - ACCESSED
- AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
- DPL - DESCRIPTOR PRIVILEGE LEVEL
- G - GRANULARITY
- P - SEGMENT PRESENT

### 全局描述符表

全局描述符表的第一项是不能被 CPU 使用，所以当一段选择子的索引（Index）部分和表指示位（Table Indicator）都为 0 的时（即段选择子指向全局描述符表的第一项时），可以当做一个空的选择子。当一个段寄存器被加载一个空选择子时，处理器并不会产生一个异常。但是，当用一个空选择子去访问内存时，则会产生异常。在 proj1 的实验中，值设置了三个段描述符，即 NULL 段、TEXT 段和 DATA 段（都是 4GB 的访问范围）。

## 4.分段机制的系统寄存器

80386 有 4 个寄存器来寻址描述发表等系统数据结构，用来实现段式内存管理。内存管理寄存器包括：

- ◆ 全局描述符表寄存器 (Global Descriptor Table Register, GDTR)：指向全局段描述符表 GDT
- ◆ 局部描述符表寄存器 (Local Descriptor Table Register, LDTR)：指向局部段描述符表 LDT（目前用不上）
- ◆ 中断门描述符表寄存器 (Interrupt Descriptor Table Register, IDTR)：指向一张包含中断处理子程序入口点的表（IDT）
- ◆ 任务寄存器 (Task Register, TR)：这个寄存器指向当前任务信息存放处，这些信息是处理器进行任务切换所需要的。（目前用不上）

80386 有四个 32 位的控制寄存器，分别命名位 CR0、CR1、CR2 和 CR3。CR0 包含指示处理器工作方式、启用和禁止分页管理机制、控制浮点协处理器操作的控制位。具体描述如下：

- ◆ PE（保护模式允许 Protection Enable，比特位 0）：设置 PE 将让处理器工作在保护模式下。复位 PE 将返回到实模式工作。
- ◆ PG（分页允许 Paging，比特位 31）：PG 指明处理器是否通过页表来转换线性地址到物理地址。在后续试验中将讲述如何设置 PG 位。

CR0 中的位 5~位 30 是保留位，这些位的值必须为 0。CR2 及 CR3 由分页管理机制使用，将在后续试验中讲述。在 80386 中不能使用 CR1，否则会引起无效指令操作异常。

### 3.1.6 【实现】实模式到保护模式的切换

BIOS 把 bootloader 从硬盘（即是我们刚才生成的 ucore.img）的第一个扇区（即是我们刚才生成的 bootblock）读出来并拷贝到内存一个特定的地址 0x7c00 处，然后 BIOS 会跳转到那个地址（（即 CS=0，EIP=0x7c00））继续执行。至此 BIOS 的初始化工作做完了，进一步的工作交给了 ucore 的 bootloader。

bootloader 从哪里开始执行呢？我们【实验 2-2 编译运行 bootloader】中描述 make 工作过程的第五步就是生成了一个 bootblock.asm，它的前面几行是：

```
obj/bootblock.o:      file format elf32-i386

Disassembly of section .text:

00007c00 <start>:

.set CR0_PE_ON,      0x1          # protected mode enable flag

.globl start
start:

.code16                # Assemble for 16-bit mode

cli                    # Disable interrupts

7c00: fa               cli
```



上述代码片段指出了 bootblock（即 bootloader）在 0x7c00 虚拟地址（在这里虚拟地址=线性地址=物理地址）处的指令为“cli”，如果读者再回头看看 bootasm.S 中的 12~15 行：

```
.globl start
start:
    .code16                # Assemble for 16-bit mode
    cli                    # Disable interrupts
    cld                    # String operations increment
```

就可以发现二者是完全一致的。而这个虚拟地址的设定是通过链接器 ld 完成的，我们【实验 2-2 编译运行 bootloader】中描述 make 工作过程的第四步：

```
i386-elf-ld -N -e start -Ttext 0x7C00 -o obj/bootblock.o obj/bootasm.o obj/bootmain.o
```

其中“-e start”指出了 bootblock 的入口地址为 start，而“-Ttext 0x7C00”指出了代码段的起始地址为 0x7c00，这也就导致 start 位置的虚拟地址为 0x7c00。

从 0x7c00 开始，bootloader 用了 21 条汇编指令完成了初始化和切换到保护模式的工作。其具体步骤如下：

1. 关中断，并清除方向标志，即将 DF 置“0”，这样(E)SI 及(E)DI 的修改为增量。

```
cli    # Disable interrupts
cld    # String operations increment
```

2. 清零各数据段寄存器：DS、ES、FS

```
xorw    %ax,%ax    # Segment number zero
movw    %ax,%ds    # -> Data Segment
movw    %ax,%es    # -> Extra Segment
movw    %ax,%ss    # -> Stack Segment
```

3. 使能 A20 地址线，这样 80386 就可以突破 1MB 访存现在，而可访问 4GB 的 32 位地址空间了。可回顾 2.2.1 节的【历史：A20 地址线与处理器向下兼容】。

```
seta20.1:
    inb    $0x64,%al    # Wait for not busy
    testb  $0x2,%al
    jnz    seta20.1
    movb   $0xd1,%al    # 0xd1 -> port 0x64
    outb   %al,$0x64
seta20.2:
    inb    $0x64,%al    # Wait for not busy
    testb  $0x2,%al
    jnz    seta20.2
    movb   $0xdf,%al    # 0xdf -> port 0x60
    outb   %al,$0x60
```

4. 建立全局描述符表（可回顾 2.2.3 节对全局描述符表的介绍），使能 80386 的保护模式（可回顾 2.2.4 节对 CR0 寄存器的介绍）。lgdt 指令把 gdt 表的起始地址和界限（gdt 的大小-1）装入 GDTR 寄存器中。而指令“movl %eax, %cr0”把保护模式开启位置为 1，这时已经做好进入 80386 保护模式的准备，但还没有进入 80386 保护模式

```
lgdt    gdt desc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

`gdt desc` 指出了全局描述符表（可以看成是段描述符组成的一个数组）的起始位置在 `gdt` 符号处，而 `gdt` 符号处放置了三个段描述符的信息

```
gdt:
    SEG_NULLASM                      # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
    SEG_ASM(STA_W, 0x0, 0xffffffff)      # data seg
```

每个段描述符占 8 个字节，第一个是 NULL 段描述符，没有意义，表示全局描述符表的开始，紧接着是代码段描述符（位于全局描述符表的 0x8 处的位置），具有可读（`STA_R`）和可执行（`STA_X`）的属性，并且段起始地址为 0，段大小为 4GB；接下来是数据段描述符（位于全局描述符表的 0x10 处的位置），具有可读（`STA_R`）和可写（`STA_W`）的属性，并且段起始地址为 0，段大小为 4GB。

- 通过长跳转指令进入保护模式。80386 在执行长跳转指令时，会重新加载 `$PROT_MODE_CSEG` 的值（即 0x8）到 CS 中，同时把 `$protcseg` 的值赋给 EIP，这样 80386 就会把 CS 的值作为全局描述符表的索引来找到对应的代码段描述符，设定当前的 EIP 为 0x7c32（即 `protcseg` 标号所在的段内偏移），根据 2.2.3 节描述的分段机制中虚拟地址到线性地址转换转换的基本过程，可以知道线性地址（即物理地址）为：

`gdt[CS].base_addr+EIP=0x0+0x7c32=0x7c32`

```
ljmp    $PROT_MODE_CSEG, $protcseg
```

- 执行完上面的这条汇编语句后，bootloader 让 80386 从实模式进入了保护模式。由于在访问数据或栈时需要用 DS/ES/FS/GS 和 SS 段寄存器作为全局描述符表的下标来找到相应的段描述符，所以还需要对 DS/ES/FS/GS 和 SS 段寄存器进行初始化，使它们都指向位于 0x10 处的段描述符（即 `gdt` 中的数据段描述符）。

```
movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
movw    %ax, %ds                # -> DS: Data Segment
movw    %ax, %es                # -> ES: Extra Segment
movw    %ax, %fs                # -> FS
movw    %ax, %gs                # -> GS
movw    %ax, %ss                # -> SS: Stack Segment
```

在保护模式下，所有的内存寻址将经过分段机制的存储管理来完成，即每个虚拟地址访问将经过分段机制转换成线性地址，由于这时还没有启动分页模式，所以线性地址就是物理地址。

### 3.1.7 【实现】设置栈

这里为了减少汇编代码量，我们就通过 C 代码来完成显示。由于需要调用 C 语言的函数，所以需要自己建立好栈空间。设置栈的代码如下：

```
movl    $start, %esp
```

由于 `start` 位置（0x7c00）前的地址空间没有用到，所以可以用来作为 bootloader 的栈，需

要注意栈是向下长的，所以不会破坏 `start` 位置后面的代码。在后面的小节还会对栈进行更加深入的讲解。我们可以通过用 `gdb` 调试 `bootloader` 来进一步观察栈的变化：

【实验】用 `gdb` 调试 `bootloader` 观察栈信息

1. 开两个窗口；在一个窗口中，在 `proj1` 目录下执行命令 `make`；
2. 在 `proj1` 目录下执行 “`qemu -hda bin/ucore.img -S -s`”，这时会启动一个 `qemu` 窗口界面，处于暂停状态，等待 `gdb` 链接；
3. 在另外一个窗口中，在 `proj1` 目录下执行命令 `gdb obj/bootblock.o`；
4. 在 `gdb` 的提示符下执行如下命令，会有一定的输出：

```
(gdb) target remote :1234    #与 qemu 建立远程链接
(gdb) break bootasm.S:68    #在 bootasm.S 的第 68 行“movl $start, %esp”设置一个断点
(gdb) continue              #让 qemu 继续执行
```

这时 `qemu` 会继续执行，但执行到 `bootasm.S` 的第 68 行时会暂停，等待 `gdb` 的控制。这时可以在 `gdb` 中继续输入如下命令来分析栈的变化：

```
(gdb) info registers esp
esp                0xffd6  0xffd6  #没有执行第 68 行代码前的 esp 值
(gdb) si            #执行第 68 行代码
69      call bootmain
(gdb) info registers esp
esp                0x7c00  0x7c00  #当前的 esp 值，即栈顶
(gdb) si            #执行 call 汇编指令
bootmain () at boot/bootmain.c:87
87      bootmain(void) {
(gdb) info registers esp
esp                0x7bfc  0x7bfc  #当前的 esp 值 0x7bfc, 0x7bfc 处存放了 bootmain 函数的返回地址
0x7c4a, 这可以通过下面两个命令了解
(gdb) x /4x 0x7bfc
0x7bfc: 0x00007c4a    0xc031fcfa    0xc08ed88e    0x64e4d08e
(gdb) x /4i 0x7c40
0x7c40 <protcseg+14>:  mov    $0x7c00,%esp
0x7c45 <protcseg+19>:  call   0x7c6c <bootmain>
0x7c4a <spin>:        jmp     0x7c4a <spin>
0x7c4c <gdt>:        add     %al,(%eax)
```

【提示】

在 `proj1` 中执行  
`make debug`

则自动完成上述大部分前期工作，即 `qemu` 和 `gdb` 的加载，且 `gdb` 会自动建立于 `qemu` 的联接并设置好断点。具体实现可参看 `proj1` 的 `Makefile` 中于 `debug` 相关的内容和 `tools/gdbinit` 中的内容。

### 3.1.8 【实现】显示字符串

`bootloader` 只在 CPU 和内存中打转无法让读者很容易知道 `bootloader` 的工作是否正常，为此在成功完成了保护模式的转换后，就需要通过显示字符串来展示一下自己了。`bootloader`

---

设置好栈后，就可以调用 `bootmain` 函数显示字符串了。在 `proj1` 中使用了显示器和并口两种外设来显示字符串，主要的代码集中在 `bootmain.c` 中。

这里采用的是很简单的基于 **Programmed I/O (PIO)** 方式，PIO 方式是一种通过 CPU 执行 I/O 端口指令来进行数据读写的数据交换模式，被广泛应用于硬盘、光驱等设备的基础传输模式中。这种 I/O 访问方式使用 CPU I/O 端口指令来传送所有的命令、状态和数据，需要 CPU 全程参与，效率较低，但编程很简单。后面讲到的中断方式将更加高效。

在 `bootmain.c` 中的 `lpt_putc` 函数完成了并口输出字符的工作。输出一个字符的流程（可参看 `bootmain.c` 中的 `lpc_putc` 函数实现）大致如下：

1. 读 I/O 端口地址 `0x379`，等待并口准备好；
2. 向 I/O 端口地址 `0x378` 发出要输出的字符；
3. 向 I/O 端口地址 `0x37A` 发出控制命令，让并口处理要输出的字符。

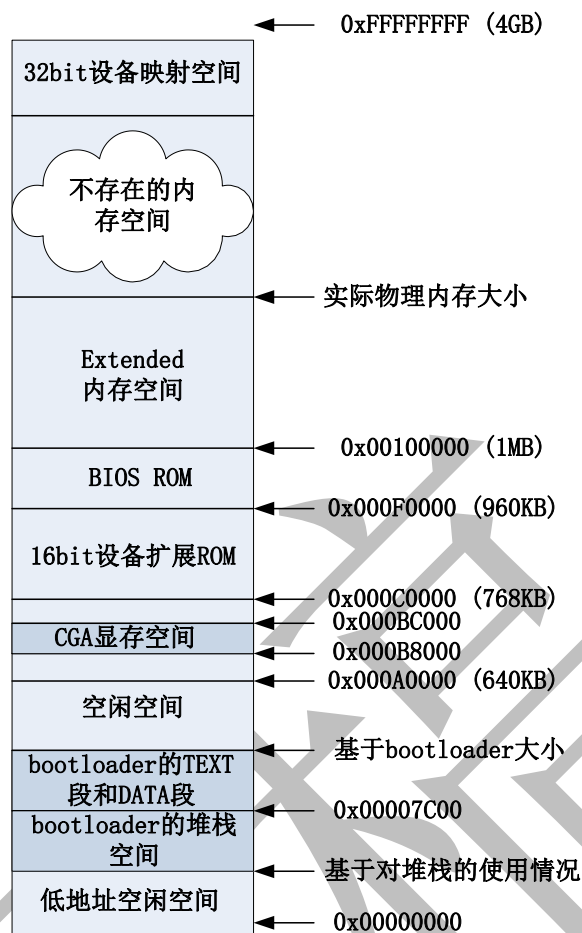
在 `bootmain.c` 中的 `serial_putc` 函数完成了串口输出字符的工作。输出一个字符的流程（可参看 `bootmain.c` 中的 `serial_putc` 函数实现）大致如下：

1. 读 I/O 端口地址(`0x3f8+5`)获得 LSR 寄存器的值，等待串口输出准备好；
2. 向 I/O 端口地址 `0x3f8` 发出要输出的字符；

在 `bootmain.c` 中的 `cga_putc` 函数完成了 CGA 字符方式在某位置输出字符的工作。输出一个字符的流程（可参看 `bootmain.c` 中的 `cga_putc` 函数实现）大致如下：

1. 写 I/O 端口地址 `0x3d4`，读 I/O 端口地址 `0x3d5`，获得当前光标位置；
2. 在光标的下一位置的显存地址空间上写字符，格式是黑色背景/白色字符；
3. 设置当前光标位置为下一位置。

`proj1` 启动后的 PC 机内存布局如下图所示：



自此,我们了解了一个小巧的bootloader的实现过程,但这还仅仅是百尺竿头的第一步,它还只能显示字符串,不能加载操作系统。我们还需要扩展 bootloader 的功能,让它能够加载操作系统。

## 3.2 可读 ELF 格式文件的 bootloader 和可完成 I/O 操作的 ucore

### 3.2.1 实验目标

接下来,我们需要完成一个能够读取位于硬盘中 OS 的代码内容并加载运行 OS 的 bootloader,这需要 bootloader 能够读取硬盘扇区中的数据。由于 OS 采用 ELF 执行文件格式,所以 bootloader 能够解析 ELF 格式文件,把其中的代码和数据放到内存中正确的位置。Bootloader 虽然增加了这么多功能,但整个 bootloader 的大小还是必须小于 512 个字节,这样才能放到只有 512 字节大小的硬盘主引导扇区中。通过分析和实现这个 bootloader,读者可以了解到如何在保护模式下通过 PIO 方式访问块设备硬盘;理解如何在保护模式下解析并加载一个简单的 ELF 执行文件。

## 3.2.2 proj2/3 概述

### 1. 实现描述

proj2 基于 proj1 的主要实现一个可读硬盘并可分析 ELF 执行文件格式的 bootloader，由于 bootloader 要放在 512 字节大小的主引导扇区中，所以不得不去掉部分显示输出的功能，确保整个 bootloader 的大小小于 510 个字节（最后两个字节用于硬盘主引导扇区标识，即“55AA”）。proj3 在 proj2 的基础上增加了一个只能显示字符的第一代幼稚型操作系统 ucore，用来验证 proj2 实现的 bootloader 能够正确从硬盘读出 ucore 并加载到正确的内存位置，并能把 CPU 控制权交给 ucore。ucore 在获得 CPU 控制权后，能够在保护模式下显示一个字符串，表明自己能够正常工作了。

### 2. 项目组成

这里我们分了两个 project 来完成此事。proj2 是一个可分析 ELF 执行文件格式的例子，proj2 整体目录结构如下所示：

```
proj2/  
|-- boot  
|   |-- asm.h  
|   |-- bootasm.S  
|   |-- bootmain.c  
|-- libs  
|   |-- elf.h  
|   |-- types.h  
|   |-- x86.h  
|-- Makefile  
.....
```

proj2 与 proj1 类似，只是增加了 libs/elf.h 文件，并且 bootmain.c 中增加了对 ELF 执行文件的简单解析功能和读磁盘功能。

proj3 建立在 proj2 基础之上，增加了一个只能显示字符的 ucore 操作系统，让 bootloader 能够把这个操作系统从硬盘上读到内存中，并跳转到 ucore 的起始处执行 ucore 的功能。proj3 整体目录结构如下所示：

```
proj3  
|-- boot  
|   |-- asm.h  
|   |-- bootasm.S  
|   |-- bootmain.c  
|-- kern  
|   |-- driver
```

```

| | |-- console.c
| | `-- console.h
| |-- init
| | `-- init.c
| `-- libs
|     `-- stdio.c
|-- libs
| |-- elf.h
| |-- error.h
| |-- printfmt.c
| |-- stdarg.h
| |-- stdio.h
| |-- string.c
| |-- string.h
| |-- types.h
| `-- x86.h
|-- Makefile
.....

```

proj3 相对于 proj2 增加了 ucore 相关的文件，下面简要说明一下：

- libs 目录下的 `printfmt.c`：完成类似 C 语言的 `printf` 中的格式化处理；
- libs 目录下的 `string.c`：完成类似 C 语言的 `str***` 相关的字符串处理函数；
- libs 目录下的 `st*.h`：是支持上述两个库函数(可被内核和用户应用共享)的.h 文件；
- kern/init 目录下的 `init.c`：完成 ucore 的初始化工作；
- kern/driver 目录下的 `console.c`：提供并口/串口/CGA 方式的字符输出的 console 驱动；
- kern/libs/stdio.c：提供内核方式下的 `cprintf` 函数功能；

### 3. 编译运行

那接下来是如何生成一个包含了 bootloader 和 ucore 操作系统的硬盘镜像呢？我们先修改 proj3 目录下的 Makefile，在其第五行

```
V      := @
```

的最前面增加一个“#”（目的是让 make 工具程序详细显示整个 project 的编译过程），这样就把这行给注释了。然后在 proj3 目录下执行 make，可以看到：

```

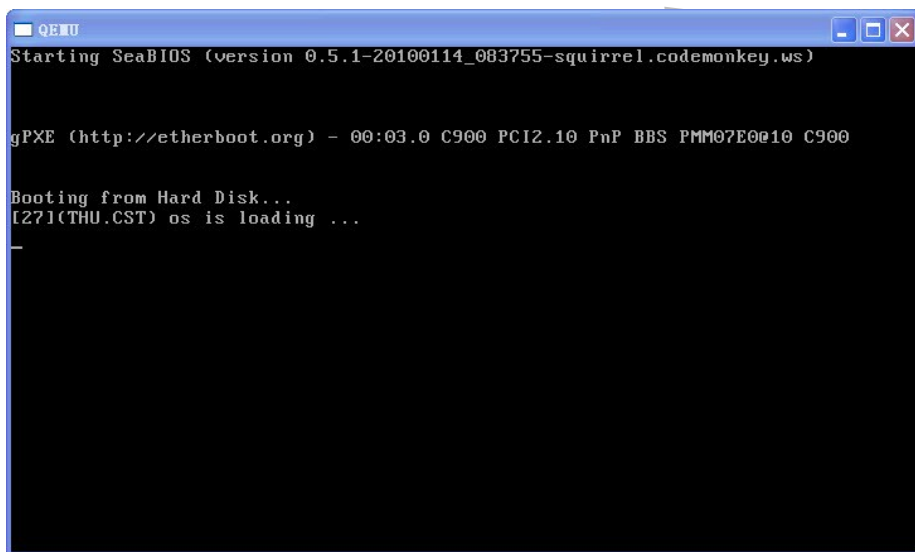
.....
ld -m      elf_i386 -Ttext 0x100000 -e kern_init -o bin/kernel obj/kern/init/init.o
obj/kern/libs/printf.o obj/kern/driver/console.o obj/libs/printfmt.o obj/libs/string.o
.....

```

```
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
```

这两步是生成 ucore 的关键。第一步把 ucore 涉及各个.o 目标文件链接起来，并在 bin 目录下形成 ELF 文件格式的文件 kernel，这就是我们第一个 ucore 操作系统，而且设定 ucore 的执行入口地址在 0x10000，即 kern\_init 函数的起始位置。这也就意味着 bootloader 需要把读出的 kernel 文件的代码段+数据段放置在 0x10000 起始的内存空间。第二步是把 bin 目录下的 kernel 文件直接覆盖到 ucore.img（虚拟硬盘的文件）的 bootloader 所处扇区（即第一个扇区，主引导扇区）之后的扇区（第二个扇区）。如果一个扇区大小为 512 字节，这 kernel 覆盖的扇区数为上取整（kernel 的大小/512 字节）。

编译后运行 proj3 的示意图如下所示：



### 3.2.3 【背景】访问硬盘数据控制

bootloader 让 80386 处理器进入保护模式后，下一步的工作就是从硬盘上加载并运行 OS。考虑到实现的简单性，bootloader 的访问硬盘都是 LBA 模式的 PIO（Program IO）方式，即所有的 I/O 操作是通过 CPU 访问硬盘的 I/O 地址寄存器完成。

一般主板有 2 个 IDE 通道（即 2.1.4 节介绍的 I/O 控制器），每个通道可以接 2 个 IDE 硬盘。第一个 IDE 通道通过访问 I/O 地址 0x1f0-0x1f7 来实现，第二个 IDE 通道通过访问 0x170-0x17f 实现。每个通道的主从盘的选择通过第 6 个 I/O 偏移地址寄存器来设置。具体参数见下表。

| I/O 地址 | 功能                                                                                                     |
|--------|--------------------------------------------------------------------------------------------------------|
| 0x1f0  | 读数据，当 0x1f7 不为忙状态时，可以读。                                                                                |
| 0x1f2  | 要读写的扇区数，每次读写前，需要指出要读写几个扇区。                                                                             |
| 0x1f3  | 如果是 LBA 模式，就是 LBA 参数的 0-7 位                                                                            |
| 0x1f4  | 如果是 LBA 模式，就是 LBA 参数的 8-15 位                                                                           |
| 0x1f5  | 如果是 LBA 模式，就是 LBA 参数的 16-23 位                                                                          |
| 0x1f6  | 第 0~3 位：如果是 LBA 模式就是 24-27 位    第 4 位：为 0 主盘；为 1 从盘<br>第 6 位：为 1=LBA 模式；0 = CHS 模式    第 7 位和第 5 位必须为 1 |
| 0x1f7  | 状态和命令寄存器。操作时先给命令，再读取内容；如果不是忙状态就从 0x1f0                                                                 |



|  |       |
|--|-------|
|  | 端口读数据 |
|--|-------|

硬盘数据是储存在硬盘扇区中，一个扇区大小为 512 字节。读一个扇区的流程大致为通过 outb 指令访问 I/O 地址:0x1f2~0x1f7 来发出读扇区命令，通过 in 指令了解硬盘是否空闲且就绪，如果空闲且就绪，则通过 inb 指令读取硬盘扇区数据都内存中。可进一步参看 bootmain.c 中的 readsect 函数实现来了解通过 PIO 方式访问硬盘扇区的过程。

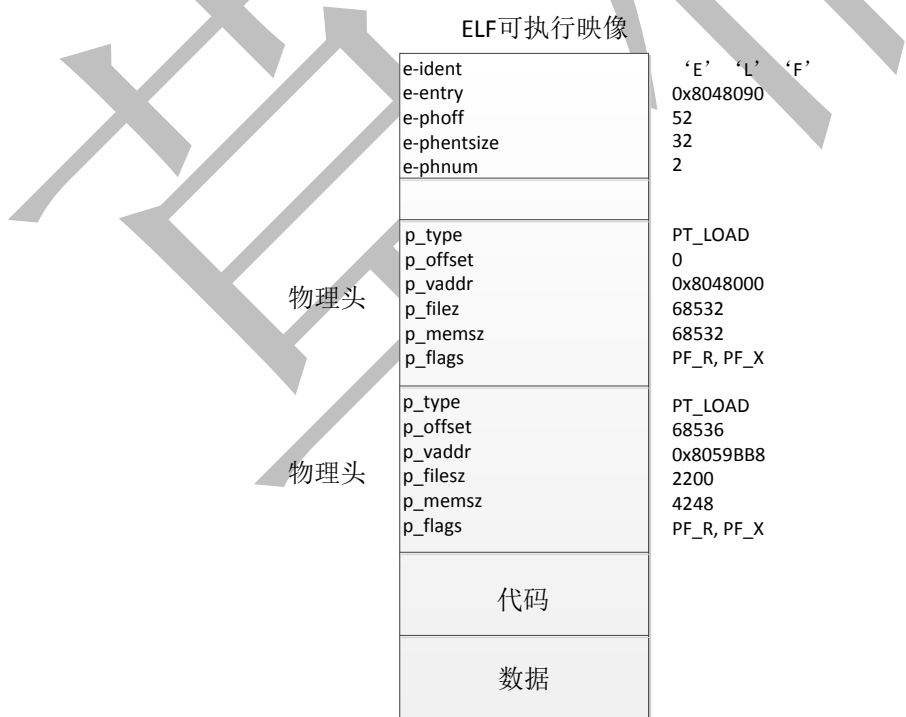
### 3.2.4 【背景】理解 ELF 文件格式

由于本章的 project 中，bootloader 会访问 ELF(Executable and linking format)格式的 ucore，并把 ucore 加载到内存中。所以，在这里我们需要简单介绍一下 ELF 文件格式，以帮助我们理解 ucore 的整个编译、链接和加载的过程，特别是希望读者对 ld 链接器用到的链接地址（Link address）和操作系统相关的加载地址（Load address）有更清楚的了解。

ELF 文件格式是 Linux 系统下的一种常用目标文件(object file)格式，有三种主要类型。可重定位文件(relocatable file)类型和共享目标文件(shared object file)类型在本实验中没有涉及。本实验的 OS 文件类型是可执行文件(executable file)类型，这种 ELF 文件格式类型提供程序的进程映像，加载程序的内存地址描述等。

简单地说，bootloader 通过解析 ELF 格式的 ucore，可以了解到 ucore 的代码段(机器码)/数据段（初始化的变量）等在文件中的位置和大小，以及应该放到内存中的位置；可了解 ucore 的 BSS 段（未初始化的变量，具体内容没有保存在文件中）的内存位置和大小。这样 bootloader 就可以把 ucore 正确地放置到内存中，便于 ucore 的正确执行。

这里只分析与本章相关的 ELF 可执行文件类型。ELF 的执行文件映像如下所示：



一个简单的ELF可执行文件的布局

ELF 的文件头包含整个执行文件的数据结构 elf header，描述了整个执行文件的组织结构。其定义在 proj2/3 中的 elf.h 文件中：

```

struct elfhdr {

    uint32_t e_magic;      // must equal ELF_MAGIC

    uint8_t e_elf[12];

    uint16_t e_type;       // 1=relocatable, 2=executable, 3=shared object, 4=core image

    uint16_t e_machine;    // 3=x86, 4=68K, etc.

    uint32_t e_version;    // file version, always 1

    uint32_t e_entry;      // entry point if executable

    uint32_t e_phoff;      // file position of program header or 0

    uint32_t e_shoff;      // file position of section header or 0

    uint32_t e_flags;      // architecture-specific flags, usually 0

    uint16_t e_ehsize;     // size of this elf header

    uint16_t e_phentsize;  // size of an entry in program header

    uint16_t e_phnum;      // number of entries in program header or 0

    uint16_t e_shentsize;  // size of an entry in section header

    uint16_t e_shnum;      // number of entries in section header or 0

    uint16_t e_shstrndx;   // section number that contains section name strings

};

```

program header 描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。可执行文件的程序前面部分有一个 program header 结构的数组，每个结构描述了一个“段”（segment）或者准备程序执行所必需的其它信息。目标文件的“段”（segment）包含一个或者多个“节区”（section），也就是“段内容（Segment Contents）”。program header 仅对于可执行文件和共享目标文件有意义。可执行目标文件在 elfhdr 的 e\_phentsize 和 e\_phnum 成员中给出其自身程序头部的大小。程序头部的数据结构如下表所示：

```

struct proghdr {

    uint32_t p_type;      // loadable code or data, dynamic linking info,etc.

    uint32_t p_offset;    // file offset of segment

    uint32_t p_va;        // virtual address to map segment

    uint32_t p_pa;        // physical address, not used

    uint32_t p_filesz;    // size of segment in file

    uint32_t p_memsz;     // size of segment in memory (bigger if contains bss)

    uint32_t p_flags;     // read/write/execute bits

    uint32_t p_align;     // required alignment, invariably hardware page size

```

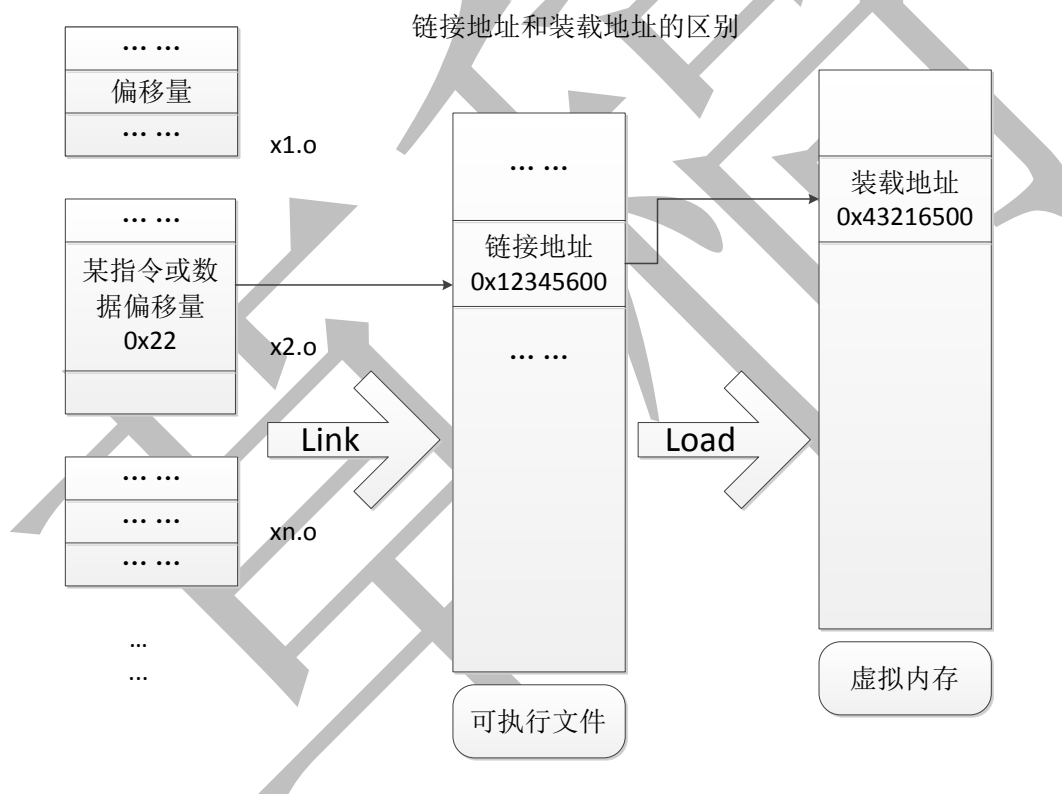
```
};
```

### 链接地址（Link address）和加载地址（Load address）

Link Address 是指编译器指定代码和数据所需要放置的内存地址，由链接器配置。Load Address 是指程序被实际加载到内存的位置。一般由可执行文件结构信息和加载器可保证这两个地址相同。Link Addr 和 LoadAddr 不同会导致：

- 直接跳转位置错误
- 直接内存访问(只读数据区或 bss 等直接地址访问)错误
- 堆和栈等的使用不受影响，但是可能会覆盖程序、数据区域

也存在 Link 地址和 Load 地址不一样的情况(如动态链接库)。在 proj3 中, bootloader 和 ucore 的链接地址和加载地址是一致的。



### 3.2.5 【背景】操作系统执行代码的组成

ucore 通过 gcc 编译和 ld 链接，形成了 ELF 格式执行文件 kernel（位于 bin 目录下），这样 kernel 的内部组成与一般的应用程序差别不大。一般而言，一个执行程序的内容是至少由 bss 段、data 段、text 段三大部分组成。

- BSS 段：BSS（Block Started by Symbol）段通常是指用来存放执行程序中未初始化的全局变量的一块存储区域。BSS 段属于静态内存分配的存储空间。
- 数据段：数据段（Data Segment）通常是指用来存放执行程序中已初始化的全局变量的一块存储区域。数据段属于静态内存分配的存储空间。

- 代码段：代码段（Code Segment/Text Segment）通常是指用来存放程序执行代码的一块存储区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些 CPU 架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

ucore 和一般应用程序一样，首先是保存在像硬盘这样的非易失性存储介质上，当需要运行时，被加载到内存中。这时，需要把代码段、数据段的内容拷贝到内存中。对于位于 BSS 段中的未初始化的全局变量，执行程序一般认为其值为零。所以需要把 BSS 段对应的内存空间清零，确保执行代码的正确运行。可查看 init 文件中的 kern\_init 函数的第一个执行语句“memset(edata, 0, end - edata);”。

随着 ucore 的执行，可能需要进行函数调用，这就需要用到栈（stack）；如果需要动态申请内存，这就需要用到堆（heap）。堆和栈是在操作系统执行过程中动态产生和变化的，并不存在于表示内核的执行文件中。栈又称堆栈，是用户存放程序临时创建的局部变量，即函数中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用函数的栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。可以把栈看成一个寄存、交换临时数据的内存区。堆是用于存放运行中被动态分配的内存空间，它的大小并不固定，可动态扩张或缩减，这需要操作系统自己进行有效的管理。

### 3.2.6 【实现】bootloader 加载并运行 ucore

了解完 proj2/3 的组成与编译，并大致理解上述两个背景知识后，我们就可以分析 bootloader 加载并运行 ucore 操作系统的工作流程。

硬盘数据是储存在硬盘扇区中，一个扇区大小为 512 字节。读一个扇区的流程可参看 bootmain.c 中的 readsect 函数实现。大致如下：

1. 读 I/O 地址 0x1f7，等待磁盘准备好；
2. 写 I/O 地址 0x1f2~0x1f5, 0x1f7，发出读取第 offset 个扇区处的磁盘数据的命令；
3. 读 I/O 地址 0x1f7，等待磁盘准备好；
4. 连续读 I/O 地址 0x1f0，把磁盘扇区数据读到指定内存。

这个函数是被 bootloader 用于读取硬盘上的 ucore 操作系统。bootloader 为了读取硬盘上的 ucore 操作系统，将调用 bootmain 函数首先读取了位于主引导扇区的后的连续 8 个扇区（可参见 bootmain 函数中的第一条语句），并把数据放到 0x10000 处（可回顾一下 2.7.1 中描述链接 bin/kernel 的过程），并按照数据结构 elfhdr 来解析这块 4KB 大小的数据；如果其 e\_magic 数据域不等于 ELF\_MAGIC（即 0x464C457F），则表示这个不是标准的 ELF 格式的文件；如果等于 ELF\_MAGIC，则继续解析，并根据其 e\_phnum 数据域的值来读取多个 program header，并根据 program header 的信息，了解到 ucore 中各个 segment 的起始位置和大小，然后把放在硬盘上的相关 segment 读入到内存中。

【实验】分析 kernel 并在 bootloader 中显示 kernel 的 segment 信息

1. 在 proj3 目录下执行命令 make，则会在 bin 目录下生成 kernel，即 ELF 执行格式文件的操作系统 ucore；
2. 在 proj3 目录下执行命令 readelf -h bin/kernel，可得到有关 elf header 的如下信息

|             |
|-------------|
| ELF Header: |
|-------------|

```

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x100000
Start of program headers: 52 (bytes into file)
Start of section headers: 19872 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 3
Size of section headers: 40 (bytes)
Number of section headers: 17
Section header string table index: 14

```

从中，我们可以看到 kernel 的入口点在 0x100000，program header 相对文件的偏移位置在 52，elf header 的大小为 52 字节，program header 的大小为 32 字节。

3. 在 proj3 目录下执行命令 `readelf -l bin/kernel`，可得到有关 program header 的如下信息

```

Elf file type is EXEC (Executable file)
Entry point 0x100000
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset    VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x01038 0x01038 R E 0x1000
  LOAD           0x002038 0x00102038 0x00102038 0x00004 0x00004 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

Section to Segment mapping:
Segment Sections...
  00      .text .rodata
  01      .data
  02

```

从中，我们可以看到 kernel 的入口点在 0x100000，代码段位于 0x100000，大小为 0x1038；数据段位于 0x102038，大小为 0x04。

【实验】用 gdb 调试 bootloader，并在 gdb 中显示 kernel 的 segment 信息

我们还可通过用 gdb 调试 bootloader 进行验证，具体步骤如下：

5. 开两个窗口；在一个窗口中，在 proj3 目录下执行命令 make；
6. 在 proj3 目录下执行 “qemu -hda bin/ucore.img -S -s”，这时会启动一个 qemu 窗口界面，处于暂停状态，等待 gdb 链接；
7. 在另外一个窗口中，在 proj3 目录下执行命令 gdb obj/bootblock.o；
8. 在 gdb 的提示符下执行如下命令，会有一定的输出：

```
(gdb) target remote :1234    #与 qemu 建立远程链接
(gdb) break bootmain.c:100   #在 bootmain.c 的第 100 行设置一个断点
(gdb) continue               #让 qemu 继续执行
```

这时 qemu 会继续执行，但执行到 bootmain.c 的第 100 行时会暂停，等待 gdb 的控制。这时可以在 gdb 中继续输入如下命令来参考 kernel 的信息：

```
(gdb) p/x *(struct elfhdr *)0x10000    #按 struct elfhdr 结构显示 0x10000 处内容
$7 = {e_magic = 0x464c457f, e_elf = {0x1, 0x1, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
e_type = 0x2, e_machine = 0x3, e_version = 0x1, e_entry = 0x100000, e_phoff = 0x34, e_shoff =
0x4550, e_flags = 0x0, e_ehsize = 0x34, e_phentsize = 0x20, e_phnum = 0x3, e_shentsize = 0x28,
e_shnum = 0x11, e_shstrndx = 0xe}
```

查看 bootmain 函数，可以知道，此时在 0x10000 处已经读入了 kernel 的 ELF 头信息，有三个 program header 表(e\_phnum 值),继续在 gdb 中敲入命令，可以得到更多信息：

```
(gdb) next                    #执行下一条指令
(gdb) p /x *ph                #获得 text 段的 program header 表信息
$5 = {p_type = 0x1, p_offset = 0x1000, p_va = 0x100000, p_pa = 0x100000, p_filesz = 0x1038,
p_memsz = 0x1038, p_flags = 0x5, p_align = 0x1000}
(gdb) next                    #执行下一条指令
(gdb) next                    #执行下一条指令
(gdb) p /x *ph                #获得 data 段的 program header 表信息
$6 = {p_type = 0x1, p_offset = 0x2038, p_va = 0x102038, p_pa = 0x102038, p_filesz = 0x4,
p_memsz = 0x4, p_flags = 0x6, p_align = 0x1000}
```

对照 readelf 命令输出的信息，可以发现 bootloader 正确读出了 text 段和 data 段的 program header 表信息，并根据这些信息调用如下函数

```
-->readseg(ph->p_va, ph->p_memsz, ph->p_offset);
-->readsect((uint8_t *)va, offset);
```

把这两个段的内容读入到正确的线性内存地址中。然后再根据 e\_entry = 0x100000，跳转到 0x100000 处去执行，这其实就是把处理器控制权转移给了 ucore 了。

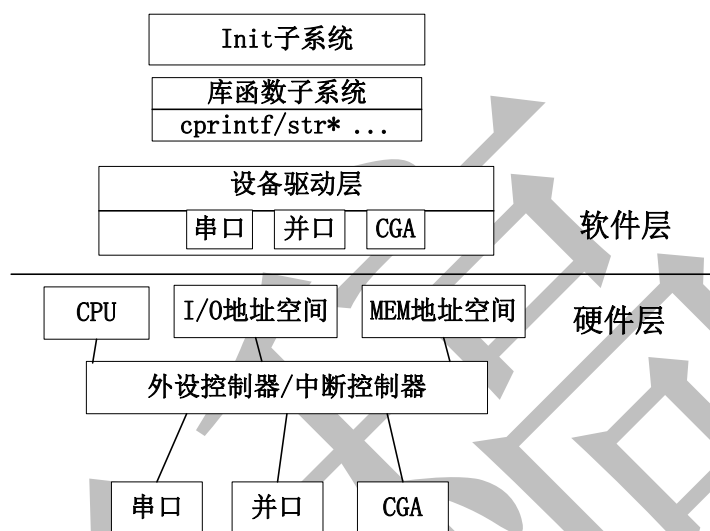
### 3.2.7 【实现】可输出字符串的 ucore

proj3 包含了一个只能输出字符串的简单 ucore 操作系统，虽然简单，但它也体现了操作系统的一些结构和特征，比如它具有：

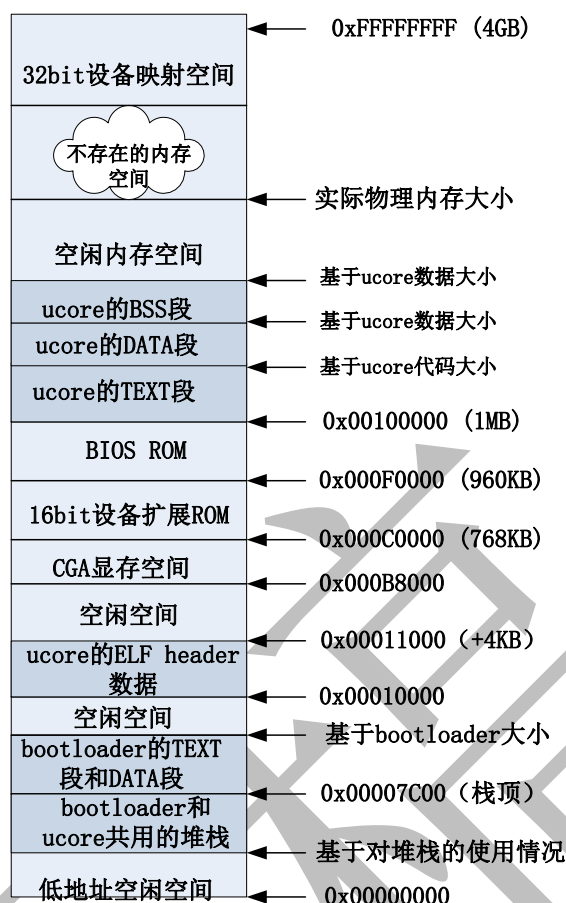
- 完成给 ucore 的 BSS 段清零并显示一个字符串的内核初始化子系统（init.c）
- 提供串口/并口/CGA 显示的驱动程序子系统（console.c）

- 提供公共服务的操作系统函数库子系统（printf.c printfmt.c string.c）

这体现了操作系统的一个基本特征：资源管理器。从操作系统原理我们可以知道一台计算机就是一组资源，这些资源用于对数据的移动、存储和处理并进行控制。在 proj3 中的 ucore 操作系统目前只提供了对串口/并口/CGA 这三种 I/O 设备的硬件资源的访问，每个 I/O 设备的操作都有自己特有的指令集或控制信号（对照一下 serial\_putc/lpt\_putc/cga\_putc 函数的实现），操作系统隐藏这些细节，并提供了统一的接口（看看 cprintf 函数的实现），因此程序员可以使用简单的 printf 函数来写这些设备，达到显示数据的效果。目前操作系统的逻辑结构图架构如下图所示：



在 PC 中的地址空间布局图如下所示：



### 3.3 能显示函数调用关系的 ucore

操作系统中存在很多函数,通过函数间的调用来完成各种功能。在操作系统运行过程中,维持函数之间的调用关系,以及函数内部的局部变量是栈(stack)的基本功能。我们需要能够理解在操作系统中栈的实现细节和功能,这样能够更好地理解操作系统中函数如何相互调用,发现可能存在的问题。

#### 3.3.1 实验目标

为了理解操作系统中的函数调用关系、传递参数和函数局部变量,我们设计了 proj3.1,在 ucore 中增加了一个 monitor 子功能模块,能够分析出 ucore 在执行过程中的函数调用关系和函数传递的参数。通过分析 proj3.1 的实现,读者可了解基本栈结构,栈处理流程,GCC 编译器的参数传递约定和构建函数调用关系的具体实现。



### 3.3.2 proj3.1 概述

#### 1. 实现描述

proj3.1 建立在 proj3 的基础上，通过增加了一个 monitor 功能子模块，实现了一个能显示函数调用关系的 ucore。简单地说 proj3.1 根据 GCC 生成的栈构建代码、函数参数压栈约定和实际函数调用过程中的栈结构内存空间，分析并显示函数调用关系。具体完成此工作的是 print\_stackframe 函数。

#### 2. 项目组成

proj3.1 整体目录结构中新增加的主要内容如下所示：

```
proj3.1
|-- kern
|   |-- debug
|   |   |-- assert.h
|   |   |-- kdebug.c
|   |   |-- kdebug.h
|   |   |-- monitor.c
|   |   |-- monitor.h
|   |   |-- panic.c
|   |   |-- stab.h
|   |-- driver
|   |   |-- kbdreg.h
|   |-- init
|   |   |-- init.c
|   |-- libs
|   |   |-- readline.c
|   |   |-- stdio.c
|   |-- trap
|   |   |-- trap.h
|-- Makefile
|-- tools
|   |-- kernel.ld
.....
```

proj3.1 是基于 proj3 进一步扩展完成的。相对于 proj3，一共增加了 10 个文件，主要集中在 debug 目录下，这一个比较大的跨越。不过仔细看来主要增加和修改的文件不多，具体新增内容如下所示：

- kern/debug/monitor.[ch]：监视系统运行的 monitor 交互子模块
- kern/debug/debug.[ch]：实现内存地址到函数名的映射和分析显示函数调用关系；

- kern/libs/readline.c: 实现 monitor 的接收字符输入的功能;
- kern/driver/kbdreg.h: 定义了键盘的键值;
- kern/trap/trap.h: 为了向后兼容中断的处理, 先在此处写了一个空的 trapframe 结构;
- tools/kernel.ld: 指导 ld 工具软件链接各个.o 目标文件形成 ucore 的 kernel 的链接脚本;

### 3. 编译运行

编译并运行 proj3.1 的命令如下:

```
cd proj3.1
make
make qemu
```

当 “K>” 提示符出现后, 可以敲入 “help” 字符串, 可以看到当前的 monitor 有三个命令可以输入: help、kerninfo、backtrace。我们敲入 “backtrace” 字符串, 则可以得到如下显示界面:

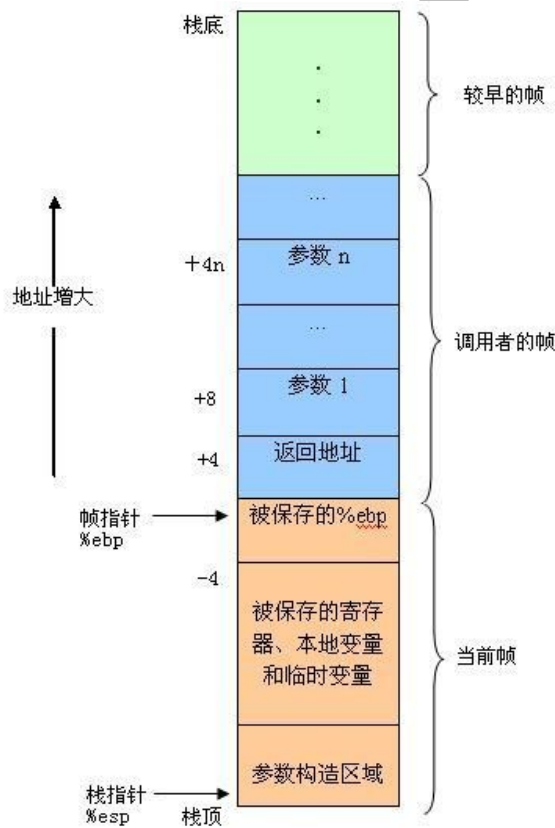
```
(THU.CST) os is loading ...

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> help
help - Display this list of commands.
kerninfo - Display information about the kernel.
backtrace - Print backtrace of stack frame.
K> backtrace
ebp:0x00007b08 eip:0x0010073a args:0x00010094 0x00000000 0x00007b88 0x00100985
    kern/debug/kdebug.c:216: print_stackframe+25
ebp:0x00007b18 eip:0x00100a76 args:0x00000000 0x00007b3c 0x00000000 0x0000000a
    kern/debug/monitor.c:94: mon_backtrace+11
ebp:0x00007b88 eip:0x00100985 args:0x00108560 0x00000000 0x00007bb8 0x0010124c
    kern/debug/monitor.c:55: runcmd+135
ebp:0x00007bb8 eip:0x001009f8 args:0x00000000 0x00101ef0 0x0000065c 0x00000000
    kern/debug/monitor.c:70: monitor+75
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:22: kern_init+89
ebp:0x00007bf8 eip:0x00007d5b args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
```

通过上图可以看到 monitor 能够把当前的函数调用关系给显示出来, 而且不仅仅给出函数调用返回处的 eip 地址, 还显示出了在实际源代码处的文件名和行号。如果 ucore 在执行过程中由于某种异常错误激发 monitor 执行 (以后有这样的实验), 我们就可以很容易找到问题出现在什么地方了。下面我们将从栈的基本概念、栈结构和栈处理过程等方面来分析上图中背后的东西。

### 3.3.3 【背景】栈结构和处理过程

根据数据结构课程中的描述，栈是限定仅在表尾进行插入（即入栈操作）或删除操作（即出栈操作）的线性表[严蔚敏著的《数据结构》书]。因此，栈的表尾端成为称为栈顶（stack top），表头端称为栈底（stack bottom）。在 X86CPU 架构中有专门的指令“push”来完成入栈操作，“pop”指令来完成出栈操作，栈顶指针寄存器 ESP 时刻指向栈的栈顶。比较有趣的是，在 x86 中，采用的是满降序栈（full descending stack）机制，即栈底在高地址，栈顶在低地址，入栈的方向是向低地址进行，出栈的方向是向高地址进行，栈指针指向上次写的最后一个数据单元。GCC 编译器规定的函数栈帧（stack frame）是一块存放某函数的局部变量、参数、返回地址和其它临时变量的内存空间，栈帧的大致结构和操作如下图所示：



操作系统中使用栈的目的与一般应用程序类似，不外乎包括：支持函数调用、传递函数参数、局部变量（也称自动变量）存储、函数返回值存储、在函数内部保存可能被修改的寄存器的值供恢复。除此之外，在后续讲到的中断处理、内核态/用户态切换、进程切换等方面，也需要使用栈来保存被打断的执行所用到的寄存器等硬件信息。由于在操作系统中要完成许多非常规的栈空间数据处理，比如直接修改保存在栈帧中的返回地址，使得函数返回到不同的地方去，所以我们需要在计算机体系结构和机器代码级别更加深入地理解操作系统是如何具体进行栈处理的。下面，我们将结合调试运行 proj3.1 来分析内核中的函数调用关系。

### 3.3.4 【实现】分析内核函数调用关系

首先，ucore 需要建立一个空的栈空间，然后才能进行函数调用、参数传递等处理工作。ucore 是在哪里建立的栈呢？其实 ucore 是借用了 bootloader 的栈空间，而 bootloader 在 bootasm.S 中的如下语句建立的栈空间：

```
# Set up the stack pointer and call into C.
```

```
movl $0x0, %ebp
```

```
movl $start, %esp
```

可以看到 `bootloader` 把栈底设置到了 `$start` 地址处，正好是 `bootloader` 的起始地址 `0x7c00`。不过由于入栈操作中的 `esp` 是向下增长的，所以不会覆盖 `bootloader` 的内容。那 `ebp` 有何作用呢？我们先暂时放在一边，继续跟踪代码的执行。

接下来，`bootloader` 会调用 `bootmain()` 函数，而 `bootmain()` 函数会在加载完 `ucore` 后，调用 `ucore` 的起始函数 `kern_init()`。在 `ucore` 的继续执行过程中，还将有如下的函数调用过程：`kern_init-->monitor-->runcmd-->mon_backtrace-->print_stackframe`。这通过看源代码或执行 `monitor` 中的 `backtrace` 命令都可以了解到。

我们可以结合前面的实验来说明 `ucore` 是如何分析出这样的调用关系的。

操作系统中的中断（也称异常）技术是操作系统的重要功能，是计算机硬件和软件相配合产生的重要技术。简单地说，中断处理是指由于有紧急事件产生，需要打断当前 CPU 的正常执行，转而处理紧急事件，处理完毕后，恢复到被打断的地方继续执行。通过中断机制，计算机系统可以高效地处理外设请求，可以快速响应应用软件的异常或请求，也可以有规律地打断应用程序的执行，把执行 CPU 控制权还到操作系统手中，从而使得整个计算机系统的资源可控。但单纯的操作系统原理书籍很难深入分析中断的处理细节。我们希望通过后续的 `proj4/4.1.1` 等的实验，让读者了解到 `ucore` 操作系统如何完成上述事情。

首先我们需要了解 GCC 生成的 C 函数调用过程：

- 1、调用函数为了传递参数给被调用函数，需要执行 0 到 n 个 `push` 指令把函数参数入栈，然后会执行一个 `call` 指令，在 `call` 指令内部执行过程中，还把返回地址（即 `CALL` 指令下一条指令的地址）也入栈了。
- 2、GCC 编译器会在每个函数的起始部分插入类似如下指令（可参看 `obj/kernel.asm` 文件内容）：

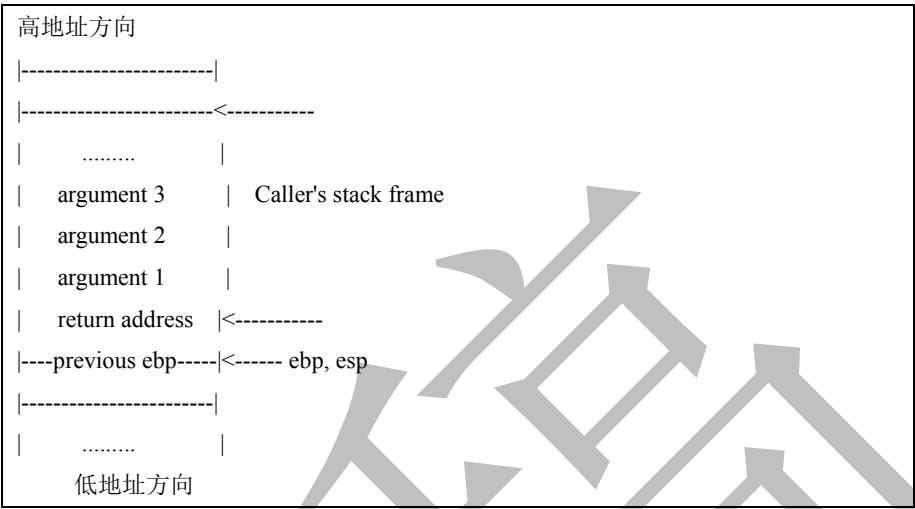
```
push    %ebp
mov     %esp,%ebp
sub     $NUM,%esp
```

在 `ucore` 执行到一个函数的函数体时，已经有以下数据顺序入栈：调用函数的参数，函数返回地址。由此得到类似如下的栈结构（参数入栈顺序跟调用方式有关，这里以 C 语言默认的 `CDECL` 为例）：

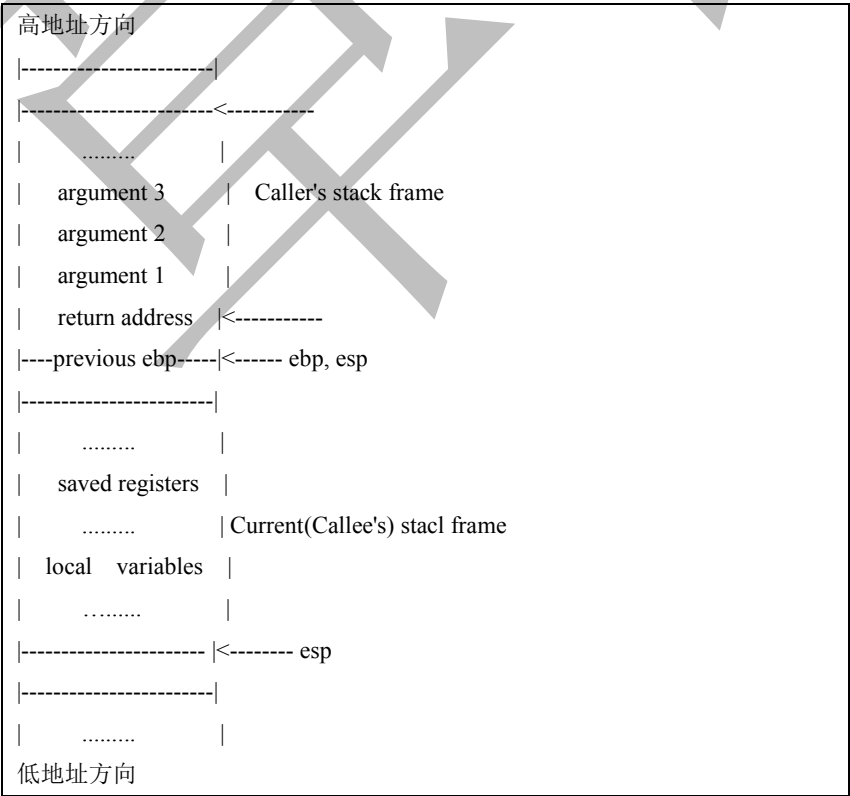
| 高地址方向          |                      |
|----------------|----------------------|
| -----          |                      |
| -----          | <-----               |
| .....          |                      |
| argument 3     | Caller's stack frame |
| argument 2     |                      |
| argument 1     |                      |
| return address | <----- esp           |
| -----          |                      |
| .....          |                      |

低地址方向

“push %ebp”和“mov %esp,%ebp”这两条指令实在隐含了对函数调用关系链的建立：首先将 ebp 入栈，然后将栈顶指针 esp 赋值给 ebp，此时的栈结构如下所示：



“mov %esp,%ebp”这条指令表面上看是用 esp 把 ebp 的旧值覆盖了，但在这条语句之前，ebp 旧值已经被压栈（位于栈顶），而新的 ebp 又恰恰指向栈顶。第三条语句“sub \$NUM,%esp”把 esp 减少了 NUM 个值，这其实是建立了函数的局部变量、寄存器保存的空间。此时的栈结构如下所示：



---

到此时为止，`ebp` 寄存器处于函数调用关系链中一个非常重要的地位。`ebp` 寄存器中存储着栈中的一个地址（栈帧分界处），此地址是“老”`ebp` 入栈后的栈顶。那么以该地址为基准，向高地址方向（即栈底方向）能获取返回地址、参数值，向低地址方向（栈顶方向）能获取函数局部变量值，而该地址处又存放着上一层函数调用时的 `ebp` 值。由于 `ebp` 中的地址总是“上一层函数调用时的 `ebp` 值”，这样，就能通过把 `ebp` 的内容作为寻找上一个调用函数的栈帧的指针，如此形成递归，直至到达栈底。这就可找到整个的函数调用栈。这也是 `kdebug.c` 中 `print_stackframe` 函数的实现内容。

## 3.4 可管理中断并处理中断方式 I/O 的 ucore

### 3.4.1 实验目标

前面的 `project` 都没有引入中断机制，所以 `bootloader` 和 `ucore` 都是正常地顺序执行，不会受到外界（比如外设）的“干扰”。虽然实现简单，但无法解决上述问题。我们需要扩展 `ucore` 的功能，让 `ucore` 能够支持中断，这需要读者了解基本的 80386 硬件中断机制，对保护模式有更深入的了解；需要清楚在中断的处理过程中，硬件主动完成了什么事情，软件在硬件完成的基础上又要完成哪些事情。通过学习和实践，读者可以了解清楚上述问题，并进一步知道通过操作系统的中断处理例程（Interrupt Process Routine, IPR）完成设备请求处理的方法等。

### 3.4.2 proj4 概述

#### 1. 实现描述

`proj4` 建立在 `proj3.1` 的基础上，实现了一个通过中断机制完成设备（键盘、串口和时钟）中断请求处理的 `ucore`。简单地说 `proj4` 扩展与中断相关的工作有两个，一个是初始化中断，涉及初始化中断控制器 8259A（打通外设与 CPU 的通路）和中断门描述符表（建立外设中断与中断服务例程的联系）和各种外设。以 `proj4` 的 `ucore` 为例，操作系统内核启动以后，`kern_init` 函数（`kern/init/init.c`）通过调用 `pic_init` 函数完成对中断控制器的初始化工作，调用 `idt_init` 函数完成了对整个中断门描述符表的创建，调用 `cons_init` 和 `clock_init` 函数完成对串口、键盘和时钟外设的中断初始化工作。

`ucore` 的另一个重要工作是中断服务，即收到中断后，对中断进行处理的中断服务例程（比如收到 100 个时钟中断后，显示一个字符串“100 ticks”）等。这主要集中在 `vectors.S`（包括 256 个中断服务例程的入口地址和第一步初步处理实现）、`trapentry.S`（紧接着第一步初步处理后，进一步完成第二步初步处理的实现以及中断处理完毕后的返回准备工作）和 `trap.c` 中（紧接着第二步初步处理后，继续完成具体的各种中断处理操作）。

## 2. 项目组成

proj4 整体目录结构如下所示：

```
proj4
|-- kern
|   |-- driver
|   |   |-- clock.c
|   |   |-- clock.h
|   |   |-- console.c
|   |   |-- console.h
|   |   |-- picirq.c
|   |   `-- picirq.h
|   |-- init
|   |   `-- init.c
|   |-- mm
|   |   |-- memlayout.h
|   |   `-- mmu.h
|   `-- trap
|       |-- trap.c
|       |-- trapentry.S
|       |-- trap.h
|       `-- vectors.S
|-- tools
|   `-- vector.c
.....
```

proj4 是基于 proj3.1（会在内置监控自身运行状态的 ucore 一节中进一步说明）进一步扩展完成的。相对于 proj3.1，增加了大约 10 个文件，相关增加和改动主要集中在 kern/driver 和 kern/trap 目录下，使得 ucore 具有外设中断处理功能，这一个比较大的跨越。主要增加和修改的文件如下所示：

- tools/vector.c：生成 vectors.S，此文件包含了中断向量处理的统一实现。
- kern/driver/intr.[ch]：实现了通过设置 CPU 的 eflags 来屏蔽和使能中断的函数；
- kern/driver/picirq.[ch]：实现了对中断控制器 8259A 的初始化和使能操作；
- kern/driver/clock.[ch]：实现了对时钟控制器 8253 的初始化操作；
- kern/driver/console.[ch]：实现了对串口和键盘的中断方式的处理操作；
- kern/trap/vectors.S：包括 256 个中断服务例程的入口地址和第一步初步处理实现；
- kern/trap/trapentry.S：紧接着第一步初步处理后，进一步完成第二步初步处理；并且有恢复中断上下文的处理，即中断处理完毕后的返回准备工作；
- kern/trap/trap.[ch]：紧接着第二步初步处理后，继续完成具体的各种中断处理操作；

### 3. 编译运行

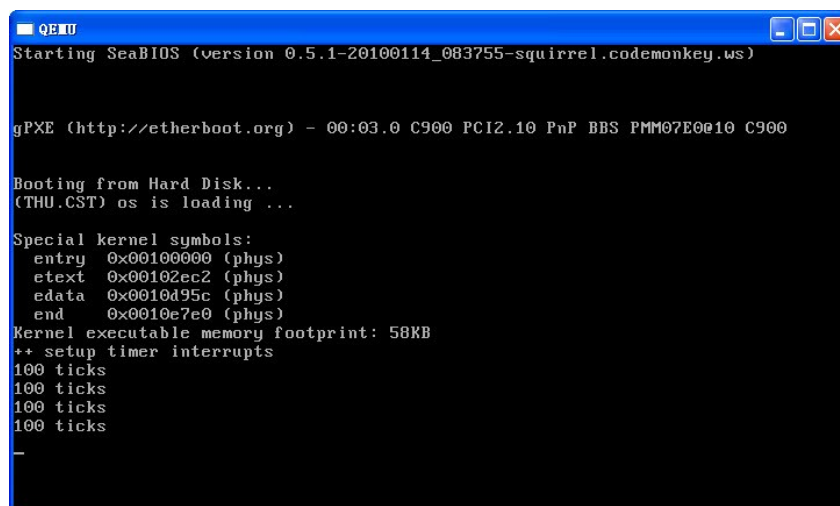
编译运行

编译并运行 proj4 的命令如下：

make

make qemu

则可以得到如下显示界面



