Removing the TerminateThread from code that waits for a job object to empty

devblogs.microsoft.com/oldnewthing/20180831-00

August 31, 2018

Raymond Chen

Some time ago I showed how to wait until all processes in a job have exited. Consider the following code which wants to create a job, put a single process in it, and then return a handle that will become signaled when that process and all its child processes have exited.

This exercise is inspired by actual production code, so we're solving a real problem here.

```
template<typename T>
struct scope_guard
{
  scope_guard(T&& t) : t_{std::move(t)} {}
  ~scope_guard() { if (!cancelled_) t_(); }

  // Move operators are auto-deleted when we delete copy operators.
  scope_guard(const scope_guard& other) = delete;
  scope_guard& operator=(const scope_guard& other) = delete;

  void cancel() { cancelled_ = true; }

private:
  bool cancelled_ = false;
  T t_;
};

template<typename T>
scope_guard<T> make_scope_guard(T&& t)
{ return scope_guard<T>{std::move(t)}; }
```

This `scope_guard` class is similar to every other `scope_guard` class you've seen: It babysits a functor and calls it at destruction. We do add a wrinkle that the guard can be cancelled, which means that the functor is not called after all.

```
struct handle_deleter
{
 void operator()(HANDLE h) { CloseHandle(h); }
};

using unique_handle = std::unique_ptr<void, handle_deleter>;
```

The `unique_handle` class is a specialization of `std::unique_ptr` for Windows handles that can be closed by `CloseHandle`. Note that it will attempt to close `INVALID_HANDLE_VALUE`, so don't use it for file handles.

```
struct WaitForJobToEmptyInfo
{
 unique_handle job;
 unique_handle ioPort;
};

DWORD CALLBACK WaitForJobToEmpty(void* parameter)
{
 std::unique_ptr<WaitForJobToEmptyInfo> info(
    reinterpret_cast<WaitForJobToEmptyInfo*>(parameter));

 DWORD completionCode;
 ULONG_PTR completionKey;
 LPOVERLAPPED overlapped;

 while (GetQueuedCompletionStatus(info->ioPort.get(), &completionCode,
          &completionKey, &overlapped, INFINITE) &&
          !(completionKey == (ULONG_PTR)info->job.get() &&
           completionCode == JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO)) {
   /* keep waiting */
 }

 return 0;
}
```

The `WaitForJobToEmpty` starts by taking ownership of the `WaitForJobToEmptyInfo` structure it is passed as a thread parameter by wrapping it inside a `std::unique_ptr`. Next, it monitors the I/O completion port until the job reports that there are no more processes in it. Once that happens, the thread exits, which sets the thread handle to the signaled state.

```
HANDLE CreateProcessAndReturnWaitableHandle(PWSTR commandLine)
{
 auto info = std::make_unique<WaitForJobToEmptyInfo>();

 info->job.reset(CreateJobObject(nullptr, nullptr));
 if (!info->job) {
  return nullptr;
 }

 info->ioPort.reset(
   CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                          nullptr, 0, 1));
 if (!info->ioPort) {
  return nullptr;
 }

 JOBOBJECT_ASSOCIATE_COMPLETION_PORT port;
 port.CompletionKey = info->job.get();
 port.CompletionPort = info->ioPort.get();
 if (!SetInformationJobObject(info->job.get(),
       JobObjectAssociateCompletionPortInformation,
       &port, sizeof(port))) {
  return nullptr;
 }

 DWORD threadId;
 unique_handle thread(CreateThread(nullptr, 0, WaitForJobToEmpty,
                            info.get(), CREATE_SUSPENDED,
                            &threadId));
 if (!thread) {
  return nullptr;
 }

 // Code in italics is wrong
 auto ensureTerminateWorkerThread = make_scope_guard([&]{
  TerminateThread(thread.get());
 });

 PROCESS_INFORMATION processInformation;
 STARTUPINFO startupInfo = { sizeof(startupInfo) };
 if (!CreateProcess(nullptr, commandLine, nullptr, nullptr,
                    FALSE, CREATE_SUSPENDED, nullptr, nullptr,
                    &startupInfo, &processInformation)) {
  return nullptr;
 }

 auto ensureCloseHandles = make_scope_guard([&]{
  CloseHandle(processInformation.hThread);
  CloseHandle(processInformation.hProcess);
 });

 auto ensureTerminateProcess = make_scope_guard([&]{
```

```
  TerminateProcess(processInformation.hProcess);
 });

 if (!AssignProcessToJobObject(info->job.get(),
         processInformation.hProcess)) {
  return nullptr;
 }

 info.release();
 ensureTerminateProcess.cancel();
 ensureTerminateWorkerThread.cancel();

 ResumeThread(processInformation.hThread);
 ResumeThread(thread.get());

 return thread.release();
}
```

