# 线程的几种状态

|  |
| --- |
| typedef enum \_KTHREAD\_STATE {  Initialized, // 创建时  Ready, // 就绪  Running, // 运行中  Standby, // 备用  Terminated, // 结束  Waiting, // 等待  Transition, // 换入换出磁盘时  DeferredReady, // 延时就绪，少见  GateWait // 少见  } KTHREAD\_STATE; |

待会分析线程切换时，会用到这个枚举.

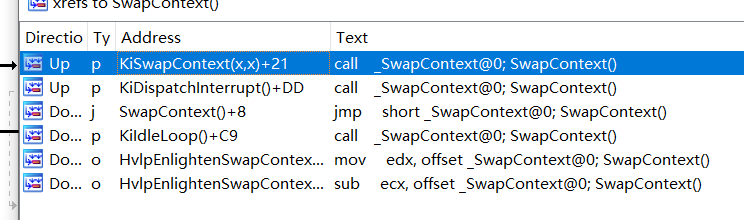
# 线程切换的两种情况

两种情况会导致线程切换：

1. 时钟中断
2. 主动切换

这次课要分析第二种，除了执行体函数Rtlxxx，其他大部分内核函数都会导致线程切换。

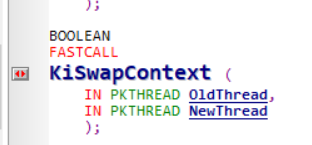
线程切换的核心函数是 SwapContext，看一下这个函数的交叉引用



比较关注 KiSwapContext，因为更上层的调用就涉及取线程的操作，属于调度算法，如果要分析线程切换的过程，从 KiSwapContext 开始分析足矣。

### KiSwapContext

从WRK得知，这个函数的原型是:



使用快速调用

Ecx: OldThread

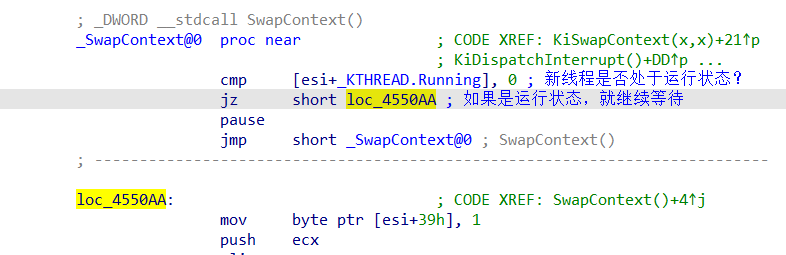
Edx:NewThread



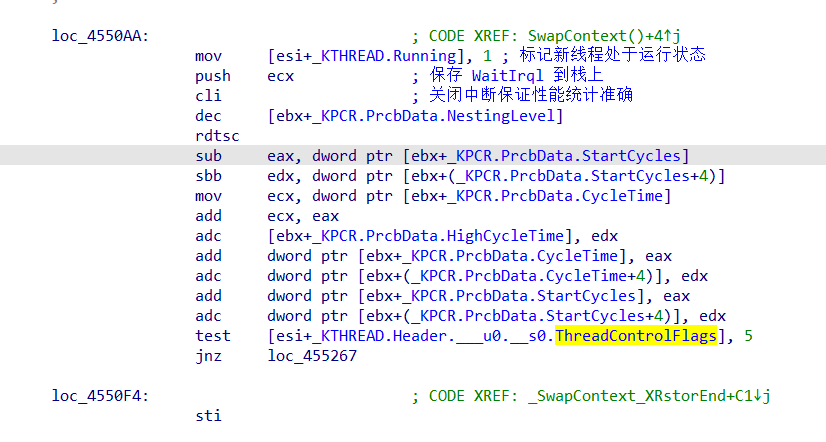
这个函数使用栈保存非易失寄存器，然后调用SwapContext切换线程，返回后，esp已经是新线程的环境，后面恢复的也是新线程的非易失寄存器。

