
2023 级 NKU 操作系统实验指导书

发行版本 1.1.0

NKU 助教团队

2023 年 09 月 05 日

1	欢迎来到 ucore step-by-step 的世界	1
2	lab0: 预备起	3
2.1	lab0: 预备起	3
2.1.1	溯源: ucore 的历史	3
2.1.2	概览: 指导书的结构	4
2.1.3	Linux 安装与熟悉	5
2.1.3.1	安装使用 Linux 实验环境	5
2.1.3.2	使用 Linux	5
2.1.4	实验中可能使用的软件	10
2.1.4.1	编辑器	10
2.1.4.2	<i>exuberant-ctags</i>	11
2.1.4.3	<i>diff & patch</i>	12
2.1.5	开搞: 搭建实验环境	12
2.1.5.1	编译器	12
2.1.5.2	模拟器	13
2.1.5.2.1	安装模拟器 Qemu	13
2.1.5.2.2	使用 OpenSBI	13
2.1.6	调试工具介绍	14
2.1.6.1	gdb 使用	14
2.1.6.2	结合 gdb 和 qemu 源码级调试 ucore	15
2.1.6.2.1	编译可调试的目标文件	15
2.1.6.2.2	使用远程调试	15
3	lab0.5: 比麻雀更小的麻雀 (最小可执行内核)	17
3.1	本章内容	17
3.1.1	实验目的:	17
3.1.2	实验内容	18
3.1.2.1	本节内容	18
3.1.2.1.1	练习	18
3.1.2.1.1.1	练习 1: 使用 GDB 验证启动流程	18
3.1.2.1.2	lab0 项目组成和执行流	19
3.1.2.1.2.1	lab0 的项目组成如下:	19
3.1.2.1.2.2	内核启动	20
3.1.2.1.2.3	设备驱动	20
3.1.2.1.2.4	编译、链接脚本	20
3.1.2.1.2.5	执行流	20

3.1.3	从机器启动到操作系统运行的过程	21
3.1.3.1	本节内容	21
3.1.3.1.1	OpenSBI, bin, elf	21
3.1.3.1.2	内存布局, 链接脚本, 入口点	23
3.1.3.1.3	“真正的”入口点	27
3.1.3.1.4	从 SBI 到 stdio	27
3.1.3.1.5	Just make it	30
3.1.3.1.5.1	make 和 Makefile	30
3.1.3.1.5.2	makefile 的基本规则简介	31
3.1.3.1.5.3	Runing ucore	31
3.1.4	实验报告要求	32
4	lab1: 断, 都可以断	33
4.1	本章内容	33
4.1.1	实验目的:	33
4.1.2	实验目的	33
4.1.2.1	本节内容	34
4.1.2.1.1	练习	34
4.1.2.1.1.1	练习 1: 理解内核启动中的程序入口操作	34
4.1.2.1.1.2	练习 2: 完善中断处理 (需要编程)	34
4.1.2.1.1.3	扩展练习 Challenge1: 描述与理解中断流程	34
4.1.2.1.1.4	扩展练习 Challenge2: 理解上下文切换机制	34
4.1.2.1.1.5	扩展练习 Challenge3: 完善异常中断	35
4.1.2.1.1.6		35
4.1.2.1.2	项目组成和执行流	35
4.1.2.1.2.1	Lab1 项目组成	35
4.1.2.1.2.2	硬件驱动层	36
4.1.2.1.2.3	初始化	36
4.1.2.1.2.4	中断处理	36
4.1.2.1.2.5	执行流	37
4.1.3	中断与中断处理流程	37
4.1.3.1	本节内容	37
4.1.3.1.1	riscv64 中断介绍	37
4.1.3.1.1.1	中断概念	37
4.1.3.1.1.2	中断机制	37
4.1.3.1.1.3	中断分类	38
4.1.3.1.1.4	riscv64 权限模式	38
4.1.3.1.1.5	riscv64 的 M Mode	38
4.1.3.1.1.6	riscv64 的 S Mode	38
4.1.3.1.1.7	寄存器	39
4.1.3.1.1.8	特权指令	40
4.1.3.1.2	掉进兔子洞 (中断入口点)	40
4.1.3.1.3	中断处理程序	44
4.1.3.1.4	滴答滴答 (时钟中断)	49
4.1.4	实验报告要求	52
5	lab2: 物理内存和页表	53
5.1	本章内容	53
5.1.1	实验目的	53
5.1.2	实验内容	53
5.1.2.1	本节内容	54
5.1.2.1.1	练习	54
5.1.2.1.1.1	练习 0: 填写已有实验	54
5.1.2.1.1.2	练习 1: 理解 first-fit 连续物理内存分配算法 (思考题)	54

5.1.2.1.1.3	练习 2: 实现 Best-Fit 连续物理内存分配算法 (需要编程)	54
5.1.2.1.1.4	扩展练习 Challenge: buddy system (伙伴系统) 分配算法 (需要编程)	54
5.1.2.1.1.5	扩展练习 Challenge: 任意大小的内存单元 slab 分配算法 (需要编程)	55
5.1.2.1.1.6	扩展练习 Challenge: 硬件的可用物理内存范围的获取方法 (思考题)	55
5.1.2.1.2	项目组成	55
5.1.3	物理内存管理	57
5.1.3.1	本节内容	57
5.1.3.1.1	基本原理概述	57
5.1.3.1.1.1	物理内存管理	57
5.1.3.1.1.2	物理地址和虚拟地址	57
5.1.3.1.2	实验执行流程概述	58
5.1.3.1.3	以页为单位管理物理内存	58
5.1.3.1.3.1	页表项	58
5.1.3.1.3.2	多级页表	59
5.1.3.1.3.3	页表基址	60
5.1.3.1.4	建立快表以加快访问效率	60
5.1.3.1.5	分页机制的设计思路	61
5.1.3.1.5.1	本小节内容	61
5.1.3.1.5.2	建立段页式管理中需要考虑的关键问题	61
5.1.3.1.5.3	实现分页机制	61
5.1.3.1.6	物理内存管理的设计思路	64
5.1.3.1.6.1	本小节内容	64
5.1.3.1.6.2	物理内存管理的实现	64
5.1.3.1.6.3	物理内存探测的设计思路	68
5.1.3.1.6.4	页面分配算法	72
5.1.4	实验报告要求	75
6	lab3: 缺页异常和页面置换	77
6.1	本章内容	77
6.1.1	实验目的	77
6.1.2	实验内容	77
6.1.2.1	本节内容	78
6.1.2.1.1	练习	78
6.1.2.1.1.1	练习 0: 填写已有实验	78
6.1.2.1.1.2	练习 1: 理解基于 FIFO 的页面替换算法 (思考题)	78
6.1.2.1.1.3	练习 2: 深入理解不同分页模式的工作原理 (思考题)	78
6.1.2.1.1.4	练习 3: 给未被映射的地址映射上物理页 (需要编程)	79
6.1.2.1.1.5	练习 4: 补充完成 Clock 页替换算法 (需要编程)	79
6.1.2.1.1.6	练习 5: 阅读代码和实现手册, 理解页表映射方式相关知识 (思考题)	79
6.1.2.1.1.7	扩展练习 Challenge: 实现不考虑实现开销和效率的 LRU 页替换算法 (需要编程)	79
6.1.2.1.2	项目组成	79
6.1.3	虚拟内存管理	82
6.1.3.1	本节内容	82
6.1.3.1.1	基本原理概述	82
6.1.3.1.1.1	虚拟内存	82
6.1.3.1.2	实验执行流程概述	82
6.1.3.1.3	关键数据结构和相关函数分析	83
6.1.3.1.4	页表项设计思路	85
6.1.3.1.5	使用多级页表实现虚拟存储	87

6.1.3.1.6	处理缺页异常 (page fault 异常)	90
6.1.4	页面置换机制的实现	94
6.1.4.1	本节内容	94
6.1.4.1.1	页替换算法设计思路	94
6.1.4.1.2	页面置换机制设计思路	95
6.1.4.1.3	FIFO 页面置换算法	100
6.1.5	实验报告要求	105
7	lab4: 进程管理	107
7.1	本章内容	107
7.1.1	实验目的	107
7.1.2	实验内容	107
7.1.2.1	提前说明	108
7.1.2.2	本节内容	108
7.1.2.2.1	练习	108
7.1.2.2.1.1	练习 0: 填写已有实验	108
7.1.2.2.1.2	练习 1: 分配并初始化一个进程控制块 (需要编码)	108
7.1.2.2.1.3	练习 2: 为新创建的内核线程分配资源 (需要编码)	108
7.1.2.2.1.4	练习 3: 编写 proc_run 函数 (需要编码)	109
7.1.2.2.1.5	扩展练习 Challenge:	109
7.1.2.2.2	项目组成	109
7.1.3	内核线程管理	111
7.1.3.1	进程与线程	111
7.1.3.2	我们为什么需要进程	112
7.1.3.3	本节内容	112
7.1.3.3.1	实验执行流程概述	112
7.1.3.3.1.1	idle 进程创建	112
7.1.3.3.1.2	第一个内核线程的初始化	113
7.1.3.3.2	PCB	113
7.1.3.3.3	设计关键数据结构	113
7.1.3.3.3.1	进程控制块	113
7.1.3.3.3.2	进程上下文	114
7.1.3.3.4	创建并执行内核线程	115
7.1.3.3.5	创建第 0 个内核线程 idleproc	115
7.1.3.3.6	创建第 1 个内核线程 initproc	116
7.1.3.3.7	调度并执行内核线程 initproc	117
7.1.4	实验报告要求	119
8	lab5: 用户程序	121
8.1	本章内容	121
8.1.1	实验目的	121
8.1.2	实验内容	121
8.1.2.1	本节内容	122
8.1.2.1.1	练习	122
8.1.2.1.1.1	练习 0: 填写已有实验	122
8.1.2.1.1.2	练习 1: 加载应用程序并执行 (需要编码)	122
8.1.2.1.1.3	练习 2: 父进程复制自己的内存空间给予进程 (需要编码)	122
8.1.2.1.1.4	练习 3: 阅读分析源代码, 理解进程执行 fork/exec/wait/exit 的实现, 以及系统调用的实现 (不需要编码)	123
8.1.2.1.1.5	扩展练习 Challenge	123
8.1.2.1.2	项目组成	123
8.1.3	用户进程管理	125
8.1.3.1	本节内容	125
8.1.3.1.1	实验执行流程概述	125

8.1.3.1.2	系统调用 (system call)	125
8.1.3.1.3	从内核线程到用户进程	126
8.1.3.1.4	让用户进程正常运行的用户环境	126
8.1.3.1.5	用户态进程的执行过程分析	127
8.1.3.1.6	用户进程的运行状态分析	128
8.1.3.1.7	创建用户进程	128
8.1.3.1.7.1	1. 应用程序的组成和编译	128
8.1.3.1.7.2	2. 用户进程的虚拟地址空间	129
8.1.3.1.7.3	3. 创建并执行用户进程	130
8.1.3.1.8	进程退出和等待进程	132
8.1.3.1.9	系统调用实现	134
8.1.4	实验报告要求	139
9	lab6 进程调度	141
9.1	本章内容	141
9.1.1	实验目的	141
9.1.2	实验内容	141
9.1.2.1	本节内容	142
9.1.2.1.1	练习	142
9.1.2.1.1.1	练习 0: 填写已有实验	142
9.1.2.1.1.2	练习 1: 使用 Round Robin 调度算法 (不需要编码)	142
9.1.2.1.1.3	练习 2: 实现 Stride Scheduling 调度算法 (需要编码)	142
9.1.2.1.1.4	扩展练习 Challenge 1: 实现 Linux 的 CFS 调度算法	143
9.1.2.1.1.5	扩展练习 Challenge 2: 在 ucore 上实现尽可能多的各种基本调度算法 (FIFO, SJF, ...), 并设计各种测试用例, 能够定量地分析出各种调度算法在各种指标上的差异, 说明调度算法的适用范围。	143
9.1.2.1.2	项目组成	143
9.1.3	调度框架和调度算法设计与实现	146
9.1.3.1	本节内容	146
9.1.3.1.1	进程状态	146
9.1.3.1.2	深入理解进程切换	146
9.1.3.1.3	调度算法框架	147
9.1.3.1.4	RR 调度算法实现	148
9.1.3.1.5	stride 调度算法	149
9.1.3.1.5.1	本小节内容	149
9.1.3.1.5.2	基本思路	149
9.1.3.1.5.3	使用优先队列实现 Stride Scheduling	151
9.1.4	附录: 执行 make qemu 的大致输出	152
10	lab7 同步互斥	153
10.1	本章内容	153
10.1.1	实验目的	153
10.1.2	实验内容	153
10.1.2.1	本节内容	154
10.1.2.1.1	练习	154
10.1.2.1.1.1	练习 0: 填写已有实验	154
10.1.2.1.1.2	练习 1: 理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题 (不需要编码)	154
10.1.2.1.1.3	练习 2: 完成内核级条件变量和基于内核级条件变量的哲学家就餐问题 (需要编码)	154
10.1.2.1.1.4	扩展练习 Challenge 1: 在 ucore 中实现简化的死锁和重入探测机制	155

10.1.2.1.1.5	扩展练习 Challenge 2：参考 Linux 的 RCU 机制，在 ucore 中实现简化的 RCU 机制	155
10.1.2.1.2	项目组成	155
10.1.3	同步互斥的一些基本概念	158
10.1.3.1	时钟中断管理	158
10.1.3.2	屏蔽使能中断	159
10.1.3.3	等待队列：	159
10.1.3.3.1	调用关系举例	160
10.1.4	信号量	162
10.1.5	管程	163
10.1.6	附录：执行 make qemu 的大致输出	165
11	Lab8 文件系统	167
11.1	本章内容	167
11.1.1	实验目的	167
11.1.2	实验内容	167
11.1.2.1	本节内容	168
11.1.2.1.1	练习	168
11.1.2.1.1.1	练习 0：填写已有实验	168
11.1.2.1.1.2	练习 1：完成读文件操作的实现（需要编码）	168
11.1.2.1.1.3	练习 2：完成基于文件系统的执行程序机制的实现（需要编码）	168
11.1.2.1.1.4	扩展练习 Challenge1：完成基于“UNIX 的 PIPE 机制”的设计方案	168
11.1.2.1.1.5	扩展练习 Challenge2：完成基于“UNIX 的软连接和硬连接机制”的设计方案	168
11.1.2.1.2	项目组成	169
11.1.2.1.2.1	Lab8 项目组成	169
11.1.2.1.2.2	Lab8 文件系统初始化过程	173
11.1.3	文件系统	174
11.1.4	文件系统概述	174
11.1.4.1	为啥需要文件/文件系统？	174
11.1.4.2	虚拟文件系统	174
11.1.4.3	UNIX 文件系统	176
11.1.4.4	ucore 文件系统总体介绍	176
11.1.4.5	ucore 文件系统总体结构	178
11.1.5	文件系统抽象层 VFS	180
11.1.5.1	file & dir 接口	180
11.1.5.2	inode 接口	180
11.1.6	硬盘文件系统 SFS	181
11.1.6.1	索引节点	183
11.1.7	设备	187
11.1.7.1	设备的定义	187
11.1.7.2	stdin 设备	189
11.1.7.3	stdout 设备	193
11.1.7.4	disk0 设备	193
11.1.8	open 系统调用的执行过程	195
11.1.8.1	通用文件访问接口层的处理流程	195
11.1.8.2	文件系统抽象层的处理流程	195
11.1.8.3	SFS 文件系统层的处理流程	199
11.1.9	Read 系统调用执行过程	201
11.1.9.1	通用文件访问接口层的处理流程	201
11.1.9.2	文件系统抽象层的处理流程	201
11.1.9.3	SFS 文件系统层的处理流程	203
11.1.10	从 zhong duan 到 zhong duan	205

11.1.10.1	程序的执行	206
11.1.10.2	终端的实现	208
11.1.11	实验报告要求	213
12	附录	215
12.1	本章内容	215
12.1.1	makefile 简介	215
12.1.1.1	makefile 文件的基本结构	215
12.1.1.2	使用宏变量	216
12.1.1.3	控制流和函数	216
12.1.1.4	实验 Makefile 解析	217
12.1.1.5	参考资料	221
12.1.2	附录：【原理】进程的属性与特征解析	221
12.1.2.1	1. 资源管理	221
12.1.2.2	2. 进程状态管理	221
12.1.2.3	3. 进程与线程	222
12.1.3	附录：【原理】用户进程的特征	222
12.1.3.1	从内核线程到用户进程	222
12.1.3.2	让用户进程正常运行的用户环境	223
12.1.3.3	用户态进程的执行过程分析	223
12.1.3.4	用户进程的运行状态分析	224

CHAPTER 1

欢迎来到 ucore step-by-step 的世界

step by step, to the light of u — 《Good Night》

你好，冒险者。让我们一起进入 ucore 的内部，一步一步构建自己的操作系统！

step by step, to the light of you(u)

1. 溯源: ucore 的历史
2. 概览: ucore step by step 指导书的架构
3. 开搞: 工作环境的搭建

2.1 lab0: 预备起

2.1.1 溯源: ucore 的历史

2006 年, MIT 的 Frans Kaashoek 等人参考 PDP-11 上的 UNIX Version 6 写了一个可在 x86 指令集架构上运行的操作系统 xv6 (基于 MIT License)。(cited from [classical ucore](#))

2010 年, 清华大学操作系统教学团队参考 MIT 的教学操作系统 xv6, 开发了在 x86 指令集架构上运行的操作系统 ucore, 多年来作为操作系统课程的实验框架使用, 已经成为了大家口中的”祖传实验”。

ucore 麻雀虽小, 五脏俱全。在不超过 5k 的代码量中包含虚拟内存管理、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, 充分体现了“小而全”的指导思想。

到了如今, x86 指令集架构的问题渐渐开始暴露出来。虽然在 PC 平台上占据绝对主流, 但出于兼容性考虑 x86 架构仍然保留了许多历史包袱, 用于教学的时候总有些累赘。另一方面, 为了更好的和计算机组成原理课程衔接, 将 ucore 移植到 RISC-V 架构势在必行。

趣闻

Intel 曾经在 1989 到 2000 年之间开发过一种 Itanium 处理器, 它基于全新的 IA-64 架构, 但这个指令集/处理器产品线以失败告终。其中一个失败原因就是向后兼容性太差, 2001 年有人测试在 Itanium 处理器上运行原先的 x86 软件时, 性能仅为同时代 x86 奔腾处理器的十分之一。

由图灵奖得主设计的年轻的精简指令集架构 RISC-V(reduced instruction set computer V) 具有诸多优势, 特别是在教学方面。其设计优雅, 开源共享, 没有 x86 指令集那样的历史包袱, 学起来更加轻松, 为计算机组成原理和操作系统课程的教学提供了一种新思路。

趣闻

RISCV 在嵌入式领域异军突起。嵌入式领域的巨头 ARM 公司受到威胁后, 上线了一个网站 riscv-basics.com, 把 RISC-V 批判了一番, 批判的方面包括: 成本、生态系统、碎片化风险、安全性问题、设计验证。但最终迫于业界舆论恶评, ARM 关闭了该网站。(cited from [Wikipedia: RISCV](#))

我们将 ucore 的代码移植到 64 位的 RISC-V 指令集架构, 并借鉴 [rcore tutorial](#) 的做法, 改进实验指导书。通过本教程, 你可以一步一步, 一个模块一个模块地从零开始搭建出一个可以运行简单命令行的操作系统。由于使用了 **step by step** 的组织方式, 每一章都假设你已经完成了前一章的内容阅读和实践, 所以建议完成了前一章的实验学习再进入下一章。

那么还等什么, 我们现在开始吧!

step by step, to the light of ucore!

2.1.2 概览: 指导书的结构

我们沿用 ucore 原先的设计, 将代码模块化, 并归到 8 个 lab 中。还有一个额外的 lab0, 就是你现在正在阅读的部分, 帮助你熟悉 ucore 实验, 完成实验环境的搭建。

从 lab1 到 lab8, 每个 lab 都会解决一个问题。

- lab1: 操作系统怎样处理**中断**?
- lab2: 操作系统怎样分配**物理内存**?
- lab3: 操作系统怎样使用**页表**抽象出**虚拟内存空间**?
- lab4: 操作系统怎样将计算资源和计算任务抽象为”**线程**“?
- lab5: 操作系统怎样创建和管理**用户态的线程/进程**?
- lab6: 操作系统怎样通过**调度算法**, 让众多线程共享一个 CPU?
- lab7: 操作系统怎样管理**并发**带来的风险?
- lab8: 操作系统怎样将存储资源(如”磁盘上的数据“)抽象为”**文件**“?

当你解决了这 8 个问题, 恭喜你, 你在操作系统方面已经有了一个不错的基础。

每个 lab 的实验指导书都按下面的结构编写:

1. 概览。介绍这个 lab 的目标, 代码结构, 主要原理。
2. 详细梳理代码, 但略过繁琐并且和操作系统核心功能无关的部分。可以跟着指导”抄“出一个你自己的 ucore。
3. 练习题。ucore 实验的成绩作为操作系统课程成绩的一部分, 而考核的依据就是同学们对练习题完成情况(包括代码和实验报告)。

在主要文本之外, 我们会在指导书里加入一些”趣闻”, ”须知”和”扩展”。

”趣闻”是为了帮助同学们开阔视野, 也让同学们在学习过程中更加轻松。

”须知”是对做实验有帮助的知识, 供不熟悉的同学参考。如果熟悉, 可以跳过不看。

”扩展”是较为深入的相关知识, 供感兴趣的同学进一步了解。

2.1.3 Linux 安装与熟悉

2.1.3.1 安装使用 Linux 实验环境

这里我们主要以 Ubuntu Linux (14.04 及其以后版本) (64 bit) 作为整个实验的系统软件环境。如果你使用 windows 系统, 推荐在 windows subsystem for linux (WSL) 下进行开发。

2.1.3.2 使用 Linux

在实验过程中, 我们需要了解基于命令行方式的编译、调试、运行操作系统的实验方法。为此, 需要了解基本的 Linux 命令行使用。

命令模式的基本结构和概念 Ubuntu 是图形界面友好和易操作的 linux 发行版, 但有时只需执行几条简单的指令就可以完成繁琐的鼠标点击才能完成的操作。linux 的命令行操作模式功能可以实现你需要的所有操作。简单的说, 命令行就是基于字符命令的用户界面, 也被称为文本操作模式。绝大多数情况下, 用户通过输入一行或多行命令直接与计算机互动, 来实现对计算机的操作。

如何进入命令模式 假设使用默认的图形界面为 GNOME 的任意版本 Ubuntu Linux。点击 GNOME 菜单-> 附件-> 终端, 就可以启动名为 `gnome-terminal` 程序, 从而可以在此软件界面中进行命令行操作。打开 `gnome-terminal` 程序后你首先可能会注意到类似下面的界面:

```
luciferlaker@luciferlaker-VirtualBox:~$
```

你所看到的这些被称为命令终端提示符, 它表示计算机已就绪, 正在等待着用户输入操作指令。以我的屏幕画面为例, “luciferlaker” 是当前所登录的用户名, “VirtualBox” 是这台计算机的主机名, “~” 表示当前目录。此时输入任何指令按回车之后该指令将会提交到计算机运行, 比如你可以输入命令: `ls` 再按下回车:

```
ls [ENTER]
```

注意: [ENTER] 是指输入完 `ls` 后按下回车键, 而不是叫你输入这个单词, `ls` 这个命令将会列出你当前所在目录里的所有文件和子目录列表。

下面介绍 `bash shell` 程序的基本使用方法, 它是 `ubuntu` 缺省的外壳程序。

常用指令

(1) 查询文件列表: (`ls`)

```
luciferlaker@luciferlaker-VirtualBox:~$ ls
file1.txt  file2.txt  file3.txt  tools
```

`ls` 命令默认状态下将按首字母升序列出你当前文件夹下面的所有内容, 但这样直接运行所得到的信息也是比较少的, 通常它可以结合以下这些参数运行以查询更多的信息:

```
ls /                # 将列出根目录 '/' 下的文件清单. 如果给定一个参数, 则命令行会把该参数
→ 当作命令行的工作目录。换句话说, 命令行不再以当前目录为工作目录。
ls -l              # 将给你列出一个更详细的文件清单。
ls -a              # 将列出包括隐藏文件 (以 . 开头的文件) 在内的所有文
```

件. `ls -h` # 将以 KB/MB/GB 的形式给出文件大小, 而不是以纯粹的 Bytes。

(2) 查询当前所在目录: (`pwd`)

```
luciferlaker@luciferlaker-VirtualBox:~$ pwd
/home/luciferlaker
```

(3) 进入其他目录: (`cd`)

```
luciferlaker@luciferlaker-VirtualBox:~$ pwd
/home/luciferlaker
luciferlaker@luciferlaker-VirtualBox:~$ cd /home
luciferlaker@luciferlaker-VirtualBox:/home$ pwd
/home
```

上面例子中, 当前目录原来是/home/luciferlaker, 执行 cd /home 之后再运行 pwd 可以发现, 当前目录已经改为/home 了。

(4) 在屏幕上输出字符: (echo)

```
luciferlaker@luciferlaker-VirtualBox:/home$ echo "Hello World"
Hello World
```

这是一个很有用的命令, 它可以在屏幕上输入你指定的参数 (“ ” 号中的内容), 当然这里举的这个例子中它没有多大的实际意义, 但随着你对 LINUX 指令的不断深入, 就会发现它的价值所在。

(5) 显示文件内容: cat

```
luciferlaker@luciferlaker-VirtualBox:~$ cat file1.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

也可以使用 less 或 more 来显示比较大的文本文件内容。

(6) 复制文件: cp

```
luciferlaker@luciferlaker-VirtualBox:~$ cp file1.txt file1_copy.txt
luciferlaker@luciferlaker-VirtualBox:~$ cat file1_copy.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

(7) 移动文件: mv

```
luciferlaker@luciferlaker-VirtualBox:~$ ls
file1.txt
file2.txt
luciferlaker@luciferlaker-VirtualBox:~$ mv file1.txt new_file.txt
luciferlaker@luciferlaker-VirtualBox:~$ ls
file2.txt
new_file.txt
```

注意: 在命令操作时系统基本上不会给你什么提示, 当然, 绝大多数的命令可以通过加上一个参数-v 来要求系统给出执行命令的反馈信息;

```
luciferlaker@luciferlaker-VirtualBox:~$ mv -v file1.txt new_file.txt
`file1.txt' -> `new_file.txt'
```

(8) 建立一个空文本文件: touch

```
luciferlaker@luciferlaker-VirtualBox:~$ ls
file1.txt
luciferlaker@luciferlaker-VirtualBox:~$ touch tempfile.txt
luciferlaker@luciferlaker-VirtualBox:~$ ls
file1.txt
tempfile.txt
```


(9) 建立一个目录: mkdir

```
luciferlaker@luciferlaker-VirtualBox:~$ ls
file1.txt
tempfile.txt
luciferlaker@luciferlaker-VirtualBox:~$ mkdir test_dir
luciferlaker@luciferlaker-VirtualBox:~$ ls
file1.txt
tempfile.txt
test_dir
```

(10) 删除文件/目录: rm

```
luciferlaker@luciferlaker-VirtualBox:~$ ls -p
file1.txt
tempfile.txt
test_dir/
luciferlaker@luciferlaker-VirtualBox:~$ rm -i tempfile.txt
rm: remove regular empty file `test.txt'? y
luciferlaker@luciferlaker-VirtualBox:~$ ls -p
file1.txt
test_dir/
luciferlaker@luciferlaker-VirtualBox:~$ rm test_dir
rm: cannot remove `test_dir': Is a directory
luciferlaker@luciferlaker-VirtualBox:~$ rm -R test_dir
luciferlaker@luciferlaker-VirtualBox:~$ ls -p
file1.txt
```

在上面的操作: 首先我们通过 `ls` 命令查询可知当前目下有两个文件和一个文件夹;

- [1] 你可以用参数 `-p` 来让系统显示某一项的类型, 比如是文件/文件夹/快捷链接等等;
- [2] 接下来我们用 `rm -i` 尝试删除文件, `-i` 参数是让系统在执行删除操作前输出一条确认提示; `i` (interactive) 也就是交互性的意思;
- [3] 当我们尝试用上面的命令去删除一个文件夹时会得到错误的提示, 因为删除文件夹必须使用 `-R` (recursive, 循环) 参数

特别提示: 在使用命令操作时, 系统假设你很明确自己在做什么, 它不会给你太多的提示, 比如你执行 `rm -Rf /`, 它将会删除你硬盘上所有的东西, 并且不会给你任何提示, 所以, 尽量在使用命令时加上 `-i` 的参数, 以让系统在执行前进行一次确认, 防止你干一些蠢事。如果你觉得每次都要输入 `-i` 太麻烦, 你可以执行以下的命令, 让 `-i` 成为默认参数:

```
alias rm='rm -i'
```

(11) 查询当前进程: ps

```
luciferlaker@luciferlaker-VirtualBox:~$ ps
PID TTY          TIME CMD
21071 pts/1        00:00:00 bash
22378 pts/1        00:00:00 ps
```

这条命令会列出你所启动的所有进程;

```
ps -a          #可以列出系统当前运行的所有进程, 包括由其他用户启动的进程;
ps auxww      #是一条相当人性化的命令, 它会列出除一些很特殊进程以外的所有进程, 并会
               ↪ 以一个高可读的形式显示结果, 每一个进程都会有较为详细的解释;
```

基本命令的介绍就到此为止, 你可以访问网络得到更加详细的 Linux 命令介绍。

控制流程 (1) 输入/输出

input 用来读取你通过键盘（或其他标准输入设备）输入的信息，output 用于在屏幕（或其他标准输出设备）上输出你指定的输出内容。另外还有一些标准的出错提示也是通过这个命令来实现的。通常在遇到操作错误时，系统会自动调用这个命令来输出标准错误提示；

我们能重定向命令中产生的输入和输出流的位置。

(2) 重定向

如果你想把命令产生的输出流指向一个文件而不是（默认的）终端，你可以使用如下的语句：

```
luciferlaker@luciferlaker-VirtualBox:~$ ls >file4.txt
luciferlaker@luciferlaker-VirtualBox:~$ cat file4.txt
file1.txt file2.txt file3.txt
```

以上例子将创建文件 file4.txt 如果 file4.txt 不存在的话。注意：如果 file4.txt 已经存在，那么上面的命令将覆盖文件的内容。如果你想将内容添加到已存在的文件内容的最后，那你可以用下面这个语句：

```
command >> filename
```

示例：

```
luciferlaker@luciferlaker-VirtualBox:~$ ls >> file4.txt
luciferlaker@luciferlaker-VirtualBox:~$ cat file4.txt
file1.txt file2.txt file3.txt
file1.txt file2.txt file3.txt file4.txt
```

在这个例子中，你会发现原有的文件中添加了新的内容。接下来我们会见到另一种重定向方式：我们将把一个文件的内容作为将要执行的命令的输入。以下是这个语句：

```
command < filename
```

示例：

```
luciferlaker@luciferlaker-VirtualBox:~$ cat > file5.txt
a3.txt
a2.txt
file2.txt
file1.txt
<Ctrl-D> # 这表示敲入Ctrl+D键
luciferlaker@luciferlaker-VirtualBox:~$ sort < file5.txt
a2.txt
a3.txt
file1.txt
file2.txt
```

(3) 管道

Linux 的强大之处在于它能把几个简单的命令联合成为复杂的功能，通过键盘上的管道符号‘|’完成。现在，我们来排序上面的”grep”命令：

```
grep -i command < myfile | sort > result.text
```

搜索 myfile 中的命令，将输出分类并写入分类文件到 result.text 。有时候用 ls 列出很多命令的时候很不方便这时 “|” 就充分利用到了 ls -llless 慢慢看吧。

(4) 后台进程

CLI 不是系统的串行接口。您可以在执行其他命令时给出系统命令。要启动一个进程到后台，追加一个 “&” 到命令后面。

```
sleep 60 &
ls
```

睡眠命令在后台运行，您依然可以与计算机交互。除了不同步启动命令以外，最好把 ‘&’ 理解成 ‘;’。

如果您有一个命令将占用很多时间，您想把它放入后台运行，也很简单。只要在命令运行时按下 `ctrl-z`，它就会停止。然后键入 `bg` 使其转入后台。`fg` 命令可使其转回前台。

```
sleep 60
<ctrl-z> # 这表示敲入 Ctrl+Z 键
bg
fg
```

最后，您可以使用 `ctrl-c` 来杀死一个前台进程。

环境变量

特殊变量。PATH, PS1, ...

(1) 不显示中文

可通过执行如下命令避免显示乱码中文。在一个 shell 中，执行：

```
export LANG=""
```

这样在这个 shell 中，output 信息缺省时英文。

获得软件包

(1) 命令行获取软件包

Ubuntu 下可以使用 `apt-get` 命令，`apt-get` 是一条 Linux 命令行命令，适用于 deb 包管理式的操作系统，主要用于自动从互联网软件库中搜索、安装、升级以及卸载软件或者操作系统。一般需要 root 执行权限，所以一般跟随 `sudo` 命令，如：

```
sudo apt-get install gcc [ENTER]
```

常见的以及常用的 `apt` 命令有：

```
apt-get install <package>
    下载 <package> 以及所依赖的软件包，同时进行软件包的安装或者升级。
apt-get remove <package>
    移除 <package> 以及所有依赖的软件包。
apt-cache search <pattern>
    搜索满足 <pattern> 的软件包。
apt-cache show/showpkg <package>
    显示软件包 <package> 的完整描述。
```

例如：

```
luciferlaker@luciferlaker-VirtualBox:~$apt-cache search gcc
gcc-4.8 - The GNU C compiler
gcc-4.8-base - The GNU Compiler Collection (base package)
gcc-4.8-doc - Documentation for the GNU compilers (gcc, gobjc, g++)
gcc-4.8-multilib - The GNU C compiler (multilib files)
gcc-4.8-source - Source of the GNU Compiler Collection
gcc-4.8-locales - The GNU C compiler (native language support files)
chy@chyhome-PC:~$
```

(2) 图形界面软件包获取新立得软件包管理器，是 Ubuntu 下面管理软件包得图形界面程序，相当于命令行中得 `apt` 命令。进入方法可以是

菜单栏 > 系统管理 > 新立得软件包管理器
(System > Administration > Synaptic Package Manager)

使用更新管理器可以通过标记选择适当的软件包进行更新操作。

(3) 配置升级源

Ubuntu 的软件包获取依赖升级源, 可以通过修改 “/etc/apt/sources.list” 文件来修改升级源 (需要 root 权限); 或者修改新立得软件包管理器中 “设置 > 软件库”。

查找帮助文件 Ubuntu 下提供 man 命令以完成帮助手册得查询。man 是 manual 的缩写, 通过 man 命令可以对 Linux 下常用命令、安装软件、以及 C 语言常用函数等进行查询, 获得相关帮助。

例如:

```
luciferlaker@luciferlaker-VirtualBox:~$man printf
PRINTF(1)                                BSD General Commands Manual                                PRINTF(1)

NAME
    printf -- formatted output

SYNOPSIS
    printf format [arguments ...]

DESCRIPTION
    The printf utility formats and prints its arguments, after the first, under contr
    ↳ ol of the format. The format is a character string which contains three types of obj
    ↳ ects: plain characters, which are simply copied to standard output, character escape
    ↳ sequences which are converted and copied to the standard output, and format specifi
    ↳ cations, each of which causes ...

    ...
    The characters and their meanings are as follows:
        \e      Write an <escape> character.
        \a      Write a <bell> character.
    ...
```

通常可能会用到的帮助文件例如:

```
gcc-doc cpp-doc glibc-doc
```

上述帮助文件可以通过 apt-get 命令或者软件包管理器获得。获得以后可以通过 man 命令进行命令或者参数查询。

2.1.4 实验中可能使用的软件

2.1.4.1 编辑器

(1) Ubuntu 下自带的编辑器可以作为代码编辑的工具。例如 gedit 是 gnome 桌面环境下兼容 UTF-8 的文本编辑器。它十分的简单易用, 有良好的语法高亮, 对中文支持很好。通常可以通过双击或者命令行打开目标文件进行编辑。

(2) Vim 编辑器: Vim 是一款极方便的文本编辑软件, 是 UNIX 下的同类型软件 VI 的改进版本。Vim 经常被视为 “专门为程序员打造的文本编辑器”, 功能强大且方便使用, 便于进行程序开发。Ubuntu 下默认安装的 vi 版本较低, 功能较弱, 建议在系统内安装或者升级到最新版本的 Vim。

[1] 关于 Vim 的常用命令以及使用, 可以通过网络进行查找。

[2] 配置文件: Vim 的使用需要配置文件进行设置, 例如:

```

set nocompatible
set encoding=utf-8
set fileencodings=utf-8,chinese
set tabstop=4
set cindent shiftwidth=4
set backspace=indent,eol,start
autocmd FileType c set omnifunc=ccomplete#Complete
autocmd FileType cpp set omnifunc=cppcomplete#Complete
set incsearch
set number
set display=lastline
set ignorecase
syntax on
set nobackup
set ruler
set showcmd
set smartindent
set hlsearch
set cmdheight=1
set laststatus=2
set shortmess=atI
set formatoptions=tcrqn
set autoindent

```

可以将上述配置文件保存到:

```
~/.vimrc
```


注意: `.vimrc` 默认情况下隐藏不可见, 可以在命令行中通过 “`ls -a`” 命令进行查看。如果 ‘`~`’ 目录下不存在该文件, 可以手动创建。修改该文件以后, 重启 Vim 可以使配置生效。

2.1.4.2 *exuberant-ctags*

`exuberant-ctags` 可以为程序语言对象生成索引, 其结果能够被一个文本编辑器或者其他工具简捷迅速的定位。支持的编辑器有 Vim、Emacs 等。实验中, 可以使用命令:

```
ctags -h=.h.c.S -R
```

默认的生成文件为 `tags` (可以通过 `-f` 来指定), 在相同路径下使用 Vim 可以使用改索引文件, 例如:

使用 “`ctrl +]`” 可以跳转到相应的声明或者定义处, 使用 “`ctrl + t`” 返回 (查询堆栈) 等
。

提示: 习惯 GUI 方式的同学, 可采用图形界面的 `understand`、`source insight` 等软件。

2.1.4.3 diff & patch

diff 为 Linux 命令, 用于比较文本或者文件夹差异, 可以通过 man 来查询其功能以及参数的使用。使用 patch 命令可以对文件或者文件夹应用修改。

例如实验中可能会在 proj_b 中应用前一个实验 proj_a 中对文件进行的修改, 可以使用如下命令:

```
diff -r -u -P proj_a_original proj_a_mine > diff.patch
cd proj_b
patch -p1 -u < ../diff.patch
```

注意: proj_a_original 指 proj_a 的源文件, 即未经修改的源码包, proj_a_mine 是修改后的代码包。第一条命令是递归的比较文件夹差异, 并将结果重定向输出到 diff.patch 文件中; 第三条命令是将 proj_a 的修改应用到 proj_b 文件夹中的代码中。

提示: 习惯 GUI 方式的同学, 可采用图形界面的 meld、kdiff3、UltraCompare 等软件。

2.1.5 开搞: 搭建实验环境

说了这么多, 现在该动手了。Make your hands dirty!

方便起见, 可以先在终端里设置一个叫做 **RISCV** 的环境变量 (在 bash 命令里可以通过 ****\$RISCV**** 使用), 作为你安装所有和 riscv 有关的软件的路径。在 /etc/profile 里面写一行 `export RISCV=/your/path/to/riscv` 之类的东西就行。后面安装各个项目最好也放在上面的路径里面。

最小的软件开发环境需要: 能够编译程序, 能够运行程序。开发操作系统这样的系统软件也不例外。

2.1.5.1 编译器

问题在于: 我们使用的计算机都是基于 x86 架构的。如何把程序编译到 riscv64 架构的汇编? 这需要我们使用“目标语言为 riscv64 机器码的编译器”, 在我们的电脑上进行**交叉编译**。

放心, 这里不需要你自己写编译器。我们使用现有的 riscv-gcc 编译器即可。从 <https://github.com/riscv/riscv-gcc> clone 下来, 然后在 x86 架构上编译 riscv-gcc 编译器为可执行的 x86 程序, 就可以运行它, 来把你的程序源代码编译成 riscv 架构的可执行文件了。这有点像绕口令, 但只要有一点编译原理的基础就可以理解。不过, 这个 riscv-gcc 仓库很大, 而且自己编译工具链总是一件麻烦的事。

其实, 没必要那么麻烦, 我们大可以使用别人已经编译好的编译器的可执行文件, 也就是所谓的**预编译 (prebuilt) 工具链**, 下载下来, 放在你喜欢的地方 (比如之前定义的 **\$RISCV**), 配好路径 (把编译器的位置加到系统的 **PATH** 环境变量里), 就能在终端使用了。我们推荐使用 sifive 公司提供的预编译工具链, 进入 <https://d2pn104n81t9m2.cloudfront.net/products/tools/>, 找到 “Prebuilt RISC-V GCC Toolchain and Emulator”, 下载 “GNU Embedded Toolchain” 中适合你的操作系统的版本即可。(注意, 如果你是 wsl, 需要下载适合 ubuntu 版本的编译器) 将 **RISCV/bin** 添加到 **bashrc** 当中, 首先利用 vim 进入 ~/.bashrc 文档, 摁住 ctrl+g, 直接跳到最后一行, 摁一下 i 键, 进入插入模式, 现在可以编辑文档了

```
vim ~/.bashrc
```

我们在 bashrc 的最后添加路径

```
export RISCV=PATH_TO_INSTALL (你RISCV预编译链下载的路径)
export PATH=$RISCV/bin:$PATH
```

路径添加好了, 该关闭 ~/.bashrc 了, 摁一下 esc 键, 退出插入模式; 输入冒号:wq, 关闭 bashrc; 这时候还没有生效! 需要 source 一下, source 命令的含义的博客:

```
source ~/.bashrc
```

配置好后, 在终端输入 `riscv64-unknown-elf-gcc -v` 查看安装的 `gcc` 版本, 如果输出一大堆东西且最后一行有 `gcc version` 某个数字. 某个数字. 某个数字, 说明 `gcc` 配置成功, 否则需要检查一下哪里做错了, 比如环境变量 **PATH** 配置是否正确。一般需要把一个形如 `.../bin` 的目录加到 **PATH** 里。

2.1.5.2 模拟器

如何运行 `riscv64` 的代码? 我们当然可以给大家每个人发一块 `riscv64` 架构处理器的开发板, 再给大家一根 `JTAG` 线, 让大家把程序烧写到上面去跑, 然后各凭本事 `debug` (手动狗头), 但还是使用 **** 模拟器 (emulator) **** 更方便一些。模拟器也就是在 `x86` 架构的计算机上, 通过软件模拟一个 `riscv64` 架构的硬件平台, 从而能够运行 `riscv64` 的目标代码。

我们选择的是 `QEMU` 模拟器。它的优点在于, 内置了一套 `OpenSBI` 固件的实现, 可以简化我们的代码, 也为了和 `rcore tutorial` 保持一致。

下面我们从 `rcore tutorial` 抄写了一段 `qemu` 安装的教程。

2.1.5.2.1 安装模拟器 Qemu

如果你在使用 `Linux (Ubuntu)`, 需要到 `Qemu` 官方网站下载源码并自行编译, 因为 `Ubuntu` 自带的软件包管理器 `apt` 中的 `Qemu` 的版本过低无法使用。参考命令如下:

```
$ wget https://download.qemu.org/qemu-4.1.1.tar.xz
$ tar xvjf qemu-4.1.1.tar.xz
$ cd qemu-4.1.1
$ ./configure --target-list=riscv32-sofmmu,riscv64-sofmmu
$ make -j
$ export PATH=$PWD/riscv32-sofmmu:$PWD/riscv64-sofmmu:$PATH
```

可查看更多详细的安装和使用命令。

如果你在使用 `macOS`, 只需要 `Homebrew` 一个命令即可:

```
$ brew install qemu
```

最后确认一下 `Qemu` 已经安装好, 且版本在 `4.1.0` 以上:

```
$ qemu-system-riscv64 --version
QEMU emulator version 4.1.1
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
```

2.1.5.2.2 使用 OpenSBI

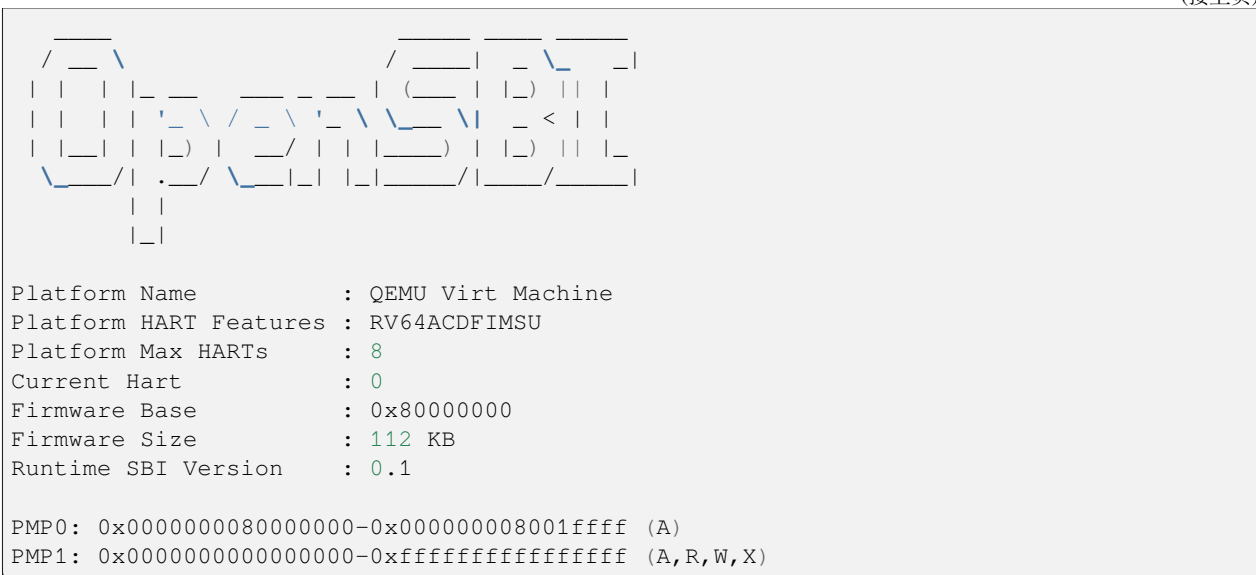
新版 `Qemu` 中内置了 `OpenSBI` 固件 (firmware), 它主要负责在操作系统运行前的硬件初始化和加载操作系统的功能。我们使用以下命令尝试运行一下:

```
$ qemu-system-riscv64 \
  --machine virt \
  --nographic \
  --bios default

OpenSBI v0.4 (Jul  2 2019 11:53:53)
```

(续下页)

(接上页)



可以看到我们已经在 qemu-system-riscv64 模拟的 virt machine 硬件上将 OpenSBI 这个固件跑起来了。Qemu 可以使用 Ctrl+a 再按下 x 退出（注意要松开 Ctrl 再单独按 x）。

如果无法正常使用 Oemu，可以尝试下面这个命令。

```
$ sudo sysctl vm.overcommit_memory=1
```

扩展

如果对 OpenSBI 的内部实现感兴趣，可以看看[RISC-V OpenSBI Deep Dive](#) 介绍文档。

2.1.6 调试工具介绍

2.1.6.1 gdb 使用

gdb 是功能强大的调试程序，可完成如下的调试任务：

- 设置断点
- 监视程序变量的值
- 程序的单步 (step in/step over) 执行
- 显示/修改变量的值
- 显示/修改寄存器
- 查看程序的堆栈情况
- 远程调试
- 调试线程

在可以使用 `gdb` 调试程序之前，必须使用 `-g` 或 `-ggdb` 编译选项编译源文件。运行 `gdb` 调试程序时通常使用如下的命令：

```
gdb progname
```

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有：

- aliases: 命令别名
- breakpoints: 断点定义;
- data: 数据查看;
- files: 指定并查看文件;
- internals: 维护命令;
- running: 程序执行;
- stack: 调用栈查看;
- status: 状态查看;
- tracepoints: 跟踪程序执行。

键入 `help` 后跟命令的分类名, 可获得该类命令的详细清单。gdb 的常用命令如下表所示。

表 gdb 的常用命令

用 gdb 查看源代码可以用 `list` 命令, 但是这个不够灵活。可以使用 `layout src` 命令, 或者按 `Ctrl-X` 再按 `A`, 就会出现一个窗口可以查看源代码。也可以用使用 `-tui` 参数, 这样进入 gdb 里面后就能直接打开代码查看窗口。其他代码窗口相关命令:

2.1.6.2 结合 gdb 和 qemu 源码级调试 ucore

2.1.6.2.1 编译可调试的目标文件

为了使得编译出来的代码是能够被 gdb 这样的调试器调试, 我们需要在使用 gcc 编译源文件的时候添加参数: `-g`。这样编译出来的目标文件中才会包含可以用于调试器进行调试的相关符号信息。

2.1.6.2.2 使用远程调试

为了与 qemu 配合进行源代码级别的调试, 需要先让 qemu 进入等待 gdb 调试器的接入并且还不能让 qemu 中的 CPU 执行, 因此启动 qemu 的时候, 我们需要使用参数 `-S -s` 这两个参数来做到这一点。`-s` 可以使 Qemu 监听本地 TCP 端口 1234 等待 GDB 客户端连接, 而 `-S` 可以使 Qemu 在收到 GDB 的请求后再开始运行。因此, Qemu 暂时没有任何输出。

```
$qemu-system-riscv64 \
    -machine virt \
    -nographic \
    -bios default \
    -device loader,file=${UCOREIMG},addr=0x80200000 \
    -s -S
```

在使用了前面提到的参数启动 qemu 之后, qemu 中的 CPU 并不会马上开始执行, 这时我们启动 gdb, 然后在 gdb 命令行界面下, 使用下面的命令连接到 qemu:

```
riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'
```

`riscv64-unknown-elf-gdb`: 这是 GDB 调试器的可执行文件名。它用于启动 GDB 调试器。

`-ex 'file bin/kernel'`: 这个选项告诉 GDB 加载名为 `bin/kernel` 的目标文件。目标文件通常是编译后的可执行文件或库文件, 这里指定的是一个名为 `kernel` 的文件。

-ex ‘set arch riscv:rv64’ : 这个选项设置 GDB 的目标体系结构为 RISC-V 的 64 位版本 (rv64)。这样, GDB 将使用 RISC-V 体系结构的调试器特性。

-ex ‘target remote localhost:1234’ : 这个选项告诉 GDB 连接到本地主机的端口 1234, GDB 将尝试通过与本地主机的端口 1234 建立连接来远程调试该目标文件。

lab0.5: 比麻雀更小的麻雀 (最小可执行内核)

放了一把火, 烧得只剩个架架

相对于上百万行的现代操作系统 (linux, windows), 几千行的 ucore 是一只”麻雀”。但这只麻雀依然是一只胖麻雀, 我们一眼看不过来几千行的代码。所以, 我们要再做简化, 先用好刀法, 片掉麻雀的血肉, 搞出一个”麻雀骨架”, 看得通透, 再像组装哪吒一样, 把血肉安回去, 变成一个活生生的麻雀。这就是我们的 ucore step-by-step tutorial 的思路。

lab0.5 是 lab1 的预备, 我们构建一个最小的可执行内核 (”麻雀骨架”), 它能够进行格式化的输出, 然后进入死循环。

下面我们就开始解剖麻雀。

3.1 本章内容

3.1.1 实验目的:

实验 0.5 主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行, 它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上, 需要了解 Qemu 模拟器的启动流程, 还需要一些程序内存布局和编译流程 (特别是链接) 相关知识。

本章你将学到:

- 使用链接脚本描述内存布局
- 进行交叉编译生成可执行文件, 进而生成内核镜像
- 使用 OpenSBI 作为 bootloader 加载内核镜像, 并使用 Qemu 进行模拟
- 使用 OpenSBI 提供的服务, 在屏幕上格式化打印字符串用于以后调试

3.1.2 实验内容

实验 0.5 主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行, 它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上, 需要了解 Qemu 模拟器的启动流程, 还需要一些程序内存布局和编译流程 (特别是链接) 相关知识, 以及通过 opensbi 固件来通过服务。

3.1.2.1 本节内容

3.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 ucore_lab 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

3.1.2.1.1.1 练习 1: 使用 GDB 验证启动流程

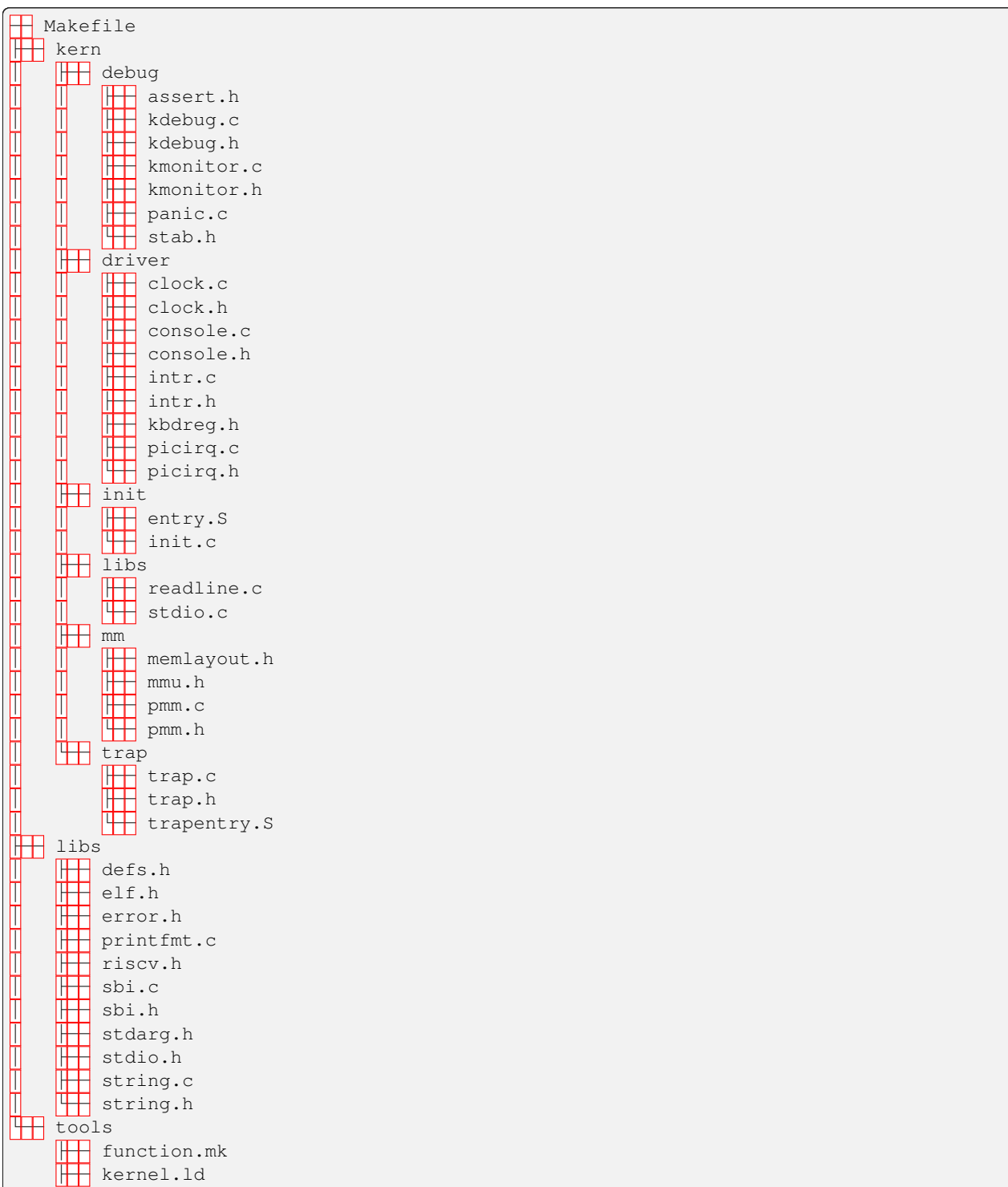
为了熟悉使用 qemu 和 gdb 进行调试工作, 使用 gdb 调试 QEMU 模拟的 RISC-V 计算机加电开始运行到执行应用程序的第一条指令 (即跳转到 0x80200000) 这个阶段的执行过程, 说明 RISC-V 硬件加电后的几条指令在哪里? 完成了哪些功能? 要求在报告中简要写出练习过程和回答。

tips:

- 可以使用示例代码 Makefile 中的 make debug 和 make gdb 指令。
- 一些可能用到的 gdb 指令:
 - x/10i 0x80000000 : 显示 0x80000000 处的 10 条汇编指令。
 - x/10i \$pc : 显示即将执行的 10 条汇编指令。
 - x/10xw 0x80000000 : 显示 0x80000000 处的 10 条数据, 格式为 16 进制 32bit。
 - info register: 显示当前所有寄存器信息。
 - info r t0: 显示 t0 寄存器的值。
 - break funcname: 在目标函数第一条指令处设置断点。
 - break *0x80200000: 在 0x80200000 处设置断点。
 - continue: 执行直到碰到断点。
 - si: 单步执行一条汇编指令。

3.1.2.1.2 lab0 项目组成和执行流

3.1.2.1.2.1 lab0 的项目组成如下:



3.1.2.1.2.2 内核启动

`kern/init/entry.S`: OpenSBI 启动之后将要跳转到的一段汇编代码。在这里进行内核栈的分配, 然后转入 C 语言编写的内核初始化函数。

`kern/init/init.c`: C 语言编写的内核入口点。主要包含 `kern_init()` 函数, 从 `kern/entry.S` 跳转过来完成其他初始化工作。

3.1.2.1.2.3 设备驱动

`kern/driver/console.c(h)`: 在 QEMU 上模拟的时候, 唯一的“设备”是虚拟的控制台, 通过 OpenSBI 接口使用。简单封装了 OpenSBI 的字符读写接口, 向上提供给输入输出库。

库文件

`libs/riscv.h`: 以宏的方式, 定义了 riscv 指令集的寄存器和指令。如果在 C 语言里使用 riscv 指令, 需要通过内联汇编和寄存器的编号。这个头文件把寄存器编号和内联汇编都封装成宏, 使得我们可以用类似函数的方式在 C 语言里执行一句 riscv 指令。

`libs/sbi.c(h)`: 封装 OpenSBI 接口为函数。如果想在 C 语言里使用 OpenSBI 提供的接口, 需要使用内联汇编。这个头文件把 OpenSBI 的内联汇编调用封装为函数。

`libs/defs.h`: 定义了一些常用的类型和宏。例如 `bool` 类型 (C 语言不自带, 这里 `typedef int bool`)。

`libs/string.c(h)`: 一些对字符数组进行操作的函数, 如 `memset()`, `memcpy()` 等, 类似 C 语言的 `string.h`。

`kern/libs/stdio.c`, `libs/readline.c`, `libs/printfmt.c`: 实现了一套标准输入输出, 功能类似于 C 语言的 `printf()` 和 `getchar()`。需要内核为输入输出函数提供两个桩函数 (stub): 输出一个字符的函数, 输入一个字符的函数。在这里, 是 `cons_getc()` 和 `cons_putc()`。

`kern/errors.h`: 定义了一些内核错误类型的宏。

3.1.2.1.2.4 编译、链接脚本

`tools/kernel.ld`: ucore 的链接脚本 (link script), 告诉链接器如何将目标文件的 section 组合为可执行文件。

`tools/function.mk`: 定义 Makefile 中使用的一些函数

Makefile: GNU make 编译脚本

3.1.2.1.2.5 执行流

最小可执行内核的执行流为:

加电 -> OpenSBI 启动 -> 跳转到 `0x80200000` (`kern/init/entry.S`) -> 进入 `kern_init()` 函数 (`kern/init/init.c`) -> 调用 `cprintf()` 输出一行信息 -> 结束

`cprintf()` 函数的执行流为:

接受一个格式化字符串和若干个需要输出的变量作为参数 -> 解析格式化的字符串, 把需要输出的各种变量转化为一串字符 -> 调用 `console.c` 提供的字符输出接口依次输出所有字符 (实际上 `console.c` 又封装了 `sbi.c` 向上提供的 OpenSBI 接口)

3.1.3 从机器启动到操作系统运行的过程

3.1.3.1 本节内容

3.1.3.1.1 OpenSBI, bin, elf

最小可执行内核里, 我们主要完成两件事:

1. 内核的内存布局和入口点设置
2. 通过 sbi 封装好输入输出函数

首先我们回顾计算机的组成:

CPU, 存储设备 (粗略地说, 包括断电后遗失的内存, 和断电后不遗失的硬盘), 输入输出设备, 总线。

QEMU 会帮助我们模拟一块 riscv64 的 CPU, 一块物理内存, 还会借助你的电脑的键盘和显示屏来模拟命令行的输入和输出。虽然 QEMU 不会真正模拟一堆线缆, 但是总线的通信功能也在 QEMU 内部实现了。

还差什么呢? 硬盘。

我们需要硬盘上的程序和数据。比如崭新的 windows 电脑里 C 盘已经被占据的二三十 GB 空间, 除去预装的应用软件, 还有一部分是 windows 操作系统的内核。在插上电源开机之后, 就需要运行操作系统的内核, 然后由操作系统来管理计算机。

问题在于, 操作系统作为一个程序, 必须加载到内存里才能执行。而“把操作系统加载到内存里”这件事情, 不是操作系统自己能做到的, 就好像你不能拽着头发把自己拽离地面。

因此我们可以想象, 在操作系统执行之前, 必然有一个其他程序执行, 他作为“先锋队”, 完成“把操作系统加载到内存”这个工作, 然后他功成身退, 把 CPU 的控制权交给操作系统。

这个“其他程序”, 我们一般称之为 **bootloader**。很好理解: 他负责 boot(开机), 还负责 load(加载 OS 到内存里), 所以叫 bootloader。

简单说明

对于有复杂的读写时序要求的设备, 比如硬盘、U 盘等, CPU 是无法直接读取的, 需要借助驱动程序来告知 CPU 如何与设备进行通信从而读到数据。

而这些驱动程序, 就需要存储在一个不需要驱动程序, CPU 就可以直接读取的地方, 即一块像 Memory 一样, 可以直接使用 load/store 命令可以访问, 又不会担心数据会丢失的区域

在现代计算机中, 有一些设备具备这样的特性, 如早期的 ROM 芯片, 以及后期相对普及的 NOR Flash 芯片

这些芯片与 CPU 连接之前, 首先会进行初始化, 在其中预置好对 CPU 的初始程序, 即 bootloader, 然后再焊接至电路板上

在 CPU 启动时, 会首先运行这些代码, 用这些代码实现对硬盘、内存和其他复杂设备的读取

在 QEMU 模拟的 riscv 计算机里, 我们使用 QEMU 自带的 bootloader: OpenSBI 固件, 那么在 Qemu 开始执行任何指令之前, 首先两个文件将被加载到 Qemu 的物理内存中: 即作为 bootloader 的 OpenSBI.bin 被加载到物理内存以物理地址 0x80000000 开头的区域上, 同时内核镜像 os.bin 被加载到以物理地址 0x80200000 开头的区域上

须知

在计算机中, 固件 (firmware) 是一种特定的计算机软件, 它为设备的特定硬件提供低级控制, 也可以进一步加载其他软件。固件可以为设备更复杂的软件 (如操作系统) 提供标准化的操作环境。对于不太复杂的设备, 固件可以直接充当设备的完整操作系统, 执行所有控制、监视和数据操作功能。在基于 x86 的计算机系统中, BIOS 或 UEFI 是固件; 在基于 riscv 的计算机系统中, OpenSBI 是固件。OpenSBI 运行在 **M 态** (M-mode), 因为固件需要直接访问硬件。

RISCV 有四种特权级 (privilege level)。

Level	Encoding	全称	简称
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved(目前未使用, 保留)	
3	11	Machine	M

粗略的分类:

U-mode 是用户程序、应用程序的特权级, S-mode 是操作系统内核的特权级, M-mode 是固件的特权级。

关于这个地址的选择, 其实有软件工的工作, 也有硬件人的工作。

实现细节

需要解释一下的是, QEMU 模拟的这款 riscv 处理器的复位地址是 0x1000, 而不是 0x80000000

所谓复位地址, 指的是 CPU 在上电的时候, 或者按下复位键的时候, PC 被赋的初始值

这个值的选择会因为厂商的不同而发生变化, 例如, 80386 的复位地址是 0xFFFF0 (因为复位时是 16 位模式, 写成 32 位时也作 0xFFFFFFFF0), MIPS 的复位地址是 0x00000000

RISCV 的设计标准相对灵活, 它允许芯片的实现者自主选择复位地址, 因此不同的芯片会有一些差异。QEMU-4.1.1 是选择了一种实现方式进行模拟

在 QEMU 模拟的这款 riscv 处理器中, 将复位向量地址初始化为 0x1000, 再将 PC 初始化为该复位地址, 因此处理器将从此处开始执行复位代码, 复位代码主要是将计算机系统的各个组件 (包括处理器、内存、设备等) 置于初始状态, 并且会启动 Bootloader, 在这里 QEMU 的复位代码指定加载 Bootloader 的位置为 0x80000000, Bootloader 将加载操作系统内核并启动操作系统的执行。

我们可以想象这样的过程: 操作系统的二进制可执行文件被 OpenSBI 加载到内存中, 然后 OpenSBI 会把 CPU 的“当前指令指针” (pc, program counter) 跳转到内存里的一个位置, 开始执行内存中那个位置的指令。

OpenSBI 怎样知道把操作系统加载到内存的什么位置? 总不能随便选个位置。也许你会觉得可以把操作系统的代码总是加载到固定的位置, 比如总是加载到内存地址最高的地方。

问题在于, 之后 OpenSBI 还要把 CPU 的 program counter 跳转到一个位置, 开始操作系统的执行。如果加载操作系统到内存里的时候随便加载, 那么 OpenSBI 怎么知道把 program counter 跳转到哪里去呢? 难道操作系统的二进制可执行文件需要提供“program counter 跳转到哪里”这样的信息?

实际上, 我们有两种不同的可执行文件格式: elf (e 是 executable 的意思, l 是 linkable 的意思, f 是 format 的意思) 和 bin (binary), 为了正确地与上一阶段的 OpenSBI 对接, 我们需要保证内核的第一条指令位于物理地址 0x80200000 处, 因为这里的代码是地址相关的, 这个地址是由处理器, 即 Qemu 指定的。为此, 我们需要将内核镜像预先加载到 Qemu 物理内存以地址 0x80200000 开头的区域上。一旦 CPU 开始执行内核的第一条指令, 证明计算机的控制权已经被移交给我们的内核。

代码的地址相关性

这里我们需要解释一个小问题, 为什么内核一定要被加载到 0x80200000 的位置开始呢? 加载到别的位置行不行呢?

这要从高级语言被编译成为汇编和机器指令的过程说起了

以一行 C 语言代码为例, `int sum = 0;`, 这句的意义是, 在内存中分配一份可以容纳一个整数的空间, 将其初始化为 0。

那么, 什么时候才会确定 sum 被分配的具体的地址呢? 在编译和链接的时候就会分配, 假设这个地址是 `pa_sum`

接下来, 当出现类似 `sum += 5;` 这样的语句时, 相应的机器指令是, 将 `pa_sum` 位置的值加载入寄存器, 完成加法计算, 再将这个值写回内存中对应的 `pa_sum` 处

而此时生成的指令, 会将 `pa_sum` 的值 (假设为 `0x55aa55aa`), 直接写入到指令的编码中, 如 `load r1 0x55aa55aa`

如此一来, 则这一段代码就成了**地址相关代码**, 即指令中的访存信息在编译完成后即已成为绝对地址, 那么在运行之前, 自然需要将所需要的代码加载到指定的位置

与之相对的, 自然会产生**地址无关代码**的技术需求, 即在加载前才代码确定好被加载的位置, 然后代码就可以在被加载处运行, 这需要编译和加载时做一些处理工作, 同学们可以思考一下这个技术需求的实现思路

`elf` 文件 ([wikipedia: elf](#)) 比较复杂, 包含一个文件头 (ELF header), 包含冗余的调试信息, 指定程序每个 section 的内存布局, 需要解析 `program header` 才能知道各段 (section) 的信息。如果我们已经有一个完整的操作系统来解析 `elf` 文件, 那么 `elf` 文件可以直接执行。但是对于 `OpenSBI` 来说, `elf` 格式还是太复杂了。

`bin` 文件就比较简单了, 简单地在文件头之后解释自己应该被加载到什么起始位置。

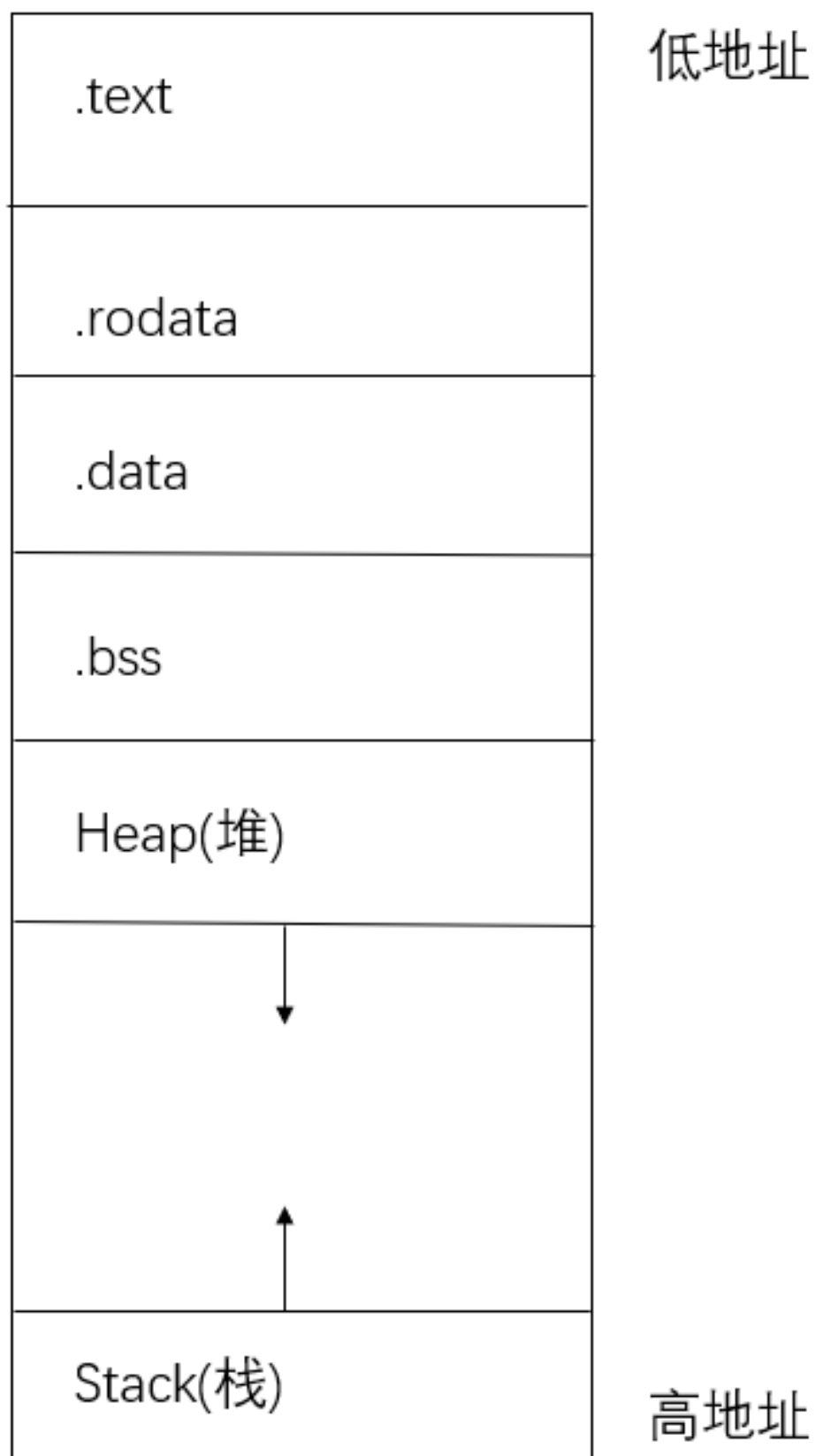
我们举一个例子解释 `elf` 和 `bin` 文件的区别: 初始化为零的一个大数组, 在 `elf` 文件里是 `bss` 数据段的一部分, 只需要记住这个数组的起点和终点就可以了, 等到加载到内存里的时候分配那一段内存。但是在 `bin` 文件里, 那个数组有多大, 有多少个字节的 0, `bin` 文件就要对应有多少个零。所以如果一个程序里声明了一个大全局数组 (默认初始化为 0), 那么可能编译出来的 `elf` 文件只有几 KB, 而生成 `bin` 文件之后却有几 MB, 这是很正常的。实际上, 可以认为 `bin` 文件会把 `elf` 文件指定的每段的内存布局都映射到一块线性的数据里, 这块线性的数据 (或者说程序) 加载到内存里就符合 `elf` 文件之前指定的布局。

那么我们的任务就明确了: 得到内存布局合适的 `elf` 文件, 然后把它转化成 `bin` 文件 (这一步通过 `objcopy` 实现), 然后加载到 `QEMU` 里运行 (`QEMU` 自带的 `OpenSBI` 会干这个活)。下面我们来看如何设置 `elf` 文件的内存布局。

3.1.3.1.2 内存布局, 链接脚本, 入口点

一般来说, 一个程序按照功能不同会分为下面这些段:

`.text` 段, 即代码段, 存放汇编代码; `.rodata` 段, 即只读数据段, 顾名思义里面存放只读数据, 通常是程序中的常量; `.data` 段, 存放被初始化的可读写数据, 通常保存程序中的全局变量; `.bss` 段, 存放被初始化为 00 的可读写数据, 与 `.data` 段的不同之处在于我们知道它要被初始化为 00, 因此在可执行文件中只需记录这个段的大小以及所在位置即可, 而不用记录里面的数据。`stack`, 即栈, 用来存储程序运行过程中的局部变量, 以及负责函数调用时的各种机制。它从高地址向低地址增长; `heap`, 即堆, 用来支持程序运行过程中内存的动态分配, 比如说你要读进来一个字符串, 在你写程序的时候你也不知道它的长度究竟为多少, 于是你只能在运行过程中, 知道了字符串的长度之后, 再在堆中给这个字符串分配内存。内存布局, 也就是指这些段各自所放的位置。一种典型的内存布局如下:



gnu 工具链中, 包含一个链接器 ld

如果你很好奇, 可以看linker script 的详细语法

链接器的作用是把输入文件 (往往是.o 文件) 链接成输出文件 (往往是 elf 文件)。一般来说, 输入文件和输出文件都有很多 section, 链接脚本 (linker script) 的作用, 就是描述怎样把输入文件的 section 映射到输出文件的 section, 同时规定这些 section 的内存布局。

如果你不提供链接脚本, ld 会使用默认的一个链接脚本, 这个默认的链接脚本适合链接出一个能在现有操作系统下运行的应用程序, 但是并不适合链接一个操作系统内核。你可以通过 ld --verbose 来查看默认的链接脚本。

下面给出我们使用的链接脚本

```
/* tools/kernel.ld */

OUTPUT_ARCH(riscv) /* 指定输出文件的指令集架构, 在riscv平台上运行 */
ENTRY(kern_entry) /* 指定程序的入口点, 是一个叫做kern_entry的符号。我们之后会在汇编代
→ 码里定义它 */

BASE_ADDRESS = 0x80200000; /* 定义了一个变量BASE_ADDRESS并初始化 */

/* 链接脚本剩余的部分是一整条SECTIONS指令, 用来指定输出文件的所有SECTION
"." 是SECTIONS指令内的一个特殊变量/计数器, 对应内存里的一个地址。*/
SECTIONS
{
    /* Load the kernel at this address: "." means the current address */
    . = BASE_ADDRESS; /* 对 "." 进行赋值 */
    /* 下面一句的意思是: 从.的当前值 (当前地址) 开始放置一个叫做text的section.
    花括号内部的*(.text.kern_entry .text .stub .text.* .gnu.linkonce.t.*)是正则表
→ 达式
    如果输入文件中有一个section的名称符合花括号内部的格式
    那么这个section就被加到输出文件的text这个section里
    输入文件中section的名称, 有些是编译器自动生成的, 有些是我们自己定义的 */
    .text : {
        *(.text.kern_entry) /* 把输入中kern_entry这一段放到输出中text的开头 */
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }

    PROVIDE(etext = .); /* Define the 'etext' symbol to this value */
    /* read only data, 只读数据, 如程序里的常量 */
    .rodata : {
        *(.rodata .rodata.* .gnu.linkonce.r.*)
    }

    /* 进行地址对齐, 将 "." 增加到 2的0x1000次方的整数倍, 也就是下一个内存页的起始处 */
    . = ALIGN(0x1000);

    .data : {
        *(.data)
        *(.data.*)
    }

    /* small data section, 存储字节数小于某个标准的变量, 一般是char, short等类型的
→ */
    .sdata : {
        *(.sdata)
        *(.sdata.*)
    }
}
```

(续下页)

(接上页)

```

PROVIDE(edata = .);
    /* 初始化为零的数据 */
.bss : {
    *(.bss)
    *(.bss.*)
    *(.sbss*)
}

PROVIDE(end = .);
    /* /DISCARD/表示忽略, 输入文件里 *(.eh_frame .note.GNU-stack)这些section都被忽
    ↪略, 不会加入到输出文件中 */
/DISCARD/ : {
    *(.eh_frame .note.GNU-stack)
}
}

```

趣闻

为什么把初始化为 0（或者说，无需初始化）的数据段称作 bss？

CSAPP 7.4 Relocatable Object files

Aside Why is uninitialized data called .bss? The use of the term .bss to denote uninitialized data is universal. It was originally an acronym for the “block started by symbol” directive from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the .data and .bss sections is to think of “bss” as an abbreviation for “Better Save Space!”

我们在链接脚本里把程序的入口点定义为 kern_entry, 那么我们的程序里需要有一个名称为 kern_entry 的符号。我们在 kern/init/entry.S 编写一段汇编代码, 作为整个内核的入口点。

```

# kern/init/entry.S
#include <mmu.h>
#include <memlayout.h>

# The , "ax", @progbits tells the assembler that the section is allocatable ("a"), execu
    ↪table ("x") and contains data ("@progbits").
# 从这里开始.text 这个section, "ax" 和 %progbits描述这个section的特征
# https://www.nongnu.org/avr-libc/user-manual/mem_sections.html
.section .text, "ax", %progbits
    .globl kern_entry # 使得ld能够看到kern_entry这个符号所在的位置, globl和global同义
    # https://sourceware.org/binutils/docs/as/Global.html#Global
kern_entry:
    la sp, bootstacktop
    tail kern_init
#开始data section
.section .data
    .align PGSHIFT #按照2^PGSHIFT进行地址对齐, 也就是对齐到下一页 PGSHIFT在 mmu.h定义
    .global bootstack #内核栈
bootstack:
    .space KSTACKSIZE #留出KSTACKSIZE这么多个字节的内存
    .global bootstacktop #之后内核栈将要从高地址向低地址增长, 初始时的内核栈为空
bootstacktop:

```

现在这个入口点, 作用就是分配好内核栈, 然后跳转到 kern_init, 看来这个 kern_init 才是我们真正的入口点。下面我们就来看看它。

3.1.3.1.3 “真正的”入口点

我们在 `kern/init/init.c` 编写函数 `kern_init`, 作为“真正的”内核入口点。为了让我们能看到一些效果, 我们希望它能在命令行进行格式化输出。

如果我们在 `linux` 下运行一个 C 程序, 需要格式化输出, 那么大一学生都知道我们应该 `#include<stdio.h>`。于是我们在 `kern/init/init.c` 也这么写一句。且慢! `linux` 下, 当我们调用 C 语言标准库的函数时, 实际上依赖于 `glibc` 提供的运行时环境, 也就是一定程度上依赖于操作系统提供的支持。可是我们并没有把 `glibc` 移植到 `ucore` 里!

怎么办呢? 只能自己动手, 丰衣足食。`QEMU` 里的 `OpenSBI` 固件提供了输入一个字符和输出一个字符的接口, 我们一会把这个接口一层层封装起来, 提供 `stdio.h` 里的格式化输出函数 `cprintf()` 来使用。这里格式化输出函数的名字不使用原先的 `printf()`, 强调这是我们在 `ucore` 里重新实现的函数。

```
// kern/init/init.c
#include <stdio.h>
#include <string.h>
//这里include的头文件, 并不是C语言的标准库, 而是我们自己编写的!

//noreturn 告诉编译器这个函数不会返回
int kern_init(void) __attribute__((noreturn));

int kern_init(void) {
    extern char edata[], end[];
    //这里声明的两个符号, 实际上由链接器ld在链接过程中定义, 所以加了extern关键字
    memset(edata, 0, end - edata);
    //内核运行的时候并没有c标准库可以使用, memset函数是我们自己在string.h定义的

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message); //cprintf是我们自己定义的格式化输出函数
    while (1)
        ;
}
```

接下来就去看看, 我们是怎么从 `OpenSBI` 的接口一层层封装到格式化输入输出函数的。

3.1.3.1.4 从 SBI 到 stdio

`OpenSBI` 作为运行在 M 态的软件 (或者说固件), 提供了一些接口供我们编写内核的时候使用。

我们可以通过 `ecall` 指令 (environment call) 调用 `OpenSBI`。通过寄存器传递给 `OpenSBI` 一个“调用编号”; 如果编号在 0-8 之间, 则由 `OpenSBI` 进行处理, 否则交给我们自己的中断处理程序处理 (暂未实现)。有时 `OpenSBI` 调用需要像函数调用一样传递参数, 这里传递参数的方式也和函数调用一样, 按照 `riscv` 的函数调用约定 (calling convention) 把参数放到寄存器里。可以阅读[SBI 的详细文档](#)。

须知 `ecall`

ecall(environment call), 当我们在 S 态执行这条指令时, 会触发一个 `ecall-from-s-mode-exception`, 从而进入 M 模式中的中断处理流程 (如设置定时器等); 当我们在 U 态执行这条指令时, 会触发一个 `ecall-from-u-mode-exception`, 从而进入 S 模式中的中断处理流程 (常用来进行系统调用)。

C 语言并不能直接调用 `ecall`, 需要通过内联汇编来实现。

```
// libs/sbi.c
#include <sbi.h>
#include <defs.h>
```

(续下页)

(接上页)

```

//SBI编号和函数的对应
uint64_t SBI_SET_TIMER = 0;
uint64_t SBI_CONSOLE_PUTCHAR = 1;
uint64_t SBI_CONSOLE_GETCHAR = 2;
uint64_t SBI_CLEAR_IPI = 3;
uint64_t SBI_SEND_IPI = 4;
uint64_t SBI_REMOTE_FENCE_I = 5;
uint64_t SBI_REMOTE_SFENCE_VMA = 6;
uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;
uint64_t SBI_SHUTDOWN = 8;
//sbi_call函数是我们关注的核心
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n" //mv操作把参数的数值放到寄存器里
        "ecall\n" //参数放好之后, 通过ecall, 交给OpenSBI来执行
        "mv %[ret_val], x10"
        //OpenSBI按照riscv的calling convention,把返回值放到x10寄存器里
        //我们还需要自己通过内联汇编把返回值拿到我们的变量里
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r"
        ↪ (arg2)
        : "memory"
    );
    return ret_val;
}

void sbi_console_putchar(unsigned char ch) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0); //注意这里ch隐式类型转换为int64_t
}

void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}

```

须知函数调用与 calling convention

我们知道, 编译器将高级语言源代码翻译成汇编代码。对于汇编语言而言, 在最简单的编程模型中, 所能够利用的只有指令集中提供的指令、各通用寄存器、CPU 的状态、内存资源。那么, 在高级语言中, 我们进行一次函数调用, 编译器要做哪些工作利用汇编语言来实现这一功能呢?

显然并不是仅用一条指令跳转到被调用函数开头地址就行了。我们还需要考虑:

- 如何传递参数?
- 如何传递返回值?
- 如何保证函数返回后能从我们期望的位置继续执行?

等更多事项。通常编译器按照某种规范去翻译所有的函数调用, 这种规范被称为 **calling convention**。值得一提的是, 为了实现函数调用, 我们需要预先分配一块内存作为 **调用栈**, 后面会看到调用栈在函数调用过程中极其重要。你也可以理解为什么第一章刚开始我们就要分配栈了。

可以参考 **riscv calling convention**

这样我们就可以通过 `sbi_console_putchar()` 来输出一个字符。接下来我们要做的事情就像月饼包装, 把它封了一层又一层。

console.c 只是简单地封装一下

```
// kern/driver/console.c
#include <sbi.h>
#include <console.h>

void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }
```

stdio.c 里面实现了一些函数, 注意我们已经实现了 ucore 版本的 puts 函数: cputs()

```
// kern/libs/stdio.c
#include <console.h>
#include <defs.h>
#include <stdio.h>

/* HIGH level console I/O */

/* *
 * cputch - writes a single character @c to stdout, and it will
 * increase the value of counter pointed by @cnt.
 * */
static void cputch(int c, int *cnt) {
    cons_putc(c);
    (*cnt)++;
}

/* cputchar - writes a single character to stdout */
void cputchar(int c) { cons_putc(c); }

int cputs(const char *str) {
    int cnt = 0;
    char c;
    while ((c = *str++) != '\0') {
        cputch(c, &cnt);
    }
    cputch('\n', &cnt);
    return cnt;
}
```

我们还在 libs/printfmt.c 实现了一些复杂的格式化输入输出函数。最后得到的 cprintf() 函数仍在 kern/libs/stdio.c 定义, 功能和 C 标准库的 printf() 基本相同。

可能你注意到我们用到一个头文件 defs.h, 我们在里面定义了一些有用的宏和类型

```
// libs/defs.h
#ifndef __LIBS_DEFS_H__
#define __LIBS_DEFS_H__
...
/* Represents true-or-false values */
typedef int bool;
/* Explicitly-sized versions of integer types */
typedef char int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short uint16_t;
typedef int int32_t;
typedef unsigned int uint32_t;
typedef long long int64_t;
typedef unsigned long long uint64_t;
```

(续下页)

(接上页)

```

...
/* *
 * Rounding operations (efficient when n is a power of 2)
 * Round down to the nearest multiple of n
 * */
#define ROUNDDOWN(a, n) ({ \
    size_t __a = (size_t)(a); \
    (typeof(a))(__a - __a % (n)); \
})
...
#endif

```

printfmt.c 还依赖一个头文件 `riscv.h`, 这个头文件主要定义了若干和 `riscv` 架构相关的宏, 尤其是将一些内联汇编的代码封装成宏, 使得我们更方便地使用内联汇编来读写寄存器。当然这里我们还没有用到它的强大功能。

```

// libs/riscv.h
...
#define read_csr(reg) ({ unsigned long __tmp; \
    asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
    __tmp; })
//通过内联汇编包装了 csrr 指令为 read_csr() 宏
#define write_csr(reg, val) ({ \
    if (__builtin_constant_p(val) && (unsigned long)(val) < 32) \
        asm volatile ("csrw " #reg ", %0" :: "i"(val)); \
    else \
        asm volatile ("csrw " #reg ", %0" :: "r"(val)); })
...

```

到现在, 我们已经看过了一个最小化的内核的各个部分, 虽然一些部分没有逐行细读, 但我们也知道它在做什么。但一直到现在我们还没进行过编译。下面就把它编译一下跑起来。

3.1.3.1.5 Just make it

我们需要: 编译所有的源代码, 把目标文件链接起来, 生成 `elf` 文件, 生成 `bin` 硬盘镜像, 用 `qemu` 跑起来。这一系列复杂的命令, 我们不想每次用到的时候都敲一遍, 所以我们使用 ~~ 魔改的 ~~~~ 祖传 ~~Makefile。

3.1.3.1.5.1 make 和 Makefile

GNU make(简称 `make`) 是一种代码维护工具, 在大中型项目中, 它将根据程序各个模块的更新情况, 自动的维护和生成目标代码。

`make` 命令执行时, 需要一个 `makefile` (或 `Makefile`) 文件, 以告诉 `make` 命令需要怎么样的去编译和链接程序。我们的 `Makefile` 还依赖 `tools/function.mk`。只要我们的 `makefile` 写得够好, 所有的这一切, 我们只用一个 `make` 命令就可以完成, `make` 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译, 从而自己编译所需要的文件和链接目标程序。

3.1.3.1.5.2 makefile 的基本规则简介

在使用这个 makefile 之前，还是让我们先来粗略地看一看 makefile 的规则。

```
target ... : prerequisites ...
      command
      ...
      ...
```

target 也就是一个目标文件，可以是 object file，也可以是执行文件。还可以是一个标签 (label)。prerequisites 就是，要生成那个 target 所需要的文件或是目标。command 也就是 make 需要执行的命令 (任意的 shell 命令)。这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。如果 prerequisites 中有一个以上的文件比 target 文件要新，那么 command 所定义的命令就会被执行。这就是 makefile 的规则。也就是 makefile 中最核心的内容。

3.1.3.1.5.3 Runing ucore

在源代码的根目录下 `make qemu`, 在 `makefile` 中运行的代码为

```
.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#      $(V) $(QEMU) -kernel $(UCOREIMG) -nographic
#      $(V) $(QEMU) \
#          -machine virt \
#          -nographic \
#          -bios default \
#          -device loader,file=$(UCOREIMG),addr=0x80200000
```

这段代码就是我们启动 `qemu` 的命令，这段代码首先通过宏定义 `$(UCOREIMG)` `$(SWAPIMG)` `$(SFSIMG)` 的函数进行目标文件的构建，然后使用 `qemu` 语句进行 `qemu` 启动加载内核。，我们就把 `ucore` 跑起来了，运行结果如下。

```
+ cc kern/init/init.c
+ cc libs/sbi.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
```

OpenSBI v0.6

[illegible]

```
Platform Name       : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs  : 8
Current Hart        : 0
Firmware Base       : 0x80000000
Firmware Size       : 120 KB
Runtime SBI Version  : 0.2
```

(续下页)

(接上页)

```
MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0     : 0x0000000080000000-0x000000008001ffff (A)
PMP1     : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

它输出一行 (THU.CST) os is loading, 然后进入死循环。

目前为止的代码可以在[这里](#)找到, 遇到困难可以参考。

tips:

- 关于 Makefile 的语法, 如果不熟悉, 可以查看 GNU 手册, 或者这份中文教程: <https://seisman.github.io/how-to-write-makefile/overview.html>
- 对实验中 Makefile 感兴趣的同学可以阅读附录中的 *Makefile*

3.1.4 实验报告要求

从 git server 网站上取得 ucore_lab 后, 进入目录 labcodes/lab0.5, 完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 labcodes/lab0.5 下存放实验报告, 实验报告文档命名为 lab0.5-学堂在线 ID.md。推荐用 **markdown** 格式。对于 lab1 中编程任务, 完成编写之后, 再通过 git push 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

注意有“LAB0.5”的注释, 代码中所有需要完成的地方 (challenge 除外) 都有“LAB0.5”和“YOUR CODE”的注释, 请在提交时特别注意保持注释, 并将“YOUR CODE”替换为自己的学号, 并且将所有标有对应注释的部分填上正确的代码。

lab1: 断, 都可以断

弱水汨其为难兮，路中断而不通。

现在我们要在最小可执行内核的基础上, 支持中断机制, 并且用时钟中断来检验我们的中断处理系统。

4.1 本章内容

4.1.1 实验目的:

实验 1 主要讲解的是中断处理机制。操作系统是计算机系统的监管者, 必须能对计算机系统状态的突发变化做出反应, 这些系统状态可能是程序执行出现异常, 或者是突发的外设请求。当计算机系统遇到突发情况时, 不得不停止当前的正常工作, 应急响应一下, 这是需要操作系统来接管, 并跳转到对应处理函数进行处理, 处理结束后再回到原来的地方继续执行指令。这个过程就是中断处理过程。

本章你将学到:

- riscv 的中断相关知识
- 中断前后如何进行上下文环境的保存与恢复
- 处理最简单的断点中断和时钟中断

4.1.2 实验目的

实验 1 主要讲解的是中断处理机制。通过本章的学习, 我们了解了 riscv 的中断处理机制、相关寄存器与指令。我们知道在中断前后需要恢复上下文环境, 用一个名为中断帧 (TrapFrame) 的结构体存储了要保存的各寄存器, 并用了很大篇幅解释如何通过精巧的汇编代码实现上下文环境保存与恢复机制。最终, 我们通过处理断点和时钟中断验证了我们正确实现了中断机制。

4.1.2.1 本节内容

4.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 ucore_lab 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

4.1.2.1.1.1 练习 1: 理解内核启动中的程序入口操作

阅读 kern/init/entry.S 内容代码, 结合操作系统内核启动流程, 说明指令 `la sp, bootstacktop` 完成了什么操作, 目的是什么? `tail kern_init` 完成了什么操作, 目的是什么?

4.1.2.1.1.2 练习 2: 完善中断处理 (需要编程)

请编程完善 trap.c 中的中断处理函数 `trap`, 在对时钟中断进行处理的部分填写 kern/trap/trap.c 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, 调用 `print_ticks` 子程序, 向屏幕上打印一行文字 “100 ticks”, 在打印完 10 行后调用 `sbi.h` 中的 `shut_down()` 函数关机。

要求完成问题 1 提出的相关函数实现, 提交改进后的源代码包 (可以编译执行), 并在实验报告中简要说明实现过程和定时器中断中断处理的流程。实现要求的部分代码后, 运行整个系统, 大约每 1 秒会输出一行 “100 ticks”, 输出 10 行。

4.1.2.1.1.3 扩展练习 Challenge1: 描述与理解中断流程

回答: 描述 ucore 中处理中断异常的流程 (从异常的产生开始), 其中 `mov a0, sp` 的目的是什么? `SAVE_ALL` 中寄存器保存在栈中的位置是什么确定的? 对于任何中断, `__alltraps` 中都需要保存所有寄存器吗? 请说明理由。

4.1.2.1.1.4 扩增练习 Challenge2: 理解上下文切换机制

回答: 在 `trapentry.S` 中汇编代码 `csrw sscratch, sp; csrrw s0, sscratch, x0` 实现了什么操作, 目的是什么? `save all` 里面保存了 `stval` `scause` 这些 `csr`, 而在 `restore all` 里面却不还原它们? 那这样 `store` 的意义何在呢?

4.1.2.1.1.5 扩展练习 Challenge3: 完善异常中断

编程完善在触发一条非法指令异常 mret 和, 在 kern/trap/trap.c 的异常处理函数中捕获, 并对其进行处理, 简单输出异常类型和异常指令触发地址, 即 “Illegal instruction caught at 0x(地址)”, “ebreak caught at 0x (地址)” 与 “Exception type:Illegal instruction”, “Exception type: breakpoint”。(

tips:

- 参考资料
 - RV 硬件简要手册-中文：重点第 10 章
- 非法指令可以加在任意位置, 比如在通过内联汇编加入, 也可以直接修改汇编。但是要注意, 什么时候异常触发了才会被处理?
- 查阅参考资料, 判断自己触发的异常属于什么类型的, 在相应的情况下进行代码修改。

4.1.2.1.1.6

4.1.2.1.2 项目组成和执行流

4.1.2.1.2.1 Lab1 项目组成

```
lab1
├── Makefile
├── kern
│   ├── debug
│   │   ├── assert.h
│   │   ├── kdebug.c
│   │   ├── kdebug.h
│   │   ├── kmonitor.c
│   │   ├── kmonitor.h
│   │   ├── panic.c
│   │   └── stab.h
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   ├── console.c
│   │   ├── console.h
│   │   ├── intr.c
│   │   └── intr.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   │   └── stdio.c
│   ├── mm
│   │   ├── memlayout.h
│   │   ├── mmu.h
│   │   ├── pmm.c
│   │   └── pmm.h
│   └── trap
│       ├── trap.c
│       ├── trap.h
│       └── trapentry.S
└── lab1.md
```

(续下页)

(接上页)

```
├── libs
│   ├── defs.h
│   ├── error.h
│   ├── printfmt.c
│   ├── readline.c
│   ├── riscv.h
│   ├── sbi.c
│   ├── sbi.h
│   ├── stdarg.h
│   ├── stdio.h
│   ├── string.c
│   └── string.h
├── readme.md
└── tools
    ├── function.mk
    ├── gdbinit
    ├── grade.sh
    ├── kernel.ld
    ├── sign.c
    └── vector.c
```

9 directories, 43 files

只介绍新增的或变动较大的文件。

4.1.2.1.2.2 硬件驱动层

kern/driver/clock.c(h): 通过 OpenSBI 的接口, 可以读取当前时间 (rdtime), 设置时钟事件 (sbi_set_timer), 是时钟中断必需的硬件支持。

kern/driver/intr.c(h): 中断也需要 CPU 的硬件支持, 这里提供了设置中断使能位的接口 (其实只封装了一句 riscv 指令)。

4.1.2.1.2.3 初始化

kern/init/init.c: 需要调用中断机制的初始化函数。

4.1.2.1.2.4 中断处理

kern/trap/trapentry.S: 我们把中断入口点设置为这段汇编代码。这段汇编代码把寄存器的数据挪来挪去, 进行上下文切换。

kern/trap/trap.c(h): 分发不同类型的中断给不同的 handler, 完成上下文切换之后对中断的具体处理, 例如外设中断要处理外设发来的信息, 时钟中断要触发特定的事件。中断处理初始化的函数也在这里, 主要是把中断向量表 (stvec) 设置成所有中断都要跳到 trapentry.S 进行处理。

4.1.2.1.2.5 执行流

内核初始化函数 `kern_init()` 的执行流: (从 `kern/init/entry.S` 进入) -> 输出一些信息说明正在初始化 -> 设置中断向量表 (`stvec`) 跳转到的地方为 `kern/trap/trapentry.S` 里的一个标记 -> 在 `kern/driver/clock.c` 设置第一个时钟事件, 使能时钟中断-> 设置全局的 `S mode` 中断使能位-> 现在开始不断地触发时钟中断

产生一次时钟中断的执行流: `set_sbi_timer()` 通过 OpenSBI 的时钟事件触发一个中断, 跳转到 `kern/trap/trapentry.S` 的 `__alltraps` 标记 -> 保存当前执行流的上下文, 并通过函数调用, 切换为 `kern/trap/trap.c` 的中断处理函数 `trap()` 的上下文, 进入 `trap()` 的执行流。切换前的上下文作为一个结构体, 传递给 `trap()` 作为函数参数 -> `kern/trap/trap.c` 按照中断类型进行分发 (`trap_dispatch()`, `interrupt_handler()`) -> 执行时钟中断对应的处理语句, 累加计数器, 设置下一次时钟中断 -> 完成处理, 返回到 `kern/trap/trapentry.S` -> 恢复原先的上下文, 中断处理结束。

4.1.3 中断与中断处理流程

4.1.3.1 本节内容

4.1.3.1.1 riscv64 中断介绍

4.1.3.1.1.1 中断概念

4.1.3.1.1.2 中断机制

中断 (interrupt) 机制, 就是不管 CPU 现在手里在干啥活, 收到“中断”的时候, 都先放下来去处理其他事情, 处理完其他事情可能再回来干手头的活。

例如, CPU 要向磁盘发一个读取数据的请求, 由于磁盘速度相对 CPU 较慢, 在“发出请求”到“收到磁盘数据”之间会经过很多时间周期, 如果 CPU 干等着磁盘干活就相当于 CPU 在磨洋工。因此我们可以让 CPU 发出读数据的请求后立刻开始干另一件事情。但是, 等一段时间之后, 磁盘的数据取到了, 而 CPU 在干其他的事情, 我们怎么办才能让 CPU 知道之前发出的磁盘请求已经完成了呢? 我们可以让磁盘给 CPU 一个“中断”, 让 CPU 放下手里的事情来接受磁盘的数据。

再比如, 为了保证 CPU 正在执行的程序不会永远运行下去, 我们需要定时检查一下它是否已经运行“超时”。想象有一个程序由于 bug 进入了死循环, 如果 CPU 一直运行这个程序, 那么其他的所有程序都会因为等待 CPU 资源而无法运行, 造成严重的资源浪费。但是检查是否超时, 需要 CPU 执行一段代码, 也就是让 CPU 暂停当前执行的程序。我们不能假设当前执行的程序会主动地定时让出 CPU, 那么就需要 CPU 定时“打断”当前程序的执行, 去进行一些处理, 这通过时钟中断来实现。

从这些描述我们可以看出, 中断机制需要软件硬件一起来支持。硬件进行中断和异常的发现, 然后交给软件来进行处理。回忆一下组成原理课程中学到的各个控制寄存器以及他们的用途 (下一小节会进行简单回顾), 这些寄存器构成了重要的**硬件/软件接口**。由此, 我们也可以得到在一般 OS 中进行中断处理支持的方法:

- 编写相应的中断处理代码
- 在启动中正确设置控制寄存器
- CPU 捕获异常
- 控制转交给相应中断处理代码进行处理
- 返回正在运行的程序

4.1.3.1.1.3 中断分类

异常 (Exception), 指在执行一条指令的过程中发生了错误, 此时我们通过中断来处理错误。最常见的异常包括: 访问无效内存地址、执行非法指令 (除零)、发生缺页等。他们有的可以恢复 (如缺页), 有的不可恢复 (如除零), 只能终止程序执行。

陷入 (Trap), 指我们主动通过一条指令停下来, 并跳转到处理函数。常见的形式有通过 `ecall` 进行系统调用 (`syscall`), 或通过 `ebreak` 进入断点 (`breakpoint`)。

外部中断 (Interrupt), 简称中断, 指的是 CPU 的执行过程被外设发来的信号打断, 此时我们必须先停下来对该外设进行处理。典型的有定时器倒计时结束、串口收到数据等。

外部中断是异步 (asynchronous) 的, CPU 并不知道外部中断将何时发生。CPU 也并不需要一直在原地等着外部中断的发生, 而是执行代码, 有了外部中断才去处理。我们知道, CPU 的主频远高于 I/O 设备, 这样避免了 CPU 资源的浪费。

由于中断处理需要进行较高权限的操作, 中断处理程序一般处于**内核态**, 或者说, 处于“比被打断的程序更高的特权级”。注意, 在 RISC-V 里, 中断 (interrupt) 和异常 (exception) 统称为“trap”。

扩展

The RISC-V Instruction Set Manual Volume I: Unprivileged ISA (Document Version 20191213)

1.6

We use the term **exception** to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.

We use the term **interrupt** to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term **trap** to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.

4.1.3.1.1.4 riscv64 权限模式

4.1.3.1.1.5 riscv64 的 M Mode

M-mode(机器模式, 缩写为 M 模式) 是 RISC-V 中 hart(hardware thread, 硬件线程) 可以执行的最高权限模式。在 M 模式下运行的 hart 对内存、I/O 和一些对于启动和配置系统来说必要的底层功能有着完全的使用权。默认情况下, 发生所有异常 (不论在什么权限模式下) 的时候, 控制权都会被移交到 M 模式的异常处理程序。它是唯一所有标准 RISC-V 处理器都必须实现的权限模式。

4.1.3.1.1.6 riscv64 的 S Mode

S-mode(监管者模式, 缩写为 S 模式) 是支持现代类 Unix 操作系统的权限模式, 支持基于页面的虚拟内存机制是其核心。Unix 系统中的大多数例外都应该进行 S 模式下的系统调用。M 模式的异常处理程序可以将异常重新导向 S 模式, 也支持通过异常委托机制 (Machine Interrupt Delegation, 机器中断委托) 选择性地将中断和同步异常直接交给 S 模式处理, 而完全绕过 M 模式。

4.1.3.1.1.7 寄存器

除了 32 个通用寄存器之外, RISC-V 架构还有大量的 **控制状态寄存器 Control and Status Registers(CSRs)**。其中有几个重要的寄存器和中断机制有关。

有些时候, 禁止 CPU 产生中断很有用。(就像你在做重要的事情, 如操作系统 lab 的时候, 并不想被打断)。所以, `sstatus` 寄存器 (Supervisor Status Register) 里面有一个二进制位 `SIE`(supervisor interrupt enable, 在 RISC-V 标准里是 2^1 对应的二进制位), 数值为 0 的时候, 如果当程序在 S 态运行, 将禁用全部中断。(对于在 U 态运行的程序, `SIE` 这个二进制位的数值没有任何意义), `sstatus` 还有一个二进制位 `UIE`(user interrupt enable) 可以在置零的时候禁止用户态程序产生中断。

在中断产生后, 应该有个**中断处理程序**来处理中断。CPU 怎么知道中断处理程序在哪? 实际上, RISC-V 架构有个 CSR 叫做 `stvec`(Supervisor Trap Vector Base Address Register), 即所谓的”中断向量表基址”。中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序, 那么可以让 `stvec` 直接指向那个中断处理程序的地址。

对于 RISC-V 架构, `stvec` 会把最低位的两个二进制位用来编码一个“模式”, 如果是“00”就说明更高的 `SXLEN-2` 个二进制位存储的是唯一的中断处理程序的地址 (`SXLEN` 是 `stval` 寄存器的位数), 如果是“01”说明更高的 `SXLEN-2` 个二进制位存储的是中断向量表基址, 通过不同的异常原因来索引中断向量表。但是怎样用 62 个二进制位编码一个 64 位的地址? RISC-V 架构要求这个地址是四字节对齐的, 总是在较高的 62 位后补两个 0。

扩展

在旧版本的 RISC-V privileged ISA 标准中 (1.9.1 及以前), RISC-V 不支持中断向量表, 用最后两位数编码一个模式是 1.10 版本加入的。可以思考一下这个改动如何保证了后向兼容。[历史版本的 ISA 手册](#)

1.9.1 版本的 RISC-V privileged architecture 手册:

4.1.3 Supervisor Trap Vector Base Address Register (stvec) The stvec register is an XLEN-bit read/write register that holds the base address of the S-mode trap vector. When an exception occurs, the pc is set to stvec. The stvec register is always aligned to a 4-byte boundary

当我们触发中断进入 S 态进行处理时, 以下寄存器会被硬件自动设置, 将一些信息提供给中断处理程序:

sepc(supervisor exception program counter), 它会记录触发中断的那条指令的地址;

scause, 它会记录中断发生的原因, 还会记录该中断是不是一个外部中断;

stval, 它会记录一些中断处理所需要的辅助信息, 比如指令获取 (instruction fetch)、访存、缺页异常, 它会把发生问题的目标地址或者出错的指令记录下来, 这样我们在中断处理程序中就知道处理目标了。

扩展

The RISC-V Instruction Set Manual Volume II: Privileged Architecture

(Document Version 20190608-Priv-MSU-Ratified)

4.1.1 Supervisor Status Register (ssstatus)

The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the sie CSR. The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1. The UIE bit enables or disables user-mode interrupts. User-level interrupts are enabled only if UIE is set and the hart is running in user-mode. The UPIE bit indicates whether user-level interrupts were enabled prior to taking a user-level trap. When a URET instruction is executed, UIE is set to UPIE, and UPIE is set to 1. User-level interrupts are optional. If omitted, the UIE and UPIE bits are hardwired to zero.

4.1.9 Supervisor Exception Program Counter (sepc)

When a trap is taken into S-mode, sepc is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, sepc is never written by the implementation, though it may be explicitly written by software.

4.1.10 Supervisor Cause Register (scause)

When a trap is taken into S-mode, scause is written with a code indicating the event that caused the trap. Otherwise, scause is never written by the implementation, though it may be explicitly written by software.

4.1.11 Supervisor Trap Value (stval) Register

When a trap is taken into S-mode, stval is written with exception-specific information to assist software in handling the trap. Otherwise, stval is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set stval informatively and which may unconditionally set it to zero. When a hardware breakpoint is triggered, or an instruction-fetch, load, or store address-misaligned, access, or page-fault exception occurs, stval is written with the faulting virtual address. On an illegal instruction trap, stval may be written with the first XLEN or ILEN bits of the faulting instruction as described below. For other exceptions, stval is set to zero, but a future standard may redefine stval's setting for other exceptions.

4.1.3.1.1.8 特权指令

RISC-V 支持以下和中断相关的特权指令：

ecall(environment call)，当我们在 S 态执行这条指令时，会触发一个 **ecall-from-s-mode-exception**，从而进入 M 模式中的中断处理流程（如设置定时器等）；当我们在 U 态执行这条指令时，会触发一个 **ecall-from-u-mode-exception**，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

sret，用于 S 态中断返回到 U 态，实际作用为 $pc \leftarrow sepc$ ，回顾 **sepc** 定义，返回到通过中断进入 S 态之前的地址。

ebreak(environment break)，执行这条指令会触发一个断点中断从而进入中断处理流程。

mret，用于 M 态中断返回到 S 态或 U 态，实际作用为 $pc \leftarrow mepc$ ，回顾 **sepc** 定义，返回到通过中断进入 M 态之前的地址。（一般不用涉及）

4.1.3.1.2 掉进兔子洞 (中断入口点)

我们已经知道，在发生中断的时候，CPU 会跳到 **stvec**。我们准备采用 **Direct** 模式，也就是只有一个中断处理程序，**stvec** 直接跳到中断处理程序的入口点，那么我们需要对 **stvec** 寄存器做初始化。

中断的处理需要“放下当前的事情但之后还能回来接着之前往下做”，对于 CPU 来说，实际上只需要把原先的寄存器保存下来，做完其他事情把寄存器恢复回来就可以了。这些寄存器也被叫做 CPU 的 **context**(上下文，情境)。我们要用汇编实现上下文切换 (context switch) 机制，这包含两步：

- 保存 CPU 的寄存器（上下文）到内存中（栈上）
- 从内存中（栈上）恢复 CPU 的寄存器

为了方便我们组织上下文的数据（几十个寄存器），我们定义一个结构体。

sscratch 寄存器在处理用户态程序的中断时才起作用。在目前其实用处不大。

须知 RISC-V 汇编的通用寄存器别名和含义

The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213

Chapter 25 RISC-V Assembly Programmer's Handbook

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

```
// kern/trap/trap.h
#ifndef __KERN_TRAP_TRAP_H__
#define __KERN_TRAP_TRAP_H__

#include <defs.h>

struct pushregs {
    uintptr_t zero; // Hard-wired zero
    uintptr_t ra; // Return address
    uintptr_t sp; // Stack pointer
    uintptr_t gp; // Global pointer
    uintptr_t tp; // Thread pointer
    uintptr_t t0; // Temporary
    uintptr_t t1; // Temporary
    uintptr_t t2; // Temporary
    uintptr_t s0; // Saved register/frame pointer
    uintptr_t s1; // Saved register
    uintptr_t a0; // Function argument/return value
    uintptr_t a1; // Function argument/return value
    uintptr_t a2; // Function argument
    uintptr_t a3; // Function argument
    uintptr_t a4; // Function argument
    uintptr_t a5; // Function argument
    uintptr_t a6; // Function argument
    uintptr_t a7; // Function argument
    uintptr_t s2; // Saved register
    uintptr_t s3; // Saved register
    uintptr_t s4; // Saved register
    uintptr_t s5; // Saved register
    uintptr_t s6; // Saved register
    uintptr_t s7; // Saved register
    uintptr_t s8; // Saved register
    uintptr_t s9; // Saved register
    uintptr_t s10; // Saved register
    uintptr_t s11; // Saved register
    uintptr_t t3; // Temporary
    uintptr_t t4; // Temporary
    uintptr_t t5; // Temporary
    uintptr_t t6; // Temporary
}
```

(续下页)

(接上页)

```
};

struct trapframe {
    struct pushregs gpr;
    uintptr_t status; //sstatus
    uintptr_t epc; //sepc
    uintptr_t badvaddr; //sbadvaddr
    uintptr_t cause; //scause
};

void trap(struct trapframe *tf);
```

C 语言里面的结构体, 是若干个变量在内存里直线排列。也就是说, 一个 trapFrame 结构体占据 36 个 uintptr_t 的空间 (在 64 位 RISC-V 架构里我们定义 uintptr_t 为 64 位无符号整数), 里面依次排列通用寄存器 x0 到 x31, 然后依次排列 4 个和中断相关的 CSR, 我们希望中断处理程序能够利用这几个 CSR 的数值。

首先我们定义一个汇编宏 SAVE_ALL, 用来保存所有寄存器到栈顶 (实际上把一个 trapFrame 结构体放到了栈顶)。

```
# kern/trap/trapentry.S
#include <riscv.h>

.macro SAVE_ALL #定义汇编宏

    csrw sscratch, sp #保存原先的栈顶指针到sscratch

    addi sp, sp, -36 * REGBYTES #REGBYTES是riscv.h定义的常量, 表示一个寄存器占据几个字节
→ 节
    #让栈顶指针向低地址空间延伸 36个寄存器的空间, 可以放下一个trapFrame结构体。
    #除了32个通用寄存器, 我们还要保存4个和中断有关的CSR

    #依次保存32个通用寄存器。但栈顶指针需要特殊处理。
    #因为我们想在trapFrame里保存分配36个REGBYTES之前的sp
    #也就是保存之前写到sscratch里的sp的值
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    STORE x3, 3*REGBYTES(sp)
    STORE x4, 4*REGBYTES(sp)
    STORE x5, 5*REGBYTES(sp)
    STORE x6, 6*REGBYTES(sp)
    STORE x7, 7*REGBYTES(sp)
    STORE x8, 8*REGBYTES(sp)
    STORE x9, 9*REGBYTES(sp)
    STORE x10, 10*REGBYTES(sp)
    STORE x11, 11*REGBYTES(sp)
    STORE x12, 12*REGBYTES(sp)
    STORE x13, 13*REGBYTES(sp)
    STORE x14, 14*REGBYTES(sp)
    STORE x15, 15*REGBYTES(sp)
    STORE x16, 16*REGBYTES(sp)
    STORE x17, 17*REGBYTES(sp)
    STORE x18, 18*REGBYTES(sp)
    STORE x19, 19*REGBYTES(sp)
    STORE x20, 20*REGBYTES(sp)
    STORE x21, 21*REGBYTES(sp)
    STORE x22, 22*REGBYTES(sp)
```

(续下页)

(接上页)

```

STORE x23, 23*REGBYTES(sp)
STORE x24, 24*REGBYTES(sp)
STORE x25, 25*REGBYTES(sp)
STORE x26, 26*REGBYTES(sp)
STORE x27, 27*REGBYTES(sp)
STORE x28, 28*REGBYTES(sp)
STORE x29, 29*REGBYTES(sp)
STORE x30, 30*REGBYTES(sp)
STORE x31, 31*REGBYTES(sp)
# RISC-V不能直接从CSR写到内存, 需要csrr把CSR读取到通用寄存器, 再从通用寄存器STORE到内存
csrrw s0, sscratch, x0
csrr s1, sstatus
csrr s2, sepc
csrr s3, sbadaddr
csrr s4, scause

