CM32M4xxR PMP RTOS StackOverflow监测应用样例说明

Create Time	@July 21, 2021 12:00 PM
End Time	
Start Date	@July 21, 2021
Tags	已完成

0.功能说明

该应用样例展示了如何利用PMP的内存隔离特性，针对RTOS的改造实现任务栈的栈溢出检测，从而保障漏洞任务无法干扰的其他正常任务的执行；

1.运行环境

Name	版本
软件开发环境	NucleiStudio
RTOS	RT-Thread 3.0.4

2.外设资源

Name	配置	描述
串口	UART5（TX-PE8, RX-PE9）	115200-8-1-n

3.功能描述

该应用例程中共启动了“**green**”，“**red**”两个测试任务：

- "**green**"任务

> 💡 周期每隔1秒打印“green task is running”
> 10秒后触发陷阱函数，执行无限递归调用

- ”**red“**任务

> 💡 周期每隔1秒打印“red task is running”

由于本应用样例中针对RTOS进行了改造，特别针对任务切换过程中的任务堆栈添加的PMP写入保护，使得在"**green**"任务在触发递归调用，造成任务栈溢出后能够进行异常捕获；并在异常捕获后将任务从调度队列中删除，避免了整个系统的崩溃；

运行日志如下：

```
\ | /
- RT -     Thread Operating System
 / | \     4.0.3 build Jul 21 2021
 2006 - 2021 Copyright by rt-thread team
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
green task is running
red task is running
enter recursive call, trigger stack overflow
PMP Exception Occures Run at 0x2000069c, Fetching Memory at 0x2001907c
!!!
########################################################
!!!!!!!!!thread green stack overflow!!!!!!!!!!!!!!!!!!thread green terminated!!!!!!!!!
########################################################
red task is running
red task is running
red task is running
red task is running
```

4.示例设计说明

4.1 任务栈设计

由于PMP对保护地址有对齐要求，并且CM32M4xxR目前的PMP最小保护单元为128bytes，特此设计了示例任务栈的管理模块。

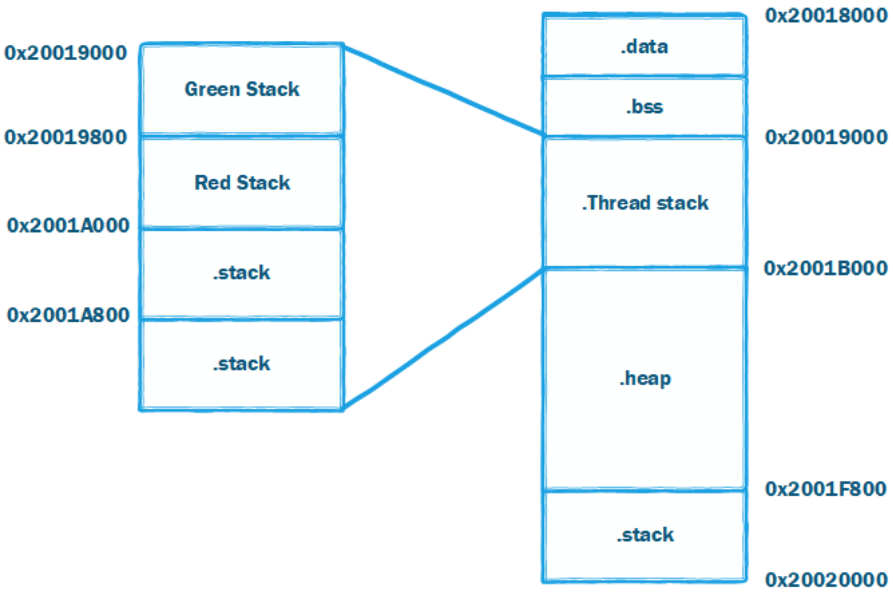
该模块在.bss段中开辟了共8KB的静态数组，定位在0x20019000地址；并实现了为创建任务分配任务栈空间地址，保障任务栈空间地址对齐；

```
MEMORY
{
  ilm (rxai!w) : ORIGIN = __ILM_RAM_BASE, LENGTH = __ILM_RAM_SIZE
  ram (wxa!ri) : ORIGIN = __RAM_BASE, LENGTH = __RAM_SIZE

  tstack(wxa!ri) : ORIGIN = 0x20019000, LENGTH = 0x2000
}

.tstack_ram :
  {
    . = ALIGN(4);
   KEEP( *(.tstack))
   KEEP( *(.tstack.*))
    . = ALIGN(4);
  } > tstack
```

