龙芯操作系统终期进度总结汇报

一、基本要求与实验目标

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。

二、项目命名与内涵阐释

1 Satori Operating System

> 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。

2 Echo Operating System

> 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。

三、平台搭建与环境配置

0 现有系统编译运行

这是一个攀登到巨人肩膀上的工作。

0.1 交叉编译环境配置

这里创建一个shell脚本，用于设置交叉编译器的路径和环境。

env.sh:

```
CC_PREFIX=/opt/cross-tools
export PATH=$CC_PREFIX/bin:$PATH
export LD_LIBRARY_PATH=$CC_PREFIX/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$CC_PREFIX/loongarch64-unknown-linux-gnu/lib/:$LD_LIBRARY_PATH
export ARCH=loongarch
export CROSS_COMPILE=loongarch64-unknown-linux-gnu-
export LC_ALL=C; export LANG=C; export LANGUAGE=C
```

0.2 交叉编译配置文件准备

交叉编译过程中需要的配置文件

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

其内容大体如下：

```
……
CONFIG_CRASH_CORE=y
CONFIG_GENERIC_ENTRY=y
CONFIG_HAVE_64BIT_ALIGNED_ACCESS=y
CONFIG_ARCH_USE_BUILTIN_BSWAP=y
CONFIG_HAVE_IOREMAP_PROT=y
CONFIG_HAVE_NMI=y
CONFIG_TRACE_IRQFLAGS_SUPPORT=y
CONFIG_HAVE_ARCH_TRACEHOOK=y
CONFIG_HAVE_DMA_CONTIGUOUS=y
……
```

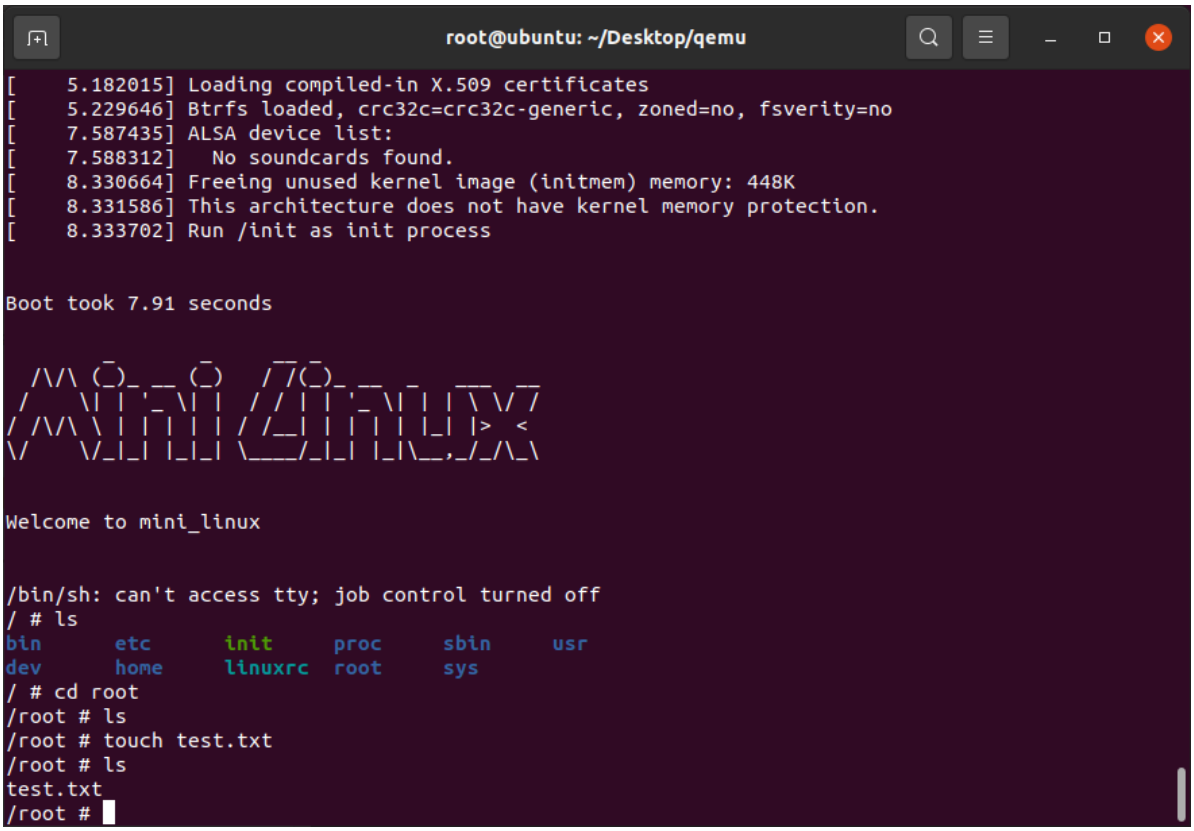
0.3 相关工具和库的安装

```
sudo apt install make
sudo apt install make-doc
sudo apt install make-guile
sudo apt install gcc
sudo apt install flex
sudo apt install bison
sudo apt install libssl-dev
```

