Avoiding Heap Contention Among Threads

<keywords: threading, heap contention, synchronization, dynamic memory allocation, lock contention, stack allocation>

Abstract

Allocating memory from the system heap can be an expensive operation due to a lock used by system runtime libraries to synchronize access to the heap. Contention on this lock can limit the performance benefits from multithreading. To solve this problem, apply an allocation strategy that avoids using shared locks, or use third party heap managers.

This article is part of the larger series, "The Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

The system heap (as used by `malloc`) is a shared resource. To make it safe to use by multiple threads, it is necessary to add synchronization to gate access to the shared heap. Synchronization (in this case lock acquisition), requires two interactions (i.e., locking and unlocking) with the operating system, which is an expensive overhead. Serialization of all memory allocations is an even bigger problem, as threads spend a great deal time waiting on the lock, rather than doing useful work.

The screenshots from Intel® Parallel Amplifier in Figures 1 and 2 illustrate the heap contention problem in a multithreaded CAD application.

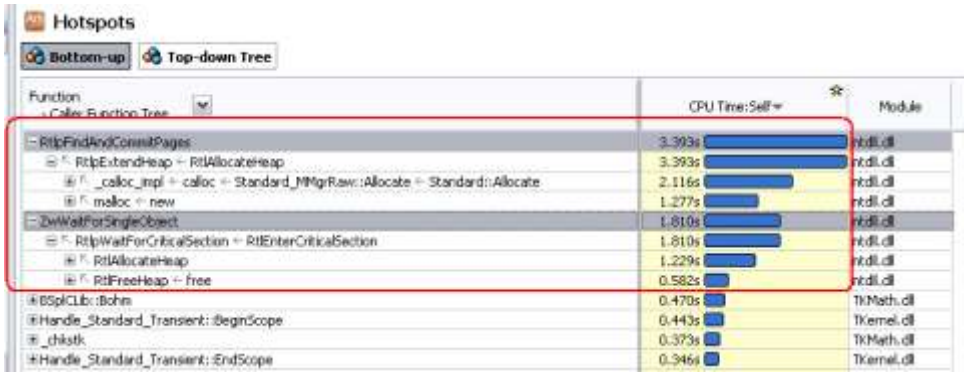

Figure 1. Heap allocation routines and kernel functions called from them are the bottleneck consuming most of the application execution time.

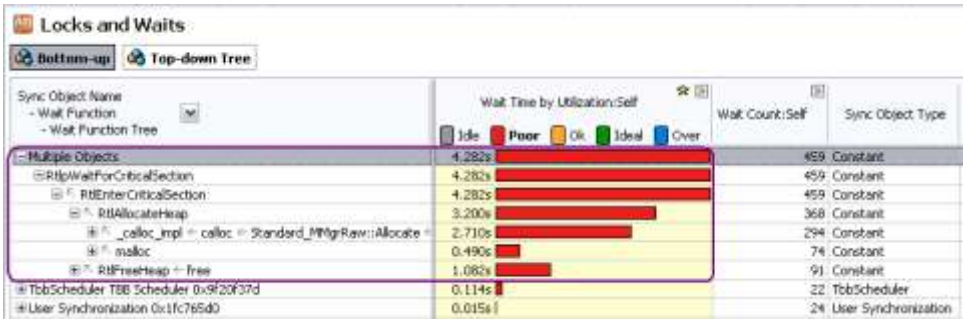

Figure 2. The critical section used in the heap allocation routines was the most contended synchronization object, causing a significant amount of wait time and poor CPU utilization.

Advice

The OpenMP* implementation in the Intel® Compilers exports two functions: `kmp_malloc` and `kmp_free`. These functions maintain a per-thread heap attached to each thread utilized by OpenMP and avoid the use of the lock that protects access to the standard system heap.

The Win32* API function `HeapCreate` can be used to allocate separate heaps for all of the threads used by the application. The flag `HEAP_NO_SERIALIZE` is used to disable the use of synchronization on this new heap, since only a single thread will access it. The heap handle can be stored in a Thread Local Storage (TLS) location in order to use this heap whenever an application thread needs to allocate or free memory. Note that memory allocated in this manner must be explicitly released by the same thread that performs the allocation.

