

目录

- [目录](#)
- [第 3 章 编译与链接](#)
 - [练习 3-1](#)
 - [练习 3-2](#)
- [第 4 章 嵌入式开发介绍](#)
 - [练习 4-1](#)
 - [练习 4-2](#)
 - [练习 4-3](#)
- [第 5 章 RISC-V 汇编语言编程](#)
 - [练习 5-1](#)
 - [练习 5-2](#)
 - [练习 5-3](#)
 - [练习 5-4](#)
 - [练习 5-5](#)
 - [练习 5-6](#)
 - [练习 5-7](#)
 - [练习 5-8](#)
 - [练习 5-9](#)
- [第 7 章 Hello RVOS](#)
 - [练习 7-1](#)
 - [练习 7-2](#)
- [第 8 章 内存管理](#)
 - [练习 8-1](#)
- [第 9 章 上下文切换和协作式多任务](#)
 - [练习 9-1](#)
 - [练习 9-2](#)
- [第 11 章 外部设备中断](#)
 - [练习 11-1](#)
- [第 12 章 硬件定时器](#)
 - [练习 12-1](#)
- [第 13 章 抢占式多任务](#)
 - [练习 13-1](#)
- [第 14 章 任务同步和锁](#)
 - [练习 14-1](#)
- [第 15 章 软件定时器](#)
 - [练习 15-1](#)
 - [练习 15-2](#)
 - [练习 15-3](#)
- [第 16 章 系统调用](#)
 - [练习 16-1](#)
- [A 综合练习](#)
 - [练习 A-1](#)
 - [练习 A-2](#)

- [练习 A-3](#)

第 3 章 编译与链接

练习 3-1

使用 gcc 编译代码并使用 binutils 工具对生成的目标文件和可执行文件（ELF 格式）进行分析。具体要求如下：

- 编写一个简单的打印“hello world！”的程序源文件：[hello.c](#)
- 对源文件进行本地编译，生成针对支持 x86_64 指令集架构处理器的目标文件 [hello.o](#)。
- 查看 [hello.o](#) 的文件的文件头信息。
- 查看 [hello.o](#) 的 Section header table。
- 对 [hello.o](#) 反汇编，并查看 [hello.c](#) 的 C 程序源码和机器指令的对应关系。

练习 3-2

如下例子 C 语言代码：

```
#include <stdio.h>

int global_init = 0x11111111;
const int global_const = 0x22222222;

void main()
{
    static int static_var = 0x33333333;
    static int static_var_uninit;

    int auto_var = 0x44444444;

    printf("hello world!\n");
    return;
}
```

请问编译为 .o 文件后，[global_init](#), [global_const](#), [static_var](#), [static_var_uninit](#), [auto_var](#) 这些变量分别存放在那些 section 里，“[hello world!\n](#)”这个字符串又在哪里？并尝试用工具查看并验证你的猜测。

第 4 章 嵌入式开发介绍

练习 4-1

熟悉交叉编译概念，使用 riscv gcc 编译代码并使用 binutils 工具对生成的目标文件和可执行文件（ELF 格式）进行分析。具体要求如下：

- 编写一个简单的打印“hello world！”的程序源文件：[hello.c](#)
- 对源文件进行编译，生成针对支持 rv32ima 指令集架构处理器的目标文件 [hello.o](#)。

- 查看 `hello.o` 的文件的文件头信息。
- 查看 `hello.o` 的 Section header table。
- 对 `hello.o` 反汇编，并查看 `hello.c` 的 C 程序源码和机器指令的对应关系。

练习 4-2

基于 练习 4-1 继续熟悉 qemu/gdb 等工具的使用，具体要求如下：

- 将 `hello.c` 编译成可调试版本的可执行程序 `a.out`
- 先执行 `qemu-riscv32` 运行 `a.out`。
- 使用 `qemu-riscv32` 和 `gdb` 调试 `a.out`。

练习 4-3

自学 Makefile 的语法，理解在 `riscv-operating-system-mooc` 仓库的根目录下执行 `make` 会发生什么。

第 5 章 RISC-V 汇编语言编程

练习 5-1

- 对 `code/asm/sub` 执行反汇编，查看 `sub x5, x6, x7` 这条汇编指令对应的机器指令的编码，并对照 RISC-V 的 specifaion 自己解析该条指令的编码。
- 现知道某条 RISC-V 的机器指令在内存中的值为 `b3 05 95 00`，从左往右为从低地址到高地址，单位为字节，请将其翻译为对应的汇编指令。

练习 5-2

假设有如下这么一段 C 语言程序代码，尝试编写一段汇编代码，达到等效的结果，并采用 `gdb` 调试查看执行结果，注意请使用寄存器来存放变量的值：

```
register int a, b, c, d, e;
b = 1;
c = 2;
e = 3;
a = b + c;
d = a - e;
```

练习 5-3

假设有如下这么一段 C 语言程序代码，尝试编写一段汇编代码，达到等效的结果，并采用 `gdb` 调试查看执行结果，注意请使用寄存器来存放变量的值：

```
register int a, b, c, d, e;
b = 1;
c = 2;
d = 3;
```

```
e = 4;
a = (b + c) - (d + e);
```

