
内核实验手册

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 @2019



WWW.RT-THREAD.ORG

Wednesday 13th October, 2021

版本和修订

Date	Version	Author	Note
2018-12-29	v1.0.0	yangjie	初始版本

目录

版本和修订	i
目录	ii
1 RT-Thread 实验环境搭建	1
1.1 目的	1
1.2 MDK5 安装	1
1.3 运行仿真	5
1.4 FinSH 命令行中启动线程	7
1.5 SystemView 工具介绍	9
2 实验：线程的使用	10
2.1 实验目的	10
2.2 实验原理及程序结构	10
2.2.1 实验设计	10
2.2.2 源程序说明	11
2.2.2.1 示例代码框架	11
2.2.2.2 示例源码	11
2.3 编译、仿真运行和观察示例应用输出	13
2.4 附件	15
3 实验：线程的时间片轮转调度	16
3.1 实验目的	16
3.2 实验原理及程序结构	16
3.2.1 实验设计	16
3.2.2 源程序说明	17
3.2.2.1 RT-Thread 示例代码框架	17

3.2.2.2 示例源码	17
3.3 编译、仿真运行和观察示例应用输出	19
3.4 附件	20
4 实验：定时器的使用	21
4.1 实验目的	21
4.2 实验原理及程序结构	21
4.2.1 实验设计	21
4.2.2 源程序说明	22
4.2.2.1 RT-Thread 示例代码框架	22
4.2.2.2 示例源码	22
4.3 编译、仿真运行和观察示例应用输出	24
4.4 附件	25
5 实验：信号量——生产者消费者问题	26
5.1 实验目的	26
5.2 实验原理及程序结构	26
5.2.1 实验设计	26
5.2.2 源程序说明	28
5.2.2.1 RT-Thread 示例代码框架	28
5.2.2.2 示例源码	28
5.3 编译、仿真运行和观察示例应用输出	31
5.4 附件	32
6 实验：互斥量——优先级继承	33
6.1 实验目的	33
6.2 实验原理及程序结构	33
6.2.1 实验设计	33
6.2.2 源程序说明	34
6.2.2.1 RT-Thread 示例代码框架	34
6.2.2.2 示例源码	34
6.3 编译、仿真运行和观察示例应用输出	37
6.4 附件	40

7 实验：事件集的使用	41
7.1 实验目的	41
7.2 实验原理及程序结构	41
7.2.1 实验设计	41
7.2.2 源程序说明	42
7.2.2.1 RT-Thread 示例代码框架	42
7.2.2.2 示例源码	42
7.3 编译、仿真运行和观察示例应用输出	45
7.4 附件	46
8 实验：邮箱的使用	47
8.1 实验目的	47
8.2 实验原理及程序结构	47
8.2.1 实验设计	47
8.2.2 源程序说明	48
8.2.2.1 RT-Thread 示例代码框架	48
8.2.2.2 示例源码	48
8.3 编译、仿真运行和观察示例应用输出	51
8.4 附件	53
9 实验：消息队列的使用	54
9.1 实验目的	54
9.2 实验原理及程序结构	54
9.2.1 实验设计	54
9.2.2 源程序说明	55
9.2.2.1 示例代码框架	55
9.2.2.2 示例源码	55
9.3 编译、仿真运行和观察示例应用输出	58
9.4 附件	60
10 实验：动态内存堆的使用	61
10.1 实验目的	61
10.2 实验原理及程序结构	61
10.2.1 实验设计	61
10.3 源程序说明	61

10.3.1 示例代码框架	61
10.3.2 示例源码	62
10.4 编译、运行和观察示例应用输出	63

第 1 章

RT-Thread 实验环境搭建

1.1 目的

- 本章的目的是让初学者了解 RT-Thread 运行环境，将以 MDK5 为例，搭建 RT-Thread 运行环境。

1.2 MDK5 安装

已经安装 MDK5 的可以直接略过此步骤。

在运行 RT-Thread 操作系统前，我们需要安装 MDK-ARM 5.24（正式版或评估版，5.14 版本及以上版本均可），这个版本也是当前比较新的版本，它能够提供更完善的调试功能。这里采用了 16K 编译代码限制的评估版 5.24 版本，如果要解除 16K 编译代码限制，请购买 MDK-ARM 正式版。先从 www.keil.com 官方网站下载 MDK-ARM 评估版：<http://www.keil.com/download/>。

在下载时，需要填一些个人基本信息，请填写相应的完整信息，然后开始下载。下载完成后，鼠标双击运行，会出现如下图所示的软件安装画面：



图 1.1: MDK 安装图 1

步骤 1 这是 MDK-ARM 的安装说明，点击“Next>>”进入下一画面，如下图所示：



图 1.2: MDK 安装图 2

步骤 2 在“I agree to all the terms of the preceding License Agreement”前的选择框中点击选择“☒”，并点击“Next >>”进入下一步安装，如下图所示：

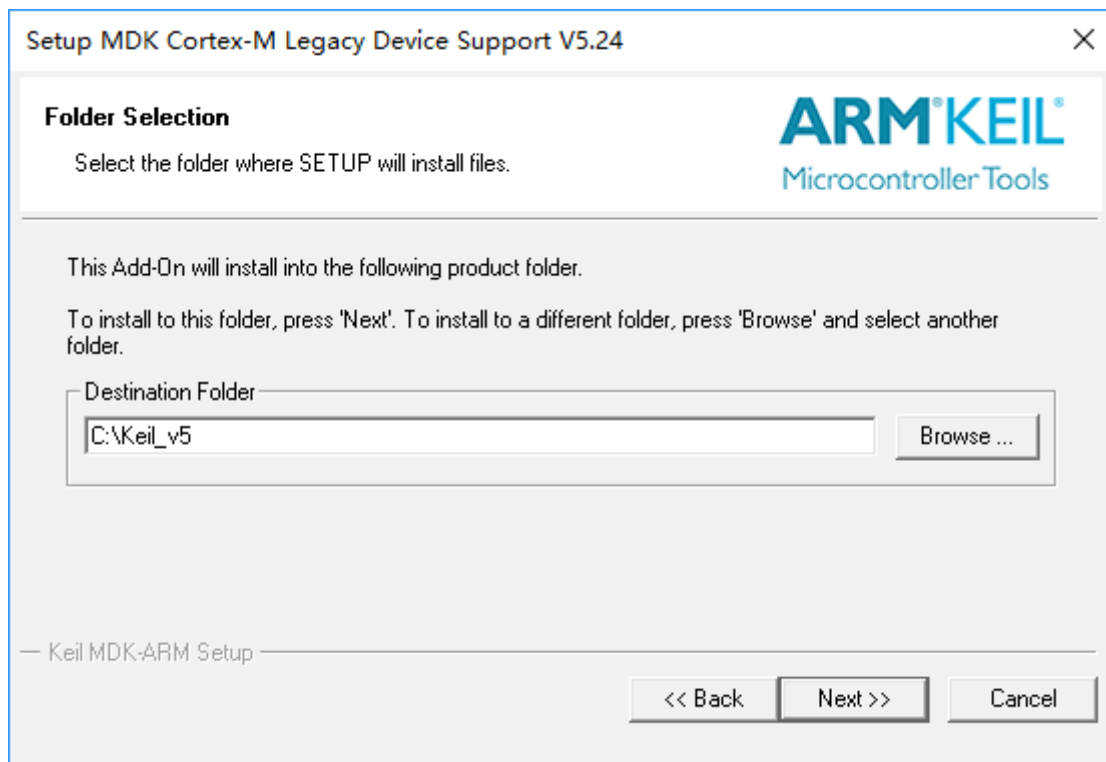


图 1.3: MDK 安装图 3

步骤 3 点击“Browse...”选择 MDK-ARM 的安装目录或者直接在“Destination Folder”下的文本框中输入安装路径，这里我们默认“C:/Keil”即可，然后点击“Next>>”进入下一步安装，如下图所示：

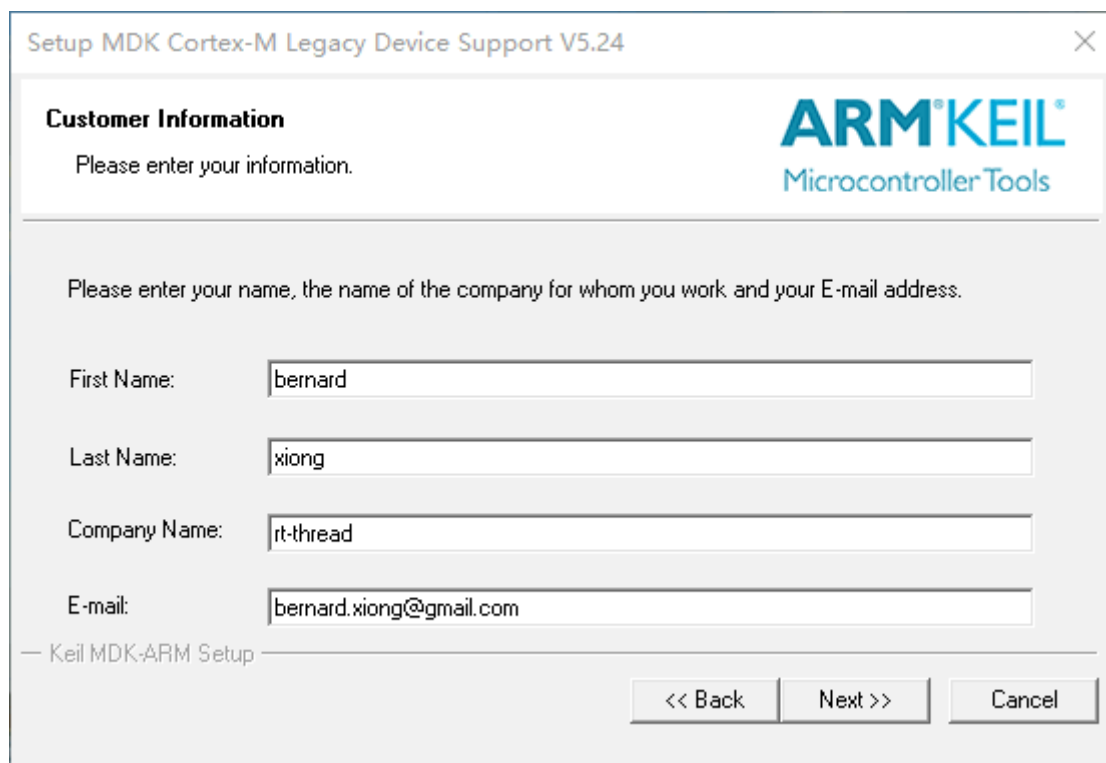


图 1.4: MDK 安装图 4

步骤 4 在“First Name”后输入您的名字，“Last Name”后输入您的姓，“Company Name”后输入您

的公司名称，“E-mail”后输入您的邮箱地址，然后点击“Next>>”进行安装，等待一段时间后，安装结束，出现如下图所示画面：

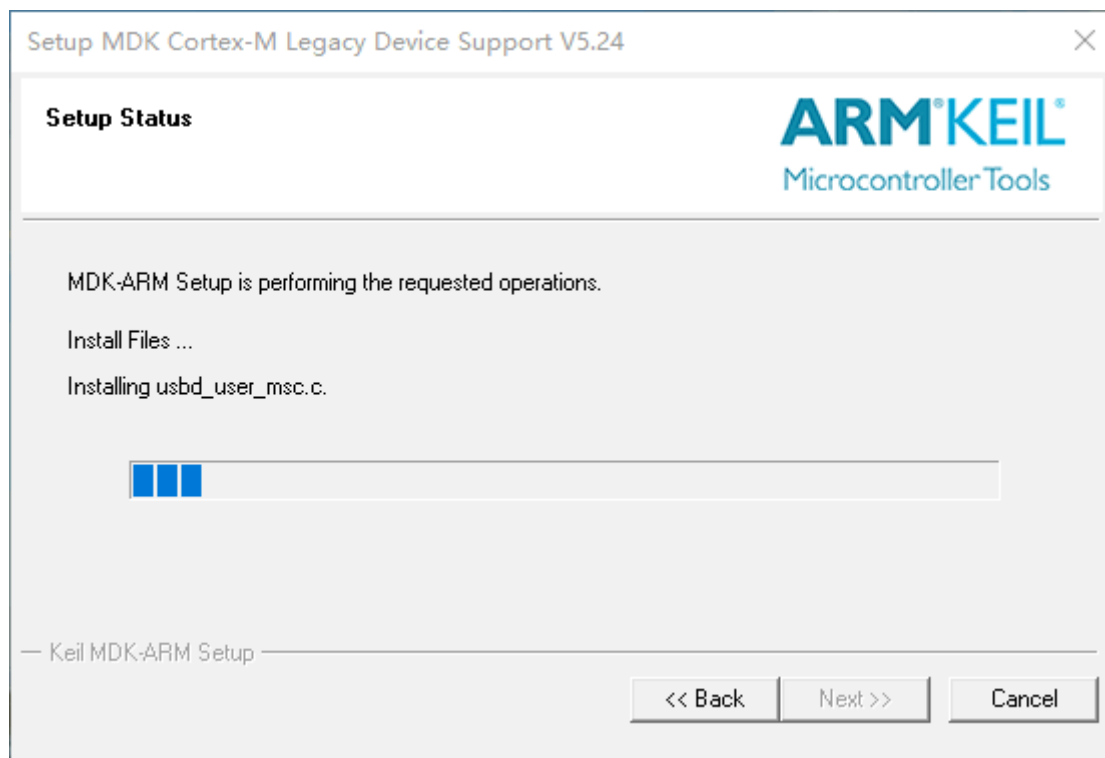


图 1.5: MDK 安装图 5

步骤 5 图中的默认选择不需改动，直接点击“Next”进入如下图所示画面：

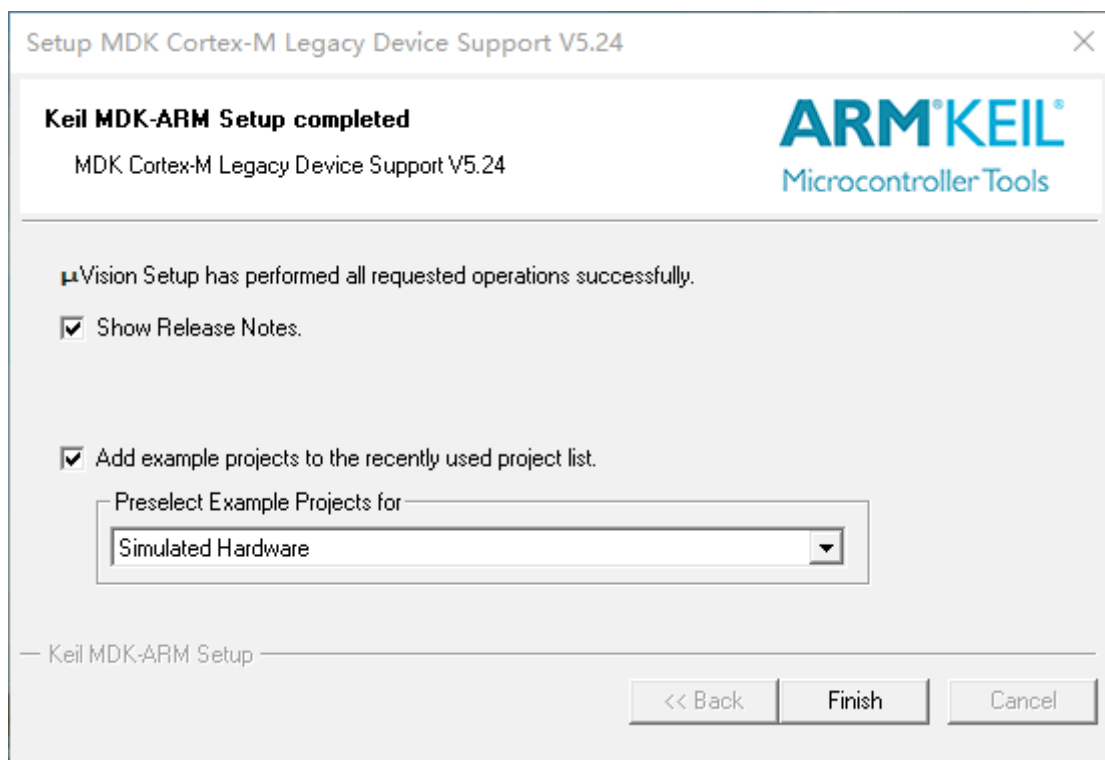


图 1.6: MDK 安装图 6

步骤 6 在这里可以点击“Finish”完成整个 MDK-ARM 软件的安装。

有了 MDK-ARM 利器，就可以轻松开始 RT-Thread 操作系统之旅，一起探索实时操作系统的奥秘。

注：MDK-ARM 正式版是收费的，如果您希望能够编译出更大体积的二进制文件，请购买 MDK-ARM 正式版。RT-Thread 操作系统也支持自由软件基金会的 GNU GCC 编译器，这是一款开源的编译器，想要了解如何使用 GNU 的相关工具请参考 RT-Thread 网站上的相关文档。

