SMP: Two Processors and Beyond

Authored by: Mario Charest
Updated by: Thomas Fletcher

So you bought a top-of-the-line PC with a 319.76GHz CPU and you still need more power. Using another machine's processor over a network won't cut it. You'll need to handle more data than your network can support. One solution would be to add a second CPU to your computer, or perhaps even a third or a fourth. That's just what symmetric multi-processing, or SMP, allows you to do (as long as your motherboard supports it, of course).

Compared to a network of computers, SMP offers many advantages. For starters, each CPU in the system has access to the same memory, PCI card, I/O port, and other hardware. But you need an operating system that can support SMP - and there are a good number that do, including Windows Vista, Windows NT, Linux, VxWorks, and of course, QNX Neutrino.

These days, SMP is becoming extremely cost-effective. I'm currently typing this document on a dual processor machine, in fact most desktop processors sold today are multi-core processors.

SMP also lets you do more. In general, SMP won't make a process run faster, unless it was written specifically for SMP. So the game Unreal Tournament won't be faster on an SMP computer than on a computer with a single CPU. But while you play Unreal Tournament, you can burn a CD and download from the Internet without affecting the frame rate (well almost)! On Windows, when I print a document on a single-CPU machine, the machine becomes unresponsive because it has to deal with the parallel port. But with an SMP machine, Windows runs very smoothly because one CPU is handling the parallel port and the other is free to run any other program.

If you have a dual processor machine using the QNX real-time platform, running make -j2 will compile a project up to 60 percent faster.

At this point, I highly recommend you jump to the SMP section (http://www.qnx.com/developers/docs/momentics621_docs/neutrino/sys_arch/smp.html) of the Neutrino manual before continuing. That should give you a good technical introduction to SMP and allow me to focus on the fun stuff.

Memory bottleneck

One of the limitations of SMP is memory bottleneck. As you might imagine, memory can't be accessed at the same time by each CPU. CPUs either have to take turns or use memory caching to reduce access bottlenecks.

The following program is a very crude example that demonstrates the memory-bottleneck effect:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#define LOOP 50
#define SIZE 50000000
int main(int argc, char* argv[])
```

```
{
int t;
int i;
float elapse;
static char dummy[SIZE];
t = clock();
for(i=0; i<LOOP; i++ ) {
memset( dummy, 0, sizeof( dummy) );
}
elapse = ( clock()-t ) / (float)CLOCKS_PER_SEC;
printf("Duration %f sec, %.1f MBytes/sec \n",
elapse, sizeof(dummy)*LOOP/elapse/1000000);
return 0;
}
```

The program fills 50M of RAM with zeros, 50 times. When this program is run once, it takes 14.3 seconds with a memory throughput of 175Mbytes/sec. If I run the program twice, simultaneously, each program now takes 24 seconds with a throughput of 100Mbytes/sec. The total throughput is 200Mbytes/sec - a 12.5 percent increase - which is pretty negligible.

This is a worst-case scenario, however, as the data area won't fit into a processor's cache (thus there are few cache hits). Also note that my machine has an Intel® Celeron® processor. It has little cache and a slow bus speed compared to the Intel® Pentium® or Athlon processors. Hence, SMP is of little benefit here. Running the program twice in a row would take 28.6 seconds, but running it simultaneously took only 24 seconds.

If we go to a best-case scenario, setting size to 50000, the memory filled by the program should now fit in the Intel® Celeron® processor's cache. Thus the program will not be overly affected by the memory bottleneck. Loop is increased to 250000 to have a similar duration. Running a single instance of the program now takes 10.6 seconds at 1180Mbytes/sec. When run simultaneously, two programs take 10.6 sec at 1180Mbytes/sec, for a total throughput of 2360Mbytes/sec. As you can see, because each CPU needed to do little to access RAM for this test, they didn't affect one another. In fact, the performance doubled!

Of course these are extreme examples that rarely happen in real life. Hopefully they will give you some understanding of the basic nature of SMP and how it can help you with your design. Estimating the performance gain obtained by moving to SMP is almost impossible.

Cache

The cache is one of the reasons why today's CPUs run as fast as they do. With CPUs beyond 1GHz and memory in the 100MHz range, CPUs would definitely starve for data. The cache's job is to hold the data most recently accessed or most likely to be needed. To give you an idea of the benefit of the cache, turn the CPU cache off in your BIOS. You will notice a dramatic effect.

When it comes to SMP, the kernel has to make intelligent use of the cache. Any threads, depending on CPU availability, can be "moved" from CPU to CPU, in an attempt to keep every CPU busy. However, thread migration has its disadvantages. For example, if a thread's data is in the first CPU cache, when it is moved to a different CPU, the data in the first cache has to be invalidated (thus flushed from memory). The second CPU doesn't have any data related to the thread in its cache, so it will have to get it from memory. Thus, thread migration is to be avoided whenever possible. Through some heuristics, the Neutrino SMP kernel attempts to find the right balance between CPU utilization and thread migration.

Luckily, system designers don't really have to deal with cache details - that's up to the kernel and compiler. The QNX2000 conferences offered many seminars; one was about optimization. A few tips given during that conference really got my attention by their simplicity, the type of simple ideas that had you wondering why you didn't think of them yourself.

You've probably seen this type of programming often:

