

**操作系统实验报告**

**实验题目**  进程的创建

**学生姓名**  付炎平

**学 号**  2019217819

**专业班级**  物联网工程19-2班

**指导教师**  田卫东

**完成日期**  2021.11.10

**合肥工业大学 计算机与信息学院**

**实验 3 进程的创建**

# 实验目的和任务要求

1. 练习使用 EOS API 函数 CreateProcess 创建一个进程，掌握创建进程的方法，理解进程和程序的区别。
2. 调试跟踪 CreateProcess 函数的执行过程，了解进程的创建过程，理解进程是资源分配的基本单位。
3. 调试跟踪 CreateThread 函数的执行过程，了解线程的创建过程，理解线程是调度的基本单位。

# 实验原理

1. 理解程序和进程的关系，进程控制块结构体以及进程创建的过程
2. CreateProcess 函数和其它与创建进程相关的函数的说明
3. 线程控制块结构体以及线程创建的过程
4. CreateThread 函数，参数和返回值的含义。

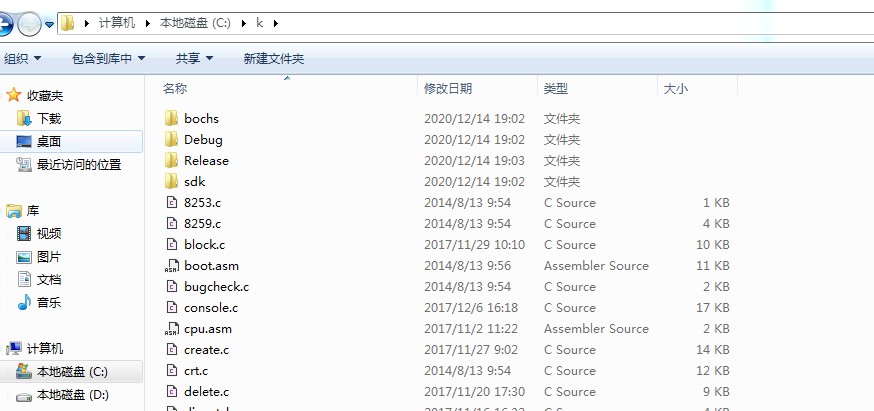
# 实验内容

## 准备实验

按照下面的步骤准备本次实验：

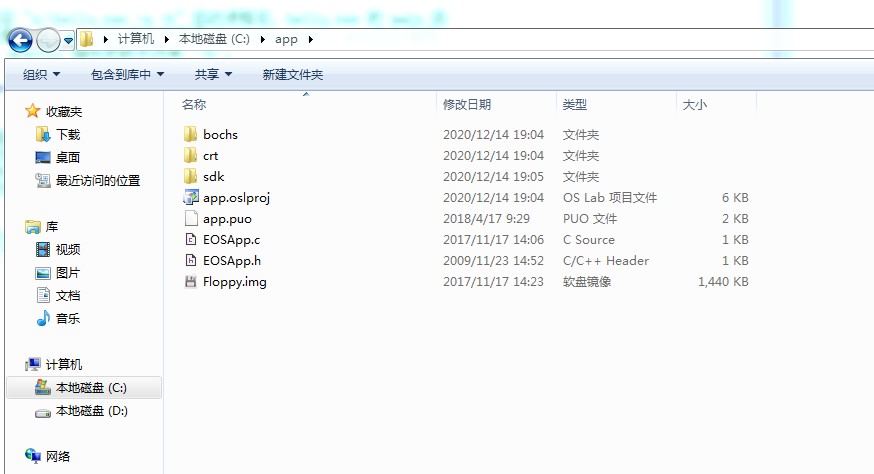
* + 1. 启动 OS Lab。
    2. 新建一个 EOS Kernel 项目。
    3. 分别使用 Debug 配置和 Release 配置生成此项目，从而在该项目文件夹中生成完全版本的 EOS SDK 文件夹。

生成结果如下：



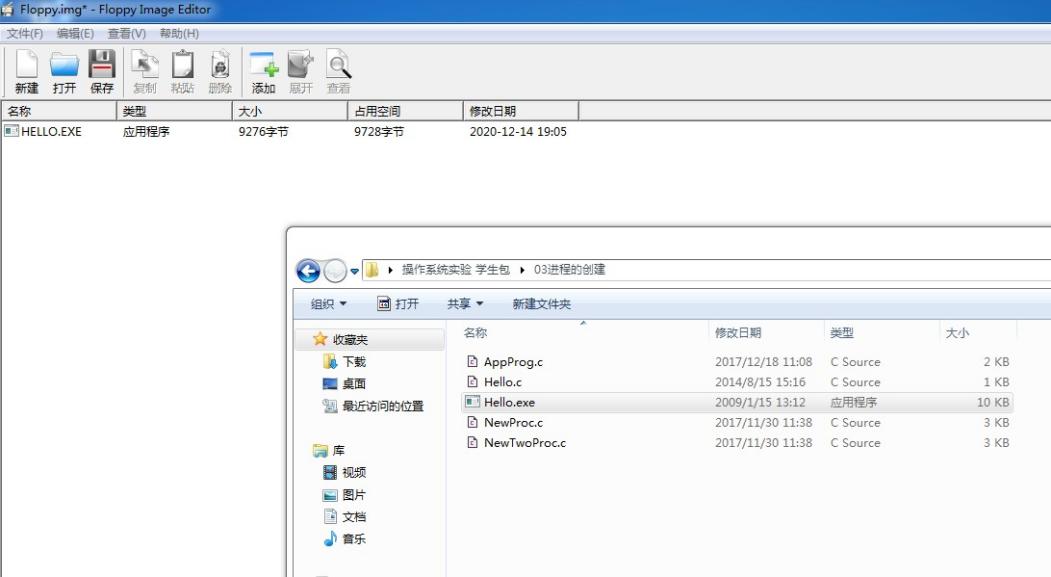
* + 1. 新建一个 EOS 应用程序项目。
    2. 使用第 3 步生成的 EOS 内核项目文件夹中的 SDK 文件夹，覆盖第 4

步新建的 EOS 应用程序项目文件夹中的 SDK 文件夹。覆盖结果如下：



## 练习使用控制台命令创建一个 EOS 应用程序的进程步骤如下：

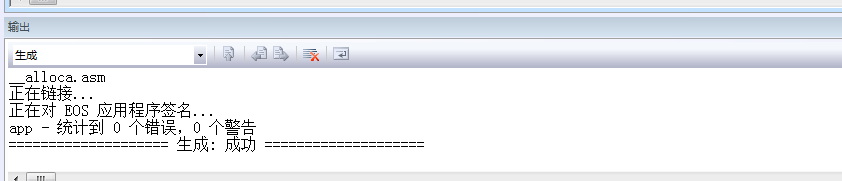
* + 1. 继续使用本实验第 3.1 节中第 4 步新建的 EOS 应用程序项目，在EOS 应用程序项目的“项目管理器”窗口中双击 Floppy.img 文件，使用FloppyImageEditor 工具打开此软盘镜像文件。
    2. 将本实验文件夹中的 Hello.exe 文件拖动到 FloppyImageEditor 工具窗口的文件列表中释放，Hello.exe 文件即被添加到软盘镜像文件中。Hello.exe是一个 EOS 应用程序的可执行文件，其源代码可以参见本实验文件夹中的Hello.c 源文件。

添加结果如下：

* + 1. 在 FloppyImageEditor 中选择“文件”菜单中的“保存” 后关闭

FloppyImageEditor。

* + 1. 按 F7 生成 EOS 应用程序项目。

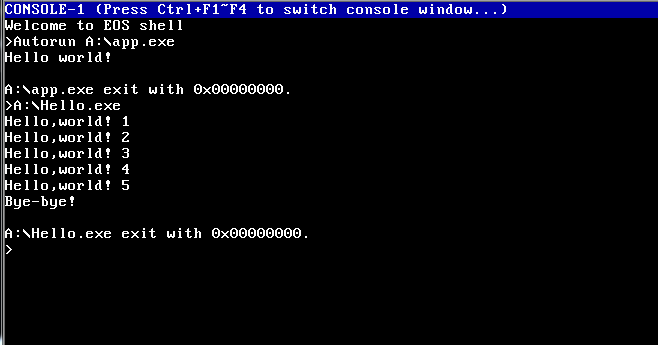


* + 1. 按 F5 启动调试。
    2. 激活虚拟机窗口，待该应用程序执行完毕后，在 EOS 的控制台中输入命令“A:\Hello.exe”后回车。

应用程序执行结果如下：

* + 1. Hello.exe 应用程序开始执行，其输出如图 11-1 所示。
    2. 待 Hello.exe 执行完毕后可以重复第 6 步，或者结束此次调试。