The example below illustrates how to use the Win32 API features mentioned above to avoid heap contention. It uses a dynamic load library (.DLL) to register new threads at the point of creation, requests independently managed unsynchronized heap for each thread, and uses TLS to remember the heap assigned to the thread.

```
#include <windows.h>

static DWORD tls_key;

__declspec(dllexport) void *
thr_malloc( size_t n )
{
  return HeapAlloc( TlsGetValue( tls_key ), 0, n );
}

__declspec(dllexport) void
thr_free( void *ptr )
{
  HeapFree( TlsGetValue( tls_key ), 0, ptr );
}

// This example uses several features of the WIN32 programming API
// It uses a .DLL module to allow the creation and destruction of
// threads to be recorded.

BOOL WINAPI DllMain(
  HINSTANCE hinstDLL, // handle to DLL module
```

```
  DWORD fdwReason, // reason for calling function
  LPVOID lpReserved ) // reserved
{
  switch( fdwReason ) {
    case DLL_PROCESS_ATTACH:
    // Use Thread Local Storage to remember the heap
    tls_key = TlsAlloc();
    TlsSetValue( tls_key, GetProcessHeap() );
      break;

    case DLL_THREAD_ATTACH:
      // Use HEAP_NO_SERIALIZE to avoid lock overhead
    TlsSetValue( tls_key, HeapCreate( HEAP_NO_SERIALIZE, 0, 0 ) );
      break;

  case DLL_THREAD_DETACH:
    HeapDestroy( TlsGetValue( tls_key ) );
      break;

  case DLL_PROCESS_DETACH:
    TlsFree( tls_key );
      break;
  }
  return TRUE; // Successful DLL_PROCESS_ATTACH.
}
```

The `pthread_key_create` and `pthread_{get|set}specific` API can be used to obtain access to TLS in applications using POSIX* threads (Pthreads*), but there is no common API to create independent heaps. It is possible to allocate a big portion of memory for each thread and store its address in TLS, but the management of this storage is the programmer's responsibility.

In addition to the use of multiple independent heaps, it is also possible to incorporate other techniques to minimize the lock contention caused by a shared lock that is used to protect the system heap. If the memory is only to be accessed within a small lexical context, the `alloca` routine can sometimes be used to allocate memory from the current stack frame. This memory is automatically deallocated upon function return.

```
// Uses of malloc() can sometimes be replaces with alloca()
{
  …
  char *p = malloc( 256 );

  // Use the allocated memory
  process( p );

  free( p );
  …
}

// If the memory is allocated and freed in the same routine.

{
  …
  char *p = alloca( 256 );

  // Use the allocated memory
```

```
  process( p );
  …
}
```

Note, however that Microsoft has deprecated `_alloca` and recommends using the security enhanced routine called `_malloca` instead. It allocates either from the stack or from the heap depending on the requested size; therefore, the memory obtained from `_malloca` should be released with `_freea`.

A per-thread free list is another technique. Initially, memory is allocated from the system heap with `malloc`. When the memory would normally be released, it is added to a per-thread linked list. If the thread needs to reallocate memory of the same size, it can immediately retrieve the stored allocation from the list without going back to the system heap.

```
struct MyObject {
  struct MyObject *next;
  …
};

// the per-thread list of free memory objects
static __declspec(thread)
struct MyObject *freelist_MyObject = 0;

struct MyObject *
malloc_MyObject( )
{
  struct MyObject *p = freelist_MyObject;

  if (p == 0)
    return malloc( sizeof( struct MyObject ) );

  freelist_MyObject = p->next;

  return p;
}

void
free_MyObject( struct MyObject *p )
{
  p->next = freelist_MyObject;
  freelist_MyObject = p;
}
```

If the described techniques are not applicable (e.g., the thread that allocates the memory is not necessarily the thread that releases the memory) or memory management still remains a bottleneck, then it may be more appropriate to look into using a third party replacement to the heap manager. Intel® Threading Building Blocks (Intel® TBB) offers a multithreading-friendly memory manager than can be used with Intel TBB-enabled applications as well as with OpenMP and manually threaded applications. Some other third-party heap managers are listed in the Additional Resources section at the end of this article.

Usage Guidelines

With any optimization, you encounter trade-offs. In this case, the trade-off is in exchanging lower contention on the system heap for higher memory usage. When each thread is maintaining its

own private heap or collection of objects, these areas are not available to other threads. This may result in a memory imbalance between the threads, similar to the load imbalance you encounter when threads are performing varying amounts of work. The memory imbalance may cause the working set size and the total memory usage by the application to increase. The increase in memory usage usually has a minimal performance impact. An exception occurs when the increase in memory usage exhausts the available memory. If this happens, it may cause the application to either abort or swap to disk.

Additional Resources

Intel® Software Network Parallel Programming Community

Microsoft Developer Network: HeapAlloc, HeapCreate, HeapFree

Microsoft Developer Network: TlsAlloc, TlsGetValue, TlsSetValue

Microsoft Developer Network: _alloca, _malloca, _freea

MicroQuill SmartHeap for SMP

The HOARD memory allocator

Intel® Threading Building Blocks

Intel Threading Building Blocks for Open Source

James Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc. Sebastopol, CA, 2007.