```
QEMU
Starting SeaBIOS (version 0.5.1-20100114_083755-squirrel.codemonkey.ws)

gPXE (http://etherboot.org) - 00:03.0 C900 PCI2.10 PnP BBS PMM07E0e10 C900

Booting from Hard Disk...
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x00102ec2 (phys)
  edata  0x0010d95c (phys)
  end    0x0010e7e0 (phys)
Kernel executable memory footprint: 58KB
** setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
-
```

通过上图可以看到时钟中断已经能够正常相应，每隔 100 个时钟中断会显示一次“100 ticks”的信息。一个简单的显示信息的背后蕴藏着中断处理的复杂实现。下面我们将从中断基本概念、中断控制器、保护模式的中断处理机制等方面来分析上图中背后的东西。

#### 3.4.3 【背景】理解 CPU 对外设中断的硬件支持

操作系统需要对计算机系统中的各种外设进行管理，这就需要 CPU 和外设能够相互通信才行。一般外设的速度远慢于 CPU 的速度。如果让操作系统通过 CPU“主动关心”外设的事件，即采用通常的轮询(polling)机制，则太浪费 CPU 资源了。所以需要操作系统和 CPU 能够一起提供某种机制，让外设在需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后在恢复操作系统和应用的正常执行。在操作系统中，这种机制称为中断机制。中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石。但中断引入的不确定性和异步性导致了设计和实现操作系统更加困难。

本章只描述保护模式下的中断处理过程。当 CPU 收到外设中断（可通过可编程中断控制器芯片 8259A 发给 CPU 中断信息）、CPU 自身产生的故障（Fault）或 CPU 自身“有意”产生的陷阱（trap）时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个事件的相关处理例程中，在完成对这个事件的处理后再跳回到刚才被打断的程序或任务中。



中断向量和中断服务例程的对应关系主要是由 IDT（中断门描述符表）来描述。操作系统在 IDT 中设置好各种中断向量对应的中断描述符，而中断描述符指出了中断服务例程的起始地址，留待 CPU 在产生中断后查询对应中断服务例程的起始地址。而 IDT 本身的起始地址保存在 IDTR 寄存器中。

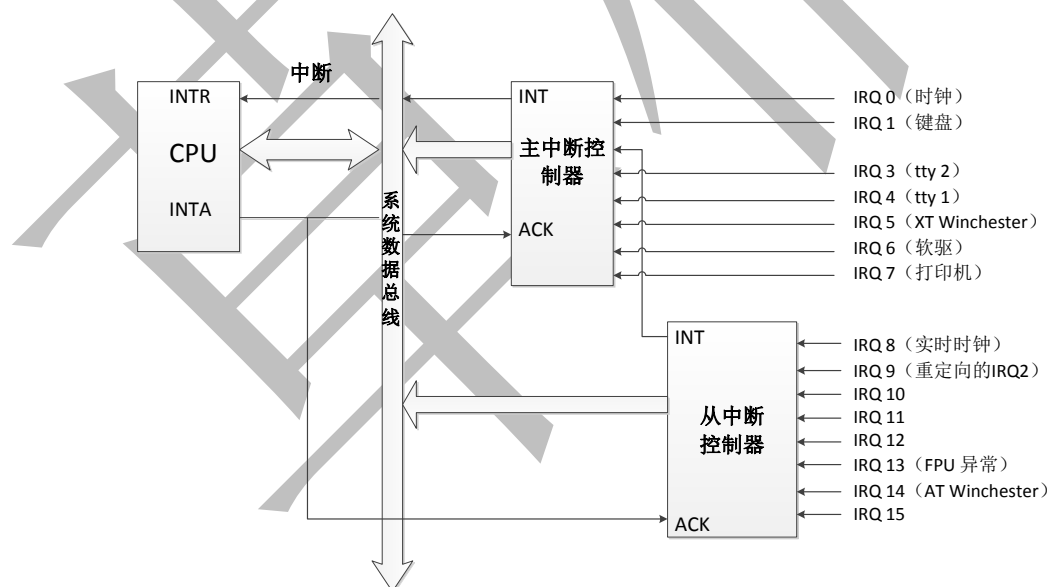
80386 共支持 256 种中断，其中故障（Fault）和陷阱(Trap)由 CPU 自身产生，不使用中断控制器，也不能被屏蔽。外设中断又分为可屏蔽中断（INTR）和非屏蔽中断（NMI），I/O 设备产生的中断请求（IRQ）引起可屏蔽中断，而紧急的外设事件（如掉电故障）引起的中断事件引起非屏蔽中断。

非屏蔽中断和异常的编号是固定的，而屏蔽中断的编号可以通过对中断控制器的编程来调整。256 个中断的分配如下：

- 0~31 号的中断对应于故障、陷阱和非屏蔽外设中断。
- 32~47 号的中断分配给可屏蔽外设中断。
- 48~255 号的中断可以用软件来设置。比如 ucore 可用其中的一个中断号来实现系统调用。

## 1. 外设可屏蔽中断

80386 通过两片中断控制器 8259A 来响应 15 个外中断源，每个 8259A 可管理 8 个中断源。第一级（称主片）的第二个中断请求输入端，与第二级 8259A（称从片）的中断输出端 INT 相连，如下图所示。IRQ 号和中断号之间的映射关系可以通过中断控制器来调整。



级联的 8259A 架构

在中断产生过程中，中断控制器 8259A 监视外设产生的中断请求（IRQ）信号，如果外设产生了一个中断请求信号，则 8259A 执行如下操作：

- 把接受到的 IRQ 信号转换成一个对应的中断编号；
- 把这个中断编号值存放在中断控制器的一个 I/O 地址单元中，CPU 通过数据/地址总线可访问到此 I/O 地址单元；

- c) 给 CPU 的 INTR 引脚触发信号，即发出一个中断；
- d) 等待直到 CPU 通过 INTA 引脚确认这个中断信号，清除 INTR 引脚上的触发信号。

屏蔽外部 I/O 请求有两种方法。一种是从 CPU 的角度清零 CPU 的 EFLAG 的中断标志位(IF)；另一种是从中断控制器的角度，即通过把中断控制器中的中断屏蔽寄存器（IMR）相应位置 1，则表示禁用某条中断线。

## 2. 陷阱、故障和非屏蔽中断

陷阱和故障是 CPU 内部执行指令的过程中产生的中断事件。非屏蔽中断就是计算机内部硬件出错时引起的紧急故障情况。80386 处理器发布了大约 20 种陷阱、故障或非屏蔽中断。在某些故障产生时，CPU 会产生一个硬件错误码并压入内核栈中。

在下表中给出了在实验中可能碰到的 80386 中陷阱的中断号、名称、类别及简单描述。更多的信息可以在 Intel 的技术文档中找到。

表 ucore 中异常的简单描述

| 中断号 | 名称   | 类别 | 简单描述                 |
|-----|------|----|----------------------|
| 8   | 双重故障 | 故障 | 在处理故障中又产生了故障         |
| 11  | 段不存在 | 故障 | 访问一个不存在的段            |
| 12  | 栈段异常 | 故障 | 超过栈段界限，或由 ss 标识的段不存在 |
| 13  | 通用保护 | 故障 | 违反了保护模式下的某种保护规则      |
| 14  | 页异常  | 故障 | 页不在内存，或违反了一种分页保护机制   |

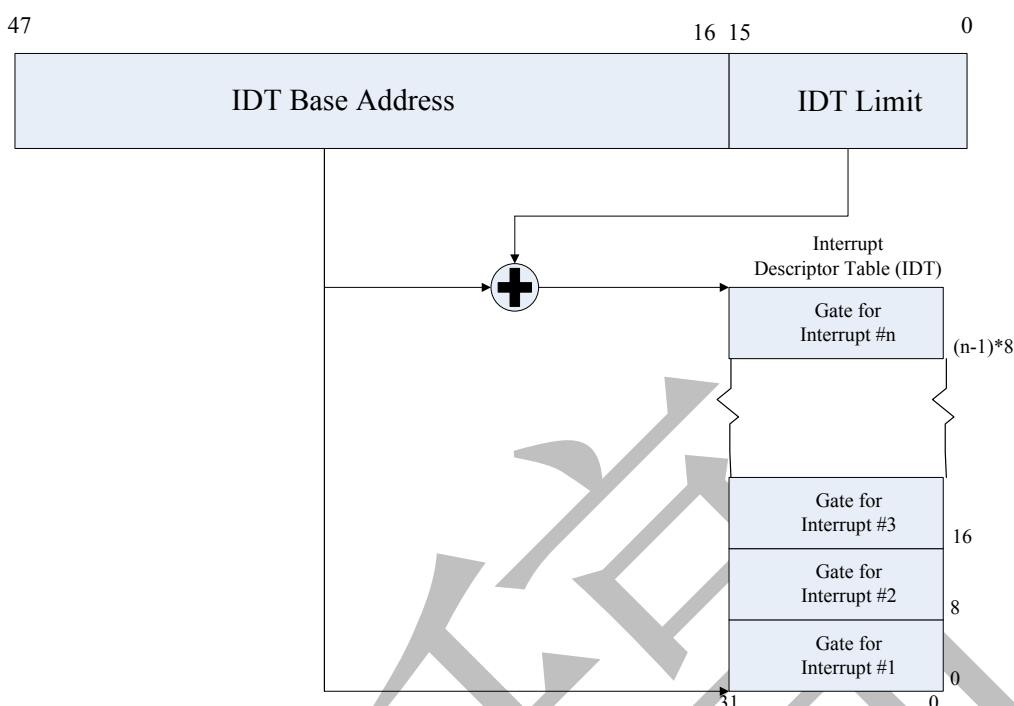
## 3. 中断门描述符表（Interrupt Descriptor Table）

中断门描述符表把每个中断或异常编号和一个指向中断服务例程的描述符联系起来。同 GDT 一样，IDT 是一个 8 字节的描述符数组，但 IDT 的第一项可以包含一个描述符。CPU 把中断（异常）号乘以 8 做为 IDT 的索引。IDT 可以位于内存的任意位置，CPU 通过 IDT 寄存器（IDTR）的内容来寻址 IDT 的起始地址。指令 LIDT 和 SIDT 用来操作 IDTR。两条指令都有一个显示的操作数：一个 6 字节表示的内存地址。指令的含义如下：

- **LIDT（Load IDT Register）指令：**使用一个包含线性地址基址和界限的内存操作数来加载 IDT。操作系统创建 IDT 时需要执行它来设定 IDT 的起始地址。这条指令只能在特权级 0 执行。
- **SIDT（Store IDT Register）指令：**拷贝 IDTR 的基址和界限部分到一个内存地址。这条指令可以在任意特权级执行。

IDT 和 IDTR 寄存器的结构和关系如下图所示：

## IDT Register



在保护模式下，最多会存在 256 个 Interrupt/Exception Vectors。范围[0, 31]内的 32 个向量被故障中断和 NMI（不可屏蔽）中断使用，但当前并非所有这 32 个向量都被使用，有几个当前没有被使用。范围[32, 255]内的向量被保留给用户定义的中断，可将它们用作外部 I/O 设备中断（8259A IRQ），或者系统调用（System Call、Software Interrupts）等。

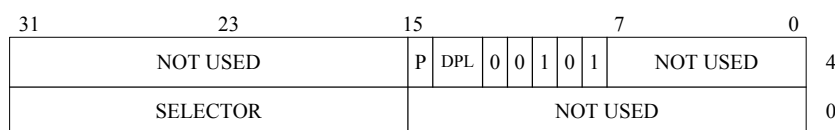
### 4. 门描述符（Gate Descriptors）

在保护模式下，中断门描述符表（IDT）中的每个表项由 8 个字节组成，其中的每个表项叫做一个门描述符（Gate Descriptor），“门”的含义是指当中断发生时必须先访问这些“门”，能够“开门”（即将要进行的处理需通过特权检查，符合设定的权限等约束）后，然后才能进入相应的处理程序。而门描述符则描述了“门”的属性（如特权级、段内偏移量等）。在 IDT 中，可以包含如下 3 种类型的系统段描述符：

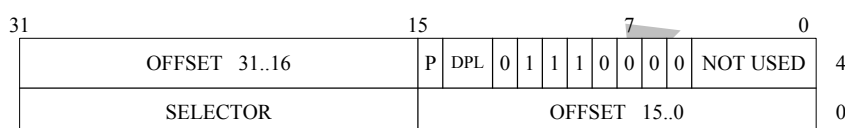
- 中断门描述符（Interrupt-gate descriptor）：用于中断处理，其类型码为 110，中断门包含了一个外设中断或故障中断的处理程序所在段的选择子和段内偏移量。当控制权通过中断门进入中断处理程序时，处理器清 IF 标志，即关中断，以避免嵌套中断的发生。中断门中的 DPL（Descriptor Privilege Level）为 0，因此用户态的进程不能访问中断门。所有的中断处理程序都由中断门激活，并全部限制在内核态。
- 陷阱门描述符（Trap-gate descriptor）：用于系统调用，其类型码为 111，与中断门类似，其唯一的区别是，控制权通过陷阱门进入处理程序时维持 IF 标志位不变，也就是说，不关中断。
- 任务门描述符（Task-gate descriptor）和调用门描述符（Call-gate descriptor）：这两种主要是 Intel 设置的“任务”切换的手段，在本书中暂时没有使用。

下图图显示了 80386 的中断门描述符、陷阱门描述符的格式：

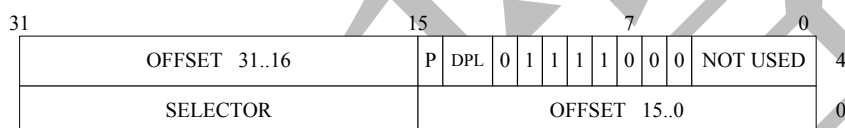
80386 TASK GATE



80386 INTERRUPT GATE



80386 TRAP GATE



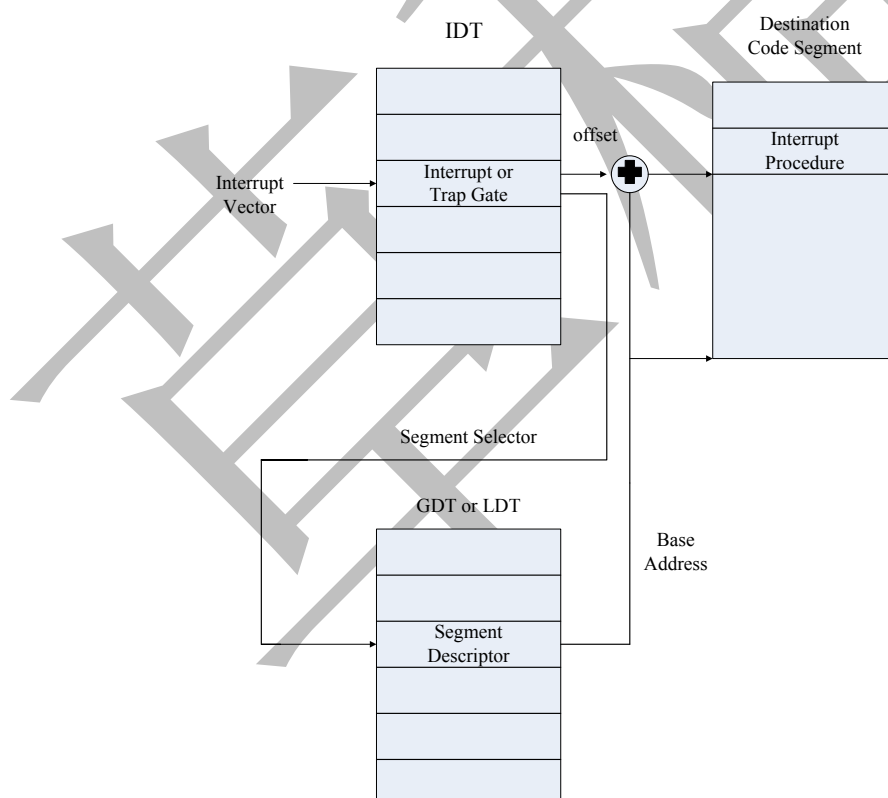
## 5. 中断处理中硬件负责完成的工作

中断服务例程包括具体负责处理中断（异常）的代码是操作系统的重要组成部分。需要注意区别的是，有两个过程由硬件完成：

- 硬件中断处理过程 1（起始）：从 CPU 收到中断事件后，打断当前程序或任务的执行，根据某种机制跳转到中断服务例程去执行的过程。其具体流程如下：
  - 1) CPU 在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如 8259A）是否发送中断请求过来，如果有那么 CPU 就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量；
  - 2) CPU 根据得到的中断向量（以此为索引）到 IDT 中找到该向量对应的中断描述符，中断描述符里保存着中断服务例程的段选择子；
  - 3) CPU 使用 IDT 查到的中断服务例程的段选择子从 GDT 中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，段描述符的基址+中断描述符中的偏移地址形成了中断服务例程的起始地址；
  - 4) CPU 会根据 CPL 和中断服务例程的段描述符的 DPL 信息确认是否发生了特权级的转换。比如当前应用程序正运行在用户态，而中断服务例程是运行在内核态的，则意味着发生了特权级的转换，这时 CPU 会从当前应用程序的 TSS 信息（该信息在内存中的起始地址存在 TR 寄存器中）里取得该程序的内核栈地址，即包括内核态的 ss 和 esp 的值，并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的用户态的 ss 和 esp 压到新的内核栈中保存起来；如果当前程序运行在内核态，则不会发生特权转移

- 5) CPU 需要开始保存当前被打断的用户态程序的现场（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的 `eflags`, `cs`, `eip`, `errorCode`（如果是有错误码的异常）信息；
  - 6) CPU 把中断服务例程的地址加载到 `cs` 和 `eip` 寄存器中，开始执行中断服务例程。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。
- 硬件中断处理过程 2（结束）：每个中断服务例程在有中断处理工作完成后需要通过 `iret`（或 `iretd`）指令恢复被打断的程序的执行。CPU 执行 `IRET` 指令的具体过程如下：
    - 1) 程序执行这条 `iret` 指令时，首先会从内核栈里弹出先前保存的被打断的程序的现场信息，即 `eflags`, `cs`, `eip` 重新开始执行；
    - 2) 如果存在特权级转换（从内核态转换到用户态），则还需要从内核栈中弹出用户态栈的 `ss` 和 `esp`，这样也意味着栈也被切换回原先使用的用户态的栈了；
    - 3) 如果此次处理的是带有错误码（`errorCode`）的异常，CPU 在恢复先前程序的现场时，并不会弹出 `errorCode`。这一步需要通过软件完成，即要求相关的中断服务例程在调用 `iret` 返回之前添加出栈代码主动弹出 `errorCode`。

下图显示了从中断向量到 GDT 中相应中断服务程序起始位置的定位方式：



## 6. 中断处理的特权级转换

中断处理的特权级转换是通过**门描述符**（`gate descriptor`）和**相关指令**来完成的。一个门描述符就是一个系统类型的段描述符，一共有 4 个子类型：调用门描述符（`call-gate`

descriptor)，中断门描述符 (interrupt-gate descriptor)，陷阱门描述符 (trap-gate descriptor) 和任务门描述符 (task-gate descriptor)。与中断处理相关的是中断门描述符和陷阱门描述符。这些门描述符被存储在中断门描述符表 (Interrupt Descriptor Table, 简称 IDT) 当中。CPU 把中断向量作为 IDT 表项的索引, 用来指出当中断发生时使用哪一个门描述符来处理中断。中断门描述符和陷阱门描述符几乎是一样的。中断发生时实施特权检查的过程如下图所示:

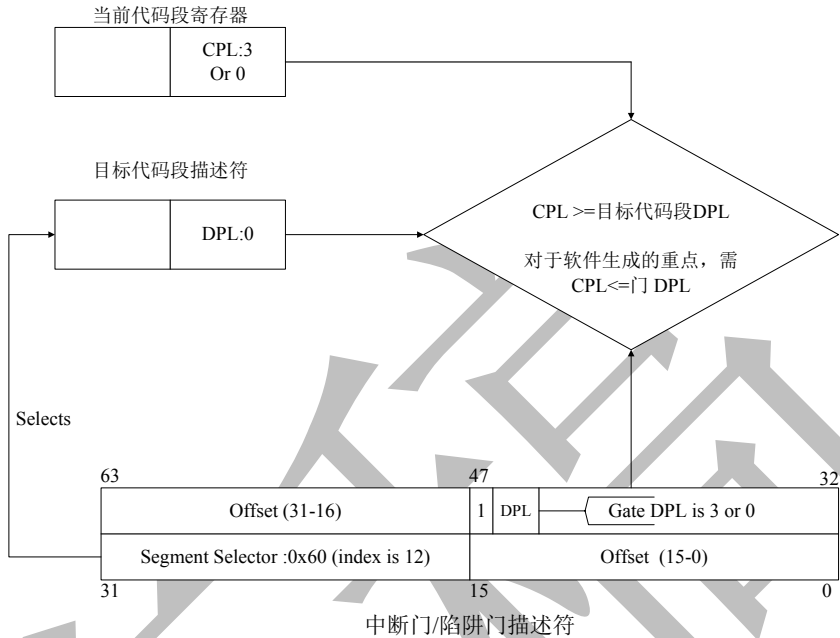


图 中断发生时实施特权检查的过程

门中的 DPL 和段选择符一起控制着访问, 同时, 段选择符结合偏移量 (Offset) 指出了中断处理例程的入口点。内核一般在门描述符中填入内核代码段的段选择子。产生中断后, CPU 一定不会将运行控制从高特权级转向低特权级, 特权级必须要么保持不变 (当操作系统内核自己被中断的时候), 或被提升 (当用户态程序被中断的时候)。无论哪一种情况, 作为结果的 CPL 必须等于目的代码段的 DPL。如果 CPL 发生了改变 (比如从用户态到内核态), 一个栈切换操作 (通过 TSS 完成) 就会发生。如果中断是被用户态程序中的指令所触发的 (比如软件执行 INT n 产生的中断), 还会增加一个额外的检查: 门的 DPL 必须具有与 CPL 相同或更低的特权。这就防止了用户代码随意触发中断。如果这些检查失败, 就会产生一个一般保护异常 (general-protection exception)。

### 3.4.4 【实现】初始化中断控制器

80386 把中断号 0~31 分配给陷阱、故障和非屏蔽中断, 而把 32~47 之间的中断号分配给可屏蔽中断。可屏蔽中断的中断号是通过中断控制器的编程来设置的。下面描述了对 8259A 中断控制器初始化过程。

8259A 通过两个 I/O 地址来进行中断相关的数据传送, 对于单个的 8259A 或者是两级级联中的主 8259A 而言, 这两个 I/O 地址是 0x20 和 0x21。对于两级级联的从 8259A 而言, 这两个 I/O 地址是 0xA0 和 0xA1。8259A 有两种编程方式, 一是初始化方式, 二是工作方式。在操作系统启动时, 需要对 8259A 做一些初始化工作, 即实现 8259A 的初始化方式编程。8259A 中的四个中断命令字 (ICW) 寄存器用来完成初始化编程, 其含义如下:

- ICW1: 初始化命令字。
- ICW2: 中断向量寄存器, 初始化时写入高五位作为中断向量的高五位, 然后在中断响应时由 8259 根据中断源 (哪个管脚) 自动填入形成完整的 8 位中断向量 (或叫中断类型号)。
- ICW3: 8259 的级联命令字, 用来区分主片和从片。
- ICW4: 指定中断嵌套方式、数据缓冲选择、中断结束方式和 CPU 类型。

8259A 初始化的过程就是写入相关的命令字, 8259A 内部储存这些命令字, 以控制 8259A 工作。有关的硬件可看附录补充资料。这里只把 ucore 对 8259A 的初始化过程 (在 picirq.c 中的 pic\_init 函数实现) 描述一下:

```
//此时系统尚未初始化完毕, 故屏蔽主从8259A的所有中断
    outb(IO_PIC1 + 1, 0xFF);
    outb(IO_PIC2 + 1, 0xFF);
// 设置主 8259A 的 ICW1, 给 ICW1 写入 0x11, 0x11 表示 (1) 外部中断请求信号为上升沿触发有效, (2)
// 系统中有多片 8259A 级联, (3) 还表示要向 ICW4 送数据
// ICW1设置格式为: 0001g0hi
//   g: 0 = edge triggering, 1 = level triggering
//   h: 0 = cascaded PICs, 1 = master only
//   i: 0 = no ICW4, 1 = ICW4 required
    outb(IO_PIC1, 0x11);

// 设置主 8259A 的 ICW2: 给 ICW2 写入 0x20, 设置中断向量偏移值为 0x20, 即把主 8259A 的 IRQ0-7 映
// 射到向量 0x20-0x27
    outb(IO_PIC1 + 1, IRQ_OFFSET);

// 设置主 8259A 的 ICW3: ICW3 是 8259A 的级联命令字, 给 ICW3 写入 0x4, 0x4 表示此主中断控制器的
// 第 2 个 IR 线 (从 0 开始计数) 连接从中断控制器。
    outb(IO_PIC1 + 1, 1 << IRQ_SLAVE);

//设置主8259A的ICW4: 给ICW4写入0x3, 0x3表示采用自动EOI方式, 即在中断响应时, 在8259A送出中断
// 矢量后, 自动将ISR相应位复位; 并且采用一般嵌套方式, 即当某个中断正在服务时, 本级中断及更低级
// 的中断都被屏蔽, 只有更高的中断才能响应。
// ICW4设置格式为: 000nbmap
//   n: 1 = special fully nested mode
//   b: 1 = buffered mode
//   m: 0 = slave PIC, 1 = master PIC
//       (ignored when b is 0, as the master/slave role
//       can be hardwired).
//   a: 1 = Automatic EOI mode
//   p: 0 = MCS-80/85 mode, 1 = intel x86 mode
    outb(IO_PIC1 + 1, 0x3);

//设置从8259A的ICW1: 含义同上
    outb(IO_PIC2, 0x11); // ICW1
//设置从8259A的ICW2: 给ICW2写入0x28, 设置从8259A的中断向量偏移值为0x28
```

```

        outb(IO_PIC2 + 1, IRQ_OFFSET + 8);    // ICW2
//0x2 表示此从中断控制器链接主中断控制器的第 2 个 IR 线
        outb(IO_PIC2 + 1, IRQ_SLAVE);    // ICW3
//设置主8259A的ICW4: 含义同上
        outb(IO_PIC2 + 1, 0x3);    // ICW4

//设置主从8259A的OCW3: 即设置特定屏蔽位（值和英文解释不一致），允许中断嵌套；不查询；将读入
其中断请求寄存器IRR的内容
        // OCW3设置格式为: 0ef01prs
        //  ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
        //    p:  0 = no polling, 1 = polling mode
        //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
        outb(IO_PIC1, 0x68);    // clear specific mask
        outb(IO_PIC1, 0x0a);    // read IRR by default

        outb(IO_PIC2, 0x68);    // OCW3
        outb(IO_PIC2, 0x0a);    // OCW3
//初始化完毕，使能主从8259A的所有中断
        if (irq_mask != 0xFFFF) {
            pic_setmask(irq_mask);
        }

```

### 3.4.5 【实现】初始化中断门描述符表

ucore 操作系统如果要正确处理各种不同的中断事件，就需要安排应该由哪个中断服务例程负责处理特定的中断事件。系统将所有的中断事件统一进行了编号（0~255），这个编号称为中断号或中断向量。

为了完成中断号和中断服务例程起始地址的对应关系，首先需要建立 256 个中断处理例程的入口地址。为此，通过一个 C 程序 `tools/vector.c` 生成了一个文件 `vectors.S`，在此文件中的 `__vectors` 地址处开始处连续存储了 256 个中断处理例程的入口地址数组，且在此文件中的每个中断处理例程的入口地址处，实现了中断处理过程的第一步初步处理。

有了中断服务例程的起始地址，就可以建立对应关系了，这部分的实现在 `trap.c` 文件中的 `idt_init` 函数中实现：

```

//全局变量：中断门描述符表
static struct gatedesc idt[256] = {{0}};

.....

void idt_init(void) {
//保存在vectors.S中的256个中断处理例程的入口地址数组
    extern uint32_t __vectors[];

    int i;

//在中断门描述符表中通过建立中断门描述符，其中存储了中断处理例程的代码段GD_KTEXT和偏移量

```



```

__vectors[i], 特权级为DPL_KERNEL。这样通过查询idt[i]就可定位到中断服务例程的起始地址。

    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
//建立好中断门描述符表后，通过指令lidt把中断门描述符表的起始地址装入IDTR寄存器中，从而完成中段
描述符表的初始化工作。

    lidt(&idt_pd);
}

```

### 3.4.6 【实现】外设的相关中断初始化

串口的初始化函数 `serial_init`（位于 `/kern/driver/console.c`）中涉及中断初始化工作的很简单：

```

.....
// 使能串口1接收字符后产生中断
    outb(COM1 + COM_IER, COM_IER_RDI);
.....
// 通过中断控制器使能串口 1 中断
    pic_enable(IRQ_COM1);

```

键盘的初始化函数 `kbd_init`（位于 `kern/driver/console.c` 中）完成了对键盘的中断初始化工作，具体操作更加简单：

```

.....
// 通过中断控制器使能键盘输入中断
    pic_enable(IRQ_KBD);

```

时钟是一种有着特殊作用的外设，其作用并不仅仅是计时。在后续章节中将讲到，正是由于有了规律的时钟中断，才使得无论当前 CPU 运行在哪里，操作系统都可以在预先确定的时间点上获得 CPU 控制权。这样当一个应用程序运行了一定时间后，操作系统会通过时钟中断获得 CPU 控制权，并可把 CPU 资源让给更需要 CPU 的其他应用程序。时钟的初始化函数 `clock_init`（位于 `kern/driver/clock.c` 中）完成了对时钟控制器 8253 的初始化：

```

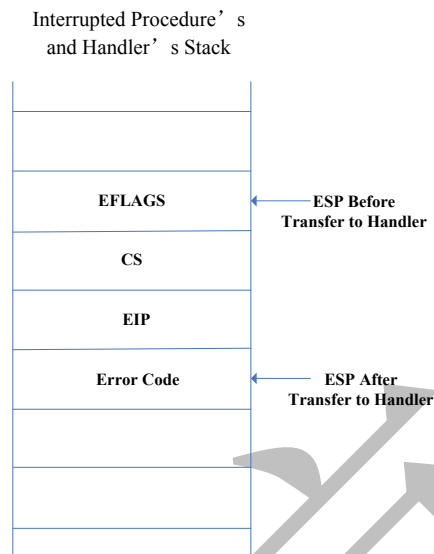
.....
//设置时钟每秒中断100次
    outb(IO_TIMER1, TIMER_DIV(100) % 256);
    outb(IO_TIMER1, TIMER_DIV(100) / 256);
// 通过中断控制器使能时钟中断
    pic_enable(IRQ_TIMER);

```

### 3.4.7 【实现】中断处理过程

当中断产生后，首先硬件要完成一系列的工作（如小节“中断处理中硬件负责完成的工作”所描述的“硬件中断处理过程 1（起始）”内容），由于中断发生在内核态执行过程中，所以特权级没有变化，所以 CPU 在跳转到中断处理例程之前，还会在内核栈中依次压入错误码（可选）、EIP、CS 和 EFLAGS，下图显示了在相同特权级下中断产生后的栈变化示意图：

## 统一特权级下中断产生后的栈变化



然后 CPU 就跳转到 IDT 中记录的中断号  $i$  所对应的中断服务例程入口地址处继续执行。`vector.S` 文件中定义了每个中断的中断处理例程的入口地址 (保存在 `vectors` 数组中)。其中, 中断可以分成两类: 一类是压入错误编码的 (error code), 另一类不压入错误编码。对于第二类, `vector.S` 自动压入一个 0。此外, 还会压入相应中断的中断号。在内核栈中压入一个或两个必要的参数之后, 都会跳转到统一的入口 `__alltraps` 处 (位于 `trapentry.S` 中) 继续执行。

CPU 从 `__alltraps` 处开始, 在栈中按照 `trapframe` 结构压入各个寄存器, 此时内核栈的结构如下所示:

```
uint32_t reg_edi;
uint32_t reg_esi;
uint32_t reg_ebp;
uint32_t reg_oesp;      /* Useless */
uint32_t reg_ebx;
uint32_t reg_edx;
uint32_t reg_ecx;
uint32_t reg_eax;
uint16_t tf_es;
uint16_t tf_padding1;
uint16_t tf_ds;
uint16_t tf_padding2;
uint32_t tf_trapno;
/* below here defined by x86 hardware */
uint32_t tf_err;
uintptr_t tf_eip;
uint16_t tf_cs;
uint16_t tf_padding3;
uint32_t tf_eflags;
```

此时, 为了将来能够恢复被打断的内核执行过程所需的寄存器内容都保存好了。为了正

确进行中断处理，把 DS 和 ES 寄存器设置为 GD\_KDATA，这是为了预防从用户态产生的中断(当然，到目前为止，ucore 都在内核态执行，还不会发生这种情况)。把刚才保存的 trapframe 结构的起始地址（即当前 SP 值）压栈，然后调用 trap 函数（定义在 trap.c 中），就开始了  
对具体中断的处理。trap 进一步调用 trap\_dispatch 函数，完成对具体中断的处理。在相应的  
处理过程结束以后，trap 将会返回，在\_\_trapret:中，完成对返回前的寄存器和栈的回复准备  
工作，最后通过 iret 指令返回到中断打断的地方继续执行。整个中断处理流程大致如下：

| trapasm.S                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | trap.c                                                                                                                           |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|-----|--------------|-----|--|------------|-------|---------|-------|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|--|
| <p>1)产生中断后，CPU 跳转到相应的中断处理入口 (vectors)，并在栈中压入相应的 error_code（是否存在与异常号相关）以及 trap_no，然后跳转到 alltraps 函数入口：</p> <p>注意：此处的跳转是 jmp 过程</p> <table><tr><td>(high)</td><td>...</td></tr><tr><td>产生中断时的 eip →</td><td>eip</td></tr><tr><td></td><td>error_code</td></tr><tr><td>esp →</td><td>trap_no</td></tr><tr><td>(low)</td><td>..</td></tr></table> <p>在栈中保存当前被打断程序的 trapframe 结构(参<br/>见过程 trapasm.S)。设置 kernel (内核) 的数据段寄存<br/>器，最后压入 esp，作为 trap 函数参数(struct<br/>trapframe *tf) 并跳转到中断处理函数 trap 处：</p> <p>注意：此时的跳转是 call 调用，会压入返回地址<br/>eip，注意区分此处 eip 与 trapframe 中 eip：</p> <p>trapframe 的结构为：</p> <table><tr><td>Struct trapframe<br/>{<br/>uint edi;<br/>uint esi;<br/>uint ebp;<br/>...<br/>ushort es;<br/>ushort padding1;<br/>ushort ds;<br/>ushort padding2;<br/>uint trapno;<br/>uint err;<br/>uint eip;<br/>...<br/>}</td><td>观察 trapframe 结构与<br/>中断产生过程的压栈顺<br/>序。<br/>需要明确 pushal 指令<br/>都保存了哪些寄存器，按<br/>照什么顺序？<br/><br/>← trap_no<br/>← trap_error<br/>← 产生中断处的 eip</td></tr></table> | (high)                                                                                                                           | ... | 产生中断时的 eip → | eip |  | error_code | esp → | trap_no | (low) | .. | Struct trapframe<br>{<br>uint edi;<br>uint esi;<br>uint ebp;<br>...<br>ushort es;<br>ushort padding1;<br>ushort ds;<br>ushort padding2;<br>uint trapno;<br>uint err;<br>uint eip;<br>...<br>} | 观察 trapframe 结构与<br>中断产生过程的压栈顺<br>序。<br>需要明确 pushal 指令<br>都保存了哪些寄存器，按<br>照什么顺序？<br><br>← trap_no<br>← trap_error<br>← 产生中断处的 eip |  |
| (high)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | ...                                                                                                                              |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |
| 产生中断时的 eip →                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | eip                                                                                                                              |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | error_code                                                                                                                       |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |
| esp →                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | trap_no                                                                                                                          |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |
| (low)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | ..                                                                                                                               |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |
| Struct trapframe<br>{<br>uint edi;<br>uint esi;<br>uint ebp;<br>...<br>ushort es;<br>ushort padding1;<br>ushort ds;<br>ushort padding2;<br>uint trapno;<br>uint err;<br>uint eip;<br>...<br>}                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | 观察 trapframe 结构与<br>中断产生过程的压栈顺<br>序。<br>需要明确 pushal 指令<br>都保存了哪些寄存器，按<br>照什么顺序？<br><br>← trap_no<br>← trap_error<br>← 产生中断处的 eip |     |              |     |  |            |       |         |       |    |                                                                                                                                                                                               |                                                                                                                                  |  |

|                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>} </pre>                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                             |
| <p>进入 <code>trap</code> 函数，对中断进行相应的处理：</p> <p>3)结束 <code>trap</code> 函数的执行后，通过 <code>ret</code> 指令返回到 <code>alltraps</code> 执行过程。</p> <p>从栈中恢复所有寄存器的值。</p> <p>调整 <code>esp</code> 的值：跳过栈中的 <code>trap_no</code> 与 <code>error_code</code>，使 <code>esp</code> 指向中断返回 <code>eip</code>，通过 <code>iret</code> 调用恢复 <code>cs</code>、<code>eflag</code> 以及 <code>eip</code>，继续执行。</p> | <p>2)详细的中断分类以及处理流程如下：</p> <p>根据中断号对不同的中断进行处理。其中，若中断号是 <code>IRQ_OFFSET + IRQ_TIMER</code> 为时钟中断，则把 <code>ticks</code> 将增加一。</p> <p>若中断号是 <code>IRQ_OFFSET + IRQ_COM1</code> 为串口中断，则显示收到的字符。</p> <p>若中断号是 <code>IRQ_OFFSET + IRQ_KBD</code> 为键盘中断，则显示收到的字符。</p> <p>若为其他中断且产生在内核状态，则挂起系统；</p> |

## 3.5 可在内核态和用户态之间进行切换的 ucore

在操作系统原理中，一直强调操作系统运行在内核态（特权态），应用程序运行在用户态（非特权态）。但为什么说处于用户态的应用程序就不能访问内核态的数据，而内核态的操作系统可以访问用户态的数据？我们有没有一个 `project` 来体验内核态和用户态的区别是什么？更进一步体验如何在内核态和用户态之间进行切换呢？`project4.1.1` 为此进行了尝试。

### 3.5.1 实验目标

通过学习和实践，读者可以了解如何 `CPU` 处不同特权态下的执行特点和限制，理解如何从内核态切换到用户态，以及如何从用户态切换到内核态。

### 3.5.2 proj4.1.1 概述

#### 1. 实现描述

`proj4.1.1` 建立在 `proj4.1`（当然是基于 `proj4`）基础之上，主要完成了用户态（非特权态）与内核态相互切换的过程。相对于 `proj4`，主要增加了两部分工作，一部分是从用户态返回内核态的准备工作，即建立任务段（`Task Segment`）和任务段描述符（`SEG_TSS`），设置陷阱中断号（`T_SWITCH_TOK`）和对应的中断处理例程。另外一部分是对内核栈进行各种特殊处理，使得能够完成内核态切换到用户态或用户态切换到内核态的工作。

## 2. 项目组成

这里我们通过 proj4.1.1 来完成此事。proj4.1.1 整体目录结构如下所示：

```
proj4.1.1
|-- kern
|   |-- init
|   |   |-- init.c
|   |-- mm
|   |   |-- memlayout.h
|   |   |-- mmu.h
|   |   |-- pmm.c
|   |   |-- pmm.h
|   |-- trap
|   |   |-- trap.c
|   |   |-- trapentry.S
|   |   |-- trap.h
|   |   |-- vectors.S
|   .....
```

相对于 proj4，改动不多，主要修改和增加的文件如下：

- **memlayout.h**: 定义了全局描述符的索引值和一些段描述符的属性。
- **pmm[ch]**: 为了能够使 CPU 从用户态转换到内核态，在 ucore 初始化时，设置任务段和任务段描述符，重新加载任务段和段描述符表；
- **trap.c**: 设置自定义的陷阱中断 T\_SWITCH\_TOK（用于用户态切换到内核态）和实现对自定义的陷阱中断 T\_SWITCH\_TOK/T\_SWITCH\_TOU 的中断处理例程，使得 CPU 能够在内核态和用户态之间切换。

## 3. 编译运行

编译并运行 proj4.1.1 的命令如下：

```
make
```

```
make qemu
```

则可以得到如下显示界面

```
QEMU
Special kernel symbols:
  entry 0x00100000 (phys)
  etext 0x001031fe (phys)
  edata 0x0010e998 (phys)
  end 0x0010fca0 (phys)
Kernel executable memory footprint: 64KB
++ setup timer interrupts
@ring 0
  cs = 8
  ds = 10
  es = 10
  ss = 10
--- switch to user mode ---
@ring 3
  cs = 1b
  ds = 23
  es = 23
  ss = 23
--- switch to kernel mode ---
@ring 0
  cs = 8
  ds = 10
  es = 10
  ss = 10
```

通过上图，我们可以看到 ucore 在切换到用户态之前，先显示了当前 CPU 的特权级（CS 的最低两位），CS/DS/ES/SS 的值（即对应的段描述符表的索引值），可以看到特权级为 0。根据 lgdt 函数（位于 kern/mm/pmm.c 中）的处理，CS 的值是内核代码段描述符的索引下标，DS/ES/SS 的值是内核数据段描述符的索引下标；而在切换到用户态后，又显示了一下，当前 CPU 的特权级为 3，CS 的值为 1b，DS/ES/SS 的值为 23，把这四个寄存器的值 & 0xfc，则分别为 0x18（SEG\_UTEXT）和 0x20（SEG\_UDATA），说明确实运行在用户态了。在执行了系统调用 T\_SWITCH\_TOK 后，又回到了内核态。下面我们将分析到底发生了什么事情。

### 3.5.3 【背景】分段机制的特权限制

在保护模式下，特权级总共有 4 个，编号从 0（最高特权）到 3（最低特权）。三类主要的资源，即内存、I/O 地址空间以及特权指令需要保护。特权指令如果被用户态的程序所使用，就会受到保护模式的保护机制限制，导致一个故障中断（general-protection exception）。对内存和 I/O 端口的访问存在类似的特权级限制。为了更好地理解不同特权级，这里先介绍三个概念

- CPL：当前特权级（Current Privilege Level）保存在 CS 段寄存器（选择子）的最低两位，CPL 就是当前活动代码段的特权级，并且它定义了当前所执行程序的特权级别）
- DPL：描述符特权（Descriptor Privilege Level）存储在段描述符中的权限位，用于描述对应段所属的特权等级，也就是段本身真正的特权级。
- RPL：请求特权级 RPL(Request Privilege Level) RPL 保存在选择子的最低两位。RPL 说明的是进程对段访问的请求权限，意思是当前进程想要的请求权限。RPL 的值由程序员自己来自由的设置，并不一定  $RPL \geq CPL$ ，但是当  $RPL < CPL$  时，实际起作用的就是 CPL 了，因为访问时的特权检查是判断： $\max(RPL, CPL) \leq DPL$  是否成立，所以 RPL 可以看成是每次访问时的附加限制，RPL=0 时附加限制最小，RPL=3 时附加限制最大。

### 3.5.4 【背景】80386 的任务切换

任务是 80386 硬件描述中的一个名词，在这里我们可以简单地把运行在内核态的 ucore 成为一个任务，把运行在用户态的应用称为另外一任务。任务寄存器(Task Register，简称 TR) 储存了一个 16 位的选择子(对软件可见)，用来索引全局描述符表(GDT)中的一项。TR 对应的描述符描述的一个任务状态段(TSS:Task Status Segment)。

TSS 任务状态段(Task State Segment，简称 TSS)。任务状态段 (TSS) 是位于 GDT 中的一个系统段描述符。任务状态段是做什么的呢？任务状态段就是内存中的一个数据结构。这个结构中保存着和任务相关的信息。当发生任务切换的时候会把当前任务用到的寄存器内容(CS/ EIP/ DS/SS/EFLAGS...)等保存在 TSS 中以便任务切换回来时候继续使用。ucore 根据 80386 硬件手册建立的 TSS 数据结构如下所示：

```
struct taskstate {
    uint32_t ts_link;        // 链接字段
    uintptr_t ts_esp0;        // 0级栈指针
    uint16_t ts_ss0;         // 0级栈段寄存器
    uint16_t ts_padding1;
    uintptr_t ts_esp1;
    uint16_t ts_ss1;
    uint16_t ts_padding2;
    uintptr_t ts_esp2;
    uint16_t ts_ss2;
    uint16_t ts_padding3;
    physaddr_t ts_cr3;        // 页目录基址寄存器
    uintptr_t ts_eip;         // 切换的上次EIP
    uint32_t ts_eflags;
    uint32_t ts_eax;          // 保存的通用寄存器eax
    uint32_t ts_ecx;
    uint32_t ts_edx;
    uint32_t ts_ebx;
    uintptr_t ts_esp;
    uintptr_t ts_ebp;
    uint32_t ts_esi;
    uint32_t ts_edi;
    uint16_t ts_es;           // 保存的段寄存器
    uint16_t ts_padding4;
    uint16_t ts_cs;
    uint16_t ts_padding5;
    uint16_t ts_ss;
    uint16_t ts_padding6;
    uint16_t ts_ds;
    uint16_t ts_padding7;
    uint16_t ts_fs;
    uint16_t ts_padding8;
```

```

uint16_t ts_gs;
uint16_t ts_padding9;
uint16_t ts_ldt;
uint16_t ts_padding10;
uint16_t ts_t;           // 调试陷阱标志(只用位0)
uint16_t ts_iomb;       // i/o map 基地址
};

```

从上图中可以，TSS 的基本格式由 104 字节组成。这 104 字节的基本格式是不可改变的，但在此之外系统软件还可定义若干附加信息。基本的 104 字节可分为链接字段区域、内层栈指针区域、地址映射寄存器区域、寄存器保存区域和其它字段等五个区域。

其中比较重要的是内层栈指针区域，为了有效地实现保护，同一个任务在不同的特权级下使用不同的栈。例如，当从外层特权级 3 变换到内层特权级 0 时，任务使用的栈也同时从 3 级变换到 0 级栈；当从内层特权级 0 变换到外层特权级 3 时，任务使用的栈也同时从 0 级栈变换到 3 级栈。所以 ucore 使用的是 0 级栈，用户态应用使用的是 3 级栈。

TSS 的内层栈指针区域中有三个栈指针，它们都是 48 位的全指针(16 位的选择子和 32 位的偏移)，分别指向 0 级、1 级和 2 级栈的栈顶，依次存放在 TSS 中偏移为 4、12 及 20 开始的位置。当发生从 3 级向 0 级转移时，把 0 级栈指针装入 0 级的 SS 及 ESP 寄存器以变换到 0 级栈。没有指向 3 级栈的指针，因为 3 级是最外层，所以任何一个向内层的转移都不可能转移到 3 级。但是，当特权级由 0 级向 3 级变换时，并不把 0 级栈的指针保存到 TSS 的栈指针区域。这表明向 3 级向 0 级转移时，总是把 0 级栈认为是一个空栈。

当发生任务切换时，80386 中各寄存器的当前值被自动保存到 TR 所指定的 TSS 中，然后下一任务的 TSS 的选择子被装入 TR；最后，从 TR 所指定的 TSS 中取出各寄存器的值送到处理器的各寄存器中。由此可见，通过在 TSS 中保存任务现场各寄存器状态的完整映象，实现任务的切换。

### 3.5.5 【实现】内核态切换到用户态

在 kern/init.c 中的 switch\_test 函数完成了内核态<-->用户态之间的切换。内核态切换到用户态是通过 switch\_to\_user 函数，执行指令“int T\_SWITCH\_TOU”。当 CPU 执行这个指令时，由于是在 switch\_to\_user 执行在内核态，所以不存在特权级切换问题，硬件只会在内核栈中压入 Error Code（可选）、EIP、CS 和 EFLAGS（如下图所示），然后跳转到到 IDT 中记录的中断号 T\_SWITCH\_TOU 所对应的中断服务例程入口地址处继续执行。通过 2.3.7 小节“中断处理过程”可知，会执行到 trap\_dispatch 函数（位于 trap.c）：

```

case T_SWITCH_TOU:
    if (tf->tf_cs != USER_CS) {
        //当前在内核态，需要建立切换到用户态所需的trapframe结构的数据switchk2u
        switchk2u = *tf;
        switchk2u.tf_cs = USER_CS;
        switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
        switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;
        //设置EFLAG的I/O特权位，使得在用户态可使用in/out指令
        switchk2u.tf_eflags |= (3 << 12);
        //设置临时栈，指向switchk2u，这样iret返回时，CPU会从switchk2u恢复数据，
    }
}

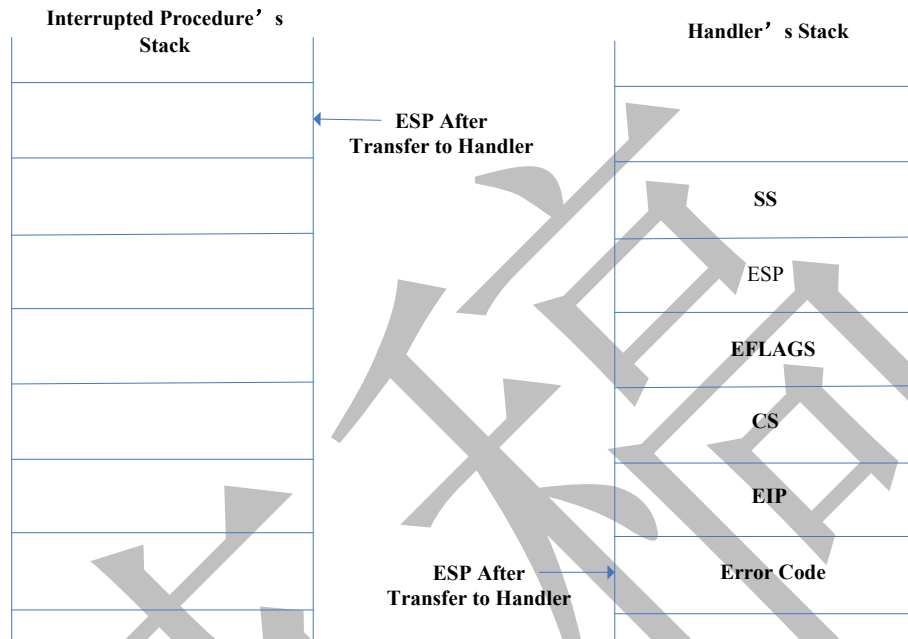
```



```
//而不是从现有栈恢复数据。
*((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
}
```

这样在 `trap` 将会返回，在 `__trapret` 中，根据 `switchk2u` 的内容完成对返回前的寄存器和栈的回复准备工作，最后通过 `iret` 指令，CPU 返回“`int T_SWITCH_TOU`”的后一条指令处，以用户态模式继续执行。

### 有特权级变化的中断产生后堆栈内容



### 3.5.6 【实现】用户态切换到内核态

CPU 在用户态执行到 `switch_to_kernel()` 函数（即执行“`int T_SWITCH_TOK`”）时，由于当前处于用户态，而中断产生后，CPU 会进入内核态，所以存在特权级转换。硬件会在内核栈中压入 `Error Code`（可选）、`EIP`、`CS` 和 `EFLAGS`、`ESP`（用户态特权级的）、`SS`（用户态特权级的）（如下图所示），然后跳转到到 `IDT` 中记录的中断号 `T_SWITCH_TOK` 所对应的中断服务例行程入口地址处继续执行。通过 2.3.7 小节“中断处理过程”可知，会执行到 `trap_disptach` 函数（位于 `trap.c`）：

```
case T_SWITCH_TOK:
    if (tf->tf_cs != KERNEL_CS) {
        //发出中断时，CPU处于用户态，我们希望处理完此中断后，CPU继续在内核态运行，
        //所以把tf->tf_cs和tf->tf_ds都设置为内核代码段和内核数据段
        tf->tf_cs = KERNEL_CS;
        tf->tf_ds = tf->tf_es = KERNEL_DS;
        //设置EFLAGS，让用户态不能执行in/out指令
        tf->tf_eflags &= ~(3 << 12);

        switchu2k = (struct trapframe *) (tf->tf_esp - (sizeof(struct trapframe) - 8));
```

```

//设置临时栈，指向switchu2k，这样iret返回时，CPU会从switchu2k恢复数据，
//而不是从现有栈恢复数据。
memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
*((uint32_t *)tf - 1) = (uint32_t)switchu2k;
}
break;

```

这样在 `trap` 将会返回，在 `__trapret` 中，根据 `switchk2u` 的内容完成对返回前的寄存器和栈的回复准备工作，最后通过 `iret` 指令，CPU 返回“`int T_SWITCH_TOU`”的后一条指令处，以内核态模式继续执行。

## 3.6 原理归纳与小结

读者通过阅读本章和做实验，相信已经尝试了加载操作系统、操作系统访问外设、操作系统处理中断、用户态与内核态切换等一系列小 `project`。这里面体现了哪些操作系统的原理和概念呢？

**系统软件：**在这里读者应该体会到操作系统是一个软件，采用编译器生成可执行代码，通过 `bootloader` 加载到内存中执行，可以执行所有 CPU 指令（包括特权指令），访问到所有硬件资源。相对于一般的应用软件，它主要完成的工作是对计算机的硬件资源管理，并给上层应用提供服务（这里还体现不够）。所以我们将操作系统看成是提供计算机系统级管理和基础共性服务的一种系统软件。

**段式的内存管理：**在这里读者应该体会到 CPU 访问的地址首先是一个虚拟地址，然后通过 MMU 的段式地址保护检查和变换（这要看段描述符表中的描述符是如何设置段基址和段范围的）才能把虚拟地址变换成线性地址（由于没有启动分页机制，线性地址就是物理地址），如果虚拟地址的偏移值超过了段范围值，这会出现故障中断。

**中断处理：**在这里读者应该体会到中断包含了外设产生的外设中断（来的时机不确定）和 CPU 主动产生的陷阱中断（执行特定指令就会产生），有了中断，操作系统就可以随时打断 CPU 正常的顺序执行，转而处理相对更加紧急的中断事件。为了能够让 CPU 继续正常执行，在中断处理前需要保存足够的硬件信息（主要是 CPU 各种寄存器的值，一般放在内核栈中），以便于中断处理完毕后，通过恢复这些硬件信息，能够回到被打断的地方继续执行。

**特权级：**在这里读者应该体会到在不同的特权级可以完成的事情是不一样的。操作系统需要管理整个计算机资源，所以它应该运行在 CPU 的最高特权级，而应用程序不必也不能使用特权指令或访问操作系统的地址空间，所以它应该运行在 CPU 的非特权级。如果应用程序执行特权指令或访问操作系统的地址空间，则 CPU 硬件的安全检查机制会阻止应用程序的执行并尝试故障中断，把 CPU 控制权转给操作系统进行进一步处理。如果应用程序需要访问计算机资源，可以通过执行特定的指令，产生陷阱中断，从而完成从用户态到内核态的切换，让操作系统为应用程序提供服务。

## 3.7 习题

- [1] `ucore.img` 是如何一步一步生成的？(需要比较详细地解释 `Makefile` 中每一条相关命令和命令参数的含义，以及说明命令导致的结果)
- [2] 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？
- [3] DOS 是一个具有保护能力(即用户应用不能直接访问/执行操作系统的数据/代码)

---

的操作系统吗？

- [4] 能否扩展 proj3.1 中的 `print_stackframe` 函数，能够显示出实际传递参数名字和内容？
- [5] 中断向量表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？
- [6] 完成 `trap.c` 中对中断向量表进行初始化的函数 `tvinit`。在 `tvinit` 函数中，依次对所有中断入口进行初始化。使用 `mmu.h` 中的 `SETGATE` 宏，填充 `idt` 数组内容。注意除了系统调用中断(`T_SYSCALL`)以外，其它中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常，权限为陷阱门描述符。每个中断的入口由 `vectors.pl` 生成，使用 `trap.c` 中声明的 `vectors` 数组即可。
- [7] 完成 `trap.c` 中的中断处理函数 `trap` 对时钟中断进行处理的部分填写 `trap` 函数中处理时钟中断的部分，使操作系统每遇到 100 次时钟中断后，便向屏幕上打印一行文字“tic...”
- [8] (Challenge A)重新设计 `cprintf` 函数，适当的增强功能，使得它能够按照不同的颜色输出字符。通常的做法是在待输出的字符中，插入一些 [ANSI escape](#) 符号，不过，这不是强制要求，你可以按照任何你认为合适的方式定义颜色的表示方法。关于 VGA 显示的资料，可以借助互联网。`reference` 中是一些与 `ucore` 和 `jost` 有关的资料。(2 周内完成：网络学堂一起提交)
- [9] (Challenge B)增加 `syscall` 功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务（通过网络查询所需信息，可找老师咨询。如果完成，且有兴趣做代替考试的实验，可找老师商量）。需写出详细的设计和分析报告。完成出色的可获得适当加分。(3 周内完成：单独提交)
- [10] (Challenge B)在真实机器上通过 U 盘启动并运行 `proj3`。

---

## 3.8 附录 补充材料

---

### 1. 如何调试 Makefile

当执行 `make` 时，一般只会显示输出，不会显示 `make` 到底执行了哪些命令。这时，只需执行

```
$ make -d
```

就可以看到 `make` 是如何决定决定以及按照什么执行命令的。但是，这时输出信息会特别多，可以执行

```
$ make --debug=b
```

要获取更多有关 `make` 的信息，请执行

```
$ man make
```

### 2. gcc\ld\objcopy 执行参数说明

gcc 执行参数：

- `-fno-builtin`：表示不使用不以 `__builtin` 为前缀的函数，
- `-MD`：生成表示文件依赖关系的 `.d` 文件
- `-ggdb`：提供了编译信息，用于 `gdb` 调试
- `-M32`：指明生成的目标代码为 32 位
- `-fno-stack-protector`：不检测缓存溢出
- `-nostdinc`：只为头文件寻找 `-I` 选项标明的目录
- `-c`：表示完成编译工作，生成 `.o` 目标文件，但是不执行 `ld` 来链接 `.o` 目标文件。

ld 执行参数：

- `-m`：指明文件类型为何种硬件平台（比如 `i386`）文件格式
- `-N`：允许代码段数据段可读可写，取消数据段页对齐，禁止链接共享库
- `-e`：确定程序入口地址
- `-Ttext`：设定代码段起始地址。

objcopy 执行参数：

- `-S`：尽可能反汇编出源代码，尤其当编译的时候指定了 `-g` 这种调试参数时，会产生比较明显的效果
-

### 3. 串口

串口又称 RS-232 接口、RS-232 口、异步口或 COM（通信）口。RS-232C 标准（协议）的全称是 EIA-RS-232C 标准，定义是“数据终端设备（DTE）和数据通讯设备（DCE）之间串行二进制数据交换接口技术标准”。PC 机一般使用 8250 或 16550 的作为串行通讯控制器。

8250（16550）的寄存器如下表所示：

| 基地址 | 读/写        | 寄存器缩写 | 注释               |
|-----|------------|-------|------------------|
| 0   | Write      | -     | 发送保持寄存器（DLAB=0）  |
| 0   | Read       | -     | 接收数据寄存器（DLAB=0）  |
| 0   | Read/Write | -     | 波特率低八位（DLAB=1）   |
| 1   | Read/Write | IER   | 中断允许寄存器          |
| 1   | Read/Write | -     | 波特率高八位（DLAB=1）   |
| 2   | Read       | IIR   | 中断标识寄存器          |
| 2   | Write      | FCR   | FIFO 控制寄存器       |
| 3   | Read/Write | LCR   | 线路控制寄存器          |
| 4   | Read/Write | MCR   | MODEM 控制寄存器      |
| 5   | Read       | LSR   | 线路状态寄存器          |
| 6   | Read       | MSR   | MODEM 状态寄存器      |
| 7   | Read/Write | -     | Scratch Register |

PC 机支持 4 个串行口，即 COM1-COM4，，对应 I/O 地址分别为 0x3F8/0x2F8/0x3E8/0x2E8，COM1 及 COM3 使用 PC 机中断 4，COM2 及 COM4 使用中断 3。在上表中，8250 共有 12 个寄存器，使用了 8 个地址，其中部分寄存器共用一个地址，由 DLAB=0/1 来区分，在 DLAB=1 用于设定通讯所需的波特率。常用的波特率参数见下表：

| 速率（BPS） | 波特率高八位 | 波特率低八位 |
|---------|--------|--------|
| 50      | 09h    | 00h    |
| 300     | 01h    | 80h    |
| 600     | 00h    | C0h    |
| 2400    | 00h    | 30h    |
| 4800    | 00h    | 18h    |
| 9600    | 00h    | 0Ch    |
| 19200   | 00h    | 06h    |
| 38400   | 00h    | 03h    |
| 57600   | 00h    | 02h    |
| 115200  | 00h    | 01h    |

以下几个表格为 8250 的寄存器的功能描述：

中断允许寄存器（IER）：

| 位 | 注释             |
|---|----------------|
| 7 | 未使用            |
| 6 | 未使用            |
| 5 | 进入低功耗模式（16750） |
| 4 | 进入睡眠模式（16750）  |

- |   |               |
|---|---------------|
| 3 | 允许 MODEM 状态中断 |
| 2 | 允许接收线路状态中断    |
| 1 | 允许发送保持器空中断    |
| 0 | 允许接收数据就绪中断    |

Bit0 置 1 将允许接收到数据时产生中断，Bit1 置 1 时允许发送保持寄存器空时产生中断，Bit2 置 1 将在 LSR 变化时产生中断，相应的 Bit3 置位将在 MSR 变化时产生中断。

#### FIFO 控制寄存器（FCR）：

- | 位         | 注释            |
|-----------|---------------|
| Bit7:6=00 | 1Byte 产生中断    |
| Bit7:6=01 | 4Byte 产生中断    |
| Bit7:6=10 | 8Byte 产生中断    |
| Bit7:6=11 | 14Byte 产生中断   |
| Bit5      | 允许 64 字节 FIFO |
| Bit4      | 未使用           |
| Bit3      | DMA 模式选择      |
| Bit2      | 清除发送 FIFO     |
| Bit1      | 清除接收 FIFO     |
| Bit0      | 允许 FIFO       |

FCR 可写但不可以读，该寄存器用来控制 16550 或 16750 的 FIFO 寄存器。Bit0 置 1 将允许发送/接收的 FIFO 工作，Bit1 和 Bit2 置 1 分别用来清除接收及发送 FIFO。清除接收及发送 FIFO 并不影响移位寄存器。Bit1:2 可自行复位，因此无需使用软件对其清零。Bit6:7 用来设定产生中断的级别，发送/接收中断将在发送/接收到对应字节数时产生。

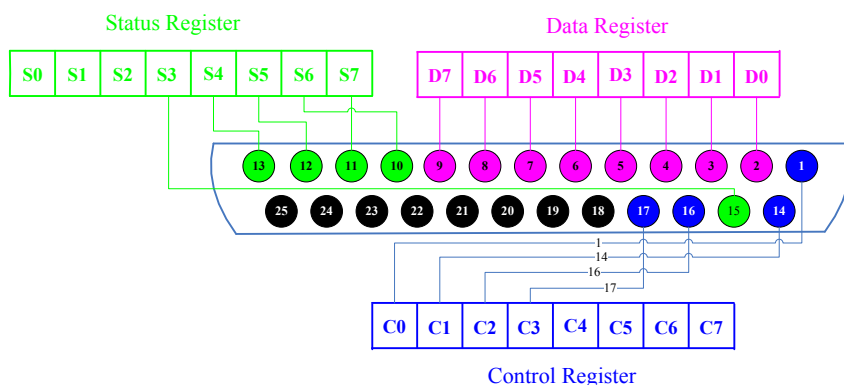
#### 线路状态寄存器（LSR）：

- | 位    | 注释           |
|------|--------------|
| Bit7 | FIFO 中接收数据错误 |
| Bit6 | 发送移位寄存器空     |
| Bit5 | 发送保持寄存器空     |
| Bit4 | 间断           |
| Bit3 | 帧格式错         |
| Bit2 | 奇偶错          |
| Bit1 | 超越错          |
| Bit0 | 接收数据就绪       |

LSR 为只读寄存器，当发生错误时 Bit7 为 1，Bit6 为 1 时标示发送保持及发送移位寄存器均空，Bit5 为 1 时标示仅发送保持寄存器空，此时，可以由软件发送下一数据。当线路状态为 0 时 Bit4 置位为 1，帧格式错时 Bit3 置位为 1，奇偶错和超越错分别将 Bit2 及 Bit1 置位为 1。Bit0 置位为 1 表示接收数据就绪。

## 4. 并口

为了理解针对并口的编程，我们需要了解并口的基本硬件知识。并口的硬件连接器为 DB25，被分成了三组线路，如下图所示：



- 1) Data lines (data bus)      数据线
- 2) Control lines              控制线
- 3) Status lines                状态线

数据线用来传输数据，控制线用来控制外围设备，外围设备通过状态线返回状态信号，这些线分别与内部的数据寄存器，控制寄存器和状态寄存器相连。所以可以通过编程来操作这些寄存器。标准并口寄存器有：

- 1) data register              数据寄存器
- 2) Status register            状态寄存器
- 3) Control register          控制寄存器

通过写这些寄存器，将在相应线上传输相应的电压信号，可以通过万用表来测量，给并口的这些信号也可以通过这些寄存器读出来，例如：我们写入“1”到数据寄存器，Data0 将被提高为+5V 电压，这样，我们就可以通过编程打开或关闭数据线或控制线。

在基于 80386 的计算机中，这些寄存器是通过 I/O 地址进行映射的，LPT1（第一个并口）的 I/O 基地址是 0x378，LPT2（第二个并口）的 I/O 基地址是 0x278，LPT3（第三个并口）的 I/O 基地址是 0x3bc。我们这里只使用了 LPT1。数据寄存器在基地址上，状态寄存器在基地址+1 控制寄存器在基地址+2，因此，当我们一旦知道基地址，我们可以通过计算出其他寄存器的地址。

状态寄存器 I/O 地址：

|              |             |              |           |              |              |    |    |    |
|--------------|-------------|--------------|-----------|--------------|--------------|----|----|----|
| <b>0x379</b> | (S7)11 线: 忙 | (S6)10 线: 应答 | (S5)12:缺纸 | (S4) 13 线:联机 | (S3)15 线: 错误 | S2 | S1 | S0 |
|--------------|-------------|--------------|-----------|--------------|--------------|----|----|----|

0x379 为打印机状态寄存器的 I/O 地址。标记为 S7 的信号表示最高位，S0 表示为最低位，只有 S3—S7 五个信号值才是真正有用的。其中 bootloader 和 ucore 用到的 S7\*(busy)即表示并口是否空闲，这样我们可以读状态寄存器 I/O 地址，通过查看该位来了解打印机是否处于忙状态。

控制寄存器 I/O 地址：

|              |   |   |      |    |      |      |      |     |
|--------------|---|---|------|----|------|------|------|-----|
| <b>0x37A</b> | 1 | 1 | 输入控制 | 中断 | 17 线 | 16 线 | 14 线 | 1 线 |
|--------------|---|---|------|----|------|------|------|-----|

这个 0x37A 计算机控制并口的地址，可以产生对并口进行控制的必要信号。最高两位常为 1；第 5 位写 1 表示可以向并口输入数据；第 4 位表示对并口是否产生中断信号；第 3/2/1/0 位 分别控制并口的第 17/16/14/线。

## 5. 8259A 可编程中断控制器

【提示】下述内容来自互联网

对 80386 而言，中断是由异步的外部事件引起的，80386 有两根引脚 INTR 和 NMI 接受外部中断请求信号。INTR 接受可屏蔽中断请求，NMI 接受不可屏蔽中断请求。在 80386 中，标志寄存器 EFLAGS 中的 IF 标志(可通过 CLI/STI 指令来改变)决定是否屏蔽可屏蔽中断请求。外部硬件在通过 INTR 发出中断请求信号的同时，还要向处理器给出一个 8 位的中断向量。处理器在响应可屏蔽中断请求时，读取这个由外部硬件给出的中断向量号。

### ● 8259A 内部结构

可编程中断控制器芯片 8259A 可配合 80386 工作，能够根据设置向处理器提供上述中断向量号，还能处理中断请求的优先级。每个 8259A 芯片可以支持 8 路中断请求信号。8259A 内部结构如下图所示：

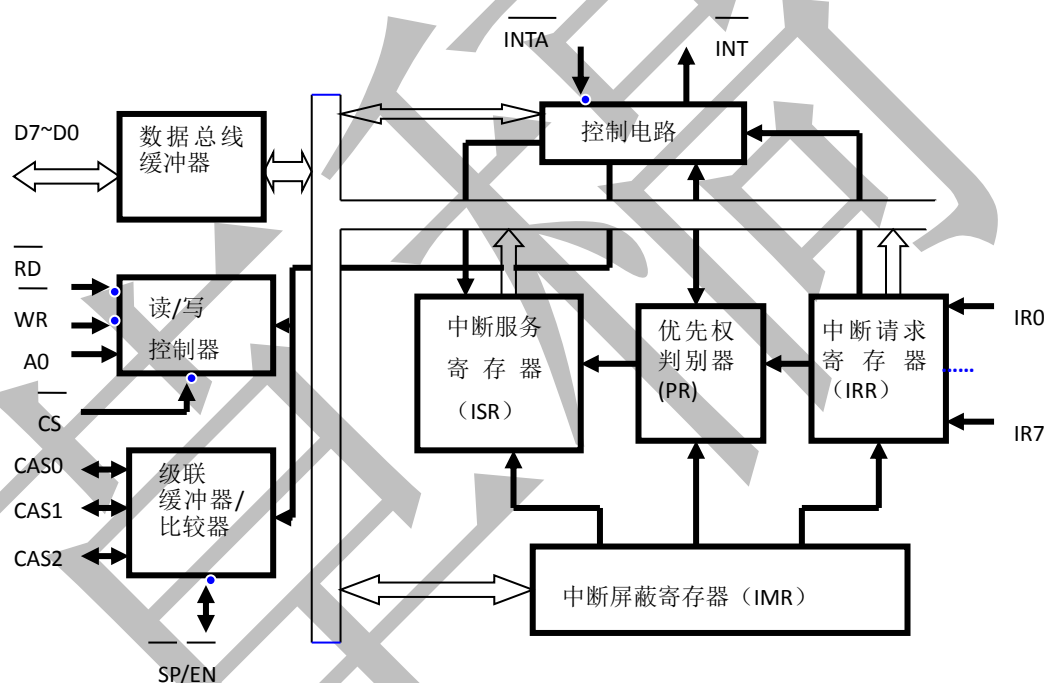


图 8259A 内部结构图

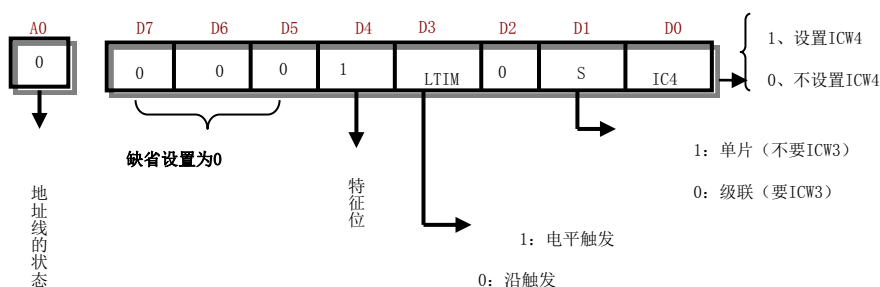
一片 8259A 芯片只能管理 8 级中断，在多于 8 级中断的系统中，必须将多片 8259A 级联使用。，如果使用 9 个 8259A 芯片(一个主片，8 个从片)，就可使 80386 在单个引脚 INTR 上接受多达 64 个中断源的中断请求信号。

最简单的级联方式见下图，为两级级联方式。在 ucore 中，假定系统采用了 2 个 8259A 芯片。





## (1) ICW1——芯片控制初始化命令字



在地址 A0 为 0 时，若对 8259A 写入一个 D4=1 的字节，则启动了其初始化编程。写入的这个字节被当成 ICW1，D4=1 是其特征位，以区别 OCW2 和 OCW3 控制字的设置。

- A0=0，写入命令字的端口地址，ICW1 必须写入 8259A 的偶地址端口。
- D0=1 表示需要 ICW4，否则不送。在不送 ICW4 时，ICW4 的各位默认值均为 0。
- D1=0 表示系统中有多片 8259A 级联，则表示单片工作，此时不送 ICW3。
- D3 规定 IR7~IR0 信号的触发方式。D3=1 为高电平触发，否则为上升沿触发。

在 ucore 中的 8259A 初始化函数 pic\_init() 中，通过执行

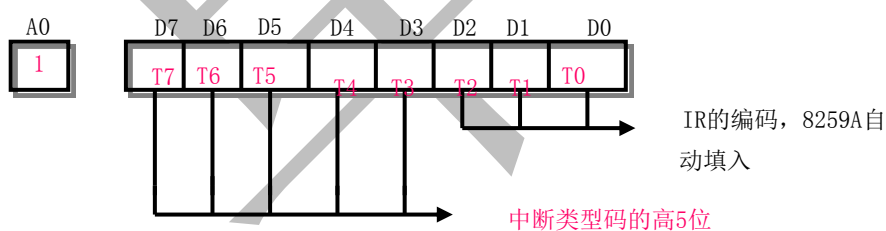
```
outb(IO_PIC1, 0x11); //主中断控制器
```

```
outb(IO_PIC2, 0x11); //从中断控制器
```

完成了对 ICW1 的初始化。

## (2) ICW<sub>2</sub>——设置中断类型码基值命令字

ICW<sub>2</sub> 是中断矢量基值寄存器，用于设置中断类型码的命令字。其形式如下：



A<sub>0</sub>=1，ICW<sub>2</sub> 必须写入 8259A 的奇地址端口中。D<sub>2</sub>~D<sub>0</sub> 不需要编程，固定为 000，在响应时由中断源序号填入相应值。D<sub>7</sub>~D<sub>3</sub> 表示中断矢量的高五位。

在 ucore 中的 8259A 初始化函数 pic\_init() 中，通过执行

```
outb(IO_PIC1 + 1, IRQ_OFFSET); //主中断控制器
```

```
outb(IO_PIC2 + 1, IRQ_OFFSET + 8); //从中断控制器
```

完成了对 ICW2 的初始化。

### (3) ICW<sub>3</sub>——标识主片/从片初始化命令字

ICW<sub>3</sub> 是 8259A 的级联命令字，用于主片/从片的初始化命令字。单片 8259A 工作时不需写入，多片 8259A 级联时，有主片从片之分，需要分别写入 ICW<sub>3</sub>。主片/从片初始化命令字格式见下图。



A0=1, ICW<sub>3</sub> 必须写入 8259A 的奇地址端口中。主片 ICW<sub>3</sub> 的 D7~D0 对应其八条中断请求线 IR7~IR0, 若某根 IR 线上接有从 8259A 片, 则 ICW<sub>3</sub> 的相应位应写成 1, 否则写成 0。各从片的 ICW<sub>3</sub> 仅 D2~D0 有意义, 作为其从片标识码, 高位固定为 0, 这个从片标识码须和本片所接主片 IR 线的序号一致。中断时响应时, 主片通过级联线 CAS2~CAS0 送出被允许中断的从片标识码。各从片用自己的 ICW<sub>3</sub> 和 CAS2~CAS0 比较, 二者一致的从片被确定为当前中断源, 才可发送自己的中断矢量。

在 ucore 中的 8259A 初始化函数 pic\_init() 中, 通过执行

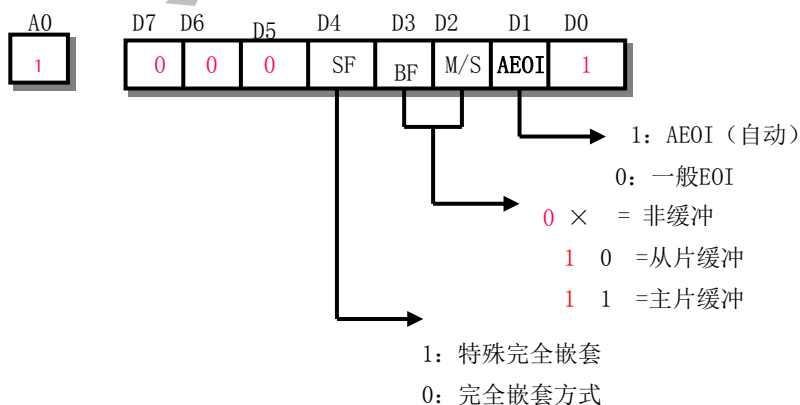
```
outb(IO_PIC1 + 1, 1 << IRQ_SLAVE); //主中断控制器
```

```
outb(IO_PIC2 + 1, IRQ_SLAVE); //从中断控制器
```

完成了对 ICW<sub>3</sub> 的初始化。

### (4) ICW<sub>4</sub>——方式控制初始化命令字

ICW<sub>4</sub> 是方式控制初始化命令字, 当 ICW<sub>1</sub> 中的 D0 为 1 时, 则在初始化时需要设置 ICW<sub>4</sub>。其格式如下图示。



A0=1, ICW4 必须写入 8259A 的奇地址端口中。

D7~D5, 这 3 位总为 0, 用来表示 ICW4 的标识码。

D4 指定了中断的嵌套方式。D4=0 为一般嵌套方式, 当某个中断正在服务时, 本级中断及更低级的中断都被屏蔽, 只有更高的中断才能响应。对于单片 8259A 的中断系统, 这种安排没有问题。但对多片 8259A 级联组成的中断系统, 当某从片中一个中断正在服务时, 主片即将这个从片的所有中断屏蔽。因此即使本从片中有比正在服务的中断级别更高的中断源发出请求, 也不能得到响应, 即不能中断嵌套。D4=1 则是特殊嵌套方式, 仅仅屏蔽比当前中断源低级的中断, 于是上述情况就可以中断嵌套。

D3 为数据缓冲选择。D3=1 时 8259A 的数据线和系统总线之间要加上三态缓冲器, 此时 8259A 的 SP/EN 引脚变成输出线, 以作为缓冲器的选通信号。每当 8259A 的数据送往系统总线时, SP/EN 引脚输出有效的低电平。这种情况用于多片 8259A 的级联, 此时主从片的区分就不能再靠 SP/EN 固定接高或低电平, 而用 ICW4 的 D2 位。规定主片的 D2=1, 从片的 D2=0。D3=0 时不加缓冲器, D2 无意义。

D1 说明了中断结束的方式。D1=0 是正常方式, 即在中断服务结束时, 8259A 送一个 EOI 命令字 (OCW2), 于是中断服务寄存器 ISR 中与中断源相对应的位被清除。D1=1 是自动 EOI 方式, 即在中断响应时, 在 8259A 送出中断矢量后, 自动将 ISR 相应位复位。

D0 指定了系统中所用 CPU 的模式, D0=0 时, 8080/8085CPU, D0=1 时用 8086/8088CPU。

若在某种应用场合, 正好需要 ICW4 各位都为 0, 则可以不写 ICW4。因为 8259A 在进入初始化时, 已自动将 ICW4 全部复位。

在 ucore 中的 8259A 初始化函数 pic\_init() 中, 通过执行