1.3 运行仿真

打开配合本实验的代码工程 [RT-Thread Simulator 例程](#)，例程源码中包含：RT-Thread 内核、FinSH 控制台、串口驱动、GPIO 驱动这些内容，支持 STM32F10X 系列 MCU，源码的目录结构如下图所示：

名称	修改日期	类型	大小
 applications	2018/08/28 15:19	文件夹	
 drivers	2018/08/28 15:19	文件夹	
 Libraries	2018/08/28 15:19	文件夹	
 packages	2018/08/28 15:18	文件夹	
 rt-thread	2018/08/28 15:19	文件夹	
 project.uvprojx	2018/08/27 18:00	Microvision5 Project	39 KB
 rtconfig.h	2018/08/21 18:40	H 文件	2 KB

图 1.7: 源码的目录结构

在目录下，有一个 project.uvprojx 文件，它是本文内容所引述的例程中的一个 MDK5 工程文件，双击“project.uvprojx”图标，打开此工程文件，如下图所示：

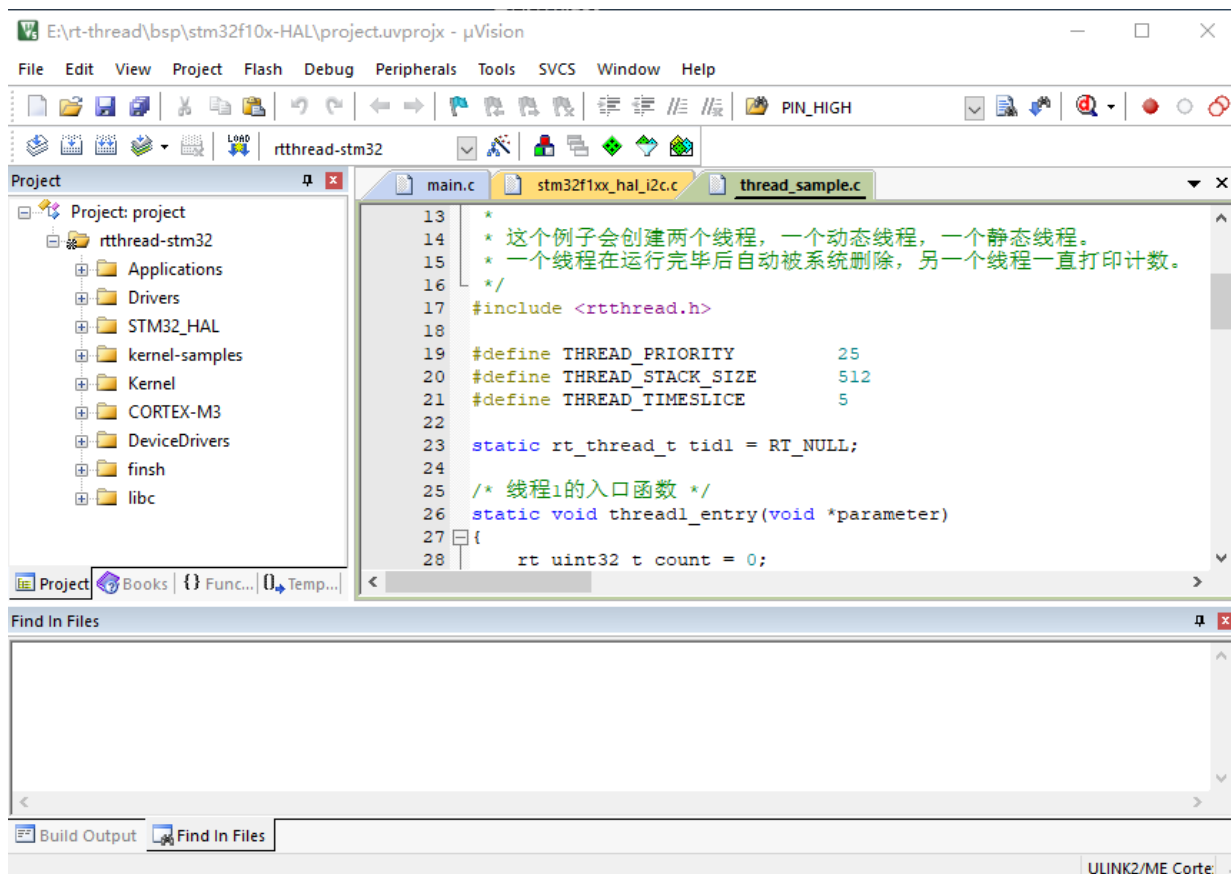



图 1.8: 工程文件

现在我们点击一下窗口上方工具栏中的按钮 ，对该工程进行编译，如下图所示：

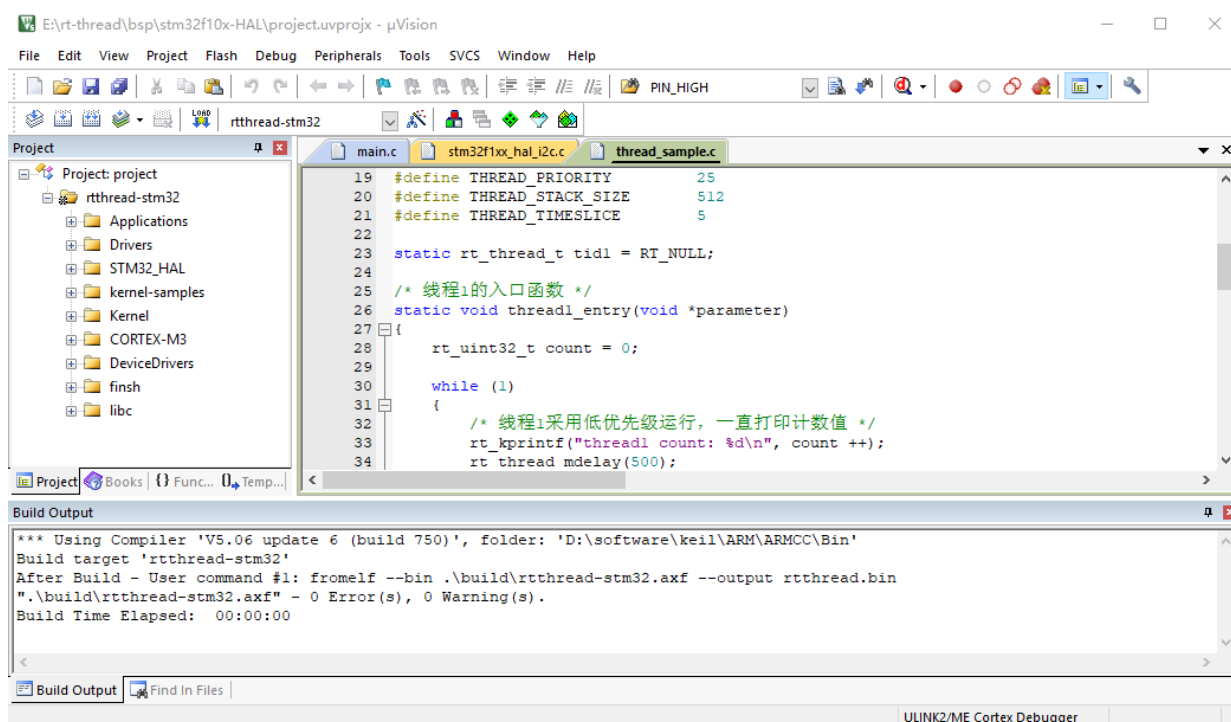


图 1.9: 编译结果

编译的结果显示在窗口下方的“Build”栏中，没什么意外的话，最后一行会显示“0 Error(s), *Warning(s).”，即无任何错误和警告。

在编译完 RT-Thread/STM32 后，我们可以通过 MDK-ARM 的模拟器来仿真运行 RT-Thread:

- 点击下图中的按钮 1 或直接按“Ctrl+F5”进入仿真界面。
- 点击下图中的按钮 2 或直接按“F5”开始仿真。
- 点击下图中的按钮 3 或者选择菜单栏中的“View □ Serial Windows □ UART#1”，打开串口 1 窗口。



图 1.10: button

可以看到串口输出了 RT-Thread 的 LOGO，其模拟运行的结果如下图所示:

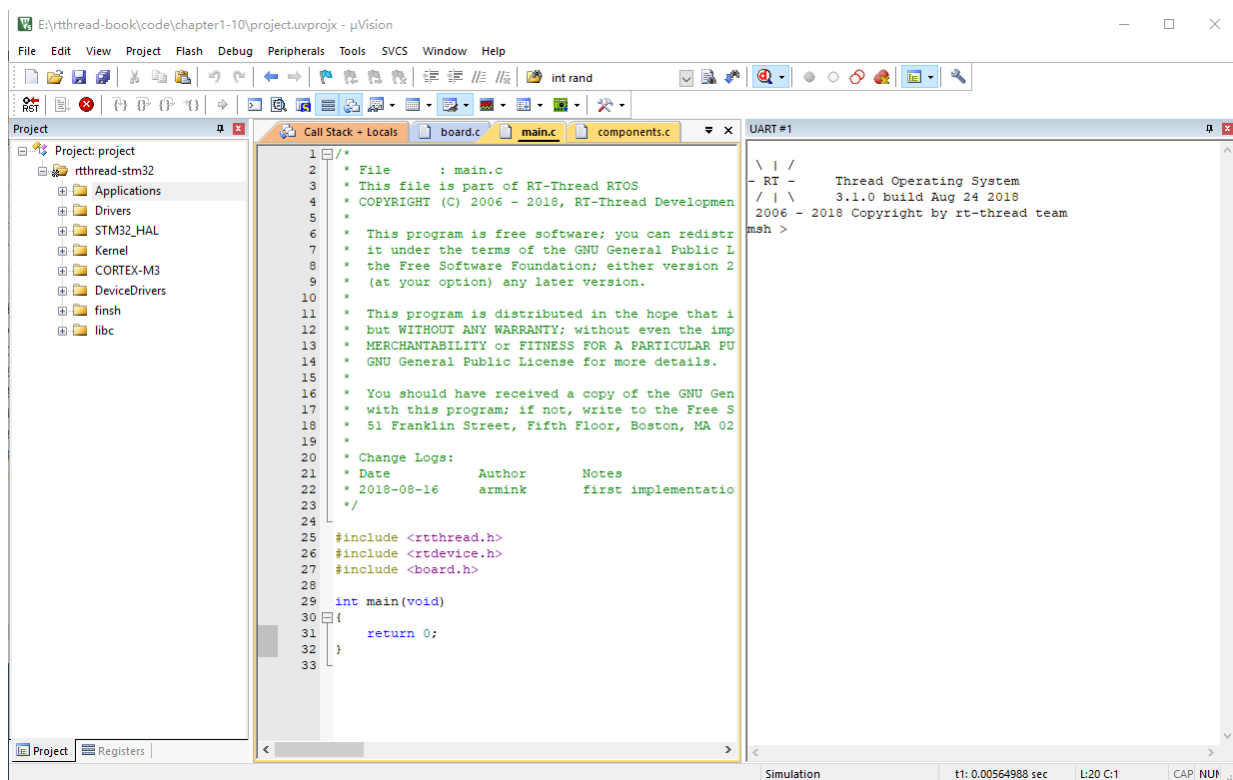


图 1.11: 模拟运行的结果图

1.4 FinSH 命令行中启动线程

RT-Thread 提供 FinSH 功能，用于调试或查看系统信息，msh 表示 FinSH 处于一种传统命令行模式，此模式下可以使用类似于 dos/bash 等传统的 shell 命令。

比如，我们可以通过输入“help + 回车”或者直接按下 Tab 键，输出当前系统所支持的所有命令，如下:

```

msh >help
RT-Thread shell commands:
thread_sample      - thread sample
timer_sample       - timer sample
semaphore_sample   - semaphore sample
mutex_sample       - mutex sample
event_sample       - event sample
mailbox_sample     - mailbox sample
msgq_sample        - msgq sample
signal_sample      - signal sample
mempool_sample     - mempool sample
dynmem_sample      - dynmem sample
interrupt_sample   - interrupt sample
idle_hook_sample   - idle hook sample
producer_consumer  - producer_consumer sample
timeslice_sample   - timeslice sample
scheduler_hook     - scheduler_hook sample
prio_inversion     - prio_inversion sample
version            - show RT-Thread version information
list_thread        - list thread
list_sem           - list semaphore in system
list_event         - list event in system
list_mutex         - list mutex in system
list_mailbox       - list mail box in system
list_msgqueue      - list message queue in system
list_memheap       - list memory heap in system
list_mempool       - list memory pool in system
list_timer         - list timer in system
list_device        - list device in system
help              - RT-Thread shell help.
ps                - List threads in the system.
time              - Execute command with time.
free              - Show the memory usage in the system.

msh >

```

此时可以输入列表中的命令，如输入 `list_thread` 命令显示系统当前正在运行的线程，结果显示为 `tshell`（shell 线程）线程与 `tidle`（空闲线程）线程：

```

msh >list_thread
thread pri status sp stack size max used left tick error
-----
tshell 20 ready 0x00000080 0x00001000 07% 0x0000000a 000
tidle 31 ready 0x00000054 0x00000100 32% 0x00000016 000
msh >

```

FinSH 具有命令自动补全功能，输入命令的部分字符（前几个字母，注意区分大小写），按下 `Tab` 键，则系统会根据当前已输入的字符，从系统中查找已经注册好的相关命令，如果找到与输入相关的命令，则

会将完整的命令显示在终端上。

如：要使用 `version` 命令，可以先输入“`v`”，再按下 `Tab` 键，可以发现系统会在下方补全了有关“`v`”开头的命令：`version`，此时只需要回车，即可查看该命令的执行结果。

每一个实验都会导出一个命令，做某个实验时，键入该实验对应的命令并回车，就会对该实验开始仿真。复位程序可以点击“`RST`”按钮，退出仿真需要再次点击仿真按钮。

1.5 SystemView 工具介绍

SystemView 是一个可以在线调试嵌入式系统的工具，它可以分析有哪些中断、任务执行了，以及这些中断、任务执行的先后关系。还可以查看一些内核对象持有和释放的时间点，比如信号量、互斥量、事件、消息队列等，这在开发和处理具有多个线程和事件的复杂系统时尤其有效，能帮助用户进行系统调试和分析、显著缩短开发和调试时间，提高开发效率。

本实验采用 **SystemView** 对系统执行的线程及其状态进行可视化监控分析，通过该工具将实验的运行状态与时间的关系保存下来，大家可以下载安装该工具，打开实验附带的附件，了解实验过程及细节。

下载 **SystemView** 分析工具：<https://www.segger.com/products/development-tools/systemview/>

第 2 章

实验：线程的使用

2.1 实验目的

- 理解线程创建、初始化与自动脱离的基本原理；
- 理解高优先级线程抢占低优先级线程运行；
- 掌握 RT-Thread 中线程的动态创建、静态初始化；
- 在 RT-Thread 中熟练使用线程来完成需求。

2.2 实验原理及程序结构

线程，即任务的载体。一般被设计成 **while(1)** 的循环模式，但在循环中一定要有让出 CPU 使用权的动作。如果是可以执行完毕的线程，则系统会自动将执行完毕的线程进行删除 / 脱离。

