QNX SOFTWARE SYSTEMS

Using POSIX Threading to Build Scalable Multi-Core Applications

Kerry Johnson
Senior Product Manager
QNX Software Systems
kjohnson@qnx.com

Introduction

Until recently, the vast majority of embedded systems employed uniprocessor designs. The situation is changing quickly, however, with the availability of multi-core processors that offer significantly better performance per watt than conventional processor architectures. Fortunately, the programming techniques required to leverage the performance benefits of multi-core chips are well established. Together, POSIX threading and symmetric multiprocessing (SMP) offer a scalable approach to achieve the highest possible performance from processors based on two, four, or more cores.

POSIX Threads: An Overview

The base POSIX standard, IEEE Std 1003.1, defines a process model and a number of core operating system services for handling signals, timers, input, and output. A POSIX process is a running instance of a program. It encapsulates a number of resources that characterize a running entity, including program instructions, data space, heap space, and file descriptors associated with I/O devices. A POSIX operating environment typically protects all process resources: One process cannot see or access resources that belong to other processes, unless the programmer explicitly allows it (as in, for example, shared memory).

Every process has one or more flows of execution, known as *threads.* The operating system (OS) schedules each thread for execution independently of all other threads, including other threads in the same process. However, all threads within a process share the process's resources, such as file descriptors and data space. Since threads share these resources, changes made by one thread, such as closing a file, can affect other threads. In addition, multiple threads within a process can access the same data space and can simultaneously update any global variable. Therefore, developers must synchronize these threads to ensure data integrity.

The IEEE POSIX 1003.1c standard provides a portable programming interface for threads, commonly referred to as POSIX threads, or pthreads. The POSIX pthread interface defines a set of C language types and functions that handle thread creation, termination, synchronization, and other thread services. POSIX 1003.1c further defines a priority-based thread scheduling model. Realtime operating systems (RTOSs) such as the QNX® Neutrino® RTOS provide a preemptive, priority-based scheduler that dispatches the highest-priority thread to the CPU as soon as the thread becomes ready to run. (Additional information about POSIX threads is widely available through many publications and on the Web.)

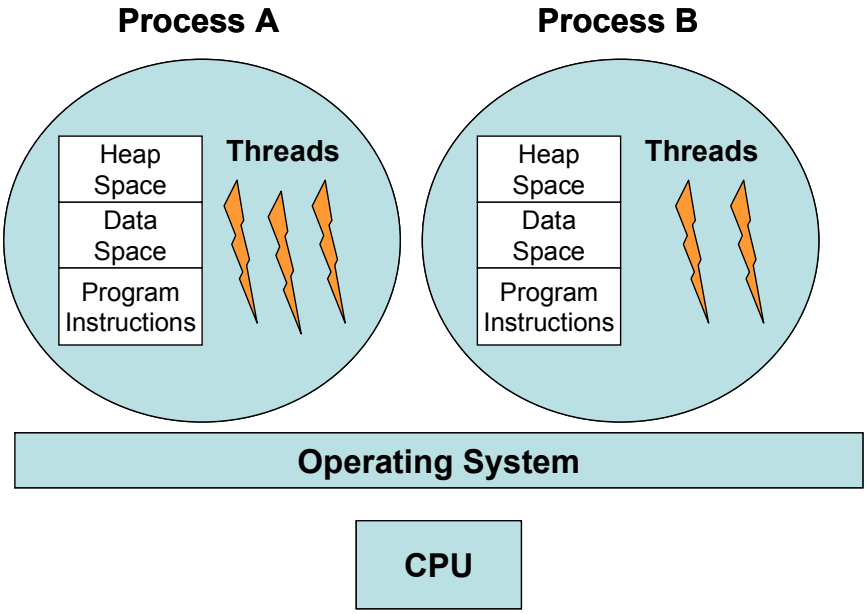

Figure 1 — A POSIX process encapsulates a number of resources and can comprise multiple threads. All threads within a process share the process's resources.