```
outb(IO_PIC1 + 1, 0x3); //主中断控制器
```

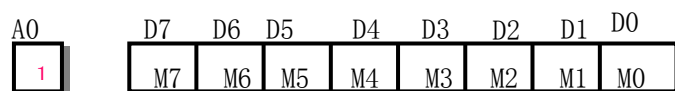
```
outb(IO_PIC2 + 1, 0x3); //从中断控制器
```

完成了对 ICW4 的初始化。

8259A 在初始化编程后, 应再进行工作编程, 即写入工作命令字, 用于对中断处理过程作动态控制。工作命令字有三个, 它们各有自己的特征位, 因此写入的顺序没有要求。在中断系统工作中, 某些工作命令字可能需要重复多次地写入。

#### (1) OCW1——中断屏蔽操作命令字

用来实现对中断源的屏蔽功能, OCW1 的内容直接写入屏蔽寄存器 IMR, OCW1 的作用见下图。

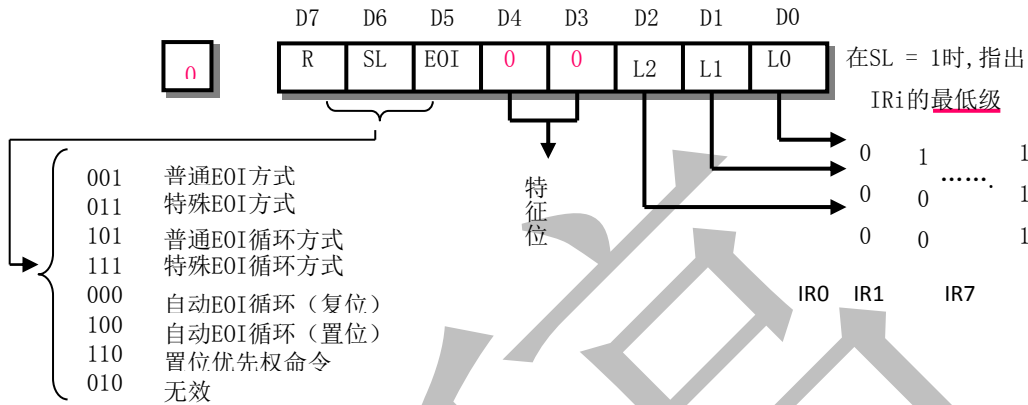


Mi = 1 禁止中断 (屏蔽), Mi = 0 允许中断

A0=1, 表示 OCW1 必须写入 8259A 的奇地址端口中 (三个命令字中只有 OCW1 占有奇地址)。

M7~M0 对应于中断屏蔽寄存器 IMR 各位，其每一位控制一根中断请求输入线，屏蔽字为“1”的位，对应的中断请求线被屏蔽；否则，被允许。（在初始化开始时，默认屏蔽字各位全为 0）。ucore 没有使用。

（2）OCW2——优先级循环方式和中断结束方式操作命令字（设置 EOI、SEOI、循环，即定义中断结束方式）



OCW2 主要作用是设置中断结束方式。其中 A0=0，表示 OCW2 应写入偶地址端口。

D4=D3=0，是 OCW2 的标志位，它虽然和 OCW3 都占有偶地址（A0=0），但其特征为 D4D3=00，因此不会发生混淆。

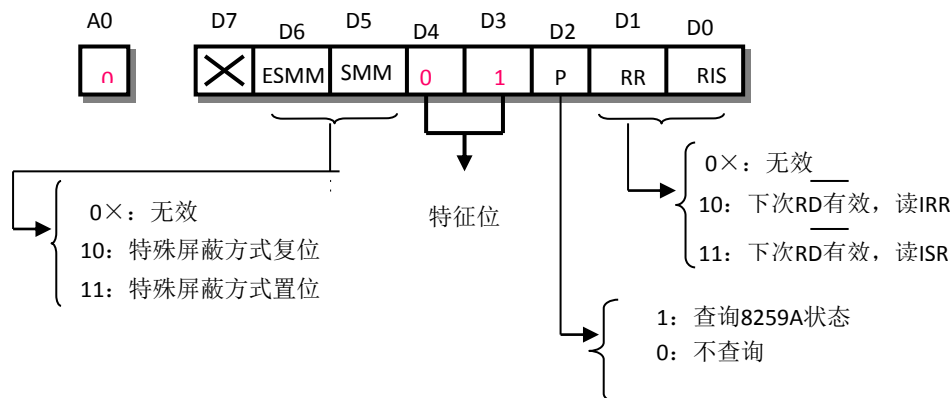
D7 位表示中断优先级循环。当 D7=0 时，八个中断请求 IR7~IR0 的优先级固定不变，IR7 最低，IR0 最高。当 D7=1 时，优先级可以循环，即 IR7 和 IR0 首尾相接成一闭环，各级的优先级在其中循环移位。循环到什么情况停止，还与其他位有关。

D6 位表示特殊循环。当 D6=1 时，最低三位 D2~D0 的二进制编码指定了一条外部中断请求线 IRi。此时若 D7=1，则称特殊循环，即优先级的循环移位一直进行到最低优先级对准 IRi 为止，于是最高优先级也就移到 IR (i-1)。D6=0 时，最低优先级自动循环到当前服务的中断请求线，D2~D0 无意义。

D5 位是中断结束位。D5=1 表示中断结束 (EOI 命令)。当用 8259A 来实现中断管理时，中断服务程序结束时 (返回指令 IRET 前)，必须给 8259A 送一条 EOI 命令 (即 D5=1 的 OCW2)。8259A 收到这条命令后，将中断服务寄存器中的相应位清除，然后才为其他中断源服务。若 D6D5=11，则称为特殊的中断结束 (SEOI 命令)，它将复位 ISR 中由 OCW2 的 D2~D0 编码指定的位。

ucore 没有使用。

（3）OCW3——特殊屏蔽方式和中断查询方式操作命令字



OCW3 的作用是写入多功能操作命令字: 设置/取消特殊屏蔽方式; 设置中断查询方式; 设置对 8259A 内部寄存器的读出命令。

A0=0, OCW3 必须写入 8259A 的偶地址端口。

D4D3=01。写入 OCW3 的地址和 OCW2 相同, 但其特征为 D4D3=01, 以区别 OCW2。

OCW3 经常用来配合读 8259A 内部寄存器的内容。D1D0=10, 则将读入其中断请求寄存器的内容; 若 D1D0=11, 则读入的是中断服务寄存器的状态。

OCW3 中的 D2 位表示查询。8259A 也可以不工作于中断方式, 而工作于查询方式, 此时应写入 D2=1。CPU 可以反复对 8259A 查询, 但每次查询前都应先送一次 D2=1 的 OCW3。

OCW3 的 D6D5 用来控制特殊屏蔽功能。当 D6D5=11 时, 设置特殊屏蔽; 当 D6D5=10 时, 清除特殊屏蔽。当一个优先级较高的中断源正在服务的过程中, 若设置了特殊屏蔽功能, 则允许优先级较低的中断源产生中断嵌套。

上述操作控制字 OCW1~OCW3 可安排在预置命令字之后, 用户可根据需要在程序的任何位置上设置它们, 当需要读取 ISR 或 IRR 寄存器内容或需要查询当前 8259A 的中断状态时, 都必须先定义 OCW3, 然后用 IN 指令读入。如果只需要读 IMR 寄存器内容, 则不需要定义 OCW3。所在并不是任何时候都需要设置 OCW3 操作命令字。

在 ucore 中的 8259A 初始化函数 pic\_init() 中, 通过执行

```
outb(IO_PIC1, 0x68); //主中断控制器
outb(IO_PIC1, 0x0a); //主中断控制器
outb(IO_PIC2, 0x68); //从中断控制器
outb(IO_PIC2, 0x0a); //从中断控制器
```

完成了对 ICW4 的初始化。在 ucore 中的 8259A 初始化函数 pic\_init() 函数的最后, ucore 还通过执行“pic\_setmask(irq\_mask);”语句把 salve 8259A 连接的第二个中断线使能。

---

# 操作系统设计与实现

## The Design & Implementation of Operating System

陈渝 向勇

清华大学计算机系  
2011 年

## 第3章 内存管理

一个编程人员希望拥有容量无限大、速度无限快，而且是非易失型的(nonvolatile)的内存空间，到 lab1 为止，这个梦想还无法轻易满足。为此绝大多数的计算机采用了一种折衷的方法，即建立一个分层的存储器结构，最高层是 CPU 内部的一些寄存器，它们的访问速度是最快的，但容量不是很大，一般小于 1KB；第二层是高速缓存（即硬件 cache），实现在 CPU 内部，接近寄存器速度，容量一般小于 4MB；第三层是主存储器（内存），其访问速度比寄存器小一个数量级、价格便宜，目前几百块人民币就可以买到 4GB。以上这三种存储器都是易失型的，即在断电后，其内容全部会丢失掉。第四层是磁盘，它的访问速度较慢、价格较便宜，目前花几百块钱就可以买到存储容量>1TB 的硬盘，而且是非易失型的。

操作系统需要尽量满足编程人员的梦想，为此它需要管理上述存储器层次结构形成的存储空间，并完成如下主要任务：

- 记录存储空间的使用情况，即记录哪些部分正在被使用，哪些部分还空闲；
- 当需求方需要存储空间时，能快速地分配给它合适大小的空间；在需求方显式表示不需要申请到的存储空间时，能把存储空间回收，便于以后的分配；
- 隔离不同的内存区域，确保在限制在一个内存区域中运行的软件无法访问区域以外的内存空间。这种机制称为地址保护（地址隔离）机制。
- 如果内存太小，就需要把内存当中使用较少的数据所占空间送到磁盘上，给使用较多的数据腾出内存空间来；如果将来又访问到缓存到硬盘的数据，需要把这些数据重新加载到内存中进行访问。这种机制称为换入换出（swap in/out）机制，并涉及页替换算法。
- 即使需求方表明了需要内存，但如果需求方没有实际访问所需内存前，则并不完成实际的物理内存分配。这种机制称为按需分配（如果是基于很分页机制，也称为按需分页）。
- 设两个具有父子关系的程序共享同一地址空间（子程序同享父程序的地址空间），若二程序只是读此地址空间，则地址空间不会有变化；若其中一个程序对此地址空间某地址进行了写操作，则要把包含此地址的页空间复制一份给执行写操作的程序，这时此二程序将有不同的地址空间，可独立运行，相互不干扰。这种机制减少了父程序创建子程序地址空间的开销，称为写时复制（Copy On Write，简称 COW）机制。

本章内容主要涉及操作系统的内存管理，包括物理内存管理和基于分页机制的虚拟内存管理。读者通过阅读本章的内容并动手实践相关的 5 个 project 实验：

- proj5：能够探测物理内存并建立页表，实现分页管理
- proj5.1/5.1.1/5.1.2：实现基于连续物理页的 first/best/worst-fit 分配算法
- proj5.2：实现基于连续物理页的 buddy 分配算法
- proj6：实现任意大小内存分配的 slab 分配算法
- proj7：实现缺页中断服务例程和虚拟内存管理结构（VMM struct），提供按需分页的支持
- proj8：实现类似改进时钟算法的页面置换算法并支持页粒度的换入换出机制



- proj9/proj9.1/proj9.2: 完善虚存管理(proj9)并逐步实现了进程间内存共享(proj9.1)和 copy on write (COW) 机制(proj9.2)

可以掌握如下知识:

- 与操作系统原理相关
  - 内存管理: 基于分页机制的内存管理
  - 内存管理: 连续内存分配算法
  - 内存管理: 非连续内存分配算法
  - 内存管理和中断: 缺页中断服务例程
  - 内存管理: 虚存管理中的页面置换算法和页换入换出机制
  - 内存管理: 按需分页机制和写时复制机制
- 操作系统原理之外
  - 80386 对分页管理(页表等)的硬件支持
  - 页粒度的页面置换策略和页换入换出的具体实现

本章内容中主要涉及内存管理的重要功能主要有两个:

- 提供空闲内存空间: 这样给操作系统和应用程序的代码和数据足够的存放“地方”, 使得二者能够正常高效地运行, 为此需要完成内存/外存的空间的分配、管理、释放等算法。
- 提供内存空间隔离保护: 隔离用户态应用程序和内核态操作系统之间, 以及不同应用程序之间的内存空间, 使得不会出现访问冲突, 为此需要为不同的应用程序和操作系统划分不同的地址空间, 一个应用程序越界规定的地址空间会出现内存访问故障中断。

为了让读者能够从实践上来理解内存管理的基本原理, 我们设计了上述实验, 主要的功能逐步实现如下所示:

- 首先是扩展 ucore 的功能, 使它能够发现并管理 PC 系统中可用的空闲物理内存;
- 然后是建立分页机制, 建立线性地址(分段机制已经完成了逻辑地址到线性地址的转换)到物理地址的映射关系和具体转换操作, 这样使得应用程序无法直接访问到物理地址, 而是只能访问由操作系统设定好的物理地址空间, 从而使得应用程序的访问空间可控;
- 为了高效地完成操作系统的其他功能单元和应用程序的空闲内存空间需求, 需要设计以页(4096 字节)为最小分配单位的面向连续物理地址空间的内存分配算法;
- 还要设计面向任意大小的内存空间(在物理地址空间上不一定连续)的虚拟内存分配算法;
- 为了给应用程序提供超过实际物理内存空间大小的虚拟内存空间, 需要把临时不常用到的内存换出(swap out)到硬盘(也称外存)中, 等到需要访问的时候, 再换入(swap in)到内存中。设计高效的页面置换算法会尽量保存经常访问的数据在内存中, 而不经常访问的数据会换出到硬盘中。

## 3.9 建立分页管理机制

### 3.9.1 试验目标

操作系统和应用程序都需要内存空间来存放代码和数据,这要求操作系统能够高效管理和保护整个计算机中的物理内存,并给自己和上层应用提供简洁安全的内存申请和释放的服务接口。通过分段机制可以完成虚拟地址到线性地址的转换,而通过分页机制可以进一步完成线性地址到物理地址的转换。分段机制中对每段的大小是可变的,分页机制中页的大小固定为 4KB,这样在操作系统对实现内存管理上会更加简洁。所以在建立好分页机制后,分段机制的映射功能退化为对等映射,即虚拟地址=线性地址,这样实际的地址映射工作将由分页机制来完成。

为此 ucore 需要在已有的分段机制的基础上,进一步加入分页机制,达到可以通过分页完成对不同应用程序执行的内存空间进行隔离(其实分段也能够达到此目的,但相对实现的开销会比较大,受到的限制(比如支持的应用执行个数)也较多)的目标。并且为后续的虚存管理提供基础支持。

### 3.9.2 proj5/5.1/5.1.1/5.1.2/5.2 概述

#### 一 实现描述

proj5 基于 proj4.3 实现,主要完成了对计算机实际物理内存大小与分布的探测,实现以页(大小为 4KB)为单位的简单物理内存管理,并通过建立二级页表,实现了分页内存管理,为将来试验中实现虚存管理打下一个基础。通过分析和实现 proj5,读者可以了解到:

- 物理内存空间布局的探测;
- 基于分页机制的存储管理;
- 32 位地址空间的二级页表结构;

proj5.1 在 proj5 的基础上实现了基于 best fit 内存分配算法的页级内存分配和释放功能;proj5.1.1 在 proj5.1 的基础上实现了 first fit 内存分配算法的页级内存分配和释放功能;proj5.1.2 在 proj5.1 的基础上实现了 worst fit 内存分配算法的页级内存分配和释放功能;proj5.2 在 proj5.1 的基础上实现了更加实用和强大的 buddy 内存分配算法的页级内存分配和释放功能。这些 proj 是操作系统原理相关算法的具体实现。读者可以参考这些实现完成新的内存分配算法。在这里通过讲解 proj5 的实现,让读者理解如何基于一个内存分配管理框架来实现不同的内存分配算法。

#### 二 项目组成

```
proj5
|-- boot
|
|-- asm.h
```

```

| |-- bootasm.S
| `-- bootmain.c
|-- kern
| |-- init
| | |-- entry.S
| | `-- init.c
| |-- mm
| | |-- default_pmm.c
| | |-- default_pmm.h
| | |-- memlayout.h
| | |-- mmu.h
| | |-- pmm.c
| | `-- pmm.h
| |-- sync
| | `-- sync.h
| `-- trap
|     |-- trap.c
|     |-- trapentry.S
|     |-- trap.h
|     `-- vectors.S
|-- libs
| |-- atomic.h
| |-- list.h
`-- tools
    |-- kernel.ld

```

相对与 proj4.3, proj5 增加了 6 个文件。主要修改和增加的文件如下:

- boot/bootasm.S: 增加了对计算机系统中物理内存布局的探测功能;
- kern/init/entry.S: 根据临时段表重新暂时建立好新的段空间, 为进行分页做好准备。
- kern/mm/default\_pmm.[ch]: 提供基本的基于链表方法的物理内存管理(分配单位为页, 即 4096 字节);
- kern/mm/pmm.[ch]: pmm.h 定义物理内存管理类框架 struct pmm\_manager, 基于此通用框架可以实现不同的物理内存管理策略和算法(default\_pmm.[ch] 实现了一个基于此框架的简单物理内存管理策略); pmm.c 包含了对此物理内存管理类框架的访问, 以及与建立、修改、访问页表相关的各种函数实现。
- kern/sync/sync.h: 为确保内存管理修改相关数据时不被中断打断, 提供两个功能, 一个是保存 eflag 寄存器中的中断屏蔽位信息并屏蔽中断的功能, 另一个是根据保存的中断屏蔽位信息来使能中断的功能;
- libs/list.h: 定义了通用双向链表结构以及相关的查找、插入等基本操作, 这是建立基于链表方法的物理内存管理(以及其他内核功能)的基础。其他有类似双向链表需求的内核功能模块可直接使用 list.h 中定义的函数。
- libs/atomic.h: 定义了对一个变量进行读写的原子操作, 确保相关操作不被中断打断。
- tools/kernel.ld: 修改了 ucore 的起始入口和代码段的起始地址

### 三 编译运行

编译并运行 proj5 的命令如下：

```
make
```

```
make qemu
```

则可以得到如下显示界面

```
chenyu@chenyu-laptop:~/oscourse/ucore-svn/lab2_memory/proj5$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc010002c (phys)
  etext 0xc010537f (phys)
  edata 0xc01169b8 (phys)
  end    0xc01178dc (phys)
Kernel executable memory footprint: 95KB
memory managment: default_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- BEGIN -----
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----

++ setup timer interrupts
100 ticks
100 ticks
.....
```

通过上图，我们可以看到 ucore 在显示其 entry(入口地址)、etext(代码段截止处地址)、edata(数据段截止处地址)、和 end(ucore 截止处地址)的值后，探测出计算机系统物理内存的布局(e820map 下的显示内容)。接下来 ucore 会以页为最小分配单位实现一个简单的内存分配管理，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最

后显示 ucore 建立好的二级页表内容，并在分页模式下响应时钟中断。下面我们将分析到底发生了什么事情。

### 3.9.3 【背景】探测计算机系统物理内存分布和大小

在 proj5 中，操作系统需要知道了解整个计算机系统物理内存如何分布的，哪些被可用，哪些不可用。其基本方法是通过 BIOS 中断调用来帮助完成的。其中 BIOS 中断调用必须在实模式下进行，所以在 bootloader 进入保护模式前完成这部分工作相对比较合适。这部分由 boot/bootasm.S 中从 probe\_memory 处到 finish\_probe 处的代码部分完成完成。通过 BIOS 中断获取内存可用参数为 e820h 的 INT 15h BIOS 中断。BIOS 通过系统内存映射地址描述符（Address Range Descriptor）格式来表示系统物理内存布局，其具体表示如下：

| Offset | Size | Description           |           |
|--------|------|-----------------------|-----------|
| 00h    | 8 字节 | base address          | #系统内存块基地址 |
| 08h    | 8 字节 | length in bytes       | #系统内存大小   |
| 10h    | 4 字节 | type of address range | #内存类型     |

看下面的(Values for System Memory Map address type)88

Values for System Memory Map address type:

|       |                                                                           |
|-------|---------------------------------------------------------------------------|
| 01h   | memory, available to OS                                                   |
| 02h   | reserved, not available (e.g. system ROM, memory-mapped device)           |
| 03h   | ACPI Reclaim Memory (usable by OS after reading ACPI tables)              |
| 04h   | ACPI NVS Memory (OS is required to save this memory between NVS sessions) |
| other | not defined yet -- treat as Reserved                                      |

INT15h BIOS 中断的详细调用参数:

- eax: e820h: INT 15 的中断调用参数;
- edx: 534D4150h (即 4 个 ASCII 字符“SMAP”)，这只是一个签名而已;
- ebx: 如果是第一次调用或内存区域扫描完毕，则为 0。如果不是，则存放上次调用之后的计数值;
- ecx: 保存地址范围描述符的内存大小,应该大于等于 20 字节;
- es:di: 指向保存地址范围描述符结构的缓冲区，BIOS 把信息写入这个结构的起始地址。

此中断的返回值为:

- cflags 的 CF 位: 若 INT 15 中断执行成功，则不置位，否则置位;
- eax: 534D4150h ('SMAP') ;
- es:di: 指向保存地址范围描述符的缓冲区,此时缓冲区内的数据已由 BIOS 填写完毕
- ebx: 下一个地址范围描述符的计数地址
- ecx : 返回 BIOS 往 ES:DI 处写的地址范围描述符的字节大小
- ah: 失败时保存出错代码

这样，我们通过调用 INT 15h BIOS 中断，递增 di 的值（20 的倍数），让 BIOS 帮我们找出一个一个的内存布局 entry，并放入到一个保存地址范围描述符结构的缓冲区中，供后续的

ucore 进一步进行物理内存管理。这个缓冲区结构定义在 memlayout.h 中：

```
struct e820map {
    int nr_map;
    struct {
        long long addr;
        long long size;
        long type;
    } map[E820MAX];
};
```

### 3.9.4 【实现】物理内存探测

物理内存探测是在 bootasm.S 中实现的，相关代码很短，如下所示：

```
probe_memory:
    //对 0x8000 处的 32 位单元清零,即给位于 0x8000 处的
    //struct e820map 的结构域 nr_map 清零
    movl $0, 0x8000
    xorl %ebx, %ebx
    //表示设置调用 INT 15h BIOS 中断后，BIOS 返回的映射地址描述符的起始地址
    movw $0x8004, %di
start_probe:
    movl $0xE820, %eax // INT 15 的中断调用参数
    //设置地址范围描述符的大小为 20 字节，其大小等于 struct e820map 的结构域 map 的大
    //小
    movl $20, %ecx
    //设置 edx 为 534D4150h (即 4 个 ASCII 字符“SMAP”)，这是一个约定
    movl $SMAP, %edx
    //调用 int 0x15 中断，要求 BIOS 返回一个用地址范围描述符表示的内存段信息
    int $0x15
    //如果 eflags 的 CF 位为 0，则表示还有内存段需要探测
    jnc cont
    //探测有问题，结束探测
    movw $12345, 0x8000
    jmp finish_probe
cont:
    //设置下一个 BIOS 返回的映射地址描述符的起始地址
    addw $20, %di
    //递增 struct e820map 的结构域 nr_map
    incl 0x8000
    //如果 INT0x15 返回的 ebx 为零，表示探测结束，否则继续探测
    cmpl $0, %ebx
    jnz start_probe
finish_probe:
```

---

上述代码正常执行完毕后，在 0x8000 地址处保存了从 BIOS 中获得的内存分布信息，此信息按照 struct e820map 的设置来进行填充。这部分信息将在 bootloader 启动 ucore 后，由 ucore 的 page\_init 函数来根据 struct e820map 的 memmap（定义了起始地址为 0x8000）来完成对整个机器中的物理内存的总体管理。

### 3.9.5 【原理】分页内存管理

在分页内存管理中，一方面把实际物理内存（也称主存）划分为许多个固定大小的内存块，称为物理页面，或者是页框（page frame）；另一方面又把 CPU（包括程序员）看到的虚拟地址空间也划分为大小相同的块，称为虚拟页面，或者简称为页面、页（page）。页面的大小要求是 2 的整数次幂，一般在 256 个字节到 4M 字节之间。在本书中，页面的大小设定为 4KB。在 32 位的 86x86 中，虚拟地址空间是 4GB，物理地址空间也是 4GB，因此在理论上程序可访问到 1M 个虚拟页面和 1M 个物理页面。软件的每一物理页面都可以放置在主存中的任何地方，分页系统（需要 CPU 等硬件系统提供相应的分页机制硬件支持，详见下一节）提供了程序中使用的虚地址和主存中的物理地址之间的动态映射。这样当程序访问一个虚拟地址时，支持分页机制的相关硬件自动把 CPU 访问的虚拟地址拆分为页号（可能有多级页号）和页内偏移量，再把页号映射为页帧号，最后加上页内偏移组成一个物理地址，这样最终完成对这个地址的读/写/执行等操作。

假设程序在运行时要读地址 0x100 的内容到寄存器 1（用 REG1 表示）中，执行如下的指令：

```
mov 0x100, REG1
```

虚拟地址 0x100 被发送给 CPU 内部的内存管理单元（MMU），然后 MMU 通过支持分页机制的相关硬件逻辑就会把这个虚拟地址是位于第 0 个虚拟页面当中（设页大小为 4KB），页内偏移是 0x100；而操作系统的分页管理子系统已经设置好第 0 个虚拟页面对应的是第 2 个物理页帧，物理页帧的起始地址是 0x2000，然后再加上页内的偏移地址 0x100，所以最后得到的物理地址就是 0x2100。然后 MMU 就会把这个真正的物理地址发送到计算机系统上的地址总线上，从而可正确访问相应的物理内存单元。

如果操作系统的分页管理子系统没有设置第 0 个虚拟页面对应的物理页帧，则表示第 0 个虚拟页面当前没有对应的物理页帧，这会导致 CPU 产生一个缺页异常，由操作系统的缺页处理服务例程来选择如何处理。如果缺页处理服务例程认为这是一次非法访问，它将报错，终止软件运行；如果它认为是一次合理的访问，则它会采用分配物理页等手段建立正确的页映射，使得能够重新正确执行产生异常的访存指令。

### 3.9.6 【背景】X86 的分页硬件支持

X86 CPU 对实际物理内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的，在前端总线上传输的内存地址是物理内存地址。物理内存地址被北桥映射到实际的内存条中的内存单元相应位置上。然而，在 CPU 内执行带来软件所使用的是虚拟内存地址（也称逻

辑内存地址)，它必须被转换成物理地址后，才能用于实际内存访问。

前面已经讲过了 80x86 的分段机制，80x86 的分页机制建立在其分段机制基础之上，提供了更加强大的内存管理支持。需要注意的是，在 x86 中，必须先有分段机制，才能有分页机制。在分段机制中，虚地址会转换为线性地址。如果不启动分页机制，那么线性地址就是最终在前端总线上的物理地址；如果启动了分页机制，则线性地址还会经过页映射被转换为物理地址。

那如果启动分页机制呢？在 80x86 中有一个 CR0 控制寄存器，它包含一个 PG 位，如果 PG=1，启用分页机制；如果 PG=0，禁用分页机制。不像分段机制管理大小不固定的内存卡，分页机制以固定大小的存储块为最小管理单位，即把整个地址空间（包括线性地址和物理地址）都看成由固定大小的存储块组成。在 80x86 中，这个固定大小一般设定为 4096 字节。在线性地址空间中的最小管理单位（称为页（page）），可以映射到物理地址空间中的任何一个最小管理单位（称为页帧（page frame））。页/页帧的 32 位地址由 20 位的页号/页帧号和 12 位的页/页帧内偏移组成。

80x86 分页机制中的分页转换功能（即线性地址到物理地址的映射功能）需采用驻留在内存中的数组来描述，该数组称为页表（page table）。每个数组项就是一个页表项。由于页/页帧基地址按 4096 字节对齐，因此页/页帧的基地址的低 12 位是 0。页地址<->页帧地址的转换过程以简单地看做 80x86 对页表的一个查找过程。页地址（线性地址）的高 20 位（即页号，or 页的基地址）构成这个数组的索引值，用于选择对应页帧的页帧号（即页帧的基地址）。页地址的低 12 位给出了页内偏移量，加上对应的页帧基地址就最终形成对应的页帧地址（即物理地址）。

由于 80x86 的地址空间可达到 4GB，按页大小（4KB）划分为 1M 个页。如果用一个页表来描述这种映射，那么该表就要有 1M 个表项，若每个表项占用 4 个字节，那么该映射表就要占用 4M 字节。考虑到将来一个进程就需要一个地址映射表，若有多个进程，那地址映射表所占的总空间将非常巨大。为避免地址映射表占用过多的内存资源，80x86 把地址映射表设定为两级。地址映射表的第一级称为页目录表，存储在一个 4KB 的物理页中，页目录表共有 1K 个表项，其中每个表项为 4 字节长，页表项中包含对应第二级表所在的基地址。地址映射表的第二级称为页表，每个页表也安排在一个 4K 字节的页中，每张页表中有 1K 个表项，每个表项为 4 字节长，包含对应页帧的基地址。由于页目录表和页表均由 1K 个表项组成，所以使用 10 位的索引就能指定表项，即用 10 位的索引值乘以 4 加基地址就得到了表项的物理地址。按上述的地址转换描述，一个页表项只需 20 位，但实际的页表项是 32 位，那其他的 12 位有何用途呢？

在 80x86 中的页目录表项结构定义如下所示：

|        | 7          | 6 | 5 | 4   | 3    | 2   | 1   | 0 |
|--------|------------|---|---|-----|------|-----|-----|---|
| 7~0位   | PSE        | 0 | A | PCD | PWT  | U/S | R/W | P |
| 15~8位  | 3~0位页表地址   |   |   |     | OS专用 |     |     | 0 |
| 23~16位 | 11~4位页表地址  |   |   |     |      |     |     |   |
| 31~24位 | 19~12位页表地址 |   |   |     |      |     |     |   |

在 80x86 中的页表项结构定义如下所示：



|        |            |   |   |     |      |     |     |   |
|--------|------------|---|---|-----|------|-----|-----|---|
|        | 7          | 6 | 5 | 4   | 3    | 2   | 1   | 0 |
| 7~0位   | 0          | D | A | PCD | PWT  | U/S | R/W | P |
| 15~8位  | 3~0位页面地址   |   |   |     | OS专用 |     |     | 0 |
| 23~16位 | 11~4位页面地址  |   |   |     |      |     |     |   |
| 31~24位 | 19~12位页面地址 |   |   |     |      |     |     |   |

其中低 12 位的相应属性位含义如下：

- **P 位：**存在（Present）标志，用于指明此表项是否有效。P=1 表示有效；P=0 表示无效。如果 80x86 访问一个无效的表项，则会产生一个异常。如果 P=0，那么除表示表项无效外，其余位用于其他用途（比如 swap in/out 中，用来保存已存储在磁盘上的页面的序号）。
- **R/W：**读/写（Read/Write）标志，如果 R/W=1，表示页的内容可以被读、写或执行。如果 R/W=0，表示页的内容只读或可执行。当处理器运行在特权级（级别 0、1 或 2）时，则 R/W 位不起作用。
- **U/S：**是用户态/特权态（User/Supervisor）标志。如果 U/S=1，那么在用户态和特权态都可以访问该页。如果 U/S=0，那么只能在特权态（0、1 或 2）可访问该页。
- **A：**是已访问（Accessed）标志。当 CPU 访问页表项映射的物理页时，页表项的这个标志就会被置为 1。可通过软件把该标志位清零，并且操作系统可通过该标志来统计页的使用情况，用于页替换策略。
- **D：**是页面已被修改（Dirty）标志。当 CPU 写页表项映射的物理页内容时，页表项的这个标志就会被置为 1。可通过软件把该标志位清零，并且操作系统可通过该标志来统计页的修改情况，用于页替换策略。

下图显示了由页目录表和页表构成的二级页表映射架构。

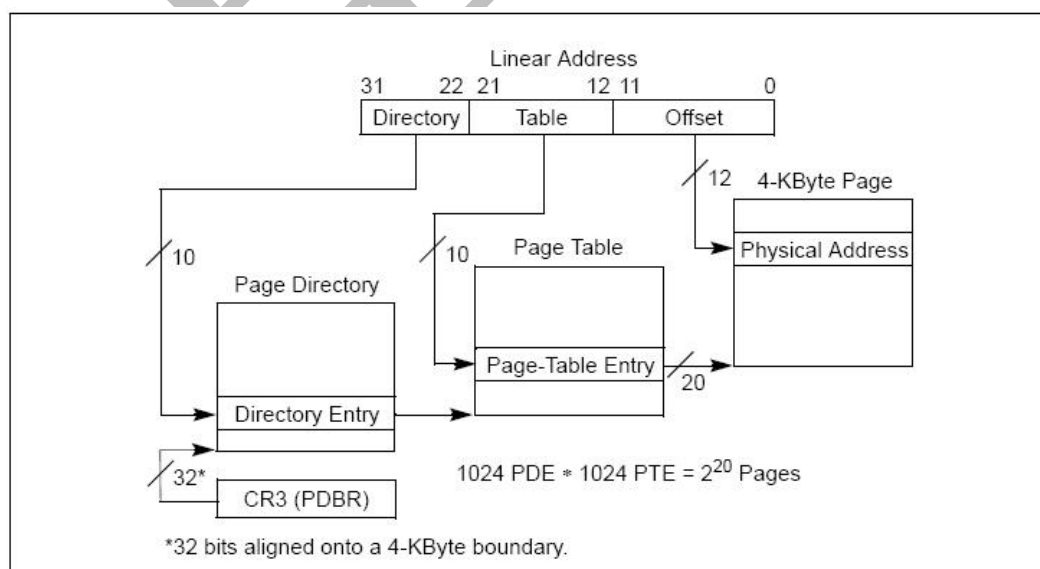


图 页目录表和页表构成的二级页表映射架构

从图中可见，控制寄存器 CR3 的内容是对应页目录表的物理基地址；页目录表可以指定 1K 个页表，这些页表可以分散存放在任意的物理页中，而不需要连续存放；每张页表可以指定 1K 个任意物理地址空间的页。存储页目录表和页表的基地址是按 4KB 对齐。当采用上述页表结构后，基于分页的线性地址到物理地址的转换过程如下图所示：

首先，CPU 把控制寄存器 CR3 的高 20 位作为页目录表所在物理页的物理基地址，再把需要进行地址转换的线性地址的最高 10 位(即 22~ 31 位)作为页目录表的索引，查找到对应的页目录表项，这个表项中所包含的高 20 位是对应的页表所在物理页的物理基地址；然后，再把线性地址的中间 10 位(即 12~21 位)作为页表中的页表项索引，查找到对应的页表项，这个表项所包含的高 20 位作为线性地址的基地址(即页号)对应的物理地址的基地址(即页帧号)；最后，把页帧号作为 32 位物理地址的高 20 位，把线性地址的低 12 位不加改变地作为 32 位物理地址的低 12 位，形成最终的物理地址。

如果每次访问内存单元都要访问位于内存中的页表，则访存开销太大。为了避免这类开销，x86 CPU 把最近使用的地址映射数据存储在内部的页转换高速缓存(页转换查找缓存，简称 TLB)中。这样在访问存储器页表之前总是先查阅高速缓存，仅当必须的转换不在高速缓存中时，才访问存储器中的两级页表。

### 3.9.7 【实现】实现分页内存管理

#### 一 重新建立段映射

前面已经介绍了如何探测物理内存，接下来 ucore 需要根据物理内存的情况来建立分页管理机制。首先观察一下 tools/kernel.ld 文件在 proj4.1 和 proj5 中的区别，在 proj4.1 中：

```
ENTRY(kern_init)

SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0x100000;

    .text : {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
}
```

在 proj5 中：

```
ENTRY(kern_entry)

SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0xC0100000;

    .text : {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
}
```

这意味着 gcc 编译出 ucore 的起始地址从 0xC0100000 开始，入口函数为 kern\_entry 函数。

这与 proj4.1 有很大差别。这实际上说明 ucore 在建立好页映射关系后，虚拟地址空间和物理地址空间之间存在如下的映射关系：

Virtual Address=LinearAddress=0xC0000000+Physical Address

另外，ucore 的入口地址也改为了 kern\_entry 函数，这个函数位于 init/entry.S 中，分析代码可以看出，entry.S 重新建立了段映射关系，从以前的

Virtual Address= Linear Address

改为

Virtual Address=Linear Address-0xC0000000

由于 gcc 编译出的虚拟起始地址从 0xC0100000 开始，ucore 被 bootloader 放置在从物理地址 0x100000 处开始的物理内存中。所以当 kern\_entry 函数完成新的段映射关系后，且 ucore 在没有建立好页映射机制前，CPU 按照 ucore 中的虚拟地址执行，能够被分段机制映射到正确的物理地址上，确保 ucore 运行正确。

## 二 初始化物理内存页分配管理

为了与以后的分页机制配合，我们首先需要建立对整个计算机的页级物理内存分配管理。这部分代码的实现在 kern/default\_pmm.[ch]。首先我们需要用一个数据结构来描述每个物理页（也称页帧），这里用了双向链表结构来表示每个页。链表头用 free\_area\_t 结构来表示，包含了一个 list\_entry 结构的双向链表指针和记录当前空闲页的个数的无符号整型变量 nr\_free。

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;           // the list header
    unsigned int nr_free;             // # of free pages in this free list
} free_area_t;
```

每一个物理页的属性用结构 Page 来表示，它包含了映射此物理页的虚拟页个数，描述物理页属性的 flags 和双向链接各个 Page 结构的 page\_link 双向链表。

```
struct Page {
    atomic_t ref; // page frame's reference counter
    uint32_t flags; // array of flags that describe the status of the page frame
    list_entry_t page_link; // free list link
};
```

有了这两个数据结构，ucore 就可以管理起来整个以页为单位的物理内存空间。接下来需要解决两个问题：

- 管理页级物理内存空间所需的 Page 结构的内存空间从哪里开始，占多大空间？
- 空闲内存空间的起始地址在哪里？

对于这两个问题，我们首先根据 bootloader 给出的内存布局信息找出最大的物理内存地址 maxpa（定义在 page\_init 函数中的局部变量），由于 x86 的起始物理内存地址为 0，所以可以得知需要管理的物理页个数为

$$npage = maxpa / PGSIZE$$

这样，我们就可以预估出管理页级物理内存空间所需的 Page 结构的内存空间所需的内存大小为：

```
sizeof(struct Page) * npage)
```

由于 bootloader 加载 ucore 的结束地址（用全局指针变量 end 记录）以上的空间没有被使用，所以我们可以把 end 按页大小为边界去整后，作为管理页级物理内存空间所需的 Page 结构的内存空间，记为：

```
pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
```

为了简化起见，从地址 0 到地址 pages+ sizeof(struct Page) \* npage)结束的物理内存空间设定为已占用物理内存空间（起始 0~640KB 的空间是空闲的），地址 pages+ sizeof(struct Page) \* npage)以上的空间为空闲物理内存空间，这时的空闲空间起始地址为

```
uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);
```

为此我们需要把这两部分空间给标识出来。对于已占用物理空间，通过如下语句即可实现占用标记：

```
for (i = 0; i < npage; i++) {  
    SetPageReserved(pages + i);  
}
```

对于空闲物理空间，通过如下语句即可实现空闲标记：

```
//获得空闲空间的起始地址 begin 和结束地址 end
```

```
.....
```

```
init_memmap(pa2page(begin), (end - begin) / PGSIZE);
```

其实 SetPageReserved 只需把物理地址对应的 Page 结构中的 flags 标志设置为 PG\_reserved，表示这些页已经被使用了。而 init\_memmap 函数则是把空闲物理页对应的 Page 结构中的 flags 和引用计数 ref 清零，并加到 free\_area.free\_list 指向的双向列表中，为将来的空闲页管理做好初始化准备工作。

### 三 物理内存页分配与释放

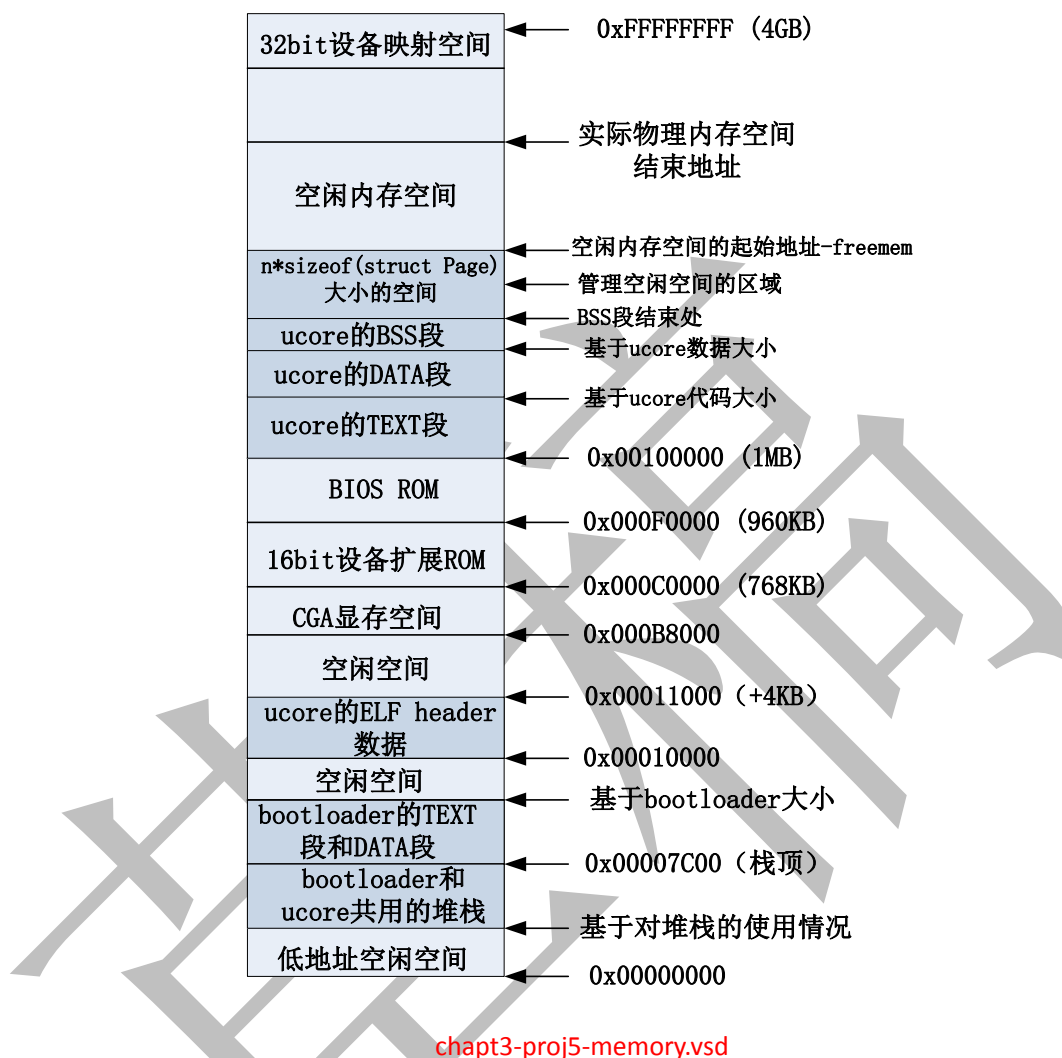
关于内存分配的操作系统原理方面的知识有很多，但在 proj5 中只实现了最简单的内存页分配算法，即每次只分配一页或释放一页的内存页分配算法。相应的实现在 default\_pmm.c 中的 default\_alloc\_pages 函数和 default\_free\_pages 函数，相关实现很简单，这里就不具体分析了，直接看源码，应该很好理解。

其实 proj5 在内存分配和释放方面最主要的作用是建立了一个物理内存页管理器框架，这实际上是一个函数指针列表，定义如下：

```
struct pmm_manager {  
    const char *name; //物理内存页管理器的名字  
    void (*init)(void); //初始化内存管理器  
    void (*init_memmap)(struct Page *base, size_t n); //初始化管理空闲内存页的数据结构  
  
    struct Page *(*alloc_pages)(size_t n); //分配 n 个物理内存页  
    void (*free_pages)(struct Page *base, size_t n); //释放 n 个物理内存页  
    size_t (*nr_free_pages)(void); //返回当前剩余的空闲页数
```

```
void (*check)(void); //用于检测分配/释放实现是否正确的辅助函数
};
```

重点是实现 `init_memmap/ alloc_pages/ free_pages` 这三个函数。当完成物理内存页管理初始化工作后，计算机系统的内存布局如下图所示：



读者可进一步通过分析 `proj5.1/5.1.1/5.1.2/5.2` 中 `firstfit_pmm[ch]/bestfit_pmm[ch]/worstfit_pmm[ch]/ buddy_pmm[ch]` 文件中对应函数实现来体会原理课中的连续空间内存分配中各种分配算法的设计思路和实现。

## 四 建立二级页表

为了实现分页机制，需要建立好虚拟内存和物理内存的页映射关系，即建立二级页表。这需要解决如下问题：

- 对于哪些物理内存空间需要建立页映射关系？
- 具体的页映射关系是什么？
- 页目录表的起始地址设置在哪里？
- 页表的起始地址设置在哪里，需要多大空间？

- 如何设置页目录表项的内容？
- 如何设置页目录表项的内容？

下面我们逐一解决上述问题。由于物理内存页管理器管理了从 0 到实际可用物理内存大小的物理内存空间，所以对于这些物理内存空间都需要建立好页映射关系。由于目前 ucore 只运行在内核空间，所以可以建立一个一一映射关系。假定虚拟内核地址的起始地址为 0xC0000000，这虚拟内存和物理内存的具体页映射关系为：

$$\text{Virtual Address} = \text{Physical Address} + 0xC0000000$$

由于我们已经具有了一个物理内存页管理器 default\_pmm\_manager，我们就可以用它来获得所需的空闲物理页。在二级页表结构中，页目录表占 4KB 空间，ucore 就可通过 default\_pmm\_manager 的 default\_alloc\_pages 函数获得一个空闲物理页，这个页的起始物理地址就是页目录表的起始地址。同理，ucore 也通过这种方式获得各个页表所需的物理页。页表的空间大小取决于页表要管理的物理页数 n，一个页表项（32 位，即 4 字节）可管理一个物理页，页表需要占 n/256 个物理页空间。这样页目录表和页表所占的总大小为 4096+1024\*n 字节。

为把 0~KERN\_SIZE（明确 ucore 设定实际物理内存不能超过 KERN\_SIZE 值，即 0x38000000 字节，896MB，3670016 个物理页）的物理地址一一映射到页目录表项和页表项的内容，其大致流程如下：

1. 先通过 default\_pmm\_manager 获得一个空闲物理页，用于页目录表；
2. 调用 boot\_map\_segment 函数建立一一映射关系，具体处理过程以页为单位进行设置，即

$$\text{Virtual Address} = \text{Physical Address} + 0xC0000000$$

- a) 设一个逻辑地址 la（按页对齐，故低 12 位为零）对应的物理地址 pa（按页对齐，故低 12 位为零），如果在页目录表项（la 的高 10 位为索引值）中的存在位（PTE\_P）为 0，表示缺少对应的页表空间，则可通过 default\_pmm\_manager 获得一个空闲物理页给页表，页表起始物理地址是按 4096 字节对齐的，这样填写页目录表项的内容为

$$\text{页目录表项内容} = \text{页表起始物理地址} \mid \text{PTE\_U} \mid \text{PTE\_W} \mid \text{PTE\_P}$$

进一步对于页表中对应页表项（la 的中 10 位为索引值）的内容为

$$\text{页表项内容} = \text{pa} \mid \text{PTE\_P} \mid \text{PTE\_W}$$

其中：

- PTE\_U：位 3，表示用户态的软件可以读取对应地址的物理内存页内容
- PTE\_W：位 2，表示物理内存页内容可写
- PTE\_P：位 1，表示物理内存页存在

建立好一一映射的二级页表结构后，接下来就要使能分页机制了，这主要是通过 enable\_paging 函数实现的，这个函数主要做了两件事：

1. 通过 lcr3 指令把页目录表的起始地址存入 CR3 寄存器中；
2. 通过 lcr0 指令把 cr0 中的 CR0\_PG 标志位设置上。

执行完 enable\_paging 函数后，计算机系统进入了分页模式！但到这一步还不够，还记得 ucore 在最开始通过 kern\_entry 函数设置了临时的新段映射机制吗？这个临时的新段映射机制不是最简单的对等映射，导致虚拟地址和线性地址不相等。而刚才建立的页映射关系是建立在简单的段对等映射，即虚拟地址=线性地址的假设基础之上的。所以我们需要进一步调整段映射关系，即重新设置新的 GDT，建立对等段映射。

这里需要注意：在进入分页模式到重新设置新 GDT 的过程是一个过渡过程。在这个过渡过程中，已经建立了页表机制，所以通过现在的段机制和页机制实现的地址映射关系为：

Virtual Address=Linear Address + 0xC0000000 = Physical Address +0xC0000000+0xC0000000

在这个特殊的阶段，如果不把段映射关系改为 Virtual Address = Linear Address，则通过段页式两次地址转换后，无法得到正确的物理地址。为此我们需要进一步调用 gdt\_init 函数，根据新的 gdt 全局段描述符表内容（gdt 定义位于 pmm.c 中），恢复以前的段映射关系，即使得 Virtual Address = Linear Address。这样在执行完 gdt\_init 后，通过的段机制和页机制实现的地址映射关系为：

Virtual Address=Linear Address = Physical Address +0xC0000000

这里存在的一个问题是，在调用 enable\_page 函数使能分页机制后到执行完毕 gdt\_init 函数重新建立好段页式映射机制的过程中，内核使用的还是旧的段表映射，也就是说，enable paging 之后，内核使用的是页表的低地址 entry。如何保证此时内核依然能够正常工作呢？其实只需让低地址目录表项的内容等于以 KERNBASE 开始的高地址目录表项的内容即可。目前内核大小不超过 4M（实际上是 3M，因为内核从 0x100000 开始编址），这样就只需要让页表在 0~4MB 的线性地址与 KERNBASE ~ KERNBASE+4MB 的线性地址获得相同的映射即可，都映射到 0~4MB 的物理地址空间，具体实现在 pmm.c 中 pmm\_init 函数的语句：

```
boot_pgdir[0] = boot_pgdir[PDX(KERNBASE)];
```

实际上这种映射也限制了内核的大小。当内核大小超过预期的 3MB 就可能导致打开分页之后内核 crash，在后面的试验中，也的确出现了这种情况。解决方法同样简单，就是拷贝更多的高地址项到低地址。

当执行完毕 gdt\_init 函数后，新的段页式映射已经建立好了，上面的 0~4MB 的线性地址与 0~4MB 的物理地址一一映射关系已经没有用了。所以可以通过如下语句解除这个老的映射关系。

```
boot_pgdir[0] = 0;
```

## 五 自映射机制

上一小节讲述了通过 boot\_map\_segment 函数建立了基于一一映射关系的页目录表项和页表项，这里的映射关系为：

Virtual addr (KERNBASE~KERNBASE+KMEMSIZE) = Physical\_addr (0~KMEMSIZE)

这样只要给出一个虚地址和一个物理地址，就可以设置相应 PDE 和 PTE，就可完成正确的映射关系。

如果我们这时需要按虚拟地址的地址顺序显示整个页目录表和页表的内容，则要查找页目录表的页目录表项内容，根据页目录表项内容找到页表的物理地址，再转换成对应的虚地址，然后访问页表的虚地址，搜索整个页表的每个页目录项。这样过程比较繁琐。我们有没有一个简洁的方法来实现这个查找呢？ucore 做了一个很巧妙的地址自映射设计，把页目录表和页表放在一个连续的 4MB 虚拟地址空间中，并设置页目录表自身的虚地址<-->物理地址映射关系。这样在已知页目录表起始虚地址的情况下，通过连续扫描这特定的 4MB 虚拟地址空间，就很容易访问每个页目录表项和页表项内容。

具体而言，ucore 是这样设计的，首先设置了一个常量（memlayout.h）：

VPT=0xFAC00000, 这个地址的二进制表示为:

1111 1010 1100 0000 0000 0000 0000

高 10 位为 1111 1010 11, 即 10 进制的 1003, 中间 10 位为 0, 低 12 位也为 0。在 pmm.c 中有两个全局初始化变量

```
pte_t * const vpt = (pte_t *)VPT;
pde_t * const vpd = (pde_t *)PGADDR(PDX(VPT), PDX(VPT), 0);
并在 pmm_init 函数执行了如下语句:
boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;
```

这些变量和语句有何特殊含义呢? 其实 vpd 变量的值就是页目录表的起始虚地址 0xFAFEB000, 且它的高 10 位和中 10 位是相等的, 都是 10 进制的 1003。当执行了上述语句, 就确保了 vpd 变量的值就是页目录表的起始虚地址, 且 vpt 是页目录表中第一个目录表项指向的页表的起始虚地址。此时描述内核虚拟空间的页目录表的虚地址为 0xFAFEB000, 大小为 4KB。页表的理论连续虚拟地址空间 0xFAC00000~0xFB000000, 大小为 4MB。因为这个连续地址空间的大小为 4MB, 可有 1M 个 PTE, 即可映射 4GB 的地址空间。

但 ucore 实际上不会用完这么多项, 在 memlayout.h 中定义了常量

```
#define KMEMSIZE 0x38000000
```

表示 ucore 只支持 896MB 的物理内存空间, 这个 896MB 只是一个设定, 可以根据情况改变。则最大的内核虚地址为常量

```
#define KERNTOP (KERNBASE + KMEMSIZE)=0xF8000000
```

所以最大内核虚地址 KERNTOP 的页目录项虚地址为

```
vpd+0xF8000000/0x400000=0xFAFEB000+0x3E0=0xFAFEB3E0
```

最大内核虚地址 KERNTOP 的页表项虚地址为:

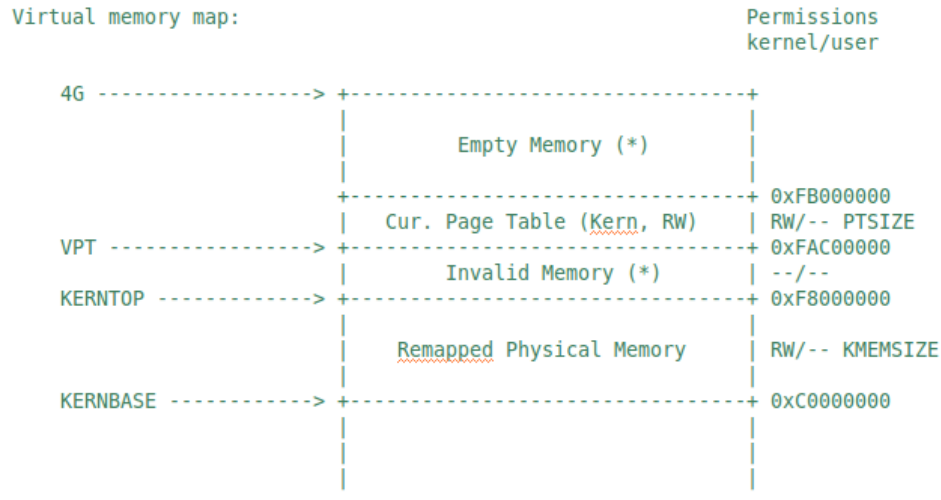
```
vpt+0xF8000000/0x1000=0xFAC00000+0xF8000=0xFACF8000
```

在 pmm.c 中的函数 print\_pgdir 就是基于 ucore 的页表自映射方式完成了对整个页目录表和页表的内容扫描和打印。注意, 这里不会出现某个页表的虚地址与页目录表虚地址相同的情况。

自映射机制还可方便用户态程序访问页表。因为页表是内核维护的, 用户程序很难知道自己页表的映射结构。VPT 实际上在内核地址空间的, 我们可以用同样的方式实现一个用户地址空间的映射 (比如 pgdir[UVPT] = PADDR(pgdir) | PTE\_P | PTE\_U, 注意, 这里不能给写权限, 并且 pgdir 是每个进程的 page table, 不是 boot\_pgdir), 这样, 用户程序就可以用和内核一样的 print\_pgdir 函数遍历自己的页表结构了。

在 page\_init 函数建立完实现物理内存一一映射和页目录表自映射的页目录表和页表后, 一旦使能分页机制, 则 ucore 看到的内核虚拟地址空间如下图所示:





proj5 使能分页机制后的虚拟地址空间图

### 3.9.8 【原理】页内存分配算法

在 proj5 中进行在动态分配内存时，存在很多限制，效率很低。在操作系统原理中，为了有效地分配内存，首先需要了解和跟踪空闲内存和分布情况，一般可采用位图（bit map）和双向链表两种方式跟踪内存使用情况。若采用位图方式，则每个页对应位图区域的一个 bit，如果此位为 0，表示空闲，如果为 1，表示被占用。采用位图方式很省空间，但查找 n 个长度为 0 的位串的开销比较大。而双向链表在查询或修改操作方面灵活性和效率较高，所以 ucore 采用双向链表来跟踪跟踪内存使用情况。

假设整个物理内存空闲空间的以页为单位被一个双向链表管理起来，每个表项管理一个物理页。这需要设计某种算法来查找空闲页和回收空闲页。ucore 实现了首次适配（first fit）算法、最佳适配（best fit）算法、最差适配（worst fit）算法和兄弟（buddy）算法，这些算法都可以实现在 ucore 提供的物理内存页管理器框架 pmm\_manager 下。

首次适配（first fit）算法的分配内存的设计思路是物理内存页管理器顺着双向链表进行搜索空闲内存区域，直到找到一个足够大的空闲区域，这是一种速度很快的算法，因为它尽可能少地搜索链表。如果空闲区域的大小和申请分配的大小正好一样，则把这个空闲区域分配出去，成功返回；否则将该空闲区分为两部分，一部分区域与申请分配的大小相等，把它分配出去，剩下的一部分区域形成新的空闲区。其释放内存的设计思路很简单，只需把这块区域重新放回双向链表中即可。

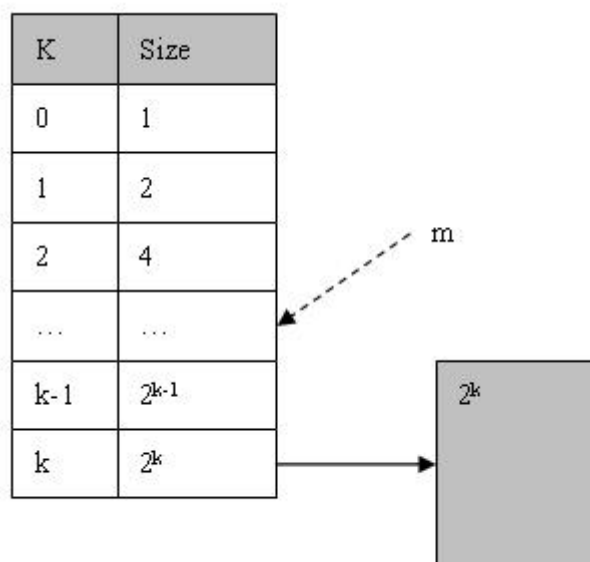
最佳适配（best fit）算法的设计思路是物理内存页管理器搜索整个双向链表（从开始到结束），找出能够满足申请分配的空间大小的最小空闲区域。找到这个区域后的处理以及释放内存的处理与上面类似。最佳适配算法试图找出最接近实际需要的空闲区，名字上听起来很好，其实在查询速度上较慢，且较易产生多的内存碎片。

最差适配（worst fit）算法与最佳适配（best fit）算法的设计思路相反，物理内存页管理器搜索整个双向链表，找出能够满足申请分配的空间大小的最大空闲区域，使新的空闲区比较大从而可以继续使用。在实际效果上，查询速度上也较慢，产生内存碎片相对少些。

上述三种算法在实际应用中都会产生碎片较多，效率不高的问题。为此一般操作系统会采用 buddy 算法来改进上述问题。buddy 算法的基本设计思想是：在 buddy 系统中，被占用的内存空间和空闲内存空间的大小均为 2 的 k 次幂（k 是正整数）。这样在 ucore 中，若申请 n 个页的内存空间，则实际可能分配的空间大小为  $2^k$  个页（ $2^{k-1} < n \leq 2^k$ ）。若初始化时的空

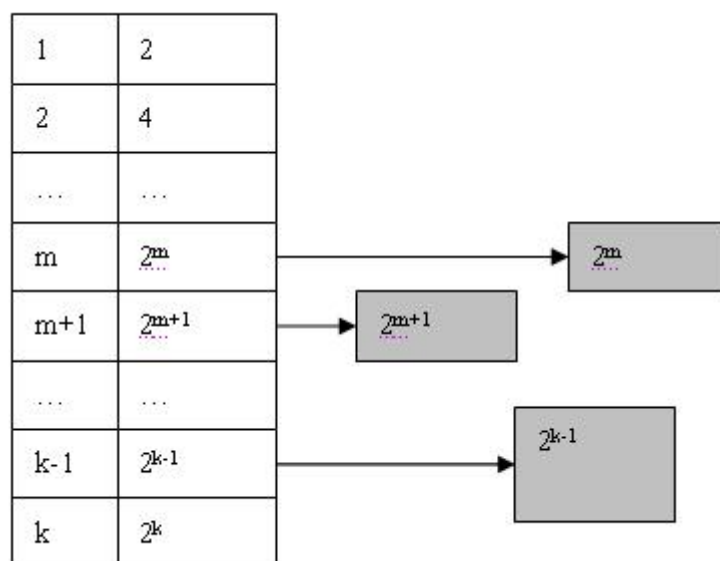
闲内存空间容量为  $2^m$  个页，这空闲块的大小只可能是  $2^0$ 、 $2^1$ 、...、 $2^m$  个页。

假定内存一开始是一个连续地址空间（大小为  $2^k$  个页）的大空闲块，且最小分配单位为 1 个页（4KB），则 buddy system 初始化时将生成一个长度为  $k+1$  的可用空间表 List，并将全部可用空间作为一个大小为  $2^k$  个页的空闲块 Bk 挂接在空闲块数组链表 List 的最后一个节点上，如下图：



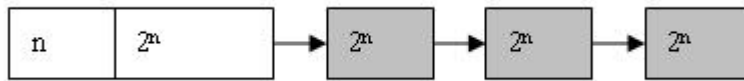
当 ucore 其他子系统申请  $n$  个字节的存储空间时，buddy system 分配的空闲块大小为  $2^m$  个页， $m$  满足条件： $2^{m-1} < n \leq 2^m$

此时 buddy system 将在 list 中的  $m$  位置寻找可用的空闲块。初始化时 List 中这个位置为空，于是 buddy system 就向上查找  $m+1$ ，...，直到达到  $k$  位置为止。找到  $k$  位置后，便得到可用空闲块 Bk，此时 Bk 将分裂成两个大小为  $2^{k-1}$  的空闲块 Bk-1a 和 Bk-1b，并将其中一个插入到 List 中  $k-1$  位置，同时对另外一个继续进行分裂。如此以往直到得到两个大小为  $2^m$  个页的块为止，并把其中一个空闲块分配给需求方。此时的内存如下图所示：



如果 buddy system 在运行一段时间之后，List 中某个位置  $t$  可能会出现多个块，则将其他块依次链接可用块链表的末尾。当 buddy system 要在  $t$  位置取可用块时，直接从链表头取

一个即可。



当一个存储块被释放时, buddy system 将把此内存块回收到空闲块链表 List 中。此时 buddy system 系统将根据此存储块的大小计算出其在 List 中的位置, 然后插入到空闲块链表的末尾。在这一步完成后, 系统立即开始合并尝试操作, 该操作是将地址相邻且大小相等的空闲块 (简称 buddy, 即"伙伴"空闲块) 合并到一起, 形成一个更大的空闲块, 并重新放到空闲块链表 List 的对应位置中, 并继续对更大的块进行合并, 直到无法合并为止。

严蔚敏老师的“数据结构”一书第 8 章第 4 节对 buddy 算法有详尽的解释, “understanding linux kernel”此书对此也有很好的描述, 读者可以进一步参考。

对于上述 4 个内存分配算法, 可参考对应的 proj5.1/5.1.1/5.1.2/5.2 中的 kern/mm/\*\_pmm.[ch]的具体实现来进一步了解。

(可以进一步描述三种算法的具体实现)

## 3.10 支持任意大小内存分配

### 3.10.1 试验目标

上一节描述了如何进行页级内存的分配与释放管理, 可以比较有效地完成以页大小为最小单位 (粒度) 的内存分配和回收工作, 这样可以很好地与分页机制配合在一起提供有效的分页管理。但在操作系统的实际运行过程中, 还有很多对小于一页的任意大小内存的动态申请需求, 则以页为最小单位就无法适应这类需求了。当然, 我们可以直接把物理内存页管理器改为粒度为 1 字节的物理内存管理器, 这主要存在两个问题:

- 由于页目录表和页表的大小是 4KB, 且一个页表项管理的内存空间大小也是 4KB, 故需要扩展新的函数和结构匹配对页表的管理支持, 导致代码复杂;
- 即使采用上述 4 种内存分配算法, 在支持任意大小的内存分配请求上, 依然存在效率不高, 有外碎片和内碎片等问题。

所以, 一个更加合理的办法是在物理内存页管理器的基础之上建立一个支持任意大小的内存分配管理器, 形成二级内存管理, 满足高效支持任意大小的内存分配请求。这就对动态内存分配管理提出了新的挑战, 即花尽量少的时间完成对内存的分配和回收, 且保证能够产生的内外碎片尽量小。

proj6 中参考 Jeff Bonwick 为 Sun OS 操作系统首次引入的一种算法: slab 算法。slab 算法的基本思路有两个, 一个是通过缓存实现“对象”重用, 另一个是在一个连续页空间放同样类型的“对象”。

Jeff Bonwick 在 SUN OS 内核中观察到内核在运行时会为有限的对象集 (内核中各种常见的数据结构) 分配大量内存, 且对内核中这些数据结构进行初始化所需的时间超过了对其进行分配和释放所需的时间。因此他的结论是**不应该将内存释放回一个全局的空闲内存池**, 而是将内存保持为针对特定目而初始化的状态。例如, 如果内存被分配给了一个变量, 那么只需在为此变量首次分配内存时执行一次变量初始化函数即可, 当该变量被回收并进一步被后续分配时就不需要执行这个初始化函数, 因为从上次释放和调

---

用析构之后，它已经处于所需的状态中了。

在一个连续地址空间放同样类型的“对象”有助于快速查找和修改同样类型的“对象”，提高分配的时间效率，并减少碎片。

### 3.10.2 proj6 概述

#### 一 实现描述

proj6 基于 proj5.2 实现，主要在 buddy 物理内存页管理器的基础上，增加了一级任意大小内存分配管理，通过 slab 算法实现对小内存的简洁分配，为后续的运行时动态内存管理提供通用的内存申请和释放接口、在 proj6 中，可以了解到：

- slab 算法的数据结构和具体实现；
- 小内存分配管理器与物理内存页管理器的接口和交互过程。

#### 二 项目组成

```
proj6
├── boot
│   └── .....
├── kern
│   ├── debug
│   │   └── .....
│   ├── driver
│   │   └── .....
│   ├── init
│   │   └── .....
│   ├── libs
│   │   └── .....
│   ├── mm
│   │   ├── .....
│   │   ├── memlayout.h
│   │   ├── slab.c
│   │   └── slab.h
│   ├── sync
│   │   └── .....
│   └── trap
│       └── .....
├── libs
│   └── .....
├── Makefile
└── .....
```

---

11 directories, 58 files

相对与 proj5.2, proj6 增加了 slab.[ch]两个文件, 主要完成对 slab 内存管理算法的简单实现。

### 三 编译运行

(THU.CST) os is loading ...

Special kernel symbols:

```
entry 0xc010002c (phys)
etext 0xc0109530 (phys)
edata 0xc0122aa0 (phys)
end    0xc0123cb8 (phys)
```

Kernel executable memory footprint: 144KB

memory managment: buddy\_pmm\_manager

e820map:

```
memory: 0009f400, [00000000, 0009f3ff], type = 1.
memory: 00000c00, [0009f400, 0009ffff], type = 2.
memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07efd000, [00100000, 07ffcfff], type = 1.
memory: 00003000, [07ffd000, 07ffffff], type = 2.
memory: 00040000, [fffc0000, ffffffff], type = 2.
```

check\_alloc\_page() succeeded!

check\_pgdir() succeeded!

check\_boot\_pgdir() succeeded!

```
----- BEGIN -----
PDE(0e0) c0000000-f8000000 38000000 urw
|-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----
```

check\_slab() succeeded!