0.4 正式编译

```
make clean
source env.sh
make
```

0.5 linux-loongarch-v2022-03-10-1运行结果展示



自此，我们的平台基础和开发环境正式测试完成。

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】

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

虚拟机平台：VMware Workstation 16 Pro V16.2.4

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

四、过关斩将与模块设计

1 文件组织与编译运行（`kernel`）

1.1 项目文件结构介绍

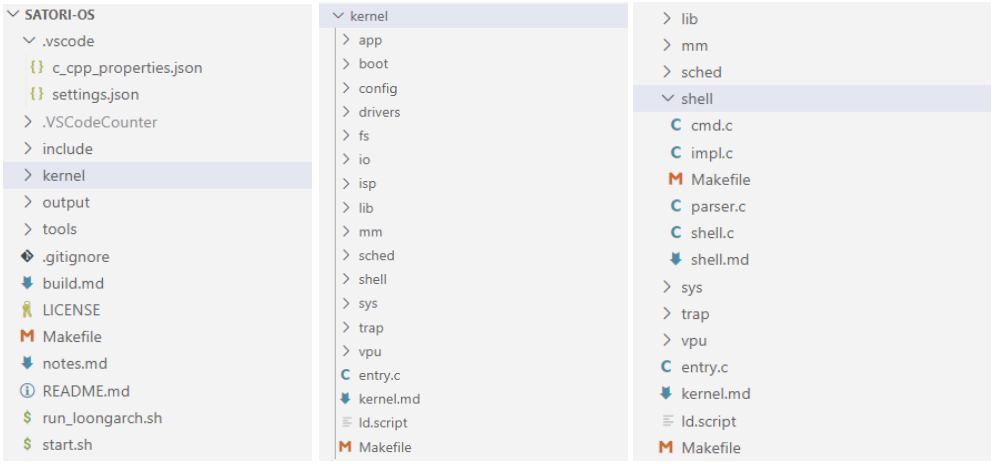
项目的文件结构如下图所示：（以`Satorios`为例）

学习Linux的源文件组织方式，将头文件和C文件分开管理。`iuclude`文件夹用于存放所有的头文件，而`kernel`文件夹用于存放所有的C文件，以及记录参考资料的md文件。

所有的编译中间产生文件均被放置在`build`文件夹中，该文件夹会被动态创建和删除。

`tools`文件夹用于放置一些辅助开发的工具，一般用`python`语言实现，其中包含一个查看内存分布的小工具。

`output`文件夹用于放置一些中间产物（主要是由tools内的工具产生的），暂时并未启用。



为了方便IDE定位头文件所在位置，可在`VSCode`中做如下配置：

```
{
    "configurations": [
        {
            ...
            "includePath": [
                "${workspaceFolder}/include/**",
            ],
            ...
        }
    ],
    ...
}
```

