一、实验名称

Shell

二、实验目的

1. 理解Shell程序的原理、底层逻辑和Shell依赖的数据结构等
2. 在操作系统内核MiniEuler上实现一个可用的Shell程序
3. 能够根据相关原理编写一条可用的Shell指令

三、实验任务

新建 src/include/prt_shell.h 头文件

```
// 如果未定义 _HWLITEOS_SHELL_H 宏，则进行定义，防止头文件重复包含
#ifndef _HWLITEOS_SHELL_H
// 定义 _HWLITEOS_SHELL_H 宏，表示该头文件已被包含
#define _HWLITEOS_SHELL_H

// 包含 prt_typedef.h 头文件
#include "prt_typedef.h"

// 定义常量 SHELL_SHOW_MAX_LEN，值为 272，用于限制 shell 缓冲区的最大长度
#define SHELL_SHOW_MAX_LEN    272
// 定义常量 PATH_MAX，值为 1024，用于限制文件路径的最大长度
#define PATH_MAX        1024

// 定义 ShellCB 结构体，用于存储 shell 相关的控制块信息
typedef struct {
    // 控制台 ID，用于标识控制台
    U32   consoleID;
    // shell 任务的句柄，用于标识 shell 任务
    U32   shellTaskHandle;
    // shell 条目的句柄，用于标识 shell 条目
    U32   shellEntryHandle;
    // 指向命令键链的指针
    void     *cmdKeyLink;
    // 指向命令历史键链的指针
    void     *cmdHistoryKeyLink;
    // 指向命令屏蔽键链的指针
    void     *cmdMaskKeyLink;
```

```
    // shell 缓冲区的偏移量
    U32   shellBufOffset;
    // shell 缓冲区的读取偏移量
    U32   shellBufReadOffset;
    // 键类型
    U32   shellKeyType;
    // 用于存储 shell 输入的字符缓冲区，最大长度为 SHELL_SHOW_MAX_LEN
    char     shellBuf[SHELL_SHOW_MAX_LEN];
    // 存储 shell 的当前工作目录，最大长度为 PATH_MAX
    char     shellWorkingDirectory[PATH_MAX];
} ShellCB;

// 结束条件编译，若定义了 _HWLITEOS_SHELL_H 宏，则不再包含该头文件内容
#endif /* _HWLITEOS_SHELL_H */
```

接收输入

QEMU的virt机器默认没有键盘作为输入设备，但当我们执行QEMU使用 -nographic 参数（disable graphical output and redirect serial I/Os to console）时QEMU会将串口重定向到控制台，因此我们可以使用UART作为输入设备。

