

第5章 作 业

通常，必须将一组进程当作单个实体来处理。例如，当让 Microsoft Developer Studio 为你创建一个应用程序项目时，它会生成 Cl.exe，Cl.exe 则必须生成其他的进程（比如编译器的各个函数传递）。如果用户想要永远停止该应用程序的创建，那么 Developer Studio 必须能够终止 Cl.exe 和它的所有子进程的运行。在 Windows 中解决这个简单（和常见的）的问题是极其困难的，因为 Windows 并不维护进程之间的父/子关系。即使父进程已经终止运行，子进程仍然会继续运行。

当设计一个服务器时，也必须将一组进程作为单个进程组来处理。例如，客户机可能要求服务器执行一个应用程序（这可以生成它自己的子应用程序），并给客户机返回其结果。由于可能有许多客户机与该服务器相连接，如果服务器能够限制客户机的要求，即用什么手段来防止任何一个客户机垄断它的所有资源，那么这是非常有用的。这些限制包括：可以分配给客户机请求的最大 CPU 时间，最小和最大的工作区的大小，防止客户机的应用程序关闭计算机，以及安全性限制等。

Microsoft Windows 2000 提供了一个新的作业内核对象，使你能够将进程组合在一起，并且创建一个“沙框”，以便限制进程能够进行的操作。最好将作业对象视为一个进程的容器。但是，创建包含单个进程的作业是有用的，因为这样一来，就可以对该进程加上通常情况下不能加的限制。

我的 StartRestrictedProcess 函数（见清单 5-1）将一个进程放入一个作业，以限制该进程进行某些操作的能力。

Windows 98 Windows 98 不支持作业的操作。

清单 5-1 StartRestrictedProcess 函数

```
void StartRestrictedProcess() {
    // Create a job kernel object.
    HANDLE hjob = CreateJobObject(NULL, NULL);

    // Place some restrictions on processes in the job.

    // First, set some basic restrictions.
    JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };

    // The process always runs in the idle priority class.
    jobli.PriorityClass = IDLE_PRIORITY_CLASS;

    // The job cannot use more than 1 second of CPU time.
    jobli.PerJobUserTimeLimit.QuadPart = 10000000; // 1 sec in 100-ns intervals

    // These are the only 2 restrictions I want placed on the job (process).
    jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS
        | JOB_OBJECT_LIMIT_JOB_TIME;
    SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
        sizeof(jobli));
}
```

```

// Second, set some UI restrictions.
JOB_OBJECT_BASIC_UI_RESTRICTIONS jobuir;
jobuir.UIRestrictionsClass = JOB_OBJECT_UILIMIT_NONE;           // A fancy zero

// The process can't log off the system.
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

// The process can't access USER objects (such as other windows)
// in the system.
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_HANDLES;

SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
    sizeof(jobuir));

// Spawn the process that is to be in the job.
// Note: You must first spawn the process and then place the process in
//       the job. This means that the process's thread must be initially
//       suspended so that it can't execute any code outside of the job's
//       restrictions.
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, "CMD", NULL, NULL, FALSE,
    CREATE_SUSPENDED, NULL, NULL, &si, &pi);
// Place the process in the job.
// Note: If this process spawns any children, the children are
//       automatically part of the same job.
AssignProcessToJobObject(hjob, pi.hProcess);

// Now we can allow the child process's thread to execute code.
ResumeThread(pi.hThread);
CloseHandle(pi.hThread);

// Wait for the process to terminate or
// for all the job's allotted CPU time to be used.
HANDLE h[2];
h[0] = pi.hProcess;
h[1] = hjob;
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_OBJECT_0) {
    case 0:
        // The process has terminated...
        break;
    case 1:
        // All of the job's allotted CPU time was used...
        break;
}

// Clean up properly.
CloseHandle(pi.hProcess);
CloseHandle(hjob);
}

```

现在，解释一下StartRestrictedProcess函数是如何工作的。首先，调用下面的代码，创建一个新作业内核对象：

```
HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);
```

与所有的内核对象一样，它的第一个参数将安全信息与新作业对象关联起来，并且告诉系统，是否想要使返回的句柄成为可继承的句柄。最后一个参数用于给作业对象命名，使它可以供另一个进程通过下面所示的OpenJobObject函数进行访问。

```
HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

与平常一样，如果知道你将不再需要访问代码中的作业对象，那么就必须通过调用CloseHandle来关闭它的句柄。可以在我的StartRestrictedProcess函数的结尾处看到这个代码的情况。应该知道，关闭作业对象并不会迫使作业中的所有进程终止运行。该作业对象实际上做上了删除标记，只有当作业中的所有进程全部终止运行之后，该作业对象才被自动撤消。

注意，关闭作业的句柄后，尽管该作业仍然存在，但是该作业将无法被所有进程访问。请看下面的代码：

```
// Create a named job object.
HANDLE hjob = CreateJobObject(NULL, TEXT("Jeff"));

// Put our own process in the job.
AssignProcessToJobObject(hjob, GetCurrentProcess());

// Closing the job does not kill our process or the job.
// But the name ("Jeff") is immediately disassociated with the job.
CloseHandle(hjob);

// Try to open the existing job.
hjob = OpenJobObject(JOB_OBJECT_ALL_ACCESS, FALSE, TEXT("Jeff"));
// OpenJobObject fails and returns NULL here because the name ("Jeff")
// was disassociated from the job when CloseHandle was called.
// There is no way to get a handle to this job now.
```

5.1 对作业进程的限制

进程创建后，通常需要设置一个沙框（设置一些限制），以便限制作业中的进程能够进行的操作。可以给一个作业加上若干不同类型的限制：

- 基本限制和扩展基本限制，用于防止作业中的进程垄断系统的资源。
- 基本的UI限制，用于防止作业中的进程改变用户界面。
- 安全性限制，用于防止作业中的进程访问保密资源（文件、注册表子关键字等）。

通过调用下面的代码，可以给作业加上各种限制：

```
BOOL SetInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pJobObjectInformation,
    DWORD cbJobObjectInformationLength);
```

第一个参数用于标识要限制的作业。第二个参数是个枚举类型，用于指明要使用的限制类型。第三个参数是包含限制设置值的数据结构的地址，第四个参数用于指明该结构的大小（用于确定版本）。表5-1列出了如何来设置各种限制条件。

表5-1 设置限制条件

限制类型	第二个参数的值	第三个参数的结构
基本限制	JobObjectBasicLimitInformation	JOB_OBJECT_BASIC_LIMIT_INFORMATION
扩展基本限制	JobObjectExtendedLimitInformation	JOB_OBJECT_EXTENDED_LIMIT_INFORMATION
基本UI限制	JobObjectBasicUIRestrictions	JOB_OBJECT_BASIC_UI_RESTRICTIONS
安全性限制	JobObjectSecurityLimitInformation	JOB_OBJECT_SECURITY_LIMIT_INFORMATION

在StartRestrictedProcess函数中，我只对作业设置了一些最基本的限制。指定了一个JOB_OBJECT_BASIC_LIMIT_INFORMATION结构，对它进行了初始化，然后调用SetInformationJobObject函数。JOB_OBJECT_BASIC_LIMIT_INFORMATION结构类似下面的样子：