```
char array[256000];
memset( array, 0, sizeof( array ) );
for ( cnt = 0; cnt < sizeof( array ); cnt++ )
do_something( array[cnt] );
```

Take 20 seconds to find a way to potentially increase the performance of this program.

The solution is: `char array[256000];`

```
memset( array, 0, sizeof( array ) );
for ( cnt = sizeof( array )-1; cnt >=0; cnt--)
do_something( array[cnt] );
```

Let's go back to the first example. Array is 256000 bytes; it won't fit in an Intel® Celeron® processor's cache. Thus when memset does its thing, only a certain portion of the array can sit in the cache. When memset is setting the last byte of the array (array[255999]) the byte located at array[0] won't be in the cache. The **for** loop starts and first accesses array [0], but it's not in the cache anymore; valuable time is lost. I made the array 256K to make it simpler to understand, but the same principal applies for smaller blocks of data. In an OS like Neutrino, you have other things happening like interrupt handlers, threads, etc. A 16K block of data may have been half flushed to make room for an interrupt handler. Even if the data appears to fit in the cache, it won't hurt to apply the same principal.

One other idea that would apply to this sample is to write a **memset_r** function that fills data backward. (That's actually what I did in my own code.) It's simple to implement in already existing code: just replace the existing **memset**() routines with the new **memset_r**().

Other tips, also relating to the cache, involve processing data in small packets. In essence it's faster to process smaller blocks of data many times than to process one huge block. I have demonstrated this in the following piece of code:

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#define BLOCK_SIZE 10000
void work1( char *data, size_t size ) {
size_t cnt;
for( cnt = 0 ; cnt < size; cnt++, data++ ) {
*data ^= 1;
}
}
void work2( char *data, size_t size ) {
size_t cnt;
for( cnt = 0 ; cnt < size; cnt++, data++ ) {
*data |= 2;
}
}
void work3( char *data, size_t size ) {
size_t cnt;
for( cnt = 0 ; cnt < size; cnt++, data++ ) {
*data += 3;
}
}
int main(int argc, char* argv[]) {
int t;
int i;
float elapse;
static char dummy[BLOCK_SIZE * 10000];
t = clock();
// --- one big block
memset( dummy, 0, sizeof( dummy ) );
work1( dummy, sizeof( dummy ) );
work2 ( dummy,sizeof( dummy ) );
work3 ( dummy,sizeof ( dummy ) );
elapse = ( clock()-t ) / (float)CLOCKS_PER_SEC;
printf("Duration %f sec(s)\n", elapse );
t = clock();
// --- smaller blocks
for( i=0; i<sizeof( dummy ) ; i+= BLOCK_SIZE ) {
memset( &dummy[i], 0, BLOCK_SIZE );
work1 ( &dummy[i], BLOCK_SIZE );
work2 ( &dummy[i], BLOCK_SIZE);
work3 ( &dummy[i], BLOCK_SIZE);
}
elapse = ( clock()-t ) / (float)CLOCKS_PER_SEC;
printf("Duration %f sec(s)\n" , elapse );
return 0;
}
```