在 src/bsp/print.c 中的 PRT_UartInit 添加初始化代码，使其支持接收数据中断。 同时定义了用于串口接收的信号量 sem_uart_rx。

```
// 包含自定义的头文件 "prt_sem.h"
#include "prt_sem.h"
// 包含自定义的头文件 "prt_shell.h"
#include "prt_shell.h"

// 定义 UART 控制寄存器使能位，将第 0 位设为 1，用于使能 UART
#define UARTCR_UARTEN (1 << 0)
// 定义 UART 控制寄存器发送使能位，将第 8 位设为 1，用于使能发送功能
#define UARTCR_TXE (1 << 8)
// 定义 UART 控制寄存器接收使能位，将第 9 位设为 1，用于使能接收功能
#define UARTCR_RXE (1 << 9)

// 定义 UART 中断清除寄存器全部清除位，将第 0 位设为 1，用于清除所有中断
#define UARTICR_ALL (1 << 0)

// 定义 UART 中断屏蔽寄存器接收中断使能位，将第 4 位设为 1，用于使能接收中断
#define UARTIMSC_RXIM (1 << 4)

// 定义 UART 整数波特率除数寄存器掩码，用于设置波特率的整数部分
```

```
#define UARTIBRD_IBRD_MASK 0xFFFF
// 定义 UART 分数波特率除数寄存器掩码，用于设置波特率的小数部分
#define UARTFBRD_FBRD_MASK 0x3F

// 定义 UART 线控制寄存器高字节掩码，用于设置数据位长度、奇偶校验等
#define UARTLCR_H_WLEN_MASK (3 << 5)
// 定义 UART 线控制寄存器奇偶校验使能位，将第 1 位设为 1，用于使能奇偶校验
#define UARTLCR_H_PEN (1 << 1)
// 定义 UART 线控制寄存器停止位选择位，将第 3 位设为 0，选择 1 个停止位
#define UARTLCR_H_STP1 (0 << 3)

// 定义信号量句柄，用于 UART 数据接收信号量
SemHandle sem_uart_rx;

// 声明外部函数 OsGicIntSetConfig，用于设置中断配置
extern void OsGicIntSetConfig(uint32_t interrupt, uint32_t config);
// 声明外部函数 OsGicIntSetPriority，用于设置中断优先级
extern void OsGicIntSetPriority(uint32_t interrupt, uint32_t priority);
// 声明外部函数 OsGicEnableInt，用于使能中断
extern void OsGicEnableInt(U32 intId);
// 声明外部函数 OsGicClearInt，用于清除中断
extern void OsGicClearInt(uint32_t interrupt);//在我的代码中需要改为
OsGicClearIntPending(uint32_t interrupt)

// 声明外部函数 PRT_Printf，用于格式化输出信息
extern U32 PRT_Printf(const char *format,...);

// 函数 PRT_UartInit 用于初始化 UART，返回值为 U32 类型
U32 PRT_UartInit(void)
{
    // 定义变量 result 用于存储函数执行结果，初始化为 0
    U32 result = 0;
    // 定义变量 reg_base 存储 UART 寄存器基地址，初始化为 UART_0_REG_BASE
    U32 reg_base = UART_0_REG_BASE;

    // 向 UART 控制寄存器写入 0，禁用 pl011（通用异步收发传输器）
    UART_REG_WRITE(0, (unsigned long)(reg_base + 0x30));
    // 向 UART 中断清除寄存器写入 0x7ff，清空中断状态
    UART_REG_WRITE(0x7ff, (unsigned long)(reg_base + 0x44));
    // 向 UART 中断屏蔽寄存器写入 UARTIMSC_RXIM，设定中断 mask，使能接收中断