```
typedef struct _JOB_OBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD         LimitFlags;
    DWORD         MinimumWorkingSetSize;
    DWORD         MaximumWorkingSetSize;
    DWORD         ActiveProcessLimit;
    DWORD_PTR     Affinity;
    DWORD         PriorityClass;
    DWORD         SchedulingClass;
} JOB_OBJECT_BASIC_LIMIT_INFORMATION, *PJOB_OBJECT_BASIC_LIMIT_INFORMATION;
```

表5-2简单地描述了它的各个成员的情况。

表5-2 JOB_OBJECT_BASIC_LIMIT_INFORMATION结构的成员

成员	描述	说明
PerProcessUserTimeLimit	设定分配给每个进程的用户方式的最大时间（以100ns为间隔时间）	任何进程占用的时间如果超过了分配给它的时间，系统将自动终止它的运行。若要设置这个限制条件，请在LimitFlags成员中设定JOB_OBJECT_LIMIT_PROCESS_TIME
PerJobUserTimeLimit	设定该作业中可以使用多少用户方式的时间(以100ns为间隔时间)	按照默认设置，当达到该时间限制时，系统将自动终止所有进程的运行。可以在作业运行时定期改变这个值。若要设置该限制条件，请在LimitFlags成员中设定JOB_OBJECT_LIMIT_JOB_TIME
LimitFlags	指明哪些限制适用于该作业	详细说明参见本表下面的一段
MinimumWorkingSetSize/MaximumWorkingSetSize	设定每个进程（不是作业中的所有进程）的最小和最大工作区的大小	通常，进程的工作区可能扩大而超过它的最小值。设置MaximumWorkingSetSize后，就可以实施硬限制。一旦进程的工作区达到该限制值，进程就会对此作出页标记。各个进程对SetProcessWorkingSetSize的调用将被忽略，除非该进程只是试图清空它的工作区。若要设置该限制，请在LimitFlags成员中设定JOB_OBJECT_LIMIT_WORKINGSET
ActiveProcessLimit	设定作业中可以同时运行的进	超过这个限制的任何尝试都会导致新进程被迫终止

(续)

成 员	描 述	说 明
	程的最大数量	运行, 并产生一个“定额不足”的错误。若要设置这个限制, 请在 <code>LimitFlags</code> 成员中设定 <code>JOB_OBJECT_LIMIT_ACTIVE_PROCESS</code>
<code>Affinity</code>	设定能够运行的进程的CPU子集	单个进程甚至能够进一步对此作出限制。若要设置这个限制, 请在 <code>LimitFlags</code> 成员中设定 <code>JOB_OBJECT_LIMIT_AFFINITY</code>
<code>PriorityClass</code>	设定所有进程使用的优先级	如果进程调用 <code>SetPriorityClass</code> 函数, 即使该函数调用失败, 它也能成功地返回。如果进程调用 <code>GetPriorityClass</code> 函数, 该函数将返回进程已经设置的优先级类, 尽管这可能不是进程的实际优先级类。此外, <code>SetThreadPriority</code> 无法将线程的优先级提高到正常的优先级之上, 不过它可以用于降低线程的优先级。若要设置这个限制, 请在 <code>LimitFlags</code> 成员中设定 <code>JOB_OBJECT_LIMIT_PRIORITY_CLASS</code>
<code>SchedulingClass</code>	设定分配给作业中的线程的相对时段差	它的值可以在0到9之间(包括0和9), 默认值是5。详细说明参见本表后面的文字。若要设置这个限制, 请在 <code>LimitFlags</code> 成员中设定 <code>JOB_OBJECT_LIMIT_SCHEDULING_CLASS</code>

关于这个结构的某些问题在 Platform SDK 文档中并没有说清楚, 因此在这里作一些说明。你在 `LimitFlags` 成员中设置了一些信息, 来指明想用于作业的限制条件。我设置了 `JOB_OBJECT_LIMIT_PRIORITY_CLASS` 和 `JOB_OBJECT_LIMIT_JOB_TIME` 这两个标志。这意味着它们是我用于该作业的唯一两个限制条件。我没有对 CPU 的亲缘关系、工作区的大小、每个进程占用的 CPU 时间等作出限制。

当作业运行时, 它会维护一些统计信息, 比如作业中的进程已经使用了多少 CPU 时间。每次使用 `JOB_OBJECT_LIMIT_JOB_TIME` 标志来设置基本限制时, 作业就会减去已经终止运行的进程的 CPU 时间的统计信息。这显示当前活动的进程使用了多少 CPU 时间。如果想改变作业运行所在的 CPU 的亲缘关系, 但是没有重置 CPU 时间的统计信息, 那将要如何处理呢? 为了处理这种情况, 必须使用 `JOB_OBJECT_LIMIT_AFFINITY` 标志来设置新的基本限制条件, 并且必须退出 `JOB_OBJECT_LIMIT_JOB_TIME` 标志的设置。这样一来, 就告诉作业, 不再想要使用 CPU 的时间限制。这不是你想要的。

你想要的是改变 CPU 亲缘关系的限制, 保留现有的 CPU 时间限制。你只是不想减去已终止运行的进程的 CPU 时间的统计信息。为了解决这个问题, 可以使用一个特殊标志, 即 `JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME`。这个标志与 `JOB_OBJECT_LIMIT_JOB_TIME` 标志是互斥的。 `JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME` 标志表示你想改变限制条件, 而不减去已经终止运行的进程的 CPU 时间的统计信息。

现在介绍一下 `JOB_OBJECT_BASIC_LIMIT_INFORMATION` 结构的 `SchedulingClass` 成员。假如你有两个正在运行的作业, 你将两个作业的优先级类都设置为 `NORMAL_PRIORITY_CLASS`。但是你还想让一个作业中的进程获得比另一个进程多的 CPU 时间。可以使用 `SchedulingClass` 成员来改变拥有相同优先级的作业的相对调度关系。可以设置一个 0 至 9 之间的值(包括 0 和 9), 5 是默认值。在 Windows 2000 上, 如果这个设置值比较大, 那么系统就会给某个作业的进程中的线程提供较长的 CPU 时间量。如果设置的值比较小, 就减少该线程的 CPU 时间量。

例如，我有两个拥有正常优先级类的作业。每个作业包含一个进程，每个进程只有一个（拥有正常优先级的）线程。在正常环境下，这两个线程将按循环方式进行调度，每个线程获得相同的CPU时间量。但是，如果将第一个作业的 SchedulingClass成员设置为3，那么，当该作业中的线程被安排CPU时间时，它得到的时间量将比第二个作业中的线程少。

如果使用 SchedulingClass成员，应该避免使用大数字即较大的时间量，因为较大的时间量会降低系统中的其他作业、进程和线程的总体响应能力。另外，我只是介绍了 Windows 2000 中的情况。Microsoft 计划在将来的 Windows 版本中对线程调度程序进行更重要的修改，因为它认为操作系统应该为作业、进程和线程提供更宽松的线程调度环境。

需要特别注意的最后一个限制是 JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION限制标志。这个限制可使系统为与作业相关的每个进程关闭“未处理的异常情况”对话框。系统通过调用 SetErrorMode 函数，将作业中的每个进程的 SEM_NOGPFAULTERRORBOX 标志传递给它。作业中产生未处理的异常情况的进程会立即终止运行，不显示任何用户界面。对于服务程序和其他面向批处理的作业来说，这是个非常有用的限制标志。如果没有这个标志，作业中的进程就会产生一个异常情况，并且永远不会终止运行，从而浪费了系统资源。

除了基本限制外，还可以使用 JOBOBJECT_EXTENDED_LIMIT_INFORMATION 结构对作业设置扩展限制：

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

如你所见，该结构包含一个 JOBOBJECT_BASIC_LIMIT_INFORMATION 结构，它构成了基本限制的一个超集。这个结构有点儿特殊，因为它包含的成员与设置作业的限制毫无关系。首先，IoInfo 成员保留不用，无论如何不能访问它。本章后面将要介绍如何查询 I/O 计数器信息。此外，PackProcessMemoryUsed 和 PackJobMemoryUsed 成员是只读成员，分别告诉你作业中的任何一个进程和所有进程需要使用的已确认的内存最大值。

另外两个成员 ProcessMemoryLimit 和 JobMemoryLimit 分别用于限制作业中的任何一个进程和所有进程使用的已确认的内存量。若要设置这些限制值，可以在 LimitFlags 成员中分别设定 JOB_OBJECT_LIMIT_JOB_MEMORY 和 JOB_OBJECT_LIMIT_PROCESS_MEMORY 两个标志。

现在看一下可以对作业设置的另一些限制。下面是 JOBOBJECT_BASIC_UI_RESTRICTIONS 结构的样子：

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

这个结构只有一个数据成员，即 UIRestrictionsClass，它用于存放表 5-3 中简单描述的一组位标志。

最后一个标志 JOB_OBJECT_UILIMIT_HANDLES 是特别有趣的。这个限制意味着作业中没有一个进程能够访问该作业外部的进程创建的 USER 对象。因此，如果试图在作业内部运行 Microsoft Spy++，那么除了 Spy++ 自己创建的窗口外，你看不到任何别的窗口。图 5-1 显示的

Spy++中打开了两个MDI子窗口。注意，Threads 1的窗口包含一个系统中的线程列表。这些线程中只有一个线程，即000006AC SPYXX似乎创建了一些窗口。这是因为我是在它自己的作业中运行Spy++的，并且限制了它对UI句柄的使用。在同一个窗口中，可以看到MSDEV和EXPLORER两个线程，但是看来它们尚未创建任何窗口。可以保证，这些线程肯定创建了窗口，但是Spy++无法访问它们。在对话框的右边，可以看到Windows 3窗口，在这个窗口中，Spy++显示了桌面上存在的所有窗口的层次结构。注意，它只有一个项目，即00000000。Spy++必须将它作为占位符放在这里。

表5-3 用于作业对象的基本用户界面限制的位标志

标 志	描 述
JOB_OBJECT_UILIMIT_EXITWINDOWS	用于防止进程通过ExitWindowsEx函数退出、关闭、重新引导或关闭系统电源
JOB_OBJECT_UILIMIT_READCLIPBOARD	防止进程读取剪贴板的内容
JOB_OBJECT_UILIMIT_WRITECLIPBOARD	防止进程删除剪贴板的内容
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS	防止进程通过SystemParametersInfo函数来改变系统参数
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS	防止进程通过ChangeDisplaySettings函数来改变显示设置
JOB_OBJECT_UILIMIT_GLOBALATOMS	为作业赋予它自己的基本结构表，使作业中的进程只能访问该作业的表
JOB_OBJECT_UILIMIT_DESKTOP	防止进程使用CreateDesktop或SwitchDesktop函数创建或转换桌面
JOB_OBJECT_UILIMIT_HANDLES	防止作业中的进程使用同一作业外部的进程创建的USER对象（如HWND）

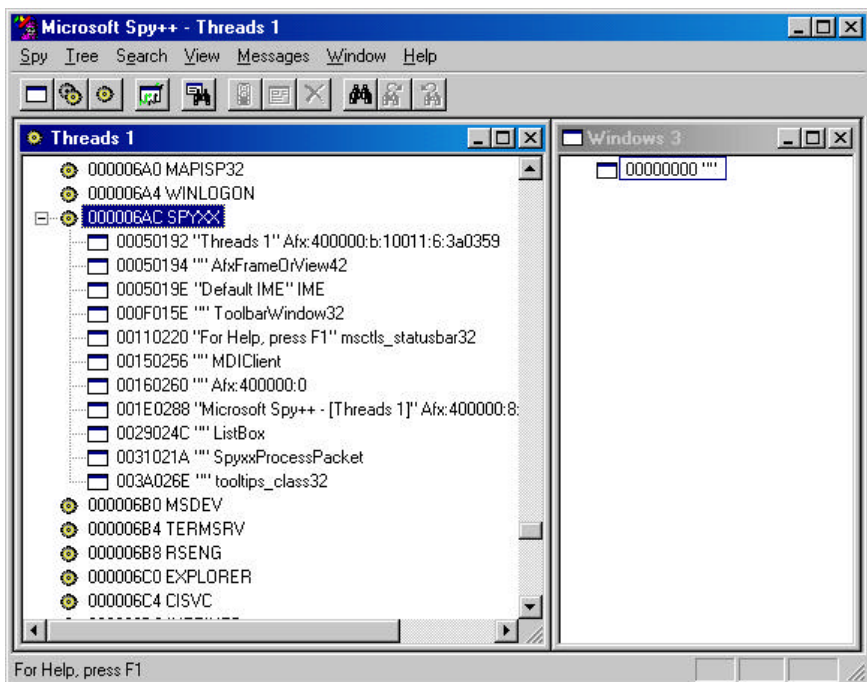


图5-1 在作业中运行的Microsoft Spy++可以限制对UI句柄的访问

注意，这个UI限制是单向的。这就是说，作业外部的进程能够看到作业内部的进程创建的USDR对象。例如，如果我在一个作业中运行Notepad，并在作业的外部运行Spy++，那么，如

果Notepad所在的作业设定了JOB_OBJECT_UILIMIT_HANDLES标志，Spy++将能够看到Notepad的窗口。同样，如果Spy++在它自己的作业中，那么它也可以看到Notepad的窗口，只要它设定了JOB_OBJECT_UILIMIT_HANDLES标志。

如果想为作业进程的操作创建真正的沙框，那么限制UI句柄是可怕的。但是，如果作为作业组成部分的一个进程要与作业外部的进程进行通信，就可以使用这种限制。

实现这个目的有一个简便的方法，那就是使用窗口消息，但是，如果作业的进程不能访问UI句柄，那么作业中的进程就无法将窗口消息发送或显示在作业外部的进程创建的窗口中。不过，可以使用下面这个新函数来解决这个问题：

