

小型内核实现设计文档

上海电力大学

T202410256994312

蒙的全队

OS 原理（内核实现方向）

小型内核实现

目录

1. 项目简介	1
1.1 项目选题	1
1.2 设计内容及要求	1
2. 原理概述	1
2.1 操作系统启动流程	1
3. 详细设计	10
3.1 数据结构和算法	10
3.2 关键代码	13
3.3 程序运行说明	17
4. 测试与结果分析	18
4.1 内核、BOOT 和 PRINT 部分	18
4.2 内存管理部分	20
4.3 进程与异常部分	22
5. 总结	26
5.1 设计过程中遇到的问题及解决方法	26
5.2 团队合作情况	27
5.3 设计总结	27
5.4 心得体会	28

1. 项目简介

1.1 项目选题

我们队伍选择的赛道是 OS 原理赛道，选择的方向是小型内核实现，代码原型来自于北京航空航天大学操作系统内核实验。

本项目的框架以及部分源代码是北京航空航天大学操作系统内核实验自带的，但原型代码并没有实现内核。准确来说，北航提供的实验代码并不是一个可以运行的操作系统内核，需要设计者根据理解以及自己的设计需求，自行修改或者补全代码，以实现操作系统内核。

因此，在本项目中，我参赛队着眼于实现操作系统内核的基本功能，如内存管理、进程调度、异常处理等，在实验代码上进行修改，实现了相关功能。在报告中，图 1、图 2 等彩色图片来源于北航实验的 guide book，而图 16 等图片则由我队自行绘制。

1.2 设计内容及要求

理解 MIPS 体系架构，掌握操作系统启动流程，掌握 ELF 文件的结构和功能。了解内存访问原理，了解 MIPS 内存映射布局，掌握使用空闲链表的管理物理内存的方法，建立页表实现分页式虚存管理，实现内存分配和释放的函数，实现时钟中断与进程调度。

2. 原理概述

2.1 操作系统启动流程

从操作系统的角度看，boot loader 的总目标就是正确地调用内核来执行。另外，由于 boot loader 的实现依赖于 cpu 的体系结构，因此大多数 boot loader 都分为 stage1 和 stage2 两大部分。

在 stage1 时，此时需要初始化硬件设备，包括 watchdog timer、中断、时钟、内存等。Stage2 运行在 RAM 中，此时有足够的运行环境从而可以用 C 语言来实现较为复杂的功能。

本设计使用的操作系统环境是 Linux，在执行 BIOS 代码完成硬件初始化后，Linux 的 BIOS 会从 MBR 读取开机信息，Grub 和 Lilo 这两种开机管理

程序就是存在 MBR 里面。操作系统的启动步骤示意图如下图所示。

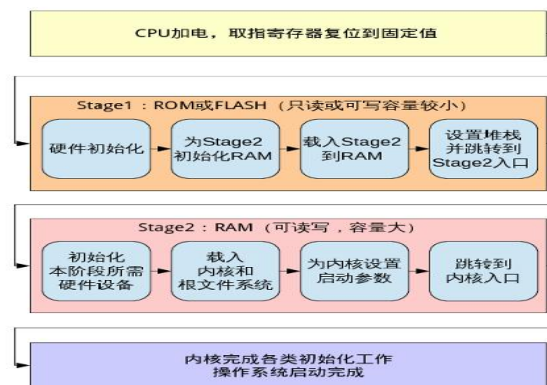


图 1 启动的步骤图

2.2.1 内存访问原理

在介绍内存访问原理前先介绍 MMU, TLB, cache 的相关概念。

MMU 是内存管理单元，MMU 是硬件设备，它的功能是把逻辑地址映射为虚拟地址，并提供了一套硬件机制来实现内存访问的权限检查

TLB 中文为翻译后备存储器，有的时候也叫相联存储器，或者快表，通常被安装在 MMU 的内部，其中记录了虚拟地址到物理地址的映射，并且访存速度要快于内存。

cache 记录了物理地址到物理地址数据的映射，同时它的访存速度也快于内存。

那么概述一下内存访问的流程。

1) 虚拟地址的产生：CPU 执行程序时，会根据程序的逻辑地址计算出页号和偏移量。

2) TLB 查询：CPU 将虚拟地址发送给翻译后备缓冲区（TLB），TLB 是一个高速缓存，用于存储最近使用的页表项。如果 TLB 命中，即找到了对应的页表项，就可以直接获取物理块号。

3) 页表查询：如果 TLB 没有命中，那么需要通过内存管理单元（MMU）访问页表。MMU 会根据页号在页表中查找对应的物理块号。这个过程可能涉及到操作系统的介入，以获取进程页表的起始地址和页表项长度。

4) 物理地址的计算：一旦获取到物理块号，结合偏移量，就可以计算出完整的物理地址。

5) Cache 查询：接下来，使用物理地址在 Cache 中查询。Cache 是一个更快的存储层，用于存储最近访问的数据。

6) Cache 命中：如果 Cache 命中，即在 Cache 中找到了对应的数据，那么就直接将数据返回给 CPU。

7) Cache 未命中: 如果 Cache 未命中, 那么需要访问主内存 (RAM) 来获取数据。同时, 根据 Cache 替换策略 (如 LRU, 最近最少使用), 可能需要将新获取的数据块放入 Cache, 并替换掉旧的数据块。

8) 数据返回: 无论是从 Cache 还是主内存获取的数据, 最终都会返回给 CPU, 以便继续执行程序。

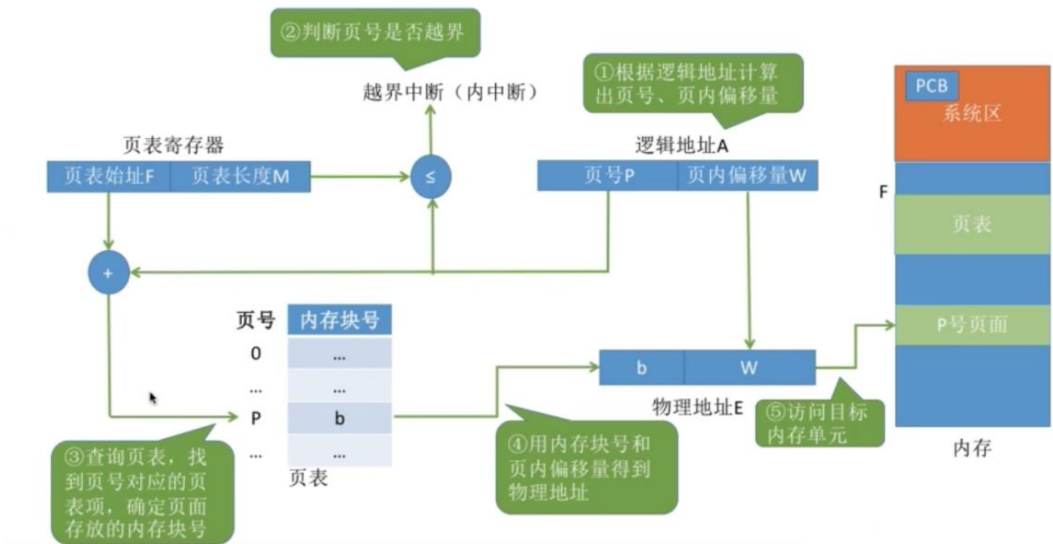


图 2 内存访问流程

2.2.2 MIPS 内存映射布局

内存映射布局如下图所示。

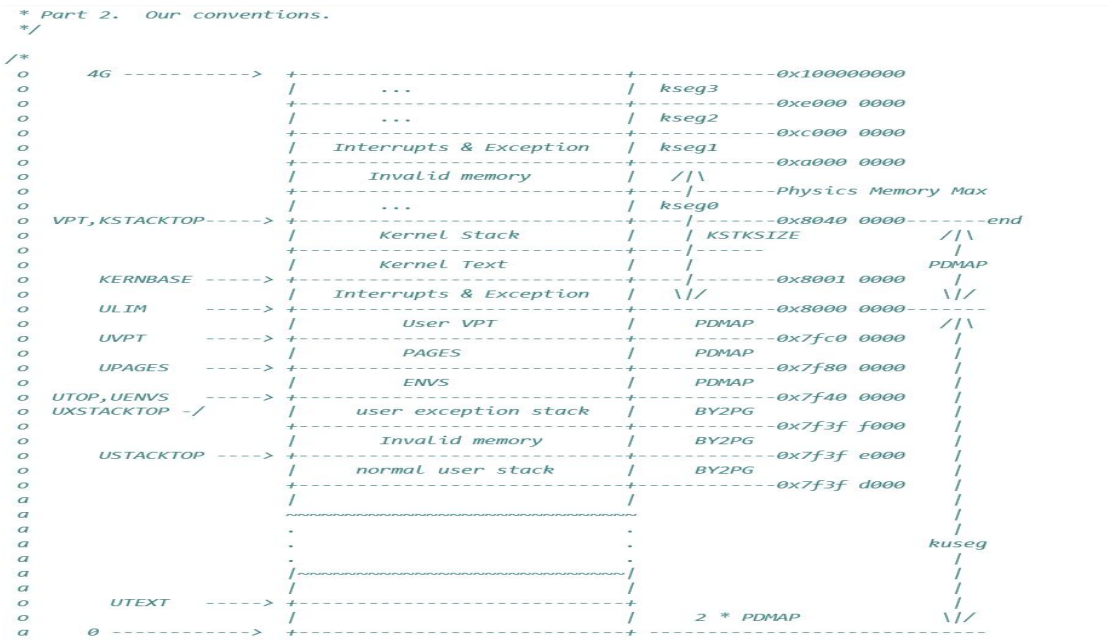


图 3 MIPS 内存映射布局

kuseg:(0x0000 0000—0x7FFF FFFF) 这一段是用户模式下可用的地址，大小为 2G,也就是 MIPS 约定的用户内存空间，需要通过 MMU 实现虚拟地址到物理地址的转换。