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

```
export TOOLPREFIX := loongarch64-unknown-linux-gnu-

export CC = $(TOOLPREFIX)gcc
export LD = $(TOOLPREFIX)ld

export CFLAGS = -Wall -O2 -g3 \
    -march=loongarch64 -mabi=lp64s \
    -ffreestanding -fno-common \
    -nostdlib \
    -I../include \ # 告知gcc头文件的相对路径
    -fno-stack-protector \
    -fno-pie -no-pie

```

```
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

```
TGTDIR := ../build/ # 编译目标输出文件夹
TARGET := $(TGTDIR)kernel
SOURCE := $(wildcard *.c)
# 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
SUBOBJS = $(filter %.o, $(patsubst ./%, %.o, $(shell find -maxdepth 1 -type d)))
OBJECTS = $(patsubst %.c, $(TGTDIR)%.o, $(SOURCE))
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

```
$(TARGET): $(OBJECTS) ld.script # 根据链接脚本进行链接
    @echo Linking $(TARGET)
    $(ECHOPRE)$(LD) $(LDFLAGS) -T ld.script -o $(TARGET) $(OBJECTS)

$(TGTDIR)%.o : %.c
    @echo Compiling $<
    $(ECHOPRE)$(CC) $(CFLAGS) -c -o $@ $<

$(TGTDIR)%.o : # 递归地执行编译
    @echo -------------------------- == $(subst .o,,$@) == ----------------
-------------
    @mkdir -p $(subst .o,,$@)
    @(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

```
SECNME := mm # 该模块名称
TGTDIR := ../../build/ # 目标产物文件夹
INCDIR := ../../include # 头文件目录
SUBDIR := $(TGTDIR)$(SECNME)/
TARGET := $(TGTDIR)$(SECNME).o
SOURCE := $(wildcard *.c)
OBJECTS := $(patsubst %.c, $(SUBDIR)%.o, $(SOURCE))
CFLAGS += -I../../include

all: $(TARGET)

$(TARGET): $(OBJECTS)
    @echo Linking $(TARGET)
    $(ECHOPRE)$(LD) -r -o $(TARGET) $(OBJECTS)

$(SUBDIR)%.o : %.c
    @echo Compiling $<
    $(ECHOPRE)$(CC) $(CFLAGS) -c -o $@ $<
```

1.3 运行脚本编写说明

2 固件简介与启动装载（`start.sh`）

2.1 UEFI固件简介

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BIOS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vmlinux.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-xxxxxxxx和0x9000xxxx-xxxxxxxx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：

```
OUTPUT_ARCH( "loongarch" )
ENTRY( kernel_entry )

SECTIONS
{
  . = 0x92000000;

  .text : {
    *(.text .text.*)
    PROVIDE(etext = .);
  }

  .rodata : {
    . = ALIGN(16);
    *(.srodata .srodata.*)
    . = ALIGN(16);
    *(.rodata .rodata.*)
  }

  .data : {
    . = ALIGN(16);
    *(.sdata .sdata.*)
    . = ALIGN(16);
```

```
    *(.data .data.*)
  }

  .bss : {
    . = ALIGN(16);
    *(.sbss .sbss.*)
    . = ALIGN(16);
    *(.bss .bss.*)
  }

  PROVIDE(end = .);
}
```

3 内核信息输出模块设计（io）

3.1 显示第一个字符

此时，我们设置了内核入口，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。亟待解决的第一个问题就是——printf。

作为一个操作系统，我们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，需要通过串口通信。

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：

```c
static const unsigned long uart_base = 0x1fe001e0;

#define UART0_THR  (uart_base + 0)
#define UART0_LSR  (uart_base + 5)
#define LSR_TX_IDLE  (1 << 5)

static char io_readb()
{
    return *(volatile char*)UART0_LSR;
}

static void io_writeb(char c)
{
    while ((io_readb() & LSR_TX_IDLE) == 0){
        asm volatile("nop\n" : : : );
    }
    *(char*)UART0_THR = c;
}
```

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）：