```
BOOL UserHandleGrantAccess(  
    HANDLE hUserObj,  
    HANDLE hJob,  
    BOOL fGrant);
```

hUserObj参数用于指明一个USER对象，可以为作业中的进程赋予或者撤消对该对象的访问权。它几乎总是一个窗口句柄，但是它可以是另一个USER对象，比如桌面、挂钩、图标或菜单。最后两个参数hJob和fGrant用于指明你赋予或撤消对哪个作业的访问权。注意，如果从hJob标识的作业中的一个进程来调用该函数，该函数的运行就会失败——这可以防止作业中的进程总是为它自己赋予访问一个对象的权限。

对作业施加的最后一种限制类型与安全性相关（注意，一旦使用这种限制，就无法取消安全性限制）。JOB_OBJECT_SECURITY_LIMIT_INFORMATION的结构类似下面的形式：

```
typedef struct _JOB_OBJECT_SECURITY_LIMIT_INFORMATION {  
    DWORD SecurityLimitFlags;  
    HANDLE JobToken;  
    PTOKEN_GROUPS SidsToDisable;  
    PTOKEN_PRIVILEGES PrivilegesToDelete;  
    PTOKEN_GROUPS RestrictedSids;  
} JOB_OBJECT_SECURITY_LIMIT_INFORMATION, *PJOB_OBJECT_SECURITY_LIMIT_INFORMATION;
```

表5-4简单地描述了它的各个成员。

表5-4 JOB_OBJECT_SECURITY_LIMIT_INFORMATION 的成员

成 员	描 述
SecurityLimitFlags	指明是否不允许管理员访问、不允许无限制的标记访问、强制使用特定的访问标记，或者停用某些安全性标识符和优先权
JobToken	作业中的所有进程使用的访问标记
SidsToDisable	指明为访问检查停用哪些SID
PrivilegesToDelete	指明要从访问标记中删除哪些优先权
RestrictedSids	指明应该添加给访问标记的一组仅为拒绝(deny only)的SID

当然，一旦给作业设置了限制条件，就可以查询这些限制。通过调用下面的代码，就可以进行这一操作

```
BOOL QueryInformationJobObject(  
    HANDLE hJob,  
    JOB_OBJECT_INFOCLASS JobObjectInformationClass,  
    PVOID pvJobObjectInformation,  
    DWORD cbJobObjectInformationLength,  
    PDWORD pdwReturnLength);
```

你为该函数传递作业的句柄（就像你对SetInformationJobObject操作时那样），这些句柄包

括用于指明你想要的限制信息的枚举类型，函数要进行初始化的数据结构的地址，以及包含该结构的数据块的长度。最后一个参数是 `pdwReturnLength`，用于指向该函数填写的 `DWORD`，它告诉你有多少字节放入了缓存。如果你愿意的话，可以（并且通常）为该参数传递 `NULL`。

注意 作业中的进程可以调用 `QueryInformationJobObject`，以便通过为作业的句柄参数传递 `NULL`，获取关于该进程所属的作业的信息。由于它使进程能够看到已经对它实施了哪些限制，所以这个函数非常有用。但是，如果为作业句柄参数传递 `NULL`，那么 `SetInformationJobObject` 函数运行就会失败，因为这将允许进程删除对它实施的限制。

5.2 将进程放入作业

上面介绍的是设置和查询限制方面的信息。现在回到 `StartRestrictedProcess` 这个函数的操作上来。当对作业实施一些限制之后，通过调用 `CreateProcess`，生成了一个进程，我想将它放入作业。但是，注意，当调用 `CreateProcess` 时，我使用了 `CREATE_SUSPENDED` 标志。这样，创建了一个新进程，但是不允许它执行任何代码。由于 `Start-RestrictedProcess` 函数是从不属于作业组成部分的进程来执行的，因此子进程也不属于作业的组成部分。如果准备立即允许子进程开始执行代码，那么它将跑出我的沙框，并且能够成功地执行我想限制它做的工作。因此，当创建子进程之后，在我允许它开始运行之前，我必须显式地将该进程放入我新创建的作业，方法是调用下面的代码：

