

龙芯操作系统终期进度总结汇报

课程名称:	操作系统
任课教师:	翟高寿
小组成员:	胡栩贤、魏振杰
成员学号:	20231159、20241068
专业年级:	计算机科学与技术
学院名称:	詹天佑学院
提交时间:	2022年12月18日

龙芯操作系统终期进度总结汇报

一、实验基本要求与思路

- 实验基本要求
- 实验基本思路
 - 集大家之所成
 - 做有意思的工作
- 实验基本方法

二、项目命名与内涵阐释

- Satori Operating System (by.魏振杰)
- Echo Operating System (by.胡栩贤)

三、平台搭建与环境配置

- 宿主机基础环境
- 虚拟机环境搭建
- 现有系统编译运行
 - 交叉编译环境配置
 - 交叉编译配置文件准备
 - 相关工具和库的安装
 - 正式编译
 - linux-loongarch-v2022-03-10-1运行结果展示

四、过关斩将与模块设计

- 文件组织与编译运行 (`kernel`)
 - 项目文件结构介绍
 - 编译脚本编写说明
 - 运行脚本编写说明
- 固件简介与启动装载 (`start.sh`)
 - UEFI固件简介
 - 龙芯操作系统装载流程
- 内核信息输出模块设计 (`io`)
 - 显示第一个字符
 - 格式化输出及控制
 - 内核信息打印模块
- 中断处理与驱动设计 (`trap/drivers`)
 - 中断处理流程概述
 - 中断控制过程初始化
 - 键盘驱动设计与实现
 - 键盘驱动设计概述

- 4.3.2 键盘扫描码转换过程
 - 4.3.3 键盘事件注册与分发
 - 4.3.4 键盘驱动上层应用举例
 - 4.4 鼠标驱动设计与实现
 - 5 内核标准库设计与实现 (lib)
 - 5.1 基本字符串处理函数
 - 5.2 标准缓冲区设计与实现
 - 5.3 标准可编辑文本结构设计与实现
 - 6 命令解释器设计与实现 (shell)
 - 6.1 核心常量及数据结构定义
 - 6.2 相关功能的设计与实现
 - 7 内存管理设计与实现 (mm)
 - 7.1 Satorios/EchoOS 内存管理架构概述
 - 7.2 连续物理内存分配器 (kmalloc) 设计与实现
 - 7.2.0 Bit Allocator 设计与实现
 - 7.2.1 Buddy System 设计与实现
 - 7.2.2 Slab Allocator 设计与实现
 - 7.3 连续虚拟内存分配器 (vmalloc) 设计与实现
 - 7.4 用户地址空间分配器 (malloc) 设计与实现
 - 8 进程管理设计与实现 (sched)
 - 8.1 进程控制块设计
 - 8.2 进程调度算法设计与实现
 - 8.3 基于VPU的进程调度设计
 - 9 文件系统设计与实现 (fs)
 - 9.1 文件系统架构概述
 - 9.2 虚拟文件系统 (vfs) 设计与实现
 - 9.3 内存虚拟硬盘 (tfs) 设计与实现
 - 9.4 简易文件系统 (FAT32) 设计与实现
 - 10 虚拟处理单元设计与实现 (vpu)
 - 10.1 虚拟处理单元想法概述
 - 10.1.1 虚拟处理单元设计
 - 10.1.2 虚拟处理执行循环
 - 10.2 虚拟段页式内存管理机构
 - 10.2.1 虚拟逻辑地址组成
 - 10.2.2 分段分页数据结构
 - 10.2.3 虚拟地址变换机构
 - 10.2.4 虚拟快表查找机构
 - 10.3 虚拟执行过程设计与实现
 - 10.3.1 虚拟指令集设计与实现
 - 10.3.2 虚拟系统调用设计与实现
 - 10.4 简易汇编器设计
 - 10.4.1 简易汇编器设计思路
 - 11 富文本图形库设计与实现 (rtx)
 - 11.1 富文本图形库架构概述
 - 11.2 富文本图形库设计与实现
 - 12 简易vim设计与实现
- ## 五、项目统计与心得总结
- 1 项目代码统计报告
 - 1.1 Languages
 - 1.2 Directories
 - 2 项目心得总结

一、实验基本要求与思路

1 实验基本要求

基于龙芯LoongArch64的操作系统的构建：要求实现启动初始化、进程管理、内存管理、显示器与键盘驱动、文件系统、系统调用及命令解释器等主要模块，并在QEMU虚拟机或龙芯处理器计算机上测试验证。

2 实验基本思路

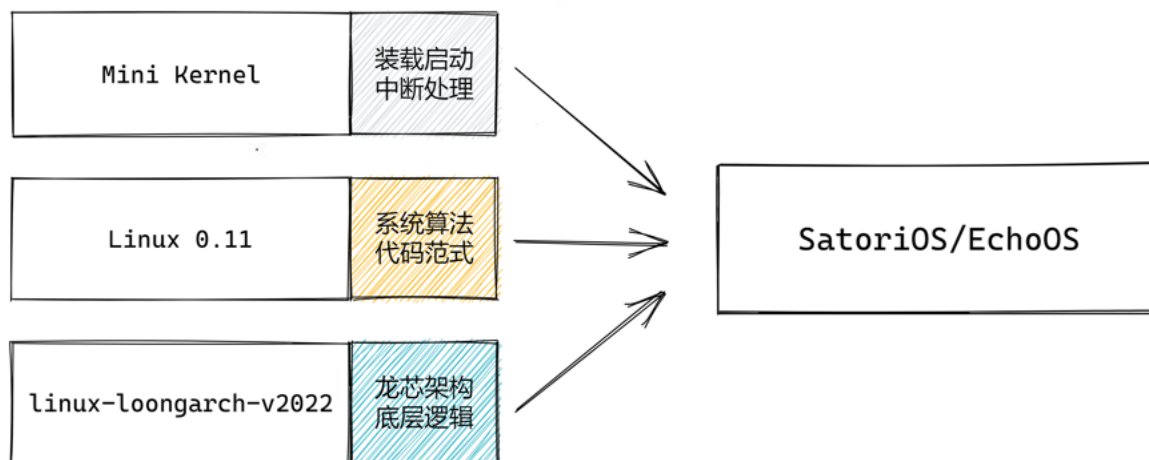
2.1 集大家之所成

Mini Kernel：Mini Kernel 是一个研究系统装载和中断处理的微内核，其本身基于 loongarch，且代码只有500行左右。麻雀虽小，五脏俱全，其实现了基本的键盘中断和时钟中断，同时简单实现了输出函数，为我们的开头工作提供了极大的便利。

Linux 0.11：Linux 0.11 作为最早一批相对完整的操作系统，其思想历经三十多年仍熠熠生辉。是现有许多操作系统只是的基础。我们从中学习了许多数据结构、代码规范、管理算法。

linux-loongarch-v2022：linux-loongarch-v2022 作为较新的基于 loongarch 的linux操作系统，与龙芯架构的底层逻辑更为贴近，是代码实现细节学习的基础。

龙芯官方文档：龙芯官方文档 包括《龙芯架构参考手册》、《龙芯处理器寄存器使用手册》、《龙芯处理器数据手册》等文件，为我们的实际考证提供了莫大的帮助。



2.2 做有意思的工作

为了保证在学习开发过程中有足够的成就感推动我们攻克难关，我们会在核心工作（如中断管理、内存管理、进程管理、文件管理、标准库实现）遇到阻碍时转而尝试开发一些简单的、可视化的、有趣的模块（如shell、vim、图形界面之类），以免长期困于同一个挫折中而丧失了推进工作的动力。

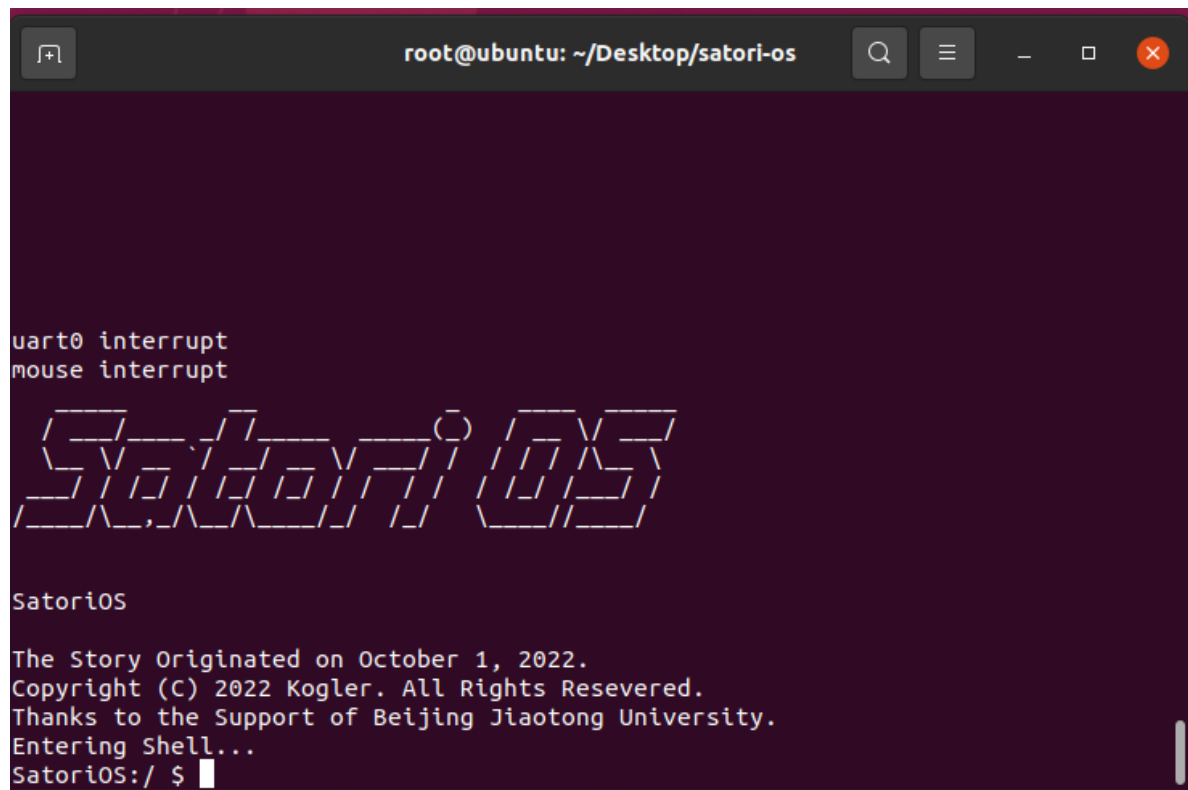
3 实验基本方法

- 底层和上层结合，核心模块与直观模块相结合
- 开发步骤：确定核心概念，定义模块行为，设计数据结构，确定模块方法，进行开发优化，系统联调测试

二、项目命名与内涵阐释

1 Satori Operating System (by.魏振杰)

Satori: 佛教禅宗用语，指心灵之顿悟、开悟；期盼在设计学习的过程中有所顿悟。



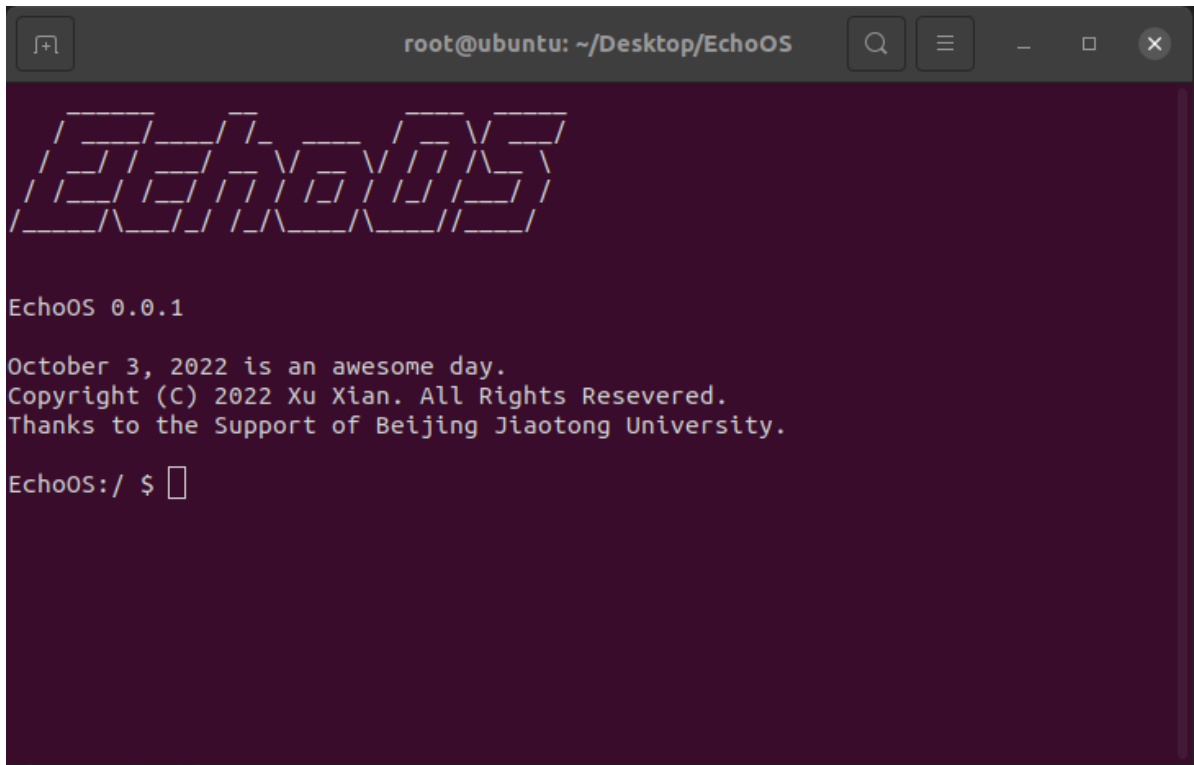
开源地址:

[SatoriOS: 开悟操作系统，仅供个人学习使用（暂时停止更新）。\(github.com\)](#)

[SatoriOS: 开悟操作系统，仅供个人学习使用。\(gitee.com\)](#)

2 Echo Operating System (by.胡栩贤)

Echo: 指回声，灵感来源于一款名为《Dark Echo》的解密游戏，寓意在黑暗中通过努力，获得反馈，不断探索。



开源地址：

[EchoOS: my first OS \(github.com\)](https://github.com/xuxian/EchoOS)

三、平台搭建与环境配置

1 宿主机基础环境

台式计算机

【处理器：Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz；内存：16GB，15.8GB可用】

【处理器：AMD Ryzen 7 4800U with Radeon Graphics @ 1.80 GHz；内存：16GB，15.4GB可用】

Windows 10 21H2

系统类型：基于x64的处理器，64位操作系统

2 虚拟机环境搭建

平台需求	具体平台
虚拟机平台：	VMware Workstation 16 Pro V16.2.4
虚拟机系统环境：	Ubuntu20.04
交叉编译工具：	loongarch64-clfs-2021-12-18-cross-tools-gcc-full
自制操作系统运行平台：	QEMU 6.2.50