Let's walk through this function.

First, we create the `WaitForJobToEmptyInfo` object that contains the information we are passing to the worker thread.

We initialize the job and the I/O completion port, and associate the job with the completion port. If anything goes wrong, we bail out.

Next, we create the worker thread that will wait for the signal from the I/O completion port that the job is empty.

Here is the sticking point: We aren't finished setting up everything yet, and if it turns out we can't create the process or can't put the process in the job, then that thread will be waiting around for a notification that will never happen. But we want to pre-create all the resources we need before creating the process, so that we don't find ourselves later with a process that has already been created, but not enough resources to monitor that process.

Okay, so the idea is that we create the thread suspended so that it is "waiting" and hasn't actually started doing anything yet. That way, if it turns out we need to abandon the operation, we can terminate the thread. (Uh-oh, he talked about terminating threads.)

Okay, now that we have all our resources reserved, we can create the process. If that fails, then we bail out, and the `ensureTerminateWorkerThread` will terminate our worker thread as part of the cleanup.

If the process was created successfully, then we create a `scope_ guard` object to remember to close the handles in the `PROCESS_ INFORMATION` structure. And we also remember to terminate the process in case something goes wrong.

Next, we put the process in the job. If this fails, we bail out, and our various `scope_guard` objects will make sure that everything gets cleaned up properly.

Once the process is in the job, we have succeeded, so resume the process and the worker thread, and return the worker thread to the caller so it can be waited on.

The problem with this plan, of course, is that pesky call to `TerminateThread`, which is a function so awful it should never be called because there is basically no safe way of calling it.

So how do we get rid of the `TerminateThread`?

One solution is to tweak the algorithm so the thread is the last thing we create. That way, we never have to back out of the thread creation.

```
HANDLE CreateProcessAndReturnWaitableHandle(PWSTR commandLine)
{
 auto info = std::make_unique<WaitForJobToEmptyInfo>();

 info->job.reset(CreateJobObject(nullptr, nullptr));
 if (!info->job) {
  return nullptr;
 }

 info->ioPort.reset(
   CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                          nullptr, 0, 1));
 if (!info->ioPort) {
  return nullptr;
 }

 JOBOBJECT_ASSOCIATE_COMPLETION_PORT port;
 port.CompletionKey = info->job.get();
 port.CompletionPort = info->ioPort.get();
 if (!SetInformationJobObject(info->job.get(),
       JobObjectAssociateCompletionPortInformation,
       &port, sizeof(port))) {
  return nullptr;
 }

 // DWORD threadId;
 // unique_handle thread(CreateThread(nullptr, 0, WaitForJobToEmpty,
 //                                   info.get(), CREATE_SUSPENDED,
 //                                   &threadId));
 // if (!thread) {
 //  return nullptr;
 // }
 //
 // auto ensureTerminateWorkerThread = make_scope_guard([&]{
 //  TerminateThread(thread.get());
 // });

 PROCESS_INFORMATION processInformation;
 STARTUPINFO startupInfo = { sizeof(startupInfo) };
 if (!CreateProcess(nullptr, commandLine, nullptr, nullptr,
                    FALSE, CREATE_SUSPENDED, nullptr, nullptr,
                    &startupInfo, &processInformation)) {
  return nullptr;
 }

 auto ensureCloseHandles = make_scope_guard([&]{
  CloseHandle(processInformation.hThread);
  CloseHandle(processInformation.hProcess);
 });

 auto ensureTerminateProcess = make_scope_guard([&]{
  TerminateProcess(processInformation.hProcess);
```

```
 });

 if (!AssignProcessToJobObject(info->job.get(),
         processInformation.hProcess)) {
  return nullptr;
 }

 // Code moved here
 DWORD threadId;
 unique_handle thread(CreateThread(nullptr, 0, WaitForJobToEmpty,
                             info.get(), 0, // not suspended
                             &threadId));
 if (!thread) {
  return nullptr;
 }

 info.release();
 ensureTerminateProcess.cancel();
 // ensureTerminateWorkerThread.cancel();

 ResumeThread(processInformation.hThread);
 // ResumeThread(thread.get());

 return thread.release();
}
```