kseg0:(0x8000 0000—0x9FFF FFFF)：这一段是内核地址，其内存虚存地址到物理内存地址的映射转换不通过 MMU，使用时只需要将地址的最高位清零，这些地址就被转换为物理地址。也就是说，这段逻辑地址被连续地映射到物理内存的低端 512MB 空间。对这段地址的存取都会通过高速缓存（cached）。通常在没有 MMU 的系统中，这段代码用于存放大多数的程序和数据。对于有 MMU 的系统，操作系统内核会存放在这个区域。

kseg1:(0xA000 0000—0xBFFF FFFF)与 kseg0 类似，这段地址也是内核地址，将虚拟地址的高三位清零，就可以转换到物理地址，这段逻辑地址也是连续映射到物理内存的低端 512M 空间。当时 kseg1 不使用缓存，访问速度比较慢，但对硬件 I/O 寄存器来说也就不存在 Cache 一致性的问题了，这段内存通常被映射到 I/O 寄存器，用来实现对外设的访问。

kseg2:(0xC000 0000—0xFFFF FFFF)这段地址只能在内核态使用，并且需要 MMU 转换。

2.2.3 物理内存管理

(1) 初始化流程说明：

操作系统启动后要完成内存管理相关数据结构的初始化。主要通过以下三个函数实现。

```
mips_delete_memory() /*内存清空*/  
mips_vm_init()        /*初始化二级页表*/  
page_init()           /*页数据结构的初始化*/
```

(2) 内存控制块：

在 MIPS CPU 中，每一页的大小为 4KB。大多数的内存分配都是以页为单位进行分配的。此时我们就需要用 Page 数据结构来记录一页内存的相关信息。

```
1  typedef LIST_ENTRY(Page) Page_LIST_entry_t;  
2  
3  struct Page {  
4      Page_LIST_entry_t pp_link; /* free list link */  
5      u_short pp_ref;  
6  };
```

图 4 Page 数据结构

pp_ref 用来记录这个物理页的引用次数。pp_link 是当前节点中指向链表中下一节点的指针。在 include/queue.h 中定义了一系列对于链表的操作。

在 queue.h 中定义了两种数据结构：列表（LIST）和尾队列（Tail Queue）。

其中值得注意的是 List_ENTRY(type) 中的 **le_pre, 它是之前它的前一个节点的后一个节点，简而言之就是指向自己所在的节点，这个在补全 LIST 函数的时候起到关键作用。

（3）内存分配和释放：

首先，我们需要注意在 mm/pmap.c 中定义的与内存相关的全局变量：

```
1  u_long maxpa;           /* Maximum physical address */
2  u_long npage;           /* Amount of memory(in pages) */
3  u_long basemem;         /* Amount of base memory(in bytes) */
4  u_long extmem;          /* Amount of extended memory(in bytes) */
```

图 5 pmap.c 中的全局变量

在初始化的过程中设置为如下图所示：

```
void mips_detect_memory()
{
    /* Step 1: Initialize basemem.
     * (When use real computer, CMOS tells us how many kilobytes there are). */
    maxpa = 0x4000000;
    npage = 0x4000;
    basemem = 0x4000000;
    extmem = 0;
    // Step 2: Calculate corresponding npage value.

    printf("Physical memory: %dK available, ", (int)(maxpa / 1024));
    printf("base = %dK, extended = %dK\n", (int)(basemem / 1024),
           (int)(extmem / 1024));
}
```

图 6 mips_detect_memory()

我们在启动 Gxemul 时设置的仿真内存的大小是 64MB, 所以最大物理地址和基本内存大小都是 0x4000000, 存储内存的页数就是内存大小/页面大小（4KB），扩展内存设置为 0。

2.2.4 虚拟内存管理

（1）两级页表机制：

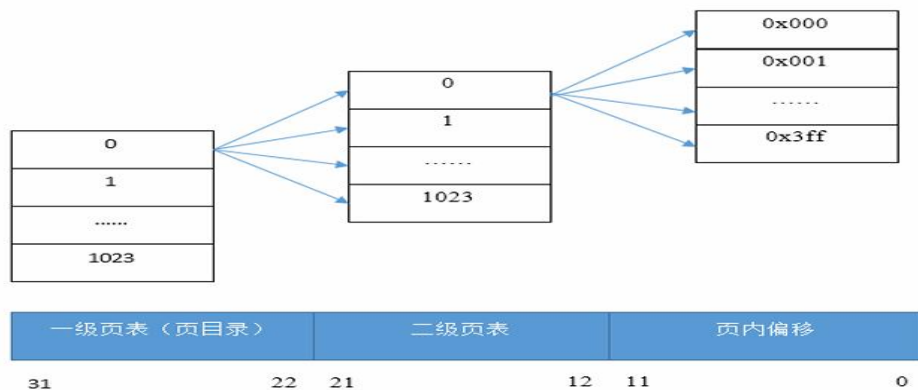


图 7 二级页表结构示意图

第一级表成为页目录 (page directory)，一共 1024 个页目录项，每个页目录项 32 位 (4Byte)，页目录项存储的值为其对应的二级页表入口的物理地址。整个页目录存放在一个页面 (4KB) 中，也就是我们在 `mips_vm_init` 函数中为其分配了相应的物理内存。第二级表成为页表 (page table)，每一张页表有 1024 个页表项，每个页表项 32 位 (4 Byte)，页表项存储的是对应页面的页框号 (20 位) 以及标志位 (12 位)。每张页表占用一个页面大小 (4KB) 的内存。

(2) 页目录自映射

对于二级页表来说，页目录有 1024 个页目录项，每个页目录项 4B,需要 4KB 内存，二级页表每个页表有 1024 个页表项，一共有 1024 个二级页表，每个页表项 4B,共需要 4MB 内存。那么理论上需要 4KB+4MB 内存。但实际上内存一共只有 1M 个页，每一页的地址需要 4B 标识，所以应该只要 4MB 就可以标识 1M 个页。那么为什么会需要 4KB 的内存？

这是因为内存一共 4GB,按字节编址需要 4G 个地址，也就是 32 位的二进制才能标识一个地址。在二级页表中，我们抽象地分成一级页表和二级页表，但实际上是读取一个虚拟地址 (4B) 的高 10 位作为一级页表，次高 10 位作为 2 级页表，最后 12 位作为偏移量。因此，一级页表和二级页表本质上是一张页表，只是我们隐式地将这张页表分成的两张页表，因此只需要 4MB 内存就可以映射 4GB 内存。由于 4MB 的页表在内存中，而 4MBd 的大小刚好是一张二级页表可以映射的内存地址，因而内存会有 4MB 不能被占用，这 4MB 还要映射 1M 个物理页。示意图如下：

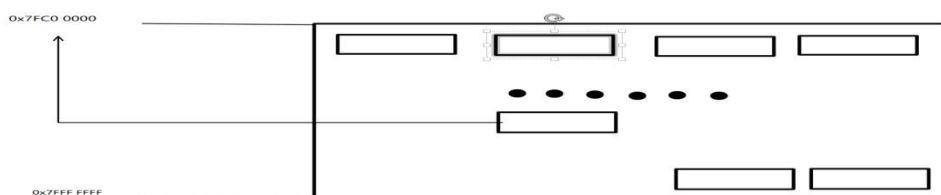


图 8 页目录示意图

(3) 创建页表

将虚拟地址转换为物理地址的过程中，如果这个虚拟地址所对应的二级页表不存在，有时，我们可能需要为这个虚拟地址创建一个新的页表。我们需要申请一页物理内存来存放这个页表，然后将他的物理地址赋值给对应的页目录项，最后设置好页目录项的权限位即可。

2.3.1 进程

进程是基本的分配单元，也是基本的执行单元。进程控制块（PCB）是系统为了管理进程设置的一个专门的数据结构，它用来记录进程的外部特征，描述进程的运动变化过程。系统利用 PCB 来控制和管理进程，所以 PCB 是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的。PCB 的数据结构以及信息将在后面介绍。