STORE s0, 2*REGBYTES(sp)
STORE s1, 32*REGBYTES(sp)
STORE s2, 33*REGBYTES(sp)
STORE s3, 34*REGBYTES(sp)
STORE s4, 35*REGBYTES(sp)
.endm #汇编宏定义结束

```

然后是恢复上下文的汇编宏, 恢复的顺序和当时保存的顺序反过来, 先加载两个 CSR, 再加载通用寄存器。

```

# kern/trap/trapentry.S
.macro RESTORE_ALL

LOAD s1, 32*REGBYTES(sp)
LOAD s2, 33*REGBYTES(sp)

# 注意之前保存的几个CSR并不都需要恢复
csrr sstatus, s1
csrr sepc, s2

# 恢复sp之外的通用寄存器, 这时候还需要根据sp来确定其他寄存器数值保存的位置
LOAD x1, 1*REGBYTES(sp)
LOAD x3, 3*REGBYTES(sp)
LOAD x4, 4*REGBYTES(sp)
LOAD x5, 5*REGBYTES(sp)
LOAD x6, 6*REGBYTES(sp)
LOAD x7, 7*REGBYTES(sp)
LOAD x8, 8*REGBYTES(sp)
LOAD x9, 9*REGBYTES(sp)
LOAD x10, 10*REGBYTES(sp)
LOAD x11, 11*REGBYTES(sp)
LOAD x12, 12*REGBYTES(sp)
LOAD x13, 13*REGBYTES(sp)
LOAD x14, 14*REGBYTES(sp)
LOAD x15, 15*REGBYTES(sp)
LOAD x16, 16*REGBYTES(sp)
LOAD x17, 17*REGBYTES(sp)
LOAD x18, 18*REGBYTES(sp)
LOAD x19, 19*REGBYTES(sp)
LOAD x20, 20*REGBYTES(sp)
LOAD x21, 21*REGBYTES(sp)

```

(续下页)

(接上页)

```

LOAD x22, 22*REGBYTES(sp)
LOAD x23, 23*REGBYTES(sp)
LOAD x24, 24*REGBYTES(sp)
LOAD x25, 25*REGBYTES(sp)
LOAD x26, 26*REGBYTES(sp)
LOAD x27, 27*REGBYTES(sp)
LOAD x28, 28*REGBYTES(sp)
LOAD x29, 29*REGBYTES(sp)
LOAD x30, 30*REGBYTES(sp)
LOAD x31, 31*REGBYTES(sp)
# 最后恢复sp
LOAD x2, 2*REGBYTES(sp)
.endm

```

现在我们编写真正的中断入口点

```

.globl __alltraps

.align(2) #中断入口点 __alltraps必须四字节对齐
__alltraps:
    SAVE_ALL #保存上下文

    move a0, sp #传递参数。
    #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
    #trap是trap.c里面的一个C语言函数, 也就是我们的中断处理程序
    jal trap
    #trap函数指向完之后, 会回到这里向下继续执行__trapret里面的内容, RESTORE_ALL, sret

.globl __trapret
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret

```

我们可以看到, trapentry.S 这个中断入口点的作用是保存和恢复上下文, 并把上下文包装成结构体送到 trap 函数那里去。下面我们就去看看 trap 函数里面做些什么。

4.1.3.1.3 中断处理程序

中断处理需要初始化, 所以我们在 init.c 里调用一些初始化的函数

```

// kern/init/init.c
#include <trap.h>
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    cons_init(); // init the console

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);

    print_kerninfo();

    // grade_backtrace();

```

(续下页)

(接上页)

```

//trap.h的函数, 初始化中断
idt_init(); // init interrupt descriptor table

//clock.h的函数, 初始化时钟中断
clock_init();
//intr.h的函数, 使能中断
intr_enable();
while (1)
    ;
}
// kern/trap/trap.c
void idt_init(void) {
    extern void __alltraps(void);
    //约定: 若中断前处于S态, sscratch为0
    //若中断前处于U态, sscratch存储内核栈地址
    //那么之后就可以通过sscratch的数值判断是内核态产生的中断还是用户态产生的中断
    //我们现在是内核态所以给sscratch置零
    write_csr(sscratch, 0);
    //我们保证__alltraps的地址是四字节对齐的, 将__alltraps这个符号的地址直接写到stvec
    ↪寄存器
    write_csr(stvec, &__alltraps);
}
//kern/driver/intr.c
#include <intr.h>
#include <riscv.h>
/* intr_enable - enable irq interrupt, 设置sstatus的Supervisor中断使能位 */
void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
/* intr_disable - disable irq interrupt */
void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }

```

trap.c 的中断处理函数 trap, 实际上把中断处理, 异常处理的工作分发给了 interrupt_handler(), exception_handler(), 这些函数再根据中断或异常的不同类型来处理。

```

// kern/trap/trap.c
/* trap_dispatch - dispatch based on what type of trap occurred */
static inline void trap_dispatch(struct trapframe *tf) {
    //scause的最高位是1, 说明trap是由中断引起的
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}

/* *
 * trap - handles or dispatches an exception/interrupt. if and when trap()
 * returns,
 * the code in kern/trap/trapentry.S restores the old CPU state saved in the
 * trapframe and then uses the ired instruction to return from the exception.
 * */
void trap(struct trapframe *tf) { trap_dispatch(tf); }

```

我们可以看到, interrupt_handler() 和 exception_handler() 的实现还比较简单, 只是简单地根据

scause 的数值更仔细地分了下类, 做了一些输出就直接返回了。switch 里的各种 case, 如 IRQ_U_SOFT, CAUSE_USER_ECALL, 是 riscv ISA 标准里规定的。我们在 riscv.h 里定义了这些常量。我们接下来主要关注时钟中断的处理。

在这里我们对时钟中断进行了一个简单的处理, 即每次触发时钟中断的时候, 我们会给一个计数器加一, 并且设定好下一次时钟中断。当计数器加到 100 的时候, 我们会输出一个 100ticks 表示我们触发了 100 次时钟中断。通过在模拟器中观察输出我们即刻看到是否正确触发了时钟中断, 从而验证我们实现的异常处理机制。

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1; //抹掉scause最高位代表“这是中断不是异常”
    ↪ 的1
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_SOFT:
            cprintf("Supervisor software interrupt\n");
            break;
        case IRQ_H_SOFT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_SOFT:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_TIMER:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_TIMER:
            //时钟中断
            /* LAB1 EXERCISE2    YOUR CODE :    */
            /* (1)设置下次时钟中断
             * (2)计数器 (ticks) 加一
             * (3)当计数器加到100的时候, 我们会输出一个`100ticks`表示我们触发了100次时
            ↪ 钟中断, 同时打印次数 (num) 加一
             * (4)判断打印次数, 当打印次数为10时, 调用<sbi.h>中的关机函数关机
             */
            break;
        case IRQ_H_TIMER:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_TIMER:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_EXT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_EXT:
            cprintf("Supervisor external interrupt\n");
            break;
        case IRQ_H_EXT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_EXT:
            cprintf("Machine software interrupt\n");
            break;
        default:
            print_trapframe(tf);
    }
}
```

(续下页)

(接上页)

```

        break;
    }
}

void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            break;
        case CAUSE_FAULT_FETCH:
            break;
        case CAUSE_ILLEGAL_INSTRUCTION:
            //非法指令异常处理
            /* LAB1 CHALLENGE3 YOUR CODE : */
            /* (1) 输出指令异常类型 ( Illegal instruction)
             * (2) 输出异常指令地址
             * (3) 更新 tf->epc 寄存器
             */
            break;
        case CAUSE_BREAKPOINT:
            //非法指令异常处理
            /* LAB1 CHALLENGE3 YOUR CODE : */
            /* (1) 输出指令异常类型 ( breakpoint)
             * (2) 输出异常指令地址
             * (3) 更新 tf->epc 寄存器
             */
            break;
        case CAUSE_MISALIGNED_LOAD:
            break;
        case CAUSE_FAULT_LOAD:
            break;
        case CAUSE_MISALIGNED_STORE:
            break;
        case CAUSE_FAULT_STORE:
            break;
        case CAUSE_USER_ECALL:
            break;
        case CAUSE_SUPERVISOR_ECALL:
            break;
        case CAUSE_HYPERVISOR_ECALL:
            break;
        case CAUSE_MACHINE_ECALL:
            break;
        default:
            print_trapframe(tf);
            break;
    }
}

```

下面是 RISC-V 标准里 `scause` 的部分, 可以看到有个 `scause` 的数值与中断/异常原因的对应表格。

4.1.10 Supervisor Cause Register (scause)

The `scause` register is an SXLEN-bit read-write register formatted as shown in Figure 4.9. When a trap is taken into S-mode, `scause` is written with a code indicating the event that caused the trap. Otherwise, `scause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `scause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table 4.2 lists the possible exception codes for the current supervisor ISAs. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

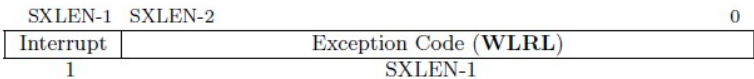


Figure 4.9: Supervisor Cause register `scause`.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	<i>Reserved for future standard use</i>
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	<i>Reserved for future standard use</i>
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10–15	<i>Reserved for future standard use</i>
1	≥ 16	<i>Reserved for platform use</i>
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	<i>Reserved for future standard use</i>
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved for future standard use</i>
0	15	Store/AMO page fault
0	16–23	<i>Reserved for future standard use</i>
0	24–31	<i>Reserved for custom use</i>
0	32–47	<i>Reserved for future standard use</i>
0	48–63	<i>Reserved for custom use</i>
0	≥ 64	<i>Reserved for future standard use</i>

Table 4.2: Supervisor cause register (`scause`) values after trap. Synchronous exception priorities are given by Table 3.7.

下一节我们将仔细讨论如何设置好时钟模块。

4.1.3.1.4 滴答滴答（时钟中断）

时钟中断需要 CPU 硬件的支持。CPU 以“时钟周期”为工作的基本时间单位，对逻辑门的时序电路进行同步。

我们的“时钟中断”实际上就是“每隔若干个时钟周期执行一次的程序”。

“若干个时钟周期”是多少个？太短了肯定不行。如果时钟中断处理程序需要 100 个时钟周期执行，而你每 50 个时钟周期就触发一个时钟中断，那么间隔时间连一个完整的时钟中断程序都跑不完。如果你 200 个时钟周期就触发一个时钟中断，那么 CPU 的时间将有一半消耗在时钟中断，开销太大。一般而言，可以设置时钟中断间隔设置为 CPU 频率的 1%，也就是每秒触发 100 次时钟中断，避免开销过大。

我们用到的 RISC-V 对时钟中断的硬件支持包括：

- OpenSBI 提供的 `sbi_set_timer()` 接口，可以传入一个时刻，让它在那个时刻触发一次时钟中断

- `rdtime` 伪指令, 读取一个叫做 `time` 的 CSR 的数值, 表示 CPU 启动之后经过的真实时间。在不同硬件平台, 时钟频率可能不同。在 QEMU 上, 这个时钟的频率是 10MHz, 每过 1s, `rdtime` 返回的结果增大 10000000

趣闻

在 RISC-V32 和 RISC-V64 架构中, `time` 寄存器都是 64 位的。

`rdcycle` 伪指令可以读取经过的时钟周期数目, 对应一个寄存器 `cycle`

注意, 我们需要“每隔若干时间就发生一次时钟中断”, 但是 OpenSBI 提供的接口一次只能设置一个时钟中断事件。我们采用的方式是: 一开始只设置一个时钟中断, 之后每次发生时钟中断的时候, 设置下一次的时钟中断。

在 `clock.c` 里面初始化时钟并封装一些接口

```
//libs/sbi.c

//当time寄存器(rdtimer的返回值)为stime_value的时候触发一个时钟中断
void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}

// kern/driver/clock.c
#include <clock.h>
#include <defs.h>
#include <sbi.h>
#include <stdio.h>
#include <riscv.h>

//volatile告诉编译器这个变量可能在其他地方被瞎改一通, 所以编译器不要对这个变量瞎优化
volatile size_t ticks;

//对64位和32位架构, 读取time的方法是不同的
//32位架构下, 需要把64位的time寄存器读到两个32位整数里, 然后拼起来形成一个64位整数
//64位架构简单的一句rdtime就可以了
//__riscv_xlen是gcc定义的一个宏, 可以用来区分是32位还是64位。
static inline uint64_t get_time(void) { //返回当前时间
    #if __riscv_xlen == 64
        uint64_t n;
        __asm__ __volatile__ ("rdtime %0" : "=r"(n));
        return n;
    #else
        uint32_t lo, hi, tmp;
        __asm__ __volatile__ (
            "1:\n"
            "rdtimeh %0\n"
            "rdtime %1\n"
            "rdtimeh %2\n"
            "bne %0, %2, 1b"
            : "=&r"(hi), "=&r"(lo), "=&r"(tmp));
        return ((uint64_t)hi << 32) | lo;
    #endif
}

// Hardcode timebase
static uint64_t timebase = 100000;
```

(续下页)

(接上页)

```

void clock_init(void) {
    // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭的
    // 所以我们要在初始化的时候, 使能时钟中断
    set_csr(sie, MIP_STIP); // enable timer interrupt in sie
    //设置第一个时钟中断事件
    clock_set_next_event();
    // 初始化一个计数器
    ticks = 0;

    cprintf("++ setup timer interrupts\n");
}
//设置时钟中断: timer的数值变为当前时间 + timebase 后, 触发一次时钟中断
//对于QEMU, timer增加1, 过去了10^-7 s, 也就是100ns
void clock_set_next_event(void) { sbi_set_timer(get_time() + timebase); }

```

回来看 trap.c 里面时钟中断处理的代码, 还是很简单的: 每秒 100 次时钟中断, 触发每次时钟中断后设置 10ms 后触发下一次时钟中断, 每触发 100 次时钟中断 (1 秒钟) 输出一行信息到控制台。

```

// kern/trap/trap.c
#include<clock.h>

#define TICK_NUM 100
static void print_ticks() {
    cprintf("%d ticks\n", TICK_NUM);
#ifdef DEBUG_GRADE
    cprintf("End of Test.\n");
    panic("EOT: kernel seems ok.");
#endif
}

void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        /* blabla 其他case*/
        case IRQ_S_TIMER:
            clock_set_next_event(); //发生这次时钟中断的时候, 我们要设置下一次时钟中断
            if (++ticks % TICK_NUM == 0) {
                print_ticks();
            }
            break;
        /* blabla 其他case*/
    }
}

```

现在执行 make qemu, 应该能看到打印一行行的 100 ticks

目前为止的代码可以在[这里](#)找到, 遇到困难可以参考。

4.1.4 实验报告要求

从 git server 网站上取得 ucore_lab 后, 进入目录 labcodes/lab1, 完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 labcodes/lab1 下存放实验报告, 实验报告文档命名为 lab1-学堂在线 ID.md。推荐用 **markdown** 格式。对于 lab1 中编程任务, 完成编写之后, 再通过 git push 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

注意有“LAB1”的注释, 代码中所有需要完成的地方(challenge 除外)都有“LAB1”和“YOUR CODE”的注释, 请在提交时特别注意保持注释, 并将“YOUR CODE”替换为自己的学号, 并且将所有标有对应注释的部分填上正确的代码。

lab2: 物理内存和页表

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。在实验二中大家会了解并且自己动手完成一个简单的物理内存管理系统。

5.1 本章内容

5.1.1 实验目的

- 理解页表的建立和使用方法
- 理解物理内存的管理方法
- 理解页面分配算法

5.1.2 实验内容

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。本次实验我们会了解如何发现系统中的物理内存，然后学习如何建立对物理内存的初步管理，即了解连续物理内存管理，最后掌握页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，帮助我们对段页式内存管理机制有一个比较全面的了解。本次的实验主要是在实验一的基础上完成物理内存管理，并建立一个最简单的页表映射。

5.1.2.1 本节内容

5.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 ucore_lab 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

5.1.2.1.1.1 练习 0: 填写已有实验

本实验依赖实验 1。请把你做的实验 1 的代码填入本实验中代码中有“LAB1”的注释相应部分并按照实验手册进行进一步的修改。具体来说, 就是跟着实验手册的教程一步步做, 然后完成教程后继续完成完成 exercise 部分的剩余练习。

5.1.2.1.1.2 练习 1: 理解 first-fit 连续物理内存分配算法 (思考题)

first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法, 需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合 kern/mm/default_pmm.c 中的相关代码, 认真分析 default_init, default_init_memmap, default_alloc_pages, default_free_pages 等相关函数, 并描述程序在进行物理内存分配的过程以及各个函数的作用。请在实验报告中简要说明你的设计实现过程。请回答如下问题:

- 你的 first fit 算法是否有进一步的改进空间?

5.1.2.1.1.3 练习 2: 实现 Best-Fit 连续物理内存分配算法 (需要编程)

在完成练习一后, 参考 kern/mm/default_pmm.c 对 First Fit 算法的实现, 编程实现 Best Fit 页面分配算法, 算法的时空复杂度不做要求, 能通过测试即可。请在实验报告中简要说明你的设计实现过程, 阐述代码是如何对物理内存进行分配和释放, 并回答如下问题:

- 你的 Best-Fit 算法是否有进一步的改进空间?

5.1.2.1.1.4 扩展练习 Challenge: buddy system (伙伴系统) 分配算法 (需要编程)

Buddy System 算法把系统中的可用存储空间划分为存储块 (Block) 来进行管理, 每个存储块的大小必须是 2 的 n 次幂 ($Pow(2, n)$), 即 1, 2, 4, 8, 16, 32, 64, 128...

- 参考伙伴分配器的一个极简实现, 在 ucore 中实现 buddy system 分配算法, 要求有比较充分的测试用例说明实现的正确性, 需要有设计文档。

5.1.2.1.1.5 扩展练习 Challenge: 任意大小的内存单元 slub 分配算法 (需要编程)

slub 算法, 实现两层架构的高效内存单元分配, 第一层是基于页大小的内存分配, 第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现, 能够体现其主体思想即可。

- 参考[linux 的 slub 分配算法](#), 在 ucore 中实现 slub 分配算法。要求有比较充分的测试用例说明实现的正确性, 需要有设计文档。

5.1.2.1.1.6 扩展练习 Challenge: 硬件的可用物理内存范围的获取方法 (思考题)

- 如果 OS 无法提前知道当前硬件的可用物理内存范围, 请问你有何办法让 OS 获取可用物理内存范围?

Challenges 是选做, 完成 Challenge 的同学可单独提交 Challenge。完成得好的同学可获得最终考试成绩的加分。

5.1.2.1.2 项目组成

表 1: 实验二文件列表

```

— Makefile
├── kern
│   ├── debug
│   │   ├── assert.h
│   │   ├── kdebug.c
│   │   ├── kdebug.h
│   │   ├── kmonitor.c
│   │   ├── kmonitor.h
│   │   ├── panic.c
│   │   └── stab.h
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   ├── console.c
│   │   ├── console.h
│   │   ├── intr.c
│   │   └── intr.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   │   └── stdio.c
│   ├── mm
│   │   ├── best_fit_pmm.c
│   │   ├── best_fit_pmm.h
│   │   ├── default_pmm.c
│   │   ├── default_pmm.h
│   │   ├── memlayout.h
│   │   ├── mmu.h
│   │   ├── pmm.c
│   │   └── pmm.h
│   └── trap
│       ├── trap.c
│       ├── trap.h
│       └── trapentry.S
└── libs

```

(续下页)

(接上页)

```

|   |— atomic.h
|   |— defs.h
|   |— error.h
|   |— list.h
|   |— printfmt.c
|   |— readline.c
|   |— riscv.h
|   |— sbi.c
|   |— sbi.h
|   |— stdarg.h
|   |— stdio.h
|   |— string.c
|   |— string.h
|— tools
|   |— boot.ld
|   |— function.mk
|   |— gdbinit
|   |— grade.sh
|   |— kernel.ld
|   |— kernel_nopage.ld
|   |— kflash.py
|   |— rustsbi-k210.bin
|   |— sign.c
|   |— vector.c

```

编译方法

编译并运行代码的命令如下:

```

make
make qemu

```

则可以得到如下显示界面 (仅供参考)

```

chenyu$ make qemu
(THU.CST) os is loading ...
Special kernel symbols:
  entry 0xffffffffc0200036 (virtual)
  etext 0xffffffffc0201ad2 (virtual)
  edata 0xffffffffc0206010 (virtual)
  end   0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x00000000080200000, 0x00000000087fffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x00000000080205000
++ setup timer interrupts
100 ticks
100 ticks
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, `ucore` 显示了物理内存的布局信息, 其中包含了内存范围。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 建立好的页表内容, 并在分页模式下响应时钟中断。

5.1.3 物理内存管理

5.1.3.1 本节内容

5.1.3.1.1 基本原理概述

5.1.3.1.1.1 物理内存管理

什么是物理内存管理? 如果我们只有物理内存空间, 不进行任何的管理操作, 那么我们也可以写程序。但这样显然会导致所有的程序, 不管是内核还是用户程序都处于同一个地址空间中, 这样显然是不好的。

举个例子: 如果系统中只有一个程序在运行, 那影响自然是有限的。但如果很多程序使用同一个内存空间, 比如此时内核和用户程序都想访问 `0x80200000` 这个地址, 那么因为它们处于一个地址空间中就会导致互相干扰, 甚至是互相破坏。

那么如何消除这种影响呢? 大家显然可以想象得到, 我们可以通过让用户程序访问的 `0x80200000` 和内核访问的 `0x80200000` 不是一个地址来解决这个问题。但是如果我们只有一块内存, 那么为了创造两个不同的地址空间, 我们可以引入一个”翻译“机制: 程序使用的地址需要经过一步”翻译“才能变成真正的内存的物理地址。这个”翻译“过程, 我们可以用一个”词典“实现。通过这个”词典“给出翻译之前的地址, 可以在词典里查找翻译后的地址。而对每个程序往往都有着唯一的一本”词典“, 而它能使用的内存也就只有他的”词典“所包含的。

”词典“是否对能使用的每个字节都进行翻译? 我们可以想象, 存储每个字节翻译的结果至少需要一个字节, 那么使用 1MB 的内存将至少需要构造 1MB 的”词典“, 这效率太低了。观察到, 一个程序使用内存的数量级通常远大于字节, 至少以 KB 为单位 (所以上古时代的人说的是”640K 对每个人都够了“而不是”640B 对每个人都够了“)。

那么我们可以考虑, 把连续的很多字节合在一起翻译, 让他们翻译前后的数值之差相同, 这就是“页”。

5.1.3.1.1.2 物理地址和虚拟地址

在本次实验中, 我们使用的是 RISC-V 的 `sv39` 页表机制, 每个页的大小是 4KB, 也就是 4096 个字节。通过之前的介绍相信大家对于物理地址和虚拟地址有了一个初步的认识了, 页表就是那个“词典”, 里面有程序使用的虚拟页号到实际内存的物理页号的对应关系, 但并不是所有的虚拟页都有对应的物理页。虚拟页可能的数目远大于物理页的数目, 而且一个程序在运行时, 一般不会拥有所有物理页的使用权, 而只是将部分物理页在它的页表里进行映射。

在 `sv39` 中, 定义物理地址 (Physical Address) 有 56 位, 而虚拟地址 (Virtual Address) 有 39 位。实际使用的时候, 一个虚拟地址要占用 64 位, 只有低 39 位有效, 我们规定 63-39 位的值必须等于第 38 位的值 (大家可以将它类比为有符号整数), 否则会认为该虚拟地址不合法, 在访问时会产生异常。不论是物理地址还是虚拟地址, 我们都可以认为, 最后 12 位表示的是页内偏移, 也就是这个地址在它所在页帧的什么位置 (同一个位置的物理地址和虚拟地址的页内偏移相同)。除了最后 12 位, 前面的部分表示的是物理页号或者虚拟页号。

5.1.3.1.2 实验执行流程概述

本次实验主要完成 ucore 内核对物理内存的管理工作。我们要在 lab1 实验的工作上对 ucore 进行相关拓展, 修改 ucore 总控函数 kern_init 的代码。

kernel 在后续执行中能够探测出的物理内存情况进行物理内存管理初始化工作。其次, bootloader 不像 lab1 那样, 直接调用 kern_init 函数, 而是先调用 entry.S 中的 kern_entry 函数。kern_entry 函数的主要任务是为执行 kern_init 建立一个良好的 C 语言运行环境 (设置堆栈), 而且临时建立了一个段映射关系, 为之后建立分页机制的过程做一个准备。完成这些工作后, 才调用 kern_init 函数。

kern_init 函数在完成一些输出并对 lab1 实验结果的检查后, 将进入物理内存管理初始化的工作, 即调用 pmm_init 函数完成物理内存的管理。接着是执行中断和异常相关的初始化工作, 即调用 pic_init 函数和 idt_init 函数等。

为了完成物理内存管理, 这里首先需要探测可用的物理内存资源; 了解到物理内存位于什么地方, 有多大之后, 就以固定页面大小来划分整个物理内存空间, 并准备以此为最小内存分配单位来管理整个物理内存, 管理在内核运行过程中每页内存, 设定其可用状态 (free 的, used 的, 还是 reserved 的), 这其实就对应了我们在课本上讲到的连续内存分配概念和原理的具体实现; 接着 ucore kernel 就要建立页表, 启动分页机制, 让 CPU 的 MMU 把预先建立好的页表中的页表项读入到 TLB 中, 根据页表项描述的虚拟页 (Page) 与物理页帧 (Page Frame) 的对应关系完成 CPU 对内存的读、写和执行操作。这一部分其实就对应了我们在课本上讲到的内存映射、页表、多级页表等概念和原理的具体实现。

ucore 在实现上述技术时, 需要解决两个关键问题:

- 如何建立虚拟地址和物理地址之间的联系
- 如何在现有 ucore 的基础上实现物理内存页分配算法

接下来将进一步分析完成 lab2 主要注意的关键问题和涉及的关键数据结构。

5.1.3.1.3 以页为单位管理物理内存

5.1.3.1.3.1 页表项

一个页表项是用来描述一个虚拟页号如何映射到物理页号的。如果一个虚拟页号通过某种手段找到了一个页表项, 并通过读取上面的物理页号完成映射, 那么我们称这个虚拟页号通过该页表项完成映射。而我们的”词典“(页表) 存储在内存里, 由若干个格式固定的”词条“也就是页表项 (PTE, Page Table Entry) 组成。显然我们需要给词典的每个词条约定一个固定的格式 (包括每个词条的大小, 含义), 这样查起来才方便。

那么在 sv39 的一个页表项占据 8 字节 (64 位), 那么页表项结构是这样的:

63-54	53-28	27-19	18-10	9-8	7	6	5	4	3	2	1	0
<i>Reserved</i>	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

我们可以看到 sv39 里面的一个页表项大小为 64 位 8 字节。其中第 53-10 位共 44 位为一个物理页号, 表示这个虚拟页号映射到的物理页号。后面的第 9-0 位共 10 位则描述映射的状态信息。

介绍一下映射状态信息各位的含义:

- RSW: 两位留给 S Mode 的应用程序, 我们可以用来进行拓展。
- D: 即 Dirty, 如果 D=1 表示自从上次 D 被清零后, 有虚拟地址通过这个页表项进行写入。
- A, 即 Accessed, 如果 A=1 表示自从上次 A 被清零后, 有虚拟地址通过这个页表项进行读、或者写、或者取指。

- G, 即 Global, 如果 G=1 表示这个页表项是“全局”的, 也就是所有的地址空间 (所有的页表) 都包含这一项
- U, 即 user, U 为 1 表示用户态 (U Mode) 的程序可以通过该页表项进行映射。在用户态运行时也只能通过 U=1 的页表项进行虚实地址映射。注意, S Mode 不一定可以通过 U=1 的页表项进行映射。我们需要将 S Mode 的状态寄存器 sstatus 上的 SUM 位手动设置为 1 才可以做到这一点 (通常情况不会把它置 1)。否则通过 U=1 的页表项进行映射也会报出异常。另外, 不论 sstatus 的 SUM 位如何取值, S Mode 都不允许执行 U=1 的页面里包含的指令, 这是出于安全的考虑。
- R,W,X 为许可位, 分别表示是否可读 (Readable), 可写 (Writable), 可执行 (Executable)。

以 W 这一位为例, 如果 W=0 表示不可写, 那么如果一条 store 的指令, 它通过这个页表项完成了虚拟页号到物理页号的映射, 找到了物理地址。但是仍然会报出异常, 是因为这个页表项规定如果物理地址是通过它映射得到的, 那么不准写入! R,X 也是同样的道理。

根据 R,W,X 取值的不同, 我们可以分成下面几种类型:

X	W	R	Meaning
0	0	0	指向下一级页表的指针
0	0	1	这一页只读
0	1	0	保留 (<i>reserved for future use</i>)
0	1	1	这一页可读可写 (不可执行)
1	0	0	这一页可读可执行 (不可写)
1	0	1	这一页可读可执行
1	1	0	保留 (<i>reserved for future use</i>)
1	1	1	这一页可读可写可执行

- V 表示这个页表项是否合法。如果为 0 表示不合法, 此时页表项其他位的值都会被忽略。

5.1.3.1.3.2 多级页表

在实际使用中显然如果只有一级页表, 那么我们构建出来的虚拟地址空间毕竟还是过于有限, 因此我们需要引入多级页表以实现更大规模的虚拟地址空间。

但相比于可用的物理内存空间, 我们的虚拟地址空间太大, 不可能为每个虚拟内存页都分配一个页表项。在 Sv39 中, 因为一个页表项占据 8 字节 (64 位), 而虚拟地址有 39 位, 后 12 位是页内偏移, 那么还剩下 27 位可以编码不同的虚拟页号。

如果开个大数组 `Pagetable[]`, 给个 2^{27} 虚拟页号都分配 8 字节的页表项, 其中 `Pagetable[vpn]` 代表虚拟页号为 `vpn` 的虚拟页的页表项, 那就是整整 1 GiB 的内存。但这里面其实很多虚拟地址我们没有用到, 会有大片大片的页表项的标志位为 0 (不合法), 显然我们不应该为那么多非法页表项浪费宝贵的内存空间。

因此, 我们可以对页表进行“分级”, 让它变成一个树状结构。也就是把很多页表项组合成一个“大页”; 如果这些页表项都非法 (没有对应的物理页), 那么只需要用一个非法的页表项来覆盖这个大页, 而不需要分别建立一大堆非法页表项。很多个大页 (megapage) 还可以组合起来变成大大页 (gigapage!), 继而可以有更大的页, 以此类推, 当然肯定不是分层越多越好, 因为随着层数增多, 开销也会越大。

在本次实验中, 我们使用的 sv39 权衡各方面效率, 使用三级页表。有 4KiB=4096 字节的页, 大小为 2MiB= 2^{21} 字节的大页, 和大小为 1 GiB 的大大页。

原先的一个 39 位虚拟地址, 被我们看成 27 位的页号和 12 位的页内偏移。那么在三级页表下, 我们可以把它看成 9 位的“大大页页号”, 9 位的“大页页号” (也是大大页内的页内偏移), 9 位的“页号” (大页的页内偏移), 还有 12 位的页内偏移。这是一个递归的过程, 中间的每一级页表映射是类似的。也就是说, 整个 Sv39 的虚拟内存空间里, 有 512 (2^9) 个大大页, 每个大大页里有 512 个大页, 每个大页里有 512 个页, 每个页里有 4096 个字节, 整个虚拟内存空间里就有 $512 \times 512 \times 512 \times 4096$ 个字节, 是 512GiB 的地址空间。

那么为啥是 512 呢? 注意, $4096/8 = 512$, 我们恰好可以在一页里放下 512 个页表项!

我们可以认为, Sv39 的多级页表在逻辑上是一棵树, 它的每个叶子节点 (直接映射 4KB 的页的页表项) 都对应内存的一页, 它的每个内部节点都对应 512 个更低一层的节点, 而每个内部节点向更低一层的节点的链接都使用内存里的一页进行存储。

或者说, Sv39 页表的根节点占据一页 4KiB 的内存, 存储 512 个页表项, 分别对应 512 个 1 GiB 的大大页, 其中有些页表项 (大大页) 是非法的, 另一些合法的页表项 (大大页) 是根节点的儿子, 可以通过合法的页表项跳转到一个物理页号, 这个物理页对应树中一个 “大大页” 的节点, 里面有 512 个页表项, 每个页表项对应一个 2MiB 的大页。同样, 这些大页可能合法, 也可能非法, 非法的页表项不对应内存里的页, 合法的页表项会跳转到一个物理页号, 这个物理页对应树中一个 “大页” 的节点, 里面有 512 个页表项, 每个页表项对应一个 4KiB 的页, 在这里最终完成虚拟页到物理页的映射。

三级和二级页表项不一定要指向下一级页表。我们知道每个一级页表项控制一个虚拟页号, 即控制 4KiB 虚拟内存; 每个二级页表项则控制 9 位虚拟页号, 总计控制 $4\text{KiB} \times 2^9 = 2\text{MiB}$ 虚拟内存; 每个三级页表项控制 18 位虚拟页号, 总计控制 $2\text{MiB} \times 2^9 = 1\text{GiB}$ 虚拟内存。我们可以将二级页表项的 R,W,X 设置为不是全 0 的许可要求, 那么它将与一级页表项类似, 只不过可以映射一个 2MiB 的大页 (Mega Page)。同理, 也可以将三级页表项看作一个叶子, 来映射一个 1GiB 的大大页 (Giga Page)。

5.1.3.1.3.3 页表基址

在翻译的过程中, 我们首先需要知道树状页表的根节点的物理地址。这一般保存在一个特殊寄存器里。对于 RISC-V 架构, 是一个叫做 satp (Supervisor Address Translation and Protection Register) 的 CSR。实际上, satp 里面存的不是最高级页表的起始物理地址, 而是它所在的物理页号。除了物理页号, satp 还包含其他信息。

63-60	59-44	43-0
MODE(WARL)	ASID(WARL)	PPN(WARL)
4	16	44

MODE 表示当前页表的模式:

- 0000 表示不使用页表, 直接使用物理地址, 在简单的嵌入式系统里用着很方便。
- 0100 表示 sv39 页表, 也就是我们使用的, 虚拟内存空间高达 512GiB。
- 0101 表示 Sv48 页表, 它和 Sv39 兼容。
- 其他编码保留备用 ASID (address space identifier) 我们目前用不到 OS 可以在内存中为不同的应用分别建立不同虚实映射的页表, 并通过修改寄存器 satp 的值指向不同的页表, 从而可以修改 CPU 虚实地址映射关系及内存保护的行为。

5.1.3.1.4 建立快表以加快访问效率

物理内存的访问速度要比 CPU 的运行速度慢很多, 去访问一次物理内存可能需要几百个时钟周期 (带来所谓的 “冯诺依曼瓶颈”)。如果我们按照页表机制一步步走, 将一个虚拟地址转化为物理地址需要访问 3 次物理内存, 得到物理地址之后还要再访问一次物理内存, 才能读到我们想要的数。这很大程度上降低了效率。好在, 实践表明虚拟地址的访问具有时间局部性和空间局部性。

- 时间局部性是指, 被访问过一次的地址很有可能不久的将来再次被访问;
- 空间局部性是指, 如果一个地址被访问, 则这个地址附近的地址很有可能不久的将来被访问。

因此, 在 CPU 内部, 我们使用快表 (TLB, Translation Lookaside Buffer) 来记录近期已完成的虚拟页号到物理页号的映射。由于局部性, 当我们要做一个映射时, 会有很大可能这个映射在近期被完成过, 所以我们可以先到 TLB 里面去查一下, 如果有的话我们就可以直接完成映射, 而不用访问那么多次内存了。但是, 我们如果修改了 satp 寄存器, 比如将上面的 PPN 字段进行了修改, 说明我们切换到了一个与先前映射方式完

全不同的页表。此时快表里面存储的映射结果就跟不上时代了，很可能是错误的。这种情况下我们要使用 `sfence.vma` 指令刷新整个 TLB。同样，我们手动修改一个页表项之后，也修改了映射，但 TLB 并不会自动刷新，我们也需要使用 `sfence.vma` 指令刷新 TLB。如果不加参数的，`sfence.vma` 会刷新整个 TLB。你可以在后面加上一个虚拟地址，这样 `sfence.vma` 只会刷新这个虚拟地址的映射。

5.1.3.1.5 分页机制的设计思路

5.1.3.1.5.1 本小节内容

5.1.3.1.5.2 建立段页式管理中需要考虑的关键问题

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，即正确建立三级页表。此过程涉及硬件细节，不同的地址映射关系组合，相对比较复杂。总体而言，我们需要思考如下问题：

- 如何在建立页表的过程中维护全局段描述符表（GDT）和页表的关系，确保 `ucore` 能够在各个时间段上都能正常寻址？
- 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？
- 如何设置页表项的内容？

5.1.3.1.5.3 实现分页机制

在本实验中，需要重点了解和实现基于页表的页机制和以页为单位的物理内存管理方法和分配算法等。由于 `ucore OS` 是基于 80386 CPU 实现的，所以 CPU 在进入保护模式后，就直接使能了段机制，并使得 `ucore OS` 需要在段机制的基础上建立页机制。下面比较详细地介绍了实现分页机制的过程。

接下来我们就正式开始实验啦！首先我们要做的是内核初始化的修改，我们现在需要做的就是将原本只能直接在物理地址空间上运行的内核引入页表机制。具体来说，我们现在想将内核代码放在虚拟地址空间中以 `0xfffffffffc0200000` 开头的一段高地址空间中。那怎么做呢？首先我们需要将下面的参数修改一下：

```
// tools/kernel.ld
BASE_ADDRESS = 0xFFFFFFFFFC0200000;
//之前这里是 0x80200000
```

我们修改了链接脚本中的起始地址。但是这样做的话，就能从物理地址空间转移到虚拟地址空间了吗？大家可以分析一下现在我们相当于是在 `bootloader` 的 `OpenSBI` 结束后的现状，这样就可以更好的理解接下来我们需要干什么：

- 物理内存状态：OpenSBI 代码放在 `[0x80000000, 0x80200000)` 中，内核代码放在以 `0x80200000` 开头的一块连续物理内存中。这个是实验一我们做完后就实现的效果。
- CPU 状态：处于 `S Mode`，寄存器 `satp` 的 `MODE` 被设置为 `Bare`，即无论取指还是访存我们都通过物理地址直接访问物理内存。`PC=0x80200000` 指向内核的第一条指令。栈顶地址 `SP` 处在 `OpenSBI` 代码内。
- 内核代码：这部分由于改动了链接脚本的起始地址，所以它会认为自己处在以虚拟地址 `0xfffffffffc0200000` 开头的一段连续虚拟地址空间中，以此为依据确定代码里每个部分的地址（每一段都是从 `BASE_ADDRESS` 往后依次摆开的，所以代码里各段都会认为自己在

0xfffffffffc0200000 之后的某个地址上, 或者说编译器和链接器会把里面的符号/变量地址都对应到 0xfffffffffc0200000 之后的某个地址上)

接下来, 我们需要修改 `entry.S` 文件来实现内核的初始化, 我们在入口点 `entry.S` 中所要做的事情是: 将 `SP` 寄存器从原先指向 `OpenSBI` 某处的栈空间, 改为指向我们自己在内核的内存空间里分配的栈; 同时需要跳转到函数 `kern_init` 中。

在之前的实验中, 我们已经在 `entry.S` 自己分配了一块 16KiB 的内存用来做启动栈:

```
#include <mmu.h>
#include <memlayout.h>

.section .text, "ax", %progbits
.globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

通过之前的实验大家应该都明白: 符号 `bootstacktop` 就是我们需要的栈顶地址, 符号 `kern_init` 代表了我们要跳转到的地址。之前我们直接将 `bootstacktop` 的值给到 `SP`, 再跳转到 `rust_main` 就行了。看上去上面的这个代码也能够实现我们想要的初始化效果呀? 但问题在于, 由于我们修改了链接脚本的起始地址, 编译器和链接器认为内核开头地址为 0xfffffffffc0200000, 因此这两个符号会被翻译成比这个开头地址还要高的某个虚拟地址。而我们的 CPU 目前还处于 Bare 模式, 会将地址都当成物理地址处理。这样, 我们跳转到 `rust_main` 就会跳转到比 0xfffffffffc0200000 还大的一个物理地址。但物理地址显然不可能有这么多位! 这就会出现这个问题。

于是, 我们需要想办法利用刚学的页表知识, 帮内核将需要的虚拟地址空间构造出来。也就是: 构建一个合适的页表, 让 `satp` 指向这个页表, 然后使用地址的时候都要经过这个页表的翻译, 使得虚拟地址 0xFFFFFFFFC0200000 经过页表的翻译恰好变成 0x80200000, 这个地址显然就比较合适了, 也就不会出错了。

理论知识告诉我们, 所有的虚拟地址有一个固定的偏移量。而要想实现页表结构这个偏移量显然是不可或缺的。而虚拟地址和物理地址之间的差值就可以当成是这个偏移量。

比如内核的第一条指令, 虚拟地址为 0xfffffffffc0200000, 物理地址为 0x80200000, 因此, 我们只要将虚拟地址减去 0xfffffffff40000000, 就得到了物理地址。所以当我们需要做到去访问内核里面的一个物理地址 `va` 时, 而已知虚拟地址为 `va` 时, 则 `va` 处的代码或数据就放在物理地址为 `pa = va - 0xfffffffff40000000` 处的物理内存中, 我们真正所要做的是要让 CPU 去访问 `pa`。因此, 我们要通过恰当构造页表, 来对于内核所属的虚拟地址, 实现这种 `va` 到 `pa` 的映射。

还记得之前的理论介绍的内容吗? 那时我们提到, 将一个三级页表项的标志位 `R, W, X` 不设为全 0, 可以将它变为一个叶子, 从而获得大小为 1GiB 的一个大页。

我们假定内核大小不超过 1GiB, 通过一个大页将虚拟地址区间 `[0xfffffffffc0000000, 0xffffffffffffffff]` 映射到物理地址区间 `[0x80000000, 0xc0000000)`, 而我们只需要分配一页内存用来存放三级页表, 并将其最后一个页表项 (也就是对应我们使用的虚拟地址区间的页表项) 进行适当设置即可。对应的代码如下所示:


```

#include <mmu.h>
#include <memlayout.h>

.section .text, "ax", %progbits
.globl kern_entry
kern_entry:
    # t0 := 三级页表的虚拟地址
    lui    t0, %hi(boot_page_table_sv39)
    # t1 := 0xffffffff40000000 即虚实映射偏移量
    li     t1, 0xffffffffc0000000 - 0x80000000
    # t0 减去虚实映射偏移量 0xffffffff40000000, 变为三级页表的物理地址
    sub    t0, t0, t1
    # t0 >= 12, 变为三级页表的物理页号
    srli   t0, t0, 12

    # t1 := 8 << 60, 设置 satp 的 MODE 字段为 Sv39
    li     t1, 8 << 60
    # 将刚才计算出的预设三级页表物理页号附加到 satp 中
    or     t0, t0, t1
    # 将算出的 t0(即新的MODE/页表基址物理页号) 覆盖到 satp 中
    csrw   satp, t0
    # 使用 sfence.vma 指令刷新 TLB
    sfence.vma
    # 从此, 我们给内核搭建出了一个完美的虚拟内存空间!
    #nop # 可能映射的位置有些bug。。插入一个nop

    # 我们在虚拟内存空间中: 随意将 sp 设置为虚拟地址!
    lui    sp, %hi(bootstacktop)

    # 我们在虚拟内存空间中: 随意跳转到虚拟地址!
    # 跳转到 kern_init
    lui    t0, %hi(kern_init)
    addi   t0, t0, %lo(kern_init)
    jr     t0

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:

.section .data
    # 由于我们要把这个页表放到一个页里面, 因此必须 12 位对齐
    .align PGSHIFT
    .global boot_page_table_sv39
# 分配 4KiB 内存给预设的三级页表
boot_page_table_sv39:
    # 0xffffffff_c0000000 map to 0x80000000 (1G)
    # 前 511 个页表项均设置为 0, 因此 V=0, 意味着是空的 (unmapped)
    .zero 8 * 511
    # 设置最后一个页表项, PPN=0x80000, 标志位 VRWXAD 均为 1
    .quad (0x80000 << 10) | 0xcf # VRWXAD

```

总结一下, 要进入虚拟内存访问方式, 需要如下步骤:

1. 分配页表所在内存空间并初始化页表;
2. 设置好页基址寄存器 (指向页表起始地址);
3. 刷新 TLB。

到现在为止, 看上去复杂无比的虚拟内存空间, 我们终于得以窥视一二了。

5.1.3.1.6 物理内存管理的设计思路

5.1.3.1.6.1 本小节内容

5.1.3.1.6.2 物理内存管理的实现

在管理虚拟内存之前, 我们首先需要能够管理物理内存, 毕竟所有虚拟内存页都要对应到物理内存页才能使用。

不妨把我们的内存管理模块划分为物理内存管理和虚拟内存管理两个模块。

物理内存管理应当为虚拟内存管理提供这样的接口:

- 检查当前还有多少空闲的物理页, 返回空闲的物理页数目
- 给出 n , 尝试分配 n 个物理页, 可以返回一个起始地址和连续的物理页数目, 也可能分配一些零散的物理页, 返回一个连起来的链表。
- 给出起始地址和 n , 释放 n 个连续的物理页

在 `kern_init()` 里, 我们调用一个新函数: `pmm_init()`, `kern_init()` 函数我们在之前就有学习过, 这里我们只是新增一个调用 `pmm_init()` 的接口。

```
// kern/init/init.c
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init(); // init the console
    const char *message = "(THU.CST) os is loading ...\0";
    cputs(message);
    print_kerninfo();

    idt_init(); // init interrupt descriptor table
    pmm_init(); // 新东西!
    clock_init(); // init clock interrupt
    intr_enable(); // enable irq interrupt
    /* do nothing */
    while (1)
        ;
}
```

那么 `pmm_init()` 究竟是用来干什么的呢? 其实 `pmm_init()` 主要就是用来主要负责初始化物理内存管理, 我们可以在 `pmm.c` 文件进行初始化操作。

```
// kern/mm/pmm.c
/* pmm_init - initialize the physical memory management */
void pmm_init(void) {
    // We need to alloc/free the physical memory (granularity is 4KB or other size).
    // So a framework of physical memory manager (struct pmm_manager) is defined in pm
    ↪ m.h
```

(续下页)

(接上页)

```

// First we should init a physical memory manager(pmm) based on the framework.
// Then pmm can alloc/free the physical memory.
init_pmm_manager();

// detect physical memory space, reserve already used memory,
// then use pmm->init_memmap to create free page list
page_init();

// use pmm->check to verify the correctness of the alloc/free function in a pmm
check_alloc_page();

extern char boot_page_table_sv39[]; //我们把汇编里定义的页表所在位置的符号声明进来
satp_virtual = (pte_t*)boot_page_table_sv39;
satp_physical = PADDR(satp_virtual); //然后输出页表所在的地址
cprintf("satp virtual address: 0x%016lx\nsatp physical address: 0x%016lx\n", satp_
->virtual, satp_physical);
}

```

check_alloc_page() 是对物理内存分配功能的一个测试。我们重点关注 page_init()

我们在 lab2 增加了一些功能, 方便我们编程:

- 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 sstatus 寄存器中的中断使能位 (SIE) 信息并屏蔽中断的功能, 另一个是根据保存的中断使能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: 定义了对一个二进制位进行读写的原子操作, 确保相关操作不被中断打断。包括 set_bit() 设置某个二进制位的值为 1, change_bit() 给某个二进制位取反, test_bit() 返回某个二进制位的值。

```

// kern/sync/sync.h
#ifndef __KERN_SYNC_SYNC_H__
#define __KERN_SYNC_SYNC_H__

#include <defs.h>
#include <intr.h>
#include <riscv.h>

static inline bool __intr_save(void) {
    if (read_csr(ssstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void __intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}

//思考: 这里宏定义的 do{}while(0)起什么作用?
#define local_intr_save(x) \
    do {

```

(续下页)

(接上页)

```

    x = __intr_save(); \
} while (0)
#define local_intr_restore(x) __intr_restore(x);

#endif /* !__KERN_SYNC_SYNC_H__ */

```

list.h 里面实现了一个简单的双向链表。虽然接口很多, 但是只要对链表熟悉, 不难理解。如果理解不了, 可以先去学学数据结构这门课。

```

// libs/list.h
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct list_entry list_entry_t;

static inline void list_init(list_entry_t *elm) __attribute__((always_inline));
static inline void list_add(list_entry_t *listelm, list_entry_t *elm) __attribute__((always_inline));
static inline void list_add_before(list_entry_t *listelm, list_entry_t *elm) __attribute__((always_inline));
static inline void list_add_after(list_entry_t *listelm, list_entry_t *elm) __attribute__((always_inline));
static inline void list_del(list_entry_t *listelm) __attribute__((always_inline));
static inline void list_del_init(list_entry_t *listelm) __attribute__((always_inline));
static inline bool list_empty(list_entry_t *list) __attribute__((always_inline));
static inline list_entry_t *list_next(list_entry_t *listelm) __attribute__((always_inline));
static inline list_entry_t *list_prev(list_entry_t *listelm) __attribute__((always_inline));
//下面两个函数仅在内部使用, 不对外开放作为接口。
static inline void __list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t *next) __attribute__((always_inline));
static inline void __list_del(list_entry_t *prev, list_entry_t *next) __attribute__((always_inline));

```

看起来 list.h 里面定义的 list_entry 并没有数据域, 但是, 如果我们把 list_entry 作为其他结构体的成员, 就可以利用 C 语言结构体内存连续布局的特点, 从 “list_entry” 的地址获得它所在的上一级结构体。于是我们定义了可以连成链表的 Page 结构体和一系列对它做操作的宏。这个结构体用来管理物理内存。

```

// libs/defs.h

/* Return the offset of 'member' relative to the beginning of a struct type */
#define offsetof(type, member) \
    ((size_t) (&((type *)0)->member))

/* *
 * to_struct - get the struct from a ptr
 * @ptr:      a struct pointer of member
 * @type:     the type of the struct this is embedded in
 * @member:   the name of the member within the struct
 * */
#define to_struct(ptr, type, member) \

```

(续下页)

(接上页)

```

((type *) ((char *) (ptr) - offsetof(type, member)))

// kern/mm/memlayout.h
/* *
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as physical address.
 * */
struct Page {
    int ref;                // page frame's reference counter
    uint64_t flags;         // array of flags that describe the status of the page frame
    unsigned int property;  // the num of free block, used in first fit pm manager
    list_entry_t page_link; // free list link
};

/* Flags describing the status of a page frame */
#define PG_reserved 0 // if this bit=1: the Page is reserved for
    ↪ kernel, cannot be used in alloc/free_pages; otherwise, this bit=0
#define PG_property 1 // if this bit=1: the Page is the head page
    ↪ of a free memory block (contains some continuous address pages), and can be used in
    ↪ alloc_pages; if this bit=0: if the Page is the the head page of a free memory block,
    ↪ then this Page and the memory block is allocated. Or this Page isn't the head page.
//这几个对page操作的宏用到了atomic.h的原子操作
#define SetPageReserved(page) set_bit(PG_reserved, &((page)->flags))
#define ClearPageReserved(page) clear_bit(PG_reserved, &((page)->flags))
#define PageReserved(page) test_bit(PG_reserved, &((page)->flags))
#define SetPageProperty(page) set_bit(PG_property, &((page)->flags))
#define ClearPageProperty(page) clear_bit(PG_property, &((page)->flags))
#define PageProperty(page) test_bit(PG_property, &((page)->flags))

// convert list entry to page
#define le2page(le, member) \
    to_struct((le), struct Page, member)

/* 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;

```

我们知道, 物理内存通常是一片 RAM, 我们可以把它看成一个以字节为单位的大数组, 通过物理地址找到对应的位置进行读写。但是, 物理地址并不仅仅只能访问物理内存, 也可以用来访问其他的外设, 因此你也可以认为物理内存也算是一种外设。

这样设计是因为: 如果访问其他外设要使用不同的指令 (如 x86 单独提供了 **in, out** 指令来访问不同于内存的 IO 地址空间), 会比较麻烦, 于是很多 CPU (如 RISC-V, ARM, MIPS 等) 通过 MMIO (Memory Mapped I/O) 技术将外设映射到一段物理地址, 这样我们访问其他外设就和访问物理内存一样啦!