```
BOOL AssignProcessToJobObject(  
    HANDLE hJob,  
    HANDLE hProcess);
```

该函数告诉系统，将该进程（由 `hProcess` 标识）视为现有作业（由 `hJob` 标识）的一部分。注意，该函数只允许将尚未被赋予任何作业的进程赋予一个作业。一旦进程成为一个作业的组成部分，它就不能转到另一个作业，并且不能是无作业的进程。另外，当作为作业的一部分的进程生成另一个进程的时候，新进程将自动成为父作业的组成部分。不过可以用下面的方法改变它的行为特性：

- 打开 `JOB_OBJECT_BASIC_LIMIT_INFORMATION` 的 `LimitFlags` 成员中的 `JOB_OBJECT_BREAKAWAY_OK` 标志，告诉系统，新生成的进程可以在作业外部运行。若要做到这一点，必须用新的 `CREATE_BREAKAWAY_FROM_JOB` 标志来调用 `CreateProcess`。如果用 `CREATE_BREAKAWAY_FROM_JOB` 标志调用 `CreateProcess` 函数，但是该作业并没有打开 `CREATE_BREAKAWAY_FROM_JOB` 这个标志，那么 `CreateProcess` 函数运行就会失败。如果新生成的进程也能控制作业，那么这个机制是有用的。
- 打开 `JOB_OBJECT_BASIC_LIMIT_INFORMATION` 的 `LimitFlags` 成员中的 `JOB_OBJECT_SILENT_BREAKAWAY_OK` 标志。该标志也告诉系统，新生成的进程不应该是作业的组成部分。但是没有必要将任何其他标志传递给 `CreateProcess`。实际上，该标志将使新进程不能成为作业的组成部分。该标志可以用于原先对作业对象一无所知的进程。

至于 `StartRestrictedProcess` 函数，当调用 `AssignProcessToJobObject` 后，新进程就成为受限的作业的组成部分。然后调用 `ResumeThread`，这样，进程的线程就可以在作业的限制下执行代码。这时，也可以关闭线程的句柄，因为不再需要它了。

5.3 终止作业中所有进程的运行

当然，想对作业进行的最经常的操作是撤消作业中的所有进程。本章开头讲过，Developer

Studio没有配备任何便于使用的方法，来停止进程中的某个操作，因为它不知道哪个进程是由第一个进程生成的（这非常复杂。我在 Microsoft Systems Journal 期刊1998年9月号上 Win32 问与答栏中介绍了 Developer Studio 是如何做到这一点的）。我认为，Developer Studio 的将来版本将会改用作业来进行操作，因为代码的编写要容易得多，可以用它做更多的工作。

若要撤消作业中的进程，只需要调用下面的代码：

```
BOOL TerminateJobObject(
    HANDLE hJob,
    UINT uExitCode);
```

这类似为作业中的每个进程调用 TerminateProcess 函数，将它们的所有退出代码设置为 uExitCode。

5.4 查询作业统计信息

前面已经介绍了如何使用 QueryInformationJobObject 函数来获取对作业的当前限制信息。也可以使用它来获取关于作业的统计信息。例如，若要获取基本的统计信息，可以调用 QueryInformationJobObject，为第二个参数传递 JobObjectBasicAccountingInformation，并传递 JOBOBJECT_BASIC_ACCOUNTING_INFORMATION 结构的地址：

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
    LARGE_INTEGER TotalUserTime;
    LARGE_INTEGER TotalKernelTime;
    LARGE_INTEGER ThisPeriodTotalUserTime;
    LARGE_INTEGER ThisPeriodTotalKernelTime;
    DWORD TotalPageFaultCount;
    DWORD TotalProcesses;
    DWORD ActiveProcesses;
    DWORD TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION,
*PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

表5-5简要描述了它的各个成员。

表5-5 JOBOBJECT_BASIC_ACCOUNTING_INFORMATION 的成员

成 员	描 述
TotalUserTime	设定作业中的进程已经使用多少用户方式 CPU 时间
TotalKernelTime	设定作业中的进程已经使用多少内核方式 CPU 时间
ThisPeriodTotalUserTime	与 TotalUserTime 的作用相同，差别是，当调用 SetInformation-JobObject 以便改变基本限制信息并且不使用 JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME 限制标记时，本值重置为 0
ThisPeriodTotalKernelTime	与 ThisPeriodTotalUserTime 相同，差别是，本值显示的是内核方式时间
TotalPageFaultCount	设定作业中的进程已经产生的页面故障数量
TotalProcesses	设定曾经成为作业组成部分的进程总数
ActiveProcesses	设定当前作为作业的组成部分的进程的数量
TotalTerminatedProcesses	设定由于超过分配给它们的 CPU 时间限制而被撤消的进程的数量

除了查询这些基本统计信息外，可以进行一次函数调用，以同时查询基本统计信息和 I/O 统计信息。为此，必须为第二个参数传递 JobObjectBasicAndIoAccountingInformation，并传递 JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION 结构的地址：

```
typedef struct JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
```

```
IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

如你所见，这个结构只返回一个 JOBOBJECT_BASIC_ACCOUNTING_INFORMATION 结构和 IO_COUNTERS 结构：

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

该结构告诉你作业中的进程已经执行的读、写和非读 / 写操作的数量（以及在这些操作期间传送的字节数）。另外，可以使用下面这个新的 GetProcessIoCounters 函数，以便获取不是这些作业中的进程的这些信息：

```
BOOL GetProcessIoCounters(
    HANDLE hProcess,
    PIO_COUNTERS pIoCounters);
```

也可以随时调用 QueryInformationJobObject 函数，以便获取当前在作业中运行的进程的一组进程 ID。若要进行这项操作，首先必须确定你想在作业中看到多少进程，然后必须分配足够的内存块，来放置这些进程 ID 的数组，并指定 JOBOBJECT_BASIC_PROCESS_ID_LIST 结构的大小：

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    DWORD ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

因此，若要获得当前作业中的一组进程 ID，必须执行类似下面的代码：

```
void EnumProcessIdsInJob(HANDLE hjob) {

    // I assume that there will never be more
    // than 10 processes in this job.
    #define MAX_PROCESS_IDS    10

    // Calculate the number of bytes needed for structure & process IDs.
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (MAX_PROCESS_IDS - 1) * sizeof(DWORD);

    // Allocate the block of memory.
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil = _alloca(cb);

    // Tell the function the maximum number of processes
    // that we allocated space for.
    pjobpil->NumberOfAssignedProcesses = MAX_PROCESS_IDS;

    // Request the current set of process IDs.
    QueryInformationJobObject(hjob, JobObjectBasicProcessIdList,
        pjobpil, cb, &cb);

    // Enumerate the process IDs.
```

```
for (int x = 0; x < pjobpil->NumberOfProcessIdsInList; x++) {  
    // Use pjobpil->ProcessIdList[x]...  
}  
  
// Since _alloca was used to allocate the memory,  
// we don't need to free it here.  
}
```

这就是你使用这些函数的所有实现方法，不过操作系统实际上保存了更多的关于作业的信息。它是使用性能计数器来进行这项操作的。可以使用 Performance Data Helper 函数库 (PDH.dll) 中的函数来检索这些信息。也可以使用 Microsoft Management Console (MMC) 的 Performance Monitor Snap-In 来查看作业信息。图 5-2 显示了系统中的作业对象可以使用的一些计数器。图 5-3 显示了可以使用的一些作业对象的明细计数器。也可以看到 Jeff 的作业里有 4 个进程，即 calc、cmd、notepad 和 wordpad。