2.2.1 实验设计

本实验使用的例程为：[thread_sample.c](#)

为了体现线程的创建、初始化与脱离，本实验设计了 **thread1**、**thread2** 两个线程。**thread1** 是创建的动态线程，优先级为 25；**Thread2** 初始化的静态线程，优先级为 24。

优先级较高的 **Thread2** 抢占低优先级的 **thread1**，执行完毕一段程序后自动被系统脱离。

优先级较低的 **thread1** 被设计成死循环，循环中有让出 CPU 使用权的动作 – 使用了 **delay** 函数。该线程在 **thread2** 退出运行之后开始运行，每隔一段时间运行一次，并一直循环运行下去。

通过本实验，用户可以清晰地了解到线程在本实验中的状态变迁情况。

整个实验运行过程如下图所示，描述如下：

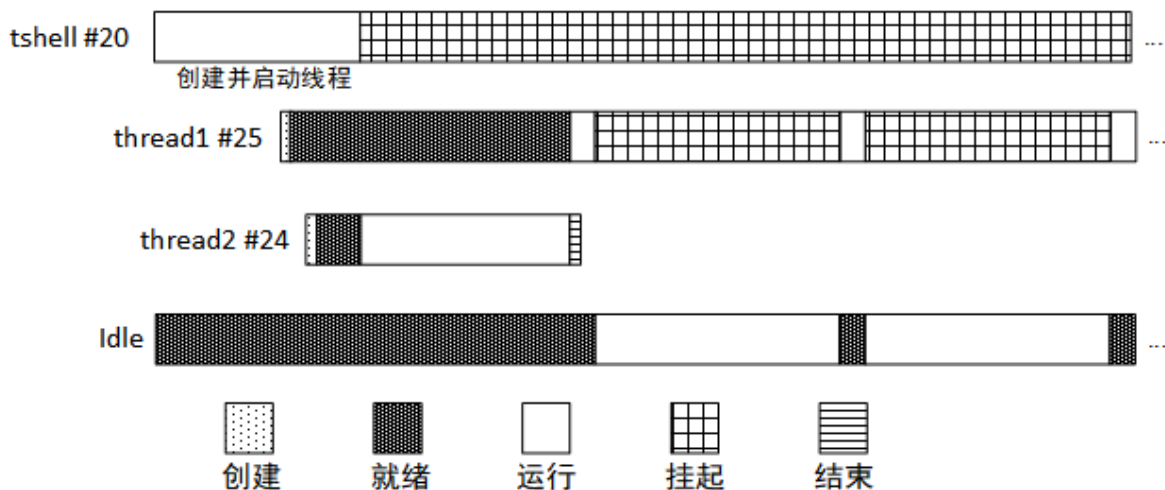


图 2.1: 实验运行过程

- (1) 在 `tshell` 线程 (优先级 20) 中创建线程 `thread1` 和初始化 `thread2`, `thread1` 优先级为 25, `thread2` 优先级为 24;
- (2) 启动线程 `thread1` 和 `thread2`, 使 `thread1` 和 `thread2` 处于就绪状态;
- (3) 随后 `tshell` 线程挂起, 在操作系统的调度下, 优先级较高的 `thread2` 首先被投入运行;
- (4) `thread2` 是可执行完毕线程, 运行完毕打印之后, 系统自动删除 `thread2`;
- (5) `thread1` 得以运行, 打印信息之后执行延时将自己挂起;
- (6) 系统中没有优先级更高的就绪队列, 开始执行空闲线程;
- (7) 延时时间到, 执行 `thread1`;
- (8) 循环 (5) ~ (7)。

2.2.2 源程序说明

2.2.2.1 示例代码框架

RT-Thread 示例代码都通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

2.2.2.2 示例源码

定义了待创建线程需要用到的优先级，栈空间，时间片的宏，定义线程 `thread1` 的线程句柄：

```
# include <rtthread.h>
# define THREAD_PRIORITY 25
# define THREAD_STACK_SIZE 512
# define THREAD_TIMESLICE 5
static rt_thread_t tid1 = RT_NULL;
```

线程 thread1 入口函数，每 500ms 打印一次计数值

```

/* 线程 1 的入口函数 */
static void thread1_entry(void *parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 线程 1 采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread1 count: %d\n", count ++);
        rt_thread_mdelay(500);
    }
}

```

线程 **thread2** 线程栈、控制块以及线程 2 入口函数的定义，线程 2 打印计数，10 次后退出。

```

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_entry(void *param)
{
    rt_uint32_t count = 0;

    /* 线程 2 拥有较高的优先级，以抢占线程 1 而获得执行 */
    for (count = 0; count < 10 ; count++)
    {
        /* 线程 2 打印计数值 */
        rt_kprintf("thread2 count: %d\n", count);
    }
    rt_kprintf("thread2 exit\n");

    /* 线程 2 运行结束后也将自动被系统脱离 */
}

```

例程代码，其中创建了线程 **thread1**，初始化了线程 **thread2**，并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

/* 线程示例 */
int thread_sample(void)
{
    /* 创建线程 1，名称是 thread1，入口是 thread1_entry*/
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);

    /* 如果获得线程控制块，启动这个线程 */
    if (tid1 != RT_NULL)

```

```
    rt_thread_startup(tid1);

    /* 初始化线程 2，名称是 thread2，入口是 thread2_entry */
    rt_thread_init(&thread2,
                  "thread2",
                  thread2_entry,
                  RT_NULL,
                  &thread2_stack[0],
                  sizeof(thread2_stack),
                  THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(thread_sample, thread sample);
```

2.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 `thread_sample` 命令启动示例应用，示例输出结果如下：

```
\ | /
- RT -   Thread Operating System
/ | \    3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >thread_sample
msh >thread2 count: 0
thread2 count: 1
thread2 count: 2
thread2 count: 3
thread2 count: 4
thread2 count: 5
thread2 count: 6
thread2 count: 7
thread2 count: 8
thread2 count: 9
thread2 exit
thread1 count: 0
thread1 count: 1
thread1 count: 2
thread1 count: 3
...
```

使用 **SystemView** 工具可以监测示例实际运行过程，如下三图所示，可以看到实验的实际运行流程与实验设计的流程一致，`thread2` 运行一段时间结束，`thread1` 每隔一段时间运行一次，并一直循环运行下

去。

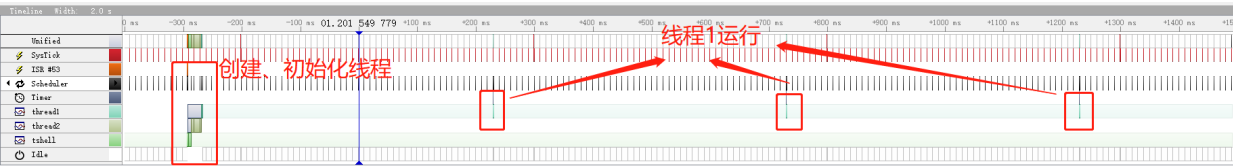


图 2.2: 实验总过程

将创建、初始化的部分放大看，如下两张图：

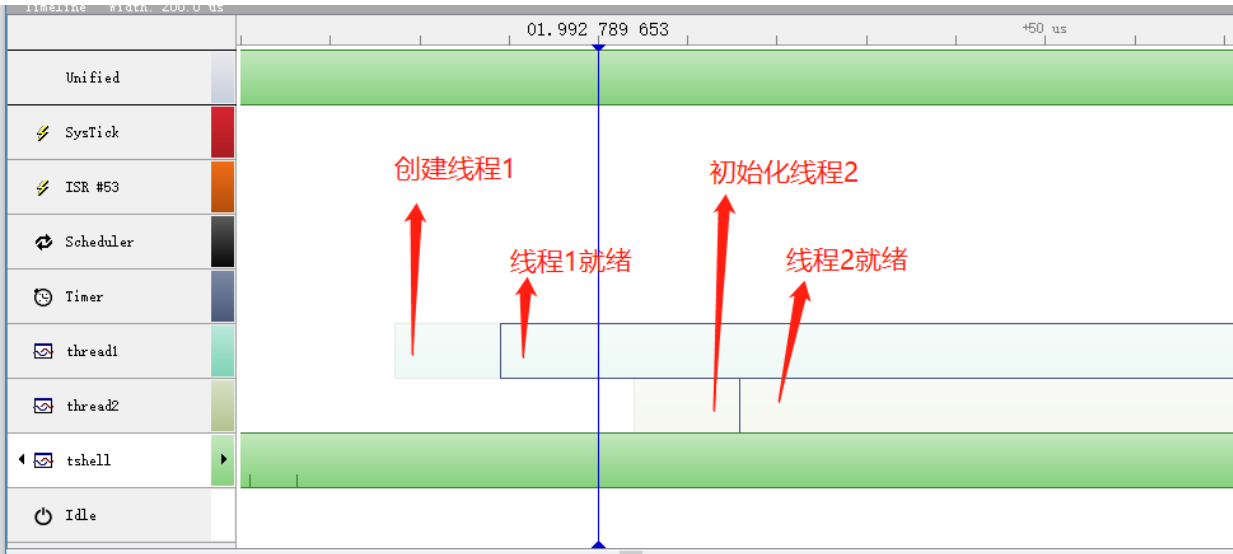


图 2.3: 线程创建、初始化细节

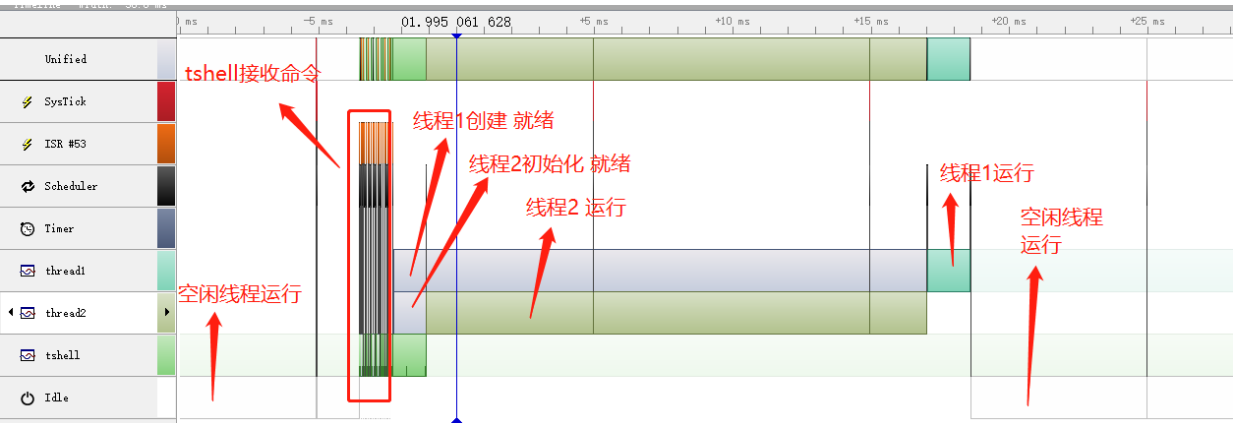


图 2.4: 线程间切换

图中各名称对应描述如下表：

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断

名称	描述
SysTick	系统时钟
Scheduler	调度器
thread1	线程 thread1
thread2	线程 thread2
Timer	定时器
tshell	线程 tshell
Idle	空闲线程

2.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [thread_sample.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 3 章

实验：线程的时间片轮转调度

3.1 实验目的

- 理解多线程时间片轮转的基本原理；
- 理解同优先级线程间的时间片轮转机制；
- 在 RT-Thread 中熟练使用时间片轮转来完成需求。

3.2 实验原理及程序结构

对优先级相同的线程采用时间片轮转的方式进行调度。

3.2.1 实验设计

本实验使用的例程为：[timeslice_sample.c](#)

为了体现时间片轮转，本实验设计了 **thread1**、**thread2** 两个相同优先级的线程，**thread1** 时间片为 10，**thread2** 时间片为 5，如果就绪列表中该优先级最高，则这两个线程会按照时间片长短被轮番调度。两个线程采用同一个入口函数，分别打印一条带有累加计数的信息（每个线程进入一次入口函数会将计数 **count++**，**count>200** 时线程退出）遵循时间片轮转调度机制。

通过本实验，用户可以清晰地了解到，同优先级线程在时间片轮转调度时刻的状态变迁。

整个实验运行过程如下图所示，OS Tick 为系统滴答时钟（精度 10ms），下面以实验开始后第一个到来的 OS Tick 为第 1 个 OS Tick，过程描述如下：

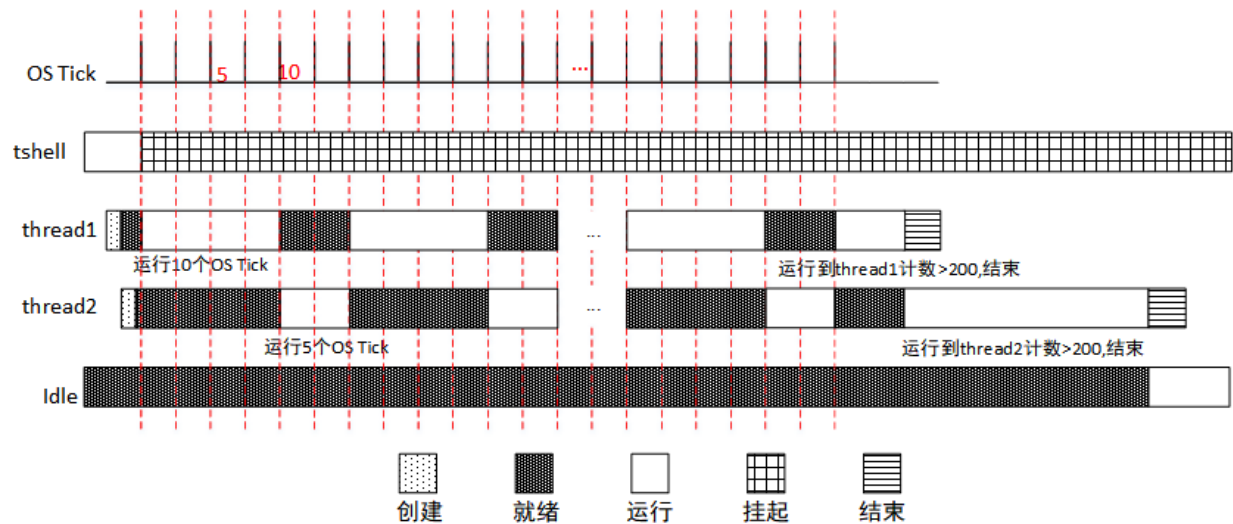


图 3.1: 实验运行过程

- (1) 在 tshell 线程中创建线程 thread1 和 thread2, 优先级相同为 20, thread1 时间片为 10, thread2 时间片为 5;
- (2) 启动线程 thread1 和 thread2, 使 thread1 和 thread2 处于就绪状态;
- (3) 在操作系统的调度下, thread1 首先被投入运行;
- (4) thread1 循环打印带有累计计数的信息, 当 thread1 运行到第 10 个时间片时, 操作系统调度 thread2 投入运行, thread1 进入就绪状态;
- (5) thread2 开始运行后, 循环打印带有累计计数的信息, 直到第 15 个 OS Tick 到来, thread2 已经运行了 5 个时间片, 操作系统调度 thread1 投入运行, thread2 进入就绪状态;
- (6) thread1 运行直到计数值 count>200, 线程 thread1 退出, 接着调度 thread2 运行直到计数值 count>200, thread2 线程退出; 之后操作统调度空闲线程投入运行;

注意: 时间片轮转机制, 在 OS Tick 到来时, 正在运行的线程时间片减 1。

3.2.2 源程序说明

3.2.2.1 RT-Thread 示例代码框架

RT-Thread 示例代码都通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

3.2.2.2 示例源码

定义了待创建线程需要用到的优先级, 栈空间, 时间片的宏:

```
#include <rtthread.h>

#define THREAD_STACK_SIZE    1024
#define THREAD_PRIORITY      20
#define THREAD_TIMESLICE     10
```

两个线程公共的入口函数，线程 **thread1** 和 **thread2** 采用同一个入口函数，但是变量分别存在不同的堆空间

