Adrian Guthals

Brina Lee

10/25/11

CSE 221

**Part 1 of System Measurement Project**

**Introduction:** The goal of this project is to determine the performance characteristics of Intel® Core™2 Quad CPU Q6600 and other components working with the operating system, a version of Ubuntu. We will be benchmarking this operating system with the following hardware specifications in our machine description. In our set of experiments, our goal is to understand the performance issues and benefits of the operating system. With our experiments, we will hopefully gain intuition about the relative speed and responsiveness of our system and be able to identify performance bottlenecks.

We used C language to implement our measurements because C has a complier that has low-level access to memory and maps efficiently to machine instructions. C also has many low-level capabilities, like minimizes variations in the runtime and compile in binary. To compile we used g++ with no optimizations by declaring “-O0”. We turned off all the complier optimization because we did not want the compiler’s optimization settings to affect our benchmarks’ run-time. We also added the “-finline” flag to make sure that the code for recording time (in CPU clock cycles) is copied and inserted where it is called, rather than being called as a function (which adds more cycles to time-recording overhead). We also used the “-lpthread” flag to include the pthread library for thread manipulation.

Also, our machine has 4 cores, but we turned of 3 of the cores to make our system a single-core. Having multiple cores running simultaneously would have severely affected our benchmarks because in a multi-core system, the OS scheduler is constantly switching running processes in and out of all available cores. Since it is not guaranteed that clock cycle counters are synchronized across multiple cores, we would have had no confidence in our benchmarks if we did not disable all but one core. Also there is no hyper-threading in our system, so we did not have to worry about simultaneous processes or threads running on a single core. Finally, our system was not equipped with power saving features that might have required the CPU to throttle up when processes start.   
  
All these tests were written together using the software engineering technique, pair programming.

**Machine Description:**

|  |  |
| --- | --- |
| Processor Model: | Intel® Core™2 Quad CPU Q6600 |
| Cycle Time: | 2.40 GHz |
| Number of Cores: | 4 |
| L1 instruction cache size: | 4 x 32KB |
| L1 data cache size: | 4 x 32KB |
| L2 Cache Size: | 2 x 4096KB (each L2 cache is shared between 2 cores) |
| Bus Speed: | 1066 MHz |
| RAM size: | 3916 MB |
| RAM Description: | 4 x 1GB DDR2 667MHz |

|  |  |
| --- | --- |
| Disk Model: | Hitachi HDT725025VLA380 |
| Disk Firmware Revision: | V5DOA73A |
| Disk Capacity: | 250 GB |
| Disk RPM: | 7,200 |
| Disk Cache Size: | 8 MB |
| Disk Average Seek Time: | 8.5 ms |

|  |  |
| --- | --- |
| Network Card Speed: | 1000 Mbps |

|  |  |
| --- | --- |
| Operating System: | Ubuntu |
| Release: | 10.04.3 |
| Version: | 2 |
| Kernel Version: | Linux 2.6.32-28-generic |
| Major Revision: | 6 |
| Minor Revision: | 32 |
| Immediate bug fixing version: | 28 |

**CPU, Scheduling, and OS Services:**

1. *Measurement Overhead:* To report the measurement of time, we used the rdtsc method. This method allows us to measure relative time by measuring the number of clock cycles executed since start-up, which is more fine grain than other time measurement through milliseconds. This has a low overhead for reading timestamps because it reads the actual ‘rdtsc’ register (which is automatically updated by the CPU). We also made rdtsc function an inline function, so we could get rid of the cost of calling the function.  
     
   We called rdtsc() twice consecutively in a loop, which we ran over a 100,00 times. After both timestamp calls, we deducted the time from the first time from the second one. Then we got the average of the differences to get our measurement overhead in cycles. To get the overhead of the loop, we called rdtsc() before and after the loop and got the average number of cycles per loop iteration.  
   *Estimates:*  
   We estimated the overhead of reading time to be about 10 cycles since the procedure involves reading a register into two 32-bit integers i1 and i2, right shifting i2, and storing the OR of i1 and i2. We estimated the overhead of using a loop to about the same as the overhead of reading time, as they both require roughly the same number of operations. For the measurement overhead, we knew calling the time function would still cost a bit but be fairly faster than other time functions because it has low overhead. We also needed to take into consideration the loop.   
   ( <http://en.wikipedia.org/wiki/Time_Stamp_Counter> )  
     
   *Actuals:*  
   - The overhead of reading time is 77 cycles.  
   - The overhead of using a loop is 7 cycles.
2. *Procedure Call Overhead:* As the number of integer arguments increase, our expectation was for the cycles to slowly increase. Because the CPU can only move one value at a time, every function is pushing another argument on to the stack. Each procedure has a stack of activation records that include return addresses, local variables, register values, etc. So, every argument is creating a new activation record that initializes the necessary items, therefore the clock cycles should monotonically increase for each argument added to the procedure.   
     