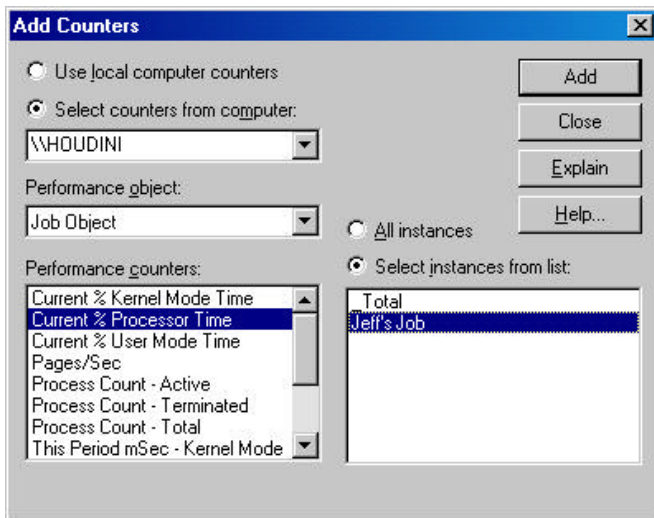


图5-2 MMC的性能监控器：作业对象计数器

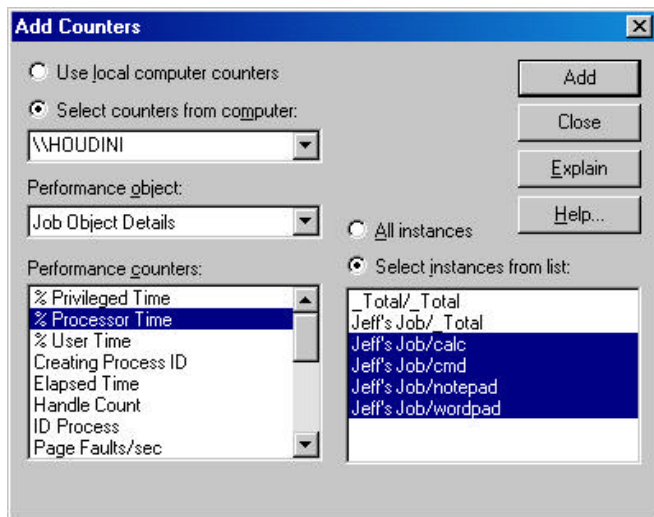


图5-3 MMC的性能监控器：作业对象明细计数器

注意，当调用 `CreateJobObject` 函数时，只能为已经赋予名字的作业获取性能计数器信息。由于这个原因，即使不打算按名字来共享跨越进程的作业对象，也应该创建带有名字的这些对象。

5.5 作业通知信息

现在，已经知道了关于作业对象的基本知识，剩下要介绍的内容是关于通知的问题。例如，是否想知道作业中的所有进程何时终止运行或者分配的全部 CPU 时间是否已经到期呢？也许想知道作业中何时生成新进程或者作业中的进程何时终止运行。如果不关心这些通知信息（而且许多应用程序也不关心这些信息），作业的操作非常容易。如果关心这些事件，那么还有一些工作要做。

如果关心的是分配的所有 CPU 时间是否已经到期，那么可以非常容易地得到这个通知信息。当作业中的进程尚未用完分配的 CPU 时间时，作业对象就得不到通知。一旦分配的所有 CPU 时间已经用完，Windows 就强制撤消作业中的所有进程，并将情况通知作业对象。通过调用 `WaitForSingleObject`（或类似的函数），可以很容易跟踪这个事件。有时，可以在晚些时候调用 `SetInformationJobObject` 函数，使作业对象恢复未通知状态，并为作业赋予更多的 CPU 时间。

当开始对作业进行操作时，我觉得当作业中没有任何进程运行时，应该将这个事件通知作业对象。毕竟当进程和线程停止运行时，进程和线程对象就会得到通知。因此，当作业停止运行时它也应该得到通知。这样，就能够很容易确定作业何时结束运行。但是，Microsoft 选择在分配的 CPU 时间到期时才向作业发出通知，因为这显示了一个错误条件。由于许多作业启动时有一个父进程始终处于工作状态，直到它的所有子进程运行结束，因此只需要在父进程的句柄上等待，就可以了解整个作业何时运行结束。`StartRestrictedProcess` 函数用于显示分配给作业的 CPU 时间何时到期，或者作业中的进程何时终止运行。

前面介绍了如何获得某些简单的通知信息，但是尚未说明如何获得更高级的通知信息，如进程创建/终止运行等。如果想要得到这些通知信息，必须将更多的基础结构放入应用程序。特别是，必须创建一个 I/O 完成端口内核对象，并将作业对象或多个作业对象与完成端口关联起来。然后，必须让一个或多个线程在完成端口上等待作业通知的到来，这样它们才能得到处理。

一旦创建了 I/O 完成端口，通过调用 `SetInformationJobObject` 函数，就可以将作业与该端口关联起来，如下面的代码所示：

```
JOB_OBJECT_ASSOCIATE_COMPLETION_PORT joacp;  
joacp.CompletionKey = 1; // Any value to uniquely identify this job  
joacp.CompletionPort = hIOCP; // Handle of completion port that  
// receives notifications  
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortInformation,  
    &joacp, sizeof(joacp));
```

当上面的代码运行时，系统将监视该作业的运行，当事件发生时，它将事件送往 I/O 完成端口（顺便说一下，可以调用 `QueryInformationJobObject` 函数来检索完成关键字和完成端口句柄。但是，这样做的机会很少）。线程通过调用 `GetQueuedCompletionStatus` 函数来监控 I/O 完成端口：

```
BOOL GetQueuedCompletionStatus(  
    HANDLE hIOCP,  
    PDWORD pNumBytesTransferred,
```

```
PULONG_PTR pCompletionKey,
POVERLAPPED *pOverlapped,
DWORD dwMilliseconds);
```

当该函数返回一个作业事件通知时，*pCompletionKey包含了调用SetInformationJobObject时设置的完成关键字值，用于将作业与完成端口关联起来。它使你能够知道哪个作业存在一个事件。*pNumBytesTransferred中的值用于指明发生了哪个事件。根据事件（见表5-6），*pOverlapped 中的值将指明一个进程ID。

表5-6 系统可以发送给作业的相关完成端口的作业事件通知

事 件	描 述
JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO	当作业中没有进程运行时发送
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	当超过分配给进程的 CPU时间时发送。进程终止运行，并赋予进程的ID
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	当试图超过作业中运行的进程数量时发送
JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT	当进程试图占用超过限额的内存时发送。给出进程的ID
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	当进程试图占用的内存超过作业的内存限制时发送。给出进程的ID
JOB_OBJECT_MSG_NEW_PROCESS	当一个进程添加给作业时发送。给出进程的ID
JOB_OBJECT_MSG_EXIT_PROCESS	当进程终止运行时发送。给出进程的ID
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	当进程由于未处理的异常事件而终止运行时发送。给出进程的ID
JOB_OBJECT_MSG_END_OF_JOB_TIME	当超过分配给作业的CPU时间时发送。这些进程没有终止运行。可以允许它们继续运行，设置一个新的时间限制，或者自己调用TerminateJobObject函数

最后要说明的一点是，按照默认设置，作业对象是这样配置的：当分配给作业的 CPU时间已经到期时，作业的所有进程均自动停止运行，而 JOB_OBJECT_MSG_END_OF_JOB_TIME 通知尚未发送。如果想要防止作业对象撤消进程而只是通知你时间已经超过，必须执行下面这样的代码：

```
// Create a JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure
// and initialize its only member.
JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti;
joeojti.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;

// Tell the job object what we want it to do when the job time is
// exceeded.
SetInformationJobObject(hJob, JobObjectEndOfJobTimeInformation,
    &joeojti, sizeof(joeojti));
```

为作业设定结束时间而使用的另一个值是 JOB_OBJECT_TERMINATE_AT_END_OF_JOB，这是作业创建时的默认值。

5.6 JobLab示例应用程序

JobLab应用程序“05JobLab.exe”（在本章末尾处清单5-2中列出）使你能够很容易地对作业进行实验性操作。该应用程序的源代码和资源文件放在本书所附光盘上 05-JobLab目录中。当启动该程序时，出现图5-4所示的窗口。

当进程被初始化时，它创建一个作业对象。我创建的这个作业对象的名字是JobLab，这样，就可以使用MMC的Performance Monitor Snap-In来观察和监控它的性能。该应用程序还创建了

