《操作系统》

实验二报告

目录

1 实验代码分析

1.1 实验目的

理解操作系统与硬件的接口方法，并实现一个可打印字符的函数（非系统调用），用于后续的调试和开发。

1.2 了解 virt 机器

virt 是一个与任何真实硬件不相关的平台，专为虚拟机设计。这种板类型提供了一个与真实硬件无关的、标准化的虚拟环境，使得虚拟机可以在不同的物理硬件上运行而无需进行修改。

```
qemu-system-aarch64 -machine virt,dumpdtb=virt.dtb -cpu cortex-a53 -nographic
dtc -I dtb -O dts -o virt.dts virt.dtb
```

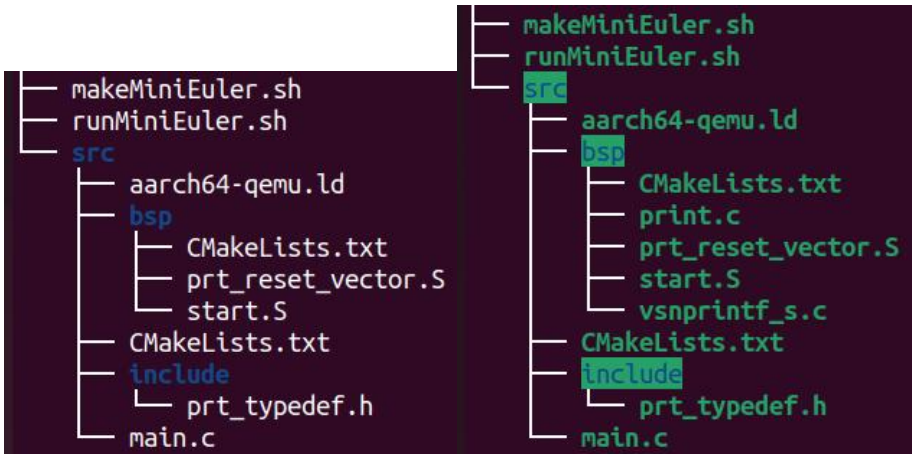
第一条命令使用 QEMU 来模拟一个基于 ARM 64 位（aarch64）的虚拟机。指定使用“virt”这个机器类型、在 QEMU 启动过程中，将 Device Tree Blob (DTB) 导出到名为“virt.dtb”的文件中、指定虚拟机使用的 CPU 型号为 Cortex-A53、指定不使用图形界面而使用命令行内控制。第二条命令使用 Device Tree Compiler (dtc) 将 Device Tree Blob (DTB) 文件转换为 Device Tree Source (DTS) 文件。

```
15         pl011@9000000 {
14                 clock-names = "uartclk\0apb_pclk";
13                 clocks = <0x8000 0x8000>;
12                 interrupts = <0x00 0x01 0x04>;
11                 reg = <0x00 0x9000000 0x00 0x1000>;
10                 compatible = "arm,pl011\0arm,primecell";
 9         };
```

```
5         chosen {
6                 stdout-path = "/pl011@9000000";
7                 kaslr-seed = <0x4bed8c96 0xa4d1309>;
8         };
```

打开转换后的 virt.dts 文件，可以找到如上内容。virt 机器包含有 pl011 的设备，该设备的寄存器在 0x9000000 开始处。它是一个 UART 设备，即串口。virt 选择使用 pl011 作为标准输出。

1.3 代码结构



可以对应 lab1 的结构来看，make 和 run 两个脚本文件用于自动化编译和执行；aarch64-qemu.ld 是创建的自定义链接器脚本，通常包含对内存布局的描述、符号解析的规则、代码和数据段的布局以及如何将这些段映射到输出文件中的特定位置；**bsp 目录存放与硬件紧密相关的代码**，include 目录中存放项目的大部分头文件；prt_typedef.h 头文件：它是 UniProton 所使用的基本数据类型和结构的定义，如 U8、U16、U32、U64 等；main.c 是我们的主函数，Start 和 prt_reset_vector.S 也是执行的一部分。