Symmetric Multiprocessing (SMP)

An SMP system consists of two or more tightly coupled CPUs. These CPUs can be either discrete, single-processor chips or the individual cores in a multi-core processor. An SMP system, by definition, has multiple identical processors connected to a set of peripherals, such as memory and I/O devices, on a common high-speed bus. The term *symmetric* denotes that each processor has the same access mechanism to the shared memory and peripheral devices.

Several differences exist between an SMP system and a distributed multiprocessor environment. In a distributed multiprocessor system, the individual processing units typically function as separate nodes. Each node can have a different processor and its own memory and I/O devices. Each processor runs its own OS and synchronizes with other processors using messages over an interconnect such as Ethernet.

In comparison, a tightly coupled shared-memory SMP system runs a single copy of the OS, which coordinates the simultaneous activity occurring across all of the identical CPUs. And because these tightly coupled CPUs access a common memory region, the system doesn't need a complex networking protocol to provide communications between different cores. Communications and synchronization can take the form of simple POSIX primitives (such as semaphores) or a native local transport capability, both of which offer much higher performance than networking protocols.

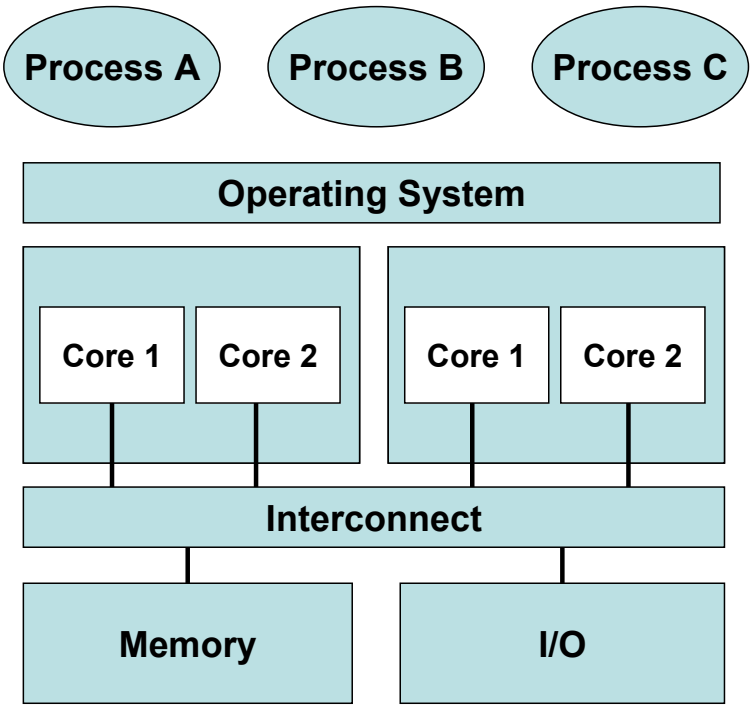

Figure 2 — Two dual-core processors configured as a shared-memory symmetric multiprocessor system. A standalone dual-core processor (or multiple single-core CPUs) could also form a symmetric system, provided that all CPU cores are identical and share memory and I/O.

Symmetric multiprocessing offers several key benefits:

- **Parallel processing** — Multiple processing units running in parallel can complete a set of tasks more quickly than one processor.

- **Scalable application software** — By using techniques such as worker threads (described below), application software can easily take advantage of the additional processors. Software can scale from two cores to four or more cores by creating additional threads and increasing parallelism.

- **Transparency** — From the developer's perspective, an SMP system is relatively straightforward to use. A well-designed, SMP-capable OS hides the application programmer from the actual number of hardware processing units and manages the complex task of allocating and sharing resources among the processors. As a result, the OS can run existing applications without requiring any SMP-specific modifications to application source code.

- **Scalable computing power** — An SMP system that can support additional processing cores or peripheral devices to execute more tasks in parallel offers a viable platform for scalability.