在实际生活当中，一台电脑中会有很多执行不同任务的进程同时存在，所以需要设置进程标识（id）来对每个进程加以区分。

在对不同进程加以区分后，就可以创建进程了，创建进程的流程如下：

①申请一个空闲的 PCB

②手动设置进程拥有的所有信息，需要手动初始化 PCB

③为新进程分配资源，为新进程的程序和数据以及用户栈分配必要的内存空间。

④将 PCB 从空闲链表中除去，即可投入使用。

注：此处指的创建进程是指系统创建进程，并非 fork 等进程创建进程的方式。

创建进程的第②步是关键，这里涉及到了初始化新进程的地址空间以及从用户态提升到内核态的问题。众所周知，操作系统当中有用户态和内核态这两个概念，内核态也就是特权态，mips 的特权指令都会陷入内核态。本项目的布局是 2G/2G 模式，即用户态和内核态都占用 2G。对于不同的进程而言，在虚拟地址 ULIM 以上的地方，虚实映射的关系是一样的，2G 的虚拟空间都是由内核管理，在这种布局模式下，并没有严格意义上的内核进程，因为即使进入了内核态，地址空间没有变化，并不会切换 CR3 寄存器。

那么，当一个进程从用户态提升到内核态的时候，进程其实并没有切换到所谓的内核进程，而是改变了进程的权限，让这个进程临时拥有了内核的权限。换言之，每个完整进程都有成为这种所谓的“临时内核”的资格，都可以发出申请成为内核态下的进程。有一种申请方式就叫做“系统调用”。

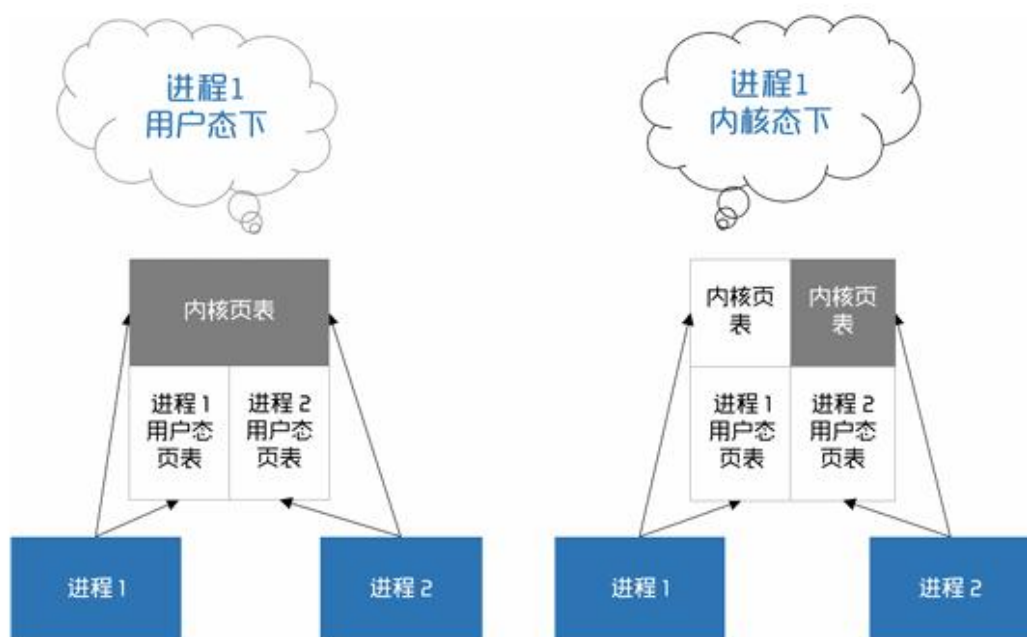


图 9 进程页表和内核页表

除此之外，还需要在空闲进程链表里面设置 status。在讲这点之前，请先看下方的 SR 寄存器示意图。

SR Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
CU3	CU2	CU1	CU0	0	RE	0	BEV	TS	PE	CM	PZ	SwC	IsC			
15								8	7	6	5	4	3	2	1	0
IM							0	KUo	IEo	KUp	IEp	KUc	IEc			

图 10 SR 寄存器结构

先解释一下该图的某些关键点：

28 设置为 1，表示处在用户模式

12 设置为 1，表示 4 号中断可以被响应

R3000 的 SR 寄存器的低 6 位的结构是二重栈。KUo 和 IEo 是一组，每当中断的时候，硬件会自动将 KUp 和 IEp 的数值拷贝到这里；KUp 和 IEp 也是一组，当中断时，硬件会把 KUc 和 IEc 的数值拷贝到这里。

KU 的含义是是否处在内核模式，1 则在内核模式；IE 表示中断是否开启，0 闭 1 开。

每当 rfe 指令调用时，就会进行上面操作的逆操作。所以在本项目当中，我们设置了 status 后六位为 000100b，每当运行第一个进程之前，运行代码到 rfe 指令的时候，就会把 KUp 和 IEp 拷贝回 KUc 和 IEc，令 status 为 000001b，最后两位 KUc 和 IEc 为【0，1】，表示开启中断。

创建进程，实际上就是实现对函数的封装、分配一个新的 Env 结构体、设

置 PCB，并将二进制代码载入到对应地址空间。

在本项目中，`env_run` 是进程运行使用的基本函数，包含保存当前进程上下文和回复要启动的进程上下文两个功能。在这里，运行一个新进程并不只是单纯的进程运行，而是意味着进程切换。在进程切换时，自然需要保存被切换的进程的信息被保留，以便下一次启动时还能从离开的地方继续，需要保存的信息包括进程本身的状态和进程周围的状态。

进程本身的状态其实就是 PCB 中的那些，而进程的上下文（寄存器）状态保存的地方是 TIMES-TACK 区域。

```
struct Trapframe *old;

old = (struct Trapframe *) (TIMESTACK - sizeof(struct Trapframe));
```

图 11 TIMES-TACK 区域

上图中的 `old` 就是当前进程上下文所存放的区域。

2.3.2 中断与异常

在内核的设计过程中，进程的中断和异常时必须考虑到的，一旦发生了进程的中断或者是异常，内核应当有能力进行处理。

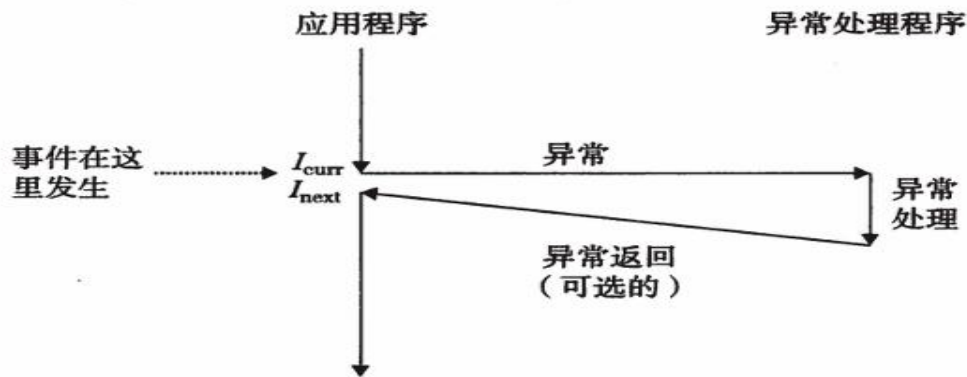


图 12 异常处理流程

异常的产生和返回涉及以下几个要点：异常的分发、异常向量组、时钟中断和进程调度。

异常的分发流程：

- ①取得异常码，这是区别不同异常的重要标志
- ②以得到的异常码作为索引去 `exception_handlers` 数组中找到对应的中断处理函数。
- ③跳转到对应的中断处理函数中，从而响应异常，将异常交给了对应的异常处理函数去处理。

异常向量组:

实现对全局变量数组的初始化工作, 其实就是把响应的处理函数的地址填到对应数组项当中

时钟中断:

时钟中断和时间片轮转算法密切相关。通过定时器, CPU 可以知晓一个进程的时间片何时结束, 当时钟中断产生时候, 当前运行的进程被挂起, 在调度队列中选取一个合适的进程进行, 即为**进程调度**。

进程调度:

在本项目中, 选取时间片轮转算法作为进程调度算法。将优先级设置为时间片大小, 1 表示 1 个时间片长度, 2 表示 2 个时间片长度, 以此类推。每当一个进程状态变为 ENV_RUNNABLE, 就将其插入第一个就绪状态进程列表。调用 sched_yield 函数时候, 先判断当前时间片是否用完, 如果用完了就将其插入另一个就绪状态进程链表。之后再判断当前就绪状态进程链表是否为空, 若是则切换到另一个链表。

3. 详细设计

3.1 数据结构和算法

(1) 操作系统启动流程

补全 scse_03.lds :