我们先不管那些外设, 目前我们只关注物理内存。

5.1.3.1.6.3 物理内存探测的设计思路

操作系统怎样知道物理内存所在的那段物理地址呢？在 RISC-V 中，这个一般是由 bootloader，即 OpenSBI 来完成的。它来完成对于包括物理内存在内的各外设的扫描，将扫描结果以 DTB(Device Tree Blob) 的格式保存在物理内存中的某个地方。随后 OpenSBI 会将其地址保存在 a1 寄存器中，给我们使用。

这个扫描结果描述了所有外设的信息，当中也包括 Qemu 模拟的 RISC-V 计算机中的物理内存。

扩展 Qemu 模拟的 RISC-V virt 计算机中的物理内存

通过查看 virt.c 的 virt_memmap[] 的定义，可以了解到 Qemu 模拟的 RISC-V virt 计算机的详细物理内存布局。可以看到，整个物理内存中有不少内存空洞（即含义为 unmapped 的地址空间），也有很多外设特定的地址空间，现在我们看不懂没有关系，后面会慢慢涉及到。目前只需关心最后一块含义为 DRAM 的地址空间，这就是 OS 将要管理的 128MB 的内存空间。

起始地址	终止地址	含义
0x0	0x100	QEMU VIRT_DEBUG
0x100	0x1000	unmapped
0x1000	0x12000	QEMU MROM (包括 hard-coded reset vector; device tree)
0x12000	0x100000	unmapped
0x100000	0x101000	QEMU VIRT_TEST
0x101000	0x2000000	unmapped
0x2000000	0x2010000	QEMU VIRT_CLINT
0x2010000	0x3000000	unmapped
0x3000000	0x3010000	QEMU VIRT_PCIE_PIO
0x3010000	0xc000000	unmapped
0xc000000	0x10000000	QEMU VIRT_PLIC
0x10000000	0x10000100	QEMU VIRT_UART0
0x10000100	0x10001000	unmapped
0x10001000	0x10002000	QEMU VIRT_VIRTIO
0x10002000	0x20000000	unmapped
0x20000000	0x24000000	QEMU VIRT_FLASH
0x24000000	0x30000000	unmapped
0x30000000	0x40000000	QEMU VIRT_PCIE_ECAM
0x40000000	0x80000000	QEMU VIRT_PCIE_MMIO
0x80000000	0x88000000	DRAM 缺省 128MB，大小可配置

不过为了简单起见，我们并不打算自己去解析这个结果。因为我们知道，Qemu 规定的 DRAM 物理内存的起始物理地址为 0x80000000。而在 Qemu 中，可以使用 -m 指定 RAM 的大小，默认是 128MiB。因此，默认的 DRAM 物理内存地址范围就是 [0x80000000, 0x88000000)。我们直接将 DRAM 物理内存结束地址硬编码到内核中：

```
// kern/mm/memlayout.h

#define KERNBASE          0xFFFFFFFFFC020000
#define KMEMSIZE          0x7E00000
#define KERNTOP           (KERNBASE + KMEMSIZE)

#define PHYSICAL_MEMORY_END      0x88000000
#define PHYSICAL_MEMORY_OFFSET  0xFFFFFFFF40000000 //物理地址和虚拟地址的偏移量
#define KERNEL_BEGIN_PADDR      0x80200000
#define KERNEL_BEGIN_VADDR      0xFFFFFFFFFC020000
```

但是，有一部分 DRAM 空间已经被占用，不能用来存别的东西了！

- 物理地址空间 [0x80000000, 0x80200000) 被 OpenSBI 占用;
- 物理地址空间 [0x80200000, KernelEnd) 被内核各代码与数据段占用;
- 其实设备树扫描结果 DTB 还占用了一部分物理内存, 不过由于我们不打算使用它, 所以可以将它所占用的空间用来存别的东西。

于是, 我们可以用来存别的东西的物理内存的物理地址范围是: [KernelEnd, 0x88000000)。这里的 KernelEnd 为内核代码结尾的物理地址。在 kernel.ld 中定义的 end 符号为内核代码结尾的虚拟地址。

为了管理物理内存, 我们需要在内核里定义一些数据结构, 来存储”当前使用了哪些物理页面, 哪些物理页面没被使用”这样的信息, 使用的是 Page 结构体。我们将一些 Page 结构体在内存里排列在内核后面, 这要占用一些内存。而摆放这些 Page 结构体的物理页面, 以及内核占用的物理页面, 之后都无法再使用了。我们用 page_init() 函数给这些管理物理内存的结构体做初始化。下面是代码:

```
// kern/mm/pmm.h

/* *
 * PADDR - takes a kernel virtual address (an address that points above
 * KERNBASE),
 * where the machine's maximum 256MB of physical memory is mapped and returns
 * the
 * corresponding physical address. It panics if you pass it a non-kernel
 * virtual address.
 */
#define PADDR(kva) \
({ \
    uintptr_t __m_kva = (uintptr_t)(kva); \
    if (__m_kva < KERNBASE) { \
        panic("PADDR called with invalid kva %08lx", __m_kva); \
    } \
    __m_kva - va_pa_offset; \
})

/* *
 * KADDR - takes a physical address and returns the corresponding kernel virtual
 * address. It panics if you pass an invalid physical address.
 */
/*
#define KADDR(pa) \
({ \
    uintptr_t __m_pa = (pa); \
    size_t __m_ppn = PPN(__m_pa); \
    if (__m_ppn >= npage) { \
        panic("KADDR called with invalid pa %08lx", __m_pa); \
    } \
    (void *) (__m_pa + va_pa_offset); \
})
*/
extern struct Page *pages;
extern size_t npage;
```

```
// kern/mm/pmm.c

// pages指针保存的是第一个Page结构体所在的位置, 也可以认为是Page结构体组成的数组的开头
// 由于C语言的特性, 可以把pages作为数组名使用, pages[i]表示顺序排列的第i个结构体
struct Page *pages;
size_t npage = 0;
```

(续下页)

(接上页)

```

uint64_t va_pa_offset;
// memory starts at 0x80000000 in RISC-V
const size_t nbase = DRAM_BASE / PGSIZE;
// (npage - nbase) 表示物理内存的页数

static void page_init(void) {
    va_pa_offset = PHYSICAL_MEMORY_OFFSET; // 硬编码 0xFFFFFFFF40000000

    uint64_t mem_begin = KERNEL_BEGIN_PADDR; // 硬编码 0x80200000
    uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;
    uint64_t mem_end = PHYSICAL_MEMORY_END; // 硬编码 0x88000000

    cprintf("physical memory map:\n");
    cprintf("  memory: 0x%016lx, [0x%016lx, 0x%016lx].\n", mem_size, mem_begin,
        mem_end - 1);

    uint64_t maxpa = mem_end;

    if (maxpa > KERNTOP) {
        maxpa = KERNTOP;
    }

    npage = maxpa / PGSIZE;

    extern char end[];
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
    // 把 pages 指针指向内核所占内存空间结束后的第一页

    // 一开始把所有页面都设置为保留给内核使用的，之后再设置哪些页面可以分配给其他程序
    for (size_t i = 0; i < npage - nbase; i++) {
        SetPageReserved(pages + i); // 记得吗？在 kern/mm/memlayout.h 定义的
    }

    // 从这个地方开始才是我们可以自由使用的物理内存
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * (npage - nbase)
    ↪);

    // 按照页面大小 PGSIZE 进行对齐，ROUNDUP, ROUNDDOWN 是在 libs/defs.h 定义的
    mem_begin = ROUNDUP(freemem, PGSIZE);
    mem_end = ROUNDDOWN(mem_end, PGSIZE);
    if (freemem < mem_end) {
        // 初始化我们可以自由使用的物理内存
        init_memmap(pa2page(mem_begin), (mem_end - mem_begin) / PGSIZE);
    }
}

```

在 `page_init()` 的代码里，我们调用了函数 `init_memmap()`，这和我们的另一个结构体 `pmm_manager` 有关。虽然 C 语言基本上不支持面向对象，但我们可以用类似面向对象的思路，把“物理内存管理”的功能集中给一个结构体。我们甚至可以让函数指针作为结构体的成员，强行在 C 语言里支持了“成员函数”。可以看到，我们调用的 `init_memmap()` 实际上又调用了 `pmm_manager` 的一个“成员函数”。

```

// kern/mm/pmm.c

// physical memory management
const struct pmm_manager *pmm_manager;

```

(续下页)

(接上页)

```

// init_memmap - call pmm->init_memmap to build Page struct for free memory
static void init_memmap(struct Page *base, size_t n) {
    pmm_manager->init_memmap(base, n);
}

// kern/mm/pmm.h
#ifndef __KERN_MM_PMM_H__
#define __KERN_MM_PMM_H__

#include <assert.h>
#include <atomic.h>
#include <defs.h>
#include <memlayout.h>
#include <mmu.h>
#include <riscv.h>

// pmm_manager is a physical memory management class. A special pmm manager -
// XXX_pmm_manager
// only needs to implement the methods in pmm_manager class, then
// XXX_pmm_manager can be used
// by ucore to manage the total physical memory space.
struct pmm_manager {
    const char *name; // XXX_pmm_manager's name
    void (*init)(
        void); // 初始化XXX_pmm_manager内部的数据结构 (如空闲页面的链表)
    void (*init_memmap)(
        struct Page *base,
        size_t n); //知道了可用的物理页面数目之后, 进行更详细的初始化
    struct Page *(*alloc_pages)(
        size_t n); // 分配至少n个物理页面, 根据分配算法可能返回不同的结果
    void (*free_pages)(struct Page *base, size_t n); // free >=n pages with
                                                    // "base" addr of Page
                                                    // descriptor
                                                    // structures(memlayout.h)
    size_t (*nr_free_pages)(void); // 返回空闲物理页面的数目
    void (*check)(void); // 测试正确性
};

extern const struct pmm_manager *pmm_manager;

void pmm_init(void);

struct Page *alloc_pages(size_t n);
void free_pages(struct Page *base, size_t n);
size_t nr_free_pages(void); // number of free pages

#define alloc_page() alloc_pages(1)
#define free_page(page) free_pages(page, 1)

```

pmm_manager 提供了各种接口: 分配页面, 释放页面, 查看当前空闲页面数。但是我们好像始终没看见 pmm_manager 内部对这些接口的实现, 其实是因为那些接口只是作为函数指针, 作为 pmm_manager 的一部分, 我们需要把那些函数指针变量赋值为真正的函数名称。

还记得最早我们在 pmm_init() 里首先调用了 init_pmm_manager(), 在这里面我们把 pmm_manager 的指针赋值成 &default_pmm_manager, 看起来我们在这里实现了那些接口。

```

// init_pmm_manager - initialize a pmm_manager instance
static void init_pmm_manager(void) {
    pmm_manager = &default_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}

// alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE
// memory
struct Page *alloc_pages(size_t n) {
    // 在这里编写你的物理内存分配算法。
    // 你可以参考nr_free_pages() 函数进行设计,
    // 了解物理内存管理器的工作原理, 然后在这里实现自己的分配算法。
    // 实现算法后, 调用 pmm_manager->alloc_pages(n) 来分配物理内存,
    // 然后返回分配的 Page 结构指针。
}

// free_pages - call pmm->free_pages to free a continuous n*PAGESIZE memory
void free_pages(struct Page *base, size_t n) {
    // 在这里编写你的物理内存释放算法。
    // 你可以参考nr_free_pages() 函数进行设计,
    // 了解物理内存管理器的工作原理, 然后在这里实现自己的释放算法。
    // 实现算法后, 调用 pmm_manager->free_pages(base, n) 来释放物理内存。
}

// nr_free_pages - call pmm->nr_free_pages to get the size (nr*PAGESIZE)
// of current free memory
size_t nr_free_pages(void) {
    size_t ret;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        ret = pmm_manager->nr_free_pages();
    }
    local_intr_restore(intr_flag);
    return ret;
}

```

到现在, 我们距离完整的内存管理, 就只差 default_pmm_manager 结构体的实现了, 也就是我们要在里面实现页面分配算法。

5.1.3.1.6.4 页面分配算法

我们在 default_pmm.c 定义了一个 pmm_manager 类型的结构体, 并实现它的接口

```

// kern/mm/default_pmm.h
#ifndef __KERN_MM_DEFAULT_PMM_H__
#define __KERN_MM_DEFAULT_PMM_H__

#include <pmm.h>

extern const struct pmm_manager default_pmm_manager;

#endif /* ! __KERN_MM_DEFAULT_PMM_H__ */

```

较为关键的, 是一开始如何初始化所有可用页面, 以及如何分配和释放页面。大家可以学习下面的代码, 其实现了 First Fit 算法。

```

// kern/mm/default_pmm.c
free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0; //nr_free 可以理解为在这里可以使用的一个全局变量, 记录可用的物理页面数
}

static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}

static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
    }
}

```

(续下页)

(接上页)

```

        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}

static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}

```

(续下页)

(接上页)

```
    }  
}  
  
const struct pmm_manager default_pmm_manager = {  
    .name = "default_pmm_manager",  
    .init = default_init,  
    .init_memmap = default_init_memmap,  
    .alloc_pages = default_alloc_pages,  
    .free_pages = default_free_pages,  
    .nr_free_pages = default_nr_free_pages,  
    .check = default_check,  
};
```

所谓 First Fit 算法就是当需要分配页面时, 它会从空闲页块链表中找到第一个适合大小的空闲页块, 然后进行分配。当释放页面时, 它会将释放的页面添加回链表, 并在必要时合并相邻的空闲页块, 以最大限度地减少内存碎片。

完成页面分配算法后我们的物理内存管理算是基本实现了, 接下来请同学们完成本次实验练习。

5.1.4 实验报告要求

从 git server 网站上取得 ucore_lab 后, 进入目录 labcodes/lab2, 完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 labcodes/lab2 下存放实验报告, 实验报告文档命名为 lab2-学生 ID.md。推荐使用 **markdown** 格式。对于 lab2 中编程任务, 完成编写之后, 再通过 git push 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

lab3: 缺页异常和页面置换

做完实验二后，大家可以了解并掌握物理内存管理中页表的建立过程以及页面分配算法的具体实现。本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，学习如何在磁盘上缓存内存页，从而能够支持虚拟内存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。

6.1 本章内容

6.1.1 实验目的

- 了解虚拟内存的 Page Fault 异常处理实现
- 了解页替换算法在操作系统中的实现
- 学会如何使用多级页表，处理缺页异常（Page Fault），实现页面置换算法。

6.1.2 实验内容

本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成 Page Fault 异常处理和部分页面替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交叉访问。如果大家有余力，可以尝试完成扩展练习，实现 LRU 页替换算法。

6.1.2.1 本节内容

6.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 `ucore_lab` 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

6.1.2.1.1.1 练习 0: 填写已有实验

本实验依赖实验 1/2。请把你做的实验 1/2 的代码填入本实验中代码中有“LAB1”, “LAB2”的注释相应部分。

6.1.2.1.1.2 练习 1: 理解基于 FIFO 的页面替换算法 (思考题)

描述 FIFO 页面置换算法下, 一个页面从被换入到被换出的过程中, 会经过代码里哪些函数/宏的处理 (或者说, 需要调用哪些函数/宏), 并用简单的一两句话描述每个函数在过程中做了什么? (为了方便同学们完成练习, 所以实际上我们的项目代码和实验指导的还是略有不同, 例如我们将 FIFO 页面置换算法头文件的大部分代码放在了 `kern/mm/swap_fifo.c` 文件中, 这点请同学们注意)

- 至少正确指出 10 个不同的函数分别做了什么? 如果少于 10 个将酌情给分。我们认为只要函数原型不同, 就算两个不同的函数。要求指出对执行过程有实际影响, 删去后会导致输出结果不同的函数 (例如 `assert`) 而不是 `cprintf` 这样的函数。如果你选择的函数不能完整地体现“从换入到换出”的过程, 比如 10 个函数都是页面换入的时候调用的, 或者解释功能的时候只解释了这 10 个函数在页面换入时的功能, 那么也会扣除一定的分数

6.1.2.1.1.3 练习 2: 深入理解不同分页模式的工作原理 (思考题)

`get_pte()` 函数 (位于 `kern/mm/pmm.c`) 用于在页表中查找或创建页表项, 从而实现对指定线性地址对应的物理页的访问和映射操作。这在操作系统中的分页机制下, 是实现虚拟内存与物理内存之间映射关系非常重要的内容。

- `get_pte()` 函数中有两段形式类似的代码, 结合 `sv32`, `sv39`, `sv48` 的异同, 解释这两段代码为什么如此相像。
- 目前 `get_pte()` 函数将页表项的查找和页表项的分配合并在一个函数里, 你认为这种写法好吗? 有没有必要把两个功能拆开?

Chapter 6. lab3: 缺页异常和页面置换

编译方法

编译并运行代码的命令如下:

```
make
make qemu
```

则可以得到如下显示界面 (仅供参考)

```
chenyu$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xc0200036 (virtual)
  etext 0xc02042cc (virtual)
  edata 0xc020a040 (virtual)
  end   0xc02115a0 (virtual)
Kernel executable memory footprint: 70KB
memory management: default_pmm_manager
membegin 80200000 memend 88000000 mem_size 7e00000
physcial memory map:
  memory: 0x07e00000, [0x80200000, 0x87ffffff].
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_vma_struct() succeeded!
Store/AMO page fault
page fault at 0x00000100: K/W
check_pgfault() succeeded!
check_vmm() succeeded.
SWAP: manager = clock swap manager
BEGIN check_swap: count 2, total 31661
setup Page Table for vaddr 0x1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
Store/AMO page fault
page fault at 0x00001000: K/W
Store/AMO page fault
page fault at 0x00002000: K/W
Store/AMO page fault
page fault at 0x00003000: K/W
Store/AMO page fault
page fault at 0x00004000: K/W
set up init env for check_swap over!
Store/AMO page fault
page fault at 0x00005000: K/W
curr_ptr 0xffffffffc02258a8
curr_ptr 0xffffffffc02258a8
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
Load page fault
page fault at 0x00001000: K/R
curr_ptr 0xffffffffc02258f0
curr_ptr 0xffffffffc02258f0
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 1, total is 8
check_swap() succeeded!
```

(续下页)

(接上页)

```

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

```

通过上述运行结果，我们可以看到 `ucore` 在显示特殊内核符号地址后，展示了物理内存的映射信息，并进行了内存管理相关的检查。接着 `ucore` 进入页面置换和缺页异常处理的测试，并成功完成了页面置换测试。最后，`ucore` 设置了定时器中断，在分页模式下响应时钟中断。

6.1.3 虚拟内存管理

6.1.3.1 本节内容

6.1.3.1.1 基本原理概述

6.1.3.1.1.1 虚拟内存

什么是虚拟内存？简单地说是指程序员或 CPU “看到” 的内存。但有几点需要注意：

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有实际的物理内存单元，那二者的地址一般是不相等的；
3. 通过操作系统实现的某种内存映射可建立虚拟内存与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自动转换为一个物理内存地址。

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，在有了分页机制后，程序员或 CPU “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页表项来限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，这种技术称为按需分页（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）。这种内存管理技术给了程序员更大的内存“空间”，从而可以让更多的程序在内存中并发运行。

6.1.3.1.2 实验执行流程概述

本次实验主要完成 `ucore` 内核对虚拟内存的管理工作。首先我们要完成初始化虚拟内存管理机制，即需要设置好哪些页需要放在物理内存中，哪些页不需要放在物理内存中，而是可被换出到硬盘上，并涉及完善建立页表映射、页访问异常处理操作等函数的实现。然后执行一组访存测试，看看我们建立的页表项是否能够正确完成虚实地址映射，是否正确描述了虚拟内存页在物理内存中还是在硬盘上，是否能够正确把虚拟内存页在物理内存和硬盘之间进行传递，是否正确实现了页面替换算法等。lab3 的总体执行流程如下：

首先，整个实验过程以 `ucore` 的总控函数 `init` 为起点。在初始化阶段，首先调用 `pmm_init` 函数完成物理内存的管理初始化。接下来，执行中断和异常相关的初始化工作。此过程涉及调用 `pic_init` 函数和 `idt_init` 函数，用于初始化处理器中断控制器（PIC）和中断描述符表（IDT），与之前的 lab1 中断和异常初始化工作相同。随后，调用 `vmm_init` 函数进行虚拟内存管理机制的初始化。在此阶段，主要是建立虚拟地址到物理地址的映射关系，为虚拟内存提供管理支持。继续执行初始化过程，接下来调用 `ide_init` 函数完成对用于页面换入

和换出的硬盘（通常称为 swap 硬盘）的初始化工作。在这个阶段，ucore 准备好了对硬盘数据块的读写操作，以便后续页面置换算法的实现。最后，完成整个初始化过程，调用 swap_init 函数用于初始化页面置换算法，这其中包括 Clock 页替换算法的相关数据结构和初始化步骤。通过 swap_init，ucore 确保页面置换算法准备就绪，可以在需要时执行页面换入和换出操作，以优化内存的利用。

下面我们就来看看如何使用多级页表进行虚拟内存管理和页面置换，ucore 在实现上述技术时，需要解决三个关键问题：

1. 当程序运行中访问内存产生 page fault 异常时，如何判定这个引起异常的虚拟地址内存访问是越界、写只读页的“非法地址”访问还是由于数据被临时换出到磁盘上或还没有分配内存的“合法地址”访问？
2. 何时进行请求调页/页换入换出处理？
3. 如何在现有 ucore 的基础上实现页替换算法？

接下来将进一步分析完成 lab3 主要注意的关键问题和涉及的关键数据结构。

6.1.3.1.3 关键数据结构和相关函数分析

大家在之前的实验中都尝试了 make qemu 这条指令，但实际上我们在 QEMU 里并没有真正模拟“硬盘”。为了实现“页面置换”的效果，我们采取的措施是，从内核的静态存储(static)区里面分出一块内存，声称这块存储区域是“硬盘”，然后包裹一下给出“硬盘 IO”的接口。这听上去很令人费解，但如果仔细思考一下，内存和硬盘，除了一个掉电后数据易失一个不易失，一个访问快一个访问慢，其实并没有本质的区别。对于我们的页面置换算法来说，也不要求硬盘上存多余页面的交换空间能够“不易失”，反正这些页面存在内存里的时候就是易失的。理论上，我们完全可以把一块机械硬盘加以改造，写好驱动之后，插到主板的内存插槽上作为内存条使用，当然我们要忽视性能方面的差距。那么我们就把 QEMU 模拟出来的一块 ram 叫做“硬盘”，用作页面置换时的交换区，完全没有问题。你可能会觉得，这样折腾一通，我们总共能使用的页面数并没有增加，原先能直接在内存里使用的一些页面变成了“硬盘”，只是在自娱自乐。这样的想法当然是没错的，不过我们在这里只是想介绍页面置换的原理，而并不关心实际性能。

那么模拟二等这个过程我们在 driver/ide.h driver/ide.c fs/fs.h fs/swapfs.h fs/swapfs.c 通过修改代码一步步实现。

首先我们先了解一下各个文件名的含义，虽然这些文件名你可以自定义设置，但我们还是一般采用比较规范的方式进行命名，这样可以帮助你更好的理解项目结构。

ide 在这里不是 integrated development environment 的意思，而是 Integrated Drive Electronics 的意思，表示的是一种标准的硬盘接口。我们这里写的东西和 Integrated Drive Electronics 并不相关，这个命名是 ucore 的历史遗留。

fs 全称为 file system，我们这里其实并没有“文件”的概念，这个模块称作 fs 只是说明它是“硬盘”和内核之间的接口。

```
// kern/driver/ide.c
/*
#include"s
*/

void ide_init(void) {}

#define MAX_IDE 2
#define MAX_DISK_NSECS 56
static char ide[MAX_DISK_NSECS * SECTSIZE];

bool ide_device_valid(unsigned short ideno) { return ideno < MAX_IDE; }

size_t ide_device_size(unsigned short ideno) { return MAX_DISK_NSECS; }
```

(续下页)

(接上页)

```

int ide_read_secs(unsigned short ideno, uint32_t secno, void *dst,
                 size_t nsecs) {
    //ideno: 假设挂载了多块磁盘, 选择哪一块磁盘 这里我们其实只有一块“磁盘”, 这个参数
    ↪ 就没用到
    int iobase = secno * SECTSIZE;
    memcpy(dst, &ide[iobase], nsecs * SECTSIZE);
    return 0;
}

int ide_write_secs(unsigned short ideno, uint32_t secno, const void *src,
                  size_t nsecs) {
    int iobase = secno * SECTSIZE;
    memcpy(&ide[iobase], src, nsecs * SECTSIZE);
    return 0;
}

```

可以看到, 我们这里所谓的“硬盘 IO”, 只是在内存里用 `memcpy` 把数据复制来复制去。同时为了逼真地模拟磁盘, 我们只允许以磁盘扇区为数据传输的基本单位, 也就是一次传输的数据必须是 512 字节的倍数, 并且必须对齐。

```

// kern/fs/fs.h
#ifndef __KERN_FS_FS_H__
#define __KERN_FS_FS_H__

#include <mmu.h>

#define SECTSIZE          512
#define PAGE_NSECT        (PGSIZE / SECTSIZE) //一页需要几个磁盘扇区?

#define SWAP_DEV_NO       1

#endif /* !__KERN_FS_FS_H__ */

// kern/mm/swap.h
extern size_t max_swap_offset;
// kern/mm/swap.c
size_t max_swap_offset;

// kern/fs/swapfs.c
#include <swap.h>
#include <swapfs.h>
#include <mmu.h>
#include <fs.h>
#include <ide.h>
#include <pmm.h>
#include <assert.h>

void swapfs_init(void) { //做一些检查
    static_assert((PGSIZE % SECTSIZE) == 0);
    if (!ide_device_valid(SWAP_DEV_NO)) {
        panic("swap fs isn't available.\n");
    }
    //swap.c/swap.h里的全局变量
    max_swap_offset = ide_device_size(SWAP_DEV_NO) / (PAGE_NSECT);
}

```

(续下页)

(接上页)

```

int swapfs_read(swap_entry_t entry, struct Page *page) {
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT, page2kva(page),
        ↪ PAGE_NSECT);
}

int swapfs_write(swap_entry_t entry, struct Page *page) {
    return ide_write_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT, page2kva(page)
        ↪, PAGE_NSECT);
}

```

6.1.3.1.4 页表项设计思路

我们定义一些对 sv39 页表项 (Page Table Entry) 进行操作的宏。这里的定义和我们在 Lab2 里介绍的定义一致。

有时候我们把多级页表中较高级别的页表 (“页表的页表”) 叫做 Page Directory。在实验二中我们知道 sv39 中采用的是三级页表, 那么在这里我们把页表项里从高到低三级页表的页码分别称作 PDX1, PDX0 和 PTX (Page Table Index)。

```

// kern/mm/mmu.h
#ifndef __KERN_MM_MMU_H__
#define __KERN_MM_MMU_H__

#ifndef __ASSEMBLER__
#include <defs.h>
#endif /* !__ASSEMBLER__ */

// A linear address 'la' has a four-part structure as follows:
//
// +-----9-----+-----9-----+-----9-----+-----12-----+
// | Page Directory | Page Directory |   Page Table   | Offset within Page |
// |      Index 1   |   Index 2     |               |                       |
// +-----+-----+-----+-----+
// \-- PDX1(la) --/ \-- PDX0(la) --/ \-- PTX(la) --/ \-- PGOFF(la) ----/
// \-----PPN(la)-----/
//
// The PDX1, PDX0, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).

// RISC-V uses 39-bit virtual address to access 56-bit physical address!
// Sv39 virtual address:
// +---9---+---9---+---9---+---12---+
// |  VPN[2] | VPN[1] | VPN[0] | PGOFF |
// +-----+-----+-----+-----+
//
// Sv39 physical address:
// +---26---+---9---+---9---+---12---+
// |  PPN[2] | PPN[1] | PPN[0] | PGOFF |
// +-----+-----+-----+-----+
//
// Sv39 page table entry:
// +---26---+---9---+---9---+---2---+---8-----+
// |  PPN[2] | PPN[1] | PPN[0] |Reserved|D|A|G|U|X|W|R|V|
// +-----+-----+-----+-----+-----+

```

(续下页)

(接上页)

```

// page directory index
#define PDX1(la) (((uintptr_t)(la)) >> PDX1SHIFT) & 0x1FF)
#define PDX0(la) (((uintptr_t)(la)) >> PDX0SHIFT) & 0x1FF)

// page table index
#define PTX(la) (((uintptr_t)(la)) >> PTXSHIFT) & 0x1FF)

// page number field of address
#define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)

// offset in page
#define PGOFF(la) (((uintptr_t)(la)) & 0xFFF)

// construct linear address from indexes and offset
#define PGADDR(d1, d0, t, o) ((uintptr_t)((d1) << PDX1SHIFT | (d0) << PDX0SHIFT | (t)
→ << PTXSHIFT | (o)))

// address in page table or page directory entry
// 把页表项里存储的地址拿出来
#define PTE_ADDR(pte) (((uintptr_t)(pte) & ~0x3FF) << (PTXSHIFT - PTE_PPN_SHIFT))
#define PDE_ADDR(pde) PTE_ADDR(pde)

/* page directory and page table constants */
#define NPDEENTRY 512 // page directory entries per page dir
→ ectory
#define NPTEENTRY 512 // page table entries per page table

#define PGSIZE 4096 // bytes mapped by a page
#define PGSHIFT 12 // log2(PGSIZE)
#define PTSIZE (PGSIZE * NPTEENTRY) // bytes mapped by a page directory e
→ ntry
#define PTSHIFT 21 // log2(PTSIZE)

#define PTXSHIFT 12 // offset of PTX in a linear address
#define PDX0SHIFT 21 // offset of PDX0 in a linear address
#define PDX1SHIFT 30 // offset of PDX1 in a linear address
#define PTE_PPN_SHIFT 10 // offset of PPN in a physical address

// page table entry (PTE) fields
#define PTE_V 0x001 // Valid
#define PTE_R 0x002 // Read
#define PTE_W 0x004 // Write
#define PTE_X 0x008 // Execute
#define PTE_U 0x010 // User

```

上述代码定义了一些与内存管理单元 (Memory Management Unit, MMU) 相关的宏和常量, 用于操作线性地址和物理地址, 以及页表项的字段。最后我们看一下 kern/init/init.c 里的变化:

```

// kern/init/init.c
int kern_init(void) {
    /* blabla */
    pmm_init(); // init physical memory management
    // 我们加入了多级页表的接
→ 口和测试

```

(续下页)

(接上页)

```

idt_init();                // init interrupt descriptor table

vmm_init();                // init virtual memory management
                           // 新增函数, 初始化虚拟内
→ 存管理并测试

ide_init();                // init ide devices. 新增函数, 初始化"硬盘".
                           // 其实这个函数啥也没做, 属
→ 于"历史遗留"

swap_init();               // init swap. 新增函数, 初始化页面置换机制并测试

clock_init();              // init clock interrupt
/* blabla */
}

```

可以看到的是我们在原本的基础上又新增了一个用来初始化“硬盘”的函数 `ide_init()`。

6.1.3.1.5 使用多级页表实现虚拟存储

要想实现虚拟存储, 我们需要把页表放在内存里, 并且需要有办法修改页表, 比如在页表里增加一个页面的映射或者删除某个页面的映射。

要想实现页面映射, 我们最主要需要修改的是两个接口:

- `page_insert()`, 在页表里建立一个映射
- `page_remove()`, 在页表里删除一个映射

这些内容都需要在 `kern/mm/pmm.c` 里面编写。然后我们可以在虚拟内存空间的第一个大大页 (Giga Page) 中建立一些映射来做测试。通过编写 `page_ref()` 函数用来检查映射关系是否实现, 这个函数会返回一个物理页面被多少个虚拟页面所对应。

```

static void check_pgdir(void) {
    // assert(npag<= KMEMSIZE / PGSIZE);
    // The memory starts at 2GB in RISC-V
    // so npag is always larger than KMEMSIZE / PGSIZE
    assert(npag<= KERNTOP / PGSIZE);
    //boot_pgdir是页表的虚拟地址
    assert(boot_pgdir != NULL && (uint32_t)PGOFF(boot_pgdir) == 0);
    assert(get_page(boot_pgdir, 0x0, NULL) == NULL);
    //get_page()尝试找到虚拟内存0x0对应的页, 现在当然是没有的, 返回NULL

    struct Page *p1, *p2;
    p1 = alloc_page(); //拿过来一个物理页面
    assert(page_insert(boot_pgdir, p1, 0x0, 0) == 0); //把这个物理页面通过多级页表映射
→ 到0x0
    pte_t *ptep;
    assert((ptep = get_pte(boot_pgdir, 0x0, 0)) != NULL);
    assert(pte2page(*ptep) == p1);
    assert(page_ref(p1) == 1);

    ptep = (pte_t *)KADDR(PDE_ADDR(boot_pgdir[0]));
    ptep = (pte_t *)KADDR(PDE_ADDR(ptep[0])) + 1;
    assert(get_pte(boot_pgdir, PGSIZE, 0) == ptep);
    //get_pte查找某个虚拟地址对应的页表项, 如果不存在这个页表项, 会为它分配各级的页表

```

(续下页)

(接上页)

```

p2 = alloc_page();
assert(page_insert(boot_pgdir, p2, PGSIZE, PTE_U | PTE_W) == 0);
assert((ptep = get_pte(boot_pgdir, PGSIZE, 0)) != NULL);
assert(*ptep & PTE_U);
assert(*ptep & PTE_W);
assert(boot_pgdir[0] & PTE_U);
assert(page_ref(p2) == 1);

assert(page_insert(boot_pgdir, p1, PGSIZE, 0) == 0);
assert(page_ref(p1) == 2);
assert(page_ref(p2) == 0);
assert((ptep = get_pte(boot_pgdir, PGSIZE, 0)) != NULL);
assert(pte2page(*ptep) == p1);
assert((*ptep & PTE_U) == 0);

page_remove(boot_pgdir, 0x0);
assert(page_ref(p1) == 1);
assert(page_ref(p2) == 0);

page_remove(boot_pgdir, PGSIZE);
assert(page_ref(p1) == 0);
assert(page_ref(p2) == 0);

assert(page_ref(pde2page(boot_pgdir[0])) == 1);
free_page(pde2page(boot_pgdir[0]));
boot_pgdir[0] = 0; //清除测试的痕迹

cprintf("check_pgdir() succeeded!\n");
}

```

在映射关系建立完成后, 如何新增一个映射关系和删除一个映射关系也是非常重要的内容, 我们来看 `page_insert()`, `page_remove()` 的实现。它们都涉及到调用两个对页表项进行操作的函数: `get_pte()` 和 `page_remove_pte()`

```

int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    //pgdir是页表基址(satp), page对应物理页面, la是虚拟地址
    pte_t *ptep = get_pte(pgdir, la, 1);
    //先找到对应页表项的位置, 如果原先不存在, get_pte()会分配页表项的内存
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page); //指向这个物理页面的虚拟地址增加了一个
    if (*ptep & PTE_V) { //原先存在映射
        struct Page *p = pte2page(*ptep);
        if (p == page) { //如果这个映射原先就有
            page_ref_dec(page);
        } else { //如果原先这个虚拟地址映射到其他物理页面, 那么需要删除映射
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = pte_create(page2ppn(page), PTE_V | perm); //构造页表项
    tlb_invalidate(pgdir, la); //页表改变之后要刷新TLB
    return 0;
}

void page_remove(pde_t *pgdir, uintptr_t la) {

```

(续下页)

(接上页)

```

pte_t *ptep = get_pte(pgdir, la, 0); //找到页表项所在位置
if (ptep != NULL) {
    page_remove_pte(pgdir, la, ptep); //删除这个页表项的映射
}
}
//删除一个页表项以及它的映射
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_V) { // (1) check if this page table entry is valid
        struct Page *page = pte2page(*ptep); // (2) find corresponding page to pte
        page_ref_dec(page); // (3) decrease page reference
        if (page_ref(page) == 0) {
            // (4) and free this page when page reference reaches 0
            free_page(page);
        }
        *ptep = 0; // (5) clear page table entry
        tlb_invalidate(pgdir, la); // (6) flush tlb
    }
}
//寻找(有必要的时候分配)一个页表项
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /* LAB2 EXERCISE 2: YOUR CODE
    *
    * If you need to visit a physical address, please use KADDR()
    * please read pmm.h for useful macros
    *
    * Maybe you want help comment, BELOW comments can help you finish the code
    *
    * Some Useful MACROS and DEFINES, you can use them in below implementation.
    * MACROS or Functions:
    *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
    *   KADDR(pa) : takes a physical address and returns the corresponding
    *   kernel virtual address.
    *   set_page_ref(page, 1) : means the page be referenced by one time
    *   page2pa(page) : get the physical address of memory which this (struct
    *   Page *) page manages
    *   struct Page * alloc_page() : allocation a page
    *   memset(void *s, char c, size_t n) : sets the first n bytes of the
    *   memory area pointed by s
    *
    *                                     to the specified value c.
    * DEFINES:
    *   PTE_P           0x001                // page table/directory entry
    *   flags bit : Present
    *   PTE_W           0x002                // page table/directory entry
    *   flags bit : Writeable
    *   PTE_U           0x004                // page table/directory entry
    *   flags bit : User can access
    */
    pde_t *pdep1 = &pgdir[PDX1(la)]; //找到对应的 Giga Page
    if (!(*pdep1 & PTE_V)) { //如果下一级页表不存在, 那就给它分配一页, 创造新页表
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        //我们现在在虚拟地址空间中, 所以要转化为KADDR再memset.
    }
}

```

(续下页)

(接上页)

```

//不管页表怎么构造, 我们确保物理地址和虚拟地址的偏移量始终相同, 那么就可以用这
→ 种方式完成对物理内存的访问。
*pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V); //注意这里 R, W, X 全零
}
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(1a)]; //再下一级页表
//这里的逻辑和前面完全一致, 页表不存在就现在分配一个
if (!(*pdep0 & PTE_V)) {
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) {
        return NULL;
    }
    set_page_ref(page, 1);
    uintptr_t pa = page2pa(page);
    memset(KADDR(pa), 0, PGSIZE);
    *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
}
//找到输入的虚拟地址 1a 对应的页表项的地址 (可能是刚刚分配的)
return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(1a)];
}

```

在 `entry.S` 里, 我们虽然构造了一个简单映射使得内核能够运行在虚拟空间上, 但是这个映射是比较粗糙的。

我们知道一个程序通常含有下面几段:

- `.text` 段: 存放代码, 需要是可读、可执行的, 但不可写。
- `.rodata` 段: 存放只读数据, 顾名思义, 需要可读, 但不可写亦不可执行。
- `.data` 段: 存放经过初始化的数据, 需要可读、可写。
- `.bss` 段: 存放经过零初始化的数据, 需要可读、可写。与 `.data` 段的区别在于由于我们知道它被零初始化, 因此在可执行文件中可以只存放该段的开头地址和大小而不用存全为 0 的数据。在执行时由操作系统进行处理。

我们看到各个段需要的访问权限是不同的。但是现在使用一个大大页 (Giga Page) 进行映射时, 它们都拥有相同的权限, 那么在现在的映射下, 我们甚至可以修改内核 `.text` 段的代码, 因为我们通过一个标志位 `W=1` 的页表项就可以完成映射, 但这显然会带来安全隐患。

因此, 我们考虑对这些段分别进行重映射, 使得他们的访问权限可以被正确设置。虽然还是每个段都还是映射以同样的偏移量映射到相同的地方, 但实现过程需要更加精细。

这里还有一个小坑: 对于我们最开始已经用特殊方式映射的一个大大页 (Giga Page), 该怎么对那里面的地址重新进行映射? 这个过程比较麻烦。但大家可以基本理解为放弃现有的页表, 直接新建一个页表, 在新页表里面完成重映射, 然后把 `satp` 指向新的页表, 这样就实现了重新映射。

6.1.3.1.6 处理缺页异常 (page fault 异常)

什么是缺页异常?

缺页异常是指 CPU 访问的虚拟地址时, MMU 没有办法找到对应的物理地址映射关系, 或者与该物理页的访问权不一致而发生的异常。

CPU 通过地址总线可以访问连接在地址总线上的所有外设, 包括物理内存、IO 设备等等, 但从 CPU 发出的访问地址并非是这些外设的地址总线上的物理地址, 而是一个虚拟地址, 由 MMU 将虚拟地址转换成物理地址再从地址总线上发出, MMU 上的这种虚拟地址和物理地址的转换关系是需要创建的, 并且还需要设置这个物理页的访问权限。

在实验二中我们有关内存的所有数据结构和相关操作都是直接针对实际存在的资源, 即针对物理内存空间的管理, 而没有从一般应用程序对内存的“需求”考虑, 所以我们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”。而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, 在 ucore 中我们是通过 page fault 异常处理来间接完成这二者之间的衔接。但需要注意的是在 lab3 中仅实现了简单的页面置换机制, 还没有涉及 lab4 和 lab5 才实现的内核线程和用户进程, 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。

须知: 哪些页面可以被换出?

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。

但在实验三实现的 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, 所以我们在实验三中仅仅通过执行 check_swap 函数在内核中分配一些页, 模拟对这些页的访问, 然后通过 do_pgfault 来调用 swap_map_swappable 函数来查询这些页的访问情况并间接调用相关函数, 换出“不常用”的页到磁盘上。

当我们引入了虚拟内存, 就意味着虚拟内存的空间可以远远大于物理内存, 也意味着程序可以访问“不对应物理内存页帧的虚拟内存地址”, 这时 CPU 应当抛出 Page Fault 这个异常。

回想一下, 我们处理异常的时候, 是在 kern/trap/trap.c 的 exception_handler() 函数里进行的。按照 scause 寄存器对异常的分类里, 有 CAUSE_LOAD_PAGE_FAULT 和 CAUSE_STORE_PAGE_FAULT 两个 case。之前我们并没有真正对异常进行处理, 只是简单输出一下就返回了, 但现在我们要真正进行 Page Fault 的处理。

```
// kern/trap/trap.c
static inline void print_pgfault(struct trapframe *tf) {
    cprintf("page fault at 0x%08x: %c/%c\n", tf->badvaddr,
            trap_in_kernel(tf) ? 'K' : 'U',
            tf->cause == CAUSE_STORE_PAGE_FAULT ? 'W' : 'R');
}

static int pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
    }
    panic("unhandled page fault.\n");
}

void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        /* .... other cases */
        case CAUSE_FETCH_PAGE_FAULT: // 取指令时发生的Page Fault先不处理
            cprintf("Instruction page fault\n");
            break;
        case CAUSE_LOAD_PAGE_FAULT:
            cprintf("Load page fault\n");
            if ((ret = pgfault_handler(tf)) != 0) {
                print_trapframe(tf);
                panic("handle pgfault failed. %e\n", ret);
            }
    }
}
```

(续下页)

(接上页)

```

        break;
    case CAUSE_STORE_PAGE_FAULT:
        cprintf("Store/AMO page fault\n");
        if ((ret = pgfault_handler(tf)) != 0) { //do_pgfault() 页面置换成功时返回 0
            print_trapframe(tf);
            panic("handle pgfault failed. %e\n", ret);
        }
        break;
    default:
        print_trapframe(tf);
        break;
}
}

```

这里的异常处理程序会把 Page Fault 分发给 kern/mm/vmm.c 的 do_pgfault() 函数并尝试进行页面置换。

之前我们进行物理页帧管理时有个功能没有实现，那就是动态的内存分配。管理虚拟内存的数据结构（页表）需要有空间进行存储，而我们又没有给它预先分配内存（也无法预先分配，因为事先不确定我们的页表需要分配多少内存），于是我们就需要设置接口来负责分配释放内存，这里我们选择的是 malloc/free 的接口。同样，我们也要在 pmm.h 里编写对物理页面和虚拟地址，物理地址进行转换的一些函数。

```

// kern/mm/pmm.c
void *kmalloc(size_t n) { //分配至少n个连续的字节，这里实现得不精细，占用的只能是整数
    ↪个页。
    void *ptr = NULL;
    struct Page *base = NULL;
    assert(n > 0 && n < 1024 * 0124);
    int num_pages = (n + PGSIZE - 1) / PGSIZE; //向上取整到整数个页
    base = alloc_pages(num_pages);
    assert(base != NULL); //如果分配失败就直接panic
    ptr = page2kva(base); //分配的内存的起始位置（虚拟地址），
    //page2kva，就是page_to_kernel_virtual_address
    return ptr;
}

void kfree(void *ptr, size_t n) { //从某个位置开始释放n个字节
    assert(n > 0 && n < 1024 * 0124);
    assert(ptr != NULL);
    struct Page *base = NULL;
    int num_pages = (n + PGSIZE - 1) / PGSIZE;
    /*计算num_pages和kmalloc里一样，
    但是如果程序员写错了呢？调用kfree的时候传入的n和调用kmalloc传入的n不一样？
    就像你平时在windows/linux写C语言一样，会出各种奇奇怪怪的bug。
    */
    base = kva2page(ptr); //kernel_virtual_address_to_page
    free_pages(base, num_pages);
}

// kern/mm/pmm.h
/*
KADDR, PADDR进行的是物理地址和虚拟地址的互换
由于我们在ucore里实现的页表映射很简单，所有物理地址和虚拟地址的偏移值相同，
所以这两个宏本质上只是做了一步加法/减法，额外还做了一些合法性检查。
*/
/* *
* PADDR - takes a kernel virtual address (an address that points above
* KERNBASE),

```

(续下页)

(接上页)

```

* where the machine's maximum 256MB of physical memory is mapped and returns
* the
* corresponding physical address. It panics if you pass it a non-kernel
* virtual address.
* */
#define PADDR(kva) \
({ \
    uintptr_t __m_kva = (uintptr_t)(kva); \
    if (__m_kva < KERNBASE) { \
        panic("PADDR called with invalid kva %08lx", __m_kva); \
    } \
    __m_kva - va_pa_offset; \
})

/* *
* KADDR - takes a physical address and returns the corresponding kernel virtual
* address. It panics if you pass an invalid physical address.
* */
#define KADDR(pa) \
({ \
    uintptr_t __m_pa = (pa); \
    size_t __m_ppn = PPN(__m_pa); \
    if (__m_ppn >= npage) { \
        panic("KADDR called with invalid pa %08lx", __m_pa); \
    } \
    (void *) (__m_pa + va_pa_offset); \
})

extern struct Page *pages;
extern size_t npage;
extern const size_t nbase;
extern uint_t va_pa_offset;

/*
我们曾经在内存里分配了一堆连续的Page结构体, 来管理物理页面。可以把它们看作一个结构体数
→组。
pages指针是这个数组的起始地址, 减一下, 加上一个基准值nbase, 就可以得到正确的物理页号。
pages指针和nbase基准值我们都在其他地方做了正确的初始化
*/
static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }

/*
指向某个Page结构体的指针, 对应一个物理页面, 也对应一个起始的物理地址。
左移若干位就可以从物理页号得到页面的起始物理地址。
*/
static inline uintptr_t page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}

/*
倒过来, 从物理页面的地址得到所在的物理页面。实际上是得到管理这个物理页面的Page结构体。
*/
static inline struct Page *pa2page(uintptr_t pa) {
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa");
    }
    return &pages[PPN(pa) - nbase]; //把pages指针当作数组使用
}

```

(续下页)

(接上页)

```
static inline void *page2kva(struct Page *page) { return KADDR(page2pa(page)); }

static inline struct Page *kva2page(void *kva) { return pa2page(PADDR(kva)); }

//从页表项得到对应的页, 这里用到了 PTE_ADDR(pte)宏, 对页表项做操作, 在mmu.h里
//定义
static inline struct Page *pte2page(pte_t pte) {
    if (!(pte & PTE_V)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte));
}

//PDE(Page Directory Entry)指的是不在叶节点的页表项 (指向低一级页表的页表项)
static inline struct Page *pde2page(pde_t pde) { //PDE_ADDR这个宏和PTE_ADDR是一样的
    return pa2page(PDE_ADDR(pde));
}
```

接下来需要我们处理的是多级页表。之前的初始页表占据一个页的物理内存，只有一个页表项是有用的，映射了一个大大页 (Giga Page)。我们将在接下来的内容中实现页面置换机制。

6.1.4 页面置换机制的实现

6.1.4.1 本节内容

6.1.4.1.1 页替换算法设计思路

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的数据或代码时，如果这些数据或代码不在内存中，则根据上一小节介绍，会产生页访问异常。这时，操作系统必须能够应对这种页访问异常，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替换算法着重考虑的问题。容易理解，一个好的页替换算法会导致页访问异常次数少，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。

本次实验涉及的页替换算法（包括扩展练习）：

- 先进先出 (First In First Out, FIFO) 页替换算法：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)，即在增加放置页的物理页帧的情况下，反而使页访问异常次数增多。
- 最久未使用 (least recently used, LRU) 算法：利用局部性，通过过去的访问情况预测未来的访问情况，我们可以认为最近还被访问过的页面将来被访问的可能性大，而很久没访问过的页面将来不太可能被访问。于是我们比较当前内存里的页面最近一次被访问的时间，把上一次访问时间离现在最久的页面置换出去。

- 时钟 (Clock) 页替换算法: 是 LRU 算法的一种近似实现。时钟页替换算法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 (简称当前指针) 指向最老的那个页面, 即最先进来的那个页面。另外, 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”。当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, 如果该页被写过, 则还要把它换出到硬盘上; 如果访问位为 “1”, 则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, 需要硬件支持来设置访问位。时钟页替换算法在本质上与 FIFO 算法是类似的, 不同之处是在时钟页替换算法中跳过了访问位为 1 的页。
- 改进的时钟 (Enhanced Clock) 页替换算法: 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, 所以优先淘汰没有修改的页, 减少磁盘操作次数。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。

6.1.4.1.2 页面置换机制设计思路

现在来看看 ucore 页面置换机制的实现。

页面置换机制中, 我们需要维护一些 “不在内存当中但是也许会用到” 的页, 它们存储在磁盘的交换区里, 也有对应的虚拟地址, 但是因为它们不在内存里, 在页表里并没有对它们的虚拟地址进行映射。但是在发生 Page Fault 之后, 会把访问到的页放到内存里, 这时也许会把其他页扔出去, 来给要用的页腾出地方。页面置换算法的核心任务, 主要就是确定 “把谁扔出去”。

页表里的信息大家都知道内容方面是比较有限的, 基本上可以理解为 “当前哪些数据在内存条里以及它们物理地址和虚拟地址的对应关系”, 这里我们显然需要一些页表之外的数据结构来维护当前页表里没映射的页。也就是要存储以下这些信息:

- 有哪些虚拟地址对应的页当前在磁盘上, 分别在磁盘上的哪个位置?
- 有哪些虚拟地址对应的页面当前放在内存里?

这两类页面 (位于内存/磁盘) 因为会相互转换 (换入/换出内存), 所以我们将这两类页面一起维护, 也就是维护 “所有可用的虚拟地址/虚拟页的集合” (不论当前这个虚拟地址对应的页在内存上还是在硬盘上)。之后我们将要实现进程机制, 对于不同的进程, 可用的虚拟地址 (虚拟页) 的集合常常是不一样的, 因此每个进程需要一个页表, 也需要一个数据结构来维护 “所有可用的虚拟地址”。

因此, 我们在 vmm.h 定义两个结构体 (vmm: virtual memory management)。

- vma_struct 结构体描述一段连续的虚拟地址, 从 vm_start 到 vm_end。通过包含一个 list_entry_t 成员, 我们可以把同一个页表对应的多个 vma_struct 结构体串成一个链表, 在链表里把它们按照区间的起始点进行排序。
- vm_flags 表示的是一段虚拟地址对应的权限 (可读, 可写, 可执行等), 这个权限在页表项里也要进行对应的设置。

我们注意到, 每个页表 (每个虚拟地址空间) 可能包含多个 vma_struct, 也就是多个访问权限可能不同的, 不相交连续地址区间。我们用 mm_struct 结构体把一个页表对应的信息组合起来, 包括 vma_struct 链表的首指针, 对应的页表在内存里的指针, vma_struct 链表的元素个数。

(参考 libs/list.h)

```

// kern/mm/vmm.h
//pre define
struct mm_struct;

// the virtual continuous memory area(vma), [vm_start, vm_end),
// addr belong to a vma means vma.vm_start<= addr <vma.vm_end
struct vma_struct {
    struct mm_struct *vm_mm; // the set of vma using the same PDT
    uintptr_t vm_start;      // start addr of vma
    uintptr_t vm_end;        // end addr of vma, not include the vm_end itself
    uint_t vm_flags;         // flags of vma
    list_entry_t list_link;  // linear list link which sorted by start addr of vma
};

#define le2vma(le, member) \
    to_struct((le), struct vma_struct, member)

#define VM_READ          0x00000001
#define VM_WRITE         0x00000002
#define VM_EXEC          0x00000004

// the control struct for a set of vma using the same Page Table
struct mm_struct {
    list_entry_t mmap_list;      // linear list link which sorted by start addr of v
    ↪ma struct vma_struct *mmap_cache; // current accessed vma, used for speed purpose
    pde_t *pgdir;               // the Page Table of these vma
    int map_count;              // the count of these vma
    void *sm_priv;              // the private data for swap manager
};

```

除了以上内容, 我们还需要为 vma_struct 和 mm_struct 定义和实现一些接口: 包括它们的构造函数, 以及如何把新的 vma_struct 插入到 mm_struct 对应的链表里。注意这两个结构体占用的内存空间需要用 kmalloc() 函数动态分配。

```

// kern/mm/vmm.c
// mm_create - alloc a mm_struct & initialize it.
struct mm_struct *
mm_create(void) {
    struct mm_struct *mm = kmalloc(sizeof(struct mm_struct));

    if (mm != NULL) {
        list_init(&(mm->mmap_list));
        mm->mmap_cache = NULL;
        mm->pgdir = NULL;
        mm->map_count = 0;

        if (swap_init_ok) swap_init_mm(mm); //我们接下来解释页面置换的初始化
        else mm->sm_priv = NULL;
    }
    return mm;
}

// vma_create - alloc a vma_struct & initialize it. (addr range: vm_start~vm_end)
struct vma_struct *
vma_create(uintptr_t vm_start, uintptr_t vm_end, uint_t vm_flags) {
    struct vma_struct *vma = kmalloc(sizeof(struct vma_struct));
}

```

(续下页)

(接上页)

```

if (vma != NULL) {
    vma->vm_start = vm_start;
    vma->vm_end = vm_end;
    vma->vm_flags = vm_flags;
}
return vma;
}

```

在插入一个新的 vma_struct 之前, 我们要保证它和原有的区间都不重合。

```

// kern/mm/vmm.c
// check_vma_overlap - check if vma1 overlaps vma2 ?
static inline void
check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
    assert(prev->vm_start < prev->vm_end);
    assert(prev->vm_end <= next->vm_start);
    assert(next->vm_start < next->vm_end); // next 是我们想插入的区间, 这里顺便检验了 st
    art < end
}

```

我们可以插入一个新的 vma_struct, 也可以查找某个虚拟地址对应的 vma_struct 是否存在。

```

// kern/mm/vmm.c
// insert_vma_struct -insert vma in mm's list link
void
insert_vma_struct(struct mm_struct *mm, struct vma_struct *vma) {
    assert(vma->vm_start < vma->vm_end);
    list_entry_t *list = &(mm->mmap_list);
    list_entry_t *le_prev = list, *le_next;

    list_entry_t *le = list;
    while ((le = list_next(le)) != list) {
        struct vma_struct *mmap_prev = le2vma(le, list_link);
        if (mmap_prev->vm_start > vma->vm_start) {
            break;
        }
        le_prev = le;
    }
    //保证插入后所有 vma_struct 按照区间左端点有序排列
    le_next = list_next(le_prev);

    /* check overlap */
    if (le_prev != list) {
        check_vma_overlap(le2vma(le_prev, list_link), vma);
    }
    if (le_next != list) {
        check_vma_overlap(vma, le2vma(le_next, list_link));
    }

    vma->vm_mm = mm;
    list_add_after(le_prev, &(vma->list_link));

    mm->map_count ++; //计数器
}

// find_vma - find a vma (vma->vm_start <= addr <= vma->vm_end)

```

(续下页)

(接上页)

```

//如果返回NULL, 说明查询的虚拟地址不存在/不合法, 既不对应内存里的某个页, 也不对应硬盘
→里某个可以换进来的页
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;
    if (mm != NULL) {
        vma = mm->mmap_cache;
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
            bool found = 0;
            list_entry_t *list = &(mm->mmap_list), *le = list;
            while ((le = list_next(le)) != list) {
                vma = le2vma(le, list_link);
                if (vma->vm_start <= addr && addr < vma->vm_end) {
                    found = 1;
                    break;
                }
            }
            if (!found) {
                vma = NULL;
            }
        }
        if (vma != NULL) {
            mm->mmap_cache = vma;
        }
    }
    return vma;
}

```

如果此时发生 Page Fault 怎么办? 我们可以回顾之前的异常处理部分的知识。我们的 `trapFrame` 传递了 `badvaddr` 给 `do_pgfault()` 函数, 而这实际上是 `stval` 这个寄存器的数值 (在旧版的 RISC-V 标准里叫做 `sbadvaddr`), 这个寄存器存储一些关于异常的数据, 对于 PageFault 它存储的是访问出错的虚拟地址。

```

// kern/trap/trap.c
static int pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct; //当前使用的mm_struct的指针, 在vmm.c定义
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
    }
    panic("unhandled page fault.\n");
}

// kern/mm/vmm.c
struct mm_struct *check_mm_struct;

// check_pgfault - check correctness of pgfault handler
static void
check_pgfault(void) {
    /* ..... */
    check_mm_struct = mm_create();
    /* ..... */
}

```

`do_pgfault()` 函数在 `vmm.c` 定义, 是页面置换机制的核心。如果过程可行, 没有错误值返回, 我们就可对页表做对应的修改, 通过加入对应的页表项, 并把硬盘上的数据换进内存, 这时还可能涉及到要把内存里的一个页换出去, 而 `do_pgfault()` 函数就实现了这些功能。如果你对这部分内容相当了解的话, 那可以说你对于页面置换机制的掌握程度已经很棒了。

```

// kern/mm/vmm.c
int do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    //addr: 访问出错的虚拟地址
    int ret = -E_INVALID;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);
    //我们首先要做的就是判断在mm_struct里判断这个虚拟地址是否可用
    pgfault_num++;
    //If the addr is not in the range of a mm's vma?
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and can not find it in vma\n", addr);
        goto failed;
    }

    /* IF (write an existed addr ) OR
     * (write an non_existed addr && addr is writable) OR
     * (read an non_existed addr && addr is readable)
     * THEN
     * continue process
     */
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }
    addr = ROUNDDOWN(addr, PGSIZE); //按照页面大小把地址对齐

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;

    ptep = get_pte(mm->pgdir, addr, 1); // (1) try to find a pte, if pte's
    //PT(Page Table) isn't existed, then
    //create a PT.

    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    else {
        /*
         * Now we think this pte is a swap entry, we should load data from disk
         * to a page with phy addr,
         * and map the phy addr with logical addr, trigger swap manager to record
         * the access situation of this page.
         *
         * swap_in(mm, addr, &page) : alloc a memory page, then according to
         * the swap entry in PTE for addr, find the addr of disk page, read the
         * content of disk page into this memroy page
         *
         * page_insert : build the map of phy addr of an Page with the virtual
        ↪ addr la
         * swap_map_swappable : set the page swappable
         */
        if (swap_init_ok) {
            struct Page *page = NULL;
            //在swap_in()函数执行完之后, page保存换入的物理页面。
            //swap_in()函数里面可能把内存里原有的页面换出去
            swap_in(mm, addr, &page); // (1) According to the mm AND addr, try

```

(续下页)

(接上页)

```

//to load the content of right disk page
//into the memory which page managed.
page_insert(mm->pgdir, page, addr, perm); //更新页表, 插入新的页表项
// (2) According to the mm, addr AND page,
// setup the map of phy addr <---> virtual addr
swap_map_swappable(mm, addr, page, 1); // (3) make the page swappable.
//标记这个页面将来是可以再换出的
page->pra_vaddr = addr;
} else {
    cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
    goto failed;
}
}

ret = 0;
failed:
    return ret;
}

```

接下来我们看看 FIFO 页面置换算法是怎么在 ucore 里实现的。

6.1.4.1.3 FIFO 页面置换算法

所谓 FIFO(First in, First out) 页面置换算法, 相当简单, 就是把所有页面排在一个队列里, 每次换入页面的时候, 把队列里最靠前 (最早被换入) 的页面置换出去。

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有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。

目前的框架支持第二种情况, 在 alloc_pages() 里面, 如果此时试图得到空闲页且没有空闲的物理页时, 我们才尝试换出页面到硬盘上。

```

// kern/mm/pmm.c
// alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE memory
struct Page *alloc_pages(size_t n) {
    struct Page *page = NULL;
    bool intr_flag;

    while (1) {
        local_intr_save(intr_flag);
        { page = pmm_manager->alloc_pages(n); }
        local_intr_restore(intr_flag);
        //如果有足够的物理页面, 就不必换出其他页面
        //如果 n>1, 说明希望分配多个连续的页面, 但是我们换出页面的时候并不能换出连续的
        //页面
        //swap_init_ok标志是否成功初始化了
        if (page != NULL || n > 1 || swap_init_ok == 0) break;

        extern struct mm_struct *check_mm_struct;
        swap_out(check_mm_struct, n, 0); //调用页面置换的”换出页面“接口。这里必有 n=1
    }
}

```

(续下页)

(接上页)

```

    }
    return page;
}

```

类似 pmm_manager, 我们定义 swap_manager, 组合页面置换需要的一些函数接口。

```

// kern/mm/swap.h

struct swap_manager
{
    const char *name;
    /* Global initialization for the swap manager */
    int (*init) (void);
    /* Initialize the priv data inside mm_struct */
    int (*init_mm) (struct mm_struct *mm);
    /* Called when tick interrupt occurred */
    int (*tick_event) (struct mm_struct *mm);
    /* Called when map a swappable page into the mm_struct */
    int (*map_swappable) (struct mm_struct *mm, uintptr_t addr, struct Page *page,
↪ int swap_in);
    /* When a page is marked as shared, this routine is called to
     * delete the addr entry from the swap manager */
    int (*set_unswappable) (struct mm_struct *mm, uintptr_t addr);
    /* Try to swap out a page, return then victim */
    int (*swap_out_victim) (struct mm_struct *mm, struct Page **ptr_page, int in_tick
↪ );
    /* check the page replacement algorithm */
    int (*check_swap) (void);
};