Pthreads, SMP, and the RTOS

With multi-threading, developers can fully leverage the scalable performance offered by symmetric multiprocessing. An SMP-capable RTOS running on an SMP system schedules threads dynamically on any core. Much like its uniprocessor equivalent, the RTOS's pre-emptive scheduler guarantees that, at any given time, threads of the highest priority are running. However, because each processor in an SMP system can run any thread, multiple threads can, in fact, run at the same time. If a thread of higher priority becomes ready to run, it will preempt the running thread of the lowest priority. Figure 3 illustrates RTOS scheduling on an SMP system.

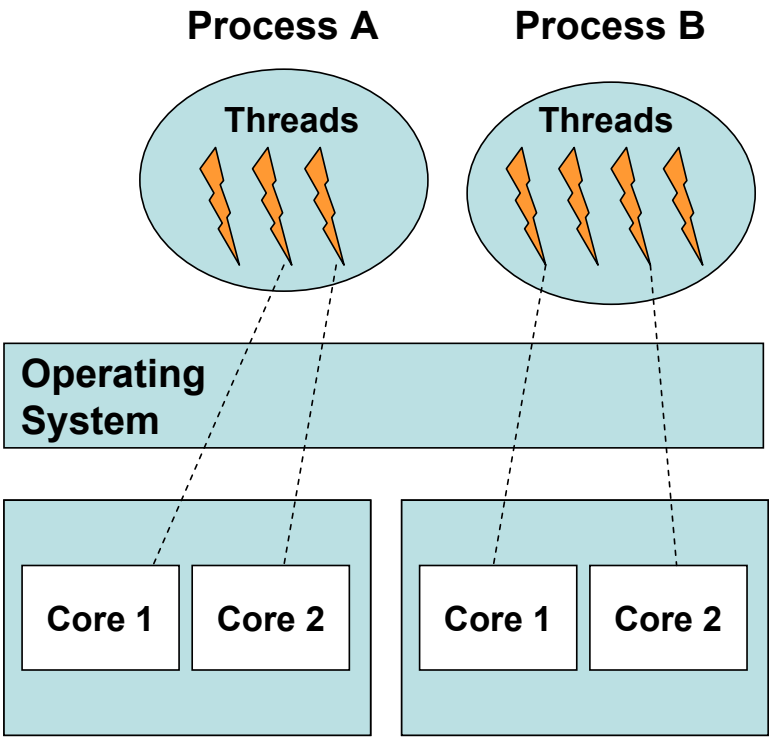

Figure 3 — The operating system schedules threads on any available core. An RTOS ensures that threads are scheduled according to their priority.

To improve processing throughput and speed in an SMP system, developers can introduce threads to take advantage of the multiple cores or CPUs. Non-threaded software will still benefit from the increased processing capacity offered by the SMP system, since the system can run multiple processes in parallel. But to truly speed up an existing process, the developer must divide the process into multiple parallel threads.

When moving to a parallel model, the developer can employ a variety of design patterns. For example:

- **Worker threads** — A main thread creates several threads to execute a workload in parallel. The number of worker threads should equal the number of CPU cores so that each core can handle an equal share of the work. The main thread may monitor the worker threads and/or wait for them to complete their work.

- **Pipelining** — The developer breaks a computation down into steps, where a different thread handles each step. Different data sets will exist in different stages of a pipeline.

- **Peer** — A thread creates other threads and participates as a peer, sharing the work.