```c
void putc(char c)
{
    // wait for Transmit Holding Empty to be set in LSR.
    while ((io_readb() & LSR_TX_IDLE) == 0);
    io_writeb(c);
}

void puts(char *str)
{
    while (*str != 0)
    {
        putc(*str);
        str++;
    }
}
```

3.2 格式化输出及控制

格式化输出函数设计实现如下：

```c
void printf(const char *fmt, ...)
{
    va_list ap;
    int i, c, n;
    char *s;

    if (fmt == 0)
    {
        intr_on();
        return;
    }

    va_start(ap, fmt);
```

```
    for (i = 0; (c = fmt[i] & 0xff) != 0; i++)
    {
        if (c != '%')
        {
            putc(c);
            continue;
        }
        c = fmt[++i] & 0xff;
        if (c == 0)
            break;
        switch (c)
        {
        case 'd':                                               // 打印10进制整数
            print_int(va_arg(ap, int), 10, 1);
            break;
        case 'x':                                               // 打印16进制整数
            print_int(va_arg(ap, int), 16, 1);
            break;
        case 'P':                                               // 打印64位地址
            print_ptr(va_arg(ap, unsigned long));
            break;
        case 'p':                                               // 打印32位地址
            print_ptr_short(va_arg(ap, unsigned long));
            break;
        case 's':                                               // 打印字符串
            if ((s = va_arg(ap, char *)) == 0)
                s = "(null)";
            for (; *s; s++)
                putc(*s);
            break;
        case 'c':                                               // 打印字符
            putc((char)(va_arg(ap, int)));
            break;
        case 'O':                                               // 打印字符串（16位左对齐）
            n = 16;
            if ((s = va_arg(ap, char *)) == 0)
                s = "(null)  ";
            for (; n; s++, n--)
                putc(*s ? *s : ' ');
            break;
        case 'o':                                               // 打印字符串（8位左对齐）
            n = 8;
            if ((s = va_arg(ap, char *)) == 0)
                s = "(null)  ";
            for (; n; s++, n--)
                putc(*s ? *s : ' ');
            break;
```

```
        case '%':
            putc('%');
            break;
        default:
            putc('%');
            putc(c);
            break;
        }
    }
}
```

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：

```
static inline void cursor_move_to(sint x, sint y);           // 光标位置移动到
static inline void clear_screen();                           // 清屏
static inline void set_cursor_style(sint style);             // 设置光标样式
static inline void set_cursor_color(sint color);             // 设置光标（字体）颜色
static inline void set_cursor_background_color(sint color); // 设置光标（字体）背景颜色
void save_cursor_style();       // 保存光标样式
void restore_cursor_style();    // 恢复光标样式
void save_cursor_color();       // 保存光标（字体）颜色
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：

```
#define pr_info(src, fmt, ...)          \
    {                                   \
        save_cursor_color();            \
        set_cursor_color(ANSI_GREEN);   \
        printf("[" #src "] | INFO | "); \
        printf(fmt, ##__VA_ARGS__);     \
        put_char('\n');                 \
        restore_cursor_color();         \
    }

#define pr_warn(src, fmt, ...)          \
    {                                   \
        save_cursor_color();            \
        set_cursor_color(ANSI_YELLOW);  \
        printf("[" #src "] | WARN | "); \
        printf(fmt, ##__VA_ARGS__);     \
        put_char('\n');                 \
        restore_cursor_color();         \
    }

#define pr_error(src, fmt, ...)          \
```

```
    {                                       \
        save_cursor_color();                \
        set_cursor_color(ANSI_RED);         \
        printf("[" #src "] | ERROR | "); \
        printf(fmt, ##__VA_ARGS__);         \
        put_char('\n');                     \
        restore_cursor_color();             \
        die();                              \
    }

#ifdef CONFIG_DEBUG

#define pr_debug(src, fmt, ...)             \
    {                                       \
        save_cursor_color();                \
        set_cursor_color(ANSI_BLUE);        \
        printf("[" #src "] | DEBUG | "); \
        printf(fmt, ##__VA_ARGS__);         \
        put_char('\n');                     \
        restore_cursor_color();             \
    }

#else

#define pr_debug(src, fmt, ...) \
    {                           \
    }

#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：