We don't need to create the thread suspended any more; it can hit the ground running.

Okay, so that's a solution if you can find a way to tweak your algorithm is that the thread is the last thing to be created. That way, you never have to try to roll back a thread creation. But that may not be possible. For example, maybe your algorithm involves creating multiple threads. Some thread gets to be last, but the others are now at risk of needing to be rolled back in case the last thread cannot be created.

Technique number two: Trick the thread into doing nothing if it turns out we don't want it to do anything.

In our case, what we can do is post a fake completion status to the I/O completion port to tell it, "Um, yeah, the job has no processes in it. Your job is done. Go home."

```
HANDLE CreateProcessAndReturnWaitableHandle(PWSTR commandLine)
{
 auto info = std::make_unique<WaitForJobToEmptyInfo>();

 info->job.reset(CreateJobObject(nullptr, nullptr));
 if (!info->job) {
  return nullptr;
 }

 info->ioPort.reset(
   CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                          nullptr, 0, 1));
 if (!info->ioPort) {
  return nullptr;
 }

 JOBOBJECT_ASSOCIATE_COMPLETION_PORT port;
 port.CompletionKey = info->job.get();
 port.CompletionPort = info->ioPort.get();
 if (!SetInformationJobObject(info->job.get(),
       JobObjectAssociateCompletionPortInformation,
       &port, sizeof(port))) {
  return nullptr;
 }

 DWORD threadId;
 unique_handle thread(CreateThread(nullptr, 0, WaitForJobToEmpty,
                            info.get(), 0, // not suspended
                            &threadId));
 if (!thread) {
  return nullptr;
 }

 // thread owns the info now
 auto ensureReleaseInfo = make_scope_guard([&]{
  info.release();
 });

 auto ensureTerminateWorkerThread = make_scope_guard([&]{
  // Tell the thread that there are no processes
  // so it will break out of its loop.
  PostQueuedCompletionStatus(info->ioPort.get(),
                            JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO,
                            (ULONG_PTR)info->job.get(),
                            nullptr);
 });

 PROCESS_INFORMATION processInformation;
 STARTUPINFO startupInfo = { sizeof(startupInfo) };
 if (!CreateProcess(nullptr, commandLine, nullptr, nullptr,
                   FALSE, CREATE_SUSPENDED, nullptr, nullptr,
                   &startupInfo, &processInformation)) {
```

```
  return nullptr;
 }

 auto ensureCloseHandles = make_scope_guard([&]{
  CloseHandle(processInformation.hThread);
  CloseHandle(processInformation.hProcess);
 });

 auto ensureTerminateProcess = make_scope_guard([&]{
  TerminateProcess(processInformation.hProcess);
 });

 if (!AssignProcessToJobObject(info->job.get(),
         processInformation.hProcess)) {
  return nullptr;
 }

 // info.release();
 ensureTerminateProcess.cancel();
 ensureTerminateWorkerThread.cancel();

 ResumeThread(processInformation.hThread);
 // ResumeThread(thread.get());

 return thread.release();
}
```