Hello.exe 程序执行结果如下：

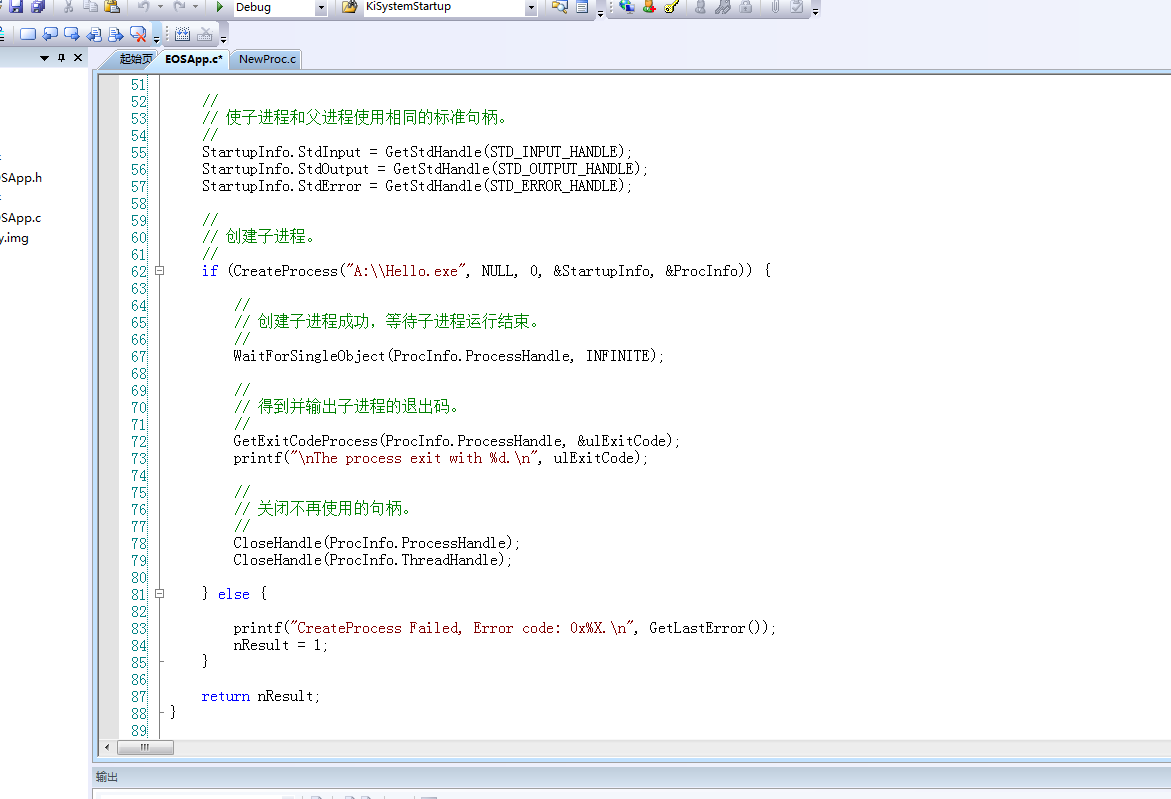


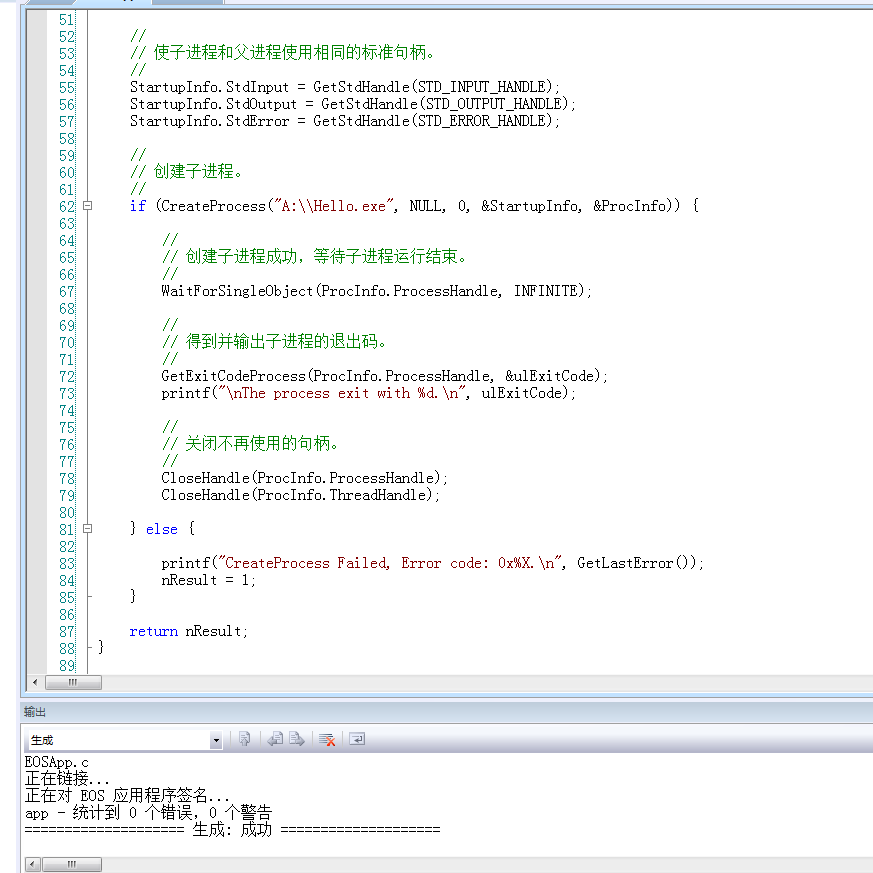
## 练习通过编程的方式让应用程序创建另一个应用程序的进程

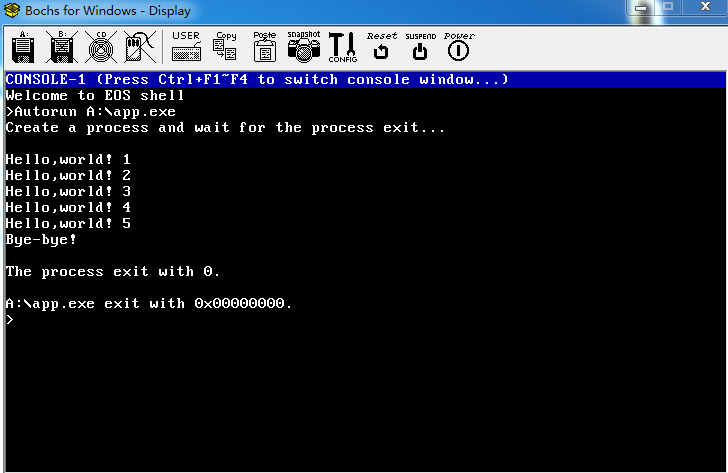
使用 OS Lab 打开本实验文件夹中的 NewProc.c 文件（将此文件拖动到 OS Lab 窗口中释放即可），仔细阅读此文件中的源代码和注释，main 函数的流程图可以参见图 11-3。按照下面的步骤查看让一个应用程序创建另一个应用程序的进程的执行结果：

* + 1. 使用 NewProc.c 文件中的源代码替换之前创建的 EOS 应用程序项目中

EOSApp.c 文件的源代码。替换过程如下：



* + 1. 按 F7 生成修改后的 EOS 应用程序项目。生成如下：
    2. 按 F5 启动调试。
    3. 激活虚拟机窗口查看应用程序输出的内容，如图 11-2 所示。结合图 11- 2，可以看到父进程（EOSApp.exe）首先开始执行并输出内容，当父进程创建了子进程（Hello.exe）后，子进程开始执行并输出内容，待子进程结束后父进程再继续执行。
    4. 结束此次调试。

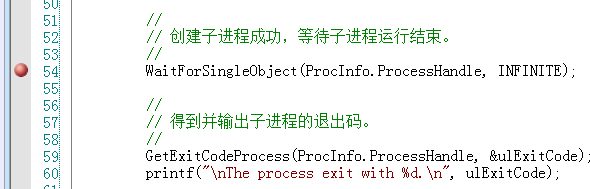


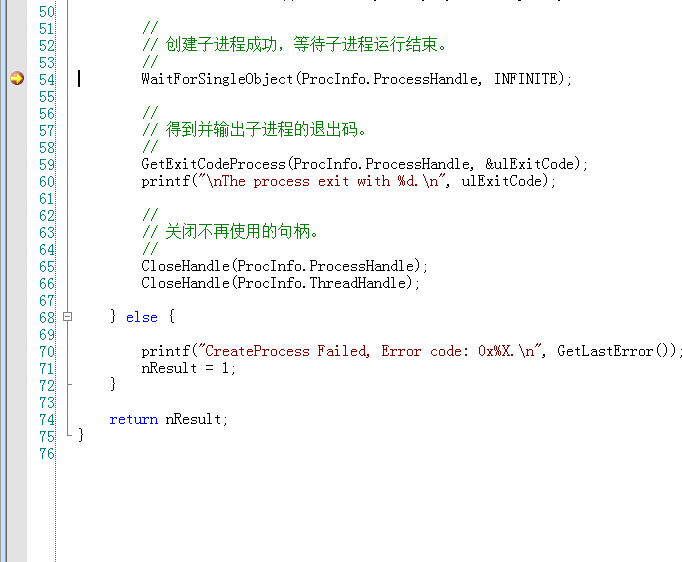
## 从应用程序的角度理解进程的创建过程

按照下面的步骤调试，从应用程序的角度理解进程的创建过程：

* + 1. 打开 EOS 应用程序的 EOSApp.c 文件 ，在 main 函数中调用

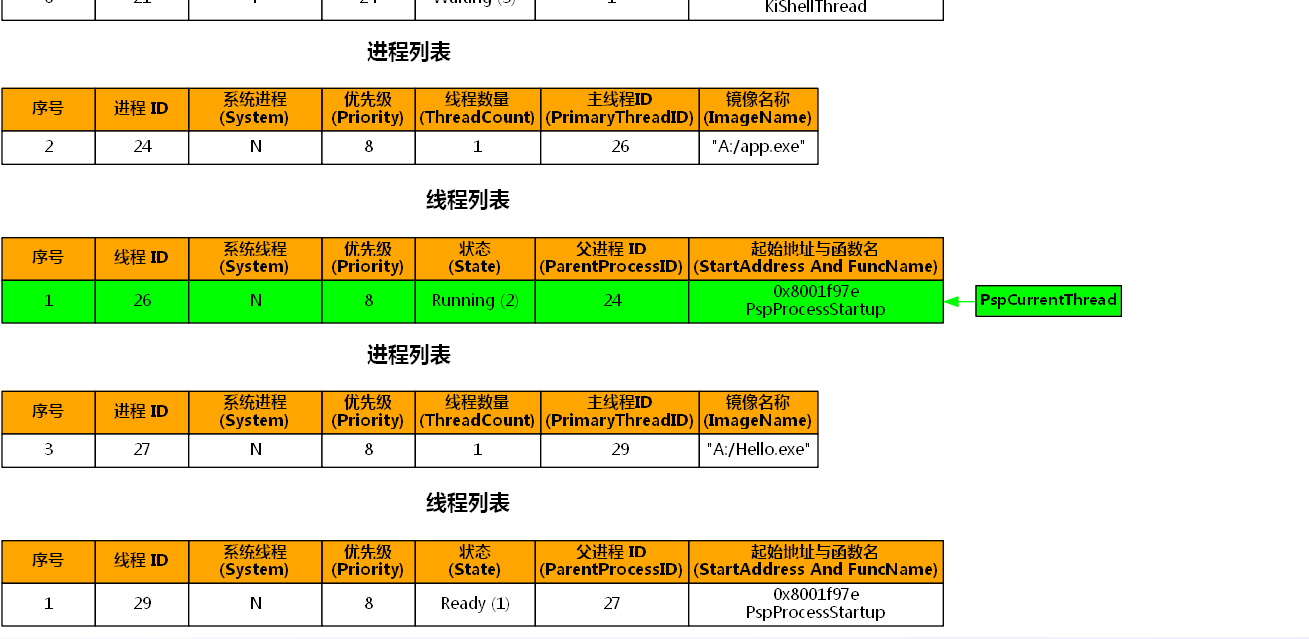
WaitForSingleObject 函数的代码行（第 54 行）添加一个断点。添加断点如下：



* + 1. 按 F5 启动调试，会在刚刚添加的断点处中断。调试状态如下：
    2. 选择“调试”菜单“窗口”中的“进程线程”菜单打开“进程线程”窗口，点击此窗口工具栏上的“刷新”按钮。可以看到当前系统中有三个进程，其中第一个是系统进程，镜像名称为空；另外两个是刚刚创建的应用程序的进程，

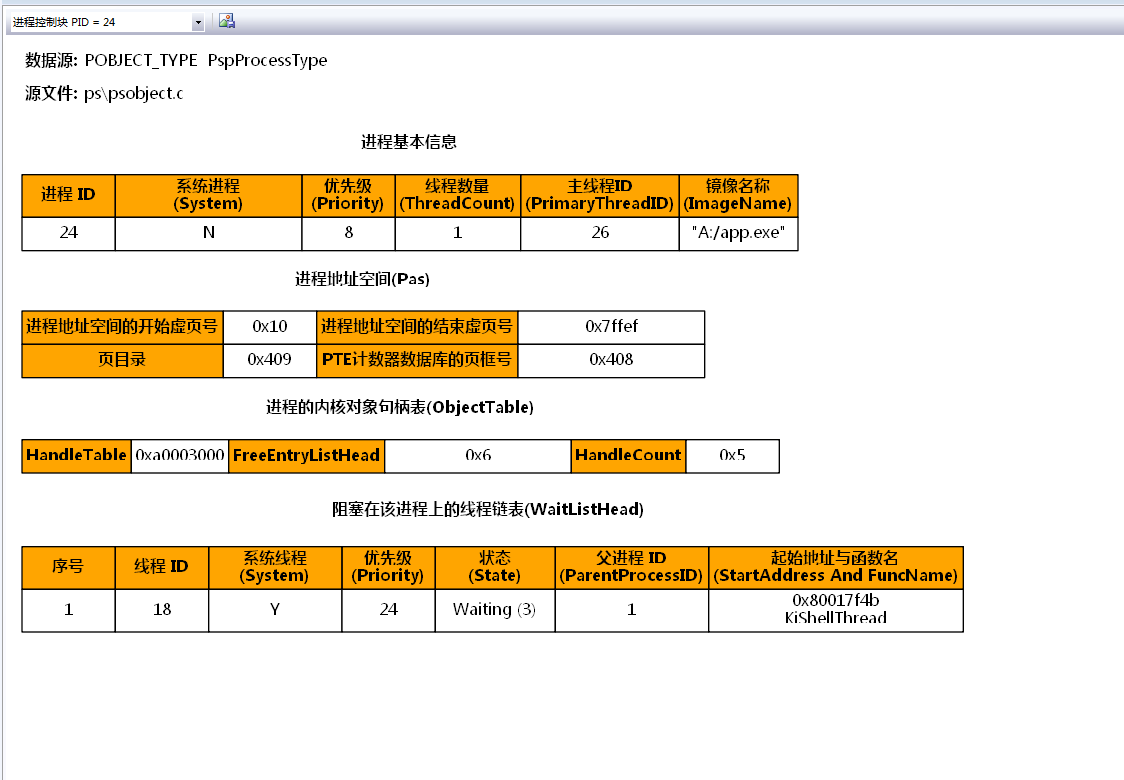
镜像名称分别为“A:\EOSApp.exe”（会根据读者创建的项目名称变化）和“A:\Hello.exe”。其中，A:\EOSApp.exe 进程是父进程，其主线程处于运行状态（正准备调用 WaitForSingleObject 函数，但是命中了断点）；A:\Hello.exe 进程是子进程， 其主线程处于就绪状态， 当父进程的主线程通过调用WaitForSingleObject 函数进入阻塞状态让出处理器后，这个处于就绪状态的线程就会占用处理器开始运行。这两个应用程序的进程和线程的优先级都为 8。