    UART_REG_WRITE(UARTIMSC_RXIM, (unsigned long)(reg_base + 0x38));
    // 向 UART 数据寄存器写入 13
    UART_REG_WRITE(13, (unsigned long)(reg_base + 0x24));
```

```
    // 向 UART 数据寄存器写入 1
    UART_REG_WRITE(1, (unsigned long)(reg_base + 0x28));

    // 从 UART 线控制寄存器高字节读取当前值
    result = UART_REG_READ((unsigned long)(reg_base + DW_UART_LCR_HR));
    // 将读取的值与 UARTLCR_H_WLEN_MASK、UARTLCR_H_PEN、UARTLCR_H_STP1 和
DW_FIFO_ENABLE 进行按位或操作
    result = result | UARTLCR_H_WLEN_MASK | UARTLCR_H_PEN | UARTLCR_H_STP1 |
DW_FIFO_ENABLE;
    // 将处理后的值写回 UART 线控制寄存器高字节，启用 8N1（8 位数据，无奇偶校验，1 个
停止位）FIFO
    UART_REG_WRITE(result, (unsigned long)(reg_base + DW_UART_LCR_HR));

    // 向 UART 控制寄存器写入 UARTCR_UARTEN | UARTCR_RXE | UARTCR_TXE，启用
pl011 的 UART 功能、接收和发送功能
    UART_REG_WRITE(UARTCR_UARTEN | UARTCR_RXE | UARTCR_TXE, (unsigned long)
(reg_base + 0x30));

    // 设置中断号为 33 的中断配置，具体配置值为 0，可省略
    OsGicIntSetConfig(33, 0);
    // 设置中断号为 33 的中断优先级为 0
    OsGicIntSetPriority(33, 0);
    // 清除中断号为 33 的中断标志，可省略
    OsGicClearInt(33);
    // 使能中断号为 33 的中断
    OsGicEnableInt(33);

    // 调用 PRT_SemCreate 函数创建 uart 数据接收信号量，初始值为 0
    U32 ret;
    ret = PRT_SemCreate(0, &sem_uart_rx);
    // 如果创建信号量失败，返回错误信息并返回 1
    if (ret != OS_OK) {
        PRT_Printf("failed to create uart_rx sem\n");
        return 1;
    }

    // 初始化成功，返回 OS_OK
    return OS_OK;
}
```

```
(base) xiaoye@localhost:~/OSlab/lab10$ grep -rn --include=\*.{c,h} 'OsGicClearInt' /home/xiaoye/OSlab/lab10
/home/xiaoye/OSlab/lab10/src/bsp/hwi_init.c:76:OS_SEC_L4_TEXT void OsGicClearIntPending(uint32_t interrupt)
/home/xiaoye/OSlab/lab10/src/bsp/print.c:45:extern void OsGicClearInt(uint32_t interrupt);
/home/xiaoye/OSlab/lab10/src/bsp/print.c:72:    OsGicClearInt(33); //可省略
/home/xiaoye/OSlab/lab10/src/bsp/timer.c:9:extern void OsGicClearIntPending(uint32_t interrupt);
grep: /home/xiaoye/OSlab/lab10/src/bsp/timer.c: binary file matches
```

