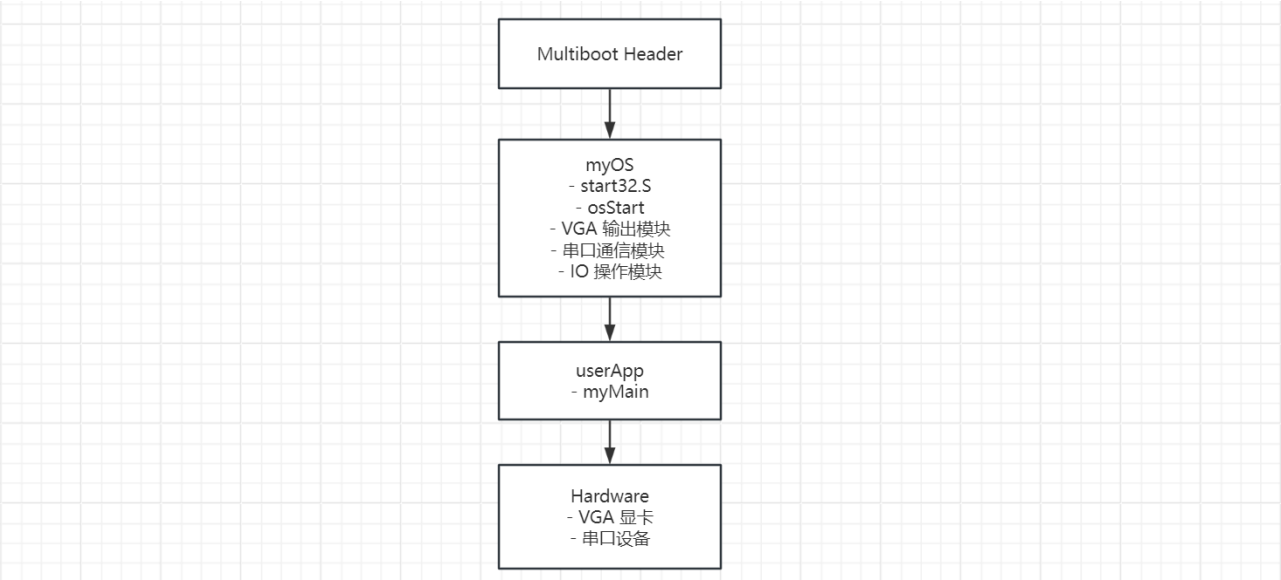
Lab2

PB22030892 刘铠瑜

一、软件架构框图与概述

1. **软件架构框图**



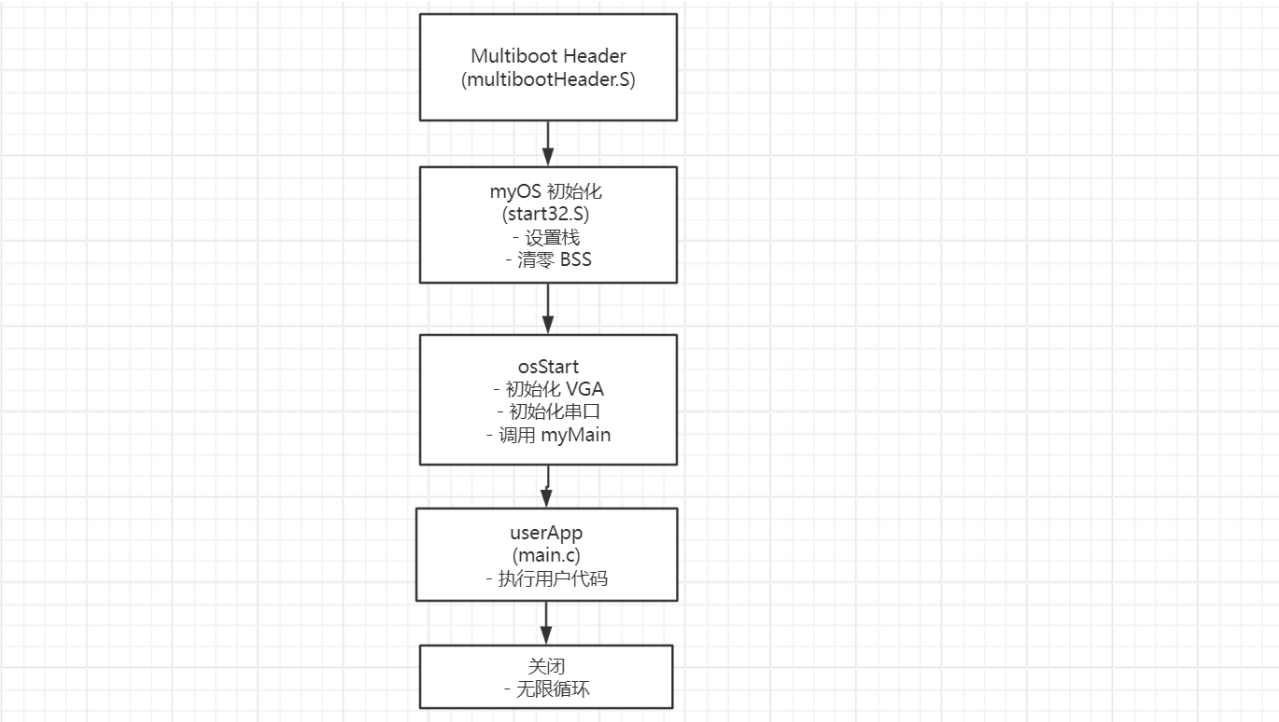
2. **概述** 本实验的软件架构分为三个主要部分：Multiboot Header、myOS 和 userApp。Multiboot Header 是启动加载器识别和加载操作系统的入口点；myOS 是操作系统的内核代码，负责初始化硬件和提供基本的系统功能；userApp 是用户代码，运行在操作系统之上，通过操作系统提供的接口与硬件交互。

二、主流程概述及流程图

1. **概述**

    1. **Multiboot Header**：启动加载器加载操作系统时，首先识别 Multiboot Header，并跳转到 _start 入口点。
    2. **myOS 初始化**：在 _start 入口点，设置栈空间，清零 BSS 段，然后跳转到 osStart 函数。
    3. **osStart**：初始化 VGA 显屏和串口通信，调用 userApp 的 myMain 函数。
    4. **userApp**：执行用户代码，通过操作系统提供的接口进行输出。
    5. **关闭**：用户代码执行完毕后，操作系统进入关闭状态。

2. **流程图**



三、主要功能模块及其实现

1. **Multiboot Header** 文件：`multibootHeader.S` 功能：提供 Multiboot 启动头，确保启动加载器能够正确加载操作系统。 代码实现：

```
.globl start

MAGIC = 0x1BADB002
FLAGS = 0
CHECKSUM = -(MAGIC + FLAGS)

.section ".multiboot_header"
.align 4
.long MAGIC
.long FLAGS
.long CHECKSUM

.text
.code32
start:
    call _start
    hlt
```

2. **VGA 输出模块** 文件：`vga.c` 功能：提供清屏和字符串输出功能，支持光标管理。 代码实现：

```
short cur_line=0;
short cur_column=0;                 //当前光标位置
char *vga_mem = (char *)0xB8000;
```

```
void update_cursor(void){        //通过当前行值cur_cline与列值cur_column回写光标
    unsigned short position = cur_line * 80 + cur_column;

    outb(0x3D4, 0x0F);
    outb(0x3D5, (unsigned char)(position & 0xFF));
    outb(0x3D4, 0x0E);
    outb(0x3D5, (unsigned char)((position >> 8) & 0xFF));
}

short get_cursor_position(void){       //获得当前光标，计算出cur_line和cur_column的
值
    unsigned char low, high;
