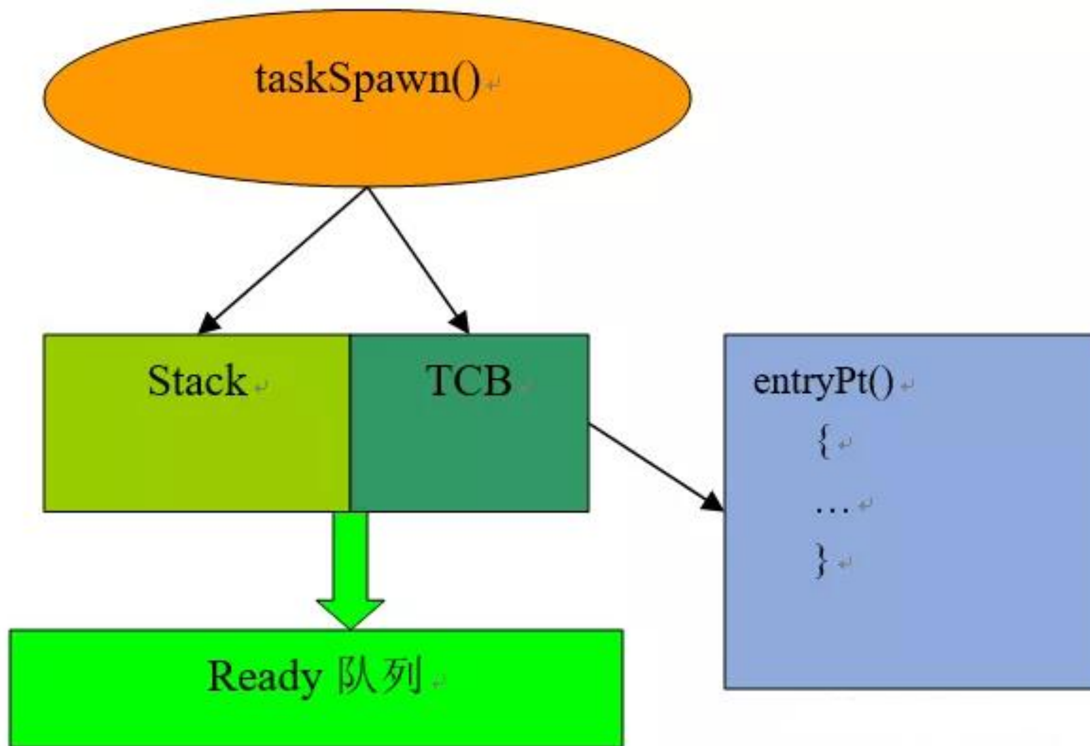
VxWorks 下任务的创建

本文我们聊一聊 VxWorks 系统里如何创建任务。
与任务相关的 API 由系统库 taskLib 提供。

常用的函数是 taskSpawn()，32 位系统里(以下同)函数原型如下：

```
01: TASK_ID taskSpawn(char *name, int priority, int options,
02:        size_t stackSize, FUNCPTR entryPt, int arg1, in
03:        int arg4, int arg5, int arg6, int arg7, int arg8, int arg9, int arg10);
```

Task 可以看作是 Stack 和 TCB 组成的。因此 taskSpawn()的第一步就是为 Stack 和 TCB 分配内存，然后初始化它们，最后将这个任务放入 Ready 队列。



这个函数的参数比较多，我们挨个看一下
name:任务的名字，主要只是开发者在 Shell 中调试时才使用它。名字中的字符和长度没有什么限制，甚至可以指定为 NULL，那么系统会自动给它分配为 t*N*，*N* 是从 1 开始递增的十进制数。甚至不同任务的名字可以是相同的，所以通过名字来管理任务的话，要注意一下。而操作系统或应用程序在管理任务时，通常使用的是它们的 ID。名字和 ID 可以通过这两个函数互相转换

```
01: char *taskName (TASK_ID tid);
02: TASK_ID taskNameToId (char *name);
```

priority:

任务的优先级，VxWorks 调度任务时就是基于它。优先级的取值范围是 0-255，可以动态改变或查询

```
01: STATUS taskPrioritySet(TASK_ID tid, int newPriority);
02: STATUS taskPriorityGet(TASK_ID tid, int *pPriority);
```

注意：多个任务间，优先级的高低是相对的。假如系统里只有两个任务 A 和 B，优先级分别是 1 和 10，或者分别是 1 和 100，这两种情况下的调度是一模一样的。都是高优先级的会抢占，必须等它退出 Ready 队列，低优先级的才有可能占用 CPU。不像有的操作系统采用分时调度，优先级的高低会影响时间片的长短。