由图可知在hwi_init.c中定义的函数是OsGicClearIntPending，而不是OsGicClearInt。

简单起见，在 src/bsp/print.c 中实现 OsUartRxHandle() 处理接收中断。

```
// 声明变量 g_shellCB，类型为 ShellCB，用于存储外壳相关的控制块信息
extern ShellCB g_shellCB;//不过这样声明之后会报错，extern需要去掉
// 定义 UART 接收处理函数 OsUartRxHandle，无返回值，无参数
void OsUartRxHandle(void)
{
    // 定义 U32 类型变量 flag，用于存储 UART 状态寄存器的值，初始化为 0
    U32 flag = 0;
    // 定义 U32 类型变量 result，用于存储从 UART 读取的数据，初始化为 0
    U32 result = 0;
    // 定义 U32 类型变量 reg_base，存储 UART 寄存器基地址，值为 UART_0_REG_BASE
    U32 reg_base = UART_0_REG_BASE;

    // 从 UART 状态寄存器（地址为 reg_base + 0x18）读取值，赋值给 flag
    flag = UART_REG_READ((unsigned long)(reg_base + 0x18));
    // 当 flag 的第 4 位为 0 时，进入循环，等待 UART 接收数据就绪
    while((flag & (1<<4)) == 0)
    {
        // 从 UART 数据寄存器（地址为 reg_base + 0x0）读取接收到的数据，赋值给
result
        result = UART_REG_READ((unsigned long)(reg_base + 0x0));
        // 注释掉的代码，若取消注释，可将接收到的字符打印到控制台
        // PRT_Printf("%c", result);

        // 将接收到的字符（转换为 char 类型）存入 g_shellCB 的缓冲区 shellBuf 中，
偏移量为 shellBufOffset
        g_shellCB.shellBuf[g_shellCB.shellBufOffset] = (char) result;
        // shellBufOffset 自增，指向下一个缓冲区位置
        g_shellCB.shellBufOffset++;
        // 如果 shellBufOffset 等于 SHELL_SHOW_MAX_LEN，说明缓冲区已满，将偏移量
重置为 0
        if (g_shellCB.shellBufOffset == SHELL_SHOW_MAX_LEN)
            g_shellCB.shellBufOffset = 0;

        // 发送信号量 sem_uart_rx，通知其他等待该信号量的任务可以继续执行
        PRT_SemPost(sem_uart_rx);
        // 再次从 UART 状态寄存器（地址为 reg_base + 0x18）读取值，更新 flag，用于
```

```
下一次循环判断
        flag = UART_REG_READ((unsigned long)(reg_base + 0x18));
    }
    // 函数执行完毕，返回
    return;
}
```

```
(base) xiaoye@localhost:~/OSlab/lab10$ grep -rn --include=\*.{c,h} 'g_shellCB' /home/xiaoye/OSlab/lab10
/home/xiaoye/OSlab/lab10/src/bsp/print.c:47:extern ShellCB g_shellCB;
/home/xiaoye/OSlab/lab10/src/bsp/print.c:216:        // 将收到的字符存到g_shellCB的缓冲区
/home/xiaoye/OSlab/lab10/src/bsp/print.c:217:        g_shellCB.shellBuf[g_shellCB.shellBufOffset] = (char) result;
/home/xiaoye/OSlab/lab10/src/bsp/print.c:218:        g_shellCB.shellBufOffset++;
/home/xiaoye/OSlab/lab10/src/bsp/print.c:219:        if (g_shellCB.shellBufOffset == SHELL_SHOW_MAX_LEN)
/home/xiaoye/OSlab/lab10/src/bsp/print.c:220:            g_shellCB.shellBufOffset = 0;
```