++ setup timer interrupts

100 ticks

100 ticks

### 3.10.3 【实现】slab 算法的简化设计实现

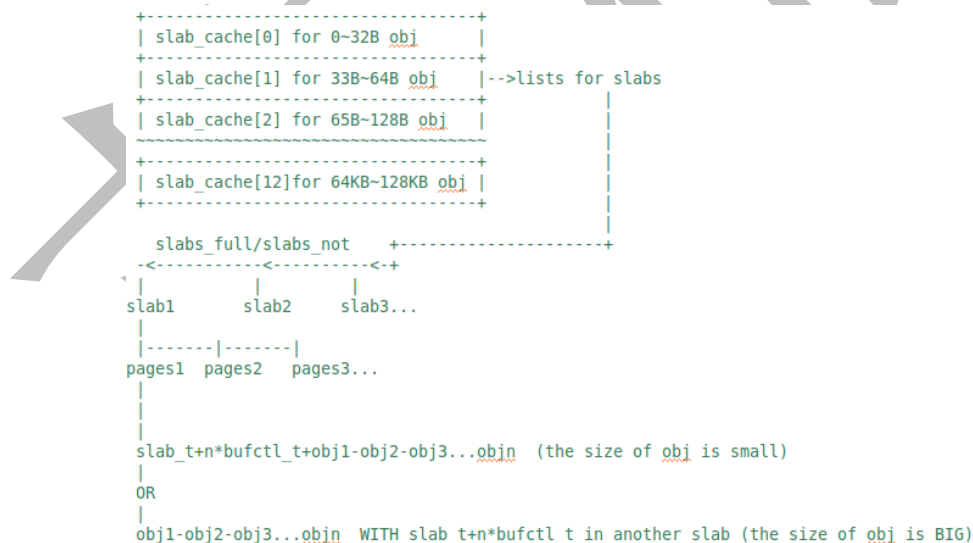
#### 一 数据结构描述

slab 算法采用了两层数据组织结构。在最高层是 `slab_cache`，这是一个不同大小 slab 缓存的链接列表数组。`slab_cache` 的每个数组元素都是一个管理和存储给定大小的空闲对象 (`obj`) 的 slab 结构链表，这样每个 slab 设定一个要管理的给定大小的对象池，占用物理空间连续的 1 个或多个物理页。`slab_cache` 的每个数组元素管理两种 slab 列表：

- `slabs_full`：完全分配的 slab
- `slabs_notfull`：部分分配的 slab

注意 `slabs_notfull` 列表中的 slab 是可以进行回收(reaping)，使得 slab 所使用的内存可被返回给操作系统供其他子系统使用。

slab 列表中的每个 slab 都是一个连续的内存块（一个或多个连续页），它们被划分成一个个 `obj`。这些 `obj` 是中进行分配和释放的基本元素。由于对象是从 slab 中进行分配和释放的，因此单个 slab 可以在 slab 链表之间进行移动。例如，当一个 slab 中的所有对象都被使用完时，就从 `slabs_notfull` 链表中移动到 `slabs_full` 链表中。当一个 slab 完全被分配并且有对象被释放后，就从 `slabs_full` 列表中移动到 `slabs_notfull` 列表中。下面是 ucore 中的 slab 架构图：



slab 架构图

#### 二 分配与释放内存实现描述

现在来看一下能够创建新 slab 缓存、向缓存中增加内存、销毁缓存的接口以及 slab 中对对象进行分配和释放操作的 slab 相关操作过程和函数。

第一个步骤是通过执行 `slab_init` 函数初始化 `slab_cache` 缓存结构。然后其他 slab 缓存函数将使用该引用进行内存分配与释放等操作。ucore 中最常用的内存管理函数是 `kmalloc` 和 `kfree` 函数。这两个函数的原型如下：

- `void *kmalloc( size_t size );`
- `void kfree(void *objp );`

在分配任意大小的空闲块时，`kmalloc` 通过调用 `kmem_cache_alloc` 函数来遍历对应 `size` 的 `slab`，来查找可以满足大小限制的缓存。如果 `kmem_cache_alloc` 函数发现 `slab` 中有空闲的 `obj`，则分配这个对象；如果没有空闲的 `obj` 了，则调用 `kmem_cache_grow` 函数分配包含 1 到多页组成的空闲 `slab`，然后继续分配。要使用 `kfree` 释放对象时，通过进一步调用 `kmem_cache_free` 把分配对象返回到 `slab` 中，并标记为空闲，建立空闲 `obj` 之间的链接关系。

(可进一步详细一些)

## 3.11 实现虚存管理功能

### 3.11.1 试验目标

有了页表的支持，我们可以使得不同用户态运行程序的内存空间之间无法访问，达到隔离和保护的作用。但页表如何仅仅只支持这个功能就太大材小用了。我们其实还可以通过页表实现更多的功能：

- 内存共享：把两个虚拟地址空间通过页表映射到同一物理地址空间。这只需通过设置不同索引的页表项的内容一致即可。
- 提供超过物理内存大小的虚拟内存空间：这一步需要结合异常中断处理和硬盘来完成。其基本思想是在内存中放置最常用的一些数据，不常用的数据会被放到硬盘上，但给用户态的软件一种感觉，觉得这些数据都在内存中。当用户态软件访问到的数据不在内存中（暂时存放在硬盘上）的时候，这条访存指令会引发异常中断，由操作系统的异常中断处理例程进行管理。这时操作系统会分析引发异常的内存地址，能够把对应缓存在硬盘中的数据重新读入这个内存地址，并让用户态软件重新执行产生访存异常的那条指令。这些由操作系统完成的工作在用户态完全“看”不到。从用户态软件的角度看，只是操作系统给用户提供了一个超出实际物理内存大小的虚拟内存空间。
- 按需分配内存：用户态软件在运行时要求操作系统提供很大的内存，操作系统“表面上”表示满足用户需求，但在背后并没有实际分配对应的物理内存空间。等到用户态软件实际执行到对这些内存的访问时，由于没有分配对应的物理内存空间，会导致产生访存异常。操作系统的异常中断处理例程发觉这是用户态软件以前确实要求过的内存空间，则在从系统管理的空闲空间中分配一页或几页物理内存给用户态软件，并让用户态软件重新执行产生访存异常的那条指令。这些由操作系统完成的工作在用户态也完全“看”不到。但从操作系统的整体管理的角度看，这种方式在用户态软件确实需要的时候把内存分配给用户态软件，提高了内存的使用率，避免了用户态软件“圈地不用”的现象。

为了高效地完成上述三件事情，操作系统需要考虑应该把哪些不常用的内存换出到硬盘上去，这就是内存的页替换算法，常见的有 LRU 算法，Clock 算法，二次机会法等。而在实现上，由于涉及异常处理和硬盘管理等，虚存管理在整个 `ucore` 实现中的相对复杂度是最大的。



### 3.11.2 【原理】虚拟内存管理

什么是虚拟内存？简单地说，是指程序员或 CPU “需要”和直接“看到”的内存，这其实暗示了两点：1、虚拟内存单元不一定有实际的物理内存单元对应，即实际的物理内存单元可能不存在；2、如果虚拟内存单元对应实际的物理内存单元，那二者的地址一般不是相等的。通过操作系统的某种内存管理和映射技术可建立虚拟内存与实际的物理内存的对应关系，使得程序员或 CPU 访问的虚拟内存地址会转换为另外一个物理内存地址。

那么这个“虚拟”的作用或意义在哪里体现呢？在操作系统中，虚拟内存其实包含多个虚拟层次，在不同的层次体现了不同的作用。首先，在有了分段或分页机制后，程序员或 CPU 直接“看到”的地址已经不是实际的物理地址了，这已经有一层虚拟化，我们可简称为内存地址虚拟化。有了内存地址虚拟化，我们就可以通过设置段界限或页表项来设定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术属于 lazy load 技术，简称按需分页（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当 CPU 访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出（page swap in/out）。两个虚拟页的数据内容相同时，可只分配一个物理页框，这样如果对两个虚拟页的访问方式是只读方式，这两个虚拟页可共享页框，节省内存空间；如果 CPU 对其中之一的虚拟页进行写操作，则这两个虚拟页的数据内容会不同，需要分配一个新的物理页框，并将物理页框标记为可写，这样两个虚拟页面将映射到不同的物理页框，确保整个内存空间的正确访问。这种技术称为写时复制（Copy On Write，简称 COW）。这三种内存管理技术给了程序员更大的内存“空间”，我们称为内存空间虚拟化。

ucore 在实现上述三种技术时，需要解决的一个关键问题是，何时进行请求调页/页换入换出/写时复制处理？其实，在程序的执行过程中由于某种原因（页框不存在/写只读页等）而使 CPU 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射时，CPU 会产生一次缺页异常，从而需要进行相应的缺页异常服务例程。这个缺页异常处理的时机就是求调页/页换入换出/写时复制处理的执行时机，当相关处理完成后，缺页异常服务例程会返回到产生异常的指令处重新执行，使得软件可以继续正常运行下去。

### 3.11.3 proj7/8/9/9.1/9.2 概述

为了实现虚存管理，首先需要能够处理缺页异常，这是需要对当前的 trap 处理进行扩展，并能够描述当前内核中“合法”的虚拟内存（不一定有对应的物理内存）。proj7 在 proj6 的基础上实现了上述过程，新增加的主要工作包括：

- 描述当前“合法”的虚拟内存的数据结构 vma\_struct 和针对 vma\_struct 的函数操作；
- 扩展 trap\_dispatch 函数，使得能够根据 vma\_struct 结构的描述，正确完成对缺页的处理（即如果发现是“合法”的虚拟内存地址，则创建或修改页表项来建立与物理内存页的对应关系）。



为了提供超过物理内存大小的虚拟内存空间，需要把不常用的页换出到硬盘上，这样当访问到这些不存在的虚存页时，会产生缺页异常，可以把这些页再从硬盘拷贝回到内存中。

proj8 在 proj7 的基础上完成上述过程的实现，新增加的主要工作包括：

- 为了准备 swap in/out，实现通过 PIO 方式读写 IDE 格式的硬盘；
- 建立 swap 相关数据结构和相关操作，确保不常用的页能够被换出（swap out）到硬盘上，并在被访问时，能够从硬盘对应的扇区中换入（swap in）到内存中；

为了实现将来不同进程（用户态程序）之间共享内存，需要对描述虚拟内存的 vma\_struct 结构进行扩展。proj9/9.1 在 proj8 的基础上完成上述过程的实现，新增加的主要工作包括：

- 增加 shmem\_node 结构的描述，确保能够描述多个虚拟页映射到一个物理页的情况，并增加针对 shmem\_node 的处理。

为了减少复制内存的开销，可通过实现写时复制（Copy On Write，简称 COW）机制来完成，其基本思路是在只读情况下，多个虚拟页只需映射到一个物理页上，当对虚拟页进行写操作时，才真正完成对物理页的复制。在实现上需要对 page 的属性进行扩展，能够在发生页保护异常时，探测出是为了“写时复制”而设置的页，这样在缺页异常处理中，会完成实际的分配新页操作。proj9.2 在 proj9.1 的基础上完成上述过程的实现，新增加的主要工作包括：

- 扩展 trap\_dispatch 函数，使得能够根据产生异常的地址的页表项内容和此地址对应的 vma 中的属性描述，正确完成对的“写时复制”处理。

### 3.11.4 proj7：支持缺页异常和 VMA 结构

#### 一 proj7 项目组成

```
proj7
|  |-- init
|  |  |-- init.c
|  |-- mm
|
|  |  |-- pmm.c
|  |  |-- pmm.h
|  |  |-- vmm.c
|  |  |-- vmm.h
|  |-- sync
|  |  |-- sync.h
|  |-- trap
|  |  |-- trap.c
|
|-- libs
|  |-- x86.h
```

相对与 proj6，proj7 主要修改和增加的文件如下：

- `init.c`: 在 `kern_init` 中增加调用初始化虚存管理函数 `vmm_init`
- `pmm.[ch]`: 增加 `pgdir_alloc_page` 函数, 完成分配一个空闲物理页, 并设置好页表项, 完成正确的虚拟地址到物理地址的转换;
- `trap.c`: 完成对缺页异常的基本操作, 调用 `vmm.c` 中的 `do_pgfault` 函数完成具体的缺页处理;
- `x86.h`: 完成对控制寄存器 `CR1` 和 `CR2` 的读操作;
- `vmm.[ch]`: 新增的文件, 主要是建立 `vma_struct` 结构, 用于描述不存在的虚拟内存, 并完成针对此结构的相关操作函数。

## 二 proj7 编译运行

编译并运行 `proj7` 的命令如下:

```
make
```

```
make qemu
```

则可以得到如下显示界面

```
chenyu@chenyu-laptop:~/oscourse/branches/testing/chyyuu/proj7$ make qemu
(THU.CST) os is loading ...
```

```
Special kernel symbols:
```

```
entry 0xc010002c (phys)
```

```
etext 0xc010ae5f (phys)
```

```
edata 0xc0127aa0 (phys)
```

```
end 0xc0128cbc (phys)
```

```
Kernel executable memory footprint: 164KB
```

```
memory managment: buddy_pmm_manager
```

```
e820map:
```

```
memory: 0009f400, [00000000, 0009f3ff], type = 1.
```

```
memory: 00000c00, [0009f400, 0009ffff], type = 2.
```

```
memory: 00010000, [000f0000, 000fffff], type = 2.
```

```
memory: 07efd000, [00100000, 07ffcfff], type = 1.
```

```
memory: 00003000, [07ffd000, 07ffffff], type = 2.
```

```
memory: 00040000, [fffc0000, ffffffff], type = 2.
```

```
check_alloc_page() succeeded!
```

```
check_pgdir() succeeded!
```

```
check_boot_pgdir() succeeded!
```

```
----- BEGIN -----
```

```
PDE(0e0) c0000000-f8000000 38000000 urw
```

```
|-- PTE(38000) c0000000-f8000000 38000000 -rw
```

```
PDE(001) fac00000-fb000000 00400000 -rw
```

```
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
```

```
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
```

```

----- END -----
check_slab() succeeded!
size of struct mm_struct is 24, size of struct vma_struct is 40
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
++ setup timer interrupts
100 ticks
100 ticks

```

通过上图，我们可以看到 `ucore` 在 `check_vma_struct` 函数中完成基于 `vma_struct` 结构的数据创建等操作，确保能够正确建立 `vma_struct` 结构，并在成功测试后打印“`check_vma_struct() succeeded!`”；接下来 `ucore` 创建一个描述了虚拟地址 `0~4K` 的 `vma_struct` 结构，这个 `0` 虚拟地址起始的虚拟页没有对应的物理页，所以在实际访问这个虚拟地址的时候会产生缺页异常，中断处理例程会经过如下调用：

```

vectorXXX(vectors.S)-->__alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)--
-->pgfault_handler(trap.c)-->print_pgfault(trap.c)

```

来显示出错的位置和原因“`page fault at 0x00000100: K/W [no page found].`”，即在内核态对虚存地址 `0x100` 处执行写操作出现了缺页异常。并进一步调用 `do_pgfault` 函数来检测时候此虚拟地址属于某个 `vma_struct` 描述的范畴，如果是，则会分配一个物理页来对应此虚拟地址所在的虚拟页，并返回继续执行引起缺页异常的指令。如果测试能够正确执行对应的写操作指令，表明能正确处理缺页异常，则显示“`check_pgfault() succeeded!`”和“`check_vmm() succeeded.`”。

### 3.11.5 【实现】`vma_struct` 数据结构和相关操作

在讲述缺页异常处理前，需要建立好虚拟内存空间描述。在 `proj7` 之前有关内存的数据结构和相关操作都是直接针对实际存在的资源--物理内存空间的管理，没有从一般应用程序对内存的“需求”考虑，即需要有相关的数据结构和操作来体现一般应用程序对虚拟内存的“需求”。一般应用程序的对虚拟内存的“需求”与物理内存空间的“供给”没有直接的对应关系，`ucore` 是通过缺页异常处理来间接完成这二者之间的衔接。

在 `ucore` 中描述应用程序对虚拟内存“需求”的数据结构是 `vma_struct`，以及针对 `vma_struct` 的函数操作。这里把一个 `vma_struct` 结构的变量简称为 `vma` 变量。`vma_struct` 的定义如下：

```

struct vma_struct {
    struct mm_struct *vm_mm;
    uintptr_t vm_start;
    uintptr_t vm_end;
    uint32_t vm_flags;
    list_entry_t list_link;
};

```

`vm_start` 和 `vm_end` 描述了一个连续地址的虚拟内存空间的起始位置和结束位置，这两个值都应该是 `PGSIZE` 对齐的，而且描述的是一个合理的地址空间范围（即严格确保 `vm_start < vm_end` 的关系）；`list_link` 是一个双向链表，按照从小到大的顺序把一系列用

`vma_struct` 表示的虚拟内存空间链接起来, 并且还要求这些链起来的 `vma_struct` 应该是不相交的, 即 `vma` 之间的地址空间无交集; `vm_flags` 表示了这个虚拟内存空间的属性, 目前的属性包括:

```
#define VM_READ 0x00000001 //只读
#define VM_WRITE 0x00000002 //可读写
#define VM_EXEC 0x00000004 //可执行
```

以后还会引入如 `VM_STACK` 的其它属性来支持动态扩展用户栈空间。

`vm_mm` 是一个指针, 指向一个比 `vma_struct` 更高的抽象层次的数据结构 `mm_struct`, 这里把一个 `mm_struct` 结构的变量简称为 `mm` 变量。这个数据结构表示了包含所有虚拟内存空间的共同属性, 具体定义如下

```
struct mm_struct {
    list_entry_t mmap_list;
    struct vma_struct *mmap_cache;
    pde_t *pgdir;
    int map_count;
};
```

`mmap_list` 是双向链表头, 链接了所有属于同一页目录表的虚拟内存空间, `mmap_cache` 是指向当前正在使用的虚拟内存空间, 由于操作系统执行的“局部性”原理, 当前正在用到的虚拟内存空间在接下来的操作中可能还会用到, 这时就不需要查链表, 而是直接使用此指针就可找到下一次要用到的虚拟内存空间。由于 `mmap_cache` 的引入, 使得 `mm_struct` 数据结构的查询加速 30% 以上。`pgdir` 所指向的就是 `mm_struct` 数据结构所维护的页表。通过访问 `pgdir` 可以查找某虚拟地址对应的页表项是否存在以及页表项的属性等。`map_count` 记录 `mmap_list` 里面链接的 `vma_struct` 的个数。

涉及 `mm_struct` 的操作函数比较简单, 只有 `mm_create` 和 `mm_destroy` 两个函数, 从字面意思就可以看出是完成 `mm_struct` 结构的变量创建和删除。在 `mm_create` 中用 `kmalloc` 分配了一块空间, 所以在 `mm_destroy` 中也要对应进行释放。在 `ucore` 运行过程中, 会产生描述虚拟内存空间的 `vma_struct` 结构, 所以在 `mm_destroy` 中也要进对这些 `mmap_list` 中的 `vma` 进行释放。涉及 `vma_struct` 的操作函数也比较简单, 主要包括三个:

- `vma_create`--创建 `vma`
- `insert_vma_struct`--插入一个 `vma`
- `find_vma`--查询 `vma`。

`vma_create` 函数根据输入参数 `vm_start`、`vm_end`、`vm_flags` 来创建并初始化描述一个虚拟内存空间的 `vma_struct` 结构变量。`insert_vma_struct` 函数完成把一个 `vma` 变量按照其空间位置 [`vma->vm_start`, `vma->vm_end`] 从小到大的顺序插入到所属的 `mm` 变量中的 `mmap_list` 双向链表中。`find_vma` 根据输入参数 `addr` 和 `mm` 变量, 查找在 `mm` 变量中的 `mmap_list` 双向链表中某个 `vma` 包含此 `addr`, 即 `vma->vm_start` <= `addr` < `vma->end`。这三个函数与后续讲到的缺页异常处理有紧密联系。

### 3.11.6 【实现】缺页异常处理

当启动分页机制以后, 如果一条指令或数据的虚拟地址所对应的物理页框不在内存中或者访问的类型有错误 (比如写一个只读页或用户态程序访问内核态的数据等), 就会发生缺页异常。产生页面异常的原因主要有:

- 目标页面不存在 (页表项全为 0, 即该线性地址与物理地址尚未建立映射或者已经

撤销);

- 相应的物理页面不在内存中(页表项非空, 但 **Present** 标志位=0, 比如在 **swap** 分区或磁盘文件上), 这将在下面介绍换页机制实现时进一步讲解如何处理;
- 访问权限不符合(此时页表项 **P** 标志=1, 比如企图写只读页面).

当出现上面情况之一, 那么就会产生页面 **page fault** (**#PF**) 异常。产生异常的线性地址存储在 **CR2** 中, 并且将 **#PF** 的类型保存在 **error code** 中, 比如 **bit 0** 表示是否 **PTE\_P** 为 0, **bit 1** 表示是否 **write** 操作。

产生缺页异常后, **CPU** 硬件和软件都会做一些事情来应对此事。首先缺页异常也是一种异常, 所以针对一般异常的硬件处理操作是必须要做的, 即 **CPU** 在当前内核栈保存当前被打断的程序现场, 即依次压入当前被打断程序使用的 **eflags**, **cs**, **eip**, **errorCode**; 由于缺页异常的中断号是 **0xE**, **CPU** 把中断 **0xE** 服务例程的地址 (**vectors.S** 中的标号 **vector14** 处) 加载到 **cs** 和 **eip** 寄存器中, 开始执行中断服务例程。这时 **ucore** 开始处理异常中断, 首先需要保存硬件没有保存的寄存器。在 **vectors.S** 中的标号 **vector14** 处先把中断号压入内核栈, 然后再在 **trapentry.S** 中的标号 **\_\_alltraps** 处把 **ds**、**es** 和其他通用寄存器都压栈。自此, 被打断的程序现场被保存在内核栈中。

接下来, 在 **trap.c** 的 **trap** 函数开始了中断服务例程的处理流程, 大致调用关系为:

**trap-->trap\_dispatch-->pgfault\_handler-->do\_pgfault**

下面需要具体分析一下 **do\_pgfault** 函数。**CPU** 把引起缺页异常的虚拟地址装到寄存器 **CR2** 中, 并给出了出错码 (**tf->tf\_err**), 指示引起缺页异常的存储器访问的类型。而中断服务例程会调用缺页异常处理函数 **do\_pgfault** 进行具体处理。缺页异常处理是实现按需分页、**swap in/out** 和写时复制的关键之处, 后面的小节将分别展开讲述。

**ucore** 中 **do\_pgfault** 函数是完成缺页异常处理的主要函数, 它根据从 **CPU** 的控制寄存器 **CR2** 中获取的缺页异常的虚拟地址以及根据 **error code** 的错误类型来查找此虚拟地址是否在某个 **VMA** 的地址范围内以及是否满足正确的读写权限, 如果在此范围内并且权限也正确, 这认为这是一次合法访问, 但没有建立虚实对应关系。所以需要分配一个空闲的内存页, 并修改页表完成虚地址到物理地址的映射, 刷新 **TLB**, 然后调用 **iret** 中断, 返回到产生缺页异常的指令处重新执行此指令。如果该虚地址不再某 **VMA** 范围内, 这认为是一次非法访问。

#### 【注意】

地址空间的管理由虚存管理和页表管理两部分组成。虚存管理限制了(程序)地址空间的范围以及权限, 而页表维护的是实际使用的地址空间以及权限, 后者不能比前者有更大的范围或者权限, 因为前者是实际管理页表的。比如权限, 虚存管理可以规定地址空间的某个范围是可写的, 但是页表中却可以标记是 **read-only** 的(比如 **copy-on-write** 的实现), 这种冲突可以被内核(通过硬件异常)轻易的捕获到, 并进行相应的处理。反过来, 如果页表权限比虚存规定的权限更大, 内核是没有办法发现这种冲突的。由于虚存管理的存在, 内核才能方便的实现更复杂和丰富的操作, 比如 **share memory**、**swap** 等。在后续的实验中还会遇到虚存管理只维护用户地址空间(也就是 **[USERBASE, USERTOP)** 区间)的情况, 因为内核地址空间包括虚存和页表都是固定的。

## 3.11.7 proj8: 支持页换入换出

### 一 proj8 项目组成

编译并运行 **proj8** 的命令如下:

---

make

make qemu

则可以得到如下显示界面

proj8

```
|  └─ driver
|  |  └─ ...
|  |  └─ ide.c
|  |    └─ ide.h
|  └─ fs
|  |  └─ fs.h
|  |  └─ swapfs.c
|  |    └─ swapfs.h
|  └─ mm
|  |  └─ .....
|  |  └─ memlayout.h
|  |  └─ pmm.c
|  |  └─ swap.c
|  |  └─ swap.h
|  |  └─ vmm.c
|  |    └─ vmm.h
|  └─ sync
|  |  └─ sync.h
|  └─ trap
|  |  └─ trap.c
|  |    └─ .....
└─ libs
|  └─ hash.c
|  └─ .....
└─ .....
```

相对于 proj7, proj8 主要修改和增加的文件如下: :

- ide.[ch]: 实现了对 IDE 硬盘的 PIO 方式的扇区读写功能, 用于支持把页换入和换出硬盘。
- swapfs.[ch]: 根据页和硬盘扇区的映射关系, 实现了在 IDE 硬盘上的 swap 文件组织, 并实现了把页写入 swap 文件和从 swap 文件读入页的功能。需要 ide.[ch] 的支持。
- swap.[ch]: 参考 Linux2.4 的页替换策略, 实现了一个简化的双链表页替换策略。
- memlayout.h: 修改 Page 等关键数据结构, 支持双链页替换策略。
- pmm.c: 修改 page\_remove\_pte 函数, 支持双链页替换策略。
- vmm.c: 修改 do\_pgfault 函数, 支持页的换入换出。
- sync.h: 增加 lock/unlock 支持, 支持页的换入换出过程不会出现 race condition 现象。

---

## 二 proj8 编译运行

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc010002c (phys)
  etext 0xc010dfec (phys)
  edata 0xc012faa8 (phys)
  end   0xc0132e20 (phys)
Kernel executable memory footprint: 204KB
memory managment: buddy_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- BEGIN -----
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----
check_slab() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:      262144(sectors), 'QEMU HARDDISK'.
page fault at 0x00000000: K/W [no page found].
page fault at 0x00000000: K/W [no page found].
page fault at 0x00001001: K/W [no page found].
page fault at 0x00001000: K/R [no page found].
page fault at 0x00000000: K/R [no page found].
check_swap() succeeded.
++ setup timer interrupts
100 ticks
```

`check_swap` 函数对 `ucore` 在 `proj8` 中建立的双链页面置换策略进行了测试，验证了其正确性，下面我们将从原理和实际实现两个方面来分析 `proj8` 中实现的页面置换算法。

### 3.11.8 【原理】页面置换算法

操作系统为何要进行页面置换呢？这是由于操作系统给用户态的应用程序提供了一个虚拟的“大容量”内存空间，而实际的物理内存空间又没有那么小。所以操作系统就“瞒着”应用程序，只把应用程序中“常用”的数据和代码放在物理内存中，而不常用的数据和代码放在了硬盘这样的存储介质上。如果应用程序访问的是“常用”的数据和代码，那么操作系统已经放置在内存中了，不会出现什么问题。但当应用程序访问它认为应该在内存中的数据和代码时，如果这些数据或代码不在内存中，则根据上一小节的介绍，会产生缺页异常。这时，操作系统必须能够应对这种缺页异常，即尽快把应用程序当前需要的数据或代码放到内存中来，然后重新执行应用程序产生异常的访存指令。如果在把硬盘中对应的数据或代码调入内存前，操作系统发现物理内存已经没有空闲空间了，这时操作系统必须把它认为“不常用”的页换出到磁盘上去，以腾出内存空闲空间给应用程序所需的数据或代码。

操作系统迟早会碰到没有内存空闲空间而必须要置换出内存中某个“不常用”的页的情况。如何判断内存中哪些是“常用”的页，哪些是“不常用”的页，把“常用”的页保持在内存中，在物理内存空闲空间不够的情况下，把“不常用”的页置换到硬盘上就是页面置换算法着重考虑的问题。容易理解，一个好的页面置换算法会导致缺页异常次数少，也就意味着访问硬盘的次数也少，从而使得应用程序执行的效率就高。

从操作系统原理的角度看，有如下一些页面置换算法：

- 最优 (Optimal) 页面置换算法：由 Belady 于 1966 年提出的一种理论上的算法。其所选择的被淘汰页面，将是以后永不使用的或许是在最长的未来时间内不再被访问的页面。采用最佳置换算法，通常可保证获得最低的缺页率。但由于操作系统其实无法预知一个应用程序在执行过程中访问到的若干页中，哪一个页是未来最长时间内不再被访问的，因而该算法是无法实际实现，但可以此算法作为上限来评价其它的页面置换算法。
- 先进先出 (First In First Out, FIFO) 页面置换算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。FIFO 算法的另一个缺点是，它有一种异常现象 (Belady 现象)，即在增加放置页的页帧的情况下，反而使缺页异常次数增多。
- 二次机会 (Second Chance) 页面置换算法：为了克服 FIFO 算法的缺点，人们对它进行了改进。此算法在页表项 (PTE) 中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当需要找到一个页淘汰时，对于最“老”的那个页面，操作系统去检查它的访问位。如果访问位是 0，说明这个页面老且无用，应该立刻淘汰出局；如果访问位是 1，这说明该页面曾经被访问过，因此就再给它一次机会。具体来说，先把访问位清零，然后把该页面放到队列的尾端，并修改它的装入时间，就好像它刚刚进入系统一样，然后继续往下搜索。二次机会算法的实质就是寻找一个比较古老的、而且从上一次缺页异常以来尚未被访问的页面。如果所有的页面都被访



问过了，它就退化为纯粹的 FIFO 算法。

- **LRU(Least Recently Used, LRU)页面置换算法：** FIFO 置换算法性能之所以较差，是因为它所依据的条件是各个页调入内存的时间，而页调入的先后顺序并不能反映页是否“常用”的使用情况。最近最久未使用（LRU）置换算法，是根据页调入内存后的使用情况进行决策页是否“常用”。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，因此，LRU 置换算法是选择最近最久未使用的页予以淘汰。该算法赋予每个页一个访问字段，用来记录一个页面自上次被访问以来所经历的时间  $t$ ，当须淘汰一个页面时，选择现有页面中其  $t$  值最大的，即最近最久未使用的页面予以淘汰。
- **时钟（Clock）页面置换算法：**也称最近未使用（Not Used Recently, NUR）页面置换算法。虽然二次机会算法是一个较合理的算法，但它经常需要在链表中移动页面，这样做既降低了效率，又是不必要的。一个更好的办法是把各个页面组织成环形链表的形式，类似于一个钟的表面。然后把一个指针指向最古老的那个页面，或者说，最先进来的那个页面。时钟算法和第二次机会算法的功能是完全一样的，只是在具体实现上有所不同。时钟算法需要在页表项（PTE）中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。然后将内存中所有的页都通过指针链接起来并形成一个循环队列。初始时，设置一个当前指针指向某页（比如最古老的那个页面）。操作系统需要淘汰页时，对当前指针指向的页所对应的页表项进行查询，如果访问位为“0”，则淘汰该页，把它换出到硬盘上；如果访问位为“1”，这将该页表项的此位置“0”，继续访问下一个页。该算法近似地体现了 LRU 的思想，且易于实现，开销少。但该算法需要硬件支持来设置访问位，且该算法在本质上与 FIFO 算法是类似的，惟一不同的是在 clock 算法中跳过了访问位为 1 的页。
- **改进的时钟（Enhanced Clock）页面置换算法：**在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当该页被“写”时，CPU 中的 MMU 硬件将把修改位置“1”。这样这两位就存在四种可能的组合情况：（0，0）表示最近未被引用也未被修改，首先选择此页淘汰；（0，1）最近未被使用，但被修改，其次选择；（1，0）最近使用而未修改，再次选择；（1，1）最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的 I/O 操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，增加了算法本身的执行开销。

### 3.11.9 【实现】页面置换机制实现（应该放在第四章进程管理与调度）

#### [下面的内容要去掉，并换成局部页置换的实现]

如果要实现页面置换机制，只考虑页面置换算法的设计与实现是远远不够的，还需考虑其他问题：

- 哪些页可以被换出？

- 一个虚拟的页如何与硬盘上的扇区建立对应关系？
- 何时进行换入和换出操作？
- 在 `proj7` 的基础上，如何设计数据结构已支持页面置换算法？
- 如何完成页的换入换出操作？

这些问题在下面会逐一进行分析。注意，在 `proj8` 中实现了换入换出机制，但现在还没有涉及 `lab3` 才实现的用户进程（激活页面置换的用户方）和内核线程（完成页面置换的服务方），所以还无法通过内核线程机制实现一个完整意义上的虚拟内存页面置换功能，并给用户进程提供大于实际物理空间的虚空间。这要等到 `lab3` 的 `proj11` 才提供上述支持。

## 一 可以被换出的页

在操作系统的设计中，一个基本的原则是：并非所有的物理页都可以交换出去的，只有映射到用户空间且被用户程序直接访问的页面才能被交换，而被内核直接使用的内核空间的页面不能被换出。这里面的原因是什么呢？操作系统是执行的关键代码，需要保证运行的高效性和实时性，如果在操作系统执行过程中，发生了缺页现象，则操作系统不得不等很长时间（硬盘的访问速度比内存的访问速度慢 2~3 个数量级），这将导致整个系统运行低效。而且，不难想象，处理缺页过程所用到的内核代码或者数据如果被换出，整个内核都面临崩溃的危险。

但在 `proj8` 实现的 `ucore` 中，我们只是实现了换入换出机制，还没有设计用户态执行的程序，所以我们在 `proj8` 中仅仅通过执行 `check_swap` 函数在内核中分配一些页，模拟对这些页的访问，然后直接调用 `page_launder` 函数来查询这些页的访问情况并执行页面置换算法换出“不常用”的页到磁盘上。

## 二 虚存中的页与硬盘上的扇区之间的映射关系

如果一个页被置换到了硬盘上，那操作系统如何能简捷来表示这种情况呢？在 `ucore` 的设计上，充分利用了页表中的 `PTE` 来表示这种情况：当一个 `PTE` 用来描述一般意义上的物理页时，显然它应该维护各种权限和映射关系，以及应该有 `PTE_P` 标记；但当它用来描述一个被置换出去的物理页时，它被用来维护该物理页与 `swap` 磁盘上扇区的映射关系，并且该 `PTE` 不应该由 `MMU` 将它解释成物理页映射（即没有 `PTE_P` 标记），与此同时对应的权限则交由 `mm_struct` 来维护，当对位于该页的内存地址进行访问的时候，必然导致 `#PF`，然后内核能够根据 `PTE` 描述的 `swap` 项将相应的物理页重新建立起来，并根据虚存所描述的权限重新设置好 `PTE` 使得内存访问能够继续正常进行。

如果一个页（4KB/页）被置换到了硬盘某 8 个扇区（0.5KB/扇区），该 `PTE` 的最低位 `--present` 位应该为 0（没有 `PTE_P` 标记），接下来的 7 位暂时保留，可以用作各种扩展；而原来用来表示页帧号的高 24 位地址，恰好可以用来表示此页在硬盘上的起始扇区的位置（其从第几个扇区开始）。为了在页表项中区别 0 和 `swap` 分区的映射，将 `swap` 分区的一个 `page` 空出来不用，也就是说一个高 24 位不为 0，而最低位为 0 的 `PTE` 表示了一个放在硬盘上的页的起始扇区号（见 `swap.h` 中对 `swap_entry_t` 的描述）：

`swap_entry_t`

```
-----
|          offset          | reserved | 0      |
```

-----  
24 bits                      7 bits      1 bit

考虑到硬盘的最小访问单位是一个扇区，而一个扇区的大小为 512 ( $2^8$ ) 字节，所以需要 8 个连续扇区才能放置一个 4KB 的页。在 ucore 中，用了第二个 IDE 硬盘来保存被换出的扇区，根据 proj8 的输出信息

```
“ide 1:      262144(sectors), 'QEMU HARDDISK'.”
```

我们可以知道 proj8 可以保存  $262144/8=32768$  个页，即 128MB 的内存空间。swap 分区的大小是 swapfs\_init 里面根据磁盘驱动接口计算出来的，目前 ucore 里面要求 swap 磁盘至少包含 1000 个 page，并且至多能使用  $1 \ll 24$  个 page。

swap.c 中维护了全局的 mem\_map 表，用来记录 swap 分区的使用，它会在 swap\_init 里面根据 swap 分区的大小进行适当的初始化。表中的每一项是一个 unsigned short 类型的整数，用来记录该 entry 的引用计数。ucore 使用一个非常大的整数 0xFFFF 表示一个 entry（这里 entry 指 swap 分区上连续的 8 个扇区）是空闲的（没有被分配出去），并用 0xFFFE 表示一个 entry 的最大引用计数，通常一个页不会有这么大的引用计数，除非内核崩了。当一个 entry 的引用计数是 0 的时候，表示该 entry 是可以被回收的，但是我们通常不是真的去回收他，只有当需要分配一个 entry，并且在 mem\_map 里面实在找不到空闲的 entry 的时候才会去回收一个引用计数为 0 的 entry。因为一个引用计数为 0 的页很可能上面的数据和对应物理页上的数据一致，这样在换出该页的时候就可以避免写磁盘的代价，而和写磁盘比起来，遍历 mem\_map 数组的时间开销总是会小很多。

此外，为了简化实现，swap 只对页表中的数据页进行换出，即只对 PTE 进行操作，而第二级页表是保留不动的。

### 三 执行换入换出的时机

check\_mm\_struct 是在 lab2 里面用来做模块测试时使用的临时的 mm\_struct。在 lab3 以后就没有用处了。在 proj8 中，check\_mm\_struct 变量这个数据结构表示了目前 ucore 认为合法的所有虚拟内存空间集合，而 mm 中的每个 vma 表示了一段地址连续的合法虚拟空间。当 ucore 或应用程序访问地址所在的页不在内存时，就会产生 #PF 异常，引起调用 do\_pgfault 函数，此函数会判断产生访问异常的地址属于 check\_mm\_struct 某个 vma 表示的合法虚拟地址空间，且保存在硬盘 swap 文件中（即对应的 PTE 的高 24 位不为 0，而最低位为 0），则是执行页换入的时机，将调用 swap\_in\_page 函数完成页面换入。

换出页面的时机相对复杂一些，针对不同的策略有不同的时机。ucore 目前大致有两种策略，即积极换出策略和消极换出策略。积极换出策略是指操作系统周期性地（或在系统不忙的时候）主动把某些认为“不常用”的页换出到硬盘上，从而确保系统中总有一定数量的空闲页存在，这样当需要空闲页时，基本上能够及时满足需求；消极换出策略是指，只是当试图得到空闲页时，发现当前没有空闲的物理页可供分配，这时才开始查找“不常用”页面，并把一个或多个这样的页换出到硬盘上。

在 proj8 中，可支持上述两种情况，但都需要到 lab3/proj11 中才会完整实现。对于第一种积极换出策略，即创建了一个每隔 1 秒执行一次的内核线程 kswapd（在 lab3 的 proj11 中第一次出现），实现积极的换出策略。对于第二种消极的换出策略，则是在 ucore 调用 alloc\_pages 函数获取空闲页时，此函数如果发现无法从页分配器（比如 buddy system）获得空闲页，就会进一步调用 try\_free\_pages 来唤醒线程 kswapd，并将 cpu 让给 kswapd 使得

换出某些页，实现一种消极的换出策略。

## 四 页面置换算法的数据结构设计

到 proj7 为止，我们知道目前表示内存中物理页使用情况的变量是基于数据结构 `Page` 的全局变量 `pages` 数组，`pages` 的每一项表示了计算机系统中一个物理页的使用情况。为了表示物理页可被换出或已被换出的情况，可对 `Page` 数据结构进行扩展：

```
struct Page {
    uint32_t flags;    // array of flags that describe the status of the page frame
    swap_entry_t index;    // stores a swapped-out page identifier
    list_entry_t swap_link; // swap hash link
    .....
};
```

首先 `flag` 的含义做了扩展：

```
// the page is in the active or inactive page list (and swap hash table)
#define PG_swap 4
// the page is in the active page list
#define PG_active 5
```

前面提到 `swap.c` 里面声明了全局的 `mem_map` 数据结构，用来存储 `swap` 分区上的页的使用计数。除此之外，`swap.c` 里还声明了两个链表，分别是 `active_list` 和 `inactive_list`，分别表示已经分配了 `swap entry` 的且处于“活跃”状态/“不活跃”状态的物理页链表。所有已经分配了 `swap entry` 的 `page` 必须处于两个链表中的一个。

当一个物理页（`struct Page`）需要被 `swap` 出去的时候，首先需要确保它已经分配了一个 `swap entry`。如果 `page` 数据结构的 `flags` 设置了 `PG_swap` 为 1，则表示该 `page` 中的 `index` 是有效的 `swap entry` 的索引值，从而该物理页上的数据可以被写出到 `index` 所表示的 `swap page` 上去。

如果一个物理页在硬盘上有一个页备份，则需要记录在硬盘中页备份的位置。`Page` 结构中的 `index` 就起到这个记录的作用，它保存了被换出的页的页表项 `PTE` 高 24 位的内容——`entry`，即硬盘中对应页备份的起始扇区位置值（以字节为单位??）。

`Page` 结构中的 `swap_link` 保存了以 `entry` 为 `hash` 索引的链表项，这样根据 `entry`，就可以快速的对 `page` 数据结构进行查找。但 `hash` 数组在哪里呢？

对于在硬盘上有页备份的物理页（简称 `swap_page`），需要统一管理起来，为此在 `proj8` 中增加了全局变量：

```
static list_entry_t hash_list[HASH_LIST_SIZE];

hash_list 数组就是我们需要的 hash 数组，根据 index (swap entry) 索引全部的 swap_page 的指针，这样通过 hash 函数
```

```
#define entry_hashfn(x) (hash32(x, HASH_SHIFT))
```

可以快速地根据 `entry` 找到对应的 `page` 数据结构。

正如页替换算法描述的那样，把“常用”的可被换出页和“不常用”的可被换出页分别集中管理起来，形成 `active_list` 和 `inactive_list` 两个链表。`ucore` 的页面置换算法会根据相应的准则把它认为“常用”的物理页放到 `active_list` 链表中，而把它认为“不常用”的物理页放到 `inactive_list` 链表中。一个标记了 `PG_swap` 的页总是需要在这两个链表之间移动。

前面介绍过 `mem_map` 数组，他是用来记录 `swap_page` 的引用次数的。因为 `swap` 分区上的 `page` 实际上是某个物理页的数据备份。所以，一个物理页 `page` 的 `page_ref` 与对应 `swap_page` 的 `mem_map[offset]` (`offset = swap_offset(entry)`) 值的和是这个数据页的真实引用计数。`page_ref` 表示 PTE 对该物理页的映射的个数；`mem_map` 表示 PTE 对该 `swap` 备份页的映射个数。当 `page_ref` 为 0 的时候，表示物理页可以被回收；当 `mem_map[offset]` 为 0 的时候，表示 `swap page` 可以被回收（前面介绍过，可以回收，但是在万不得已的情况下才真的回收）。在后面的实验中还可能涉及到，这里只是简单了解一下。

### 【注意】

`ucore` 目前使用的 PIO 的方式读写 IDE 磁盘，这样的好处是，磁盘读入、写出操作可以认为是同步的，即当前 CPU 需要等待磁盘读写完毕后再进行进一步的工作。由于磁盘操作相对 CPU 的速度而言是很慢的，这使得会浪费大量的 CPU 时间在等 IO 操作上。于是我们总是希望能够在 IO 性能上有更大的提升，比如引入 DMA 这种异步的 IO 机制，为了避免后续开发上的各种不便和冲突，我们假设所有的磁盘操作都是异步的（也包括后面的实验），即使目前是通过 PIO 完成的。

假定某 `page` 的 `flags` 中的 `PG_swap` 标志位为 1，并且 `PG_active` 标志位也为 1，则表示该 `page` 在 `swap` 的 `active_list` 中，否则在 `inactive_list` 中。`active_list` 中的页表示活跃的物理页，即页表中可能存在多个 PTE 指向该物理页（这里可以是同一个页表中的多个 `entry`，在后面 lab3 的实验里面有了进程以后，也可以是多个进程的页表的多个 `entry`）；反过来，`inactive_list` 链表所链接的 `page` 通常是指没有 PTE 再指向的页。

### 【注意】

需要强调两点设计因素：

- 1、一个 `page` 是在 `active_list` 还是在 `inactive_list` 的条件不是绝对的；
- 2、只有 `inactive_list` 上的页才会被尝试换出。

这两个设计因素的设计起因如下：

1. 我们知道一个 `page` 换出的代价是很大的（磁盘操作），并且我们假设所有的磁盘操作都是异步的，那么换出一个 `active` 的页就变得非常不值得。因为在还有多个 PTE 指向他情况下进行换出操作（异步 IO 可能导致进程切换）的较长过程中，这个页可以随时被其它进程写脏。而硬件提供给内核的接口（即页表项 PTE 的 `dirty` 位）使得内核只能知道一个页是否是脏的（不能明确知道一个页的哪个部分是脏的），当这种情况发生时，就导致了一次无效的写出。

2. `active_list` 和 `inactive_list` 的维护只能由与 `swap` 有关的集中操作来完成。特别是在 lab3/proj11 引入 `kswapd` 内核线程之后，所有的内存页换出任务都交给 `kswapd`，这样减少了复杂的同步互斥实现（在 lab5 中会重点涉及）。

3. 页面换入换出有关的操作需要做的就是尽可能的完成如下三件事情：

- (1) 将 `PG_swap` 为 0 的页转变成 `PG_swap` 为 1 的页。即尽可能的给每个物理页分配一个 `swap entry`（当然前提是有足够大的 `swap` 分区）。

- (2) 将页从 `active_list` 上移动到 `inactive_list` 上。如果一个页还在 `active_list` 上，说明还有 PTE 指向此“活跃”的物理页。所以需要在完成内存页换出时断开对这些物理页的引用，把它变成不活跃的（`inactive`）。只有把所有的 PTE 对某 `page` 的引用都断开（即 `page` 的 `page_ref` 为 0）后，就可以将此 `page` 从 `active_list` 移动到 `inactive_list` 上。

- (3) 将 `inactive_list` 上的页写出并释放掉。`inactive_list` 上的 `page` 表示已没有 PTE 指向此 `page` 了，那么该 `page` 可以被释放，如果该 `page` 被写过，那还需把此 `page` 换出

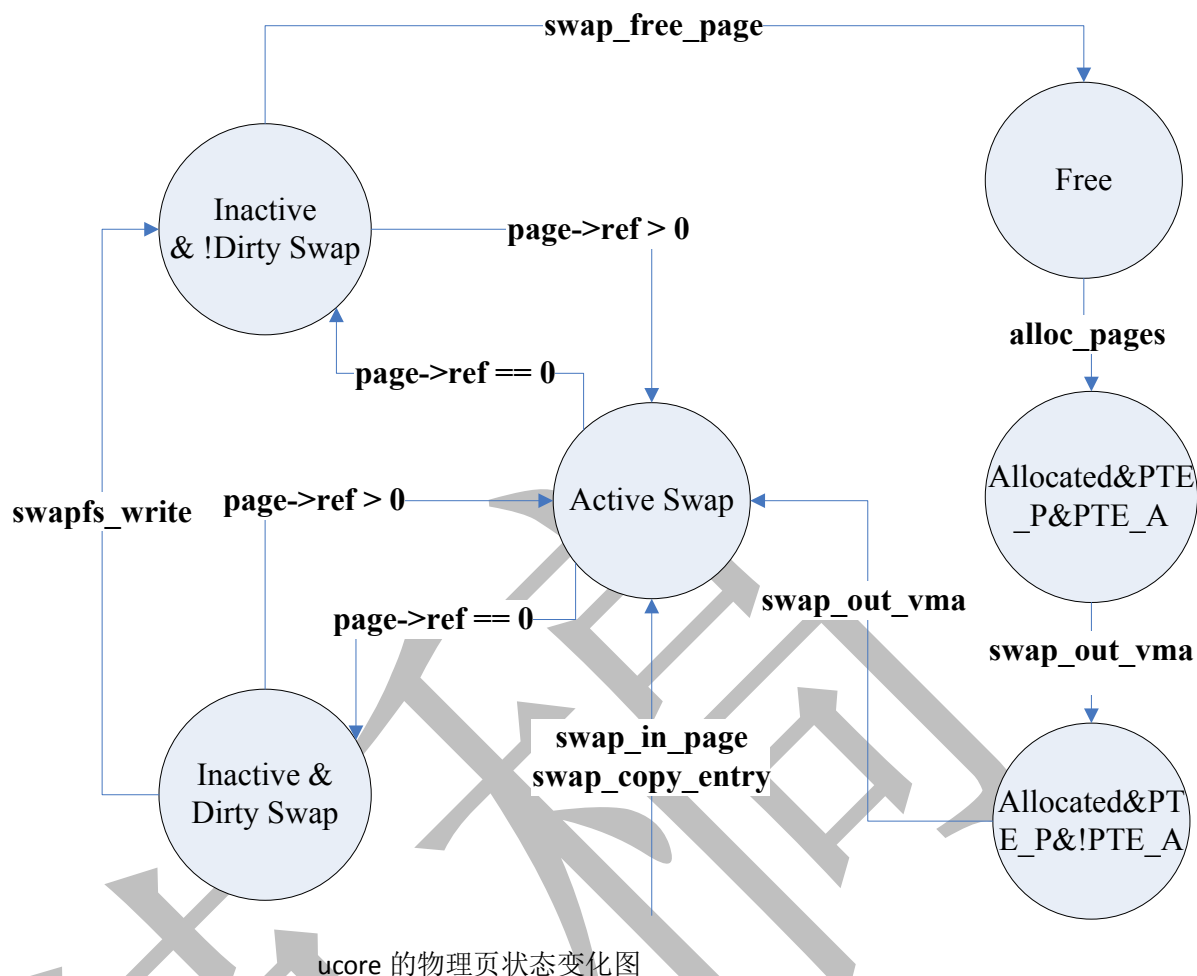
到 swap 分区上。如果在整个换出过程（异步 IO）中没有其他进程再写这个物理页（即没有 PTE 在引用它或有 PTE 引用但页没有写脏），就认为这个物理页是可以安全释放的了。那么将它从 `inactive_list` 上取下，并调用 `page_free` 函数实现 `page` 的回收。

4. 值得注意的是，内存页换出操作只有特定时候才被调用，即通过执行 `try_free_pages` 函数或者定时器机制（在 lab3/proj10.4 才引入）定期唤醒 `kswapd` 内核线程。这样会导致内存页换出操作对两个链表上的数据都不够敏感。比如处于 `active_list` 上的 `page`，可能在 `kswapd` 工作的时候，已经没有 PTE 再引用它了；再如相应的进程退出了，并且相应的地址空间已经被内核回收，从而变成了一个 `inactive` 的 `page`；还存在 `inactive_list` 上的 `page` 也可能在换出的时候，其它进程通过 `page fault`，又将 PTE 指向他，进而变成一个实际上 `active` 的页。所以说，`active` 和 `inactive` 条件并不绝对。

## 五 页面置换算法的执行逻辑

其实在 `proj8` 中并没有完全实现页面置换算法，只是实现了其中的部分关键函数，并通过 `check_swap` 来验证了这些函数的正确性。直到 lab3 的 `proj11` 才形成了完整的页面置换逻辑，而这个页面置换逻辑基本上是改进的时钟算法的一个实际扩展版本。

`ucore` 采用的页面置换算法是一个全局的页面置换算法，因为它收集了 `ucore` 中所有用户态进程（这里可理解为 `ucore` 中运行的每个用户态程序）的可换出页，并把这些可换出页中的一部分转换为空闲页。其次它考虑了页的访问情况（根据 PTE 中 `PTE_A` 位的值）和读写情况（根据 PTE 中 `PTE_D` 位的值）。如果页被访问过，则把 `PTE_A` 位清零继续找下一页；如果页没有被访问过，这此页就成为了 `active` 状态的可换出页，并放入 `active_list` 链表中，这时需要把对应的 PTE 转换成为一个 `swap entry`（高 24 位保存为硬盘缓存页的起始扇区号，`PTE_P` 位清零）；接着 `refill_inactive_scan` 函数会把处于 `active` 状态的部分可换出页转换成 `inactive` 状态，并放入 `inactive_list` 链表中；然后 `page_launer` 函数扫描 `inactive_list` 中的处于 `inactive` 状态的可换出页，如果此页不是 `dirty` 的，则把它直接转换成空闲页，如果此页是 `dirty` 的，则执行换出操作，把该页换出到硬盘上保存。这个页的状态变化图如下图所示。



下面具体讲述一下 proj11 中实现上述置换算法的页面置换逻辑。在 proj11 中同时实现了积极换出策略和消极换出策略,这都是通过在不同的时机执行 `kwapd_main` 函数来完成的。当 `ucore` 调用 `alloc_pages` 函数以获取空闲页,但物理页内存分配器无法满足请求时,`alloc_pages` 函数将调用 `tree_free_pages` 来执行 `kwapd_main` 函数(通过直接唤醒线程的方式,在 lab3 中会进一步讲解),完成对页的换出操作和生成空闲页的操作。这是一种消极换出的策略。另外,`ucore` 设立了每秒执行一次 `kwapd_main` 函数(通过设置 `timer` 来唤醒线程的方式,在 lab3 中会进一步讲解),完成对页的换出操作和生成空闲页的操作。这是一种积极换出的策略。

`kswapd_main` 是 `ucore` 整个页面置换算法的总控部分,其大致思路是根据当前的空闲页情况查找出足够多的可换出页(`swap page`),然后根据这些可换出页的访问情况确定哪些是“常用”页,哪些是“不常用”页,最后把“不常用”的页转换成空闲页。思路简单,但具体实现相对复杂。

如前所述,swap 整个流程就是把尽可能多的 `page` 从变成 `PG_active` 的,并移动到 `active list` 中;把 `active list` 中的页尽可能多的变成 `inactive list` 中页;最后把 `inactive list` 的页换出(洗净),根据情况处理洗净后的 `page` 结构(根据 `page ref` 以及相应的 `dirty bit`)。所以整个过程的核心是尽可能的断开页表中的 `PTE` 映射。

当 `alloc_pages` 执行分配 `n` 连续物理页失败的时候,则会通过调用 `tree_free_page`



来唤醒 kswapd 线程，此线程执行 kswapd\_main 函数。kswapd 会发现它现在压力很大，需要尽可能的满足分配 n 个连续物理页的需求。既然需求是 n 个连续物理页，那么 kswapd 所需要释放的物理页就应该大于 n 个；每个页可能在某个或者许多个页表的不同地方有 PTE 映射（特别是 copy on write 之后，这种情况更为普遍），那么 kswapd 所需要断开的 PTE 映射就远远不止 n 个。Linux 实现了能够根据 physical address 在页表中快速定位的数据结构，但是实现起来过于复杂，这里 ucore 采用了一个比较笨的方法，即遍历所有存在的页表结构，断开足够多的 PTE 映射。这里足够多是个经验公式，采用  $n \ll 5$ 。当然这也可能失败，那么 kswapd 就会尝试一定次数。当他实在无能为力时，也就放弃了。而 alloc\_pages 也会不停的调用 try\_free\_pages 进行尝试，当尝试不停的遭遇失败的时候，程序中会有许多句 warn 来输出这些调试信息。而 Linux 的方案是选择一个占用内存最多的进程杀掉并释放出资源，来尽可能的满足当前程序的需求（注意，这里当前程序是指内核服务或者调用），直到程序从内核态正常退出；ucore 的这种设计显然是过于简单了，不过此是后话。

扫描页表是一项艰巨的任务，因为除了内核空间，用户地址空间有将近 3G 的空间，真正的程序很少能够用这么多。因此，充分利用虚存管理能够很大的提升扫描页表的速度。

接下来，我们需要介绍一下 kswapd\_main 是如何一步一步完成 swap 的操作的。正如前面介绍的那样，swap 的任务主要分成三个过程，

现在我们来介绍以下 kswap\_main 是如何一步一步完成 swap 的操作的。正如前面介绍过的，swap 需要完成 3 件事情，下面对应的是这三个操作的具体细节：

1. kswapd\_main 函数通过循环调用函数 swap\_out\_mm 并进一步调用 swap\_out\_vma，来查找 ucore 中所有存在的虚存空间，并总共断开 m 个 PTE 到物理页的映射（这里需要和进程的概念有所结合，可以理解为每个用户程序都拥有一个自己的虚存空间。但是需要提一句的是，在后的实验中还会遇到虚存空间在多个进程之间的共享；遍历虚存空间而不是遍历每个进程是为了避免某个虚存空间被很多进程共享进而被 kswapd 过度压榨所带来的不公平。可以想象，被过度压榨的虚存空间，同时又由于被很多进程共享从而有很高的概率被使用到，最终必然会导致频繁的 #PF，给系统不必要的负担。还有就是虽然 swap 的任务是断开 m 个 PTE 映射，但是实际上它对每个虚存空间都一次至多提出断开 32 个映射的需求，并循环遍历所有的虚存空间直到 m 得到满足。这样做的目的也是为了保证公平，使得每个虚存空间被交换出去的页的几率是近似相等的。Linux 实际上应该有更好的实现，它根据虚存空间所实际使用的物理页的个数来决定断开的映射的个数。）。这些断开的 PTE 映射所指向的物理页如果没有 PG\_active 标记，则需要给他分配一个新的 swap entry，并做好标记，将 page 插入到 active\_list 中去（同时也插入到 swap 的哈希表中），然后设置好相应的 page\_ref 和 mem\_map[offset] 的值，当然，如果找不到空闲的 swap entry 可以分配（比如 swap 分区已经用光了），我们只能跳过这样的 PTE 映射，从下一个地址继续寻找出路；对于原来就已经标记了 PG\_swap 的物理页，则只需要完成后面的工作，即调整引用计数就足够了。断开的 PTE 被 swap\_entry 取代，并取消 PTE\_P 标记，这样当出现 #PF 的时候，我们能够直接根据 PTE 上的值得到该页的数据实际是在 swap 分区上的哪个位置上。

现在不必要计较一个 page 究竟是放在 active\_list 还是放在 inactive\_list 中，更不必要考虑换出这样的操作，这一阶段的工作只是断开 PTE 映射，余下的工作后面会一步步完成。

当 kswapd 断开足够数量的 PTE 映射以后，这一部分的工作也就完成了。虚存管理中维护了一个 swap\_address 的地址，表示上一次 swap 操作结束时的地址，维护这个数据是避免每次 swap 操作都从虚存空间的起始地址开始，从而导致过多数量的重复的无效的遍历。

当 kswapd 发现自己竭尽所能的遍历都无法满足断开 m 个链接的需求时，该怎么办？



我们需要明确的是 swap 操作的主要目的是释放物理页，而断开 PTE 映射是一个必要的步骤，作用是尽可能的扩大 inactive\_list 中 page 的个数，为物理页的换出提供更大的基数（操作空间），但并不是主要过程。所以为了防止在这一步陷入死循环，kswapd\_main 最多会对全部虚存空间的链表尝试 rounds=16 次遍。

2. 通过 page\_launder 函数，遍历 inactive\_list，实现页的换出。这部分和下面 refill\_inactive\_list 操作的先后顺序并不那么严格。通俗的解释就是 page\_launder 实现的是把 inactive\_list 中的 page 洗净，并完成 page 的释放，当然也顺便实现了把实际上活跃的(active)的 page 从 inactive\_list 上取下，放回 active\_list 的过程；而 refill\_inactive\_list 从函数名上可以看出，实际上就是遍历 active\_list，把实际上不活跃的(inactive) page 从 active\_list 上取下，放到 inactive\_list 上，方便下一轮 page\_launder 的操作。后者没有什么需要特别强调的，但是 page\_launder 是比较复杂的过程，我们需要仔细的分析一下。page\_launder 先检查一个 page 的 page\_ref 是否 !=0，如果满足，则表示该 page 实际上是 active 的，则把它移动到 active\_list 上去；如果不是，则需要对该页进行换出操作，过程如下：注意，下面讨论的是以 page\_ref == 0 作为前提的。

(\*) page\_launder 的实现，涉及 ucore 内核代码设计的一个重要假设前提，这是第一次涉及，以后的各个模块也会逐步大量涉及。这部分和进程调度又有一定的关联，提前了解一下，有助于理解这部分以后后续其它部分的代码。这个前提就是：ucore 的内核代码是不可抢占的，也就是，执行在内核部分的代码，只要不是以下几种情况，通常可以认为是操作不会被抢占(preemption)，即 CPU 控制权被剥夺。1. 主动释放 cpu 执行权限，比如调用 schedule 让其它程序执行；2. 进行同步互斥操作，比如争抢一个信号量、锁；3. 进行磁盘等等异步操作，由于 kmalloc 有可能会调用 alloc\_pages 来分配页，而 alloc\_pages 可能失败进而将 cpu 让渡给 kswapd，所以，内核中的 kmalloc 操作可以认为不是一个同步的操作。所有这样的非同步的操作一个可能的问题，就是所有执行比如 kmalloc 这样的函数之前所做出的各种条件判断，在 kmalloc 之后可能都不再成立了。

你可以理解下面两段程序在运行时的差异，其中 list 是一个全局变量，并且可能被任何程序在执行内核服务的时候修改掉：

```
parta:
if (!list_empty(list)) {
    ptr = (uintptr_t)kmalloc(sizeof(uint32_t));
    do_something(list_next(list), buffer);
    kfree(ptr);
}

partb:
ptr = (uintptr_t)kmalloc(sizeof(uint32_t));
if (!list_empty(list)) {
    do_something(list_next(list), buffer);
}
kfree(ptr);
```

除此之外，中断处理的代码也需要进一步考虑进来。当内核尝试修改一部分数据的时候，如果该数据是中断处理流程可能访问的数据，那么内核需要对这次修改屏蔽中断；同理，如果中断处理需可能修改一部分数据，并且内核打算尝试读取该数据，那么内核需要对读操作屏蔽中断，等等。

(1) 如果一个页的 `mem_map` 项也为 0：前面已经讨论过 `mem_map` 和 `page_ref` 之间的关系。如果此时，这个页的 `mem_map` 项也为 0，说明这个时候已经没有 PTE 映射指向它们了，无论是物理页还是 swap 备份页。那么这个页也就没有必要洗净了。可以直接释放物理页以及相应的 swap entry 了。（同时记得处理 swap 有关的链表，以下不再赘述）

(2) 如果一个页的 `mem_map` 项不为 0，但是没有 `PG_dirty` 标记：page 数据结构里面有 `PG_dirty` 标记，swap 部分的代码根据这个标记来判断一个页是否需要被洗净（写到 swap 分区上）。这个 `PG_dirty` 标记在什么情况下设置，我们稍后会讨论。这种情况可以等价于物理页上的数据和 swap 分区上的数据是一致的，所以不需要洗净该页，因为该物理页本身就已经足够干净了。所以可以安全的和 (1) 中的操作一样对该物理页进行释放。

(3) 如果一个页的确有 `PG_dirty` 标记：表示该页需要被洗净。ucore 这里的实现存在一个 bug，最新的代码已经修复了这个 bug，不过 bug 对于理解这段代码没有影响。

先清楚，洗净一个页，需要调用 `swapfs_write` 函数，完成将物理页写到磁盘上的操作。前面已经强调过，我们假设所有磁盘操作是异步 IO。先明确一下，当前的状态，`page_ref=0 &&mem_map!=0 && PG_dirty`，那么写出的过程中就可能发生下面许多种可能的场景：

- a. `swapfs_write` 操作失败了。磁盘操作不像内存操作，它应该允许发生更多的错误。
- b. 其它进程又访问到相应的数据页，前面提到过，因为 PTE 已经被修了，所以会产生 #PF，内核会根据 PTE 的内容在 swap hash 里面查找到相应的物理页，并将它重新插入到相应的页表中，并更新该 page 的 `page_ref` 和 `mem_map` 的值。整个过程发生时，`swapfs_write` 还没有结束。那么当完成洗净一个页的操作时（写到 swap 分区），swap 部分的代码应该有能力检测出这种变化。也就是在 `swapfs_write` 之后，需要再判断 `page_ref` 是否依然满足 `inactive` 的。
- c. 和 b 类似，不过不同的是，这次对物理页进行的是一个写操作。操作完成之后，进程又将该 PTE 指向的页释放掉了。那么当 `swapfs_write` 返回的时候，它面对的条件，可能就变成了 `page_ref=0 &&mem_map=0 &&PG_dirty`。它应该能够处理这个变化。
- d. 和 c 类似，不同的是，该 page 有两个不同的 PTE 映射。那么在 `swapfs_write` 操作之前，状态可能是 `page_ref =0&&mem_map=2&&!PG_dirty`，那么当 c 中的情况发生以后，该物理页的状态就可能变成了 `page_ref=0&&mem_map=1&&PG_dirty` 了。swap 应该能够处理这种变化。

综上所述，`page_laundry` 部分的代码变得相对复杂很多。大家可以参照程序了解 ucore 是怎么解决这种冲突的。

最后，提及一下那个 bug。因为 `swapfs_write` 是异步操作，并且是对该 page 的操作，ucore 为了保证在操作的过程中，该页不被释放（比如一个进程通过 #PF，增加 `page_ref` 到 1，然后又通过释放该 page 减少 `page_ref` 到 0，进而触发内核执行 `page_free` 的操作），分别在 `swapfs_write` 前后获得和释放该 page 的引用（`page_ref_inc/page_ref_dec`）。但事实证明，这种担心是多余的。理由很简单，当 `page_laundry` 操作一个页的时候，该页是被标记 `PG_swap` 的，这个标记一方面表示 page 结构中的 `index` 有意义，另一方面也代表了，这样的 page 的释放，只能由 swap 部分的代码来完成（参见 `pmm.c` 以及后面 `shmem.c` 的处理）。所以，swap 在操作该 page 的时候，不可能有程序能够调用 `free_page` 释放该 page。

而相反的，`mem_map` 是一个需要保护的数据。这个是产生 bug 的地方，有兴趣的同学可以去自己理解一下。

此外，可以翻阅一下涉及到 `page_ref` 修改的 `pmm` 部分的代码，不难发现，当一个 page 从 PTE 断开的时候，也就是 `page_ref` 下降的时候，内核会根据 PTE 上的硬件设置

的 PTE\_D 来设置 PG\_dirty。其实这就足够了。因为 PG\_dirty 并不需要时时刻刻都十分的准确，只要在 swap 尝试判断该 page 是否需要洗净的时候，PG\_dirty 是正确的，就足够了。所以只需要保证每次 page\_ref 下降的时候，PG\_dirty 是正确的即可。除此之外，在对每个页分配 swap\_entry 的时候，需要保证标记 PG\_dirty，因为毕竟是刚刚分配的，物理页的数据还从来没有写出去过。

总结一下，页换入换出的实现很复杂，但是相对独立。并且正是由于 ucore 的内核代码不可抢占使得实现变得相对容易一些。只要是不涉及 IO 操作，大部分过程都可以认为处于不可抢占的内核执行过程。

### 3.11.10 proj9.1: 实现共享内存

实现共享内存功能的目的是为将来（lab5 中才需要）不同进程（process）之间能够通过共享内存实现数据共享。共享内存机制其实是一种进程间通信的手段。proj9/proj9.1 完成了不同页表中（目前仅局限于父子进程之间，lab3 才涉及）的虚拟地址共享同一块物理地址空间的功能。由于目前的实现仅限于有亲属关系的进程，这实际上意味着这些具有共享物理地址空间的虚拟地址空间也是相同的。

#### 一 相关数据结构

这部分的具体实现工作主要在 kern/mm/shmem.[ch] 和其他一些文件中。根据前面的分析，我们知道在 mm\_struct 层面上管理了基于同一个页表的 vma 集合，这些集合表示了当前可“合法”使用（即使没有对应的物理内存）的所有虚拟地址空间集合。当前的 vma\_struct 定义如下：

```
struct vma_struct {
    struct mm_struct *vm_mm;
    uint32_t vm_start;
    uint32_t vm_end;
    uint32_t vm_flags;
    vma_entry_t vma_link;
};
```

这个是在 proj9.1 以前的 vma\_struct 结构定义。由于 vma 中并没有描述此虚拟地址对应的物理地址空间，所以无法确保不同页表中描述同一虚拟地址空间的 vma 指向同一个物理地址空间，所以不同页表间的内存无法实现共享。于是我们可以对 vma\_struct 增加下面两个域（field）：

```
struct shmem_struct *shmem;
size_t shmem_off;
```

shmem 的作用是统一维护不同 mm\_struct 结构（即不同页表）中描述具有共享属性的同一虚拟地址空间的唯一物理地址空间。如果 vma->flags 里面有 VM\_SHARE，就表示 vma->shmem 有意义，他指向一个 shmem\_struct 结构的指针。shmem\_struct 结构定义如下：

```
struct shmem_struct {
    list_entry_t shmn_list;
    shmn_t *shmn_cache;
    size_t len;
```

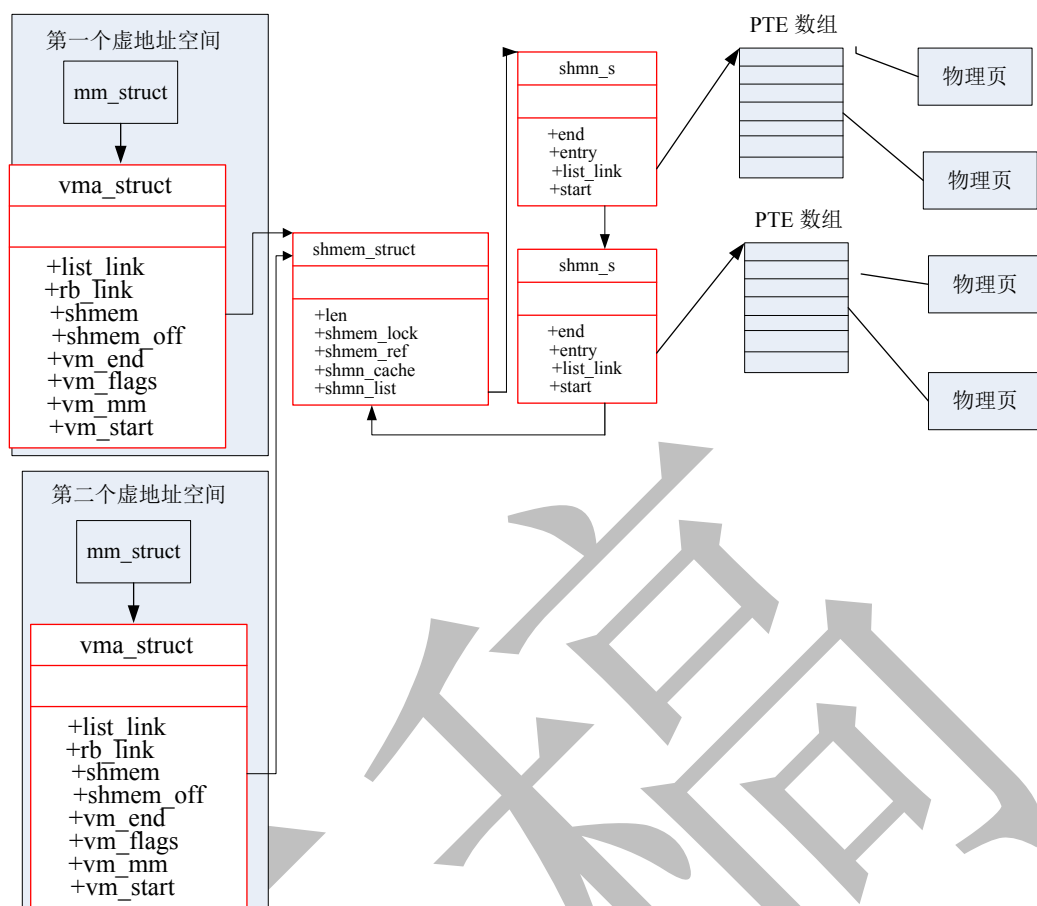
```
atomic_t shmem_ref;
lock_t shmem_lock;
};
```

shmem\_struct 包含了 list\_entry\_t 结构的 shmn\_list, 此链表的元素是 shmn\_t 结构的共享页描述 (包含某共享虚拟页的信息 (page 或者 swap entry), 为了维护起来简便, 里面借用了页表的 PTE 描述方式, 除了 PTE\_P 用来区分是否是一个物理页以外, 没有任何其它权限标记, 所以后面提到的 PTE 应该是带引号的), 所以此链表就是用来存储虚拟地址空间的 PTE 集合, 即此共享的虚拟地址空间对应的唯一的物理地址空间的映射关系。shmem\_ref 指出了当前有多少进程共享此共享虚拟空间。shmn\_t 是用来描述一段共享虚拟空间的信息, 可理解为一个 shmem node 结构, 其定义如下:

```
typedef struct shmn_s {
    uintptr_t start;
    uintptr_t end;
    pte_t *entry;
    list_entry_t list_link;
} shmn_t;
```

在这个结构中, entry 保存了一块 (4KB) 连续的虚拟空间的 PTE 数组, 可以理解为一个二级页表, 这个页表最大可以描述 4MB 的连续虚拟空间对应的物理空间地址信息。entry 中的每一项用于保存 physical address | PTE\_P 或者 swap\_entry。这样能最大限度节约内存, 并且能很快通过 entry 项计算出 对应的 struct page。

而 list\_link 是用来把自身连接属于同一 shmem\_struct 结构的域-shmn\_list 链表, 便于 shmem\_struct 结构的变量对共享空间进行管理。这样就可以形成如下图所示的共享内存布局:



## 二 创建和访问共享内存的实现逻辑

为了创建一块共享内存，首先需要在内核中调用 do\_shmem 函数(此函数要到 lab3 的 proj12 才会出现)，do\_shmem 函数并进一步调用 shmem\_create 函数创建一个 shmem\_struct 结构的变量，然后再调用 mm\_map\_shmem 函数来创建一个 vma 结构的变量，并设置 vma 的属性为 VM\_SHARE，并把 vma->shmem 指向 shmem 结构变量。这样就初步建立好了一个共享内存虚拟空间。

由于并没有给这块共享内存虚拟空间实际分配物理空间，所以在第一次访问此 vma 中某地址时，会产生缺页异常 (page fault)，并触发 do\_pgfault 函数被调用。此函数如果发现是“合法”的共享内存虚拟空间出现的地址访问错时，将调用 shmem 的处理函数 shmem\_get\_entry，来查找此共享内存虚拟子空间所在的其他虚拟地址空间中（具有不同的页表）是否此虚地址对应的 PTE 项存在，如果不存在，则表明这是第一次访问这个共享内存区域，就可以创建一个物理页，并在 shmn\_s 的 entry 中记录虚拟页与物理页的映射关系；如果存在，则在本虚拟地址空间中建立此虚地址对应的 PTE 项，这样就可以确保不同页表中的共享虚拟地址空间共享一个唯一的物理地址空间。

shmem 结构中增加一个计数器，在执行复制某虚拟地址空间的 copy\_mm 函数的时，如果 vm\_flags 有 VM\_SHARE，则仅增加 shmem 计数器 shmem\_ref，而不用再创建一个 shmem 变量；同理，释放具有共享内存属性的 vma 时，应该减少 shmem 计数器 shmem\_ref，当 shmem 计数器 shmem\_ref 减少到 0 的时候，应该释放 shmem 所占有的资源。

另外，在 shmem 里面不能记录地址，因为不同 vma 可能 map 到不同的地址上去，因此它只维护一个 page 的大小。上面提到的 shmem\_off 的作用是定位页面。具有共享属性的 vma 创建的时候，shmem\_off = 0。当 vma->vm\_start 增加的时候（只可能变大，因为内核不支持它减小，unmap 的时候可能导致 vma 前面部分的 unmap，这就可能会让 vm\_start 变大），应该将 vm\_start 的增量赋给 shmem\_off，以保证剩下的 shmem 能够访问正确的位置。这样在访问共享内存地址 addr 发生缺页异常的时候，此地址对应的页在 shmem\_struct 里面的 PTE 数组项的索引 index 应该等于  $(addr - vma->vm\_start + vma->shmem\_off) / PGSIZE$

#### 【注意】

在页换出操作中，尝试换出一页的条件是页的 page\_ref 到 0。为了防止 share memory 的 page 被意外的释放掉，shmem 结构也会增加相应数据页的引用计数。那么对于一个 share memory 的数据页，是不是就不能换出了？前面提到，页换出操作的第一步是扫描所有的虚拟地址空间，那么页换出操作就完全有能力知道当前扫描的 vma 是普通的 vma 还是对应的 share memory 的 vma。正如 swap.c 里面看到的那样，swap 断开一个 page 以后，如果发现当前 vma 是 share memory 的，并且 page\_ref 是 1，那么可以确定的是这个最后一个 page\_ref 是在 shmem 结构中。那么 swap 也同时将该 share memory 上的 PTE 断开，就满足了 page\_launer 的换出条件。

share memory 上的 entry 换成了 swap entry 带来的坏处也很明显，因为标记 share 的 vma 如果一开始没有页表内容，需要通过 #PF 从 shmem 里面得到相应的 PTE。但是不幸的是，得到的是 swap entry，那么只能通过第二次 #PF，才能将 swap entry 替换成数据页。

### 3.11.11 proj9.2: 实现写时复制

proj9.2 实现了写时复制（Copy On Write，简称 COW）的主要功能，为 lab3 高效地创建子进程打下了基础。COW 有何作用？这里又不得不提前讲讲 lab3 中的子进程创建。不同的进程应该具有不同的物理内存空间，当用户态进程发出 fork( ) 系统调用来创建子进程时，ucore 可复制当前进程（父进程）的整个地址空间，这样就有两块不同的物理地址空间了，新复制的那一块物理地址空间分配给子进程。这种行为是非常耗时和占内存资源的，因为它需要为子进程的页表分配页面，复制父进程的每一个物理内存页。如果子进程加载一个新的程序开始执行（这个过程会释放掉原来申请的全部内存和资源），这样前面的复制工作就白做了，完全没有必要。

为了解决上述问题，ucore 采用一种有效的 COW 机制。其设计思想相对简单：父进程和子进程之间共享（share）页面而不是复制（copy）页面。但只要页面被共享，它们就不能被修改，即是只读的。注意此共享是指父子进程共享一个表示内存空间的 mm\_struct 结构的变量。当父进程或子进程试图写一个共享的页面，就产生一个页访问异常，这时内核就把这个页复制到一个新的页面中并标记为可写。注意，原来的页面仍然是写保护的。当其它进程试图写入时，ucore 检查写进程是否是这个页面的唯一属主（通过判断 page\_ref 和 swap\_page\_count 即 mem\_map 中相关 entry 保存的值的和是否为 1。注意区分与 share memory 的差别，share memory 通过 vma 中的 shmem 实现，这样的 page 是直接标记为共享的，而不是 copy on write，所以没有任何冲突）；如果是，它把这个页面标记为对这个进程是可写的。

在具体实现上，ucore 调用 dup\_mmap 函数，并进一步调用 copy\_range 函数来具体完成对页表内容的复制，这样两个页表表示同一个虚拟地址空间（包括对应的物理地址空间），

---

且还需修改两个页表中每一个页对应的页表项属性为只读，但。在这种情况下，两个进程有两个页表，但这两个页表只映射了一块只读的物理内存。同理，对于换出的页，也采用同样的办法来共享一个换出页。综上所述，我们可以总结出：如果一个页的 PTE 属性是只读的，但此页所属的 VMA 描述指出其虚地址空间是可写的，则这样的页是 COW 页。

当对这样的地址空间进行写操作的时候，会触发 do\_pgfault 函数被调用。此函数如果发现是 COW 页，就会调用 alloc\_page 函数新分配一个物理页，并调用 memcpy 函数把旧页的内容复制到新页中，并最后调用 page\_insert 函数给当前产生缺页错的进程建立虚拟页地址到新物理页地址的映射关系（即改写 PTE，并设置此页为可读写）。

这里还有一个特殊情况，如果产生访问异常的页已经被换出到硬盘上了，则需要把此页通过 swap\_in\_page 函数换入到内存中来，如果进一步发现换入的页是一个 COW 页，则将其属性设置为只读，然后异常处理结束返回。但这样重新执行产生异常的写操作，又会触发一次内存访问异常，则又要执行上一段描述的过程了。

Page 结构的 ref 域用于跟踪共享相应页面的进程数目。只要进程释放一个页面或者在它上面执行写时复制，它的 ref 域就递减；只有当 ref 变为 0 时，这个页面才被释放。

## 操作系统设计与实现

## The Design & Implementation of Operating System

---

陈渝 向勇

清华大学计算机系  
2011 年

## 第4章 进程管理与调度

“进程”(process)是 20 世纪 60 年代初首先由 MIT 的 MULTICS 系统和 IBM 公司的 CTSS/360 系统率先引入的概念。简单地说,进程是一个正在运行的程序。但传统的程序本身是一组指令的集合,是一个静态的概念。程序在一方面无法描述程序在内存中的动态执行情况,即无法从程序的字面上看出它何时执行,何时结束;另一方面,在内存中可存在多个程序,这些程序分时复用 CPU,但无法清楚地表达程序间关系(比如父子关系、同步互斥关系等)。因此,程序这个静态概念已不能如实反映多程序并发执行过程的特征。

为了从根本上描述程序动态执行过程的性质,计算机科学家引入了“进程(Process)”概念。在计算机系统中,由于 CPU 的速度非常快(现在的通用 CPU 主频达到 2GHz 是很平常的事情),只让它做一件事情无法充分发挥其能力。我们可以“同时”运行多个程序,这个“同时”,其实是操作系统给用户造成的一个“错觉”。大家知道,CPU 是计算机系统中的硬件资源。为了提高 CPU 的利用率,在内存中的多个程序可分时复用 CPU,即如果一个程序因某个事件而不能继续执行时,就可把 CPU 占用权转交给另一个可运行程序。为了刻画多各程序的并发执行的过程,就引入了“进程”的概念。从操作系统原理上看,一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。操作系统中的进程管理需要协调多道程序之间的关系,解决对处理器分配调度策略、分配实施和回收等问题,从而使得处理器资源得到最充分的利用。

操作系统需要管理这些进程,使得这些进程能够公平、合理、高效地分时使用 CPU,这需要操作系统完成如下主要任务:

- 进程生命周期管理:创建进程、让进程占用 CPU 执行、让进程放弃 CPU 执行、销毁进程;
- 进程分派(dispatch)与调度(scheduling):设定进程占用/放弃 CPU 的时机、根据某种策略和算法选择将占用的 CPU(这就是调度),完成进程切换;
- 进程内存空间保护:给每个进程一个受到保护的地址空间,确保进程有独立的地址空间,不会被其他进程非法访问;
- 进程内存空间等资源共享:提供内存等资源共享机制,可以使得不同进程可共享内存等资源;
- 系统调用机制:给用户进程提供访问操作系统功能的接口,即系统调用接口。

本章内容主要涉及操作系统的进程管理与调度,并能够利用 lab2 的虚存管理功能实现高效的进程中的内存管理。读者通过阅读本章的内容并动手实践相关的 lab3 和 lab4 种的 9



个 project 实验：

- Proj10: 创建进程控制块和内核线程。
- Proj10.1: 实现用户进程、读和加载 ELF 格式执行程序、一个简单的调度器，以及提供创建（fork）/execve（执行）/ 放弃对 CPU 的占用（yield）等系统调用实现。
- Proj10.2: 完成等待子进程结束（wait），杀死进程（kill），进程自己退出（exit）等系统调用，从而完善了进程的生命周期管理。
- Proj10.3: 实现 sys\_brk 系统调用和相应的用户进程内存管理，从而与 lab2 的内存管理进一步联合在一起。
- Proj10.4: 让进程可以睡眠和被唤醒。
- Proj11: 创建 kswapd 内核线程来专门处理内存页的换入和换出。
- Proj12: 基于进程间内存共享（proj9.1）实现父子进程数据共享，并实现了用户态的线程机制。
- Proj13: 设计实现了通用的调度器框架
- Proj13.1/Proj13.2: 在通用调度器框架下实现了轮转（RoundRobin, RR）调度器/多级反馈队列（Multi Level Feedback Queue, MLFQ）调度器

可以掌握如下知识：

- 与操作系统原理相关
  - 进程管理：进程状态和进程状态转换
  - 进程管理：进程创建、进程删除、进程阻塞、进程唤醒
  - 进程管理：父子进程的关系和区别
  - 进程管理：进程中的内存管理
  - 进程管理：用户进程、内核进程、用户线程、内核线程的关系区别
  - 进程管理：线程的特征和实现机制
  - 进程调度：进程调度算法
- 操作系统原理之外
  - 页面换入换出的内核线程实现技术
  - 父子进程数据共享实现
  - 通用的调度器框架
  - 进程切换的实现细节

本章内容中主要涉及进程管理的重要功能主要有两个：

- 进程生命周期的管理：如何高效地创建进程、切换进程、删除进程和管理进程对资源的需求（内存和 CPU）涉及一系列的动态管理机制。线程的加入使得整个系统的执行效率更高，这需要根据线程的特点设计与进程不同的线程组织和管理机制。
- 进程调度算法：进程调度（部分教科书也称为处理器调度）算法主要是选择响应时间决定应该由哪个进程占用 CPU 来执行。这里需要确保通过调度来提高整个系统的吞吐量和减少响应时间。

为了让读者能够从实践上来理解进程管理和调度的基本原理，我们设计了上述实验，主要的功能逐步实现如下所示：

- 首先需要能够对运行的程序进行管理，这需要一个“档案”，进程控制块（Process Control Block）；

- 有了进程控制块，我们就可以实现不同特点的进程或线程，这里首先实现了相对简单的内核线程；
- 为了能够执行应用程序，还需通过进程控制块实现用户进程的管理，能够创建/删除/阻塞/唤醒进程，从而能够对用户进程的整个生命周期进行全程管理；
- 由于在内存中有多个进程，但只有一个 CPU，所以需要设计合理的调度器，让不同进程能够分时复用 CPU，从而提高整个系统的吞吐量和减少响应时间。

## 4.1 创建并执行内核线程

### 4.1.1 实验目标

ucore 在 lab2 完成了内存管理。一个程序如果要加载到内存中运行，通过 ucore 的内存管理就可以分配合适的空间了。接下来就需要考虑如何使用 CPU 来“并发”执行多个程序。

操作系统把一个程序加载到内存中运行，这个运行的程序会经历从“出生”到“死亡”的整个“生命”历程。这个运行程序的整个执行过程就是进程。为了记录、描述和管理程序执行的动态变化过程，需要有一个数据结构，这个就是进程控制块。一个进程与一个进程控制块一一对应。为此，ucore 就需要建立合适的进程控制块数据结构，并基于进程控制块来完成对进程的管理。

### 4.1.2 proj10 概述

#### 1. 实现描述

project10 是 lab3 的第一个项目，它基于 lab2 的最后一个项目 proj9.2。主要就是扩展了进程控制块的数据结构，并基于进程控制块实现了对进程的初步管理。并通过创建两个内核线程 idleproc 和 init\_main 来实现了对 CPU 的分时使用。

#### 2. 项目组成

```
proj10
├── .....
│   ├── process
│   │   ├── entry.S
│   │   ├── proc.c
│   │   ├── proc.h
│   │   └── switch.S
│   ├── schedule
│   │   ├── sched.c
│   │   └── sched.h
│   └── sync
```

```

|   |   └── sync.h
|   └── trap
|       ├── trap.c
|       ├── .....
|       └── trapentry.S
└── libs
    ├── .....
    └── unistd.h
        .....
14 directories, 77 files

```

相对与 proj9.2, proj10 增加了 7 个文件, 修改了相对重要的 3 个文件。主要修改和增加的文件如下:

- process/proc.[ch]: 实现了进程控制块的定义和基于进程控制块的各种进程管理函数, 实现了对进程生命周期管理的绝大部分功能。
- process/entry.S: 内核线程的起始入口处和结束处理 (通过调用 `do_exit` 完成实际结束工作)。
- process/switch.S: 实现了进程上下文 (context) 切换的函数 `switch_to`, 由于与硬件相关, 所以直接采用汇编实现。
- schedule/sched.[ch]: 实现了一个先进先出 (First In First Out) 策略的进程调度。
- trap/trap.c, trapentry.S:
- unistd.h: 定义了一系列系统调用号, 为后续用户进程访问内核功能做准备, 这里暂时用不上。

### 3. 编译运行

编译并运行 proj10 的命令如下:

```
make
```

```
make qemu
```

则可以得到如下显示界面

```

thuos:~/oscourse/ucore/i386/lab3_process/proj10$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
entry 0xc010002c (phys)
etext 0xc0113bd7 (phys)
edata 0xc013fab0 (phys)
end   0xc0144e74 (phys)
Kernel executable memory footprint: 276KB
.....

```

```

check_mm_shm_swap: step2, dup_mmap ok.
check_mm_shm_swap() succeeded.
++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:317:
process exit!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

```

从上图可以看到，proj10 创建了一个内核线程 `init_main`，然后启动内核线程 `initproc` 运行，此线程调用 `init_main` 函数完成了其主要的工作，即输出了一些信息：

```

this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"

```

然后就“死亡”了，所占用的资源被回收。由于现在没有其他值得执行的线程，所以 `ucore` 就进入到 `kernel debug monitor` 了。到底是在哪里判断并进入 `monitor` 的呢？我们只需在“K>”提示符后面输入 `backtrace`，就可得到如下输出：

```

K> backtrace
ebp:0xc7eb0ee8 eip:0xc0101c86 args:0x00000000 0x00000000 0xc7eb0f68 0xc0102489
    kern/debug/kdebug.c:298: print_stackframe+21
ebp:0xc7eb0ef8 eip:0xc010258b args:0x00000000 0xc7eb0f1c 0x00000000 0x00000000
    kern/debug/monitor.c:147: mon_backtrace+10
ebp:0xc7eb0f68 eip:0xc0102489 args:0xc013fac0 0x00000000 0xc011784a 0xc7eb0fdc
    kern/debug/monitor.c:93: runcmd+134
ebp:0xc7eb0f98 eip:0xc010250d args:0x00000000 0xc7eb0fdc 0x0000013d 0xc7eb0fcc
    kern/debug/monitor.c:114: monitor+91
ebp:0xc7eb0fc8 eip:0xc0102b2f args:0xc0117825 0x0000013d 0xc0117839 0xc0144e44
    kern/debug/panic.c:30: __panic+106
ebp:0xc7eb0fe8 eip:0xc0112bc7 args:0x00000000 0xc01178b8 0x00000000 0x00000010
    kern/process/proc.c:317: do_exit+33
K>

```

这里可以清楚的看到，在 `proc.c` 的 317 行（`do_exit` 函数内）调用了 `panic` 函数，导致进入了 `monitor`。`idleproc` 和 `initproc` 是 `ucore` 里面两个特殊的内核线程，它们是不允许退出的，所以 `do_exit` 里面直接调用 `panic` 了，对于其它普通的进程而言，`do_exit` 实际上还有很多工作要做，后续章节会进一步讲到。上述执行过程其实包含了对进程整个生命周期的管理。下面我们将从原理和实现两个方面对此进行进一步阐述。

### 4.1.3 【原理】进程的属性与特征解析

为了让多个程序能够使用 CPU 执行任务，我们需要设计进程控制块，需要进一步管理进程。但到底如何设计进程控制块，如何管理进程？如果我们对进程的属性和特征了解不够，

则无法有效地设计进程控制块和实现进程管理。

再一次回到进程的定义：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。这里有四个关键词：**程序、数据集合、执行、动态执行过程**。从 CPU 的角度来看，所谓程序就是一段特定的指令机器码序列而已。CPU 会一条一条地取出在内存中程序的指令并按照指令的含义执行各种功能。所谓数据集合就是使用的内存，所谓执行就是让 CPU 工作。那么这个数据集合和执行其实体现了进程对资源的占用。动态执行过程体现了程序执行的不同“生命”阶段：诞生、工作、休息/等待、死亡。如果这一段指令执行完毕，也就意味着进程结束了。从开始执行到执行结束是一个进程的全过程。那么操作系统需要管理进程的什么？如果计算机系统中只有一个进程，那操作系统的工作就简单了。其实就是管理进程执行的指令，进程占用的资源，进程执行的状态。这可归结为对一个进程内的管理工作。但实际上在计算机系统的内存中，可以放很多程序，这也就意味着操作系统需要管理多个进程，那么，还需要做有关进程间的其他管理工作包括：进程调度、进程间的数据共享、进程间执行的同步互斥关系（lab5 涉及）等。下面逐一进行解析。

### 1. 指令执行安全管理

CPU 的指令有一般指令和特权指令之分，那么由不包含特权指令的一般指令集合组成了用户程序，而由特权指令加上一般指令形成的核心软件就是操作系统了。CPU 在执行时可处于不同的特权模式：用户态模式和核心态模式，在用户态模式只能执行一般指令，在核心态模式除了可以执行一般指令外，还可以执行特权指令。如果这里的程序是指的一般应用程序或用户程序，不是操作系统，那么我们把在 CPU 处于用户态特权模式下的应用程序的执行过程称为用户进程。如果这里的程序指的是操作系统，那么我们把在 CPU 处于核心态特权模式下的操作系统的执行过程称为内核进程。回想 lab1 和 lab2 中内容，物理内存存在操作系统的内存管理下也有类型，可以分为用户态内存和核心态内存，这两种内存形成了 CPU 可能访问的所有内存空间。内核进程可以干任何事情，所以对于各种内存，各种指令，它都能访问和执行。用户进程和内核进程对内存和指令的权限如下所示：

|          | CPU 特权模式 | 本身的用<br>户态内存 | 其他进程的<br>用户态内存 | 核心态内<br>存 | 一般指令 | 特权指令 |
|----------|----------|--------------|----------------|-----------|------|------|
| 用户<br>进程 | 用户态模式    | 可访问          | 不可访问           | 不可访问      | 可执行  | 执行无效 |
| 内核<br>进程 | 核心态模式    | 可访问          | 可访问            | 可访问       | 可执行  | 可执行  |

用户进程本身要完成的工作细节由一条一条指令具体体现。对于一般指令完成“合法”的功能，操作系统不用管理。操作系统要管理的是用户进程的“非法”指令。简单地说，“非法”指令包括：特权指令、引起异常的一般指令（比如除零操作）和访问不属于它的内存空间。

而且操作系统会在用户进程执行前，设置一个用户进程执行的用户态环境，即如果程序的指令流一开始执行，操作系统就让 CPU 处于用户态特权模式。而且还要分配一定的物理空间，建立页表，给进程一个用户态虚拟内存空间，这样一个用户态进程就可以在这个限定的虚拟内存空间中正常执行一般指令了。一旦用户进程执行了“非法”指令，则 CPU 会产生异常，CPU 使用权将转到操作系统中，从而让操作系统能够对执行“非法”指令的用户进程进行管理，比如让进程退出、给进程分配更大的内存空间等。这样操作系统能够确保用户进程在执行过程中无法破坏在内存中的其他用户进程和操作系统本身。

## 2. 资源管理

在计算机系统中,进程会占用内存和 CPU,这都是有限的资源,如果不进行合理的管理,资源会耗尽或无法高效公平地使用,从而会导致计算机系统多个进程执行效率很低,甚至由于资源不够而无法正常运行。

对于用户态进程而言,操作系统是它的“上帝”,操作系统给了用户态进程可以运行所需的资源,最基本的资源就是内存和 CPU。在 lab2 中涉及的内存管理方法和机制可直接应用到进程的内存资源管理中。在多个进程存在的情况下,对于 CPU 这种资源,则需要通过进程调度来合理选择一个进程,并进一步通过进程分派和进程切换让不同的进程分时复用 CPU,执行各自的工作。进程调度的核心是各种进程调度算法,而调度算法的评价指标是高效、合理、公平、系统吞吐量大、响应时间短等。另外,对于无法剥夺的共享资源,如果资源管理不当,多个进程会出现死锁或饥饿现象。

## 3. 状态管理

用户进程有不同的状态(也可理解为“生命”的不同阶段),当操作系统把程序的放到内存中后,这个进程就“诞生”了,不过还没有开始执行,但已经消耗了内存资源,处于“创建”状态;当进程准备好各种资源,就等能够使用 CPU 时,进程处于“就绪”状态;当进程终于占用 CPU,程序的指令被 CPU 一条一条执行的时候,这个进程就进入了“工作”状态,也称“运行”状态,这时除了进一步占用内存资源外,还占用了 CPU 资源;当这个进程等待某个资源而无法继续执行时,进程可放弃 CPU 使用,即释放 CPU 资源,进入“等待”状态;当程序指令执行完毕,进程进入了“死亡”状态。

这些状态的转换时机需要操作系统管理起来,而且进程的创建和清除等工作必须由操作系统提供,而且从“运行”态与“就绪”态/“等待”态之间的转换,涉及到保存和恢复进程的“执行现场”,也成为进程上下文,这是确保进程即使“断断续续”地执行,也能正确完成工作的必要保证。

## 4. 系统调用

操作系统把用户进程现在在用户态特权模式下执行,这使得用户进程无法完成各种重要的工作,比如获取内存、访问硬盘内容等。为了解决这个问题,用户进程可通过执行系统调用来通知操作系统帮助它来完成这些需要在特权态下才能执行的重要工作。执行系统调用会使得 CPU 从用户态特权模式切换到核心态特权模式,在用户进程的用户态执行现场(用户态进程运行上下文)也需要保存在操作系统中,当操作系统完成用户进程请求的工作后,还需根据保存的用户态执行现场恢复用户进程的正常执行。

## 5. 进程与线程

一个进程拥有一个存放程序和数据等的虚拟地址空间以及其他资源。一个进程基于程序的指令流执行,其执行过程可能与其它进程的执行过程交替进行。因此,一个具有执行状态(运行态、就绪态等)的进程是一个被操作系统调度并分派的单位。在大多数操作系统中,这两个特点是进程的主要本质特征。但这两个特征相对独立,操作系统可以把这两个特征分别进行管理。

这样可以把拥有资源所有权的单位通常仍称作进程,对资源的管理成为进程管理;把指令执行流的单位称为线程,对线程的管理就是线程调度和线程分派。对属于同一进程的所有线程而言,这些线程共享进程的虚拟地址空间和其他资源,但每个线程都有一个独立的栈,还有独立的线程运行上下文用于包含表示线程执行现场的寄存器值等信息。

在多线程环境中,进程被定义成资源分配与保护的单位,与进程相关联的信息主要有存

放进程映像的虚拟地址空间等。在一个进程中，可能有一个或多个线程，每个线程有线程执行状态（运行、就绪、死亡等），保存上次运行时的线程上下文、线程的执行栈等。考虑到 CPU 有不同的特权模式，参照进程的分类，线程又可进一步细化为用户线程和内核线程。

到目前为止，我们就可以明确用户进程、核心进程、用户线程、核心线程的区别了。从本质上看，线程就是一个特殊的不用拥有资源的轻量级进程，在 ucore 的调度和执行管理中，并没有区分线程和进程。且由于 ucore 内核中的所有内核进程共享一个内核地址空间和其他资源，所以这些内核进程都是内核线程。理解了进程或线程的上述属性和特征，我们就可以进行进程/线程管理的设计与实现了。但是为了叙述上的简便，在分析 proj12（proj12 实现了用户线程）以前，以下用户态的进程/线程统称为用户进程，而内核进程/线程则统称为内核线程。

### 4.1.4 【实现】设计进程控制块

在 proj10 中，进程管理信息用 struct proc\_struct 表示，在 kern/process/proc.h 中定义如下：

```
struct proc_struct {
enum proc_state state;          // Process state
int pid;                        // Process ID
int runs;                       // the running times of Proces
uintptr_t kstack;              // Process kernel stack
volatile bool need_resched;    // need to be rescheduled to release CPU?
struct proc_struct *parent;    // the parent process
struct mm_struct *mm;          // Process's memory management field
struct context context;        // Switch here to run process
struct trapframe *tf;          // Trap frame for current interrupt
uintptr_t cr3;                 // the base addr of Page Directroy Table (PDT)
uint32_t flags;                // Process flag
char name[PROC_NAME_LEN + 1]; // Process name
list_entry_t list_link;        // Process link list
list_entry_t hash_link;        // Process hash list
};
```

在上述域中，与进程管理各个相关层面的对应关系如下表所示：

| 安全管理     | 资源管理                                                           | 状态管理                                  | 系统调用                        | 进程/线程                                                          |
|----------|----------------------------------------------------------------|---------------------------------------|-----------------------------|----------------------------------------------------------------|
| 内存安全: mm | 内存资源:mm<br>页表首地址:cr3<br>内核堆栈: kstack<br>统计占用 CPU 的次<br>数: runs | 进程状态:state<br>“运行”状态的执行现<br>场:context | 产生中断或执行<br>系统调用时执行<br>现场:tf | 内 核 线 程 :<br>mm=NULL<br>进 程 间 关 系 :<br>parent<br>进程 id: pid 进程 |

下面重点解释一下几个比较重要的域:

- **mm** : 内存管理的信息, 包括内存映射列表、页表指针等。mm 里有个很重要的项 pgdir, 记录的是该进程使用的一级页表的物理地址。
- **state**: 进程所处的状态。
- **parent** : 用户进程的父进程 (创建它的进程)。在所有进程中, 只有一个进程没有父进程, 就是内核创建的第一个内核线程 `idleproc`。内核根据这个父子关系建立进程的树形结构, 用于维护一些特殊的操作, 例如确定哪些进程是否可以对另外一些进程进行什么样的操作等等。
- **context**: 进程的上下文, 用于进程切换 (参见 `switch.S`)。在 `ucore` 中, 所有的进程在内核中也是相对独立的 (例如独立的内核堆栈以及上下文等等)。使用 **context** 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用 **context** 进行上下文切换的函数是 `switch_to`, 在 `kern/process/switch.S` 中定义。
- **tf**: 中断帧的指针, 总是指向内核栈的某个位置: 当进程从用户空间跳到内核空间时, 中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时, 需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外, `ucore` 内核允许嵌套中断。因此为了保证嵌套中断发生时 **tf** 总是能够指向当前的 `trapframe`, `ucore` 在内核栈上维护了 **tf** 的链, 可以参考 `trap.c::trap` 函数 做进一步的了解。
- **cr3**: `cr3` 保存页表的物理地址, 目的就是进程切换的时候方便直接使用 `lcr3` 实现页表切换, 避免每次都根据 **mm** 来计算 `cr3`。**mm** 数据结构是用来实现用户空间的虚存管理的, 但是内核线程没有用户空间, 它执行的只是内核中的一小段代码 (通常是一小段函数), 所以它没有 **mm** 结构, 也就是 `NULL`。当某个进程是一个普通用户态进程的时候, **PCB** 中的 `cr3` 就是 **mm** 中页表 (`pgdir`) 的物理地址; 而当它是内核线程的时候, `cr3` 等于 `boot_cr3`。而 `boot_cr3` 指向了 `ucore` 启动时建立好的内核虚拟空间的页目录表首地址。
- **kstack**: 每个进程都有一个内核栈, 并且位于内核地址空间的不同位置。对于内核线程, 该栈就是运行时的程序使用的栈; 而对于普通进程, 该栈是发生特权级改变的时候使保存被打断的硬件信息用的栈。`Ucore` 在创建进程时分配了 2 个连续的物理页 (参见 `memlayout.h`) 作为内核栈的空间。这个栈很小, 所以内核中的代码应该尽可能的紧凑, 并且避免在栈上分配大的数据结构, 以免栈溢出, 导致系统崩溃。**kstack** 记录了分配给该进程/线程的内核栈的位置。主要作用有以下几点。首先, 当内核准备从一个进程切换到另一个的时候, 需要根据 **kstack** 的值正确的设置好 `tss` (可以回顾一下在 `lab1` 中讲述的 `tss` 在中断处理过程中的作用), 以便在进程切换以后再次发生中断时能够使用正确的栈。其次, 内核栈位于内核地址空间, 并且是不共享的 (每个进程/线程都拥有自己的内核栈), 因此不受到 **mm** 的管理, 当进程退出的时候, 内核能够根据 **kstack** 的值快速定位栈的位置并进行回收。`ucore` 的这种内核栈的设计借鉴的是 `linux` 的方法 (但由于内存管理实现的差异, 它实现的远不如 `linux` 的灵活), 它使得每个进程/线程的内核栈在不同的位置, 这样从某种程度上方便调试, 但同时也使得内核对栈溢出变得十分不敏感, 因为一旦发生溢出, 它极可能污染内核中其它的数据使得内核崩溃。如果能够通过页表,



将所有进程的内核栈映射到固定的地址上去，能够避免这种问题，但又会使得进程切换过程中对栈的修改变得相当繁琐。感兴趣的同学可以参考 linux kernel 的代码对此进行尝试。

为了管理系统中所有的进程控制块，ucore 维护了如下全局变量（位于 *kern/process/proc.c*）：

- `static struct proc *current;` //当前占用 CPU，处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证操作的原子性，目前至少需要屏蔽中断，可以参考 `switch_to` 的实现，后面也会介绍到。linux 的实现很有意思，它将进程控制块放在进程内核栈的底部，这使得任何时候 `current` 都可以根据内核栈的位置计算出来的，而不用维护一个全局变量。这样使得一致性的维护以及多核的实现变得十分的简单和高效。感兴趣的同学可以参考 linux kernel 的代码。
- `static struct proc *initproc;` //指向第一个用户态进程（proj10 以后）
- `static list_entry_t hash_list[HASH_LIST_SIZE];` //所有进程控制块的哈希表，这样 `proc_struct` 中的域 `hash_link` 将基于 `pid` 链接入这个哈希表中。
- `list_entry_t proc_list;` //所有进程控制块的双向线性列表，这样 `proc_struct` 中的域 `list_link` 将链接入这个链表中。

#### 4.1.5 【实现】创建并执行内核线程

既然建立了进程控制块，我们就可以通过进程控制块来创建具体的进程了。首先，我们考虑最简单的内核线程，它通常只是内核中的一小段代码或者函数，没有用户空间。而由于在操作系统启动后，已经对整个核心内存空间进行了管理，通过设置页表建立了核心虚拟空间（即 `boot_cr3` 指向的二级页表描述的空间）。所以内核中的所有线程都不需要再建立各自的页表，只需共享这核心虚拟空间就可以访问整个物理内存了。

### 1. 创建第 0 个内核线程 `idleproc`

在 `init.c:kern_init` 函数调用了 `proc.c:proc_init` 函数。`proc_init` 函数启动了创建内核线程的步骤。首先当前的执行上下文（从 `kern_init` 启动至今）就可以看成是一个内核线程的上下文，为此 ucore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化而将其打造成第 0 个内核线程 —— `idleproc`。具体步骤如下：

首先调用 `alloc_proc` 函数来通过 `kmalloc` 函数获得 `proc_struct` 结构的一块内存——`proc`，这就是第 0 个进程控制块了，并把 `proc` 进行初步初始化（即把 `proc_struct` 中的各个域清零）。但有些域设置了特殊的值：

```
proc->state = PROC_UNINIT; //设置进程为“初始”态
proc->pid = -1;             //进程的 pid 还没设置好
proc->cr3 = boot_cr3;       //进程在内核中使用的内核页表的起始地址
```

上述三条语句中，第一条设置了进程的状态为“初始”态，这表示进程已经“出生”了，正在获取资源茁壮成长中；第二条语句设置了进程的 `pid` 为 -1，这表示进程的“身份证号”还没有办好；第三条语句表明进程如果在内核运行，则采用为内核建立的页表，即设置

了在内核的页表的起始地址，这也可进一步看出所有进程的内核虚地址空间（也包括物理地址空间）是相同的。既然内核进程共用一个映射内核空间的页表，这表示所有这些内核空间对所有内核进程都是“可见”的，所以更精确地说，这些内核进程都应该是内核线程。

接下来，`proc_init` 函数对 `idleproc` 内核线程进行进一步初始化：

```
idleproc->pid = 0;
idleproc->state = PROC_RUNNABLE;
idleproc->kstack = (uintptr_t)bootstack;
idleproc->need_resched = 1;
set_proc_name(idleproc, "idle");
```

需要注意前 4 条语句。第一条语句给了 `idleproc` 合法的身份证号--0，这名正言顺地表明了 `idleproc` 是第 0 个内核线程。“0”是第一个的表示方法是计算机领域所特有的，比如 C 语言定义的第一个数组元素的小标也是“0”。第二条语句改变了 `idleproc` 的状态，使得它从“出生”转到了“准备工作”，就差 `ucore` 调度它执行了。第三条语句设置了 `idleproc` 所使用的内核栈的起始地址。需要注意以后的其他进程/线程的内核栈都需要通过分配获得，因为 `ucore` 启动时设置的内核栈直接分配给 `idleproc` 使用了。第四条很重要，因为 `ucore` 希望当前 CPU 应该做更有用的工作，而不是运行 `idleproc` 这个“无所事事”的内核线程，所以把 `idleproc->need_resched` 设置为“1”，结合 `idleproc` 的执行主体--`cpu_idle` 函数的实现，可以清楚看出如果当前 `idleproc` 在执行，则只要此标志为 1，马上就调用 `schedule` 函数要求调度器切换其他进程执行。

【问题】为何说 `idleproc` 的执行主体是 `cpu_idle` 函数？

## 2. 创建第 1 个内核线程 `initproc`

第 0 个内核线程主要工作是完成内核中各个子系统的初始化，然后通过执行 `cpu_idle` 函数开始过退休生活了。但接下来还需创建其他进程来完成各种工作，但 `idleproc` 自己不想做，于是就通过调用 `kernel_thread` 函数创建了一个内核线程 `init_main`。在 `proj10` 中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看 `init_main` 函数）。但在后续的 `proj` 中，`init_main` 的工作就是创建特定的其他内核线程或用户进程。下面我们来分析一下创建内核线程的函数 `kernel_thread`：

```
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf_struct;
    memset(&tf_struct, 0, sizeof(struct trapframe));
    tf_struct.tf_cs = KERNEL_CS;
    tf_struct.tf_ds = tf_struct.tf_es = tf_struct.tf_ss = KERNEL_DS;
    tf_struct.tf_regs.reg_ebx = (uint32_t)fn;
    tf_struct.tf_regs.reg_edx = (uint32_t)arg;
    tf_struct.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf_struct);
}
```

注意，`kernel_thread` 函数采用了局部变量 `tf` 来放置保存内核线程的临时中断帧，并把中断帧的指针传递给 `do_fork` 函数，而 `do_fork` 函数会调用 `copy_thread` 函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间。

给中断帧分配完空间后，就需要构造新进程的中断帧，具体过程是：首先给 `tf` 进行清

零初始化，并设置中断帧的代码段（`tf.tf_cs`）和数据段（`tf.tf_ds/tf_es/tf_ss`）为内核空间的段（`KERNEL_CS/ KERNEL_DS`），这实际上也说明了 `initproc` 内核线程在内核空间中执行。而 `initproc` 内核线程从哪里开始执行呢？`tf.tf_eip` 的指出了是 `kernel_thread_entry`（位于 `kern/process/entry.S` 中），`kernel_thread_entry` 是 `entry.S` 中实现的汇编函数，它做的事情很简单：

```
kernel_thread_entry:          # void kernel_thread(void)

    pushl %edx                # push arg
    call *%ebx                # call fn

    pushl %eax                # save the return value of fn(arg)
    call do_exit              # call do_exit to terminate current thread
```

从上可以看出，`kernel_thread_entry` 函数主要为内核线程的主体 `fn` 函数做了一个准备开始和结束运行的“壳”，及把函数 `fn` 的参数 `arg`（保存在 `edx` 寄存器中）压栈，然后调用 `fn` 函数，把函数返回值 `eax` 寄存器内容压栈，调用 `do_exit` 函数退出线程执行。

`do_fork` 是创建线程的主要函数。`kernel_thread` 函数通过调用 `do_fork` 函数最终完成了内核线程的创建工作。下面我们来分析一下 `do_fork` 函数的实现。`do_fork` 函数主要做了以下 6 件事情：

1. 分配并初始化进程控制块（`alloc_proc` 函数）；
2. 分配并初始化内核栈（`setup_stack` 函数）；
3. 根据 `clone_flag` 标志复制或共享进程内存管理结构（`copy_mm` 换生）；
4. 设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行上下文（`copy_thread` 函数）；
5. 把设置好的进程控制块放入 `hash_list` 和 `proc_list` 两个全局进程链表中；
6. 自此，进程已经准备好执行了，把进程状态设置为“就绪”态。

这里需要注意的是，如果上述前 3 步执行没有成功，则需要做对应的出错处理，把相关已经占有的内存释放掉。`copy_mm` 函数目前只是把 `current->mm` 设置为 `NULL`，这是由于目前 `proj10` 只能创建内核线程，`proc->mm` 描述的是进程用户态空间的情况，所以目前 `mm` 还用不上。`copy_thread` 函数做的事情比较多，代码如下：

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    proc->context.eip = (uintptr_t) forkret;
    proc->context.esp = (uintptr_t) (proc->tf);
}
```

此函数首先在内核堆栈的顶部设置中断帧大小的一块栈空间，并在此空间中拷贝在 `kernel_thread` 函数建立的临时中断帧的初始值，并进一步设置中断帧中的栈指针 `esp` 和标志寄存器 `eflags`，特别是 `eflags` 设置了 `FL_IF` 标志，这表示此内核线程在执行过程中，能响应中断，打断当前的执行。执行到这步后，此进程的中断帧就建立好了，对于 `initproc`

而言，它的中断帧如下所示：

```
//所在地址位置
initproc->tf= (proc->kstack+KSTACKSIZE) - sizeof (struct trapframe);
//具体内容
initproc->tf.tf_cs = KERNEL_CS;
initproc->tf.tf_ds = initproc->tf.tf_es = initproc->tf.tf_ss = KERNEL_DS;
initproc->tf.tf_regs.reg_ebx = (uint32_t)init_main;
initproc->tf.tf_regs.reg_edx = (uint32_t) ADDRESS of "Hello world!!";
initproc->tf.tf_eip = (uint32_t)kernel_thread_entry;
initproc->tf.tf_regs.reg_eax = 0;
initproc->tf.tf_esp = esp;
initproc->tf.tf_eflags |= FL_IF;
```

设置好中断帧后，最后就是设置 `initproc` 的执行现场（也称进程上下文，`process context`）了。只有设置好执行现场后，一旦 `ucore` 调度器选择了 `initproc` 执行，就需要根据 `initproc->context` 中保存的执行现场来恢复 `initproc` 的执行这里设置了 `initproc` 的执行现场中主要的两个信息：上次停止执行时的下一条指令地址 `context.eip` 和上次停止执行时的堆栈地址 `context.esp`。其实 `initproc` 还没有执行过，所以这其实就是 `initproc` 实际执行的第一条指令地址和堆栈指针。可以看出，由于 `initproc` 的中断帧占用了实际给 `initproc` 分配的栈空间的顶部，所以 `initproc` 就只能把栈顶指针 `context.esp` 设置在 `initproc` 的中断帧的起始位置。根据 `context.eip` 的赋值，可以知道 `initproc` 实际开始执行的地方在 `forkret` 函数处。至此，`initproc` 内核线程已经做好准备执行了。

### 3. 调度并执行内核线程 `initproc`

在 `ucore` 执行完 `proc_init` 函数后，就创建好了两个内核线程：`idleproc` 和 `initproc`，这时 `ucore` 当前的执行现场就是 `idleproc`，等到执行到 `init` 函数的最后一个函数 `cpu_idle` 之前，`ucore` 的所有初始化工作就结束了，`idleproc` 将通过执行 `cpu_idle` 函数让出 CPU，给其它内核线程执行，具体过程如下：

```
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
        .....
    }
}
```

首先，判断当前内核线程 `idleproc` 的 `need_resched` 是否不为 0，回顾本节中“创建第一个内核线程 `idleproc`”中的描述，`proc_init` 函数在初始化 `idleproc` 中，就把 `idleproc->need_resched` 置为 1 了，所以会马上调用 `schedule` 函数找其他处于“就绪”态的进程执行。

`ucore` 在 `proc10` 中只实现了一个最简单的 FIFO 调度器，其核心就是 `schedule` 函数。它的执行逻辑很简单：

1. 设置当前内核线程 `current->need_resched` 为 0；

2. 在 `proc_list` 队列中查找下一个处于“就绪”态的线程或进程 `next`;
3. 找到这样的进程后,就调用 `proc_run` 函数,保存当前进程 `current` 的执行现场(进程上下文),恢复新进程的执行现场,完成进程切换。

至此,新的进程 `next` 就开始执行了。由于在 `proc10` 中只有两个内核线程,且 `idleproc` 要让出 CPU 给 `initproc` 执行,我们可以看到 `schedule` 函数通过查找 `proc_list` 进程队列,只能找到一个处于“就绪”态的 `initproc` 内核线程。并通过 `proc_run` 和进一步的 `switch_to` 函数完成两个执行现场的切换,具体流程如下:

1. 让 `current` 指向 `next` 内核线程 `initproc`;
2. 设置任务状态段 `ts` 的特权态 0 下的栈顶指针 `esp0` 为 `next` 内核线程 `initproc` 的内核栈的栈顶,即 `next->kstack + KSTACKSIZE` ;
3. 设置 `CR3` 寄存器的值为 `next` 内核线程 `initproc` 的页目录表起始地址 `next->cr3`,这实际上是完成进程间的页表切换;
4. 由 `switch_to` 函数完成具体的两个线程的执行现场切换,即切换各个寄存器,当 `switch_to` 函数执行完“`ret`”指令后,就切换到 `initproc` 执行了。

这里需要注意,在第二步设置任务状态段 `ts` 的特权态 0 下的栈顶指针 `esp0` 的目的是建立好内核线程或将来用户线程在执行特权态切换(从特权态 0<-->特权态 3,或从特权态 3<-->特权态 3)时能够正确定位处于特权态 0 时进程的内核栈的栈顶,而这个栈顶其实放了一个 `trapframe` 结构的内存空间。如果是在特权态 3 发生了中断/异常/系统调用,则 CPU 会从特权态 3->特权态 0,且 CPU 从此栈顶开始压栈来保存被中断/异常/系统调用打断的用户态执行现场;如果是在特权态 0 发生了中断/异常/系统调用,则 CPU 会从特权态还是 0,且 CPU 从当前栈指针 `esp` 所指的位置开始压栈保存被中断/异常/系统调用打断的内核态执行现场。反之,当执行完对中断/异常/系统调用打断的处理后,最后会执行一个“`iret`”指令。在执行此指令之前,CPU 的当前栈指针 `esp` 一定指向上次产生中断/异常/系统调用时 CPU 保存的被打断的指令地址 `CS` 和 `EIP`,”`iret`”指令会根据 `ESP` 所指的保存的址 `CS` 和 `EIP` 恢复到上次被打断的地方继续执行。

在页表设置方面,由于 `idleproc` 和 `initproc` 都是共用一个内核页表 `boot_cr3`,所以此时第三步其实没用,但考虑到以后的进程有各自的页表,其起始地址各不相同,只有完成页表切换,才能确保新的进程能够正常执行。

第四步 `proc_run` 函数调用 `switch_to` 函数,参数是前一个进程和后一个进程的执行现场 `context`。在上一节“设计进程控制块”中,描述了 `context` 结构包含的要保存和恢复的寄存器。我们在看看 `switch.S` 中的 `switch_to` 函数的执行流程:

```
.globl switch_to
switch_to:                                     # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax                         # eax points to from
    popl 0(%eax)                               # save eip !popl
    movl %esp, 4(%eax)
    .....
    movl %ebp, 28(%eax)
    # restore to's registers
    movl 4(%esp), %eax                         # not 8(%esp): popped return address already
                                                # eax now points to to
    movl 28(%eax), %ebp
```

```

.....
movl 4(%eax), %esp
pushl 0(%eax)          # push eip
ret

```

首先，保存前一个进程的执行现场，前两条汇编指令（如下所示）保存了进程在返回 `switch_to` 函数后的指令地址到 `context.eip` 中

```

movl 4(%esp), %eax      # eax points to from
popl 0(%eax)            # save eip !popl

```

在接下来的 7 条汇编指令完成了保存前一个进程的其他 7 个寄存器到 `context` 中的相应域中。至此前一个进程的执行现场保存完毕。接下来就是恢复向一个进程的执行现场，这其实就是上述保存过程的逆执行过程，即从 `context` 的高地址的域 `ebp` 开始，逐一把相关域的值赋值给对应的寄存器，倒数第二条汇编指令“`pushl 0(%eax)`”其实把 `context` 中保存的下一个进程要执行的指令地址 `context.eip` 放到了堆栈顶，这样接下来执行最后一条指令“`ret`”时，会把栈顶的内容赋值给 `EIP` 寄存器，这样就切换到下一个进程执行了，即当前进程已经是下一个进程了。

再回到 `proj10` 中，`ucore` 会执行进程切换，让 `initproc` 执行。在对 `initproc` 进行初始化时，设置了 `initproc->context.eip = (uintptr_t)forkret`，这样，当执行 `switch_to` 函数并返回后，`initproc` 将执行其实际上的执行入口地址 `forkret`。而 `forkret` 会调用位于 `kern/trap/trapentry.S` 中的 `forkrets` 函数执行，具体代码如下：

```

.globl __trapret
__trapret:
    # restore registers from stack
    popal

    # restore %ds and %es
    popl %es
    popl %ds

    # get rid of the trap number and error code
    addl $0x8, %esp
    iret

.globl forkrets
forkrets:
    # set stack to this new process's trapframe
    movl 4(%esp), %esp    //把 esp 指向当前进程的中断帧
    jmp __trapret

```

可以看出，`forkrets` 函数首先把 `esp` 指向当前进程的中断帧，从 `_trapret` 开始执行到 `iret` 前，`esp` 指向了 `current->tf.tf_eip`，而如果此时执行的是 `initproc`，则 `current->tf.tf_eip = kernel_thread_entry`，`initproc->tf.tf_cs = KERNEL_CS`，所以当执行完 `iret` 后，就开始在内核中执行 `kernel_thread_entry` 函数了，而 `initproc->tf.tf_regs.reg_ebx = init_main`，所以在 `kernel_thread_entry` 中执行“`call %ebx`”后，就开始执行 `initproc` 的主体了。`initproc` 的主体函数很简单就是输出一段字符串，然后就返回到 `kernel_thread_entry` 函数，并进一步调用

`do_exit` 执行退出操作了。本来 `do_exit` 应该完成一些资源回收工作等，但这些不是 `proj10` 涉及的，而是由后续的 `proj` 来完成。

至此，`proj10` 的主要工作描述完毕

## 4.2 创建并执行用户进程

### 4.2.1 实验目标

到 `proj10` 为止，`ucore` 还一直在核心态“打转”，没有到用户态执行。其实这也是对操作系统的要求，操作系统就要呆在核心态，才能管理整个计算机系统。但应用程序员也需要编写各种应用软件，且要在计算机系统上运行。如果把这些应用软件都作为内核线程来执行，那系统的安全性就无法得到保证了。操作系统的观点是：操作系统程序员编写的操作系统模块是可信的、高效的，对计算机的物理资源了如指掌，可在计算机的核心态执行；但应用程序员编写的应用软件是不可信的，不必了解计算机的物理资源，且需要操作系统提供服务，故放在用户态执行，无法轻易破坏操作系统和其他用户态的进程通过系统调用获得操作系统的服务。所以，`ucore` 要提供用户态进程的创建和执行机制，给应用程序执行提供一个用户态运行环境。

### 4.2.2 `proj10.1` 概述

#### 1. 实现描述

`proj10.1` 是 `lab3` 的第二个 `project`。它在 `proj10` 的基础上实现了对用户态进程的支持，主要扩展设计了用户进程执行的用户地址空间、对用户进程访存错误的异常处理、提供用户进程执行效率的按需分页和写时复制的支持、加载并执行依附在 `ucore` 内核文件的用户执行程序、实现系统调用机制等。

#### 2. 项目组成

```
proj10.1
├── .....
├── kern
│   ├── .....
│   ├── mm
│   │   ├── memlayout.h
│   │   ├── vmm.c
│   │   └── vmm.h
│   ├── process
│   │   ├── proc.c
│   │   ├── proc.h
│   │   └── .....
│   └── syscall
```

```

|   |   |—— syscall.c
|   |   |—— syscall.h
|   |   |—— trap
|   |   |—— trap.c
|   |   |—— .....
|   |   |—— user
|   |   |—— badsegment.c
|   |   |—— divzero.c
|   |   |—— faultread.c
|   |   |—— faultreadkernel.c
|   |   |—— hello.c
|   |   |—— libs
|   |   |—— initcode.S
|   |   |—— panic.c
|   |   |—— stdio.c
|   |   |—— syscall.c
|   |   |—— syscall.h
|   |   |—— ulib.c
|   |   |—— ulib.h
|   |   |—— umain.c
|   |   |—— pgdir.c
|   |   |—— softint.c
|   |   |—— testbss.c
|   |   |—— yield.c

```

17 directories, 97 files

相对于 proj10, proj10.1 主要增加了有关系统调用实现的 `syscall.[ch]` 和测试 `ucore` 对用户进程支持的各种用户态程序和库文件, 并对相关的内核文件进行了修改。主要修改和增加的文件如下:

- `mm/memlayout.h`: 定义了用户态空间的范围, 具体可看“Virtual memory map”的 ASCII 图注释。
- `mm/vmm.[ch]`: 在 `vmm.h` 文件中, 扩展了 `mm_struct` 数据结构, 支持多进程对 `mm_struct` 的引用计数和互斥访问, 和针对 `mm_struct` 结构的引用计数操作和互斥操作; 在 `vmm.c` 中, 主要增加了部分函数防止多进程 (比如父子进程、多线程等) 同时访问进程的 `mm` 进程内存管理数据结构。
- `process/proc.[ch]`: 对一系列进程管理相关函数进行了扩展, 并新实现了部分函数。这是内核改动最大的部分。
- `syscall/syscall.[ch]`: 新增加的部分, 提供用户态进程所需的系统服务的操作系统层接口, 根据系统调用号, 在转到具体的服务功能函数中完成用户态进程的服务请求。
- `trap/trap.c`: 增加对系统调用的初始化和处理, 扩展对访存错误异常的处理。
- `user/*`: 实现应用程序所需的基本库函数支持, 提供实现系统调用的用户层接口。



### 3. 编译运行

编译并运行 proj10.1 的命令如下：

```
make
```

```
make qemu
```

则可以得到如下显示界面

```
(THU.CST) os is loading ...
```

```
Special kernel symbols:
```

```
entry 0xc010002c (phys)
```

```
etext 0xc0114ba3 (phys)
```

```
edata 0xc018656f (phys)
```

```
end 0xc018b934 (phys)
```

```
Kernel executable memory footprint: 559KB
```

```
memory management: buddy_pmm_manager
```

```
.....
```

```
check_mm_shm_swap() succeeded.
```

```
++ setup timer interrupts
```

```
kernel_execve: pid = 1, name = "hello".
```

```
Hello world!!.
```

```
I am process 1.
```

```
hello pass.
```

```
kernel panic at kern/process/proc.c:379:
```

```
initproc exit.
```

```
Welcome to the kernel debug monitor!!
```

```
Type 'help' for a list of commands.
```

```
K>
```

上述执行输出相对于 proj10 没有太多变化，只是出现了“kernel\_execve: ... hello pass”等字符串。但其实在其背后，涉及创建用户态进程，把执行代码加载到用户态线程地址空间，执行系统调用等一系列操作。为了更好地理解 proj10.1 的设计和实现方案，我们先需要在了解一下用户态进程的特征。下面我们将从原理和实现两个方面对此进行进一步阐述。

#### 4.2.3 【原理】用户进程的特征

##### 1. 从内核线程到用户进程

在 proj10 中设计实现了进程控制块，并实现了内核线程的创建和简单的调度执行。但 proj10 中没有在用户态执行用户进程的管理机制，既无法体现用户进程的地址空间，以及用户进程间地址空间隔离的保护机制，不支持进程执行过程的用户态和核心态之间的切换，且没有用户进程的完整状态变化的生命周期。其实没有实现的原因是内核线程不需要这些功能。

---

那内核线程相对于用户态线程有何特点呢？

但其实我们已经在 `proj10` 中看到了内核线程，内核线程的管理实现相对是简单的，其特点是直接使用操作系统（比如 `ucore`）在初始化中建立的内核虚拟内存地址空间，不同的内核线程之间可以通过调度器实现线程间的切换，达到分时使用 `CPU` 的目的。由于内核虚拟内存空间是一一映射计算机系统的物理空间的，这使得可用空间的大小不会超过物理空间大小，所以操作系统程序员编写内核线程时，需要考虑到有限的地址空间，需要保证各个内核线程在执行过程中不会破坏操作系统的正常运行。这样在实现内核线程管理时，不必考虑涉及与进程相关的虚拟内存管理中的缺页处理、按需分页、写时复制、页换入换出等功能。如果在内核线程执行过程中出现了访存错误异常或内存不够的情况，就认为操作系统出现错误了，操作系统将直接宕机。在 `ucore` 中，就是调用 `panic` 函数，进入内核调试监控器 `kernel_debug_monitor`。

内核线程管理思想相对简单，但编写内核线程对程序员的要求很高。从理论上讲（理想情况），如果程序员都是能够编写操作系统级别的“高手”，能够勤俭和高效地使用计算机系统资源，且这些“高手”都为他人着想，具有奉献精神，在别的应用需要计算机资源的时候，能够从大局出发，从整个系统的执行效率出发，让出自己占用的资源，那这些“高手”编写出来的程序直接作为内核线程运行即可，也就没有用户进程存在的必要了。

但现实与理论的差距是巨大的，能编写操作系统的程序员是极少数的，与当前的应用程序员相比，估计大约差了 3~4 个数量级。如果还要求编写操作系统的程序员考虑其他未知程序员的未知需求，那这样的程序员估计可以成为是编程界的“上帝”了。

从应用程序编写和运行的角度看，既然程序员都不是“上帝”，操作系统程序员就需要给应用程序员编写的程序提供一个既“宽松”又“严格”的执行环境，让对内存大小和 `CPU` 使用时间等资源的限制没有仔细考虑的应用程序都能在操作系统中正常运行，且即使程序太可靠，也只能破坏自己，而不能破坏其他运行程序和整个系统。“严格”就是安全性保证，即应用程序执行不会破坏在内存中存在的其他应用程序和操作系统的内存空间等独占的资源；“宽松”就算是方便性支持，即提供给应用程序尽量丰富的服务功能和一个远大于物理内存空间的虚拟地址空间，使得应用程序在执行过程中不必考虑很多繁琐的细节（比如如何初始化 `PCI` 总线和外设等，如果管理物理内存等）。

## 2. 让用户进程正常运行的用户环境

在操作系统原理的介绍中，一般提到进程的概念其实主要是指用户进程。从操作系统的设计和实现的角度看，其实用户进程是指一个应用程序在操作系统提供的一个用户环境中的一次执行过程。这里的重点是用户环境。用户环境有啥功能？用户环境指的是什么？

从功能上看，操作系统提供的这个用户环境有两方面的特点。一方面与存储空间相关，即限制用户进程可以访问的物理地址空间，且让各个用户进程之间的物理内存空间访问不重叠，这样可以保证不同用户进程之间不能相互破坏各自的内存空间，利用虚拟内存的功能（页换入换出）。给用户进程提供了远大于实际物理内存空间的虚拟内存空间。

另一方面与执行指令相关，即限制用户进程可执行的指令，不能让用户进程执行特权指令（比如修改页表起始地址），从而保证用户进程无法破坏系统。但如果不能执行特权指令，则很多功能（比如访问磁盘等）无法实现，所以需要某种机制，让操作系统完成需要特权指令才能做的各种服务功能，给用户进程一个“服务窗口”，用户进程可以通过这个“窗口”向操作系统提出服务请求，由操作系统来帮助用户进程完成需要特权指令才能做的各种服务。另外，还要有一个“中断窗口”，让用户进程不主动放弃使用 `CPU` 时，操作系统

能够通过这个“中断窗口”强制让用户进程放弃使用 CPU，从而让其他用户进程有机会执行。

基于功能分析，我们就可以把这个用户环境定义为如下组成部分：

- 建立用户虚拟空间的页表和支持页换入换出机制的用户内存访问错误异常服务例程：提供地址隔离和超过物理空间大小的虚存空间。
- 应用程序执行的用户态 CPU 特权级：在用户态 CPU 特权级，应用程序只能执行一般指令，如果特权指令，结果不是无效就是产生“执行非法指令”异常；
- 系统调用机制：给用户进程提供“服务窗口”；
- 中断响应机制：给用户进程设置“中断窗口”，这样产生中断后，当前执行的用户进程将被强制打断，CPU 控制权将被操作系统的中断服务例程使用。

### 3. 用户态进程的执行过程分析

在这个环境下运行的进程就是用户进程。那如果用户进程由于某种原因下面进入内核态后，那在内核态执行的是什么呢？还是用户进程吗？首先分析一下用户进程这样会进入内核态呢？回顾一下 lab1，就可以知道当产生外设中断、CPU 执行异常（比如访存错误）、陷入（系统调用），用户进程就会切换到内核中的操作系统中来。表面上看，到内核态后，操作系统取得了 CPU 控制权，所以现在执行的应该是操作系统代码，由于此时 CPU 处于核心态特权级，所以操作系统的执行过程就应该是内核进程了。这样理解忽略了操作系统的实现。如果考虑操作系统的实现，应该如果来理解进程呢？

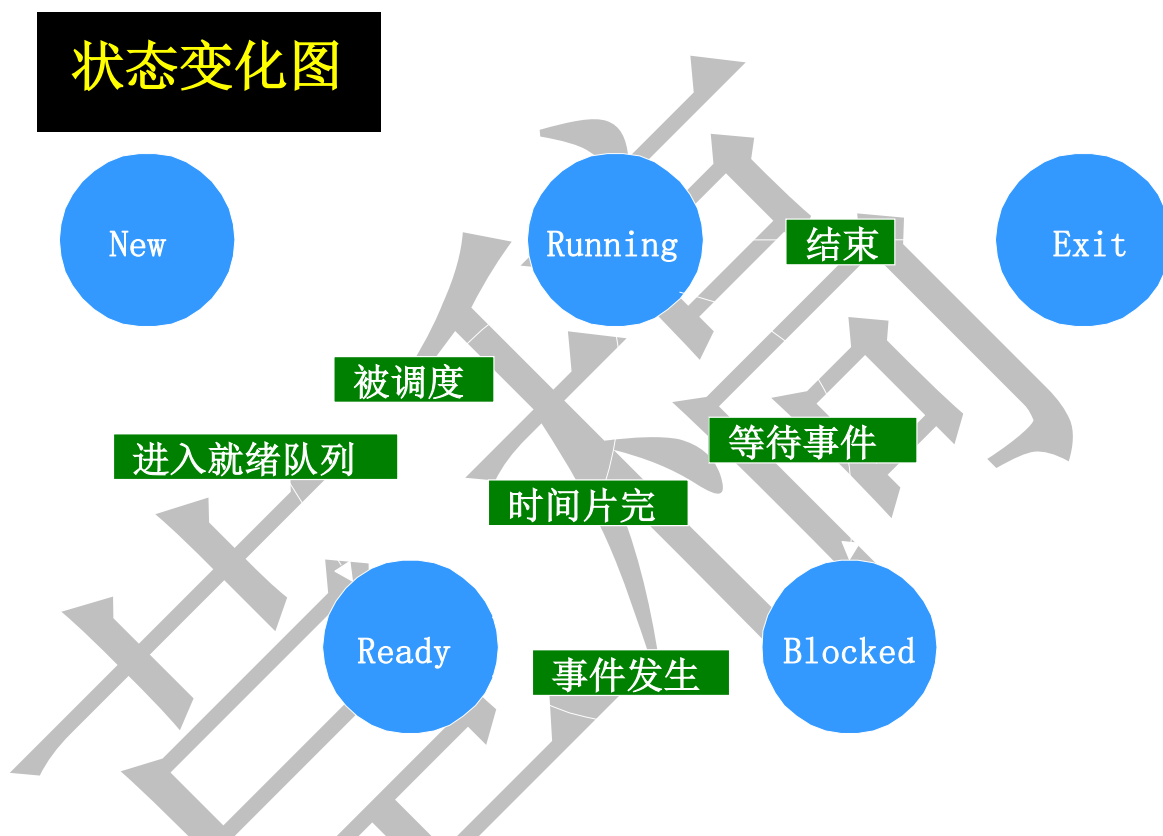
从进程控制块的角度看，如果执行了进程执行现场（上下文）的切换，就认为到另外一个进程执行了，及进程的分界点设定在执行进程切换的前后。到底切换了什么呢？其实只是切换了进程的页表和相关硬件寄存器，这些信息都保存在进程控制块中的相关域中。所以，我们可以把执行应用程序的代码一直到执行操作系统中的进程切换处为止都认为是一个应用程序的执行过程（其中有操作系统的部分代码执行过过程）即进程。因为在这个过程中，没有更换到另外一个进程控制块的进程的页表和相关硬件寄存器。

从指令执行的角度看，如果再仔细分析一下操作系统这个软件的特点并细化一下进入内核原因，就可以看出进一步进行划分。操作系统的主要功能是给上层应用提供服务，管理整个计算机系统资源。所以操作系统虽然是一个软件，但其实是一个基于事件的软件，这里操作系统需要响应的事件包括三类：外设中断、CPU 执行异常（比如访存错误）、陷入（系统调用）。如果用户进程通过系统调用要求操作系统提供服务，那么从用户进程的角度看，操作系统就是一个特殊的软件库（比如相对于用户态的 libc 库，操作系统可看作是内核态的 libc 库），完成用户进程的需求，从执行逻辑上看，是用户进程“主观”执行的一部分，即用户进程“知道”操作系统要做的事情。那么在这种情况下，进程的代码空间包括用户态的执行程序和内核态响应用户进程通过系统调用而在核心特权态执行服务请求的操作系统代码，为此这种情况下的进程的内存虚拟空间也包括两部分：用户态的虚地址空间和核心态的虚地址空间。但如果此时发生的事件是外设中断和 CPU 执行异常，虽然 CPU 控制权也转入到操作系统中的中断服务例程，但这些内核执行代码执行过程是用户进程“不知道”的，是另外一段执行逻辑。那么在这种情况下，实际上是执行了两段目标不同的执行程序，一个是代表应用程序的用户进程，一个是代表中断服务例程处理外设中断和 CPU 执行异常的内核线程。这个用户进程和内核线程在产生中断或异常的时候，CPU 硬件就完成了它们之间的指令流

切换。

## 4. 用户进程的运行状态分析

用户进程在其执行过程中会存在很多种不同的执行状态，根据操作系统原理，一个用户进程一般的运行状态有五种：创建（new）态、就绪（ready）态、运行（running）态、等待（blocked）态、退出（exit）态。各个状态之间会由于发生了某事件而进行状态转换。进程的状态转换图如下所示：



但在用户进程的执行过程中，具体在哪个时间段是出于上述状态的呢？上述状态是如何转变的呢？首先，我们看创建（new）态，操作系统完成进程的创建工作，而体现进程存在的就是进程控制块，所以一旦操作系统创建了进程控制块，则可以认为此时进程就已经存在了，但由于进程能够运行的各种资源还没准备好，所以此时的进程处于创建（new）态。创建了进程控制块后，进程并不能就执行了，还需准备好各种资源，如果把进程执行所需要的虚拟内存空间，执行代码，要处理的数据等都准备好了，则此时进程已经可以执行了，但还没有被操作系统调度，需要等待操作系统选择这个进程执行，于是把这个做好“执行准备”的进程放入到一个队列中，并可以认为此时进程处于就绪（ready）态。当操作系统的调度器从就绪进程队列中选择一个就绪进程后，通过执行进程切换，就让这个被选上的就绪进程执行了，此时进程就处于运行（running）态了。到了运行态后，会出现三种事件。如果进程需要等待某个事件（比如主动睡眠 10 秒钟，或进程访问某个内存空间，但此内存空间被换出到硬盘 swap 分区中了，进程不得不等待操作系统把缓慢的硬盘上的数据重新读回到内存中），那么操作系统会把 CPU 给其他进程执行，并把进程状态从运行（running）态转换为等待（blocked）态。如果用户进程

的应用程序逻辑流程执行结束了，那么操作系统会把 CPU 给其他进程执行，并把进程状态从运行（running）态转换为退出（exit）态，并准备回收用户进程占用的各种资源，当把表示整个进程存在的进程控制块也回收了，这进程就不存在了。在这整个回收过程中，进程都处于退出（exit）态。2 考虑到在内存中存在多个处于就绪态的用户进程，但只有一个 CPU，所以为了公平起见，每个就绪态进程都只有有限的时间片段，当一个运行态的进程用完了它的时间片段后，操作系统会剥夺此进程的 CPU 使用权，并把此进程状态从运行（running）态转换为就绪（ready）态，最后把 CPU 给其他进程执行。如果某个处于等待（blocked）态的进程所等待的事件产生了（比如睡眠时间到，或需要访问的数据已经从硬盘换入到内存中），则操作系统会通过把等待此事件的进程状态从等待（blocked）态转到就绪（ready）态。这样进程的整个状态转换形成了一个有限状态自动机。

有了上述对用户进程的特征分析后，接下来我们就通过跟踪用户进程的整个生命周期来阐述用户进程管理的设计与实现。

#### 4.2.4 创建用户进程

在 proj10 中，我们已经完成了对内核线程的创建，但与用户进程的创建过程相比，创建内核线程的过程还远远不够。而这两个创建过程的差异本质上就是用户进程和内核线程的差异决定的。

### 1. 应用程序的组成和编译

首先，我们要有一个应用程序，这里我们假定是 hello 应用程序，在 user/hello.c 中实现，代码如下：

```
#include <stdio.h>
#include <ulib.h>

int
main(void) {
    printf("Hello world!!.\n");
    printf("I am process %d.\n", getpid());
    printf("hello pass.\n");
    return 0;
}
```

hello 应用程序只是输出一些字符串，并通过系统调用 sys\_getpid(在 getpid 函数中调用) 输出代表 hello 应用程序执行的用户进程的进程标识--pid。

首先，我们需要了解 ucore 操作系统如何能够找到 hello 应用程序。这需要分析 ucore 和 hello 是如何编译的。修改 Makefile，把第六行注释掉。然后在 proj10.1 目录下执行 make，可得到如下输出：

```
.....
+ cc user/hello.c

gcc -Iuser/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/
```

```
-Iuser/include/ -Iuser/libs/ -c user/hello.c -o obj/user/hello.o
```

```
ld -m elf_i386 -nostdlib -T tools/user.ld -o obj/__user_hello.out obj/user/libs/initcode.o  
obj/user/libs/panic.o obj/user/libs/stdio.o obj/user/libs/syscall.o obj/user/libs/ulib.o  
obj/user/libs/umain.o obj/libs/hash.o obj/libs/printfmt.o obj/libs/rand.o obj/libs/string.o  
obj/user/hello.o
```

.....

```
ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/entry.o  
obj/kern/init/init.o ..... -b binary ..... obj/__user_hello.out
```

.....