本实验在 bsp 文件夹中新增了 print.c 和 vsnprintf_s.c 文件，同时在 src/bsp/CMakeLists.txt 中增加了链接方式，并修改了 main.c 文件和 start.S 文件。

1.4 PRT_Printf 函数的实现

1.4.1 print.c 定义宏

首先包含所需头文件<stdarg.h> 和 "prt_typedef.h"，然后定义后续所用的宏，与 UART（通用异步收发传输器）通信相关。

首先定义 UART _0 的基地址。

接着定义了几个具体的寄存器地址偏移量，如 DW_UART_THR、DW_UART_FR 和 DW_UART_LCR_HR，它们都是相对于 UART_0_REG_BASE 的偏移。该定义在提供的链接中（即上表）可以找到。

定义了发送 FIFO 非满的位掩码和启用发送和接收 FIFO 的位掩码。

定义了超时值和最大显示长度。

定义宏 READ 从给定的地址（addr）读取 32 位值，使用 volatile 关键字确保编译器不会优化掉这些访问。定义宏 WRITE 用于向给定的地址（addr）写入一个 32 位值（value）。同样使用了 volatile 关键字。

Summary of registers

Table 3.1 lists the UART registers.

Offset	Name	Type	Reset	Width	Description
0x000	UARTDR	RW	0x---	12/8	Data Register, UARTDR
0x004	UARTRSR/ UARTECR	RW	0x0	4/0	Receive Status Register / Error Clear Register, UARTRSR/UARTECR
0x008-0x014	-	-	-	-	Reserved
0x018	UARTFR	RO	0b--10010----	9	Flag Register, UARTFR
0x01C	-	-	-	-	Reserved
0x020	UARTILPR	RW	0x00	8	IrDA Low-Power Counter Register, UARTILPR
0x024	UARTIBRD	RW	0x0000	16	Integer Baud Rate Register, UARTIBRD
0x028	UARTFBRD	RW	0x00	6	Fractional Baud Rate Register, UARTFBRD
0x02C	UARTLCR_H	RW	0x00	8	Line Control Register, UARTLCR_H
0x030	UARTCR	RW	0x0300	16	Control Register, UARTCR

1.4.2 print.c 串口初始化

使用 UART_REG_READ 宏从 UARTLCR_H 寄存器（可见于上图）（通过基地址和偏移量计算得到实际地址）读取当前的配置值，并存储在 result 变量中。

使用位或操作(|)将 DW_FIFO_ENABLE(即启用 FIFO 的位掩码)与先前从 UARTLCR_H 寄存器读取的值合并并写回，从而启用 FIFO。

启用 FIFO 的原因：当使用 UART 进行通信时，数据通常是以字节为单位进行传输的。但是，在某些情况下，数据的产生和消耗速度可能不匹配，这可能导致数据丢失或中断。若启用 FIFO，可以防止数据丢失、平滑数据流、减少中断频率、提高数据传输效率。

若初始化成功返回 OS_OK。

1.4.3 print.c 向串口发送字符实现输出

通过轮询的方式向 PL011 的数据寄存器 DR 写入数据即可实现往串口发送字符，实

现字符输出。

uart_reg_read 读 reg_base + offset 寄存器的值（**读寄存器函数**）。uart_is_txfifo_full 通过检查 FR 寄存器的标志位确定发送缓冲是否满，满时返回 1（使用 uart_reg_read）（**检查标志位 TXFF 函数**）。uart_reg_write 往 reg_base + offset 寄存器中写入值 val（**写寄存器函数**）。

uart_poll_send 通过轮询的方式发送字符到串口，如果缓冲区没满，通过往数据寄存器写入数据发送字符到串口（使用 uart_is_txfifo_full 和 uart_reg_write）。TryPutc 作为最外层接口，调用 uart_poll_send，用轮询的方式发送字符到串口，且转义换行符。

