Managing Lock Contention: Large and Small Critical Sections

<keywords: threading, lock contention, synchronization, spin-wait, critical section, lock size>

Abstract

In multithreaded applications, locks are used to synchronize entry to regions of code that access shared resources. The region of code protected by these locks is called a critical section. While one thread is inside a critical section, no other thread can enter. Therefore, critical sections serialize execution. This topic introduces the concept of critical section size, defined as the length of time a thread spends inside a critical section, and its effect on performance.

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

Critical sections ensure data integrity when multiple threads attempt to access shared resources. They also serialize the execution of code within the critical section. Threads should spend as little time inside a critical section as possible to reduce the amount of time other threads sit idle waiting to acquire the lock, a state known as "lock contention." In other words, it is best to keep critical sections small. However, using a multitude of small, separate critical sections introduces system overheads associated with acquiring and releasing each separate lock. This article presents scenarios that illustrate when it is best to use large or small critical sections.

The thread function in Code Sample 1 contains two critical sections. Assume that the critical sections protect different data and that the work in functions `DoFunc1` and `DoFunc2` is independent. Also assume that the amount of time to perform either of the update functions is always very small.

Code Sample 1:

```
Begin Thread Function ()
  Initialize ()

  BEGIN CRITICAL SECTION 1
    UpdateSharedData1 ()
  END CRITICAL SECTION 1

  DoFunc1 ()

  BEGIN CRITICAL SECTION 2
    UpdateSharedData2 ()
  END CRITICAL SECTION 2

  DoFunc2 ()
End Thread Function ()
```

The critical sections are separated by a call to `DoFunc1`. If the threads only spend a small amount of time in `DoFunc1`, the synchronization overhead of two critical sections may not be justified. In this case, a better scheme might be to merge the two small critical sections into one larger critical section, as in Code Sample 2.

Code Sample 2:

```
Begin Thread Function ()
  Initialize ()

  BEGIN CRITICAL SECTION 1
    UpdateSharedData1 ()
    DoFunc1 ()
    UpdateSharedData2 ()
  END CRITICAL SECTION 1

  DoFunc2 ()
End Thread Function ()
```

If the time spent in `DoFunc1` is much higher than the combined time to execute both update routines, this might not be a viable option. The increased size of the critical section increases the likelihood of lock contention, especially as the number of threads increases.

Consider a variation of the previous example where the threads spend a large amount of time in the `UpdateSharedData2` function. Using a single critical section to synchronization access to `UpdateSharedData1` and `UpdateSharedData2`, as in Code Sample 2, is no longer a good solution because the likelihood of lock contention is higher. On execution, the thread that gains access to the critical section spends a considerable amount of time in the critical section, while all the remaining threads are blocked. When the thread holding the lock releases it, one of the waiting threads is allowed to enter the critical section and all other waiting threads remain blocked for a long time. Therefore, Code Sample 1 is a better solution for this case.

It is good programming practice to associate locks with particular shared data. Protecting all accesses of a shared variable with the same lock does not prevent other threads from accessing a different shared variable protected by a different lock. Consider a shared data structure. A separate lock could be created for each element of the structure, or a single lock could be created to protect access to the whole structure. Depending on the computational cost of updating the elements, either of these extremes could be a practical solution. The best lock granularity might also lie somewhere in the middle. For example, given a shared array, a pair of locks could be used: one to protect the even numbered elements and the other to protect the odd numbered elements.

In the case where `UpdateSharedData2` requires a substantial amount of time to complete, dividing the work in this routine and creating new critical sections may be a better option. In Code Sample 3, the original `UpdateSharedData2` has been broken up into two functions that operate on different data. It is hoped that using separate critical sections will reduce lock contention. If the entire execution of `UpdateSharedData2` did not need protection, rather than enclose the function call, critical sections inserted into the function at points where shared data are accessed should be considered.

Code Sample 3:

```
Begin Thread Function ()
  Initialize ()

  BEGIN CRITICAL SECTION 1
    UpdateSharedData1 ()
  END CRITICAL SECTION 1

    DoFunc1 ()
```

```
  BEGIN CRITICAL SECTION 2
    UpdateSharedData2 ()
  END CRITICAL SECTION 2

  BEGIN CRITICAL SECTION 3
    UpdateSharedData3 ()
  END CRITICAL SECTION 3

  DoFunc2 ()
End Thread Function ()
```

Advice

Balance the size of critical sections against the overhead of acquiring and releasing locks. Consider aggregating small critical sections to amortize locking overhead. Divide large critical sections with significant lock contention into smaller critical sections. Associate locks with particular shared data such that lock contention is minimized. The optimum solution probably lies somewhere between the extremes of a different lock for every shared data element and a single lock for all shared data.

Remember that synchronization serializes execution. Large critical sections indicate that the algorithm has little natural concurrency or that data partitioning among threads is sub-optimal. Nothing can be done about the former except changing the algorithm. For the latter, try to create local copies of shared data that the threads can access asynchronously.

The forgoing discussion of critical section size and lock granularity does not take the cost of context switching into account. When a thread blocks at a critical section waiting to acquire the lock, the operating system swaps an active thread for the idle thread. This is known as a context switch. In general, this is the desired behavior, because it releases the CPU to do useful work. For a thread waiting to enter a small critical section, however, a spin-wait may be more efficient than a context switch. The waiting thread does not relinquish the CPU when spin-waiting. Therefore, a spin-wait is only recommended when the time spent in a critical section is less than the cost of a context switch.

Code Sample 4 shows a useful heuristic to employ when using the Win32 threading API. This example uses the spin-wait option on Win32 CRITICAL_SECTION objects. A thread that is unable to enter a critical section will spin rather than relinquish the CPU. If the CRITICAL_SECTION becomes available during the spin-wait, a context switch is avoided. The spin-count parameter determines how many times the thread will spin before blocking. On uniprocessor systems, the spin-count parameter is ignored. Code Sample 4 uses a spin-count of 1000 for each thread in the application but a maximum spin-count of 8000.

Code Sample 4:

```
int gNumThreads;
CRITICAL_SECTION gCs;

int main ()
{
  int spinCount = 0;
  ...
  spinCount = gNumThreads * 1000;
  if (spinCount > 8000) spinCount = 8000;
  InitializeCriticalSectionAndSpinCount (&gCs, spinCount);
  ...
```

```
}

DWORD WINAPI ThreadFunc (void *data)
{
  ...
  EnterCriticalSection (&gCs);
  ...
  LeaveCriticalSection (&gCs);
}
```

Usage Guidelines

The spin-count parameter used in Code Sample 4 should be tuned differently on processors with Intel® Hyper-Threading Technology (Intel® HT Technology), where spin-waits are generally detrimental to performance. Unlike true symmetric multiprocessor (SMP) systems, which have multiple physical CPUs, Intel HT Technology creates two logical processors on the same CPU core. Spinning threads and threads doing useful work must compete for logical processors. Thus, spinning threads can impact the performance of multithreaded applications to a greater extent on systems with Intel HT Technology compared to SMP systems. The spin-count for the heuristic in Code Sample 4 should be lower or not used at all.

Additional Resources

Intel® Software Network Parallel Programming Community

Intel® Hyper-Threading Technology