Windows Internals Course – University of Tokyo – July 2003

Thread/Synchronization Exercises

Arun Kishan – 2003/07/18a

Threads

In the Windows NT operating system, *threads* represent the primary unit of execution. Threads are pre-emptively scheduled by the kernel scheduler with the goal of obtaining maximum CPU utilization. The scheduler represents the most complex piece of code in the system, as it must take into account thread priorities, the presence of multiple logical and physical processors, CPU affinities, and moving forward, real-time constraints when assigning a thread to a particular CPU for execution. In this exercise, you will explore how the NT operating system deals with priority inversion, how one can use cooperatively scheduled threads, and techniques user-mode programs can use to increase throughput and parallelism.

Priority Inversion

Priority inversion is the phenomenon that causes a high priority thread to be delayed indefinitely while awaiting a resource held by a low priority thread. The low priority thread is often unable to run due to the presence of an unrelated medium priority thread. In the situation that results, one finds that the high priority thread is effectively denied the CPU by the lower medium priority thread.

NT combats this phenomenon with the *balance set manager*, a low real-time priority (level 16) system thread created during system initialization that executes the routine *KeBalanceSetManager()*. This routine periodically scans the scheduler's ready queues in search of long waiting variable priority (15 or less) ready threads to boost, as well as various other maintenance/performance tuning tasks, such as outswapping kernel stacks, tweaking the depth of the per-processor pool lookaside lists, and invoking the working set manager. When a thread is boosted, its priority is set to 15 and its quantum set to match the appropriate value for lock ownership. Once the quantum expires, the priority of the thread reverts to it original level.

Question: *How does this combat priority inversion?*

Experiment 1: Write a small program that has a low, medium, and high priority thread that exhibits priority inversion. Choose some amount of CPU bound time for the medium priority thread. Use a synchronization primitive such as a *mutex* as the critical resource shared between the high and low priority threads and make note of the completion order of the various threads. Make sure that the minimum priority used is the lowest real-time priority level (16) in order to guarantee that the balance set manager does not moderate priority levels. Use the base priority of PROCESS_PRIORITY_CLASS_REALTIME, which requires the caller to possess the SE_INC_BASE_PRIORITY_PRIVILEGE. Take care to avoid using *printf* or similar functions until the main body of the threads has

completed, as these routines can cause an implicit yield of the thread, disrupting observance of priority inversion.

Experiment 2: Modify the program above to run at lower priority levels, such that the balance set manager moderates thread priorities to combat priority inversion. Compare the completion times of the high priority threads. Experiment with the CPU bound run time of the medium priority thread in order to exhibit priority inversion if the value from the first experiment does not yield the expected results. *Hint*: Since thread startup may incur page faults (i.e., result in implicit context switches), try to ensure the threads have all completed initialization and are beginning to execute user-specified code before beginning this experiment. For best results, use a uni-processor machine, which guarantees that the single highest priority ready thread is the only one running.

Question: *Rather than using the balance set manager, what is an alternative scheme NT could have used to combat priority inversion? What are the tradeoffs?*

Threads and Fibers

All threads in the NT operating system are *preemptively* scheduled, i.e., at IRQL below DISPATCH_LEVEL, a thread may be halted at any instant by the operating system and resumed at a later time. NT may be distinguished from other operating systems by noting that much of the core system code itself is pre-emptible. However, in certain cases, a well written application may benefit from the ability to manually instruct the system when it is done performing work and another thread can be scheduled. Such *cooperatively* scheduled threads are available in NT in the form of lightweight *fibers*, which are essentially executable user-mode *co routines* that are grafted onto an underlying pre-emptible NT thread. Co routines allow control to be transferred deterministically by the programmer between distinct pieces of code and later resumed exactly at the point of transfer. They may be distinguished from function calls as each invocation requires its own stack and program counter (i.e., state), and as such fibers assume thread-like characteristics. These “context switches” are performed entirely in user mode and require no intervention of the operating system. Note with the NT fiber scheme, processes as a whole remain preemptively scheduled and thus a rogue process cannot monopolize the CPU resource.