```
#define mm_info(fmt, ...) pr_info(MM, fmt, ##__VA_ARGS__)
#define mm_warn(fmt, ...) pr_warn(MM, fmt, ##__VA_ARGS__)
#define mm_error(fmt, ...) pr_error(MM, fmt, ##__VA_ARGS__)
#define mm_debug(fmt, ...) pr_debug(MM, fmt, ##__VA_ARGS__)
```

4 中断处理与驱动设计（trap）

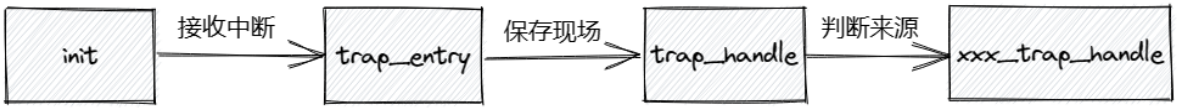
4.1 中断处理流程概述

（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：

```
void trap_init(void)
{
    /*CPU控制状态寄存器设置*/
    unsigned int ecfg = ( 0U << CSR_ECFG_VS_SHIFT ) | HWI_VEC | TI_VEC;
    unsigned long tcfg = 0x0a000000UL | CSR_TCFG_EN | CSR_TCFG_PER;
    w_csr_ecfg(ecfg);
    w_csr_tcfg(tcfg);
    w_csr_eentry((unsigned long)trap_entry);
    /*拓展io中断初始化*/
    extioi_init();
    /*桥片初始化*/
    ls7a_intc_init();
    /*键鼠控制芯片初始化*/
    i8042_init();
}
```

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`csrwr/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	R0	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间距。当 VS=0 时，所有例外和中断的入口地址是同一个。当 VS!=0 时，各例外和中断之间的入口地址间距是 2^VS 条指令。 因为 TLB 重填例外和机器错误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	R0	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 值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，还会自动定时器重新装载成 InitVal 域中配置的初始值，然后再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 的整倍数，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	只读恒为 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：

```
void extioi_init(void)
{
    iocsr_writeq((0x1UL << UART0_IRQ) | (0x1UL << KEYBOARD_IRQ) |
                 (0x1UL << MOUSE_IRQ) | (0x1UL << DISK_IRQ),
                 LOONGARCH_IOCSR_EXTIOI_EN_BASE);

    /* extioi[31:0] map to cpu irq pin INT1, other to INT0 */
    iocsr_writeq(0x01UL,LOONGARCH_IOCSR_EXTIOI_MAP_BASE);

    /* extioi IRQ 0-7 route to core 0, use node type 0 */
    iocsr_writeq(0x0UL,LOONGARCH_IOCSR_EXTIOI_ROUTE_BASE);

    /* nodetype0 set to 1, always trigger at node 0 */
```

```
    iocsr_writeq(0x1,LOONGARCH_IOCSR_EXRIOI_NODETYPE_BASE);
}
```

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：

```
void ls7a_intc_init(void)
{
    /* enable uart0/keyboard/mouse */
    *(volatile unsigned long*)(LS7A_INT_MASK_REG) = ~((0x1UL << UART0_IRQ) | (0x1UL << KEYBOARD_IRQ) |
                                                      (0x1UL << MOUSE_IRQ));

    *(volatile unsigned long*)(LS7A_INT_EDGE_REG) = (0x1UL << (UART0_IRQ | KEYBOARD_IRQ | MOUSE_IRQ));

    /* route to the same irq in extioi */
    *(volatile unsigned char*)(LS7A_INT_HTMSI_VEC_REG + UART0_IRQ) = UART0_IRQ;
    *(volatile unsigned char*)(LS7A_INT_HTMSI_VEC_REG + KEYBOARD_IRQ) = KEYBOARD_IRQ;
    *(volatile unsigned char*)(LS7A_INT_HTMSI_VEC_REG + MOUSE_IRQ) = MOUSE_IRQ;