```

我们来看 swap_in(), swap_out() 如何换入/换出一个页面. 注意我们对物理页面的 Page 结构体做了一些改动。

```

// kern/mm/memlayout.h
struct Page {
    int ref; // page frame's reference counter
    uint_t flags; // array of flags that describe the status of the pa
↪ ge frame
    unsigned int property; // the num of free block, used in first fit pm man
↪ ager
    list_entry_t page_link; // free list link
    list_entry_t pra_page_link; // used for pra (page replace algorithm)
    uintptr_t pra_vaddr; // used for pra (page replace algorithm)
};

// kern/mm/swap.c
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
    struct Page *result = alloc_page(); // 这里 alloc_page() 内部可能调用 swap_out()
    // 找到对应的一个物理页面
    assert(result != NULL);

    pte_t *ptep = get_pte(mm->pgdir, addr, 0); // 找到/构建对应的页表项
    // 将物理地址映射到虚拟地址是在 swap_in() 退出之后, 调用 page_insert() 完成的
    int r;
    if ((r = swapfs_read((ptep), result)) != 0) // 将数据从硬盘读到内存

```

(续下页)

(接上页)

```

{
    assert(r!=0);
}
cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n", (*ptep)
->>8, addr);
*ptr_result=result;
return 0;
}
int swap_out(struct mm_struct *mm, int n, int in_tick)
{
    int i;
    for (i = 0; i != n; ++ i)
    {
        uintptr_t v;
        struct Page *page;
        int r = sm->swap_out_victim(mm, &page, in_tick); //调用页面置换算法的接口
        //r=0表示成功找到了可以换出去的页面
        //要换出去的物理页面存在page里
        if (r != 0) {
            cprintf("i %d, swap_out: call swap_out_victim failed\n", i);
            break;
        }

        cprintf("SWAP: choose victim page 0x%08x\n", page);

        v=page->pra_vaddr; //可以获取物理页面对应的虚拟地址
        pte_t *ptep = get_pte(mm->pgdir, v, 0);
        assert((*ptep & PTE_V) != 0);

        if (swapfs_write( (page->pra_vaddr/PGSIZE+1)<<8, page) != 0) {
            //尝试把要换出的物理页面写到硬盘上的交换区, 返回值不为0
            cprintf("SWAP: failed to save\n");
            sm->map_swappable(mm, v, page, 0);
            continue;
        }
        else {
            //成功换出
            cprintf("swap_out: i %d, store page in vaddr 0x%x to disk swap ent
            ry %d\n", i, v, page->pra_vaddr/PGSIZE+1);
            *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
            free_page(page);
        }

        //由于页表改变了, 需要刷新TLB
        //思考: swap_in()的时候插入新的页表项之后在哪里刷新了TLB?
        tlb_invalidate(mm->pgdir, v);
    }
    return i;
}

```

kern/mm/swap.c 里其他的接口基本都是简单调用 swap_manager 的具体实现。值得一提的是 swap_init() 初始化里做的工作。

```

// kern/mm/swap.c
static struct swap_manager *sm;
int swap_init(void)

```

(续下页)

(接上页)

```

{
    swapfs_init();

    // Since the IDE is faked, it can only store 7 pages at most to pass the test
    if (!(7 <= max_swap_offset &&
        max_swap_offset < MAX_SWAP_OFFSET_LIMIT)) {
        panic("bad max_swap_offset %08x.\n", max_swap_offset);
    }

    sm = &swap_manager_fifo; // use first in first out Page Replacement Algorithm
    int r = sm->init();

    if (r == 0)
    {
        swap_init_ok = 1;
        cprintf("SWAP: manager = %s\n", sm->name);
        check_swap();
    }

    return r;
}

int swap_init_mm(struct mm_struct *mm)
{
    return sm->init_mm(mm);
}

int swap_tick_event(struct mm_struct *mm)
{
    return sm->tick_event(mm);
}

int swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
{
    return sm->map_swappable(mm, addr, page, swap_in);
}

int swap_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return sm->set_unswappable(mm, addr);
}

```

kern/mm/swap_fifo.h 完成了 FIFO 置换算法最终的具体实现。我们所做的就是维护了一个队列（用链表实现）。

```

// kern/mm/swap_fifo.h
#ifndef __KERN_MM_SWAP_FIFO_H__
#define __KERN_MM_SWAP_FIFO_H__

#include <swap.h>
extern struct swap_manager swap_manager_fifo;

#endif
// kern/mm/swap_fifo.c
/* Details of FIFO PRA

```

(续下页)

(接上页)

```

* (1) Prepare: In order to implement FIFO PRA, we should manage all swappable pages,
↳so we can
* link these pages into pra_list_head according the time order. At first you should
* be familiar to the struct list in list.h. struct list is a simple doubly linked lis
↳t
* implementation. You should know howto USE: list_init, list_add(list_add_after),
* list_add_before, list_del, list_next, list_prev. Another tricky method is to transf
↳orm
* a general list struct to a special struct (such as struct page). You can find some
↳ MACRO:
* le2page (in memlayout.h), (in future labs: le2vma (in vmm.h), le2proc (in proc.h),e
↳tc.
*/

list_entry_t pra_list_head;
/*
* (2) _fifo_init_mm: init pra_list_head and let mm->sm_priv point to the addr of pra
↳_list_head.
* Now, From the memory control struct mm_struct, we can access FIFO PRA
*/
static int
_fifo_init_mm(struct mm_struct *mm)
{
    list_init(&pra_list_head);
    mm->sm_priv = &pra_list_head;
    return 0;
}
/*
* (3)_fifo_map_swappable: According FIFO PRA, we should link the most recent arrival
↳page at the back of pra_list_head queue
*/
static int
_fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_
↳in)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && head != NULL);
    //record the page access situlation
    //(1)link the most recent arrival page at the back of the pra_list_head queue.
    list_add(head, entry);
    return 0;
}
/*
* (4)_fifo_swap_out_victim: According FIFO PRA, we should unlink the earliest arri
↳val page in front of pra_list_head queue,
* then set the addr of addr of this page to ptr_page.
*/
static int
_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    assert(head != NULL);
    assert(in_tick==0);
    /* Select the victim */
    //(1) unlink the earliest arrival page in front of pra_list_head queue

```

(续下页)

(接上页)

```

    // (2) set the addr of this page to ptr_page
    list_entry_t* entry = list_prev(head);
    if (entry != head) {
        list_del(entry);
        *ptr_page = le2page(entry, pra_page_link);
    } else {
        *ptr_page = NULL;
    }
    return 0;
}

static int _fifo_init(void) // 初始化的时候什么都不做
{
    return 0;
}

static int _fifo_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return 0;
}

static int _fifo_tick_event(struct mm_struct *mm) // 时钟中断的时候什么都不做
{
    return 0;
}

struct swap_manager swap_manager_fifo =
{
    .name           = "fifo swap manager",
    .init           = &_fifo_init,
    .init_mm        = &_fifo_init_mm,
    .tick_event     = &_fifo_tick_event,
    .map_swappable  = &_fifo_map_swappable,
    .set_unswappable = &_fifo_set_unswappable,
    .swap_out_victim = &_fifo_swap_out_victim,
    .check_swap     = &_fifo_check_swap,
};

```

我们通过 `_fifo_check_swap()`, `check_swap()`, `check_vma_struct()`, `check_pgfault()` 等接口对页面置换机制进行了简单的测试。具体测试的细节在这里不进行展示, 同学们可以尝试自己进行测试。至此, 实验三中的主要工作描述完毕, 接下来请同学们完成本次实验练习。

6.1.5 实验报告要求

从 git server 网站上取得 `ucore_lab` 后, 进入目录 `labcodes/lab3`, 完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 `labcodes/lab3` 下存放实验报告, 实验报告文档命名为 `lab3-学生 ID.md`。推荐用 **markdown** 格式。对于 `lab3` 中编程任务, 完成编写之后, 再通过 `git push` 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

实验 2/3 完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过 ucore OS 的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用 CPU 来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的 CPU。.. _ 本章内容:

7.1 本章内容

7.1.1 实验目的

- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程

7.1.2 实验内容

实验 2/3 完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过 ucore OS 的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用 CPU 来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的 CPU。

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 内核线程只运行在内核态
- 用户进程会在在用户态和内核态交替运行
- 所有内核线程共用 ucore 内核内存空间，不需为每个内核线程维护单独的内存空间
- 而用户进程需要维护各自的用户内存空间

相关原理介绍可看附录 B：【原理】进程/线程的属性与特征解析。.. _ 提前说明:

7.1.2.1 提前说明

需要注意的是, 在 ucore 的调度和执行管理中, **对线程和进程做了统一的处理**。且由于 ucore 内核中的所有内核线程共享一个内核地址空间和其他资源, 所以这些内核线程从属于同一个唯一的内核进程, 即 ucore 内核本身。

7.1.2.2 本节内容

7.1.2.2.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 ucore_lab 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

7.1.2.2.1.1 练习 0: 填写已有实验

本实验依赖实验 1/2/3。请把你做的实验 1/2/3 的代码填入本实验中代码中有 “LAB1”, “LAB2”, “LAB3” 的注释相应部分。

7.1.2.2.1.2 练习 1: 分配并初始化一个进程控制块 (需要编码)

alloc_proc 函数 (位于 kern/process/proc.c 中) 负责分配并返回一个新的 struct proc_struct 结构, 用于存储新建的内核线程的管理信息。ucore 需要对这个结构进行最基本的初始化, 你需要完成这个初始化过程。

【提示】在 alloc_proc 函数的实现中, 需要初始化的 proc_struct 结构中的成员变量至少包括: state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。

请在实验报告中简要说明你的设计实现过程。请回答如下问题:

- 请说明 proc_struct 中 struct context context 和 struct trapframe *tf 成员变量含义和在本实验中的作用是啥? (提示通过看代码和编程调试可以判断出来)

7.1.2.2.1.3 练习 2: 为新创建的内核线程分配资源 (需要编码)

创建一个内核线程需要分配和设置好很多资源。kernel_thread 函数通过调用 do_fork 函数完成具体内核线程的创建工作。do_kernel 函数会调用 alloc_proc 函数来分配并初始化一个进程控制块, 但 alloc_proc 只是找到了一小块内存用以记录进程的必要信息, 并没有实际分配这些资源。ucore 一般通过 do_fork 实际创建新的内核线程。do_fork 的作用是, 创建当前内核线程的一个副本, 它们的执行上下文、代码、数据都一样, 但是存储位置不同。因此, 我们实际需要” fork” 的东西就是 **stack 和 trapframe**。在这个过程中, 需要给新内核线程分配资源, 并且复制原进程的状态。你需要完成在 kern/process/proc.c 中的 do_fork 函数中的处理过程。它的大致执行步骤包括:

- 调用 alloc_proc, 首先获得一块用户信息块。
- 为进程分配一个内核栈。

- 复制原进程的内存管理信息到新进程（但内核线程不必做此事）
- 复制原进程上下文到新进程
- 将新进程添加到进程列表
- 唤醒新进程
- 返回新进程号

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 请说明 `ucore` 是否做到给每个新 `fork` 的线程一个唯一的 `id`？请说明你的分析和理由。

7.1.2.2.1.4 练习 3：编写 `proc_run` 函数（需要编码）

`proc_run` 用于将指定的进程切换到 CPU 上运行。它的大致执行步骤包括：

- 检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换。
- 禁用中断。你可以使用 `/kern/sync/sync.h` 中定义好的宏 `local_intr_save(x)` 和 `local_intr_restore(x)` 来实现关、开中断。
- 切换当前进程为要运行的进程。
- 切换页表，以便使用新进程的地址空间。`/libs/riscv.h` 中提供了 `lcr3(unsigned int cr3)` 函数，可实现修改 `CR3` 寄存器值的功能。
- 实现上下文切换。`/kern/process` 中已经预先编写好了 `switch.S`，其中定义了 `switch_to()` 函数。可实现两个进程的 `context` 切换。
- 允许中断。

请回答如下问题：

- 在本实验的执行过程中，创建且运行了几个内核线程？

完成代码编写后，编译并运行代码：`make qemu`

如果可以得到如附录 A 所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

7.1.2.2.1.5 扩展练习 Challenge：

- 说明语句 `local_intr_save(intr_flag); ... local_intr_restore(intr_flag);` 是如何实现开关中断的？

7.1.2.2.2 项目组成

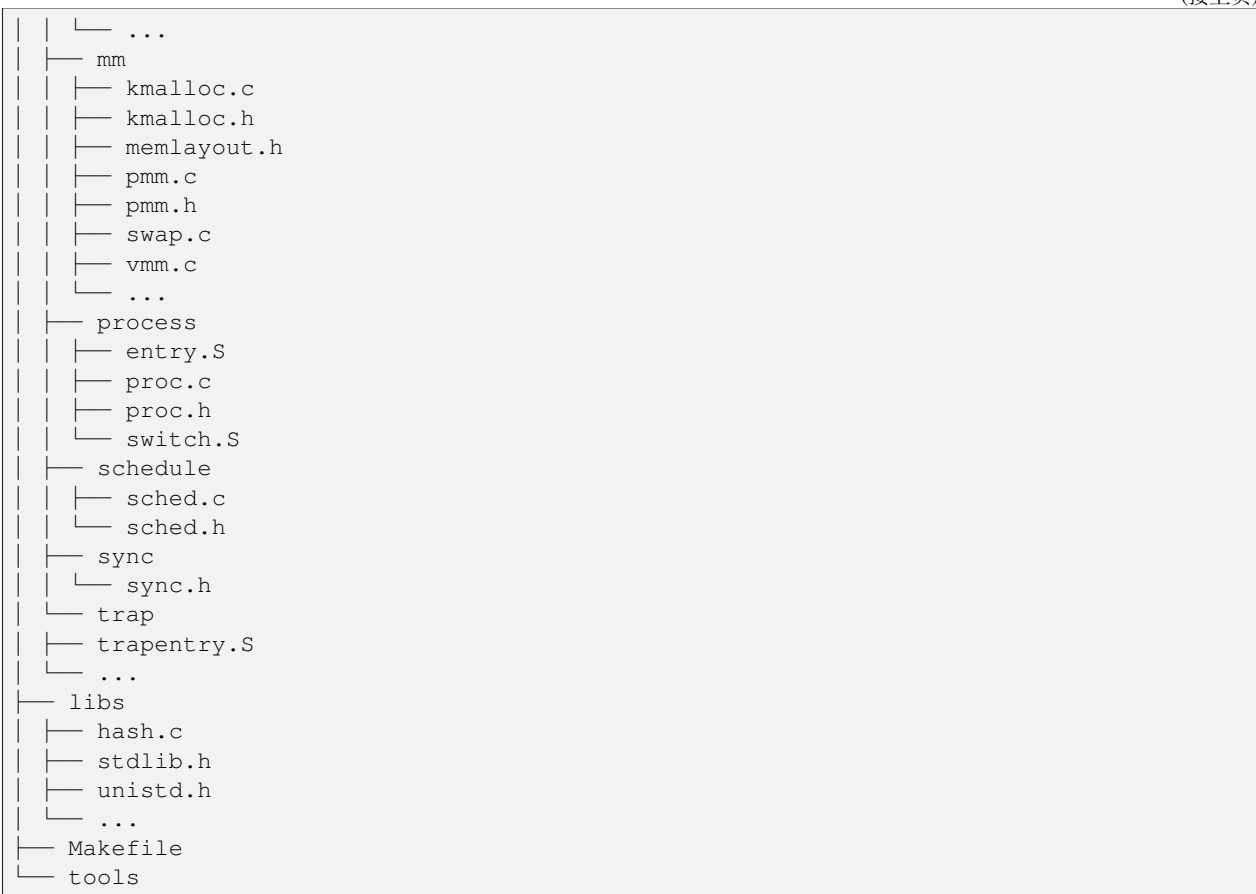
```

├─ boot
├─ kern
│   ├── debug
│   ├── driver
│   ├── fs
│   ├── init
│   │   ├── init.c
│   │   └── ...
│   └── libs
│       ├── rb\_tree.c
│       └── rb\_tree.h

```

(续下页)

(接上页)



相对与实验三，实验四中主要改动如下：

- kern/process/ （新增进程管理相关文件）
- proc.[ch]：新增：实现进程、线程相关功能，包括：创建进程/线程，初始化进程/线程，处理进程/线程退出等功能
- entry.S：新增：内核线程入口函数 `kernel_thread_entry` 的实现
- switch.S：新增：上下文切换，利用堆栈保存、恢复进程上下文
- kern/init/
- init.c：修改：完成进程系统初始化，并在内核初始化后切入 `idle` 进程
- kern/mm/ （基本上与本次实验没有太直接的联系，了解 `kmalloc` 和 `kfree` 如何使用即可）
- kmalloc.[ch]：新增：定义和实现了新的 `kmalloc/kfree` 函数。具体实现是基于 `slab` 分配的简化算法（只要求会调用这两个函数即可）
- memlayout.h：增加 `slab` 物理内存分配相关的定义与宏（可不用理会）。
- pmm.[ch]：修改：在 `pmm.c` 中添加了调用 `kmalloc_init` 函数, 取消了老的 `kmalloc/kfree` 的实现；在 `pmm.h` 中取消了老的 `kmalloc/kfree` 的定义
- swap.c：修改：取消了用于 `check` 的 Line 185 的执行
- vmm.c：修改：调用新的 `kmalloc/kfree`
- kern/trap/

- trapentry.S: 增加了汇编写的函数 forkrets, 用于 do_fork 调用的返回处理。
- kern/schedule/
- sched.[ch]: 新增: 实现 FIFO 策略的进程调度
- kern/libs
- rb_tree.[ch]: 新增: 实现红黑树, 被 slab 分配的简化算法使用 (可不用理会)

编译执行

编译并运行代码的命令如下:

```
make
make qemu
```

则可以得到如下的显示内容 (仅供参考, 不是标准答案输出)

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

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc010a708 (phys)
  edata  0xc0127ae0 (phys)
  end    0xc012ad58 (phys)

...

++ 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:354:
  process exit!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu: terminating on signal 2
```

7.1.3 内核线程管理

在真正进入本章的讨论前, 我们先来谈一谈何为进程, 为什么需要进程。

7.1.3.1 进程与线程

在操作系统中, 我们经常谈到的两个概念就是进程与线程的概念。这两个概念虽然有许多相似的地方, 但也有很多的不同。

我们平时编写的源代码, 经过编译器编译就变成了可执行文件, 我们管这一类文件叫做 **程序**。而当一个程序被用户或操作系统启动, 分配资源, 装载进内存开始执行后, 它就成为了一个 **进程**。进程与程序之间最大的不同在于进程是一个“正在运行”的实体, 而程序只是一个不动的文件。进程包含程序的内容, 也就是它的静态的代码部分, 也包括一些在运行时在可以体现出来的信息, 比如堆栈, 寄存器等数据, 这些组成了进程“正在运行”的特性。

如果我们只关注于那些“正在运行”的部分, 我们就从进程当中剥离出来了 **线程**。一个进程可以对应一个线程, 也可以对应很多线程。这些线程之间往往具有相同的代码, 共享一块内存, 但是却有不同的 CPU

执行状态。相比于线程，进程更多的作为一个资源管理的实体（因为操作系统分配网络等资源时往往是基于进程的），这样线程就作为可以被调度的最小单元，给了调度器更多的调度可能。

7.1.3.2 我们为什么需要进程

进程的一个重要特点在于其可以调度。在我们操作系统启动的时候，操作系统相当是一个初始的进程。之后，操作系统会创建不同的进程负责不同的任务。用户可以通过命令行启动进程，从而使用计算机。想想如果没有进程会怎么样？所有的代码可能需要在操作系统编译的时候就打包在一块，安装软件将变成一件非常难的事情，这显然对于用户使用计算机是不利的。

另一方面，从 2000 年开始，CPU 越来越多的使用多核心的设计。这主要是因为芯片设计师们发现一个核心上提高高频变得越来越难（这其中有许多原因，相信组成原理课上已经有过介绍），所以采用多个核心，将利用多核性能的任务交给了程序员。在这种环境下，操作系统也需要进行相应的调整，以适应这种多核的趋势。使用进程的概念有助于各个进程同时的利用 CPU 的各个核心，这是单进程系统往往做不到的。

但是，多进程的引入其实远早于多核心处理器。在计算机的远古时代，存在许多“巨无霸”计算机。但是，如果只让这些计算机服务于一个用户，有时候又有点浪费。有没有可能让一个计算机服务于多个用户呢（哪怕只有一个核心）？分时操作系统解决了这个问题，就是通过时间片轮转的方法使得多个用户可以“同时”使用计算资源。这个时候，引入进程的概念，成为操作系统调度的单元就显得十分必要了。

综合以上可以看出，操作系统的确离不开进程管理。在下一节，我们会介绍 `ucore` 中与进程相关的数据结构，看一看如果定义一个进程。

7.1.3.3 本节内容

7.1.3.3.1 实验执行流程概述

lab2 和 lab3 完成了对内存的虚拟化，但整个控制流还是一条线串行执行。lab4 将在此基础上进行 CPU 的虚拟化，即让 `ucore` 实现分时共享 CPU，实现多条控制流能够并发执行。从某种程度上，我们可以把控制流看作是一个内核线程。本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：内核线程只运行在内核态而用户进程会在用户态和内核态交替运行；所有内核线程直接使用共同的 `ucore` 内核内存空间，不需为每个内核线程维护单独的内存空间而用户进程需要维护各自的内存空间。从内存空间占用情况这个角度上看，我们可以把线程看作是一种共享内存空间的轻量级进程。

为了实现内核线程，需要设计管理线程的数据结构，即进程控制块（在这里也可叫做线程控制块）。如果要让内核线程运行，我们首先要创建内核线程对应的进程控制块，还需把这些进程控制块通过链表连在一起，便于随时进行插入，删除和查找操作等进程管理事务。这个链表就是进程控制块链表。然后在通过调度器（scheduler）来让不同的内核线程在不同的时间段占用 CPU 执行，实现对 CPU 的分时共享。那 lab4 中是如何一步一步实现这个过程的呢？

7.1.3.3.1.1 idle 进程创建

进程初始化的函数定义在文件 `kern/process/proc.c` 中的 `proc_init`。进程模块的初始化主要分为两步，首先创建第 0 个内核进程，`idle`。

我们还是从 lab4/kern/init/init.c 中的 `kern_init` 函数入手分析。在 `kern_init` 函数中，当完成虚拟内存的初始化工作后，就调用了 `proc_init` 函数，这个函数完成了 `idleproc` 内核线程和 `initproc` 内核线程的创建或复制工作，这也是本次实验要完成的练习。`idleproc` 内核线程的工作就是不停地查询，看是否有其他内核线程可以执行了，如果有，马上让调度器选择那个内核线程执行（请参考 `cpu_idle` 函数的实现）。所以 `idleproc` 内核线程是在 `ucore` 操作系统没有其他内核线程可执行的情况下才会被调用。接着就是调用 `kernel_thread` 函数来创建 `initproc` 内核线程。`initproc` 内核线程的工作就是显示“Hello World”，表明自己存在且能正常工作了。

7.1.3.3.1.2 第一个内核线程的初始化

接下来我们对于第一个真正的内核进程进行初始化（因为 idle 进程仅仅算是“继承了”ucore 的运行）。我们的目标是使用新的内核进程进行一下内核初始化的工作，但在这章我们先仅仅让它输出一个 Hello World，证明我们的内核进程实现的没有问题。

调度器是一种通过一定的调度算法、在特定的调度点上执行调度，最终完成进程切换的机制。在 lab4 中，这个调度点就一处，即在 cpu_idle 函数中，此函数如果发现当前进程（也就是 idleproc）的 need_resched 置为 1（在初始化 idleproc 的进程控制块时就置为 1 了），则调用 schedule 函数，完成进程调度和进程切换。进程调度的过程其实比较简单，就是在进程控制块链表中查找到一个“合适”的内核线程，所谓“合适”就是指内核线程处于“PROC_RUNNABLE”状态。在接下来的 switch_to 函数（在后续有详细分析，有一定难度，需深入了解一下）完成具体的进程切换过程。一旦切换成功，那么 initproc 内核线程就可以通过显示字符串来表明本次实验成功。

7.1.3.3.2 PCB

接下来将主要介绍了进程创建所需的重要数据结构-进程控制块 proc_struct，以及 ucore 创建并执行内核线程 idleproc 和 initproc 的两种不同方式，特别是创建 initproc 的方式将被延续到实验五中，扩展为创建用户进程的主要方式。另外，还初步涉及了进程调度（实验六涉及并会扩展）和进程切换内容。

7.1.3.3.3 设计关键数据结构

7.1.3.3.3.1 进程控制块

在实验四中，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;     // bool value: need to be rescheduled to r
    ↪ 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;                  // CR3 register: the base addr of Page Dir
    ↪ destroy 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
};
```

下面重点解释一下几个比较重要的成员变量：

- mm：这里面保存了内存管理的信息，包括内存映射，虚存管理等内容。具体内在实现可以参考之前的章节。
- state：进程所处的状态。uCore 中进程状态有四种：分别是 PROC_UNINIT、PROC_SLEEPING、PROC_RUNNABLE、PROC_ZOMBIE。

- `parent`: 里面保存了进程的父进程的指针。在内核中, 只有内核创建的 `idle` 进程没有父进程, 其他进程都有父进程。进程的父子关系组成了一棵进程树, 这种父子关系有利于维护父进程对于子进程的一些特殊操作。
- `context`: `context` 中保存了进程执行的上下文, 也就是几个关键的寄存器的值。这些寄存器的值用于在进程切换中还原之前进程的运行状态 (进程切换的详细过程在后面会介绍)。切换过程的实现在 `kern/process/switch.S`。
- `tfs`: `tfs` 里保存了进程的中断帧。当进程从用户空间跳进内核空间的时候, 进程的执行状态被保存在了中断帧中 (注意这里需要保存的执行状态数量不同于上下文切换)。系统调用可能会改变用户寄存器的值, 我们可以通过调整中断帧来使得系统调用返回特定的值。
- `cr3`: `cr3` 寄存器是 x86 架构的特殊寄存器, 用来保存页表所在的基址。出于 `legacy` 的原因, 我们这里仍然保留了这个名字, 但其值仍然是页表基址所在的位置。
- `kstack`: 每个线程都有一个内核栈, 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` 中 `KSTACKSIZE` 的定义) 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` (可以回顾一下在实验一中讲述的 `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 且处于“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。
- `static struct proc *initproc`: 本实验中, 指向一个内核线程。本实验以后, 此指针将指向第一个用户态进程。
- `static list_entry_t hash_list[HASH_LIST_SIZE]`: 所有进程控制块的哈希表, `proc_struct` 中的成员变量 `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, `proc_struct` 中的成员变量 `list_link` 将链接入这个链表中。

7.1.3.3.2 进程上下文

进程上下文使用结构体 `struct context` 保存, 其中包含了 `ra`, `sp`, `s0~s11` 共 14 个寄存器。

可能感兴趣的同学就会问了, 为什么我们不需要保存所有的寄存器呢? 这里我们巧妙地利用了编译器对于函数的处理。我们知道寄存器可以分为调用者保存 (`caller-saved`) 寄存器和被调用者保存 (`callee-saved`) 寄存器。因为线程切换在一个函数当中 (我们下一小节就会看到), 所以编译器会自动帮助我们生成保存和恢复调用者保存寄存器的代码, 在实际的进程切换过程中我们只需要保存被调用者保存寄存器就好啦!

下一小节, 我们来看一看到底如何进行进程的切换。

7.1.3.3.4 创建并执行内核线程

建立进程控制块（proc.c 中的 alloc_proc 函数）后，现在就可以通过进程控制块来创建具体的进程/线程了。首先，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，没有自己的“专属”空间。这是由于在 uCore OS 启动后，已经对整个内核内存空间进行了管理，通过设置页表建立了内核虚拟空间（即 boot_cr3 指向的二级页表描述的空间）。所以 uCore OS 内核中的所有线程都不需要再建立各自的页表，只需共享这个内核虚拟空间就可以访问整个物理内存了。从这个角度看，内核线程被 uCore OS 内核这个大“内核进程”所管理。

7.1.3.3.5 创建第 0 个内核线程 idleproc

idleproc 是一个在操作系统中常见的概念，用于表示空闲进程。在操作系统中，空闲进程是一个特殊的进程，它的主要目的是在系统没有其他任务需要执行时，占用 CPU 时间，同时便于进程调度的统一化。

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 uCore 内核（也可看做是内核进程）中的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，将其打造成第 0 个内核线程 idleproc。具体步骤如下：

首先需要初始化进程链表。进程链表就是把所有进程控制块串联起来的数据结构，可以记录和追踪每一个进程。

随后，调用 alloc_proc 函数来通过 kmalloc 函数获得 proc_struct 结构的一块内存块，作为第 0 个进程控制块。并把 proc 进行初步初始化（即把 proc_struct 中的各个成员变量清零）。但有些成员变量设置了特殊的值，比如：

```
proc->state = PROC_UNINIT;  设置进程为“初始”态
proc->pid = -1;              设置进程pid的未初始化值
proc->cr3 = boot_cr3;        使用内核页目录表的基址
...
```

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；第三条语句表明由于该内核线程在内核中运行，故采用为 uCore 内核已经建立的页表，即设置为在 uCore 内核页表的起始地址 boot_cr3。后续实验中可进一步看出所有内核线程的内核虚拟地址空间（也包括物理地址空间）是相同的。既然内核线程共用一个映射内核空间的页表，这表示内核空间对所有内核线程都是“可见”的，所以更精确地说，这些内核线程都应该是从属于同一个唯一的“大内核进程”——uCore 内核。

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。通常可以通过 pid 的赋值来表示线程的创建和身份确定。“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。

接下来, 在下一小节中我们将对第一个真正的内核进程进行初始化 (因为 `idle` 进程仅仅算是“继承了”`uCore` 的运行)。我们的目标是使用新的内核进程进行一下内核初始化的工作。

7.1.3.3.6 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。所以 `uCore` 接下来还需创建其他进程来完成各种工作, 但 `idleproc` 内核子线程自己不想做, 于是就通过调用 `kernel_thread` 函数创建了一个内核线程 `init_main`。在实验四中, 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)。但在后续的实验中, `init_main` 的工作就是创建特定的其他内核线程或用户进程 (实验五涉及)。下面我们来分析一下创建内核线程的函数 `kernel_thread`:

```
int kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    // 对trampoline, 也就是我们程序的一些上下文进行一些初始化
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));

    // 设置内核线程的参数和函数指针
    tf.gpr.s0 = (uintptr_t)fn; // s0 寄存器保存函数指针
    tf.gpr.s1 = (uintptr_t)arg; // s1 寄存器保存函数参数

    // 设置 trapframe 中的 status 寄存器 (SSTATUS)
    // SSTATUS_SPP: Supervisor Previous Privilege (设置为 supervisor 模式, 因为这是一
    → 个内核线程)
    // SSTATUS_SPIE: Supervisor Previous Interrupt Enable (设置为启用中断, 因为这是一
    → 个内核线程)
    // SSTATUS_SIE: Supervisor Interrupt Enable (设置为禁用中断, 因为我们不希望该线程
    → 被中断)
    tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~SSTATUS_SIE;

    // 将入口点 (epc) 设置为 kernel_thread_entry 函数, 作用实际上是将pc指针指向它(*tra
    → pentry.S会用到)
    tf.epc = (uintptr_t)kernel_thread_entry;

    // 使用 do_fork 创建一个新进程 (内核线程), 这样才真正用设置的tf创建新进程。
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

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, 随后设置设置内核线程的参数和函数指针。要特别注意对 `tf.status` 的赋值过程, 其读取 `sstatus` 寄存器的值, 然后根据特定的位操作, 设置 `SPP` 和 `SPIE` 位, 并同时清除 `SIE` 位, 从而实现特权级别切换、保留中断使能状态并禁用中断的操作。而 `initproc` 内核线程从哪里开始执行呢? `tf.epc` 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”, 并把函数 `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（练习 2）。`do_fork` 函数主要做了以下 7 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_flags` 决定是复制还是共享内存管理系统（`copy_mm` 函数）
4. 设置进程的中断帧和上下文（`copy_thread` 函数）
5. 把设置好的进程加入链表
6. 将新建的进程设为就绪态
7. 将返回值设为线程 id

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`，这是由于目前在实验四中只能创建内核线程，`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。

在这里我们需要尤其关注 `copy_thread` 函数：

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE - sizeof(struct trapframe)
    ↪e);
    *(proc->tf) = *tf;

    // Set a0 to 0 so a child process knows it's just forked
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;

    proc->context.ra = (uintptr_t)forkret;
    proc->context.sp = (uintptr_t)(proc->tf);
}
```

在这里我们首先在上面分配的内核栈上分配出一片空间来保存 `trapframe`。然后，我们将 `trapframe` 中的 `a0` 寄存器（返回值）设置为 0，说明这个进程是一个子进程。之后我们将上下文中的 `ra` 设置为了 `forkret` 函数的入口，并且把 `trapframe` 放在上下文的栈顶。在下一个小节，我们会看到这么做之后 `uCore` 是如何完成进程切换的。

7.1.3.3.7 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，回顾前面“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` 在实验四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. 将当前运行的进程设置为要切换过去的进程
2. 将页表换成新进程的页表
3. 使用 `switch_to` 切换到新进程

第三步 `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: `process 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。我们再看看 `switch.S` 中的 `switch_to` 函数的执行流程:

```
.text
# void switch_to(struct proc_struct* from, struct proc_struct* to)
.globl switch_to
switch_to:
    # save from's registers
    STORE ra, 0*REGBYTES(a0)
    STORE sp, 1*REGBYTES(a0)
    STORE s0, 2*REGBYTES(a0)
    STORE s1, 3*REGBYTES(a0)
    STORE s2, 4*REGBYTES(a0)
    STORE s3, 5*REGBYTES(a0)
    STORE s4, 6*REGBYTES(a0)
    STORE s5, 7*REGBYTES(a0)
    STORE s6, 8*REGBYTES(a0)
    STORE s7, 9*REGBYTES(a0)
    STORE s8, 10*REGBYTES(a0)
    STORE s9, 11*REGBYTES(a0)
    STORE s10, 12*REGBYTES(a0)
    STORE s11, 13*REGBYTES(a0)

    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
    LOAD s3, 5*REGBYTES(a1)
    LOAD s4, 6*REGBYTES(a1)
    LOAD s5, 7*REGBYTES(a1)
    LOAD s6, 8*REGBYTES(a1)
    LOAD s7, 9*REGBYTES(a1)
    LOAD s8, 10*REGBYTES(a1)
    LOAD s9, 11*REGBYTES(a1)
    LOAD s10, 12*REGBYTES(a1)
    LOAD s11, 13*REGBYTES(a1)

    ret
```

可以看出这段代码就是将需要保存的寄存器进行保存和调换。其中的 `a0` 和 `a1` 是 RISC-V 架构中通用寄存器, 它们用于传递参数, 也就是说 `a0` 指向原进程, `a1` 指向目的进程。

在之前我们也已经谈到过了, 这里只需要调换被调用者保存寄存器即可。由于我们在初始化时把上下文的

ra 寄存器设定成了 forkret 函数的入口, 所以这里会返回到 forkret 函数。forkrets 函数很短, 位于 kern/trap/trapentry.S:

```
.globl forkrets
forkrets:
    # set stack to this new process's trapframe
    move sp, a0
    j __trapret
```

这里把传进来的参数, 也就是进程的中断帧放在了 sp, 这样在 __trapret 中就可以直接从中断帧里面恢复所有的寄存器啦! 我们在初始化的时候对于中断帧做了一点手脚, epc 寄存器指向的是 kernel_thread_entry, s0 寄存器里放的是新进程要执行的函数, s1 寄存器里放的是传给函数的参数。在 kernel_thread_entry 函数中:

```
.text
.globl kernel_thread_entry
kernel_thread_entry:      # void kernel_thread(void)
    move a0, s1
    jalr s0

    jal do_exit
```

我们把参数放在了 a0 寄存器, 并跳转到 s0 执行我们指定的函数! 这样, 一个进程的初始化就完成了。至此, 我们实现了基本的进程管理, 并且成功创建并切换到了我们的第一个内核进程。

7.1.4 实验报告要求

从 git server 网站上取得 ucore_lab 后, 进入目录 labcodes/lab4, 完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 labcodes/lab4 下存放实验报告, 实验报告文档命名为 lab4-学生 ID.md。推荐使用 markdown 格式。对于 lab4 中编程任务, 完成编写之后, 再通过 git push 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

注意有“LAB4”的注释, 代码中所有需要完成的地方(challenge 除外)都有“LAB4”和“YOUR CODE”的注释, 请在提交时特别注意保持注释, 并将“YOUR CODE”替换为自己的学号, 并且将所有标有对应注释的部分填上正确的代码。

实验 4 完成了内核线程，但到目前为止，所有的运行都在内核态执行。实验 5 将创建用户进程，让用户进程在用户态执行，且在需要 `ucore` 支持时，可通过系统调用来让 `ucore` 提供服务。

8.1 本章内容

8.1.1 实验目的

- 了解第一个用户进程创建过程
- 了解系统调用框架的实现机制
- 了解 `ucore` 如何实现系统调用 `sys_fork/sys_exec/sys_exit/sys_wait` 来进行进程管理

8.1.2 实验内容

实验 4 完成了内核线程，但到目前为止，所有的运行都在内核态执行。实验 5 将创建用户进程，让用户进程在用户态执行，且在需要 `ucore` 支持时，可通过系统调用来让 `ucore` 提供服务。为此需要构造出第一个用户进程，并通过系统调用 `'sys_fork'/sys_exec/sys_exit/sys_wait` 来支持运行不同的应用程序，完成对用户进程的执行过程的基本管理。相关原理介绍可看附录 B。

8.1.2.1 本节内容

8.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 ucore_lab 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

8.1.2.1.1.1 练习 0: 填写已有实验

本实验依赖实验 1/2/3/4。请把你做的实验 1/2/3/4 的代码填入本实验中代码中有 “LAB1” / “LAB2” / “LAB3” / “LAB4” 的注释相应部分。注意: 为了能够正确执行 lab5 的测试应用程序, 可能需对已完成的实验 1/2/3/4 的代码进行进一步改进。

8.1.2.1.1.2 练习 1: 加载应用程序并执行 (需要编码)

`do_execv` 函数调用 `load_icode` (位于 `kern/process/proc.c` 中) 来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序。你需要补充 `load_icode` 的第 6 步, 建立相应的用户内存空间来放置应用程序的代码段、数据段等, 且要设置好 `proc_struct` 结构中的成员变量 `trapframe` 中的内容, 确保在执行此进程后, 能够从应用程序设定的起始执行地址开始执行。需设置正确的 `trapframe` 内容。

请在实验报告中简要说明你的设计实现过程。

- 请简要描述这个用户态进程被 ucore 选择占用 CPU 执行 (RUNNING 态) 到具体执行应用程序第一条指令的整个经过。

8.1.2.1.1.3 练习 2: 父进程复制自己的内存空间给予进程 (需要编码)

创建子进程的函数 `do_fork` 在执行中将拷贝当前进程 (即父进程) 的用户内存地址空间中的合法内容到新进程中 (子进程), 完成内存资源的复制。具体是通过 `copy_range` 函数 (位于 `kern/mm/pmm.c` 中) 实现的, 请补充 `copy_range` 的实现, 确保能够正确执行。

请在实验报告中简要说明你的设计实现过程。

- 如何设计实现 Copy on Write 机制? 给出概要设计, 鼓励给出详细设计。

Copy-on-write (简称 COW) 的基本概念是指如果有多个使用者对一个资源 A (比如内存块) 进行读操作, 则每个使用者只需获得一个指向同一个资源 A 的指针, 就可以该资源了。若某使用者需对这个资源 A 进行写操作, 系统会对该资源进行拷贝操作, 从而使得该 “写操作” 使用者获得一个该资源 A 的 “私有” 拷贝—资源 B, 可对资源 B 进行写操作。该 “写操作” 使用者对资源 B 的改变对于其他的使用者而言是不可见的, 因为其他使用者看到的还是资源 A。

8.1.2.1.1.4 练习 3: 阅读分析源代码, 理解进程执行 fork/exec/wait/exit 的实现, 以及系统调用的实现 (不需要编码)

请在实验报告中简要说明你对 fork/exec/wait/exit 函数的分析。并回答如下问题:

- 请分析 fork/exec/wait/exit 的执行流程。重点关注哪些操作是在用户态完成, 哪些是在内核态完成? 内核态与用户态程序是如何交错执行的? 内核态执行结果是如何返回给用户程序的?
- 请给出 ucore 中一个用户态进程的执行状态生命周期图 (包括执行状态, 执行状态之间的变换关系, 以及产生变换的事件或函数调用)。(字符方式画即可)

执行: make grade。如果所显示的应用程序检测都输出 ok, 则基本正确。(使用的是 qemu-1.0.1)

8.1.2.1.1.5 扩展练习 Challenge

1. 实现 Copy on Write (COW) 机制

给出实现源码, 测试用例和设计报告 (包括在 cow 情况下的各种状态转换 (类似有限状态自动机) 的说明)。

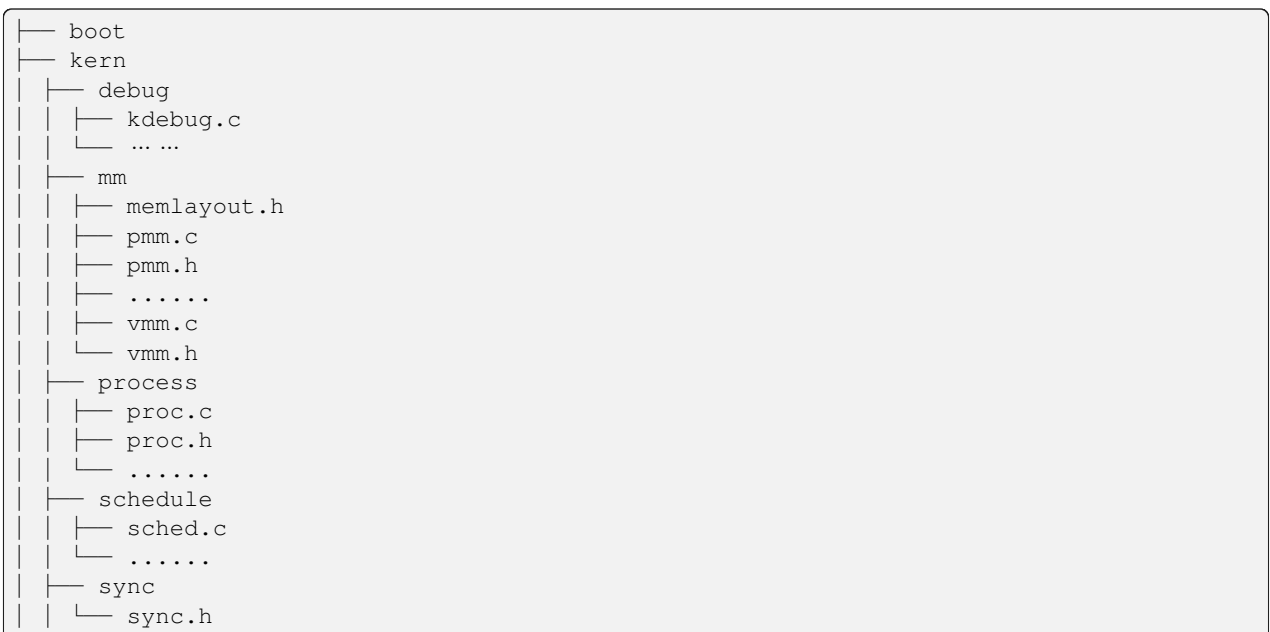
这个扩展练习涉及到本实验和上一个实验 “虚拟内存管理”。在 ucore 操作系统中, 当一个用户父进程创建自己的子进程时, 父进程会把其申请的用户空间设置为只读, 子进程可共享父进程占用的用户内存空间中的页面 (这就是一个共享的资源)。当其中任何一个进程修改此用户内存空间中的某页面时, ucore 会通过 page fault 异常获知该操作, 并完成拷贝内存页面, 使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请在 ucore 中实现这样的 COW 机制。

由于 COW 实现比较复杂, 容易引入 bug, 请参考 <https://dirtycow.ninja/> 看看能否在 ucore 的 COW 实现中模拟这个错误和解决方案。需要有解释。

这是一个 big challenge.

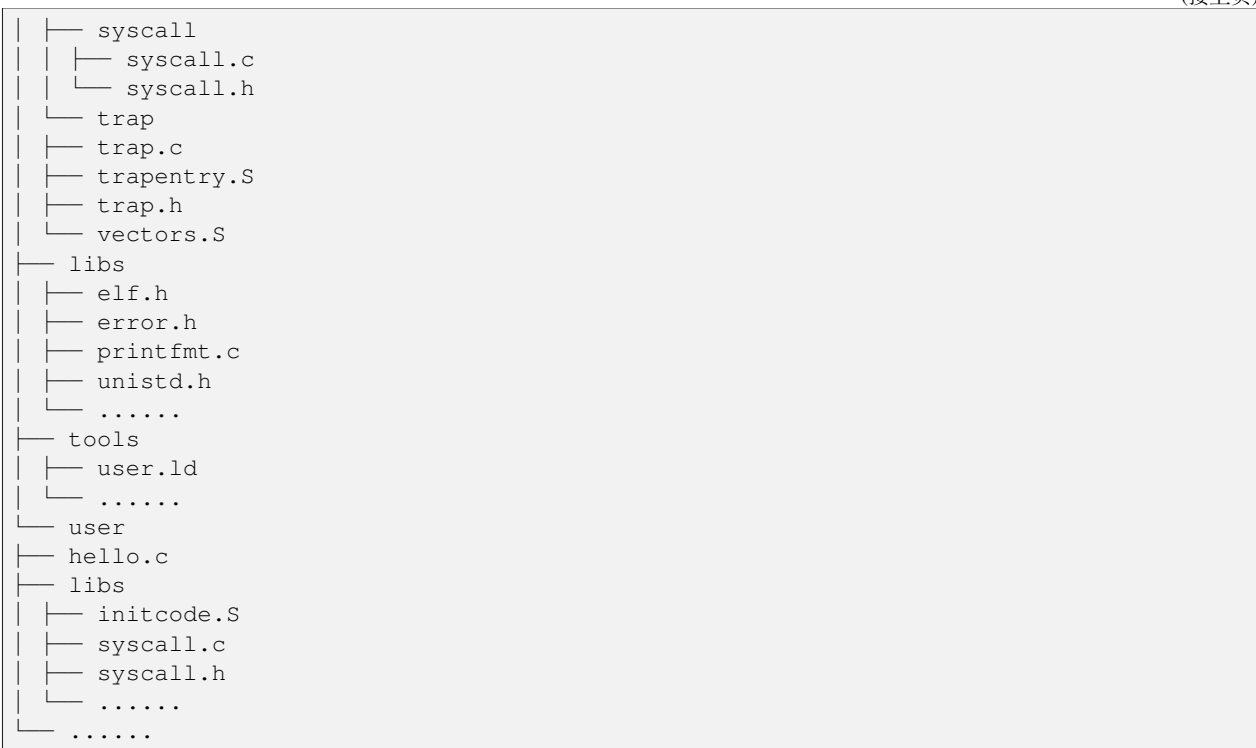
2. 说明该用户程序是何时被预先加载到内存中的? 与我们常用操作系统的加载有何区别, 原因是什么?

8.1.2.1.2 项目组成



(续下页)

(接上页)



相对与实验四，实验五主要增加的文件如上表红色部分所示，主要修改的文件如上表紫色部分所示。主要改动如下：

◆ kern/debug/

kdebug.c: 修改: 解析用户进程的符号信息表示 (可不用理会)

◆ kern/mm/ (与本次实验有较大关系)

memlayout.h: 修改: 增加了用户虚存地址空间的图形表示和宏定义 (需仔细理解)。

pmm.[ch]: 修改: 添加了用于进程退出 (do_exit) 的内存资源回收的 page_remove_pte、unmap_range、exit_range 函数和用于创建子进程 (do_fork) 中拷贝父进程内存空间的 copy_range 函数, 修改了 pgdir_alloc_page 函数

vmm.[ch]: 修改: 扩展了 mm_struct 数据结构, 增加了一系列函数

- mm_map/dup_mmap/exit_mmap: 设定/取消/复制/删除用户进程的合法内存空间
- copy_from_user/copy_to_user: 用户内存空间内容与内核内存空间内容的相互拷贝的实现
- user_mem_check: 搜索 vma 链表, 检查是否是一个合法的用户空间范围

◆ kern/process/ (与本次实验有较大关系)

proc.[ch]: 修改: 扩展了 proc_struct 数据结构。增加或修改了一系列函数

- setup_pgdir/put_pgdir: 创建并设置/释放页目录表
- copy_mm: 复制用户进程的内存空间和设置相关内存管理 (如页表等) 信息
- do_exit: 释放进程自身所占内存空间和相关内存管理 (如页表等) 信息所占空间, 唤醒父进程, 好让父进程收了自己, 让调度器切换到其他进程
- load_icode: 被 do_execve 调用, 完成加载放在内存中的执行程序到进程空间, 这涉及到对页表等的修改, 分配用户栈

- `do_execve`: 先回收自身所占用户空间, 然后调用 `load_icode`, 用新的程序覆盖内存空间, 形成一个执行新程序的新进程
- `do_yield`: 让调度器执行一次选择新进程的过程
- `do_wait`: 父进程等待子进程, 并在得到子进程的退出消息后, 彻底回收子进程所占的资源 (比如子进程的内核栈和进程控制块)
- `do_kill`: 给一个进程设置 `PF_EXITING` 标志 (“kill” 信息, 即要它死掉), 这样在 `trap` 函数中, 将根据此标志, 让进程退出
- `KERNEL_EXECVE/__KERNEL_EXECVE/__KERNEL_EXECVE2`: 被 `user_main` 调用, 执行一用户进程

◆ kern/trap/

`trap.c`: 修改: 在 `idt_init` 函数中, 对 IDT 初始化时, 设置好了用于系统调用的中断门 (`idt[T_SYSCALL]`) 信息。这主要与 `syscall` 的实现相关

◆ user/*

新增的用户程序和用户库

8.1.3 用户进程管理

8.1.3.1 本节内容

8.1.3.1.1 实验执行流程概述

之前我们已经实现了内存的管理和内核进程的建立。但是那都是在内核态。

接下来我们将在用户态运行一些程序。

用户程序, 也就是我们在计算机系前几年课程里一直在写的那些程序, 到底怎样在操作系统上跑起来?

首先需要编译器把用户程序的源代码编译为可以在 CPU 执行的目标程序, 这个目标程序里, 既要有执行的代码, 又要有关于内存分配的一些信息, 告诉我们应该怎样为这个程序分配内存。

我们先不考虑怎样在 `ucore` 里运行编译器, 只考虑 `ucore` 如何把编译好的用户程序运行起来。这需要给它分配一些内存, 把程序代码加载进来, 建立一个进程, 然后通过调度让这个用户进程开始执行。

8.1.3.1.2 系统调用 (system call)

操作系统应当提供给用户程序一些接口, 让用户程序使用操作系统提供的服务。这些接口就是**系统调用**。用户程序在用户态运行 (U mode), 系统调用在内核态执行 (S mode)。这里有一个 CPU 的特权级切换的过程, 要用到 `ecall` 指令从 U mode 进入 S mode。想想我们之前用 `ecall` 做过什么? 在 S mode 调用 OpenSBI 提供的 M mode 接口。当时我们用 `ecall` 进入了 M mode, 剩下的事情就交给 OpenSBI 来完成, 然后我们收到 OpenSBI 返回的结果。

现在我们用 `ecall` 从 U mode 进入 S mode 之后, 对应的处理需要我们编写内核系统调用的代码来完成。

另外, 我们总不能让用户程序里直接调用 `ecall`。通常我们会把这样的系统调用操作封装成一个个的函数, 作为“标准库”提供给用户使用。例如在 linux 里, 写一个 C 程序时使用 `printf()` 函数进行输出, 实际上是要进行 `write()` 的系统调用, 通过内核把输出打印到命令行或其他地方。

对于用户进程的管理, 有四个系统调用比较重要。

`sys_fork()`: 把当前的进程复制一份, 创建一个子进程, 原先的进程是父进程。接下来两个进程都会收到 `sys_fork()` 的返回值, 如果返回 0 说明当前位于子进程中, 返回一个非 0 的值 (子进程的 PID) 说明当前位于父进程中。然后就可以根据返回值的不同, 在两个进程里进行不同的处理。

`sys_exec()`: 在当前的进程下, 停止原先正在运行的程序, 开始执行一个新程序。PID 不变, 但是内存空间要重新分配, 执行的机器代码发生了改变。我们可以用 `fork()` 和 `exec()` 配合, 在当前程序不停止的情况下, 开始执行另一个程序。

`sys_exit()`: 退出当前的进程。

`sys_wait()`: 挂起当前的进程, 等到特定条件满足的时候再继续执行。

下面我们从 [ucore 实验指导书](#) 摘抄一段关于用户进程的理论讲解。

8.1.3.1.3 从内核线程到用户进程

在实验四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。但实验四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?

但其实我们已经在实验四中看到了内核线程, 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, 如何管理物理内存等)。

8.1.3.1.4 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, 所以需要提供某种机制, 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。

8.1.3.1.5 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具体实现。如果考虑操作系统的具体实现, 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。

8.1.3.1.6 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。

但在用户进程的执行过程中，具体在哪个时间段处于上述状态的呢？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。

8.1.3.1.7 创建用户进程

在实验四中，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。

8.1.3.1.7.1 1. 应用程序的组成和编译

我们首先来看一个应用程序，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) {
    cprintf("Hello world!!.\n");
    cprintf("I am process %d.\n", getpid());
    cprintf("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。然后在本实验源码目录下执行 make，可得到如下输出：

```

... ..
+ cc user/hello.c
riscv64-unknown-elf-gcc -Iuser/ -mmodel=medany -O2 -std=gnu99 -Wno-unused -fno-builiti
↪n -Wall -nostdinc -fno-stack-protector -ffunction-sections -fdata-sections -Ilibs/
↪-Iuser/include/ -Iuser/libs/ -c user/hello.c -o obj/user/hello.o
+ cc user/libs/panic.c
riscv64-unknown-elf-gcc -Iuser/libs/ -mmodel=medany -O2 -std=gnu99 -Wno-unused -fno-b
↪uiltin -Wall -nostdinc -fno-stack-protector -ffunction-sections -fdata-sections -Il
↪ibs/ -Iuser/include/ -Iuser/libs/ -c user/libs/panic.c -o obj/user/libs/panic.o
... ..

