ecos源码阅读总结

本文目的是总结ecos源码阅读过程的收获。此收获包括但不限于：

1. 以ecos为例分析理解操作系统，特别是嵌入式操作系统在完整程序中的角色
2. ecos内核调度和中断机制的分析
3. 对未来验证工作的一点思考。

本文所希望达到的效果是，读者可以在阅读之后对操作系统所必须完成的任务和其任务完成的具体技术有清晰的了解，最好能够据此完成ecos内核的重现。

不清晰的地方用红字标出，本文不断更新，欢迎提出需求和改进意见。

一、（嵌入式）操作系统必须完成的功能

对操作系统最直观的理解是：用户创建线程（进程）的执行平台。它创建、维护用户线程的运行环境，提供用户线程与操作系统和其他线程通信的支持，（系统调用，例如信号量等元件）提供硬件设备与线程之间的数据交换。为完成这些功能，操作系统必须提供的功能见“系统功能与技术实现概览.jpg”。

thread\_running→set\_need\_reschedule→reschedule→scheduler.schedule()→context switch→back to thread running ↑

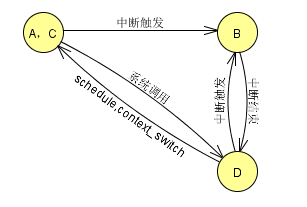
↑

interrupt coming

new thread created

lock / unlock

事实上，在操作系统上运行的代码可以分为三类：A）按部就班持续运行的代码段（thread），B）需立即执行很快结束的短代码段（ISR），C）需要尽快执行但不很快结束的代码段（DSR）。DSR可以看做是优先级最高的线程。用D表示内核代码，那么reschedule和上下文切换的场景可以用下图归纳表示：



因此，事实上我们需要考虑的调度变化，在单cpu的系统中，只有以下几种：

1. 线程被isr打断
2. dsr被isr打断
3. 系统代码被isr打断
4. isr执行完毕，系统重新调度
5. dsr执行完毕，系统重新调度
6. 线程执行完毕，系统重新调度
7. 线程进行系统调用，系统重新调度

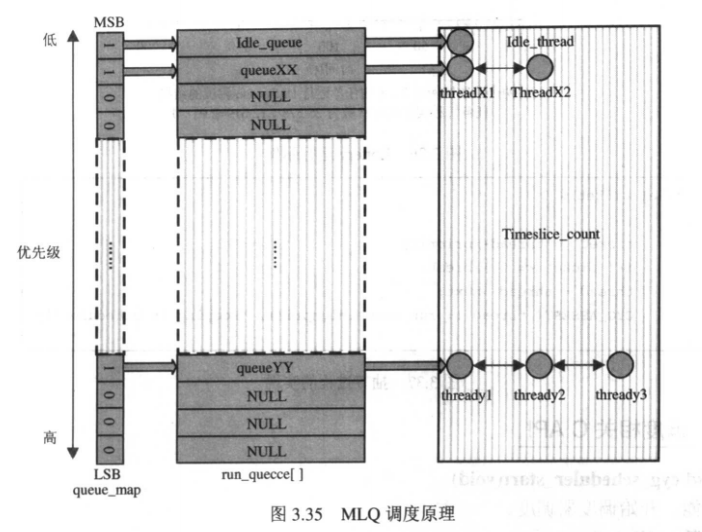
再整合一下，分以下三种情况：

1. 各种XX被ISR打断。这个要由HAL→VSR→ISR
2. XX执行完毕，系统重新调度。这个就是典型的set\_need\_reschedule, reschedule(),schedule(),context\_switch..的路线
3. 线程进行系统调用，这其中可能导致重新调度的系统调用包括：
   1. 创建线程：若更高优先级的线程被创建，则重新调度
   2. 获取锁，但被阻塞
   3. 释放锁，其他等待锁的线程被唤醒，优先级高的线程将抢占cpu。
   4. 线程主动结束，如调用yield
   5. 待补充……