一个I/O完成端口，并将作业对象与它相关联。这样就可以对来自作业的通知进行监控，并可显示在窗口底部的列表框中。

开始时，该作业不包含进程，也没有各种限制条件。顶部的各个域用于设定对作业的基本限制和扩展限制条件。要做的工作是用有效值填写这些域，然后点击 Apply Limits按钮。如果将一个域置空，那么该限制条件就不起作用。除了基本限制和扩展限制条件外，还可以打开和关闭各种UI限制。注意，Preserve Job Time When Applying Limits（当运用各个限制时保留作业时间）复选框并不用于设置限制条件。它只是让你在查询基本统计信息时可以改变作业的限制条件，而不重置 ThisPeriodTotalUserTime和ThisPeriod-TotalKernelTime成员。当运用单个作业的时间限制时，该复选框不起作用。

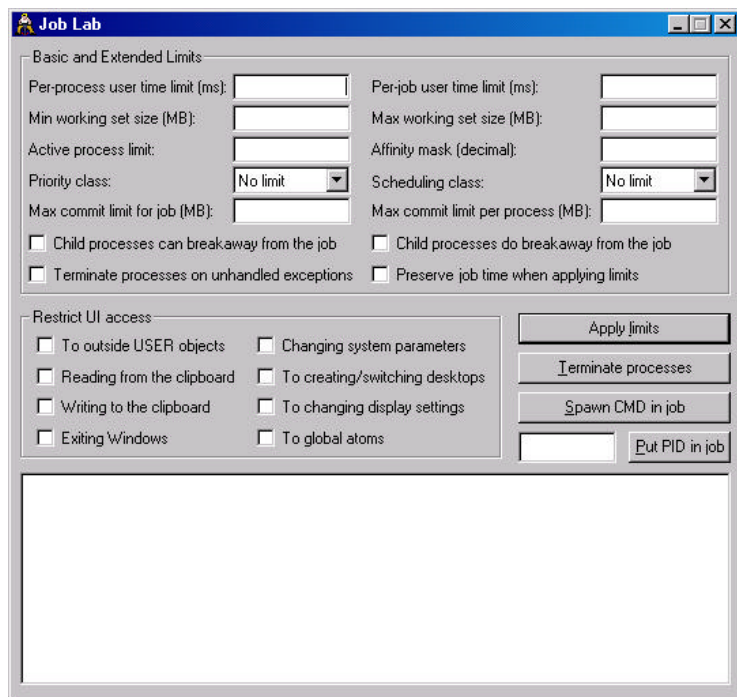


图5-4 JobLab示例应用程序

其余的按钮供你用其他方式对作业进行操作。Terminate Processes按钮用于撤消作业中的所有进程。Spawn CMD In Job按钮用于生成与作业相关的命令外壳进程。从该命令外壳程序中，可以生成更多的子进程，并且可以看到它们如何作为作业的组成部分来运行。我发现这对试验操作是非常有用的。最后一个按钮是 Put PID In Job，它用于将现有的无作业进程与作业相关联。

窗口底部的列表框显示了更新的关于作业的状态信息。每隔 10s，该窗口显示一次基本统计信息和I/O统计信息，以及进程/作业的内存峰值使用量。同时也显示作业中当前的每个进程的ID。

最后要说明的是，如果修改了源代码，并且创建一个没有名字的作业内核对象，那么可以运行该应用程序的多个拷贝，以便在同一台机器上创建两个或多个作业对象，并且进行更多的试验。

就源代码而言，没有什么特殊的東西需要介绍，因为源代码已经做了非常完善的说明。不过，我创建了一个 Job.h 文件，它定义了一个 Cjob C++ 类，用于封装操作系统的作业对象。这

使得操作起来更加容易，因为不必到处传递作业的句柄。这个类还减少了平常调用 Query InformationJobObject和 SetInformationJobObject函数时需要进行的转换工作量。

清单5-2 JobLab示例应用程序



JobLab.cpp

```

//*****
Module: JobLab.cpp
Notes: Copyright (c) 2000 Jeffrey Richter
*****

#include "JobLab.h"
#include <windows.h>
#include <process.h> // for _beginthreadex
#include <string.h>
#include <stdio.h>
#include "resource.h"
#include "job.h"

//*****

JOB g_job; // Job object

HWND g_hwnd; // Handle to dialog box (accessible by all threads)

HANDLE g_hIOCP; // Completion port that receives job notifications
HANDLE g_hInfoThread; // Completion port thread

// Completion keys for the completion port
#define COMPKEY_TERMINATE (DWORD_PTR) 0
#define COMPKEY_UPDATE (DWORD_PTR) 1
#define COMPKEY_INFORMATION (DWORD_PTR) 2

//*****

DWORD WINAPI JobNotify(PVOID) {
    TCHAR sz[2000];
    BOOL fDone = FALSE;

    while (!fDone) {
        DWORD dwBytesXferred;
        ULONG_PTR CompKey;
        LPOVERLAPPED po;
        GetQueuedCompletionStatus(g_hIOCP,
            &dwBytesXferred, &CompKey, &po, INFINITE);

        // The app is shutting down, exit this thread
        fDone = (CompKey == COMPKEY_TERMINATE);

        HWND hwndLB = FindWindow(NULL, TEXT("Job Lab"));
    }
}

```

```

hwndLB = GetDlgItem(hwndLB, IDC_STATUS);

if (CompKey == COMPKEY_JOBOBJECT) {
    lstrcpy(sz, TEXT("--> Notification: "));
    LPTSTR psz = sz + lstrlen(sz);
    switch (dwBytesXferred) {
        case JOB_OBJECT_MSG_END_OF_JOB_TIME:
            wsprintf(psz, TEXT("Job time limit reached"));
            break;

        case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
            wsprintf(psz, TEXT("Job process (Id=%d) time limit reached"), po);
            break;

        case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
            wsprintf(psz, TEXT("Too many active processes in job"));
            break;

        case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
            wsprintf(psz, TEXT("Job contains no active processes"));
            break;

        case JOB_OBJECT_MSG_NEW_PROCESS:
            wsprintf(psz, TEXT("New process (Id=%d) in Job"), po);
            break;

        case JOB_OBJECT_MSG_EXIT_PROCESS:
            wsprintf(psz, TEXT("Process (Id=%d) terminated"), po);
            break;

        case JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS:
            wsprintf(psz, TEXT("Process (Id=%d) terminated abnormally"), po);
            break;

        case JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT:
            wsprintf(psz, TEXT("Process (Id=%d) exceeded memory limit"), po);
            break;

        case JOB_OBJECT_MSG_JOB_MEMORY_LIMIT:
            wsprintf(psz,
                TEXT("Process (Id=%d) exceeded job memory limit"), po);
            break;

        default:
            wsprintf(psz, TEXT("Unknown notification: %d"), dwBytesXferred);
            break;
    }
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
    CompKey = 1;    // Force a status update when a notification arrives
}

if (CompKey == COMPKEY_STATUS) {

    static int s_nStatusCount = 0;
    _stprintf(sz, TEXT("--> Status Update (%u)"), s_nStatusCount++);
}

```

```

        jobai.BasicInfo.ActiveProcesses,
        jobai.BasicInfo.TotalTerminatedProcesses);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // Show the I/O accounting information
    _stprintf(sz, TEXT("Reads=%I64u (%I64u bytes), ")
        TEXT("Write=%I64u (%I64u bytes), Other=%I64u (%I64u bytes)"),
        jobai.IOInfo.ReadOperationCount, jobai.IOInfo.ReadTransferCount,
        jobai.IOInfo.WriteOperationCount, jobai.IOInfo.WriteTransferCount,
        jobai.IOInfo.OtherOperationCount, jobai.IOInfo.OtherTransferCount);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // Show the peak per-process and job memory usage
    JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli;
    g_job.QueryExtendedLimitInfo(&joeli);
    _stprintf(sz, TEXT("Peak memory used: Process=%I64u, Job=%I64u"),
        (__int64) joeli.PeakProcessMemoryUsed,
        (__int64) joeli.PeakJobMemoryUsed);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // Show the set of Process IDs
    DWORD dwNumProcesses = 50, dwProcessIdList[50];
    g_job.QueryBasicProcessIdList(dwNumProcesses,
        dwProcessIdList, &dwNumProcesses);
    _stprintf(sz, TEXT("PIDs: %s"),
        (dwNumProcesses == 0) ? TEXT("(none)") : TEXT(""));
    for (DWORD x = 0; x < dwNumProcesses; x++) {
        _stprintf(_tcschr(sz, 0), TEXT("%d "), dwProcessIdList[x]);
    }
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
}
}
return(0);
}