查看“进程线程”窗口：



* + 1. 选择“调试”菜单“窗口”中的“进程控制块”菜单打开“进程控制块” 窗口，在此窗口工具栏上的组合框中选择“进程控制块 PID=24”的选项，可以查看 PID 为 24 的进程的详细信息。其中，阻塞在该进程的线程链表中，可以看到 ID 为 18 的控制台线程阻塞在该进程上。这也就是当此进程结束后，控制台线程才能继续接收用户命令的原因。并且其实现方式与父进程等待子进程的方式是一致的，感兴趣的读者可以在 EOS 内核项目的 ke/sysproc.c 文件中的KiShellThread 函数中找到控制台线程根据用户输入的命令创建应用程序的进程，并等待其执行完毕后，继续接收用户命令的源代码。

查看 PID 为 24 的进程的详细信息：



* + 1. 按 F10 单步执行 WaitForSingleObject 函数，再次刷新进程线程窗口， 确认子进程已经结束运行。

单步执行结果如下：

* + 1. 选择“调试”菜单中的“删除所有断点”，删除之前添加的断点。
    2. 选择“调试”菜单中的“停止调试”，结束本次调试。

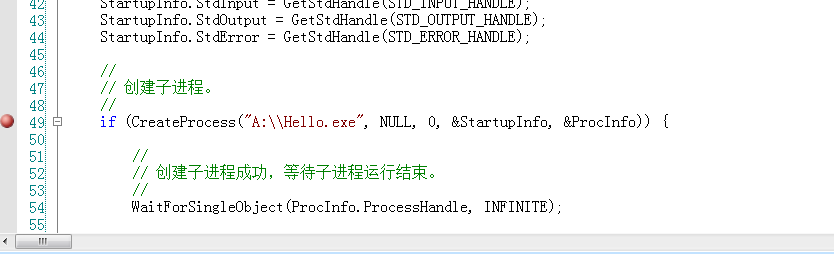
## 从内核的角度理解进程的创建过程

* + 1. **调试 CreateProcess 函数**

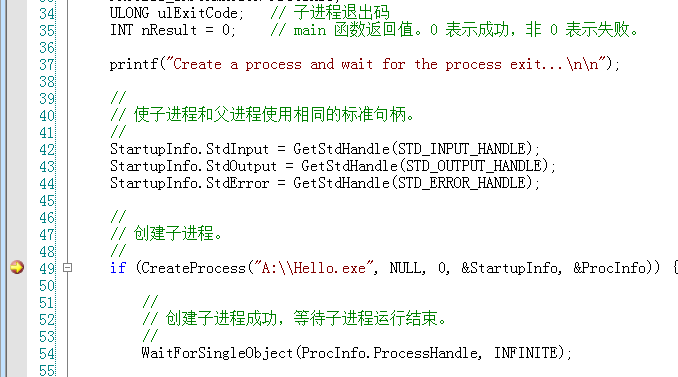
按照下面的步骤调试 CreateProcess 函数创建进程的过程：

1. 打开 EOS 应用程序的 EOSApp.c 文件，在 main 函数中调用

CreateProcess 函数的代码行（第 49 行）添加一个断点。添加断点如下：



1. 按 F5 启动调试，会在刚刚添加的断点处中断。执行结果如下：

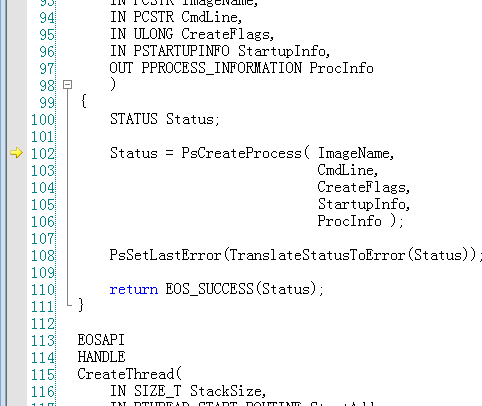


1. 选择“调试”菜单“窗口”中的“进程线程”菜单打开“进程线程”窗口，点击工具栏上的“刷新”按钮，可以看到当前系统中包含两个进程，一个是系统进程，创建了 6 个线程，另一个是应用程序进程，创建了一个线程，并处于运行态，也就是在其准备调用 CreateProcess 函数时命中了断点。

查看“进程线程”窗口：

1. 按 F11 调试进入 CreateProcess 函数。注意，此时已经进入 EOS 内核了！

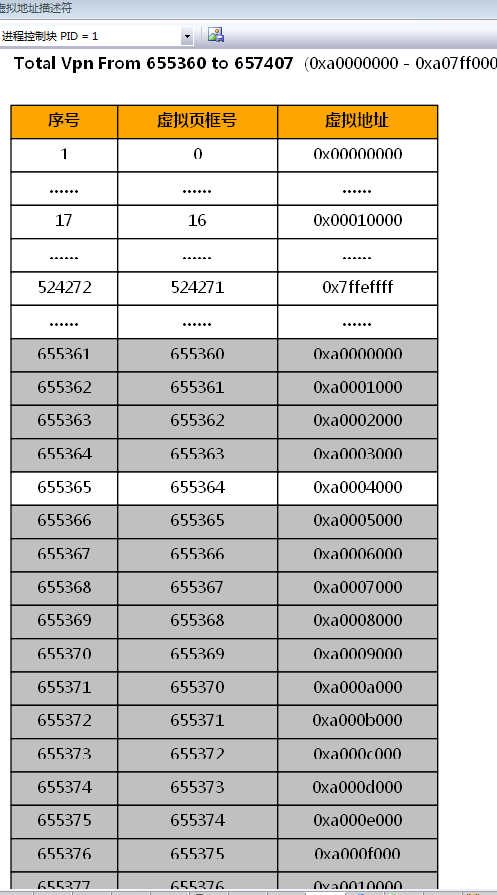
调试状态如下：



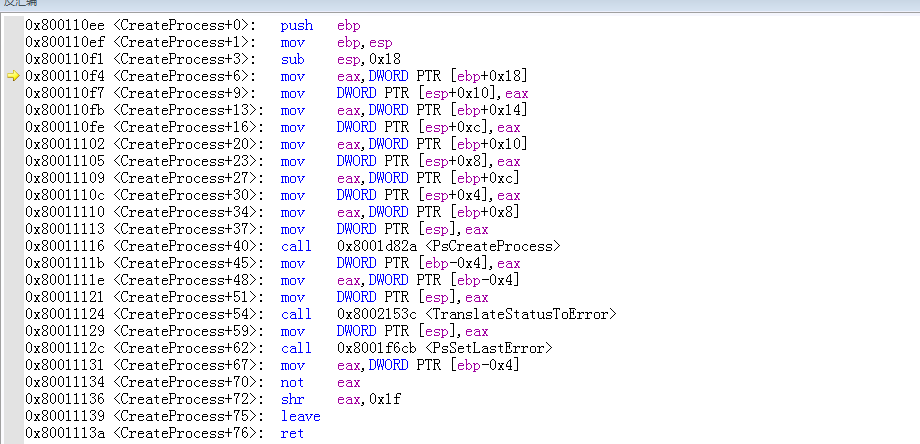
从图中可以看到，当 EOS 应用程序 eosapp.exe 存储在软盘上的时候，它是静态的，只包含应用程序的指令和数据。而创建进程后，进程不但包含应用程序的指令和数据，也会包含操作系统内核（kernel.dll）的指令和数据（参见图5-1 ）。同时， 图示也说明了一个进程可以包含多个程序， 该进程包含了eosapp.exe 和 kernel.dll 两个程序。

继续在内核中调试进程创建的过程之前，读者可以先按照下面的步骤来分别验证一下图中所示的应用程序和操作系统内核在进程的 4G 虚拟地址空间中所处的位置：

1. 由于此时是在内核中的 CreateProcess 函数内中断的，所以先来看一下内核在 4G 虚拟地址空间中的位置。选择“调试”菜单“窗口”中的“虚拟地址描述符”菜单项，打开“虚拟地址描述符”窗口，从该窗口的工具栏下拉框中选择“进程控制块 PID=1”选项，此时显示的是系统进程的虚拟地址描述符表， 其描述的就是系统进程动态管理的存储空间，用于存储内核管理的数据。可以看到，已经使用的虚拟地址空间（底色为灰色的表示已使用）都大于 0x80000000。

查看“虚拟地址描述符”窗口：

1. 在“调试”菜单“窗口”中选择“反汇编”，会在“反汇编”窗口中显示 CreateProcess 函数的指令对应的反汇编代码。在“反汇编”窗口的左侧显示的是指令所在的虚拟地址，可以看到，CreateProcess 函数中所有指令的虚拟地址都是大于 0x80000000 的，从而验证了内核程序（kernel.dll）位于高 2G 虚拟地址空间。

查看“反汇编”窗口：

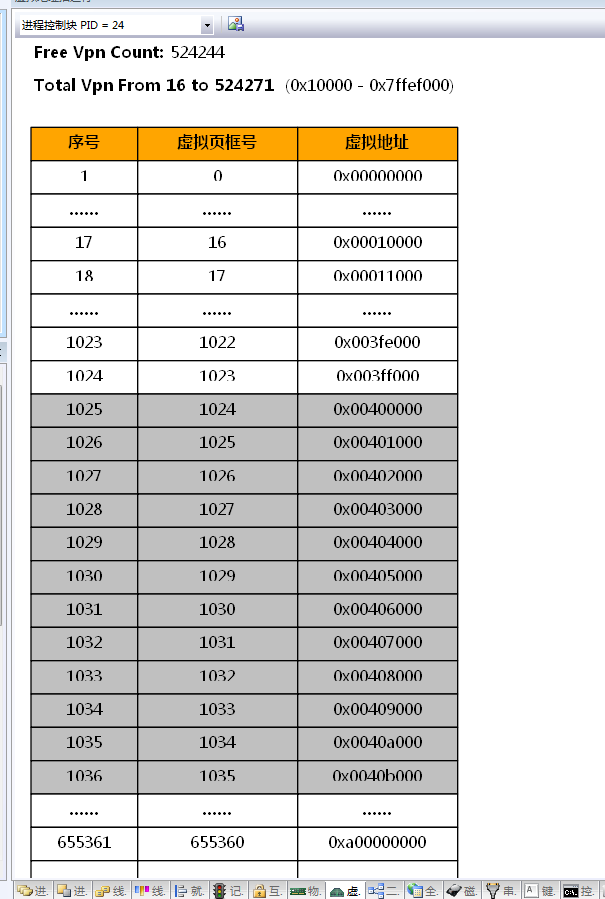
1. 内核程序（kernel.dll）在虚拟地址空间中的位置与其基地址是一致的。读者可以再启动一个 Linux Lab，打开在 3.1 节中新建的 EOS Kernel 项目， 然后在“项目管理器”窗口中右键点击项目节点（根节点），选择菜单中的“属性”，在弹出的“属性页”窗口左侧选择“链接器”，右侧的“基址”属性的值就是告诉链接器使用这个数值作为内核程序的基地址。