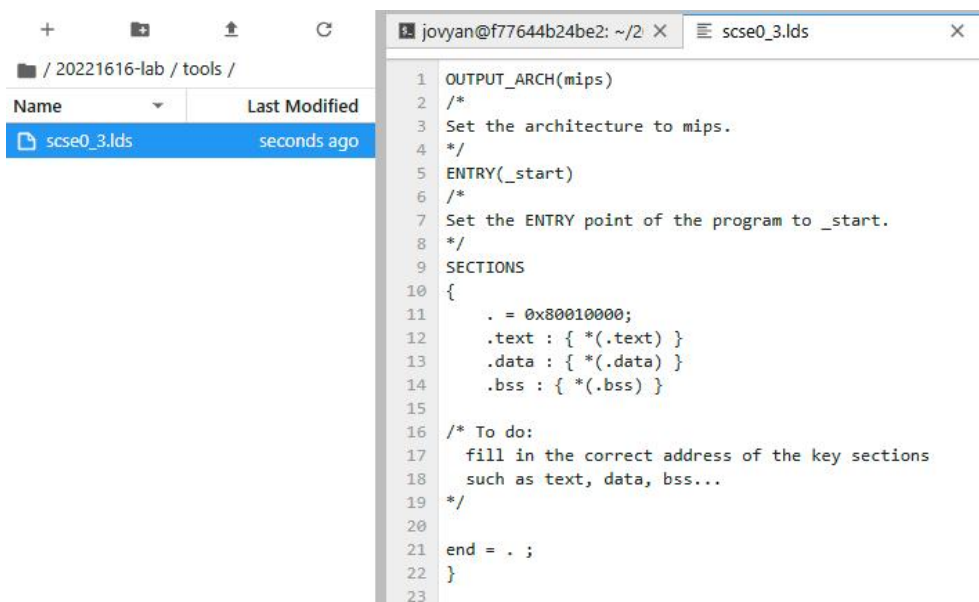
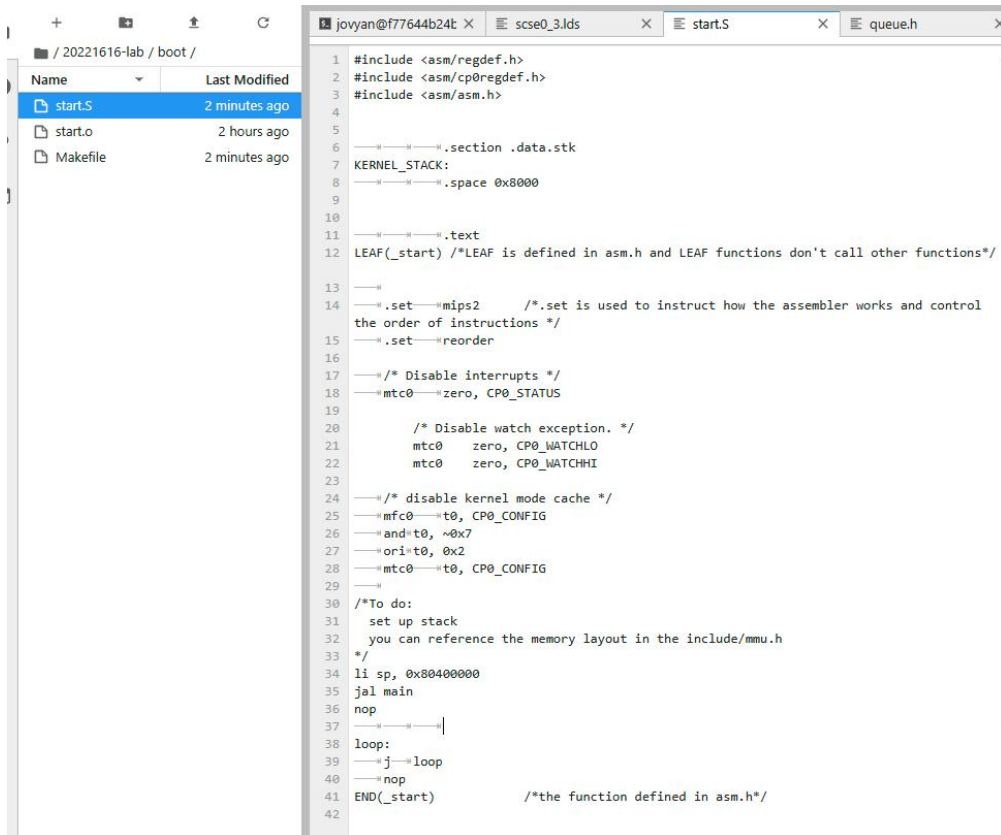


图 13 scse_03.lds 的补全

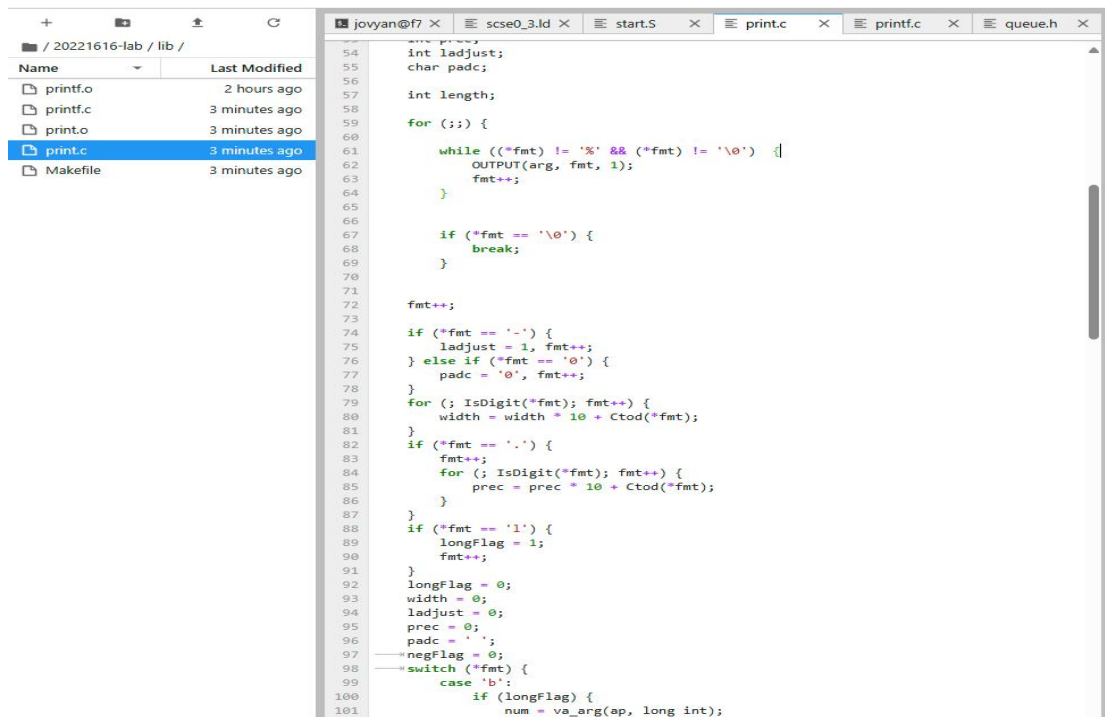
完成 boot/start.S 的空缺部分：



```
1 #include <asm/regdef.h>
2 #include <asm/cp0regdef.h>
3 #include <asm/asm.h>
4
5
6 ---.section .data.stk
7 KERNEL_STACK:
8 ---.space 0x8000
9
10
11 ---.text
12 LEAF(_start) /*LEAF is defined in asm.h and LEAF functions don't call other functions*/
13
14 ---.set mips2 /*.set is used to instruct how the assembler works and control
15 the order of instructions */
16 ---.set reorder
17
18 /* Disable interrupts */
19 mtc0 zero, CP0_STATUS
20
21 /* Disable watch exception. */
22 mtc0 zero, CP0_WATCHLO
23 mtc0 zero, CP0_WATCHHI
24
25 /* disable kernel mode cache */
26 mfc0 t0, CP0_CONFIG
27 and t0, ~0x7
28 ori t0, 0x2
29 mtc0 t0, CP0_CONFIG
30
31 /*To do:
32 set up stack
33 you can reference the memory layout in the include/mmu.h
34 */
35 li sp, 0x80400000
36 jal main
37 nop
38
39 loop:
40 j loop
41 nop
42 END(_start) /*the function defined in asm.h*/
```

图 14 完成 boot/start.S 的空缺部分

补全 lib/print.c 中的 lp_Print() 函数中的缺失部分来实现字符的输出。



```
54 int ladjust;
55 char padc;
56
57 int length;
58
59 for (;;) {
60
61     while ((*fmt) != '%' && (*fmt) != '\0') {
62         OUTPUT(arg, fmt, 1);
63         fmt++;
64     }
65
66     if (*fmt == '\0') {
67         break;
68     }
69
70     fmt++;
71
72     if (*fmt == '-') {
73         ladjust = 1;
74     } else if (*fmt == '0') {
75         padc = '0';
76     }
77
78     for (; IsDigit(*fmt); fmt++) {
79         width = width * 10 + Ctod(*fmt);
80     }
81
82     if (*fmt == '.') {
83         fmt++;
84         for (; IsDigit(*fmt); fmt++) {
85             prec = prec * 10 + Ctod(*fmt);
86         }
87     }
88
89     if (*fmt == 'l') {
90         longFlag = 1;
91     }
92
93     longFlag = 0;
94     width = 0;
95     ladjust = 0;
96     prec = 0;
97     padc = ' ';
98     negFlag = 0;
99     switch (*fmt) {
100     case 'b':
101         if (longFlag) {
102             num = va_arg(ap, long int);
```