1.4.4 print.c 支持格式化输出

为实现与 C 语言中 printf 函数类似的格式化功能，我们要用到**可变参数列表** va_list。而真正实现格式化控制转换的函数是 vsnprintf_s 函数，该函数定义为 extern 即外部定义，在 vsnprintf_s.c 文件中。

TryPrintf 函数：这个函数尝试使用给定的格式字符串和可变参数列表利用 vsnprintf_s 函数来格式化一个字符串，并通过 TryPutc 函数将格式化的结果输出。

PRT_Printf 函数：这个函数是一个典型的可变参数函数的实现，它使用 va_list 和相关的宏来处理可变参数列表。

这两个函数共同工作，以提供一个完整的可变参数格式化输出功能。PRT_Printf 处理可变参数列表，并调用 TryPrintf 进行实际的格式化和输出。这种模式在 C 语言中很常见，因为它允许将复杂的逻辑（如格式化和输出）封装在一个内部函数中，同时提供一个简单的外部接口供用户使用。实现的是同一个功能，但实现了分离。

1.4.5 实现 vsnprintf_s 函数

vsnprintf_s 函数的主要作用是依据格式控制符将可变参数列表转换成字符列表写入缓冲区。具体实现不做说明。

1.5 修改 main、将新增文件加入构建系统并启用 FPU

修改 main.c，调用 PRT_Printf 函数输出信息。并将新增文件加入构建系统（修改

src/bsp/CMakeLists.txt 文件）。

FPU 是浮点数运算单元，如果不启用 FPU，可能会导致无法正确进行浮点数运算，进而无法产生相应的输出。

在 start.S 中修改代码：

```
20  Start:
21      LDR     x1, =__os_sys_sp_end // 符号在ld文件中定义
22      BIC     sp, x1, #0xf // 设置栈指针
23
24      B       OsEnterMain
```

```
1   Start:
2       LDR     x1, =__os_sys_sp_end // ld文件中定义，堆栈设置
3       BIC     sp, x1, #0xf
4
5   //参考： https://developer.arm.com/documentation/den0024/a/Memory-Order
6   Enable_FPU:
7       MRS X1, CPACR_EL1
8       ORR X1, X1, #(0x3 << 20)
9       MSR CPACR_EL1, X1
10      ISB
11
12      B       OsEnterMain
```

1.6 构建项目与执行结果

本实验逻辑为：

print.c 文件定义与 UART（通用异步收发传输器）通信相关的宏；实现串口的初始化；使用轮询的方式向串口发送字符实现输出（TryPutc 为最外层）；实现对 vsnprintf_s 函数的调用代码，以实现格式化输出。

vsnprintf_s.c 文件实现 vsnprintf_s 函数（PRT_Printf 为最外层）。

修改 main.c 函数调用 PRT_Printf 输出信息，新增文件加入构建系统，启用 FPU 并输出。

执行结果：

```
guoruiling@guoruiling-virtual-machine:~/lab2$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc
- skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab2/build
Scanning dependencies of target bsp
[ 16%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 33%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 50%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 66%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 66%] Built target bsp
[ 83%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
```

```
guoruiling@guoruiling-virtual-machine:~/lab2$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
------------ ------------    ------    -----------
 ************ ************   ********   ************
----         ------------  -----------  ----     ---
************     ****     ****    **** *********
 ------------     ----     ------------ ----------
       *****     ****     ************ ****  ****
 ------------     ----     ----    ---- ----   ----
************     ****     ****    **** ****    ****
Test PRT_Printf int format 10
```