    *(volatile unsigned long*)(LS7A_INT_POL_REG) = 0x0UL;
}
```

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：

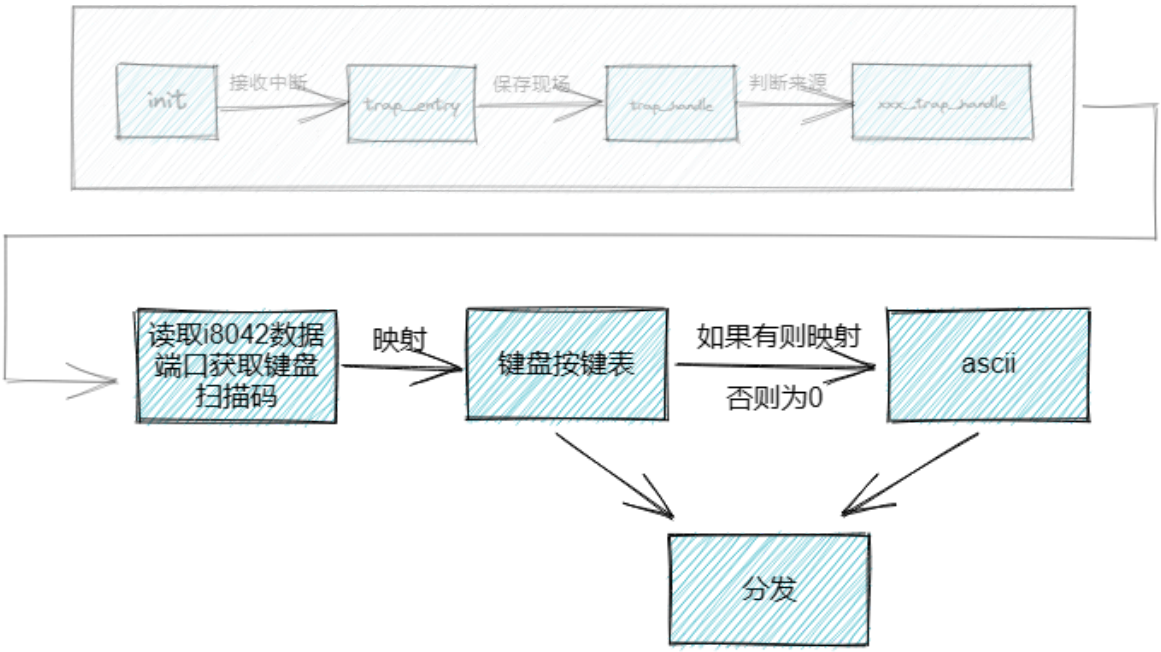
```
void i8042_init(void)
{
  unsigned char data;

  /* disable device */
  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAD;
  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xA7;
  /* flush */
  data = *(volatile unsigned char*)(LS7A_I8042_DATA);
  /* self test */
  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAA;
  data = *(volatile unsigned char*)(LS7A_I8042_DATA);

  /* set config byte, enable device and interrupt*/
  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0x20;
  data = *(volatile unsigned char*)(LS7A_I8042_DATA);
  *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0x60;
  *(volatile unsigned char*)(LS7A_I8042_DATA) = 0x07;

  /* test */
```

```
    *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAB;
    data = *(volatile unsigned char*)(LS7A_I8042_DATA);

    /* enable first port */
    *(volatile unsigned char*)(LS7A_I8042_COMMAND) = 0xAE;

    /* reset device */
    *(volatile unsigned char*)(LS7A_I8042_DATA) = 0xFF;
    data = *(volatile unsigned char*)(LS7A_I8042_DATA);
}
```

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

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。

4.4 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

为了方便内核部分功能的实现，我们首先实现了`string.h`：