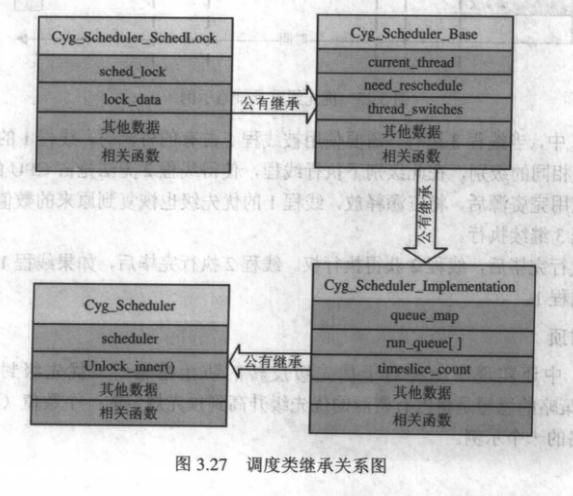
对第一种情况，cpu在执行每条指令之后都会查看中断状态字，若有中断，则保护现场，跳转至VSR，查到对应的ISR代码地址，于是跳转至ISR执行，执行完毕后挂出DSR（在DSR栈里？），返回。在这个过程中，不会为了ISR的执行单独开一个栈，因此按照时间的比率，ISR通常在用户栈中执行，较少在内核栈中执行。

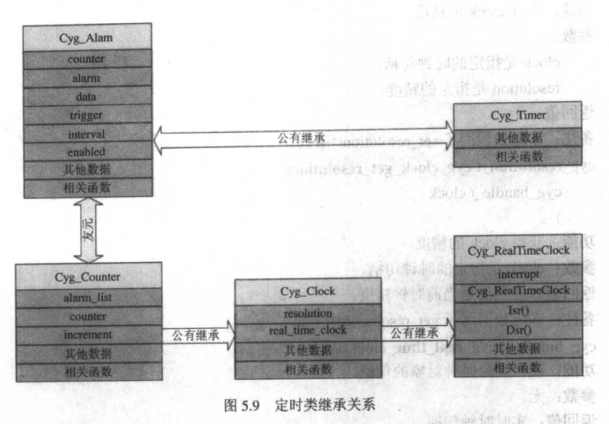
第二种和第三种可以参考已经上传的流程图。

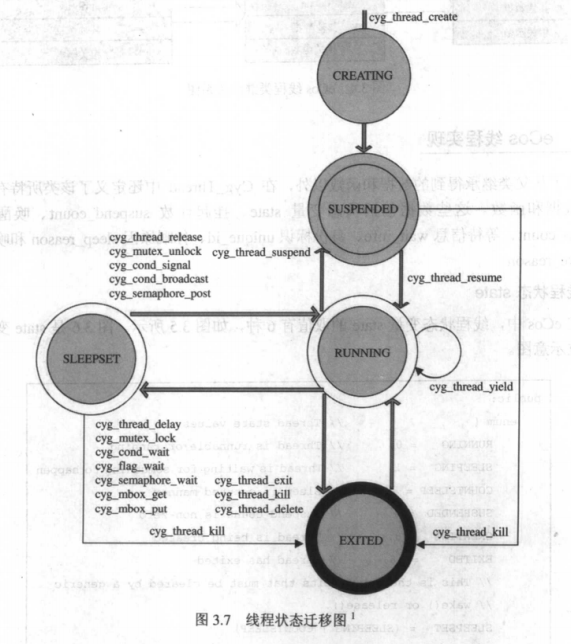
mlqueue数据结构图：



在明确了过程之后，就可以更加细致的考虑过程中的一些数据结构的维护。下面几张图说明了各个类之间的数据结构继承关系：







三、关于验证

我们所能关注的性质包括功能性、安全性。功能性包括第一张图里面提到的功能性。安全性如何说明呢？抽象出一些性质吧：

1. critical area的数据不存在共同访问共同修改的情况。
2. 内存无泄漏（在操作系统上无泄漏，用户程序若满足一定规范也无泄漏）
3. 无死锁，每次锁都有对应的解锁。
4. 不会无故异常？？？