```

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。这些用户态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 bootloader 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。而到了与文件系统相关的实验后, 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, 而可以用大家熟悉的文件方式进行加载了。

8.1.3.1.7.2 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:

```

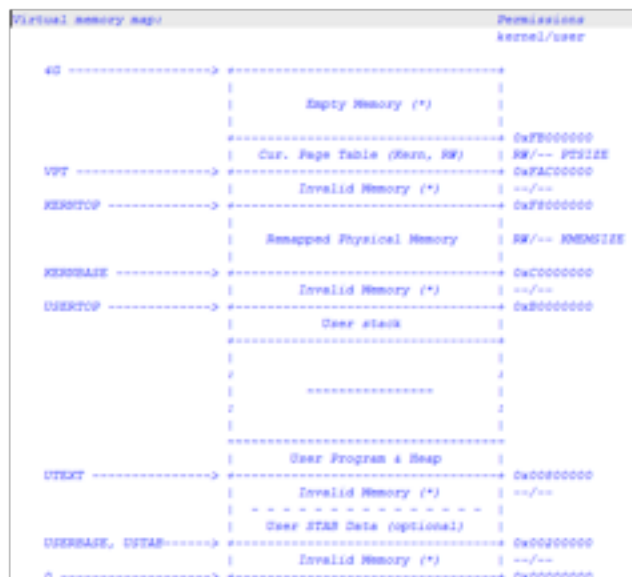
BASE_ADDRESS = 0xFFFFFFF0200000;

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

```

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:



8.1.3.1.7.3 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。在本实验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:

```
// kern/process/proc.c
#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);
})

#define __KERNEL_EXECVE2(x, xstart, xsize) ({
    extern unsigned char xstart[], xsize[];
    __KERNEL_EXECVE(#x, xstart, (size_t)xsize);
})

#define KERNEL_EXECVE2(x, xstart, xsize)    __KERNEL_EXECVE2(x, xstart, xsize)
```

(续下页)

(接上页)

```
// user_main - kernel thread used to exec a user program
static int
user_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(exit);
#endif
    panic("user_main execve failed.\n");
}
```

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;
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;
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, 并分配一定数量的物理内存且建立好栈的虚地址 <-> 物理地址映射关系;

- 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;
- 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;

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。

8.1.3.1.8 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。

```
// /user/libs/ulib.c

void
exit(int error_code) {
    sys_exit(error_code);
    cprintf("BUG: exit failed.\n");
    while (1);
}
```

首先, exit 函数会把一个退出码 error_code 传递给 ucore, ucore 通过执行位于 /proc/process/proc.c 中的内核函数 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:

```
// /proc/process/proc.c

int
do_exit(int error_code) {
    // 检查当前进程是否为idleproc或initproc, 如果是, 发出panic
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }

    // 获取当前进程的内存管理结构mm
    struct mm_struct *mm = current->mm;

    // 如果mm不为空, 说明是用户进程
    if (mm != NULL) {
        // 切换到内核页表, 确保接下来的操作在内核空间执行
        lcr3(boot_cr3);
    }
}
```

(续下页)

(接上页)

```

// 如果mm引用计数减到0, 说明没有其他进程共享此mm
if (mm_count_dec(mm) == 0) {
    // 释放用户虚拟内存空间相关的资源
    exit_mmap(mm);
    put_pgdir(mm);
    mm_destroy(mm);
}
// 将当前进程的mm设置为NULL, 表示资源已经释放
current->mm = NULL;
}

// 设置进程状态为PROC_ZOMBIE, 表示进程已退出
current->state = PROC_ZOMBIE;
current->exit_code = error_code;

bool intr_flag;
struct proc_struct *proc;

// 关中断
local_intr_save(intr_flag);
{
    // 获取当前进程的父进程
    proc = current->parent;

    // 如果父进程处于等待子进程状态, 则唤醒父进程
    if (proc->wait_state == WT_CHILD) {
        wakeup_proc(proc);
    }

    // 遍历当前进程的所有子进程
    while (current->cptr != NULL) {
        proc = current->cptr;
        current->cptr = proc->optr;

        // 设置子进程的父进程为initproc, 并加入initproc的子进程链表
        proc->yptr = NULL;
        if ((proc->optr = initproc->cptr) != NULL) {
            initproc->cptr->yptr = proc;
        }
        proc->parent = initproc;
        initproc->cptr = proc;

        // 如果子进程也处于退出状态, 唤醒initproc
        if (proc->state == PROC_ZOMBIE) {
            if (initproc->wait_state == WT_CHILD) {
                wakeup_proc(initproc);
            }
        }
    }
}

// 开中断
local_intr_restore(intr_flag);

// 调用调度器, 选择新的进程执行
schedule();

// 如果执行到这里, 表示代码执行出现错误, 发出panic

```

(续下页)

(接上页)

```
panic("do_exit will not return!! %d.\n", current->pid);
}
```

下面我们来看这些系统调用的实现。

8.1.3.1.9 系统调用实现

系统调用，是用户态 (U mode) 的程序获取内核态 (S mode) 服务的方法，所以需要在用户态和内核态都加入对应的支持和处理。我们也可以认为用户态只是提供一个调用的接口，真正的处理都在内核态进行。

首先我们在头文件里定义一些系统调用的编号。

```
// libs/unistd.h
#ifndef __LIBS_UNISTD_H__
#define __LIBS_UNISTD_H__

#define T_SYSCALL          0x80

/* syscall number */
#define SYS_exit           1
#define SYS_fork           2
#define SYS_wait           3
#define SYS_exec           4
#define SYS_clone          5
#define SYS_yield         10
#define SYS_sleep          11
#define SYS_kill           12
#define SYS_gettime        17
#define SYS_getpid         18
#define SYS_brk            19
#define SYS_mmap           20
#define SYS_munmap         21
#define SYS_shmem          22
#define SYS_putc           30
#define SYS_pgdir          31

/* SYS_fork flags */
#define CLONE_VM            0x00000100 // set if VM shared between processes
#define CLONE_THREAD        0x00000200 // thread group

#endif /* !__LIBS_UNISTD_H__ */
```

我们注意在用户态进行系统调用的核心操作是，通过内联汇编进行 `ecall` 环境调用。这将产生一个 `trap`，进入 S mode 进行异常处理。

```
// user/libs/syscall.c
#include <defs.h>
#include <unistd.h>
#include <stdarg.h>
#include <syscall.h>
#define MAX_ARGS           5
static inline int syscall(int num, ...) {
    //va_list, va_start, va_arg都是C语言处理参数个数不定的函数的宏
    //在 stdarg.h 里定义
```

(续下页)

(接上页)

```

va_list ap; //ap: 参数列表(此时未初始化)
va_start(ap, num); //初始化参数列表, 从num开始
//First, va_start initializes the list of variable arguments as a va_list.
uint64_t a[MAX_ARGS];
int i, ret;
for (i = 0; i < MAX_ARGS; i++) { //把参数依次取出
    /*Subsequent executions of va_arg yield the values of the additional
    arguments
    in the same order as passed to the function.*/
    a[i] = va_arg(ap, uint64_t);
}
va_end(ap); //Finally, va_end shall be executed before the function returns.
asm volatile (
    "ld a0, %1\n"
    "ld a1, %2\n"
    "ld a2, %3\n"
    "ld a3, %4\n"
    "ld a4, %5\n"
    "ld a5, %6\n"
    "ecall\n"
    "sd a0, %0"
    : "=m" (ret)
    : "m" (num), "m" (a[0]), "m" (a[1]), "m" (a[2]), "m" (a[3]), "m" (a[4])
    : "memory");
//num存到a0寄存器, a[0]存到a1寄存器
//ecall的返回值存到ret
return ret;
}

int sys_exit(int error_code) { return syscall(SYS_exit, error_code); }
int sys_fork(void) { return syscall(SYS_fork); }
int sys_wait(int pid, int *store) { return syscall(SYS_wait, pid, store); }
int sys_yield(void) { return syscall(SYS_yield); }
int sys_kill(int pid) { return syscall(SYS_kill, pid); }
int sys_getpid(void) { return syscall(SYS_getpid); }
int sys_putc(int c) { return syscall(SYS_putc, c); }

```

我们下面看看 trap.c 是如何转发这个系统调用的。

```

// kern/trap/trap.c
void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) { //通过中断帧里 scause寄存器的数值, 判断出当前是来自USER_ECALL
    的异常
        case CAUSE_USER_ECALL:
            //cprintf("Environment call from U-mode\n");
            tf->epc += 4;
            //sepc寄存器是产生异常的指令的位置, 在异常处理结束后, 会回到sepc的位置继续
            执行
            //对于ecall, 我们希望sepc寄存器要指向产生异常的指令(ecall)的下一条指令
            //否则就会回到ecall执行再执行一次ecall, 无限循环
            syscall(); // 进行系统调用处理
            break;
            /*other cases .... */
        }
    }
}
// kern/syscall/syscall.c
#include <unistd.h>

```

(续下页)

(接上页)

```

#include <proc.h>
#include <syscall.h>
#include <trap.h>
#include <stdio.h>
#include <pmm.h>
#include <assert.h>
//这里把系统调用进一步转发给proc.c的do_exit(), do_fork()等函数
static int sys_exit(uint64_t arg[]) {
    int error_code = (int)arg[0];
    return do_exit(error_code);
}
static int sys_fork(uint64_t arg[]) {
    struct trapframe *tf = current->tf;
    uintptr_t stack = tf->gpr.sp;
    return do_fork(0, stack, tf);
}
static int sys_wait(uint64_t arg[]) {
    int pid = (int)arg[0];
    int *store = (int *)arg[1];
    return do_wait(pid, store);
}
static int sys_exec(uint64_t arg[]) {
    const char *name = (const char *)arg[0];
    size_t len = (size_t)arg[1];
    unsigned char *binary = (unsigned char *)arg[2];
    size_t size = (size_t)arg[3];
    //用户态调用的exec(), 归根结底是do_execve()
    return do_execve(name, len, binary, size);
}
static int sys_yield(uint64_t arg[]) {
    return do_yield();
}
static int sys_kill(uint64_t arg[]) {
    int pid = (int)arg[0];
    return do_kill(pid);
}
static int sys_getpid(uint64_t arg[]) {
    return current->pid;
}
static int sys_putc(uint64_t arg[]) {
    int c = (int)arg[0];
    cputchar(c);
    return 0;
}
//这里定义了函数指针的数组syscalls, 把每个系统调用编号的下标上初始化为对应的函数指针
static int (*syscalls[])(uint64_t arg[]) = {
    [SYS_exit]      sys_exit,
    [SYS_fork]      sys_fork,
    [SYS_wait]      sys_wait,
    [SYS_exec]      sys_exec,
    [SYS_yield]     sys_yield,
    [SYS_kill]      sys_kill,
    [SYS_getpid]    sys_getpid,
    [SYS_putc]      sys_putc,
};

#define NUM_SYSCALLS ((sizeof(syscalls)) / (sizeof(syscalls[0])))

```

(续下页)

(接上页)

```

void syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0; //a0寄存器保存了系统调用编号
    if (num >= 0 && num < NUM_SYSCALLS) { //防止syscalls[num]下标越界
        if (syscalls[num] != NULL) {
            arg[0] = tf->gpr.a1;
            arg[1] = tf->gpr.a2;
            arg[2] = tf->gpr.a3;
            arg[3] = tf->gpr.a4;
            arg[4] = tf->gpr.a5;
            tf->gpr.a0 = syscalls[num](arg);
            //把寄存器里的参数取出来，转发给系统调用编号对应的函数进行处理
            return ;
        }
    }
    //如果执行到这里，说明传入的系统调用编号还没有被实现，就崩掉了。
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
        num, current->pid, current->name);
}

```

这样我们就完成了系统调用的转发。接下来就是在 `do_exit()`，`do_execve()` 等函数中进行具体处理了。

我们看看 `do_execve()` 函数

```

// kern/mm/vmm.c
bool user_mem_check(struct mm_struct *mm, uintptr_t addr, size_t len, bool write) {
    //检查从addr开始长为len的一段内存能否被用户态程序访问
    if (mm != NULL) {
        if (!USER_ACCESS(addr, addr + len)) {
            return 0;
        }
        struct vma_struct *vma;
        uintptr_t start = addr, end = addr + len;
        while (start < end) {
            if ((vma = find_vma(mm, start)) == NULL || start < vma->vm_start) {
                return 0;
            }
            if (!(vma->vm_flags & ((write) ? VM_WRITE : VM_READ))) {
                return 0;
            }
            if (write && (vma->vm_flags & VM_STACK)) {
                if (start < vma->vm_start + PGSIZE) { //check stack start & size
                    return 0;
                }
            }
            start = vma->vm_end;
        }
        return 1;
    }
    return KERN_ACCESS(addr, addr + len);
}
// kern/process/proc.c
// do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of current pro

```

(续下页)

(接上页)

```

↪cess
//      - call load_icode to setup new memory space according binary prog.
int do_execve(const char *name, size_t len, unsigned char *binary, size_t size) {
    struct mm_struct *mm = current->mm;
    if (!user_mem_check(mm, (uintptr_t)name, len, 0)) { //检查name的内存空间能否被访问
        return -E_INVAL;
    }
    if (len > PROC_NAME_LEN) { //进程名字的长度有上限 PROC_NAME_LEN, 在proc.h定义
        len = PROC_NAME_LEN;
    }
    char local_name[PROC_NAME_LEN + 1];
    memset(local_name, 0, sizeof(local_name));
    memcpy(local_name, name, len);

    if (mm != NULL) {
        cputs("mm != NULL");
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm); //把进程当前占用的内存释放, 之后重新分配内存
        }
        current->mm = NULL;
    }
    //把新的程序加载到当前进程里的工作都在load_icode()函数里完成
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit; //返回不为0, 则加载失败
    }
    set_proc_name(current, local_name);
    //如果set_proc_name的实现不变, 为什么不能直接set_proc_name(current, name)?
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}

```

那么我们如何实现 kernel_execve() 函数?

能否直接调用 do_execve()?

```

// kern/process/proc.c
static int kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
    ret = do_execve(name, len, binary, size);
    cprintf("ret = %d\n", ret);
    return ret;
}

```

很不幸。这么做行不通。do_execve() load_icode() 里面只是构建了用户程序运行的上下文, 但是并没有完成切换。上下文切换实际上要借助中断处理的返回来完成。直接调用 do_execve() 是无法完成上下文切换的。如果是在用户态调用 exec(), 系统调用的 ecall 产生的中断返回时, 就可以完成上下文切换。

由于目前我们在 S mode 下, 所以不能通过 ecall 来产生中断。我们这里采取一个取巧的办法, 用 ebreak 产生断点中断进行处理, 通过设置 a7 寄存器的值为 10 说明这不是一个普通的断点中断, 而是要转发到 syscall(), 这样用一个不是特别优雅的方式, 实现了在内核态使用系统调用。

```
// kern/process/proc.c
// kernel_execve - do SYS_exec syscall to exec a user program called by user_main kern
↳ el_thread
static int kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
    asm volatile(
        "li a0, %1\n"
        "lw a1, %2\n"
        "lw a2, %3\n"
        "lw a3, %4\n"
        "lw a4, %5\n"
        "li a7, 10\n"
        "ebreak\n"
        "sw a0, %0\n"
        : "=m"(ret)
        : "i"(SYS_exec), "m"(name), "m"(len), "m"(binary), "m"(size)
        : "memory"); //这里内联汇编的格式, 和用户态调用ecall的格式类似, 只是ecall换成
↳ 了ebreak
    cprintf("ret = %d\n", ret);
    return ret;
}

// kern/trap/trap.c
void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        case CAUSE_BREAKPOINT:
            cprintf("Breakpoint\n");
            if (tf->gpr.a7 == 10) {
                tf->epc += 4; //注意返回时要执行ebreak的下一条指令
                syscall();
            }
            break;
        /* other cases ... */
    }
}
}
```

注意我们需要让 CPU 进入 U mode 执行 `do_execve()` 加载的用户程序。进行系统调用 `sys_exec` 之后, 我们在 trap 返回的时候调用了 `sret` 指令, 这时只要 `sstatus` 寄存器的 `SPP` 二进制位为 0, 就会切换到 U mode, 但 `SPP` 存储的是“进入 trap 之前来自什么特权级”, 也就是说我们这里 `ebreak` 之后 `SPP` 的数值为 1, `sret` 之后会回到 S mode 在内核态执行用户程序。所以 `load_icode()` 函数在构造新进程的时候, 会把 `SSTATUS_SPP` 设置为 0, 使得 `sret` 的时候能回到 U mode。

至此, 实验五中的主要工作描述完毕。

8.1.4 实验报告要求

从 git server 网站上取得 `ucore_lab` 后, 进入目录 `labcodes/lab5`, 完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 `labcodes/lab5` 下存放实验报告, 实验报告文档命名为 `lab5-学生 ID.md`。推荐使用 **markdown** 格式。对于 lab5 中编程任务, 完成编写之后, 再通过 `git push` 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

注意有“LAB5”的注释, 代码中所有需要完成的地方 (challenge 除外) 都有“LAB5”和“YOUR CODE”的注释, 请在提交时特别注意保持注释, 并将“YOUR CODE”替换为自己的学号, 并且将所有标有对应注释的部分填上正确的代码。

弱水汨其为难兮，路中断而不通。

现在我们要在最小可执行内核的基础上，支持中断机制，并且用时钟中断来检验我们的中断处理系统。

9.1 本章内容

9.1.1 实验目的

- 理解操作系统的调度管理机制
- 熟悉 ucore 的系统调度器框架，以及缺省的 Round-Robin 调度算法
- 基于调度器框架实现一个 (Stride Scheduling) 调度算法来替换缺省的调度算法

9.1.2 实验内容

在前两章中，我们已经分别实现了内核进程和用户进程，并且让他们正确运行了起来。同时我们也实现了一个简单的调度算法，FIFO 调度算法，来对我们的进程进行调度，可通过阅读实验五下的 kern/schedule/sched.c 的 schedule 函数的实现来了解其 FIFO 调度策略。但是，单单如此就够了吗？显然，我们可以让 ucore 支持更加丰富的调度算法，从而满足各方面的调度需求。与实验五相比，实验六专门需要针对处理器调度框架和各种算法进行设计与实现，为此对 ucore 的调度部分进行了适当的修改，使得 kern/schedule/sched.c 只实现调度器框架，而不再涉及具体的调度算法实现。而调度算法在单独的文件（default_sched.[ch]）中实现。

在本次实验中，我们在 'init/init.c' 中加入了对 'sched_init' 函数的调用。这个函数主要完成调度器和特定调度算法的绑定。初始化后，我们在调度函数中就可以使用相应的接口了。这也是在 C 语言环境下对于面向对象编程模式的一种模仿。这样之后，我们只需要关注于实现调度类的接口即可，操作系统也同样不关心调度类具体的实现，方便了新调度算法的开发。本次实验，主要是熟悉 ucore 的系统调度器框架，以及基于此框架的 Round-Robin (RR) 调度算法。然后参考 RR 调度算法的实现，完成 Stride 调度算法。

9.1.2.1 本节内容

9.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 `ucore_lab` 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

9.1.2.1.1.1 练习 0: 填写已有实验

本实验依赖实验 1/2/3/4/5。请把你做的实验 2/3/4/5 的代码填入本实验中代码中有 “LAB1” / “LAB2” / “LAB3” / “LAB4” “LAB5” 的注释相应部分。并确保编译通过。注意: 为了能够正确执行 lab6 的测试应用程序, 可能需对已完成的实验 1/2/3/4/5 的代码进行进一步改进。

9.1.2.1.1.2 练习 1: 使用 Round Robin 调度算法 (不需要编码)

完成练习 0 后, 建议大家比较一下 (可用 `kdiff3` 等文件比较软件) 个人完成的 lab5 和练习 0 完成后的刚修改的 lab6 之间的区别, 分析了解 lab6 采用 RR 调度算法后的执行过程。执行 `make grade`, 测试用例可以通过, 但没有得分。

请在实验报告中完成:

- 比较一个在 lab5 和 lab6 都有, 但是实现不同的函数, 说说为什么要做这个改动, 不做这个改动会出什么问题
 - 提示: 如 `kern/schedule/sched.c` 里的函数。你也可以找个其他地方做了改动的函数。
- 请理解并分析 `sched_class` 中各个函数指针的用法, 并描述 `ucore` 如何通过 Round Robin 算法来调度两个进程, 并解释 `sched_class` 里的每个函数 (函数指针) 是怎么被调用的。

9.1.2.1.1.3 练习 2: 实现 Stride Scheduling 调度算法 (需要编码)

首先需要换掉 RR 调度器的实现, 即用 `default_sched_stride_c` 覆盖 `default_sched.c`。然后根据此文件和后续文档对 Stride 度器的相关描述, 完成 Stride 调度算法的实现。注意有 “LAB6” 的注释, 主要是修改 `default_sched_swide_c` 中的内容。代码中所有需要完成的地方 (challenge 除外) 都有 “LAB6” 和 “YOUR CODE” 的注释, 请在提交时特别注意保持注释, 并将 “YOUR CODE” 替换为自己的学号, 并且将所有标有对应注释的部分填上正确的代码。

后面的实验文档部分给出了 Stride 调度算法的大体描述。这里给出 Stride 调度算法的一些相关的资料 (目前网上中文的资料比较欠缺)。

- [strid-shed paper location](#)
- 也可 GOOGLE “Stride Scheduling” 来查找相关资料

执行: `make grade`。如果所显示的应用程序检测都输出 ok, 则基本正确。执行 `make qemu` 大致的显示输出见附录。

- 请在实验报告中简要说明如何设计实现”多级反馈队列调度算法“；给出概要设计，鼓励给出详细设计
- 简要证明/说明（不必特别严谨，但应当能够”说服你自己“），为什么 Stride 算法中，经过足够多的时间片之后，每个进程分配到的时间片数目和优先级成正比。（最后两题二选一做即可）

请在实验报告中简要说明你的设计实现过程。

9.1.2.1.1.4 扩展练习 Challenge 1：实现 Linux 的 CFS 调度算法

在 ucore 的调度器框架下实现下 Linux 的 CFS 调度算法。可阅读相关 Linux 内核书籍或查询网上资料，可了解 CFS 的细节，然后大致实现在 ucore 中。

9.1.2.1.1.5 扩展练习 Challenge 2：在 ucore 上实现尽可能多的各种基本调度算法 (FIFO, SJF, ...)，并设计各种测试用例，能够定量地分析出各种调度算法在各种指标上的差异，说明调度算法的适用范围。

9.1.2.1.2 项目组成

```
lab6
├── Makefile
├── kern
│   ├── debug
│   │   ├── assert.h
│   │   ├── kdebug.c
│   │   ├── kdebug.h
│   │   ├── kmonitor.c
│   │   ├── kmonitor.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
│   │   └── picirq.h
│   ├── fs
│   │   ├── fs.h
│   │   ├── swapfs.c
│   │   └── swapfs.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   │   ├── readline.c
│   │   └── stdio.c
│   └── mm
│       ├── default_pmm.c
│       └── default_pmm.h
```

(续下页)

(接上页)

```

|   |— user.ld
|   |— vector.c
|— user
|   |— badarg.c
|   |— badsegment.c
|   |— divzero.c
|   |— exit.c
|   |— faultread.c
|   |— faultreadkernel.c
|   |— forktest.c
|   |— forktree.c
|   |— hello.c
|   |— libs
|   |   |— initcode.S
|   |   |— panic.c
|   |   |— stdio.c
|   |   |— syscall.c
|   |   |— syscall.h
|   |   |— ulib.c
|   |   |— ulib.h
|   |   |— umain.c
|   |— matrix.c
|   |— pgdir.c
|   |— priority.c
|   |— softint.c
|   |— spin.c
|   |— testbss.c
|   |— waitkill.c
|   |— yield.c

```

16 directories, 105 files

相对与实验五，实验六的主要改动简单说明如下：

- `libs/skew_heap.h`: 提供了基本的优先队列数据结构，为本次实验提供了抽象数据结构方面的支持。
- `kern/process/proc.[ch]`: `proc.h` 中扩展了 `proc_struct` 的成员变量，用于 RR 和 stride 调度算法。`proc.c` 中实现了 `lab6_set_priority`，用于设置进程的优先级。
- `kern/schedule/{sched.h,sched.c}`: 定义了 `ucore` 的调度器框架，其中包括相关的数据结构（包括调度器的接口和运行队列的结构），和具体的运行时机制。
- `kern/schedule/{default_sched.h,default_sched.c}`: 具体的 round-robin 算法，在本次实验中你需要了解其实现。
- `kern/schedule/default_sched_stride.c`: Stride Scheduling 调度器的基本框架，在此次实验中你需要填充其中的空白部分以实现一个完整的 Stride 调度器。
- `kern/syscall/syscall.[ch]`: 增加了 `sys_gettime` 系统调用，便于用户进程获取当前时钟值；增加了 `sys_lab6_set_priority` 系统调用，便于用户进程设置进程优先级（给 `priority.c` 用）
- `user/{matrix.c,priority.c,...`: 相关的一些测试用户程序，测试调度算法的正确性，`user` 目录下包含但不限于这些程序。在完成实验过程中，建议阅读这些测试程序，以了解这些程序的行为，便于进行调试。

9.1.3 调度框架和调度算法设计与实现

9.1.3.1 本节内容

9.1.3.1.1 进程状态

在 `ucore` 中, 进程有如下几个状态:

- `PROC_UNINIT`: 这个状态表示进程刚刚被分配相应的进程控制块, 但还没有初始化, 需要进一步的初始化才能进入 `PROC_RUNNABLE` 的状态。
- `PROC_SLEEPING`: 这个状态表示进程正在等待某个事件的发生, 通常由于等待锁的释放, 或者主动交出 CPU 资源 (`do_sleep`)。这个状态下的进程是不会被调度的。
- `PROC_RUNNABLE`: 这个状态表示进程已经准备好要执行了, 只需要操作系统给他分配相应的 CPU 资源就可以运行。
- `PROC_ZOMBIE`: 这个状态表示进程已经退出, 相应的资源被回收 (大部分), `almost dead`。

一个进程的生命周期一般由如下过程组成:

1. 刚刚开始初始化, 进程处在 `PROC_UNINIT` 的状态
2. 进程已经完成初始化, 时刻准备执行, 进入 `PROC_RUNNABLE` 状态
3. 在调度的时候, 调度器选中该进程进行执行, 进程处在 `running` 的状态
- 4.(1) 正在运行的进程由于 `wait` 等系统调用被阻塞, 进入 `PROC_SLEEPING`, 等待相应的资源或者信号。
- 4.(2) 另一种可能是正在运行的进程被外部中断打断, 此时进程变为 `PROC_RUNNABLE` 状态, 等待下次被调用
5. 等待的事件发生, 进程又变成 `PROC_RUNNABLE` 状态
6. 重复 3~6, 直到进程执行完毕, 通过 `exit` 进入 `PROC_ZOMBIE` 状态, 由父进程对他的资源进行回收, 释放进程控制块。至此, 这个进程的生命周期彻底结束

下面我们来看一看如何实现内核对于进程的调度。

9.1.3.1.2 深入理解进程切换

我们在第四章已经简单了解过了在内核启动过程中进程切换的过程, 在这一节我们再来重新回顾一下这些内容, 并且深入讨论下其中的几个细节。

首先我们需要考虑的是, 什么时候可以进行进程的切换。这里可以主要分为下面几种情况:

1. 进程主动放弃当前的 CPU 资源, 比如显式调用 `wait` 或 `sleep` 通知操作系统当前进程需要等待
2. 进程想要获取的资源当前不可用, 比如尝试获得未被释放的锁, 或进行磁盘操作的时候
3. 进程由于外部中断被打断进入内核态, 内核发现某些条件满足 (比如当前进程时间片用尽), 进行进程切换

在我们实现的 `ucore` 中, 内核进程是**不可抢占**的。这也就意味着当内核执行的时候, 另一个内核进程不可以夺走它的 CPU 资源。但这是不是就意味着一个内核进程执行的时候, 它就会一直执行到结束呢? 虽说内核进程不可以抢占, 但是它可以主动放弃自己占有的 CPU 资源。如果不这样设计的话, 内核当中很有可能出现各种死锁导致内核崩溃。

另一方面, 内核不能相信用户进程不会无限执行下去, 所以需要提供手段在用户进程执行的时候打断他, 比如时钟中断。在 `ucore` 的实现中, 用户进程可以随时被打断进入内核, 操作系统会检查当前进程是否需要调度, 从而把运行的机会交给别的进程。

由于内核进程是不可抢占的，所以我们在内核中有许多地方使用了显式的函数调用来进行调度，主要有以下几个地方：

函数	原因
proc.c/do_exit	用户进程退出，放弃 CPU 资源
proc.c/do_wait	用户进程等待，放弃 CPU 资源
proc.c/init_main	init 线程会等待所有的用户线程执行完毕，之后调用 kswapd 内核线程回收内存资源
proc.c/cpu_idle	idle 线程等待处于就绪态的线程，如果有就调用 schedule
sync.c/lock	如果获取锁失败就进入等待
trap.c/trap	用户进程在被中断打断后，内核会检查是否需要调度，如果是则调用调度器进行调度

当用户进程 A 发生中断或系统调用之后，首先其中断帧会被保存，CPU 进入内核态，执行中断处理函数。在执行完毕中断处理函数后，操作系统检查当前进程是否需要调度，如果需要，就把当前的进程状态保存，switch 到另一个进程 B 中。注意在执行上面的操作的时候，进程 A 处于内核态，类似的，调度后我们到达的是进程 B 的内核态。进程 B 从系统调用中返回，继续执行。如果进程 B 在中断或系统调用中被调度，控制权可能转交给进程 A 的内核态，这样进程 A 从内核态返回后就可以继续执行之前的代码了。

下一小节，我们来看一看 ucore 的调度器框架以及一些简单的调度算法。

9.1.3.1.3 调度算法框架

调度算法框架实现为一个结构体，其中保存了各个函数指针。通过实现这些函数指针即可实现各个调度算法。结构体的定义如下：

```
struct sched_class {
    // 调度类的名字
    const char *name;
    // 初始化run queue
    void (*init)(struct run_queue *rq);
    // 把进程放进run queue，这个是run queue的维护函数
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // 把进程取出run queue
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // 选择下一个要执行的进程
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // 每次时钟中断调用
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

所有的进程被组织成一个 run_queue 数据结构。这个数据结构虽然没有保存在调度类中，但是是由调度类来管理的。目前 ucore 仅支持单个 CPU 核心，所以只有一个全局的 run_queue。

我们在进程控制块中也记录了一些和调度有关的信息：

```
struct proc_struct {
    // ...
    // 表示这个进程是否需要调度
    volatile bool need_resched;
    // run queue的指针
    struct run_queue *rq;
    // 与这个进程相关的run queue表项

```

(续下页)

(接上页)

```
list_entry_t run_link;
// 这个进程剩下的时间片
int time_slice;
// 以下几个都和 Stride 调度算法实现有关
// 这个进程在优先队列中对应的项
skew_heap_entry_t lab6_run_pool;
// 该进程的 Stride 值
uint32_t lab6_stride;
// 该进程的优先级
uint32_t lab6_priority;
};
```

前面的几个成员变量的含义都比较直接，最后面的几个的含义可以参见 Stride 调度算法。这也是本次 lab 的实验内容。

结构体 run_queue 实现了运行队列，其内部结构如下：

```
struct run_queue {
    // 保存着链表头指针
    list_entry_t run_list;
    // 运行队列中的线程数
    unsigned int proc_num;
    // 最大的时间片大小
    int max_time_slice;
    // Stride 调度算法中的优先队列
    skew_heap_entry_t *lab6_run_pool;
};
```

有了这些基础，我们就来实现一个最简单的调度算法：Round-Robin 调度算法，也叫时间片轮转调度算法。

9.1.3.1.4 RR 调度算法实现

时间片轮转调度 (Round-Robin Scheduling) 算法非常简单。它为每一个进程维护了一个最大运行时间片。当一个进程运行够了其最大运行时间片那么长的时间后，调度器会把它标记为需要调度，并且把它的进程控制块放在队尾，重置其时间片。这种调度算法保证了公平性，每个进程都有均等的机会使用 CPU，但是没有区分不同进程的优先级（这个也就是在 Stride 算法中需要考虑的问题）。下面我们来实现以下时间片轮转算法相对应的调度器接口吧！

首先是 enqueue 操作。RR 算法直接把需要入队的进程放在调度队列的尾端，并且如果这个进程的剩余时间片为 0（刚刚用完时间片被收回 CPU），则需要把它的剩余时间片设为最大时间片。具体的实现如下：

```
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 ++;
}
```

dequeue 操作非常普通，将相应的项从队列中删除即可：

```
static void
RR_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 --;
}
```

pick_next 选取队列头的表项, 用 le2proc 函数获得对应的进程控制块, 返回:

```
static struct proc_struct *
RR_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;
}
```

proc_tick 函数在每一次时钟中断调用。在这里, 我们需要对当前正在运行的进程的剩余时间片减一。如果在减一后, 其剩余时间片为 0, 那么我们就把这个进程标记为“需要调度”, 这样在中断处理完之后内核判断进程是否需要调度的时候就会把它进行调度:

```
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;
    }
}
```

至此我们就实现完了和时间片轮转算法相关的所有重要接口。类似于 RR 算法, 我们也可以参照这个方法实现自己的调度算法。本次实验中需要同学们自己实现 Stride 调度算法。

9.1.3.1.5 stride 调度算法

9.1.3.1.5.1 本小节内容

9.1.3.1.5.2 基本思路

【提示】 请先看练习 2 中提到的论文, 理解后在看下面的内容。

考察 round-robin 调度器, 在假设所有进程都充分使用了其拥有的 CPU 时间资源的情况下, 所有进程得到的 CPU 时间应该是相等的。但是有时候我们希望调度器能够更智能地为每个进程分配合理的 CPU 资源。假设我们为不同的进程分配不同的优先级, 则我们有可能希望每个进程得到的时间资源与他们的优先级成正比关系。Stride 调度是基于这种想法的一个较为典型和简单的算法。除了简单易于实现以外, 它还有如下的特点:

- 可控性: 如我们之前所希望的, 可以证明 Stride Scheduling 对进程的调度次数正比于其优先级。
- 确定性: 在不考虑计时器事件的情况下, 整个调度机制都是可预知和重现的。该算法的基本思想可以考虑如下:
 1. 为每个 runnable 的进程设置一个当前状态 stride, 表示该进程当前的调度权。另外定义其对应的 pass 值, 表示对应进程在调度后, stride 需要进行的累加值。
 2. 每次需要调度时, 从当前 runnable 态的进程中选择 stride 最小的进程调度。

3. 对于获得调度的进程 P, 将对应的 stride 加上其对应的步长 pass (只与进程的优先权有关系)。
4. 在一段固定的时间之后, 回到 2. 步骤, 重新调度当前 stride 最小的进程。
可以证明, 如果令 $P.\text{pass} = \text{BigStride} / P.\text{priority}$ 其中 $P.\text{priority}$ 表示进程的优先权 (大于 1), 而 BigStride 表示一个预先定义的大常数, 则该调度方案为每个进程分配的时间将与其优先级成正比。证明过程我们在这里略去, 有兴趣的同学可以在网上查找相关资料。将该调度器应用到 ucore 的调度器框架中来, 则需要将调度器接口实现如下:

- init:
 - 初始化调度器类的信息 (如果有的话)。
 - 初始化当前的运行队列为一个空的容器结构。(比如和 RR 调度算法一样, 初始化为一个有序列表)
- enqueue
 - 初始化刚进入运行队列的进程 proc 的 stride 属性。
 - 将 proc 插入放入运行队列中去 (注意: 这里并不要求放置在队列头部)。
- dequeue
 - 从运行队列中删除相应的元素。
- pick next
 - 扫描整个运行队列, 返回其中 stride 值最小的对应进程。
 - 更新对应进程的 stride 值, 即 $\text{pass} = \text{BIG_STRIDE} / P \rightarrow \text{priority}$; $P \rightarrow \text{stride} += \text{pass}$ 。
- proc tick:
 - 检测当前进程是否已用完分配的时间片。如果时间片用完, 应该正确设置进程结构的相关标记来引起进程切换。
 - 一个 process 最多可以连续运行 rq.max_time_slice 个时间片。

在具体实现时, 有一个需要注意的地方: stride 属性的溢出问题, 在之前的实现里面我们并没有考虑 stride 的数值范围, 而这个值在理论上是不断增加的, 在 stride 溢出以后, 基于 stride 的比较可能会出现错误。比如假设当前存在两个进程 A 和 B, stride 属性采用 16 位无符号整数进行存储。当前队列中元素如下 (假设当前运行的进程已经被重新放置进运行队列中):

A.stride (实际值)	A.stride (理论值)	A.pass (= $\frac{\text{BigStride}}{A.\text{priority}}$)
65534	65534	100
B.stride (实际值)	B.stride (理论值)	B.pass (= $\frac{\text{BigStride}}{B.\text{priority}}$)
65535	65535	50

此时应该选择 A 作为调度的进程, 而在一轮调度后, 队列将如下:

A.stride (实际值)	A.stride (理论值)	A.pass (= $\frac{\text{BigStride}}{A.\text{priority}}$)
98	65634	100
B.stride (实际值)	B.stride (理论值)	B.pass (= $\frac{\text{BigStride}}{B.\text{priority}}$)
65535	65535	50

可以看到由于溢出的出现, 进程间 stride 的理论比较和实际比较结果出现了偏差。我们首先在理论上分析这个问题: 令 PASS_MAX 为当前所有进程里最大的步进值。则我们可以证明如下结论: 对每次 Stride 调度器的调度步骤中, 有其最大的步进值 STRIDE_MAX 和最小的步进值 STRIDE_MIN 之差:

$$\text{STRIDE_MAX} - \text{STRIDE_MIN} \leq \text{PASS_MAX}$$

提问 1: 如何证明该结论?

有了该结论, 在加上之前对优先级有 $\text{Priority} > 1$ 限制, 我们有 $\text{STRIDE_MAX} - \text{STRIDE_MIN} \leq \text{BIG_STRIDE}$, 于是我们只要将 **BigStride** 取在某个范围之内, 即可保证对于任意两个 **Stride** 之差都会在机器整数表示的范围之内。而我们可以通过与 0 的比较结构, 来得到两个 **Stride** 的大小关系。在上例中, 虽然在直接的数值表示上 $98 < 65535$, 但是 $98 - 65535$ 的结果用带符号的 16 位整数表示的结果为 99, 与理论值之差相等。所以在这个意义下 $98 > 65535$ 。基于这种特殊考虑的比较方法, 即便 **Stride** 有可能溢出, 我们仍能够得到理论上的当前最小 **Stride**, 并做出正确的调度决定。

提问 2: 在 **ucore** 中, 目前 **Stride** 是采用无符号的 32 位整数表示。则 **BigStride** 应该取多少, 才能保证比较的正确性?

9.1.3.1.5.3 使用优先队列实现 Stride Scheduling

在上述的实现描述中, 对于每一次 **pick_next** 函数, 我们都需要完整地扫描来获得当前最小的 **stride** 及其进程。这在进程非常多的时候是非常耗时和低效的, 有兴趣的同学可以在实现了基于列表扫描的 **Stride** 调度器之后比较一下 **priority** 程序在 **Round-Robin** 及 **Stride** 调度器下各自的运行时间。考虑到其调度选择于优先队列的抽象逻辑一致, 我们考虑使用优化的优先队列数据结构实现该调度。

优先队列是这样一种数据结构: 使用者可以快速的插入和删除队列中的元素, 并且在预先指定的顺序下快速取得当前在队列中的最小 (或者最大) 值及其对应元素。可以看到, 这样的数据结构非常符合 **Stride** 调度器的实现。

本次实验提供了 **libs/skew_heap.h** 作为优先队列的一个实现, 该实现定义相关的结构和接口, 其中主要包括:

```

1 // 优先队列节点的结构
2 typedef struct skew_heap_entry skew_heap_entry_t;
3 // 初始化一个队列节点
4 void skew_heap_init(skew_heap_entry_t *a);
5 // 将节点 b 插入至以节点 a 为队列头的队列中去, 返回插入后的队列
6 skew_heap_entry_t *skew_heap_insert(skew_heap_entry_t *a,
7                                     skew_heap_entry_t *b,
8                                     compare_f comp);
9 // 将节点 b 插入从以节点 a 为队列头的队列中去, 返回删除后的队列
10 skew_heap_entry_t *skew_heap_remove(skew_heap_entry_t *a,
11                                     skew_heap_entry_t *b,
12                                     compare_f comp);

```

其中优先队列的顺序是由比较函数 **comp** 决定的, **sched_stride.c** 中提供了 **proc_stride_comp_f** 比较器用来比较两个 **stride** 的大小, 你可以直接使用它。当使用优先队列作为 **Stride** 调度器的实现方式之后, 运行队列结构也需要作相关改变, 其中包括:

- **struct run_queue** 中的 **lab6_run_pool** 指针, 在使用优先队列的实现中表示当前优先队列的头元素, 如果优先队列为空, 则其指向空指针 (**NULL**)。
- **struct proc_struct** 中的 **lab6_run_pool** 结构, 表示当前进程对应的优先队列节点。本次实验已经修改了系统相关部分的代码, 使得其能够很好地适应 **LAB6** 新加入的数据结构和接口。而在实验中我们需要做的是用优先队列实现一个正确和高效的 **Stride** 调度器, 如果用较简略的伪代码描述, 则有:
 - **init(rq):**
 - Initialize **rq->run_list**
 - Set **rq->lab6_run_pool** to **NULL**
 - Set **rq->proc_num** to 0
 - **enqueue(rq, proc)**
 - Initialize **proc->time_slice**
 - Insert **proc->lab6_run_pool** into **rq->lab6_run_pool**

- rq->proc_num ++
- dequeue(rq, proc)
 - Remove proc->lab6_run_pool from rq->lab6_run_pool
 - rq->proc_num -
- pick_next(rq)
 - If rq->lab6_run_pool == NULL, return NULL
 - Find the proc corresponding to the pointer rq->lab6_run_pool
 - proc->lab6_stride += BIG_STRIDE / proc->lab6_priority
 - Return proc
- proc_tick(rq, proc):
 - If proc->time_slice > 0, proc->time_slice -
 - If proc->time_slice == 0, set the flag proc->need_resched

大家可以根据上述伪代码，完成本次实验练习 2。

9.1.4 附录：执行 make qemu 的大致输出

```
$ make qemu
.....
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
Breakpoint
set priority to 6
main: fork ok,now need to wait pids.
set priority to 5
set priority to 4
set priority to 3
set priority to 2
set priority to 1
child pid 7, acc 944000, time 2010
child pid 6, acc 788000, time 2010
child pid 5, acc 620000, time 2010
child pid 4, acc 460000, time 2020
child pid 3, acc 316000, time 2020
main: pid 3, acc 316000, time 2020
main: pid 4, acc 460000, time 2020
main: pid 5, acc 620000, time 2030
main: pid 6, acc 788000, time 2030
main: pid 0, acc 944000, time 2030
main: wait pids over
stride sched correctresult:1 1 2 2 3
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process proc.c:468:
initproc exit.
```

10.1 本章内容

10.1.1 实验目的

- 理解操作系统的同步互斥的设计实现；
- 理解底层支撑技术：禁用中断、定时器、等待队列；
- 在 ucore 中理解信号量（semaphore）机制的具体实现；
- 理解管程机制，在 ucore 内核中增加基于管程（monitor）的条件变量（condition variable）的支持；
- 了解经典进程同步问题，并能使用同步机制解决进程同步问题。

10.1.2 实验内容

在本章中我们来实现 ucore 中的同步互斥机制。

在之前的几章中我们已经实现了进程以及调度算法，可以让多个进程并发的执行。在现实的系统当中，有许多多线程的系统都需要协同的完成某一项任务。但是在协同的过程中，存在许多资源共享的问题，比如对一个文件读写的并发访问等等。这些问题需要我们提供一些同步互斥的机制来让程序可以有序的、无冲突的完成他们的工作，这也是这一章内我们要解决的问题。

我们最终实现的目标是解决“哲学家就餐问题”。“哲学家就餐问题”是一个非常有名的同步互斥问题：有五个哲学家围成一圈吃饭，每两个哲学家中间有一根筷子。每个需要就餐的哲学家需要两根筷子才可以就餐。哲学家处于两种状态之间：思考和饥饿。当哲学家处于思考的状态时，哲学家便无欲无求；而当哲学家处于饥饿状态时，他必须通过就餐来解决饥饿，重新回到思考的状态。如何让这 5 个哲学家可以不发生死锁的把这一顿饭吃完就是我们要解决的目标。

下面，我们先从一些同步互斥实现的机制讲起，慢慢的了解 ucore 中同步互斥机制的实现，最后解决哲学家就餐问题吧！.. _ 本节内容：

10.1.2.1 本节内容

10.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 ucore_lab 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

10.1.2.1.1.1 练习 0: 填写已有实验

本实验依赖实验 1/2/3/4/5/6。请把你做的实验 1/2/3/4/5/6 的代码填入本实验中代码中有 “LAB1” / “LAB2” / “LAB3” / “LAB4” / “LAB5” / “LAB6” 的注释相应部分。并确保编译通过。注意: 为了能够正确执行 lab7 的测试应用程序, 可能需对已完成的实验 1/2/3/4/5/6 的代码进行进一步改进。

10.1.2.1.1.2 练习 1: 理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题 (不需要编码)

完成练习 0 后, 建议大家比较一下 (可用 meld 等文件 diff 比较软件) 个人完成的 lab6 和练习 0 完成后的刚修改的 lab7 之间的区别, 分析了解 lab7 采用信号量的执行过程。执行 `make grade`, 测试用例可以通过, 但没有全部得分。

- 请在实验报告中给出内核级信号量的设计描述, 并说明其大致执行流程。
- 证明/说明为什么我们给出的信号量实现的哲学家问题不会出现死锁。
- 请在实验报告中给出给用户态进程/线程提供信号量机制的设计方案, 并比较说明给内核级提供信号量机制的异同。

10.1.2.1.1.3 练习 2: 完成内核级条件变量和基于内核级条件变量的哲学家就餐问题 (需要编码)

首先掌握管程机制, 然后基于信号量实现完成条件变量实现, 然后用管程机制实现哲学家就餐问题的解决方案 (基于条件变量)。

执行: `make grade`。如果所显示的应用程序检测都输出 `ok`, 则基本正确。执行 `make qemu` 的大致输出见附录。

- 请在实验报告中给出内核级条件变量的设计描述, 并说明其大致执行流程。
- 请在实验报告中给出给用户态进程/线程提供条件变量机制的设计方案, 并比较说明给内核级提供条件变量机制的异同。
- 请在实验报告中回答: 能否不用基于信号量机制来完成条件变量? 如果不能, 请给出理由, 如果能, 请给出设计说明和具体实现。

10.1.2.1.1.4 扩展练习 Challenge 1 : 在 ucore 中实现简化的死锁和重入探测机制

在 ucore 下实现一种探测机制, 能够在多进程/线程运行同步互斥问题时, 动态判断当前系统是否出现了死锁产生的必要条件, 是否产生了多个进程进入临界区的情况。如果发现, 让系统进入 monitor 状态, 打印出你的探测信息。

10.1.2.1.1.5 扩展练习 Challenge 2 : 参考 Linux 的 RCU 机制, 在 ucore 中实现简化的 RCU 机制

在 ucore 下实现下 Linux 的 RCU 同步互斥机制。可阅读相关 Linux 内核书籍或查询网上资料, 可了解 RCU 的设计实现细节, 然后简化实现在 ucore 中。要求有实验报告说明你的设计思路, 并提供测试用例。下面是一些参考资料:

- <http://www.ibm.com/developerworks/cn/linux/l-rcu/>
- http://www.diybl.com/course/6_system/linux/Linuxjs/20081117/151814.html

10.1.2.1.2 项目组成

```
lab7
├── Makefile
├── kern
│   ├── debug
│   │   ├── assert.h
│   │   ├── kdebug.c
│   │   ├── kdebug.h
│   │   ├── kmonitor.c
│   │   ├── kmonitor.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
│   │   └── picirq.h
│   ├── fs
│   │   ├── fs.h
│   │   ├── swapfs.c
│   │   └── swapfs.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   │   ├── readline.c
│   │   └── stdio.c
│   └── mm
│       ├── default_pmm.c
│       └── default_pmm.h
```

(续下页)

(接上页)

```

├── kmalloc.c
├── kmalloc.h
├── memlayout.h
├── mmu.h
├── pmm.c
├── pmm.h
├── swap.c
├── swap.h
├── swap_fifo.c
├── swap_fifo.h
├── vmm.c
├── vmm.h
├── process
│   ├── entry.S
│   ├── proc.c
│   ├── proc.h
│   └── switch.S
├── schedule
│   ├── default_sched.h
│   ├── default_sched_c.c
│   ├── default_sched_stride
│   ├── sched.c
│   └── sched.h
├── sync
│   ├── check_sync.c
│   ├── monitor.c
│   ├── monitor.h
│   ├── sem.c
│   ├── sem.h
│   ├── sync.h
│   ├── wait.c
│   └── wait.h
├── syscall
│   ├── syscall.c
│   └── syscall.h
├── trap
│   ├── trap.c
│   ├── trap.h
│   └── trapentry.S
├── lab5.md
├── libs
│   ├── atomic.h
│   ├── defs.h
│   ├── elf.h
│   ├── error.h
│   ├── hash.c
│   ├── list.h
│   ├── printfmt.c
│   ├── rand.c
│   ├── riscv.h
│   ├── sbi.h
│   ├── skew_heap.h
│   ├── stdarg.h
│   ├── stdio.h
│   ├── stdlib.h
│   ├── string.c
│   └── string.h

```

(续下页)

(接上页)

```

├── unistd.h
├── tools
│   ├── boot.ld
│   ├── function.mk
│   ├── gdbinit
│   ├── grade.sh
│   ├── kernel.ld
│   ├── sign.c
│   ├── user.ld
│   └── vector.c
├── user
│   ├── badarg.c
│   ├── badsegment.c
│   ├── divzero.c
│   ├── exit.c
│   ├── faultread.c
│   ├── faultreadkernel.c
│   ├── forktest.c
│   ├── forktree.c
│   ├── hello.c
│   ├── libs
│   │   ├── initcode.S
│   │   ├── panic.c
│   │   ├── stdio.c
│   │   ├── syscall.c
│   │   ├── syscall.h
│   │   ├── ulib.c
│   │   ├── ulib.h
│   │   └── umain.c
│   ├── matrix.c
│   ├── pgdir.c
│   ├── priority.c
│   ├── sleep.c
│   ├── sleepkill.c
│   ├── softint.c
│   ├── spin.c
│   ├── testbss.c
│   ├── waitkill.c
│   └── yield.c

```

16 directories, 115 files

简单说明如下：

- kern/schedule/{sched.h,sched.c}: 增加了定时器 (timer) 机制, 用于进程/线程的 do_sleep 功能。
- kern/sync/sync.h: 去除了 lock 实现 (这对于不抢占内核没用)。
- kern/sync/wait.[ch]: 定义了等待队列 wait_queue 结构和等待 entry 的 wait 结构以及在此之上的函数, 这是 ucore 中的信号量 semaphore 机制和条件变量机制的基础, 在本次实验中你需要了解其实现。
- kern/sync/sem.[ch]: 定义并实现了 ucore 中内核级信号量相关的数据结构和函数, 本次试验中你需要了解其中的实现, 并基于此完成内核级条件变量的设计与实现。
- user/ libs/ {syscall.[ch],ulib.[ch]} 与 kern/sync/syscall.c: 实现了进程 sleep 相关的系统调用的参数传递和调用关系。
- user/{ sleep.c,sleepkill.c}: 进程睡眠相关的一些测试用户程序。
- kern/sync/monitor.[ch]: 基于管程的条件变量的实现程序, 在本次实验中是练习的一部分, 要求完成。

- kern/sync/check_sync.c: 实现了基于管程的哲学家就餐问题, 在本次实验中是练习的一部分, 要求完成基于管程的哲学家就餐问题。
- kern/mm/vmm.[ch]: 用信号量 mm_sem 取代 mm_struct 中原有的 mm_lock。(本次实验不用管)

10.1.3 同步互斥的一些基本概念

虽然我们经常把同步和互斥放在一起说, 但是这两个词是两个概念。**同步**指的是进程间的执行需要按照某种先后顺序, 即访问是有序的。**互斥**指的是对于某些共享资源的访问不能同时进行, 同一时间只能有一定数量的进程进行。这两种情况基本构成了我们在多线程执行中遇到的各种问题。

与同步互斥相关的另一个概念是**临界区**。临界区指的是进程的一段代码, 其特征要求了同一时间段只能有一个进程执行, 否则就有可能出现问题。一般而言, 进程处理临界区的思路是设计一个协议, 不同的进程遵守这个相同的协议来进行临界区的协调。在进入临界区前, 进程请求进入的许可, 这段代码称为**进入区**; 退出临界区时, 进程应该通过协议告知别的进程自己已经使用完临界区, 这段代码称为**退出区**; 临界区其他的部分称为**剩余区**。我们本章解决的问题, 就是在进入区和退出区为进程提供同步互斥的机制。

为了提供同步互斥机制, 操作系统有多种实现方法, 包括时钟中断管理, 屏蔽使能中断, 等待队列, 信号量, 管程等等。下面我们来分别看一看上面提到的部分机制:

10.1.3.1 时钟中断管理

在 lab1 中我们已经实现了时钟中断。在 ucore 中, 时钟 (timer) 中断给操作系统提供了有一定间隔的时间事件, 操作系统将其作为基本的调度和计时单位 (我们记两次时间中断之间的时间间隔为一个时间片, timer splice)。

通过时钟中断, 操作系统可以提供基于时间节点的事件。通过时钟中断, 操作系统可以提供任意长度的等待唤醒机制, 由此可以给应用程序机会来实现更加复杂的自定义调度操作。

- sched.h, sched.c 定义了有关 timer 的各种相关接口来使用 timer 服务, 其中主要包括:
- typedef struct {……} timer_t: 定义了 timer_t 的基本结构, 其可以用 sched.h 中的 timer_init 函数对其进行初始化。
- void timer_init(timer_t *timer, struct proc_struct *proc, int expires): 对某 timer 进行初始化, 让它在 expires 时间片之后唤醒 proc 进程。
- void add_timer(timer_t *timer): 向系统添加某个初始化过的 timer_t, 该 timer 在指定时间后被激活, 并将对应的进程唤醒至 runnable (如果当前进程处在等待状态)。
- void del_timer(timer_t *time): 向系统删除 (或者说取消) 某一个 timer。该 timer 在取消后不会被系统激活并唤醒进程。
- void run_timer_list(void): 更新当前系统时间点, 遍历当前所有处在系统管理内的 timer, 找出所有应该激活的计数器, 并激活它们。该过程在且只在每次时钟中断时被调用。在 ucore 中, 其还会调用调度器事件处理程序。

一个 timer_t 在系统中的存活周期可以被描述如下:

1. timer_t 在某个位置被创建和初始化, 并通过 add_timer 加入系统管理列表中。
2. 系统时间被不断累加, 直到 run_timer_list 发现该 timer_t 到期。
3. run_timer_list 更改对应的进程状态, 并从系统管理列表中移除该 timer_t。

尽管本次实验并不需要填充时钟相关的代码, 但是作为系统重要的组件, 你应该了解其相关机制和在 ucore 中的实现方法和使用方法。且在 trap_dispatch 函数中修改之前对时钟中断的处理, 使得 ucore 能够利用时钟提供的功能完成调度和睡眠唤醒等操作。

10.1.3.2 屏蔽使能中断

这部分主要是处理内核内的同步互斥问题。因为内核在执行的过程中可能会被外部的中断打断，我们实现的 `ucore` 也是不可抢占的系统，所以可以在操作系统进行某些需要同步互斥的操作的时候先禁用中断，等执行完之后再使能。这样保证了操作系统在执行临界区的时候不会被打断，也就实现了同步互斥。

根据操作系统原理的知识，我们知道如果没有在硬件级保证读内存-修改值-写回内存的原子性，我们只能通过复杂的软件来实现同步互斥操作。但由于有开关中断和 `test_and_set_bit` 等原子操作机器指令的存在，使得我们在实现同步互斥原语上可以大大简化。

在 `ucore` 中提供的底层机制包括中断屏蔽/使能控制等。`kern/sync.c` 中实现的开关中断的控制函数 `local_intr_save(x)` 和 `local_intr_restore(x)`，它们是基于 `kern/driver` 文件下的 `intr_enable()`、`intr_disable()` 函数实现的。具体调用关系为：

```
关中断：local_intr_save --> __intr_save --> intr_disable --> cli
开中断：local_intr_restore--> __intr_restore --> intr_enable --> sti
```

最终的 `cli` 和 `sti` 是 x86 的机器指令，最终实现了关（屏蔽）中断和开（使能）中断，即设置了 `eflags` 寄存器中与中断相关的位。通过关闭中断，可以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断，那也就意味着在内核运行的当前进程无法被打断或被重新调度，即实现了对临界区的互斥操作。所以在单处理器情况下，可以通过开关中断实现对临界区的互斥保护，需要互斥的临界区代码的一般写法为：

```
local_intr_save(intr_flag);
{
    临界区代码
}
local_intr_restore(intr_flag);
... ..
```

由于目前 `ucore` 只实现了对单处理器的支持，所以通过这种方式，就可简单地支撑互斥操作了。在多处理器情况下，这种方法是无法实现互斥的，因为屏蔽了一个 CPU 的中断，只能阻止本地 CPU 上的进程不会被中断或调度，并不意味着其他 CPU 上执行的进程不能执行临界区的代码。所以，开关中断只对单处理器下的互斥操作起作用。在本实验中，开关中断机制是实现信号量等高层同步互斥原语的底层支撑基础之一。

10.1.3.3 等待队列：

等待队列是操作系统提供了一种事件机制。一些进程可能会在执行的过程中等待某些特定事件的发生，这个时候进程进入睡眠状态。操作系统维护一个等待队列，把这个进程放进他等待的事件的等待队列中。当对应的事件发生之后，操作系统就唤醒相应等待队列中的进程。这也是 `ucore` 内部实现信号量的机制。

内核实现这一功能的一个底层支撑机制就是等待队列 `wait_queue`，等待队列和每一个事件（睡眠结束、时钟到达、任务完成、资源可用等）联系起来。需要等待事件的进程在转入休眠状态后插入到等待队列中。当事件发生之后，内核遍历相应等待队列，唤醒休眠的用户进程或内核线程，并设置其状态为就绪状态（`PROC_RUNNABLE`），并将该进程从等待队列中清除。`ucore` 在 `kern/sync/{ wait.h, wait.c}` 中实现了等待项 `wait` 结构和等待队列 `wait_queue` 结构以及相关函数，这是实现 `ucore` 中的信号量机制和条件变量机制的基础，进入 `wait queue` 的进程会被设为等待状态（`PROC_SLEEPING`），直到他们被唤醒。

数据结构定义

```
typedef struct {
    struct proc_struct *proc;           //等待进程的指针
    uint32_t wakeup_flags;              //进程被放入等待队列的原因标记
    wait_queue_t *wait_queue;           //指向此wait结构所属于的wait_queue
    list_entry_t wait_link;             //用来组织wait_queue中wait节点的连接
} wait_t;
```

(续下页)

(接上页)