Using Worker Threads: An Example

Worker threads work well when the system must perform repetitive calculations and when no dependencies exist between the calculations for the different data sets. For example, many programs iterate through an array, updating each element. If no dependencies exist between the elements (or between loop iterations), then the program is a good candidate for parallelization. Depending on the size of the array and the complexity of the calculations, a multithreaded design that uses a separate thread to calculate each portion of the array can significantly increase performance,

Consider the single-threaded example in Figure 4, where the *fill_array*() function updates a large array. This function simply iterates through the two-dimensional array, updating each element. Because the function updates the elements independently (element N value doesn't depend on N-1 element), it is easily made parallel.

```
float array[NUM_ROWS][NUM_COLUMNS];

void fill_array()
{
   int i, j;

   for ( i = 0; i < NUM_ROWS; i++ )
   {
      for ( j = 0; j < NUM_COLUMNS; j++ )
      {
        array[i][j] = ((i/2 * j) / 3.2) + 1.0;
      }
   }
}
```

Figure 4 — Single-threaded *fill_array*() function.

To increase the speed of *fill_array*(), we created a worker thread for each CPU, where each worker thread updates a portion of the array. To create worker threads, we used the POSIX *pthread_create*() call, as shown in Figure 5. The developer specifies the entry point for a

thread (where the thread starts running) by passing a function pointer to *pthread_create*(). In this case, each worker thread starts with the *fill_array_fragment*() function shown in Figure 6.

```
void multi_thread_fill_array()
{
  int        thread, rc;
  pthread_t  worker_threads[NUM_CPUS];
  int        thread_index[NUM_CPUS];

  // Synchronize a total of CPU+1 threads
  // One main thread + one worker thread per CPU
  pthread_barrier_init(&barrier, NULL, NUM_CPUS+1);

  for (thread = 0; thread < NUM_CPUS; ++thread)
  {
    thread_index[thread] = thread;
    rc = pthread_create(&worker_threads[thread],
                NULL,
                &fill_array_fragment,
                (void *)&thread_index[thread]);
    if (rc)
    {
      // handle error
    }
    // Wait at the barrier for all threads to arrive
    pthread_barrier_wait(&barrier);
    pthread_barrier_destroy(&barrier);
  }
  return;
}
```

Figure 5 — Main thread creates worker threads to update the array in parallel. The thread also creates synchronization objects and waits for all worker threads to complete their work. Note that each worker thread starts executing the *fill_array_fragment*() function.

```
void *fill_array_fragment(int *thread_index)
{
  int col = 0;
  int start_row = 0;
  int end_row = 0;

  // Compute the rows to update by this thread

  start_row = *thread_index * (NUM_ROWS/NUM_CPUS);
  end_row = start_row + (NUM_ROWS/NUM_CPUS) - 1;

  while (start_row <= end_row)
  {
    for (col = 0; col < NUM_COLUMNS; col++)
    {
      array[start_row][col] =
        ((start_row/2 * col) / 3.2) + 1.0;
    }
    ++start_row;
  }
  // Wait at barrier for all threads to complete
  pthread_barrier_wait(&barrier);
  return NULL;
}
```

Figure 6 — Each worker thread updates a portion of the array, then waits at the barrier.

Each worker thread determines which portion of the array it should update, ensuring no overlap with other worker threads. All worker threads can then proceed in parallel, taking advantage of SMP's ability to schedule any thread on any available CPU core.

Since the main thread must wait for the array to be fully updated before performing additional operations, we needed to implement synchronization to ensure all worker threads complete their work before the main thread proceeds. POSIX pthreads provides a number of synchronization primitives, including mutexes, semaphores, and thread joins. The example uses barrier synchronization, *pthread_barrier_wait*(), where any thread that has finished its work waits at a barrier until all other threads have also finished.

Now that we've converted the software to a multi-threaded approach, it can scale with the number of CPUs. The example in Figure 7 uses a four-CPU system, but we could just as easily adjust the number of worker threads to accommodate more or less processors.