那在我们自己的应用程序里，优先级设置为多少合适呢？很多人习惯于将内核任务设置为 100，用户态任务的稍低一些，150 或 200，这个并没有什么限制，只要平衡好多个应用任务之间的关系即可。不过建议应用任务的优先级不要高于系统任务的。例如，在《Task 之常见系统任务》里，我们提到过 WDB 任务的优先级默认为 3，所以我们应用任务尽量不要高于 4，否则可能会影响到调试了。

options

任务选项，以 Bit 为单位，每个版本的选项不尽相同，因此代码中尽量使用选项的宏定义，而**不要使用对应的数值**。例如 6.9 里可用的选项如下。

l VX_FP_TASK 使用浮点协处理器，否则含有浮点操作的任务在切换时不会保存浮点寄存器的值，导致浮点异常。特例是 C++语句被编译后，会产生浮点指令，因此任务中**调用 C++语句**时，就一定要使能这个选项。

l VX_NO_STACK_FILL 不填充 Stack

l VX_NO_STACK_PROTECT 不提供 Stack 的上溢和下溢保护

l VX_ALTIVEC_TASK 使用 Altivec 指令集(仅 PowerPC)

l VX_SPE_TASK 使用 SPE 引擎(仅 PowerPC)

l VX_DSP_TASK 使用 DSP(仅 SuperH)

l VX_PRIVATE_ENV 支持私有环境变量(6.9 中实际已取消，仅是为了兼容)

stackSize

任务的栈，单位是 Byte，从系统内存池分配。一经分配，尺寸就**固定不变**了。如果分配的数值过大，就会增加一点点初始化时间，并浪费部分内存空间，影响倒是不大；而分配的过小，就有栈溢出的风险，这可是致命的了。因此，开发过程中，要评估一下具体的数值。那如何才能知道分配的是否合适呢？可以在 Shell 里使用 checkStack()来检查，因为创建任务时，Stack 的每个 Byte 默认被填充为 0xee，checkStack()通过检查 Stack 中 0xee 的变化来判断 Stack 的使用边界。

```
01: void checkStack(long taskNameOrId);
```

```
-> checkStack
  NAME         ENTRY         TID          SIZE    CUR   HIGH   MARGIN
------------ ------------ ---------- --------- ----- ----- ---------
tRootTask    usrRoot      0x00debf50      9216   128  2644      6572
tIsr0        isrDeferTask 0x00dfaf28      8192   152   316      7876
tJobTask     jobTask      0x00e00b10      8000   240   564      7436
tExcTask     excTask      0x005c18a0      8192   272   332      7860
tErfTask     erfServiceTa 0x00e05440      4096   240   300      3796
tLogTask     logTask      0x00e095d0      5008   336   396      4612
tNet0        ipcomNetTask 0x00e10560     10000   512  2472      7528
ipcom_syslog ipcom_syslog 0x00df3f98      6144   456   844      5300
tNetConf     ipnet_config 0x00e67650      6144   688  1516      4628
ipcom_telnet ipcom_telnet 0x00e7e718      6144   664  1548      4596
tVxdbgTask   vxdbgEventTa 0x00e84150      8192   288   348      7844
tWdbTask     wdbTask      0x00e8b250      8192   272  2236      5956
tShell0      shellTask    0x00ea2d60     65536   91
ipcom tickd  ipcom tickd  0x00e33398      6144   280   652      5492
```

不过，如果使用了选项 VX_NO_STACK_FILL，或者设置 Kernel 的配置参数 VX_GLOBAL_NO_STACK_FILL 为 TRUE，Stack 的内容就不会被填充 0xee 了，checkStack()也就没办法正常工作了。但好处是，taskSpawn()创建任务时的速度会加快一些

Components

Component Configuration

Description	Name	Type	Value
⊞ job task (default)	INCLUDE_JOB_TASK		
⊟ kernel (default)	INCLUDE_KERNEL		
Disable stack fill on all task	VX_GLOBAL_NO_STACK_FILL	BOOL	FALSE
Enable compatible mode for task p	TASK_PRIORITY_SET_COMPAT...	BOOL	FALSE
INT lock level	INT_LOCK_LEVEL		0x0
ISR stack size	ISR_STACK_SIZE		0x1000
Kernel task execution stack overf	TASK_KERNEL_EXEC_STACK_O	uint	4096
Kernel task execution stack under	TASK_KERNEL_EXEC_		
Root stack size	ROOT_STACK_SIZE		10000