```
typedef struct {
    list_entry_t wait_head;          //wait_queue的队头
} wait_queue_t;

le2wait(le, member)                //实现wait_t中成员的指针向wait_t 指针的转化
```

相关函数说明与 wait 和 wait queue 相关的函数主要分为两层，底层函数是对 wait queue 的初始化、插入、删除和查找操作，相关函数如下：

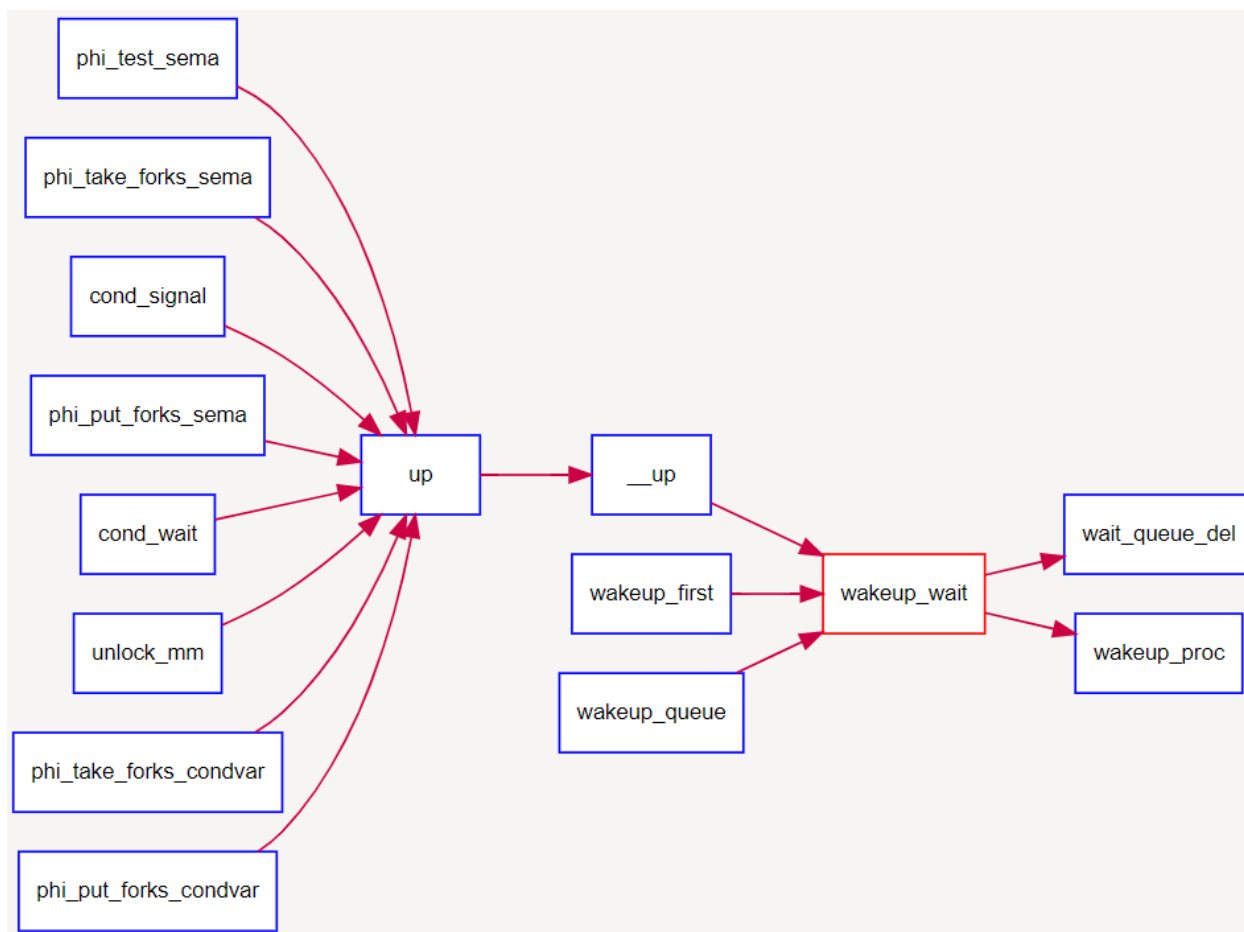
```
void wait_init(wait_t *wait, struct proc_struct *proc);    //初始化wait结构
bool wait_in_queue(wait_t *wait);                          //wait是否在wait queue中
void wait_queue_init(wait_queue_t *queue);                 //初始化wait_queue结构
void wait_queue_add(wait_queue_t *queue, wait_t *wait);    //把wait前插到wait queue中
void wait_queue_del(wait_queue_t *queue, wait_t *wait);    //从wait queue中删除wait
wait_t *wait_queue_next(wait_queue_t *queue, wait_t *wait); //取得wait的后一个链接指针
wait_t *wait_queue_prev(wait_queue_t *queue, wait_t *wait); //取得wait的前一个链接指针
wait_t *wait_queue_first(wait_queue_t *queue);              //取得wait queue的第一个wait
↪ t
wait_t *wait_queue_last(wait_queue_t *queue);               //取得wait queue的最后一个wait
↪ ait
bool wait_queue_empty(wait_queue_t *queue);                //wait queue是否为空
```

高层函数基于底层函数实现了让进程进入等待队列 wait_current_set，以及从等待队列中唤醒进程 wakeup_wait，相关函数如下：

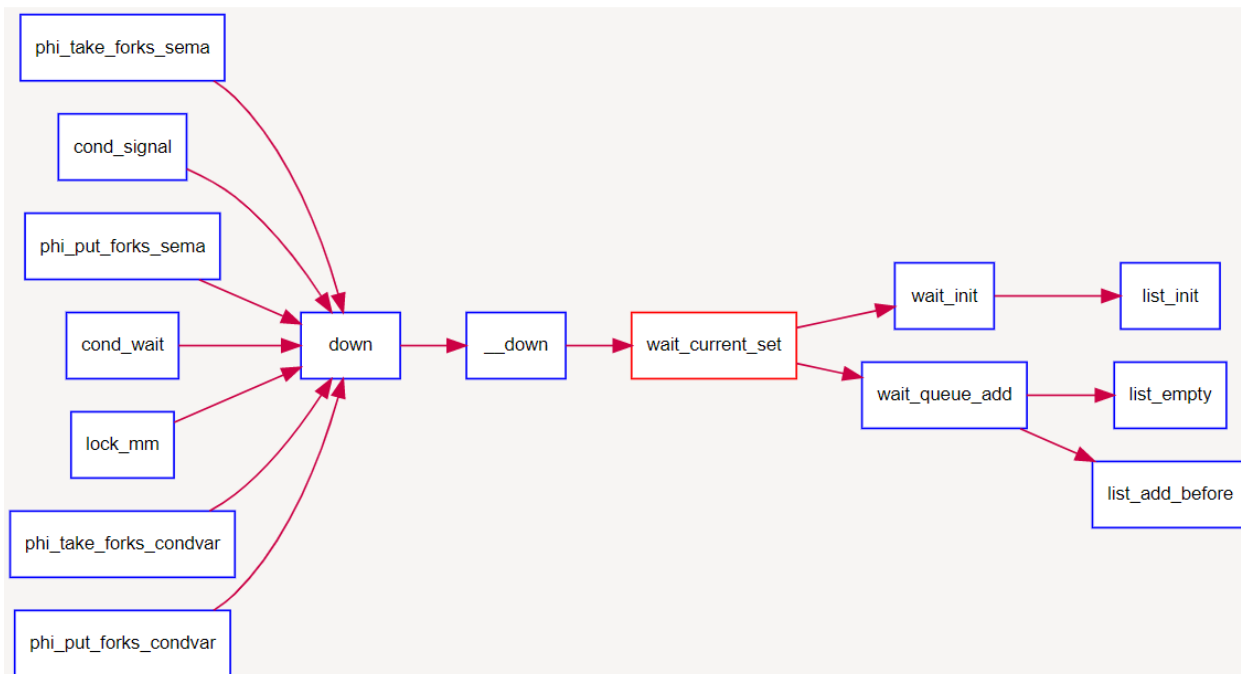
```
//让wait与进程关联，且让当前进程关联的wait进入等待队列queue，当前进程睡眠
void wait_current_set(wait_queue_t *queue, wait_t *wait, uint32_t wait_state);
//把与当前进程关联的wait从等待队列queue中删除
wait_current_del(queue, wait);
//唤醒与wait关联的进程
void wakeup_wait(wait_queue_t *queue, wait_t *wait, uint32_t wakeup_flags, bool del);
//唤醒等待队列上挂着的第一个wait所关联的进程
void wakeup_first(wait_queue_t *queue, uint32_t wakeup_flags, bool del);
//唤醒等待队列上所有的等待的进程
void wakeup_queue(wait_queue_t *queue, uint32_t wakeup_flags, bool del);
```

10.1.3.3.1 调用关系举例

如下图所示，对于唤醒进程的函数 wakeup_wait，可以看到它会被各种信号量的 V 操作函数 up 调用，并且它会调用 wait_queue_del 函数和 wakeup_proc 函数来完成唤醒进程的操作。



如下图所示，而对于让进程进入等待状态的函数 `wait_current_set`，可以看到它会被各种信号量的 P 操作函数 `down` 调用，并且它会调用 `wait_init` 完成对等待项的初始化，并进一步调用 `wait_queue_add` 来把与要处于等待状态的进程所关联的等待项挂到与信号量绑定的等待队列中。



接下来的两节，我们来仔细看一看信号量和管程的实现。

10.1.4 信号量

信号量 (semaphore) 是一种同步互斥的实现。semaphore 一词来源于荷兰语，原来是指火车信号灯。想象一些火车要进站，火车站里有 n 个站台，那么同一时间只能有 n 辆火车进站装卸货物。当火车站里已经有了 n 辆火车，信号灯应该通知后面的火车不能进站了。当有火车出站之后，信号灯应该告诉后面的火车可以进站。

这个问题放在操作系统的语境下就是有一个共享资源只能支持 n 个线程并行的访问，信号量统计目前有多少进程正在访问，当同时访问的进程数小于 n 时就可以让新的线程进入，当同时访问的进程数为 n 时想要访问的进程就需要等待。

在信号量中，一般用两种操作来刻画申请资源和释放资源：P 操作申请一份资源，如果申请不到则等待；V 操作释放一份资源，如果此时有进程正在等待，则唤醒该进程。在 ucore 中，我们使用 down 函数实现 P 操作，up 函数实现 V 操作。

首先是信号量结构体的定义：

```
typedef struct {
    int value;
    wait_queue_t wait_queue;
} semaphore_t;
```

其中的 value 表示信号量的值，其正值表示当前可用的资源数量，负值表示正在等待资源的进程数量。wait_queue 即为这个信号量相对应的等待队列。

down 函数实现的是 P 操作。首先关闭中断，然后判断信号量的值是否为正，如果是正值说明进程可以获得信号量，将信号量的值减一，打开中断然后函数返回即可。否则表示无法获取信号量，将自己的进程保存进等待队列，打开中断，调用 schedule 函数进行调度。等到 V 操作唤醒进程的时候，其会回到调用 schedule 函数后面，将自身从等待队列中删除（此过程需要关闭中断）并返回即可。具体的实现如下：

```
static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
```

(续下页)

(接上页)

```

local_intr_save(intr_flag);
if (sem->value > 0) {
    sem->value --;
    local_intr_restore(intr_flag);
    return 0;
}
wait_t __wait, *wait = &__wait;
wait_current_set(&(sem->wait_queue), wait, wait_state);
local_intr_restore(intr_flag);

schedule();

local_intr_save(intr_flag);
wait_current_del(&(sem->wait_queue), wait);
local_intr_restore(intr_flag);

if (wait->wakeup_flags != wait_state) {
    return wait->wakeup_flags;
}
return 0;
}

```

up 函数实现了 V 操作。首先关闭中断，如果释放的信号量没有进程正在等待，那么将信号量的值加一，打开中断直接返回即可。如果有进程正在等待，那么唤醒这个进程，把它放进就绪队列，打开中断并返回。具体的实现如下：

```

static __noinline void __up(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        wait_t *wait;
        if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) {
            sem->value ++;
        }
        else {
            assert(wait->proc->wait_state == wait_state);
            wakeup_wait(&(sem->wait_queue), wait, wait_state, 1);
        }
    }
    local_intr_restore(intr_flag);
}

```

下一节，我们来看看如何使用信号量实现条件变量以及管程。

10.1.5 管程

我们已经有了基本的同步互斥机制实现，下面让我们用这些机制来实现一个管程，从而解决哲学家就餐问题。

为什么要使用管程呢？引入管程相当于将底层的同步互斥机制封装了起来，对外提供已经经过同步的接口供进程使用，大大降低了并行进程开发的门槛。管程主要由四个部分组成：

- 管程内部的共享变量
- 管程内部的条件变量
- 管程内部并发执行的进程

- 对局部于管程内部的共享数据设置初始值的语句

由此可见，管程把需要互斥访问的变量直接包装了起来，对共享变量的访问只能通过管程提供的相应接口，方便了多进程的编写。但是管程只有同步互斥是不够的，可能需要条件变量。条件变量类似于信号量，只不过在信号量中进程等待某一个资源可用，而条件变量中进程等待条件变量相应的资源为真。条件变量的结构体如下：

```
typedef struct condvar{
    // 信号量
    semaphore_t sem;
    // 正在等待的线程数
    int count;
    // 自己属于哪一个管程
    monitor_t * owner;
} condvar_t;
```

我们主要需要实现两个函数：wait 函数，等待某一个条件；signal 函数，提醒某一个条件已经达成。具体实现比较简单，可以参考代码如下：

```
// wait
cv.count++;
if(monitor.next_count > 0)
    sem_signal(monitor.next);
else
    sem_signal(monitor.mutex);
sem_wait(cv.sem);
cv.count -- ;
```

```
// signal
if( cv.count > 0) {
    monitor.next_count ++;
    sem_signal(cv.sem);
    sem_wait(monitor.next);
    monitor.next_count -- ;
}
```

管程的内部实现如下所示：

```
typedef struct monitor{
    // 保证管程互斥访问的信号量
    semaphore_t mutex;
    // 里面放着正在等待进入管程执行的进程
    semaphore_t next;
    // 正在等待进入管程的进程数
    int next_count;
    // 条件变量
    condvar_t *cv;
} monitor_t;
```

条件变量 cv 被设置时，会使得当前在管程内的进程等待条件变量而睡眠，其他进程进入管程执行。当 cv 被唤醒的时候，之前等待这个条件变量的进程也会被唤醒，进入管程执行。由于管程内部只能由一个条件变量，所以通过设置 next 来维护下一个要运行的进程是哪一个。

使用了管程，我们的哲学家就餐问题可以被实现为如下：

```
monitor dp
{
    enum {THINKING, HUNGRY, EATING} state[5];
```

(续下页)

(接上页)

```

condition self[5];

void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        self[i].wait_cv();
}

void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
}

void test(int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
        state[i] = EATING;
        self[i].signal_cv();
    }
}

initialization code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
}

```

具体的实现可以在 `sync/check_sync.c` 中找到。需要注意的是, 上述只是原理描述, 与具体描述相比, 还有一定的差距。需要大家在完成练习时仔细设计和实现。

10.1.6 附录: 执行 `make qemu` 的大致输出

```

$ make qemu
.....
(THU.CST) os is loading ...
... ..
check_alloc_page() succeeded!
... ..
check_swap() succeeded!
++ setup timer interrupts
I am No.4 philosopher_sema
Iter 1, No.4 philosopher_sema is thinking
I am No.2 philosopher_sema
Iter 1, No.2 philosopher_sema is thinking
... ..
Iter 1, No.0 philosopher_sema is eating
Iter 1, No.2 philosopher_sema is eating
Iter 2, No.2 philosopher_sema is thinking
... ..
I am No.3 philosopher_condvar
Iter 1, No.3 philosopher_condvar is thinking
... ..

```

(续下页)

(接上页)

```
phi_test_condvar: state_condvar[0] will eating
phi_test_condvar: signal self_cv[0]
cond_signal begin: cvp c0443430, cvp->count 0, cvp->owner->next_count 0
cond_signal end: cvp c0443430, cvp->count 0, cvp->owner->next_count 0
Iter 1, No.0 philosopher_condvar is eating
cond_wait begin: cvp c0443458, cvp->count 0, cvp->owner->next_count 0
... ..
No.4 philosopher_condvar quit
No.1 philosopher_condvar quit
No.3 philosopher_condvar quit
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:426:
  initproc exit.
```

文件最早来自于计算机用户需要把数据持久保存在持久存储设备上的需求。由于放在内存中的数据在计算机关机或掉电后就会消失，所以应用程序要把内存中需要保存的数据放到持久存储设备的数据块（比如磁盘的扇区等）中存起来。随着操作系统功能的增强，在操作系统的管理下，应用程序不用理解持久存储设备的硬件细节，而只需对文件这种持久存储数据的抽象进行读写就可以了，由操作系统中的文件系统和存储设备驱动程序一起来完成繁琐的持久存储设备的管理与读写。

11.1 本章内容

11.1.1 实验目的

通过完成本次实验，希望能够达到以下目标

- 了解文件系统抽象层-VFS 的设计与实现
- 了解基于索引节点组织方式的 Simple FS 文件系统与操作的设计与实现
- 了解“一切皆为文件”思想的设备文件设计
- 了解简单系统终端的实现

11.1.2 实验内容

实验七完成了在内核中的同步互斥实验。本次实验涉及的是文件系统，通过分析了解 ucore 文件系统的总体架构设计，完善读写文件操作 (即实现 `sfs_io_nolock()` 函数)，从新实现基于文件系统的执行程序机制（即实现 `load_icode()` 函数），从而可以完成执行存储在磁盘上的文件和实现文件读写等功能。

与实验七相比，实验八增加了文件系统，并因此实现了通过文件系统来加载可执行文件到内存中运行的功能，导致对进程管理相关的实现比较大的调整。

11.1.2.1 本节内容

11.1.2.1.1 练习

对实验报告的要求:

- 基于 markdown 格式来完成, 以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后, 请分析 `ucore_lab` 中提供的参考答案, 并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点, 以及与对应的 OS 原理中的知识点, 并简要说明你对二者的含义, 关系, 差异等方面的理解 (也可能出现实验中的知识点没有对应的原理知识点)
- 列出你认为 OS 原理中很重要, 但在实验中没有对应上的知识点

11.1.2.1.1.1 练习 0: 填写已有实验

本实验依赖实验 1/2/3/4/5/6/7。请把你做的实验 1/2/3/4/5/6/7 的代码填入本实验中代码中有 “LAB1” / “LAB2” / “LAB3” / “LAB4” / “LAB5” / “LAB6” / “LAB7” 的注释相应部分。并确保编译通过。注意: 为了能够正确执行 lab8 的测试应用程序, 可能需对已完成的实验 1/2/3/4/5/6/7 的代码进行进一步改进。

11.1.2.1.1.2 练习 1: 完成读文件操作的实现 (需要编码)

首先了解打开文件的处理流程, 然后参考本实验后续的文件读写操作的过程分析, 填写在 `kern/fs/sfs/sfs_inode.c` 中的 `sfs_io_nolock()` 函数, 实现读文件中数据的代码。

11.1.2.1.1.3 练习 2: 完成基于文件系统的执行程序机制的实现 (需要编码)

改写 `proc.c` 中的 `load_icode` 函数和其他相关函数, 实现基于文件系统的执行程序机制。执行: `make qemu`。如果能看看到 `sh` 用户程序的执行界面, 则基本成功了。如果在 `sh` 用户界面上可以执行 “ls”, “hello” 等其他放置在 `sfs` 文件系统其他执行程序, 则可以认为本实验基本成功。

11.1.2.1.1.4 扩展练习 Challenge1: 完成基于 “UNIX 的 PIPE 机制” 的设计方案

如果要在 `ucore` 里加入 UNIX 的管道 (Pipe) 机制, 至少需要定义哪些数据结构和接口? (接口给出语义即可, 不必具体实现。数据结构的设计应当给出一个 (或多个) 具体的 C 语言 `struct` 定义。在网络上查找相关的 Linux 资料 and 实现, 请在实验报告中给出设计实现” UNIX 的 PIPE 机制 “的概要设方案, 你的设计应当体现出对可能出现的同步互斥问题的处理。)

11.1.2.1.1.5 扩展练习 Challenge2: 完成基于 “UNIX 的软连接和硬连接机制” 的设计方案

如果要在 `ucore` 里加入 UNIX 的软连接和硬连接机制, 至少需要定义哪些数据结构和接口? (接口给出语义即可, 不必具体实现。数据结构的设计应当给出一个 (或多个) 具体的 C 语言 `struct` 定义。在网络上查找相关的 Linux 资料 and 实现, 请在实验报告中给出设计实现” UNIX 的软连接和硬连接机制 “的概要设方案, 你的设计应当体现出对可能出现的同步互斥问题的处理。)

11.1.2.1.2 项目组成

11.1.2.1.2.1 Lab8 项目组成

```

lab8
├── Makefile
├── disk0
│   ├── badarg
│   ├── badsegment
│   ├── divzero
│   ├── exit
│   ├── faultread
│   ├── faultreadkernel
│   ├── forktest
│   ├── forktree
│   ├── hello
│   ├── matrix
│   ├── pgdir
│   ├── priority
│   ├── sh
│   ├── sleep
│   ├── sleepkill
│   ├── softint
│   ├── spin
│   ├── testbss
│   ├── waitkill
│   └── yield
├── giveitatri.pyq
├── kern
│   ├── debug
│   │   ├── assert.h
│   │   ├── kdebug.c
│   │   ├── kdebug.h
│   │   ├── kmonitor.c
│   │   ├── kmonitor.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
│   │   ├── picirq.h
│   │   ├── ramdisk.c
│   │   └── ramdisk.h
│   └── fs
│       ├── devs
│       │   ├── dev.c
│       │   ├── dev.h
│       │   ├── dev_disk0.c
│       │   └── dev_stdin.c

```

(续下页)

(接上页)

```

├── dev_stdout.c
├── file.c
├── file.h
├── fs.c
├── fs.h
├── iobuf.c
├── iobuf.h
├── sfs
│   ├── bitmap.c
│   ├── bitmap.h
│   ├── sfs.c
│   ├── sfs.h
│   ├── sfs_fs.c
│   ├── sfs_inode.c
│   ├── sfs_io.c
│   └── sfs_lock.c
├── swap
│   ├── swapfs.c
│   └── swapfs.h
├── sysfile.c
├── sysfile.h
├── vfs
│   ├── inode.c
│   ├── inode.h
│   ├── vfs.c
│   ├── vfs.h
│   ├── vfsdev.c
│   ├── vfsfile.c
│   ├── vfslookup.c
│   └── vfspath.c
├── init
│   ├── entry.S
│   └── init.c
├── libs
│   ├── readline.c
│   ├── stdio.c
│   └── string.c
├── mm
│   ├── default_pmm.c
│   ├── default_pmm.h
│   ├── kmalloc.c
│   ├── kmalloc.h
│   ├── memlayout.h
│   ├── mmu.h
│   ├── pmm.c
│   ├── pmm.h
│   ├── swap.c
│   ├── swap.h
│   ├── swap_fifo.c
│   ├── swap_fifo.h
│   ├── vmm.c
│   └── vmm.h
├── process
│   ├── entry.S
│   ├── proc.c
│   ├── proc.h
│   └── switch.S

```

(续下页)

(接上页)

```

├── schedule
│   ├── default_sched.h
│   ├── default_sched.c
│   ├── default_sched_stride.c
│   ├── sched.c
│   └── sched.h
├── sync
│   ├── check_sync.c
│   ├── monitor.c
│   ├── monitor.h
│   ├── sem.c
│   ├── sem.h
│   ├── sync.h
│   ├── wait.c
│   └── wait.h
├── syscall
│   ├── syscall.c
│   └── syscall.h
├── trap
│   ├── trap.c
│   ├── trap.h
│   └── trapentry.S
├── lab5.md
├── libs
│   ├── atomic.h
│   ├── defs.h
│   ├── dirent.h
│   ├── elf.h
│   ├── error.h
│   ├── hash.c
│   ├── list.h
│   ├── printfmt.c
│   ├── rand.c
│   ├── riscv.h
│   ├── sbi.h
│   ├── skew_heap.h
│   ├── stat.h
│   ├── stdarg.h
│   ├── stdio.h
│   ├── stdlib.h
│   ├── string.c
│   ├── string.h
│   └── unistd.h
├── tools
│   ├── boot.ld
│   ├── function.mk
│   ├── gdbinit
│   ├── grade.sh
│   ├── kernel.ld
│   ├── mksfs.c
│   ├── sign.c
│   ├── user.ld
│   └── vector.c
├── user
│   ├── badarg.c
│   ├── badsegment.c
│   └── divzero.c

```

(续下页)

(接上页)

```

├── exit.c
├── faultread.c
├── faultreadkernel.c
├── forktest.c
├── forktree.c
├── hello.c
├── libs
│   ├── dir.c
│   ├── dir.h
│   ├── file.c
│   ├── file.h
│   ├── initcode.S
│   ├── lock.h
│   ├── panic.c
│   ├── stdio.c
│   ├── syscall.c
│   ├── syscall.h
│   ├── ulib.c
│   ├── ulib.h
│   └── umain.c
├── matrix.c
├── pgdir.c
├── priority.c
├── sh.c
├── sleep.c
├── sleepkill.c
├── softint.c
├── spin.c
├── testbss.c
├── waitkill.c
└── yield.c

```

21 directories, 176 files

本次实验主要是理解 `kern/fs` 目录中的部分文件，并可用 `user/*.c` 测试所实现的 Simple FS 文件系统是否能够正常工作。本次实验涉及到的代码包括：

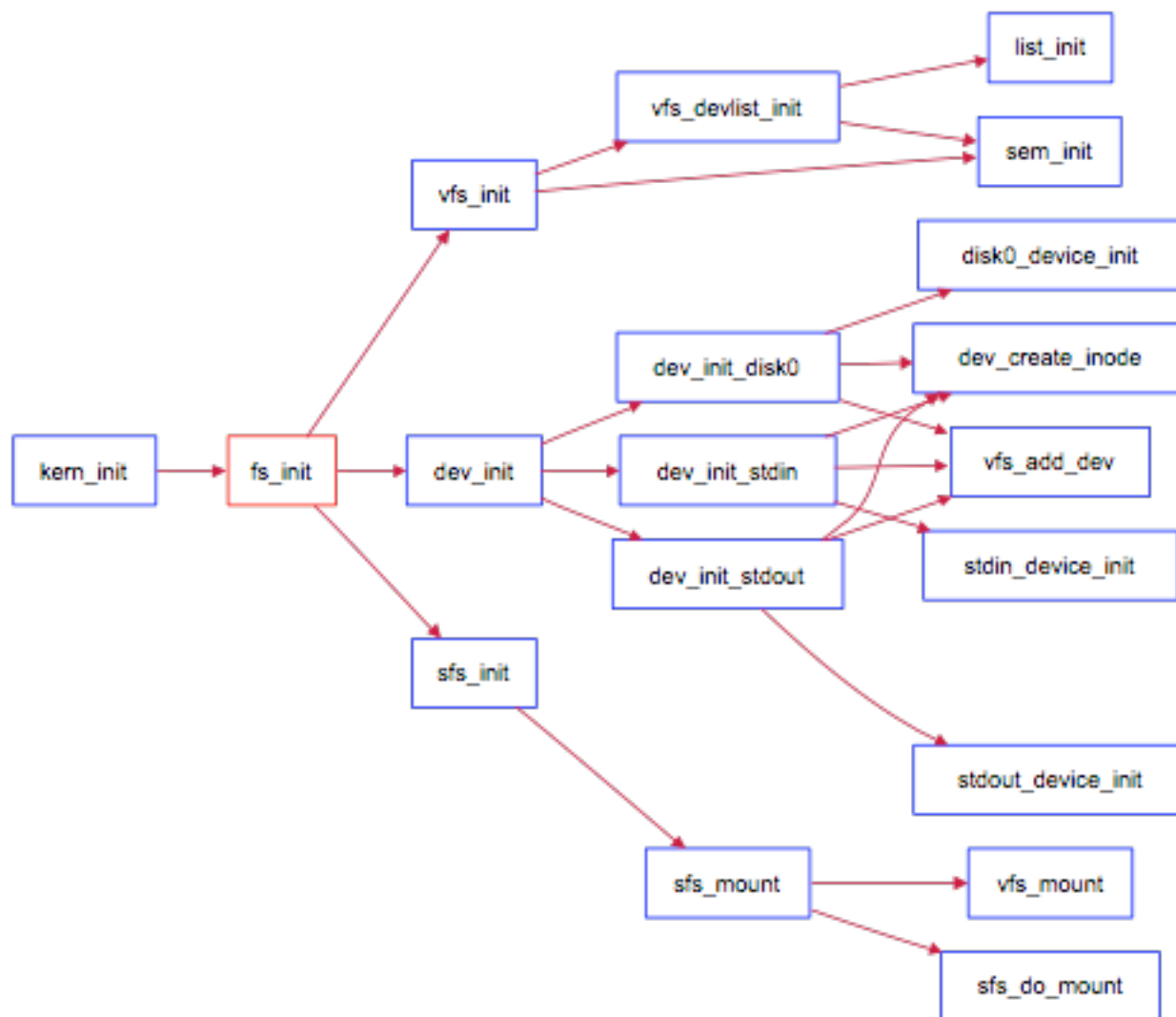
- 文件系统测试用例：`user/*.c`：对文件系统的实现进行测试的测试用例；
- 通用文件系统接口
 - n `user/libs/file.[ch]`|`dir.[ch]`|`syscall.c`：与文件系统操作相关的用户库实行；
 - n `kern/syscall.[ch]`：文件中包含文件系统相关的内核态系统调用接口
 - n `kern/fs/sysfile.[ch]`|`file.[ch]`：通用文件系统接口和实行
- 文件系统抽象层-VFS
 - n `kern/fs/vfs/*.ch`：虚拟文件系统接口与实现
- Simple FS 文件系统
 - n `kern/fs/sfs/*.ch`：SimpleFS 文件系统实现
- 文件系统的硬盘 IO 接口
 - n `kern/fs/devs/dev.[ch]`|`dev_disk0.c`：disk0 硬盘设备提供给文件系统的 I/O 访问接口和实现
- 辅助工具
 - n `tools/mksfs.c`：创建一个 Simple FS 文件系统格式的硬盘镜像。（理解此文件的实现细节对理解 SFS 文件系统很有帮助）
- 对内核其它模块的扩充
 - n `kern/process/proc.[ch]`：增加成员变量 `struct fs_struct *fs_struct`，用于支持进程对文件的访问；重写了

do_execve load_icode 等函数以支持执行文件系统中的文件。
 n kern/init/init.c: 增加调用初始化文件系统的函数 fs_init。

11.1.2.1.2.2 Lab8 文件系统初始化过程

与实验七相比，实验八增加了文件系统，并因此实现了通过文件系统来加载可执行文件到内存中运行的功能，导致对进程管理相关的实现比较大的调整。我们来简单看看文件系统是如何初始化并能在 ucore 的管理下正常工作的。

首先看看 kern_init 函数，可以发现与 lab7 相比增加了对 fs_init 函数的调用。fs_init 函数就是文件系统初始化的总控函数，它进一步调用了虚拟文件系统初始化函数 vfs_init，与文件相关的设备初始化函数 dev_init 和 Simple FS 文件系统的初始化函数 sfs_init。这三个初始化函数联合在一起，协同完成了整个虚拟文件系统、SFS 文件系统和文件系统对应的设备（键盘、串口、磁盘）的初始化工作。其函数调用关系图如下所示：



文件系统初始化调用关系图

参考上图，并结合源码分析，可大致了解到文件系统的整个初始化流程。vfs_init 主要建立了一个 device list 双向链表 vdev_list，为后续具体设备（键盘、串口、磁盘）以文件的形式呈现建立查找访问通道。dev_init 函数通过进一步调用 disk0/stdin/stdout_device_init 完成对具体设备的初始化，把它们抽象成一个设备文件，并建立对应的 inode 数据结构，最后把它们链入到 vdev_list 中。这样通过虚拟文件系统就可以方便地以文件的

形式访问这些设备了。sfs_init 是完成对 Simple FS 的初始化工作, 并把此实例文件系统挂在虚拟文件系统中, 从而让 ucore 的其他部分能够通过访问虚拟文件系统的接口来进一步访问到 SFS 实例文件系统。

11.1.3 文件系统

我们来到了最后一个 lab。文件系统 (file system), 指的是操作系统中管理 (硬盘上) 持久存储数据的模块。

11.1.4 文件系统概述

11.1.4.1 为啥需要文件/文件系统?

能够”持久存储数据的设备”, 可能包括: 机械硬盘 (HDD, hard disk drive), 固态硬盘 (solid-state storage device), 光盘 (加上它的驱动器), 软盘, U 盘, 甚至磁带或纸带等等。一般来说, 每个设备上都会连出很多针脚 (这些针脚很可能符合某个特定协议, 如 USB 协议, SATA 协议), 它们都可以按照事先约定的接口/协议把数据从设备中读到内存里, 或者把内存里的数据按照一定的格式储存到设备里。

我们希望做到的第一件事情, 就是把以上种种存储设备当作“同样的设备”进行使用。不管机械硬盘的扇区有多少个, 或者一块固态硬盘里有多少存储单元, 我们希望能用同样的接口进行使用, 提供“把硬盘的第 a 到第 b 个字节读出来”和“把内存里这些内容写到硬盘的从第 a 个字节开始的位置”的接口, 在使用时只会感受到不同设备速度的不同。否则, 我们还需要自己处理什么 SATA 协议, NVMe 协议啥的。处理具体设备, 具体协议并向上提供简单接口的软件, 我们叫做**设备驱动 (device driver)**, 简称**驱动**。

文件的概念在我们脑子里根深蒂固, 但“文件”其实也是一种抽象。理论上, 只要提供了上面提到的读写两个接口, 我们就可以进行编程, 而并不需要什么文件的概念。只要你自己在小本本上记住, 你的某些数据存储在硬盘上从某个地址开始的多么长的位置, 以及硬盘上哪些位置是没有被占用的。编程的时候, 如果用/修改硬盘上的数据, 就把小本本上记录的位置给硬编码到程序里。

但这很不灵活! 如果你的小本本丢了, 你就再也无法使用硬盘里的数据了, 因为你不知道那些数据都是谁跟谁。另外, 你也可能一不小心修改到无关的数据。最后, 这个小本本经常需要修改, 你很容易出错。

显然, 我们应该把这个小本本交给计算机来保存和自动维护。这就是**文件系统**。

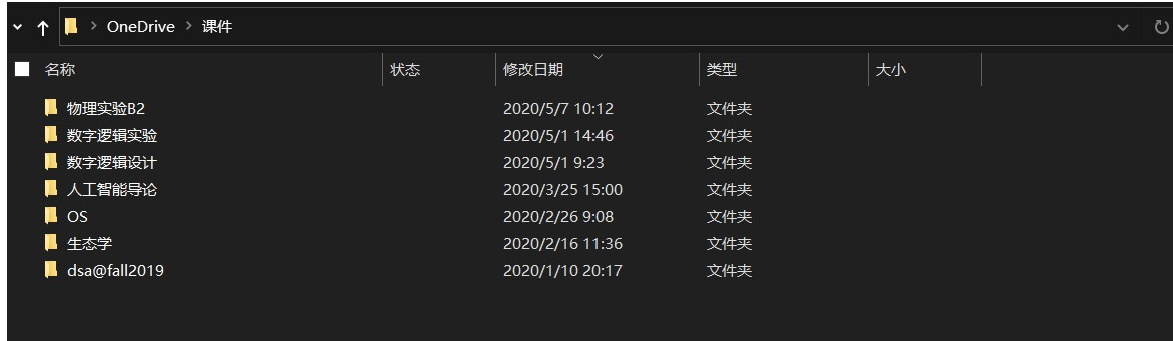
我们把一段逻辑上相关联的数据看作一个整体, 叫做**文件**。

除了硬盘, 我们还可以把其他设备 (如标准输入, 标准输出) 看成是文件, 由文件系统进行统一的管理。之前我们模拟过一个很简单的文件系统, 用来存放内存置换出的页面 (参见 lab3 页面置换)。现在我们要来实现功能更强大也更复杂的文件系统。

11.1.4.2 虚拟文件系统

我们可以实现一个**虚拟文件系统 (virtual filesystem, VFS)**, 作为操作系统和更具体的文件系统之间的接口。所谓“具体文件系统”, 更接近具体设备和文件系统的内部实现, 而“虚拟文件系统”更接近用户使用的接口。

1. 电脑上本地的硬盘和远程的云盘 (例如清华云盘, OneDrive) 的具体文件管理方式很可能不同, 但是操作系统可以让我们用相同的操作访问这两个地方的文件, 可以认为用到了虚拟文件系统。



例如, 远程的云盘 OneDrive, 在 Windows 资源管理器 (可以看作是虚拟文件系统提供给用户的接口) 里, 看起来和本地磁盘上的文件夹一模一样, 也可以做和本地的文件夹相同的操作, 复制, 粘贴, 打开。虽然背后有着网络传输, 可能速度会慢, 但是接口一致。(理论上可以在 OneDrive 的远程存储服务器上使用 linux 操作系统)。

2. 在 Linux 系统中, 如 /floppy 是一块 MS-DOS 文件系统的软盘的挂载点, /tmp/test 是 Ext2 文件系统的目录, 我们执行 `cp /floppy/TEST /tmp/test`, 进行目录的拷贝, 相当于执行下面这段代码:

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

对于不同文件系统的目录, 我们可以使用相同的 open, read, write, close 接口, 好像它们在同一个文件系统里一样。这是虚拟文件系统的功能。

扩展

最早的虚拟文件系统: Sun Microsystems' s SunOS , 1986

linux 的 VFS 支持虚拟块设备 (virtual block devices), 可以把当前文件系统中的某个文件, 认为是“一块具备虚拟文件系统的磁盘”进行挂载, 也就是说这个文件里包含一个完整的文件系统, 如同“果壳里的宇宙”。

这种设备也叫做 “loop device”, 可以用来简易地在现有文件系统上进行磁盘分区。

扩展

When a filesystem is mounted on a directory, the contents of the directory in the parent filesystem are no longer accessible, because every pathname, including the mount point, will refer to the mounted filesystem. However, the original directory' s content shows up again when the filesystem is unmounted. This somewhat surprising feature of Unix filesystems is used by system administrators to hide files; they simply mount a filesystem on the directory containing the files to be hidden.

《understanding linux kernel》

11.1.4.3 UNIX 文件系统

ucore 的文件系统模型和传统的 UNIX 文件系统类似。

UNIX 文件中的内容可理解为是一段有序的字节, 占用磁盘上可能连续或不连续的一些空间 (实际占用的空间可能比你存储的数据要多)。每个文件都有一个方便应用程序识别的文件名 (也可以称作路径 `path`), 另外有一个文件系统内部使用的编号 (用户程序不知道这个底层编号)。你可以对着一个文件又读又写 (写太多的时候可能会自动给文件分配更多的硬盘存储空间), 也可以创建或删除文件。

**** 目录 (directory)**** 是特殊的文件, 一个目录里包含若干其他文件或目录。

在 UNIX 中, 文件系统可以被安装在一个特定的文件路径位置, 这个位置就是**挂载点 (mount point)**。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。比如当你把 U 盘插进来, 系统检测到 U 盘之后, 会给 U 盘的文件系统一个挂载点, 这个挂载点是原先的 UNIX 操作系统的叶子, 但也可以认为是 U 盘文件系统的根节点。

UNIX 的文件系统中, 有一个**通用文件模型 (Common File Model)**, 所有具体的文件系统 (不管是 Ext4, ZFS 还是 FAT), 都需要提供通用文件模型所约定的行为。我们可以认为, 通用文件模型是面向对象的, 由若干对象 (Object) 组成, 每个对象有成员属性和函数接口。类 UNIX 系统的内核一般使用 C 语言实现, 成员函数 “一般体现为函数指针”。

通用文件模型定义了一些对象:

- **超级块 (superblock)**: 存储整个文件系统的相关信息。对于磁盘上的文件系统, 对应磁盘里的**文件系统控制块 (filesystem control block)**
- **索引节点 (inode)**: 存储关于某个文件的元数据信息 (如访问控制权限、大小、拥有者、创建时间、数据内容等等), 通常对应磁盘上的**文件控制块 (file control block)**。每个索引节点有一个编号, 唯一确定文件系统里的一个文件。
- **文件 (file)**: 这里定义的 `file object` 不是指磁盘上的一个 “文件”, 而是指一个进程和它打开的一个文件之间的关系, 这个对象存储在内存态的内存中, 仅当某个进程打开某个文件的时候才存在。
- **目录项 (dentry)**: 维护从 “目录里的某一项” 到 “对应的文件” 的链接/指针。一个目录也是一个文件, 包含若干个子目录和其他文件。从某个子目录、文件的名称, 对应到具体的文件/子目录的地址 (或者索引节点 `inode`) 的链接, 通过 **** 目录项 (dentry)**** 来描述。

上述抽象概念形成了 UNIX 文件系统的逻辑数据结构, 并需要通过一个具体文件系统的架构, 把上述信息映射并存储到磁盘介质上, 从而在具体文件系统的磁盘布局 (即数据在磁盘上的物理组织) 上体现出上述抽象概念。比如文件元数据信息存储在磁盘块中的索引节点上。当文件被载入内存时, 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。又比如 `dentry` 对象在磁盘上不存在, 但是当目录包含的某一项 (可能是子目录或文件) 的信息被载入到内存时, 内核会构建对应的 `dentry` 对象, 如 `/tmp/test` 这个路径, 在解析的过程中, 内核为根目录/创建一个 `dentry` 对象, 为根目录的成员 `tmp` 构建一个 `dentry` 对象, 为 `/tmp` 目录的成员 `test` 也构建一个 `dentry` 对象。

11.1.4.4 ucore 文件系统总体介绍

我们将在 ucore 里用虚拟文件系统管理三类设备:

- 硬盘, 我们管理硬盘的具体文件系统是 Simple File System (地位和 Ext2 等文件系统相同)
- 标准输出 (控制台输出), 只能写不能读
- 标准输入 (键盘输入), 只能读不能写

其中, 标准输入和标准输出都是比较简单的设备。管理硬盘的 Simple File System 相对而言比较复杂。

我们的 “硬盘” 依然需要通过用一块内存来模拟。

lab8 的 Makefile 和之前不同, 我们分三段构建内核镜像。

- `sfs.img`: 一块符合 SFS 文件系统的硬盘, 里面存储编译好的用户程序

- swap.img: 一段初始化为 0 的硬盘交换区
- kernel objects: ucore 内核代码的目标文件

这三部分共同组成 ucore.img, 加载到 QEMU 里运行。ucore 代码中, 我们通过链接时添加的首尾符号, 把 swap.img 和 sfs.img 两段“硬盘”(实际上对应两段内存空间)找出来, 然后作为“硬盘”进行管理。

注意, 我们要在 ucore 内核开始执行之前, 构造好“一块符合 SFS 文件系统的硬盘”, 这就得另外写个程序做这个事情。这个程序就是 tools/mksfs.c。它有 500 多行, 如果感兴趣的话可以通过它了解 Simple File System 的结构。

ucore 模仿了 UNIX 的文件系统设计, 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。
- 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。

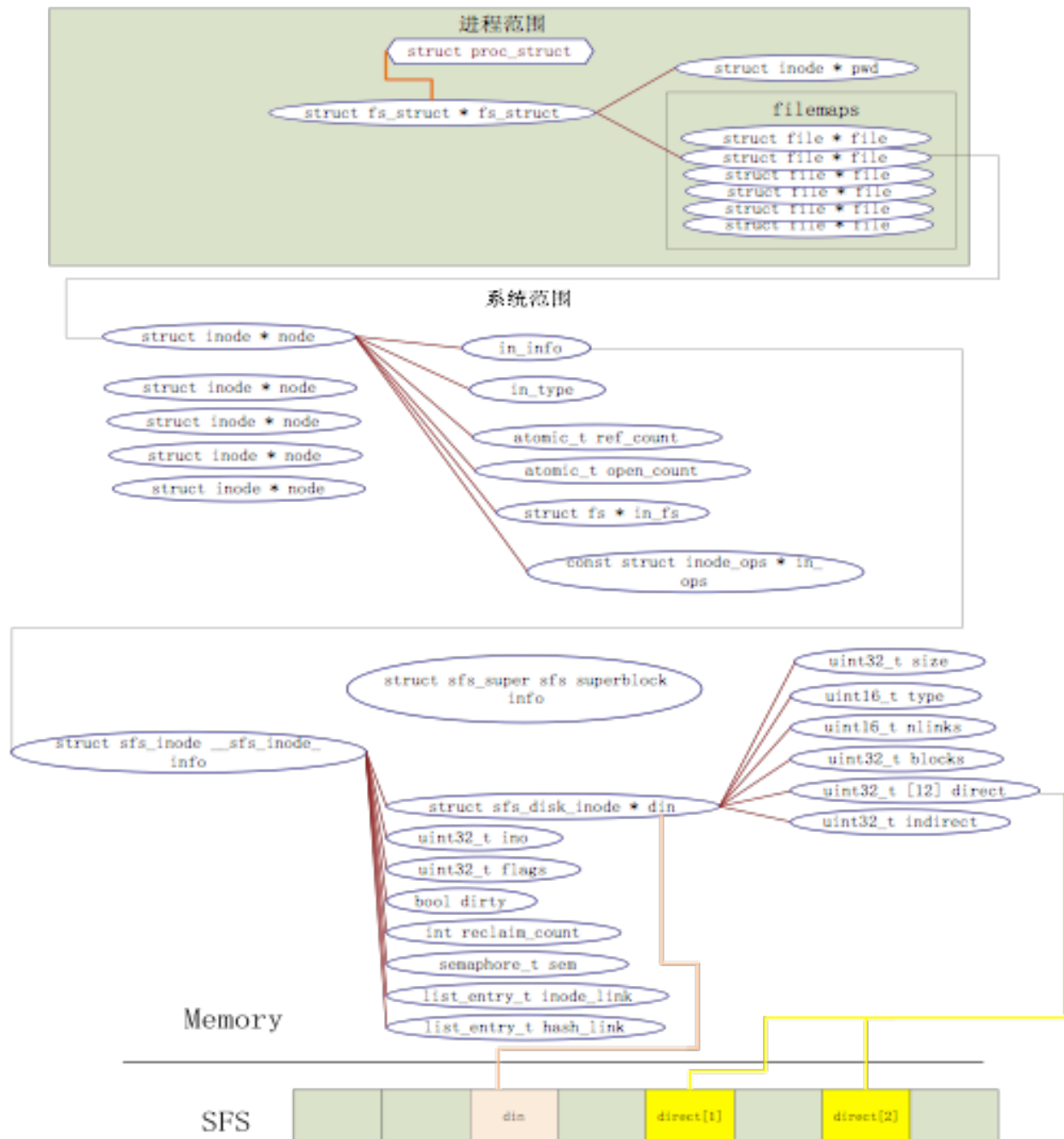


11.1.4.5 ucore 文件系统总体结构

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): 它主要从文件的文件路径的角度描述了文件路径中的一个特定的目录项 (注: 一系列目录项形成目录/文件路径)。它的作用范围是整个 OS 空间。对于 SFS 而言, inode(具体为 struct sfs_disk_inode) 对应于物理磁盘上的具体对象, dentry (具体为 struct sfs_disk_entry) 是一个内存实体, 其中的 ino 成员指向对应的 inode number, 另外一个成员是 file name(文件名)。
- 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。

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和关系如下图所示:



11.1.5 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。

11.1.5.1 file & dir 接口

file&dir 接口层定义了进程在内核中直接访问的文件相关信息, 这定义在 file 数据结构中, 具体描述如下:

```
// kern/fs/file.h
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
    int open_count;      //打开此文件的次数
};
```

而在 kern/process/proc.h 中的 proc_struct 结构中加入了描述了进程访问文件的数据接口 files_struct, 其数据结构定义如下:

```
// kern/fs/fs.h
struct files_struct {
    struct inode *pwd;           //进程当前执行目录的内存inode指针
    struct file *fd_array;      //进程打开文件的数组
    atomic_t files_count;       //访问此文件的线程个数
    semaphore_t files_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 files_struct 将会被初始化或复制父进程的 files_struct。当用户进程打开一个文件时, 将从 fd_array 数组中取得一个空闲 file 项, 然后把此 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。

11.1.5.2 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:

```
// kern/vfs/inode.h
struct inode {
    union {                //包含不同文件系统特定inode信息的union成员
        struct device __device_info; //设备文件系统内存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_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
};
```

(续下页)

(接上页)

```

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, 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_getdirent)(struct inode *node, struct iobuf *iob);
    int (*vop_create)(struct inode *node, const char *name, bool excl, struct inode **
    ↪ node_store);
    int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
    ...
};

```

参照上面对 SFS 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。

11.1.6 硬盘文件系统 SFS

我们从 ucore 实验指导书摘抄一段关于 Simple File System 的介绍。

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表所示。

superblock	root-dir inode	freemap	inode、File Data、Dir Data Blocks
超级块	根目录索引节点	空闲块映射	目录和文件的数据和索引节点

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。其定义如下:

```

struct sfs_super {
    uint32_t magic;           /* magic number, should be SFS_MAGIC */
    ↪ GIC */
    uint32_t blocks;          /* # of blocks in fs */
    uint32_t unused_blocks;   /* # of unused blocks in fs */
    char info[SFS_MAX_INFO_LEN + 1]; /* information 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"。

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。

趣闻 “魔数”是怎样工作的?

我们经常需要检查某个文件/某块磁盘是否符合我们需要的格式。一般会按照这个文件的完整格式, 进行一次全面的分析。

在一个较早的版本, UNIX 的可执行文件格式最开头包含一条 PDP-11 平台上的跳转指令, 使得在 PDP-11 硬件平台上能够正常运行, 而在其他平台上, 这条指令就是“魔数”(magic number), 只能用作文件类型的标识。

Java 类文件 (编译到字节码) 以十六进制 0xCAFEBAE 开头

JPEG 图片文件以 0xFFD8 开头, 0xFFD9 结尾

PDF 文件以 “%PDF” 的 ASCII 码开头, 十六进制 25 50 44 46

进行这样的约定之后, 我们发现, 如果文件开头的“魔数”不符合要求, 那么这个文件的格式一定不对。这让我们立刻发现文件损坏或者搞错文件类型的情况。由于不同类型的文件有不同的魔数, 当你把 JPEG 文件当作 PDF 打开的时候, 立即就会出现异常。

下面是一个摇滚乐队和巧克力豆的故事, 有助于你理解魔数的作用。

美国著名重金属摇滚乐队 Van Halen 的演出合同中有此一条: 演出后台必须提供 M&M 巧克力豆, 但是绝对不许出现棕色豆。如有违反, 根据合同, 乐队可以取消演出。实际情形中乐队甚至会借此发飙, 砸后台, 主办方也只好承担所有经济损失。这一条款长期被媒体用来作为摇滚乐队耍大牌的典型例子, 有传言指某次由于主唱在后台发现了棕色 M&M 豆, 大发其飙地砸了后台, 造成损失高达八万五千美元 (当时是八十年代, 八万五千还是不少钱)。Van Halen 乐队对此从不回应。

多年以后, 主唱 David Lee Roth 在自传中揭示了这一无厘头条款的来由: Van Halen 乐队在当时是把大型摇滚现场演唱会推向高校及二 / 三线地区的先锋, 由于常常会遇到没有处理过这种大场面的承办方, 因此合同里有大量条款来确认演出承办方把场地, 器材, 工作人员安排等等细节都严格按照要求准备好。合同里有成章成章的技术细节, 包括场地的承重要求, 各类出入口的宽度, 电源要求, 以至于插座的数量和插座之间的间隔。因此, 乐队把棕色豆条款夹带在合同里, 以确认承办方是否“仔仔细细阅读了所有条款”。David 说: “如果我在后台的 M&M 里找到棕色豆, 我就会立马知道承办方 (十有八九) 是没好好读全部技术要求, 我们肯定会碰上技术问题。某些技术问题绝对会毁了这场演出, 甚至害死人。”

回到上文, 八万五千美元的损失是怎么来的? 某次在某大学体育场办演唱会, 主唱来到后台, 发现了棕色 M&M 豆, 当即发飙, 砸了后台化妆室, 财物损坏大概值一万二。但实际上更糟糕的是, 主办方没有细读演出演出场地的承重要求, 结果整个舞台压垮 (似乎是压穿) 了体育场地面, 损失高达八万多。

事后媒体的报道是, 由于主唱看到棕色 M&M 豆后发飙砸了后台, 造成高达八万五的损失…

11.1.6.1 索引节点

在 SFS 文件系统中, 需要记录文件内容的存储位置以及文件名与文件内容的对应关系。sfs_disk_inode 记录了文件或目录的内容存储的索引信息, 该数据结构在硬盘里储存, 需要时读入内存 (从磁盘读进来的是一段连续的字节, 我们将这段连续的字节强制转换成 sfs_disk_inode 结构体; 同样, 写入的时候换一个方向强制转换)。sfs_disk_entry 表示一个目录中的一个文件或目录, 包含该项所对应 inode 的位置和文件名, 同样也在硬盘里储存, 需要时读入内存。

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:

```
// kern/fs/sfs/sfs.hc
/*inode (on disk)*/
struct sfs_disk_inode {
    uint32_t size;                //如果inode表示常规文件, 则size是文件
    ↪大小
    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_NDI
    ↪RECT个)
    uint32_t indirect;            //此inode的一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, 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。

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 $12 * 4k + 1024 * 4k = 48k + 4m$ 。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。indirect 为 0 时, 表示不使用一级索引块。(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:

```
// kern/fs/sfs/sfs.h
/* file entry (on disk) */
struct sfs_disk_entry {
    uint32_t ino;                //索引节点所占数据块索引值
    char name[SFS_MAX_FNAME_LEN + 1]; //文件名
};
```

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_disk_entry 也占用一个 block。

内存中的索引节点

```
// kern/fs/sfs/sfs.h
/* inode for sfs */
struct sfs_inode {
    struct sfs_disk_inode *din;    /* on-disk inode */
    uint32_t ino;                  /* inode number */
    uint32_t flags;                /* inode flags */
};
```

(续下页)

(接上页)

```

bool dirty;                                /* true if inode modified */
int reclaim_count;                         /* kill inode if it hits zero */
semaphore_t sem;                          /* semaphore for din */
list_entry_t inode_link;                  /* entry for linked-list in sfs_fs */
list_entry_t hash_link;                   /* entry for hash linked-list in sfs_fs */
→ */
};

```

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：

(在 kern/fs/sfs/sfs_inode.c 实现)

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_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

```

// kern/fs/sfs/sfs_inode.c
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


```
// kern/fs/sfs/sfs_inode.c
static const struct inode_ops sfs_node_dirops = {
    .vop_magic          = VOP_MAGIC,
    .vop_open           = sfs_opendir,
    .vop_close          = sfs_close,
    .vop_getdirent      = sfs_getdirent,
    .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`、`sys_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_getdirent`(), 其主要工作是获取目录下的文件 `inode` 信息。

这里用到的 `inode_ops` 结构体, 在 `kern/fs/vfs/inode.h` 定义, 作用是: 把关于 `inode` 的操作接口, 集中在一个结构体里, 通过这个结构体, 我们可以把 Simple File System 的接口 (如 `sfs_openfile()`) 提供给上层的 VFS 使用。可以想象我们除了 Simple File System, 还在另一块磁盘上使用完全不同的文件系统 Complex File System, 显然 `vop_open()`、`vop_read()` 这些接口的实现都要不一样了。对于同一个文件系统这些接口都是一样的, 所以我们可以提供“属于 SFS 的文件的 `inode_ops` 结构体”, “属于 CFS 的文件的 `inode_ops` 结构体”。

下面的注释里详细解释了每个接口的用途。当然, 不必现在就详细了解每一个接口。

```
// kern/fs/vfs/inode.h
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_fstat)(struct inode *node, struct stat *stat);
    int (*vop_fsync)(struct inode *node);
    int (*vop_namefile)(struct inode *node, struct iobuf *iob);
    int (*vop_getdirent)(struct inode *node, struct iobuf *iob);
    int (*vop_reclaim)(struct inode *node);
    int (*vop_gettype)(struct inode *node, uint32_t *type_store);
    int (*vop_tryseek)(struct inode *node, off_t pos);
    int (*vop_truncate)(struct inode *node, off_t len);
    int (*vop_create)(struct inode *node, const char *name, bool excl, struct inode **
    ↪ node_store);
    int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
    int (*vop_ioctl)(struct inode *node, int op, void *data);
};

/*
 * Abstract operations on a inode.
 *
 * These are used in the form VOP_FOO(inode, args), which are macros
 * that expands to inode->inode_ops->vop_foo(inode, args). The operations
 * "foo" are:
 *
 *     vop_open          - Called on open() of a file. Can be used to
 *                         reject illegal or undesired open modes. Note that
 *                         various operations can be performed without the
 *                         file actually being opened.
 *                         The inode need not look at O_CREAT, O_EXCL, or
```

(续下页)

(接上页)

```

*          O_TRUNC, as these are handled in the VFS layer.
*
*          VOP_EACHOPEN should not be called directly from
*          above the VFS layer - use vfs_open() to open inodes.
*          This maintains the open count so VOP_LASTCLOSE can
*          be called at the right time.
*
* vop_close      - To be called on *last* close() of a file.
*
*          VOP_LASTCLOSE should not be called directly from
*          above the VFS layer - use vfs_close() to close
*          inodes opened with vfs_open().
*
* vop_reclaim    - Called when inode is no longer in use. Note that
*                  this may be substantially after vop_lastclose is
*                  called.
*
*****
*
* vop_read        - Read data from file to uio, at offset specified
*                  in the uio, updating uio_resid to reflect the
*                  amount read, and updating uio_offset to match.
*                  Not allowed on directories or symlinks.
*
* vop_getdirentry - Read a single filename from a directory into a
*                  uio, choosing what name based on the offset
*                  field in the uio, and updating that field.
*                  Unlike with I/O on regular files, the value of
*                  the offset field is not interpreted outside
*                  the filesystem and thus need not be a byte
*                  count. However, the uio_resid field should be
*                  handled in the normal fashion.
*                  On non-directory objects, return ENOTDIR.
*
* vop_write       - Write data from uio to file at offset specified
*                  in the uio, updating uio_resid to reflect the
*                  amount written, and updating uio_offset to match.
*                  Not allowed on directories or symlinks.
*
* vop_ioctl       - Perform ioctl operation OP on file using data
*                  DATA. The interpretation of the data is specific
*                  to each ioctl.
*
* vop_fstat       - Return info about a file. The pointer is a
*                  pointer to struct stat; see stat.h.
*
* vop_gettype     - Return type of file. The values for file types
*                  are in sfs.h.
*
* vop_tryseek     - Check if seeking to the specified position within
*                  the file is legal. (For instance, all seeks
*                  are illegal on serial port devices, and seeks
*                  past EOF on files whose sizes are fixed may be
*                  as well.)
*
* vop_fsync       - Force any dirty buffers associated with this file
*                  to stable storage.