```
/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_uint32_t value;
    rt_uint32_t count = 0;

    value = (rt_uint32_t)parameter;
    while (1)
    {
        if(0 == (count % 5))
        {
            rt_kprintf("thread %d is running ,thread %d count = %d\n", value , value
                , count);

            if(count> 200)
                return;
        }
        count++;
    }
}
```

线程时间片的示例函数，示例函数首先创建并启动了线程 **thread1**，然后创建并启动了线程 **thread2**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```
int timeslice_sample(void)
{
    rt_thread_t tid = RT_NULL;
    /* 创建线程 1 */
    tid = rt_thread_create("thread1",
                           thread_entry, (void*)1,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);

    if (tid != RT_NULL)
        rt_thread_startup(tid);

    /* 创建线程 2 */
    tid = rt_thread_create("thread2",
                           thread_entry, (void*)2,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE-5);

    if (tid != RT_NULL)
        rt_thread_startup(tid);

    return 0;
}

/* 导出到 msh 命令列表中 */
```



```
MSH_CMD_EXPORT(timeslice_sample, timeslice sample);
```

3.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 作为 msh 终端，可以看到系统的启动日志，输入 timeslice_sample 命令启动示例应用，示例输出结果如下：

```
\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Jun 14 2018
2006 - 2018 Copyright by rt-thread team
msh >timeslice_sample
msh >thread 1 is running ,thread 1 count = 0
thread 1 is running ,thread 1 count = 5
thread 1 is running ,thread 1 count = 10
thread 1 is running ,thread 1 count = 15
thread 1 is running ,thread 1 count = 20
...
thread 1 is running ,thread 1 count = 125
thread 1 is running ,thread 1 count = 1thread 2 is running ,thread 2 count = 0
thread 2 is running ,thread 2 count = 5
thread 2 is running ,thread 2 count = 10
...
thread 2 is running ,thread 2 count = 60
thread 2 is running ,thread 2 co30
thread 1 is running ,thread 1 count = 135
thread 1 is running ,thread 1 count = 140
thread 1 is running ,thread 1 count = 145
...
thread 1 is running ,thread 1 count = 205
unt = 205thread 2 is running ,thread 2 count = 70
thread 2 is running ,thread 2 count = 75
thread 2 is running ,thread 2 count = 80
...
thread 2 is running ,thread 2 count = 200
thread 2 is running ,thread 2 count = 205
```

线程 thread1 在 10 个 OS Tick 中，可计数约 125 左右，计数 > 200 会退出，所以下一次执行不了 10 个 OS Tick 就会退出了。由于“计数 > 200 会退出”，Thread1 与 thread2 只会轮番调度一次就会先后退出了。

使用 SystemView 工具可以监测示例实际运行过程，示例开始之后现象如下图，与实验设计相同，轮流切换打印的同时（只是由于 OS Tick 的精度 10ms，导致在计数 200 内，thread1 与 thread2 时间片只轮番调度了一次），遵循时间片轮转规则。

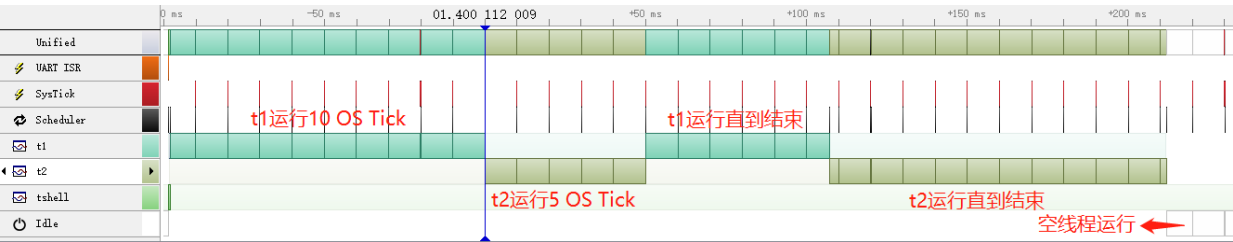


图 3.2: 示例运行流程图

图中各名称对应描述如下表:

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器
thread1	线程 thread1
thread2	线程 thread2
tshell	线程 tshell
Idle	空闲线程

3.4 附件

整个示例运行流程可以使用工具 SystemView 工具打开附件文件 [timeslice_sample.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

注：如果将 OS Tick 设置 1ms，又或者将示例代码中 `if(count> 200)` 中的 200 改成更大的数值（如 2000），那么实验效果就很明显了。OS Tick 设置为 1ms 效果如下图所示，附件详见 [timeslice_sample1.SVDat](#)

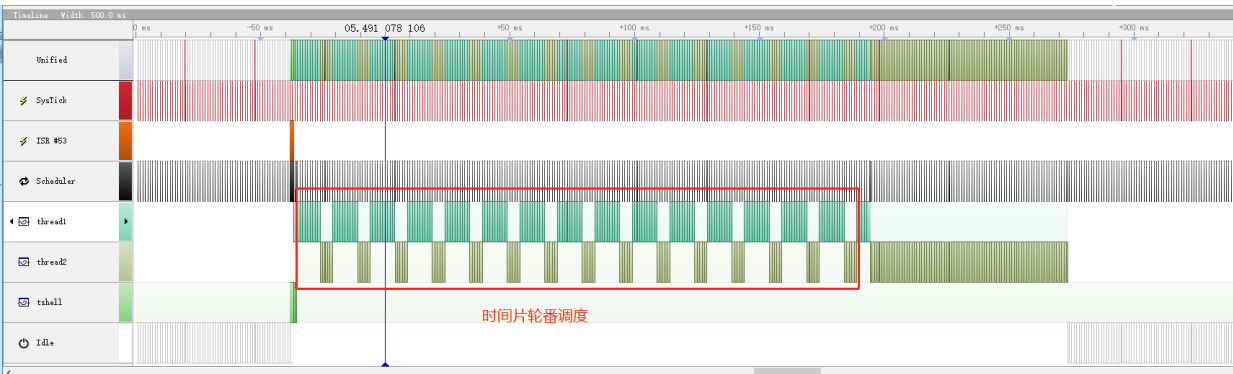


图 3.3: OS Tick 设置为 1ms 时的结果

第 4 章

实验：定时器的使用

4.1 实验目的

- 理解动态定时器的基本原理；
- 掌握 RT-Thread 中动态定时器的创建与使用；
- 在 RT-Thread 中熟练使用动态定时器来完成需求。

4.2 实验原理及程序结构

RT-Thread 定时器由操作系统提供的一类系统接口（函数），它构建在芯片的硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。

RT-Thread 定时器分为 `HARD_TIMER` 与 `SOFT_TIMER`，可以设置为单次定时与周期定时，这些属性均可在创建 / 初始化定时器时设置；而如果没有设置 `HARD_TIMER` 或 `SOFT_TIMER`，则默认使用 `HARD_TIMER`。

4.2.1 实验设计

本实验使用的例程为：[timer_sample.c](#)

为了体现动态定时器的单次定时与周期性定时，本实验设计了 `timer1`、`timer2` 两定时器。

周期性定时器 1 的超时函数，每 5 个 OS Tick 运行 1 次，共运行 5 次（5 次后调用 `rt_timer_stop` 使定时器 1 停止运行）；单次定时器 2 的超时函数在第 15 个 OS Tick 时运行一次。

通过本实验，用户可以清晰地了解到定时器的工作过程，以及使用定时器相关 API 动态更改定时器属性。

整个实验运行过程如下图所示，描述如下：

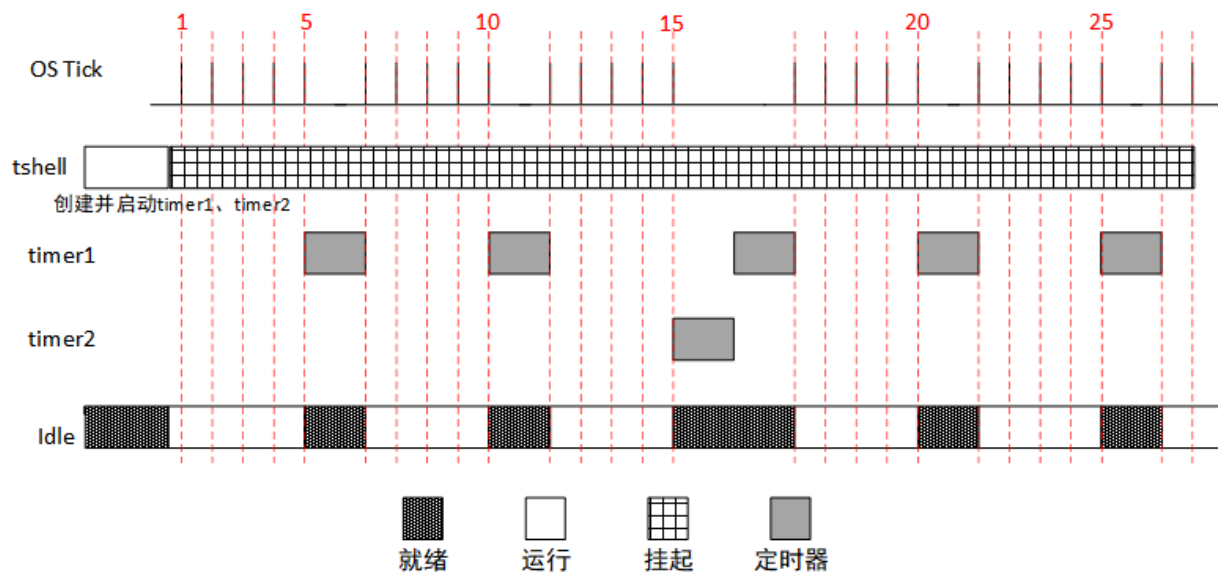


图 4.1: 实验运行过程

- (1) 在 **tshell** 线程中创建定时器 **timer1** 和 **timer2**, **timer1** 周期定时 5 OS Tick, **timer2** 单次定时 15 OS Tick; 启动定时器 **timer1**、**timer2**;
- (2) 定时器的定时时间均为到, 在操作系统的调度下, **Idle** 投入运行;
- (3) 每 5 个 OS Tick 到来时, 定时器 **timer1** 定时时间到, 调用超时函数打印一段信息, **timer1** 定时器重置;
- (4) 在第 15 个 OS Tick 到来时, **timer1** 第 3 次超时, 调用超时函数打印一段信息; **timer2** 第一次超时, 调用超时函数打印一段信息且超时函数运行完删除;
- (5) 在第 25 个 OS Tick 到来时, 定时器 **timer1** 第 5 次超时, 调用超时函数打印一段信息, 并使用 **rt_timer_stop0** 接口将定时器停止, 超时函数运行完后自行删除;

4.2.2 源程序说明

4.2.2.1 RT-Thread 示例代码框架

RT-Thread 示例代码都通过 **MSH_CMD_EXPORT** 将示例初始函数导出到 **msh** 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

4.2.2.2 示例源码

头文件以及定义了待创建定时器控制块以及实验需要用到的变量

```
#include <rtthread.h>

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_timer_t timer2;
static int cnt = 0;
```

周期定时器 **timer1** 的超时函数，**timer1** 定时时间到会执行次函数，10 次之后停止定时器 **timer1**。

```
/* 定时器 1 超时函数 */
static void timeout1(void *parameter)
{
    rt_kprintf("periodic timer is timeout %d\n", cnt);

    /* 运行第 10 次，停止周期定时器 */
    if (cnt++>= 9)
    {
        rt_timer_stop(timer1);
        rt_kprintf("periodic timer was stopped! \n");
    }
}
```

单次定时器 **timer2** 的超时函数，**timer2** 定时时间到会执行次函数

```
/* 定时器 2 超时函数 */
static void timeout2(void *parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}
```

定时器的示例代码，示例函数首先创建并启动了线程 **timer1**，然后创建并启动了线程 **timer2**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```
int timer_sample(void)
{
    /* 创建定时器 1 周期定时器 */
    timer1 = rt_timer_create("timer1", timeout1,
                             RT_NULL, 10,
                             RT_TIMER_FLAG_PERIODIC);

    /* 启动定时器 1 */
    if (timer1 != RT_NULL)
        rt_timer_start(timer1);

    /* 创建定时器 2 单次定时器 */
    timer2 = rt_timer_create("timer2", timeout2,
                             RT_NULL, 30,
                             RT_TIMER_FLAG_ONE_SHOT);

    /* 启动定时器 2 */
    if (timer2 != RT_NULL)
        rt_timer_start(timer2);

    return 0;
}

/* 导出到 msh 命令列表中 */
```

```
MSH_CMD_EXPORT(timer_sample, timer sample);
```

以上为示例函数，可以看到将函数使用 `MSH_CMD_EXPORT` 导出命令，示例函数首先创建并启动了定时器 `timer1`，然后创建并启动了定时器 `timer2`。

4.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 `UART#1` 做为 `msh` 终端，可以看到系统的启动日志，输入 `timer_sample` 命令启动示例应用，示例输出结果如下：