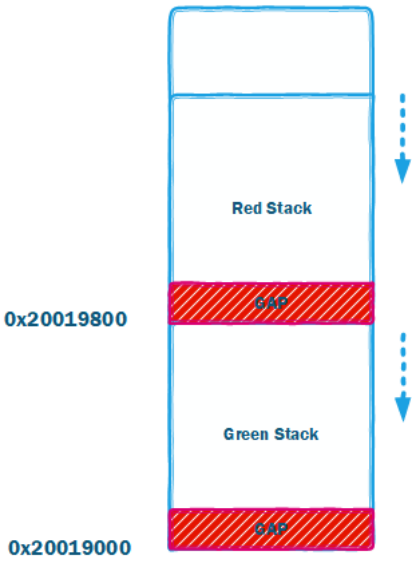
若采用RT-Thread的动态堆空间分配创建任务栈空间，可能造成栈空间地址非对齐，从而造成PMP配置失效；
或用户可选择修改RT_ALIGN 为128字节对齐，来实现地址对齐，但将会造成空间资源浪费；
本示例中主要展示PMP在RTOS上的应用，故采用简单方式进行实现，其他栈空间对齐实现，本示例中将不做展开；



4.2 PMP配置

CM32M4xxR采用向下生长栈结构，为保障每个任务栈不会出现溢出，为此在每个栈的底部添加一块128Bytes的缓存空间，并将该区域设置为只读区域。

当任务调用栈出现溢出后，SP将进入GAP区域，该区域的PMP属性为ReadOnly，因此将触发PMP的非法访问异常，CM32M4xxR栈信息如下图：



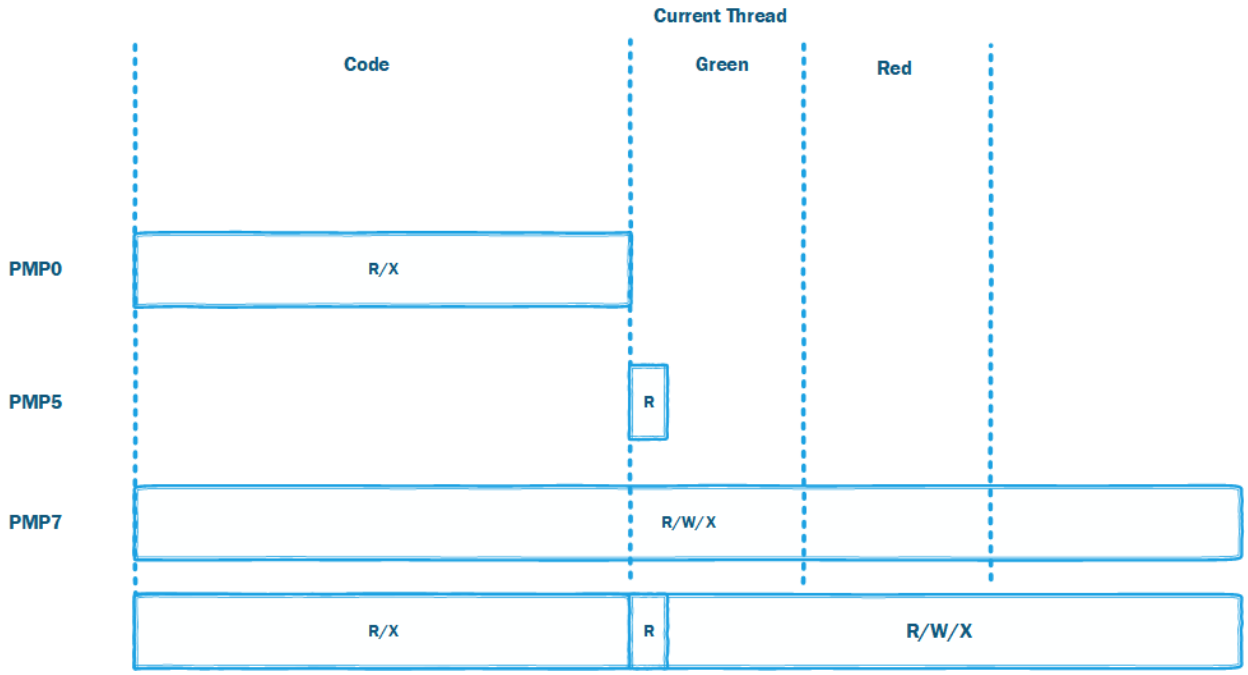
在RT-Thread启动初期，在板级初始化时，定义了一个背景PMP配置项（PMP7）支持所有空间的读写权限；同时Code区配置为可读、可执行权限；