Question: *What is a disadvantage of using co routines?*

Experiment 3: Write a small function that accepts as input two binary trees and returns a boolean value to indicate whether or not the sequence of values are the same when each tree is traversed in-order. Generate two binary trees containing 15 random elements in the range [75, 99] (discarding duplicates) and empirically verify the output of the function by displaying the in-order traversal of both trees. The function must return FALSE upon the first discovery of inequality in the in-order tree structure, e.g., it is unacceptable to determine the complete in-order traversal of both trees and then proceed to compare them. Use fibers (co routines) to simplify the code structure.

Question: Comment on a purely threaded approach to solving the same problem. Would it be more complex? How would the performance compare?

Non-blocking Data Structures

Most often shared data structures in user mode programs are protected by synchronization primitives such as a mutex or critical section. The minimum number of atomic (interlocked) operations these synchronization primitives require is typically two – one at the time of acquisition and another at the time of release. These interlocked operations typically atomically test and modify a memory location, requiring expensive interprocessor synchronization. Thus, whenever possible, it is favorable to reduce the requisite number of interlocked operations while still maintaining correctness. Additionally, should a thread fail while holding a synchronization primitive, or is preempted in favor of a higher priority thread or interrupt handler in the CPU, no other threads ready to access the queue will be able to make forward progress. Non-blocking data structures attempt to address both issues – wherever possible, they reduce the required number of interlocked operations while also reducing the size of the critical region to a single instruction (the interlocked operation). Consult the table below for a comparison between interlocked LIFO lists and traditional lock-based lists:

	Lock-Based	Lock-Free

Push	AcquireLock();
	NewNode->Next = Head;
	Head = NewNode;
	ReleaseLock();
		do {
		  OldHead = Head;
		  NewNode->Next = OldHead;
		  old = CmpExch(&Head, NewNode, OldHead)
		} while (old != OldHead);

Pop	AcquireLock();
	Top = Head;
	if (Top != NULL)
	  Head = Top->Next;
	ReleaseLock();
	return Top;
		while (Top = Head) {
		  old = CmpExch(&Head, Top->Next, Top)
		  If (old == Top)
		  break;
		}
		return Top;

(The semantics of *CmpExch(address, newvalue, oldvalue)* are as follows: the processor atomically tests if contents of *address* matches *oldvalue*, and if so, the contents is set to *newvalue*. Regardless of whether or not the operation succeeds, the value returned by *CmpExch* indicates the original contents of *address*. By definition, if *CmpExch* is successful, the returned value is *oldvalue*.)

Note the paradigm in each model: in the lock-based model, exactly one thread may be executing the code within the locked region at any time, whereas within the lock-free model, up to *n* threads (where *n* is the number of processors) may execute any instruction within the routine at the same instant. In the first model, the thread need not worry about the possibility of an operation failing once the lock is obtained, whereas in the second, a thread is not guaranteed that the *AtomicSwap* operation will succeed, and may potentially need to repeat the loop several times before the operation is successfully completed.

Question: *In the lock-free model, how many threads can concurrently fail to successfully complete an* AtomicSwap *operation? Since many threads can fail and may need to*

subsequently repeat the code, what is the advantage of the lock-free model over the lock-based model?

Experiment 4: Write a program that executes 128 threads for 10 seconds and demonstrates the throughput changes observed when using either a lock-free data structure or a lock-based data structure. Each thread should perform a push and followed by a pop, simulating work in between each (perform CPU bound work for some interval), repeating this sequence of operations until the timer expires. In the lock-based model, use any synchronization primitive in conjunction with the basic code structure above. For the lock-free case, use the InterlockedSLists provided by the platform SDK. In both cases, track the total number of completed pushes and pops using the *InterlockedIncrement* primitives. Compare the total throughput in both cases and comment on the differences. Is it as you expected? (Note: Since more than 64 thread handles will exist, the main thread cannot wait for the slave threads to complete using a single *WaitForMultipleObjects*. Design an alternate scheme that will avoid this limitation.)

Question: *The actual implementation of interlocked lists is slightly more complex than outlined above. Stored in the bottom bits of the* Head *pointer is a sequence number that is incremented in conjunction with each successful* Push *operation. What purpose does this sequence number serve? What problem with the above code does the sequence number not solve?* Hint: *Consider what could happen to nodes that are freed once they are removed from a non-blocking data structure.*