由图片可知，在print.c文件中，g_shellCB是第一次出现，去掉前面的extern。不然，链接的时候会报错。

在 src/bsp/prt_exc.c 中OsHwiHandleActive() 链接中断和处理函数OsUartRxHandle()

```
// 声明外部函数 OsTickDispatcher，该函数无参数和返回值
extern void OsTickDispatcher(void);
// 声明外部函数 OsUartRxHandle，该函数无参数和返回值
extern void OsUartRxHandle(void);
// 定义一个内联函数 OsHwiHandleActive，用于处理硬件中断，参数 irqNum 为中断号，类型为 U32
OS_SEC_ALW_INLINE INLINE void OsHwiHandleActive(U32 irqNum)
{
    // 根据中断号 irqNum 进行分支处理
    switch(irqNum){
        // 当中断号为 30 时
        case 30:
            // 调用 OsTickDispatcher 函数处理系统时钟节拍中断
            OsTickDispatcher();
            // 注释掉的打印语句，可能用于调试，打印一个点表示时钟节拍中断处理
            // PRT_Printf(".");
            // 跳出 switch 语句
            break;
        // 当中断号为 33 时
        case 33:
            // 调用 OsUartRxHandle 函数处理 UART 接收中断
            OsUartRxHandle();
        // 当中断号不是 30 也不是 33 时，执行 default 分支
        default:
            // 不做任何处理，直接跳出 switch 语句
            break;
```

```
    }
}
```

在 src/kernel/task/prt_task.c 中加入函数

```
// 声明外部函数 PRT_Printf，该函数用于格式化输出，返回值为 U32 类型
// 参数 format 为格式化字符串，... 表示可变参数列表
extern U32 PRT_Printf(const char *format,...);
// 定义一个操作系统安全级别的函数 OsDisplayTasksInfo，无返回值，无参数
OS_SEC_TEXT void OsDisplayTasksInfo(void)
{
    // 定义一个指向 struct TagTskCb 结构体的指针 taskCb，并初始化为 NULL
    struct TagTskCb *taskCb = NULL;
    // 定义一个 U32 类型的变量 cnt，用于计数，初始化为 0
    U32 cnt = 0;

    // 调用 PRT_Printf 函数，输出表头信息，包含 PID、Priority、Stack Size 三列
    PRT_Printf("\nPID\t\tPriority\tStack Size\n");
    // 使用 LIST_FOR_EACH 宏遍历 g_runQueue 队列
    // taskCb 为当前遍历到的节点指针，&g_runQueue 为队列头地址
    // struct TagTskCb 为结构体类型，pendList 为节点中用于链表的成员名
    // 遍历过程中，每次将当前节点的 taskPid、priority、stackSize 信息输出
    LIST_FOR_EACH(taskCb, &g_runQueue, struct TagTskCb, pendList) {
        // 计数器 cnt 自增，记录任务数量
        cnt++;
        // 调用 PRT_Printf 函数，输出当前任务的 PID、优先级、堆栈大小信息
        PRT_Printf("%d\t\t%d\t\t%d\n", taskCb->taskPid, taskCb->priority,
taskCb->stackSize);
    }
    // 输出任务总数信息，使用之前计数的 cnt 变量
    PRT_Printf("Total %d tasks", cnt);
}
```