注意此时ecx值是旧线程的WaitIrql，ebx是当前核KPCR，SwapContext要用。

### SwapContext

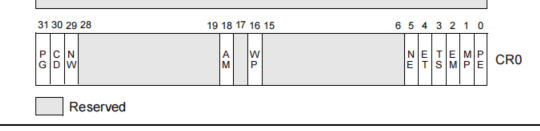


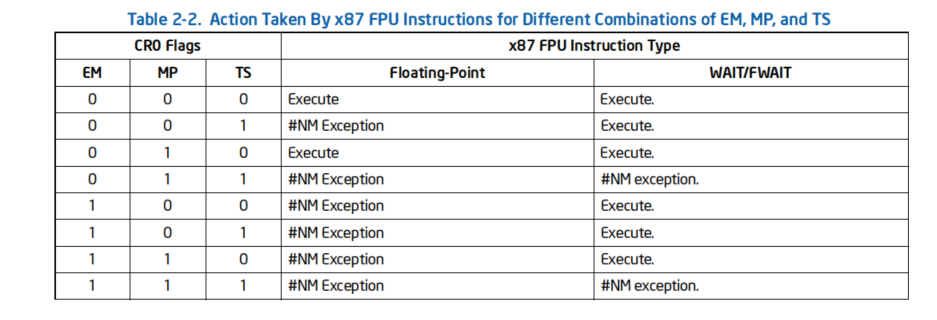
首先是一个循环，等待直到新线程不再处于运行状态



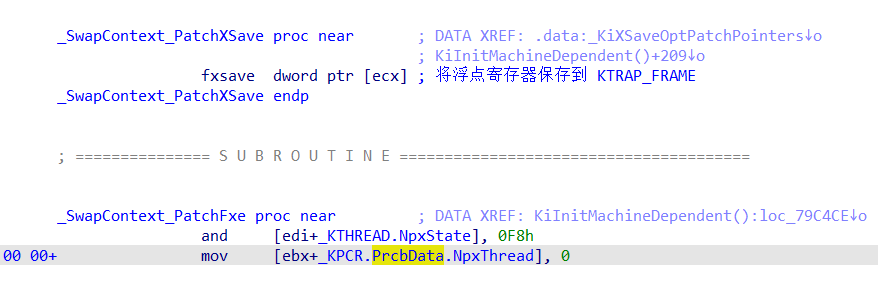
然后是性能统计，不管它。

接下来是协处理器相关，听说家用电脑基本没有这玩意，不过还是看看，万一有呢？







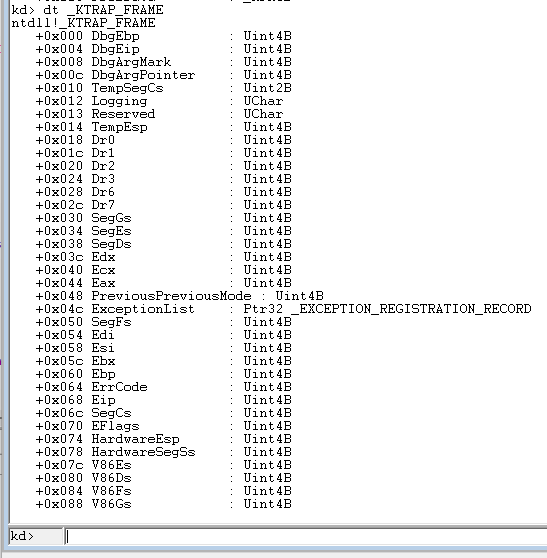


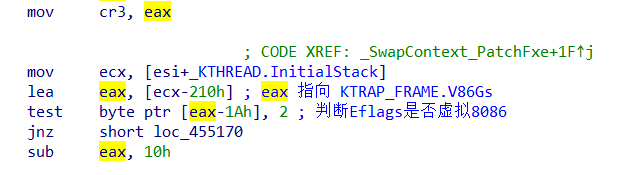
修改一下协处理器相关的字段，不知道有啥用。

接下来就是关键步骤了，切换esp，如果不属于同一个挂靠进程，还需要切换CR3：

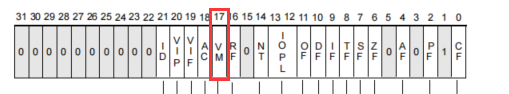


切换CR3后，开始操作KTRAP\_FRAME:

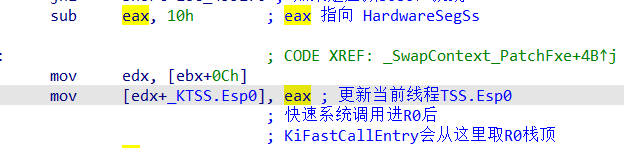




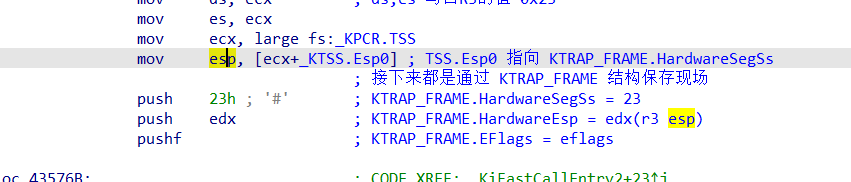
这里老是用减法，IDA没法识别，只能对着结构体看，-210h指向 KTRAP\_FRAME.V86Gs，他在+88h的位置，减去18h就是+70h的位置也就是指向了EFlags,而指令里的-1Ah就是Eflags的高2位，所以这里就是判断是否为虚拟8086模式。



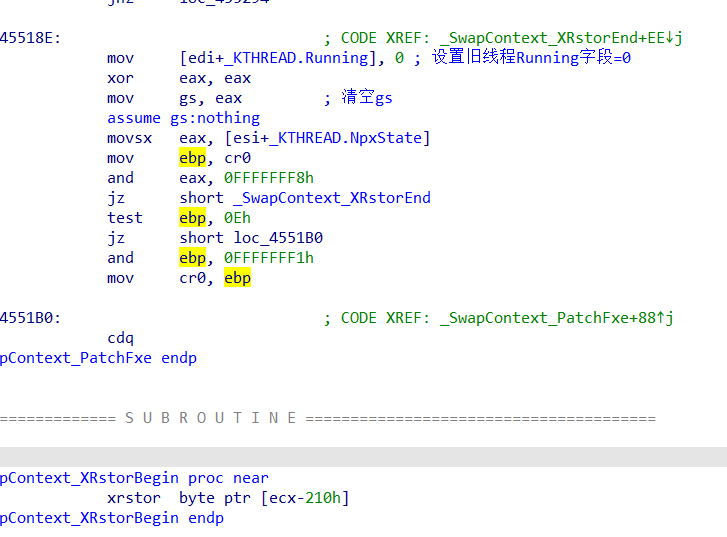
接下来，给TSS.Esp0填充值，这个字段在系统调用KiFastCallEntry里取出来用过：



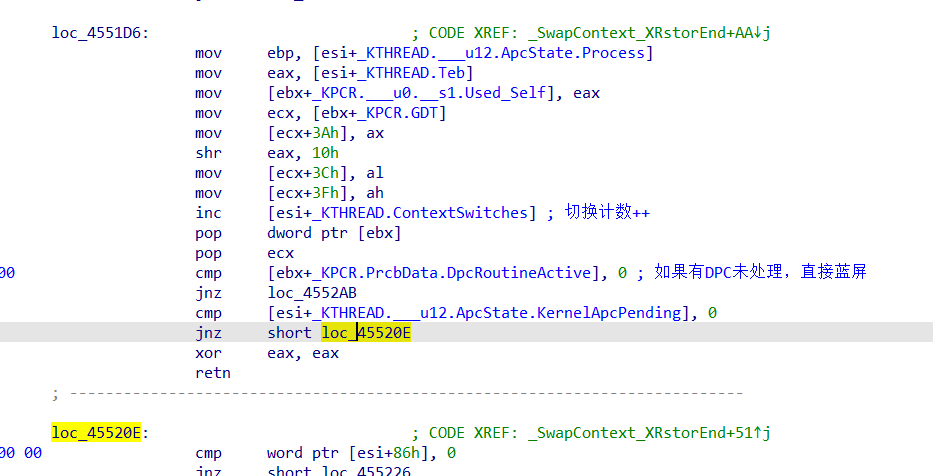
对着KiFastCallEntry看就明白了，KTSS.Esp0实际上指向KTRAP\_FRAME.HardwareSegSs



设置一下旧线程Running字段，然后是一些协处理器和浮点寄存器的操作：



如果有DPC未处理，就蓝屏，如果有APC，就继续处理，没有就返回了。



# 模拟线程切换