3 现有系统编译运行

为了验证环境配置已经就绪，我们实现了对Mini Linux系统的编译运行，并将编译过程记录如下。

这是一个攀登到巨人肩膀上的工作。

3.1 交叉编译环境配置

创建一个shell脚本，用于设置交叉编译器的路径和环境。

env.sh:

```
1 CC_PREFIX=/opt/cross-tools
2 export PATH=$CC_PREFIX/bin:$PATH
3 export LD_LIBRARY_PATH=$CC_PREFIX/lib:$LD_LIBRARY_PATH
4 export LD_LIBRARY_PATH=$CC_PREFIX/loongarch64-unknown-linux-
  gnu/lib/:$LD_LIBRARY_PATH
5 export ARCH=loongarch
6 export CROSS_COMPILE=loongarch64-unknown-linux-gnu-
7 export LC_ALL=C; export LANG=C; export LANGUAGE=C
```

3.2 交叉编译配置文件准备

交叉编译过程中需要的配置文件

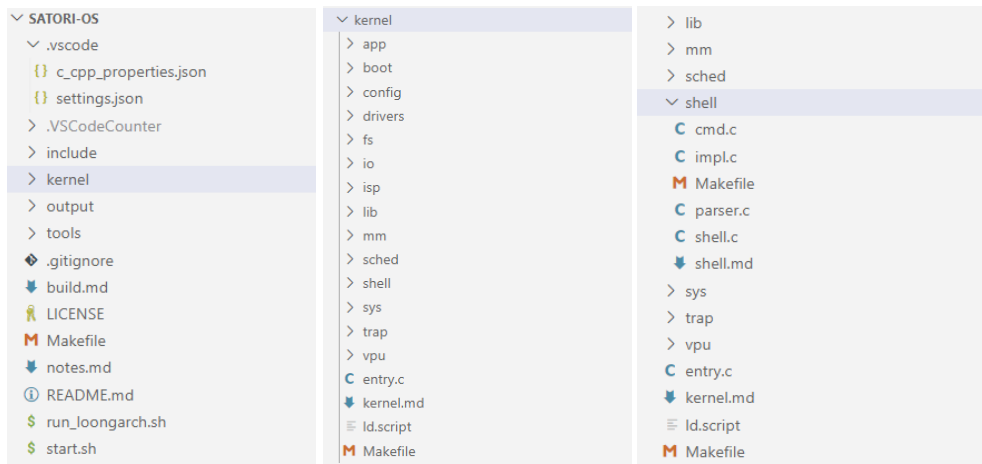
```
1 cp arch/loongarch/configs/loongson3_defconfig_qemu .config
```

其内容大体如下：

```
1 .....
2 CONFIG_CRASH_CORE=y
3 CONFIG_GENERIC_ENTRY=y
4 CONFIG_HAVE_64BIT_ALIGNED_ACCESS=y
5 CONFIG_ARCH_USE_BUILTIN_BSWAP=y
6 CONFIG_HAVE_IOREMAP_PROT=y
7 CONFIG_HAVE_NMI=y
8 CONFIG_TRACE_IRQFLAGS_SUPPORT=y
9 CONFIG_HAVE_ARCH_TRACEHOOK=y
10 CONFIG_HAVE_DMA_CONTIGUOUS=y
11 .....
```

3.3 相关工具和库的安装

```
1 sudo apt install make
2 sudo apt install make-doc
3 sudo apt install make-guile
4 sudo apt install gcc
5 sudo apt install flex
6 sudo apt install bison
7 sudo apt install libssl-dev
```

为了方便IDE定位头文件所在位置，可在 VSCode 中做如下配置：

```
1 {
2     "configurations": [
3         {
4             ...
5             "includePath": [
6                 "${workspaceFolder}/include/**",
7             ],
8             ...
9         }
10    ],
11    ...
12 }
```

1.2 编译脚本编写说明

本系统通过编写多级 Makefile 文件实现对源代码的编译和链接，详细编译过程如下。

在 kernel 文件夹中添加负责内核总体编译的 Makefile 文件，其主要功能是

- 定义基本编译/链接指令参数，并将其 export 至工作空间

```
1 export TOOLPREFIX := loongarch64-unknown-linux-gnu-
2
3 export CC = $(TOOLPREFIX)gcc
4 export LD = $(TOOLPREFIX)ld
5
6 export CFLAGS = -Wall -O2 -g3 \
7     -march=loongarch64 -mabi=lp64s \
8     -ffreestanding -fno-common \
9     -nostdlib \
10    -I../include \ # 告知gcc头文件的相对路径
11    -fno-stack-protector \
12    -fno-pie -no-pie
13
14 export LDFLAGS = -z max-page-size=4096
```

- 遍历 kernel 文件夹，确定需要编译链接的中间目标文件，这里默认将每个子文件夹的名字作为目标文件的文件名


```

1  TGTDIR := ../build/ # 编译目标输出文件夹
2  TARGET := $(TGTDIR)kernel
3  SOURCE := $(wildcard *.c)
4  # 遍历所有的文件夹，并将文件夹的名字作为中间目标产物的名字
5  SUBOBJS = $(filter %.o, $(patsubst ./%, %.o, $(shell find -maxdepth 1 -type
d)))
6  OBJECTS = $(patsubst %.c, $(TGTDIR)%.o, $(SOURCE))
7  OBJECTS += $(patsubst %.o, $(TGTDIR)%.o, $(SUBOBJS))

```

- 遍历并进入到每一个子文件夹中执行 make 指令，递归地完成编译，并将所有中间产物最终链接为内核

```

1  $(TARGET): $(OBJECTS) ld.script # 根据链接脚本进行链接
2      @echo Linking $(TARGET)
3      $(ECHOPRE)$(LD) $(LDFLAGS) -T ld.script -o $(TARGET) $(OBJECTS)
4
5  $(TGTDIR)%.o : %.c
6      @echo Compiling $<
7      $(ECHOPRE)$(CC) $(CFLAGS) -c -o $@ $<
8
9  $(TGTDIR)%.o : # 递归地执行编译
10     @echo ----- == $(subst .o,, $@) == -----
11     -----
12     @mkdir -p $(subst .o,, $@)
13     @(cd $(subst $(TGTDIR),, $(subst .o,, $@)); make) # 进入到子文件夹并执行
`make` 指令

```

每一个子文件夹中的 Makefile 文件内容类似如下

```

1  SECNME := mm # 该模块名称
2  TGTDIR := ../../build/ # 目标产物文件夹
3  INCDIR := ../../include # 头文件目录
4  SUBDIR := $(TGTDIR)$(SECNME)/
5  TARGET := $(TGTDIR)$(SECNME).o
6  SOURCE := $(wildcard *.c)
7  OBJECTS := $(patsubst %.c, $(SUBDIR)%.o, $(SOURCE))
8  CFLAGS += -I../../include
9
10 all: $(TARGET)
11
12 $(TARGET): $(OBJECTS)
13     @echo Linking $(TARGET)
14     $(ECHOPRE)$(LD) -r -o $(TARGET) $(OBJECTS)
15
16 $(SUBDIR)%.o : %.c
17     @echo Compiling $<
18     $(ECHOPRE)$(CC) $(CFLAGS) -c -o $@ $<

```

1.3 运行脚本编写说明

基本参数设置如下：

```
1 MEM="4G"
2 CPUS="1"
3 BIOS="/opt/Satorivenv/loongarch_bios_0310.bin"
4 #BIOS="/opt/Satorivenv/loongarch_bios_0310_debug.bin"
5 KERNEL="./build/kernel"
6 INITRD="/opt/Satorivenv/busybox-rootfs.img"
7 USE_GRAPHIC="yes"
8 DEBUG=''
9 QEMU="qemu-system-loongarch64"
```

由于键盘驱动需要QEMU启动可视化设备，因此可设置相关参数如下：

```
1 if [ $USE_GRAPHIC = "no" ] ; then
2     # run without graphic
3     CMDLINE="root=/dev/ram console=ttyS0,115200 rdinit=/init"
4     GRAPHIC="-vga none -nographic"
5 else
6     # run with graphic
7     CMDLINE="root=/dev/ram console=tty0 rdinit=/init"
8     GRAPHIC="-vga virtio -serial stdio"
```

启动命令（借助QEMU）

```
1 $QEMU -m $MEM -smp $CPUS -bios $BIOS -kernel $KERNEL -append "$CMDLINE"
   $GRAPHIC $DEBUG
```

2 固件简介与启动装载（start.sh）

2.1 UEFI固件简介

BIOS（Basic Input Output System，基本输入输出系统）诞生于1975年的CP/M计算机。UEFI，全称 Unified Extensible Firmware Interface，即“统一的可扩展固件接口”，是一种详细描述全新类型接口的标准，这种接口用于操作系统自动从预启动的操作环境，加载到一种操作系统上。是适用于电脑的标准固件接口，旨在代替 BIOS（基本输入/输出系统）。

UEFI 是 BIOS 的一种升级替代方案。UEFI 本身已经相当于一个微型操作系统，相比 BIOS 具有更强大的功能支持，能为操作系统提供更多的便利（当然，由于其权限极大，不得不让人质疑其安全性）。首先，UEFI 已具备文件系统的支持，它能够直接读取 FAT 分区中的文件。其次，可开发出直接在 UEFI 下运行的应用程序，这类程序文件通常以efi结尾。

相比之下，基于 BIOS 要完成这些工作将会复杂许多。因为 BIOS 下启动操作系统之前，必须从硬盘上指定扇区读取系统启动代码(包含在主引导记录中)，然后从活动分区中引导启动操作系统。对扇区的操作远比不上对分区中文件的操作更直观更简单，所以在 BIOS 下引导安装系统，我们不得不使用一些工具对设备进行配置以达到启动要求。而在 UEFI 下，这些统统都不需要，不再需要主引导记录，不再需要活动分区，不需要任何工具，只要复制安装文件到一个FAT32(主)分区/U盘中，然后从这个分区/U盘启动即可。

传统BIOS开机流程



UEFI开机流程



2.2 龙芯操作系统装载流程

龙芯之前定义了一个启动规范，定义了固件和内核的交互接口，但是在推动相关补丁进入上游社区时，内核的维护者们提出了不同意见。社区倾向于采用EFI标准提供的启动功能，即编译内核时生成一个 `kernel.efi` 这样的EFI模块，它可以不用任何grub之类的装载器实现启动。因为还没有最终定论，导致龙芯开源版本的内核和BIOS互相没有直接支持。因此我们不得不从github.com/loongson fork了相应的软件，进行了一点修改。这里对目前的启动约定做一个简单的说明：

- UEFI bios装载内核时，会把从内核elf文件获取的入口点地址（可以用`readelf -h`或者`-l vmlinux`看到）抹去高32位使用。比如vmlinux链接的地址是`0x9000000001034804`，实际bios跳转的地址将是`0x1034804`，代码装载的位置也是物理内存`0x1034804`。BIOS这么做是因为它逻辑上相当于用物理地址去访问内存，高的虚拟地址空间没有映射不能直接用。
- 内核启动入口代码需要做两件事：（参见arch/loongarch/kernel/head.S）
 1. 设置一个直接地址映射窗口（参见loongarch体系结构手册，5.2.1节），把内核用到的64地址抹去高位映射到物理内存。目前linux内核是设置`0x8000xxxx-xxxxxxx`和`0x9000xxxx-xxxxxxx`地址抹去最高的8和9为其物理地址，前者用于uncache访问(即不通过高速缓存去load/store)，后者用于cache访问。
 2. 做个代码自跳转，使得后续代码执行的PC和链接用的虚拟地址匹配。BIOS刚跳转到内核时，用的地址是抹去了高32位的地址（相当于物理地址），步骤1使得链接时的高地址可以访问到同样的物理内存，这里则换回到原始的虚拟地址。

我们这里使用链接脚本（`ld.script`）设置内核入口：

```
1 OUTPUT_ARCH( "loongarch" )
2 ENTRY( kernel_entry )
3
4 SECTIONS
5 {
6     . = 0x92000000;
7
8     .text : {
9         *(.text .text.*)
10        PROVIDE(etext = .);
11    }
12
13    .rodata : {
14        . = ALIGN(16);
15        *(.srodata .srodata.*)
16        . = ALIGN(16);
```

```

17     *(&.rodata .rodata.*)
18 }
19
20 .data : {
21     . = ALIGN(16);
22     *(&.sdata .sdata.*)
23     . = ALIGN(16);
24     *(&.data .data.*)
25 }
26
27 .bss : {
28     . = ALIGN(16);
29     *(&.sbss .sbss.*)
30     . = ALIGN(16);
31     *(&.bss .bss.*)
32 }
33
34 PROVIDE(end = .);
35 }

```

3 内核信息输出模块设计（io）

3.1 显示第一个字符

通过上述步骤，我们成功完成了内核装载并启动，同时设置了内核入口，就相当于给我们的操作系统“程序”设置了一个“main”函数，但是在这样的命令行窗口，没有输出，我们无法看到任何东西，也就无法做任何有意义的交互，甚至无法进行基本的调试工作。因此，摆在我们面前亟待解决的第一个问题就是——实现 printf。

作为一个从0开始实现的操作系统，一开始并没有标准的输入输出库供我们使用，经过研究，想要在命令行窗口实现输出，我们决定通过串口实现输出与通信。

对于串口的通信，龙芯3A5000提供了两块UART(Universal Asynchronous Receiver Transmitter)控制器进行控制，分别为UART0和UART1从《龙芯3A5000_3B5000处理器寄存器使用手册》中我们可以找到UART0控制器的物理地址为0x1FE00100，在输出过程中，涉及到的两个重要寄存器如下图：

15.1.1 数据寄存器（DAT）

中文名： 数据传输寄存器
寄存器位宽： [7: 0]
偏移量： 0x00
复位值： 0x00

15.1.7 线路状态寄存器（LSR）

中文名： 线路状态寄存器
寄存器位宽： [7: 0]
偏移量： 0x05
复位值： 0x00

其中DAT寄存器，负责输入数据的传输，是我们向命令行窗口输出的端口，接收8位宽的 **ascii码**；LSR寄存器则负责在传输前后检测输出状态，其第5个标志位代表当前串口是否为空，即是否能够传输数据，防止对前面传入但未处理的数据造成直接覆盖。