在 src/kernel/tick/prt_tick.c 中加入函数

```
// 声明外部函数 PRT_Printf，该函数用于格式化输出信息，返回值为 U32 类型
// 参数 format 为格式化字符串，... 表示可变参数列表
extern U32 PRT_Printf(const char *format,...);
// 定义操作系统安全级别的函数 OsDisplayCurTick，无返回值，无参数
OS_SEC_TEXT void OsDisplayCurTick(void)
{
    // 调用 PRT_Printf 函数，在控制台输出换行符和提示信息 "Current Tick: "
    // 并调用 PRT_TickGetCount() 函数获取当前系统时钟节拍数，作为参数传递给
```

```
PRT_Printf 函数进行输出
    PRT_Printf("\nCurrent Tick: %d", PRT_TickGetCount());
}
```

shell 处理

新建 src/shell/shmsg.c 文件

```
// 包含头文件 "prt_typedef.h"
#include "prt_typedef.h"
// 包含头文件 "prt_shell.h"
#include "prt_shell.h"
// 包含头文件 "os_attr_armv8_external.h"
#include "os_attr_armv8_external.h"
// 包含头文件 "prt_task.h"
#include "prt_task.h"
// 包含头文件 "prt_sem.h"
#include "prt_sem.h"

// 声明外部信号量句柄 sem_uart_rx，用于 UART 接收信号量
extern SemHandle sem_uart_rx;
// 声明外部函数 PRT_Printf，用于格式化输出信息，返回值为 U32 类型
extern U32 PRT_Printf(const char *format,...);
// 声明外部函数 OsDisplayTasksInfo，用于显示任务信息，无返回值，无参数
extern void OsDisplayTasksInfo(void);
// 声明外部函数 OsDisplayCurTick，用于显示当前系统时钟节拍，无返回值，无参数
extern void OsDisplayCurTick(void);

// 定义操作系统安全级别的函数 ShellTask，用于实现 shell 任务
// 参数 param1、param2、param3、param4 为 uintptr_t 类型，用于传递任务相关参数
OS_SEC_TEXT void ShellTask(uintptr_t param1, uintptr_t param2, uintptr_t
param3, uintptr_t param4)
{
    // 定义 U32 类型变量 ret，用于存储函数执行结果，初始化为 0
    U32 ret;
    // 定义字符变量 ch，用于存储从 UART 接收到的字符
    char ch;
    // 定义字符数组 cmd，大小为 SHELL_SHOW_MAX_LEN，用于存储用户输入的命令
    char cmd[SHELL_SHOW_MAX_LEN];
    // 定义 U32 类型变量 idx，用于记录命令数组的索引，初始化为 0
    U32 idx;
    // 将 param1 强制转换为 ShellCB 类型的指针，赋值给 shellCB，用于操作外壳控制块
    ShellCB *shellCB = (ShellCB *)param1;
```

```
    // 进入无限循环，持续处理 shell 任务
    while (1) {
        // 调用 PRT_Printf 函数，在控制台输出提示信息 "miniEuler # "
        PRT_Printf("\nminiEuler # ");
        // 将 idx 重置为 0，用于清空命令数组
        idx = 0;
        // 循环清空命令数组 cmd 的每个元素
        for(int i = 0; i < SHELL_SHOW_MAX_LEN; i++)
        {
            // 将命令数组的每个元素赋值为 0
            cmd[i] = 0;
        }

        // 进入无限循环，等待并处理 UART 接收到的数据
        while (1){
            // 等待 UART 接收信号量 sem_uart_rx，阻塞直到信号量可用
            PRT_SemPend(sem_uart_rx, OS_WAIT_FOREVER);

            // 从 shellCB 的缓冲区中读取字符，偏移量为 shellCB-
>shellBufReadOffset
            ch = shellCB->shellBuf[shellCB->shellBufReadOffset];
            // 将读取的字符存入命令数组 cmd 的当前索引位置
            cmd[idx] = ch;
            // 命令数组索引 idx 自增
            idx++;
            // shellCB 的缓冲区读取偏移量自增
            shellCB->shellBufReadOffset++;
            // 如果读取偏移量达到命令数组的最大长度
            if(shellCB->shellBufReadOffset == SHELL_SHOW_MAX_LEN)
                // 将读取偏移量重置为 0，实现循环缓冲
                shellCB->shellBufReadOffset = 0;

            // 调用 PRT_Printf 函数，将接收到的字符回显到控制台
            PRT_Printf("%c", ch);
            // 如果接收到的字符为回车符 '\r'
            if (ch == '\r'){
                // 判断命令是否为 "top"，如果是则调用 OsDisplayTasksInfo 函数显示
任务信息
                if(cmd[0]=='t' && cmd[1]=='o' && cmd[2]=='p'){
                    OsDisplayTasksInfo();
                // 判断命令是否为 "tick"，如果是则调用 OsDisplayCurTick 函数显示
当前时钟节拍
```

```
                } else if(cmd[0]=='t' && cmd[1]=='i' && cmd[2]=='c' &&
cmd[3]=='k'){
                    OsDisplayCurTick();
                }
                // 跳出内层循环，继续等待下一次 UART 接收数据
                break;
            }

        }
    }
}

// 定义操作系统安全级别的函数 ShellTaskInit，用于初始化 shell 任务
// 参数 shellCB 为指向 ShellCB 结构体的指针，用于传递外壳控制块信息
OS_SEC_TEXT U32 ShellTaskInit(ShellCB *shellCB)
{
    // 定义 U32 类型变量 ret，用于存储函数执行结果，初始化为 0
    U32 ret = 0;
    // 定义 TskInitParam 结构体变量 param，并初始化为 0
    struct TskInitParam param = {0};

    // 设置任务入口函数为 ShellTask
    param.taskEntry = (TskEntryFunc)ShellTask;
    // 设置任务优先级为 9
    param.taskPrio = 9;
    // 设置任务堆栈大小为 0x1000（4096 字节）
    param.stackSize = 0x1000;
    // 将 shellCB 的地址赋值给任务的参数数组 args 的第一个元素
    param.args[0] = (uintptr_t)shellCB;

    // 定义 TskHandle 类型变量 tskHandle1，用于存储任务句柄
    TskHandle tskHandle1;
    // 调用 PRT_TaskCreate 函数创建任务，将任务句柄存入 tskHandle1
    ret = PRT_TaskCreate(&tskHandle1, &param);
    // 如果任务创建失败，返回错误码
    if (ret) {
        return ret;
    }

    // 调用 PRT_TaskResume 函数恢复任务，使其进入就绪状态
    ret = PRT_TaskResume(tskHandle1);
    // 如果任务恢复失败，返回错误码
    if (ret) {
```

```
        return ret;
    }
}
```