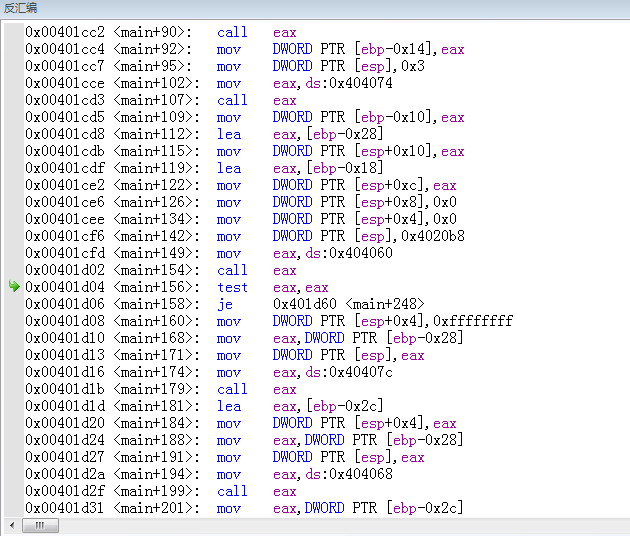
但是，只在链接时指定内核程序的基地址是不够的，加载程序还需要根据此值将内核程序加载到物理内存中的对应位置，然后将其映射到虚拟地址空间中的正确位置。感兴趣的读者可以在完成实验 8 后，在理解了分页存储器管理的机制后再回到这里进行深入研究。

1. 接下来确认应用程序在 4G 虚拟地址空间中的位置。在“虚拟地址描述符”窗口工具栏的下拉列表中选择“进程控制块 PID=24”的选项，此时显示的是应用程序进程的虚拟地址描述符表。

可以看到已经使用的虚拟地址都小于 0x80000000。在“调用堆栈”窗口中双击 main 函数项，设置 main 函数的调用堆栈帧为激活状态，在“反汇编”窗口中可以查看应用程序 main 函数的反汇编，可以看到 main 函数所有指令的虚拟地址都是小于 0x80000000 的，这就说明应用程序（eosapp.exe）位于低 2G 的虚拟地址空间中。读者也可以使用类似上一个步骤中的方法,查看 EOS 应用程序项目链接器属性中基址的值。

查看“进程控制块 PID=24”：



查看“反汇编”窗口：

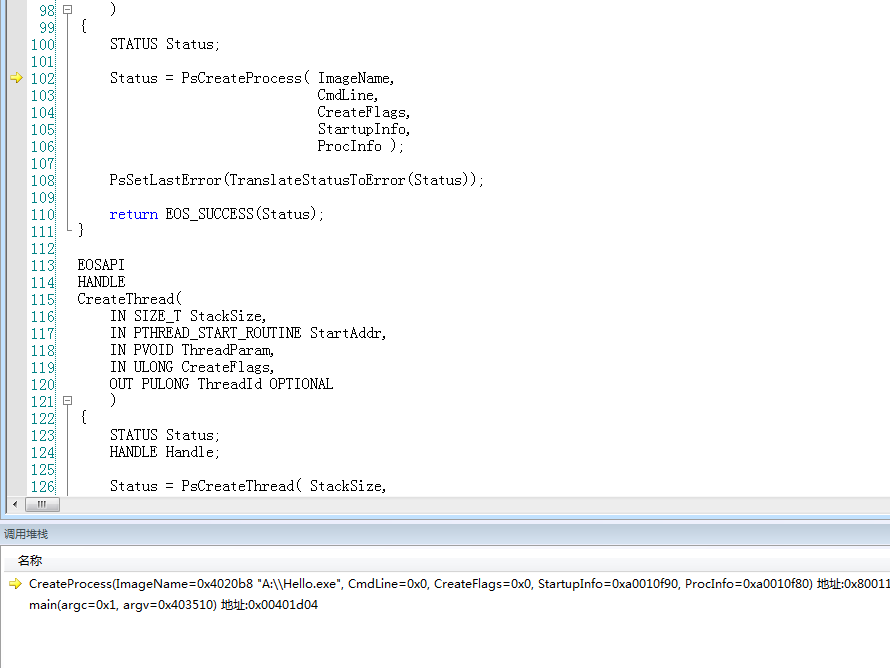
1. 选择“调试”菜单中的“窗口”中的“物理内存”菜单项，打开“物理内存”窗口，可以查看当前系统物理内存的使用情况。

查看“物理内存”菜单项：



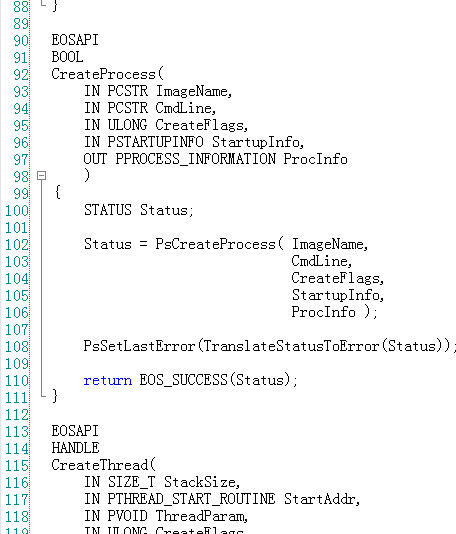
1. 在“调用堆栈”窗口中双击 CreateProcess 函数项，重新设置

CreateProcess 函数的调用堆栈帧为激活的。设置过程如下：

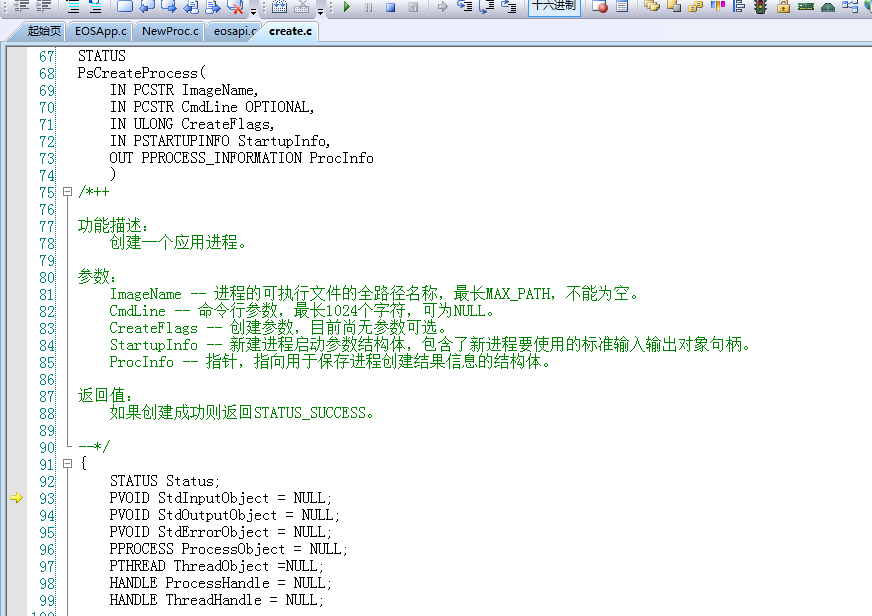


1. 为了在接下来的调试过程中避免反汇编代码影响调试效率，读者可以关闭“反汇编”窗口。接下来观察 eosapi.c 文件中 CreateProcess 函数的源代码，可以看到此函数只是调用了 EOS 内核函数 PsCreateProcess 并将创建进程所用到的参数传递给了此函数。所以，此时读者可以按 F11 调试进入create.c 文件中的 PsCreateProcess 函数，在此函数中才真正开始执行创建进程的各项操作。

观察 CreateProcess 函数：



观察 PsCreateProcess 函数：



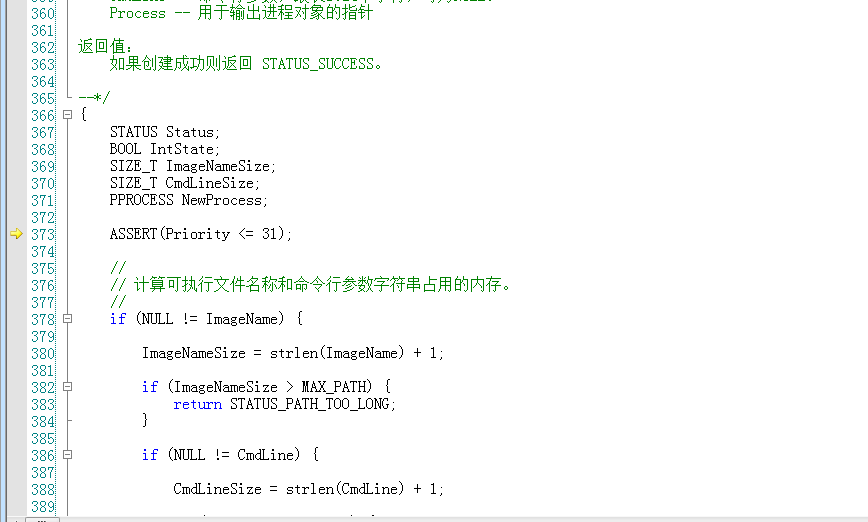
## 调试 PsCreateProcess 函数

创建进程最主要的操作就是创建进程控制块(PCB),并初始化其中的各种信息（也就是为进程分配各种资源）。所以在 PsCreateProcess 函数中首先调用了 PspCreateProcessEnvironment 函数来创建进程控制块。

调试 PspCreateProcessEnvironment 函数的步骤如下：

1. 在 PsCreateProcess 函数中找到调用 Ps
2. pCreateProcessEnvironment 函数的代码行（create.c 文件的第 163

行），并在此行添加一个断点。添加断点如下图所示：

1. 按 F5 继续调试，到此断点处中断。调试状态如下：
2. 按 F11 调试进入 PspCreateProcessEnvironment 函数。 由于PspCreateProcessEnvironment 函数的主要功能是创建进程控制块并初始化其中的部分信息，所以在此函数的开始，定义了一个进程控制块的指针变量NewProcess。在此函数中查找到创建进程控制块的代码行（create.c 文件的第418 行）

Status = ObCreateObject( PspProcessType,NULL, sizeof(PROCESS) + ImageNameSize + CmdLineSize,0, (PVOID\*)&NewProcess );

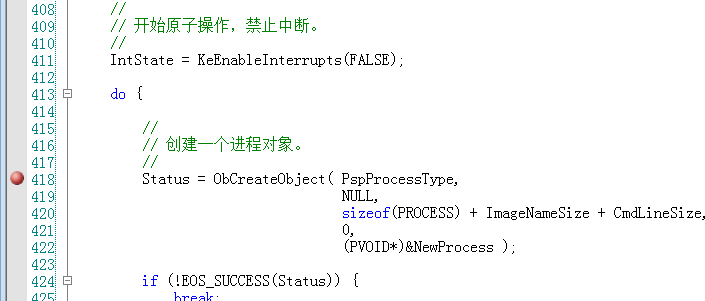
这里的 ObCreateObject 函数会在由 EOS 内核管理的内存中创建了一个新的进程控制块（也就是分配了一块内存），并由 NewProcess 返回进程控制块的指针（也就是返回所分配内存的起始地址）。