Technique number three: If all else fails, then just have a special flag to tell the thread, “I don’t want you to do anything. Just get out as quickly as you can.”

```
struct WaitForJobToEmptyInfo
{
 unique_handle job;
 unique_handle ioPort;
 bool active = false;
};

DWORD CALLBACK WaitForJobToEmpty(void* parameter)
{
 std::unique_ptr<WaitForJobToEmptyInfo> info(
    reinterpret_cast<WaitForJobToEmptyInfo>(parameter));

 // If we are not active, then do nothing.
 if (!info->active) return 0;

 DWORD completionCode;
 ULONG_PTR completionKey;
 LPOVERLAPPED overlapped;

 while (GetQueuedCompletionStatus(info->ioPort.get(), &completionCode,
          &completionKey, &overlapped, INFINITE) &&
          !(completionKey == (ULONG_PTR)info->job.get() &&
           completionCode == JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO)) {
   /* keep waiting */
 }

 return 0;
}

HANDLE CreateProcessAndReturnWaitableHandle(PWSTR commandLine)
{
 auto info = std::make_unique<WaitForJobToEmptyInfo>();

 info->job.reset(CreateJobObject(nullptr, nullptr));
 if (!info->job) {
  return nullptr;
 }

 info->ioPort.reset(
   CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                          nullptr, 0, 1));
 if (!info->ioPort) {
  return nullptr;
 }

 JOBOBJECT_ASSOCIATE_COMPLETION_PORT port;
 port.CompletionKey = info->job.get();
 port.CompletionPort = info->ioPort.get();
 if (!SetInformationJobObject(info->job.get(),
       JobObjectAssociateCompletionPortInformation,
       &port, sizeof(port))) {
  return nullptr;
```

```
 }

 DWORD threadId;
 unique_handle thread(CreateThread(nullptr, 0, WaitForJobToEmpty,
                            info.get(), CREATE_SUSPENDED,
                            &threadId));
 if (!thread) {
  return nullptr;
 }

 // auto ensureTerminateWorkerThread = make_scope_guard([&]{
 //  TerminateThread(thread.get());
 // });

 auto ensureResumeWorkerThread = make_scope_guard([&]{
  ResumeThread(thread.get());
 });

 PROCESS_INFORMATION processInformation;
 STARTUPINFO startupInfo = { sizeof(startupInfo) };
 if (!CreateProcess(nullptr, commandLine, nullptr, nullptr,
                    FALSE, CREATE_SUSPENDED, nullptr, nullptr,
                    &startupInfo, &processInformation)) {
  return nullptr;
 }

 auto ensureCloseHandles = make_scope_guard([&]{
  CloseHandle(processInformation.hThread);
  CloseHandle(processInformation.hProcess);
 });

 auto ensureTerminateProcess = make_scope_guard([&]{
  TerminateProcess(processInformation.hProcess);
 });

 if (!AssignProcessToJobObject(info->job.get(),
         processInformation.hProcess)) {
  return nullptr;
 }

 info->active = true; // tell the thread that it has work to do
 info.release();
 ensureTerminateProcess.cancel();
 // ensureTerminateWorkerThread.cancel();
 ensureResumeWorkerThread.cancel();

 ResumeThread(processInformation.hThread);
 ResumeThread(thread.get());

 return thread.release();
}
```

We could have signaled the thread that it should not do anything by closing the handles in the `WaitForJobToEmptyInfo` structure, but I want to demonstrate the most general possible solution.

There is some subtlety in resuming the worker thread: We need the `ResumeThread` to happen before the `thread.release()` because the `thread.release()` causes the `thread` to relinquish knowledge of the kernel thread. I probably could have fixed this some more scoping, but I tried to change the existing code as little as possible.

So there you go: Three ways of getting rid of the `TerminateThread` from this specific algorithm. The general-purpose trick works if the reason you were terminating a thread was to prevent it from starting. Instead of terminating the thread, resume it, but make sure it does nothing.

Raymond Chen