5	TFE	1	R	传输 FIFO 位空表示位 '1' - 当前传输 FIFO 为空，给传输 FIFO 写数
---	-----	---	---	---

				据时清零 '0' - 有数据
--	--	--	--	-----------------------

具体实现和管理代码如下：

```

1  static const unsigned long uart_base = 0x1fe001e0;
2
3  #define UART0_THR  (uart_base + 0)
4  #define UART0_LSR  (uart_base + 5)
5  #define LSR_TX_IDLE  (1 << 5)
6
7  static char io_readb()
8  {
9      return *(volatile char*)UART0_LSR;
10 }
11
12 static void io_writeb(char c)
13 {
14     while ((io_readb() & LSR_TX_IDLE) == 0){
15         asm volatile("nop\n" : : : );
16     }
17     *(char*)UART0_THR = c;
18 }

```

在此基础上，我们进一步封装出putc（输出单个字符）和puts（输出字符串）：

```

1  void putc(char c)
2  {
3      // wait for Transmit Holding Empty to be set in LSR.
4      while ((io_readb() & LSR_TX_IDLE) == 0);
5      io_writeb(c);
6  }
7
8  void puts(char *str)
9  {
10     while (*str != 0)
11     {
12         putc(*str);
13         str++;
14     }
15 }

```

3.2 格式化输出及控制

格式化输出函数设计实现如下：

```
1 void printf(const char *fmt, ...)
2 {
3     va_list ap;
4     int i, c, n;
5     char *s;
6
7     if (fmt == 0)
8     {
9         intr_on();
10        return;
11    }
12
13    va_start(ap, fmt);
14    for (i = 0; (c = fmt[i] & 0xff) != 0; i++)
15    {
16        if (c != '%')
17        {
18            putc(c);
19            continue;
20        }
21        c = fmt[++i] & 0xff;
22        if (c == 0)
23            break;
24        switch (c)
25        {
26            case 'd': // 打印10进制整数
27                print_int(va_arg(ap, int), 10, 1);
28                break;
29            case 'x': // 打印16进制整数
30                print_int(va_arg(ap, int), 16, 1);
31                break;
32            case 'P': // 打印64位地址
33                print_ptr(va_arg(ap, unsigned long));
34                break;
35            case 'p': // 打印32位地址
36                print_ptr_short(va_arg(ap, unsigned long));
37                break;
38            case 's': // 打印字符串
39                if ((s = va_arg(ap, char *)) == 0)
40                    s = "(null)";
41                for (; *s; s++)
42                    putc(*s);
43                break;
44            case 'c': // 打印字符
45                putc((char)(va_arg(ap, int)));
46                break;
47            case 'o': // 打印字符串（16位左对
48                // 齐）
49                n = 16;
50                if ((s = va_arg(ap, char *)) == 0)
51                    s = "(null) ";
```

```

51         for (; n; s++, n--)
52             putc(*s ? *s : ' ');
53         break;
54     case 'o':                                     // 打印字符串（8位左对
齐）
55         n = 8;
56         if ((s = va_arg(ap, char *)) == 0)
57             s = "(null) ";
58         for (; n; s++, n--)
59             putc(*s ? *s : ' ');
60         break;
61     case '%':
62         putc('%');
63         break;
64     default:
65         putc('%');
66         putc(c);
67         break;
68     }
69 }
70 }

```

在此基础上，我们了解到可以通过 ANSI 控制码实现对串口窗口的多样化控制，我们将相关方法进行了封装，其中一些示例如下：

```

1  static inline void cursor_move_to(sint x, sint y);           // 光标位置移动到
2  static inline void clear_screen();                           // 清屏
3  static inline void set_cursor_style(sint style);             // 设置光标样式
4  static inline void set_cursor_color(sint color);             // 设置光标（字体）
颜色
5  static inline void set_cursor_background_color(sint color); // 设置光标（字体）
背景颜色
6  void save_cursor_style();                                     // 保存光标样式
7  void restore_cursor_style();                                  // 恢复光标样式
8  void save_cursor_color();                                     // 保存光标（字体）颜色
9  void restore_cursor_color();                                  // 恢复光标（字体）颜色

```

3.3 内核信息打印模块

在以上输出控制模块的基础上，我们统一了内核信息输出的标准格式，定义了四种输出类型，其实现如下：

```

1  #define pr_info(src, fmt, ...) \
2      { \
3          save_cursor_color(); \
4          set_cursor_color(ANSI_GREEN); \
5          printf("[ " #src " ] | INFO | "); \
6          printf(fmt, ##__VA_ARGS__); \
7          put_char('\n'); \
8          restore_cursor_color(); \
9      }
10

```

```

11 #define pr_warn(src, fmt, ...) \
12     { \
13         save_cursor_color(); \
14         set_cursor_color(ANSI_YELLOW); \
15         printf("[ " #src " ] | WARN | "); \
16         printf(fmt, ##__VA_ARGS__); \
17         put_char('\n'); \
18         restore_cursor_color(); \
19     }
20
21 #define pr_error(src, fmt, ...) \
22     { \
23         save_cursor_color(); \
24         set_cursor_color(ANSI_RED); \
25         printf("[ " #src " ] | ERROR | "); \
26         printf(fmt, ##__VA_ARGS__); \
27         put_char('\n'); \
28         restore_cursor_color(); \
29         die(); \
30     }
31
32 #ifdef CONFIG_DEBUG
33
34 #define pr_debug(src, fmt, ...) \
35     { \
36         save_cursor_color(); \
37         set_cursor_color(ANSI_BLUE); \
38         printf("[ " #src " ] | DEBUG | "); \
39         printf(fmt, ##__VA_ARGS__); \
40         put_char('\n'); \
41         restore_cursor_color(); \
42     }
43
44 #else
45
46 #define pr_debug(src, fmt, ...) \
47     { \
48     }
49
50 #endif /* !CONFIG_DEBUG */

```

内核各模块在使用上述基本模块时，可以进行封装，例如：

```

1 #define mm_info(fmt, ...) pr_info(MM, fmt, ##__VA_ARGS__)
2 #define mm_warn(fmt, ...) pr_warn(MM, fmt, ##__VA_ARGS__)
3 #define mm_error(fmt, ...) pr_error(MM, fmt, ##__VA_ARGS__)
4 #define mm_debug(fmt, ...) pr_debug(MM, fmt, ##__VA_ARGS__)

```

4 中断处理与驱动设计 (trap/drivers)

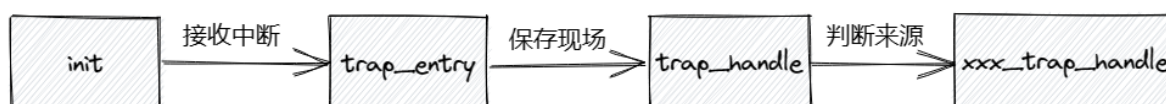
4.1 中断处理流程概述

(1) 为了实现中断处理，首先是对中断的初始化，为各个中断相关的芯片写入控制方式，同时为CPU设置中断入口、中断使能等中断基本信息；

(2) 当接收到中断后，CPU会自动跳转到设置的中断入口处运行；

(3) 当前我们使用的是单个中断，在进入中断后，各个中断统一进入trap_entry函数进行现场保护，而后进入中断处理函数trap_handle；

(4) 在trap_handle函数中，我们通过读取例外配置寄存器获取当前中断状态并和配置的中断配置寄存器以及各中断设备进行比较，确定触发中断的设备并进入对应设备的中断处理函数。



4.2 中断控制过程初始化

对于中断过程的初始化可以分为四部分：

```
1 void trap_init(void)
2 {
3     /*CPU控制状态寄存器设置*/
4     unsigned int ecfg = ( 0U << CSR_ECFG_VS_SHIFT ) | HWI_VEC | TI_VEC;
5     unsigned long tcfg = 0x0a000000UL | CSR_TCFG_EN | CSR_TCFG_PER;
6     w_csr_ecfg(ecfg);
7     w_csr_tcfg(tcfg);
8     w_csr_eentry((unsigned long)trap_entry);
9     /*拓展io中断初始化*/
10    extioi_init();
11    /*桥片初始化*/
12    ls7a_intc_init();
13    /*键鼠控制芯片初始化*/
14    i8042_init();
15 }
```

首先是对**控制状态寄存器**的设置，不同于8086简单基础的架构，龙芯对CPU本身设置了大量的可配置内容，其通过**控制状态寄存器**进行设置。所有的**控制状态寄存器**需要通过龙芯的 `csrrw/csrrd` 指令进行控制。

`ecfg` 个 `tcfg` 分别为例外配置寄存器和时钟配置寄存器，这样我们就实现了对于中断的基础配置和一个十分重要的中断源——时钟中断源：

7.4.5 例外配置 (ECFG)

该寄存器用于控制例外和中断的入口计算方式以及各中断的局部使能位。

表 7-6 例外配置寄存器定义

位	名字	读写	描述
12:0	LIE	RW	局部中断使能位，高有效。这些局部中断使能位与 CSR_ESTAT 中 IS 域记录的 13 个中断源一一对应，每一位控制一个中断源。
15:13	0	RO	保留域，读返回 0，且软件不允许改变其值。
18:16	VS	RW	配置例外和中断入口的偏移。当 VS=0 时，所有例外和中断的入口地址是同一个；当 VS=1 时，各例外和中断之间的入口地址则是 2 ^{VS} 条指令。 因为 TLB 重填例外和机器断点例外其独立的入口基址，所以二者的例外入口不受 VS 域的影响。
31:19	0	RO	保留域，读返回 0，且软件不允许改变其值。

7.6.2 定时器配置 (TCFG)

该寄存器是软件配置定时器的接口。定时器的有效位数由实现决定，因此该寄存器中 TimeVal 域的位宽也将随之变化。

表 7-45 定时器配置寄存器定义

位	名字	读写	描述
0	En	RW	定时器使能位。仅当该位为 1 时，定时器才会进行倒计时自减，并在减为 0 时置起定时中断信号。
1	Periodic	RW	定时器循环模式控制位。若该位为 1，定时器在倒计时自减至 0 时，在置起定时中断信号的同时，还会自动对定时器重新装载或 InitVal 域中配置的初始值，然后下一个时钟周期继续自减。若该位为 0，定时器在倒计时自减至 0 时，将停止计数直至软件再次配置该定时器。
n-1:2	InitVal	RW	定时器倒计时自减计数的初始值。要求该初始值必须是 4 的整数倍，硬件将自动在该域数值的最低位补上两比特 0 后再使用。
GRLEN-1:n	0	R	只读域为 0，写被忽略。

同时我们还需要向 `eentry` 控制状态寄存器中写入我们编写的中断入口函数地址。

对CPU的控制状态寄存器配置完成后，我们需要对CPU的IO端口的控制状态进行配置：

```
1 void extioi_init(void)
```

```

2  {
3      iocsr_writeq((0x1UL << UART0_IRQ) | (0x1UL << KEYBOARD_IRQ) |
4                  (0x1UL << MOUSE_IRQ) | (0x1UL << DISK_IRQ),
5                  LOONGARCH_IOCSR_EXTIOI_EN_BASE);
6
7      /* extioi[31:0] map to cpu irq pin INT1, other to INT0 */
8      iocsr_writeq(0x01UL, LOONGARCH_IOCSR_EXTIOI_MAP_BASE);
9
10     /* extioi IRQ 0-7 route to core 0, use node type 0 */
11     iocsr_writeq(0x0UL, LOONGARCH_IOCSR_EXTIOI_ROUTE_BASE);
12
13     /* nodetype0 set to 1, always trigger at node 0 */
14     iocsr_writeq(0x1, LOONGARCH_IOCSR_EXRIOI_NODETYPE_BASE);
15 }

```

这里分别对**拓展IO中断使能寄存器**、**中断路由寄存器**、**中断目标处理器核路由寄存器地址**和**中断目标结点映射方式寄存器**进行了配置。

对CPU的配置完成后，我们还需要对桥片芯片进行配置，配置其中断使能寄存器、中断向量寄存器等：

```

1  void ls7a_intc_init(void)
2  {
3      /* enable uart0/keyboard/mouse */
4      *(volatile unsigned long*)(LS7A_INT_MASK_REG) = ~((0x1UL << UART0_IRQ) |
5                                                         (0x1UL << KEYBOARD_IRQ) |
6                                                         (0x1UL << MOUSE_IRQ));
7
8      *(volatile unsigned long*)(LS7A_INT_EDGE_REG) = (0x1UL << (UART0_IRQ |
9                                                         KEYBOARD_IRQ | MOUSE_IRQ));
10
11     /* route to the same irq in extioi */
12     *(volatile unsigned char*)(LS7A_INT_HTMSI_VEC_REG + UART0_IRQ) =
13     UART0_IRQ;
14     *(volatile unsigned char*)(LS7A_INT_HTMSI_VEC_REG + KEYBOARD_IRQ) =
15     KEYBOARD_IRQ;
16     *(volatile unsigned char*)(LS7A_INT_HTMSI_VEC_REG + MOUSE_IRQ) =
17     MOUSE_IRQ;
18
19     *(volatile unsigned long*)(LS7A_INT_POL_REG) = 0x0UL;
20 }

```

自此，中断还不能够正常运行，我们还需要对外围设备进行配置。我们知道不同设备都会有不同的控制状态和输入输出端口，对应设备控制芯片的控制寄存器、数据寄存器和状态寄存器（i8042的控制寄存器和状态寄存器共用一个端口），这里我们最直接涉及到的就是对键鼠进行控制的i8042芯片，我们需要在其控制端口写入我们需要的控制方式：

```

1  void i8042_init(void)
2  {
3      unsigned char data;
4
5      /* disable device */
6      *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAD;
7      *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xA7;
8      /* flush */