```
\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >timer_sample
msh >periodic timer is timeout 0
periodic timer is timeout 1
one shot timer is timeout
periodic timer is timeout 2
periodic timer is timeout 3
periodic timer is timeout 4
periodic timer was stopped!
```

使用 `SystemView` 工具可以监测示例实际运行过程，如下图所示。`Systemview` 没有区分定时器名称，详细时间信息可以看到定时器标号。

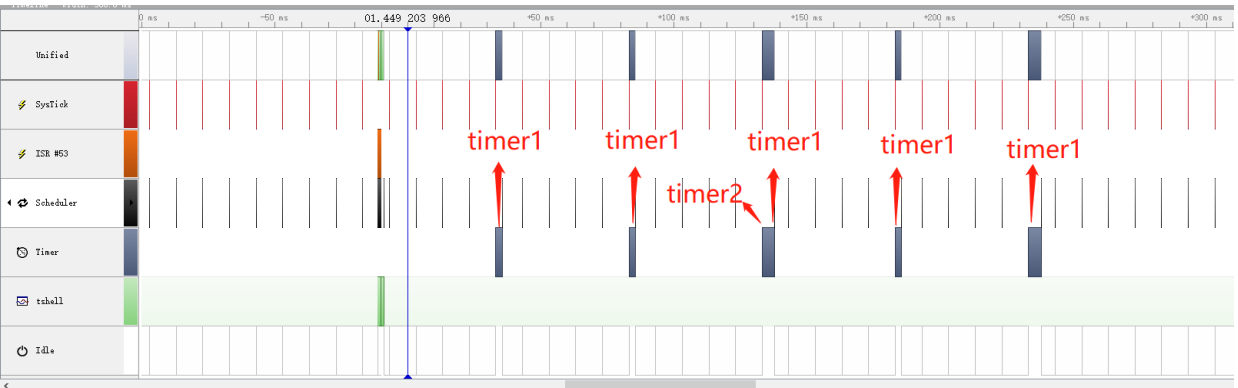


图 4.2: 示例运行流程图

图中各名称对应描述如下表：

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器

名称	描述
Timer	定时器
tshell	线程 tshell
Idle	空闲线程

4.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [timer_sample.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 5 章

实验：信号量—生产者消费者问题

5.1 实验目的

- 理解信号量的基本原理；
- 使用信号量来达到线程间同步；
- 理解资源计数适合于线程间工作处理速度不匹配的场所；
- 在 RT-Thread 中熟练使用信号量来完成需求。

5.2 实验原理及程序结构

信号量在大于 0 时才能获取，在中断、线程中均可释放信号量。

5.2.1 实验设计

本实验使用的例程为：[producer_consumer.c](#)

为了体现使用信号量来达到线程间的同步，本实验设计了 `producer`、`consumer` 两个线程，`producer` 优先级为 24，`consumer` 优先级为 26。线程 `producer` 每生产一个数据进入 20ms 延时，生产 10 个数据后结束。线程 `consumer` 每消费一个数据进入 50ms 延时，消费 10 个数据后结束。通过本实验，用户可以清晰地了解到，信号量在线程同步以及资源计数时起到的作用。

整个实验运行过程如下图所示，OS Tick 为系统滴答时钟，下面以实验开始后第一个到来的 OS Tick 为第 1 个 OS Tick，过程描述如下：

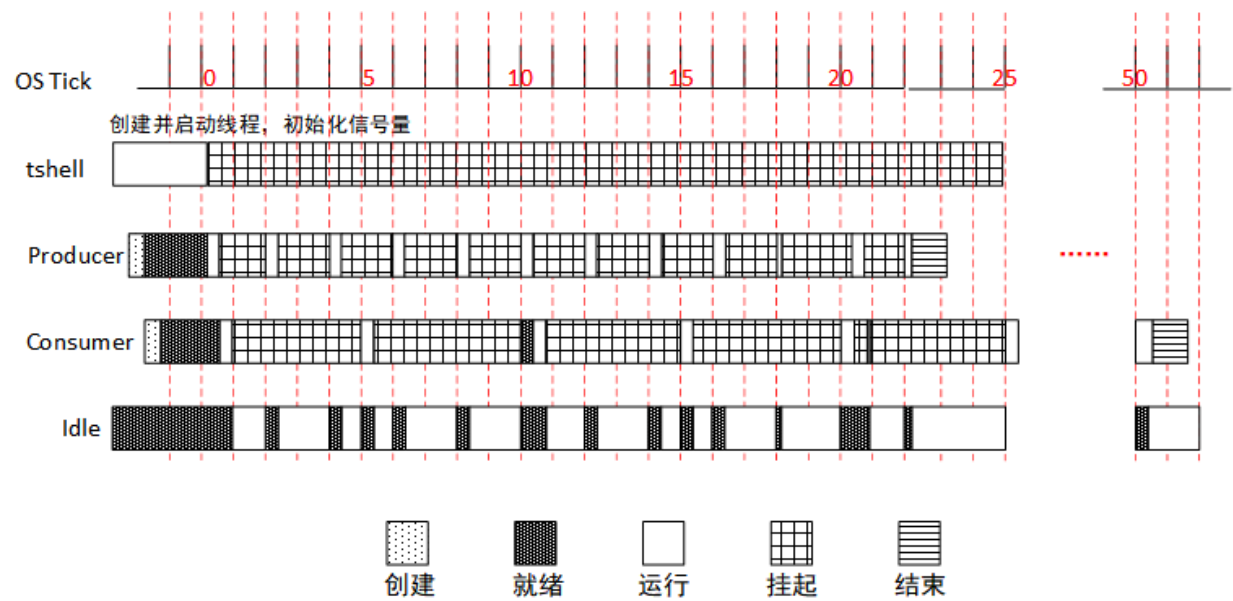


图 5.1: 实验运行过程

(1) 在 tshell 线程中初始化 3 个信号量，lock 初始化为 1（用作保护临界区，保护数组），empty 初始化为 5，full 初始化为 0；信号量情况：

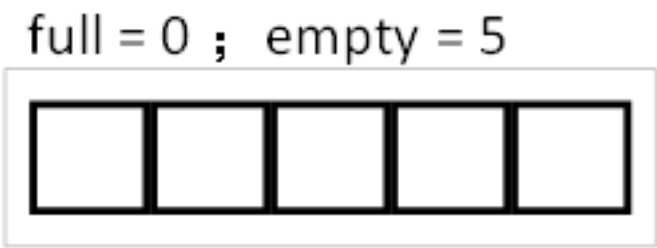


图 5.2: 信号量情况 1

- (2) 创建并启动线程 producer，优先级为 24；创建并启动线程和 consumer，优先级为 26；
- (3) 在操作系统的调度下，producer 优先级高，首先被投入运行；
- (4) producer 获取一个 empty 信号量，产生一个数据放入数组，再释放一个 full 信号量，然后进入 2 OS Tick 延时；之后的信号量情况：

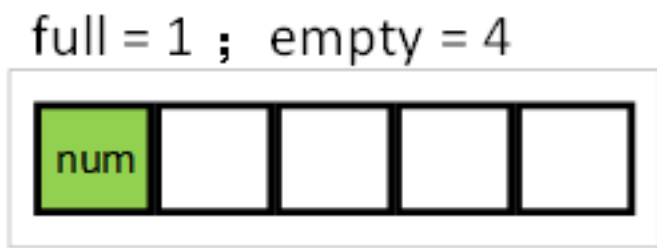


图 5.3: 信号量情况 2

- (5) 随后 consumer 投入运行，获取一个 full 信号量，消费一个数据用于累加，再释放一个 empty 信

号量，然后进入 5 OS Tick 延时；之后的信号量情况：

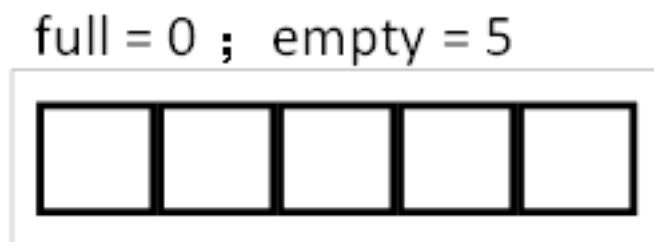


图 5.4: 信号量情况 3

(6) 由于生产速度 > 消费速度，所以在某一时刻会存在 full = 5 / empty = 0 的情况，如下：

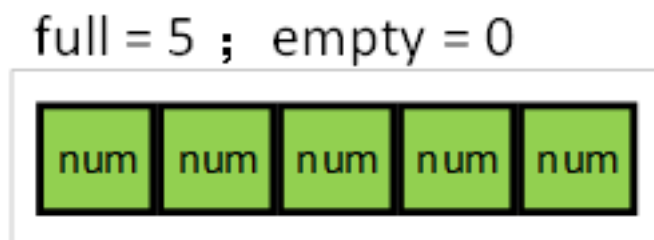


图 5.5: 信号量情况 4

比如第 18 个 OS Tick 时，producer 延时结束，操作系统调度 producer 投入运行，获取一个 empty 信号量，由于此时 empty 信号量为 0，producer 由于获取不到信号量挂起；等待有 empty 信号时，才可以继续生产。

(7) 直到 producer 产生 10 个 num 后，producer 线程结束，被系统删除。

(8) 直到 consumer 消费 10 个 num 后，consumer 线程结束，被系统删除。

5.2.2 源程序说明

5.2.2.1 RT-Thread 示例代码框架

RT-Thread 示例代码都通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

5.2.2.2 示例源码

定义了待创建线程需要用到的优先级，栈空间，时间片的宏，以及生产消费过程中用于存放产生数据的数字和相关变量、线程句柄、信号量控制块。

```
#include <rtthread.h>

#define THREAD_PRIORITY      6
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5
```

```
/* 定义最大 5 个元素能够被产生 */
#define MAXSEM 5

/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];

/* 指向生产者、消费者在 array 数组中的读写位置 */
static rt_uint32_t set, get;

/* 指向线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;
```

生产者 **producer** 线程的入口函数，每 20ms 就获取一个空位（获取不到时挂起），上锁，产生一个数字写入数组，解锁，释放一个满位，10 次后结束。

```
/* 生产者线程入口 */
void producer_thread_entry(void *parameter)
{
    int cnt = 0;

    /* 运行 10 次 */
    while (cnt < 10)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改 array 内容，上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set % MAXSEM] = cnt + 1;
        rt_kprintf("the producer generates a number: %d\n", array[set % MAXSEM]);
        set++;
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        rt_thread_mdelay(20);
    }

    rt_kprintf("the producer exit!\n");
}
```

消费者 **consumer** 线程的入口函数，每 50ms 获取一个满位（获取不到时挂起），上锁，将数组中的内容相加，解锁，释放一个空位，10 次后结束。

```
/* 消费者线程入口 */
void consumer_thread_entry(void *parameter)
{
    rt_uint32_t sum = 0;

    while (1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区，上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get % MAXSEM];
        rt_kprintf("the consumer[%d] get a number: %d\n", (get % MAXSEM), array[get % MAXSEM]);
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到 10 个数目，停止，消费者线程相应停止 */
        if (get == 10) break;

        /* 暂停一小会时间 */
        rt_thread_mdelay(50);
    }

    rt_kprintf("the consumer sum is: %d\n", sum);
    rt_kprintf("the consumer exit!\n");
}
```

生产者与消费者问题的示例函数，示例函数首先初始化了 3 个信号量，创建并启动生产者线程 **producer**，然后创建、启动消费者线程 **consumer**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```
int producer_consumer(void)
{
    set = 0;
    get = 0;

    /* 初始化 3 个信号量 */
    rt_sem_init(&sem_lock, "lock", 1, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_empty, "empty", MAXSEM, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_full, "full", 0, RT_IPC_FLAG_FIFO);

    /* 创建生产者线程 */
    producer_tid = rt_thread_create("producer",
```

```

                                producer_thread_entry, RT_NULL,
                                THREAD_STACK_SIZE,
                                THREAD_PRIORITY - 1, THREAD_TIMESLICE);

if (producer_tid != RT_NULL)
    rt_thread_startup(producer_tid);

/* 创建消费者线程 */
consumer_tid = rt_thread_create("consumer",
                                consumer_thread_entry, RT_NULL,
                                THREAD_STACK_SIZE,
                                THREAD_PRIORITY + 1, THREAD_TIMESLICE);

if (consumer_tid != RT_NULL)
    rt_thread_startup(consumer_tid);

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(producer_consumer, producer_consumer sample);

```

5.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 **producer_consumer** 命令启动示例应用，示例输出结果如下：

```

\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >producer_consumer
the producer generates a number: 1
the consumer[0] get a number: 1
msh >the producer generates a number: 2
the producer generates a number: 3
the consumer[1] get a number: 2
the producer generates a number: 4
the producer generates a number: 5
the producer generates a number: 6
the consumer[2] get a number: 3
the producer generates a number: 7
the producer generates a number: 8
the consumer[3] get a number: 4
the producer generates a number: 9
the consumer[4] get a number: 5
the producer generates a number: 10
the producer exit!
the consumer[0] get a number: 6
the consumer[1] get a number: 7

```

```
the consumer[2] get a number: 8
the consumer[3] get a number: 9
the consumer[4] get a number: 10
the consumer sum is: 55
the consumer exit!
```

使用 **SystemView** 工具可以监测示例实际运行过程，示例开始之后现象与实验设计相同，生产者每 20ms 生产一个数据，生产 10 个数据后结束，且最多存在 5 个未被消费的数据。消费者每 50ms 消费一个数据。如下图所示，图中红色数字表示当前线程执行之后 **empty** 信号量的值。



图 5.6: 示例运行流程图

图中各名称对应描述如下表：

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器
producer	线程 producer
consumer	线程 consumer
Timer	定时器
tshell	线程 tshell
Idle	空闲线程

5.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [producer_consumer.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 6 章

实验：互斥量——优先级继承

6.1 实验目的

- 理解互斥量的基本原理；
- 使用互斥量来达到线程间同步并探索其中的优先级继承问题；
- 在 RT-Thread 中熟练使用互斥量来完成需求。

6.2 实验原理及程序结构

互斥量是一种特殊的二值信号量。它和信号量不同的是：拥有互斥量的线程拥有互斥量的所有权，互斥量支持递归访问且能防止线程优先级翻转；并且互斥量只能由持有线程释放，而信号量则可以由任何线程释放。

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁的状态，而被线程持有的时候则立刻转为闭锁的状态。

注意：需要切记的是互斥量不能在中断服务例程中使用。

6.2.1 实验设计

本实验使用的例程为：[priority_inversion.c](#)

为了体现使用互斥量来达到线程间的同步，并体现优先级继承的现象，本实验设计了 `thread1`、`thread2`、`thread3` 三个线程，优先级分别为 9、10、11，设计了一个互斥量 `mutex`。

线程 `thread1` 优先级最高，先执行 100ms 延时，之后再打印线程 2 与线程 3 的优先级信息——用于检查线程 `thread3` 的优先级是否被提升为 `thread2` 的优先级。

线程 `thread2` 进入后先打印自己的优先级，然后进入 50ms 延时，延时结束后获取互斥量 `mutex`，获取到互斥量之后再释放互斥量 `mutex`。

线程 `thread3` 进入后先打印自己的优先级，然后获取互斥量 `mutex`，获取到互斥量之后进入 500ms 的循环，循环结束后将互斥量释放。

整体情况就是：线程 3 先持有互斥量，而后线程 2 试图持有互斥量，此时线程 3 的优先级应该被提升为和线程 2 的优先级相同，然后线程 1 打印线程 2 与线程 3 的优先级信息。

通过本实验，用户可以清晰地了解到，互斥量在线程间同步的作用、互斥量的优先级继承性以及互斥量连续获取不会造成死锁。

整个实验运行过程如下图所示，过程描述如下：

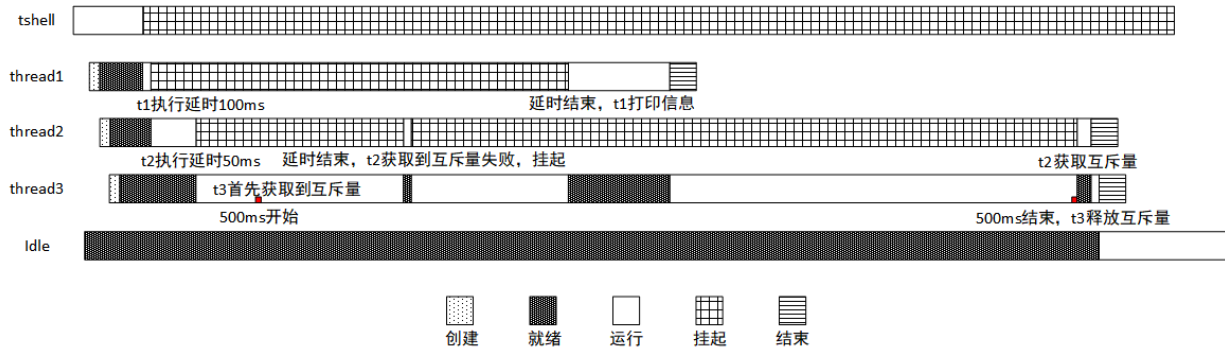


图 6.1: 实验运行过程

(1) 在 tshell 线程中创建一个互斥量 mutex，初始化为先进先出型；并分别创建、启动线程 thread1、thread2、thread3，优先级分别为 9、10、11；

(2) thread1 开始执行，延时 100ms 将自己挂起；

(3) thread2 开始执行，打印自己的优先级信息，开始延时 50ms 将自己挂起；

(4) thread3 获取互斥量，然后使用循环 500ms 来模拟 thread3 运行 500ms，之后释放互斥量。

(5) 在 thread2 延时 50ms 结束时，试图获取互斥量，由于互斥量被 thread3 持有，所以获取失败，自身挂起。（此时，thread3 的优先级应该是被提升为和 thread2 的优先级相同）。

(6) 在 thread1 延时 100ms 结束时，打印 thread2 与 thread3 的优先级信息，检查两者优先级是否相同。如果相同，那么说明互斥量确实解决了优先级翻转的问题，进行了优先级继承。

6.2.2 源程序说明

6.2.2.1 RT-Thread 示例代码框架

RT-Thread 示例代码都通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

6.2.2.2 示例源码

定义了待创建线程需要用到的优先级，栈空间，时间片的宏，以及线程控制块句柄和互斥量控制块句柄

```
#include <rtthread.h>

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
```



```
static rt_thread_t tid3 = RT_NULL;
static rt_mutex_t mutex = RT_NULL;

#define THREAD_PRIORITY      10
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5
```

线程 **thread1** 入口函数，首先让低优先级先运行，之后打印 **thread2** 与 **thread3** 的优先级，验证互斥量优先级继承。

```
/* 线程 1 入口 */
static void thread1_entry(void *parameter)
{
    /* 先让低优先级线程运行 */
    rt_thread_mdelay(100);

    /* 此时 thread3 持有 mutex，并且 thread2 等待持有 mutex */

    /* 检查 thread2 与 thread3 的优先级情况 */
    if (tid2->current_priority != tid3->current_priority)
    {
        /* 优先级不相同，测试失败 */
        rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);
        rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);
        rt_kprintf("test failed.\n");
        return;
    }
    else
    {
        rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);
        rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);
        rt_kprintf("test OK.\n");
    }
}
```