图 15 lib/print.c 中函数的补全

(2) 队列 (LIST)

结构定义:

LIST_HEAD(name, type): 定义一个列表头, 包含一个指向第一个元素的指针。

LIST_ENTRY(type): 定义一个列表元素的节点, 包含指向下一个元素的指针和指向前一个元素指针的地址。

操作:

LIST_EMPTY(head): 检查列表是否为空。

LIST_FIRST(head): 返回列表的第一个元素。

LIST_FOREACH(var, head, field): 遍历列表中的所有元素。

LIST_INIT(head): 初始化列表头, 使其成为一个空列表。

LIST_INSERT_AFTER(listelm, elm, field): 在指定元素后插入一个新元素。

LIST_INSERT_HEAD(head, elm, field): 在列表头部插入一个新元素。

LIST_INSERT_TAIL(head, elm, field): 在列表尾部插入一个新元素

LIST_NEXT(elm, field): 获取当前元素的下一个元素。

LIST_REMOVE(elm, field): 从列表中移除一个元素。

尾队列 (Tail Queue)

结构定义:

TAILQ_HEAD(name, type): 定义一个尾队列头, 包含指向第一个元素和最后一个元素的指针。

TAILQ_ENTRY(type): 定义一个尾队列元素的节点, 包含指向下一个元素的指针和指向前一个元素指针的地址。

操作:

代码中未提供尾队列的具体操作宏定义, 但通常包括插入、删除和遍历操作, 类似于列表操作。

Page_list 结构大致如下:

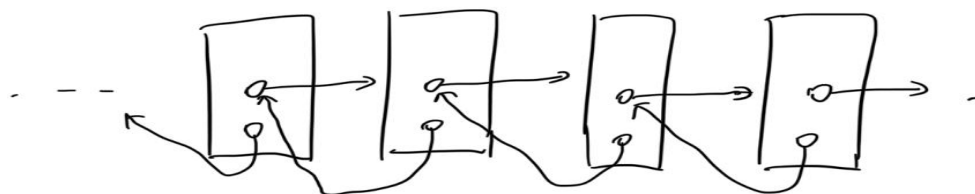


图 16 page_list 结构

Page 结构体只是信息的载体，它只代表了相应物理页内存的信息，它本身并不是物理内存页。通过 page2ppn 函数将指针转换为页号，再将转成虚拟地址，将虚拟地址右移 12 位后加上页目录的起始地址就是页的虚存地址，虚存地址存放的就是对应页号的实际物理地址。

(3) PCB（进程控制块）

```
struct Env {
    struct Trapframe env_tf;
    LIST_ENTRY(Env) env_link;
    u_int env_id;
    u_int env_parent_id;
    u_int env_status;
    Pde *env_pgdir;
    u_int env_cr3;
};
```

图 17 PCB 结构

上图展示的是本项目的 PCB 结构，下面对这些域作简单说明。

①env_tf: 在进程因为时间片用光不再运行时，将其当时的进程上下文环境保存在该变量当中。当从用户模式切换到内核模式时，内核也会保存进程上下文，因此进程返回时上下文从中恢复

②env_link: 使用它和 env_free_list 来构造空闲进程链表

③env_id: 每个进程独一无二的标识符

④env_parent_id: 存储了创建本进程的 id

⑤env_status: 分为三种

ENV_FREE: 表明进程不活动，PCB 处于进程空闲链表

ENV_NOT_RUNNABLE: 表明进程处于阻塞

ENV_RUNNABLE: 表明进程就绪，正在等待被调度，但是并不能代表进程一定正在运行

⑥env_pgdir: 保存该进程页目录的虚拟地址

⑦env_cr3: 保存该进程页目录的物理地址

⑧env_sched_link: 构造就绪状态进程链表

⑨env_pri: 保存了该进程的优先级

3.2 关键代码

(1) 内核、BOOT 和 PRINT 部分

Scse_03.ldp 代码见图 12。以下是代码的解释。

这段代码是为 MIPS 架构编写的汇编链接脚本，主要功能是指定程序的内存布局和入口点：

OUTPUT_ARCH(mips): 设置目标架构为 MIPS。

ENTRY(_start): 定义程序的入口点为 _start, 程序从这个位置开始执行。

SECTIONS: 定义程序的各个段 (如 .text、.data、.bss) 的内存布局:

.text 段存储可执行代码。

.data 段存储已初始化的变量。

.bss 段存储未初始化的变量。

. = 0x80010000: 将程序的起始地址设置为 0x80010000。

end = .: 定义一个符号 end, 表示程序的结束位置。

简单来说, 这段代码主要配置了程序的内存布局和入口点。

Start.S 代码见图 13。以下是对代码的解释。

这段代码的作用是设置堆栈并调用 main 函数:

li sp, 0x80400000: 将栈指针 sp 设置为 0x80400000, 即为程序分配栈空间。

jal main: 跳转到 main 函数并保存返回地址。

nop: 执行一个空操作, 通常用于避免流水线冲突。

总体来说, 代码先配置栈, 再跳转并执行 main 函数。

Print.c 代码见图 14。以下是对代码的解释。

这段代码实现了格式化输出的解析过程, 主要用于处理类似 printf 的格式化字符串。具体流程如下:

扫描格式化字符串: 循环读取字符, 直到遇到 % 或字符串结束。每个非格式字符被输出或处理。

遇到 % 后处理格式说明符:

解析格式标志, 如左对齐 (-) 和填充字符 (0)。

解析字段宽度和精度 (如果存在数字或 .)。

处理长整型标志 (l)。

重置并初始化格式化标志和参数: 在每次格式说明符解析后, 重置相关标志和变量, 以准备处理下一个格式。

总体来说, 这段代码是一个格式化输出字符串解析器, 用于提取各种格式选项 (如宽度、精度等), 并为后续的输出做好准备。

(2) 内存管理部分

```
static inline u_long
```

```
page2ppn(struct Page *pp)/*将结构体指针转换为物理页号*/
```

```
{
```

```
    return pp - pages;
```



```

}

/* Get the physical address of Page 'pp'.*/
static inline u_long
page2pa(struct Page *pp)/*将结构体指针转换为物理地址*/
{
    return page2ppn(pp) << PGSHIFT;
}

/* Get the Page struct whose physical address is 'pa'.*/
static inline struct Page *
pa2page(u_long pa)/*将物理地址转换为页结构体指针*/
{
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa: %x", pa);
    }

    return &pages[PPN(pa)];
}

/* Get the kernel virtual address of Page 'pp'. */
static inline u_long
page2kva(struct Page *pp)/*将物理地址转换为内核虚拟地址*/
{
    return KADDR(page2pa(pp));
}

int
page_alloc(struct Page **pp)
{
    struct Page *ppage_temp;

    /* Step 1: Get a page from free memory. If fails, return the error code.*/
    if (LIST_EMPTY(&page_free_list)) {
        return -E_NO_MEM;
    }
}

```

```

    }
    ppage_temp = LIST_FIRST(&page_free_list);
    LIST_REMOVE(ppage_temp,pp_link);
    /* Step 2: Initialize this page.
     * Hint: use `bzero`. */
    bzero((void*)page2kva(ppage_temp),BY2PG);//清空的是对应的 4k 空间的
    那一页，而不是我们存储页信息的结构体
    *pp = ppage_temp;
    return 0;

```

```

}

```

```

/*Overview:

```

```

    Release a page, mark it as free if it's `pp_ref` reaches 0.

```

```

Hint:

```

```

    When to free a page, just insert it to the page_free_list.*/

```

```

void

```

```

page_free(struct Page *pp)

```

```

{

```

```

    /* Step 1: If there's still virtual address refers to this page, do nothing. */

```

```

    if (pp->pp_ref>=1) return;

```

```

    /* Step 2: If the `pp_ref` reaches to 0, mark this page as free and return. */

```

```

    if (pp->pp_ref==0) {

```

```

        LIST_INSERT_HEAD(&page_free_list,pp,pp_link);

```

```

        return;
    }

```

```

    /* If the value of `pp_ref` less than 0, some error must occurred before,

```

```

        * so PANIC !!! */

```

```

    panic("cgh:pp->pp_ref is less than zero\n");

```

```

}

```

(3) 进程与中断部分

```

lw k0,TF_STATUS(K0) # 恢复 CP0_STATUS 寄存器

```

```

mtc0 k0,CP0_STATUS

```

j k l

rfe

该段代码在运行第一个进程前是一定要执行的，而 rfe 指令是用于进行操作的逆操作的。这段代码的目的是开启中断，当第一个进程成功运行后，这时候操作系统可以正常响应中断。