```

(续下页)

(接上页)

```

*
*   vop_truncate    - Forcibly set size of file to the length passed
*                     in, discarding any excess blocks.
*
*   vop_namefile    - Compute pathname relative to filesystem root
*                     of the file and copy to the specified io buffer.
*                     Need not work on objects that are not
*                     directories.
*
*****
*
*   vop_creat       - Create a regular file named NAME in the passed
*                     directory DIR. If boolean EXCL is true, fail if
*                     the file already exists; otherwise, use the
*                     existing file if there is one. Hand back the
*                     inode for the file as per vop_lookup.
*
*****
*
*   vop_lookup      - Parse PATHNAME relative to the passed directory
*                     DIR, and hand back the inode for the file it
*                     refers to. May destroy PATHNAME. Should increment
*                     refcount on inode handed back.
*/

```

11.1.7 设备

在本实验中, 为了统一地访问设备 (device), 我们可以把一个设备看成一个文件, 通过访问文件的接口来访问设备。目前实现了 stdin 设备文件、stdout 设备文件、disk0 设备。stdin 设备就是键盘, stdout 设备就是控制台终端的文本显示, 而 disk0 设备是承载 SFS 文件系统的磁盘设备。下面看看 ucore 是如何让用户把设备看成文件来访问。

11.1.7.1 设备的定义

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, 如下:

可以认为 struct device 是一个比较抽象的“设备”的定义。一个具体设备, 只要实现了 d_open() 打开设备, d_close() 关闭设备, d_io() (读写该设备, write 参数是 true/false 决定是读还是写), d_ioctl() (input/output control) 四个函数接口, 就可以被文件系统使用了。

```

// kern/fs/devs/dev.h
/*
 * Filesystem-namespace-accessible device.
 * d_io is for both reads and writes; the iobuf will indicates the direction.
 */
struct device {
    size_t d_blocks;
    size_t d_blocksize;
    int (*d_open)(struct device *dev, uint32_t open_flags);
    int (*d_close)(struct device *dev);
    int (*d_io)(struct device *dev, struct iobuf *iob, bool write);
    int (*d_ioctl)(struct device *dev, int op, void *data);
};

```

(续下页)

(接上页)

```

#define dop_open(dev, open_flags)      ((dev)->d_open(dev, open_flags))
#define dop_close(dev)                 ((dev)->d_close(dev))
#define dop_io(dev, iob, write)        ((dev)->d_io(dev, iob, write))
#define dop_ioctl(dev, op, data)       ((dev)->d_ioctl(dev, op, data))

```

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）的表示，完成对设备的基本操作。

但这个设备描述没有与文件系统以及表示一个文件的 inode 数据结构建立关系，为此，还需要另外一个数据结构把 device 和 inode 联通起来，这就是 `vfs_dev_t` 数据结构。

利用 `vfs_dev_t` 数据结构，就可以让文件系统通过一个链接 `vfs_dev_t` 结构的双向链表找到 device 对应的 inode 数据结构，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。

```

// kern/fs/vfs/vfsdev.c
// device info entry in vdev_list
typedef struct {
    const char *devname;
    struct inode *devnode;
    struct fs *fs;
    bool mountable;
    list_entry_t vdev_link;
} vfs_dev_t;
#define le2vdev(le, member) \
    to_struct((le), vfs_dev_t, member) //为了使用链表定义的宏，做到现在应该对它很熟悉
→了

static list_entry_t vdev_list; // device info list in vfs layer
static semaphore_t vdev_list_sem; // 互斥访问的 semaphore
static void lock_vdev_list(void) {
    down(&vdev_list_sem);
}
static void unlock_vdev_list(void) {
    up(&vdev_list_sem);
}

```

ucore 虚拟文件系统为了把这些设备链接在一起，还定义了一个设备链表，即双向链表 `vdev_list`，这样通过访问此链表，可以找到 ucore 能够访问的所有设备文件。

注意这里的 `vdev_list` 对应一个 `vdev_list_sem`。在文件系统中，互斥访问非常重要，所以我们将看到很多的 semaphore。

我们使用 `iobuf` 结构体传递一个 IO 请求（要写入设备的数据当前所在内存的位置和长度/从设备读取的数据需要存储到的位置）

```

struct iobuf {
    void *io_base; // the base addr of buffer (used for Rd/Wr)
    off_t io_offset; // current Rd/Wr position in buffer, will have been incremented
→ by the amount transferred
    size_t io_len; // the length of buffer (used for Rd/Wr)
    size_t io_resid; // current resident length need to Rd/Wr, will have been decrem
→ented by the amount transferred.
};

```

注意设备文件的 inode 也有一个 `inode_ops` 成员，提供设备文件应具备的接口。

```
// kern/fs/devs/dev.c
/*
 * Function table for device inodes.
 */
static const struct inode_ops dev_node_ops = {
    .vop_magic          = VOP_MAGIC,
    .vop_open           = dev_open,
    .vop_close          = dev_close,
    .vop_read           = dev_read,
    .vop_write          = dev_write,
    .vop_fstat          = dev_fstat,
    .vop_ioctl          = dev_ioctl,
    .vop_gettype        = dev_gettype,
    .vop_tryseek        = dev_tryseek,
    .vop_lookup         = dev_lookup,
};
```

11.1.7.2 stdin 设备

trap.c 改变了对 stdin 的处理, 将 stdin 作为一个设备 (也是一个文件), 通过 sys_read() 接口读取标准输入的数据。

注意, 既然我们把 stdin, stdout 看作文件, 那么也需要先打开文件, 才能进行读写。在执行用户程序之前, 我们先执行了 umain.c 建立一个运行时环境, 这里主要做的工作, 就是让程序能够使用 stdin, stdout。

```
// user/libs/file.c
//这是用户态程序可以使用的“系统库函数”, 从文件fd读取len个字节到base这个位置。
//当fd = 0的时候, 表示从stdin读取
int read(int fd, void *base, size_t len) {
    return sys_read(fd, base, len);
}

// user/libs/umain.c
int main(int argc, char *argv[]) {

static int initfd(int fd2, const char *path, uint32_t open_flags) {
    int fd1, ret;
    if ((fd1 = open(path, open_flags)) < 0) {
        return fd1;
    }//我们希望文件描述符是fd2, 但是分配的fd1如果不等于fd2, 就需要做一些处理
    if (fd1 != fd2) {
        close(fd2);
        ret = dup2(fd1, fd2);//通过sys_dup让两个文件描述符指向同一个文件
        close(fd1);
    }
    return ret;
}

void umain(int argc, char *argv[]) {
    int fd;
    if ((fd = initfd(0, "stdin:", O_RDONLY)) < 0) {
        warn("open <stdin> failed: %e.\n", fd);
    }//0用于描述stdin, 这里因为第一个被打开, 所以stdin会分配到0
    if ((fd = initfd(1, "stdout:", O_WRONLY)) < 0) {
        warn("open <stdout> failed: %e.\n", fd);
    }//1用于描述stdout
}
```

(续下页)

(接上页)

```

int ret = main(argc, argv); //真正的“用户程序”
exit(ret);
}

```

这里我们需要把命令行的输入转换成一个文件，于是需要一个缓冲区：把已经在命令行输入，但还没有被读取的数据放在缓冲区里。这里遇到一个问题：每当控制台输入一个字符，我们都要及时把它放到 `stdin` 的缓冲区里。一般来说，应当有键盘的外设中断来提醒我们。但是我们在 QEMU 里收不到这个中断，于是采取一个措施：借助时钟中断，每次时钟中断检查是否有新的字符，这效率比较低，不过也还可以接受。

```

// kern/trap/trap.c
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        /*...*/
        case IRQ_S_TIMER:
            clock_set_next_event();

            if (++ticks % TICK_NUM == 0 && current) {
                // print_ticks();
                current->need_resched = 1;
            }
            run_timer_list();
            //按理说用户程序看到的stdin是“只读”的
            //但是，一个文件，只往外读，不往里写，是不是会导致数据“不守恒”？
            //我们在这里就是把控制台输入的数据“写到”stdin里(实际上是写到一个缓冲区里
            ↪)

            //这里的cons_getc()并不一定能返回一个字符,可以认为是轮询
            //如果cons_getc()返回0,那么dev_stdin_write()函数什么都不做
            dev_stdin_write(cons_getc());
            break;
    }
}

// kern/driver/console.c

#define CONSBUFSIZE 512

static struct {
    uint8_t buf[CONSBUFSIZE];
    uint32_t rpos;
    uint32_t wpos; //控制台的输入缓冲区是一个队列
} cons;

/* *
 * cons_intr - called by device interrupt routines to feed input
 * characters into the circular console input buffer.
 * */
void cons_intr(int (*proc)(void)) {
    int c;
    while ((c = (*proc)()) != -1) {
        if (c != 0) {
            cons.buf[cons.wpos++] = c;
            if (cons.wpos == CONSBUFSIZE) {
                cons.wpos = 0;
            }
        }
    }
}

```

(续下页)

(接上页)

```

    }
}
/* serial_proc_data - get data from serial port */
int serial_proc_data(void) {
    int c = sbi_console_getchar();
    if (c < 0) {
        return -1;
    }
    if (c == 127) {
        c = '\b';
    }
    return c;
}

/* serial_intr - try to feed input characters from serial port */
void serial_intr(void) {
    cons_intr(serial_proc_data);
}

/* *
 * cons_getc - return the next input character from console,
 * or 0 if none waiting.
 * */
int cons_getc(void) {
    int c = 0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        // poll for any pending input characters,
        // so that this function works even when interrupts are disabled
        // (e.g., when called from the kernel monitor).
        serial_intr();

        // grab the next character from the input buffer.
        if (cons.rpos != cons.wpos) {
            c = cons.buf[cons.rpos++];
            if (cons.rpos == CONSBUFSIZE) {
                cons.rpos = 0;
            }
        }
    }
    local_intr_restore(intr_flag);
    return c;
}

```

我们来看 stdin 设备的实现:

```

// kern/fs/devs/dev_stdin.c

#define STDIN_BUFSIZE 4096
static char stdin_buffer[STDIN_BUFSIZE];
//这里又有一个stdin设备的缓冲区, 能否和之前console的缓冲区合并?
static off_t p_rpos, p_wpos;
static wait_queue_t __wait_queue, *wait_queue = &__wait_queue;

void dev_stdin_write(char c) { //把其他地方的字符写到stdin缓冲区, 准备被读取
    bool intr_flag;
    if (c != '\0') {

```

(续下页)

(接上页)

```

    local_intr_save(intr_flag); //禁用中断
    {
        stdin_buffer[p_wpos % STDIN_BUFSIZE] = c;
        if (p_wpos - p_rpos < STDIN_BUFSIZE) {
            p_wpos ++;
        }
        if (!wait_queue_empty(wait_queue)) {
            wakeup_queue(wait_queue, WT_KBD, 1);
            //若当前有进程在等待字符输入, 则进行唤醒
        }
    }
    local_intr_restore(intr_flag);
}

static int dev_stdin_read(char *buf, size_t len) { //读取len个字符
    int ret = 0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        for (; ret < len; ret ++, p_rpos ++ ) {
            try_again:
            if (p_rpos < p_wpos) { //当前队列非空
                *buf ++ = stdin_buffer[p_rpos % STDIN_BUFSIZE];
            }
            else {
                //希望读取字符, 但是当前没有字符, 那么进行等待
                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; //再次尝试
                }
                break;
            }
        }
    }
    local_intr_restore(intr_flag);
    return ret;
}

static int stdin_io(struct device *dev, struct iobuf *iob, bool write) {
    //对应 struct device 的 d_io()
    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;
}

```


11.1.7.3 stdout 设备

stdout 设备只需要支持写操作, 调用 `cputchar()` 把字符打印到控制台。

```
// kern/fs/devs/dev_stdout.c
static int stdout_io(struct device *dev, struct iobuf *iob, bool write) {
    //对应struct device 的d_io()
    if (write) {
        char *data = iob->io_base;
        for (; iob->io_resid != 0; iob->io_resid --) {
            cputchar(*data ++);
        }
        return 0;
    }
    //if !write:
    return -EINVAL; //对stdout执行读操作会报错
}
```

11.1.7.4 disk0 设备

封装了一下 ramdisk 的接口, 每次读取或者写入若干个 block。

```
// kern/fs/devs/dev_disk0.c
static char *disk0_buffer;
static semaphore_t disk0_sem;

static void
lock_disk0(void) {
    down(&(disk0_sem));
}

static void
unlock_disk0(void) {
    up(&(disk0_sem));
}

static void disk0_read_blks_nolock(uint32_t blkno, uint32_t nblks) {
    int ret;
    uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
    if ((ret = ide_read_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0) {
        panic("disk0: read blkno = %d (sectno = %d), nblks = %d (nsecs = %d): 0x%08x.\n",
            blkno, sectno, nblks, nsecs, ret);
    }
}

static void disk0_write_blks_nolock(uint32_t blkno, uint32_t nblks) {
    int ret;
    uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
    if ((ret = ide_write_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0) {
        panic("disk0: write blkno = %d (sectno = %d), nblks = %d (nsecs = %d): 0x%08x.\n",
            blkno, sectno, nblks, nsecs, ret);
    }
}
```

(续下页)

(接上页)

```

static int disk0_io(struct device *dev, struct iobuf *iob, bool write) {
    off_t offset = iob->io_offset;
    size_t resid = iob->io_resid;
    uint32_t blkno = offset / DISK0_BLKSIZE;
    uint32_t nblks = resid / DISK0_BLKSIZE;

    /* don't allow I/O that isn't block-aligned */
    if ((offset % DISK0_BLKSIZE) != 0 || (resid % DISK0_BLKSIZE) != 0) {
        return -E_INVAL;
    }

    /* don't allow I/O past the end of disk0 */
    if (blkno + nblks > dev->d_blocks) {
        return -E_INVAL;
    }

    /* read/write nothing ? */
    if (nblks == 0) {
        return 0;
    }

    lock_disk0();
    while (resid != 0) {
        size_t copied, alen = DISK0_BUFSIZE;
        if (write) {
            iobuf_move(iob, disk0_buffer, alen, 0, &copied);
            assert(copied != 0 && copied <= resid && copied % DISK0_BLKSIZE == 0);
            nblks = copied / DISK0_BLKSIZE;
            disk0_write_blks_nolock(blkno, nblks);
        }
        else {
            if (alen > resid) {
                alen = resid;
            }
            nblks = alen / DISK0_BLKSIZE;
            disk0_read_blks_nolock(blkno, nblks);
            iobuf_move(iob, disk0_buffer, alen, 1, &copied);
            assert(copied == alen && copied % DISK0_BLKSIZE == 0);
        }
        resid -= copied, blkno += nblks;
    }
    unlock_disk0();
    return 0;
}

```

这些设备的实现看起来比较复杂, 实际上属于比较麻烦的设备驱动的部分。

11.1.8 open 系统调用的执行过程

下面我们通过打开文件的系统调用 `open()` 的执行过程, 看看文件系统的不同层次是如何交互的。

11.1.8.1 通用文件访问接口层的处理流程

首先, 经过 `syscall.c` 的处理之后, 进入内核态, 执行 `sysfile_open()` 函数

```
// kern/fs/sysfile.c
/* sysfile_open - open file */
int sysfile_open(const char *__path, uint32_t open_flags) {
    int ret;
    char *path;
    if ((ret = copy_path(&path, __path)) != 0) {
        return ret;
    }
    ret = file_open(path, open_flags);
    kfree(path);
    return ret;
}
```

到了这里, 需要把位于用户空间的字符串 `__path` 拷贝到内核空间中的字符串 `path` 中, 然后调用了 `file_open`, `file_open` 调用了 `vfs_open`, 使用了 VFS 的接口, 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。

11.1.8.2 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 `fd`。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。

```
// kern/fs/file.c
// open file
int file_open(char *path, uint32_t open_flags) {
    bool readable = 0, writable = 0;
    switch (open_flags & O_ACCMODE) { //解析 open_flags
        case O_RDONLY: readable = 1; break;
        case O_WRONLY: writable = 1; break;
        case O_RDWR:
            readable = writable = 1;
            break;
        default:
            return -E_INVALID;
    }
    int ret;
    struct file *file;
    if ((ret = fd_array_alloc(NO_FD, &file)) != 0) { //在当前进程分配file descriptor
        return ret;
    }
    struct inode *node;
    if ((ret = vfs_open(path, open_flags, &node)) != 0) { //打开文件的工作在vfs_open完
        ↪成
        fd_array_free(file); //打开失败, 释放file descriptor
    }
}
```

(续下页)

(接上页)

```

        return ret;
    }
    file->pos = 0;
    if (open_flags & O_APPEND) {
        struct stat __stat, *stat = &__stat;
        if ((ret = vop_fstat(node, stat)) != 0) {
            vfs_close(node);
            fd_array_free(file);
            return ret;
        }
        file->pos = stat->st_size; //追加写模式, 设置当前位置为文件尾
    }
    file->node = node;
    file->readable = readable;
    file->writable = writable;
    fd_array_open(file); //设置该文件的状态为“打开”
    return file->fd;
}

```

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。 `vfs_open` 是一个比较复杂的函数，这里我们使用的打开文件的 `flags`，基本是参照 `linux`，如果希望详细了解，可以阅读 `linux manual: open`。

```

// kern/fs/vfs/vfsfile.c

// open file in vfs, get/create inode for file with filename path.
int vfs_open(char *path, uint32_t open_flags, struct inode **node_store) {
    bool can_write = 0;
    // 解析open_flags并做合法性检查
    switch (open_flags & O_ACCMODE) {
        case O_RDONLY:
            break;
        case O_WRONLY:
            break;
        case O_RDWR:
            can_write = 1;
            break;
        default:
            return -E_INVAL;
    }

    if (open_flags & O_TRUNC) {
        if (!can_write) {
            return -E_INVAL;
        }
    }
}

/*
linux manual
O_TRUNC
    If the file already exists and is a regular file and the
    access mode allows writing (i.e., is O_RDWR or O_WRONLY) it
    will be truncated to length 0.  If the file is a FIFO or ter-
    minal device file, the O_TRUNC flag is ignored.  Otherwise,
    the effect of O_TRUNC is unspecified.

```

(续下页)

(接上页)

```

*/
int ret;
struct inode *node;
bool excl = (open_flags & O_EXCL) != 0;
bool create = (open_flags & O_CREAT) != 0;
ret = vfs_lookup(path, &node); // vfs_lookup根据路径构造inode

if (ret != 0) { //要打开的文件还不存在, 可能出错, 也可能需要创建新文件
    if (ret == -16 && (create)) {
        char *name;
        struct inode *dir;
        if ((ret = vfs_lookup_parent(path, &dir, &name)) != 0) {
            return ret; //需要在已经存在的目录下创建文件, 目录不存在, 则出错
        }
        ret = vop_create(dir, name, excl, &node); //创建新文件
    } else return ret;
} else if (excl && create) {
    return -E_EXISTS;
}
/*
linux manual
    O_EXCL Ensure that this call creates the file: if this flag is
    specified in conjunction with O_CREAT, and pathname already
    exists, then open() fails with the error EEXIST.
*/
}
assert(node != NULL);

if ((ret = vop_open(node, open_flags)) != 0) {
    vop_ref_dec(node);
    return ret;
}

vop_open_inc(node);
if (open_flags & O_TRUNC || create) {
    if ((ret = vop_truncate(node, 0)) != 0) {
        vop_open_dec(node);
        vop_ref_dec(node);
        return ret;
    }
}
*node_store = node;
return 0;
}

```

`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 文件系统中的目录下的文件。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。通过调用 `vop_lookup` 函数来查找到根目录 “/” 下对应文件 `sfs_filetest1` 的索引节点, 如果找到就返回此索引节点。

```

/*
 * get_device- Common code to pull the device name, if any, off the front of a
 *              path and choose the inode to begin the name lookup relative to.
 */

```

(续下页)

(接上页)

```

static int get_device(char *path, char **subpath, struct inode **node_store) {
    int i, slash = -1, colon = -1;
    for (i = 0; path[i] != '\0'; i++) {
        if (path[i] == ':') { colon = i; break; }
        if (path[i] == '/') { slash = i; break; }
    }
    if (colon < 0 && slash != 0) {
        /* *
         * No colon before a slash, so no device name specified, and the slash isn't lead
         ↪ing
         * or is also absent, so this is a relative path or just a bare filename. Start f
         ↪rom
         * the current directory, and use the whole thing as the subpath.
         * */
        *subpath = path;
        return vfs_get_curdir(node_store); //把当前目录的inode存到node_store
    }
    if (colon > 0) {
        /* device:path - get root of device's filesystem */
        path[colon] = '\0';

        /* device:/path - skip slash, treat as device:path */
        while (path[++ colon] == '/');
        *subpath = path + colon;
        return vfs_get_root(path, node_store);
    }

    /* *
     * we have either /path or :path
     * /path is a path relative to the root of the "boot filesystem"
     * :path is a path relative to the root of the current filesystem
     * */
    int ret;
    if (*path == '/') {
        if ((ret = vfs_get_bootfs(node_store)) != 0) {
            return ret;
        }
    }
    else {
        assert(*path == ':');
        struct inode *node;
        if ((ret = vfs_get_curdir(&node)) != 0) {
            return ret;
        }
        /* The current directory may not be a device, so it must have a fs. */
        assert(node->in_fs != NULL);
        *node_store = fsop_get_root(node->in_fs);
        vop_ref_dec(node);
    }

    /* ///... or :/... */
    while (*(++ path) == '/');
    *subpath = path;
    return 0;
}

```

(续下页)

(接上页)

```

/*
 * vfs_lookup - get the inode according to the path filename
 */
int vfs_lookup(char *path, struct inode **node_store) {
    int ret;
    struct inode *node;
    if ((ret = get_device(path, &path, &node)) != 0) {
        return ret;
    }
    if (*path != '\0') {
        ret = vop_lookup(node, path, node_store);
        vop_ref_dec(node);
        return ret;
    }
    *node_store = node;
    return 0;
}

/*
 * vfs_lookup_parent - Name-to-vnode translation.
 * (In BSD, both of these are subsumed by namei().)
 */
int vfs_lookup_parent(char *path, struct inode **node_store, char **endp) {
    int ret;
    struct inode *node;
    if ((ret = get_device(path, &path, &node)) != 0) {
        return ret;
    }
    *endp = path;
    *node_store = node;
    return 0;
}

```

我们注意到, 这个流程中, 有大量以 vop 开头的函数, 它们都通过一些宏和函数的转发, 最后变成对 inode 结构体里的 inode_ops 结构体的“成员函数”(实际上是函数指针)的调用。对于 SFS 文件系统的 inode 来说, 会变成对 sfs 文件系统的操作。

11.1.8.3 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。注意: 在 lab8 中, 为简化代码, sfs_lookup 函数中并没有实现能够对多级目录进行查找的控制逻辑 (在 ucore_plus 中有实现)。

```

/*
 * sfs_lookup - Parse path relative to the passed directory
 *              DIR, and hand back the inode for the file it
 *              refers to.
 */
static int
sfs_lookup(struct inode *node, char *path, struct inode **node_store) {
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    assert(*path != '\0' && *path != '/');
    vop_ref_inc(node);

```

(续下页)

(接上页)

```

    struct sfs_inode *sin = vop_info(node, sfs_inode);
    if (sin->din->type != SFS_TYPE_DIR) {
        vop_ref_dec(node);
        return -E_NOTDIR;
    }
    struct inode *subnode;
    int ret = sfs_lookup_once(sfs, sin, path, &subnode, NULL);

    vop_ref_dec(node);
    if (ret != 0) {
        return ret;
    }
    *node_store = subnode;
    return 0;
}

```

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 sfs_filetest1 的绝对路径/sfs_filetest1, 而 node_store 是经过查找获得的 sfs_filetest1 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。在本例中是调用 sfs_lookup_once 查找以根目录下的文件 sfs_filetest1 所对应的 inode 节点。当无法分解 path 后, 就意味着找到了 sfs_filetest1 对应的 inode 节点, 就可顺利返回了。

```

/*
 * sfs_lookup_once - find inode corresponding the file name in DIR's sin inode
 * @sfs:          sfs file system
 * @sin:          DIR sfs inode in memory
 * @name:         the file name in DIR
 * @node_store:   the inode corresponding the file name in DIR
 * @slot:         the logical index of file entry
 */
static int
sfs_lookup_once(struct sfs_fs *sfs, struct sfs_inode *sin, const char *name, struct in
↳ode **node_store, int *slot) {
    int ret;
    uint32_t ino;
    lock_sin(sin);
    { // find the NO. of disk block and logical index of file entry
        ret = sfs_dirent_search_nolock(sfs, sin, name, &ino, slot, NULL);
    }
    unlock_sin(sin);
    if (ret == 0) {
        // load the content of inode with the the NO. of disk block
        ret = sfs_load_inode(sfs, node_store, ino);
    }
    return ret;
}

```

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。

11.1.9 Read 系统调用执行过程

读文件其实就是读出目录中的目录项，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。下面来分析一下读文件的实现。

11.1.9.1 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。

```
int
read(int fd, void *base, size_t len) {
    return sys_read(fd, base, len);
}
```

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。

```
static int
sys_read(uint64_t arg[]) {
    int fd = (int)arg[0];
    void *base = (void *)arg[1];
    size_t len = (size_t)arg[2];
    return sysfile_read(fd, base, len);
}
```

11.1.9.2 文件系统抽象层的处理流程

- 1) 检查错误，即检查读取长度是否为 0 和文件是否可读。
- 2) 分配 buffer 空间，即调用 kmalloc 函数分配 4096 字节的 buffer 空间。
- 3) 读文件过程

[1] 实际读文件

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。

```
int
sysfile_read(int fd, void *base, size_t len) {
    struct mm_struct *mm = current->mm;
    if (len == 0) {
        return 0;
    }
    if (!file_testfd(fd, 1, 0)) {
        return -E_INVALID;
    }
    void *buffer;
    if ((buffer = kmalloc(IOBUF_SIZE)) == NULL) {
        return -E_NO_MEM;
    }
}
```

(续下页)

(接上页)

```

    }

    int ret = 0;
    size_t copied = 0, alen;
    while (len != 0) {
        if ((alen = IOBUF_SIZE) > len) {
            alen = len;
        }
        ret = file_read(fd, buffer, alen, &alen);
        if (alen != 0) {
            lock_mm(mm);
            {
                if (copy_to_user(mm, base, buffer, alen)) {
                    assert(len >= alen);
                    base += alen, len -= alen, copied += alen;
                }
                else if (ret == 0) {
                    ret = -E_INVAL;
                }
            }
            unlock_mm(mm);
        }
        if (ret != 0 || alen == 0) {
            goto out;
        }
    }

out:
    kfree(buffer);
    if (copied != 0) {
        return copied;
    }
    return ret;
}

```

[2] 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。

```

// read file
int
file_read(int fd, void *base, size_t len, size_t *copied_store) {
    int ret;
    struct file *file;
    *copied_store = 0;
    if ((ret = fd2file(fd, &file)) != 0) {
        return ret;
    }
    if (!file->readable) {
        return -E_INVAL;
    }
    fd_array_acquire(file);

```

(续下页)

(接上页)

```

struct iobuf __iob, *iob = iobuf_init(&__iob, base, len, file->pos);
ret = vop_read(file->node, iob);

size_t copied = iobuf_used(iob);
if (file->status == FD_OPENED) {
    file->pos += copied;
}
*copied_store = copied;
fd_array_release(file);
return ret;
}

```

11.1.9.3 SFS 文件系统层的处理流程

vop_read 函数实际上是对 sfs_read 的包装。在 sfs_inode.c 中 sfs_node_fileops 变量定义了 .vop_read = sfs_read, 所以下面来分析 sfs_read 函数的实现。

```

static int
sfs_read(struct inode *node, struct iobuf *iob) {
    return sfs_io(node, iob, 0);
}

```

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 表示是读还是写的布尔值 (0 表示读, 1 表示写), 这里是 0。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。

```

/*
 * sfs_io - Rd/Wr file. the wrapper of sfs_io_nolock
 *          with lock protect
 */
static inline int
sfs_io(struct inode *node, struct iobuf *iob, bool write) {
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    struct sfs_inode *sin = vop_info(node, sfs_inode);
    int ret;
    lock_sin(sin);
    {
        size_t alen = iob->io_resid;
        ret = sfs_io_nolock(sfs, sin, iob->io_base, iob->io_offset, &alen, write);
        if (alen != 0) {
            iobuf_skip(iob, alen);
        }
    }
    unlock_sin(sin);
    return ret;
}

```

在 sfs_io_nolock 函数中完成操作如下:

1. 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。
2. 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。

3. 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。
4. 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。

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t offset, size
↳_t *alenp, bool write) {
    struct sfs_disk_inode *din = sin->din;
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
    // calculate the Rd/Wr end position
    if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
        return -E_INVALID;
    }
    if (offset == endpos) {
        return 0;
    }
    if (endpos > SFS_MAX_FILE_SIZE) {
        endpos = SFS_MAX_FILE_SIZE;
    }
    if (!write) {
        if (offset >= din->size) {
            return 0;
        }
        if (endpos > din->size) {
            endpos = din->size;
        }
    }

    int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno, off_t
↳ offset);
    int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t nblks)
↳;
    if (write) {
        sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
    }
    else {
        sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
    }

    int ret = 0;
    size_t size, alen = 0;
    uint32_t ino;
    uint32_t blkno = offset / SFS_BLKSIZE; // The NO. of Rd/Wr begin block
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno; // The size of Rd/Wr blocks

    //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf, sfs_rblock,et
↳c. read different kind of blocks in file
    /*
     * (1) If offset isn't aligned with the first block, Rd/Wr some content from
     ↳offset to the end of the first block
     * NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
     * Rd/Wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset)
     * (2) Rd/Wr aligned blocks
     * NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op

```

(续下页)

(接上页)

```

    * (3) If end position isn't aligned with the last block, Rd/Wr some content from
    ↪begin to the (endpos % SFS_BLKSIZE) of the last block
    * NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
    */

out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}

```

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。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。

```

static int
sfs_bmap_load_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, uint32_t index, uint32_t
    ↪ino_store) {
    struct sfs_disk_inode *din = sin->din;
    assert(index <= din->blocks);
    int ret;
    uint32_t ino;
    bool create = (index == din->blocks);
    if ((ret = sfs_bmap_get_nolock(sfs, sin, index, create, &ino)) != 0) {
        return ret;
    }
    assert(sfs_block_inuse(sfs, ino));
    if (create) {
        din->blocks++;
    }
    if (ino_store != NULL) {
        *ino_store = ino;
    }
    return 0;
}

```

11.1.10 从 zhong duan 到 zhong duan

这是 ucore step by step tutorial 的最后一节: 实现一个简单的终端 (shell)。

可以说, 我们的操作系统之旅, 从 zhong duan(中断) 开始, 也在 zhong duan(终端) 结束。

11.1.10.1 程序的执行

我们的终端需要实现这样的功能: 根据输入的程序名称, 从文件系统里加载对应的程序并执行。我们采取 `fork()` `exec()` 的方式来加载执行程序, `exec()` 的一系列接口都需要重写来使用文件系统。以 `do_execve()` 为例,

以前的函数原型从内存的某个位置加载程序

```
int do_execve(const char *name, size_t len, unsigned char *binary, size_t size) ;
```

现在则调用文件系统接口加载程序, 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。

```
// kern/process/proc.c
// do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of current pro
→cess
//          - call load_icode to setup new memory space accroding binary prog.
int
do_execve(const char *name, int argc, const char **argv) {
    static_assert(EXEC_MAX_ARG_LEN >= FS_MAX_FPATH_LEN);
    struct mm_struct *mm = current->mm;
    if (!(argc >= 1 && argc <= EXEC_MAX_ARG_NUM)) {
        return -E_INVAL;
    }

    char local_name[PROC_NAME_LEN + 1];
    memset(local_name, 0, sizeof(local_name));

    char *kargv[EXEC_MAX_ARG_NUM];
    const char *path;

    int ret = -E_INVAL;

    lock_mm(mm);
    if (name == NULL) {
        snprintf(local_name, sizeof(local_name), "<null> %d", current->pid);
    }
    else {
        if (!copy_string(mm, local_name, name, sizeof(local_name))) {
            unlock_mm(mm);
            return ret;
        }
    }
    if ((ret = copy_kargv(mm, argc, kargv, argv)) != 0) {
        unlock_mm(mm);
        return ret;
    }
    path = argv[0];
    unlock_mm(mm);
    files_closeall(current->files);

    /* sysfile_open will check the first argument path, thus we have to use a user-spa
    →ce pointer, and argv[0] may be incorrect */
    int fd;
    if ((ret = fd = sysfile_open(path, O_RDONLY)) < 0) {
```

(续下页)

(接上页)

```

        goto execve_exit;
    }
    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    ret = -E_NO_MEM;
    if ((ret = load_icode(fd, argc, kargv)) != 0) {
        goto execve_exit;
    }
    put_kargv(argc, kargv);
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    put_kargv(argc, kargv);
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}

```

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。

```

static int
load_icode(int fd, int argc, char **kargv)

```

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；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；
2. 根据应用程序文件的文件描述符 (fd) 通过调用 load_icode_read () 函数来加载和解析此 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；
3. 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；
4. 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，并分配一定数量的物理内存且建立好栈的虚地址 <-> 物理地址映射关系；
5. 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，，设置 uargc 和 uargv 在用户栈中。此时的 initproc 已经被代码和数据覆盖，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；
6. 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；

实验 8 和实验 5 中 `load_icode()` 函数代码最大不同的地方在于读取 ELF 文件的方式，实验 5 中是通过获取 ELF 在内存中的位置，根据 ELF 的格式进行解析，而在实验 8 中则是通过 ELF 文件的文件描述符调用 `load_icode_read()` 函数来进行解析程序。`load_icode_read()` 的函数实现如下：

```
static int
load_icode_read(int fd, void *buf, size_t len, off_t offset) {
    int ret;
    if ((ret = sysfile_seek(fd, offset, LSEEK_SET)) != 0) {
        return ret;
    }
    if ((ret = sysfile_read(fd, buf, len)) != len) {
        return (ret < 0) ? ret : -1;
    }
    return 0;
}
```

这个函数是用来从文件描述符 `fd` 指向的文件中读取数据到 `buf` 中的。函数的入参包括 `fd`（文件描述符）、`buf`（指向读取数据的缓冲区）、`len`（要读取的数据长度）和 `offset`（文件中的偏移量）。

函数首先调用 `sysfile_seek` 函数将文件指针移动到指定的偏移量处。如果 `sysfile_seek` 返回错误码，则 `load_icode_read` 直接返回该错误码。

接着，函数调用 `sysfile_read` 函数从文件中读取指定长度的数据到 `buf` 中。如果读取的数据长度不等于要求的长度 `len`，则 `load_icode_read` 返回错误码 -1；如果读取过程中发生了错误，则 `load_icode_read` 返回 `sysfile_read` 函数的错误码。

最后，如果函数执行成功，则 `load_icode_read` 返回 0。

11.1.10.2 终端的实现

我们还要看一下终端程序的实现。可以发现终端程序需要对命令进行词法和语法分析。

```
// user/sh.c
#include <ulib.h>
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include <file.h>
#include <error.h>
#include <unistd.h>

#define printf(...)      fprintf(1, __VA_ARGS__)
#define putc(c)          printf("%c", c)

#define BUFSIZE          4096
#define WHITESPACE       " \t\r\n"
#define SYMBOLS           "</>&";

char showcwd[BUFSIZE];

int
gettoken(char **p1, char **p2) {
    char *s;
    if ((s = *p1) == NULL) {
```

(续下页)

(接上页)

```

        return 0;
    }
    while (strchr(WHITESPACE, *s) != NULL) {
        *s ++ = '\\0';
    }
    if (*s == '\\0') {
        return 0;
    }

    *p2 = s;
    int token = 'w';
    if (strchr(SYMBOLS, *s) != NULL) {
        token = *s, *s ++ = '\\0';
    }
    else {
        bool flag = 0;
        while (*s != '\\0' && (flag || strchr(WHITESPACE SYMBOLS, *s) == NULL)) {
            if (*s == '"') {
                *s = ' ', flag = !flag;
            }
            s ++;
        }
    }
    *p1 = (*s != '\\0' ? s : NULL);
    return token;
}

char * readline(const char *prompt) {
    static char buffer[BUFSIZE];
    if (prompt != NULL) {
        printf("%s", prompt);
    }
    int ret, i = 0;
    while (1) {
        char c;
        if ((ret = read(0, &c, sizeof(char))) < 0) {
            return NULL;
        }
        else if (ret == 0) {
            if (i > 0) {
                buffer[i] = '\\0';
                break;
            }
            return NULL;
        }

        if (c == 3) {
            return NULL;
        }
        else if (c >= ' ' && i < BUFSIZE - 1) {
            putc(c);
            buffer[i ++] = c;
        }
        else if (c == '\\b' && i > 0) {
            putc(c);
            i --;
        }
    }
}

```

(续下页)

(接上页)

```

        else if (c == '\n' || c == '\r') {
            putc(c);
            buffer[i] = '\0';
            break;
        }
    }
    return buffer;
}

void
usage(void) {
    printf("usage: sh [command-file]\n");
}

int
reopen(int fd2, const char *filename, uint32_t open_flags) {
    int ret, fd1;
    close(fd2);
    if ((ret = open(filename, open_flags)) >= 0 && ret != fd2) {
        close(fd2);
        fd1 = ret, ret = dup2(fd1, fd2);
        close(fd1);
    }
    return ret < 0 ? ret : 0;
}

int
testfile(const char *name) {
    int ret;
    if ((ret = open(name, O_RDONLY)) < 0) {
        return ret;
    }
    close(ret);
    return 0;
}

int
runcmd(char *cmd) {
    static char argv0[BUFSIZE];
    static const char *argv[EXEC_MAX_ARG_NUM + 1]; // must be static!
    char *t;
    int argc, token, ret, p[2];
again:
    argc = 0;
    while (1) {
        switch (token = gettoken(&cmd, &t)) {
            case 'w':
                if (argc == EXEC_MAX_ARG_NUM) {
                    printf("sh error: too many arguments\n");
                    return -1;
                }
                argv[argc++] = t;
                break;
            case '<':
                if (gettoken(&cmd, &t) != 'w') {
                    printf("sh error: syntax error: < not followed by word\n");
                    return -1;
                }

```

(续下页)

(接上页)

```

    }
    if ((ret = reopen(0, t, O_RDONLY)) != 0) {
        return ret;
    }
    break;
case '>':
    if (gettoken(&cmd, &t) != 'w') {
        printf("sh error: syntax error: > not followed by word\n");
        return -1;
    }
    if ((ret = reopen(1, t, O_RDWR | O_TRUNC | O_CREAT)) != 0) {
        return ret;
    }
    break;
case '|':
    // if ((ret = pipe(p)) != 0) {
    //     return ret;
    // }
    if ((ret = fork()) == 0) {
        close(0);
        if ((ret = dup2(p[0], 0)) < 0) {
            return ret;
        }
        close(p[0]), close(p[1]);
        goto again;
    }
    else {
        if (ret < 0) {
            return ret;
        }
        close(1);
        if ((ret = dup2(p[1], 1)) < 0) {
            return ret;
        }
        close(p[0]), close(p[1]);
        goto runit;
    }
    break;
case 0:
    goto runit;
case ';':
    if ((ret = fork()) == 0) {
        goto runit;
    }
    else {
        if (ret < 0) {
            return ret;
        }
        waitpid(ret, NULL);
        goto again;
    }
    break;
default:
    printf("sh error: bad return %d from gettoken\n", token);
    return -1;
}
}

```

(续下页)

(接上页)

```

runit:
    if (argc == 0) {
        return 0;
    }
    else if (strcmp(argv[0], "cd") == 0) {
        if (argc != 2) {
            return -1;
        }
        strcpy(shcwd, argv[1]);
        return 0;
    }
    if ((ret = testfile(argv[0])) != 0) {
        if (ret != -E_NOENT) {
            return ret;
        }
        snprintf(argv0, sizeof(argv0), "%s", argv[0]);
        argv[0] = argv0;
    }
    argv[argc] = NULL;
    return __exec(argv[0], argv);
}

int main(int argc, char **argv) {
    cputs("user sh is running!!!");
    int ret, interactive = 1;
    if (argc == 2) {
        if ((ret = reopen(0, argv[1], O_RDONLY)) != 0) {
            return ret;
        }
        interactive = 0;
    }
    else if (argc > 2) {
        usage();
        return -1;
    }
    //shcwd = malloc(BUFSIZE);
    assert(shcwd != NULL);

    char *buffer;
    while ((buffer = readline((interactive) ? "$ " : NULL)) != NULL) {
        shcwd[0] = '\0';
        int pid;
        if ((pid = fork()) == 0) {
            ret = runcmd(buffer);
            exit(ret);
        }
        assert(pid >= 0);
        if (waitpid(pid, &ret) == 0) {
            if (ret == 0 && shcwd[0] != '\0') {
                ret = 0;
            }
            if (ret != 0) {
                printf("error: %d - %e\n", ret, ret);
            }
        }
    }
}

```

(续下页)

(接上页)

```
return 0;  
}
```

如果我们能够把终端运行起来，并能输入命令执行用户程序，就说明程序运行正常。

目前的代码可以在[这里](#)找到。

11.1.11 实验报告要求

从 git server 网站上取得 ucore_lab 后，进入目录 labcodes/lab8，完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录 labcodes/lab8 下存放实验报告，实验报告文档命名为 lab8-学堂在线 ID.md。推荐用 markdown 格式。对于 lab8 中编程任务，完成编写之后，再通过 git push 命令把代码同步回 git server 网站。最后请一定提前或按时提交到 git server 网站。

注意有“LAB8”的注释，这是需要主要修改的内容。代码中所有需要完成的地方 challenge 除外) 都有“LAB8”和“YOUR CODE”的注释，请在提交时特别注意保持注释，并将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

这部分是一些可能对于实验框架理解有帮助的知识。

12.1 本章内容

12.1.1 makefile 简介

makefile 是一种自动构建项目的工具。在我们的 ucore 实现中，我们使用了 makefile 进行项目的自动构建。下面我们来讨论一些关于 makefile 文件的基本知识。

12.1.1.1 makefile 文件的基本结构

makefile 文件的基本结构是目标 (target)、依赖 (prerequisites) 和命令 (command)。每一个 makefile 文件的内部都包含了许多这样的生成规则。

```
target ... : prerequisites ...  
    command  
    ...  
    ...
```

这三个要件描述了一种依赖关系：当依赖比目标文件新的时候，则命令被执行。当我们在命令行输入 make 的时候，make 程序会自动寻找目录下的 makefile 文件，依次检查依赖关系（以及依赖关系的依赖关系），最终按照合适的顺序执行命令，保证所需要的目标文件被生成。

12.1.1.2 使用宏变量

为了方便把重复的内容进行统一的管理, 可以使用 `makefile` 的变量。这样, 对于重复内容的修改就变成了对变量值的修改。我们一般使用大写的字符串表示变量, 比如 `OBJ`, `OBJECTS` 等。变量定义的方法非常简单, 只需要通过等号赋值即可, 如下

```
OBJ = a.o b.o
```

在需要使用变量的地方, 我们用 `$(变量名)` 来使用变量。比如一段 `makefile` 代码如下所示:

```
main: a.o b.o
    gcc a.o b.o -o main
```

使用了宏变量后就变成了下面这样:

```
OBJ = a.o b.o
main: $(OBJ)
    gcc $(OBJ) -o main
```

如果我们需要添加一个新的 `.o` 文件, 我们只需要修改第一行代码就可以啦 ~

变量也可以使用 `shell` 命令来定义, 比如:

```
cur_dir := $(shell pwd)
```

使用这条命令会把当前所在的目录赋给变量 `cur_dir`, 这也是一种非常好的将 `shell` 指令和 `makefile` 结合起来的方式。

12.1.1.3 控制流和函数

在 `makefile` 中, 可以使用 `if` 等方法控制 `makefile` 执行的控制流。比如下面这个例子:

```
foo: $(objects)
    ifeq ($(CC), gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
    else
        $(CC) -o foo $(objects) $(normal_libs)
    endif
```

这里使用了 `ifeq` 指令判断变量 `CC` 是否为 `gcc`, 如果是则执行链接 `gcc` 相关的库, 否则采取另外一套操作。类似的关键字还有 `ifneq` (如果不相等)、`ifdef` (如果某个变量已定义)、`ifndef` (如果某个变量未定义)。

`makefile` 中也可以使用函数。函数调用遵循下面的范式:

```
$(<function> <arguments>)
```

`makefile` 提供了许多函数, 如字符处理函数, 目录操作函数等等, 可以帮助我们完成一些 `makefile` 中的常用操作。关于如何自定义函数的内容, 感兴趣的同学可以自己去了解。

12.1.1.4 实验 Makefile 解析

下面是对 Lab 中的 makefile 部分代码进行解析:

```
PROJ      := lab1
EMPTY     :=
SPACE     := $(EMPTY) $(EMPTY)
SLASH     := /
```

定义了变量 PROJ, 其值为 lab1。定义了变量 EMPTY, 其为空字符串。定义了变量 SPACE, 其值为两个空格。定义了变量 SLASH, 其值为斜杠。

```
V      := @
```

定义了变量 V, 值为 @。这个变量在后续的命令执行中用于控制命令是否显示在输出中。

```
#ifndef GCCPREFIX
GCCPREFIX := riscv64-unknown-elf-
#endif
```

如果变量 GCCPREFIX 未定义, 则将其设置为 riscv64-unknown-elf-。这个变量用于指定编译器的前缀。

```
ifndef QEMU
QEMU := qemu-system-riscv64
endif
```

如果变量 QEMU 未定义, 则将其设置为 qemu-system-riscv64。这个变量用于指定 QEMU 仿真器的名称。

```
.SUFFIXES: .c .S .h
```

定义了文件后缀名的规则, .c、.S 和 .h 文件被视为源文件。

```
.DELETE_ON_ERROR:
```

如果在执行过程中发生错误或中断, 删除目标文件。

```
HOSTCC      := gcc
HOSTCFLAGS  := -Wall -O2
```

定义了主机编译器的名称和编译选项。

```
GDB      := $(GCCPREFIX)gdb
```

定义了调试器的名称, 使用了前缀变量 GCCPREFIX。

```
CC      := $(GCCPREFIX)gcc
CFLAGS  := -mmodel=medany -std=gnu99 -Wno-unused -Werror
CFLAGS  += -fno-builtin -Wall -O2 -nostdinc $(DEFS)
CFLAGS  += -fno-stack-protector -ffunction-sections -fdata-sections
CTYPE   := c S
```

定义了编译器的名称和编译选项。CFLAGS 包含了一系列编译选项, 如代码模型、标准、警告设置等。CTYPE 定义了源文件的类型。

```
LD      := $(GCCPREFIX)ld
LDFLAGS := -m elf64lriscv
LDFLAGS += -nostdlib --gc-sections
```

定义了链接器的名称和链接选项。

```
OBJCOPY := $(GCCPREFIX)objcopy
OBJDUMP := $(GCCPREFIX)objdump
```

定义了目标文件复制和转储的工具名称。

```
COPY      := cp
MKDIR     := mkdir -p
MV        := mv
RM        := rm -f
AWK       := awk
SED       := sed
SH        := sh
TR        := tr
TOUCH     := touch -c
```

定义了文件复制、目录创建、文件移动、文件删除等命令的名称。

```
OBJDIR    := obj
BINDIR    := bin
```

定义了目标文件和二进制文件的存放目录。

```
ALLOBSJS      :=
ALLDEPS       :=
TARGETS       :=
```

定义了用于存放所有目标文件、依赖文件和目标的变量。

```
include tools/function.mk
```

包含了 tools/function.mk 文件, 该文件包含了 makefile 中使用的函数定义。

```
listf_cc = $(call listf,$(1),$(CTYPE))
```

定义了一个函数宏 listf_cc, 用于列出指定目录中指定类型的文件。

```
add_files_cc = $(call add_files,$(1),$(CC),$(CFLAGS) $(3),$(2),$(4))
create_target_cc = $(call create_target,$(1),$(2),$(3),$(CC),$(CFLAGS))
```

定义了两个函数宏 add_files_cc 和 create_target_cc, 用于添加文件和创建目标。

```
# for hostcc
add_files_host = $(call add_files,$(1),$(HOSTCC),$(HOSTCFLAGS),$(2),$(3))
create_target_host = $(call create_target,$(1),$(2),$(3),$(HOSTCC),$(HOSTCFLAGS))
```

这段代码继续定义了一些函数和变量, 与主机编译器相关。

add_files_host 函数: 它是在之前定义的 **add_files** 函数的基础上进行了扩展。**add_files_host** 函数用于添加主机编译器编译的文件。该函数将使用主机编译器和选项编译源文件, 并将生成的目标文件添加到文件列表中。

create_target_host 函数: 它是在之前定义的 **create_target** 函数的基础上进行了扩展。**create_target_host** 函数用于创建主机编译的目标。该函数将使用主机编译器和选项创建目标文件。这些函数和变量与主机编译器相关, 用于在构建过程中使用主机编译器编译文件。它们提供了用于添加主机编译文件和创建主机编译目标的功能。

```

cctype = $(patsubst %.$(2),%.$(3),$(1))

objfile = $(call toobj,$(1))

asmfile = $(call cctype,$(call toobj,$(1)),o,asm)

outfile = $(call cctype,$(call toobj,$(1)),o,out)

symfile = $(call cctype,$(call toobj,$(1)),o,sym)

# for match pattern

match = $(shell echo $(2) | $(AWK) '{for(i=1;i<=NF;i++){if(match("$(1)","^"$$i) "$$")
→{exit 1;}}}' ; echo $$?)

```

这段代码定义了一些函数和变量，用于处理文件名和模式匹配。

cctype 函数：它使用 patsubst 函数将文件名中的后缀名替换为指定的新后缀名。

objfile 函数：它调用 toobj 函数将文件名转换为目标文件名（以.o 结尾）。

asmfile、outfile 和 symfile 函数：它们使用 cctype 函数来生成特定类型的文件名。它们接受一个参数：\$(1) 代表文件名。asmfile 用于生成汇编文件名（以.asm 结尾），outfile 用于生成输出文件名（以.out 结尾），symfile 用于生成符号文件名（以.sym 结尾）。

match 函数：它使用 awk 命令来进行模式匹配。它接受两个参数：\$(1) 代表模式，\$(2) 代表要匹配的字符串。它返回一个值，如果匹配成功则为 0，否则为 1。该函数用于判断目标是否与指定的模式匹配。

```

# create kernel target
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call sym
→file,kernel)

$(call create_target,kernel)

```

这段代码用于构建内核（kernel）目标文件。首先，它使用 \$(call totarget,kernel) 宏来将目标名称转换为目标文件名。然后，它定义了一个规则，指定了目标文件（\$(kernel)）的依赖关系和构建命令。

依赖关系：

1. tools/kernel.ld：作为目标文件的一个依赖项，表示该文件需要先于目标文件构建完成。
2. \$(KOBJS)：这是一个变量，表示内核目标文件的一组依赖项。具体的依赖项可能在 Makefile 的其他地方定义。

构建命令：

1. @echo + ld \$@：打印构建命令，其中 \$@ 表示目标文件名。
2. \$(V)\$(LD) \$(LDFLAGS) -T tools/kernel.ld -o \$@ \$(KOBJS)：这是实际的构建命令。它使用 LD 变量指定的链接器（可能是 GNU ld）来链接目标文件。LDFLAGS 变量包含了链接器的一些选项。-T 选项指定链接脚本文件为 tools/kernel.ld，-o 选项指定输出文件为目标文件名，\$(KOBJS) 表示依赖项的列表。

3. `@$(OBJDUMP) -S $@ > $(call asmfile,kernel)`: 使用 **OBJDUMP** 变量指定的反汇编工具将目标文件反汇编, 并将结果重定向到一个文件中, 该文件由 `$(call asmfile,kernel)` 宏指定。
4. `@$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call symfile,kernel)`: 使用 **OBJDUMP** 变量指定的工具获取目标文件的符号表, 并使用 **SED** 变量指定的工具对符号表进行一些处理, 最后将结果重定向到一个文件中, 该文件由 `$(call symfile,kernel)` 宏指定。

```
# create ucore.img
UCOREIMG      := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel)
    $(OBJCOPY) $(kernel) --strip-all -O binary $@

$(call create_target,ucore.img)
```

这段代码用于创建 `ucore.img` 文件。首先, 它使用 `$(call totarget,ucore.img)` 宏来将目标名称转换为目标文件名。这个宏可能在 **Makefile** 的其他地方定义。然后, 它定义了一个规则, 指定了 `ucore.img` 目标文件的依赖关系和构建命令。

依赖关系:

1. `$(kernel)`: 作为目标文件的一个依赖项, 表示该文件需要先于目标文件构建完成。

构建命令:

1. `$(OBJCOPY) $(kernel) --strip-all -O binary $@`: 这是实际的构建命令。它使用 **OBJCOPY** 变量指定的工具, 将 `$(kernel)` 目标文件进行处理。`--strip-all` 选项表示移除所有符号表和调试信息, `-O binary` 选项表示将目标文件输出为二进制文件。最后, `$@` 表示目标文件名, 即 `ucore.img`。

```
.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#    $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
#    $(V)$(QEMU) \
#        -machine virt \
#        -nographic \
#        -bios default \
#        -device loader,file=$(UCOREIMG),addr=0x80200000
```

这段代码就是我们启动 `qemu` 的命令, 这段代码首先通过宏定义 `$(UCOREIMG)` `$(SWAPIMG)` `$(SFSIMG)` 的函数进行目标文件的构建, 然后使用 `qemu` 语句进行 `qemu` 启动加载内核。其中:

- `-machine virt` 表示将模拟的 64 位 RISC-V 计算机设置为名为 `virt` 的通用虚拟平台。
- `-nographic` 表示模拟器不需要提供图形界面, 而只需要对外输出字符流。
- 通过 `-bios` 可以设置 `Qemu` 模拟器开机时用来初始化的引导加载程序 (bootloader), 这里我们使用默认的 `Opensbi`
- 通过 `-device loader` 可以在 `Qemu` 模拟器开机之前将一个宿主机上的文件载入到 `Qemu` 的物理内存的指定位置中, `file` 和 `addr` 分别可以设置待载入文件的路径以及将文件载入到的 `Qemu` 物理内存上的物理地址。

12.1.1.5 参考资料

跟我一起写 makefile

12.1.2 附录：【原理】进程的属性与特征解析

操作系统负责进程管理，即从程序加载到运行结束的全过程，这个程序运行过程将经历从“出生”到“死亡”的完整“生命”历程。所谓“进程”就是指这个程序运行的整个执行过程。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，这就是进程控制块。进程与进程控制块是一一对应的。为此，ucore 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。

为了让多个程序能够使用 CPU 执行任务，需要设计用于进程管理的内核数据结构“进程控制块”。但到底如何设计进程控制块，如何管理进程？如果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、执行和动态执行过程。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。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。进程管理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，为了协调各进程对系统资源的使用，进程管理还需要做一些与进程协调有关的其他管理工作，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（后续相关实验涉及）等。下面逐一进行解析。

12.1.2.1 1. 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。

对于用户进程而言，操作系统是它的“上帝”，操作系统给了用户进程可以运行所需的资源，最基本的资源就是内存和 CPU。在实验二/三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。在有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。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。

12.1.2.2 2. 进程状态管理

用户进程有不同的状态（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“运行”状态，这时除了继续占用内存资源外，还占用了 CPU 资源；当进程由于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，由操作系统回收进程所占用的资源时，进程进入了“死亡”状态。

这些进程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服务必须由操作系统提供，而且在“运行”与“就绪”/“等待”状态之间的转换，涉及到保存和恢复进程的“执行现场”，也就是进程上下文，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。

12.1.2.3 3. 进程与线程

一个进程拥有一个存放程序和数据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分配资源（比如分配内存）并调度（比如分时使用 CPU）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、等待等），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、内核进程（可把 `ucore` 看成一个内核进程）、用户线程、内核线程的区别了。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` 内核中的所有内核线程共享一个内核地址空间和其他资源，所以这些内核线程从属于同一个唯一的内核进程，即 `ucore` 内核本身。理解了进程或线程的上述属性和特征，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，以下用户态的进程/线程统称为用户进程。

12.1.3 附录：【原理】用户进程的特征

12.1.3.1 从内核线程到用户进程

在实验四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。但实验四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？

但其实我们已经在实验四中看到了内核线程，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 and 外设等, 如何管理物理内存等)。

12.1.3.2 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, 所以需要提供某种机制, 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。

12.1.3.3 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具体实现。如果考虑操作系统的具体实现, 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。

12.1.3.4 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。

但在用户进程的执行过程中，具体在哪个时间段处于上述状态的呢？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。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。这样进程的状态转换形成了一个有限状态自动机。