线程 **thread2** 的入口函数，打印优先级信息之后，先让低优先级的 **thread3** 先运行，然后尝试获取互斥量，获取到后释放互斥量。

```
/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    rt_err_t result;

    rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);

    /* 先让低优先级线程运行 */
    rt_thread_mdelay(50);

    /*
```

```

    * 试图持有互斥锁，此时 thread3 持有，应把 thread3 的优先级提升
    * 到 thread2 相同的优先级
    */
    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);

    if (result == RT_EOK)
    {
        /* 释放互斥锁 */
        rt_mutex_release(mutex);
    }
}

```

线程 **thread3** 的入口函数，先打印自身优先级信息，然后获取互斥量，获取到互斥量之后进行 500ms 的长时间循环，使 **thread3** 运行 500ms 左右，之后释放互斥量。

```

/* 线程 3 入口 */
static void thread3_entry(void *parameter)
{
    rt_tick_t tick;
    rt_err_t result;

    rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);

    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        rt_kprintf("thread3 take a mutex, failed.\n");
    }

    /* 做一个长时间的循环，500ms */
    tick = rt_tick_get();
    while (rt_tick_get() - tick < (RT_TICK_PER_SECOND / 2)) ;

    rt_mutex_release(mutex);
}

```

互斥量优先级继承的例子，解决优先级翻转问题。示例函数首先创建互斥量，再创建、启动了线程 **thread1**、**thread2**、**thread3**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

int pri_inversion(void)
{
    /* 创建互斥锁 */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_FIFO);
    if (mutex == RT_NULL)
    {
        rt_kprintf("create dynamic mutex failed.\n");
        return -1;
    }
}

```

```

/* 创建线程 1 */
tid1 = rt_thread_create("thread1",
                        thread1_entry,
                        RT_NULL,
                        THREAD_STACK_SIZE,
                        THREAD_PRIORITY - 1, THREAD_TIMESLICE);

if (tid1 != RT_NULL)
    rt_thread_startup(tid1);

/* 创建线程 2 */
tid2 = rt_thread_create("thread2",
                        thread2_entry,
                        RT_NULL,
                        THREAD_STACK_SIZE,
                        THREAD_PRIORITY, THREAD_TIMESLICE);

if (tid2 != RT_NULL)
    rt_thread_startup(tid2);

/* 创建线程 3 */
tid3 = rt_thread_create("thread3",
                        thread3_entry,
                        RT_NULL,
                        THREAD_STACK_SIZE,
                        THREAD_PRIORITY + 1, THREAD_TIMESLICE);

if (tid3 != RT_NULL)
    rt_thread_startup(tid3);

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pri_inversion, pri_inversion sample);

```

6.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 `mutex_simple_init` 命令启动示例应用，示例输出结果如下：

```

\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >pri_inversion
the priority of thread2 is: 10
the priority of thread3 is: 11
the priority of thread2 is: 10
the priority of thread3 is: 10
test OK.

```

例程演示了互斥量的使用方法。线程 3 先持有互斥量，而后线程 2 试图持有互斥量，此时线程 3 的优先级被提升为和线程 2 的优先级相同。

使用 **SystemView** 工具可以监测示例实际运行过程，示例开始之后现象与实验设计相同，如下几张图所示。

几个阶段详细调度信息如下：

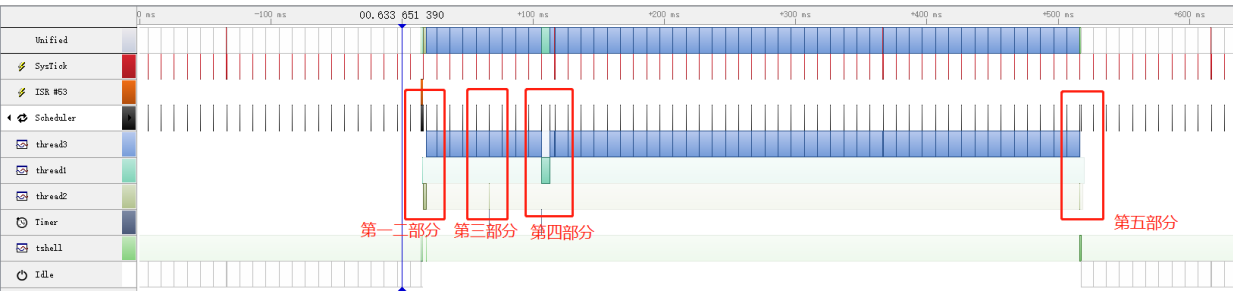


图 6.2: 运行整体图

第一、二部分放大图如下：

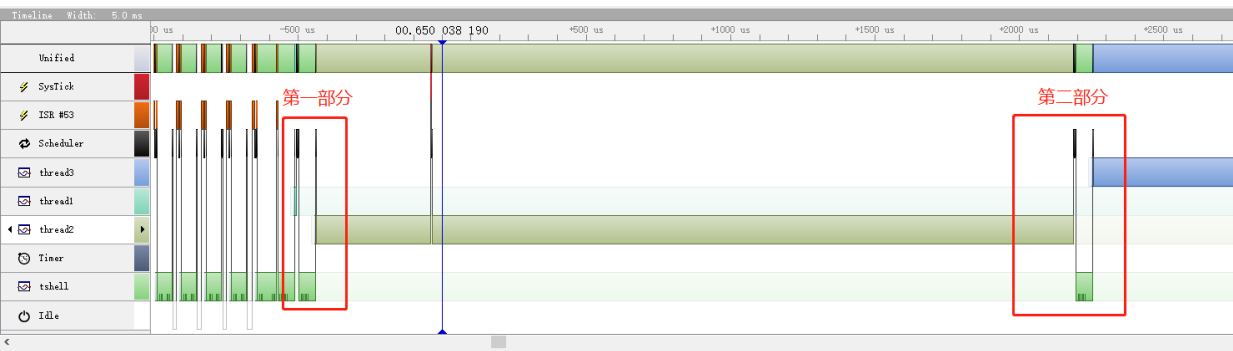


图 6.3: 第一二部分放大图

第一部分细节图如下：



图 6.4: 第一部分细节图 - 创建线程 1、2

第二部分细节图如下：

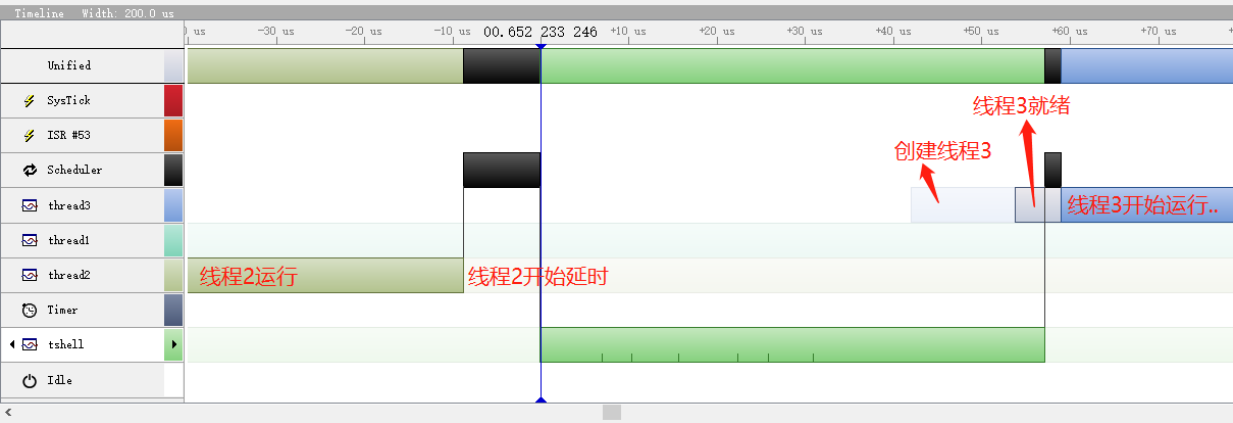


图 6.5: 第二部分细节图

第三部分细节图如下，thread2 获取互斥量详细过程：



图 6.6: 第三部分细节图

第四部分细节图如下：

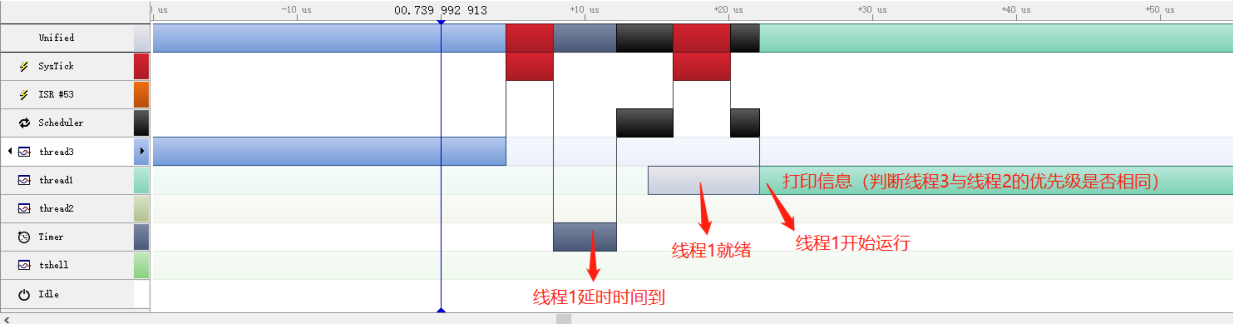


图 6.7: 第四部分细节图

第五部分细节图如下，结束阶段详细调度过程：



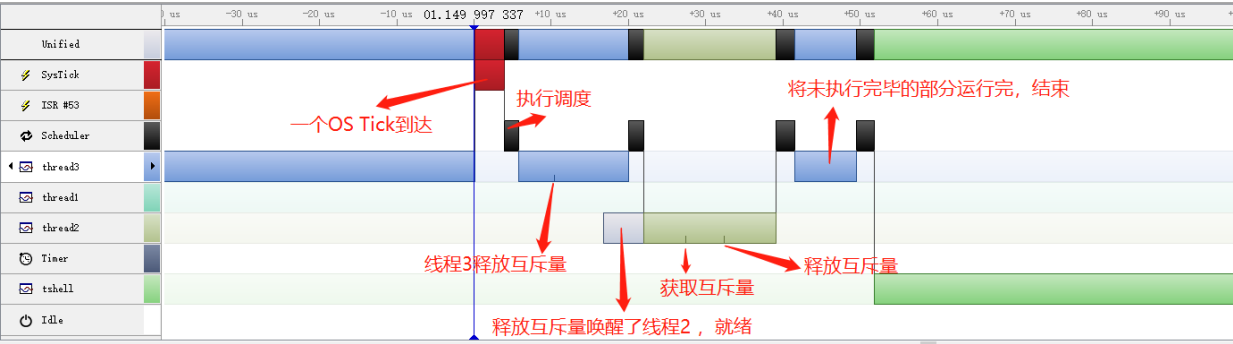


图 6.8: 第五部分细节图

图中各名称对应描述如下表:

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器
Timer	定时器
thread3	线程 thread3
thread2	线程 thread2
thread1	线程 thread1
tshell	线程 tshell
Idle	空闲线程

6.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [pri_inversion.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 7 章

实验：事件集的使用

7.1 实验目的

- 理解事件集的基本原理；
- 使用事件集来达到多条件情况下线程间同步；
- 在 RT-Thread 中熟练使用事件集来完成需求。

7.2 实验原理及程序结构

事件集主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。即一个线程与多个事件的关系可设置为：其中任意一个事件唤醒线程，或几个事件都到达后才唤醒线程进行后续的处理；同样，事件也可以是多个线程同步多个事件。

7.2.1 实验设计

本实验使用的例程为：[event_sample.c](#)

为了体现使用事件集来达到线程间的同步，本实验设计了 `thread1`、`thread2` 两个线程，优先级分别为 8、9，设计了一个事件集 `event`。

线程 `thread1` 进入后接收事件组合“事件 3 或事件 5”，接收到事件时候进行 100ms 延时，然后接收事件组合“事件 3 与事件 5”，接收完成后结束线程。

线程 `thread2` 进入后发送事件 3，延时 200ms；发送事件 5，延时 200ms；发送事件 3，完成后结束线程。

整体情况：`thread1` 首先等待“事件 3 或事件 5”的到来，`thread2` 发送事件 3，唤醒 `thread1` 接收事件，之后 `thread1` 等待“事件 3 与事件 5”；`thread2` 再发送事件 5，进行延时，`thread2` 发送事件 3，等 `thread1` 延时结束就能接收事件组合“事件 3 与事件 5”。

通过本实验，用户可以清晰地了解到，线程在同时接收多个事件和接收多个事件中的一个时的运行情况。

整个实验运行过程如下图所示，过程描述如下：

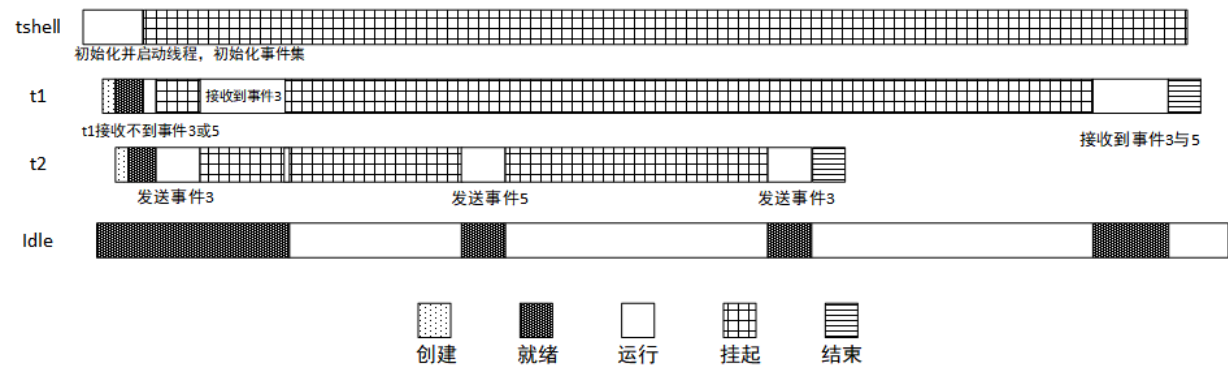


图 7.1: 实验运行过程

- (1) 在 `tshell` 线程中初始化一个事件集 `event`，初始化为先进先出型；并分别初始化、启动线程 `thread1`、`thread2`，优先级分别为 8、9；
- (2) 在操作系统的调度下，`thread1` 优先级高，首先被投入运行；`thread1` 开始运行后接收事件集 3 或 5，由于未接收到事件集 3 或 5，线程 `thread1` 挂起；
- (3) 随后操作系统调度 `thread2` 投入运行，`thread2` 发送事件 3，然后执行延时将自己挂起 200ms；
- (4) `thread1` 接收到事件 3，打印相关信息，并开始等待“事件 3 与事件 5”；
- (5) `thread2` 的延时时间到，发送事件 5；延时，发送事件 3；
- (6) 等待 `thread1` 的延时结束后，可以马上接收到“事件 3 与事件 5”，打印信息，结束运行；

7.2.2 源程序说明

7.2.2.1 RT-Thread 示例代码框架

RT-Thread 示例代码都通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

7.2.2.2 示例源码

例子中初始化一个事件集，初始化两个静态线程。一个线程等待自己关心的事件的发生，另外一个线程发生事件。以下定义了待创建线程需要用到的优先级，栈空间，时间片的宏，事件控制块。

```
#include <rtthread.h>
#define THREAD_PRIORITY      9
#define THREAD_TIMESLICE    5
#define EVENT_FLAG3 (1 << 3)
#define EVENT_FLAG5 (1 << 5)

/* 事件控制块 */
static struct rt_event event;
```

线程 `thread1` 的栈空间、线程控制块以及线程 `thread1` 的入口函数，共接收两次事件，第一次永久等待“事件 3 或事件 5”，第二次永久等待“事件 3 与事件 5”


```

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口函数 */
static void thread1_recv_event(void *param)
{
    rt_uint32_t e;