练习 5-4

给定一个 32 位数 `0x87654321`，先编写 c 程序，将其低 16 位 (`0x4321`) 和高 16 位 (`0x8765`) 分别分离出来保存到独立的变量中；完成后再尝试采用汇编语言实现类似的效果。

练习 5-5

假设有如下这么一段 C 代码：

```
int array[2] = {0x11111111, 0xffffffff};
int i = array[0];
int j = array[1];
```

尝试用汇编语句实现等效的功能。

练习 5-6

在内存中定义一个结构体变量，编写汇编程序，用宏方式 (`.macro/.endm`) 实现对结构体变量的成员赋值以及读取该结构体变量的成员的值到寄存器变量中。等价的 c 语言的示例代码如下，供参考：

```
struct S {
    unsigned int a;
    unsigned int b;
};

struct S s = {0};

#define set_struct(s) \
    s.a = a; \
    s.b = b;

#define get_struct(s) \
    a = s.a; \
    b = s.b;

void foo()
{
    register unsigned int a = 0x12345678;
    register unsigned int b = 0x87654321;
    set_struct(s);
    a = 0;
    b = 0;
    get_struct(s);
}
```

练习 5-7

编写汇编指令，使用条件分支指令循环遍历一个字符串数组，获取该字符串的长度。等价的 c 语言的示例代码如下，供参考：

```
char array[] = {'h', 'e', 'l', 'l', 'o', ',', 'w', 'o', 'r', 'l', 'd', '!', '\0'};
int len = 0;
while (array[len] != '\0') {
    len++;
}
```

练习 5-8

要求：阅读 [code/asm/cc_leaf](#) 和 [code/asm/cc_nested](#) 的例子代码，理解 RISC-V 的函数调用约定。在此基础上编写汇编程序实现以下功能，等价的 c 语言的示例代码如下，供参考：

```
unsigned int square(unsigned int i)
{
    return i * i;
}

unsigned int sum_squares(unsigned int i)
{
    unsigned int sum = 0;
    for (int j = 1; j <= i; j++) {
        sum += square(j);
    }
    return sum;
}

void _start()
{
    sum_squares(3);
}
```

练习 5-9

在 C 函数中嵌入一段汇编，实现 `foo` 函数中的 `c = a * a + b * b`；这句 c 语言的同等功能。

```
int foo(int a, int b)
{
    int c;
    c = a * a + b * b;
    return c;
}
```

第 7 章 Hello RVOS

练习 7-1

要求：完全采用汇编语言重写 `code/os/01-helloRVOS`，运行后在控制台上打印输出 "Hello, RVOS!"

练习 7-2

要求：参考 `code/os/01-helloRVOS`，在此基础上增加采用轮询方式读取控制台上输入的字符并 **回显** 在控制台上。另外用户按下回车后能够另起一行从头开始。

第 8 章 内存管理

练习 8-1

要求：参考 `code/os/02-memmanagement`，在 `page` 分配的基础上实现更细颗粒度的，精确到字节为单位的内存管理。要求实现如下接口，具体描述参考 `man(3) malloc`：

```
void *malloc(size_t size);
void free(void *ptr);
```

第 9 章 上下文切换和协作式多任务

练习 9-1

要求：参考 `code/os/04-multitask`，在此基础上进一步改进任务管理功能。具体要求：

- 改进 `task_create()`，提供更多的参数，具体改进后的函数如下所示：

```
int task_create(void (*task)(void* param),
               void *param,
               uint8_t priority);
```

其中：

- `param` 用于在创建任务执行函数时可带入参数，如果没有参数则传入 `NULL`。
- `priority` 用于指定任务的优先级，目前要求最多支持 256 级，0 最高，依次类推。同时修改任务调度算法，在原先简单轮转的基础上支持按照优先级排序，优先选择优先级高的任务运行，同一级多个任务再轮转。
- 增加任务退出接口 `task_exit()`，当前任务可以通过调用该接口退出执行，内核负责将该任务回收，并调度下一个可运行任务。建议的接口函数如下：

```
void task_exit(void);
```

练习 9-2

目前 `code/os/04-multitask` 实现的任务调度中，前一个用户任务直接调用 `task_yield()` 函数并最终调用 `switch_to()` 切换到下一个用户任务。`task_yield()` 作为内核路径借用了用户任务的栈，当用户任务的函数调用层次过多或者 `task_yield()` 本身函数内部继续调用函数，可能会导致用户任务的栈空间溢出。参考 "`mini-riscv-os`" 的 `03-MultiTasking` 的实现，为内核调度单独实现一个任务，在任务切换中，前一个用户任务首先切换到内核调度任务，然后再由内核调度任务切换到下一个用户任务，这样就可以避免前面提到的问题了。

要求：参考以上设计，并尝试实现之。

第 11 章 外部设备中断

练习 11-1

要求：采用中断方式实现对 UART 的写操作。

第 12 章 硬件定时器

练习 12-1

要求：参考 `code/os/07-hwtimer`，利用系统提供的 TICK 机制实现一个数字时钟，通过串口显示在控制台上，随着 TICK 的增加以墙上时钟的效果显示当前时间，显示效果如下：

初始状态：

```
$ ./a.out
Press Ctrl-A and then X to exit QEMU
-----
Hello, RVOS!
00:00:00
```

一秒钟后：

```
$ ./a.out
Press Ctrl-A and then X to exit QEMU
-----
Hello, RVOS!
00:00:01
```

两秒钟后：

```
$ ./a.out
Press Ctrl-A and then X to exit QEMU
-----
Hello, RVOS!
00:00:02
```

.....