```

```

9   data = *(volatile unsigned char*)(LS7A_I8042_DATA);
10  /* self test */
11  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAA;
12  data = *(volatile unsigned char*)(LS7A_I8042_DATA);
13
14  /* set config byte, enable device and interrupt*/
15  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0x20;
16  data = *(volatile unsigned char*)(LS7A_I8042_DATA);
17  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0x60;
18  *(volatile unsigned char*)(LS7A_I8042_DATA) = 0x07;
19
20  /* test */
21  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAB;
22  data = *(volatile unsigned char*)(LS7A_I8042_DATA);
23
24  /* enable first port */
25  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAE;
26
27  /* reset device */
28  *(volatile unsigned char*)(LS7A_I8042_DATA) = 0xFF;
29  data = *(volatile unsigned char*)(LS7A_I8042_DATA);
30  }

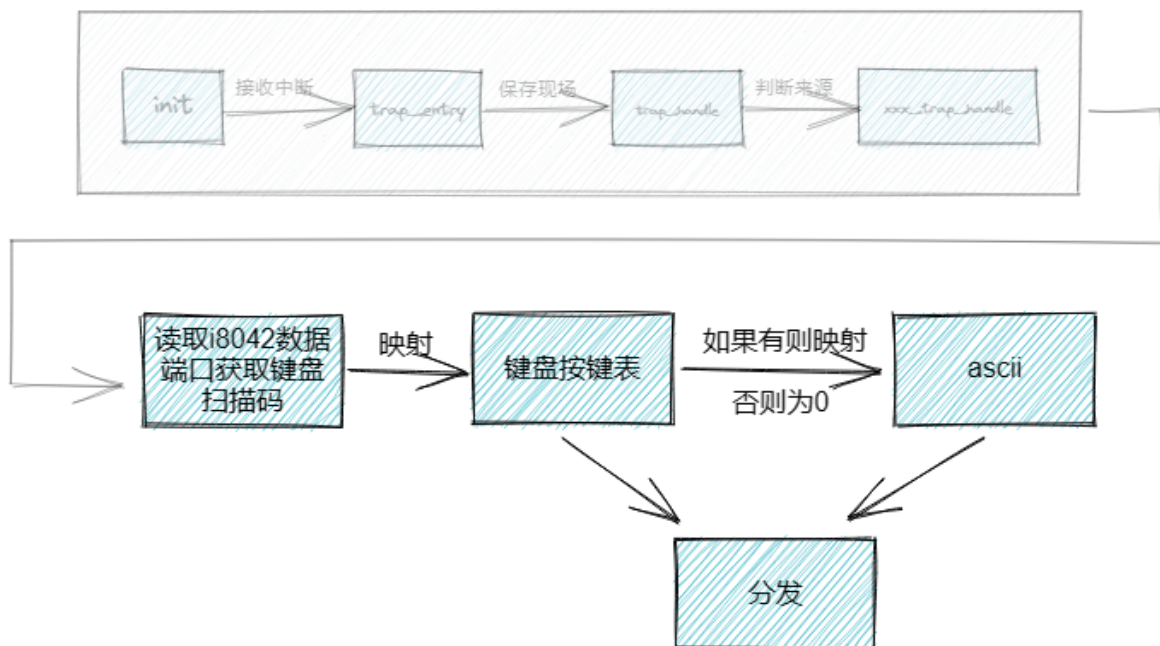
```

这样我们就是完整实现了键鼠中断以及时钟中断并将中断定向到我们设置的中断入口处。

4.3 键盘驱动设计与实现

4.3.1 键盘驱动设计概述

在操作系统的运行过程中，因为处于命令行状态下，我们最主要的交互方式就是通过键盘实现输入，下图为设计的键盘驱动处理过程。



当我们通过中断进入键盘的中断处理程序后，会先通过状态端口判断 i8042 的数据端口是否有未读的数据，如果有，则可以通过数据端口读取数据。

此时我们读取到的数据为键盘扫描码，我们需要通过键盘扫描码映射到键盘按键表，其中包含了每一个按键的 `ascii` 码（若没有则为0）。然后外部的应用可以通过注册键盘的回调，在键盘中断的过程中接收键盘驱动分发的键盘数据。

4.3.2 键盘扫描码转换过程

首先将键盘扫描码映射为键号：

```
1  const unsigned int keymap[] = {
2      ...
3      /* 28 */ KEY_RESERVED, KEY_SPACE,      KEY_V,          KEY_F,          KEY_T,
4              KEY_R,          KEY_5,          KEY_F6,
5      /* 30 */ KEY_RESERVED, KEY_N,          KEY_B,          KEY_H,          KEY_G,
6              KEY_Y,          KEY_6,          KEY_F7,
7      /* 38 */ KEY_RESERVED, KEY_RIGHTALT, KEY_M,          KEY_J,          KEY_U,
8              KEY_7,          KEY_8,          KEY_F8,
9      /* 40 */ KEY_RESERVED, KEY_COMMA,      KEY_K,          KEY_I,          KEY_O,
10             KEY_0,          KEY_9,          KEY_F9,
11     ...
12 };
```

进一步将键号映射为ASCII码：

```
1  const char kbd_US[128] =
2  {
3      0, 27, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=',
4      '\b',
5      '\t', /* <-- Tab */
6      'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n',
7      0, /* <-- control key */
8      'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`', 0, '\\',
9      'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0,
10     ...
11 };
```

若同时按下 `shift`，则使用另一个表完成映射：

```
1  const char kbd_US_shift[128] =
2  {
3      0, 27, '!', '@', '#', '$', '%', '^', '&', '*', '(', ')', '_', '+', 0,
4      0, /* <-- Tab */
5      'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', '{', '}', '\n',
6      0, /* <-- control key */
7      'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ':', '\'', '~', 0, '|',
8      'Z', 'X', 'C', 'V', 'B', 'N', 'M', '<', '>', '?', 0,
9      ...
10 };
```

至此，完成了从键盘扫描码到 `ASCII` 的转换。

4.3.3 键盘事件注册与分发

我们设计的键盘驱动采用了事件注册与分发的机制，键盘事件的数据结构如下：

```
1 typedef struct kbd_event {
2     char key;    // ASCII
3     int state;   // 键盘是否按下？
4     int key_no;  // 键号，用于判断特殊按键是否按下
5 } kbd_event;
```

上层应用可以通过注册事件回调函数的方式来获取键盘按下的信息：

```
1 int register_kbd_cbk(void (*cbk_func)(kbd_event));
2 void unregister_kbd_cbk(int cbk_id);
```

当键盘中断发生时，键盘驱动会完成键盘信息的解析，并向注册了键盘事件的应用程序逐一分发键盘事件（即依次调用其注册的键盘回调）。键盘驱动处理中断的核心函数实现如下：

```
1 void handle_kbd_irq(void)
2 {
3     while (kbd_has_data()) // while 用于稳定键盘中断，具体原理不详
4     {
5         kbd_event e;
6         unsigned char code = kbd_read_byte();
7         e.key_no = keymap[(unsigned int)code];
8
9         e.state = KEY_STATE_DOWN;
10
11        if (code == 0xF0 && kbd_has_data()) // 处理由于键盘抬起准备的二次数据
12        {
13            e.key_no = keymap[(unsigned int)kbd_read_byte()];
14            e.state = KEY_STATE_UP;
15        }
16
17        ... // 一些特殊按键的判断
18        else
19        {
20            ... // 根据特殊按键的状态进行进一步转换
21            invoke_kbd_cbk(e); // 完成键盘事件生成后，调用注册的回调函数
22        }
23    }
24 }
```

4.3.4 键盘驱动上层应用举例

键盘驱动的最直接应用是 `gets` 函数的实现（是的没错，相比于 `puts` 函数，`gets` 函数的实现要绕更大的弯子）。

`stdin` 的实现基于标准缓冲区库的实现，有关标准缓冲区库的讨论，请参见本章5.2小节。

在有了标准缓冲区的基础上，`stdin` 会动态地向键盘驱动注册回调函数，相关代码如下：

```

1 void stdin_kbd_cbk(kbd_event e)
2 {
3     if (stdin_enabled && e.state == KEY_STATE_DOWN)
4     {
5         if (e.key == '\b')
6         {
7             if (std_buffer_empty(stdin_buffer))
8                 return;
9             std_buffer_pop(stdin_buffer);
10            putc('\b');
11            putc(' ');
12            putc('\b');
13        }
14        else
15        {
16            std_buffer_put(stdin_buffer, e.key);
17            putc(e.key);
18            if (e.key == '\n')
19                putc('\r');
20        }
21    }
22 }

```

此时，便可借助标准缓冲区的相关方法实现 `gets` 函数：

```

1 char getc()
2 {
3     stdin_enable();
4     byte c = 0;
5     c = std_buffer_wait_char(stdin_buffer);
6     stdin_disable();
7     return c;
8 }
9
10 int gets(char *str, int size)
11 {
12     stdin_enable();
13     int n = std_buffer_wait_line(stdin_buffer, str, size);
14     stdin_disable();
15     return n;
16 }

```

4.4 鼠标驱动设计与实现

由于时间原因，鼠标驱动并未完全实现，在此不便介绍。

5 内核标准库设计与实现 (lib)

5.1 基本字符串处理函数

为了方便内核部分功能的实现，我们首先实现了 `string.h`：

```
1  #ifndef _SYSTEM_LIB_STRING_H_
2  #define _SYSTEM_LIB_STRING_H_
3
4  int strcmp(const char *str1, const char *str2);
5  int strcpy(char *dst, const char *src);
6  int strlen(const char *str);
7  int strncmp(const char *str1, const char *str2, int n);
8  int strncpy(char *dst, const char *src, int n);
9  void split(char *str, char *delim, char result[][100], int *result_len);
10
11 void memset(void *ptr, char c, unsigned long size);
12 int memcmp(void *ptr1, void *ptr2, unsigned long size);
13 void memcpy(void *ptr1, void *ptr2, unsigned long size);
14
15 #endif /* !_SYSTEM_LIB_STRING_H_ */
```

以上这些函数的功能人尽皆知，在此便不赘述。

5.2 标准缓冲区设计与实现

为了方便标准输入模块的开发，我们首先设计实现了标准字符输入缓冲区数据结构及其相关操作函数。缓冲区由顺序队列实现，并提供了自旋等待数据的API。

```
1  typedef struct std_buffer
2  {
3      byte *data;        // 数据
4      int size;          // 已装入数据大小（单位：字节）
5      int capacity;      // 缓冲区容量（单位：字节）
6      int head;          // 缓冲区头
7      int tail;          // 缓冲区尾
8      int peek;          // 访问指针
9  } std_buffer;
10
11 std_buffer *std_buffer_create(int capacity);
12 void std_buffer_destroy(std_buffer *buffer);
13 void std_buffer_clear(std_buffer *buffer);
14
15 void std_buffer_put(std_buffer *buffer, const byte data);
16 void std_buffer_puts(std_buffer *buffer, const char *data);
17
18 byte std_buffer_pop(std_buffer *buffer);
19 byte std_buffer_get(std_buffer *buffer);
20
21 int std_buffer_gets(std_buffer *buffer, char *data, int size);
22
23 byte std_buffer_peek(std_buffer *buffer);
24 void std_buffer_back(std_buffer *buffer);
25
26 char std_buffer_wait_char(std_buffer *buffer);
27
```

```

28 int std_buffer_wait_line(std_buffer *buffer, char *data, int size);
29
30 static inline int std_buffer_full(std_buffer *buffer)
31 {
32     return buffer->size == buffer->capacity;
33 }
34
35 static inline int std_buffer_full_p(std_buffer *buffer)
36 {
37     return buffer->peek == buffer->head;
38 }
39
40 static inline int std_buffer_empty(std_buffer *buffer)
41 {
42     return buffer->size == 0;
43 }
44
45 static inline int std_buffer_empty_p(std_buffer *buffer)
46 {
47     return buffer->peek == buffer->tail;
48 }

```

5.3 标准可编辑文本结构与实现

为了支持建议文本编辑器 `vim` 的设计实现，我们利用二维双向链表设计了可编辑的文本数据结构，其核心数据结构定义与相关操作函数设计与实现如下。

核心数据结构：

```

1  typedef struct text_cursor // 光标位置
2  {
3      int x;
4      int y;
5  } text_cursor;
6
7  typedef struct text_char // 字符结点
8  {
9      char ch;
10     struct text_char *next;
11     struct text_char *prev;
12 } text_char;
13
14 typedef struct text_line // 行节点
15 {
16     int nr_chars;
17     text_char *fst_char;
18     text_char *lst_char;
19     struct text_line *next;
20     struct text_line *prev;
21 } text_line;
22
23 typedef struct text_buffer // 可编辑文本缓冲
24 {

```



```

25     int nr_lines;
26     text_line *fst_line;
27     text_line *lst_line;
28     text_line *cur_line;
29     text_char *cur_char;
30     text_cursor cursor;
31 } text_buffer;

```

相关操作函数:

```

1  text_buffer *text_buffer_create();
2
3  int text_buffer_count_lines(text_buffer *buffer);
4  int text_buffer_count_chars(text_buffer *buffer);
5
6  void text_buffer_load_text(text_buffer *buffer, char *str);
7  void text_buffer_save_text(text_buffer *buffer, char *str, int size);
8
9  void text_buffer_save_line(text_line *line, char *str, int size);
10
11 void text_buffer_clear(text_buffer *buffer);
12 void text_buffer_destroy(text_buffer *buffer);
13 void text_buffer_free_line(text_line *line);
14
15 void text_buffer_write_char(text_buffer *buffer, char c);
16
17 void text_buffer_insert_line(text_buffer *buffer);
18 void text_buffer_insert_char(text_buffer *buffer, char c);
19 void text_buffer_insert_string(text_buffer *buffer, char *str);
20
21 void text_buffer_split_line(text_buffer *buffer);
22 void text_buffer_merge_line(text_buffer *buffer);
23
24 static inline void text_buffer_newline(text_buffer *buffer)
25 {
26     text_buffer_split_line(buffer);
27 }
28
29 void text_buffer_delete_char(text_buffer *buffer);
30 void text_buffer_delete_line(text_buffer *buffer);
31
32 void text_buffer_backspace(text_buffer *buffer);
33
34 void text_buffer_cursor_up(text_buffer *buffer);
35 void text_buffer_cursor_down(text_buffer *buffer);
36 void text_buffer_cursor_prev(text_buffer *buffer);
37 void text_buffer_cursor_next(text_buffer *buffer);
38
39 void text_buffer_cursor_move_to(text_buffer *buffer, int x, int y);
40 void text_buffer_cursor_to_line(text_buffer *buffer, int line);
41 void text_buffer_cursor_to_col(text_buffer *buffer, int col);
42
43 void text_buffer_cursor_home(text_buffer *buffer);
44 void text_buffer_cursor_end(text_buffer *buffer);
45

```

```

46 void text_buffer_cursor_line_home(text_buffer *buffer);
47 void text_buffer_cursor_line_end(text_buffer *buffer);
48
49 void text_buffer_print_info(text_buffer *buffer);
50
51 void text_buffer_print_line(text_line *line);
52 void text_buffer_print_text(text_buffer *buffer);
53
54 void text_buffer_relocate_cursor(text_buffer *buffer);

```

函数详细实现过程不再赘述，感兴趣的同学可以参考源码。

6 命令解释器设计与实现 (shell)

6.1 核心常量及数据结构定义

核心常量及数据结构定义如下：(shell.h)

```

1  #define SHELL_BUFFER_SIZE 256 // shell命令输入缓冲区大小
2  #define SHELL_CMD_MAX 64 // 最大支持内置命令的数量
3  #define CMD_PARAM_MAX 8 // 每条命令最大支持的参数数量
4  #define NAME_LEN_MAX 16 // 命令或参数名最大长度
5  #define DESC_LEN_MAX 64 // 命令或参数描述信息最大长度
6
7  typedef struct {
8      char sign;
9      char name[NAME_LEN_MAX];
10     char desc[DESC_LEN_MAX];
11 } cmd_param; // 命令参数定义数据结构，sign代表该命令的缩写
12
13 typedef struct {
14     char sign;
15     char param[DESC_LEN_MAX];
16 } param_unit; // 从命令中解析得到的参数数据结构，由参数缩写和参数附加值组成
17
18 typedef struct {
19     char cmd[NAME_LEN_MAX]; // 命令名
20     char desc[DESC_LEN_MAX]; // 命令描述信息
21     cmd_param params[CMD_PARAM_MAX]; // 命令附带的参数定义
22     void (*func)(); // 命令的执行函数
23 } shell_cmd; // 命令定义数据结构
24
25 extern char input_buff[SHELL_BUFFER_SIZE]; // 输入缓冲区
26
27 extern shell_cmd shell_cmds[SHELL_CMD_MAX]; // 内置命令
28
29 extern param_unit param_buff[CMD_PARAM_MAX]; // 参数缓冲区
30
31 extern int shell_exit_flag; // shell退出标志