查看该函数如下：

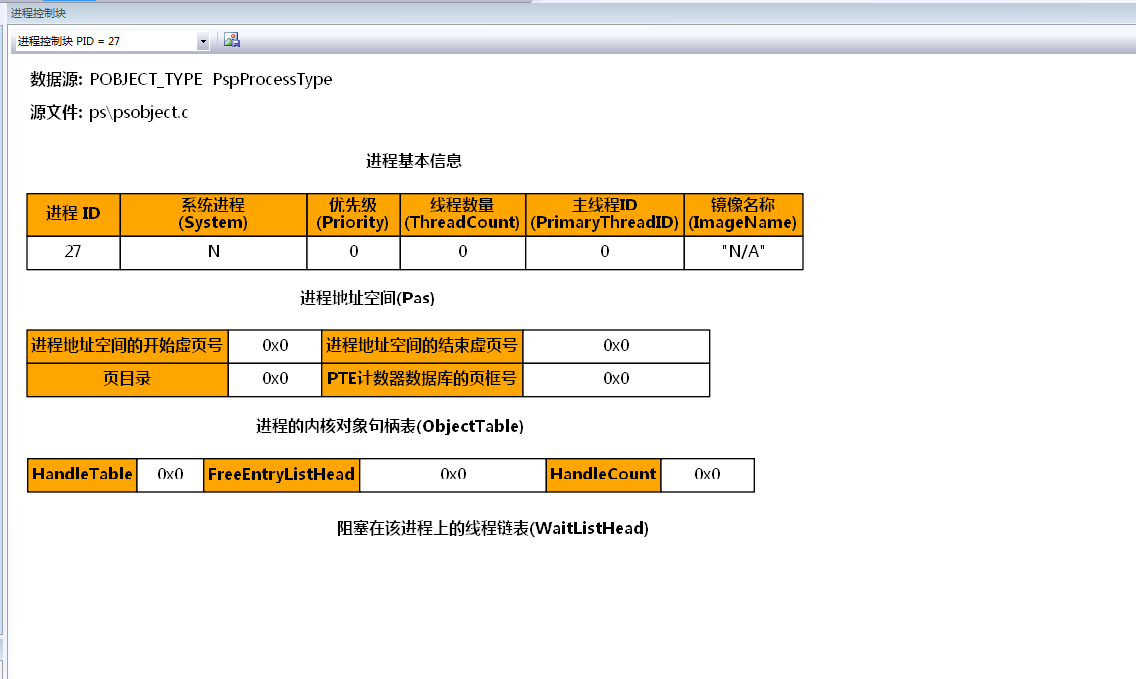
按照下面的步骤调试进程控制块的创建过程：

1. 在调用 ObCreateObject 函数的代码行（create.c 文件的第 418 行） 添加一个断点。

添加断点如下：



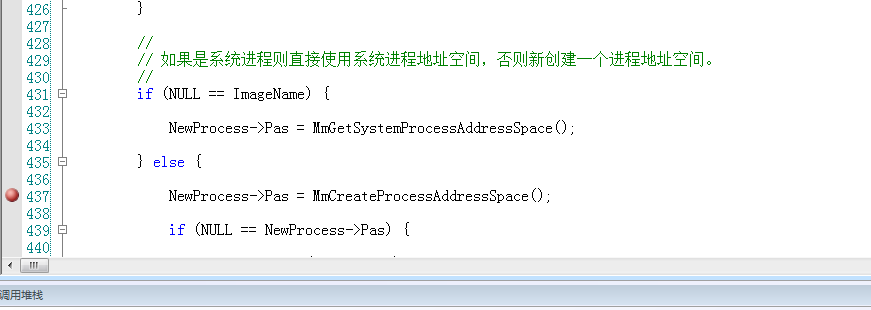
1. 按 F5 继续调试，到此断点处中断。
2. 按 F10 执行此函数后中断。此时调试状态如下：
3. 此时为了查看进程控制块中的信息，选择“调试”菜单“窗口”中的“进程控制块”，打开“进程控制块”窗口，在该窗口工具栏的下拉列表中选择“进程控制块 PID=27”，可以看到该进程控制块中各个成员变量的值。由于目前只是新建了进程控制块，还没有为其中的成员变量赋值，所以值都为 0。

查看“进程控制块”窗口如下：

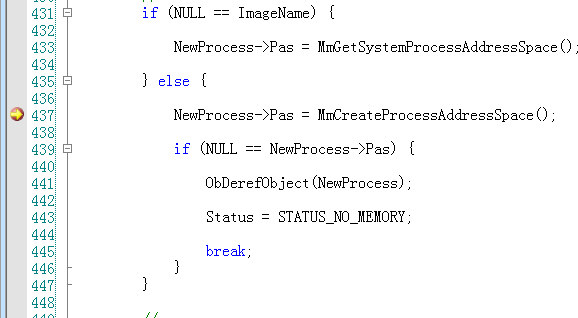
接下来调试初始化进程控制块中各个成员变量的过程：

1. 首先创建进程的地址空间，即 4G 虚拟地址空间。在代码行（create.c 文件的第 437 行）NewProcess->Pas = MmCreateProcessAddressSpace();添加一个断点。

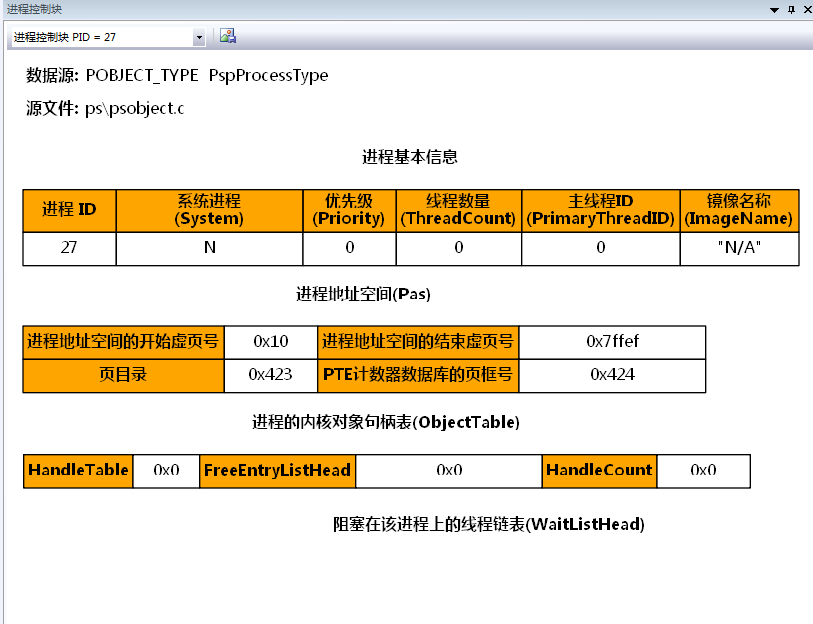
添加断点如下：



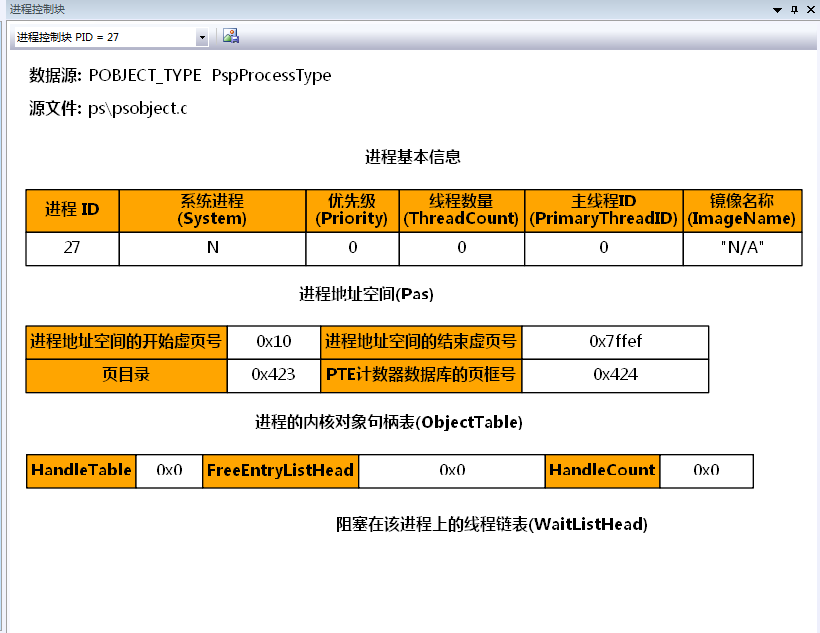
1. 按 F5 继续调试，到此断点处中断。
2. 按 F10 执行此行代码后中断。调试状态如下：



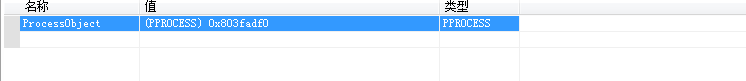
1. 在“进程控制块”窗口中，选择“进程控制块 PID=27”，可以看到该进程控制块的“进程地址空间”的值已经不再是 0。说明已经初始化了进程的 4G 虚拟地址空间。还可以在“监视”窗口中查看进程控制块的成员变量 Pas 的值已经不再是 0。

查看“进程控制块”窗口如下：

1. 使用 F10 一步步调试 PspCreateProcessEnvironment 函数中后面的代码，在调试的过程中根据执行的源代码，可以在“进程控制块”窗口中查看“进程控制块 PID=27”的信息，或者在“监视”窗口中查看\*NewProcess 表达式的值，观察进程控制块中哪些成员变量是被哪些代码初始化的，哪些成员变量还没有被初始化。

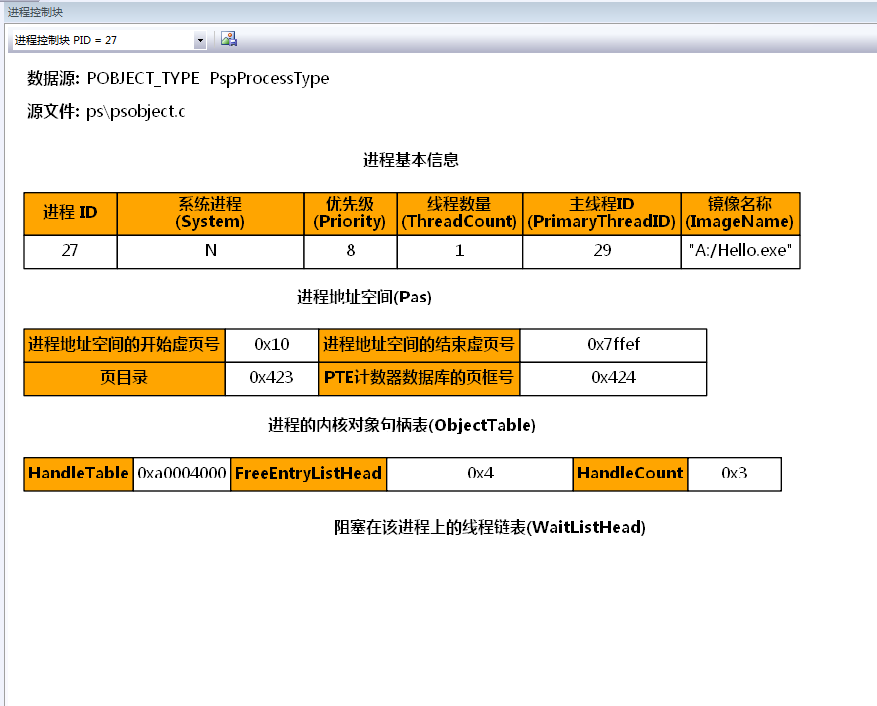
查看“进程控制块 PID=27”的信息如下：

1. 当从 PspCreateProcessEnvironment 函数返回到 PsCreateProcess 函数后，停止按 F10。此时“监视”窗口中已经不能再显示表达式\*NewProcess 的值了，在 PsCreateProcess 函数中是使用 ProcessObject 指针指向进程控制块的，所以将表达式\*ProcessObject 添加到“监视”窗口中就可以继续观察新建进程控制块中的信息。

