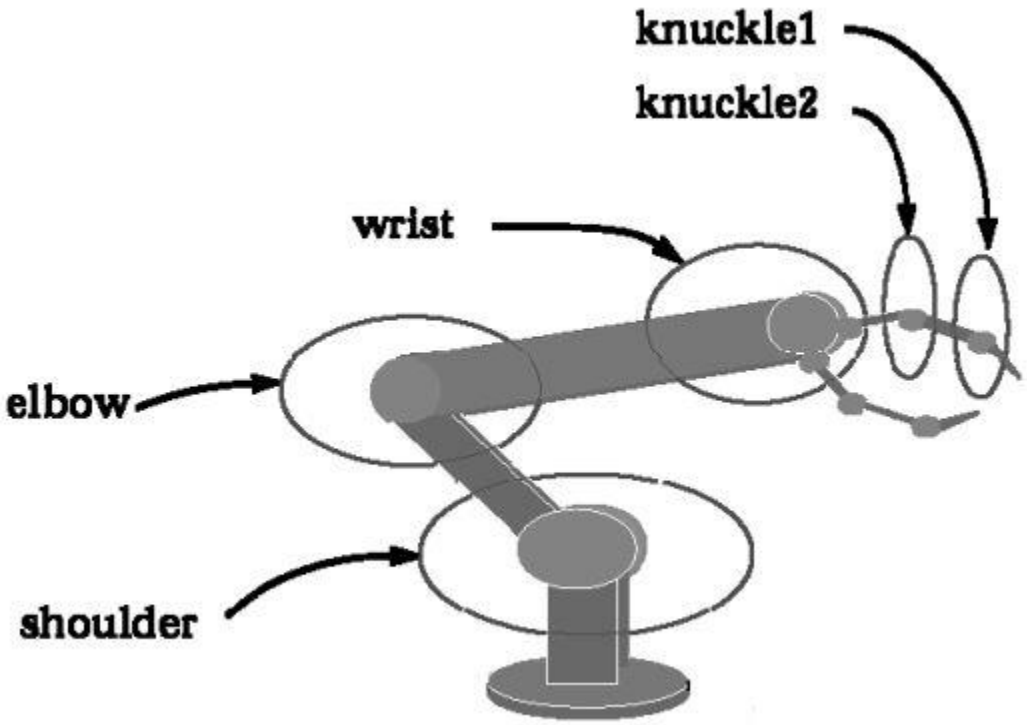
VxWorks 下的 Task 是什么

VxWorks 的 Task，也就是任务，是系统里最基本的执行单元，类似于其它操作系统的 Thread(线程)。现代的RTOS基本都通过提供一个多任务环境来支撑上层的应用程序。而上层的应用程序则使用不同的任务来模拟真实世界里各种各样的分离事件。每个任务就是一个执行线程，使用着自己的系统资源。

那如果不使用多任务呢？也就是单任务系统又是如何工作的？我们看看这个案例：假设需要实现一个控制多节机械手臂的程序。



在单任务环境下，通常的代码实现就是一个无限循环，循环里轮询每节的状态。伪代码如：

```
arm()
{
        for (;;)
        {
                if (shoulder needs moving)
                        moveShoulder( );
                if (elbow needs moving)
                        moveElbow( );
                if (wrist needs moving)
                        moveWrist( );
                ...
                ...
                ...
        }
}
```

代码结构比较简单，很容易理解，这就是它的优势。另外，因为只有一个任务，也就不需要切换任务了，CPU 就可以专注于执行这个任务。
但是，单任务的缺点也很明显。从代码来看，每一节的地位平等，没有轻重缓急的概念，而真实世界里，每一节的重要程度可能是有区别的。也就是说，这种代码结构，不具备抢占能力，每一节必须等到被轮询时，才有可能去执行。而且，每一节的执行频率也不容易控制，很多条件语句会经常出现假值判断的情况，这种代码就是在浪费 CPU 了。