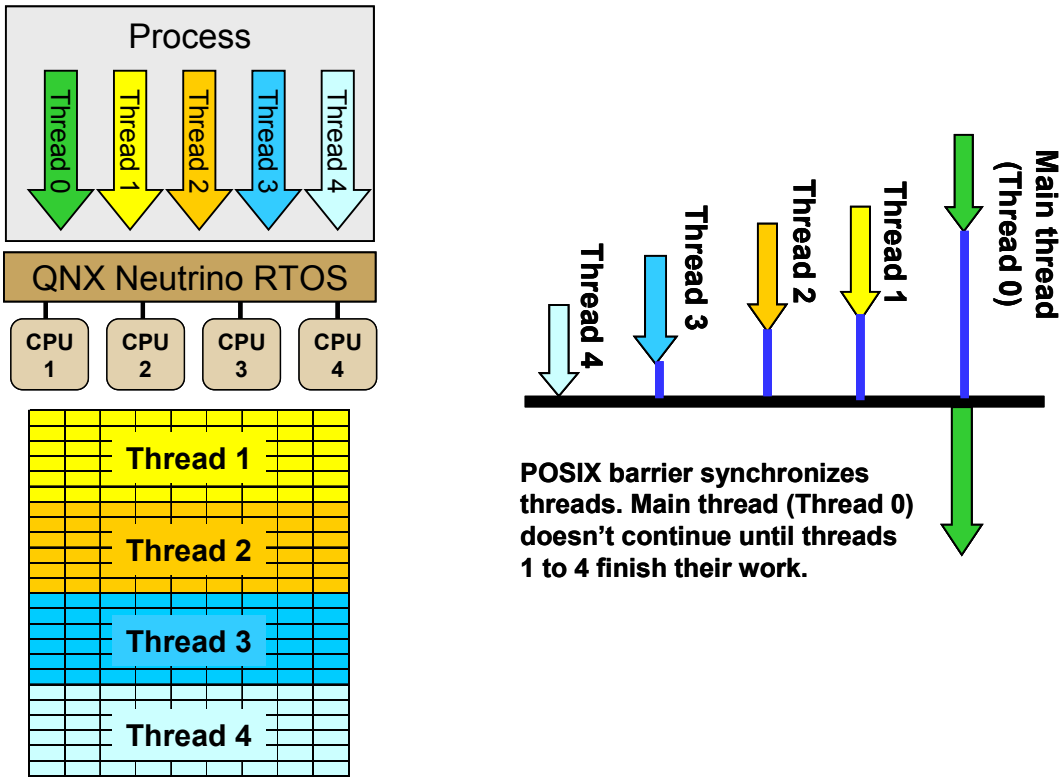

Figure 7 — Multi-threaded approach where each worker thread updates a portion of the array. Barrier synchronization ensures that all worker threads have finished updating the array before the main thread proceeds.

Visualizing Multi-Core Execution

Although traditional process-level debuggers can help diagnose some problems in a multi-core SMP system, they cannot provide insight into the complex system-level behaviors that arise from multiple threads interacting across multiple cores. In fact, a traditional debugger may indicate that a problem exists in one part of the multi-core system when the problem actually resides somewhere else.

To understand these behaviors and simplify the debugging and optimization of their multi-threaded, multi-core systems, developers need a system-tracing tool such as the QNX Momentics® system profiler. The system profiler works in conjunction with the instrumented version of the QNX Neutrino® RTOS kernel. The instrumented kernel logs all system calls and events to determine when threads are created, how long they run, and when they complete their work. The logging function provides high-resolution timestamps to ensure accurate timing information.

Figure 8 shows a system profiler trace for the multi-threaded, multiprocessing system. From this, you can quickly see that the RTOS scheduler has scheduled threads on each available core (indicated by color coding). You can also see how the worker threads wait at the barrier and how the main thread resumes execution after the barrier synchronization. The multi-threaded application is clearly performing as expected.

