How do I wait until all processes in a job have exited?

devblogs.microsoft.com/oldnewthing/20130405-00

April 5, 2013

Raymond Chen

A customer was having trouble with job objects, specifically, the customer found that a `WaitForSingleObject` on a job object was not completing even though all the processes in the job had exited.

This is probably the most frustrating part of job objects: A job object does not become signaled when all processes have exited.

> The state of a job object is set to signaled when all of its processes are terminated because the specified end-of-job time limit has been exceeded. Use **WaitForSingleObject** or **WaitForSingleObjectEx** to monitor the job object for this event.

The job object becomes signaled only if the end-of-job time limit has been reached. If the processes exit without exceeding the time limit, then the job object remains unsignaled. This is a historical artifact of the original motivation for creating job objects, which was to manage batch style server applications which were short-lived and usually ran to completion. The original purpose of job objects was to keep those processes from getting into a runaway state and consuming excessive resources. Therefore, the interesting thing from a job object's point of view was whether the process being managed in the job had to be killed for exceeding its resource allocation.

Of course, nowadays, most people use job objects just to wait for a process tree to exit, not for keeping a server batch process from going runaway. The original motivation for job objects has vanished into the mists of time.

In order to wait for all processes in a job object to exit, you need to listen for job completion port notifications. Let's try it:

```
#define UNICODE
#define _UNICODE
#define STRICT
#include <windows.h>
#include <stdio.h>
#include <atlbase.h>
#include <atlalloc.h>
#include <shlwapi.h>


int __cdecl wmain(int argc, PWSTR argv[])
{
 CHandle Job(CreateJobObject(nullptr, nullptr));
 if (!Job) {
  wprintf(L"CreateJobObject, error %d\n", GetLastError());
  return 0;
 }


 CHandle IOPort(CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                                       nullptr, 0, 1));
 if (!IOPort) {
  wprintf(L"CreateIoCompletionPort, error %d\n",
          GetLastError());
  return 0;
 }


 JOBOBJECT_ASSOCIATE_COMPLETION_PORT Port;
 Port.CompletionKey = Job;
 Port.CompletionPort = IOPort;
 if (!SetInformationJobObject(Job,
       JobObjectAssociateCompletionPortInformation,
       &Port, sizeof(Port))) {
  wprintf(L"SetInformation, error %d\n", GetLastError());
  return 0;
 }


 PROCESS_INFORMATION ProcessInformation;
 STARTUPINFO StartupInfo = { sizeof(StartupInfo) };
 PWSTR CommandLine = PathGetArgs(GetCommandLine());


 if (!CreateProcess(nullptr, CommandLine, nullptr, nullptr,
                    FALSE, CREATE_SUSPENDED, nullptr, nullptr,
                    &StartupInfo, &ProcessInformation)) {
  wprintf(L"CreateProcess, error %d\n", GetLastError());
  return 0;
 }
 if (!AssignProcessToJobObject(Job,
         ProcessInformation.hProcess)) {
  wprintf(L"Assign, error %d\n", GetLastError());
  return 0;
 }


 ResumeThread(ProcessInformation.hThread);
 CloseHandle(ProcessInformation.hThread);
 CloseHandle(ProcessInformation.hProcess);


 DWORD CompletionCode;
 ULONG_PTR CompletionKey;
 LPOVERLAPPED Overlapped;


 while (GetQueuedCompletionStatus(IOPort, &CompletionCode,
          &CompletionKey, &Overlapped, INFINITE) &&
          !((HANDLE)CompletionKey == Job &&
           CompletionCode == JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO)) {
  wprintf(L"Still waiting…\n");
 }


 wprintf(L"All done\n");


 return 0;
}
```

The first few steps are to create a job object, then associate it with a completion port. We set the completion key to be the job itself, just in case some other I/O gets queued to our port that we aren't expecting. (Not sure how that could happen, but we'll watch out for it.)

Next, we launch the desired process into the job. It's important that we create it suspended so that we can put it into the job before it exits or does something else that would mess up our bookkeeping. After it is safely assigned to the job, we can resume the process's main thread, at which point we have no use for the thread and process handles.

Finally, we go into a loop pulling events from the I/O completion port. If the event is not "this job has no more active processes", then we just keep waiting.

Officially, the last parameter to `GetQueuedCompletionStatus` is `lpNumberOfBytes`, but the job notifications are posted via `PostQueuedCompletionStatus`, and the parameters to `PostQueuedCompletionStatus` can mean anything you want. In particular, when the job object posts notifications, it puts the notification code in the "number of bytes" field.

Run this program with, say, `cmd` on the command line. From the nested `cmd` prompt, type `start notepad`. Then type `exit` to exit the nested command prompt. Observe that our program is still waiting, because it's waiting for Notepad to exit. When you exit Notepad, our program finally prints "`All done`".

Exercise: The statement "Not sure how that could happen" is a lie. Name a case where a spurious notification could arrive, and how the code can protect against it.

Raymond Chen

Follow