提示：将新增文件加入构建系统
更改main.c文件如下：

```
#include "prt_typedef.h"

#include "prt_tick.h"

#include "prt_task.h"

#include "prt_sem.h"

#include "prt_shell.h"

extern U32 PRT_Printf(const char *format, ...);

extern void PRT_UartInit(void);

extern U32 OsActivate(void);

extern U32 OsTskInit(void);

extern U32 OsSemInit(void);

extern U32 OsHwiInit(void);

extern void ShellTask(uintptr_t param1, uintptr_t param2, uintptr_t param3,
uintptr_t param4);

extern U32 ShellTaskInit(ShellCB *shellCB);

extern void CoreTimerInit(void);

extern ShellCB g_shellCB;

static SemHandle sem_sync;

static SemHandle sem_uart_rx;
```

```
S32 main(void)

{



    OsHwiInit();

    OsTskInit();

    OsSemInit();

    CoreTimerInit();

    PRT_UartInit();




    U32 ret;

    ret = ShellTaskInit(&g_shellCB);

    if(ret != OS_OK){

        PRT_Printf("ERROR:falied to create a ShellTask \n");

        return 1;

    }

    OsActivate();

    while(1);

    return 0;

}
```

运行结果如下：

```
[100%] Built target miniEuler
(base) xiaoye@localhost:~/OSlab/lab10$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s

 miniEuler # top
PID             Priority        Stack Size
1               71              4096
0               21              4096
Total 2 tasks
 miniEuler # tick
Current Tick: 9008
```

作业

作业1

实现一条有用的 shell 指令。

> **思路**
>
> 1.实现相应命令字符串的匹配
> 2.调用相应的函数/直接在ShellTask中实现相应的操作

clear

clear - Clear the tick

```
else if(cmd[0]=='c' && cmd[1]=='l' && cmd[2]=='e' && cmd[3]=='a' &&
cmd[4]=='r'){

                    OsClearTick();

   }
```

```
OS_SEC_TEXT void OsClearTick(void)

{

    g_uniTicks = 0;//清零

    PRT_Printf("Tick count cleared.\n");

    PRT_Printf("Current Tick: 0");

}
```

help

help - Display help message

```
else if(cmd[0]=='h' && cmd[1]=='e' && cmd[2]=='l' && cmd[3]=='p'){

            PRT_Printf("\ntop - Display tasks information");

            PRT_Printf("\ntick - Display current tick count");

            PRT_Printf("\nclear - Clear the tick");

            PRT_Printf("\nhelp - Display help message");

            PRT_Printf("\nquit - Exit the process");

            }
```

quit

quit - Exit the process

```
else if(cmd[0]=='q' && cmd[1]=='u' && cmd[2]=='i' && cmd[3]=='t')

        {

            PRT_Printf("\nThank you for using it! Looking forward to next
use!")    ;

            flag=0;

         }
```