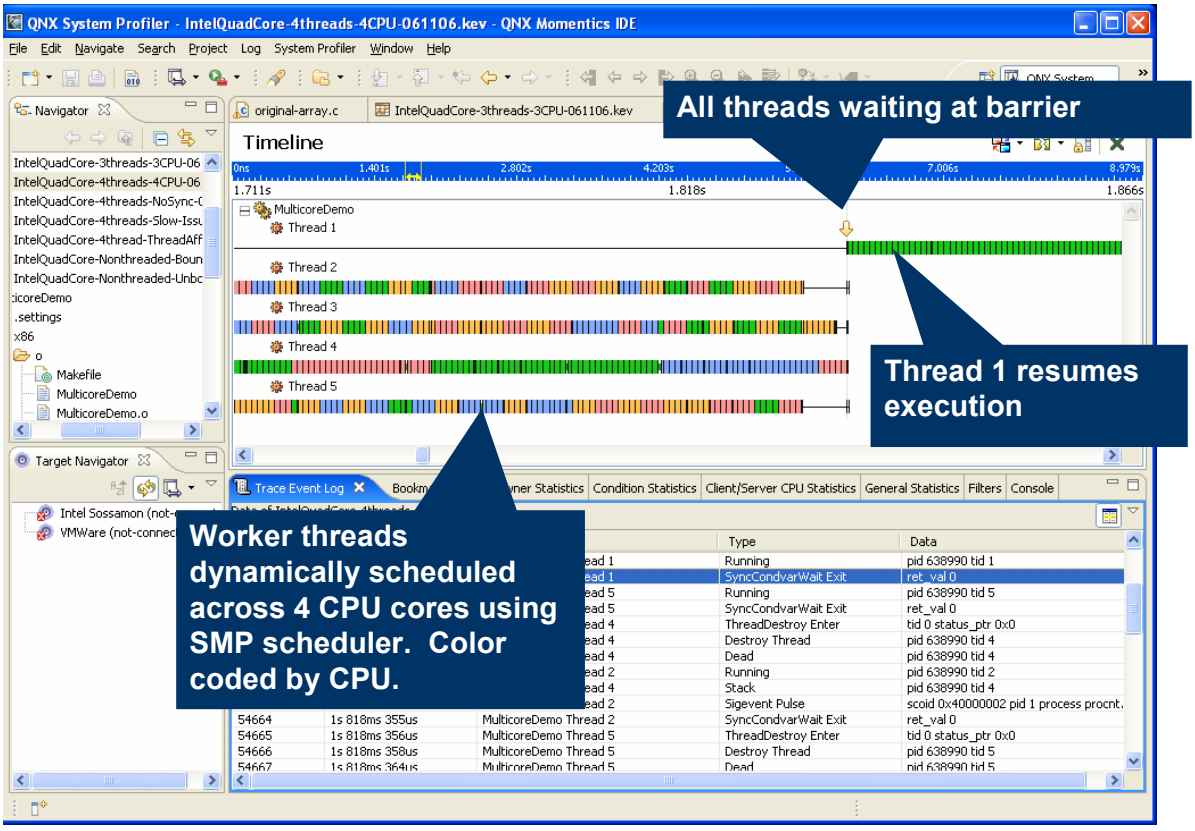

Figure 8 — Using the system profiler to visualize a multi-threaded application on an SMP system.

To measure the performance improvement achieved in the above multi-threading example, we used the QNX Momentics system profiler. First, we measured the timing of a single-threaded process that calls the *fill_array*() function to initialize the array and then proceeds with other work; see Figure 9. The system profiler indicates that the total execution time is 5.494739 seconds.