```

6.2 相关功能的设计与实现

在 `cmd.c` 中定义内置命令如下：（截取部分）

```
1  shell_cmd shell_cmds[SHELL_CMD_MAX] = {
2      ...
3      {
4          .cmd = "info",
5          .desc = "show the information of SatorIOS",
6          .params = {
7              {
8                  .sign = 'c',
9                  .name = "cpu",
10                 .desc = "show cpu information",
11             },
12             {
13                 .sign = 'm',
14                 .name = "memory",
15                 .desc = "show memory information",
16             },
17             {
18                 .sign = 'b',
19                 .name = "boot",
20                 .desc = "show boot information",
21             }
22         },
23         .func = show_satori_info
24     }
25     ...
26 };
```

该命令对应的实现在 `impl.c` 中，也可直接调用内核其他部分的函数。shell的主函数如下。

注意：由于我们并未进入保护模式，所以此处并不涉及内核态到用户态的切换。

```
1  void entry_shell()
2  {
3      puts("Entering Shell...");
4      shell_exit_flag = 0;
5      while (!shell_exit_flag)
6      {
7          printf("SatorIOS:%s $ ", shell_path);
8          int n = gets(input_buff, SHELL_BUFFER_SIZE);
9          if (n == SHELL_BUFFER_SIZE)
10         {
11             puts("\n\rInput overflowed!");
12             continue;
13         }
14         if (input_buff[0] == 0)
15             continue;
16         parse_command();
17     }
18     puts("Exiting Shell...");
19 }
```

此处，为方便起见，shell直接使用的我们实现的 `gets` 方法获取键盘输入并进行解析。解析过程在 `parser.c` 中实现，其中重要函数的含义如下：

```
1 void parse_command();           // 解析输入缓冲区中的命令，并将格式化的参数存入参数缓冲
   区，而后调用命令执行函数
2 void parse_params(int cmd_id);  // 由parse_command调用，负责解析命令参数
3 int has_param(int cmd_id);     // 由命令执行函数调用，判断参数缓冲区中是否含有某个参
   数
4 char *get_param(char sign);    // 由命令执行函数调用，获取某个参数的附加值
```

对于每一个命令，可以参照如下格式进行实现，其余不多赘述。

```
1 void show_about_info(int cmd_id)
2 {
3     if (!has_param(cmd_id))
4     {
5         puts("Satori OS is a simple OS written by C.");
6         print_info();
7     }
8     else
9     {
10        if (get_param('v') != 0)
11            puts(VERSION);
12        if (get_param('a') != 0)
13            puts(AUTHOR);
14        if (get_param('c') != 0)
15            puts(COPYRIGHT);
16        if (get_param('l') != 0)
17            puts(LOGO);
18        if (get_param('o') != 0)
19            puts(ORIGIN);
20    }
21 }
```

SatoriOS shell 实际运行截图如下：

```
root@ubuntu: ~/Desktop/satori-os
SatoriOS:/ $ help
test      test the shell command parser
          [-x]
help      show help
echo      echo the input
cls       clear the screen
exit      exit the shell
clock     show clock counter info
          [-s] reset          reset and start clock counter
          [-p] present       show present clock counts
          [-f] speed         modify the speed level
about     show info about Satori OS
          [-v] version       show version info
          [-a] author        show author info
          [-c] copyright     show copyright info
          [-l] logo          show Satori logo
          [-o] origin        show Satori origin
info      show the information of SatoriOS
          [-c] cpu           show cpu information
          [-m] memory        show memory information
          [-b] boot          show boot information
time      show the datetime
ls        list the files in the current directory
          [-l] long          use a long listing format
```

7 内存管理设计与实现（mm）

7.1 SatoriOS/EchoOS 内存管理架构概述

在设计内存管理系统时，我们着重参考了Linux相关的设计理念和设计思路，但由于Linux的内存管理系统过于庞大复杂，我们并未完全遵循Linux的实现方式，而是混入了大量的自己的理解和思考。在实现上，我们以根据现有需求自行实现代码为主，以参考Linux相关概念命名为辅，设计了一个简易可用的内存管理系统。

由于时间原因，这个系统尚有诸多等待完善的部分。同时，由于资料和时间的双重匮乏，我们目前并未对龙芯的硬件架构有透彻的研究，因此并未针对龙芯的相关控制寄存器进行设置，也并未启用硬件的MMU，无法对地址进行翻译转换，从而无法真正意义上进入保护模式，对内存进行段页式的管理。对于这个问题，一方面我们只能使系统仍然在内核态运行，另一方面我们提出了vpu的概念，希望以模拟硬件逻辑的方式来虚拟地实现段页式的内存管理，该想法将在本章第10小节进行详细讨论和阐述。

下面先对内存分配系统整体的设计思路进行介绍。

研究Linux内核可以发现，Linux中重要的内存分配接口主要由kmalloc、vmalloc、malloc等组成。其中kmalloc用于内核空间动态内存分配，其本质是分配大小不定的连续物理内存，依靠slab系统实现。在Linux中，slab系统是以对象为单位的，在buddy system基础上分配较小连续物理内存的系统。Linux底层所使用的页分配器是buddy system。在Linux系统中，vmalloc负责分配虚拟连续，但物理上并不一定连续的内存空间，而malloc则是用于在用户空间进行连续内存分配的接口。

用户/内核		API名称	物理连续？	大小限制	单位	场景
用户空间		malloc/calloc/realloc/free	不保证	堆申请	字节	calloc初始化为0；realloc改变内存大小。
		alloca		栈申请	字节	向栈申请内存
		mmap/munmap				将文件利用虚拟内存技术映射到内存中去。
		brk, sbrk				虚拟内存到内存的映射。sbrk(0)返回program break地址，sbrk调整对的大小。
内核空间		vmalloc/vfree	虚拟连续 物理不定	vmalloc区大小限制	页 VMALLOC区域	可能睡眠，不能从中断上下文调用，或其他不允许阻塞情况下调用。 VMALLOC区域vmalloc_start~vmalloc_end之间，vmalloc比kmalloc慢，适用于分配大内存。
	slab	kmalloc/kcalloc/krealloc/kfree	物理连续	64B-4MB (随slab而变)	2^order字节 Normal区域	大小有限，不如vmalloc/malloc大。 最大/小值由KMALLOC_MIN_SIZE/KMALLOC_SHIFT_MAX，对应64B/4MB。 从/proc/slabinfo中的kmalloc-xxxx中分配，建立在kmem_cache_create基础之上。
		kmem_cache_create	物理连续	64B-4MB	字节大小，需对齐 Normal区域	便于固定大小数据的频繁分配和释放，分配时从缓存池中获取地址，释放时也不一定真正释放内存。通过slab进行管理。
	伙伴系统	__get_free_page/__get_free_pages	物理连续	4MB(1024页)	页 Normal区域	__get_free_pages基于alloc_pages，但是限定不能使用HIGHMEM。
		alloc_page/alloc_pages/free_pages	物理连续	4MB	页 Normal/Vmalloc都可	CONFIG_FORCE_MAX_ZONEORDER定义了最大页面数2^11，一次能分配到的最大页面数是1024。

<https://blog.csdn.net/hithogoo>

在我们设计的操作系统中，我们模仿Linux设计了最简单的 Buddy System 和 Slab Allocator，并在此基础上实现了 kmalloc 函数。由于时间原因，我们并未实现 vmalloc 和 malloc 两个分配器。

在我们实践的过程中，我们发现内存管理实际上存在鸡生蛋蛋生鸡的问题，也就是说，内存管理系统的搭建是为了能够动态的申请和释放内存，而内存管理系统本身在一定程度上也依赖于动态的内存申请和释放。对此，Linux的解决方案是设计了一个 boot_cache 的过程，而我们则采用了更为简单的方式，即在内核空间划分出一段内存用于临时的内核数据结构的动态内存申请，而管理这片内存的分配器是基于位示图原理的 Bit Allocator。

7.2 连续物理内存分配器（kmalloc）设计与实现

我们设计的连续物理内存分配器主要由底层的页分配器 Buddy System、上层的对象分配器 Slab Allocator 以及初始化阶段的临时分配器 Bit Allocator 三个部分组成。

7.2.0 Bit Allocator 设计与实现

位分配器的实现非常简单，其针对位示图进行位的分配操作。其中负责分配的函数主要有 alloc_bits 和 alloc_aligned_bits 两个函数，前者负责忠实的分配内核需要的内存大小，保证内存的利用率，后者则在内核需求的基础上考虑了内存对齐以及分配性能等因素，可以相对快速地分配大块的内存。两个函数的实现如下：

```

1  int alloc_bits(byte *bitmap, int map_size, int size, int *last)
2  {
3      int last_i = *last / 8;
4      int i = last_i, j = *last % 8, k = 0;
5      while (k < size)
6      {
7          if (j == 8)
8          {
9              j = 0;
10             i = (i + 1) % map_size;
11             if (i == last_i)
12                 return -1;
13         }
14         if ((bitmap[i] & (1 << j)) == 0)
15             k++;
16         else
17             k = 0;

```

```

18     j++;
19 }
20 *last = i * 8 + j;
21 for (int l = 0; l < size; l++)
22     set_bit(bitmap, *last - l - 1, 1);
23 return *last - size;
24 }

```

```

1 int alloc_aligned_bits(byte *bitmap, int map_size, int size, int *last)
2 {
3     int last_i = (*last + 7) / 8 % map_size;
4     size = (size + 7) / 8;
5     int i = last_i, k = 0;
6     while (k < size)
7     {
8         if (bitmap[i] == 0)
9             k++;
10        else
11            k = 0;
12        i = (i + 1) % map_size;
13        if (i == last_i)
14            return -1;
15    }
16    *last = i * 8;
17    for (int l = 0; l < size; l++)
18        bitmap[i - l - 1] = 0xFF;
19    return *last - size * 8;
20 }

```

此外，分别有两个函数负责内存的释放，并与上述两个函数——匹配，一般不可混用。由于释放函数地实现较为简单，在此便不做展示。

系统启动之初的位分配器内存分配情况：

```

root@ubuntu: ~/Desktop/satori-os

SatoriOS

The Story Originated on October 1, 2022.
Copyright (C) 2022 Kogler. All Rights Reserved.
Thanks to the Support of Beijing Jiaotong University.
Entering Shell...
SatoriOS:/ $ info -m sys_heap
[MM] | INFO | System heap usage: 288 bytes used, 67108576 bytes free.
- 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
- 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
- 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
- 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
- 11111111 11111111 11111111 11111111 00000000 00000000 00000000 00000000
- 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
- 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
- 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
SatoriOS:/ $

```

7.2.1 Buddy System设计与实现

在我们实现的系统中，伙伴系统是建立在树结构的基础上实现的。使用树实现的优点是代码结构简单，可靠性强，但缺点是内存分配效率相对较低。后续我们会择机对其进行优化改进。我们目前实现的伙伴系统的数据结构及方法的定义如下：

```
1  #define ALLOCATED 1
2  #define FREE 0
3
4  typedef struct buddy_node
5  {
6      struct buddy_node *left;
7      struct buddy_node *right;
8      int order;
9      int state;
10     void* start;
11 } buddy_node;
12
13 void init_buddy();
14 int split_buddy(buddy_node *node);
15 void merge_buddy(buddy_node *node);
16 buddy_node *alloc_buddy(buddy_node *root, int order);
17 void free_buddy(void* start, int order);
18
19 void *buddy_alloc(int size);
20 void buddy_free(void *addr, int size);
21 void* buddy_realloc(void* addr, int old_size, int new_size);
22 void* buddy_calloc(int size);
```

除去一些基本的树结构操作，我们的伙伴系统几个核心的方法实现如下：

```
1  buddy_node *alloc_buddy(buddy_node *root, int order)
2  {
3      if (order > root->order)
4          return NULL;
5      if (order == root->order)
6      {
7          root->state = ALLOCATED;
8          return root;
9      }
10     else
11     {
12         if (root->left == NULL)
13         {
14             if (split_buddy(root) == -1)
15                 return NULL;
16         }
17         void *ret = alloc_buddy(root->left, order);
18         if (ret == NULL)
19             ret = alloc_buddy(root->right, order);
20         return ret;
21     }
22     return NULL;
23 }
```