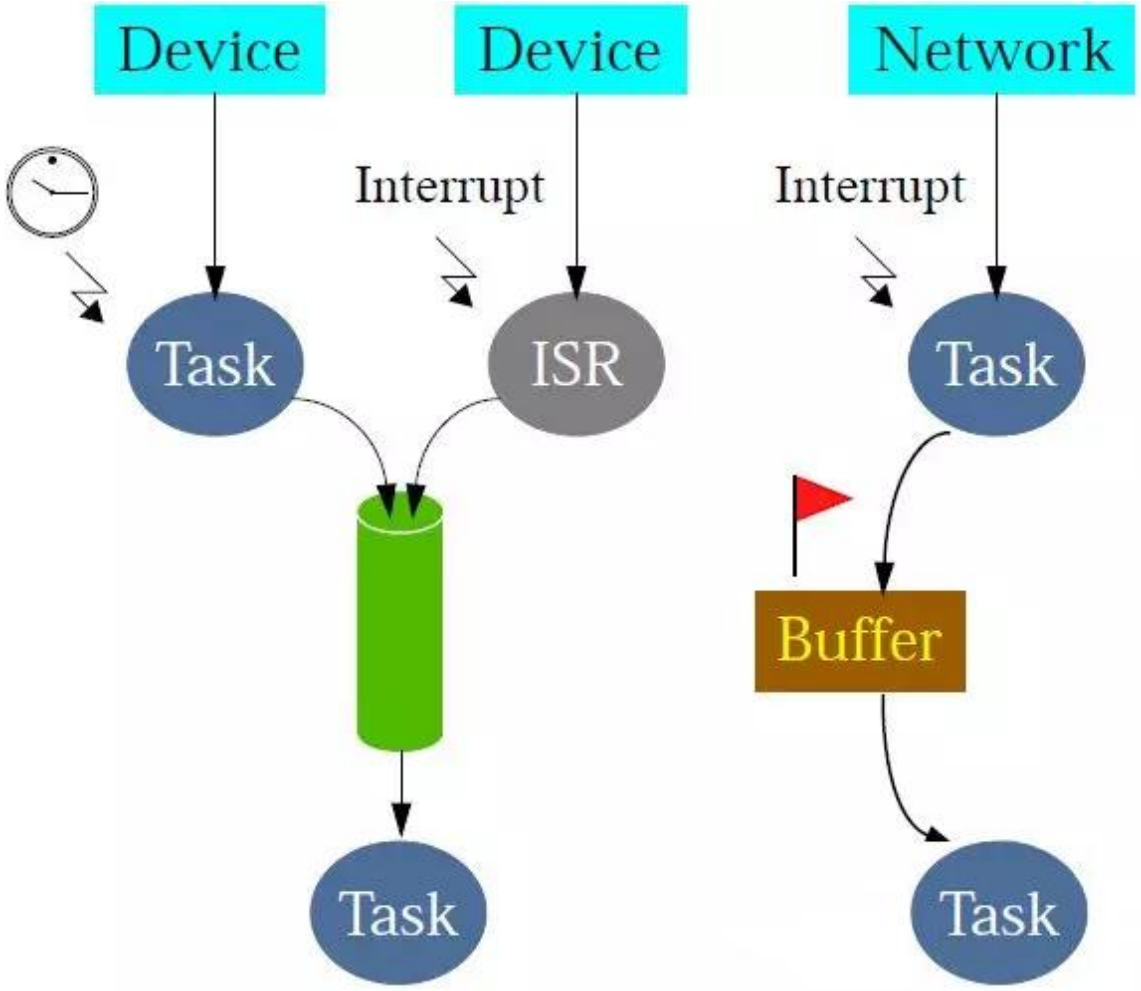
如果在多任务环境里，代码大致如下：

```
joint()
{
        for (;;)
        {
                waitForMove( );  /* Until joint needs moving */
                moveJoint( );
        }
}
```

需要给每一节制作一个任务，每个任务等待一个通知，然后才能执行，也就是说每个任务都是一直在 Waiting 和 Ready 两种状态之间切换。Waiting 表示它正在等待通知；Ready 表示它等到通知了，可以去执行了，这个时候操作系统可以决定是不是要把 CPU 分配给它了。
可以看到，这种实现对操作系统的要求就高一些了。需要操作系统提供通知机制，让不同的任务等待不同的通知。另外，如果有多个任务都等到通知了，操作系统还得提供排队机制，以便于安排哪个任务可以使用 CPU 来执行了。