从中可以看出，hello 应用程序不仅仅是 hello.c，还包含了支持 hello 应用程序的用户态库：

- user/libs/initcode.S: 所有应用程序的起始用户态执行地址 “\_start”，调整了 EBP 和 ESP 后，调用 umain 函数。
- user/libs/umain.c: 实现了 umain 函数，这是所有应用程序执行的第一个 C 函数，它将调用应用程序的 main 函数，并在 main 函数结束后调用 exit 函数，而 exit 函数最终将调用 sys\_exit 系统调用，让操作系统回收进程资源。
- user/libs/ulib.[ch]: 实现了最小的 C 函数库，除了一些与系统调用无关的函数，其他函数是对访问系统调用的包装。
- user/libs/syscall.[ch]: 用户层发出系统调用的具体实现。
- user/libs/stdio.c: 实现 cprintf 函数，通过系统调用 sys\_putc 来完成字符输出。
- user/libs/panic.c: 实现 \_\_panic/ \_\_warn 函数，通过系统调用 sys\_exit 完成用户进程退出。

除了这些用户态库函数实现外，还有一些 libs/\*.ch 是操作系统内核和应用程序共用的函数实现。这些用户库函数其实在本质上与 UNIX 系统中的标准 libc 没有区别，只是实现得很简单，但 hello 应用程序的正确执行离不开这些库函数。

【注意】libs/\*.ch、user/libs/\*.ch、user/\*.ch 的源码中没有任何特权指令。

在 make 的最后一步执行了一个 ld 命令，把 hello 应用程序的执行码 obj/\_\_user\_hello.out 连接在了 ucore kernel 的末尾。且 ld 命令会在 kernel 中会把 \_\_user\_hello.out 的位置和大小记录在全局变量 \_binary\_obj\_\_user\_hello\_out\_start 和 \_binary\_obj\_\_user\_hello\_out\_size 中，这样这个 hello 用户程序就能够和 ucore 内核一起被 boot loader 加载到内存里中，并且通过这两个全局变量定位 hello 用户程序执行码的起始位置和大小。而到了 lab6 的实验后，ucore 会提供一个简单的文件系统，那时所有的用户程序就都不再用这种方法进行加载了。

## 2. 用户进程的虚拟地址空间

在 tools/user.ld 描述了用户程序的用户虚拟空间的执行入口虚拟地址：

```
SECTIONS {
```

```
    /* Load programs at this address: "." means the current address */
```

```
    . = 0x800020;
```

在 tools/kernel.ld 描述了操作系统的内核虚拟空间的起始入口虚拟地址：

```
SECTIONS {
```

```
    /* Load the kernel at this address: "." means the current address */
```

. = 0xC0100000;

这样 ucore 把用户进程的虚拟地址空间分了两块，一块与内核线程一样，是所有用户进程都共享的内核虚拟地址空间，映射到同样的物理内存空间中，这样在物理内存中只需放置一份内核代码，使得用户进程从用户态进入核心态时，内核代码可以统一应对不同的内核程序；另外一块是用户虚拟地址空间，虽然虚拟地址范围一样，但映射到不同且没有交集的物理内存空间中。这样当 ucore 把用户进程的执行代码（即应用程序的执行代码）和数据（即应用程序的全局变量等）放到用户虚拟地址空间中时，确保了各个进程不会“非法”访问到其他进程的物理内存空间。

这样 ucore 给一个用户进程具体设定的虚拟内存空间（kern/mm/memlayout.h）如下所示：

| Virtual memory map:    |                            | Permissions    |
|------------------------|----------------------------|----------------|
|                        |                            | kernel/user    |
| 4G ----->              | Empty Memory (*)           |                |
|                        | Cur. Page Table (Kern, RW) | RW/-- PTSIZE   |
| VPT ----->             | Invalid Memory (*)         | --/--          |
| KERNTOP ----->         | Remapped Physical Memory   | RW/-- KMEMSIZE |
| KERNBASE ----->        | Invalid Memory (*)         | --/--          |
| USERTOP ----->         | User stack                 | 0xB0000000     |
|                        | :                          | :              |
|                        | ~~~~~                      | :              |
|                        | :                          | :              |
|                        | User Program & Heap        | 0x00800000     |
| UTEXT ----->           | Invalid Memory (*)         | --/--          |
|                        | User STAB Data (optional)  |                |
| USERBASE, USTAB -----> | Invalid Memory (*)         | --/--          |
| 0 ----->               |                            | 0x00000000     |

画一幅图，表示用户进程的虚拟地址空间和物理地址空间的映射关系和空间布局

### 3. 创建并执行用户进程

在确定了用户进程的执行代码和数据，以及用户进程的虚拟空间布局后，我们可以来创建用户进程了。在 proj10.1 中第一个用户进程是由第二个内核线程 initproc 通过把 hello 应用程序执行码覆盖到 initproc 的用户虚拟内存空间来创建的，相关代码如下所示：

```
// kernel_execve - do SYS_exec syscall to exec a user program called
by user_main kernel_thread

static int
kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int ret, len = strlen(name);
    asm volatile (
```

```

        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL), "0" (SYS_exec), "d" (name), "c" (len), "b" (binary), "D" (size)
        : "memory");
    return ret;
}

#define __KERNEL_EXECVE(name, binary, size) ({
    cprintf("kernel_execve: pid = %d, name = \"%s\".\n",
        current->pid, name);
    kernel_execve(name, binary, (size_t)(size));
})

#define KERNEL_EXECVE(x) ({
    extern unsigned char _binary_obj__user_##x##_out_start[], \
        _binary_obj__user_##x##_out_size[];
    __KERNEL_EXECVE(#x, _binary_obj__user_##x##_out_start, \
        _binary_obj__user_##x##_out_size);
})

.....
// init_main - the second kernel thread used to create kswapd_main & user
_main kernel threads
static int
init_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(hello);
#endif
    panic("kernel_execve failed.\n");
    return 0;
}

```

对于上述代码，我们需要从后向前按照函数/宏的实现一个一个来分析。Initproc 的执行主体是 `init_main` 函数，这个函数在缺省情况下是执行宏 `KERNEL_EXECVE(hello)`，而这个宏最终是调用 `kernel_execve` 函数来调用 `SYS_exec` 系统调用，由于 `ld` 在链接 `hello` 应用程序执行码时定义了两全局变量：

- `_binary_obj__user_hello_out_start`: `hello` 执行码的起始位置
- `_binary_obj__user_hello_out_size` 中: `hello` 执行码的大小

`kernel_execve` 把这两个变量作为 `SYS_exec` 系统调用的参数，让 `ucore` 来创建此用户进程。当 `ucore` 收到此系统调用后，将依次调用如下函数

```

vector128(vectors.S)--> __alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)--
-->syscall(syscall.c)-->sys_exec (syscall.c) -->do_execve(proc.c)

```

最终通过 `do_execve` 函数来完成用户进程的创建工作。此函数的主要工作流程如下：



- 首先为加载新的执行码做好用户态内存空间清空准备。如果 `mm` 不为 `NULL`，则设置页表为内核空间页表，且进一步判断 `mm` 的引用计数减 1 后是否为 0，如果为 0，则表明没有进程再需要此进程所占用的内存空间，为此将根据 `mm` 中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的 `mm` 内存管理指针为空。由于此处的 `initproc` 是内核线程，所以 `mm` 为 `NULL`，整个处理都不会做。
- 接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。这里涉及到读 ELF 格式的文件，申请内存空间，建立用户态虚存空间，加载应用程序执行码等。`load_icode` 函数完成了整个复杂的工作。

`load_icode` 函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。此函数有一百多行，完成了如下重要工作：

1. 调用 `mm_create` 函数来申请进程的内存管理数据结构 `mm` 所需内存空间，并对 `mm` 进行初始化；（`mm_struct` 的介绍载第三章 3.3.5 小节）
2. 调用 `setup_pgdir` 来申请一个页目录表所需的一个页大小的内存空间，并把描述 `ucore` 内核虚空间映射的内核页表（`boot_pgdir` 所指）的内容拷贝到此新目录表中，最后让 `mm->pgdir` 指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间；
3. 根据应用程序执行码的起始位置来解析此 ELF 格式的执行程序，并调用 `mm_map` 函数根据 ELF 格式的执行程序说明的各个段（代码段、数据段、BSS 段等）的起始位置和大小建立对应的 `vma` 结构，并把 `vma` 插入到 `mm` 结构中，从而表明了用户进程的合法用户态虚拟地址空间；（`vma_struct` 的介绍载第三章 3.3.5 小节）
4. 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；
5. 需要给用户进程设置用户栈，为此调用 `mm_mmap` 函数建立用户栈的 `vma` 结构，明确用户栈的位置在用户虚空间的顶端，大小为 256 个页，即 1MB，但要注意，并没有给用户栈分配实际的物理内存；
6. 至此，进程内的内存管理 `vma` 和 `mm` 数据结构已经建立完成，于是把 `mm->pgdir` 赋值到 `cr3` 寄存器中，即更新了用户进程的虚拟内存空间，此时的 `initproc` 已经被 `hello` 的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；
7. 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“`iret`”后，能够让 CPU 转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断；

```
tf->tf_cs = USER_CS;
tf->tf_ds = USER_DS;
tf->tf_es = USER_DS;
tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;
tf->tf_eip = elf->e_entry;
tf->tf_eflags = FL_IF;
```

至此，用户进程的用户环境已经搭建完毕。此时 `initproc` 将按产生系统调用的函数调用路径原路返回，执行中断返回指令“`iret`”（位于 `trapentry.S` 的最后一句）后，将切换到用户进程 `hello` 的第一条语句位置 `_start` 处（位于 `user/libs/initcode.S` 的第三句）开始执行。

## 4.2.5 基于时间事件的等待与唤醒

Clock（时钟）中断(irq0，可回顾第二章 2.4 节) 可给操作系统提供有一定间隔的时间事件，操作系统将其作为基本的计时单位，这里把两次时钟中断之间的时间间隔为一个时间片（timer splice）。基于此时间片，操作系统得以向上提供基于时间点的事件，并实现基于固定时间长度的等待和唤醒机制。在每个时钟中断发生时，操作系统可产生对应时间长度的时间事件，这样操作系统和应用程序可基于这些时间事件来构建基于时间的软件设计。在 proj10.4 中，实现了定时器 timer 的支持。

### 1. timer 数据结构

sched.h 定义了有关 timer 数据结构，

```
typedef struct {
    unsigned int expires;           //到期时间
    struct proc_struct *proc;      //等待时间到期的进程
    list_entry_t timer_link;       //链接到 timer_list 的链表项指针
} timer_t;
```

这几个成员变量描述了一个 timer（定时器）涉及的相关因素，首先一个 expires 表明了这个定时器何时到期，而第二个成员变量描述了定时器到期后需要唤醒的进程，最后一个参数是一个链表项，用于把自身挂到系统的 timer 链表上，以便于扫描查找特定的 timer。

### 2. timer 相关操作

一个 timer 在 ucore 中的生存周期可以被描述如下：

1. 某进程创建和初始化 timer\_t 结构的一个 timer，并把 timer 被加入系统 timer 管理列表 timer\_list 中，进程设置为基于 timer 事件的阻塞态（即睡眠了），这样这个 timer 就诞生了；
2. 系统时间通过时钟中断被不断累加，且 ucore 定期检查是否有某个 timer 的到期时间已经到了，如果没有到期，则 ucore 等待下一次检查，此 timer 会挂在 timer\_list 上继续存在；
3. 如果到期了，则对应的进程又处于就绪态了，并从系统 timer 管理列表 timer\_list 中移除该 timer，自此 timer 就死亡退出了。

基于上述 timer 生存周期的流程，与 timer 相关的函数如下：

- timer init: 对 timer 的成员变量进行初始化，设定了在 expires 时间之后唤醒 proc 进程
- add\_timer: 向系统 timer 链表 timer\_list 添加某个初始化过的 timer，这样该 timer 计时器将在指定时间 expires 后被扫描到，如果等待则这个定时器 timer 的进程处在等待状态，并将进程唤醒，进程将处于就绪态。
- del\_timer: 向系统 timer 链表 timer\_list 删除（或者说取消）某一个计时器。该计时器在取消后，对应的进程不会被系统在指定时刻 expires 唤醒。
- run\_timer\_list: 被 trap 函数调用，遍历系统 timer 链表 timer\_list 中的 timer 计时器，找出所有应该到时的 timer 计时器，并唤醒与此计时器相关的等待进程，再删除此

timer 计时器。在 lab4/proj13 以后，还增加了对进程调度器在时间事件产生后的处理函数的调用（在后续有进一步分析）。

有了这些函数的支持，我们就可以实现进程睡觉并被定时唤醒的功能了。比如 ucore 在用户函数库中提供了 sleep 函数，当用户进程调用 sleep 函数后，会进一步调用 sys\_sleep 系统调用，在内核中完成 sys\_sleep 系统调用服务的是 do\_sleep 内核函数，其实现如下：

```
int
do_sleep(unsigned int time) {
.....
    timer_t __timer, *timer = timer_init(&__timer, current, time);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_TIMER;
    add_timer(timer);
.....
    schedule();
    del_timer(timer);
    return 0;
}
```

可以看出，do\_sleep 首先初始化了一个定时器 timer，设置了 timer 的 proc 是当前进程，到期时间 expires 是参数 time；然后把当前进程的状态设置为等待状态，且等待原因是等某个定时器到期；再调用 schedule 完成进程调度与切换，这时当前进程已经不占用 CPU 执行了。当定时器到期后，run\_timer\_list 会删除 timer 且唤醒 timer 对应的当前进程，从而使得当前进程可以继续执行。

## 4.2.6 进程退出和等待进程

当进程执行完它的工作后，就需要执行退出操作，释放进程占用的资源。Ucore 分了两步来完成这个工作，首先由进程本身完成大部分资源的占用内存回收工作，然后由此进程的父进程完成剩余资源占用内存的回收工作。为何不让进程本身完成所有的资源回收工作呢？这是因为进程要执行回收操作，就表明此进程还存在，还在执行指令，这就需要内核栈的空间不能释放，且表示进程存在的进程控制块不能释放。所以需要父进程来帮忙释放子进程无法完成的这两个资源回收工作。

为此在用户态的函数库中提供了 exit 函数，此函数最终访问 sys\_exit 系统调用接口让操作系统来帮助当前进程执行退出过程中的部分资源回收。我们来看看 ucore 是如何做进程退出工作的。需要注意，这部分实现在 proj10.2 中才完成。所以我们这里是基于 proj10.2 的代码来进行分析。

首先，exit 函数会把一个退出码 error\_code 传递给 ucore，ucore 通过执行内核函数 do\_exit 来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作，具体流程如下：

1. 如果 current->mm != NULL，表示是用户进程，则开始回收此用户进程所占用的用户态虚拟内存空间；
  - a) 首先执行“lcr3(boot\_cr3)”，切换到内核态的页表上，这样当前用户进程目前只能在内核虚拟地址空间执行了，这是为了确保后续释放用户态内存和进程页表的工作能够正常执行；

- b) 如果当前进程控制块的成员变量 `mm` 的成员变量 `mm_count` 减 1 后为 0 (表明这个 `mm` 没有再被其他进程共享, 可以彻底释放进程所占的用户虚拟空间了。), 则开始回收用户进程所占的内存资源:
  - i. 调用 `exit_mmap` 函数释放 `current->mm->vma` 链表中每个 `vma` 描述的进程合法空间中实际分配的内存, 然后把对应的页表项内容清空, 最后还把页表所占用的空间释放并把对应的页目录表项清空;
  - ii. 调用 `put_pgdir` 函数释放当前进程的页目录所占的内存;
  - iii. 调用 `mm_destroy` 函数释放 `mm` 中的 `vma` 所占内存, 最后释放 `mm` 所占内存;
- c) 此时设置 `current->mm` 为 `NULL`, 表示与当前进程相关的用户虚拟内存空间和对应的内存管理成员变量所占的内核虚拟内存空间已经回收完毕;
2. 这时, 设置当前进程的执行状态 `current->state=PROC_ZOMBIE`, 当前进程的退出码 `current->exit_code=error_code`。此时当前进程已经不能被调度的了, 需要此进程的父进程来做最后的回收工作 (即回收描述此进程的内核栈和进程控制块);
3. 如果当前进程的父进程 `current->parent` 处于等待子进程状态 (即 `current->parent->wait_state==WT_CHILD`), 则唤醒父进程 (即执行 “`wakup_proc(current->parent)`”), 让父进程帮助自己完成最后的资源回收工作;
4. 如果当前进程还有子进程, 则需要把这些子进程的父进程指针设置为内核线程 `initproc`, 且各个子进程指针需要插入到 `initproc` 的子进程链表中。如果某个子进程的执行状态是 `PROC_ZOMBIE`, 则需要唤醒 `initproc` 来完成对此子进程的最后回收工作。
5. 执行 `schedule()` 函数, 选择新的进程执行。

那么父进程如何完成对子进程的最后回收工作呢? 这要求父进程要执行 `wait` 用户函数或 `wait_pid` 用户函数, 这两个函数的区别是, `wait` 函数等待任意子进程的结束通知, 而 `wait_pid` 函数等待进程 `id` 号为 `pid` 的子进程结束通知。这两个函数最终访问 `sys_wait` 系统调用接口让 `ucore` 来完成对子进程的最后回收工作, 即回收子进程的内核栈和进程控制块所占内存空间, 具体流程如下:

2. 如果 `pid!=0`, 表示只找一个进程 `id` 号为 `pid` 的退出状态的子进程, 否则找任意一个处于退出状态的子进程;
3. 如果此子进程的执行状态不为 `PROC_ZOMBIE`, 表明此子进程还没有退出, 则当前进程只好设置自己的执行状态为 `PROC_SLEEPING`, 睡眠原因为 `WT_CHILD` (即等待子进程退出), 调用 `schedule()` 函数选择新的进程执行, 自己睡眠等待, 如果被唤醒, 则重复跳回步骤 1 处执行;
4. 如果此子进程的执行状态为 `PROC_ZOMBIE`, 表明此子进程处于退出状态, 需要当前进程 (即子进程的父进程) 完成对子进程的最终回收工作, 即首先把子进程控制块从两个进程队列 `proc_list` 和 `hash_list` 中删除, 并释放子进程的内核堆栈和进程控制块。自此, 子进程才彻底地结束了它的执行过程, 消除了它所占用的所有资源。

【问题】哪些资源是子进程无法回收, 需要父进程帮忙回收的?

【问题】当子进程执行了 `sys_exit` 系统调用, 但父进程还没有执行 `sys_wait` 系统调用的情况下, 子进程还能正常通知父进程让父进程回收子进程最后无法回收的子进程所占资源吗?

## 4.2.7 系统调用实现

系统调用的英文名字是 System Call。操作系统为什么需要实现系统调用呢？其实这是实现了用户进程后，自然引申出来需要实现的操作系统功能。用户进程只能在操作系统给它圈定好的“用户环境”中执行，但“用户环境”限制了用户进程能够执行的指令，即用户进程只能执行一般的指令，无法执行特权指令。如果用户进程想执行一些需要特权指令的任务，比如通过网卡发网络包等，只能让操作系统来代劳了。于是就需要一种机制来确保用户进程不能执行特权指令，但能够请操作系统“帮忙”完成需要特权指令的任务，这种机制就是系统调用。

采用系统调用机制为用户进程提供一个获得操作系统服务的统一接口层，这样一来可简化用户进程的实现，把一些共性的、繁琐的、与硬件相关、与特权指令相关的任务放到操作系统层来实现，但提供一个简洁的接口给用户进程调用；二来这层接口事先可规定好，且严格检查用户进程传递进来的参数和操作系统要返回的数据，使得让操作系统给用户进程服务的同时，保护操作系统不会被用户进程破坏。

从硬件层面上看，需要硬件能够支持在用户态的用户进程通过某种机制切换到内核态。在第二章的 2.4 节和 2.5 节讲述中断硬件支持和软件处理过程其实就可以用来完成系统调用所需的软硬件支持。下面我们来看看如何在 ucore 中实现系统调用。

### 1. 初始化系统调用对应的中断描述符

在 ucore 初始化函数 kern\_init 中调用了 idt\_init 函数来初始化中断描述符表。在 proj10.1 以前的一个 proj4.4.1（其实 proj4.4.1 就是为 proj10.1 做准备的）中，为了实现了从用户态返回到内核态的功能，设置了一个特定中断号 T\_SWITCH\_TOK 的中断门，让用户态程序通过执行一个特殊的指令“INT 中断号”来完成从用户态到内核态的切换，这个中断号就是 T\_SWITCH\_TOK。proj10.1 参考 proj4.4.1 的做法，首先要设置一个特定中断号的中断门，专门用于用户进程访问系统调用。此事由 ide\_init 函数完成：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
        SETGATE(idt[i], 1, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
    lidt(&idt_pd);
}
```

在上述代码中，可以看到在执行加载中断描述符表 lidt 指令前，专门设置了一个特殊的中断描述符 idt[T\_SYSCALL]，它的特权级设置为 DPL\_USER，中断向量处理地址在 \_\_vectors[T\_SYSCALL] 处。这样建立好这个中断描述符后，一旦用户进程执行“INT T\_SYSCALL”后，由于此中断允许用户态进程产生（注意它的特权级设置为 DPL\_USER），所以 CPU 就会从用户态切换到内核态，保存相关寄存器，并跳转到 \_\_vectors[T\_SYSCALL] 处开始执行，形成如下执行路径：

vector128(vectors.S)--> \_\_alltraps(trapentry.S)-->trap(trap.c)-->trap\_dispatch(trap.c)--

-->syscall(syscall.c)-

在 syscall 中，根据系统调用号来完成不同的系统调用服务。

## 2. 建立系统调用的用户库准备

在操作系统中初始化好系统调用相关的中断描述符、中断处理起始地址等后，还需在用户态的应用程序中初始化好相关工作，简化应用程序访问系统调用的复杂性。为此在用户态建立了一个中间层，即简化的 libc 实现，在 user/libs/ulib.[ch]和 user/libs/syscall.[ch]中完成了对访问系统调用的封装。用户态最终的访问系统调用函数是 syscall，实现如下：

```
static inline int
syscall(int num, ...) {
    va_list ap;
    va_start(ap, num);
    uint32_t a[MAX_ARGS];
    int i, ret;
    for (i = 0; i < MAX_ARGS; i++) {
        a[i] = va_arg(ap, uint32_t);
    }
    va_end(ap);

    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a[0]),
          "c" (a[1]),
          "b" (a[2]),
          "D" (a[3]),
          "S" (a[4])
        : "cc", "memory");
    return ret;
}
```

从中可以看出，应用程序调用的 exit/fork/wait/getpid 等库函数最终都会调用 syscall 函数，只是调用的参数不同而已，如果看最终的汇编代码会更清楚：

```
.....
34: 8b 55 d4          mov     -0x2c(%ebp),%edx
37: 8b 4d d8          mov     -0x28(%ebp),%ecx
3a: 8b 5d dc          mov     -0x24(%ebp),%ebx
3d: 8b 7d e0          mov     -0x20(%ebp),%edi
40: 8b 75 e4          mov     -0x1c(%ebp),%esi
43: 8b 45 08          mov     0x8(%ebp),%eax
46: cd 80            int     $0x80
```

```
48: 89 45 f0          mov     %eax,-0x10(%ebp)
```

.....

可以看到其实是把系统调用号放到 EAX，其他 5 个参数 a[0]~a[4] 分别保存到 EDX/ECX/EBX/EDI/ESI 五个寄存器中，及最多用 6 个寄存器来传递系统调用的参数，且系统调用的返回结果是 EAX。比如对于 getpid 库函数而言，系统调用号（SYS\_getpid=18）是保存在 EAX 中，返回值（调用此库函数的的当前进程号 pid）也在 EAX 中。

### 3. 与用户进程相关的系统调用

在 proj10.1 中，与进程相关的各个系统调用属性如下所示：

| 系统调用名      | 含义                                        | 具体完成服务的函数                                                               |
|------------|-------------------------------------------|-------------------------------------------------------------------------|
| SYS_exit   | process exit                              | do_exit                                                                 |
| SYS_fork   | create child process, dup mm              | do_fork-->wakeup_proc                                                   |
| SYS_wait   | wait child process                        | do_wait                                                                 |
| SYS_exec   | after fork, process execute a new program | load a program and refresh the mm                                       |
| SYS_clone  | create child thread                       | do_fork-->wakeup_proc                                                   |
| SYS_yield  | process flag itself need resequeduling    | proc->need_sched=1, then scheduler will rescheule this process          |
| SYS_sleep  | process sleep                             | do_sleep                                                                |
| SYS_kill   | kill process                              | do_kill-->proc->flags  = PF_EXITING, -->wakeup_proc-->do_wait-->do_exit |
| SYS_getpid | get the process's pid                     |                                                                         |

通过这些系统调用，可方便地完成从进程/线程创建到退出的整个运行过程。

### 4. 系统调用的执行过程

与用户态的函数库调用执行过程相比，系统调用执行过程的有四点主要的不同：

- 不是通过“CALL”指令而是通过“INT”指令发起调用；
- 不是通过“RET”指令，而是通过“IRET”指令完成调用返回；
- 当到达内核态后，操作系统需要严格检查系统调用传递的参数，确保不破坏整个系统的安全性；
- 执行系统调用可导致进程等待某事件发生，从而可引起进程切换；

下面我们以 getpid 系统调用的执行过程大致看看操作系统是如何完成整个执行过程的。当用户进程调用 getpid 函数，最终执行到“INT T\_SYSCALL”指令后，CPU 根据操作系统建立的系统调用中断描述符，转入内核态，并跳转到 vector128 处（kern/trap/vectors.S），开始了操作系统的系统调用执行过程，函数调用和返回操作的关系如下所示：

```
vector128(vectors.S)--> __alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)-->
-->syscall(syscall.c)-->sys_getpid(syscall.c)-->.....-->__trapret(trapentry.S)
```

在执行 trap 函数前，软件还需进一步保存执行系统调用前的执行现场，即把与用户进程继续执行所需的相关寄存器等当前内容保存到当前进程的中断帧 trapframe 中（注意，在创建进程是，把进程的 trapframe 放在给进程的内存栈分配的空间的顶部）。软件做的工作在



vector128 和 \_\_alltraps 的起始部分:

```
vectors.S::vector128 起始处:
    pushl $0
    pushl $128
    .....
trapentry.S::__alltraps 起始处:
    pushl %ds
    pushl %es
    pushal
    .....
```

自此, 用于保存用户态的用户进程执行现场的 `trapframe` 的内容填写完毕, 操作系统可开始完成具体的系统调用服务。在 `sys_getpid` 函数中, 简单地把当前进程的 `pid` 成员变量做为函数返回值就是一个具体的系统调用服务。完成服务后, 操作系统按调用关系的路径原路返回到 `__alltraps` 中。然后操作系统开始根据当前进程的中断帧内容做恢复执行现场操作。其实就是把 `trapframe` 的一部分内容保存到寄存器内容。恢复寄存器内容结束后, 调整内核堆栈指针到中断帧的 `tf_eip` 处, 这是内核栈的结构如下:

```
/* below here defined by x86 hardware */
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
/* below here only when crossing rings */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
```

这时执行“IRET”指令后, CPU 根据内核栈的情况回复到用户态, 并把 EIP 指向 `tf_eip` 的值, 即“INT T\_SYSCALL”后的那条指令。这样整个系统调用就执行完毕了。

## 4.3 基于内核线程实现全局内存页替换机制

### 4.3.1 实验目标

到 `proj11` 为止, 还没有能够在 `ucore` 中实现一个完整的内存页替换机制。但其实在 `lab2` 的 `proj8` 中, 已经为 `ucore` 实现内存页替换机制提供了大量的支持, 并在相关测试函数 `kern/mm/swap.c::check_swap` 中进行了检查。但这个检查只是说明了 `proj8` 提供了能够完成内存页替换机制的数据结构和函数支持, 即已经一砖一瓦地完成了门窗、墙壁等建筑工作, 还差把相关部件完整组织起来实现成一个完整的房子。`proj11` 就是完成这最后一步, 采用内核线程来实现内存页替换机制, 使得用户进程在快用完内存后, 可以通过内存页替换机制把不常用的页换出到硬盘 `swap` 分区中, 常用的页保存在内存中, 保持系统中有足够的内存给用户进程使用。

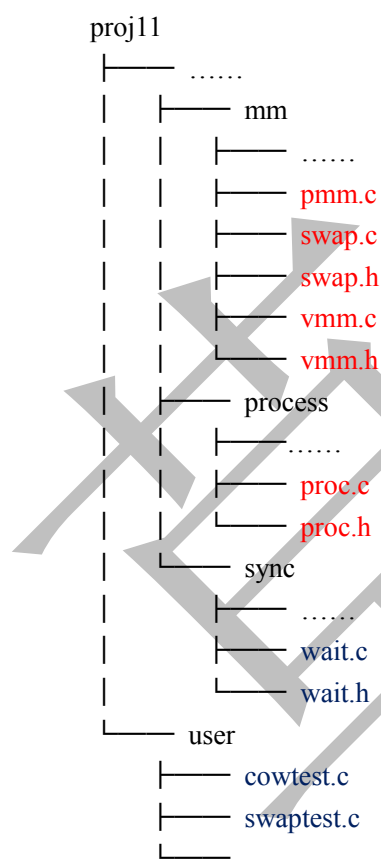


## 4.3.2 proj11 概述

### 1. 实现描述

proj11 是 lab3 的第六个 project。它在 proj10.4 的基础上实现了基于内核线程的内存页替换机制，主要扩展设计了专门用于执行内存页替换的内核线程 `kswapd`，并增加了等待队列、扩展了进程控制块的成员变量 `mm` 的等，使得在用户进程申请内存不足或系统空闲内存不足的情况下，通过执行 `kswapd` 内存线程，实现内存页替换，把不常用的页放到硬盘 `swap` 分区上，给系统提供足够的空闲空间。

### 2. 项目组成



17 directories, 114 files

相对于 proj10.4，proj11 在内核方面主要增加了有关 `kswapd` 内核线程和相关函数以及等待队列实现，在用户程序方面，增加测试 `ucore` 的 `COW` 实现的用户程序 `cowtest.c` 和测试内存页置换实现的 `swaptest.c`。主要修改和增加的文件如下：

- `kern/mm/pmm.c`: 扩展了 `alloc_pages` 函数，使得它能够在没有获得所需空闲内存页后，进一步调用 `try_free_pages` 来要求 `ucore` 释放足够的空闲页，从而再次要求所需空闲页，直到要求得到满足为止。
- `kern/mm/swap.[ch]`: 更新 `try_free_pages` 的实现，完成让当前进程睡眠，并唤醒

---

kswapd 内核线程，让它完成对空闲页的回收。同时实现了内核线程 kswapd 的执行主体 kswapd\_main 函数，此函数完成具体的内存页置换机制。

- kern/sync/wait.[ch]: 实现等待队列机制，使得内存页等资源无法得到满足的进程能够处于等待状态，并在资源得到满足后让进程继续执行。
- kern/mm/vmm.[ch]: 扩展了 mm\_struct 结构，并修改相关函数，使得所有进程的成员变量 mm 能够链入到全局 mm\_struct 结构的链表 proc\_mm\_list 中。

### 3. 编译运行

编译并运行 proj11 的命令如下：

```
make
```

```
make qemu
```

则可以得到如下显示界面

```
(THU.CST) os is loading ...
```

```
Special kernel symbols:
```

```
.....
```

```
check_vmm() succeeded.
```

```
ide 0:      10000(sectors), 'QEMU HARDDISK'.
```

```
ide 1:     262144(sectors), 'QEMU HARDDISK'.
```

```
check_swap() succeeded.
```

```
.....
```

```
++ setup timer interrupts
```

```
kernel_execve: pid = 3, name = "swaptest".
```

```
buffer size = 00500000
```

```
parent init ok.
```

```
child 9 fork ok, pid = 13.
```

```
child 8 fork ok, pid = 12.
```

```
child 7 fork ok, pid = 11.
```

```
child 6 fork ok, pid = 10.
```

```
child 5 fork ok, pid = 9.
```

```
child 4 fork ok, pid = 8.
```

```
child 3 fork ok, pid = 7.
```

```
child 2 fork ok, pid = 6.
```

```
child 1 fork ok, pid = 5.
```

```
child 0 fork ok, pid = 4.
```

```
check cow ok.
```

```
round 0
```

```
round 1
```

```
round 2
```

```
round 3
```

```
round 4
```

```
child check ok.
wait ok.
check buffer ok.
swapttest pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:430:
    initproc exit.
```

Welcome to the kernel debug monitor!!

Type 'help' for a list of commands.

K>

表面上看不出上述输出对内存页置换算法实现的具体体现。不过通过 `Makefile` 和对 `swapttest.c` 程序的分析，还是能够看出 `proj11` 的执行与其他进程的执行不同：

`Makefile`:

```
.....
QEMUOPTS      =      -m      48m      -hda      $(UCOREIMG)      -drive
file=$(SWAPIMG),media=disk,cache=writeback
```

`swapttest.c`

```
.....
const int size = 5 * 1024 * 1024;
char *buffer;
.....
main(void){
.....
    (buffer = malloc(size))
.....
    for (i = 0; i < pids; i++) {
        if ((pid[i] = fork()) == 0) {
.....
        }
    }
}
```

通过 `Makefile`，可以看到 `qemu` 只模拟出了 48MB 的物理内存空间，但 `swapttest.c` 创建了 10 个子进程，且每个子进程都会复制全局变量 `buffer`，且会对 `buffer` 中的所有元素进行写操作。由于每个 `buffer` 的空间大小为 5MB，所以 10 个子进程和 1 个父进程的 `buffer` 所占虚拟空间总和为 55MB，大于实际的 48MB 物理内存空间。而在操作系统设计上，用户进程的用户空间是没必要都保存在内存中，这使得必须把某些页换出到硬盘 `swap` 分区才能确保所有子进程都能正常执行完毕。为此，我们还需进一步分析 `proj11` 中 `ucore` 具体的内存页置换机制的实现和执行过程。下面将从实现方面对此进行进一步阐述。

### 4.3.3 等待队列设计与实现

为了支持用户进程完成特定事件的等待和唤醒操作，`ucore` 设计了等待队列，从而使得用户进程可以方便地实现由于某事件没有完成而睡眠，并且在事件完成后被唤醒的整个操作过程。

其基本设计思想是：当一个进程由于某个事件没有产生而需要在某个睡眠等待时，设置自身运行状态为 `PROC_SLEEPING`，等待原因为某事件，然后将自己的进程控制块指针和等待标记组装到一个数据结构为 `wait_t` 的等待项数据中，并把这个等待项的挂载到等待队列 `wait_queue` 的链表中，再执行 `schedule` 函数完成调度切换；当某些事件发生后，另一个任务（进程）会唤醒等待队列 `wait_queue` 上的某个或者所有进程，唤醒操作就是将等待队列 `wait_queue` 中的等待项中的进程运行状态设置为可调度的状态，并且把等待项从等待队列中删除。下面是等待队列的设计与实现分析。

## 1. 数据结构描述

等待项的定义：

```
typedef struct {
    struct proc_struct *proc;
    uint32_t wakeup_flags;
    wait_queue_t *wait_queue;
    list_entry_t wait_link;
} wait_t;
```

这里等待项的成员变量 `proc` 表明了等待某事件的进程控制块指针，`wakeup_flags` 是唤醒进程的事件标志（多个标志可以有逻辑或的关系，形成复合事件标志），`wait_queue` 是此等待项所属的等待队列，`wait_link` 用于链接到等待队列 `wait_queue` 中。

等待队列的定义：

```
typedef struct {
    list_entry_t wait_head;
} wait_queue_t;
```

等待队列就是一个双向链表的头指针。

## 2. 等待队列相关操作函数

### ● 初始化

如果要使用等待队列，首先需要声明并初始化等待队列。以 `proj11` 为例，在 `kern/mm/swap.c` 中有一个等待队列的变量声明和在 `swap_init` 函数中执行的对应初始化：

```
static wait_queue_t kswapd_done;
...
wait_queue_init(&kswapd_done);
```

### ● 执行等待

如果某进程需要等待某事件，则需要设置自己的运行状态为 `PROC_SLEEPING`，构建并初始化一个等待项，再挂入到某个等待队列中。以 `proj11` 为例，某进程申请内存资源无法满足，需要等待 `kswapd` 内核线程给系统更多的内核资源，于是在 `try_free_pages` 函数中执行了如下操作：

```
wait_t __wait, *wait = &__wait;
wait_init(wait, current);
current->state = PROC_SLEEPING;
```

```
current->wait_state = WT_KSWAPD;
wait_queue_add(&kswapd_done, wait);
```

这里可以看到，首先声明了一个等待项 `__wait`，然后调用 `wait_init` 函数对此等待项进行了初始化；并进一步把当前进程的运行状态设置为 `PROC_SLEEPING`，睡眠原因设置为 `WT_KSWAPD`，即等待 `kswapd` 释放出更多的空闲内存；最后把此等待项加入到等待队列 `kwapd_done` 中。

- 执行唤醒

当某个事件产生后，需要唤醒等待在等待队列中的睡眠进程。以 `proj11` 为例，当 `kswapd` 内核线程释放出更多的空闲内存后，就需要唤醒等待更多内存的进程，在 `kswapd` 内核线程的主体执行函数 `kswapd_main` 中调用了 `kswapd_wakeup_all` 函数：

```
wakeup_queue(&kswapd_done, WT_KSWAPD, 1);
```

这个函数就完成了唤醒功能，它会遍历 `kswapd_done` 等待队列上的所有等待项，找到一个就执行 `wakeup_wait` 函数，来进一步调用 `wakeup_proc` 函数来唤醒挂在等待项上的睡眠进程。

上面是使用等待队列的基本流程。为了更好地完善整个基于等待队列的等待唤醒机制，在 `wait.[ch]` 中提供了一系列函数：

- `void wait_init`: 初始化等待项
- `void wait_queue_init`: 初始化等待队列
- `void wait_queue_add`: 把一个等待项加入到一个等待队列中
- `void wait_queue_del`: 从一个等待队列中删除一个等待项
- `wait_t *wait_queue_next`: 查找挂在某等待队列中的等待项指向的下一个等待项
- `wait_t *wait_queue_prev`: 查找挂在某等待队列中的等待项指向的前一个等待项
- `wait_t *wait_queue_first`: 查找挂在某等待队列中的第一个等待项
- `wait_t *wait_queue_last`: 查找挂在某等待队列中的最后一个等待项
- `bool wait_queue_empty`: 判断等待队列是否为空
- `bool wait_in_queue`: 单晶某等待项是否在等待队列中
- `void wakeup_wait`: 唤醒等待项中的睡眠进程，删除等待队列中的等待项（参数 `del` 确定是否删除）
- `void wakeup_first`: 唤醒等待队列中第一个等待项中的睡眠进程，删除等待队列中的这个等待项（参数 `del` 确定是否删除）
- `void wakeup_queue`: 唤醒等待队列中所有等待项中的睡眠进程，删除等待队列中的对应等待项（参数 `del` 确定是否删除）

### 4.3.4 内存页置换机制的执行过程

在 `lab2/proj8` 中已经完成了大部分内存页置换所需的功能函数，但还没有有机地整和在 `ucore` 中，成为 `ucore` 内存管理子系统的组成部分。当有了内核线程机制后，我们就可以把内存页置换的功能整合到一个内核线程中，从而实现一个内存页置换线程 `kswapd`，专门负责完成内存页置换的工作。

## 1. 创建 `kswapd` 内核线程

在第 1 个内核线程 `initproc` 的主体执行函数 `init_main` 中，完成了对 `kswapd` 内核线程的创建工作：

```

static int
init_main(void *arg) {
    int pid;
    if ((pid = kernel_thread(kswapd_main, NULL, 0)) <= 0) {
        panic("kswapd init failed.\n");
    }
    kswapd = find_proc(pid);
    set_proc_name(kswapd, "kswapd");
    .....

```

并且保存了创建用户进程前剩余页的情况和已经分配的 slab 的容量计数，这样在接下来创建完用户进程，并且所有用户进程执行完毕后，再判断所有进程结束后的剩余页的情况和已经分配的 slab 的容量计数，如果二者相等，这表明内存管理功能基本正确。

```

    size_t nr_free_pages_store = nr_free_pages();
    size_t slab_allocated_store = slab_allocated();
    .....
    cprintf("all user-mode processes have quit.\n");
    .....
    assert(nr_free_pages_store == nr_free_pages());
    assert(slab_allocated_store == slab_allocated());
    cprintf("init check memory pass.\n");
    return 0;

```

## 2. 触发 kswapd 内核线程

ucore 目前大致有两种触发 kswapd 内核线程的策略，即积极策略和消极策略。积极策略是指 ucore 周期性地（或在系统不忙的时候）唤醒 kswapd 内核线程，让它主动把某些认为“不常用”的页换出到硬盘上，从而确保系统中总有一定数量的空闲页存在，这样当需要空闲页时，基本上能够及时满足需求；消极换出策略是指，ucore 试图得到空闲页时，发现当前没有空闲的物理页可供分配，这时才唤醒 kswapd 内核线程，让 kswapd 开始查找“不常用”页面，并把一个或多个这样的页换出到硬盘上。

对于积极策略，即每隔 1 秒唤醒一次内核线程 kswapd。在实现上是基于定时器的触发，即在 kswapd 的主体执行函数 kswapd\_main 的末尾执行了“do\_sleep(1000);”，这表明了每隔 1 秒，kswapd 就要执行一个回收空闲线程的迭代执行过程。对于消极策略，则是在 ucore 调用 alloc\_pages 函数获取空闲页时，此函数如果发现无法从页分配器获得空闲页，就会进一步调用 try\_free\_pages 来唤醒线程 kswapd，让 kswapd 换出某些页。在执行 try\_free\_pages 函数时，由于当前用户进程要求的空闲内存空间无法得到满足，所以首先让当前用户进程睡眠并加入到等待队列中，睡眠原因设置为 WT\_KSWAPD，即等待 kswapd 释放出更多的空闲内存，然后唤醒 kswapd:

```

    .....
    wait_init(wait, current);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_KSWAPD;

```

```

wait_queue_add(&kswapd_done, wait);
if (kswapd->wait_state == WT_TIMER) {
    wakeup_proc(kswapd);
}
.....

```

### 3. 全局页面置换算法的数据结构设计

根据 lab2 的设计,我们可以知道 ucore 中表示内存中物理页使用情况的变量是基于数据结构 Page 的全局变量 pages 数组, pages 的每一项表示了计算机系统中一个物理页的使用情况。如果一个物理页在硬盘上有一个页备份,则 ucore 需要记录在硬盘中页备份的位置,同时还需要记录 swap 分区上的页备份的使用计数。

为了表示物理页可被换出或已被换出的情况,ucore 对部分内存相关数据和数据结构进行了扩展。如果一个页被换出了,则页的内容保存在硬盘 swap 分区上有一个连续扇区中(称为一个 swap page),而对应此页的页表项 PTE 的 Present 位为 0,表示已经没有对应的物理内存页映射关系了,但如果高 24 位不为 0,则表示这个页虽然在物理内存中不存在了,但保存在 swap 分区中以 PTE 高 24 位为偏移位置的 swap page 中。这里我们把 Present 位为 0 且高 24 位不为 0 的 PTE 称为一个 swap entry。一个有效的 swap entry 对应着一个 swap page。而且不同页表中的 swap entry 可对已一个 swap page。

为了表示存储 swap 分区上的页的使用计数,在 swap.c 里面声明了全局的 mem\_map 数据结构。如果一个 ucore 在 swap 分区上分配了一个

swap.c 里还声明了两个链表,分别是 active\_list 和 inactive\_list,分别表示已经有对应 swap page 的且处于“活跃”状态/“不活跃”状态的物理页所形成的链表。所有已经有对应 swap page 的物理页 page 必须处于两个链表中的一个。

为了更好地对应 swap page/swap entry,在描述物理页的 Page 数据结构专门设立了几个与页面置换相关的成员变量:

```

struct Page {
    uint32_t flags;    // array of flags that describe the status of the page
    frame
    swap_entry_t index;    // stores a swapped-out page identifier
    list_entry_t swap_link; // swap hash link
    .....
};

```

首先 flag 的含义做了扩展:

```

// the page is in the active or inactive page list (and swap hash table)
#define PG_swap    4
// the page is in the active page list
#define PG_active  5

```

Page 结构中的 index 值表示物理页在硬盘中页备份的位置,它保存了被换出的页的页表项 PTE (即 swap entry) 高 24 位的内容,即硬盘中对应页备份的起始扇区位置值(以扇区为单位)。如果 page 数据结构的 flags 设置了 PG\_swap 为 1,则表示该 page 中的 index 是有效的 swap entry 的索引值,从而该物理页上的数据可以被写出到 index 所表示的 swap page 上去。

Page 结构中的 swap\_link 保存了以 entry 为 hash 索引的链表项，这样根据 entry，就可以快速的对 page 数据结构进行查找。但 hash 数组在哪里呢？对于在硬盘上有页备份的物理页（简称 swap\_page），需要统一管理起来，为此在 proj8 中就增加了全局变量：

```
static list_entry_t hash_list[HASH_LIST_SIZE];
```

hash\_list 数组就是我们需要的 hash 数组，根据 index(swap entry) 索引全部的 swap\_page 的指针，这样通过 hash 函数

```
#define entry_hashfn(x) (hash32(x, HASH_SHIFT))
```

可以快速地根据 entry 找到对应的 page 数据结构。

正如页替换算法描述的那样，把“常用”的可被换出页和“不常用”的可被换出页分别集中管理起来，形成 active\_list 和 inactive\_list 两个链表。ucore 的页面置换算法会根据相应的准则把它认为“常用”的物理页放到 active\_list 链表中，而把它认为“不常用”的物理页放到 inactive\_list 链表中。一个标记了 PG\_swap 的页总是需要在这两个链表之间移动。

前面介绍过 mem\_map 数组，他是用来记录 swap\_page 的引用次数的。因为 swap 分区上的 swap page 实际上是某个物理页的数据备份。所以，一个物理页 page 的 page\_ref 与对应 swap page 的 mem\_map[offset] (offset = swap\_offset(entry)) 值的和是这个页的真实引用计数。page\_ref 表示 PTE 对该物理页的映射的个数；mem\_map 表示 PTE 对该 swap 备份页的映射个数。当 page\_ref 为 0 的时候，表示物理页可以被回收；当 mem\_map[offset] 为 0 的时候，表示 swap page 可以被重新存储其他的物理页内容（前面介绍过，可以回收，但是在万不得已的情况下才真的回收）。

#### 【注意】

ucore 目前使用的 PIO 的方式读写 IDE 磁盘，这样的好处是，磁盘读入、写出操作可以认为是同步的，即当前 CPU 需要等待磁盘读写完毕后再进行进一步的工作。由于磁盘操作相对 CPU 的速度而言是很慢的，这使得会浪费大量的 CPU 时间在等 IO 操作上。于是我们总是希望能够在 IO 性能上有更大的提升，比如引入 DMA 这种异步的 IO 机制，为了避免后续开发上的各种不便和冲突，我们假设所有的磁盘操作都是异步的（也包括后面的实验），即使目前是通过 PIO 完成的。

假定某 page 的 flags 中的 PG\_swap 标志位为 1，并且 PG\_active 标志位也为 1，则表示该 page 在 swap 的 active\_list 中，否则在 inactive\_list 中。active\_list 中的页表示活跃的物理页，即页表中可能存在多个 PTE 指向该物理页（这里可以是同一个页表中的多个 entry，在后面 lab3 的实验里面有了进程以后，也可以是多个进程的页表的多个 entry）；反过来，inactive\_list 链表所链接的 page 通常是指没有 PTE 再指向的页。

#### 【注意】

需要强调两点设计因素：

- 1、一个 page 是在 active\_list 还是在 inactive\_list 的条件不是绝对的；
- 2、只有 inactive\_list 上的页才会被尝试换出。

这两个设计因素的设计起因如下：

1. 我们知道一个 page 换出的代价是很大的（磁盘操作），并且我们假设所有的磁盘操作都是异步的，那么换出一个 active 的页就变得非常不值得。因为在还有多个 PTE 指向他情况下进行换出操作（异步 IO 可能导致进程切换）的较长过程中，这个页可以随时被其它进程写脏。而硬件提供给内核的接口（即页表项 PTE 的 dirty 位）使得内核只能知道一个页是否是脏的（不能明确知道一个页的哪个部分是脏的），当这种情况发生时，就导致了一次无效的写出。

2. active\_list 和 inactive\_list 的维护只能由与 swap 有关的集中操作来完成。



特别是在 lab3/proj11 引入 kswapd 内核线程之后，所有的内存页换出任务都交给 kswapd，这样减少了复杂的同步互斥实现（在 lab5 中会重点涉及）。

3. 页面换入换出有关的操作需要做的就是尽可能的完成如下三件事情：

(1) 将 `PG_swap` 为 0 的页转变成 `PG_swap` 为 1 的页。即尽可能的给每个物理页分配一个 `swap entry`（当然前提是有足够大的 `swap` 分区）。

(2) 将页从 `active_list` 上移动到 `inactive_list` 上。如果一个页还在 `active_list` 上，说明还有 PTE 指向此“活跃”的物理页。所以需要在完成内存页换出时断开对这些物理页的引用，把它变成不活跃的（inactive）。只有把所有的 PTE 对某 page 的引用都断开（即 page 的 `page_ref` 为 0）后，就可以将此 page 从 `active_list` 移动到 `inactive_list` 上。

(3) 将 `inactive_list` 上的页写出并释放掉。`inactive_list` 上的 page 表示已没有 PTE 指向此 page 了，那么该 page 可以被释放，如果该 page 被写过，那还需把此 page 换出到 swap 分区上。如果在整个换出过程（异步 IO）中没有其他进程再写这个物理页（即没有 PTE 在引用它或有 PTE 引用但页没有写脏），就认为这个物理页是可以安全释放的了。那么将它从 `inactive_list` 上取下，并调用 `page_free` 函数实现 page 的回收。

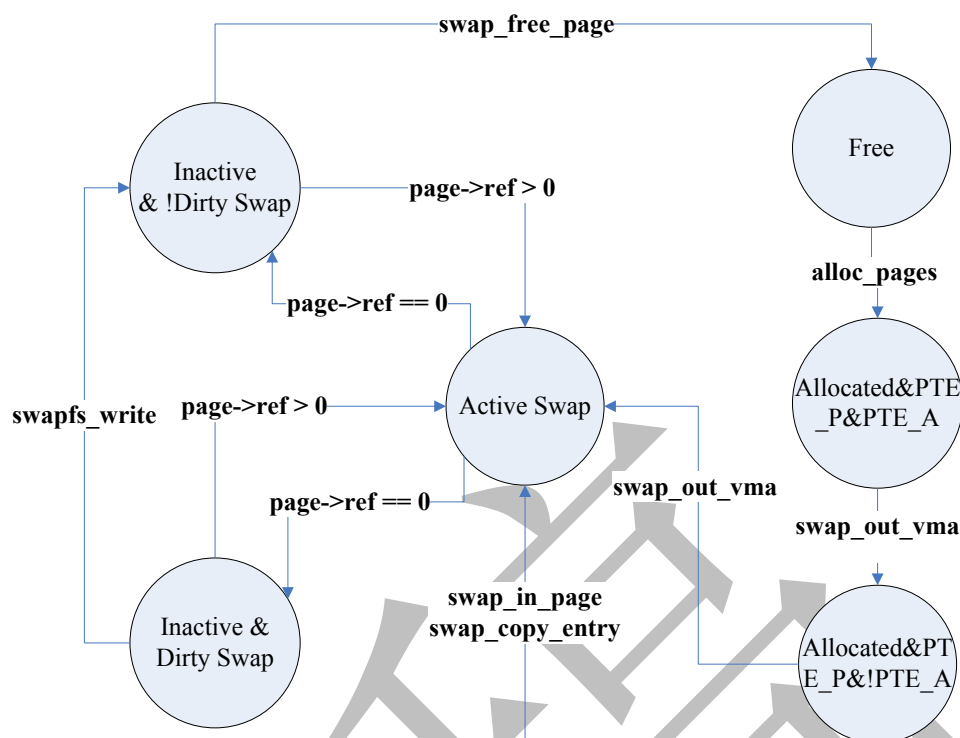
4. 值得注意的是，内存页换出操作只有特定时候才被调用，即通过执行 `try_free_pages` 函数或者定时器机制（在 lab3/proj10.4 才引入）定期唤醒 kswapd 内核线程。这样会导致内存页换出操作对两个链表上的数据都不够敏感。比如处于 `active_list` 上的 page，可能在 kswapd 工作的时候，已经没有 PTE 再引用它了；再如相应的进程退出了，并且相应的地址空间已经被内核回收，从而变成了一个 inactive 的 page；还存在 `inactive_list` 上的 page 也可能在换出的时候，其它进程通过 page fault，又将 PTE 指向他，进而变成一个实际上 active 的页。所以说，active 和 inactive 条件并不绝对。

## 4. 全局页面置换算法的执行逻辑

其实在 lab2/proj8 中并没有完全实现页面置换算法，只是实现了其中的部分关键函数，并通过 `check_swap` 来验证了这些函数的正确性。直到 lab3 的 proj11 才形成了完整的页面置换逻辑，而这个页面置换逻辑基本上是改进的时钟算法的一个实际扩展版本。

### 页状态变化关系

ucore 采用的页面置换算法是一个全局的页面置换算法，因为它收集了 ucore 中所有用户态进程（这里可理解为 ucore 中运行的每个用户态程序）的可换出页，并把这些可换出页中的一部分转换为空闲页。其次它考虑了页的访问情况（根据 PTE 中 `PTE_A` 位的值）和读写情况（根据 PTE 中 `PTE_D` 位的值）。如果页被访问过，则把 `PTE_A` 位清零继续找下一页；如果页没有被访问过，这此页就成为了 active 状态的可换出页，并放入 `active_list` 链表中，这时需要把对应的 PTE 转换成为一个 `swap entry`（高 24 位保存为硬盘缓存页的起始扇区号，`PTE_P` 位清零）；接着 `refill_inactive_scan` 函数会把处于 active 状态的部分可换出页转换成 inactive 状态，并放入 `inactive_list` 链表中；然后 `page_launder` 函数扫描 `inactive_list` 中的处于 inactive 状态的可换出页，如果此页不是 dirty 的，则把它直接转换成空闲页，如果此页是 dirty 的，则执行换出操作，把该页换出到硬盘上保存。这个页的状态变化图如下图所示。



ucore 的物理页状态变化图

### 确定腾出的空闲页数和断开的 PTE 映射数

在 proj11 中同时实现了积极换出策略和消极换出策略，这都是通过在不同的时机执行 `kwapd_main` 函数来完成的。当 `ucore` 调用 `alloc_pages` 函数以获取空闲页，但物理页内存分配器无法满足请求时，`alloc_pages` 函数将调用 `try_free_pages` 来通过直接唤醒内核线程 `kswapd` 的方式执行 `kwapd_main` 函数，完成对页的换出操作和生成空闲页的操作。这是一种消极换出的策略。另外，`ucore` 也设立了通过设置 `timer` 来唤醒线程的方式每秒执行一次 `kwapd_main` 函数，完成对页的换出操作和生成空闲页的操作。这是一种积极换出的策略。

如果 `alloc_pages` 执行分配 `n` 连续物理页失败，则会通过调用 `tree_free_page` 来唤醒 `kswapd` 线程。`kswapd` 需要尽可能的满足分配 `n` 个连续物理页的需求。既然需求是 `n` 个连续物理页，那么 `kswapd` 所需要释放的物理页就应该大于 `n` 个；每个页可能在某个或者许多个页表的不同地方有 PTE 映射（特别是 `copy on write` 之后，这种情况更为普遍），那么 `kswapd` 所需要断开的 PTE 映射就远远不止 `n` 个。

Linux 实现了能够根据 `physical address` 在页表中快速定位的数据结构，但是实现起来过于复杂，这里 `ucore` 采用了一个比较笨的方法，即遍历所有存在的页表结构，断开足够多的 PTE 映射。这里足够多是个经验公式，采用 `n << 5`。当然这也可能失败，那么 `kswapd` 就会尝试一定次数。当他实在无能为力时，也就放弃了。而 `alloc_pages` 也会不停的调用 `try_free_pages` 进行尝试，当尝试不停的遭遇失败时，程序中会有许多句 `warn` 来输出这些调试信息。而 Linux 的方案是选择一个占用内存最多的进程杀掉并释放出资源，来尽可能的满足当前程序的需求（注意，这里当前程序是指内核服务或者调用），直到程序从内核态正常退出；`ucore` 的这种设计显然是相对简单了

### 页面置换大致流程

`kswapd_main` 是 `ucore` 整个页面置换算法的总控部分，其大致思路是根据当前的空闲

页情况查找出足够多的可换出页（swap page，即 page 的 PG\_swap 标志位为 1），然后根据这些可换出页的访问情况确定哪些是“常用”页（即 page 的 PG\_swap 标志位和 PG\_active 标志位为 1），哪些是“不常用”页，最后把“不常用”的页转换成空闲页。思路简单，但具体实现相对复杂。kswapd 腾出更多空闲页的核心是尽可能的断开页表中的 PTE 映射，其整个处理流程分为三步：

1. 就是把尽可能多的 page（有对应合法 PTE 项，即对应 PTE 项的 PTE\_P 标志位为 1）转变成 具有 PG\_active 的 page（此时的 page PG\_swap 标志位和 PG\_active 标志位都为 1，但对应的 PTE 项的 PTE\_P 标志位为 0），并移动到 active list 中；
2. 接着把 active list 中的页（这些 page 的 PG\_swap 标志位和 PG\_active 标志位为 1）尽可能多的变成 inactive list 中页（这些 page 的 PG\_swap 标志位为 1 且 PG\_active 标志位为 0）；
3. 最后把 inactive list 的页（这些 page 的 PG\_swap 标志位为 1 且 PG\_active 标志位为 0）转换成空闲页，如果这些页是 dirty 的，则在转换为空闲页之前，先要把页的内容换出（也称为 launder，洗净）到 swap 分区对应的 swap page 中。

扫描页表是一项艰巨的任务，因为除了内核空间，用户地址空间有将近 3G 的空间，真正的程序很少能够用这么多。因此，充分利用虚存管理能够提升扫描页表的速度。

### 页面置换具体流程

现在我们来介绍以下 kswap\_main 是如何一步一步完成 swap 的操作的。正如前面介绍过的，swap 需要完成 3 件事情，下面对应的是这三个操作的具体细节：

#### 1. 断开足够多的页表项 PTE

kswapd\_main 函数通过循环调用函数 swap\_out\_mm 并进一步调用 swap\_out\_vma，来查找 ucore 中所有存在的虚存空间，并总共断开 m 个 PTE 到物理页的映射。为何是查找 ucore 中所有存在的虚存空间，而不是直接扫描每个进程的虚存空间呢？这是因为虽然每个用户进程都拥有一个自己的虚存空间，但还存在虚存空间在多个进程之间的共享的情况。所以遍历虚存空间而不是遍历每个进程可避免某个虚存空间被很多进程共享进而被 kswapd 过度压榨所带来的不公平情况。可以想象，被过度压榨的虚存空间，同时又由于被很多进程共享从而有很高的概率被使用到，最终必然会导致频繁的内存访问错误异常#PF，给系统不必要的负担。

另外虽然 kswapd\_main 的第一个任务是断开 m 个 PTE 映射，但是实际上它对每个虚存空间都一次至多提出断开 32 个映射的需求，并循环遍历所有的虚存空间直到 m 得到满足。这样做的目的也是为了保证公平，使得每个虚存空间被交换出去的页的几率是近似相等的。Linux 实际上应该有更好的实现，它根据虚存空间所实际使用的物理页的个数来决定断开的映射的个数。这些断开的 PTE 映射所指向的物理页如果没有 PG\_active 标记，则需要给它分配一个新的 swap entry，并做好标记，将 page 插入到 active\_list 中去（同时也插入到 swap 的哈希表中），然后设置好相应的 page\_ref 和 mem\_map[offset] 的值。

当然，如果找不到空闲的 swap entry 可以分配（比如 swap 分区已经用光了），我们只能跳过这样的 PTE 映射，从下一个地址继续寻找出路；对于原来就已经标记了 PG\_swap 的物理页，则只需要完成后面的工作，即调整引用计数就足够了。

断开的 PTE 被 swap\_entry 取代，并取消 PTE\_P 标记，，这样当出现#PF 的时候，我们能够直接根据 PTE 上的值得到该页的数据实际是在 swap 分区上的哪个位置上。

现在这一阶段的工作只是断开 PTE 映射，余下的工作后面会一步步完成。需要注意现在

还不需要考虑一个 page 究竟是放在 active\_list 还是放在 inactive\_list 中,也还没涉及页换出操作。当 kswapd 断开足够数量的 PTE 映射以后,第一阶段的工作也就完成了。

当 kswapd 发现自己竭尽所能的遍历都无法满足断开 m 个链接的需求时,该怎么办? 我们需要明确的是 swap 操作的主要目的是释放物理页,而断开 PTE 映射是一个必要的步骤,作用是尽可能的扩大 inactive\_list 中 page 的个数,为物理页的换出提供更大的基数(操作空间),但这并不是换页主要过程。所以为了防止在这一步陷入死循环,kswapd\_main 最多会对全部虚存空间的链表尝试 rounds=16 次遍。

【注意】虚存管理数据结构 mm\_struct 新增加了一个 swap\_address,表示上一次页换出操作结束时的地址。维护这个数据是避免每次 swap 操作都从虚存空间的起始地址开始,从而导致过多数量的重复且无效的遍历。

## 2. 转换 inactive page

refill\_inactive\_list 从函数名上可以看出,实际上就是遍历 active\_list,把实际上不活跃的(inactive) page 从 active\_list 上取下,放到 inactive\_list 上,方便下一轮 page\_laundry 的操作。

## 3. 页换出和释放页

通过 page\_laundry 函数,完成遍历 inactive\_list 并实现页的释放与换出。page\_laundry 的实现,涉及 ucore 内核代码设计的一个重要假设前提,ucore 的内核代码是不可抢占的(具体细节请参考 4.5.4 小节)。这部分和上述的 refill\_inactive\_list 函数操作的先后顺序并不那么严格。通俗的解释就是 page\_laundry 实现的是把 inactive\_list 中的 page 洗净(即完成 page 的释放和换出),当然也顺便实现了把实际上活跃的(active)的 page 从 inactive\_list 上取下,放回 active\_list 的过程。page\_laundry 其实是比较复杂的过程,需要仔细的分析一下。page\_laundry 先检查一个 page 的 page\_ref 是否 != 0。如果是,则表示该 page 实际上是 active 的,则把它移动到 active\_list 上去;如果不是,则需要对该页进行释放和换出操作。具体过程如下

【注意】下面讨论的是以 page\_ref == 0 作为前提)

(1) 如果一个页的 mem\_map 项为 0,说明这个时候已经没有 PTE 映射指向它了,无论是物理页还是 swap 备份页。那么这个页也就没有必要洗净了。可以直接释放物理页以及相应的表示为 swap entry 的 PTE 了。(同时还需处理 swap page 有关的链表,以下不再赘述)

(2) 如果一个页的 mem\_map 项不为 0,但是没有 PG\_dirty 标记:page 数据结构里面有 PG\_dirty 标记,swap 部分的代码根据这个标记来判断一个页是否需要被洗净(写到 swap 分区上)。这个 PG\_dirty 标记在什么情况下设置,我们稍后会讨论。这种情况可以等价于物理页上的数据和 swap 分区上的数据是一致的,所以不需要洗净该页,因为该物理页本身就已经足够干净了。所以可以安全的和(1)中的操作一样对该物理页进行释放。

(3) 如果一个页的确有 PG\_dirty 标记:表示该页需要被洗净,这需要调用 swapfs\_write 函数,完成将物理页写到磁盘上的操作。前面已经强调过,我们假设所有磁盘操作是异步 IO。先明确一下,当前的状态,page\_ref=0 &&mem\_map!=0 &&PG\_dirty,那么在执行写到磁盘的过程中就可能发生下面许多种可能的场景:

- swapfs\_write 操作失败了。磁盘操作不像内存操作,它应该允许发生更多的错误。
- 其它进程又访问到相应的数据页,前面提到过,因为 PTE 的 PTE\_P 标志位为 0 了,所以会产生#PF,内核会根据 PTE 的内容在 swap hash 里面查找到相应的物理

页，并将它重新插入到相应的页表中，并更新该 page 的 page\_ref 和 mem\_map 的值。整个过程发生时，swapfs\_write 还没有结束。那么当完成洗净一个页的操作时（写到 swap 分区），swap 部分的代码应该有能力检测出这种变化。也就是在 swapfs\_write 之后，需要再判断 page\_ref 是否依然满足 inactive 的。

c. 和 b 类似，不过不同的是，这次对物理页进行的是一个写操作。操作完成之后，进程又将该 PTE 指向的页释放掉了。那么当 swapfs\_write 返回的时候，它面对的条件，可能就变成了 page\_ref=0 &&mem\_map=0 &&PG\_dirty。它应该能够处理这个变化。

d. 和 c 类似，不同的是，该 page 有两个不同的 PTE 映射。那么在 swapfs\_write 操作之前，状态可能是 page\_ref =0&&mem\_map=2&&PG\_dirty，那么当 c 中的情况发生以后，该物理页的状态就可能变成了 page\_ref=0&&mem\_map=1&&PG\_dirty 了。swap 应该能够处理这种变化。

综上所述，page\_launder 部分的代码变得相对复杂很多。大家可以参照程序了解 ucore 是怎么解决这种冲突的。

### 其他注意事项

因为 swapfs\_write 是异步操作，并且是对该 page 的操作，ucore 为了保证在操作的过程中，该页不被释放（比如一个进程通过 #PF，增加 page\_ref 到 1，然后又通过释放该 page 减少 page\_ref 到 0，进而触发内核执行 page\_free 的操作），分别在 swapfs\_write 前后获得和释放该 page 的引用（page\_ref\_inc/page\_ref\_dec）。但事实证明，这种担心是多余的。理由很简单，当 page\_launder 操作一个页的时候，该页是被标记 PG\_swap 的，这个标记一方面表示 page 结构中的 index 有意义，另一方面也说明这样的 page 的释放，只能由 swap 部分的代码来完成（参见 pmm.c 以及后面 shmem.c 的处理）。所以，swap 在操作该 page 的时候，不可能有程序能够调用 free\_page 释放该 page。

而相反的，mem\_map 是一个需要保护的数据。此外，可以翻阅一下涉及到 page\_ref 修改的 pmm 部分的代码，不难发现，当一个 page 从 PTE 断开的时候，也就是 page\_ref 下降的时候，ucore 会根据 PTE 上的硬件设置的 PTE\_D 来设置 PG\_dirty。其实这就足够了。因为 PG\_dirty 并不需要时时刻刻都十分的准确，只要在 swap 尝试判断该 page 是否需要洗净的时候，PG\_dirty 是正确的，就足够了。所以只需要保证每次 page\_ref 下降的时候，PG\_dirty 是正确的即可。除此之外，在对每个页分配 swap\_entry 的时候，需要保证标记 PG\_dirty，因为毕竟是刚刚分配的，物理页的数据还从来没有写出去过。

总结一下，页换入换出的实现很复杂，但是相对独立。并且正是由于 ucore 的内核代码不可抢占使得实现变得相对容易一些。只要是不涉及 IO 操作，大部分过程都可以认为处于不可抢占的内核执行过程。

## 4.4 创建并执行用户线程

### 4.4.1 实验目标

到 proj12 为止，ucore 还一直没有用户线程。而用户线程与用户进程的区别在于操作系统用户进程管理除了涉及与执行过程相关的调度、进程上下文切换、执行状态变化外，还需管理内存、文件等资源，而用户线程只管理与执行过程相关的调度、上下文切换、执行状态变化。这样使得在执行线程创建、删除和线程上下文切换时的开销比对进程做类似的事情要

少很大的开销。从而在操作系统的用户线程管理下，可以让应用程序开发人员开发多线程应用软件的执行效率更高。

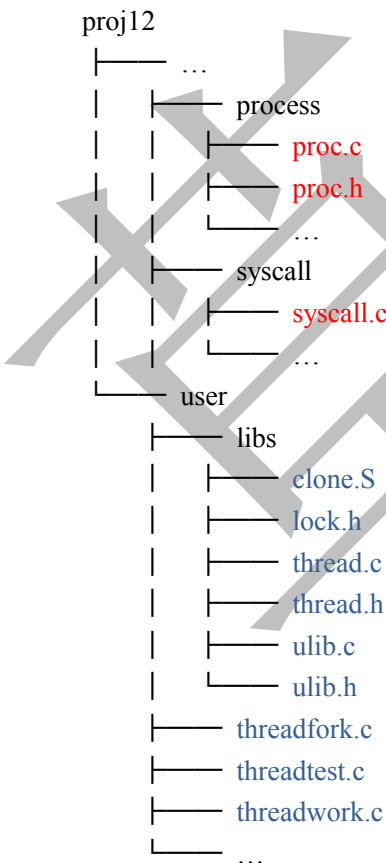
为了支持用户线程，我们还需对现有的进程管理进行有限扩展，在了解线程基本原理的情况下，设计并实现 ucore 对用户线程的基本支持。

### 4.4.2 proj12 概述

#### 1. 实现描述

proj12 是 lab3 的最后一个 project。它在 proj10.1（中间还有 proj10.2/10.3/10.4/11）的基础上实现了对用户线程的支持，主要参考了 Linux 的线程实现思路，把线程作为一个共享内存等资源的轻量级进程看待，扩展设计了进程控制块中支持用户线程的成员变量和与进程管理相关的系统调用，使得在现有进程管理的基础上，做相对较小的改动，就支持线程模型了。

#### 2. 项目组成



- 相对于 proj11，主要增加和扩展的文件如下：
- kern/proc[ch]: 扩展进程控制块，增加线程组成员变量，并扩展和增加与线程管理相关的函数；
  - kern/syscall.c: 增加用于线程创建的 sys\_clone 系统调用；

- user/libs/\*.chS: 实现用户态创建线程的库调用函数、访问系统调用函数;
- user/thread\*.c: 线程支持用户测试用例。

### 3. 编译运行

首先确保 proj12 的 kern/proc.c 中的 user\_main 函数中的代码为:

```
static int
user_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(threadtest);
#endif
    panic("user_main execve failed.\n");
}
```

编译并运行 proj12 的命令如下:

```
make
```

```
make qemu
```

则可以得到如下显示界面:

```
thuos:~/oscourse/ucore/i386/lab3_process/proj12$ make qemu
(THU.CST) os is loading ...
.....
++ setup timer interrupts
kernel_execve: pid = 3, name = "threadtest".
thread ok.
child ok.
threadtest pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:454:
    initproc exit.
```

```
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

这其实是 ucore 首先创建了一个用户进程 usermain, 然后此用户进程通过调用 sys\_execv 执行的是 users/threadtest.c 中的代码:

```
#include <ulib.h>
#include <stdio.h>
#include <thread.h>
```

```

int
test(void *arg) {
    cprintf("child ok.\n");
    return 0xbee;
}

int
main(void) {
    thread_t tid;
    assert(thread(test, NULL, &tid) == 0);
    cprintf("thread ok.\n");

    int exit_code;
    assert(thread_wait(&tid, &exit_code) == 0 && exit_code == 0xbee);

    cprintf("threadtest pass.\n");
    return 0;
}

```

usermain 用户进程调用了 thread 用户库函数来创建了一个用户线程 test，这个用户线程同享了 usermain 用户进程的地址空间，然后 usermain 用户进程就调用 thread\_wait 函数等待用户线程结束。用户线程 test 在结束执行时，会设置退出码为 0xbee。用户进程 usermain 会检查此退出码，看用户线程是否结束。上述执行过程其实包含了对用户线程整个生命周期的管理。下面我们将从原理和实现两个方面对此进行进一步阐述。

#### 4.4.3 【原理】线程的属性与特征分析

线程概念的提出是计算机系统的技术发展和操作系统对进程管理优化过程的自然产物。随着计算机处理能力的提高，内存容量的加大，在一个计算机系统中会存在大量的进程，为了提高整个系统的执行效率，需要进程间能够进行简洁高效的数据共享和进程切换，进程创建和进程退出。这样就会产生一个自然的想法：能否改进进程模型，提供一个简单的数据共享机制，能否加快进程管理（特别是进程切换）的速度？再仔细看看进程管理的核心数据结构进程控制块和处理的流程，就可以发现，在某种情况下，进程的地址空间隔离不一定是一个必须的需求。假定进程间是可以相互“信任”的（即进程间是可信的），那么我们就不需要地址空间隔离，而是让这些相互信任的进程共用（共享）一个地址空间。这样一下子就解决上述两个问题：在一个地址空间内，一个进程对某内存单元的修改（即写内存单元操作）可以让其他进程马上看见（即读内存单元操作），这是共享地址空间带来的天然好处；另外由于进程共享了地址空间，所以在创建进程或回收进程的时候，只要不是相互信任的进程组的第一个或最后一个，就没必要再创建一个地址空间或回收地址空间，节省了创建进程和退出进程的执行开销；而且对于频繁发生的进程切换操作而言，由于不需要切换页表，所以 TLB 中缓存的虚拟地址—物理地址映射关系（页表项的缓存）不必清除或失效，减少了进程切换的开销。

为了区别已有的进程概念，我们把这些共享资源的进程称为线程。从而把进程的概念进行了细化。原有的进程概念缩小为资源管理的最小单位，而线程是指令执行控制流的最小单位。且线程属于进程，是进程的一部分。一个进程至少需要一个线程作为它的指令执行单元，



进程管理主要是资源（如内存空间等）分配、使用和回收的管理，而线程管理主要是指令执行过程和切换过程的管理。一个进程可以拥有多个线程，而这些线程共享其所属进程所拥有的资源。这样，采用多线程模型来设计应用程序，可以使得程序的执行效率也更高。

从执行线程调度时 CPU 所处特权级角度看，有内核级线程模型和用户级线程模型两种线程模型。内核级线程模型由操作系统在核心态进行调度，每个线程有对应的进程控制块结构。用户级线程模型有用户态的线程管理库在用户态进行调度，操作系统不能“感知”到这样的线程存在，所以也就没有对应进程控制块来描述它。相对而言，由于用户级线程模型在执行线程调度切换时不需从用户态转入核心态，开销相对较小，而内核级线程模型虽然开销相对较大，但由于操作系统直接负责管理，所以在执行的灵活性上由于用户级线程模型。比如用户级线程模型中某线程执行系统调用而可能被操作系统阻塞，这会引起同属于一个进程的其他线程都被阻塞。但内核级线程模型不会有这种情况后发生。这两种线程模型还可以结合在一起形成一种“混合”线程模型。“混合”线程模型通常都能带来更高的效率，但也带来更大的实现难度和实现代价。ucore 出于“简单”的设计思路，参考 Linux 实现了内核级线程模型。

从线程执行时 CPU 所处特权级角度看，有内核线程和用户线程之分，内核线程共享操作系统内核运行过程中的所有资源，最主要的就是内核虚拟地址空间，但没有内核线程有各自独立的核心栈，且没有用户态的地址空间。用户线程共享属于同一用户进程的所有资源，最主要的就是用户虚拟地址空间，所以同享同一用户进程的进程控制块所描述的一个页表和一个内存管理数据结构。接下来我们看看 ucore 中如何具体实现用户线程这个概念。

#### 4.4.4 【实现】创建并执行用户线程

### 1. 数据结构扩展：进程控制块和用户线程数据结构

由于 ucore 采用的是内核级线程模型，所以一个用户线程有一个进程控制块，但由于用户线程不同于用户进程，所以要对已有进程控制块进行简单扩展，这个扩展其实就是要表达共享了同一进程的线程组的关系：

```
struct proc_struct {  
    // the threads list including this proc which share resource  
    list_entry_t thread_group;  
};
```

这样便于查找属于同一用户进程的所有用户线程。另外，属于同一用户进程的用户线程能够共享的进程控制块的主要成员变量包括：

```
struct mm_struct *mm; // Process's memory management field  
uintptr_t cr3; // CR3 register: the base addr of Page Directroy Table(PDT  
)
```

这样确保了属于同一用户进程的用户线程能共享同一用户地址空间。在对于其他一些与线程执行相关的成员变量，比如 state、kstack、tf 等，都有各自独立的数据存在。由于属于同一用户进程的不同用户线程除了在内核地址空间需要有不同的内核栈空间外，在用户地址空间也需要有不同的用户栈空间。为此，我们还需有一个数据结构来描述。ucore 把这个数据机构放在用户态函数库中（user/libs/thread.h）：

```

typedef struct {
    int pid;
    void *stack;
} thread_t;

```

`thread_t` 是对进程控制块的补充，`pid` 是此线程的 `id` 标识（与进程控制块的 `pid` 一致），`stack` 是用户栈的起始地址。通过这两方面的扩展，在数据结构层面就已经做好对用户线程的支持了。

## 2. 创建用户线程

创建用户线程的执行流程重用了创建用户进程的执行过程，但有一些微小的差别，下面我们逐一进行分析。用户态函数库提供了 `thread` 函数来给应用程序提供创建线程的接口。

```

int
thread(int (*fn)(void *), void *arg, thread_t *tidp) {
    if (fn == NULL || tidp == NULL) {
        return -EINVAL;
    }
    int ret;
    uintptr_t stack = 0;
    if ((ret = mmap(&stack, THREAD_STACKSIZE, MMAP_WRITE | MMAP_READ, 0, 0, 0)) != 0) {
        return ret;
    }
    assert(stack != 0);

    if ((ret = clone(CLONE_VM | CLONE_THREAD, stack + THREAD_STACKSIZE, fn, arg)) < 0) {
        munmap(stack, THREAD_STACKSIZE);
        return ret;
    }

    tidp->pid = ret;
    tidp->stack = (void *)stack;
    return 0;
}

```

从中可以看出，`thread` 函数首先需要给用户线程分配用户栈，这里采用的是 `mmap` 函数（最终访问 `sys_mmap` 系统调用）来完成栈空间分配。其实 `ucore` 并没有真的给用户线程分配栈空间，这里采用了 **Demanding Paging**（按需分页）技术来减少分配栈空间的时间和空间开销。另外，还调用了 `clone` 函数来完成用户线程的创建，`clone` 进一步调用了 `sys_clone` 系统调用接口来要求 `ucore` 完成创建线程的服务。`sys_clone` 与创建进程的 `sys_fork` 系统调用接口有何区别？它们的区别是创建标志位 `clone_flags`：

- 调用 `clone` 创建线程的创建标志位：`CLONE_VM | CLONE_THREAD`

- 调用 fork 创建进程的创建标志位 : 0

【注意】另外 clone 函数要完成的具体工作是放在/usr/libs/clone.S 中的函数\_\_clone 来实现的。为何\_\_clone 函数的内容不直接放到 clone 函数中用 C 语言来实现呢？这里其实只能用汇编来实现。因为对于用户进程而言，采用 fork 函数创建用户进程时，对子进程用户空间的设置采用的是整体复制或基于 COW 机制的整体共享方式，这样就可保证不同进程的堆和栈最终位于不同的用户地址空间。而对于用户线程而言，它需要共享父进程的地址空间，但又需要有自己的栈空间（此空间虽然父进程也可以“看到”，但不能用，只能归线程自己专用），所以在访问 sys\_clone 系统调用前，用户线程用的是父进程的用户堆栈，在此系统调用返回后，用户线程已经被创建并开始为用户态执行，此时 sp 已经指向了新进程建立的新用户栈（见 usr/libs/thread.c），而不是父进程的用户栈，这也就意味着在从系统调用返回后，子进程已经无法读位于父进程栈上的局部变量等数据。这样新线程需要使用寄存器（其实全局变量也行，只要避免使用位于进程栈上的局部变量即可）来进行数据处理，完成用户线程的继续执行。为了能够精确控制这个执行过程，不得不采用汇编来写，如果用 C 语言来写，你无法确保编译器不创建或使用局部变量来完成处理过程。

在内核中，创建线程的服务和创建进程的服务都是通过 do\_fork 来实现的，而这个创建标志位 clone\_flags 就决定了如何创建线程，查看 do\_fork 函数：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    .....
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    .....
    {
        .....
        if (clone_flags & CLONE_THREAD) {
            list_add_before(&(current->thread_group), &(proc->thread_group));
        }
    }
    .....
}
```

如果进一步分析 copy\_mm 函数，如果 clone\_flags 有 CLONE\_VM 位，则执行：

```
.....
    mm_count_inc(mm);
    proc->mm = mm;
    proc->cr3 = PADDR(mm->pgdir);
    .....
```

后两条语句说明了被创建的新线程重用了当前进程控制块的 mm 成员变量，并重用同样的页表。而如果 clone\_flags 有 CLONE\_THREAD 位，则会把被创建的新线程的进程控制块成员变量 thread\_group 链接到当前进程控制块的牵头的链表 thread\_group 中，这样所有当前进程创建的线程都会在当前进程牵头的链表 thread\_group 中找到。而

在其他操作中，创建线程与创建进程的处理逻辑是一样的。

### 3. 退出用户线程

在退出线程方面，与进程退出相比，区别不大。只是在执行 `do_exit` 的时候，调用 `de_thread(current)` 函数，进一步把自身从线程组链表中删除，其他方面都与进程执行 `do_exit` 一致；而父进程执行用户库函数 `thread_wait` 来通过 `sys_wait` 系统调用接口进一步调用内核函数 `do_exit` 完成对退出的子进程资源的最后回收。与进程的 `wait` 函数相比，主要的区别是：用户库函数 `thread_wait` 通过 `munmap` 函数完成了对线程用户栈空间的回收。与 `proj12` 以前实现的 `do_exit` 函数相比，主要的区别是：在查找是否有子线程的执行状态处于 `PROC_ZOMBIE` 时，除了搜索此进程的每个子进程外，还搜索此进程所属线程组的每个线程的所有子进程。其他方面的处理都是大致相同的。

#### 4.4.5 进程运行状态转变过程

分析完从进程/线程从创建到退出的整个过程，我们需要从全局的角度来看看进程/线程在做整个运行过程中的运行状态转变过程。在执行状态转变过程中，`ucore` 在调度过程总，并没有区分线程和进程，所以进程和线程的执行状态转变是一致的，分析的结果适合用户线程和用户进程的执行过程。

首先为了描述进程/线程的整个状态集合，`ucore` 在 `kern/process/proc.h` 中定义了进程/线程的运行状态：

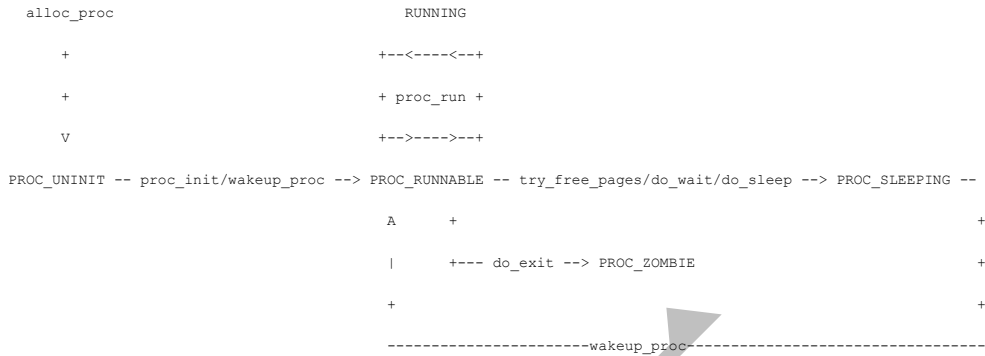
```
// process's state in his life cycle
enum proc_state {
    PROC_UNINIT = 0, // uninitialized
    PROC_SLEEPING, // sleeping
    PROC_RUNNABLE, // runnable(maybe running)
    PROC_ZOMBIE, // almost dead, and wait parent proc to
    reclaim his resource
};
```

这与操作系统原理讲解的五进程执行状态相比，少了一个 `PROC_RUNNING` 态（表示正在占用 CPU 执行），这是由于在 `ucore` 中，用 `current`（基于 `proc_struct` 数据结构）进程控制块指针指向了当前正在运行的进程/线程 `PROC_RUNNING` 态，所以就没必要再增加一个 `PROC_RUNNING` 态了。那么那些事件或内核函数会触发状态的转变呢？通过分析 `ucore` 源码，我们可以得到如下表示：

| 进程状态                       | 转变原因                                                                                 |
|----------------------------|--------------------------------------------------------------------------------------|
| <code>PROC_UNINIT</code>   | 执行 <code>alloc_proc</code> 函数                                                        |
| <code>PROC_SLEEPING</code> | 执行 <code>try_free_pages</code> , <code>do_wait</code> , <code>do_sleep</code> 函数     |
| <code>PROC_RUNNABLE</code> | 执行 <code>proc_init</code> , <code>wakeup_proc</code> 函数，有此状态的进程可处于就绪准备状态或占用 CPU 运行状态 |
| <code>PROC_ZOMBIE</code>   | 执行 <code>do_exit</code> 函数                                                           |

当父进程得到子进程的通知，回收完子进程控制块所占内存后，这个进程就彻底消失了。我们也可以用类似有限状态自动机来表示状态的变化：（需要用 visio 重画）

process state changing:



## 4.5 进程调度

### 4.5.1 实验目标

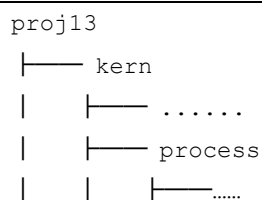
在只有一个或几个 CPU 的计算机系统中，进程数量一般远大于 CPU 数量，CPU 是一个稀缺资源，多个进程不得分时占用 CPU 执行各自的工作，操作系统必须提供一种手段来保证各个进程“和谐”地共享 CPU。为此，需要在 lab3 的基础上设计实现 ucore 进程调度类框架以便于设计各种进程调度算法，同时以提高计算机整体效率和尽量保证各个进程“公平”地执行作为目标来设计各种进程调度算法。

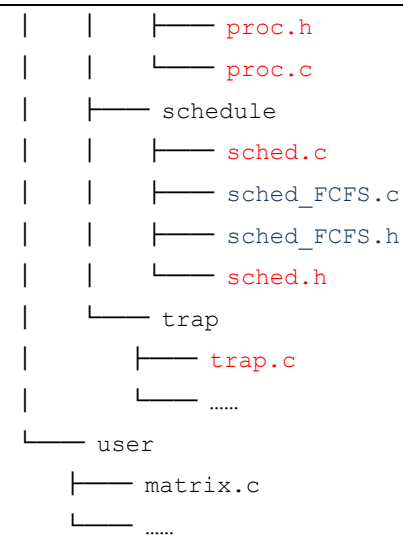
### 4.5.2 proj13/13.1/13.2 概述

#### 1. 实现描述

project13 是 lab4 的第一个项目，它基于 lab3 的最后一个项目 proj12。主要参考了 Linux-2.6.23 设计了一个简化的进程调度类框架，能够在此框架下实现不同的调度算法，并在此框架下实现了一个简单的 FIFO 调度算法。接下来的 proj13.1 在此调度框架下实现了轮转（Round Robin，简称 RR）调度算法，proj13.2 在此调度框架下实现了多级反馈队列（Multi-Level Feed Back，简称 MLFB）调度算法。

#### 2. 项目组成





相对与 proj12, proj13 增加了 3 个文件, 修改了相对重要的 5 个文件。主要修改和增加的文件如下:

- `process/proc.[ch]`: 扩展了进程控制块的定义能够把处于就绪态的进程放入到一个就绪队列中，并在创建进程时对新扩展的成员变量进行初始化。
- `schedule/sched.[ch]`: 增加进程调度类的定义和相关进程调度类的共性函数。
- `schedule/sched_FCFS.[ch]`: 基于进程调度类的 FCFS 调度算法实例的设计实现；
- `schedule/sched.[ch]`: 实现了一个先来先服务（First Come First Serve）策略的进程调度。
- `user/matrix.c`: 矩阵乘用户测试程序

### 3. 编译运行

编译并运行 proj13 的命令如下:

make

make qemu

则可以得到如下显示界面

(THU.CST) os is loading ...

Special kernel symbols:

• • • • •

```
++ setup timer interrupts
```

```
kernel execve: pid = 3, name = "matrix".
```

fork ok.

pid 4 is running (1000 times)!.

pid 4 done!.