    short position;

    outb(0x3D4, 0x0F);
    low = inb(0x3D5);
    outb(0x3D4, 0x0E);
    high = inb(0x3D5);

    position = (high << 8) | low;
    cur_line = position / 80;
    cur_column = position % 80;

    return position;
}

void clear_screen(void) {
    for(int i = 0; i < 25 * 80; i++) {
        vga_mem[i * 2] = ' ';
        vga_mem[i * 2 + 1] = 0x07;
    }
    cur_line = 0;
    cur_column = 0;
    update_cursor();
}

void append2screen(char *str,int color) {
    while(*str != '\0') {
        if((*str == '\n') || (cur_column == 80)) {
            // 遇到换行符或行满时，进行换行操作
            cur_column = 0;
            cur_line++;
            if (cur_column == 80) {
                str--;
            }
            if (cur_line >= 25) {
                // 到底部时，进行滚屏操作
                for(int i = 0; i < 24 * 80; i++) {        // 把现在的25行都复制到上一
行
                    vga_mem[i * 2] = vga_mem[(i + 80) * 2];
                    vga_mem[i * 2 + 1] = vga_mem[(i + 80) * 2 + 1];
                }
                for(int i = 24 * 80; i < 25 * 80; i++) {     // 再把第25行清空
                    vga_mem[i * 2] = ' ';
```

```
                    vga_mem[i * 2 + 1] = 0x07;
                }
                cur_line = 24;
            }
            update_cursor();
        } else {
            vga_mem[(cur_line * 80 + cur_column) * 2] = *str;
            vga_mem[(cur_line * 80 + cur_column) * 2 + 1] = color;
            cur_column++;
            update_cursor();
        }
        str++;
    }
}
```