添加“监视”窗口过程如下：

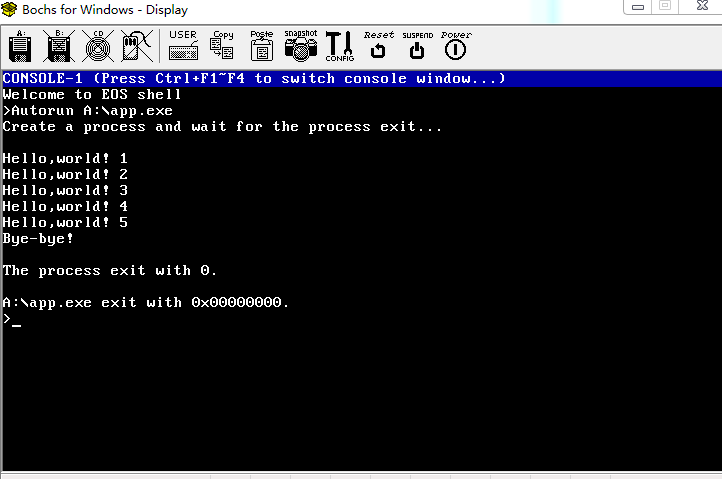
1. 接下来继续使用 F10 一步步调试 PsCreateProcess 函数中的代码，同样要注意观察执行后的代码修改了进程控制块中的哪些成员变量。当调试到PsCreateProcess 函数的最后一行代码时，在监视窗口总查看进程控制块中的信息，此时所有的成员变量都已经被初始化了（注意观察成员 ImageName 的值）。

查看进程控制块状态如下：



1. 按 F5 继续执行，EOS 内核会为刚刚初始化完毕的进程控制块新建一个进程。激活虚拟机窗口查看新建进程执行的结果。

执行结果如下：

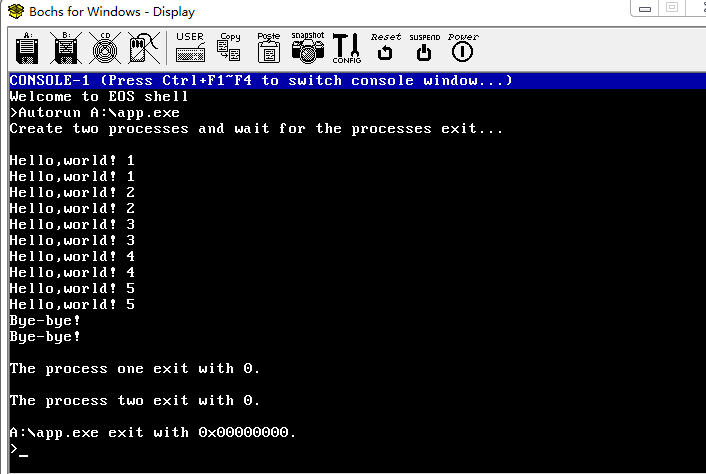


1. 在 OS Lab 中选择“调试”菜单中的“停止调试”结束此次调试。
2. 选择“调试”菜单中的“删除所有断点”。

## 练习通过编程的方式创建一个应用程序的多个进程

使用 OS Lab 打开本实验文件夹中的参考源代码文件 NewTwoProc.c，仔细阅读此文件中的源代码。使用 NewTwoProc.c 文件中的源代码替换 EOS 应用程序项目中 EOSApp.c 文件内的源代码，生成后启动调试，查看多个进程并发执行的结果。多个进程并发时，EOS 操作系统中运行的用户进程可以参见图示，从而验证一个程序（hello.exe）可以同时创建多个进程。

执行结果如下：

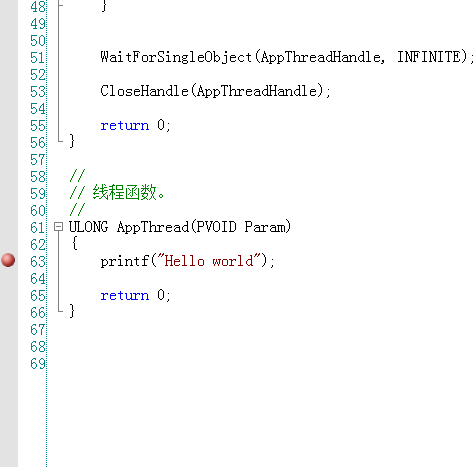


## 在应用程序进程中创建一个工作线程（ Worker Thread ）

前面的练习是通过编程的方式在一个应用程序进程中创建另一个应用程序的进程，接下来请读者按照下面的步骤练习在一个应用程序进程中创建一个工作线程的方法：

* + 1. 打开“学生包”本实验对应的文件夹，使用其中的 AppProg.c 文件中的源代码替换之前创建的 EOS 应用程序项目中 EOSApp.c 文件内的源代码。
    2. 按 F7 生成项目。
    3. 删除之前添加的所有断点后，在工作线程函数 AppThread 中（第 63 行） 添加一个断点。

添加断点如下：



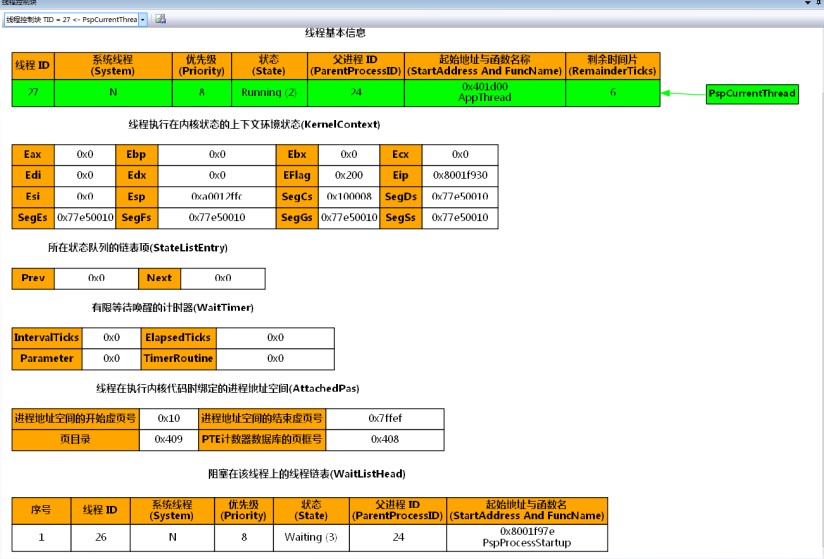
* + 1. 按 F5 启动调试，在断点的位置中断。
    2. 刷新“进程线程”窗口，可以看到如图所示的内容。在应用程序进程的线程列表中，包含两个线程，一个是应用程序的主线程，处于阻塞状态，说明其正在等待工作线程结束；另一个是在应用程序中通过调用 CreateThread 函数创建的工作线程，处于运行状态，说明其正在执行线程函数并命中了断点。

查看“进程线程”窗口如下：



* + 1. 在“线程控制块”窗口工具栏的组合框中选择“线程控制块 TID=27”的选项，可以查看工作线程的详细信息，如图所示，包括：线程的基本信息、线程执行在内核状态的上下文环境状态、所在状态队列的链表项（由于该线程处于运行态，也就不在任何状态队列中，所以 Prev 和 Next 指针的值都为 0）、有限等待唤醒的计时器、线程在执行内核代码时绑定的地址空间、阻塞在该线程上的线程链表。其中，阻塞在该线程上的线程链表中显示了应用程序的主线程阻塞在该工作线程上，也就是主线程在等待该工作线程结束。

查看“线程控制块 TID=27”如下：

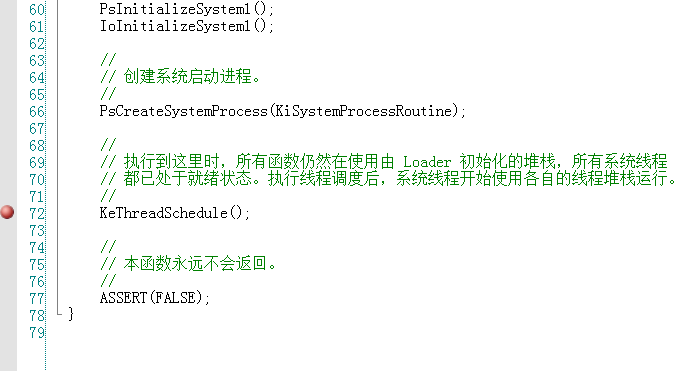


## 系统线程的创建过程

通过前面的实验，相信读者已经对进程的创建过程和线程的创建过程有了一个全面而深入的理解。接下来，通过调试 EOS 内核的初始化过程来理解几个重要的系统线程的创建过程，这些线程包括：系统初始化线程、控制台派遣线程、控制台线程。

* + 1. 新建一个 EOS Kernel 项目。
    2. 打开 ke\start.c 文件，在第 72 行代码 KeThreadSchedule();处添加一个断点。

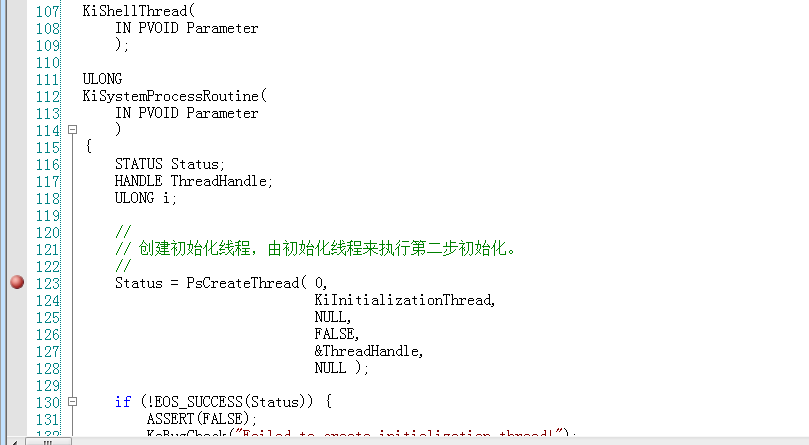
添加断点如下：



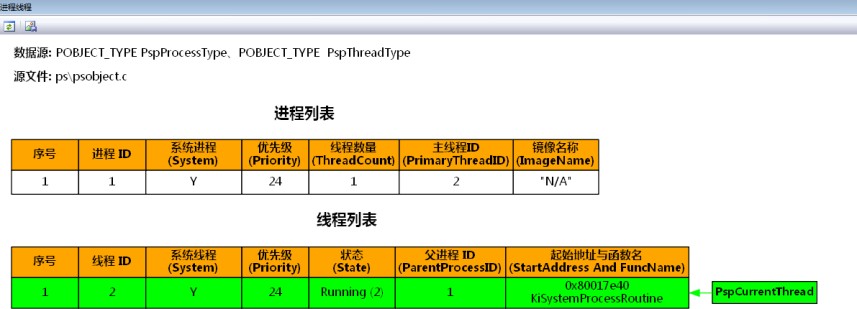
* + 1. 按 F5 启动调试，在断点处中断。
    2. 刷新“进程线程”窗口，显示如图所示的内容。可以看到当前系统中， 只创建了一个线程，该线程就是由第 66 行的 PsCreateSystemProcess 函数在创建系统进程后，为系统进程创建的第一个子线程，同时也是系统进程的主线程。

查看“进程线程”窗口如下：

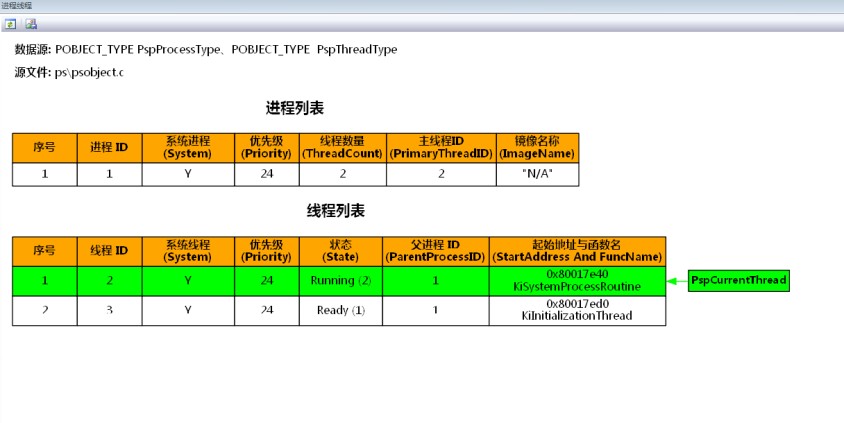
* + 1. 由于第一个系统线程处于就绪状态，并且是当前系统中唯一的线程，所以在执行第 72 行的线程调度函数后，就会开始执行其线程函数。接下来，在其线 程 函 数 中 添 加 一 个 断 点 ： 打 开 ke\sysproc.c 文 件 ， 在KiSystemProcessRoutine 函数中的（第 123 行代码处）添加一个断点。