```

////////////////////////////////////

```

BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_JOBLAB);

    // Save our window handle so that the completion port thread can access it
    g_hwnd = hwnd;

    HWND hwndPriorityClass = GetDlgItem(hwnd, IDC_PRIORITYCLASS);
    ComboBox_AddString(hwndPriorityClass, TEXT("No limit"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Idle"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Below normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Above normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("High"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Realtime"));
    ComboBox_SetCurSel(hwndPriorityClass, 0); // Default to "No Limit"

    HWND hwndSchedulingClass = GetDlgItem(hwnd, IDC_SCHEDULINGCLASS);
    ComboBox_AddString(hwndSchedulingClass, TEXT("No limit"));
    for (int n = 0; n <= 9; n++) {
        TCHAR szSchedulingClass[2] = { (TCHAR) (TEXT('0') + n), 0 };
        ComboBox_AddString(hwndSchedulingClass, szSchedulingClass);
    }
    ComboBox_SetCurSel(hwndSchedulingClass, 0); // Default to "No Limit"
    SetTimer(hwnd, 1, 10000, NULL);           // 10 second accounting update
    return(TRUE);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void Dlg_ApplyLimits(HWND hwnd) {
    const int nNanosecondsPerSecond = 1000000000;
    const int nMillisecondsPerSecond = 1000;
    const int nNanosecondsPerMillisecond =
        nNanosecondsPerSecond / nMillisecondsPerSecond;
    BOOL f;
    __int64 q;
    SIZE_T s;
    DWORD d;

    // Set Basic and Extended Limits
    JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli = { 0 };
    joeli.BasicLimitInformation.LimitFlags = 0;

    q = GetDlgItemInt(hwnd, IDC_PERPROCESSUSERTIMELIMIT, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PROCESS_TIME;
        joeli.BasicLimitInformation.PerProcessUserTimeLimit.QuadPart =
            q * nNanosecondsPerMillisecond / 100;
    }

    q = GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_TIME;
        joeli.BasicLimitInformation.PerJobUserTimeLimit.QuadPart =
            q * nNanosecondsPerMillisecond / 100;
    }
}

```

```
s = GetDlgItemInt(hwnd, IDC_MINWORKINGSETSIZE, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_WORKINGSET;
    joeli.BasicLimitInformation.MinimumWorkingSetSize = s * 1024 * 1024;
    s = GetDlgItemInt(hwnd, IDC_MAXWORKINGSETSIZE, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.MaximumWorkingSetSize = s * 1024 * 1024;
    } else {
        joeli.BasicLimitInformation.LimitFlags &=~JOB_OBJECT_LIMIT_WORKINGSET;
        chMB("Both minimum and maximum working set sizes must be set.\n"
            "The working set limits will NOT be in effect.");
    }
}

d = GetDlgItemInt(hwnd, IDC_ACTIVEPROCESSLIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_ACTIVE_PROCESS;
    joeli.BasicLimitInformation.ActiveProcessLimit = d;
}

s = GetDlgItemInt(hwnd, IDC_AFFINITYMASK, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_AFFINITY;
    joeli.BasicLimitInformation.Affinity = s;
}

joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PRIORITY_CLASS;
switch (ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_PRIORITYCLASS))) {
    case 0:
        joeli.BasicLimitInformation.LimitFlags &=
            ~JOB_OBJECT_LIMIT_PRIORITY_CLASS;
        break;

    case 1:
        joeli.BasicLimitInformation.PriorityClass =
            IDLE_PRIORITY_CLASS;
        break;

    case 2:
        joeli.BasicLimitInformation.PriorityClass =
            BELOW_NORMAL_PRIORITY_CLASS;
        break;

    case 3:
        joeli.BasicLimitInformation.PriorityClass =
            NORMAL_PRIORITY_CLASS;
        break;

    case 4:
        joeli.BasicLimitInformation.PriorityClass =
            ABOVE_NORMAL_PRIORITY_CLASS;
        break;

    case 5:
        joeli.BasicLimitInformation.PriorityClass =
            HIGH_PRIORITY_CLASS;
        break;
}
```



```

case 6:
    joeli.BasicLimitInformation.PriorityClass =
        REALTIME_PRIORITY_CLASS;
    break;
}

int nSchedulingClass =
    ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_SCHEDULINGCLASS));
if (nSchedulingClass > 0) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SCHEDULING_CLASS;
    joeli.BasicLimitInformation.SchedulingClass = nSchedulingClass - 1;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERJOB, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_MEMORY;
    joeli.JobMemoryLimit = s * 1024 * 1024;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERPROCESS, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_PROCESS_MEMORY;
    joeli.ProcessMemoryLimit = s * 1024 * 1024;
}

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESCANBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESDOBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_TERMINATEPROCESSONEXCEPTIONS))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION;

f = g_job.SetExtendedLimitInfo(&joeli,
    ((joeli.BasicLimitInformation.LimitFlags & JOB_OBJECT_LIMIT_JOB_TIME)
    != 0) ? FALSE :
    IsDlgButtonChecked(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS));
chASSERT(f);

// Set UI Restrictions
DWORD jobuir = JOB_OBJECT_UILIMIT_NONE; // A fancy zero (0)
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTACCESSTOOUTSIDEUSEROBJECTS))
    jobuir |= JOB_OBJECT_UILIMIT_HANDLES;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTREADINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_READCLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTWRITINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_WRITECLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTEXITWINDOW))
    jobuir |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

```

```

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTCHANGINGSYSTEMPARAMETERS))
    jobuir |= JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDESKTOPS))
    jobuir |= JOB_OBJECT_UILIMIT_DESKTOP;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDISPLAYSETTINGS))
    jobuir |= JOB_OBJECT_UILIMIT_DISPLAYSETTINGS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTGLOBALATOMS))
    jobuir |= JOB_OBJECT_UILIMIT_GLOBALATOMS;

chVERIFY(g_job.SetBasicUIRestrictions(jobuir));
}

////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    switch (id) {
        case IDCANCEL:
            // User is terminating our app, kill the job too.
            KillTimer(hwnd, 1);
            g_job.Terminate(0);
            EndDialog(hwnd, id);
            break;

        case IDC_PERJOBUSERTIMELIMIT:
            {
                // The job time must be reset if setting a job time limit
                BOOL f;
                GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
                EnableWindow(
                    GetDlgItem(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS), !f);
            }
            break;

        case IDC_APPLYLIMITS:
            Dlg_ApplyLimits(hwnd);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_TERMINATE:
            g_job.Terminate(0);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_SPAWNCMDINJOB:
            {
                // Spawn a command shell and place it in the job
                STARTUPINFO si = { sizeof(si) };
                PROCESS_INFORMATION pi;
                TCHAR sz[] = TEXT("CMD");
                CreateProcess(NULL, sz, NULL, NULL,
                    FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
                g_job.AssignProcess(pi.hProcess);
                ResumeThread(pi.hThread);
                CloseHandle(pi.hProcess);
            }
    }
}

```

```

        CloseHandle(pi.hThread);
    }
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
    break;
case IDC_ASSIGNPROCESSTOJOB:
    {
        DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
        HANDLE hProcess = OpenProcess(
            PROCESS_SET_QUOTA | PROCESS_TERMINATE, FALSE, dwProcessId);
        if (hProcess != NULL) {
            chVERIFY(g_job.AssignProcess(hProcess));
            CloseHandle(hProcess);
        } else chMB("Could not assign process to job.");
    }
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
    break;
}
}

////////////////////////////////////

void WINAPI Dlg_OnTimer(HWND hwnd, UINT id) {

    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
}

////////////////////////////////////

INT_PTR WINAPI Dlg_Proc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_TIMER,      Dlg_OnTimer);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
    }

    return(FALSE);
}

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    // Create the completion port that receives job notifications
    g_hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

    // Create a thread that waits on the completion port
    g_hThreadIOCP = chBEGINTHREDEX(NULL, 0, JobNotify, NULL, 0, NULL);
    // Create the job object
    g_job.Create(NULL, TEXT("JobLab"));
    g_job.SetEndOfJobInfo(JOB_OBJECT_POST_AT_END_OF_JOB);
    g_job.AssociateCompletionPort(g_hIOCP, COMPKEY_JOBOBJECT);

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_JOBLAB), NULL, Dlg_Proc);

    // Post a special key that tells the completion port thread to terminate