3. **串口通信模块** 文件：uart.c 功能：提供串口发送和接收功能。 代码实现：

```
#define uart_base 0x3F8     // 串口基地址

void uart_put_char(unsigned char c){            // 向串口发送一个字符
    while ((inb(uart_base + 5) & 0x20) == 0);   // 等待发送缓冲区空
    outb(uart_base, c);                         // 发送字符
}

unsigned char uart_get_char(void){              // 从串口接收一个字符
    while ((inb(uart_base + 5) & 0x01) == 0);   // 等待接收缓冲区有数据
    return inb(uart_base);                      // 读取字符 并返回
}

void uart_put_chars(char *str){                 // 向串口发送字符串
    while(*str) {
        uart_put_char(*str++);
    }
}
```

4. **IO 操作模块** 文件：io.c 功能：提供端口输入输出功能。 代码实现：

```
unsigned char inb(unsigned short int port_from){
    unsigned char _in_value;
    __asm__ __volatile__ ("inb %w1,%0":"=a"(_in_value):"Nd"(port_from));
    return _in_value;
}

void outb (unsigned short int port_to, unsigned char value){
    __asm__ __volatile__ ("outb %b0,%w1"::"a" (value),"Nd" (port_to));
}
```

5. **用户程序模块** 文件：main.c 功能：用户代码，通过操作系统接口输出内容。

```c
void myMain(void){
    int i;
    /*NOTE:用户使用myPrintf而不使用myPrink接口*/
    myPrintf(0x7,"main\n");
    for (i=1;i<30;i++) myPrintf(i,"%d\n",i);
    myPrintf(0x7,"PB22030892_LiuKaiyu\n");
    return;
}
```

四、源代码说明

1. **目录组织**

```
src_1/
├── multibootheader/          # Multiboot Header 相关代码
│   └── multibootHeader.S     # Multiboot 启动头
├── myOS/                     # 操作系统内核代码
│   ├── dev/                  # 设备驱动模块
│   │   ├── uart.c            # 串口通信模块
│   │   └── vga.c             # VGA 显示模块
│   ├── i386/                 # x86 架构相关代码
│   │   └── io.c              # IO 操作模块
│   ├── printk/               # 格式化输出模块
│   │   ├── myPrintk.c        # 格式化输出函数
│   │   └── vsprintf.c        # 格式化字符串处理
│   ├── start32.S             # 汇编入口文件
│   └── osStart.c             # 操作系统第一个 C 入口
├── userApp/                  # 用户程序代码
│   └── main.c                # 用户程序入口
├── Makefile                  # 顶层 Makefile
└── source2run.sh             # 编译和运行脚本
```