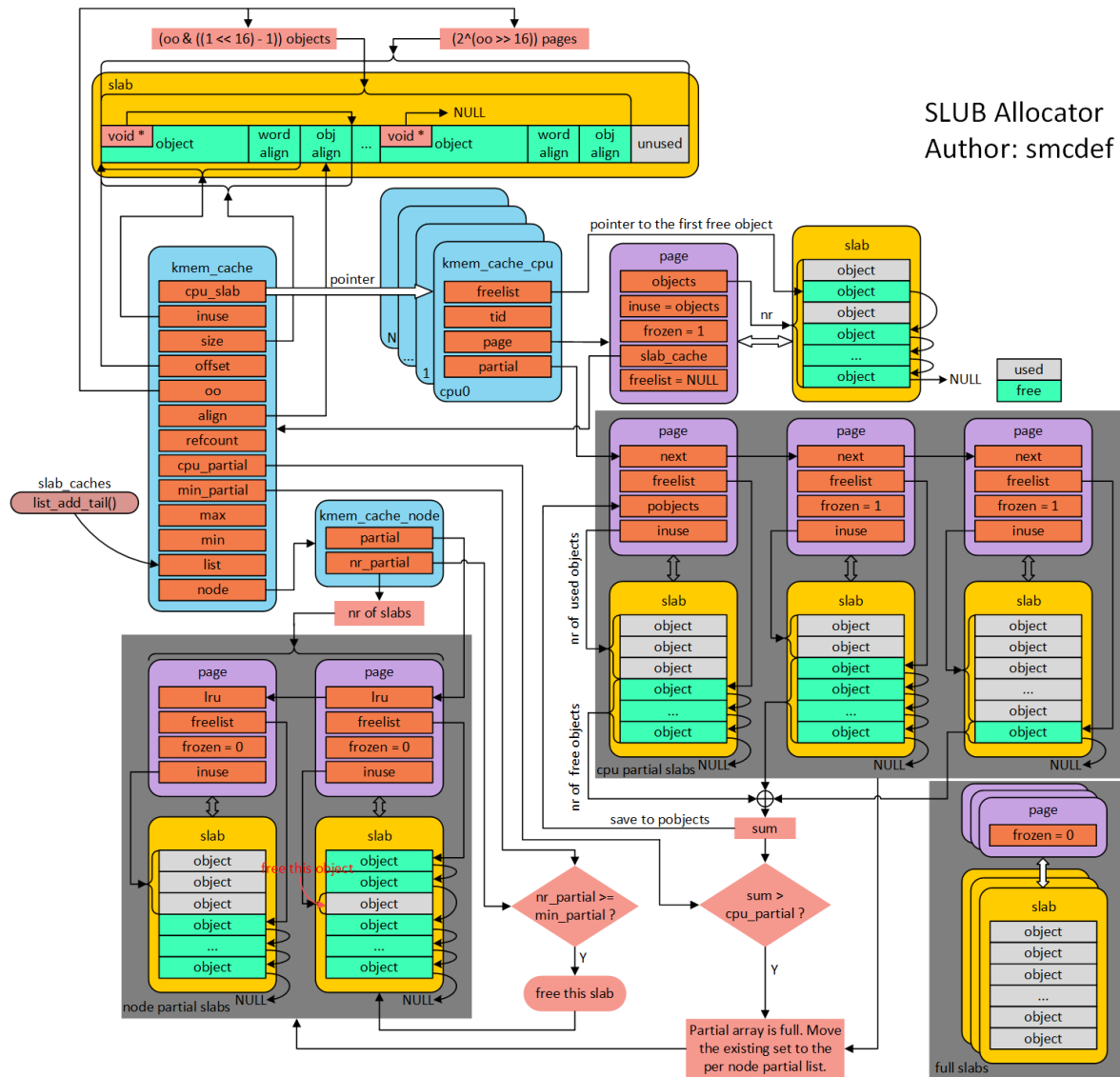
```

24
25 void free_buddy(void *start, int order)
26 {
27     buddy_node *root = &buddy_root;
28     buddy_node *node = root;
29     while (node->order != order)
30     {
31         if (node->left == NULL)
32         {
33             mm_error("free_buddy: invalid operation. (start: %p, order:
34             %d)", start, order);
35             return;
36         }
37         root = node;
38         if (start < node->right->start)
39             node = node->left;
40         else
41             node = node->right;
42     }
43     node->state = FREE;
44     if (root->left->state == FREE && root->right->state == FREE)
45         merge_buddy(root);
46 }
47
48 void *buddy_alloc(int size)
49 {
50     int order = 0;
51     while ((1 << order) < size)
52         order++;
53     buddy_node *node = alloc_buddy(&buddy_root, order);
54     if (node == NULL)
55         return NULL;
56     set_dead_beef(node->start + (1 << order) - 1);
57     return (void *)node->start;
58 }
59
60 void buddy_free(void *addr, int size)
61 {
62     int order = 0;
63     while ((1 << order) < size)
64         order++;
65     if (!check_dead_beef(addr + (1 << order) - 1))
66     {
67         mm_error("buddy_free: Dead beef check failure. (addr: %p, size:
68         %d)", addr, size);
69     }
70     free_buddy(addr, order);
71 }

```

7.2.2 Slab Allocator 设计与实现

由于时间原因，我们自己的Slab分配器尚未完全实现，下面附上Linux的具体实现示意图，感兴趣的同学可以深入了解。



7.3 连续虚拟内存分配器 (vmalloc) 设计与实现

由于时间原因，本功能还处于实验开发阶段，在此不便展示。

7.4 用户地址空间分配器 (malloc) 设计与实现

由于时间原因，本功能还处于实验开发阶段，在此不便展示。

8 进程管理设计与实现 (sched)

8.1 进程控制块设计

进程控制块的设计比较常规，根据龙芯的架构做了相应的调整，这其中也借鉴了 Linux 的设计思路。

- 进程标识符：内/外部、父/子进程、用户标识符
- 处理器状态信息：通用、PC、PSW、用户栈指针寄存器、龙芯控制寄存器
- 进程调度信息：进程状态、进程优先级、事件及其它

- 进程控制信息：程序和数据地址、进程同步通信机制、资源清单、链接指针

```
1 struct loongarch_fpu {
2     unsigned int    fcsr;
3     unsigned int    vcsr;
4     unsigned long int    fcc;    /* 8x8 */
5     union fpureg fpr[NUM_FPU_REGS];
6 };
7
8 struct thread_struct {
9     /* 保存主要的处理器寄存器 */
10    unsigned long reg01, reg02, reg03, reg22; /* ra tp sp fp */
11    unsigned long reg04, reg05, reg06, reg07; /* a0-a3 */
12    unsigned long reg23, reg24, reg25, reg26; /* s0-s3 */
13    unsigned long reg27, reg28, reg29, reg30, reg31; /* s4-s8 */
14    /* 保存控制状态寄存器 */
15    unsigned long csr_prmd;
16    unsigned long csr_crmd;
17    unsigned long csr_euen;
18    unsigned long csr_ecfg;
19    unsigned long csr_badvaddr; //Last user fault
20    /* 保存特权级寄存器 */
21    unsigned long scr0;
22    unsigned long scr1;
23    unsigned long scr2;
24    unsigned long scr3;
25    /* 保存标志寄存器 */
26    unsigned long eflags;
27    /* 其他与进程相关的内容 */
28    unsigned long trap_nr;
29    unsigned long error_code;
30
31    struct loongarch_fpu fpu FPU_ALIGN;
32 };
33
34 struct task_struct
35 {
36     long state;    /* -1 不可运行, 0 可运行, >0 终止 */
37     long counter;
38     long priority;
39     long signal; /* 挂起信号位图 */
40     struct sigaction sigaction[32]; /* 信号的相关信息 */
41     long blocked;    /* 屏蔽信号位图 */
42     /* 进程信息 */
43     int exit_code;
44     unsigned long start_code, end_code, end_data, brk, start_stack;
45     long pid, father, pgrp, session, leader;
46     unsigned short uid, euid, suid;
47     unsigned short gid, egid, sgid;
48     long utime, stime, cutime, cstime, start_time;
49
50     struct thread_struct tss;
51 };
```

8.2 进程调度算法设计与实现

进程调度算法采用时间片轮转，在上述进程控制块的基础上，每当触发时钟中断达到指定的倒计时次数后，会进入进程调度算法，调度算法首先会判断现有各个进程的信号量，判断是否存在未被阻塞且可中断的进程，需要将其置为运行状态。

而后正式进入进程调度的部分，通过判断哪一个进程的计数器值大，则将其确定为下一个占用CPU的程序，调用 `switch_to` 切换进程。

```
1 void schedule (void)
2 {
3     int i, next, c;
4     struct task_struct **p;
5
6     /*通过信号量激活进程，略*/
7
8     /*调度*/
9     while (1)
10    {
11        c = -1;
12        next = 0;
13        i = NR_TASKS;
14        p = &task[NR_TASKS];
15        while (--i)
16        {
17            if (!*--p)
18                continue;
19            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
20                c = (*p)->counter, next = i;
21        }
22        if (c)
23            break;
24        for (p = &LAST_TASK; p > &FIRST_TASK; --p)
25            if (*p)
26                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
27    }
28    switch_to (current, task[next]);
29 }
```

8.3 基于VPU的进程调度设计

VPU 的想法阐述请参见本章第十小节。在 VPU 的基础上，进程切换将变得非常简单。在虚拟进程进行切换时，不再需要进行额外的现场保护，只需将CPU的执行权交由不同的VPU执行即可。

9 文件系统设计与实现 (fs)

由于时间原因，本功能还处于实验开发阶段，在此不便展示。

9.1 文件系统架构概述

9.2 虚拟文件系统（vfs）设计与实现

9.3 内存虚拟硬盘（tfs）设计与实现

9.4 简易文件系统（FAT32）设计与实现

10 虚拟处理单元设计与实现（vpu）

10.1 虚拟处理单元想法概述

由于资料和时间的双重匮乏，我们对龙芯架构的了解还很浅薄，尚不能完成对龙芯CPU的完全控制。对我们来说，龙芯CPU就像是一个黑盒子，我们无法在短期内摸清其关键操纵方法，而这样的盲人摸象可能会在很大程度上阻碍我们研发的进度。进行操作系统实验，一方面是希望对龙芯架构有一定的研究和理解，另一方面，我们也希望在实践的过程中更深入地理解课堂中学习到的知识，去进一步对操作系统这一非具体的概念感兴趣。因此，我们决定走一个折中的路线——既然操纵龙芯CPU存在一定的难度，那我们就用软件的方式构建一个虚拟的处理单元（Virtual Processor Unit, VPU），并在此基础上完成虚拟的指令执行、地址转换、段页管理及用户-内核态的转换，等相关的算法成熟之后再移植到真正的CPU上。我们必须承认，这个想法的工作量很大，我们投入了大量的时间也未能将其完全实现，下面我们仅就已经完成的工作作为简要介绍和展示。

10.1.1 虚拟处理单元设计

经过综合考虑，我们最终确定VPU设计如下：

```
1  typedef struct vpu
2  {
3      u32 gpr[GENERAL_REGISTERS]; // General purpose registers
4      u32 scr[SYSCALL_REGISTERS]; // System call registers
5      segment_registers_t sgr;    // Segment registers
6      segment_register_t *csr;    // Current segment register
7      int ip;                     // Instruction pointer
8      int sp;                     // Stack pointer
9      int bp;                     // Base pointer
10     sint cpl;                   // Current privilege level
11     sint asid;                  // Address space identifier
12     vpu_flags_t flags;          // Flags
13     vdt_entry_t gdtr;           // Global descriptor table register
14     selector_t ldtr;            // Local descriptor table register
15 } vpu_t;
```

下面进行逐一介绍。

```
1  u32 gpr[GENERAL_REGISTERS]; // General purpose registers
```

通用寄存器。每个寄存器占用32位空间，共计16个。

```
1  u32 scr[SYSCALL_REGISTERS]; // system call registers
```

系统调用专用寄存器。用于系统调用参数传递，共8个。这样的设计可能并不规范，仅仅是为了方便。

```
1 | segment_registers_t sgr;    // Segment registers
```

段寄存器组。其详细定义如下，本VPU设计共将应用程序划分成了四段，即代码段、数据段、栈段、堆段。

```
1 | typedef struct segment_registers
2 | {
3 |     segment_register_t cs; // Code segment
4 |     segment_register_t ds; // Data segment
5 |     segment_register_t ss; // Stack segment
6 |     segment_register_t hs; // Heap segment
7 | } segment_registers_t;
```

其中，每一个段寄存器由一个段选择子和一个段描述符组成。这里是参照了8086的经典设计，在设置段选择子时，VPU会自动将该选择子对应的段描述符填入段寄存器中。

```
1 | typedef struct segment_register
2 | {
3 |     selector_t selector;
4 |     virtual_descriptor_t descriptor;
5 | } segment_register_t;
```

其余有关分段设计将在10.2.2进行阐述。

```
1 | segment_register_t *csr;    // Current segment register
```

当前段寄存器。用于标记CPU当前正处于工作的段。该设计暂时没有参考依据，仅仅是为了方便而设置。

```
1 | int ip;                    // Instruction pointer
2 | int sp;                    // Stack pointer
3 | int bp;                    // Base pointer
```

程序计数器、栈顶指针和栈底指针。

```
1 | sint cpl;                  // Current privilege level
2 | sint asid;                  // Address space identifier
```

当前工作特权级、进程地址空间ID。

```
1 | vpu_flags_t flags;        // Flags
```

标志寄存器，详细定义如下：

```

1  typedef struct vpu_flags
2  {
3      u8 cf : 1; // carry flag
4      u8 zf : 1; // zero flag
5      u8 of : 1; // overflow flag
6      u8 sf : 1; // sign flag
7      u8 pf : 1; // parity flag
8      u8 tf : 1; // trace flag
9      u8 rf : 1; // interrupt flag
10 } vpu_flags_t;

```

以上定义参考了经典的CPU设计，并结合实际需求制定。

```

1  vdt_entry_t gdtr;           // Global descriptor table register
2  selector_t ldtr;           // Local descriptor table register

```

全局描述符表寄存器、局部描述符表寄存器。根据8086经典设计，`gdtr`中保存的是全局描述符表的入口信息，而`ldtr`中保存的是其对应的局部描述符表在全局描述符表中的位置，即段选择子。

10.1.2 虚拟处理执行循环

在VPU中，我们模拟了微处理器处理解析并运行指令的完整过程，并将其封装成一个循环，规定每一个循环都是一个原子操作，在执行循环的过程中不可被打断执行。（注意，此处的不可被中断是指，不可被同样是运行在VPU上的其他虚拟进程中断，而并非屏蔽了硬件意义上的中断。事实上，硬件中断后会恢复现场，并不影响虚拟进程的正常执行。）

```

1  void vpu_cycle()
2  {
3      logi_addr_t ip = cur_vpu->ip;
4      phys_addr_t *phys_ip = get_phys_addr(ip);
5      vpu_instr_t *instr = (vpu_instr_t *)phys_ip;
6      cur_vpu->ip += sizeof(vpu_instr_t);
7      vpu_exec_instr(*instr);
8  }

```

那在什么情况下CPU会进入`vpu_cycle`中执行呢？答案是当内核代码在执行空操作时。我们暂时约定，当内核代码在执行空操作时，便会跳转到VPU并执行一个Cycle，这样便可利用CPU的空闲实现来执行VPU的相关任务。事实上，空操作函数已经被替换为如下操作：

```

1  void go_to_vpu_cycle()
2  {
3      if(vpu_exit_flag || !cur_vpu)
4      {
5          return;
6      }
7      if (vpu_switch_flag)
8      {
9          vpu_switch_flag = false;
10         cur_vpu = next_vpu;
11     }
12     vpu_cycle();
13 }

```

从上面的代码中也可以看到，虚拟进程的切换是通过不通过VPU之间交换CPU的执行权来实现的，而且这个过程发生在一个VPU cycle之外。

10.2 虚拟段页式内存管理机构

在建立了初步的VPU概念之后，我们就可以开始大展拳脚，开始模拟实现真正的段页式内存管理系统了。由于由软件模拟实现的虚拟执行单元的所有细节都是由我们决定的，VPU相对于龙芯的CPU来说，对于我们已经不再是一个黑盒子，其所有细节都可以被跟踪确定的。基于以上设计，我们设计了虚拟的段页式内存管理架构如下。

