Are there alternatives to _lock and _unlock in Visual Studio 2015?

devblogs.microsoft.com/oldnewthing/20170125-00

January 25, 2017

Raymond Chen

A customer was using the `_lock` and `_unlock` functions in the C runtime library to take internal locks in order to avoid deadlocking with a thread they were suspending. They included this demonstration program:

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

unsigned int __stdcall ThreadFunc(void *);

#define _HEAP_LOCK 4    /* lock for heap allocator routines */

extern "C" void _lock(int);
extern "C" void _unlock(int);

int __cdecl main(int, char**)
{
  HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0,
      ThreadFunc, NULL, 0, NULL);

  for (;;) {
    printf(".");
    _lock(_HEAP_LOCK);
    SuspendThread(hThread);
    _unlock(_HEAP_LOCK);
    void *p = malloc(8);
    free(p);
    ResumeThread(hThread);
  }
  return 0;
}

unsigned int __stdcall ThreadFunc(void *)
{
  for (;;) {
    void *p = malloc(8);
    free(p);
  }
}
```

This sample program starts a worker thread that continuously allocates and frees memory from the heap. The main thread is in a loop that suspends the worker thread, and then tries to allocate memory from the heap while the worker is suspended.

Normally, this would be a problem if the worker thread happens to be in the middle of a heap operation, then the main thread will deadlock because it wants the heap lock, but the heap lock is owned by the worker thread, which is suspended.

The program addresses the problem by explicitly taking the heap lock before suspending the thread. That way, we are sure that the thread does not hold the heap lock before we suspend it. The definition of the magic number that represents the heap lock comes from the internal `mtdll.h` header file that comes with the C runtime source code.

The customer found that the version of the Visual C++ compiler that comes with Visual Studio 2015 no longer has the `_lock` and `_unlock` functions. As a result, their program doesn't compile any more. How can they suspend the thread without deadlocking on the heap?

The customer liaison pointed the customer to this article and suggested to the customer that they use other synchronization mechanisms instead of `SuspendThread`. The customer responded that they are developing a simulator for their product, and they need to suspend a thread as accurately as possible, so they need to use the `SuspendThread` function.

What they're doing now is not going to work well long-term, because it's taking dependencies upon the internals of the C runtime library. The C runtime library team explained,

> There is no replacement for `_lock` and `_unlock`. The Universal CRT does not expose its internal locks as older CRTs did. Note that their current approach of acquiring the CRTs heap lock before suspending the thread is not sufficient to avoid deadlock. The CRT `malloc` calls the Windows `HeapAlloc`, and the Windows heap has its own locks that it uses for synchronization.

On top of this, the documentation for `SuspendThread` cautions directly against this usage (emphasis mine):

> This function is primarily designed for use by debuggers. It is not intended to be used for thread synchronization. Calling **SuspendThread** on a thread that owns a synchronization object, such as a mutex or critical section, can lead to a deadlock if the calling thread tries to obtain a synchronization object owned by a suspended thread. To avoid this situation, a thread within an application that is not a debugger should signal the other thread to suspend itself. The target thread must be designed to watch for this signal and respond appropriately.

(Further discussion here.)

We didn't understand what the customer meant by "they are developing a simulator for their product" and how that required them to suspend a thread "as accurately as possible." We asked for clarification in the hopes that understanding their scenario would help us come up with a solution, but we never did get a clarification. That didn't stop us from trying to help anyway:

If the customer is really insistent on suspending the thread in order to do inspection of the process state, there are a few options.

One option, as noted in MSDN, is to coordinate with the thread so it is suspended at only well-defined points where it does not own any locks or resource. This is what the CLR does.

Another option is to make sure that while any thread is suspended, you never take any locks or more generally try to acquire any resources. Among other things, this means preallocating memory before suspending the thread. It also means that you cannot call into external functions because you have no idea what locks those external functions may take. Suspend the thread, memcpy the results into preallocated memory, resume the thread, and then process the results. Do not call the heap or anything else[1] that may require a lock.[2]

The best option is to do the suspension and inspection from another process. Even in that case, you need to be careful if the inspecting process requires locks that may be owned by the process being inspected, such as shared mutexes.

Suspending a thread at a random point in its execution, and then trying to do anything interesting from within the same process is a bad idea and has high deadlock potential.

The customer liaison thanked us for the information and explained that the customer wants to suspend the thread at arbitrary points in its execution because it is an application requirement.[3] They will take our recommendations into consideration while they decide what to do next.

[1] Furthermore, you would be best served to take the heap lock (`HeapLock`) before suspending the thread, because the Detours library will allocate memory during thread suspension.

[2] It may be difficult to avoid allocating memory. You can at least avoid the heap lock by using `VirtualAlloc` instead of `HeapAlloc` .

[3] This sounds like a circular argument, but hey.

Raymond Chen

Follow