```

```

PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_TERMINATE, NULL);

// Wait for the completion port thread to terminate
WaitForSingleObject(g_hThreadIOCP, INFINITE);

// Clean up everything properly
CloseHandle(g_hIOCP);
CloseHandle(g_hThreadIOCP);

// NOTE: The job is closed when the g_job's destructor is called.
return(0);
}

//////////////////////////////////// End Of File //////////////////////////////////////

```

Job.h

```

/*****
Module: Job.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#pragma once

////////////////////////////////////

#include <malloc.h> // for _alloca

////////////////////////////////////

class CJob {
public:
    CJob(HANDLE hJob = NULL);
    ~CJob();

    operator HANDLE() const { return(m_hJob); }

    // Functions to create/open a job object
    BOOL Create(LPSECURITY_ATTRIBUTES psa = NULL, LPCTSTR pszName = NULL);
    BOOL Open(LPCTSTR pszName, DWORD dwDesiredAccess,
        BOOL fInheritHandle = FALSE);

    // Functions that manipulate a job object
    BOOL AssignProcess(HANDLE hProcess);
    BOOL Terminate(UINT uExitCode = 0);

    // Functions that set limits/restrictions on the job
    BOOL SetExtendedLimitInfo(PJOB_OBJECT_EXTENDED_LIMIT_INFORMATION pjoeli,
        BOOL fPreserveJobTime = FALSE);
    BOOL SetBasicUIRestrictions(DWORD fdwLimits);
    BOOL GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant = TRUE);
    BOOL SetSecurityLimitInfo(PJOB_OBJECT_SECURITY_LIMIT_INFORMATION pjosli);

```

```
// Functions that query job limits/restrictions
BOOL QueryExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli);
BOOL QueryBasicUIRestrictions(PDWORD pfdwRestrictions);
BOOL QuerySecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// Functions that query job status information
BOOL QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai);
BOOL QueryBasicProcessIdList(DWORD dwMaxProcesses,
    PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned = NULL);

// Functions that set/query job event notifications
BOOL AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey);
BOOL QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp);
BOOL SetEndOfJobInfo(
    DWORD fdwEndOfJobInfo = JOB_OBJECT_TERMINATE_AT_END_OF_JOB);
BOOL QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo);
```

```
private:
```

```
    HANDLE m_hJob;
```

```
};
```

```
////////////////////////////////////
```

```
inline CJob::CJob(HANDLE hJob) {
```

```
    m_hJob = hJob;
```

```
}
```

```
////////////////////////////////////
```

```
inline CJob::~CJob() {
```

```
    if (m_hJob != NULL)
```

```
        CloseHandle(m_hJob);
```

```
}
```

```
////////////////////////////////////
```

```
inline BOOL CJob::Create(PSECURITY_ATTRIBUTES psa, PCTSTR pszName) {
```

```
    m_hJob = CreateJobObject(psa, pszName);
```

```
    return(m_hJob != NULL);
```

```
}
```

```
////////////////////////////////////
```

```
inline BOOL CJob::Open(
```

```

PCTSTR pszName, DWORD dwDesiredAccess, BOOL fInheritHandle) {

    m_hJob = OpenJobObject(dwDesiredAccess, fInheritHandle, pszName);
    return(m_hJob != NULL);
}

////////////////////////////////////

inline BOOL CJob::AssignProcess(HANDLE hProcess) {

    return(AssignProcessToJobObject(m_hJob, hProcess));
}

////////////////////////////////////

inline BOOL CJob::AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey) {

    JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp = { (PVOID) CompKey, hIOCP };
    return(SetInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, &joacp, sizeof(joacp)));
}

////////////////////////////////////

inline BOOL CJob::SetExtendedLimitInfo(
    JOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli, BOOL fPreserveJobTime) {

    if (fPreserveJobTime)
        pjoeli->BasicLimitInformation.LimitFlags |=
            JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME;

    // If we are to preserve the job's time information,
    // the JOB_OBJECT_LIMIT_JOB_TIME flag must not be on
    const DWORD fdwFlagTest =
        (JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME | JOB_OBJECT_LIMIT_JOB_TIME);

    if ((pjoeli->BasicLimitInformation.LimitFlags & fdwFlagTest)
        == fdwFlagTest) {
        // These flags are mutually exclusive but both are on, error
        DebugBreak();
    }

    return(SetInformationJobObject(m_hJob,
        JobObjectExtendedLimitInformation, pjoeli, sizeof(*pjoeli)));
}

////////////////////////////////////

inline BOOL CJob::SetBasicUIRestrictions(DWORD fdwLimits) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir = { fdwLimits };
    return(SetInformationJobObject(m_hJob,

```



```

        JobObjectBasicUIRestrictions, &jobuir, sizeof(jobuir)));
    }

    //////////////////////////////////////

inline BOOL CJob::SetEndOfJobInfo(DWORD fdwEndOfJobInfo) {
    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti = { fdwEndOfJobInfo };
    joeojti.EndOfJobTimeAction = fdwEndOfJobInfo;
    return(SetInformationJobObject(m_hJob,
        JobObjectEndOfJobTimeInformation, &joeojti, sizeof(joeojti)));
}

    //////////////////////////////////////

inline BOOL CJob::SetSecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

    return(SetInformationJobObject(m_hJob,
        JobObjectSecurityLimitInformation, pjosli, sizeof(*pjosli)));
}

    //////////////////////////////////////

inline BOOL CJob::QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, pjoacp, sizeof(*pjoacp),
        NULL));
}

    //////////////////////////////////////

inline BOOL CJob::QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectBasicAndIoAccountingInformation, pjobai, sizeof(*pjobai),
        NULL));
}

    //////////////////////////////////////

inline BOOL CJob::QueryExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli) {

    return(QueryInformationJobObject(m_hJob, JobObjectExtendedLimitInformation,

```

```

        pjoeli, sizeof(*pjoeli), NULL));
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

inline BOOL CJob::QueryBasicProcessIdList(DWORD dwMaxProcesses,
    PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned) {

    // Calculate the # of bytes necessary
    DWORD cb = sizeof(JOB_OBJECT_BASIC_PROCESS_ID_LIST) +
        (sizeof(DWORD) * (dwMaxProcesses - 1));

    // Allocate those bytes from the stack
    JOB_OBJECT_BASIC_PROCESS_ID_LIST pjobpil =
        (JOB_OBJECT_BASIC_PROCESS_ID_LIST) _alloca(cb);

    // Were those bytes allocated OK? If so, keep going
    BOOL fOk = (pjobpil != NULL);

    if (fOk) {
        pjobpil->NumberOfProcessIdsInList = dwMaxProcesses;
        fOk = ::QueryInformationJobObject(m_hJob, JobObjectBasicProcessIdList,
            pjobpil, cb, NULL);

        if (fOk) {
            // We got the information, return it to the caller
            if (pdwProcessesReturned != NULL)
                *pdwProcessesReturned = pjobpil->NumberOfProcessIdsInList;
            CopyMemory(pdwProcessIdList, pjobpil->ProcessIdList,
                sizeof(DWORD) * pjobpil->NumberOfProcessIdsInList);
        }
    }
    return(fOk);
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

inline BOOL CJob::QueryBasicUIRestrictions(PDWORD pfdwRestrictions) {

    JOB_OBJECT_BASIC_UI_RESTRICTIONS jobuir;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &jobuir, sizeof(jobuir), NULL);
    if (fOk)
        *pfdwRestrictions = jobuir.UIRestrictionsClass;
    return(fOk);
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

inline BOOL CJob::QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo) {

    JOB_OBJECT_END_OF_JOB_TIME_INFORMATION joeojti;

```

JobLab.rc[illegible]

[illegible]