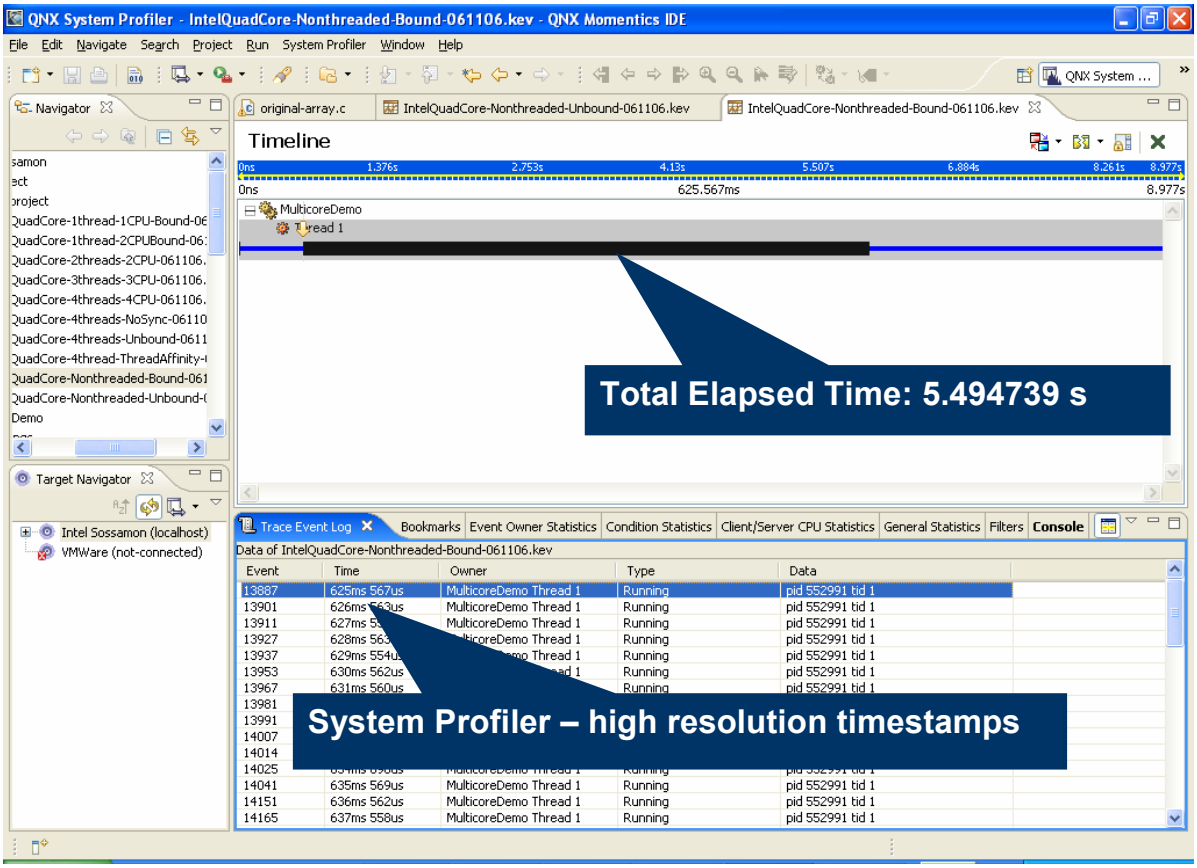

Figure 9 — Single-threaded timing. Trace shows total execution time of 5.494739 seconds.

Figure 10 illustrates the performance improvement achieved by creating four worker threads. Although the four threads ran on a four-CPU system, the profiler shows a 3.45x performance improvement, rather than a 4x improvement. Upon further analysis, the system profiler showed that the additional processing done by the main thread, after *fill_array*() completes, contributes 0.880243 seconds to the overall execution time. This time is consumed in both the single-threaded and multi-threaded implementations. Since this additional processing is single threaded, it doesn't benefit from multiple CPUs. Therefore, it isn't possible to reach a fourfold performance improvement in this case.

This outcome demonstrates Amdahl's Law, which states that the amount of speedup achieved through parallelism is limited by the amount of nonparallel portions of code. When we factored out the linear region, performance increased to 3.95x. The difference between this measured value and a theoretical fourfold improvement can be attributed to the cost of creating and synchronizing four threads.