10.2.1 虚拟逻辑地址组成

我们首先对地址类型进行了如下定义：

```
1  #define VADDR_OFT_ODR 12
2  #define VADDR_PGN_ODR 6
3  #define VADDR_PDN_ODR 14
4
5  typedef struct logi_addr
6  {
7      u32 oft : VADDR_OFT_ODR; // offset: 4k
8      u32 pgn : VADDR_PGN_ODR; // page number: 64 pages
9      u32 pdn : VADDR_PDN_ODR; // page directory number: 16k page tables
10 } logi_addr_t;
11
12 typedef addr phys_addr_t;
```

可以看到，我们采用了两级分页结构，内存页的大小为4k，逻辑地址一共占用32位内存空间。处于一些实际的考虑，我们并没有按照课本上讲述的在逻辑地址中加入段号。在我们的虚拟VPU中，分段管理是由操作系统自动完成的，这便意味着，虚拟用户程序的每个段都认为自己拥有32位总计4G的内存空间。

10.2.2 分段分页数据结构

经过对各类资料的学习和分析，我们最终确定的页表项数据结构如下所示：

```
1  typedef struct page
2  {
3      u32 present : 1; // Page present in memory
4      u32 rw : 1; // Read-only if clear, readwrite if set
5      u32 user : 1; // Supervisor level only if clear
6      u32 accessed : 1; // Has the page been accessed since last refresh?
7      u32 dirty : 1; // Has the page been written to since last refresh?
8      u32 unused : 7; // Amalgamation of unused and reserved bits
9      u32 frame : 20; // Frame address (shifted right 12 bits)
10 } page_t;
```

其中物理页框号占20位，结合12位的页内偏移，我们的分页结构可以访问32位的地址空间。

在此基础上的页表和页目录数据结构则呼之欲出：


```

1  typedef struct page_table
2  {
3      int nr_pages;
4      page_t *pages;
5  } page_table_t;
6
7  typedef struct page_directory
8  {
9      int nr_tables;
10     page_table_t *tables;
11 } page_directory_t;

```

以上是分页的数据结构。下面是分段的数据结构：

根据惯例，我们确定了16位的段选择子结构：

```

1  typedef struct selector
2  {
3      u16 rpl : 2;    // Requested Privilege Level
4      u16 tbl : 1;    // Table Indicator (0 = GDT, 1 = LDT)
5      u16 idx : 13;   // Index
6  } selector_t;

```

段选择子用于在描述符表中查找定位某个段的段描述符，而段描述符的数据结构如下：

```

1  typedef struct descriptor
2  {
3      u16 limit_low;           // 段界限的低16位
4      u16 base_low;           // 段基址的低16位
5      u8 base_mid;            // 段基址的中8位
6      descriptor_attrs_t attrs; // 段属性
7      descriptor_flags_t flags; // 段标志
8      u8 base_high;           // 段基址的高8位
9  } descriptor_t;

```

这里段描述符数据结构的设计参考了8086的经典设计，但原设计由于历史兼容原因过于复杂，结合实际的使用需求，我们实际使用的是自定义的虚拟描述符结构：

```

1  typedef struct virtual_descriptor
2  {
3      addr entry;
4      u32 limit;
5      descriptor_attrs_t attrs;
6  } attr_packed virtual_descriptor_t;

```

上述两种描述符结构共用段属性结构，其详细定义如下：

```

1  typedef struct descriptor_attrs
2  {
3      u8 accessed : 1;          // 表明该段是否被访问过，将选择子装入寄存器时，该位被置1
4      u8 writeable : 1;         // 代码段：0表示只执行，1表示可读；数据段：0表示只读，1
表示可读写
5      u8 direction : 1;        // 代码段：0表示非一致码段，1表示一致码段；数据段：0表示
向高位扩展，1表示向低位扩展
6      u8 executable : 1;       // 表明该段是否可执行，如果该位为1，则表示该段是代码段，
否则是数据段
7      u8 descriptor_type : 1; // 表明该段描述符是系统段（0）描述符还是存储段（代码或数
据）描述符
8      u8 privilege_level : 2; // 表明该段的特权级别，0表示最高级别，3表示最低级别
9      u8 present : 1;          // 表明该段是否存在内存中，如果该位为0，则表示该段不存在
内存中
10 } descriptor_attrs_t;

```

10.2.3 虚拟地址变换机构

真实的地址转换依赖于地址变换机构（一般为MMU）才得以实现，为了全面支持分段分页以及保护模式的管理，我们设计了虚拟的地址变换机构，即 Virtual Memory Management Unit, VMMU。其提供一个核心功能：

```

1  phys_addr_t get_phys_addr(logi_addr_t logi_addr)
2  {
3      virtual_descriptor_t *vdesc = cur_vpu->csr->descriptor;
4      page_directory_t *pdir = (page_directory_t *) (vdesc->entry);
5      if ((u32)logi_addr >= vdesc->limit)
6      {
7          vmmu_error("%x - Unhandled segment fault: Exceed seg limit.",
logi_addr);
8          return nullptr;
9      }
10     if (logi_addr.pdn >= pdir->nr_tables)
11     {
12         vmmu_error("%x - Unhandled page fault: Exceed page directory
limit.", logi_addr);
13         return nullptr;
14     }
15     page_table_t *ptable = (page_table_t *) (pdir->tables[logi_addr.pdn]);
16     if (logi_addr.pgn >= ptable->nr_pages)
17     {
18         vmmu_error("%x - Unhandled page fault: Exceed page table limit.",
logi_addr);
19         return nullptr;
20     }
21     page_t *page = (page_t *) (ptable->pages[logi_addr.pgn]);
22     if (!page->present)
23     {
24         vmmu_error("%x - Unhandled page fault: Page not present.",
logi_addr);
25         return nullptr;
26     }

```

```
27     return (page->frame << 12) + logi_addr.ofst;
28 }
```

由于时间原因，部分功能暂未实现。计划实现的函数如下：

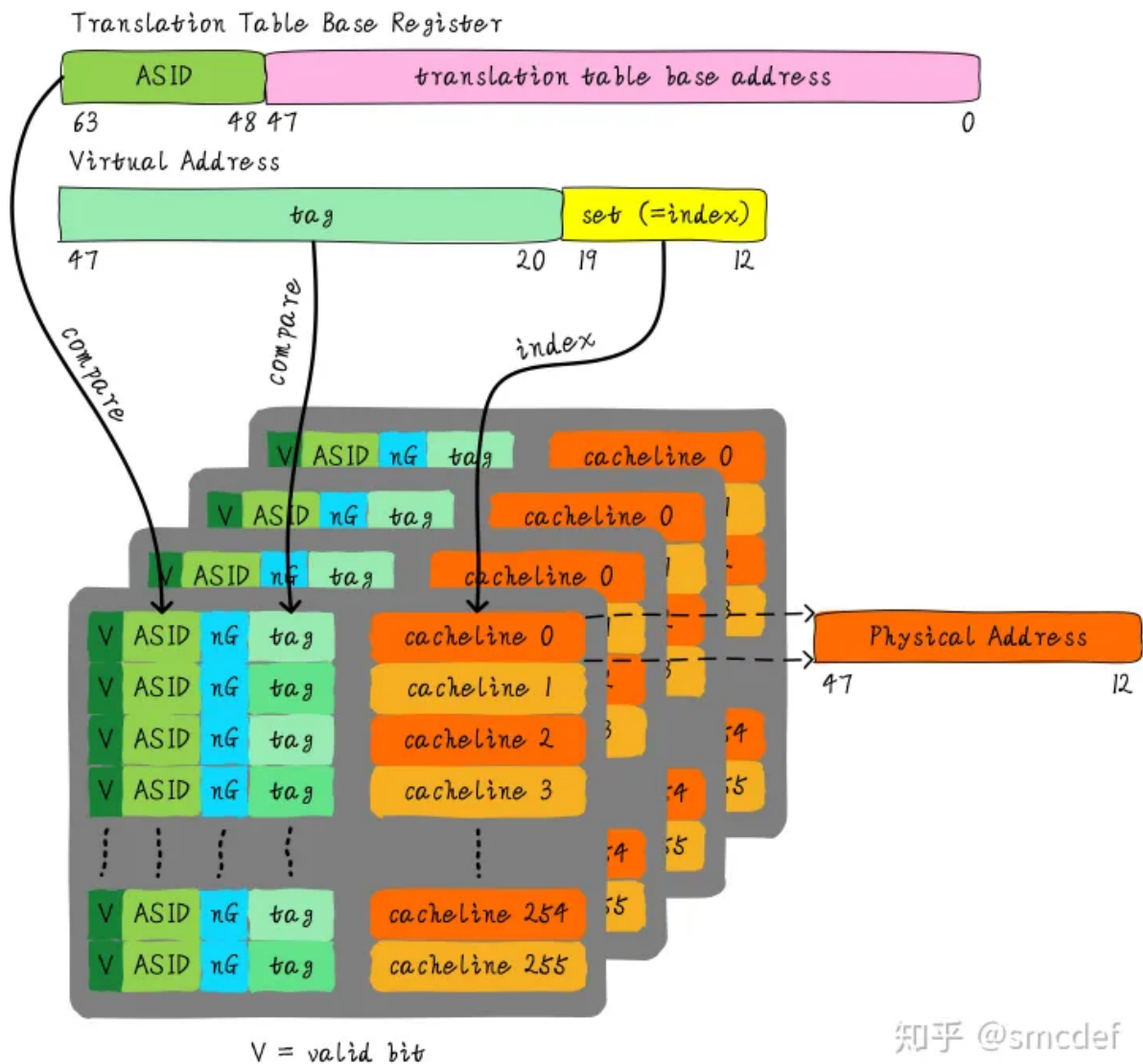
```
1 void handle_page_fault(u32 error_code);
2 void handle_tlb_miss(logi_addr_t logi_addr, phys_addr_t phys_addr);
```

10.2.4 虚拟快表查找机构

为了尽可能模仿真实计算机的运行过程，同时提高虚拟执行单元的工作效率，减轻多级地址变换带来的性能消耗，我们也在尝试设计虚拟快表查找机构。不过经老师提醒，该实现目前并不能很好的起到加速所用，下面仅作简要展示：

```
1 #define VTLB_SIZE 64 // 2^VADDR_PGN_ODR(6)=64
2
3 typedef struct vtlb_entry
4 {
5     byte valid : 1;
6     byte ng : 1; // non global
7     byte asid : 6;
8     u16 tag : VADDR_PDN_ODR;
9     addr frame;
10 } vtlb_entry_t;
11
12 void flush_tlb();
13 void vtlb_insert(logi_addr_t logi_addr, phys_addr_t frame);
14 phys_addr_t *vtlb_lookup(logi_addr_t logi_addr);
```

以上 tlb 表项的设计参考了知乎上的一篇讲解文章，其示意图如下：



我们的虚拟 tlb 查找的过程如下。大致原理是，将虚拟地址中的页号作为表项索引以避免对整个快表的遍历，并将虚拟地址中的页目录号作为tag和表项进行对比以判断是否命中。表项中存储了实际的物理页框号，其结合虚拟地址中的业内偏移量即可计算出真实的物理地址。

```

1 #define get_vtlb_entry(logi_addr) (virtual_tlb + logi_addr.pgn)
2 #define get_vtlb_tag(logi_addr) logi_addr.pdn
3
4 phys_addr_t *vtlb_lookup(logi_addr_t logi_addr)
5 {
6     vtlb_entry_t *entry = get_vtlb_entry(logi_addr);
7     if (!entry->valid || (entry->ng && entry->asid != cur_vpu->asid))
8     {
9         return nullptr;
10    }
11    if (entry->tag == get_vtlb_tag(logi_addr))
12    {
13        return entry->frame << 12 + (logi_addr & 0xfff);
14    }
15    return nullptr;
16 }

```

10.3 虚拟执行过程设计与实现

由于虚拟执行单元是由软件模拟的，所以其并不能执行机器指令。为了将以上虚拟硬件系统利用起来，我们设计了一套简易的模拟指令集，并在此基础上模拟实现了简单的系统调用。下面进行简要介绍。

10.3.1 虚拟指令集设计与实现

经过研究商讨，我们最终确定的虚拟指令集如下。其中共包含算术指令7条，位操作指令6条，逻辑指令2条，分支控制指令10条，堆栈指令2条，内存指令2条，系统调用指令1条，其他指令2条总计32条虚拟机器指令。

```
1  typedef enum vpu_opcode
2  {
3      // Arithmetic
4      ADD,
5      SUB,
6      MUL,
7      DIV,
8      MOD,
9      INC,
10     DEC,
11     // Bitwise
12     AND,
13     OR,
14     XOR,
15     NOT,
16     SHL,
17     SHR,
18     // Logical
19     CMP,
20     TEST,
21     // Control
22     JMP,
23     JZ,
24     JNZ,
25     JG,
26     JGE,
27     JL,
28     JLE,
29     CALL,
30     RET,
31     LOOP,
32     // Stack
33     PUSH,
34     POP,
35     // Memory
36     MOV,
37     LEA,
38     // System
39     SYSCALL,
40     // Misc
41     NOP,
42     HALT
43 } vpu_opcode_t;
```

虚拟指令数据结构如下（虚拟用，并未做压缩优化）：

```
1  typedef enum vpu_operand_type
2  {
3      REGISTER,
4      SYSCALL_REG,
5      IMMEDIATE,
6      MEMORY,
7  } vpu_operand_type_t;
8
9  typedef struct vpu_operand
10 {
11     vpu_operand_type_t type;
12     union
13     {
14         u32 reg;
15         u32 imm;
16         u32 mem;
17     };
18 } vpu_operand_t;
19
20 typedef struct vpu_instr
21 {
22     vpu_opcode_t opcode;
23     vpu_operand_t operands[3];
24 } vpu_instr_t;
```

由上可见，我们设计的指令采用了3操作数结构，其中第一个操作数默认为目标操作数。三个操作数均支持寄存器、立即数和内存单元三种类型。虚拟指令的执行实现如下。由于指令数较多，下面只列出几个典型代表。

```
1  #define _calc_op(op1, op2, op3, op)          \
2      do                                      \
3      {                                      \
4          vpu_switch_seg(SEG_DS);            \
5          u32 op2_val = get_op_value(op2);    \
6          u32 op3_val = get_op_value(op3);    \
7          set_op_value(op1, op2_val op op3_val); \
8      } while (0)
9
10 void vpu_instr_add(vpu_operand_t *op1, vpu_operand_t *op2, vpu_operand_t
    *op3)
11 {
12     _calc_op(op1, op2, op3, +);
13 }
14
15 void vpu_instr_sub(vpu_operand_t *op1, vpu_operand_t *op2, vpu_operand_t
    *op3)
16 {
17     _calc_op(op1, op2, op3, -);
18 }
19
20 void vpu_instr_cmp(vpu_operand_t *op1, vpu_operand_t *op2)
21 {
```

```

22     vpu_switch_seg(SEG_DS);
23     u32 op1_val = get_op_value(op1);
24     u32 op2_val = get_op_value(op2);
25     if (op1_val == op2_val)
26     {
27         cur_vpu->flags.zf = 1;
28     }
29     else
30     {
31         cur_vpu->flags.zf = 0;
32     }
33     if (op1_val < op2_val)
34     {
35         cur_vpu->flags.cf = 1;
36     }
37     else
38     {
39         cur_vpu->flags.cf = 0;
40     }
41 }