   We created 8 different functions, each with one more argument (32-bit integer) than the last one. Basically a function for no arguments, one argument, two arguments, three arguments, and so on. We called each function 100,000 times, called rdtsc() before and after the loop to measured the average. We also took into consideration the loop overhead and deducted that time from all of our measurements.  
     
   *Estimates:*

Since calling a function is an operation commonly performed by the system, we expected it to be under 10 cycles for a zero argument function, and to increase by roughly 1 cycle for every additional argument. We know the actuals should looks like they are increasing by understand the process behind adding more arguments to a procedure. Each argument needs a stack frame would cause the cycles to rise. There would also be a bit of variation due to the scheduling and other things happening in the operating system. *Actuals:*  
0 args – 1 cycle  
1 args – 4 cycles  
2 args – 5 cycles  
3 args – 6 cycles  
4 args – 6 cycles  
5 args – 9 cycles  
6 args – 11 cycles  
7 args – 12 cycles

1. *System Call Overhead:* A system call transfers control to the operating system and raises the privileges to kernel mode. In our benchmarking for the system call getpid(), the system call takes relatively the same time as our procedure calls. This tells us that the system calls are most likely optimized in this operating system, meaning there is a “fast” control transfer instruction designed to transfer control to the kernel without the overhead of an interrupt or trap that is used to usually transfer control. This is seen in the later versions of Linux 2.5 on the x86.  
     
   We used the system call “getpid()” to measure this benchmark. We took into consideration the loop overhead and the assignment of getpid() to a variable. After calling getpid() a 100,000 times, we took the time before and after the loop to get the average. After getting the average, we deducted the assignment and loop time from the numbers.   
     
   *Estimates:*Under 10 cycles because getpid is a very minimal system call. We estimated a syscall in our system to be roughly around the same time it takes for a procedure. Linux seems to have a lot of optimizations on system calls and should be fairly fast.   
   ( <http://www.ibm.com/developerworks/linux/library/l-system-calls/> )  
    *Actuals:*  
   - The system call overhead is 2 cycles.
2. Task Creation Time: In our measurements, thread creation seems to be about 10 times faster than process creation. A process has to load more resources than a kernel thread, like virtual memory mappings, file tables, and signal-handler tables.  
     
   We used the function “fork()” to create a new process, which involves copying page tables and creating copy-on-write mappings for memory. The only other option was “vfork()”, which also creates a new process but without copying the address space. Since the classic definition of a process implies they do not share the same address space, we used “fork()”. To create threads, “pthread\_create()” was used. We recorded time before creating a new process/thread and after the process/thread terminated. As soon as a process was created it called “exit(1)” to terminate. A thread immediately terminated by returning from its function.

*Estimates:*Since creating a process involves copying the address space of the parent process and threads share memory, we estimated process creation time to be ~20 times longer than thread creation time. Task creation for Linux 2.6 in much faster than previous versions. We needed to take into consideration heavy interrupt loads, scheduling, and creation.  
( <http://www.lynuxworks.com/products/whitepapers/linux-2.6.php3> )  
 *Actuals:*  
- The process run overhead is 461811 cycles.  
- The thread run overhead is 38927 cycles.

1. Context Switch Time: Our actual times for context switching a process and thread were within 30% of each other. We expected it to take longer to context switch between a process than the thread because a process has more resources to switch over than a thread, like TLB mappings. The scheduler also has to get involved with the context switch when switching a process. In later versions of Linux, threads actually don’t take up space and are considered a “thread group,” which act more like processes.  
     
   Measuring context switch time was done by the same method for processes and threads. For each, a child process/thread was created and it simply looped 5 million times and at each iteration it called “sched\_yield()” to tell the OS scheduler to move its scheduling priority to the bottom of the queue. The parent process did the same repeated call to “sched\_yield()” after creating the child process/thread so every time the child called “sched\_yield()”, a context switch occurred to the parent and every time the parent called “sched\_yield()”, a context switch occurred to the child. The total time of creating, switching in and out of, and ending a process/thread was measured and divided by 10 million (the # of total context switches) so the cost of starting and running a process/thread was divided out in both cases.  
     
   *Estimates:*We estimated the cost of switching from a thread to be ~2 times faster than switching from a process. According to this source, which has done benchmarking for Linux Intel 5150 with 4 cores.  
   ( <http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html> )  
     
   *Actuals:*- The number of cycles per thread context switch is1465 cycles.  
   - The number of cycles per process context switch is 1761 cycles.