On an Intel® Celeron® processor, the output is:

```
      Duration 8.609000 sec(s)

      Duration 6.766000 sec(s)
```

Even with the overhead of the **for** loop and the calls to four functions 10000 times over, there is a gain of about 1.8 seconds, over 20 percent. Convinced? I could have improved this further by making work1 and work3 process the data in reverse.

Another thing worth mentioning about an SMP system: if you compare two 500MHz Intel® Celeron® processors with a 1GHz Intel® Celeron® processor (if it existed), the two Intel® Celeron® processors have twice the amount of cache as the single Intel® Celeron® processor. This is why in some cases SMP makes so much sense. More cache is always better.

FIFO scheduling

On SMP machines, threads really do run at the same time. Multi-tasking is often conceptualized as running multiple programs at the same time, but that is not truly the case. The CPU is shared across different programs. One obvious example is the FIFO scheduling algorithm of QNX 4. If a group of programs are running at the same priority in FIFO mode, when one of these program gets the CPU, the other programs of that group will not get CPU time until relinquished by the currently running process. I have seen some designs rely on FIFO and have personally used it a few times. FIFO mode is useful to implement data protection with no extra overhead. But on an SMP machine, FIFO is your worst enemy; it will simply not work. The FIFO mode does not work across multiple CPUs. Two programs of the same priority, running in FIFO mode, stand a very high chance of running on different CPUs; thus running at the same time... There is a way around this problem: with Neutrino it is possible to force a process to run on a specific CPU. Thus a group of programs running in FIFO mode could be forced to run on a common CPU. Unfortunately, this has negative side effects which I will explain later. In general, stay away from FIFO scheduling. This is well-covered in the Neutrino documentation.

Priority

Just like FIFO, priorities can play tricks on you. A thread at a priority of one can run at the same time as a thread of priority 63 because each will run on a different processor. Do not rely on a priority to create critical sections - you'll get bitten.

Quad Intel® Xeon™ Processor

I had the chance to work on a project involving a Quad Intel® Xeon™ processor (1M of cache) machine. I learned a lot about SMP - most of the time, the hard way. Let me share my experience with you.

One of the first things I did when I got access to the machine, was to run a test similar to the worst-case scenario described earlier. To my surprise there was only about a 20 percent penalty. In other words, the single instance of the program took 10 seconds to run, but both programs running simultaneously only took 12 seconds. So unlike the Intel® Celeron® processor, the Intel® Xeon™ is apparently designed with SMP in mind (as shown by the large 1M cache). I started to have some respect for the beast.

The use of four processors

The Quad machine's purpose was to impersonate a robot with two arms, each with seven joints. As you might imagine, simulating such a complex machine requires a fair bit of processing power. To provide accurate simulation, the calculation loop needed to run at 1000MHz and couldn't miss a beat. On top of this intensive task, the machine had to log data on a local hard disk, support file transfer of TCP/IP, handle a high-speed link (ScramNet) and do a few other things...

When tested on a single-CPU machine, the simulation part was slower than 1000MHz. The developer of the simulation algorithm found a way to parallel some sections of the algorithm. After these sections were split in two threads, they could each run on their own CPU on the SMP machine. Thus, the simulation ran below 1000MHz. It was still a very close call - too close for comfort. Two processors weren't going to be enough. We needed to run other things on the machine as well, hence the use of the Quad.

