**#PendSV中的工作内容（汇编）：**

1. 关全局中断，用MRS指令加载PSP指针的指向地址到R0

2. 判断是不是第一次执行，如果第一次执行的话就直接切换下文

3. 保存上文：

1.R0的值-32（因为8个寄存器自动入栈）

2.使用STM命令保存R4-R11寄存器。

3.更新当前TCB的栈指针（不是指PSP指针）

4. 切换下文：

1.把当前任务的控制块指针也应更新为新任务控制块的指针。

2.提取新任务TCB中的栈顶地址给R0

2.使用LDM命令恢复R4-R11寄存器。

3.R0增加32B后（修正后的新栈顶）赋值给PSP

5.对LR寄存器的位2置位，确保程序返回时使用PSP指针

6.开中断并返回

**#使用OS前后的SP使用情况**

不使用OS时：只用到MSP（中断和非中断都使用MSP）；

使用OS时：main函数和中断使用MSP， 各个Task（线程）使用PSP（即任务栈）；

**#栈大小如何选**

最低栈大小为64B，因为CM3一共有16个寄存器要保存，每个寄存器占4B。一般情况下栈的使用率不应超过70%。空余的部分一方面是为了防止一些极端的意料不到的情况，另一方面，为添加新功能提供一定方便。

**#为什么用PendSV完成切换：**

主要是延迟切换，避免中断因抢占被耽误的问题。PendSV是一个可悬挂的可编程中断，通常将其优先级设为最低，触发PendSV后将被自动挂起，待所有中断全部响应后它才响应。因为任何芯片都不允许在中断活跃时进入线程模式如切换任务，否则fault伺候。假如，CPU在正在一个执行一个DMA中断，此时响应一个SysTick中断，如果我在SysTick进行任务切换，必将导致SysTick返回后无法继续执行DMA中断，从而耽误DMA中的相关工作，这是RTOS所不能容忍的，这也是为什么每次进出中断都需要告诉uCOS，因为任务切换时不允许有中断嵌套。可以使用Systick来替代PendSV在这里的功能吗？非常不推荐！SysTick异常的设计初衷是定时节拍的响应，它应能快速响应，如果把优先级降至最低，会和其他需要时钟节拍的任务引发伦理冲突。请使用设计者专门提供的PendSV异常

**#为什么入栈时先入栈xPSR、PC（R15）、LR（R14）、R12再入R3-R0？难道R4-R11就是下等公民？**

首先，先入xPSR、PC、LR是为了更早的预读服务指令（可以理解为跳转到中断服务函数而跳转函数会改变这几个寄存器的值）；R12、R3-R0是由一套C函数调用标准所约定的，这一约定使得我们可以用C语言来编写中断服务函数（因为R3-R0寄存器被约定作为子函数的参数，如R0用于存储子函数的第一个输入参数也用于存储子函数的返回值，如果输入参数多于4个，剩余参数由栈来保存或者由编译器帮忙使用更多的寄存器，至于R12，编译器另有他用，具体何用暂时未知）。

**#LR（R14）寄存器的作用**

LR是链接寄存器，用于CPU在进入子函数前保存PC指针的值，届时方便返回继续执行。使用BL（Branch and Link转移并链接）跳转指令，会自动把PC计数器填入LR寄存器。但是在中断中，LR不再保存PC指针值将被重新解释为EXC\_RETURN，用于切换返回后特权级/用户级、使用PSP还是MSP指针，ARM/Thumb指令集切换等，可以理解为一个控制寄存器。虽然被重新解释，BX LR指令依然可以用于中断的返回。

**# 工作模式：thread模式/handler模式、用户级/特权级**

两个模式的本意，是用于区别普通应用程序的代码和异常服务例程的代码——包括中断服务例程的代码。在线程模式＋用户级下，许多访问会受限，如读写特殊功能寄存器。利用这种特性可以确保内核的稳定运行不被破坏。当处理器处在线程状态下时，既可以使用特权级，也可以使用用户级；另一方面，handler模式总是特权级的。在复位后，处理器进入线程模式＋特权级。

**#CONTROL寄存器作用**

堆栈指针选择、用户级开关（用户级的关闭必须触发异常进入特权模式才能关闭，通常是触发SVC中断）

**#ARM指令与Thumb指令切换**

对PC指针的LSB写1/0表示在Thumb/ARM指令集模式。CM3不支持ARM指令，虽然可写0但是会触发异常

**#进出临界区函数原理**

保存PRIMASK寄存器值并关闭中断、恢复PRIMASK寄存器值并开启中断。PRIMASK是1bit中断屏蔽寄存器，置1时关闭除FAULT和NMI外所有中断。简单理解：保存先前的中断开关状态并关闭总中断，届时在恢复。（另外还有FAULTMASK也是特殊功能寄存器，它也是1Bit寄存器，置1后仅开启NMI中断）（CPSID I关中断指令等价给PRIMASK置1）