添加断点如下：

* + 1. 按 F5 继续调试，在断点处中断。
    2. 刷新“进程线程”窗口，可以看到系统中唯一的线程处于运行状态，说

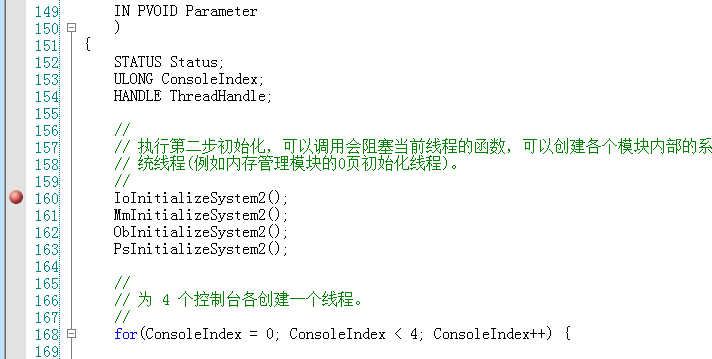
明该线程正在执行其线程函数并命中了断点。查看“进程线程”窗口如下：

* + 1. 当前中断位置处的代码行用于创建一个新的系统线程，并使用该新建线程继续进行操作系统的初始化工作。所以，按 F10 单步调试一次后，刷新“进程线程”窗口，会显示如图所示的内容，可以看到当前系统中又多出了一个线程， 并处于就绪状态。

刷新“进程线程”窗口如下：

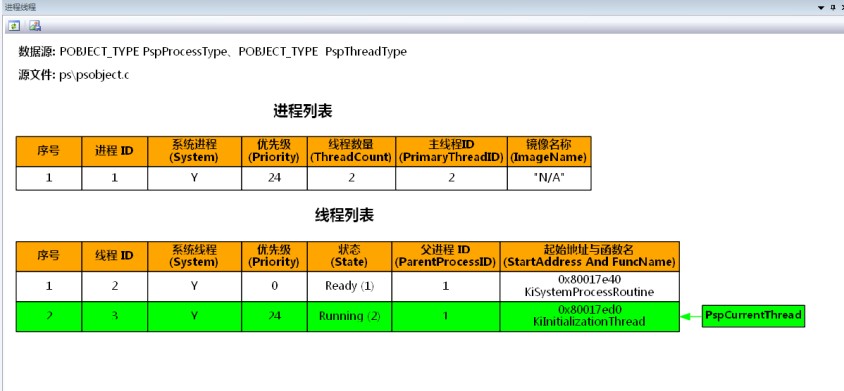
* + 1. 由于第一个系统线程会使用第 140 行的代码将自己的优先级降为 0， 退化为空闲线程。这就会让刚刚新建的还处于就绪状态的系统初始化线程抢占处理器开始运行。接下来，在初始化线程的线程函数中添加一个断点： 打开ke\sysproc.c 文件，在 KiInitializationThread 函数中（第 160 行代码处） 添加一个断点。

添加断点如下：



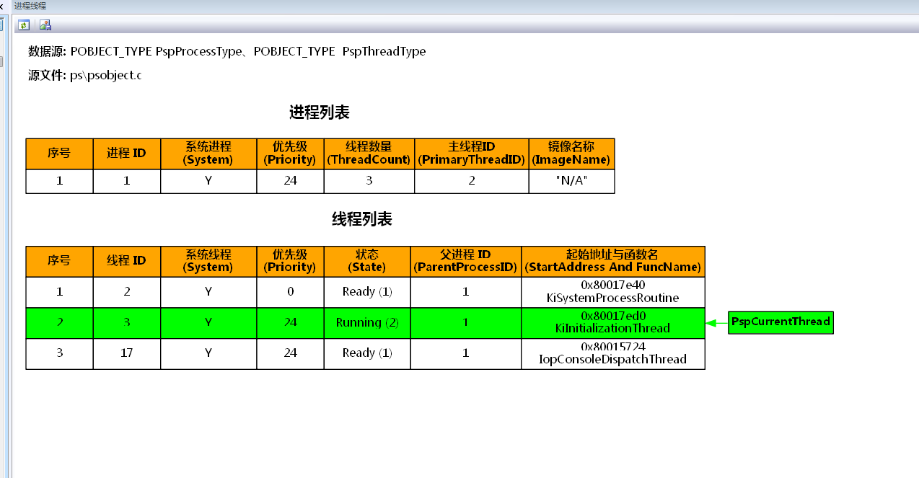
* + 1. 按 F5 继续调试，在断点处中断。
    2. 刷新“进程线程”窗口，可以看到系统初始化线程处于运行状态，说明该线程正在执行其线程函数并命中了断点。

查看“进程线程”窗口如下：



* + 1. 按 F10 单步调试执行 IoInitializeSystem2 函数，完成基本输入输出的初始化。刷新“进程线程”窗口，显示如图 11-17 所示的内容，可以看到当前系统中已经创建了控制台派遣线程，并处于就绪状态。请读者自行查找创建控制台派遣线程的代码位置，并尝试说明其函数调用层次。

查看“进程线程”窗口如下：



* + 1. 由于第 168 行的 for 语句会循环创建四个控制台线程，所以，读者可以在 184 行代码处添加一个断点，按 F5 继续调试。待命中断点后，刷新“进程线程”窗口，会显示如图 11-18 所示的内容，可以看到当前系统已经完成了四个控制台线程的创建，并且所有的控制台线程都处于就绪状态。

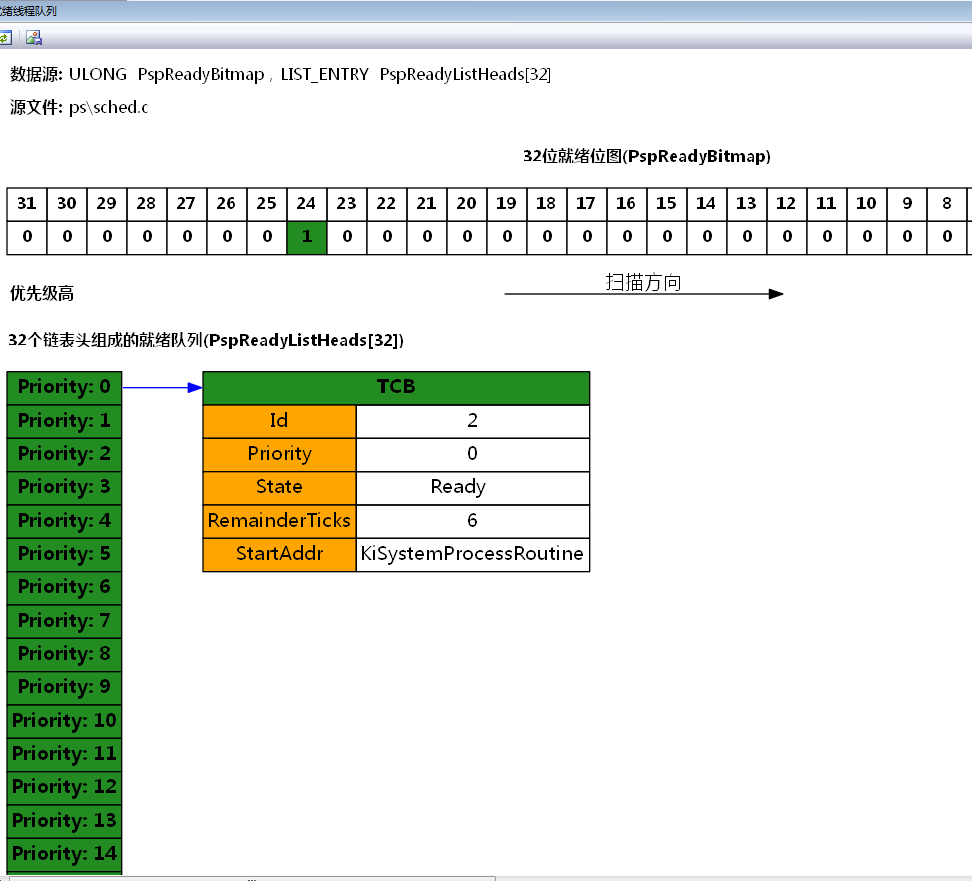
添加断点如下：

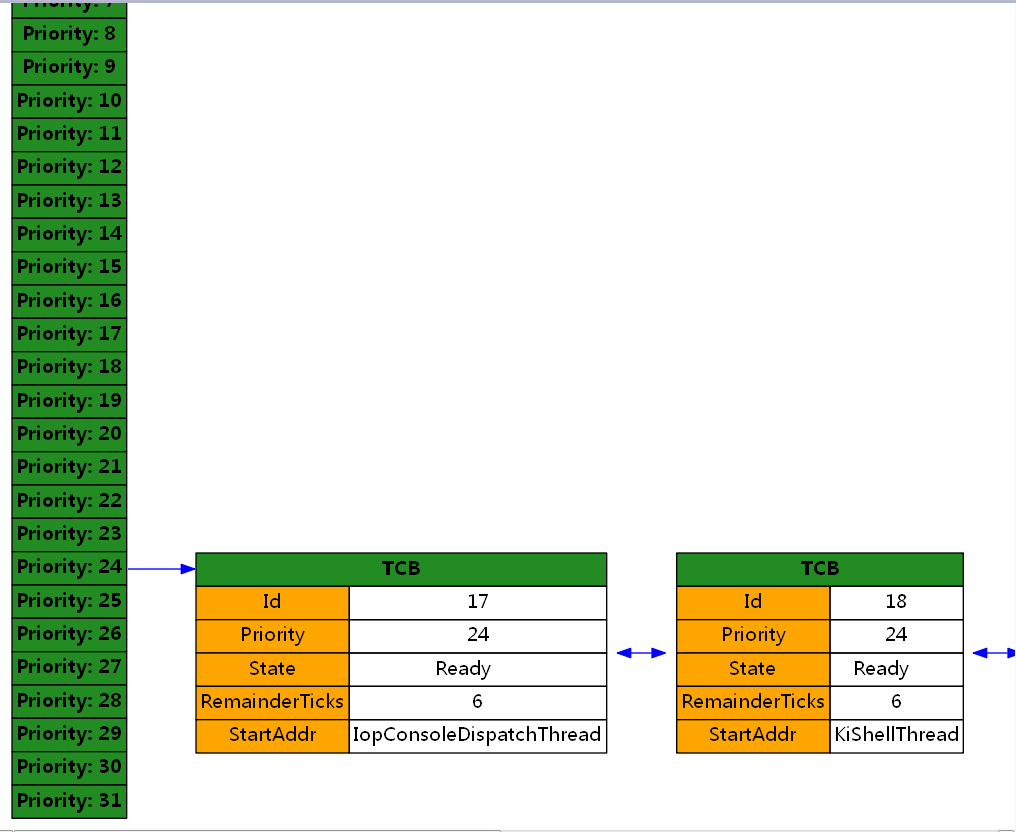


查看“进程线程”窗口如下：

* + 1. 选择“调试”菜单“窗口”中的“就绪线程队列”，打开“就绪线程队列”窗口。点击此窗口工具栏上的“刷新”按钮，会显示如图 11-19 所示的内容。顾名思义，就绪线程队列用于管理那些处于就绪状态的线程控制块，可以看到优先级为 0 的空闲线程，以及优先级为 24 的控制台派遣线程和四个控制台线程分别在其优先级对应的就绪队列中，而当前处于运行态的系统初始化线程