entryPt

任务主函数的入口地址，可以包含 10 个 int 型参数，arg1- arg10。

如果参数不是 int 型的，可以考虑使用指针。如果参数多于 10 个，可以考虑使用结构体。如果参数的类型大于 32Bit，可以考虑使用联合体，不过要注意大小端

```
01: union doubleToInts {
02:     double doubleVal;
03:     int    intVals[2];
04: };
05:
06: int entryPtRtn (int arg1, int arg2)
07:     {
08:     double result;
09:     doubleToInts uDoubleToInts;
10:
11:     uDoubleToInts.intVals[0] = arg1;
12:     uDoubleToInts.intVals[1] = arg2;
13:
14:     result = uDoubleToInts.doubleVal;
15:     return 0;
16:     }
```

返回值

taskSpawn()的返回值就是 Kernel 分配给任务的 ID，它是一个 32Bit 的数值，是全局唯一的。不过任务退出后，这个 ID 值是可以再次分配给其它任务的，也就是可以重复使用的。任务可以通过 taskIdSelf()查询自己的 ID，通过 taskIdVerify()验证某任务是否存在，通过 taskIdListGet()获取当前的任务列表。taskLib 中很多 API 使用 task ID 为参数，这个参数取值为 0 的话，一般就是表示当前任务自己了。

```
01: TASK_ID taskIdSelf (void);
02: STATUS taskIdVerify(TASK_ID tid);
03: int taskIdListGet
04:     (
05:     TASK_ID idList[],
06:     int maxTasks
07:     );
```

另外，创建任务还可以使用 taskCreate()，它只是初始化任务，需要 taskActivate()把它放到 Ready 队列。

```
01: TASK_ID taskCreate
02:     (
03:     char *name,          /* name of new task (stored at pStackBase) */
04:     int priority,        /* priority of new task */
05:     int options,         /* task option word */
06:     size_t stackSize,    /* size (bytes) of stack needed */
07:     FUNCPTR entryPt,     /* entry point of new task */
08:     _Vx_usr_arg_t arg1, /* 1st of 10 req'd args to pass to entryPt */
09:     _Vx_usr_arg_t arg2,
10:     _Vx_usr_arg_t arg3,
11:     _Vx_usr_arg_t arg4,
12:     _Vx_usr_arg_t arg5,
13:     _Vx_usr_arg_t arg6,
14:     _Vx_usr_arg_t arg7,
15:     _Vx_usr_arg_t arg8,
16:     _Vx_usr_arg_t arg9,
17:     _Vx_usr_arg_t arg10
18:     );
19:
20: STATUS taskActivate
21:     (
22:     TASK_ID tid          /* task ID of task to
23:     );
```

这样分成两步做的好处就是可以在整个系统初始化时就把任务也初始化好，需要使用的时候，再进行激活，相当于减少了激活时间。其实 taskSpawn()就是它俩的合成

```
01: TASK_ID taskSpawn(char *name, int priority, int
02:     {
03:     TASK_ID tid = taskCreate (name, priority, opti
04:
05:     if (tid == TASK_ID_NULL)     /* create failed */
06:         return (TASK_ID_ERROR);
07:
08:     (void) taskActivate (tid); /* activate task */
09:
10:     return (tid);
11:     }
```

taskSpawn()中初始化 Stack 和 TCB 的操作就是在 taskCreate()中完成的，下图是很多年前的一段 Benchmark 数据

Tasks	
taskSpawn	24.5 µs
taskInit	18.6µs
taskActivate	0.6 µs
taskDelete	19.6 µs
taskLock:	
Lock exists	0.01 µs
taskUnlock:	
No lock exists	0.03 µs
Lock exists	0.01 µs
taskSuspend:	
Ready task	

可以看到在当时的软硬件(Pentium3)配置下，Kernel 操作的耗时一般都是微秒级的，但是 taskSpawn()比其它函数要慢很多。因此，当项目的实时性需求非常高时，可以考虑使用 taskCreate()和 taskActivate()的组合。

还有一个 POSIX 风格的 API 可以用来创建任务或者获得任务句柄，taskOpen()。这个函数多数是在支持进程时使用，因为它可以把任务创建为公共对象，以便于多进程与 Kernel 间相互访问。我们在介绍 RTP 通信时，再详细介绍它

```
01: TASK_ID taskOpen (const char *name, int priority, int options,
02:     int mode, char *pStackBase, size_t stackSize,
03:     void *context, FUNCPTR entryPt,
04:     int arg1, int arg2, int arg3, int arg4, int
05:     int arg6, int arg7, int arg8, int arg9, int arg10);
```