    /* 第一次接收事件，事件 3 或事件 5 任意一个可以触发线程 1，接收完后清除事件标志 */
    if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                     RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                     RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: OR recv event 0x%x\n", e);
    }

    rt_kprintf("thread1: delay 1s to prepare the second event\n");
    rt_thread_mdelay(1000);

    /* 第二次接收事件，事件 3 和事件 5 均发生时才可以触发线程 1，接收完后清除事件标志 */
    if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                     RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                     RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: AND recv event 0x%x\n", e);
    }
    rt_kprintf("thread1 leave.\n");
}

```

线程 **thread2** 的栈空间、线程控制块以及线程 **thread1** 的入口函数，发送 3 次事件，发送事件 3，延时 200ms；发送事件 5，延时 200ms；发送事件 3，结束。

```

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_send_event(void *param)
{
    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event5\n");
    rt_event_send(&event, EVENT_FLAG5);
    rt_thread_mdelay(200);
}

```

```

    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_kprintf("thread2 leave.\n");
}

```

事件的示例代码，初始化一个事件对象，初始化并启动线程 **thread1**、**thread2**，并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

int event_sample(void)
{
    rt_err_t result;

    /* 初始化事件对象 */
    result = rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init event failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                   "thread1",
                   thread1_recv_event,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),
                   THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                   "thread2",
                   thread2_send_event,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),
                   THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(event_sample, event sample);

```

7.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 event_sample 命令启动示例应用，示例输出结果如下：

```
\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >event_sample
thread2: send eventthread3
thread1: OR recv event 0x8
thread1: delay 1s to prepare the second event
msh >thread2: send event5
thread2: send eventthread3
thread2 leave.
thread1: AND recv event 0x28
thread1 leave.
```

例程演示了事件集的使用方法。thread1 前后两次接收事件，分别使用了“逻辑或”与“逻辑与”的方法。

使用 SystemView 工具可以监测示例实际运行过程，示例开始之后现象与实验设计相同。如下图所示。

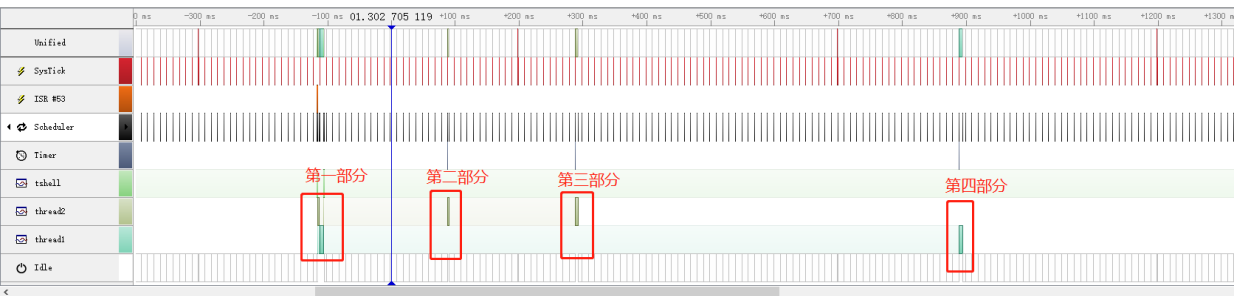


图 7.2: 运行整体图

第一部分：初始化事件与线程，详见下图；第二部分 thread2 发送事件 5；第三部分 thread2 发送事件 3，第四部分 thread1 延时结束接收到“事件 3 与事件 5”。

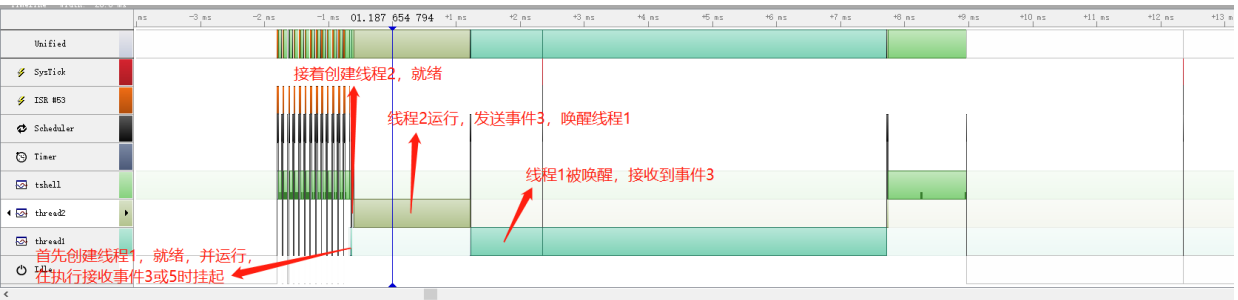


图 7.3: 第一部分细节

图中各名称对应描述如下表：

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器
Timer	定时器
thread2	线程 thread2
thread1	线程 thread1
tshell	线程 tshell
Idle	空闲线程

7.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [event_sample.SVdat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 8 章

实验：邮箱的使用

8.1 实验目的

- 理解邮箱的基本原理；
- 使用邮箱进行线程间通信；
- 在 RT-Thread 中熟练使用邮箱来完成需求。

8.2 实验原理及程序结构

邮箱是一种简单的线程间消息传递方式，特点是开销比较低，效率较高。在 RT-Thread 操作系统的实现中能够一次传递一个 4 字节大小的邮件，并且邮箱具备一定的存储功能，能够缓存一定数量的邮件数（邮件数由创建、初始化邮箱时指定的容量决定）。邮箱中一封邮件的最大长度是 4 字节，所以邮箱能够用于不超过 4 字节的消息传递。

8.2.1 实验设计

本实验使用的例程为：[mailbox_sample.c](#)

为了体现使用邮箱来达到线程间的通信，本实验设计了 `thread1`、`thread2` 两个线程，优先级同为 10，设计了一个邮箱 `mbt`。

线程 `thread1` 每 100ms 尝试接收一次邮件，如果接收到邮件就将邮件内容打印出来。在接收到结束邮件时，打印邮件信息，线程结束。

线程 `thread2` 每 200ms 发送一次邮件，发送 10 次之后，发送结束邮件（线程 2 共发送 11 封邮件），线程运行结束。

通过本实验，用户可以清晰地了解到，线程在使用邮箱时候的线程调度。

整个实验运行过程如下图所示，下面以 `thread2` 开始运行时为开始时间，过程描述如下：

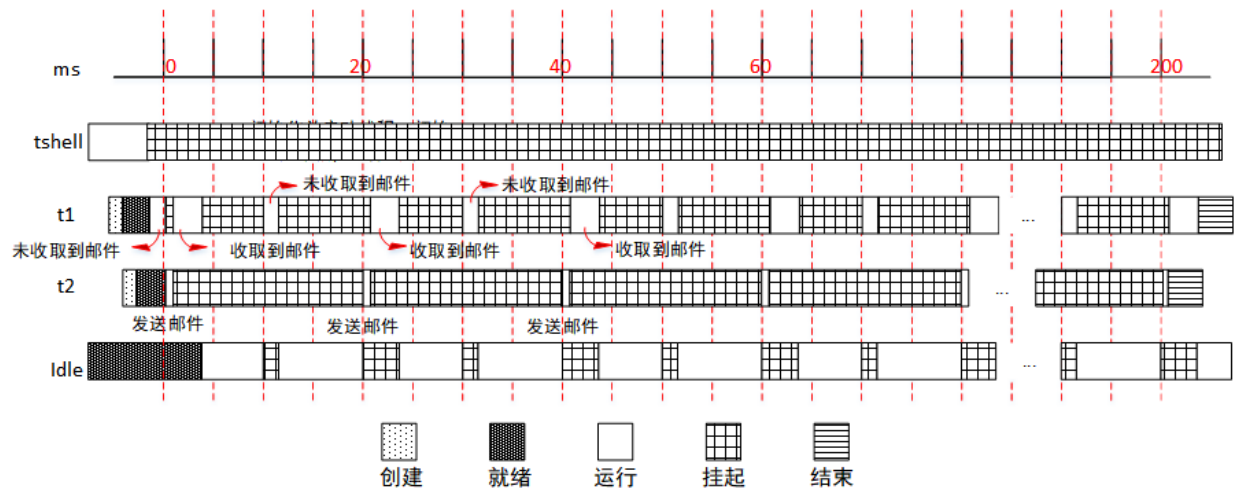


图 8.1: 实验运行过程

- (1) 在 tshell 线程中初始化一个邮箱 mbt, 采用 FIFO 方式进行线程等待; 初始化并启动线程 thread1、thread2, 优先级同为 10;
- (2) 在操作系统的调度下, thread1 首先被投入运行;
- (3) thread1 开始运行, 首先打印一段信息, 然后尝试获取邮件, 邮箱暂时没有邮件, thread1 挂起;
- (4) 随后操作系统调度 thread2 投入运行, 发送一封邮件, 随后进入 200ms 延时;
- (5) 此时线程 thread1 被唤醒, 接收到邮件, 继续打印一段信息, 然后进入 100ms 延时;
- (6) thread2 在发送 10 次邮件后, 发送一封结束内容的邮件, 线程结束。
- (7) thread1 一直接收邮件, 当接收到来自 thread2 的结束邮件后, 脱离邮箱, 线程结束。

8.2.2 源程序说明

8.2.2.1 RT-Thread 示例代码框架

RT-Thread 示例代码都通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

8.2.2.2 示例源码

以下定义了线程需要用到的优先级, 栈空间, 时间片的宏, 邮箱控制块, 存放邮件的内存池、3 份邮件内容。

```
#include <rtthread.h>

#define THREAD_PRIORITY    10
#define THREAD_TIMESLICE  5

/* 邮箱控制块 */
static struct rt_mailbox mb;
```

```
/* 用于放邮件的内存池 */
static char mb_pool[128];

static char mb_str1[] = "I'm a mail!";
static char mb_str2[] = "this is another mail!";
static char mb_str3[] = "over";
```

线程 **thread1** 使用的栈空间、线程控制块，以及线程 **thread1** 的入口函数，每 100ms 收取一次邮件并打印邮件内容，当收取到结束邮件的时候，脱离邮箱，结束运行。

```
ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口 */
static void thread1_entry(void *parameter)
{
    char *str;

    while (1)
    {
        rt_kprintf("thread1: try to recv a mail\n");

        /* 从邮箱中收取邮件 */
        if (rt_mb_recv(&mb, (rt_uint32_t *)&str, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: get a mail from mailbox, the content:%s\n", str);
            if (str == mb_str3)
                break;

            /* 延时 100ms */
            rt_thread_mdelay(100);
        }
    }
    /* 执行邮箱对象脱离 */
    rt_mb_detach(&mb);
}
```

线程 **thread2** 使用的栈空间、线程控制块，以及线程 **thread2** 的入口函数，每 200ms 发送一封邮件，10 次后发送结束邮件，结束运行

```
ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    rt_uint8_t count;
```

```

count = 0;
while (count < 10)
{
    count ++;
    if (count & 0x1)
    {
        /* 发送 mb_str1 地址到邮箱中 */
        rt_mb_send(&mb, (rt_uint32_t)&mb_str1);
    }
    else
    {
        /* 发送 mb_str2 地址到邮箱中 */
        rt_mb_send(&mb, (rt_uint32_t)&mb_str2);
    }

    /* 延时 200ms */
    rt_thread_mdelay(200);
}

/* 发送邮件告诉线程 1，线程 2 已经运行结束 */
rt_mb_send(&mb, (rt_uint32_t)&mb_str3);
}

```

邮箱的示例代码，初始化了邮箱，初始化并启动了线程 `thread1` 与 `thread2`。并将函数使用 `MSH_CMD_EXPORT` 导出命令

```

int mailbox_sample(void)
{
    rt_err_t result;

    /* 初始化一个 mailbox */
    result = rt_mb_init(&mb,
                        "mbt", /* 名称是 mbt */
                        &mb_pool[0], /* 邮箱用到的内存池是 mb_pool */
                        sizeof(mb_pool) / 4, /* 邮箱中的邮件数目，因为一封邮件占 4 字节 */
                        RT_IPC_FLAG_FIFO); /* 采用 FIFO 方式进行线程等待 */

    if (result != RT_EOK)
    {
        rt_kprintf("init mailbox failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                  "thread1",
                  thread1_entry,
                  RT_NULL,
                  &thread1_stack[0],

```



```

        sizeof(thread1_stack),
        THREAD_PRIORITY, THREAD_TIMESLICE);
rt_thread_startup(&thread1);

rt_thread_init(&thread2,
               "thread2",
               thread2_entry,
               RT_NULL,
               &thread2_stack[0],
               sizeof(thread2_stack),
               THREAD_PRIORITY, THREAD_TIMESLICE);
rt_thread_startup(&thread2);
return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(mailbox_sample, mailbox sample);

```

8.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 `mailbox_sample` 命令启动示例应用，示例输出结果如下：

```

\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh>mailbox_sample
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
msh>thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
...
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:over

```

使用 **SystemView** 工具可以监测示例实际运行过程，示例开始之后现象与实验设计相同。整体流程如下两张图所示，“接收”表示线程 t1 接收然后挂起，“收取”表示线程 t1 接收到邮件恢复运行，“发送”表示 t2 发送邮件。

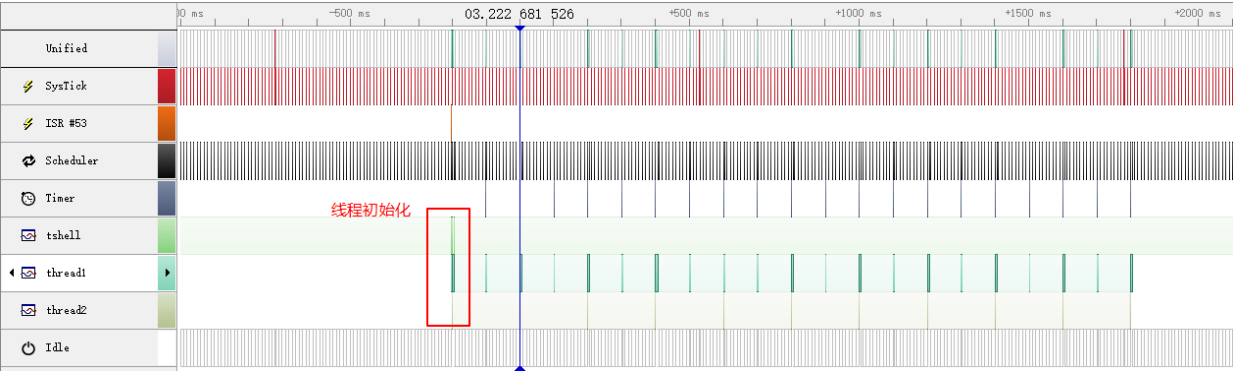


图 8.2: 运行整体图

起始阶段详细调度如下图所示：

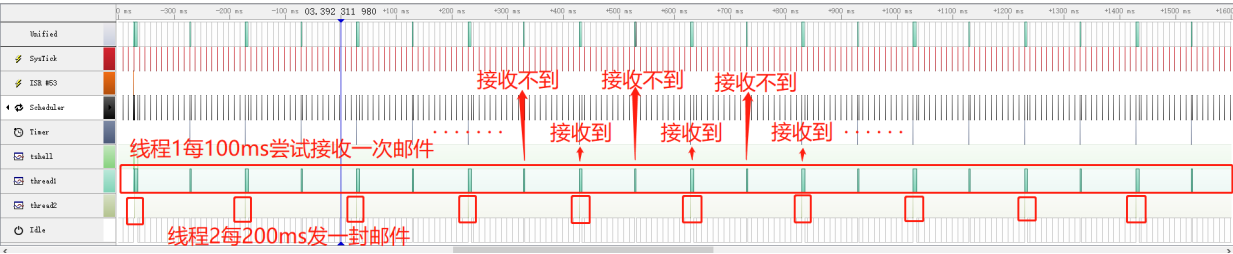


图 8.3: 运行过程

放大线程初始化部分：

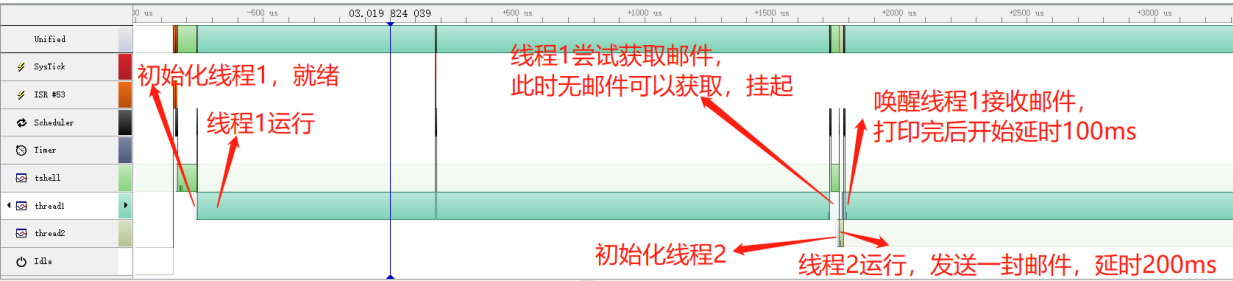


图 8.4: 线程初始化过程

图中各名称对应描述如下表：

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器
Timer	定时器
thread1	线程 thread1
Thread2	线程 thread2