2 实验任务

2.1 作业 1

题目：不启用 fifo，通过检测 UARTFR 寄存器的 TXFE 位来发送数据。

在串口的初始化部分，U32 PRT_UartInit(void)函数中，启用了 FIFO 来提高数据传输的效率和稳定性，防止数据丢失。

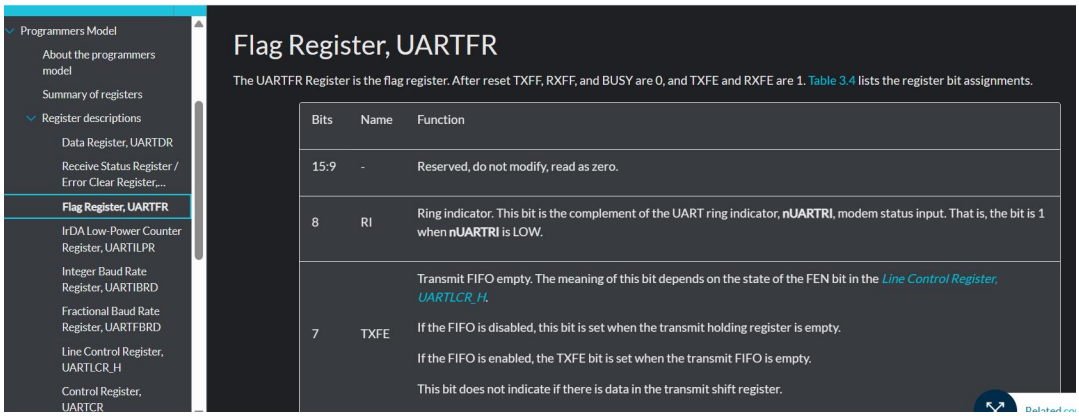
```
U32 PRT_UartInit(void)
{
    U32 result = 0;
    U32 reg_base = UART_0_REG_BASE;
    // LCR寄存器:  https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-desc
    result = UART_REG_READ((unsigned long)(reg_base + DW_UART_LCR_HR));
    UART_REG_WRITE(result | DW_FIFO_ENABLE, (unsigned long)(reg_base + DW_UART_LCR_HR)); // 启用 FIF

    return OS_OK;
}
```

对于是否启用 FIFO：

当 FIFO 被启用时，可以将多个字节的数据写入 FIFO，而不需要等待每个字节都被发送出去。UART 会按照 FIFO 中的顺序发送数据。只要 FIFO 未满（即 TXFF 未被置位），你就可以继续向 FIFO 中写入数据。

如果不使用 FIFO（或者 FIFO 的深度为 1），那么每次只能发送一个字节的数据。在发送完一个字节之前，不能写入下一个字节。这通常意味着 CPU 需要为每个字节的发送都进行干预，效率较低。

不使用 FIFO 时，可以通过直接监控 UART 的状态寄存器来确保在发送数据之前发送缓冲区是空的。UARTFR（UART FIFO 寄存器）包含 TXFE 位，用于指示发送 FIFO 是否为空。根据技术手册中 UARTFR 的介绍，我们可以知道 TXFE 为第 7 位。



原实验（使用 FIFO 方式）在实现向串口发送字符的时候，使用查看标志位 TXFF（掩码为 0x0010 0000，即第五位）进行缓存区是否满的判断，若 TXFF 为 1，返回 1。TXFF 标志位用于指示 UART 的发送 FIFO 是否为满，根据后续函数，该函数 uart_is_txfifo_full 返回 1 时，函数会自旋并计算时间，直到返回 0 时，也就是非满时，通过往数据寄存器写数据来发送字符。

5	TXFF	Transmit FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H Register. If the FIFO is disabled, this bit is set when the transmit holding register is full. If the FIFO is enabled, the TXFF bit is set when the transmit FIFO is full.

```
9   #define DW_XFIFO_NOT_FULL 0x020  // 发送缓冲区满置位
12  // 通过检查 FR 寄存器的标志位确定发送缓冲是否满，满时返回1.
13  S32 uart_is_txfifo_full(S32 uartno)
14  {
15      S32 ret;
16      U32 usr = 0;
17
18      ret = uart_reg_read(uartno, DW_UART_FR, &usr);
19      if (ret) {
20          return OS_OK;
21      }
22
23      return (usr & DW_XFIFO_NOT_FULL);
24  }
```