**#栈最大使用率统计原理**

在程序初始化前，将栈清0，随着程序运行，栈数组会填入数据也就成为非0值，最后统计连续0值占总栈大小的百分比。

**#CPU使用率统计原理**

在所有任务开始运行前，先只开启空闲任务，并运行一个统计周期，计算在该统计周期内空闲任务被运行的总次数。此时启动其他任务，以后只需要统计空闲任务的执行次数占之前总次数的百分比即可得出CPU使用率。

**#上下文切换函数OSCtxSw和中断的上下文切换函数OSIntCtxSw有什么区别**

对于STM32，没有区别，他们都是触发PendSV，依靠PendSV的可悬挂特性实现上下文切换。对于无类似PendSV可悬挂中断的芯片来说，他们区别**可能**在于，中断中切换上下文时已经自动入栈了一部分数据，因此OSIntCtxSw可以少操心一部分数据。

**#什么时候需要关全局中断？**

语句/事务不应被打断，但又需要多个指令才能执行完毕的时候。

**#MSP和PSP？**

一个是主栈指针，一个是进程栈指针。复位后是使用MSP指针，PSP则需要手动开启。使用两个指针的初衷是为了安全，保证用户代码崩溃的时候，不至于使内核或者操作系统崩溃。因为**异常和中断都是使用MSP指针**，因此可以有机会给崩溃的代码擦屁股处理后事。各种Fault异常的目的也是为了处理后事。

**#抢占式和不可抢占式区别？**

不可抢占式的只有在任务主动放弃CPU使用权时，其他任务才可以执行。一个优点是可以使用不可重入函数。抢占式的则是一旦有高优先级任务就绪则会发生任务切换。

**#优先级反转**

一个低优先级任务占用一个独享资源时，此时一个高优先级任务也需要该资源，会因为得不到该任务而被挂起，此时一个中等优先级任务就绪，它不需要使用独享资源，所以会将低优先级任务挂起，中等优先级任务开始运行，但是此时高优先级任务还在挂起中，此时中等优先级任务优于高优先级任务执行，优先级发生反转。

解决办法：首选优先级继承法，继承使用该资源的最高优先级。候选优先级天花板法，两个算法的区别是，后者需要找到所有任务中最高优先级的，遍历任务表要花时间。而前者只需要将独占资源的任务优先级提升为希望抢占它的任务的优先级

**#邮箱、消息队列、信号、事件组、标志**

邮箱传递单值信息，消息队列可以看作邮箱组，缺点在于无法传递0

信号传递多值标志，标志传递二值标志、事件组可以看作标志组。标志是特殊的信号。

ucos的这些信号状态都在TCB中的任务状态变量里，一个bit对应一种通讯方式，只有都处于复位状态时才可能进入就绪态

邮箱/消息队列都是通过提供指针来传递数据和非空检测

信号则是计数值的改变

**#何时可以用普通信号量替代互斥型信号量**

**互斥信号量的不同处就是内建了优先级继承机制**，因此如果**没有任务对共享资源的访问有截止时间**，则可以使用普通信号量。（假设低优先级任务L占据打印机，此时高优先级任务H开始执行，一段时间后也需要占据打印机，H由于得不到打印机被迫挂起，如果使用互斥信号量，接下来执行的任务必定是L，如果是普通信号量，则可能是L和H之间的其他优先级任务被执行。H得到打印机的时机将无法预估）

**#如何避免死锁**

死锁：两个任务无限等待对方控制的资源。

避免：让每个任务**按统一的顺序**获取到**全部资源**后再做下一步工作，同时请求信号量应**设超时时间**

**#SVC的作用**

通常OS和APP是隔离开的，如果APP需要通知内核做事情，一般就是通过SVC异常来完成。ucos没有使用SVC，一些OS的APP就会通过SVC通知OS进行任务调度或其他特殊请求