最后的shmsg.c文件中的ShellTask函数为：

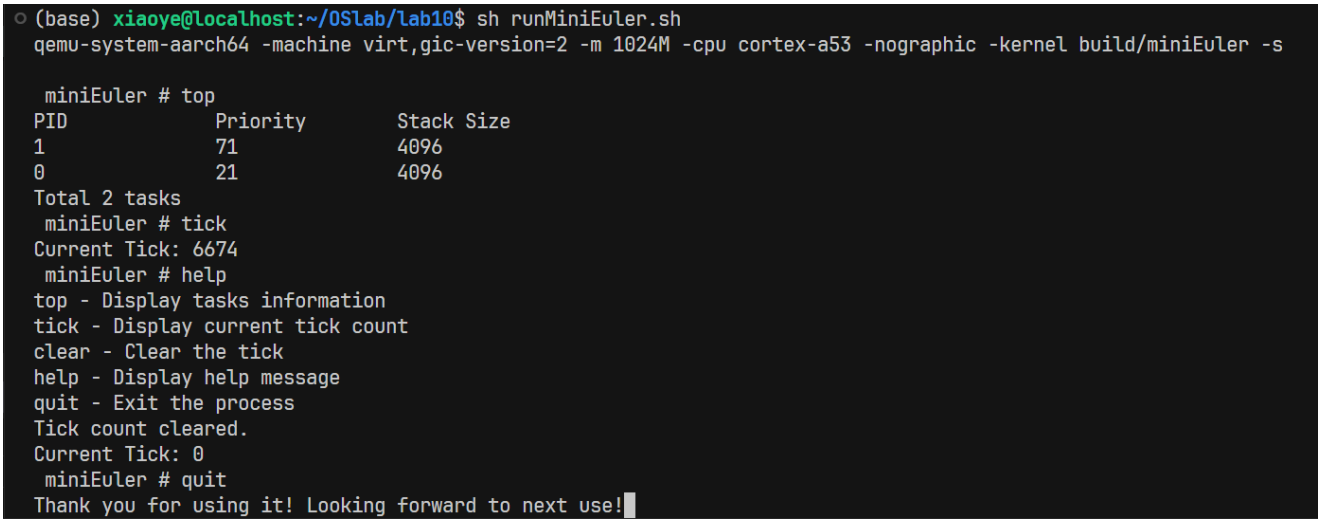
```
OS_SEC_TEXT void ShellTask(uintptr_t param1, uintptr_t param2, uintptr_t
param3, uintptr_t param4)

{

    U32 ret;

    char ch;
```

```
char cmd[SHELL_SHOW_MAX_LEN];

U32 idx;

ShellCB *shellCB = (ShellCB *)param1;

int flag=1;

while (flag) {

    PRT_Printf("\n miniEuler # ");

    idx = 0;

    for(int i = 0; i < SHELL_SHOW_MAX_LEN; i++)

    {

        cmd[i] = 0;

    }

    while (1){

        PRT_SemPend(sem_uart_rx, OS_WAIT_FOREVER);



        // 读取shellCB缓冲区的字符

        ch = shellCB->shellBuf[shellCB->shellBufReadOffset];

        cmd[idx] = ch;

        idx++;

        shellCB->shellBufReadOffset++;

        if(shellCB->shellBufReadOffset == SHELL_SHOW_MAX_LEN)

            shellCB->shellBufReadOffset = 0;
```

```
            PRT_Printf("%c", ch); //回显

             if (ch == '\r'){

                // PRT_Printf("\n");

                if(cmd[0]=='t' && cmd[1]=='o' && cmd[2]=='p'){

                    OsDisplayTasksInfo();//top

                } else if(cmd[0]=='t' && cmd[1]=='i' && cmd[2]=='c' &&
cmd[3]=='k'){

                    OsDisplayCurTick();//tick

                }

                else if(cmd[0]=='c' && cmd[1]=='l' && cmd[2]=='e' &&
cmd[3]=='a' && cmd[4]=='r'){

                   OsClearTick();//clear

                   }

                else if(cmd[0]=='h' && cmd[1]=='e' && cmd[2]=='l' &&
cmd[3]=='p'){
                //help

            PRT_Printf("\ntop - Display tasks information");

            PRT_Printf("\ntick - Display current tick count");

            PRT_Printf("\nclear - Clear the tick");

            PRT_Printf("\nhelp - Display help message");

            PRT_Printf("\nquit - Exit the process");

            }

        else if(cmd[0]=='q' && cmd[1]=='u' && cmd[2]=='i' && cmd[3]=='t')
```

```
        //quit

        {

            PRT_Printf("\nThank you for using it! Looking forward to next
use!")    ;

            flag=0;

         }

                break;

            }

    }

    }

}
```

运行结果：



OK啦~
至此，操作系统的所有实验就完成啦。

实验总结

- 实验让我深刻理解了操作系统中 Shell 的重要性，以及它如何与用户交互。我也学习到了如何在操作系统层面处理用户输入和执行命令。此外，通过实现具体的 Shell 指令，我加深了对操作系统命令处理机制的理解。
- 至此，操作系统课程的所有实验落下帷幕，我本人是感慨万分的。从最开始的什么都不懂，到逐步开始找资料、配环境，学习进步，顺利完成所有的实验，实现了一个自己的简

单操作系统内核：MiniEuler，还是非常有成就感的。十分感谢有这个机会能接触到这么底层的实验，让我逐渐了解原本神秘的操作系统。当然啦，这肯定只是前行的小小一步。未来的学习之路道阻且长。