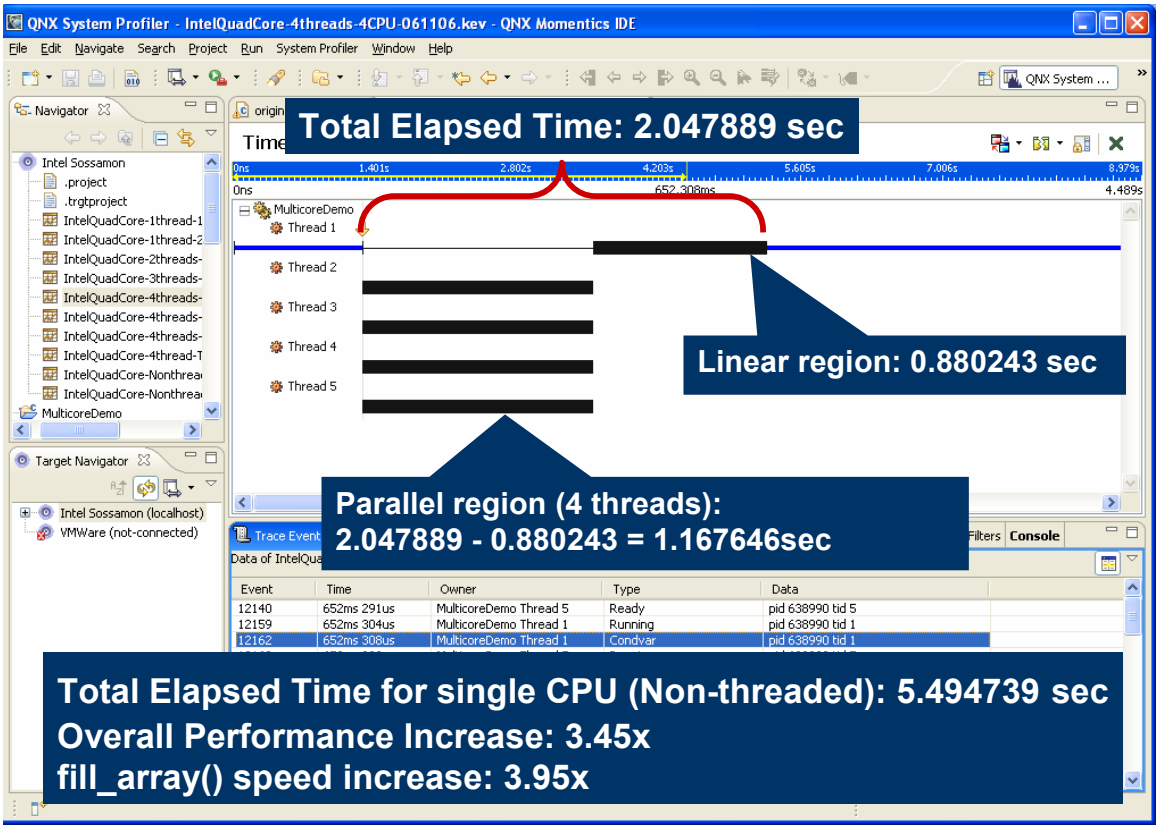

Figure 10 — Multi-threaded implementation with four worker threads and four CPUs. Total execution time = 2.047889 seconds. System profiler confirms that the parallelized portion of *fill_array*() runs 3.95x faster than the non-parallelized version.

Other Optimizations

For multiprocessing systems, the most significant benefit comes from increasing parallelism within the software. However, other optimizations can also have a large impact on overall performance. These include reducing contention for shared resources, modifying lock granularity, and optimizing use of cache. These optimization approaches require multiprocessing-capable toolsets to help diagnose and correct the software.

Conclusion

The combination of POSIX pthreads and SMP provides a programming model that can scale by adding additional processors. This has become particularly important now that semiconductor manufacturers are improving performance by adding more processing cores. Besides using an RTOS that supports SMP, developers need a rich toolset to characterize existing software and simplify optimization on multi-core systems.