```
#ifndef _SYSTEM_LIB_STRING_H_
#define _SYSTEM_LIB_STRING_H_

int strcmp(const char *str1, const char *str2);
int strcpy(char *dst, const char *src);
int strlen(const char *str);
int strncmp(const char *str1, const char *str2, int n);
int strncpy(char *dst, const char *src, int n);
void split(char *str, char *delim, char result[][100], int *result_len);

void memset(void *ptr, char c, unsigned long size);
int memcmp(void *ptr1, void *ptr2, unsigned long size);
void memcpy(void *ptr1, void *ptr2, unsigned long size);

#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。

```
typedef struct std_buffer
{
    byte *data;     // 数据
    int size;       // 已装入数据大小（单位：字节）
    int capacity;   // 缓冲区容量（单位：字节）
    int head;       // 缓冲区头
    int tail;       // 缓冲区尾
    int peek;       // 访问指针
} std_buffer;

std_buffer *std_buffer_create(int capacity);
void std_buffer_destroy(std_buffer *buffer);
void std_buffer_clear(std_buffer *buffer);

void std_buffer_put(std_buffer *buffer, const byte data);
void std_buffer_puts(std_buffer *buffer, const char *data);

byte std_buffer_pop(std_buffer *buffer);
byte std_buffer_get(std_buffer *buffer);

int std_buffer_gets(std_buffer *buffer, char *data, int size);

byte std_buffer_peek(std_buffer *buffer);
void std_buffer_back(std_buffer *buffer);

char std_buffer_wait_char(std_buffer *buffer);

```

```c
int std_buffer_wait_line(std_buffer *buffer, char *data, int size);

static inline int std_buffer_full(std_buffer *buffer)
{
    return buffer->size == buffer->capacity;
}

static inline int std_buffer_full_p(std_buffer *buffer)
{
    return buffer->peek == buffer->head;
}

static inline int std_buffer_empty(std_buffer *buffer)
{
    return buffer->size == 0;
}

static inline int std_buffer_empty_p(std_buffer *buffer)
{
    return buffer->peek == buffer->tail;
}
```

5.3 标准可编辑文本结构设计与实现

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：

```c
typedef struct text_cursor  // 光标位置
{
    int x;
    int y;
} text_cursor;

typedef struct text_char    // 字符结点
{
    char ch;
    struct text_char *next;
    struct text_char *prev;
} text_char;

typedef struct text_line    // 行节点
{
    int nr_chars;
    text_char *fst_char;
    text_char *lst_char;
    struct text_line *next;
    struct text_line *prev;
} text_line;

typedef struct text_buffer  // 可编辑文本缓冲
{
```

```
    int nr_lines;
    text_line *fst_line;
    text_line *lst_line;
    text_line *cur_line;
    text_char *cur_char;
    text_cursor cursor;
} text_buffer;
```

相关操作函数:

```
text_buffer *text_buffer_create();

int text_buffer_count_lines(text_buffer *buffer);
int text_buffer_count_chars(text_buffer *buffer);

void text_buffer_load_text(text_buffer *buffer, char *str);
void text_buffer_save_text(text_buffer *buffer, char *str, int size);

void text_buffer_save_line(text_line *line, char *str, int size);

void text_buffer_clear(text_buffer *buffer);
void text_buffer_destroy(text_buffer *buffer);
void text_buffer_free_line(text_line *line);

void text_buffer_write_char(text_buffer *buffer, char c);

void text_buffer_insert_line(text_buffer *buffer);
void text_buffer_insert_char(text_buffer *buffer, char c);
void text_buffer_insert_string(text_buffer *buffer, char *str);

void text_buffer_split_line(text_buffer *buffer);
void text_buffer_merge_line(text_buffer *buffer);

static inline void text_buffer_newline(text_buffer *buffer)
{
    text_buffer_split_line(buffer);
}

void text_buffer_delete_char(text_buffer *buffer);
void text_buffer_delete_line(text_buffer *buffer);

void text_buffer_backspace(text_buffer *buffer);

void text_buffer_cursor_up(text_buffer *buffer);
void text_buffer_cursor_down(text_buffer *buffer);
void text_buffer_cursor_prev(text_buffer *buffer);
void text_buffer_cursor_next(text_buffer *buffer);

void text_buffer_cursor_move_to(text_buffer *buffer, int x, int y);
void text_buffer_cursor_to_line(text_buffer *buffer, int line);
void text_buffer_cursor_to_col(text_buffer *buffer, int col);

void text_buffer_cursor_home(text_buffer *buffer);
void text_buffer_cursor_end(text_buffer *buffer);