3.3 程序运行说明

程序运行环境：希冀平台

包含以下几种文件：.o 文件，.c 文件，Makefile 文件，.mk 文件，ELF 文件，Gxemul 程序等。

Gxemul 程序通过 makefile 完成编译链接。

```
1  # Main makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written by Zhu Like ( zlike@cse.buaa.edu.cn )
5  #
6
7  drivers_dir      := drivers
8  boot_dir         := boot
9  init_dir         := init
10 lib_dir          := lib
11 tools_dir        := tools
12 vmlinux_elf       := gxemul/vmlinux
13
14 link_script       := $(tools_dir)/scse0_3.lds
15
16 modules           := boot drivers init lib
17 objects           := $(boot_dir)/start.o
18                   $(init_dir)/main.o
19                   $(init_dir)/init.o
20                   $(drivers_dir)/gxconsole/console.o
21                   $(lib_dir)/*.o
22
23 .PHONY: all $(modules) clean
24
25 all: $(modules) vmlinux
26
27 vmlinux: $(modules)
28         $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)
29
30 $(modules):
31         $(MAKE) --directory=$@
32
33 clean:
34         for d in $(modules); \
35         do \
36                 $(MAKE) --directory=$$d clean; \
37         done; \
38         rm -rf *.o *~ $(vmlinux_elf)
39
40 include include.mk
```

图 18 makefile 文件

通过执行 `gxemul -E testmips -C R3000 -M 64 vmlinux` 实现内核启动和内存的初始化。

4. 测试与结果分析

4.1 内核、BOOT 和 PRINT 部分

在补全 `scse_03.ldp` 文件之后执行如图 16 的操作，运行结果如下。

```
jovyan@f77644b24be2: /20221616-lab/gxemul$ readelf -S vmlinux
There are 14 section headers, starting at offset 0x8f6c:

Section Headers:
[Nr] Name                Type              Addr             Off             Size            ES Flg Lk  Inf Al
[ 0]                      NULL             00000000         000000         000000         00 000 0   0  0
[ 1] .text                  PROGBITS          00000000         000080         000950         00 WAX 0   0 16
[ 2] .reginfo              MIPS_REGINFO      00000950         0009d0         000018         18  A 0   0  4
[ 3] .rodata.str1.4         PROGBITS          00000968         0009e8         0000a2         01 AMS 0   0  4
[ 4] .rodata                PROGBITS          00000a10         000a90         000200         00  A 0   0 16
[ 5] .data                  PROGBITS          00000c10         000c90         000000         00  WA 0   0 16
[ 6] .data.stk              PROGBITS          00000c10         000c90         008000         00  WA 0   0  1
[ 7] .bss                   NOBITS            00008c10         008c90         000000         00  WA 0   0 16
[ 8] .pdr                   PROGBITS          00000000         008c90         0001a0         00  A 0   0  4
[ 9] .mdebug.abi32          PROGBITS          00000000         008e30         000000         00  A 0   0  1
[10] .comment                PROGBITS          00000000         008e30         0000c8         00  A 0   0  1
[11] .shstrtab               STRTAB            00000000         008ef8         000072         00  A 0   0  1
[12] .symtab                 SYMTAB            00000000         00919c         000250         10 13 24  4
[13] .strtab                 STRTAB            00000000         0093ec         0000c2         00  A 0   0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)
```

图 19 readelf -S vmlinux

在补全 `start.S` 文件的代码之后执行下图所示的操作，运行结果如下。

```
jovyan@f77644b24be2: /20221616-lab$ gxemul -E testmips -C R3000 -M64 gxemul/vmlinux
GXemul 0.4.6 Copyright (C) 2003-2007 Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
    simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
    using nameserver 210.35.88.5
machine "default":
    memory: 64 MB
    cpu0: R3000 (I+D = 4+4 KB)
    machine: MIPS test machine
    loading gxemul/vmlinux
    starting cpu0 at 0x80010000

-----

main.: main imain.:    main is start ...

s start ...

tart ...
```

图 20 gxemul -E testmips -C R3000 -M64 gxemul/vmlinux

补全 `print.c` 文件的代码之后运行。最后通过 `git add`、`commit`、`push` 提交测评。

```

jovyan@77644b24be2: ~/20221616-lab$ make
make --directory=boot
make[1]: Entering directory '/home/jovyan/20221616-lab/boot'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/jovyan/20221616-lab/boot'
make --directory=drivers
make[1]: Entering directory '/home/jovyan/20221616-lab/drivers'
make --directory=gxconsole
make[2]: Entering directory '/home/jovyan/20221616-lab/drivers/gxconsole'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/jovyan/20221616-lab/drivers/gxconsole'
make[1]: Leaving directory '/home/jovyan/20221616-lab/drivers'
make --directory=init
make[1]: Entering directory '/home/jovyan/20221616-lab/init'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/jovyan/20221616-lab/init'
make --directory=lib
make[1]: Entering directory '/home/jovyan/20221616-lab/lib'
/OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -I./ -I../ -I../include/ -c print.c
make[1]: Leaving directory '/home/jovyan/20221616-lab/lib'
/OSLAB/compiler/usr/bin/mips_4KC-ld -o gxemul/vmlinux -N -T tools/scse0_3.lds boot/start.o init/main.o init/init.o drivers/gxconsole
le/console.o lib/*.o
jovyan@77644b24be2: ~/20221616-lab$ gxemul -E testmips -C R3000 -M64 gxemul/vmlinux
GXemul 0.4.6 Copyright (C) 2003-2007 Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
using nameserver 210.35.88.5
machine "default":
memory: 64 MB
cpu0: R3000 (I+D = 4+4 KB)
machine: MIPS test machine
loading gxemul/vmlinux
starting cpu0 at 0x80010000

-----

main.c: main is start ...

init.c: mips_init() is called

panic at %s:%d: ~~~~~

```

图 21 gxemul -E testmips -C R3000 -M64 gxemul/vmlinux

```

remote: make[2]: Entering directory '/usr/src/workdir/drivers/gxconsole'
remote: rm -rf *.o *
remote: make[2]: Leaving directory '/usr/src/workdir/drivers/gxconsole'
remote: make[1]: Leaving directory '/usr/src/workdir/drivers'
remote: make[1]: Entering directory '/usr/src/workdir/init'
remote: rm -rf *.o *
remote: make[1]: Leaving directory '/usr/src/workdir/init'
remote: make[1]: Entering directory '/usr/src/workdir/lib'
remote: rm -rf *.o *
remote: make[1]: Leaving directory '/usr/src/workdir/lib'
remote: Building the project.....
remote: make --directory=boot
remote: make[1]: Entering directory '/usr/src/workdir/boot'
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -I../include/ -c start.S
remote: make[1]: Leaving directory '/usr/src/workdir/boot'
remote: make --directory=drivers
remote: make[1]: Entering directory '/usr/src/workdir/drivers'
remote: make --directory=gxconsole
remote: make[2]: Entering directory '/usr/src/workdir/drivers/gxconsole'
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -c -o console.o console.c
remote: make[2]: Leaving directory '/usr/src/workdir/drivers/gxconsole'
remote: make[1]: Leaving directory '/usr/src/workdir/drivers'
remote: make --directory=print
remote: make[1]: Entering directory '/usr/src/workdir/init'
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -I../include -c
init.c
remote: init.c: In function 'mips_init':
remote: init.c:17: warning: implicit declaration of function 'abcd'
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -DTEST=abcd -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -I../include -c
main.c
remote: make[1]: Leaving directory '/usr/src/workdir/init'
remote: make --directory=lib
remote: make[1]: Entering directory '/usr/src/workdir/lib'
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -I./ -I../ -I../include/ -c
print.c
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abisalls -fno-builtin -Wa,-xgot -Wall -fPIC -I./ -I../ -I../include/ -c
print.c
remote: make[1]: Leaving directory '/usr/src/workdir/lib'
remote: make --directory=/usr/src/testdir30574
remote: make[1]: Entering directory '/usr/src/testdir30574'
remote: make[1]: Nothing to be done for 'all'.
remote: make[1]: Leaving directory '/usr/src/testdir30574'
remote: /OSLAB/compiler/usr/bin/mips_4KC-ld -o gxemul/vmlinux -N -T tools/scse0_3.lds boot/start.o init/main.o init/init.o drivers
/gxconsole/console.o lib/*.o /usr/src/testdir30574/*.o
remote: End build at Thu Dec 19 14:07:15 CST 2024
remote: [ PASSED 5 ]
remote: [ TOTAL 5 ]
remote: [ You have passed all testcases of extra printf. ]
remote: [ You got 100 (of 100) this time. Thu Dec 19 14:07:25 CST 2024 ]
remote:
remote: >>>>> Collecting autotest results >>>>>
remote: Switched to a new branch 'lab1-result'
remote: Branch lab1-result set up to track remote branch lab1-result from origin.
remote: Already up-to-date.
remote: [lab1-result 4d8fc43] Judgement for lab1 at 2024-12-19T14:07:25+0800
remote: 1 file changed, 133 insertions(+)
remote: create mode 100644 log/2024-12-19T14:06:59+0800.log
remote: To git@localhost:20221616-lab
remote: ea36304..4d8fc43 lab1-result -> lab1-result
remote: Please find the autotest log in lab1-result branch.
remote:
remote: [ Congratulations! You have passed the current lab. ]
remote: Switched to a new branch 'lab2'
remote: Branch lab2 set up to track remote branch lab2 from origin.
remote: [ lab2 already exists. ]
remote: To 192.168.1.112:20221616-lab
remote: 0ce1900..d42880e lab1 -> lab1
jovyan@77644b24be2: ~/20221616-lab$