一分钟后：

```
$ ./a.out
Press Ctrl-A and then X to exit QEMU
-----
Hello, RVOS!
00:01:00
```

.....

第 13 章 抢占式多任务

练习 13-1

要求：在 [练习 9-1](#) 的基础上进一步改进任务管理功能，增加任务优先级的管理。具体要求：改进 `task_create()`，增加时间片 (`timeslice`) 参数，具体改进后的函数如下所示：

```
int task_create(void (*task)(void* param),
               void *param,
               uint8_t priority,
               uint32_t timeslice);
```

其中：

- 其他参数含义不变，见 [练习 9-1](#) 的描述。
- `timeslice`：任务的时间片大小，单位是操作系统的时钟节拍 (`tick`)，此参数指定该任务一次调度可以运行的最大时间长度。和 `priority` 相结合，调度器会首先根据 `priority` 选择优先级最高的任务运行，而 `timeslice` 则决定了当没有更高优先级的任务时，当前正在运行的任务可以运行的最大时间长度。

第 14 章 任务同步和锁

练习 14-1

阅读 [code/os/ch14-lock](#) 的代码例子，深入体会自旋锁的实现和使用，检查代码看看内核中有哪些部分会产生并发问题，是否可以采用自旋锁进行保护并实验之。

第 15 章 软件定时器

练习 15-1

参考例子 [code/os/10-swtimer](#)，自己尝试实现一个软件单次触发定时器。要求：

- 参考下面的代码实现标准的创建定时器和删除定时器接口

```
/* software timer */
struct timer {
    void (*func)(void *arg);
    void *arg;
    uint32_t timeout_tick;
};

struct timer *timer_create(
    void (*handler)(void *arg),
    void *arg,
    uint32_t timeout);

void timer_delete(struct timer *timer);
```

- 软件定时器对象列表的组织先采用简单的数组管理方式。
- 数组方式实验完成后再尝试采用链表的管理方式。为加快超时对象的搜索，要求软件定时器对象在链表中按照超时的前后顺序排序，即实现一个有序的链表。

练习 15-2

基于 [练习 15-1](#) 的有序链表实现进行进一步优化，尝试采用 **跳表 (Skip List) 算法** 进一步加快链表搜索的速度。

提示: 自行搜索并学习 **跳表 (Skip List) 算法**，设置不同的层数，比较搜索的效率。

练习 15-3

基于软件定时器重现实现 `task_delay()`。原来的 `task_delay()` 的实现十分粗糙，只是采用简单的循环来达到延时的目的，见下面的代码实现：

```
void task_delay(volatile int count)
{
    count *= 50000;
    while (count--);
}
```

我们希望利用调度机制，结合定时器，实现真正意义上的任务延迟（睡眠）。具体来说就是当某个任务调用 `task_delay()` 后，内核将该任务移出调度的队列（目前的示例代码中可以认为内核只维护了一个任务调度队

列)，并根据任务指定的 `delay` 时长对该任务设置一个定时器，定时器到期后内核再将该任务移入调度的队列，并根据当前状态重新进行调度。建议实现后新的 `task_delay()` 接口定义如下：

```
int task_delay(uint32_t tick);
```

其中 `tick` 是延迟（睡眠）的时间长度，单位是系统的 TICK。

第 16 章 系统调用

练习 16-1

阅读 `code/os/11-syscall` 的代码例子，如果我们想要对整个系统划分 User 和 Machine 两层的话，看看哪些接口函数需要采用系统调用方式来实现并实验之。

A 综合练习

练习 A-1

熟悉 rvos 的源码，阅读 RISC-V 的 spec 手册将 rvos 移植到 64 位的 qemu (qemu-system-riscv64) 上去。要求至少实现 `code/os/08-preemptive` 程度的功能。

练习 A-2

阅读 RT-Thread 的 [源码](#) 以及其 [官网学习资料](#)，了解 RT-Thread 的内核设计，主要包括：

- 启动流程
- 中断管理
- 线程管理
- 时钟管理
- 线程间同步

在理解的基础上对以上内核模块写出自己的分析报告，有条件的话尝试结合一款 riscv 的开发板进行实验。RT-Thread 支持的 RISC-V 开发板参考代码目录：https://gitee.com/rththread/rthread/tree/gitee_master/libcpu/risc-v。

练习 A-3

尝试将 RT-Thread 移植到 qemu virt 上（32 位或者 64 位皆可）。