**#何时调度/切换任务？**

在RTOS时，只有**就绪进程集合发生变动**时才有调度/切换需要，变动只会在以下情况发生：

1. 当前进程自动放弃CPU或运行受阻（Delay、MsgPend）

2. 新建进程、唤醒某进程（TaskCreat、TaskUnsuspend、MsgPost）

3. 进程自杀或被杀（TaskDel）

系统心跳抵达、中断异常的返回会使用调度器但不一定会做任务切换！

**#何时不可调度？**

1. 执行临界区代码（进程切换、关键变量修改时都需要进入临界区）

2. 中断响应期间

RTOS必须努力缩小不可调度窗口

**#任务调度算法**

任务调度函数只有一个功能，选取当前优先级最高的就绪任务。但是核心算法就是如何快速的找到这个最高的就绪任务。粗糙的方法是按优先级遍历TCB链表，但是耗时是不确定的，最明显的缺陷就是，随着任务数量增加，任务调度时间线性增加，好的算法则将其变为常数。

ucos的算法不难，主要是非常巧妙。

**一句话总结：**将64个优先级分为8组OSRdyGrp，每组8个优先级OSRdyTbl，先找出优先级最高的组，然后从这个组中找出最高的优先级。是通过查表法OSUnMapTbl[256]来找出最高的组（或组内最高的优先级）。而OSUnMapTbl[256]就是将0x00-0xff每个数据中最低位为1的位数一一列举出来

图解：

如图，有3个就绪的任务，首先查询OSRdyGrp中最低位1所在的位置，从图中可以看出是第2组，对应OSRdyGrp的bit2处，然后在该组（也就是OSRdyTbl[2]）中查询最低位1的位置，从图中可以看出最低位的1位于bit5处，因此最高的优先级即为2\*8+5=21，让我们从右往左数一下，到第2组bit5处是不是正好是21。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
| 0组 |  |  |  |  |  |  |  |  |
| 1组 |  |  |  |  |  |  |  |  |
| 2组 |  | 1 | 1 |  |  |  |  |  |
| 3组 |  |  |  |  |  |  |  |  |
| 4组 |  |  |  |  |  | 1 |  |  |
| 5组 |  |  |  |  |  |  |  |  |
| 6组 |  |  |  |  |  |  |  |  |
| 7组 |  |  |  |  |  |  |  |  |

知识背景：

在ucos里，任务最多64个，用1字节中的6个bit即可表示，将其分为2组，每组3bit，高3bit组为Y组，低3bit为X组。ucos还有一个char型就绪表OSRdyTbl[8]，8个元素代表8个组，每个组有8bit，1个bit表示一个优先级，正好可表示64个优先级，当一个任务就绪后，会在表中对应bit位写1。还有一个char型变量OSRdyGrp用来表示哪组存在就绪任务，1个bit表示一个组，一个组中的8个任务其中任何一个任务就绪都会将OSRdyGrp中对应的bit置1。通常OSRdyGrp存储Y组信息，OSRdyTbl[8]存储X组信息。

置任务为就绪态代码：

OSRdyGrp |= ptcb->OSTCBBitY;

OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;

其中ptcb任务控制块的内容在初始化时计算完毕

ptcb->OSTCBY = (INT8U)(prio >> 3);

ptcb->OSTCBX = (INT8U)(prio & 0x07);

ptcb->OSTCBBitY = (INT8U)(1 << ptcb->OSTCBY);

ptcb->OSTCBBitX = (INT8U)(1 << ptcb->OSTCBX);

通过下面的算法即可在常数时间内寻找到最高优先级

找到最高优先级的核心代码：

y = OSUnMapTbl[OSRdyGrp];

OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);

OSUnMapTbl[256]是一个表，就是将0x00-0xff每个数据中最低位为1的位数一一列举出来，它存的是下标对应最低位的1在第几个位置，比如下标3存的数应该是0，因为3d=11b，又比如下标4应该为2，应为4d=100b。一来相当于把低优先级任务标志给复位，二来又把OSTCBBitX转换回了OSTCBY，得到OSTCBX和OSTCBY就可以写出OSPrioHighRdy值。现在可以看代码了，OSUnMapTbl[OSRdyGrp]表示高Y组最低位的1的位置（也就是高Y组中最高的优先级），OSUnMapTbl[OSRdyTbl[y]]表示先提取OSRdyTbl[y]高Y组对应的X组，然后找到X组中最高优先级的任务。

一个更加神仙的方法，使用CLZ（计算前导0的数目）、RBIT（位逆序），如上图OSRdyGrp=0x04，OSRdyTbl[2]=0x60, 使用如下汇编代码：

\_\_asm int \_\_rt\_ffs(int value)