```

图 22 git 上传测试

4.2 内存管理部分

```
167 void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
168 {
169     int i, va_temp;
170     Pte *pgtable_entry;
171
172     /* Step 1: Check if `size` is a multiple of BY2PG. */
173     if (size % BY2PG != 0) {
174         panic("Size must be a multiple of BY2PG\n");
175         return;
176     }
177     /* Step 2: Map virtual address space to physical address. */
178     /* Hint: Use `boot_pgdir_walk` to get the page table entry of virtual address `va`. */
179     for (i = 0; i < size; i += BY2PG) {
180         // 获取对应虚拟地址 va+i 的页表项指针
181         pgtable_entry = boot_pgdir_walk(pgdir, va + i, 1);
182         if (pgtable_entry == NULL) {
183             panic("boot_pgdir_walk failed\n");
184             return;
185         }
186         // 设置页表项，映射到对应的物理地址，并设置权限位
187         *pgtable_entry = (pa + i) | (perm | PTE_V);
188     }
189 }
190 }
191
```

图 23 boot_map_segment

```
void mips_detect_memory()
{
    /* Step 1: Initialize basemem.
     * Base memory is set to 64MB (64 * 1024 * 1024 bytes).
     */
    basemem = 64 * 1024 * 1024;

    /* Step 2: Set maxpa to basemem (assuming the system only has 64MB physical memory). */
    maxpa = basemem;

    /* Step 3: Calculate npage.
     * Each page is 4KB, so npage = maxpa / 4KB.
     */
    npage = maxpa / (4 * 1024);

    /* Step 4: Calculate extmem.
     * Since the entire memory is considered base memory, extmem = 0.
     */
    extmem = 0;

    /* Print memory information. */
    printf("Physical memory: %dK available, ", (int)(maxpa / 1024));
    printf("base = %dK, extended = %dK\n", (int)(basemem / 1024),
           (int)(extmem / 1024));
}
```

图 24 mips_detect_memory


```

341 int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
342 {
343     Pte *pgtable_entry;
344     u_int PERM = perm | PTE_V;
345
346     /* Step 1: Get corresponding page table entry. */
347     if (pgdir_walk(pgdir, va, 1, &pgtable_entry) < 0) {
348         return -E_NO_MEM; // 如果无法分配页表, 返回内存不足错误
349     }
350
351     /* Check if there is already a valid mapping at 'va'. */
352     if (*pgtable_entry & PTE_V) {
353         struct Page *existing_pp = pa2page(PTE_ADDR(*pgtable_entry));
354         if (existing_pp != pp) {
355             // If the existing mapping is different from the one we want to insert, remove it.
356             page_remove(pgdir, va);
357         } else {
358             // If the existing mapping is the same as the one we want to insert, only update
359             permissions.
360             tlb_invalidate(pgdir, va);
361             *pgtable_entry = page2pa(pp) | PERM;
362             return 0;
363         }
364     }
365
366     /* Step 2: Insert the new page and increment its reference count. */
367     *pgtable_entry = page2pa(pp) | PERM;
368     pp->pp_ref++; // Increment the reference count of the physical page.
369
370     /* Step 3: Update TLB. */
371     tlb_invalidate(pgdir, va);
372
373     /* Step 4: Validate the insertion by re-checking the page table entry. */
374     if ((*pgtable_entry & PTE_V) == 0 || pa2page(PTE_ADDR(*pgtable_entry)) != pp) {
375         // If the insertion failed for some reason, decrement the ref count and return error.
376         pp->pp_ref--;
377         return -E_NO_MEM;
378     }
379
380     return 0; // Success.
381 }

```

图 25 page_inisert

```

1  main.c: main is start ...
2  init.c: mips_init() is called
3  Physical memory: 65536K available, base = 65536K, extended = 0K
4  to memory 80401000 for struct page directory.
5  to memory 80431000 for struct Pages.
6  mips_vm_init:boot_pgdir is 80400000
7  pmap.c: mips vm init success
8  start page_insert
9  va2pa(boot_pgdir, 0x0) is 3ffe000
10 page2pa(pp1) is 3ffe000
11 pp2->pp_ref 0
12 end page_insert
13 page_check() succeeded!
14 panic at init.c:55: ~~~~~

```

图 26 执行结果图

4.3 进程与异常部分

```
void mips_vm_init()
{
    extern char end[];
    extern int mCONTEXT;
    extern struct Env *envs;

    Pde *pgdir;
    u_int n;

    /* Step 1: Allocate a page for page directory(first level page table). */
    pgdir = alloc(BY2PG, BY2PG, 1);
    printf("to memory %x for struct page directory.\n", freemem);
    mCONTEXT = (int)pgdir;

    boot_pgdir = pgdir;

    /* Step 2: Allocate proper size of physical memory for global array `pages`,
     * for physical memory management. Then, map virtual address `UPAGES` to
     * physical address `pages` allocated before. For consideration of alignment,
     * you should round up the memory size before map. */
    pages = (struct Page *)alloc(npages * sizeof(struct Page), BY2PG, 1);
    printf("to memory %x for struct Pages.\n", freemem);
    n = ROUND(npages * sizeof(struct Page), BY2PG);
    boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);

    /* Step 3, Allocate proper size of physical memory for global array `envs`,
     * for process management. Then map the physical address to `UENVS`. */
    envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
    n = ROUND(NENV * sizeof(struct Env), BY2PG);
    boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);

    printf("pmap.c:\t mips vm init success\n");
}
```

图 27 mips_vm_init, 用于为 envs 数组分配空间

```
void
env_init(void)
{
    int i;
    /*Step 1: Initial env_free_list. */
    LIST_INIT(&env_free_list);

    /*Step 2: Travel the elements in 'envs', init every element(mainly initial its status, mark it as free)
     * and inserts them into the env_free_list as reverse order. */
    for(i = NENV - 1; i >= 0; -- i)
    {
        envs[i].env_status = ENV_FREE; // initial status as free
        LIST_INSERT_HEAD(&env_free_list, (envs + i), env_link);
    }
}
```

图 28 env_init


```

int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    struct Env *e;
    /* Hint:
    *   * If envid is zero, return the current environment. */
    /*Step 1: Assign value to e using envid. */
    if(envid == 0) {
        *penv = curenv;
        return 0;
    }
    e = &envs[ENVX(envid)];

    if (e->env_status == ENV_FREE || e->env_id != envid) {
        *penv = 0;
        return -E_BAD_ENV;
    }
    /* Hint:
    *   * Check that the calling environment has legitimate permissions
    *   * to manipulate the specified environment.
    *   * If checkperm is set, the specified environment
    *   * must be either the current environment.
    *   * or an immediate child of the current environment. If not, error! */
    /*Step 2: Make a check according to checkperm. */
    if(checkperm == 1 && e != curenv && curenv->env_id != e->env_parent_id) {
        *penv = 0;
        return -E_BAD_ENV;
    }

    *penv = e;
    return 0;
}

```

图 29 envid2env, 实现通过一个的 id 获取该对应的进程控制块的功能

```

static int
env_setup_vm(struct Env *e)
{
    int i, r;
    struct Page *p = NULL;
    Pde *pgdir;

    /*Step 1: Allocate a page for the page directory using a function you completed in the lab2.
     * and add its reference.
     * pgdir is the page directory of Env e, assign value for it. */
    if ( (r = page_alloc(&p)) == -E_NO_MEM) { /* Todo here*/
        panic("env_setup_vm - page alloc error\n");
        return r;
    }
    p->pp_ref++;
    pgdir = (Pde *)page2kva(p);    //将页面p转化为虚拟地址

    /*Step 2: Zero pgdir's field before UTOP. */
    for(i = 0; i < PDX(UTOP); ++ i) pgdir[i] = 0; // UTOP之前的清零

    /*Step 3: Copy kernel's boot_pgdir to pgdir. */
    /* Hint:
     * The VA space of all envs is identical above UTOP
     * (except at VPT and UVPT, which we've set below).
     * See ./include/mmu.h for layout.
     * Can you use boot_pgdir as a template?
     */
    for(i = PDX(UTOP); i < 1024; ++ i) pgdir[i] = boot_pgdir[i];    //UTOP之后的复制过来

    /*Step 4: Set e->env_pgdir and e->env_cr3 accordingly. */
    e->env_pgdir = pgdir;    // 虚拟地址
    e->env_cr3 = PADDR(pgdir);    //物理地址

    /*VPT and UVPT map the env's own page table, with
     * different permissions. */
    e->env_pgdir[PDX(VPT)] = e->env_cr3;
    e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_V | PTE_R;
    return 0;
}