修改对应函数：增加一个掩码，禁用 FIFO，修改检查标志位函数，修改轮询发送函数。

```
9 #define DW_XFIFO_Empty 0x080  // 发送缓冲区kong置位
```

```
U32 PRT_UartInit(void)
{
    U32 result = 0;
    U32 reg_base = UART_0_REG_BASE;
    // LCR寄存器： https://developer.arm.com/documentatic
s/line-control-register--uartlcr-h?lang=en
   // result = UART_REG_READ((unsigned long)(reg_base + D
   // UART_REG_WRITE(result | DW_FIFO_ENABLE, (unsigned

    return OS_OK;
}
```

```
 7 // 通过检查 FR 寄存器的标志位确定发送缓冲是否kong，kong时返回1.
 8 S32 uart_is_txfifo_full(S32 uartno)
 9 {
10     S32 ret;
11     U32 usr = 0;
12
13     ret = uart_reg_read(uartno, DW_UART_FR, &usr);
14     if (ret) {
15         return OS_OK;
16     }
17
18     return (usr & DW_XFIFO_Empty);
19 }
```

```c
// 通过轮询的方式发送字符到串口
void uart_poll_send(unsigned char ch)
{

    S32 timeout = 0;
    S32 max_timeout = UART_BUSY_TIMEOUT;

    // 轮询发送缓冲区是否kong
    int result = uart_is_txfifo_full(0);
    while (!result) {
        timeout++;
        if (timeout >= max_timeout) {
            return;
        }
        result = uart_is_txfifo_full(0);
    }

    // 如果缓冲区kong，通过往数据寄存器写入数据发送字符到串口
    uart_reg_write(0, DW_UART_THR, (U32)(U8)ch);
    return;
}
```

编译，正常运行。

```
guoruiling@guoruiling-virtual-machine:~/lab2$ sh makeMiniEuler.sh
mkdir: cannot create directory 'build': File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab2/build
Scanning dependencies of target bsp
[ 16%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 33%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 50%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 66%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 66%] Built target bsp
[ 83%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
guoruiling@guoruiling-virtual-machine:~/lab2$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
Test PRT_Printf int format 10

QEMU: Terminated
```

2.2 作业 2

题目：采用 UniProton 提供的 libboundscheck 库实现 vsnprintf_s 函数。

UniProton 的 Gitee 链接：https://gitee.com/openeuler/libboundscheck

下载了 UniProton 的有关代码，尝试实现时发现有头文件未导入的问题，尝试将头文件相关部分代码导入和下载头文件修改，均未调试成功，还发现了嵌套头文件，要花很大精力将代码移植，故放弃。

```c
#include "secureprintoutput.h"

#if SECUREC_ENABLE_VSNPRINTF
/*
 * <FUNCTION DESCRIPTION>
 *    The vsnprintf_s function is equivalent to the vsnprintf function
 *     except for the parameter destMax/count and the explicit runtime-constraints violation
 *    The vsnprintf_s function takes a pointer to an argument list, then formats
 *    and writes up to count characters of the given data to the memory pointed
 *    to by strDest and appends a terminating null.
 *
```

```
int vsnprintf_s(char *strDest, size_t destMax, size_t count, const char *format, va_list argList)
{
    int retVal;

    if (SECUREC_VSNPRINTF_PARAM_ERROR(format, strDest, destMax, count, SECUREC_STRING_MAX_LEN)) {
        SECUREC_VSPRINTF_CLEAR_DEST(strDest, destMax, SECUREC_STRING_MAX_LEN);
        SECUREC_ERROR_INVALID_PARAMTER("vsnprintf_s");
        return -1;
    }

    if (destMax > count) {
        retVal = SecVsnprintfImpl(strDest, count + 1, format, argList);
        if (retVal == SECUREC_PRINTF_TRUNCATE) {  /* To keep dest buffer not destroyed 2014.2.18 */
            /* The string has been truncated, return  -1 */
            return -1;          /* To skip error handler,  return strlen(strDest) or -1 */
        }
    } else {
        retVal = SecVsnprintfImpl(strDest, destMax, format, argList);
#ifdef SECUREC_COMPATIBLE_WIN_FORMAT
```