这种多任务的优点有很多：
1　任意频率，每个任务的频率取决于自己被通知的频率，与其它任务关系不大；
2　可以排队，给每个任务分配不同的优先级，这样高优先级的任务就可以抢占正在执行的低优先级任务；
3　易于扩展，不同任务代码之间的耦合性比较小，增加新任务时，对已有任务的影响比较小。

正所谓，春兰秋菊,各擅胜场。而我们的真实世界，本身是一个并行的世界，许许多多事件都是同时在发生的。在模拟真实世界的软件世界里，大量的场景也是更适合于用多任务来实现，例如下图：



VxWorks 也正是采用这种多任务方式来保证它的实时性的。使用多任务把复杂问题分解，提供多种多任务通信机制来保障任务间的协作，提供高效的调度策略来保障重要任务的及时响应。这些任务可以分别独立地**等待**系统资源(包括 CPU、I/O 设备及内存空间等)，或者**使用**资源一起在宏观上并发运行。当然了，在微观上，并发运行的任务数量取决于 CPU 的内核数量。不过这不影响我们对任务的理解，关于多核的内容，我们以后会有专门的 SMP 章节，目前我们假定 CPU 都是单核的。

VxWorks 的任务由两部分组成：Stack 和 TCB。

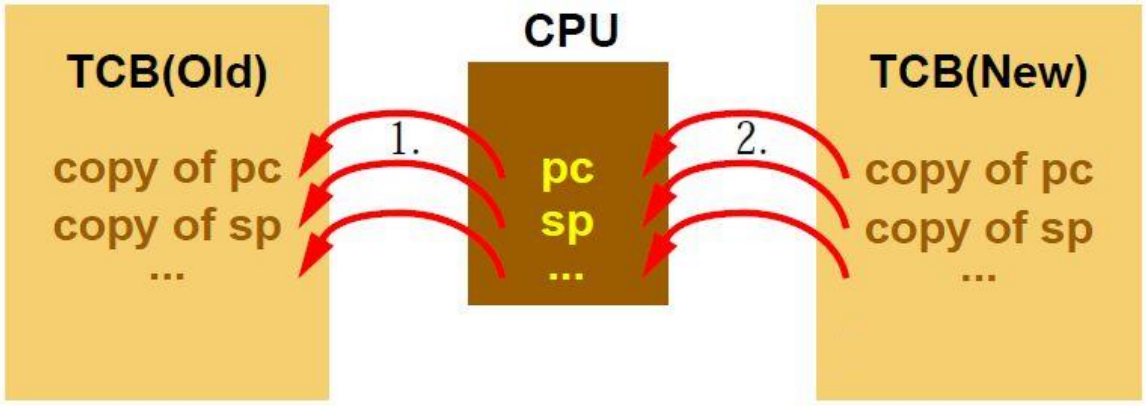
Stack(栈)用来存放任务的动态变量和函数调用关系。

TCB 全称是任务控制块(Task Control Block)，在代码中用一个庞大的结构体来表示，windTcb，在 target/h/taskLib.h 中可以看到它的完整定义。TCB 里记录的信息是任务的 Context(上下文)：

- 任务的 PC 指针
- CPU 寄存器
- 协处理器的寄存器(可选)
- 标准 IO 的分配情况
- 延时计数器
- 时间片计数器
- 内核控制结构
- 信号量回调函数
- 任务的私有环境变量
- Errno 状态
- 调试和性能监测信息
- 虚拟内存上下文(支持 RTP 时)

以及 Stack 的信息，所以也可能把 Stack 看作 Context 的一部分。

VxWorks 的调度策略进行任务调度时，就是让正在使用 CPU 执行的任务停下来，然后把 CPU 分配给另一个任务来执行。这个过程就叫做上下文切换。



可以看到，上下文切换就是对两个任务 Context 的读写操作。为了保证实时性，这个切换过程必须准确、快速。