Interrupts

One of our concerns was how interrupts would affect the performance of the simulation. Remember, we couldn't afford to miss one loop. Luckily, it turns out that all the interrupts are handled by the first processor (this is set up by the kernel). There might be an occasion where the kernels running on each CPU could lock on each other to handle an interrupt-related event. In fact, the interrupt on the first CPU never affected the other CPUs as far as we could measure. That's not to say the other CPUs would never be disturbed. What may happen is if a thread or process located on a CPU was set up to receive the interrupt event, the kernel on the first CPU would have to interface with the kernel on the other CPU. This will slightly disturb the realtime behavior of the second CPU. In our case, because the other CPUs were so busy, thread-related interrupts mostly resided on the first CPU.

Affinity

It is possible to force a thread to run on a specific process and never be migrated to another CPU. This is called affinity. In general, you shouldn't have to deal with affinity, since the kernel usually does a capable job of handling it.

Tests showed the simulation machine performed better if we forced the two simulation threads to run on processors two and three. Each of these threads were set at priority 63. Because these threads had the highest priorities on the system, we were sure no other threads would disrupt the simulation. Actually we would have liked to have had the capability of preventing any other threads from using the CPU, a kind of exclusive mode. We actually were able to measure the effect of cache misses caused by other threads running on the processor while the simulation threads were in receive mode. These tests were performed on an early beta of Neutrino 2.1. It is quite possible by now that we could avoid using the affinity mask altogether.

Side effects

During the project, I got bitten hard once by SMP. I learned my lesson and it didn't happen again. To make matters worse, everything was running fine but I was sure something was wrong. I even spent three days searching for a bug that didn't exist.

I wrote a program to handle data logging. To make sure logging wouldn't interfere with the simulation, the logger was composed of two threads. The main thread was a resource manager handling the write operation performed by the client. It stuffed the data in a circular buffer. The other thread, set at a low priority, was set to flush the circular buffer on the HD. That way clients would never block waiting for the logger to write to disk. In fact, they could be forced to wait if the circular buffer got full.

I started to test the program piece by piece; everything looked good. I wasn't expecting any trouble **-** it was, after all, a simple program. Then came time to test the buffer full case and to make sure my index math was OK. I started putting **printf**s here and there to watch over the indexes. At that point I wasn't writing to disk yet. This allowed me to test the overhead of the threads, compared to having the client writing directly to the HD. Then I started to notice the indexes didn't look right. My brain told me something was wrong. I checked and rechecked everything assuming I forgot to create a critical section, but to no avail. I gave up and went to work on something else for a few hours in the hope of clearing my mind. When I came back to hunt the bug, my tactics worked. I figured it out right away. It was so simple. Because the program was on an SMP machine, both threads were really running at the same time. The circular queue never held more than a few kilobytes of data, since the data in the buffer got processed (thus removed) as fast as it could be filled. Actually, I didn't have any code to write the data to disk, so removing the data from the buffer was in fact faster than writing into it. I laughed (what else could I do?) ...

Polling

Polling should be avoided in the real-time world. But sometimes it can make one's design perform better. For example, if you have to process something 100,000 times a second, polling can be advantageous. This is well below the OS timer capability. You could set up hardware to generate 100 interrupts a second, but that would put a heavy burden on the CPU. Even for today's fast CPUs,100,000 interrupts are a lot. On an SMP machine, this would be no problem. You could have the program polling on a timer counter to check for a proper elapsed time and do a job when 10 **usec** expires. Of course you would need one CPU per polling process.

Unfortunately, SMP hardware isn't well-suited for embedded systems because of the extra space and heat-dissipation requirements. Luckily, with the appearance of single chips with multiple CPUs on them, we may see this change in the future.

Conclusion

If you have a chance, test all your programs on an SMP machine. You may find potential bugs that could affect a program even in a non-SMP system. Like any other tool, SMP is not the solution to all performance-related problems, but it's sure worth investigating. I hope this article gives you some ideas to help you solve your next problem. (By the way, this article was typed on an SMP machine. No CPUs were hurt or mistreated during the course of the project. :-)