```

10.3.2 虚拟系统调用设计与实现

在我们设计的虚拟执行单元中，用户程序可通过 `syscall` 指令陷入内核态，调用系统提供的功能函数。该指令接受一个操作数作为系统调用号，用户可将需要的参数事先放置在系统调用寄存器中。

系统调用指令的实现如下：

```

1 void vpu_instr_syscall(vpu_operand_t *op1)
2 {
3     syscall_entry(get_op_value(op1), cur_vpu->scr);
4 }

```

涉及到系统调用实现部分的数据和方法定义如下：

```

1 typedef void (*syscall_t)(void);
2
3 extern syscall_t syscall_table[SYSCALL_MAX];
4
5 void syscall_entry(int idx, u32* args);

```

由于时间原因，系统调用的功能并未完全实现，因此系统调用表中仅仅存放了一些空白的占位符，大致如下：

```

1 syscall_t syscall_table[SYSCALL_NUM_MAX] = {
2     [SYSCALL_EXIT]      = nullptr,
3     [SYSCALL_FORK]      = nullptr,
4     [SYSCALL_READ]      = nullptr,
5     [SYSCALL_WRITE]     = nullptr,
6     [SYSCALL_OPEN]      = nullptr,
7     [SYSCALL_CLOSE]     = nullptr,
8     [SYSCALL_WAITPID]   = nullptr,

```

```
9     [SYSCALL_CREAT]      = nullptr,  
10    [SYSCALL_LINK]       = nullptr,  
11    [SYSCALL_UNLINK]      = nullptr,  
12    [SYSCALL_EXECVE]     = nullptr,  
13    [SYSCALL_CHDIR]      = nullptr,  
14    [SYSCALL_TIME]       = nullptr,  
15    [SYSCALL_MKNOD]      = nullptr,  
16    [SYSCALL_CHMOD]      = nullptr,  
17    [SYSCALL_LSEEK]      = nullptr,  
18    [SYSCALL_GETPID]     = nullptr,  
19 }
```

10.4 简易汇编器设计

仅仅有虚拟指令并不足够，我们计划为该指令集实现一个配套的汇编器，不过该想法目前仅仅初具雏形，暂不便进行展示。

10.4.1 简易汇编器设计思路

我们计划将汇编分为预处理、符号解析、指令翻译、可执行文件生成等数个阶段。

11 富文本图形库设计与实现 (rtx)

长期以来，我们试图为自己的操作系统构建一个图形化界面。但从零开始的像素级操作的难度可想而知，于是我们打算先从文本化的界面入手。经过前期规划思考，我们大致敲定了一个名为 RTX(Rich Text Graphics) 的可视化框架，并正在设计开发中。

11.1 富文本图形库架构概述

在我们的设想中，该架构主要负责对虚拟显存的管理。用户应用程序可以向RTX申请一块特定大小的虚拟显存，并告诉RTX它想在这块虚拟显存上输出什么数据。RTX则会判断该用户程序目前是否处于最顶层，若是，则在修改显存的同时将更改映射到真正的屏幕上，否则将只会修改显存。系统会向RTX申请一块保留显存用于展示系统的状态信息，以及完成不同应用间的切换等任务。RTX目前提供总计10种窗口尺寸，分别是上中下、左中右两两组合的9种尺寸以及全屏尺寸。

11.2 富文本图形库设计与实现

RTX相关的数据结构及方法暂时定义如下：

```
1  #define hit_align(cur_align, tgt_align) (cur_align & tgt_align)  
2  
3  extern char _rtx_buffer[RTX_BUFFER_LINES][RTX_MAX_WIDTH];  
4  
5  typedef enum rtx_align  
6  {  
7      rtx_align_lft = 1 << 0, // left  
8      rtx_align_mid = 1 << 1, // middle (full width)  
9      rtx_align_rgt = 1 << 2, // right  
10     rtx_align_top = 1 << 3, // top  
11     rtx_align_ctr = 1 << 4, // center (vertical)
```



```

12     rtx_align_btm = 1 << 5, // bottom
13     rtx_align_exp = 1 << 6, // expand (full width and height)
14 } rtx_align;
15
16 typedef struct rtx_char
17 {
18     char ch;
19     sint color;
20     sint style;
21 } rtx_char;
22
23 typedef struct rtx_line
24 {
25     rtx_char *line;
26     sint width;
27     bool blank;
28     struct rtx_line *next;
29     struct rtx_line *prev;
30 } rtx_line;
31
32 typedef struct rtx_buffer
33 {
34     sint prior;
35     sint width;
36     sint height;
37     bool active;
38     sint cursor_x;
39     sint cursor_y;
40     rtx_align align;
41     rtx_line *buf_line; // buffer line
42     rtx_line *pre_line; // previous line
43     rtx_line *cur_line; // current line
44     rtx_line *exs_line; // excessive line
45 } rtx_buffer;
46
47 void rtx_init();
48
49 rtx_buffer *rtx_create_buffer(rtx_align align);
50 void rtx_destroy_buffer(rtx_buffer *buffer);
51
52 void rtx_clear_buffer(rtx_buffer *buffer);
53 void rtx_clear_line(rtx_buffer *buffer, int line);
54 void rtx_clear_char(rtx_buffer *buffer, int x, int y);
55
56 void rtx_set_char(rtx_buffer *buffer, int x, int y, char c);
57 void rtx_set_string(rtx_buffer *buffer, int x, int y, char *str);
58
59 void rtx_append_char(rtx_buffer *buffer, char c);
60 void rtx_append_string(rtx_buffer *buffer, char *str);
61
62 void rtx_set_active(rtx_buffer *buffer);
63 void rtx_set_inactive(rtx_buffer *buffer);
64
65 void rtx_render_buffer(rtx_buffer *buffer);
66 void rtx_render_back(rtx_buffer *buffer);

```

```

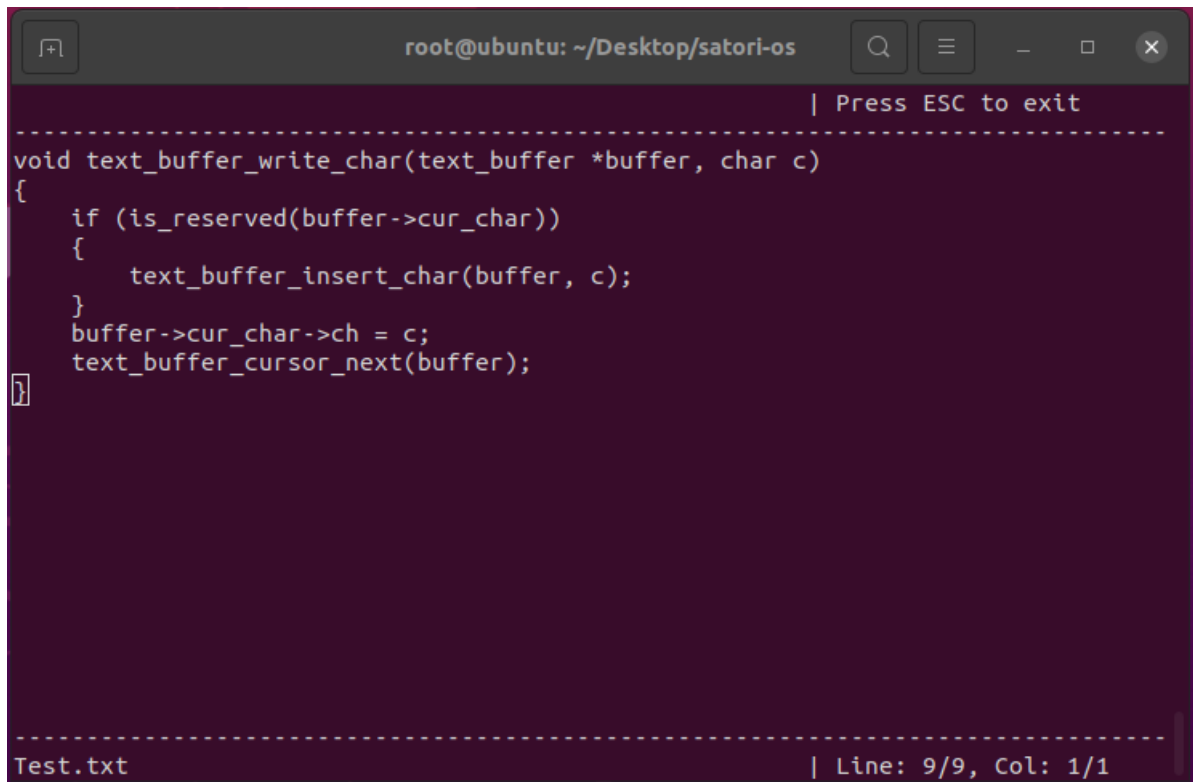
67 void rtx_render_line(rtx_buffer *buffer, int line);
68
69 void rtx_set_cursor(rtx_buffer *buffer, int x, int y);
70 void rtx_move_cursor(rtx_buffer *buffer, int x, int y);
71 void rtx_hide_cursor(rtx_buffer *buffer);
72 void rtx_show_cursor(rtx_buffer *buffer);
73
74 void rtx_set_cursor_style(rtx_buffer *buffer, int style);
75 void rtx_save_cursor_style(rtx_buffer *buffer);
76 void rtx_restore_cursor_style(rtx_buffer *buffer);
77
78 void rtx_set_cursor_color(rtx_buffer *buffer, int color);
79 void rtx_save_cursor_color(rtx_buffer *buffer);
80 void rtx_restore_cursor_color(rtx_buffer *buffer);
81
82 void rtx_roll_up(rtx_buffer *buffer, int lines);
83 void rtx_roll_down(rtx_buffer *buffer, int lines);
84
85 void rtx_render_all();

```

由于时间原因，相关函数仍在开发测试中，在此不便详细展示。

12 简易vim设计与实现

基于上述的可编辑文本数据结构、ANSI控制码，综合键盘驱动等已经实现的模块，我们设计了简易的vim应用，支持基础的文本编辑操作，其效果图如下：



```

root@ubuntu: ~/Desktop/satori-os | Press ESC to exit
-----
void text_buffer_write_char(text_buffer *buffer, char c)
{
    if (is_reserved(buffer->cur_char))
    {
        text_buffer_insert_char(buffer, c);
    }
    buffer->cur_char->ch = c;
    text_buffer_cursor_next(buffer);
}
-----
Test.txt | Line: 9/9, Col: 1/1

```

五、项目统计与心得总结

1 项目代码统计报告

以 `SatoriOS` 为例。

统计工具采用 `VSCodeCounter`

Date : 2022-12-18

`SatoriOS` Total : 140 files, 7279 codes, 456 comments, 1508 blanks, all 9243 lines

`EchooS` Total : 82 files, 4263 codes, 960 comments, 827 blanks, all 6050 lines

1.1 Languages

`SatoriOS`

language	files	code	comment	blank	total
C	69	4,847	429	673	5,949
Markdown	15	1,219	0	466	1,685
C++	36	726	14	247	987
Makefile	16	294	0	98	392
Python	1	112	0	16	128
Shell Script	2	65	13	7	85
CSV	1	16	0	1	17

`EchooS`

language	files	code	comment	blank	total
C	46	3,183	844	538	4,565
C++	19	703	103	144	950
Makefile	13	240	0	80	320
Markdown	2	70	0	57	127
Shell Script	2	67	13	8	88

1.2 Directories

`SatoriOS`

path	files	code	comment	blank	total
.	140	7,279	456	1,508	9,243
include	60	2,071	152	564	2,787
include\app	1	4	0	2	6

path	files	code	comment	blank	total
include\arch	3	211	17	49	277
include\boot	1	81	6	20	107
include\config	3	275	16	20	311
include\drivers	6	346	31	89	466
include\fs	6	56	1	22	79
include\io	5	118	0	38	156
include\isp	3	26	0	12	38
include\lib	4	132	0	45	177
include\mm	9	141	0	53	194
include\sched	2	19	0	6	25
include\shell	3	50	0	14	64
include\sys	1	36	0	13	49
include\temp	1	178	72	47	297
include\vpu	10	342	9	116	467
kernel	71	3,971	291	585	4,847
kernel\app	3	122	19	12	153
kernel\boot	2	161	16	36	213
kernel\config	4	189	6	24	219
kernel\drivers	5	239	5	36	280
kernel\fs	4	283	3	35	321
kernel\io	4	246	8	33	287
kernel\isp	4	19	0	8	27
kernel\lib	4	822	47	70	939
kernel\mm	13	504	127	126	757
kernel\sched	3	22	4	10	36
kernel\shell	6	514	12	31	557
kernel\sys	3	51	0	10	61
kernel\trap	8	295	40	80	415
kernel\vpu	5	438	0	56	494

path	files	code	comment	blank	total
report	1	1,009	0	325	1,334
tools	2	128	0	17	145

EchoOS

path	files	code	comment	blank	total
.	82	4,263	960	827	6,050
include	33	1,530	313	340	2,183
include\app	1	12	0	9	21
include\arch	2	170	16	46	232
include\boot	1	74	13	19	106
include\config	1	4	0	2	6
include\drivers	3	489	43	62	594
include\fs	1	27	1	7	35
include\mm	9	147	0	54	201
include\sched	3	204	132	45	381
include\serial	1	18	1	5	24
include\shell	1	11	0	5	16
include\sysio	2	49	0	8	57
include\temp	1	203	72	47	322
include\utils	3	27	0	7	34
kernel	45	2,653	634	474	3,761
kernel\app	4	303	62	27	392
kernel\boot	2	127	21	32	180
kernel\config	2	43	0	13	56
kernel\drivers	4	243	24	30	297
kernel\fs	2	221	3	24	248
kernel\mm	13	500	128	122	750
kernel\sched	4	361	274	80	715
kernel\shell	2	265	23	21	309
kernel\sysio	2	114	10	20	144

path	files	code	comment	blank	total
kernel\trap	5	247	40	63	350
kernel\utils	3	164	16	19	199

2 项目心得总结

经历过了操作系统的磨练，我们无形之中收获了很多勇气和坚毅。我们经历了数次举步维艰思绪混乱和数次突破重围理清思路，并最终找到了一套架构设计方法：理清概念关系、设计数据结构、设计方法接口、实现完善方法、测试和联调。经此一役，我们历经从一头雾水到柳暗花明、从抵触硬件到得心应手、从一无所有到渐入佳境.....相信这样的一段经历将会是我们大学生活中最为浓墨重彩的一笔！