{   CMP     r0, #0x00 ;r0是0的话不必比较直接返回

   BEQ     exit ;不等于跳转exit

   RBIT    r0, r0 ;逆序r0

   CLZ     r0, r0 ;求r0前导0

exit

   BX      lr}; 返回子程序

传入OSRdyGrp即可得到2，因此8乘2得到16，这个2也将作为OSRdyTbl[]的索引值，即索引结果0x60传入函数，嘿嘿，就得到5，加上16就得到21。这个逻辑比C语言版本更容易理解、更加暴力。

实用代码：

OS\_GetHighRdyPrio

LDR R0, =OSRdyGrp

LDR R0, [R0]

RBIT R0, R0 ;求OSRdyGrp

CLZ R0, R0

MOV R1, R0 ;保存就绪组到R1

LDR R0, =OSRdyTbl ;求就绪位

ADD R0, R1

LDR R0, [R0]

RBIT R0, R0

CLZ R0, R0

MOV R3, #8

MUL R1, R3

ADD R0, R1

BX LR

**#入栈顺序**

xPSR，PC，LR，R12，R3-R0，R11-R4

**#.s启动文件基本工作**

定义堆栈大小、初始化中断向量表（初始化跳转函数地址）、预先写一些弱函数形式的异常，做默认异常处理函数（复位函数：调用时钟初始化、调用main函数；NMI异常、Fault异常）

**#ucos启动函数基本工作OSStart()**

1. 寻找最高优先级就绪任务
2. 将当前任务TCB更新为刚刚找到的就绪任务
3. 初始化PendSV、PSP指针，并触发PendSV

**#ucos调度函数基本工作OS\_Sched()**

1. 在无嵌套的情况下（带PendSV的芯片可能无需考虑嵌套？）
   1. 寻找最高优先级就绪任务
   2. 更新当前任务控制块为刚才找到的就绪任务
   3. 触发PendSV进行上下文切换

**#ucos进中断函数基本工作OSIntEnter()**

1. 中断嵌套数+1

**#ucos退出中断函数基本工作OSIntExit()**

1. 中断嵌套层-1
2. 无嵌套时，寻找最高优先级任务并触发PendSV中断（任务调度）

**#ucos空闲任务工作OS\_TaskIdle()**

1. 统计执行次数（也可以进入休眠态）

**#ucos延时函数基本工作OSTimeDly()**

1. 复位当前任务就绪态
2. 设置当前任务延时节拍数
3. 调度一个新任务

**#ucos心跳SysTick中断基本工作**

1. 所有任务延时节拍-1，并设延时数为0的任务为就绪态（未考虑信号量挂起等）
2. 退出中断函数OSIntExit()
   1. 中断嵌套层-1
   2. 无嵌套时，寻找最高优先级任务并触发PendSV中断以切换上下文（任务调度）

**#ucos任务控制块初始化函数基本工作OS\_TCBInit()**

1. 初始化栈指针、优先级、延时节拍、任务状态、上下TCB链、优先级位图索引、栈统计信息、任务间通信Flag等

**#ucos创建一个任务基本工作OSTaskCreate()**

1. 初始化任务堆栈
2. 初始化TCB
3. 将TCB加入TCB列表
4. 进行任务调度

**#ucos删除一个任务基本工作OSTaskDel()**

1. 复位就绪态
2. 将TCB从TCB列表中删除
3. DeInit TCB
4. 进行任务调度

**#简易操作系统的设计**

1. 上下文切换：在PendSV中进行上下文切换，使用汇编完成
2. 任务调度器：使用优先级位图法来寻找最高优先级任务，与ucos不同，这也使用汇编完成
3. 系统节拍：对任务中的延时节拍数减1，并把延时节拍为0的任务设为就绪态，最后触发任务调度
4. 延时函数：取消任务就绪态并挂起并设置延时节拍，最后触发任务调度

**#ARM和MCU的区别**

ARM基本上只是一个MPU，相当于CPU，还需要自己加RAM、FLASH等设备才能组成一个系统，MCU则集成了这些外设。另外MCU没有MMU，在安全性、灵活性等方面不及ARM。

遗留问题：

#ucos如果关闭了中断嵌套层数的记录，而此时又恰好发生了中断嵌套，ucos在每个中断结束前都应调用OSIntExt()，该函数除了对嵌套层数减1，更重要的是进行任务切换。也就是说如果在中断嵌套时强行执行任务切换函数会引起异常吗？还是依靠PendSV的优越性，上下文切换过程会自动悬挂到所有中断响应完成后才开始执行？