```

图 30 env_setup_vm，设置进程控制块

```

int
env_alloc(struct Env **new, u_int parent_id)
{
    int r;
    struct Env *e;

    /*Step 1: Get a new Env from env_free_list*/

    e = LIST_FIRST(&env_free_list);
    if(e == NULL) {
        *new = NULL;
        return - E_NO_FREE_ENV;
    }

    /*Step 2: Call certain function(has been implemented) to init kernel memory layout for this new Env.
    *The function mainly maps the kernel address to this new Env address. */

    if( env_setup_vm(e) == -E_NO_MEM){
        *new = NULL;
        return - E_NO_FREE_ENV;
    }

    /*Step 3: Initialize every field of new Env with appropriate values*/

    e->env_id = mkenvid(e);
    e->env_parent_id = parent_id;
    e->env_status = ENV_RUNNABLE;

    /*Step 4: focus on initializing env_tf structure, located at this new Env.
    * especially the sp register,CPU status. */
    e->env_tf.cp0_status = 0x10001004;
    e->env_tf.regs[29] = USTACKTOP;

    /*Step 5: Remove the new Env from Env free List*/
    *new = e;
    LIST_REMOVE((e), env_link);

    return 0;
}

```

图 31 env_alloc

```

static int load_icode_mapper(u_long va, u_int32_t sgsz,
                             u_char *bin, u_int32_t bin_size, void *user_data)
{
    /*失败就返回小于0的数 */
    //printf("*****--Start load_icode_mapper()--*****\n");
    struct Env *env = (struct Env *)user_data;
    struct Page *p = NULL;
    u_long i;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG);
    if( bin == NULL ) return - 1;
    u_long size = 0;
    /*Step 1: Load all content of bin into memory. */

    /* 第一部分, 不对齐单独处理*/
    if(offset > 0) {
        size = BY2PG - offset; // 剩余的部分
        if( page_alloc(&p) == - E_NO_MEM) return - E_NO_MEM;
        p->pp_ref++;
        /* 将虚拟地址va与申请的物理页建立映射 */
        page_insert(env->env_pgdir, p, va - offset, PTE_R);
        /* 将bin里的内容存到申请的物理页内存中 */
        bcopy((void *)bin, (void *)page2kva(p) + offset, MIN(bin_size, size));
    }

    /*第二部分, 已经对齐了, 就正常处理*/
    for( i = size; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a page and increase the reference count of it. */
        if( page_alloc(&p) == - E_NO_MEM) return - E_NO_MEM;
        p->pp_ref++;
        /* 将虚拟地址va + i与申请的物理页建立映射 */
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        /* 将bin里的内容存到申请的物理页内存中 */
        bcopy((void *)bin + i, (void *)page2kva(p), MIN(bin_size - i, BY2PG)); //这里也有个地方要注意,末尾可能不对齐
    }

    /*Step 2: alloc pages to reach `sgsz` when `bin_size` < `sgsz`.
    * i has the value of `bin_size` now. */
    /*第三部分, 物理页填充0*/
    while( i < sgsz) {
        if( page_alloc(&p) == - E_NO_MEM) return - E_NO_MEM;
        p->pp_ref++;
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        /*page_alloc 函数已经调用了清0函数, 这里就不再需要了 */
        i += BY2PG;
    }

    //printf("*****--End load_icode_mapper()--*****\n");
    return 0;
}

```

图 32 load_icode_mapper

```

OUTPUT_ARCH(mips)
ENTRY(_start)

SECTIONS
{
    . = 0x80000080;
    .except_vec3 : {
        *(.text.exc_vec3)
    }

    . = 0x80010000;
    .text : {
        *(.text)
    }

    .bss : {
        *(.bss)
    }

    .data : {
        *(.data)
    }

    .sdata : {
        *(.sdata)
    }

    . = 0x80400000;
    end = . ;
}

```

图 33 修改 lds 代码，使得异常后可以调到异常分发代码

5. 总结

5.1 设计过程中遇到的问题及解决方法

1) 遇到交叉编译器路径不正确的问题。

解决办法：通过修改 `include.mk` 文件中的 `CROSS_COMPILE` 变量为正确的路径，解决了这个问题。

2) 编写和调试汇编代码。

解决办法：在补全 `boot/start.S` 时，学会了如何编写 MIPS 汇编代码，并理解了汇编层面的函数跳转。

3) TLB 到底是如何转换的？

TLB 中的项由两部分组成：标识和数据。标识中存放的是虚地址的一部分，而数据部分中存放物理页号、存储保护信息以及其他一些辅助信息。虚地址与 TLB 中项的映射方式有三种：全关联方式、直接映射方式、分组关联方式。

以直接映射方式是指每一个虚拟地址只能映射到 TLB 中唯一的一个表项。假设内存页大小是 8KB，TLB 中有 64 项，采用直接映射方式时的 TLB 变换原理如下图所示：

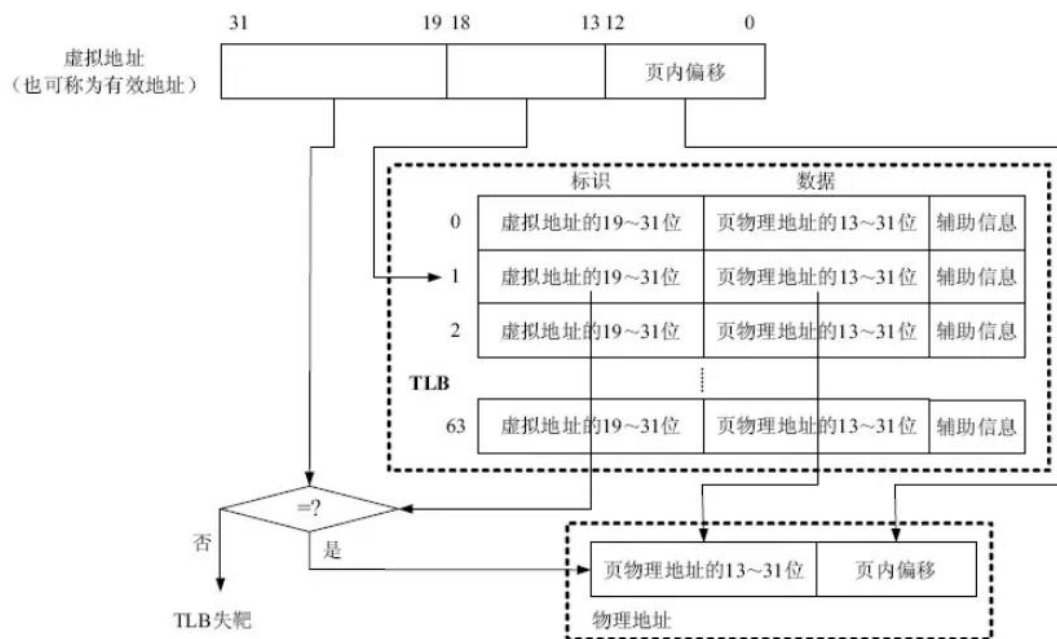


图 34 采用直接映射方式时的 TLB 变换原理

在地址翻译的过程中还会结合 TLB 项中的辅助信息进行判断。数据区的辅助信息包括有效位、引用位和脏位，内容如下：

有效位：对于操作系统，所有的数据都不会加载进内存，当数据不在内存的时候，就需要到硬盘查找并加载到内存。当为 1 时，表示在内存上，为 0 时，该页不在内存，就需要到硬盘查找。

引用位：由于 TLB 中的项数是一定的，所以当有新的 TLB 项需要进来但是又满了的话，如果根据 LRU 算法，就将最近最少使用的项替换成新的项。故需要引用位。同时要注意的是，页表中也有引用位。

脏位：当内存上的某个块需要被新的块替换时，它需要根据脏位判断这个块之前有没有被修改过，如果被修改过，先把这个块更新到硬盘再替换，否则就直接替换。

5.2 团队合作情况

团队合作非常重要，因为实现内核本身难度较大，时间紧任务重，如果单一的从头做到尾一旦出现一时无法解决的问题就十分浪费时间，让团队分工合作，总结汇报汲取他人已有的经验，可以加快设计速度，同时再相互讨论中也加深自己的理解，发现实验中的不足。

5.3 设计总结

在设计中虽然是三人分工，但设计内容内容是层次递进的，因此后续的功能

比如进程调度需要前置的知识,需要小组成员讲述自己设计的重要内容以便后续设计的正常进行。同时在设计中对于某一困难的问题,小组成员一同探讨问题的解决办法。

5.4 心得体会

在本次小型内核实现的过程中,组员之间密切合作,解决了过程中遇到的各种困难,学到了许多新知识,包括 Vim 编辑器的使用、Makefile 文件的功能、make 指令的应用,以及 ELF 文件的作用等,这些都加深了对操作系统实现的理解。

同时,在本次实现内核的过程中,我们也学会了阅读那些已经写好的是代码注释。由于本项目的原型是来自北航的实验,刚上手的时候我们是很陌生的。而代码注释透露出设计者的意图或者整个工程的思想,对于我们完成功能的实现起到了很大的帮助。