```
pid 5 is running (1400 times)!.
pid 5 done!.
.....
pid 22 is running (33400 times)!.
pid 22 done!.
pid 23 is running (33400 times)!.
pid 23 done!.
matrix pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:456:
    initproc exit.
```

```
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

这其实是在采用简单的 FCFS 调度方法来执行 `matrix` 用户进程，这个 `matrix` 进程将创建 20 个进程来各自执行二维矩阵乘的工作。`Ucore` 将按照 FCFS 的调度方法，一个一个地按创建顺序执行每个子进程。一个子进程结束后，再调度另外一个子进程运行。这实际上看不出 `ucore` 是如何具体实现进程调度类框架和 FCFS 调度算法的。下面我们首先介绍一下进程调度的基本原理，然后再分析 `ucore` 的调度框架和调度算法的实现。

### 4.5.3 【原理】进程调度

需要进程调度的理由很简单，即充分利用计算机系统中的 CPU 资源，让计算机系统能够多快好省地完成我们让它做的各种任务。为此，可在内存中可存放数目远大于计算机系统内 CPU 个数的进程，让这些进程在操作系统的进程调度器的调度下，能够让进程高效（高的吞吐量--throughput）、及时（低延迟--latency）、公平（fairness）地使用 CPU。为此调度器可设计不同的调度算法来选择进程，这体现了进程调度的策略，同时还需并进一步通过进程的上下文切换（context switch）来完成进程切换，这体现了进程调度的机制。总体上说，我们需要何时调度（调度的时机）、是否能够在内核执行的任意位置进行调度（调度的方式）、如果完成进程切换（上下文切换）、如果选择“合适”的进程执行（调度策略/调度算法）、如果评价选择的合理性（进程调度的指标）。了解上述细节，也就可以说是了解了进程调度。

## 1. 进程调度的指标

不同的进程调度算法具有不同的特征，为此需要建立衡量一个算法的基本指标。一般而言，衡量和比较各种进程调度算法性能的主要因素如下所示：

- CPU 利用率：CPU 是计算机系统中的稀缺资源，所以应在有具体任务的情况下尽可能使 CPU 保持忙，从而使得 CPU 资源利用率最高。
- 吞吐量：CPU 运行时的工作量大小是以每单位时间所完成的进程数目来描述的，即称为吞吐量。
- 周转时间：指从进程创建到作进程结束所经过的时间，这期间包括了由于各种因素（比如等待 I/O 操作完成）导致的进程阻塞，处于就绪态并在就绪队列中排队，在处理机上

---

运行所花时间的总和。

- 等待时间：即进程在就绪队列中等待所花的时间总和。因此衡量一个调度算法的简单方法就是统计进程在就绪队列上的等待时间。
- 响应时间：指从事件（比如产生了一次时钟中断事件）产生到进程或系统作出响应所经过的时间。在交互式桌面计算机系统中，用户希望响应时间越快越好，但这常常要以牺牲吞吐量为代价。

这些指标其实是相互有冲突的，响应时间短也就意味着在相关事件产生后，操作系统需要迅速进行进程切换，让对应的进程尽快响应产生的事件，从而导致进程调度与切换的开销增大，这会降低系统的吞吐量。

## 2. 进程调度的时机

进程调度发生的时机（也称为调度点）与进程的状态变化有直接的关系。回顾进程状态变化图，我们可以看到进程调度的时机直接与进程在运行态<-->退出态/就绪态/阻塞态的转变时机相关。简而言之，引起进程调度的时机可归结为以下几类：

- 正在执行的进程执行完毕，需要选择新的就绪进程执行。
- 正在执行的进程调用相关系统调用（包括与 I/O 操作，同步互斥操作等相关的系统调用）导致需等待某事件发生或等待资源可用，从而将自己阻塞起来进入阻塞状态。
- 正在执行的进程主动调用放弃 CPU 的系统调用，导致自己的状态为就绪态，且把自己重新放到就绪队列中。
- 等待事件发生或资源可用的进程等待队列，从而导致进程从阻塞态回到就绪态，并可参与到调度中。
- 正在执行的进程的时间片已经用完，将自己的状态为就绪态，且把自己重新放到就绪队列中。
- 在执行完系统调用后准备返回用户进程前的时刻，可调度选择一新用户进程执行
- 就绪队列中某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

## 3. 进程调度的方式

这里需要注意，存在两种进程抢占处理器的调度方式：

- 可抢占式（可剥夺式，preemptive）：就绪队列中一旦有某进程的优先级高于当前正在执行的进程的优先级时，操作系统便立即进行进程调度，完成进程切换。
- 不可抢占式（不可剥夺式 non\_preemptive）：即使在就绪队列存在有某进程优先级高于当前正在执行的进程的优先级时，当前进程仍将占用处理机执行，直到该进程自己进入阻塞状态，或时间片用完，或在执行完系统调用后准备返回用户进程前的时刻，才重新发生调度让出处理机。

显然，可抢占式调度可有效减少等待时间和响应时间，但会带来较大的其他管理开销，使得吞吐量等的性能指标比不可抢占式调度要低。所以一般在桌面计算机中都支持可抢占式调度，使得用户可以得到更好的人机交互体验，而在服务器领域不必非要可抢占式调度，而通常会采用不可抢占式调度，从而可提高系统的整体吞吐量。



## 4. 进程调度的策略/算法

在早期操作系统的调度方式大多数是非剥夺的,这是由于早期的应用一般是科学计算或事务处理,不太把人机交互的响应时间指标放在首要位置。在这种情况下,正在运行的进程可一直占用 CPU 直到进程阻塞或终止。这种方式的调度算法可以很简单,且比较适用对于响应时间不关心或者关心甚少的批处理科学计算或事务处理应用。随着计算机的应用领域进一步扩展,计算机更多地用在了多媒体等人机交互应用上,为此采用可抢占式的调度方式可在一个进程终止或阻塞之前就剥夺其执行权,把 CPU 尽快分配给另外的“更重要”进程,使得就绪队列中的进程有机会响应它们用户的 IO 事件。基于这两种方式的调度算法如下:

- 先来先服务(FCFS)调度算法:处于就绪态的进程按先后顺序链入到就绪队列中,而 FCFS 调度算法按就绪进程进入就绪队列的先后次序选择当前最先进入就绪队列的进程来执行,直到此进程阻塞或结束,才进行下一次的进程选择调度。FCFS 调度算法采用的是不可抢占的调度方式,一旦一个进程占有处理机,就一直运行下去,直到该进程完成其工作,或因等待某一事件而不能继续执行时,才释放处理机。操作系统如果采用这种进程调度方式,则一个运行时间长且正在运行的进程会使很多晚到的且运行时间短的进程的等待时间过长。
- 短作业优先(SJF)调度算法:其实目前作业的提法越来越少,我们姑且把“作业”用“进程”来替换,改称为短进程优先调度算法,此算法选择就绪队列中确切(或估计)运行时间最短的进程进入执行。它既可采用可抢占调度方式,也可采用不可抢占调度方式。可抢占的短进程优先调度算法通常也叫做最短剩余时间优先(Shortest Remaining Time First, SRTF)调度算法。短进程优先调度算法能有效地缩短进程的平均周转时间,提高系统的吞吐量,但不利于长进程的运行。而且如果进程的运行时间是“估计”出来的话,会导致由于估计的运行时间不一定准确,而不能实际做到短作业优先。
- 时间片轮转(RR)调度算法:RR 调度算法与 FCFS 调度算法在选择进程上类似,但在调度的时机选择上不同。RR 调度算法定义了一个的时间单元,称为时间片(或时间量)。一个时间片通常在 1~100 ms 之间。当正在运行的进程用完了时间片后,即使此进程还要运行,操作系统也不让它继续运行,而是从就绪队列依次选择下一个处于就绪态的进程执行,而被剥夺 CPU 使用的进程返回到就绪队列的末尾,等待再次被调度。时间片的大小可调整,如果时间片大到让一个进程足以完成其全部工作,这种算法就退化为 FCFS 调度算法;若时间片设置得很小,那么处理机在进程之间的进程上下文切换工作过于频繁,使得真正用于运行用户程序的时间减少。时间片可以静态设置好,也可根据系统当前负载状况和运行情况动态调整,时间片大小的动态调整需要考虑就绪态进程个数、进程上下文切换开销、系统吞吐量、系统响应时间等多方面因素。
- 高响应比优先(Highest Response Ratio First, HRRF)调度算法:HRRF 调度算法是介于先来先服务算法与最短进程优先算法之间的一种折中算法。先来先服务算法只考虑进程的等待时间而忽视了进程的执行时间,而最短进程优先调度算法只考虑用户估计的进程的执行时间而忽视了就绪进程的等待时间。HRRF 调度算法二者兼顾,既考虑进程等待时间,又考虑进程的执行时间,为此定义了响应比( $R_p$ )这个指标:

$$R_p = (\text{等待时间} + \text{预计执行时间}) / \text{执行时间} = \text{响应时间} / \text{执行时间}$$

上个表达式假设等待时间与预计执行时间之和等于响应时间。HRRF 调度算法将选

择  $R_p$  最大值的进程执行，这样既照顾了短进程又不使长进程的等待时间过长，改进了调度性能。但 HRRF 调度算法需要每次计算各个进程的响应比  $R_p$ ，这会带来较大的时间开销（特别是在就绪进程个数多的情况下）。

- 多级反馈队列（Multi-Level Feedback Queue）调度算法：在采用多级反馈队列调度算法的执行逻辑流程如下：

- 设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二队次之，其余队列优先级依次降低。仅当第  $1 \sim i-1$  个队列均为空时，操作系统调度器才会调度第  $i$  个队列中的进程运行。赋予各个队列中进程执行时间片的大小也各不相同。在优先级越高的队列中，每个进程的执行时间片就越小或越大（Linux-2.4 内核就是采用这种方式）。
- 当一个就绪进程需要链入就绪队列时，操作系统首先将它放入第一队列的末尾，按 FCFS 的原则排队等待调度。若轮到该进程执行且在一个时间片结束时尚未完成，则操作系统调度器便将该进程转入第二队列的末尾，再同样按先来先服务原则等待调度执行。如此下去，当一个长进程从第一队列降到最后一个队列后，在最后一个队列中，可使用 FCFS 或 RR 调度算法来运行处于此队列中的进程。
- 如果处理机正在第  $i(i>1)$  队列中为某进程服务时，又有新进程进入第  $k(k<i)$  的队列，则新进程将抢占正在运行进程的处理机，即由调度程序把正在执行进程放回第  $i$  队列末尾，重新将处理机分配给处于第  $k$  队列的新进程。

从 MLFQ 调度算法可以看出长进程无法长期占用处理机，且系统的响应时间会缩短，吞吐量也不错（前提是没有频繁的短进程）。所以 MLFQ 调度算法是一种合适不同类型应用特征的综合进程调度算法。

- 最高优先级优先调度算法：进程的优先级用于表示进程的重要性及运行的优先性。一个进程的优先级可分为两种：静态优先级和动态优先级。静态优先级是在创建进程时确定的。一旦确定后，在整个进程运行期间不再改变。静态优先级一般由用户依据包括进程的类型、进程所使用的资源、进程的估计运行时间等因素来设置。一般而言，若进程需要的资源越多、估计运行的时间越长，则进程的优先级越低；反之，对于 I/O bounded 的进程可以把优先级设置得高。动态优先级是指在进程运行过程中，根据进程执行情况的变化来调整优先级。动态优先级一般根据进程占有 CPU 时间的长短、进程等待 CPU 时间的长短等因素确定。占有处理机的时间越长，则优先级越低，等待时间越长，优先级越高。那么进程调度器将根据静态优先级和动态优先级的总和现在优先级最高的就绪进程执行。

操作系统中为了能够让每个进程都有机会运行，需要给每个进程分配一个时间片，当一个进程的时间片用完以后，操作系统的调度器就会让当前进程放弃 CPU，而选择另外一个进程占用 CPU 执行。为了有效地支持进程调度所需的时间片，ucore 设计并实现了一个 timer（计时器）功能。这样，通过 timer 就对基于时间事件的调度机制提供了基本支持。

## 4.5.4 【实现】进程调度

### 1. 内核的抢占性

调度本质上体现了对 CPU 资源的抢占。对于用户进程而言，由于有中断的产生，可以随

时打断用户进程的执行，转到操作系统内部，从而给了操作系统以调度控制权，让操作系统可以根据具体情况（比如用户进程时间片已经用完了）选择其他用户进程执行。这体现了用户进程的可抢占性（preemptive）。但如果把 ucore 操作系统也看成是一个特殊的内核进程或多个内核线程的集合，那 ucore 是否也是可抢占的呢？其实 ucore 内核执行是不可抢占的（non-preemptive），即在执行“任意”内核代码时，CPU 控制权可被强制剥夺。这里需要注意，不是在所有情况下 ucore 内核执行都是不可抢占的，有以下几种“固定”情况是例外：

- 1. 进行同步互斥操作，比如争抢一个信号量、锁（lab5 中会详细分析）；
- 2. 进行磁盘读写等耗时的异步操作，由于等待完成的耗时太长，ucore 会调用 shcedule 让其他就绪进程执行。
- 3.

这几种情况其实都是由于当前进程所需的某个资源（也可称为事件）无法得到满足，无法继续执行下去，从而不得不主动放弃对 CPU 的控制权。如果参照用户进程任何位置都可被内核打断并放弃 CPU 控制权的情况，这些在内核中放弃 CPU 控制权的执行地点是“固定”而不是“任意”的，不能体现内核任意位置都可抢占性的特点。我们搜寻一下 proj13 的代码，可发现在如下几处地方调用了 shcedule 函数：

调用进程调度函数 schedule 的位置和原因

| 编号 | 位置                    | 原因                                                                                                             |
|----|-----------------------|----------------------------------------------------------------------------------------------------------------|
| 1  | swap.c::try_free_page | 用户进程申请内存无法得到满足，主动放弃对 CPU 控制权，等待 kswapd 内核线程释放出足够多的空闲空间。                                                        |
| 2  | proc.c::do_exit       | 用户线程执行结束，主动放弃 CPU 控制权。                                                                                         |
| 3  | proc.c::do_wait       | 用户线程等待子进程结束，主动放弃 CPU 控制权。                                                                                      |
| 4  | proc.c::do_sleep      | 用户线程要睡眠一段时间，主动放弃 CPU 控制权。                                                                                      |
| 5  | proc.c::init_main     | 1. initproc 内核线程等待所有用户进程结束，如果没有结束，就主动放弃 CPU 控制权；<br>2. initproc 内核线程在所有用户进程结束后，让 kswapd 内核线程执行 10 次，用于回收空闲内存资源 |
| 6  | proc.c::cpu_idle      | idleproc 内核线程的工作就是等待有处于就绪态的进程或线程，如果有就调用 schedule 函数                                                            |
| 7  | sync.h::lock          | 在获取锁的过程中，如果无法得到锁，则主动放弃 CPU 控制权                                                                                 |
| 8  | trap.c::trap          | 如果在当前进程在用户态被打断去，且当前进程控制块的成员变量 need_resched 设置为 1，则当前线程会放弃 CPU 控制权                                              |

仔细分析上述位置，第 1、2、3、4、7 处的执行位置体现了由于访问磁盘数据的资源或获取锁资源一时等不到满足、进程要退出、进程要睡眠等原因而不得不主动放弃 CPU。第 5、6 处的执行位置比较特殊，initproc 内核线程等待用户进程结束和执行 kwapd 内核线程而执行 schedule 函数；idle 内核线程在没有进程处于就绪态时才执行，一旦有了就绪态的进程，它将执行 schedule 函数完成进程调度。这里只有第 8 处的位置比较特殊：

```
if (!in_kernel) {  
.....
```

```

        if (current->need_resched) {
            schedule();
        }
    }
}

```

这里表明了只有当进程在用户态执行到“任意”某处用户代码位置时发生了中断，且当前进程控制块成员变量 `need_resched` 为 1（表示需要调度了）时，才会执行 `shedule` 函数。这实际上体现了对用户进程的可抢占性。如果没有第一行的 `if` 语句，那么就可以体现对内核代码的可抢占性。但如果要把这一行 `if` 语句去掉，我们不得不实现对 `ucore` 中的所有全局变量的互斥访问操作，以防止所谓的 `race condition` 现象（在 lab5 中有进一步讲述），这样 `ucore` 的实现复杂度会增加不少。

## 2. 进程调度时机

我们可以知道 `ucore` 通过进程控制块的 `state` 成员变量来表示进程的运行状态，而且处于就绪态的进程和运行态的进程在 `ucore` 中的运行状态都是 `PROC_RUNNABLE`。那如何进一步区分处于就绪态的进程和运行态的进程呢？在 `ucore` 中，所有处于就绪态的进程会链接在一个就绪队列 `rq`（`run queue` 的简称），而处于运行态的进程实际上市从就绪队列 `rq` 中被进程调度器选择出来的一个进程，此进程会从此就绪队列中断开，并开始占用 CPU 执行。这说明，没有挂在就绪队列且运行状态为 `PROC_RUNNABLE` 的进程是处于运行态的进程。

根据进程状态变化过程的分析，我们可以知道选择一个进程，把它从就绪态转变到运行态是进程调度器的主要工作。在 `ucore` 内核中哪些地方会进行调度呢？这其实与进程的运行状态变化的时机相关，回想 4.4.5 小节描述的进程状态变化过程，可以知道在如下一些地方是需要进行进程调度的。这里需要更深入地分析从运行态到其他状态相关转换的情况

### 运行态到就绪态的变化过程

首先来看运行态到就绪态的变化过程。进程主动/被动放弃 CPU 回到就绪态的起因有两个。一个起因是用户进程主动从运行态回到就绪态，即用户进程调用 `yield` 用户态库函数，此库函数接下来的调用路径如下：

`yield(用户态)` → `sys_yield(用户态)` → `sys_yield(内核态)` → `do_yield(内核态)`  
`do_yield` 内核函数仅仅把当前进程的进程控制块成员变量 `need_resched` 设置为 1。

```

int
do_yield(void) {
    current->need_resched = 1;
    return 0;
}

```

另外一个起因是在进程调度器采用类似时间片轮转调度（RR）策略的前提下，用户进程由于时间片用完被动地从运行态回到就绪态的情况，即当时钟中断积累到一定的时间片段（给用户进程设置能够持续占用 CPU 运行的时间）后，如果一个用户进程还在持续运行，则 `ucore` 需要强制剥夺用户进程的 CPU 使用权，即把当前进程重新插入到就绪队列中，并从就绪队列中再选择另外一个“合适”的进程执行。时钟中断产生后，整个执行过程的函数调用路径如下：

`vetcor32` → `__alltraps` → `trap` → `trap_dispatch` → `run_timer_list`  
 → `sched_class_proc_tick` → `XXX_proc_tick`

注意 XXX\_proc\_tick 是某个具体进程调度器 XXX 对产生时钟中断后的特定处理函数。比如对于 FCFS 调度器而言，它没有考虑时间片轮转的情形，所以实际的 FCFS\_proc\_tick 函数啥也没干。但如果是 RR 调度器（在 lab4/proj13.1 中实现），这实际的 RR\_proc\_tick 则需要递减当前进程拥有的时间片，并判断当前进程的时间片是否已经用完，如果用完了，则需要把当前进程的进程控制块成员变量 need\_resched 设置为 1。

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc)
{
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

其实把当前进程的进程控制块成员变量 need\_resched 设置为 1 只是指出了需要调度，但并没有实现调度。那需要在哪里完成调度呢？注意到上一小节“内核的抢占性”中，我们可以看到，实际上 ucore 只是在中断或异常返回的时候，如果在用户态被中断的当前用户进程 need\_resched 设置为 1，则会执行 schedule 函数，完成进程调度与切换。

### 运行态到就睡眠态的变化过程

参考上表（调用进程调度函数 schedule 的位置和原因），我们可以看到有三个地方：第 1、3、4 处的执行代码会导致当前进程转变到睡眠态。第一个是由于当前进程申请内存无法得到满足，需通过执行 try\_free\_pages 函数，主动放弃对 CPU 控制权，等待 kswapd 内核线程释放出足够多的空闲空间。这里采用了等待队列的机制实现进程睡眠：

```
local_intr_save(intr_flag);
{
    wait_init(wait, current);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_KSWAPD;
    wait_queue_add(&kswapd_done, wait);
    if (kswapd->wait_state == WT_TIMER) {
        wakeup_proc(kswapd);
    }
}
local_intr_restore(intr_flag);
schedule();
```

上述代码很清楚地表明了当前进程的状态转变为睡眠状态，睡眠原因是 WT\_KSWAPD，即等待内核线程 kswapd 释放出更多的空闲内存，并执行 schedule 函数，把自身从就绪队列中删除，并选择新的就绪进程占用 CPU 执行。

第二个是 do\_wait 函数。用户进程执行 wait/waitpid 用户库函数，并进一步调用 sys\_wait 用户函数和 sys\_wait 内核函数后，将调用 do\_wait 函数完成实际的父进程等待子进程的工作。相关代码如下：

```
if (haskid) {
```

```

current->state = PROC_SLEEPING;
current->wait_state = WT_CHILD;
schedule();
.....

```

此函数判断如果当前进程有子进程，则会设置当前进程状态转变为睡眠状态，睡眠原因是 WT\_CHILD，即等待子进程结束，并执行 schedule 函数，把自身从就绪队列中删除，并选择新的就绪进程占用 CPU 执行。

第三个是 do\_sleep 函数。用户进程执行 sleep 用户库函数来实现 n 毫秒睡眠，这将进一步调用 sys\_sleep 用户函数和 sys\_sleep 内核函数后，最终调用 do\_sleep 完成实际的睡眠操作。相关代码如下：

```

do_sleep(unsigned int time) {
    if (time == 0) {
        return 0;
    }
    bool intr_flag;
    local_intr_save(intr_flag);
    timer_t __timer, *timer = timer_init(&__timer, current,
time);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_TIMER;
    add_timer(timer);
    local_intr_restore(intr_flag);

    schedule();
    .....
}

```

可以看出会设置当前进程状态转变为睡眠状态，睡眠原因是 WT\_TIMER，即等待定时器到时，并执行 schedule 函数，把自身从就绪队列中删除，并选择新的就绪进程占用 CPU 执行。这里采用了定时器机制（参见 4.2.5 小节），通过调用 timer\_init 函数来创建定时器，并调用 add\_timer 函数来设置定时器运转，当时间 time 到后，会唤醒当前进程。

### 运行态到就退出态的变化过程

当用户进程执行完毕（或者被要求强行退出）后，将执行 do\_exit 函数完成对自身所占部分资源的回收，并执行进程调度切换。参考上表（调用进程调度函数 schedule 的位置和原因），我们可以看到第 2 处的 do\_exit 函数的代码实现从运行态到退出态的变化过程，

```

current->state = PROC_ZOMBIE;
current->exit_code = error_code;
.....

schedule();

```

可以看出会设置当前进程状态转变为退出状态，并执行 schedule 函数，把自身从就绪队列中删除，并选择新的就绪进程占用 CPU 执行。



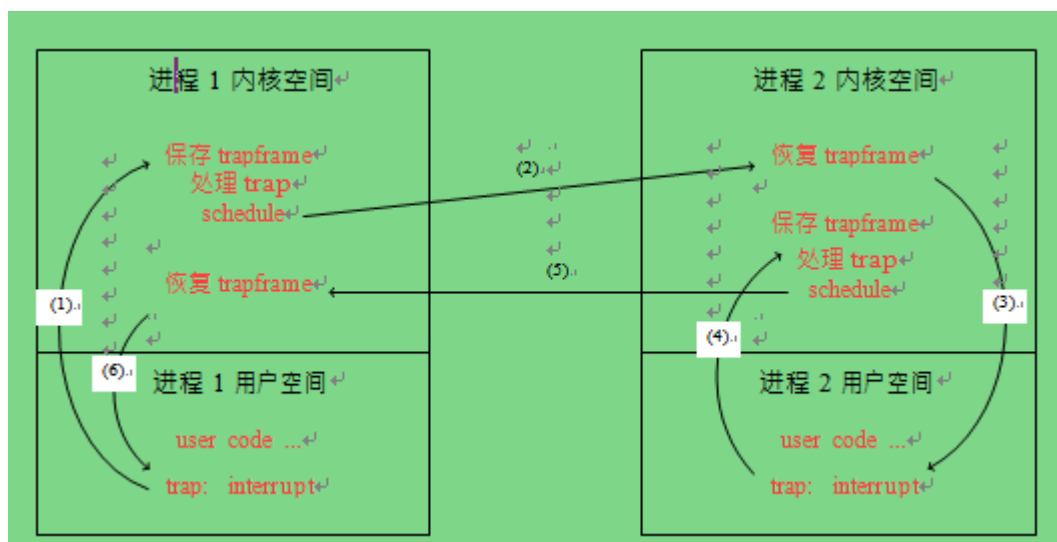
### 3. 进程切换过程

进程切换的具体过程发生在内核态,我们以对于基于时间片的进程调度过程来具体分析两个进程切换的过程。首先在执行进程 1 的用户代码时,出现了一个 trap (例如是一个 Timer 中断),这个时候就会从进程 1 的用户态切换到内核态(过程(1)),并且保存好进程 1 的 trapframe; 当 ucore 在处理中断时发现此进程的时间片已经用完,就会设置进程 1 的进程控制块的 need\_resched 为 1,表明需要进行进程调度,ucore 于是进一步执行 schedule 函数,并通过此函数把进程 1 重新放回就绪队列,选择了另一个进程—进程 2,并把进程 2 从就绪队列中摘除,这时需要调用 proc\_run 函数来具体完成进程切换了。我们来看看切换的过程:

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

从上面的代码可以看出,proc\_run 的函数参数 proc 是进程 2,而 current 是进程 1,首先调用 load\_esp0 需要设置进程 2 用户态返回到内核态的内核栈寄存器指针(即 TS 段的 ESP0);然后页表基址(CR3 寄存器的内容)设置为进程 2 的页表基址,虽然换了一个页表基址,但由于所有进程在内核中都用的是同一个内核虚拟地址空间,所以切换后只要在内核态执行,并访问内核地址空间,那实际上没啥变化,而只是在进程 2 返回到用户态的时候,所用的用户地址空间与进程 1 的用户地址空间就不一样了;最后一步是调用 switch\_to 函数切换进程运行相关的硬件寄存器内容(具体过程可回顾 4.1.5 节的第 3 小节“调度并执行内核线程 initproc”)。一旦执行完 switch\_to 的最后一个指令“RET”后,就彻底切换到进程 2 的执行环境了。

到了进程 2 的执行环境,首先继续执行进程 2 上一次在内核态的操作,并最终通过中断或异常的返回操作回到进程 2 的用户空间执行。类似上述描述过程,当进程 2 由于某种原因发生中断之后,并需要切换到进程 1;当再次切换到进程 1 时,会执行进程 1 上一次在内核调用 schedule (具体还要跟踪到 switch to 函数) 函数后的下一行代码,这行代码当然还是在进程 1 的上一次中断处理的位置处。最后当进程 1 的中断处理完毕的时候,执行权又会反交给进程 1 的用户代码。大致流程如下所示:



【问题】进程切换的工作可以在用户态实现吗？（提示，如果是用户线程，即执行环境不需要考虑页表和内核栈等，只在用户态执行，那就可以在用户态实现用户线程的切换，这就是通常用户态线程库的实现）

【问题】进程切换以后，当前进程是从哪里开始执行的？（提示，虽然还是同一个 `cpu`，但是此时使用的资源已经完全不同了。）

【问题】内核在第一个用户进程运行的时候，需要进行哪些操作？（提示，内核运行第一个用户进程的过程，实际上是从启动时的内核状态切换到该程序的内核状态的过程，而用户程序的起始状态的入口，即 `forkret` 等。）

## 4. 进程调度类框架设计

### 设计思路

进程调度类框架的设计是为了更好地实行各种进程调度策略或算法，为此需要了解为了实行一个进程调度策略，到底需要实现哪些基本功能对应的数据结构？首先考虑到一个无论哪种调度算法都需要选择一个就绪进程来占用 `CPU` 运行。为此我们可把就绪进程组织起来，可用队列（双向链表）、二叉树、红黑树、数组...等不同的组织方式。

在操作方面，如果需要选择一个就绪进程，就可以从基于某种组织方式的就绪进程集合中选择出一个进程执行。需要注意，这里“选择”和“出”是两个操作，选择是在集合中挑选一个“合适”的进程，“出”意味着离开就绪进程集合。另外考虑到一个处于运行态的进程还会由于某种原因（比如时间片用完了）回到就绪态而不能继续占用 `CPU` 执行，这就会重新进入到就绪进程集合中。这两种情况就形成了调度器相关的三个基本操作：在就绪进程集合中选择、进入就绪进程集合和离开就绪进程集合。这三个操作属于调度器的基本操作。

在进程的执行过程中，就绪进程的等待时间和执行进程的执行时间是影响调度选择的重要因素，这两个因素随着时间的流逝和各种事件的发生在不停地变化，比如处于就绪态的进程等待调度的时间在增长，处于运行态的进程所消耗的时间片在减少等。这些进程状态变化的情况需要及时让进程调度器知道，便于选择更合适的进程执行。所以这种进程变化的情况就形成了调度器相关的一个变化感知操作：`timer` 时间事件感知操作。这样在进程运行或等待的过程中，调度器可以调整进程控制块中与进程调度相关的属性值（比如消耗的时间片、进程优先级等），并可能导致对进程组织形式的调整（比如以时间片大小的顺序来重排双向



链表等), 并最终可能导致调选择新的进程占用 CPU 运行。这个操作属于调度器的进程调度属性调整操作。

## 数据结构和变量

在加上对进程调度相关所需的通用初始化操作, 就形成了进程调度类框架的 5 个指针函数, 每个具体的进程调度实例将分别实现这 5 个函数, 完成不同的进程调度策略/算法。具体而言, 在 ucore 中进程调度类定义如下:

```
struct sched_class {  
    // the name of sched_class  
    const char *name;  
    // Init the run queue  
    void (*init)(struct run_queue *rq);  
    // put the proc into runqueue, and this function must be called with rq_lock  
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);  
    // get the proc out runqueue, and this function must be called with rq_lock  
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);  
    // choose the next runnable task  
    struct proc_struct *(*pick_next)(struct run_queue *rq);  
    // dealer of the time-tick  
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);  
};
```

这其实是对 Linux 内核的调度类的一种简化设计, 需要注意这里的 run\_queue 是就绪进程集合的一种组织方式, 并不是特指队列 (queue), 我们根据调度算法的具体设计, 也可采用 Linux 的 CFS 调度器的就绪进程队列组织结构--红黑树。在 ucore 中, 不同调度算法需要设计不同的就绪进程队列组织结构。比如 lab4/proj13 实现的是 FCFS 调度算法, 所以就绪进程只需能够按创建顺序链接如一个双向链表即可, 为此设计的就绪进程队列组织结构就是一个双向链表:

```
struct run_queue {  
    list_entry_t run_list;  
    unsigned int proc_num;  
};
```

在此结构中, 还有一个 proc\_num 成员变量, 表示当前就绪进程的个数。对于 lab4/proj13.1 而言, 它实现了 RR 调度算法, 由于 RR 调度算法需要分析和调整每个进程的时间片, 所以对就绪进程队列组织结构进行了小的扩展, 增加了一个最大时间片的设定, 用于比较当前进程的时间片是否已经超过了就绪进程队列设计的最大时间片范畴。其具体的设置如下:

```
struct run_queue {  
    list_entry_t run_list;  
    unsigned int proc_num;  
    int max_time_slice;  
};
```

对于 lab4/proj13.2, 实现了 MLFQ 调度算法, 这里包含了 4 个不同级别的 RR 就绪队列, 于是需要对就绪进程队列进行进一步扩展:

```
struct run_queue {
```

```
list_entry_t run_list;
unsigned int proc_num;
int max_time_slice;
list_entry_t rq_link;
};
```

其中增加的 `rq_link` 就用于把就绪进程放入到某个运行队列中。另外，对于 RR 调度算法和 MLFQ 调度算法而言，需要基于就绪进程和执行进程的时间片来调整就绪进程组织结构和抢占当前运行进程，所以需要代表每个进程的进程控制块进行扩展，于是在 lab4/proj13.1 中对进程控制块进行了扩展，增加了 `time_slice` 成员变量来表示进程的时间片。

这样用于表示调度器操作的数据结构 `sched_class` 进程调度类和表示就绪进程组织形式的 `run_queue` 数据结构就绪进程队列就形成了进程调度类框架的主体。但如何在操作系统中运用这些数据结构，来实现与调度算法无关的调度框架呢？

### 调度点的关键调度相关函数

在本节中“进程调度时机”小节分析了进程在执行中状态变化过程的处理流程。虽然进程各种状态变化的原因和导致的调度处理各异，但其实仔细观察各个流程的共性部分，会发现其中只涉及了三个关键调度相关函数：`wakeup_proc`、`schedule`、`run_timer_list`。如果我们能够让这三个调度相关函数的实现与具体调度算法无关，那么就可以认为 `ucore` 实现了一个与调度算法无关的调度框架。

`wakeup_proc` 函数其实完成了把一个就绪进程放入到就绪进程队列中的工作，为此还调用了调度类接口函数 `sched_class_enqueue`，这使得 `wakeup_proc` 的实现与具体调度算法无关。`schedule` 函数完成了与调度框架和调度算法相关三件事情：把当前继续占用 CPU 执行的运行进程放入到就绪进程队列中，从就绪进程队列中选择一个“合适”就绪进程，把这个“合适”的就绪进程从就绪进程队列中摘除。通过调用三个调度类接口函数 `sched_class_enqueue`、`sched_class_pick_next`、`sched_class_enqueue` 来使得完成这三件事情与具体的调度算法无关。`run_timer_list` 函数在每次 timer 中断处理过程中被调用，从而可用来调用调度算法所需的 timer 时间事件感知操作，调整相关进程的进程调度相关的属性值。通过调用调度类接口函数 `sched_class_proc_tick` 使得此操作与具体调度算法无关。

这里涉及了一系列调度类接口函数：

- `sched_class_enqueue`
- `sched_class_dequeue`
- `sched_class_pick_next`
- `sched_class_proc_tick`

这 4 个函数的实现其实就是调用某基于 `sched_class` 数据结构的特定调度算法实现的 4 个指针函数。采用这样的调度类框架后，如果我们需要实现一个新的调度算法，则需要定义一个针对此算法的调度类的实例，一个就绪进程队列的组织结构描述就行了，其他的事情都可交给调度类框架来完成。

## 5. 进程调度策略/算法

### FCFS 调度算法的实现

FCFS 调度算法不需要考虑执行进程的运行时间和就绪进程的等待时间，所以其实现很简单。首先其就绪进程队列是一个双向链表：

```

    struct run_queue {
        list_entry_t run_list;
        unsigned int proc_num;
    };

```

并在 sched.c 中声明了基于此数据结构的就绪进程队列 rq。在执行 wakeup\_proc 时，把进程加入到就绪进程队列中（插入到 rq 的队列尾）。在执行 schedule 函数时，当前执行进程只会是已经退出或者已经睡眠两种情况，这时需要从就绪进程队列中选择最早进入就绪队列的进程（位于 rq 的队列头），然后把它从就绪队列中摘除。由于没有时间片的考虑，所以与 run\_timer\_list 函数相关的 FCFS\_proc\_tick 函数为空函数。我们在看看其他三个 FCFS 算法相关的函数实现。

FCFS\_enqueue 的函数实现如下：

```

static void
FCFS_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    proc->rq = rq;
    rq->proc_num ++;
}

```

即把一个就绪进程插入到就绪进程队列 rq 的队列尾，并把表示就绪进程个数的 proc\_num 加一。

FCFS\_pick\_next 的函数实现如下：

```

static struct proc_struct *
FCFS_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}

```

即选取就绪进程队列 rq 中的队头队列元素，并把队列元素转换成进程控制块指针。

FCFS\_dequeue 的函数实现如下：

```

static void
FCFS_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}

```

即把就绪进程队列 rq 的进程控制块指针的队列元素删除，并把表示就绪进程个数的 proc\_num 减一。

## RR 调度算法的实现

RR 调度算法的就绪队列在组织结构上也是一个双向链表，只是增加了一个成员变量，表明在此就绪进程队列中的最大执行时间片。而且在进程控制块 proc\_struct 中增加了一个成员变量 time\_slice，用来记录进程当前的可运行时间片段。这是由于 RR 调度算法需要考虑

执行进程的运行时间不能太长。在每个 timer 到时的时候，操作系统会递减当前执行进程的 time\_slice，当 time\_slice 为 0 时，就意味着这个进程运行了一段时间（这个时间片段称为进程的时间片），需要把 CPU 让给其他进程执行，于是操作系统就需要让此进程重新回到 rq 的队列尾，且重置此进程的时间片为就绪队列的成员变量最大时间片 max\_time\_slice 值，然后再从 rq 的队列头取出一个新的进程执行。下面来分析一下其调度算法的实现。RR\_dequeue 和 RR\_pick\_next 的函数实现与 FCFS\_dequeue 和 FCFS\_pick\_next 函数实现一致，这里不再赘述。

RR\_enqueue 的函数实现如下：

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

即把某进程的进程控制块指针放入到 rq 队列末尾，且如果进程控制块的时间片为 0，则需要把它重置为 rq 成员变量 max\_time\_slice。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。

RR\_proc\_tick 的函数实现如下：

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

即每次 timer 到时后，trap 函数将会间接调用此函数来把当前执行进程的时间片 time\_slice 减一。如果 time\_slice 降到零，则设置此进程成员变量 need\_resched 标识为 1，这样在下次中断来后执行 trap 函数时，会由于当前进程成员变量 need\_resched 标识为 1 而执行 schedule 函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行。

## MLFQ 调度算法的实现

当前 MLFQ 调度算法其实是对 RR 算法的进一步扩展，即把一个 rq 队列扩展为了 n 个 rq 队列。多个 rq 队列确保了长时间运行的进程优先级会随着执行时间的增加而降低，而短时间运行的进程会优于长时间运行的进程被先调度执行。在 proj13.2 中实现了一个比较简单的 MLFQ 调度算法，其就绪队列由 4 个双向链表组成（在 sched.c 中定义）：

```
static struct run_queue __rq[4];
```

第  $i$  个  $rq$  的最大时间片为  $8 * (1 < i)$ ，其中  $0 \leq i < 4$ 。创建的新进程首先会放到第 0 个  $rq$  中，这样新创建的进程的时间片为 8（即第 0 个  $rq$  的最大时间片），如果进程用完了其时间片，则会降到第 1 个  $rq$  中，此时这个进程的时间片设置为 16（即第 1 个  $rq$  的最大时间片），持续下去，直到第 3 个  $rq$ ，此时，此时这个进程的时间片设置为 64（即第 3 个  $rq$  的最大时间片）。

下面来分析一下其调度算法的实现。MLFQ\_dequeue 和 MLFQ\_proc\_tick 的函数实现就是直接调用 RR\_dequeue 和 RR\_proc\_tick 函数，这里不再赘述。

MLFQ\_enqueue 的函数实现如下：

```
static void
MLFQ_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    struct run_queue *nrq = rq;
    if (proc->rq != NULL && proc->time_slice == 0) {
        nrq = le2rq(list_next(&(proc->rq->rq_link)), rq_link);
        if (nrq == rq) {
            nrq = proc->rq;
        }
    }
    sched_class->enqueue(nrq, proc);
}
```

如果判断进程  $proc$  的  $rq$  不为空且  $time\_silce$  为 0，则需要降  $rq$  队列，从  $rq[i]$  调整到  $rq[i+1]$ ，如果是最后一个  $rq$ ，即  $rq[3]$ ，则继续保持在  $rq[3]$  中，然后调用  $RR\_enqueue$  把  $proc$  插入到  $rq[i+1]$  中。

MLFQ\_pick\_next 的函数实现如下：

```
static struct proc_struct *
MLFQ_pick_next(struct run_queue *rq) {
    struct proc_struct *next;
    list_entry_t *list = &(rq->rq_link), *le = list;
    do {
        if ((next = sched_class->pick_next(le2rq(le, rq_link)))
            != NULL) {
            break;
        }
        le = list_next(le);
    } while (le != list);
    return next;
}
```

按顺序从  $rq[0] \sim rq[3]$  依次搜寻，直接调用  $RR\_pick\_next$  来查找某  $rq$  链表的头指向的就绪进程，只要找到就返回。

目前的实现其实相对简化了不少。我们把睡眠时间长的进程设定为 IO-bounded 进程。如果有一个进程经常等待某 IO 事件（比如鼠标移动或按键），然后占用一部分 CPU 时间，但整体上执行了很长时间，则这类进程也算是 IO-bounded 进程，需要得到及时响应。如果

---

采用上述 MLFQ 调度算法，则此进程也会落到 `rq[3]` 中执行，导致这类进程无法及时响应 IO 事件。出现这种情况的原因是上述调度算法只考虑了对运行的进程做降级惩罚操作（即只是根据运行时间的增加来递增 `i`，使得运行时间长的进程最终都会落到 `rq[3]` 中），但对于睡眠很久的进程没有做升级奖励操作（即没有根据睡眠时间的增加来递减 `i`，使得睡眠时间长的进程能够尽量保持在 `rq[0]` 中）。

【问题】对于新创建的子进程，在 FCFS/RR/MLFQ 调度算法如何设定其时间片比较合理？

【问题】如何扩展 MLFQ 调度算法，可减少长期运行的 IO-bounded 进程的响应时间？

【问题】如果需要实现内核级抢占（kernel preemptive），需要如何设计 ucore？



# 操作系统设计与实现

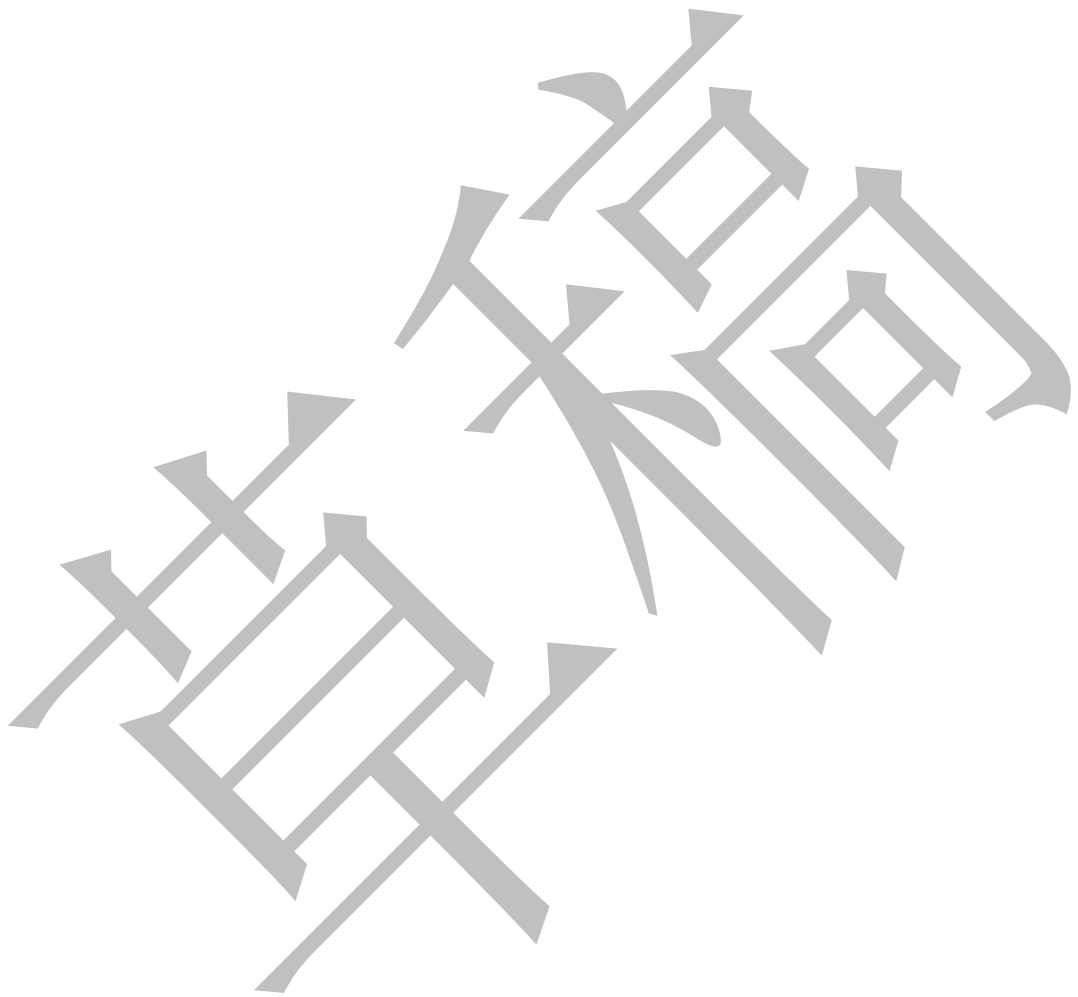
## The Design & Implementation of Operating System

陈渝 向勇

清华大学计算机系

---

2011 年





## 第 5 章 同步互斥与死锁处理

【待写中】

## 第 6 章 文件系统

文件系统是什么？从用户的角度看，文件系统是为了方便长期保存程序获得和产生的数据集合，一般驻留在类似硬盘、光盘、SSD、Flash 存储介质上，这些数据以文件的形式存在，应用程序可以很方便地通过目录名+文件名所形成的文件路径访问到文件中保存的数据。从操作系统实现的角度看，由于类似硬盘这样的存储介质访问速度远慢于内存，但掉电后数据不会丢失，所以文件系统涉及三个层面的实现。一个层面是设计数据在存储介质的物理空间的组织存放方式，便于从时间和空间上高效访问存储介质上的数据；第二个层面是如何组织内存中的数据放置方式，使得可有效地把存储介质中的数据读入到内存中或把内存中的数据写回到存储介质中；第三个层面是给应用程序提供便捷的文件抽象表示和文件访问系统调用，使得应用程序可以方便地根据文件名进行文件内容的读写访问。

本章主要针对操作系统实现的角度来分析如何进行文件系统的设计与实现，涉及基于索引方式的文件系统设计，虚拟文件系统设计，文件系统系统访问接口等。读者通过阅读本章的内容并动手实践相关的 lab6 中的几个 project 实验：

- proj17: 建立虚拟文件系统--VFS，文件系统相关的 syscall 和基于简单输入输出设备（串口、键盘和 NULL）的设备文件系统。
- proj17.1: 一个用于进程间通讯的 PIPE 文件系统；
- proj18: 一个基于索引节点组织方式的 Simple FS 文件系统；
- proj18.1: 增加 mkdir/link/rename/unlink 等目录功能的 Simple FS 文件系统；
- proj18.2/18.3: 提供执行位于 SimpleFS 文件系统中可执行文件的功能；
- proj19: 提供一个用户 shell，可以访问和执行位于 Simple FS 文件系统上的目录与文件。

可以掌握如下知识：

- 与文件系统原理相关
  - 文件抽象的概念：文件、目录项、索引节点
  - 文件空间组织：连续空间组织、链接空间组织、索引节点空间组织
  - 空闲空间管理：空闲空间位图
  - 目录实现：目录项信息存储
  - 文件实现：索引节点和数据块
  - 文件系统：虚拟文件系统的概念
- 操作系统原理之外
  - 设备文件系统
  - PIPE 文件系统
  - 跨文件系统访问
  - 文件系统的内存组织和磁盘组织



---

本章内容中主要涉及文件系统的重要功能主要有两个：

- 文件系统的访问界面：如何给上层应用提供一个简洁的访问接口，并在进程中进行实现。
- 文件系统的实现：这部分功能比较复杂，涉及如何实现一个实际文件系统并能够放在磁盘上存放；如何在一个统一的虚拟文件系统层下与其他文件系统共存；如何把设备等其他可读写资源也当成文件访问等。

为了让读者能够从实践上来理解文件系统的基本原理，我们设计了上述实验，主要的功能逐步实现如下所示：

- 首先了解用户访问文件的接口，这需要设计相关的系统调用和进程控制块对文件访问的扩充部分；
- 然后了解一下在此文件访问接口下，如何建立一个 OS 范畴内的统一的文件访问接口实现函数列表，即 VFS，并通过把设备看成是设备文件来完成与 VFS 的对接，从而让用户程序可以实现键盘输入和串口输出；
- 为了支持不同进程间对同一文件进行读写，基于内存实现 PIPE 文件系统，让父子进程能够进行数据传输；
- 上述 proj 大致完成了对一个文件的读写访问的操作系统即支持，但没有实现以磁盘为存储介质的文件系统，为此需要设计文件的组织形式，扩展目录、目录项访问支持，从而实现基于索引节点的文件系统 Simple FS。此次以前放在 ucore kernel 镜像后的那些应用程序都可以以 Simple FS 支持的文件格式放在磁盘上，并能够被 ucore 读写。

## 6.1 建立文件访问接口并实现对设备文件的访问

### 6.1.1 实验目标

文件系统中的一个文件是一种共享资源，需要应对多个用户进程并发访问的需求。由于 ucore 在 lab4 完成了同步互斥的功能支持，这为文件系统的正确实现提供了基础。为了更好地实现后续的基于磁盘介质的不同实际文件系统，这里先设计了一个简洁的文件访问接口和一个简单的 VFS 层，使得上层应用能够不用关注具体的文件系统是设备、还是其他具体的文件系统，就能够读写某个文件。

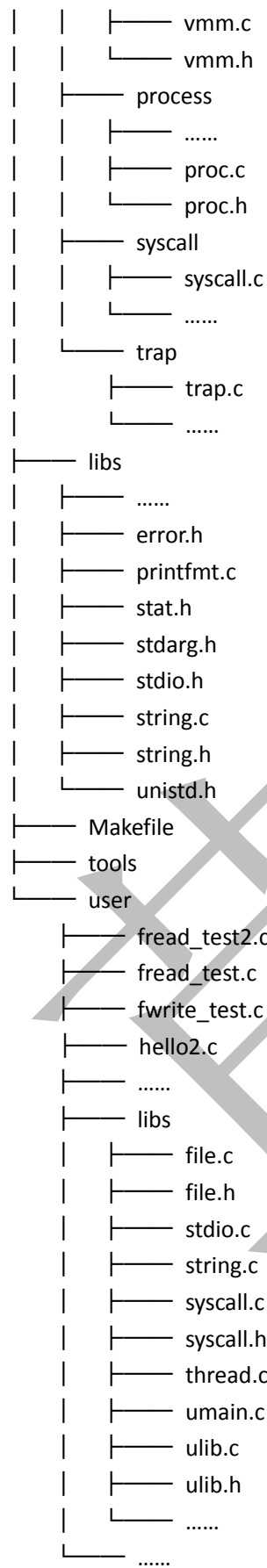
### 6.1.2 Proj17 概述

## 6. 实现描述

proj17 是 lab6 的第一个 proj，其目的是希望能够给应用程序提供一个简洁的文件访问接口，并把两个简单的设备（键盘和串口）看成是文件，让应用程序能够通过文件访问接口，即通过简单的 read 和 write 就可完成对设备的读写操作。

## 7. 项目组成

```
proj17/
├── .....
├── kern
│   ├── .....
│   ├── fs
│   │   ├── devs
│   │   │   ├── dev.c
│   │   │   ├── dev.h
│   │   │   ├── dev_null.c
│   │   │   ├── dev_stdin.c
│   │   │   └── dev_stdout.c
│   │   ├── file.c
│   │   ├── file.h
│   │   ├── fs.c
│   │   ├── fs.h
│   │   ├── iobuf.c
│   │   ├── iobuf.h
│   │   ├── swap
│   │   │   ├── swapfs.c
│   │   │   └── swapfs.h
│   │   ├── sysfile.c
│   │   ├── sysfile.h
│   │   └── vfs
│   │       ├── inode.c
│   │       ├── inode.h
│   │       ├── vfs.c
│   │       ├── vfsdev.c
│   │       ├── vfsfile.c
│   │       ├── vfs.h
│   │       ├── vfslookup.c
│   │       └── vfspath.c
│   ├── init
│   │   ├── .....
│   │   └── init.c
│   ├── libs
│   │   ├── .....
│   │   ├── stdio.c
│   │   └── string.c
│   ├── mm
│   │   ├── .....
│   │   └── pmm.c
```



相对于lab5的proj16, lab6的proj17增加和修改了相对比较多的文件, 跳跃性较大。主要修改和增加的文件如下:

- 文件系统测试用例
  - user/fread\_\*.c | fwrite\_\*.c | hello2.c: 对proj17中设备文件系统实现进行测试的测试用例;
- 通用文件系统接口
  - user/libs/file.[ch] | syscall.c: 与文件系统中文件读写操作相关的用户库实行;
  - kern/syscall.[ch]: 文件中包含文件系统相关的内核态系统调用接口
  - kern/fs/sysfile.[ch] | file.[ch] | fs.[ch] | iobuf.[ch]: 通用文件系统访问接口和实现
- 文件系统抽象层-VFS
  - kern/vfs/\*. [ch]: 虚拟文件系统接口与实现
- 设备文件系统的设备IO接口
  - dev/\*. [ch]: 键盘/串口/NULL设备提供给文件系统的I/O访问接口和实现
- 对内核其它模块的扩充
  - kern/process/proc.h: 增加成员变量 struct fs\_struct \*fs\_struct, 用于支持进程对文件的访问;
  - kern/init/init.c: 增加调用初始化文件系统的函数fs\_init。

## 8. 编译运行

编译并运行 proj17 的命令如下:

```
make
```

```
make qemu
```

则可以得到如下显示界面

```
thuos:~/oscourse/ucore-svn2/ucore-read-only/i386/lab6_filesystem/proj17$ make
qemu
(THU.CST) os is loading ...
```

Special kernel symbols:

```
entry 0xc010002c (phys)
```

```
etext 0xc011dfdc (phys)
```

```
edata 0xc041297c (phys)
```

```
end 0xc0418fcc (phys)
```

Kernel executable memory footprint: 3172KB

.....

```
sched class: MLFQ_scheduler
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:      262144(sectors), 'QEMU HARDDISK'.
check_swap() succeeded.
.....
check_mm_shm_swap() succeeded.
++ setup timer interrupts
kernel_execve: pid = 3, name = "hello2".
Hello world!!.
I am process 3.
hello2 pass.
all user-mode processes have quit.
init check memory pass.
.....
```

从上图可以看到，proj17 创建了一个用户进程 hello2，然后启动 hello2 运行，此进程调用输出了一些信息：“Hello world!!.”。这好像与 proj17 之前的用户进程的输出没啥区别，但其实整个的实现流程变了，经过了文件访问系统调用接口，虚拟文件系统 VFS 和设备文件，最后到串口设备驱动来完成了最终的字符串输出。上述执行过程其实包含了对文件访问的。下面我们将基本过程，下面将从原理和实现两个方面对此进行进一步阐述。

### 6.1.3 【原理】用户角度的文件系统

文件是具有一定名称的一组相关数据的集合。文件通常存储在外部存储介质上(如磁盘、光盘等)，也可以是某种资源或设备。简言之，具有读或写操作的资源都可以用文件来表示。下面我们从文件命名、文件构成与属性、文件访问与控制接口等方面，讨论文件相关的基本概念。

#### 1. 文件命名

为了方便用户和应用程序访问文件，每个文件都有特定的名称，即一个字符串。这样用户和应用程序就不必关心文件的具体物理存在形式，存储方法、位置等，而可以直接通过文件名来使用文件。

各种文件系统的文件命名不尽相同，对文件名称的长度限制也各不相同。如早期 MSDOS 的 FAT 文件系统采用 8.3 命名规则，即规定文件名为 8 个字符、外加句点和 3 个字符的扩展名；而现代 windows NT 中的 NTFS 文件系统则支持长达 255 个字符的文件名。

有的文件系统不区分文件名的大小写，而有的则加以区分。有的文件系统规定文件名只能使用 ASCII 字符，而有的文件系统的文件名可以使用 Unicode。

#### 2. 文件构成及其属性

文件有两类重要信息组成：一是文件所包含的内容，常称为文件内容信息；二是关于文件本身的说明信息，常称为文件元数据信息或文件属性信息。文件属性信息主要描述文件的

---

创建日期、文件长度、文件权限、文件所处位置等，这些信息主要用于操作系统管理文件。下面是一些常用的文件属性：

- **文件名称：**文件名称是供用户使用的外部身份。这是文件最基本的外在属性。文件名称通常是由 ASCII 码或 Unicode 构成的字符串组成。
- **文件拥有者 (owner)：**操作系统通常支持多用户，这样不同的用户可拥有各自的文件，且对文件的操作权限也各不相同。通常作为文件创建者（或拥有者）的用户对自己所建的文件拥有一切权限，而对其他用户所建的文件则拥有有限的权限。
- **文件权限：**这里权限主要是指对文件的读、写、执行的权限。文件拥有者可以为自己所拥有的文件赋予各种权限，如可允许自己读写执行，允许同组的用户读，不允许其他用户读写执行。
- **文件长度：**文件长度通常是其文件内容信息的大小。
- **文件时间：**文件时间有很多，如文件创建时间，最近一次的文件修改时间，最近一次的文件执行时间等。

### 3. 文件控制与访问接口

为了方便用户和应用程序控制与访问文件，操作系统系统向用户提供各种文件控制与访问系统调用接口。用户进程通过这些接口来对文件进行各种操作。对文件的操作可以分为两大类：一类是对文件的控制与管理，例如，建立新文件、打开文件、关闭文件、删除文件等，一类是对文件的访问，例如读写文件、执行文件等；另一类是对记录的操作，最简单的记录可以是一个字符，例如，查找文件中的字符串，以及插入和删除等。

### 4. 目录

目录也可形象地称为“文件夹”，一个目录或文件夹就是一个虚拟“容器”。在目录里面保存着一组文件和其它一些目录（文件夹）。这样可以应付文件数量过多或需要对文件分类的需求，可以达到有组织的存储文件的目的。在一个目录（文件夹）中的另一个目录（文件夹）被称作它的子目录（子文件夹）。这样，这些目录（文件夹）就构成了层次（**hierarchy**），或树形结构。从某种程度上可以把目录看成是一类特殊的文件，目录的文件内容是此目录下包含的文件元数据信息。

### 5. 文件共享语义

文件是一种可共享资源，当两个或更多个用户进程共享访问相同的文件时，必须精确地定义读和写的语义以避免产生访问冲突问题。在允许用户进程共享文件的操作系统（如 UNIX）中，对所有文件访问操作需强制一个文件访问的相对时间顺序，并且总是返回最近更新的文件信息。我们称这种语义为 **UNIX 文件共享语义**，此文件共享语义规定：

- 在 READ 文件操作紧跟在 WRITE 文件操作后执行时，READ 文件操作返回 WRITE 文件操作刚刚写入的值。

- 当一个 READ 文件操作跟在两个紧相连发生的 WRITE 文件操作后，读入的值就是后一个写操作写入的值。

## 6.1.4 【原理】操作系统角度的文件系统

操作系统中的每一个文件都有一个名字以供识别，但文件名是便于用户和用户进程访问文件的外在标识。由操作系统的文件管理系统掌握与管理文件的实际存储设备、位置、大小、特性等的信息。若用户进程要存取文件必须要得到文件系统的帮助，把要存取文件的有关信息和要求，诸如文件的名称、读还是写文件、以何种方式读 / 写等通过系统调用告诉文件系统，由文件系统在设备中建立、寻找、定位文件，分配存取文件的缓冲区，做好存取文件要求的一切准备工作。

文件是一个公共资源，可以保存在磁盘上，也可以就是某个外设，用户进程都可以访问。但不同进程在访问同一文件时，可能访问的位置和具体操作等是不同的，所以在操作系统实现上，首先需要考虑文件的物理实体（磁盘块或某外部设备），也需要考虑从单个进程和整个操作系统这两个范畴来动态描述和管理文件。在进程层面，需要维护一个数据结构，保存当前时间下，进程打开（如执行 `open` 用户函数）的文件列表，且需要维护每个打开文件的上次访问位置，便于各个进程根据各自需求访问同一个文件。在操作系统层面，文件需要在内存中保存相关的最新内容数据信息和元数据信息，这些信息可以被所有进程共享，所以这些信息不应该局限在某一特定进程内，而是在操作系统范畴内维持一个版本，便于文件系统的高效管理和访问。

## 6.1.5 【原理】虚拟文件系统

虚拟文件系统(VFS)是 Sun Microsystems 公司在定义网络文件系统(NFS)时创造的。NFS 是一种用于网络环境的分布式文件系统，它的实现与基于磁盘的文件系统实现很不一样，为了不用区分是 NFS 还是基于磁盘的文件系统，SUN OS 设计了一个 VFS 访问层，在 VFS 定义好的统一接口下给上层应用和下层物理文件系统提供一致的函数调用界面。这极大地简化了应用程序和内核其他部分对文件的访问接口。

简单地说，虚拟文件系统(VFS)是物理文件系统与文件服务需求方之间的一个接口层，它对访问物理文件系统的函数接口细节进行归纳与抽象，使得对不同文件系统的访问在操作系统内核以及运行的进程看来，都是相同的。严格说来，VFS 并不是一种实际的文件系统，它只是一个对文件对象和访问函数的抽象层，存在于内存中，VFS 在操作系统启动时建立，在操作系统关闭时消亡。

VFS提出了四个文件系统抽象概念：文件(file)、目录项(dentry)、索引节点(inode)和安装点(mount point)。

- 文件：文件中的内容可理解为是一有序字节buffer，文件都有一个方便应用程序识别的文件名称（也称文件路径名）。典型的文件操作有读、写、创建和删除等。
- 目录项：目录项不是目录，而是目录的组成部分。在目录被看作一种特定的文件，而目录项是文件路径中的一部分。如一个文件路径名是“/test/testfile”，则包含的目录项为：根目录“/”，目录“test”和文件“testfile”，这三个都是目录项。一般而言，目录项包含目录项的名字（文件名或目录名）和目录项的索引节点（见下面的描述）位置。
- 索引节点：将文件的相关元数据信息（如访问控制权限、大小、拥有者、创建时间、数据内容等等信息）存储在一个单独的数据结构中，该结构被称为索引节点。

- 安装点：在UNIX中，文件系统被安装在一个特定的文件路径位置，这个位置就是安装点。所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。

上述抽象概念形成了各种文件系统的共性逻辑数据结构，并需要通过一个具体文件系统的架构设计与实现把上述信息映射并储存到磁盘介质或外设上。一个具体的文件系统需要在磁盘布局也实现上述抽象概念。比如文件元数据信息存储在磁盘块中的索引节点上。当文件被载如内存时，内核需要使用磁盘块中的索引点来构造内存中的索引节点。

在proj17中，我们把外设作为一个简单的物理文件系统，只需完成读写功能，不涉及保存的功能，所以只需文件和索引节点的部分共性数据结构，而不需考虑目录项、安装点等概念，不用考虑如何完成具体的存储。

## 6.1.6 【实现】ucore 文件系统总体架构

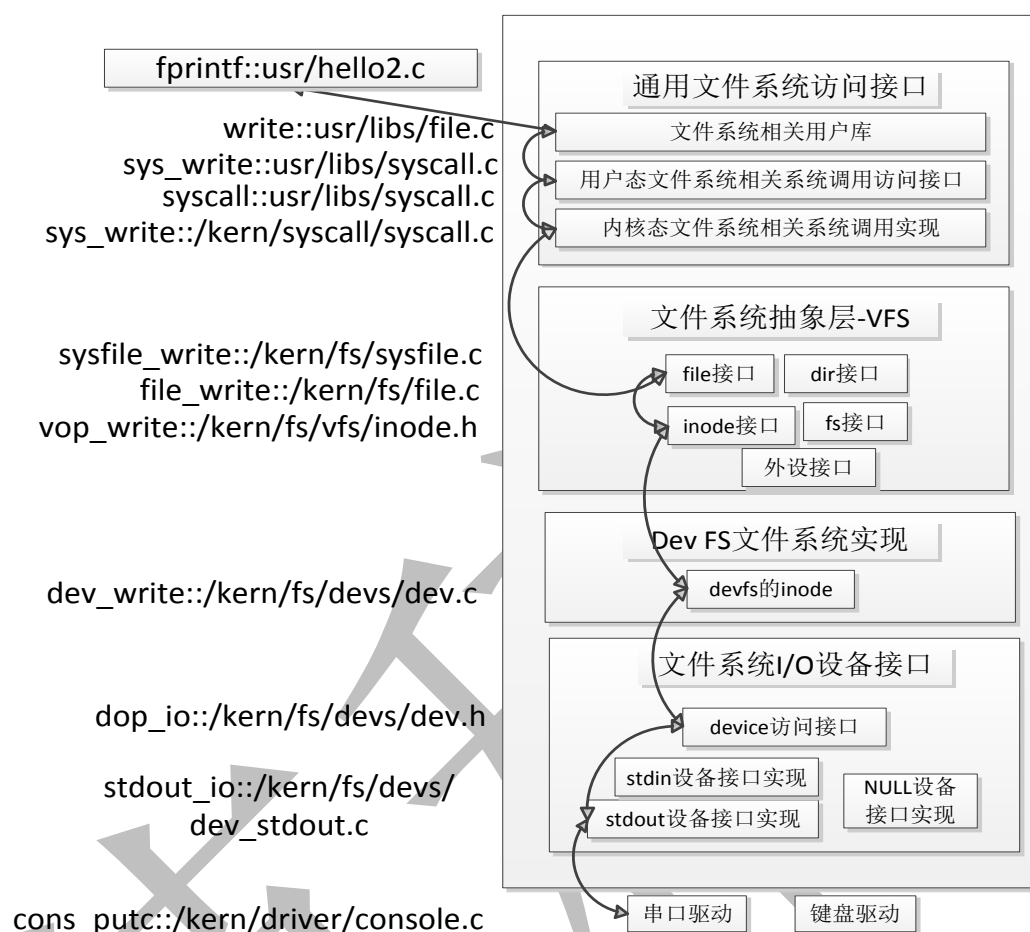
proj17不涉及面向磁盘的存储，只考虑了以文件读写的访问形式访问设备文件系统所代表的外设。所以在实现上简单了许多。但ucore依然模仿了UNIX文件系统的总体设计思路，ucore的文件系统架构主要由四部分组成：

- 通用文件系统访问接口层：该层提供了一个从用户空间到文件系统的标准访问接口。这一层访问接口让应用程序能够通过一个简单的接口获得ucore内核的文件系统服务。
- 文件系统抽象层：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块）访问。向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节。
- 设备文件系统层：一个把简单输入输出设备看成文件的设备文件系统。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问外设接口。
- 外设接口层：向上提供device访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动的接口，比如disk设备接口/串口设备接口/键盘设备接口等。

对照上面的层次我们看看文件系统的访问处理过程，可加深对文件系统的总体理解。假如应用程序操作文件（如读写），首先需要通过文件系统的通用文件系统访问接口层给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给设备文件系统，设备文件系统把应用程序的访问请求转化为对设备的I/O请求，并通过外设接口层交给外设来完成具体的I/O操作。结合用户态写文件函数write的整个执行过程，我们可以了解ucore文件系统架构的层次和依赖关系。