名称	描述
tshell	线程 tshell
Idle	空闲线程

8.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [mailbox_sample.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 9 章

实验：消息队列的使用

9.1 实验目的

- 理解消息队列的基本原理
- 使用消息队列进行线程间通信
- 在 RT-Thread 中熟练使用消息队列来完成需求

9.2 实验原理及程序结构

消息队列能够接收来自线程或中断服务例程中不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。

9.2.1 实验设计

本实验使用的例程为：[msgq_sample.c](#)

为了体现使用消息队列来达到线程间的通信，本实验设计了 `thread1`、`thread2` 两个线程，优先级同为 25，设计了一个消息队列 `mqt`。

线程 `thread1` 每 50ms 从消息队列接收一次消息，并打印接收到的消息内容，在接收 20 次消息之后，将消息队列脱离、结束线程。

线程 `thread2` 每 10ms 向 `mqt` 消息队列依次发送 20 次消息，分别是消息“A”-“T”，第 9 次发送的是一个紧急消息“T”，发送 20 次后线程运行结束。（注，虽然设置的是 5ms，但是该工程设置的一个 OS Tick 是 10ms，是最小精度）。

通过本实验，用户可以清晰地了解到，线程在使用消息队列时候的线程调度。

整个实验运行过程如下图所示，OS Tick 为系统滴答时钟，下面以实验开始后第一个到来的 OS Tick 为第 1 个 OS Tick，过程描述如下：

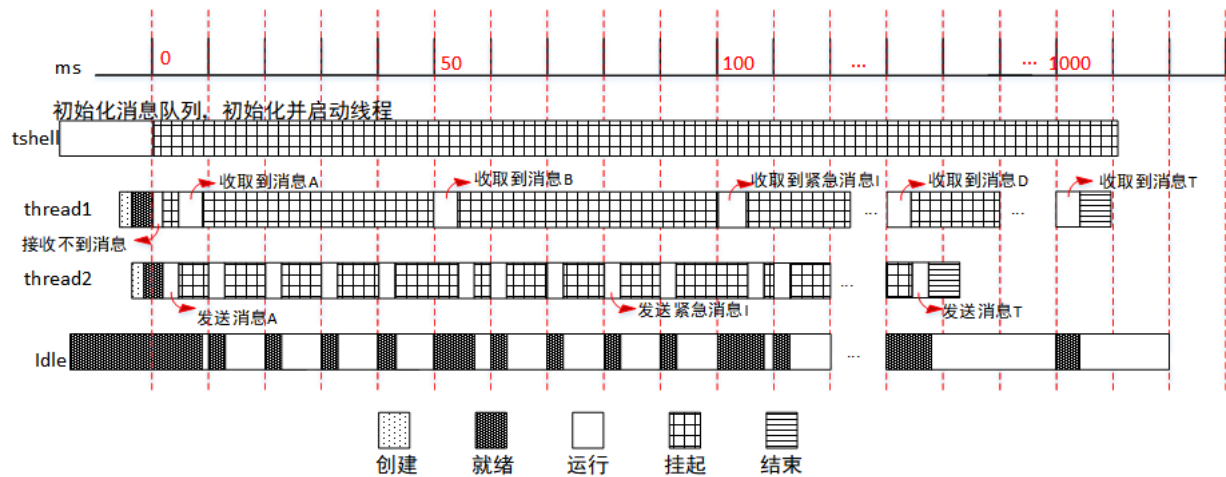


图 9.1: 实验运行过程

- (1) 在 `tshell` 线程中初始化一个消息队列 `mqt`，采用 `FIFO` 方式进行线程等待；初始化并启动线程 `thread1`、`thread2`，优先级同为 25；
- (2) 在操作系统的调度下，`thread1` 首先被投入运行，尝试从消息队列获取消息，消息队列暂时没有消息，线程挂起；
- (3) 随后操作系统调度 `thread2` 投入运行，`thread2` 发送一个消息“A”，并打印发送消息内容，随后每 10ms 发送一条消息；
- (4) 此时线程 `thread1` 接收到消息，打印消息内容“A”，然后每 50ms 接收一次消息；
- (5) 在第 100ms 时，`thread1` 本应接收消息“C”，但由于队列中有紧急消息，所以 `thread1` 先接收紧急消息“I”，之后再顺序接收其他消息。
- (6) `thread2` 发送 20 条消息后，结束线程。
- (7) `thread1` 接收 20 条消息后，结束线程。

9.2.2 源程序说明

9.2.2.1 示例代码框架

RT-Thread 示例代码都通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

9.2.2.2 示例源码

以下定义了待创建线程需要用到的优先级、时间片的宏，消息队列控制块以及存放消息用到的内存池。

```
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_TIMESLICE    5

/* 消息队列控制块 */
```

```
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static rt_uint8_t msg_pool[2048];
```

线程 **thread1** 使用的栈空间、线程控制块，以及线程 **thread1** 的入口函数，每 50ms 从消息队列中收取消息，并打印消息内容，20 次后结束。

```
ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口函数 */
static void thread1_entry(void *parameter)
{
    char buf = 0;
    rt_uint8_t cnt = 0;

    while (1)
    {
        /* 从消息队列中接收消息 */
        if (rt_mq_rcv(&mq, &buf, sizeof(buf), RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: rcv msg from msg queue, the content:%c\n", buf);
            if (cnt == 19)
            {
                break;
            }
        }
        /* 延时 50ms */
        cnt++;
        rt_thread_mdelay(50);
    }
    rt_kprintf("thread1: detach mq \n");
    rt_mq_detach(&mq);
}
```

线程 **thread2** 使用的栈空间、线程控制块，以及线程 **thread2** 的入口函数，每 5ms 向消息队列中发送消息，并打印消息内容，20 次后结束

```
ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    int result;
    char buf = 'A';
    rt_uint8_t cnt = 0;
```

```

while (1)
{
    if (cnt == 8)
    {
        /* 发送紧急消息到消息队列中 */
        result = rt_mq_urgent(&mq, &buf, 1);
        if (result != RT_EOK)
        {
            rt_kprintf("rt_mq_urgent ERR\n");
        }
        else
        {
            rt_kprintf("thread2: send urgent message - %c\n", buf);
        }
    }
    else if (cnt >= 20) /* 发送 20 次消息之后退出 */
    {
        rt_kprintf("message queue stop send, thread2 quit\n");
        break;
    }
    else
    {
        /* 发送消息到消息队列中 */
        result = rt_mq_send(&mq, &buf, 1);
        if (result != RT_EOK)
        {
            rt_kprintf("rt_mq_send ERR\n");
        }

        rt_kprintf("thread2: send message - %c\n", buf);
    }
    buf++;
    cnt++;
    /* 延时 5ms */
    rt_thread_mdelay(5);
}
}

```

消息队列的示例代码，初始化了一个消息队列，初始化并启动了 **thread1** 与 **thread2**. 并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

/* 消息队列示例的初始化 */
int msgq_sample(void)
{
    rt_err_t result;

    /* 初始化消息队列 */
    result = rt_mq_init(&mq,

```

```

        "mqt",
        &msg_pool[0],          /* 内存池指向 msg_pool */
        1,                    /* 每个消息的大小是 1 字节 */
        sizeof(msg_pool),      /* 内存池的大小是 msg_pool 的大
                                小 */
        RT_IPC_FLAG_FIFO);     /* 如果有多个线程等待，按照先来
                                先得到的方法分配消息 */

if (result != RT_EOK)
{
    rt_kprintf("init message queue failed.\n");
    return -1;
}

rt_thread_init(&thread1,
               "thread1",
               thread1_entry,
               RT_NULL,
               &thread1_stack[0],
               sizeof(thread1_stack),
               THREAD_PRIORITY, THREAD_TIMESLICE);
rt_thread_startup(&thread1);

rt_thread_init(&thread2,
               "thread2",
               thread2_entry,
               RT_NULL,
               &thread2_stack[0],
               sizeof(thread2_stack),
               THREAD_PRIORITY, THREAD_TIMESLICE);
rt_thread_startup(&thread2);

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(msgq_sample, msgq_sample);

```

9.3 编译、仿真运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 msgq_sample 命令启动示例应用，示例输出结果如下：

```

\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh > msgq_sample

```



```

msh >thread2: send message - A
thread1: rcv msg from msg queue, the content:A
thread2: send message - B
thread2: send message - C
thread2: send message - D
thread2: send message - E
thread1: rcv msg from msg queue, the content:B
thread2: send message - F
thread2: send message - G
thread2: send message - H
thread2: send urgent message - I
thread2: send message - J
thread1: rcv msg from msg queue, the content:I
thread2: send message - K
thread2: send message - L
thread2: send message - M
thread2: send message - N
thread2: send message - O
thread1: rcv msg from msg queue, the content:C
thread2: send message - P
thread2: send message - Q
thread2: send message - R
thread2: send message - S
thread2: send message - T
thread1: rcv msg from msg queue, the content:D
message queue stop send, thread2 quit
thread1: rcv msg from msg queue, the content:E
thread1: rcv msg from msg queue, the content:F
thread1: rcv msg from msg queue, the content:G
...
thread1: rcv msg from msg queue, the content:T
thread1: detach mq

```

使用 **SystemView** 工具可以监测示例实际运行过程，示例开始之后现象与实验设计相同。整体流程如下图所示，初始化部分细节见第二张图。

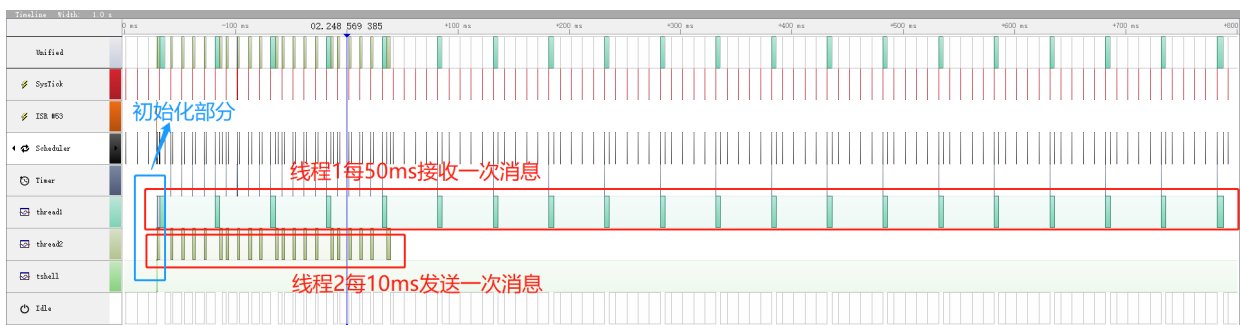


图 9.2: 运行整体图

初始化部分细节：



图 9.3: 初始化部分细节

图中各名称对应描述如下表:

名称	描述
Unified	CPU 当前运行状态
UART ISR	串口中断
SysTick	系统时钟
Scheduler	调度器
Timer	定时器
thread1	线程 thread1
thread2	线程 thread2
tshell	线程 tshell
Idle	空闲线程

9.4 附件

整个示例运行流程可以使用工具 **SystemView** 工具打开附件文件 [msgq_sample.SVDat](#) 查看具体细节。注意打开附件时，不要有中文路径。

第 10 章

实验：动态内存堆的使用

10.1 实验目的

- 理解动态内存的基本原理
- 在 RT-Thread 中熟练使用动态内存

10.2 实验原理及程序结构

动态堆管理根据具体内存设备划分为以下三种情况，本实验针对于第一种情况，前提是要开启系统 heap 功能。

第一种是针对小内存块的分配管理（小堆内存管理算法），小内存管理算法主要针对系统资源比较少，一般用于小于 2MB 内存空间的系统。

第二种是针对大内存块的分配管理（slab 管理算法），slab 内存管理算法则主要是在系统资源比较丰富时，提供了一种近似多内存池管理算法的快速算法。

第三种是针对多内存块的分配情况（memheap 管理算法），memheap 方法适用于系统存在多个内存堆的情况，它可以将多个内存“粘贴”在一起，形成一个大的内存堆，用户使用起来会感到格外便捷。

10.2.1 实验设计

本实验使用的例程为：[dynmem_sample.c](#)

实验设计一个动态的线程，这个线程会动态申请内存并释放，每次申请更大的内存，当申请不到的时候就结束。

10.3 源程序说明

10.3.1 示例代码框架

RT-Thread 示例代码都通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

10.3.2 示例源码

以下定义了线程所用的优先级、栈大小以及时间片的宏。

```
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5
```

线程入口函数，一直申请内存，申请到之后就释放内存，每次会申请更大的内存，申请不到时，将结束，申请的内存大小信息也会打印出来。

```
/* 线程入口 */
void thread1_entry(void *parameter)
{
    int i;
    char *ptr = RT_NULL; /* 内存块的指针 */

    for (i = 0; ; i++)
    {
        /* 每次分配 (1 << i) 大小字节数的内存空间 */
        ptr = rt_malloc(1 << i);

        /* 如果分配成功 */
        if (ptr != RT_NULL)
        {
            rt_kprintf("get memory :%d byte\n", (1 << i));
            /* 释放内存块 */
            rt_free(ptr);
            rt_kprintf("free memory :%d byte\n", (1 << i));
            ptr = RT_NULL;
        }
        else
        {
            rt_kprintf("try to get %d byte memory failed!\n", (1 << i));
            return;
        }
    }
}
```

动态内存管理的示例代码，创建 `thread1` 并启动。并将函数使用 `MSH_CMD_EXPORT` 导出命令。

```
int dynmem_sample(void)
{
    rt_thread_t tid = RT_NULL;

    /* 创建线程 1 */
    tid = rt_thread_create("thread1",
                           thread1_entry, RT_NULL,
```

```
        THREAD_STACK_SIZE,  
        THREAD_PRIORITY,  
        THREAD_TIMESLICE);  
  
    if (tid != RT_NULL)  
        rt_thread_startup(tid);  
  
    return 0;  
}  
  
/* 导出到 msh 命令列表中 */  
MSH_CMD_EXPORT(dynmem_sample, dynmem sample);
```

10.4 编译、运行和观察示例应用输出

编译工程，然后开始仿真。使用控制台 UART#1 做为 msh 终端，可以看到系统的启动日志，输入 dynmem_sample 命令启动示例应用，示例输出结果如下：

```
\ | /  
- RT - Thread Operating System  
/ | \ 3.1.0 build Aug 24 2018  
2006 - 2018 Copyright by rt-thread team  
msh > dynmem_sample  
msh > get memory :1 byte  
free memory :1 byte  
get memory :2 byte  
free memory :2 byte  
...  
get memory :16384 byte  
free memory :16384 byte  
get memory :32768 byte  
free memory :32768 byte  
try to get 65536 byte memory failed!
```

例程中分配内存成功并打印信息；当试图申请 65536 byte 即 64KB 内存时，由于 RAM 总大小只有 64K，而可用 RAM 小于 64K，所以分配失败。