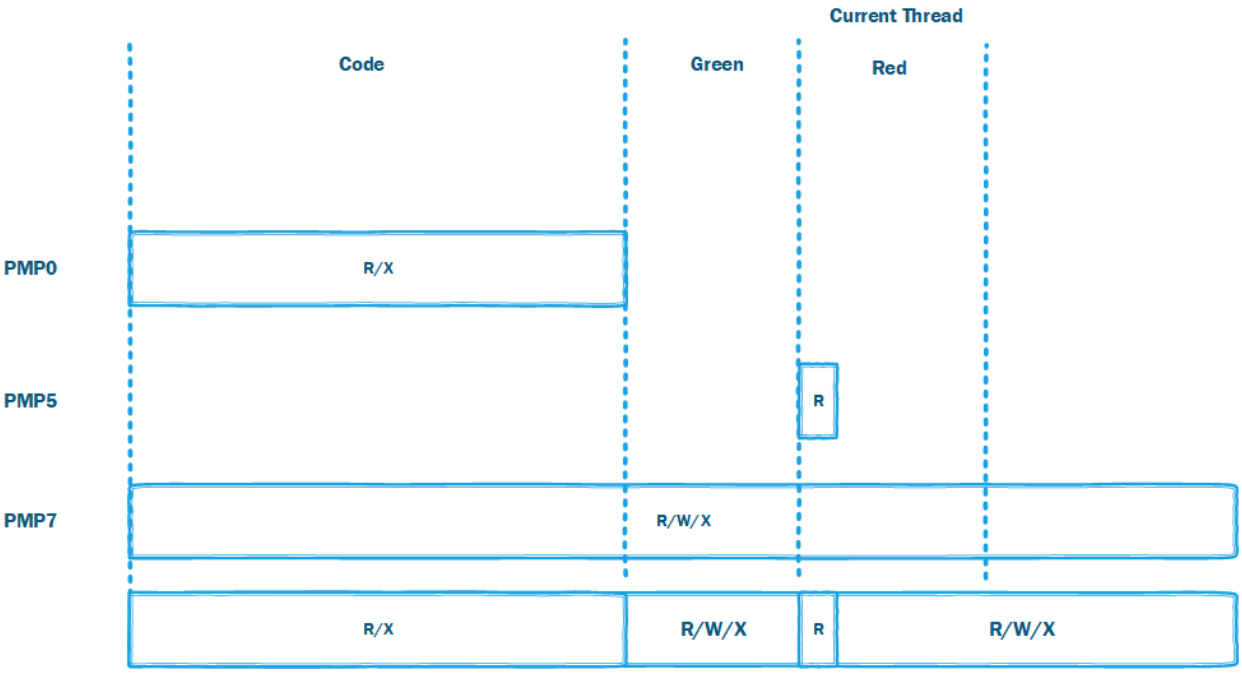
💡 RISC-V中PMP的配置匹配区域支持重叠覆盖，PMP的序号越低优先级越高；

💡 同时由于RT-Thread当前运行在Machine Mode下，默认PMP的配置项对MachineMode不起作用，为此修改Mstatus的MPRV位1，生效MachineMode的PMP匹配；

```
/* Initial CSR MSTATUS value when thread created */
#define RT_INITIAL_MSTATUS  \
              (MSTATUS_MPP | MSTATUS_MPIE | MSTATUS_FS_INITIAL | MSTATUS_MPRV)
```

在进行任务调度时，获取目标任务的栈配置起始地址，并将该目标地址的128byte空间设置为只读权限，用于栈溢出识别；在任务调度时PMP配置示意图如下：






```
rt_stack_mpu_hook(1, to_thread->stack_addr, to_thread->stack_size);

void rt_stack_mpu_hook(rt_uint8_t flag, rt_uint8_t * stack_addr, rt_uint32_t stack_size)
{
  PMP_Region_InitTypeDef pmp_init;

  pmp_init.Number = PMP_REGION_NUMBER5;
  pmp_init.Enable = PMP_REGION_ENABLE;                  // Enable Configuration
  pmp_init.Lock = PMP_REGION_UNLOCK;                    // Configuration Lock, may not modify unless reset, and also match in Machine Mode
  pmp_init.BaseAddress = stack_addr;       //Seting Thread Stack
  pmp_init.Size = PMP_REGION_SIZE_128B;                  //Setting Thread Stack GAP
  pmp_init.AddressMatching = PMP_ADDRESS_MATCHING_NAPOT; //Setting PMP Size to NAPOT mode -> 2^n
  pmp_init.AccessPermission = PMP_ACCESS_R;           //Setting array permission is Read Only

  PMP_ConfigRegion(&pmp_init);
}
```

4.3异常处理

在green任务出现递归调用后，将触发PMP非法访问异常，并进入在PMP配置时注册的异常处理函数，在该接口函数中，用户可选择对系统进行恢复性操作；例如删除任务，修改PC指针等。本示例中将打印溢出的任务名称，并将线程从调度队列中删除；保障其他线程不被干扰；

```
//PMP Exception Handler
void PMP_Exception_Handler(unsigned long mcause, unsigned long sp) {
  rt_kprintf("PMP Exception Occures Run at 0x%lx, Fetching Memory at 0x%lx\r\n!!!", __RV_CSR_READ(CSR_MEPC), __RV_CSR_READ(CSR_MBADADDR));

  rt_interrupt_enter();
  static rt_thread_t fault_thread = RT_NULL;
  rt_thread_t thread = rt_thread_self();
  rt_kprintf("\r\n#########################################################\r\n", thread->name);
  rt_kprintf("!!!!!!!!!thread %s stack overflow!!!!!!!!!", thread->name);
  rt_thread_detach(thread);
  rt_kprintf("!!!!!!!!!thread %s terminated!!!!!!!!!", thread->name);
  rt_kprintf("\r\n#########################################################\r\n", thread->name);
  fault_thread = thread;
  rt_schedule();
```

```
    rt_interrupt_leave();
}
```