```

```
void text_buffer_cursor_line_home(text_buffer *buffer);
void text_buffer_cursor_line_end(text_buffer *buffer);

void text_buffer_print_info(text_buffer *buffer);

void text_buffer_print_line(text_line *line);
void text_buffer_print_text(text_buffer *buffer);

void text_buffer_relocate_cursor(text_buffer *buffer);
```

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）

```
#define SHELL_BUFFER_SIZE   256 // shell命令输入缓冲区大小
#define SHELL_CMD_MAX       64  // 最大支持内置命令的数量
#define CMD_PARAM_MAX       8   // 每条命令最大支持的参数数量
#define NAME_LEN_MAX        16  // 命令或参数名最大长度
#define DESC_LEN_MAX        64  // 命令或参数描述信息最大长度

typedef struct {
    char sign;
    char name[NAME_LEN_MAX];
    char desc[DESC_LEN_MAX];
} cmd_param;    // 命令参数定义数据结构，sign代表该命令的缩写

typedef struct {
    char sign;
    char param[DESC_LEN_MAX];
} 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

typedef struct {
    char cmd[NAME_LEN_MAX];             // 命令名
    char desc[DESC_LEN_MAX];            // 命令描述信息
    cmd_param params[CMD_PARAM_MAX];    // 命令附带的参数定义
    void (*func)();                     // 命令的执行函数
} shell_cmd;    //命令定义数据结构

extern char input_buff[SHELL_BUFFER_SIZE];  // 输入缓冲区

extern shell_cmd shell_cmds[SHELL_CMD_MAX]; // 内置命令

extern param_unit param_buff[CMD_PARAM_MAX];// 参数缓冲区

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）

```
shell_cmd shell_cmds[SHELL_CMD_MAX] = {
    ...
    {
        .cmd = "info",
        .desc = "show the information of SatoriOS",
        .params = {
            {
                .sign = 'c',
                .name = "cpu",
                .desc = "show cpu information",
            },
            {
                .sign = 'm',
                .name = "memory",
                .desc = "show memory information",
            },
            {
                .sign = 'b',
                .name = "boot",
                .desc = "show boot information",
            }
        },
        .func = show_satori_info
    }
    ...
};
```

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。

```
void entry_shell()
{
    puts("Entering Shell...");
    shell_exit_flag = 0;
    while (!shell_exit_flag)
    {
        printf("SatoriOS:%s $ ", shell_path);
        int n = gets(input_buff, SHELL_BUFFER_SIZE);
        if (n == SHELL_BUFFER_SIZE)
        {
            puts("\n\rInput overflowed!");
            continue;
        }
        if (input_buff[0] == 0)
            continue;
        parse_command();
    }
    puts("Exiting Shell...");
}
```

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：

```
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
void parse_params(int cmd_id);  // 由parse_command调用，负责解析命令参数
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
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。

```
void show_about_info(int cmd_id)
{
    if (!has_param(cmd_id))
    {
        puts("Satori OS is a simple OS written by C.");
        print_info();
    }
    else
    {
        if (get_param('v') != 0)
            puts(VERSION);
        if (get_param('a') != 0)
            puts(AUTHOR);
        if (get_param('c') != 0)
            puts(COPYRIGHT);
        if (get_param('l') != 0)
            puts(LOGO);
        if (get_param('o') != 0)
            puts(ORIGIN);
    }
}
```

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

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

2 项目心得总结

经历过了操作系统的磨练，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

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

从一头雾水到思路清晰、从抵触硬件到实现控制、从一无所有到系统完备