2. **Makefile组织**

```
# 顶层 Makefile
SRC_RT = $(shell pwd)  # 获取当前工作目录

# 编译器和编译选项
CROSS_COMPILE =  # 交叉编译器前缀，本实验中为空
ASM_FLAGS = -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector  # 汇编文件编译选项
C_FLAGS = -m32 -fno-stack-protector -g  # C 文件编译选项

# 目标文件
.PHONY: all
all: output/myOS.elf  # 默认目标

# 包含子目录的 Makefile
MULTI_BOOT_HEADER = output/multibootheader/multibootHeader.o
```

```
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/Makefile

# 所有目标文件
OS_OBJS = ${MYOS_OBJS} ${USER_APP_OBJS}

# 最终目标文件
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf

# 汇编文件编译规则
output/%.o : %.S
    @mkdir -p $(dir $@)  # 创建目标目录
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

# C 文件编译规则
output/%.o : %.c
    @mkdir -p $(dir $@)  # 创建目标目录
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<

# 清理规则
clean:
    rm -rf output  # 删除 output 目录及其内容
```

五、代码布局说明

1. 链接脚本 (`myOS.ld`) 链接脚本是操作系统编译过程中非常重要的部分，它定义了各个段的布局和内存分配。以下是本实验中使用的链接脚本：

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS {
    . = 1M;  // 代码段从 1MB 开始
    .text : {
        *(.multiboot_header)  // Multiboot Header
        . = ALIGN(8);
        *(.text)  // 操作系统代码
    }

    . = ALIGN(16);
    .data : { *(.data*) }  // 初始化的数据段

    . = ALIGN(16);
    .bss : {
        __bss_start = .;  // BSS 段的起始地址
        _bss_start = .;
        *(.bss)
        __bss_end = .;  // BSS 段的结束地址
```

```
    }

    . = ALIGN(16);
    _end = .;  // 操作系统的结束地址
    . = ALIGN(512);
}
```

2. 地址空间布局

- **代码段（`.text`）**：
  - **起始地址**：1MB（`0x100000`）
  - **内容**：
    - **Multiboot Header**：位于代码段的最前面，用于引导加载器识别和加载操作系统。
    - **操作系统代码**：包括 `_start` 入口点、`osStart` 和其他功能模块的代码。
  - **对齐方式**：代码段的各个部分对齐到 8 字节边界。
- **数据段（`.data`）**：
  - **起始地址**：紧跟在代码段之后，对齐到 16 字节边界。
  - **内容**：包含已初始化的全局变量和静态变量。
- **BSS 段（`.bss`）**：
  - **起始地址**：紧跟在数据段之后，对齐到 16 字节边界。
  - **内容**：包含未初始化的全局变量和静态变量。
  - **符号**：
    - `__bss_start` 和 `_bss_start`：BSS 段的起始地址。
    - `__bss_end`：BSS 段的结束地址。
- **操作系统的结束地址（`_end`）**：
  - **起始地址**：紧跟在 BSS 段之后，对齐到 16 字节边界。
  - **内容**：表示操作系统代码和数据的结束位置。
- **对齐到 512 字节**：
  - **目的**：确保整个操作系统的镜像大小是 512 字节的倍数，便于在磁盘上加载和存储。

六、编译过程说明

1. 清理旧的编译结果 运行以下命令清理旧的编译文件：

```
make clean
```

此命令会删除 `output` 目录及其内容，确保没有旧的目标文件或中间文件。

2. 编译源代码 运行以下命令开始编译：

```
make
```

- **Makefile 中的关键规则**：
  - **汇编文件编译规则**：

```
output/%.o : %.S
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<
```

  - **C 文件编译规则**：

```
output/%.o : %.c
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

- **说明**：
  - 汇编文件（`.S`）和 C 文件（`.c`）分别被编译为目标文件（`.o`）。
  - 编译过程中会创建必要的目录（如 `output`）。

3. 链接目标文件 所有目标文件编译完成后，链接器会将它们链接成最终的可执行文件 `myOS.elf`：

```
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf
```

- **说明**：
  - 使用链接脚本 `myOS.ld` 定义代码段、数据段和 BSS 段的布局。
  - 最终生成的可执行文件为 `output/myOS.elf`。

七、运行及运行结果

1. 运行

```
./source2run.sh
```

- **脚本内容**：

```
#!/bin/sh
make clean
make
if [ $? -ne 0 ]; then
```

```
    echo "make failed"
else
    echo "make succeed"
    qemu-system-i386 -kernel output/myOS.elf -serial stdio
fi
```

- **说明**：
  - 脚本会先清理旧的编译结果，然后编译项目。
  - 编译成功后，启动 QEMU 模拟器运行生成的 `myOS.elf` 文件。

2. 运行结果