（线程 ID 为 3）就不在就绪队列中。查看“就绪线程队列”窗口如下：

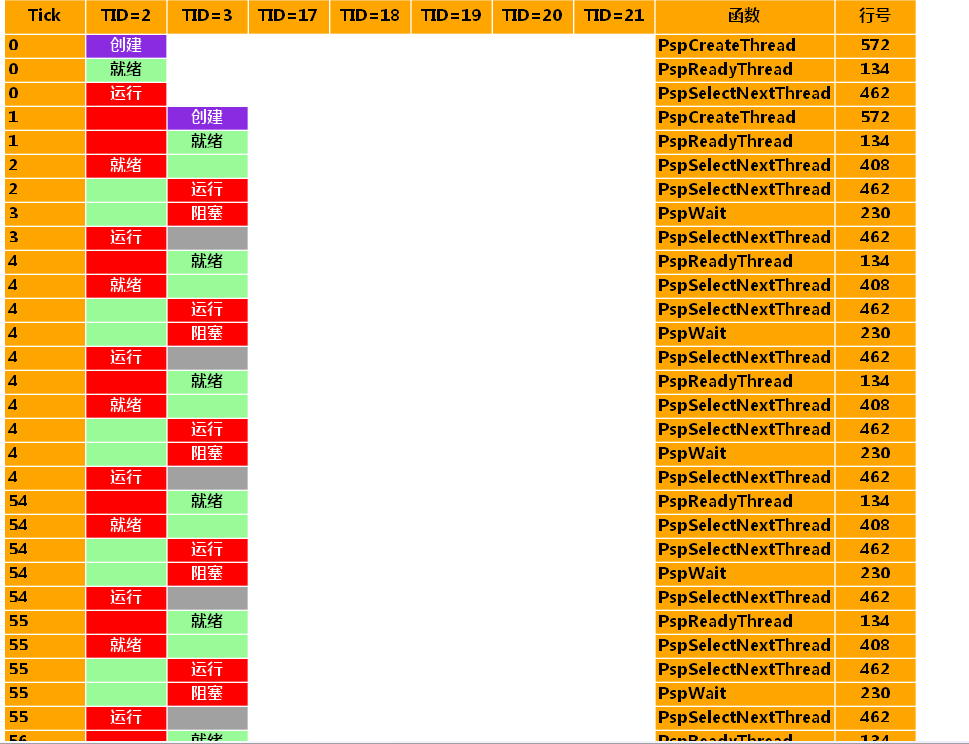


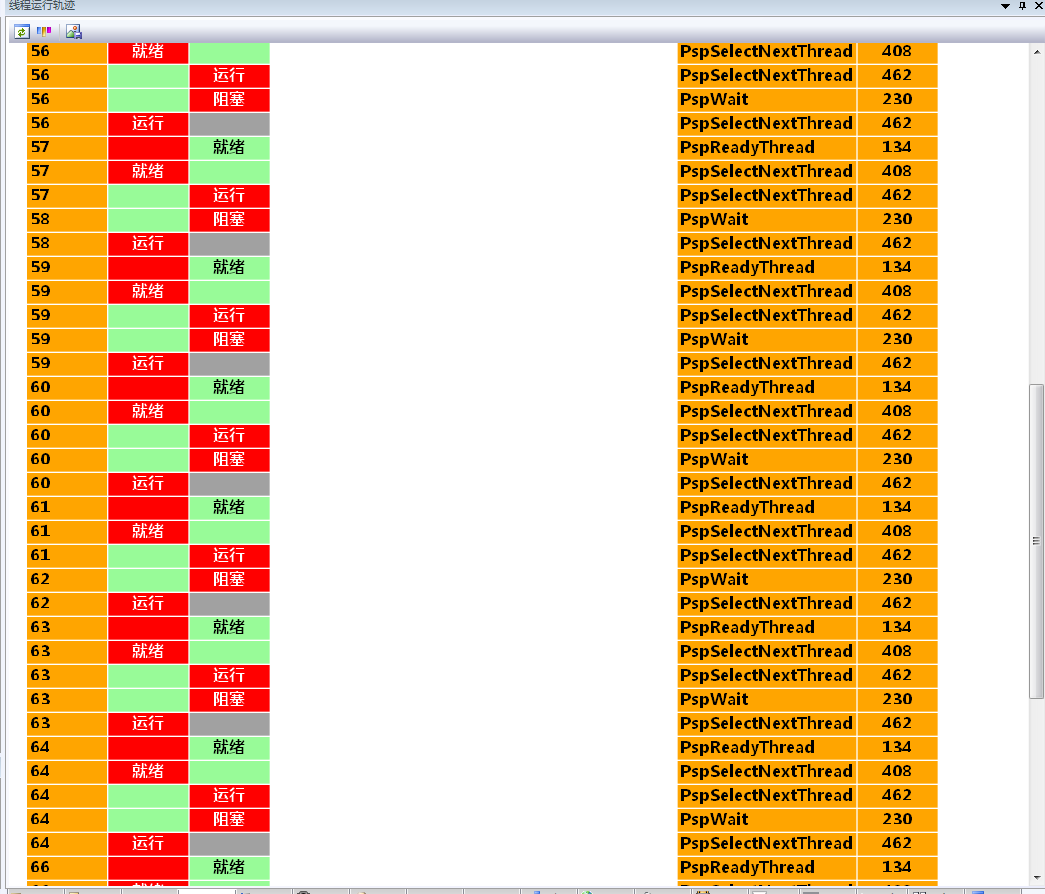


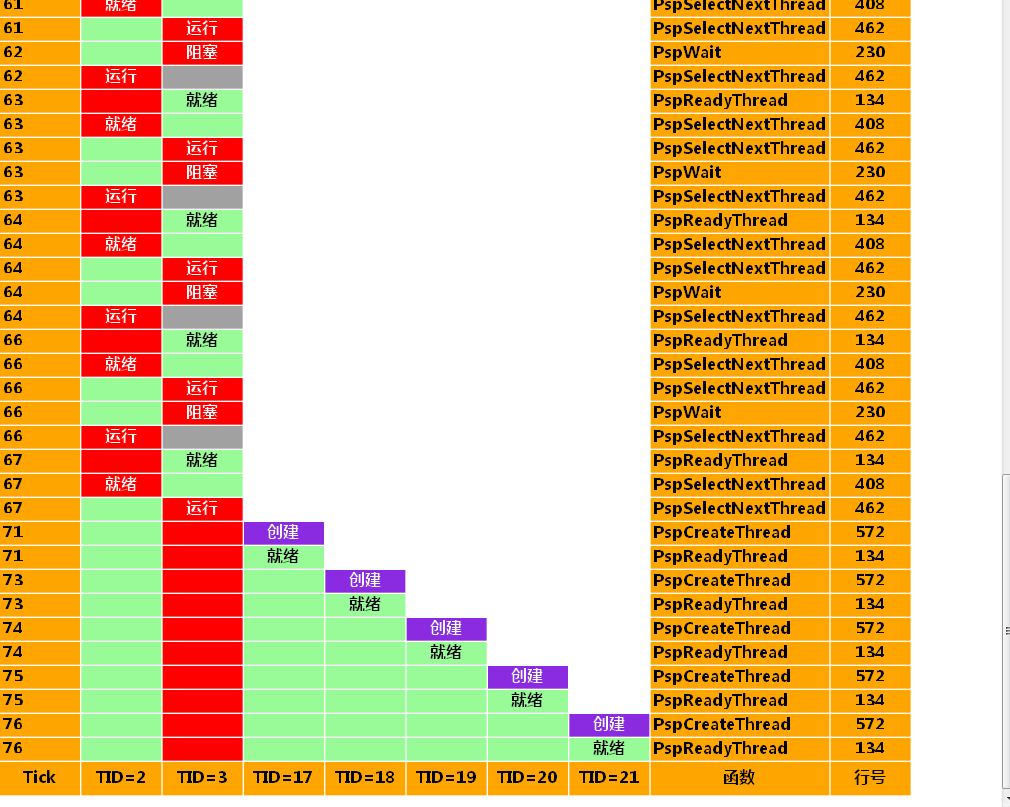
* + 1. 选择“调试”菜单“窗口”中的“线程运行轨迹”，打开“线程运行轨迹”窗口。点击此窗口工具栏上的“刷新”按钮，会显示如图 11-20 所示的内容，可以查看这些系统线程的状态转换过程和运行轨迹。注意，系统初始化线程

（TID=3）在初始化的过程中会由于等待一些硬件设备的响应，从而频繁进入阻塞状态。

查看“线程运行轨迹”如下：







至此，系统初始化线程（TID=3）已经完成其所有的工作，在其结束运行后， 处于就绪状态的控制台派遣线程和四个控制台线程会依次获得处理器，但是控制台派遣线程会在等待键盘事件时进入阻塞状态，四个控制台线程会在等待控制台派遣线程分派任务时进入阻塞状态，当它们都进入阻塞状态后，低优先级的系统空闲线程（TID=2）就会占用处理器开始运行，直到有更高优先级的线程抢占处理器为止。

通过以上对系统线程创建和运行过程的调试，读者对 EOS 内核从系统入口点函数（KiSystemStartup）开始执行到完全启动这个过程已经有了一个初步的了解，在后面的实验中，将会对这些内容进行更加深入的研究。

1. **实验的思考与问题分析**
2. 在源代码文件 NewTwoProc.c 提供的源代码基础上进行修改，要求使用

hello.exe 同时创建 10 个进程。

答：使用 PROCESS\_INFORMATION 类型定义一个有 10 个元素的数组，每一个元素对应一个进程，这就完成了进程的定义。在进行进程创建时，使用一个循环创建 10 个子进程，同样，在执行时，然后再使用一个循环等待 10 个子进程结束，得到退出码后关闭句柄。

1. 学习本书第 5 章中的 5.2 节，了解关于线程的相关知识，然后尝试调试 PspCreateThread 函数，并在“线程控制块”窗口中观察线程控制块（TCB） 初始化的过程。

答：在 EOS 内核中，线程的创建最终都是通过调用 PspCreateThread 函数

（在文件 ps/create.c 中定义）完成的。该函数的流程比较简单，首先是创建一个空白的线程控制块，然后为线程分配栈，并初始化线程的上下文环境，最后使线程进入就绪状态。

如果该函数成功的创建了一个新线程，就返回线程对象的句柄，否则就会返回 NULL。

在调用 CreateThread 函数创建线程之前，要首先按照线程入口函数类型的定义，编写一个线程入口函数。需要强调的一点是，当线程创建成功后，

CreateThread 函数会立即返回，也就是说创建者线程和被创建线程是异步执行的。

当线程入口函数返回时，新建的线程就结束执行了。此时，线程对象会由nonsignaled 状态变为 signaled 状态。注意，EOS 应用程序启动执行时会创建一个主线程，在主线程执行的过程中，使用 CreateThread 函数创建的线程可以认为是子线程。当主线程结束后，进程就会结束，此时，所有的子线程无论是否执行完毕，都会被强制结束执行。所以，一般情况下，主线程应该等待所有的子线程执行完毕后再结束执行。

1. 在 PsCreateProcess 函数中调用了 PspCreateProcessEnvironment 函数后 又 先 后 调 用 了 PspLoadProcessImage 和 PspCreateThread 函数，学习这些函数的主要功能。能够交换这些函数被调用的顺序吗？思考其中的原因。

答：PspCreateProcessEnvironment：创建进程控制块，并分配地址空间和句柄表。

PspLoadProcessImage：将进程的可执行映像加载到进程的地址空间中。PspCreateThread：创建了进程的主线程。

这三个函数的顺序不能变，加载可执行映像之前必须已经为进程创建了地址空间，在创建主线程之前必须已经加载了可执行映像。

# 总结和感想体会

(1) 通过本次实验，让我了解OSLab的基本使用方法,包括打开工程项目和新建工程项目，让我基本掌握了OSLab的基本使用，这次实验让我明白了如何通过打断点调试的方式来查看操作系统内部变量的方式，让我对操作系统有了更加深入的理。

(2) 通过本次实验，我从掌握了如何在OSLab软件上创建新的进程，并通过查看和调试 CreateProcess、PsCreateProcess 函数来掌握系统创建进程的过程，同时提高了我OS编程能力，这次实验让我收获很多。