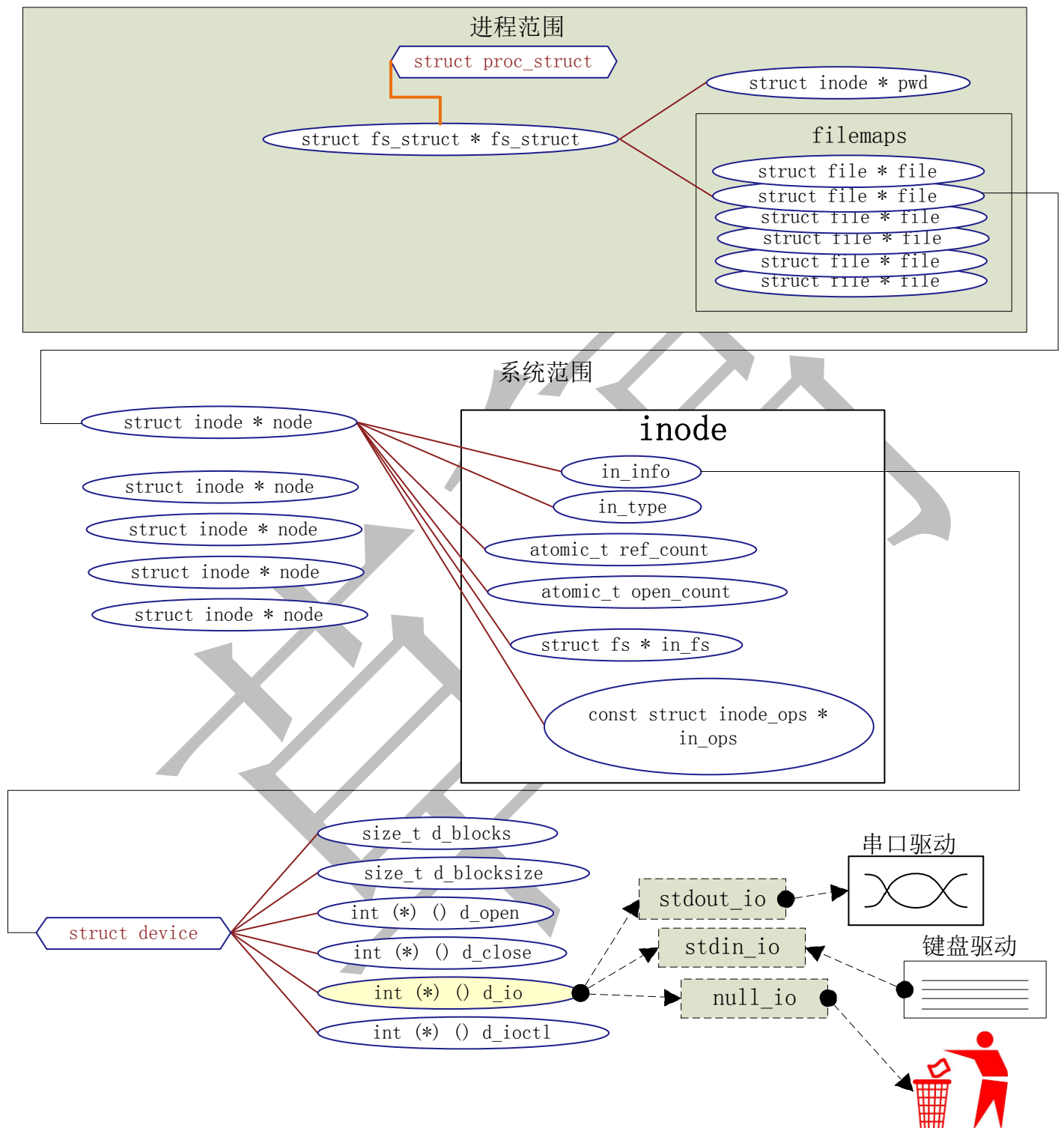
## ucore文件系统总体结构



从ucore操作系统不同的角度来看，ucore中的文件系统架构包含四类主要的数据结构，它们分别是：

- 超级块 (SuperBlock)，它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个OS空间。
- 索引节点 (inode)：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整个OS空间。
- 目录项 (dentry)：它主要从文件系统的文件路径的角度描述了文件路径中的特定目录。它的作用范围是整个OS空间。
- 文件 (file)，它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

但在proj17中，主要涉及的是file对象。如果一个用户进程打开了一个文件，那么在ucore中涉及的相关数据结构（其中相关数据结构将在下面各个小节中展开叙述）和关系如下图所示：



## 6.1.7 【实现】通用文件系统访问接口

### 文件相关用户库函数

Lab6 的 proj17 中部分用户库函数与文件系统有关，我们先讨论对单个文件进行操作的系统调用。在文件操作方面，最基本的相关函数是 `open`、`close`、`read`、`write`。在读写一个文件之前，首先要用 `open` 系统调用将其打开。`open` 的第一个参数指定文件的路径名，可使用绝对路径名；第二个参数指定打开的方式，可设置为 `O_RDONLY`、`O_WRONLY`、`O_RDWR`，分别表示只读、只写、可读可写。在打开一个文件后，就可以使用它返回的文件描述符 `fd` 对文件进行相关操作。在使用完一个文件后，还要用 `close` 系统调用把它关闭，其参数就是文件描述符 `fd`。这样它的文件描述符就可以空出来，给别的文件使用。

读写文件内容的系统调用是 `read` 和 `write`。`read` 系统调用有三个参数：一个指定所操作的文件描述符，一个指定读取数据的存放地址，最后一个指定读多少个字节。在 C 程序中调用该系统调用的方法如下：

```
count = read(filehandle, buffer, nbytes);
```

该系统调用会把实际读到的字节数返回给 `count` 变量。在正常情形下这个值与 `nbytes` 相等，但有时可能会小一些。例如，在读文件时碰上了文件结束符，从而提前结束此次读操作。

如果由于参数无效或磁盘访问错误等原因，使得此次系统调用无法完成，则 `count` 被置为 -1。而 `write` 函数的参数与之完全相同。

### 文件访问相关系统调用

与文件相关的 `open`、`close`、`read`、`write` 用户库函数对应的是 `sys_open`、`sys_close`、`sys_read`、`sys_write` 四个系统调用接口。这 4 个系统调用函数接口将通过 `syscall` 函数来获得 ucore 的内核服务。当到了 ucore 内核后，在调用文件系统抽象层的 `file` 接口。

### 文件访问相关数据结构--`iobuf`

用户态的应用程序把读出文件中数据或把用户数据写入文件的过程中，涉及到如何把数据在内核态和用户态之间安全可靠地传递。Ucore 为此设计了 `iobuf` 数据结构来描述需要在用户态和内核态之间传递数据所需的信息。`iobuf` 提供了一个文件系统与其他结构（如内核等）的读写的缓冲区。如果有操作想往文件里写东西，就先写到 `iobuf` 里面，然后文件系统再向 `iobuf` 索取内容。而有操作需要读文件或者文件夹的信息时，文件系统也先写入到 `iobuf` 中，之后外界再通过它来获得内容。`iobuf` 数据结构定义在 `iobuf.h` 中，具体描述如下：

```
struct iobuf {  
    void *io_base; //访问 buffer 的起始地址  
    off_t io_offset; //访问 buffer 的偏移量  
    size_t io_len; //访问 buffer 的可用空间大小  
    size_t io_resid; //还需访问的空间大小
```

```
};
```

这里描述了内核维护的缓冲区描述结构，用来表示在 `sysfile_read` 和 `sysfile_write` 函数中通过 `kmalloc` 分配的一个内核缓冲区--`buffer`。`iobuf` 中有 4 个成员，分别是 `io_base`，它始终指向下一个未被使用的内存空间。所以在读写操作结束后，要对它的值（即指向的内存地址）做相应的维护。`io_offset` 表示信息所在地址的偏移量。如果是读文件时，就表示从文件的第几个字节开始读，写的时候表示从第几个字节开始写。`io_len` 一般表示待写的或者待读的字节数。而 `io_resid` 则表示实际读或者写完后剩余的字节数。可以通过 `io_resid` 是否为 0 来判断是否读写了规定的字节数。除 `io_len` 外另外 3 个成员都要在文件系统的函数里进行维护。

### iobuf 初始化

一个 `iobuf` 结构的初始化过程中，将记录对应 `buffer` 的起始地址，使用中的大小等信息，并记录用户进程访问文件的偏移位置。具体实现在 `iobuf_init` 函数中。

### iobuf 访问操作

设置 `iobuf` 的目的是简化文件读写过程，为此设置了 `iobuf_move` 函数，其主要功能是完成某内存区域与 `iobuf` 描述的 `buffer` 之间的数据传递（通过 `memmove` 函数完成），并在传递完毕后调整 `iobuf` 中的 `io_base`、`io_offset`、`io_resid`（见函数 `iobuf_skip` 的实现）。

## 6.1.8 【实现】文件系统抽象层-VFS

文件系统抽象层是把不同文件系统的对外共性接口提取出来，形成一个函数指针数组，这样，通用文件系统访问接口层只需访问文件系统抽象层，而不需关心具体文件系统的实现细节和接口。

### 1. file 接口

进程访问文件需要了解到文件许多信息，这些进行记录在 `file` 这个数据结构中。`File` 数据结构定义了进程在内核中直接访问的文件相关信息，`file` 数据结构的具体描述如下：

```
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;           //访问文件的执行状态
    bool readable;       //文件是否可读
    bool writable;       //文件是否可写
    int fd;              //文件在filemap中的索引值
    off_t pos;           //访问文件的当前位置
    struct inode *node;   //该文件对应的内存inode指针
    atomic_t open_count;  //打开此文件的次数
};
```

而在 `kern/process/proc.h` 中的 `proc_struct` 结构中描述了进程访问文件的数据接口 `fs_struct`，其数据结构定义如下：

```
struct fs_struct {
```

```

struct inode *pwd;           //进程当前执行目录的内存inode指针
struct file *filemap;       //进程打开文件的数组
atomic_t fs_count;          //访问此文件的线程个数??
semaphore_t fs_sem;         //确保对进程控制块中fs_struct的互斥访问
};

```

当创建一个进程后，该进程的fs\_struct将会被初始化或复制父进程的fs\_struct。当用户进程打开一个文件时，将从filemap数组中取得一个空闲file项，然后会把此file的成员变量node指针指向一个代表此文件的inode的起始地址。

## 2. inode 接口

index node是位于内存的索引节点，它是VFS结构中的重要数据结构，因为它实际负责把不同文件系统的特定索引节点信息（甚至不能算是一个索引节点）统一封装起来，避免了进程直接访问具体文件系统。其定义如下：

```

struct inode {
    union {                                //包含不同文件系统特定inode信息的
union域
        struct device __device_info;      //设备文件系统内存inode信息
        struct pipe_root __pipe_root_info;
        struct pipe_inode __pipe_inode_info; //pipe文件系统内存inode信息
        struct sfs_inode __sfs_inode_info;  //SFS文件系统内存inode信息
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_pipe_root_info,
        inode_type_pipe_inode_info,
        inode_type_sfs_inode_info,
    } in_type;                            //此inode所属文件系统类型
    atomic_t ref_count;                   //此inode的引用计数
    atomic_t open_count;                  //打开此inode对应文件的个数
    struct fs *in_fs;                     //抽象的文件系统，包含访问文件系统的函数指针
    const struct inode_ops *in_ops;       //抽象的inode操作，包含访问inode的函数指针
};

```

在inode中，成员变量in有一成员变量为in\_ops，这是对此inode的操作函数指针列表，其数据结构定义如下：

```

struct inode_ops {
    unsigned long vop_magic;
    int (*vop_open)(struct inode *node, uint32_t open_flags);
    int (*vop_close)(struct inode *node);
    int (*vop_read)(struct inode *node, struct iobuf *iob);
    int (*vop_write)(struct inode *node, struct iobuf *iob);
};

```

```

int (*vop_mkdir)(struct inode *node, const char *name);
int (*vop_getdirent)(struct inode *node, struct iobuf *iob);
int (*vop_create)(struct inode *node, const char *name, bool excl, struct inode
**node_store);
int (*vop_unlink)(struct inode *node, const char *name);
int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
.....
};

```

参照上面对Device FS文件系统中的索引节点操作函数的说明，可以看出inode\_ops是对常规文件、目录、设备文件所有操作的一个抽象函数表示。对于某一具体的文件系统中的文件或目录，只需实现相关的函数，就可以被用户进程访问具体的文件了，且用户进程无需了解具体文件系统的实现细节。

## 6.1.9 【实现】设备文件系统-Device FS

在 proj17 中，可以把一个设备看成一个文件，目前实行了 NULL 设备文件、STDIN 设备文件文件和 STDOUT 设备文件。这 3 个文件中，NULL 是一个只写文件（读没有效果），STDIN 设备在这里就是键盘，STDOUT 在这里就是 CONSOLE（串口、并口和文本显示器）。下面我们逐一分析 ucore 是如何让用户把设备看成文件来访问。

### 1. 关键数据结构

为了表示一个设备，需要有对应的数据结构，ucore 为此定义了 struct device，其描述如下：

```

struct device {
    size_t d_blocks;    //设备占用的数据块个数
    size_t d_blocksize; //数据块的大小
    int (*d_open)(struct device *dev, uint32_t open_flags); //打开设备的函数指针
    int (*d_close)(struct device *dev); //关闭设备的函数指针
    int (*d_io)(struct device *dev, struct iobuf *iob, bool write); //读写设备的函数指针
    int (*d_ioctl)(struct device *dev, int op, void *data); //用 ioctl 方式控制设备的函数指针
};

```

这个数据结构能够支持对块设备（比如磁盘）、字符设备（比如键盘、串口）的表示，完成对设备的基本操作。一个 inode 节点的成员变量 in\_type 的值是 0x1234，则此 inode 的成员变量 in\_info 将成为一个 device 结构。这样 inode 就和一个设备建立了联系，这个 inode 就是一个设备文件。

### 2. NULL 设备文件

NULL 设备是一个虚拟输入设备，是 ucore 的垃圾箱和无底洞，如果向这个设备写数据可顺利完成，NULL 设备只是把这些数据丢到，导致数据有去无回。实现 NULL 设备可验证从用户发起读写请求到最终设备完成读写请求的基本通路是畅通的。接下来我们看看 NULL 设备的相关处理过程。

## 初始化

既然 NULL 设备是设备文件的文件，自然有自己的 inode 结构。在系统初始化时，即只需如下处理过程

kern\_init-->fs\_init-->dev\_init-->dev\_init\_null-->dev\_init\_null

在 dev\_init\_null 中完成了对 NULL 设备文件的初始化。即首先创建了一个 inode，然后通过 null\_device\_init 完成对 inode 中的成员变量 inode->\_\_device\_info 进行初始：

```
null_device_init(struct device *dev) {
    dev->d_blocks = 0;
    dev->d_blocksize = 1;
    dev->d_open = null_open;
    dev->d_close = null_close;
    dev->d_io = null_io;
    dev->d_ioctl = null_ioctl;
}
```

这里可以看到，对于与用户程序访问此设备文件所需要的 open、close、read、write 函数最终将通过 null\_open、null\_close、null\_io（应对读写两个事情）来完成具体访问工作。

## 访问操作实现

null\_open 和 null\_close 这两个函数啥也不干，直接返回 0，表示一切 ok。null\_io 是完成具体读写的，其实现如下：

```
static int
null_io(struct device *dev, struct iobuf *iob, bool write) {
    if (write) {
        iob->io_resid = 0;
    }
    return 0;
}
```

这里可以看出，如果是写操作，则把 iob->io\_resid 设置为 0，表示剩余要写的字节数为 0。这里可以看出对于用户实际要写的数据根本没有用。对于读操作，就更简单了，直接返回 0，表明读不出任何数据。user/nulldevfs\_tet.c 用户态测试用例测试了对 null 设备文件的读写操作。

## 3. STDOUT 设备文件

这里的 STDOUT 设备文件实际上就是指的 console 外设（它其实是串口、并口和 CGA 的组合型外设）。这个设备文件是一个只写设备，如果读这个设备，就会出错。接下来我们看看 STDOUT 设备的相关处理过程。

## 初始化

STDOUT 设备文件的初始化过程与 NULL 设备文件类似，stdout\_device\_init 完成了主要的初始化工作，具体实现如下：

```
Static void
stdout_device_init(struct device *dev) {
    dev->d_blocks = 0;
```

```

dev->d_blocksize = 1;
dev->d_open = stdout_open;
dev->d_close = stdout_close;
dev->d_io = stdout_io;
dev->d_ioctl = stdout_ioctl;
}

```

可以看到，`stdout_open` 函数完成设备文件打开工作，如果发现用户进程调用 `open` 函数的参数 `flags` 不是只写（`O_WRONLY`），则会报错。

## 访问操作实现

`stdout_io` 函数完成设备的写操作工作，具体实现如下：

```

static int
stdout_io(struct device *dev, struct iobuf *iob, bool write) {
    if (write) {
        char *data = iob->io_base;
        for (; iob->io_resid != 0; iob->io_resid --) {
            cputchar(*data ++);
        }
        return 0;
    }
    return -E_INVALID;
}

```

可以看到，要写的数据放在 `iob->io_base` 所指的内存区域，一直写到 `iob->io_resid` 的值为 0 为止。每次写操作都是通过 `cputchar` 来完成的，此函数最终将通过 `console` 外设驱动来完成把数据输出到串口、并口和 CGA 显示器上过程。另外，也可以注意到，如果用户想执行读操作，则 `stdout_io` 函数直接返回错误值 `-E_INVALID`。

## 4. STDIN 设备文件

这里的 `STDIN` 设备文件实际上就是指的键盘。这个设备文件是一个只读设备，如果写这个设备，就会出错。接下来我们看看 `STDIN` 设备的相关处理过程。

### 初始化

`STDIN` 设备文件的初始化过程与 `NULL` 设备文件类似，`stdin_device_init` 完成了主要的初始化工作，具体实现如下：

```

static void
stdin_device_init(struct device *dev) {
    dev->d_blocks = 0;
    dev->d_blocksize = 1;
    dev->d_open = stdin_open;
    dev->d_close = stdin_close;
    dev->d_io = stdin_io;
    dev->d_ioctl = stdin_ioctl;
}

```



```

    p_rpos = p_wpos = 0;
    wait_queue_init(wait_queue);
}

```

相对于 `null` 和 `stdout` 的初始化过程, `stdin` 的初始化相对复杂一些, 多了一个 `stdin_buffer` 缓冲区, 描述缓冲区读写位置的变量 `p_rpos`、`p_wpos` 以及用于等待缓冲区的等待队列 `wait_queue`。在 `stdin_device_init` 函数的初始化中, 也完成了对 `p_rpos`、`p_wpos` 和 `wait_queue` 的初始化。

## 访问操作实现

`stdin_io` 函数负责完成设备的读操作工作, 具体实现如下:

```

static int
stdin_io(struct device *dev, struct iobuf *iob, bool write) {
    if (!write) {
        int ret;
        if ((ret = dev_stdin_read(iob->io_base, iob->io_resid)) > 0) {
            iob->io_resid -= ret;
        }
        return ret;
    }
    return -E_INVALID;
}

```

可以看到, 如果是写操作, 则 `stdin_io` 函数直接报错返回。所以这也进一步说明了此设备文件是只读文件。如果是读操作, 则此函数进一步调用 `dev_stdin_read` 函数完成对键盘设备的读入操作。`dev_stdin_read` 函数的实现相对复杂一些, 主要的流程如下:

```

static int
dev_stdin_read(char *buf, size_t len) {
    int ret = 0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        for (; ret < len; ret++, p_rpos++) {
            try_again:
            if (p_rpos < p_wpos) {
                *buf++ = stdin_buffer[p_rpos % STDIN_BUFSIZE];
            }
            else {
                wait_t __wait, *wait = &__wait;
                wait_current_set(wait_queue, wait, WT_KBD);
                local_intr_restore(intr_flag);

```

```

        schedule();

        local_intr_save(intr_flag);
        wait_current_del(wait_queue, wait);
        if (wait->wakeup_flags == WT_KBD) {
            goto try_again;
        }
        break;
    }
}
local_intr_restore(intr_flag);
return ret;
}

```

在上述函数中可以看出，如果  $p\_rpos < p\_wpos$ ，则表示有键盘输入的新字符在 `stdin_buffer` 中，于是就从 `stdin_buffer` 中取出新字符放到 `iobuf` 指向的缓冲区中；如果  $p\_rpos \geq p\_wpos$ ，则表明没有新字符，这样调用 `read` 用户态库函数的用户进程就需要采用等待队列的睡眠操作进入睡眠状态，等待键盘输入字符的产生。

键盘输入字符后，如何唤醒等待键盘输入的用户进程呢？回顾 lab1 中的外设中断处理，可以了解到，当用户敲击键盘时，会产生键盘中断，在 `trap_dispatch` 函数中，当识别出中断是键盘中断（中断号为  $IRQ\_OFFSET + IRQ\_KBD$ ）时，会调用 `dev_stdin_write` 函数，来把字符写入到 `stdin_buffer` 中，且会通过等待队列的唤醒操作唤醒正在等待键盘输入的用户进程。

通过对 `dev_stdin_write` 和 `dev_stdin_read` 函数的分析，可以看出这是一个典型的生产者/消费者问题的具体实现。

## 6.2 实现简单的磁盘文件系统

### 6.2.1 实验目标

### 6.2.2 Proj18 概述

## 1. 实现描述

## 2. 项目组成

## 3. 编译运行

### 6.2.3 【原理】磁盘文件概念

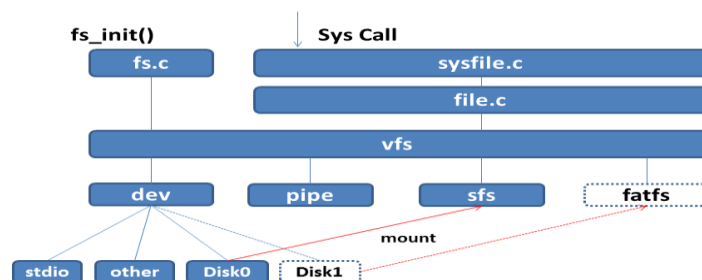
### 6.2.4 【原理】磁盘文件实现

### 6.2.5 【实现】对磁盘文件系统的扩展支持

## 1. Ucore 文件系统结构扩展

ucore模仿UNIX的文件系统设计，对proj17的文件系统架构进行了扩展，实现了简单的磁盘文件系统-SFS，主要集中在下面两层：

- Simple FS文件系统层：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问磁盘外设接口。
- 外设接口层：向上提供device访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动的接口，比如disk设备接口/串口设备接口/键盘设备接口等。



ucore 内核把所有文件都看作是字节流，任何内部逻辑结构都是专用的，由应用程序负责解释。但是 ucore 区分文件的物理结构。ucore 目前支持如下 6 种类型的文件：

- 常规文件：文件中包括的内容信息是由应用程序输入。SFS 文件系统在普通文件上不强加任何内部结构，把其文件内容信息看作为字节。
- 目录：包含一系列的 entry，每个 entry 包含文件名和指向与之相关联的索引节点（index node）的指针。目录是按层次结构组织的。
- 设备文件：不包含数据，但是提供了一个映射物理设备（如串口、键盘等）到一个文件名的机制。可通过设备文件访问外围设备。
- 管道：管道是进程间通讯的一个基础设施。管道缓存了其输入端所接受的数据，以便在管道输出端读的进程能一个先进先出的方式来接受数据。
- 链接文件：实际上一个链接文件是一个已经存在的文件的另一个可选择的文件名。

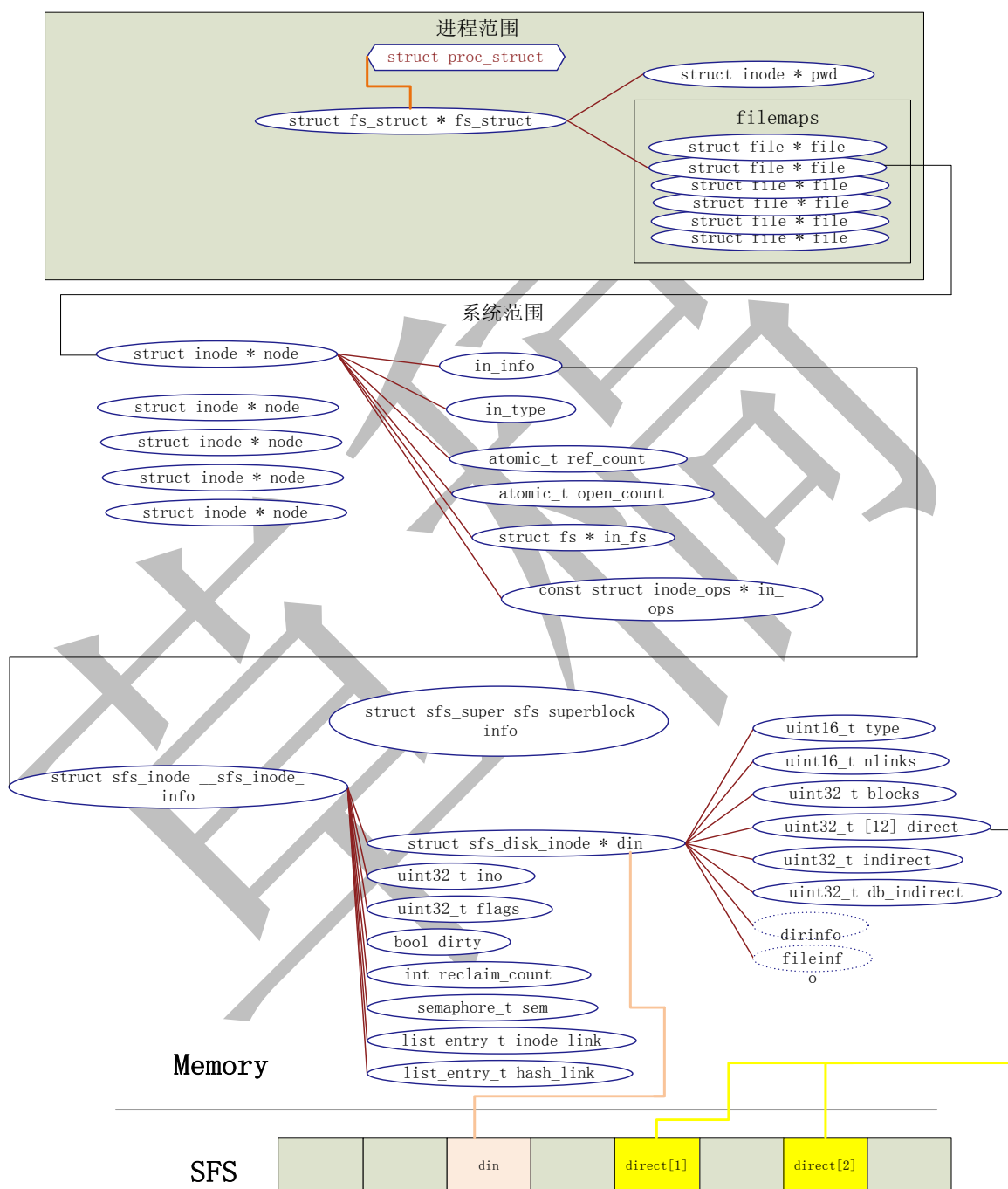
在proj18+中关注的主要是SFS支持的常规文件、目录和链接中的 **hardlink** 的设计实现。SFS 文件系统中目录和常规文件具有共同的属性，而这些属性保存在索引节点中。SFS 通过索引节点来管理目录和常规文件，索引节点包含操作系统所需要的关于某个文件的关键信息，比如文件的属性、访问许可权以及其它控制信息都保存在索引节点中。可以有多个文件名可指向一个索引节点。

## 2. Ucore 文件系统内部组成和相关关系

对照上面的层次我们再大致介绍一下文件系统的访问处理过程，加深对文件系统的总体理解。假如应用程序操作文件（打开/创建/删除/读写），首先需要通过文件系统的通用文件系统访问接口层给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给某一具体文件系统（比如 SFS 文件系统），具体文件系统（Simple FS 文件系统层）把应用程序的访问请求转化为对磁盘上的 block 的处理请求，并通过外设接口层交给磁盘驱动例程来完成具体的磁盘操作。结合用户态写文件函数 `write` 的整个执行过程，我们可以比较清楚地看出 ucore 文件系统架构的层次和依赖关系。



如果一个用户进程打开了一个文件，那么在ucore中涉及的相关数据结构（其中相关数据结构将在下面各个小节中展开叙述）和关系如下图所示：



ucore中文件相关关键数据结构及其关系

## 6.2.6 【实现】基于索引的磁盘文件系统-SFS

### 1. SFS 的磁盘物理布局

文件系统通常保存在磁盘上。在本实验中，第三个磁盘（即 disk0，前两个磁盘分别是 ucore.img 和 swap.img）用于存放一个 SFS 文件系统(Simple Filesystem)。通常文件系统中，磁盘的使用是以扇区（Sector）为单位的，但是为了实现简便，SFS 中以 block（4K，与内存 page 大小相等）为基本单位。

SFS 文件系统的布局如下图所示。

|            |                |         |                                 |
|------------|----------------|---------|---------------------------------|
| superblock | root-dir inode | freemap | Inode/File Data/Dir Data blocks |
|------------|----------------|---------|---------------------------------|

第 0 个块（4K）是超级块（superblock），它包含了关于文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存。

第 1 个块放了一个 root-dir 的 inode，用来记录根目录的相关信息。有关 inode 还将在后续部分介绍。这里只要理解 root-dir 是 SFS 文件系统的根结点，通过这个 root-dir 的 inode 信息就可以定位并查找到根目录下的所有文件信息。

从第 2 个块开始，根据 SFS 中所有块的数量，用 1 个 bit 来表示一个块的占用和未被占用的情况。这个区域称为 SFS 的 freemap 区域，这将占用若干个块空间。为了更好地记录和管理 freemap 区域，专门提供了两个文件 kern/fs/sfs/bitmap.[ch]来完成根据一个块号查找或设置对应的 bit 位的值。

最后在剩余的磁盘空间中，存放了所有其他目录和文件的 inode 信息和内容数据信息。需要注意的是虽然 inode 的大小小于一个块的大小（4096B），但为了实现简单，每个 inode 都占用一个完整的 block。

在 sfs\_fs.c 文件中的 sfs\_do\_mount 函数中，完成了加载位于硬盘上的 SFS 文件系统的超级块 superblock 和 freemap 的工作。这样，在内存中就有了 SFS 文件系统的全局信息。

### 2. SFS 的超级块（Super Block）

#### 磁盘超级块

位于磁盘第 0 块的是 SFS 的超级块（superblock），包含了关于文件系统的所有磁盘上的关键参数。其定义如下：

```
struct sfs_super {
    uint32_t magic;                /* magic number, should be SFS_MAGIC */
    uint32_t blocks;               /* # of blocks in fs */
    uint32_t unused_blocks;        /* # of unused blocks in fs */
    char info[SFS_MAX_INFO_LEN + 1]; /* infomation for sfs */
};
```

可以看到，包含一个成员变量魔数 magic，其值为 0x2f8dbe2a，内核通过它来检查磁盘

镜像是否是合法的 SFS img；成员变量 blocks 记录了 SFS 中所有 block 的数量，即 img 的大小；成员变量 unused\_block 记录了 SFS 中还没有被使用的 block 的数量；成员变量 info 包含了字符串"simple file system"。

## 内存超级块

当 ucore 进行文件系统初始化时，磁盘上的 SFS 超级块的内容就会被装入内存，保存在另外一个数据结构中，即 sfs\_fs

```
/* filesystem for sfs */
struct sfs_fs {
    struct sfs_super super;           /* on-disk superblock */
    struct device *dev;               /* device mounted on */
    struct bitmap *freemap;           /* blocks in use are mared 0 */
    bool super_dirty;                 /* true if super/freemap modified */
    void *sfs_buffer;                 /* buffer for non-block aligned io */
    semaphore_t fs_sem;               /* semaphore for fs */
    semaphore_t io_sem;               /* semaphore for io */
    semaphore_t mutex_sem;            /* semaphore for link/unlink and rename */
    list_entry_t inode_list;           /* inode linked-list */
    list_entry_t *hash_list;          /* inode hash linked-list */
};
```

看到成员变量 **super** 保存的就是磁盘上的超级块内容，此内容会在 ucore 进行 mount SFS 的时候从磁盘的第 0 块拷贝到内存中 sfs\_fs 的 super 成员变量中。dev 成员变量指向了存储 SFS 文件系统的磁盘设备，在这里是 disk0。freemap 成员变量指向了从磁盘读入内存的 freemap 区域。如果内存中的 super 或 freemap 中部分内容被修改了，则 super\_dirty 会置为 1，以便于在做文件系统同步时能够把更新的信息写回磁盘上的超级块或 freemap 区域。fs\_sem 和 io\_sem 用于确保在给进行文件系统或 I/O 操作时的同步互斥；mutex\_sem 用于确保对目录项进行相关操作时的互斥。inode\_list 和 hash\_list 用于把内存中的 inode 节点给链接起来，便于查找或进行其他处理。

## 文件系统操作函数

与文件系统相关的用户态函数主要是 mount 和 unmount，这主要是用于加载文件系统和卸载文件系统。只有在加载了文件系统，把磁盘上的超级块和 freemap 区域读入内存并进行了相关初始化后，才能进行正常的文件访问；在卸载文件系统后，如果对超级块或 freemap 区域进行了修改（即增加或减少了使用的磁盘块数），才会把内存中更新的超级块和 freemap 区域中的内容写回到磁盘上，确保 SFS 文件系统的一致性。下面具体分析一下这两个函数的设计实现。

## 加载文件系统

加载 SFS 文件系统的具体实现是在 sfs\_do\_mount 函数中完成的，其主要工作是为用户进程访问文件系统中的文件或目录做好一切准备工作。为此其工作大致如下所示：

- 1) 首先内部检查确保 SFS 定义磁盘块大小能够包含与 SFS 相关的关键数据结构：磁盘超级块--sfs\_super、磁盘索引节--sfs\_disk\_node、磁盘目录项--sfs\_disk\_entry；

- 2) 然后确保要加载文件系统所在的磁盘定义的磁盘块大小与 SFS 文件系统定义的磁盘块大小一致；
- 3) 分配一个 VFS 层的全局文件系统描述--fs, 然后读取磁盘设备第 0 块保存 SFS 磁盘超级块的内容到 fs.fs\_info.\_\_sfs\_info.super 中；
- 4) 读取磁盘设备第 2 块起保存 free bit map 的区域，并保存到 fs.fs\_info.\_\_sfs\_info.freemap 中；
- 5) 接着初始化了 fs.fs\_info.\_\_sfs\_info 中的信号量 fs\_sem、io\_sem、mutex\_sem 以及索引节点列表 hash\_list 和 inode\_list；
- 6) 最后把 SFS 文件系统相关的函数 sfs\_sync、fs\_get\_root、sfs\_umount、sfs\_cleanup 的地址赋值给 fs 的对应函数指针 fs\_sync、fs\_get\_root、fs\_umount、fs\_cleanup。

在完成上述步骤后，SFS 文件系统中位于磁盘上的整体信息：SFS 标识、SFS 的总磁盘块数、SFS 的空闲磁盘块数、SFS 空闲磁盘块的分布情况等都在全局文件系统描述--fs 中的存了下来，并且设置与 SFS 文件系统级函数的联系。这样为将来使用和卸载 SFS 文件系统做好了必须的准备。

## 卸载文件系统

当不需要访问文件系统或者要 down 机得时候，就需要卸载文件系统，其主要目标是把保存在内存中的全局文件系统描述--fs 中的最新数据和索引节点和文件内容数据等最新的内容都写回到磁盘上后，把这些内存中的数据都释放掉，从而完成对整个文件系统的卸载。

## 3. SFS 的索引节点

### 磁盘索引节点

SFS 中的磁盘索引节点代表了一个实际位于磁盘上的文件。首先我们看看在硬盘上的索引节点的内容：

```
struct sfs_disk_inode {
    union {
        struct {
            uint32_t size;           如果 inode 表示常规文件，则 size 是文件大小
        } fileinfo;
        struct {
            uint32_t slots;          如果 inode 表示目录，则 slots 表示目录下的文件项数
            uint32_t parent;         如果 inode 表示目录，则表示父目录
        } dirinfo;
    };
    uint16_t type;                  inode 的文件类型
    uint16_t nlinks;                此 inode 的硬链接数
    uint32_t blocks;                此 inode 的数据块数的个数
    uint32_t direct[SFS_NDIRECT];  此 inode 的直接数据块索引值（有 SFS_NDIRECT 个）
    uint32_t indirect;              此 inode 的一级间接数据块索引值
    uint32_t db_indirect;            此 inode 的二级间接数据块索引值
};
```

通过上表可以看出，如果 inode 表示的是文件或目录，则成员变量 direct[] 直接指向了保



存文件内容数据的数据块索引值。indirect 间接指向了保存文件内容数据的数据块，indirect 指向的是间接数据块（indirect block），此数据块实际存放的全部是数据块索引，这些数据块索引指向的数据块才被用来存放文件内容数据。db\_indirect 是所谓的 double indirect block，里面存放的全是 indirect block 索引，即二级索引。这样就可以支持很大的文件。

默认的，ucore 里 SFS\_NDIRECT 是 12，即直接索引的数据页大小为  $12 * 4k = 48k$ ；当使用一级间接数据块索引时，ucore 支持最大的文件大小为  $12 * 4k + 1024 * 4k = 48k + 4m$ ；当使用二级间接数据块索引时，ucore 支持最大的文件理论上为  $12 * 4k + 1024 * 4k + 1024 * 1024 * 4k = 48k + 4m + 4g$ ，但是由于 sfs 是一个 32bit 的文件系统，文件内的偏移量是 off\_t 类型，即有符号整数，所以即便使用二级间接索引，sfs 能够支持的最大文件也只能到 2g。

数据索引表内，0 表示一个无效的索引，inode 里 blocks 表示该文件或者目录占用的磁盘的 block 的个数。indiret 为 0 时，表示不使用一级索引块，db\_indirect 为 0 时，表示不使用二级索引块。（因为 block 0 用来保存 super block，它不可能被其他任何文件或目录使用，所以这么设计也是合理的）。

对于普通文件，索引值指向的 block 中保存的是文件中的数据。

而对于目录，索引值指向的数据保存的是目录下所有的文件名以及对应的索引节点所在的索引块（磁盘块）所形成的数组。数据结构如下：

```
/* file entry (on disk) */
struct sfs_disk_entry {
    uint32_t
    索引节点所占数据块索引值
    char name[SFS_MAX_FNAME_LEN + 1];
    文件名
};
ino;
```

操作系统中，每个文件系统下的 inode 都应该分配唯一的 inode 编号。SFS 下，为了实现的简便（偷懒），每个 inode 直接用他所在的磁盘 block 的编号作为 inode 编号。比如，root block 的 inode 编号为 1；每个 sfs\_disk\_entry 数据结构中，name 表示目录下文件或文件夹的名称，ino 表示磁盘 block 编号，通过读取该 block 的数据，能够得到相应的文件或文件夹的 inode。ino 为 0 时，表示一个无效的 entry。

此外，和 inode 相似，每个 sfs\_dirent\_entry 也占用一个 block。

## 内存中的索引节点

```
/* inode for sfs */
struct sfs_inode {
    struct sfs_disk_inode *din;
    uint32_t ino;
    uint32_t flags;
    bool dirty;
    int reclaim_count;
    semaphore_t sem;
    list_entry_t inode_link;
    list_entry_t hash_link;
    sfs_fs */
    /* on-disk inode */
    /* inode number */
    /* inode flags */
    /* true if inode modified */
    /* kill inode if it hits zero */
    /* semaphore for din */
    /* entry for linked-list in sfs_fs */
    /* entry for hash linked-list in
```

```
};
```

可以看到 SFS 中的内存 inode 包含了 SFS 的硬盘 inode 信息，而且还增加了其他一些信息，这属于是便于进行判断是否改写、互斥操作、回收和快速地定位等作用。需要注意，一个内存 inode 是在打开一个文件后才创建的，如果关机则相关信息都会消失。而硬盘 inode 的内容是保存在硬盘中的，只是在进程需要时才被读入到内存中，用于访问文件或目录的具体内容数据

为了方便实现上面提到的多级数据的访问以及目录中 entry 的操作，对 inode SFS 实现了一些辅助的函数：

1. `sfs_bmap_load_nolock`：将对应 `sfs_inode` 的第 `index` 个索引指向的 block 的索引值取出存到相应的指针指向的单元（`ino_store`）。该函数只接受 `index <= inode->blocks` 的参数。当 `index == inode->blocks` 时，该函数理解为需要为 inode 增长一个 block。并标记 inode 为 dirty（所有对 inode 数据的修改都要做这样的操作，这样，当 inode 不再使用的时候，sfs 能够保证 inode 数据能够被写回到磁盘）。`sfs_bmap_load_nolock` 调用的 `sfs_bmap_get_nolock` 来完成相应的操作，阅读 `sfs_bmap_get_nolock`，了解他是如何工作的。（`sfs_bmap_get_nolock` 只由 `sfs_bmap_load_nolock` 调用）

2. `sfs_bmap_truncate_nolock`：将多级数据索引表的最后一个 entry 释放掉。他可以认为是 `sfs_bmap_load_nolock` 中，`index == inode->blocks` 的逆操作。当一个文件或目录被删除时，sfs 会循环调用该函数直到 `inode->blocks` 减为 0，释放所有的数据页。函数通过 `sfs_bmap_free_nolock` 来实现，他应该是 `sfs_bmap_get_nolock` 的逆操作。和 `sfs_bmap_get_nolock` 一样，调用 `sfs_bmap_free_nolock` 也要格外小心。

3. `sfs_dirent_read_nolock`：将目录的第 `slot` 个 entry 读取到指定的内存空间。他通过上面提到的函数来完成。

4. `sfs_dirent_write_nolock`：用指定的 entry 来替换某个目录下的第 `slot` 个 entry。他通过调用 `sfs_bmap_load_nolock` 保证，当第 `slot` 个 entry 不存在时（`slot == inode->blocks`），SFS 会分配一个新的 entry，即在目录尾添加了一个 entry。

5. `sfs_dirent_search_nolock`：是常用的查找函数。他在目录下查找 `name`，并且返回相应的搜索结果（文件或文件夹）的 inode 的编号（也是磁盘编号），和相应的 entry 在该目录的 `index` 编号以及目录下的数据页是否有空闲的 entry。

（SFS 实现里文件的数据页是连续的，不存在任何空洞；而对于目录，数据页不是连续的，当某个 entry 删除的时候，SFS 通过设置 `entry->ino` 为 0 将该 entry 所在的 block 标记为 free，在需要添加新 entry 的时候，SFS 优先使用这些 free 的 entry，其次才会去在数据页尾追加新的 entry。

注意，这些后缀为 `nolock` 的函数，只能在已经获得相应 inode 的 semaphore 才能调用。

## Inode 的文件操作函数

```
static const struct inode_ops sfs_node_fileops = {  
    .vop_magic          = VOP_MAGIC,  
    .vop_open           = sfs_openfile,  
    .vop_close          = sfs_close,
```

```

.vop_read                = sfs_read,
.vop_write               = sfs_write,
.....
};

```

上述 `sfs_openfile`、`sfs_close`、`sfs_read` 和 `sfs_write` 分别对应用户进程发出的 `open`、`close`、`read`、`write` 操作。其中 `sfs_openfile` 不用做什么事；`sfs_close` 需要把对文件的修改内容写回到硬盘上，这样确保硬盘上的文件内容数据是最新的；`sfs_read` 和 `sfs_write` 函数都调用了一个函数 `sfs_io`，并最终通过访问硬盘驱动来完成对文件内容数据的读写。

### Inode 的目录操作函数

```

static const struct inode_ops sfs_node_dirops = {
    .vop_magic            = VOP_MAGIC,
    .vop_open             = sfs_opendir,
    .vop_close            = sfs_close,
    .vop_getdirentry      = sfs_getdirentry,
    .vop_lookup           = sfs_lookup,
    .....
};

```

对于目录操作而言，由于目录也是一种文件，所以 `sfs_opendir`、`sfs_close` 对应户进程发出的 `open`、`close` 函数。相对于 `sfs_open`，`sfs_opendir` 只是完成一些 `open` 函数传递的参数判断，没做其他更多的事情。目录的 `close` 操作与文件的 `close` 操作完全一致。由于目录的内容数据与文件的内容数据不同，所以读出目录的内容数据的函数是 `sfs_getdirentry`，其主要工作是获取目录下的文件 `inode` 信息。

## 6.2.7 【实现】SFS 文件操作

### 1. 打开文件

有了上述分析后，我们可以看看如果一个用户进程打开文件会做些什么事情？首先假定用户进程需要打开的文件已经存在在硬盘上。以 `user/sfs_filetest1.c` 为例，首先用户进程会调用在 `main` 函数中的如下语句：

```
int fd1 = safe_open("/test/testfile", O_RDWR | O_TRUNC);
```

从字面上可以看出，如果 `ucore` 能够正常查找到这个文件，就会返回一个代表文件的文件描述符 `fd1`，这样在接下来的读写文件过程中，就直接用这样 `fd1` 来代表就可以了。那这个打开文件的过程是如何一步一步实现的呢？

#### 通用文件访问接口层的处理流程

首先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：`open->sys_open->syscall`，从而引起系统调用进入到内核态。到了内核态后，通过中断处理例程，会调用到 `sys_open` 内核函数，并进一步调用 `sysfile_open` 内核函数。到了这里，需要把位于用户空间的字符串 `"/test/testfile"` 拷贝到内核空间中的字符串 `path` 中，并进入到文件

系统抽象层的处理流程完成进一步的打开文件操作中。

## 文件系统抽象层的处理流程

### 1. 分配一个空闲的 file 数据结构变量 file

在文件系统抽象层的处理中，首先调用的是 `file_open` 函数，它要给这个即将打开的文件分配一个 `file` 数据结构的变量，这个变量其实是当前进程的打开文件数组 `current->fs_struct->filemap[]` 中的一个空闲元素（即还没用于一个打开的文件），而这个元素的索引值就是最终要返回到用户进程并赋值给变量 `fd1`。到了这一步还仅仅是给当前用户进程分配了一个 `file` 数据结构的变量，还没有找到对应的文件索引节点。

为此需要进一步调用 `vfs_open` 函数来找到 `path` 指出的文件所对应的基于 `inode` 数据结构的 VFS 索引节点 `node`。`vfs_open` 函数需要完成两件事情：通过 `vfs_lookup` 找到 `path` 对应文件的 `inode`；调用 `vop_open` 函数打开文件。

### 2. 找到文件设备的根目录 “/” 的索引节点

需要注意，这里的 `vfs_lookup` 函数是一个针对目录的操作函数，它会调用 `vop_lookup` 函数来找到 SFS 文件系统下的 “/test” 目录下的 “testfile” 文件。为此，`vfs_lookup` 函数首先调用 `get_device` 函数，并进一步调用 `vfs_get_bootfs` 函数（其实调用了）来找到根目录 “/” 对应的 `inode`。这个 `inode` 就是位于 `vfs.c` 中的 `inode` 变量 `bootfs_node`。这个变量在 `init_main` 函数（位于 `kern/process/proc.c`）执行时获得了赋值。

### 3. 找到根目录 “/” 下的 “test” 子目录对应的索引节点

在找到根目录对应的 `inode` 后，通过调用 `vop_lookup` 函数来查找 “/” 和 “test” 这两层目录下的文件 “testfile” 所对应的索引节点，如果找到就返回此索引节点。

### 4. 把 file 和 node 建立联系

完成第 3 步后，将返回到 `file_open` 函数中，通过执行语句 “`file->node=node;`”，就把当前进程的 `current->fs_struct->filemap[fd]`（即 `file` 所指变量）的成员变量 `node` 指针指向了代表 “/test/testfile” 文件的索引节点 `node`。这时返回 `fd`。经过重重回退，通过系统调用返回，用户态的 `syscall->sys_open->open->safe_open` 等用户函数的层层函数返回，最终把 `fd` 赋值给 `fd1`。自此完成了打开文件操作。但这里我们还没有分析第 2 和第 3 步是如何进一步调用 SFS 文件系统提供的函数找位于 SFS 文件系统上的 “/test/testfile” 所对应的 sfs 磁盘 `inode` 的过程。下面需要进一步对此进行分析。

## SFS 文件系统层的处理流程

这里需要分析文件系统抽象层中没有彻底分析的 `vop_lookup` 函数到底做了啥。下面我们来看看。在 `sfs_inode.c` 中的 `sfs_node_dirops` 变量定义了 “`.vop_lookup = sfs_lookup`”，所以我们重点分析 `sfs_lookup` 的实现。

`sfs_lookup` 有三个参数：`node`，`path`，`node_store`。其中 `node` 是根目录 “/” 所对应的 `inode` 节点；`path` 是文件 “testfile” 的绝对路径 “/test/testfile”，而 `node_store` 是经过查找获得的 “testfile” 所对应的 `inode` 节点。

`sfs_lookup` 函数以 “/” 为分割符，从左至右逐一分解 `path` 获得各个子目录和最终文件对应的 `inode` 节点。在本例中是分解出 “test” 子目录，并调用 `sfs_lookup_once` 函数获得 “test” 子目录对应的 `inode` 节点 `subnode`，然后循环进一步调用 `sfs_lookup_once` 查找以 “test” 子

目录下的文件“testfile1”所对应的 inode 节点。当无法分解 path 后,就意味着找到了 testfile1 对应的 inode 节点,就可顺利返回了。

当然这里讲得还比较简单, sfs\_lookup\_once 将调用 sfs\_dirent\_search\_nolock 函数来查找与路径名匹配的目录项,如果找到目录项,则根据目录项中记录的 inode 所处的数据块索引值找到路径名对应的 SFS 磁盘 inode,并读入 SFS 磁盘 inode 对的内容,创建 SFS 内存 inode。

## 2. 关闭文件

### 4.1 分析关闭文件的操作

首先假定要关闭的文件在磁盘上且已打开。用户进程有如下的语句:

```
close(fd);
```

即关闭 fd 对应的文件,下面仔细分析关闭文件的实现。

通用文件访问接口层的处理流程

首先进入通用文件访问接口层的处理流程,即进一步调用如下用户态函数: close->sys\_close->syscall,从而引起系统调用进入到内核态。到了内核态以后,通过中断处理例程,会调用到 sys\_close 内核函数,并进一步调用 sysfile\_close 内核函数,又调用 file\_close 函数,进入到文件系统抽象层处理流程完成进一步关闭文件的操作。

文件系统抽象层的处理流程

#### 1、找到对应的 file 结构

调用 fd2file 函数找到 fd 对应的 file 结构

#### 2、真正关闭文件

调用 filemap\_close 函数真正关闭文件,这个函数把 file 的 status 设为 FD\_CLOSED,然后把文件打开个数减 1,如果没有其他进程打开这个文件了,就调用 filemap\_free 函数释放资源。

在 filemap\_free 函数中,调用 vfs\_close 函数释放对应的 inode,并把 file 的 status 设为 FD\_NONE。

在 vfs\_close 函数中,调用 vop\_open\_dec 函数和 vop\_ref\_dec 函数把 inode 相关计数减 1,完成关闭操作。最后函数调用层层返回至用户程序。

SFS 文件系统层的处理流程

关闭文件时没有涉及到 SFS 文件系统层。

---

`vop_reclaim` 函数对应的是 `sfs_reclaim` 函数，该函数是在某个文件的所有引用都被关闭时调用。如果该文件已经不再被使用，即 `nlinks = 0`，那么占用位图信息会首先被重新赋值成未被使用，否则，应该检查该文件内容是否被改写，如果被改写将文件内容写回磁盘。另外，当该文件不再被使用的时候除了在磁盘中进行改写之外，还需要对内存中的位图进行改写操作。

### 3. 读写文件

#### 4.2 分析读写文件的操作

##### 4.2.1 分析读文件的操作

首先假定文件在磁盘上且已经打开。用户进程有如下语句：

```
read(fd, data, len);
```

即读取 `fd` 对应文件，读取长度为 `len`，存入 `data` 中。下面仔细分析读文件的实现。

通用文件访问接口层的处理流程

先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：`read`→`sys_read`→`syscall`，从而引起系统调用进入到内核态。到了内核态以后，通过中断处理例程，会调用到 `sys_read` 内核函数，并进一步调用 `sysfile_read` 内核函数，进入到文件系统抽象层处理流程完成进一步读文件的操作。

文件系统抽象层的处理流程

##### 1、检查错误

检查读取长度是否为 0 和文件是否可读。

##### 2、分配 buffer 空间

调用 `kmalloc` 函数分配 4096 字节的 `buffer` 空间。

##### 3、实际读取文件

循环读取文件，每次读取 `buffer` 大小。每次循环中，先检查剩余部分大小，若其小于 4096 字节，则只读取剩余部分的大小。然后调用 `file_read` 函数（详细分析见后）将文件内容读取到 `buffer` 中，`alen` 为实际大小。调用 `copy_to_user` 函数将读到的内容拷贝到用户的内存空间中，调整各变量以进行下一次循环读取，直至指定长度读取完成。最后函数调用层层返回至用户程序，用户程序收到了读到的文件内容。

##### 4、`file_read` 函数

这个函数是读文件的核心函数。函数有 4 个参数，`fd` 是文件描述符，`base` 是缓存的基地址，`len` 是要读取的长度，`copied_store` 存放实际读取的长度。

函数首先调用 `fd2file` 函数找到对应的 `file` 结构，并检查是否可读。调用 `filemap_acquire` 函数使打开这个文件的计数加 1。调用 `vop_read` 函数将文件内容读到 `iob` 中（详细分析见后）。调整文件指针偏移量 `pos` 的值，使其向后移动实际读到的字节数 `iobuf_used(iob)`。最后调用 `filemap_release` 函数使打开这个文件的计数减 1，若打开计数为 0，则释放 `file`。

## SFS 文件系统层的处理流程

下面仔细分析一下 `vop_read` 函数做的事情。在 `sfs_inode.c` 中 `sfs_node_fileops` 变量定义了 `.vop_read = sfs_read`，所以重点分析 `sfs_read` 函数的实现。

`sfs_read` 函数调用 `sfs_io` 函数。它有三个参数，`node` 是对应文件的 `inode`，`iob` 是缓存，`write` 表示是读还是写，0 表示读，1 表示写。函数先找到 `inode` 对应 `sfs` 和 `sin`，然后调用 `sfs_io_nolock` 函数进行读取文件操作，最后调用 `iobuf_skip` 函数调整 `iobuf` 的指针。

在 `sfs_io_nolock` 函数中，先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有 `sfs_buf_op = sfs_rbuf`, `sfs_block_op = sfs_rblock`，设置读取的函数操作。接着进行实际操作，先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。每部分中都调用 `sfs_bmap_load_nolock` 函数得到 `blkno` 对应的 `inode` 编号，并调用 `sfs_rbuf` 或 `sfs_rblock` 函数读取数据（中间部分调用 `sfs_rblock`，起始和末尾部分调用 `sfs_rbuf`），调整相关变量。完成后如果 `offset + alen > din->fileinfo.size`（写文件时会出现这种情况，读文件时不会出现这种情况，`alen` 为实际读写的长度），则调整文件大小为 `offset + alen` 并设置 `dirty` 变量。

`sfs_rbuf` 和 `sfs_rblock` 函数最终都调用 `sfs_rwblock_nolock` 函数完成操作，而 `sfs_rwblock_nolock` 函数调用 `dop_io->disk0_io->disk0_read_blks_nolock->ide_read_secs` 完成对磁盘的操作。

`sfs_bmap_load_nolock` 函数将对应 `sfs_inode` 的第 `index` 个索引指向的 `block` 的索引值取出存到相应的指针指向的单元(`ino_store`)。它调用 `sfs_bmap_get_nolock` 来完成相应的操作。

### 4.2.2 分析写文件的操作

首先假定文件在磁盘上且已经打开。用户进程有如下语句：

```
write(fd, data, len);
```

即写入 `fd` 对应文件，写入长度为 `len`，写入数据为 `data`。下面仔细分析写文件的实现。

## 通用文件访问接口层的处理流程

先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：`write->sys_write->syscall`，从而引起系统调用进入到内核态。到了内核态以后，通过中断处理例程，会调用到 `sys_write` 内核函数，并进一步调用 `sysfile_write` 内核函数，进入到文件系统抽象层处理流程完成进一步写文件的操作。

## 文件系统抽象层的处理流程

### 1、检查错误

检查写入长度是否为 0 和文件是否可写。

### 2、分配 buffer 空间

调用 `kmalloc` 函数分配 4096 字节的 `buffer` 空间。

### 3、实际写入文件

循环写入文件，每次写入 `buffer` 大小。每次循环中，先检查剩余部分大小，若其小于 4096 字节，则只写入剩余部分的大小。然后调用 `file_write` 函数（详细分析见后）将 `buffer` 写入到文件中，`alen` 为实际大小。调用 `copy_from_user` 函数将要写的内容拷贝到内核的内存空间中，调整各变量以进行下一次循环写入，直至指定长度写入完成。最后函数调用层层返回至用户程序，用户程序的写入任务完成。

#### 4、file\_write 函数

这个函数是写文件的核心函数。函数有 4 个参数，fd 是文件描述符，base 是缓存的基地址，len 是要写入的长度，copied\_store 存放实际写入的长度。

函数首先调用 fd2file 函数找到对应的 file 结构，并检查是否可写。调用 filemap\_acquire 函数使打开这个文件的计数加 1。调用 vop\_write 函数将 iob 的内容写入文件中（详细分析见后）。调整文件指针偏移量 pos 的值，使其向后移动实际写入的字节数 iobuf\_used(iob)。最后调用 filemap\_release 函数使打开这个文件的计数减 1，若打开计数为 0，则释放 file。

#### SFS 文件系统层的处理流程

下面仔细分析一下 vop\_write 函数做的事情。在 sfs\_inode.c 中 sfs\_node\_fileops 变量定义了.vop\_write = sfs\_write，所以重点分析 sfs\_write 函数的实现。

sfs\_write 函数调用 sfs\_io 函数。它有三个参数，node 是对应文件的 inode，iob 是缓存，write 表示是读还是写，0 表示读，1 表示写。函数先找到 inode 对应 sfs 和 sin，然后调用 sfs\_io\_nolock 函数进行写入文件操作，最后调用 iobuf\_skip 函数调整 iobuf 的指针。

在 sfs\_io\_nolock 函数中，先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有 sfs\_buf\_op = sfs\_wbuf, sfs\_block\_op = sfs\_wblock，设置写入的函数操作。接着进行实际操作，先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。每部分中都调用 sfs\_bmap\_load\_nolock 函数得到 blkno 对应的 inode 编号，并调用 sfs\_wbuf 或 sfs\_wblock 函数写入数据（中间部分调用 sfs\_wblock，起始和末尾部分调用 sfs\_wbuf），调整相关变量。完成后如果 offset + alen > din->fileinfo.size（写文件时会出现这种情况，读文件时不会出现这种情况，alen 为实际读写的长度），则调整文件大小为 offset + alen 并设置 dirty 变量。

sfs\_wbuf 和 sfs\_wblock 函数最终都调用 sfs\_rwblock\_nolock 函数完成操作，而 sfs\_rwblock\_nolock 函数调用 dop\_io->disk0\_io->disk0\_write\_blks\_nolock->ide\_write\_secs 完成对磁盘的操作。

sfs\_bmap\_load\_nolock 函数将对应 sfs\_inode 的第 index 个索引指向的 block 的索引值取出存到相应的指针指向的单元(ino\_store)。它调用 sfs\_bmap\_get\_nolock 来完成相应的操作。

#### 读写文件实现流程：

在系统调用层次上的修改完全与 open 操作相同，这里不再赘述。进入内核态以后，对应的处理函数为 sysfile\_read 和 sysfile\_write。

在 sysfile\_read 函数中，会首先分配 4KB 的 buffer。随后进行一个循环，当剩余的读取字节数不为 0 时，将 alen 设定为 len 和 4KB 中的较小者，alen 即为向文件系统底层请求的读取字节数，经过 file\_read 函数进行读取后，如果实际读取到的字节数 alen 不为 0，那么将读取到的数据复制到用户态空间，并且修改下次开始读取的位置。即下次读取到用户空间的 base + alen 位置，剩余的读取量为 len - alen，已经读取的字节数为 copied + alen。该循环直到出现读取错误或者某次读取的字节数为 0 时结束，结束时需要释放掉 buffer 并且返回读取到的字节数。

在 file\_read 函数中，有 4 个参数，分别是文件描述符，读取到的地址，试图读取长度，以及通过指针方式返回实际的读取长度。该函数进一步调用 vop\_read，vop\_read 会得知一个 iobuf 型的变量指针，在 iobuf 型变量中记录了文件读取的偏移量，并且在每次读取之后，都会将文件描述符中的偏移量加上本次读取到的偏



---

移，从而实现文件的连续读取。

`vop_read` 对应于 `sfs_read`，该函数调用 `sfs_io`，后者进一步调用 `sfs_io_nolock` 进行底层的读取操作。`sfs_io_nolock` 会首先确定读取区间，由于上层保证，该读取区间长度不会超过 4096 字节，并且该区间的右边会取其与文件尾之间的较小值。在确定好区间后进行循环读取，每次读取都是以 4096 进行对齐的读取，因此读取次数并不为 1，该读取过程与 lab1 中涉及的读取 OS 内核的 `bootloader` 方法类似，即调用底层函数进行读取操作。

`sysfile_write` 的过程比较类似，因此这里写的比较简略，在 `sysfile_write` 中同样会根据用户实际给予的缓冲区的大小进行循环地向底层发出写入请求的操作，该操作通过 `file_write` 来实际执行，`file_write` 的参数与 `file_read` 类似。在 `file_write` 的具体操作流程中，同样是通过建立一个 `iobuf` 来通知写入的具体偏移量，并且每次累加当前写入的字节数。之后在 `sfs_write` 中进行底层操作，该函数与 `sfs_read` 调用的都是 `sfs_io`，并最终调用 `sfs_io_nolock`，只不过最后一个参数，写入为 1，读出为 0，因此后面的流程更为类似。不同的是，读入文件的末尾最大只能到达文件尾，而写入的时候如果到达了文件尾，说明当前写入的文件已经比原来的要大，并不能停止写入操作。随后的操作只是一个循环写入的过程，这里可以看到，为了实现不同的功能，`sys_io_nolock` 实际通过了函数指针的方式来实现对于读写操作的统一。

## 6.3 实现支持目录的磁盘文件系统

### 6.3.1 实验目标

实现基于磁盘的文件系统，使得数据可以文件系统中的文件是一种共享资源，需要应对多个用户进程并发访问的需求。由于 `ucore` 在 lab4 完成了同步互斥的功能支持，这为文件系统的正确实现提供了基础。为了更好地实现后续的基于磁盘介质的不同实际文件系统，这里先设计了一个简洁的文件访问接口和一个简单的 `VFS` 层，使得上层应用能够不用关注具体的文件系统是设备、还是其他具体的文件系统，就能够读写某个文件。

### 6.3.2 Proj18.1 概述

#### 1. 实现描述

#### 2. 项目组成

---

proj18

```
|— boot
|   |— asm.h
|   |— bootasm.S
|   |— bootmain.c
|— disk0
|   |— bin
|   |— home
|   |— test
|   |   |— testfile
|   |   |— testman
|   |   |— awk
|   |   |— coreutils
|   |   |   |— cat
|   |   |   |— cp
|   |   |   |— ls
|   |   |   |— mv
|   |   |   |— rm
|   |   |— cpp
|   |   |— gcc
|   |   |— gdb
|   |   |— ld
|   |   |— sed
|   |   |— zsh
|— kern
|   |— debug
|   |   |— assert.h
|   |   |— kdebug.c
|   |   |— kdebug.h
|   |   |— monitor.c
|   |   |— monitor.h
|   |   |— panic.c
|   |   |— stab.h
|   |— driver
|   |   |— clock.c
|   |   |— clock.h
|   |   |— console.c
|   |   |— console.h
|   |   |— ide.c
|   |   |— ide.h
|   |   |— intr.c
|   |   |— intr.h
|   |   |— kbdreg.h
|   |   |— picirq.c
```

```
graph TD;
    picirq.h --- fs;
    fs --- devs;
    devs --- dev.c;
    devs --- dev_disk0.c;
    devs --- dev.h;
    devs --- dev_null.c;
    devs --- dev_stdin.c;
    devs --- dev_stdout.c;
    fs --- file.c;
    fs --- file.h;
    fs --- fs.c;
    fs --- fs.h;
    fs --- iobuf.c;
    fs --- iobuf.h;
    fs --- pipe;
    pipe --- pipe.c;
    pipe --- pipe.h;
    pipe --- pipe_inode.c;
    pipe --- pipe_root.c;
    pipe --- pipe_state.c;
    pipe --- pipe_state.h;
    fs --- sfs;
    sfs --- bitmap.c;
    sfs --- bitmap.h;
    sfs --- sfs.c;
    sfs --- sfs_fs.c;
    sfs --- sfs.h;
    sfs --- sfs_inode.c;
    sfs --- sfs_io.c;
    sfs --- sfs_lock.c;
    fs --- swap;
    swap --- swapfs.c;
    swap --- swapfs.h;
    fs --- sysfile.c;
    fs --- sysfile.h;
    fs --- vfs;
    vfs --- inode.c;
    vfs --- inode.h;
    vfs --- vfs.c;
    vfs --- vfsdev.c;
    vfs --- vfsfile.c;
    vfs --- vfs.h;
    vfs --- vfslookup.c;
```

```
graph TD; init[init] --> entryS[entry.S]; init --> initc[init.c]; libs[libs] --> rbtreeC[rb_tree.c]; libs --> rbtreeH[rb_tree.h]; libs --> readlineC[readline.c]; libs --> stdioC[stdio.c]; libs --> stringC[string.c]; mm[mm] --> buddyPmmC[buddy_pmm.c]; mm --> buddyPmmH[buddy_pmm.h]; mm --> memlayoutH[memlayout.h]; mm --> mmuH[mmu.h]; mm --> pmmC[pmm.c]; mm --> pmmH[pmm.h]; mm --> shmемC[shmем.c]; mm --> shmемH[shmем.h]; mm --> slabC[slab.c]; mm --> slabH[slab.h]; mm --> swapC[swap.c]; mm --> swapH[swap.h]; mm --> vmmC[vmm.c]; mm --> vmmH[vmm.h]; process[process] --> procEntryS[entry.S]; process --> procC[proc.c]; process --> procH[proc.h]; process --> switchS[switch.S]; schedule[schedule] --> schedC[sched.c]; schedule --> schedH[sched.h]; schedule --> schedMLFQC[sched_MLFAQ.c]; schedule --> schedMLFQH[sched_MLFAQ.h]; schedule --> schedRRC[sched_RR.c]; schedule --> schedRRH[sched_RR.h]; sync[sync] --> eventC[event.c]; sync --> eventH[event.h]; sync --> ipcH[ipc.h]; sync --> mboxC[mbox.c]; sync --> mboxH[mbox.h]; sync --> semC[sem.c]
```

---

```

|   |   |—— sem.h
|   |   |—— sync.c
|   |   |—— sync.h
|   |   |—— wait.c
|   |   |—— wait.h
|   |—— syscall
|   |   |—— syscall.c
|   |   |—— syscall.h
|   |—— trap
|   |   |—— trap.c
|   |   |—— trapentry.S
|   |   |—— trap.h
|   |   |—— vectors.S
|—— libs
|   |—— atomic.h
|   |—— dirent.h
|   |—— elf.h
|   |—— error.h
|   |—— hash.c
|   |—— list.h
|   |—— mboxbuf.h
|   |—— printfmt.c
|   |—— rand.c
|   |—— stat.h
|   |—— stdarg.h
|   |—— stdio.h
|   |—— stdlib.h
|   |—— string.c
|   |—— string.h
|   |—— types.h
|   |—— unistd.h
|   |—— x86.h
|—— Makefile
|—— tools
|   |—— function.mk
|   |—— gdbinit
|   |—— grade.sh
|   |—— kernel.ld
|   |—— mksfs.c
|   |—— sign.c
|   |—— user.ld
|   |—— vector.c
|—— user
|   |—— badarg.c

```

---

- |— badbrktest.c
- |— badsegment.c
- |— brkfreetest.c
- |— brktest.c
- |— buggy\_event.c
- |— buggy\_wait2.c
- |— buggy\_wait.c
- |— cowtest.c
- |— divzero.c
- |— eventtest.c
- |— exit.c
- |— faultread.c
- |— faultreadkernel.c
- |— forktest.c
- |— forktree.c
- |— fread\_test2.c
- |— fread\_test.c
- |— fwrite\_test.c
- |— hello2.c
- |— hello.c
- |— libs
  - |— clone.S
  - |— dir.c
  - |— dir.h
  - |— file.c
  - |— file.h
  - |— initcode.S
  - |— lock.h
  - |— malloc.c
  - |— malloc.h
  - |— panic.c
  - |— spipe.c
  - |— spipe.h
  - |— stdio.c
  - |— string.c
  - |— syscall.c
  - |— syscall.h
  - |— thread.c
  - |— thread.h
  - |— ulib.c
  - |— ulib.h
  - |— umain.c
- |— matrix.c
- |— mboxmap.c

---

- |— mboxtest.c
- |— mmaptest.c
- |— nulldevfs\_test.c
- |— pgdir.c
- |— pipetest2.c
- |— pipetest.c
- |— primer2.c
- |— primer3.c
- |— primer.c
- |— sem\_rf.c
- |— semtest2.c
- |— semtest3.c
- |— semtest4.c
- |— semtest.c
- |— sem\_wf.c
- |— sfs\_dirtest1.c
- |— sfs\_filetest1.c
- |— sfs\_filetest2.c
- |— shmemtest.c
- |— sleep.c
- |— sleepkill.c
- |— softint.c
- |— spin.c
- |— spipetest2.c
- |— spipetest.c
- |— swaptest.c
- |— testbss.c
- |— threadfork.c
- |— threadgroup1.c
- |— threadgroup2.c
- |— threadtest.c
- |— threadwork.c
- |— waitkill.c
- |— yield.c

28 directories, 225 files

### 3. 编译运行

---

(THU.CST) os is loading ...

Special kernel symbols:

entry 0xc010002c (phys)  
etext 0xc0123f98 (phys)  
edata 0xc04bcd03 (phys)  
end 0xc04c338c (phys)

Kernel executable memory footprint: 3853KB

memory management: buddy\_pmm\_manager

e820map:

memory: 0009fc00, [00000000, 0009fbff], type = 1.  
memory: 00000400, [0009fc00, 0009ffff], type = 2.  
memory: 00010000, [000f0000, 000fffff], type = 2.  
memory: 07efd000, [00100000, 07ffcfff], type = 1.  
memory: 00003000, [07ffd000, 07ffffff], type = 2.  
memory: 00040000, [fffc0000, ffffffff], type = 2.

check\_alloc\_page() succeeded!

check\_pgdir() succeeded!

check\_boot\_pgdir() succeeded!

----- BEGIN -----

PDE(0e0) c0000000-f8000000 38000000 urw

|-- PTE(38000) c0000000-f8000000 38000000 -rw

PDE(001) fac00000-fb000000 00400000 -rw

|-- PTE(000e0) faf00000-fafe0000 000e0000 urw

|-- PTE(00001) fafeb000-fafec000 00001000 -rw

----- END -----

check\_slab() succeeded!

check\_vma\_struct() succeeded!

check\_pgfault() succeeded!

check\_vmm() succeeded.

sched class: MLFQ\_scheduler

ide 0: 10000(sectors), 'QEMU HARDDISK'.

ide 1: 262144(sectors), 'QEMU HARDDISK'.

ide 2: 262144(sectors), 'QEMU HARDDISK'.

check\_swap() succeeded.

check\_mm\_swap: step1, mm\_map ok.

check\_mm\_swap: step2, mm\_unmap ok.

check\_mm\_swap: step3, exit\_mmap ok.

check\_mm\_swap: step4, dup\_mmap ok.

check\_mm\_swap() succeeded.

check\_mm\_shm\_swap: step1, share memory ok.

check\_mm\_shm\_swap: step2, dup\_mmap ok.

check\_mm\_shm\_swap() succeeded.

sfs: mount: 'simple file system' (200/32568/32768)



---

```
vfs: mount disk0.  
++ setup timer interrupts  
kernel_execve: pid = 3, name = "hello2".  
Hello world!!.  
I am process 3.  
hello2 pass.  
sfs: cleanup: 'simple file system' (200/32568/32768)  
all user-mode processes have quit.  
init check memory pass.  
kernel panic at kern/process/proc.c:580:  
    initproc exit.
```

```
Welcome to the kernel debug monitor!!  
Type 'help' for a list of commands.  
K>
```

### 6.3.3 【原理】目录基本概念

1. 目录功能
2. 目录结构
3. 目录操作

### 6.3.4 【原理】目录实现

### 6.3.5 【实现】SFS 的目录实现

---

### 6.3.6 【实现】SFS 目录操作

4. 打开目录

5. 关闭目录

6. 读写目录

## 6.4 实现 shell

### 6.4.1 实验目标

### 6.4.2 Proj19 概述

1. 实现描述

2. 项目组成

3. 编译运行

---

6.4.3 【原理】在磁盘上组织可运行的文件

6.4.4 【实现】实现 shell

