**Adrian Guthals**

**Brina Lee**

**12/07/11**

**CSE 221**

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

**MEASUREMENT OVERHEAD**

**Method:** 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:**  
For overhead of reading time is 77 cycles.  
For overhead of using a loop is 7 cycles.

**Standard Deviation:**

For overhead of reading time: #### cycles.  
For overhead of using a loop: #### cycles.

**Discussion:**

Our actuals for the reading time was expected be a little more than what we estimated due to the time function call. The time function call was suppose to take a little longer, along with reading the register. The loop overhead was lower than the reading time as expected because we took the time function out of the loop.

**PROCEDURE CALL OVERHEAD**

**Method:** 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, we created 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 raise. There would also be a bit of variation due to the scheduling and other things happening in the operating system.**Actuals:**

|  |  |
| --- | --- |
| **Number of Arguments** | **Number of Cycles** |
| 0 Arguments | 1 Cycle |
| 1 Arguments | 4 Cycles |
| 2 Arguments | 5 Cycles |
| 3 Arguments | 6 Cycles |
| 4 Arguments | 6 Cycles |
| 5 Arguments | 9 Cycles |
| 6 Arguments | 11 Cycles |
| 7 Arguments | 12 Cycles |

**Standard Deviation:**

**Discussion:** Our actuals were as expected because the number of cycles grew as we pushed on more arguments to the stack. As more arguments increased, the time became longer because a new stack was created.

**SYSTEM CALL OVERHEAD**

**Method:** A system call transfers control to the operating system and raises the privileges to kernel mode. In our benchmarking for the system call syscall(SYS\_gettid), 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 “syscall(SYS\_gettid)” to measure this benchmark. We didn’t use getpid() because the returns will be cached and took into consideration the loop overhead and the assignment of syscall(SYS\_gettid) to a variable. After calling syscall(SYS\_gettid) 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:**We estimated for the getpid() system call to be 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 269 cycles.

**Standard Deviation:** The standard deviation for system call overhead is 269 cycles.

**Discussion:** Our estimate for the getpid() system call was as expected. The getpid() system call is almost instantaneous because of the fast control transfer optimization in the operating system. There is hardly any overhead when control is transferred to the kernel.

**TASK CREATION TIME**

**Method:**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:**

For process run overhead: 461811 cycles.

For thread run overhead: 38927 cycles.

**Standard Deviation:**   
For process run overhead: ##### cycles.

For thread run overhead: ##### cycles.

**Discussion:**  Our expectations for the process to take longer than the thread was correct. Processes involved more when copying over their addresses while threads share memory. There are also more resources a process has to load than a thread.

**CONTEXT SWITCH TIME**

**Method:**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 a 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:**

For thread context switch: 1465 cycles.

For process context switch: 1761 cycles.

**Standard Deviation:**

For thread context switch: #### cycles.

For process context switch: #### cycles.

**Discussion:**  Our estimates were as expected. Processes have more to copy over than threads and the operating system’s scheduler has to get involved when context switching processes.

Memory

**RAM ACCESS TIME**

**Method:** In our actuals, L1 was 20% faster than L2 and L2 was 600% faster than main memory. L1 was estimated to be faster than L2 because it is closer to the processor. And L2 was estimated to be faster than main memory because L2 is the closer one to the processor. Our numbers are all relative due to the design issues and some overhead we could not control.

While calculating our actuals for RAM access time, we had to worry about prefetching. At first, we did a sequential/linear access to memory, however the prefetching caught on and started to cache accesses to L2 and main memory into L1 and L2. Therefore, to trick the operating system and to have the smallest amount of prefetching, we randomly accessed our array as in the lmbench paper. Starting at index 0, we pre-loaded each integer array with fixed strides between 0 and the length of the array such that following the indices would not create a cycle. Then we made the last index reference index 0 such that traversing the array would follow 1 cycle that randomly referenced each index in the array once before returning back to index 0. Finally, we timed 100,000,000 accesses to the array in the following manner:

time1 = *rdtsc*();

*for* ( *int* i = 0; i < 100000000; i++ ) {idx = array[idx];}

time2 = *rdtsc*();

This procedure minimized overhead such that all we had to account for was the overhead of the *for* loop and the addition of each array index to the array base pointer. Since the array was pre-loaded with fixed strides into itself, there was little the pre-fetcher could do to predict array accesses.

06ency in Cycles ) s: memory.

**Estimates:** We estimated that L1 should be faster than L2 and L2 should be faster than main memory. The manufacturer of the CPU reported a 3 cycle latency for L1 accesses, 15 cycle latency for L2, and 160 cycle latency for accesses to DDR3 16000 MHZ RAM. We had approximately 600 MHZ DDR2 RAM and a slower CPU clock rate by 0.6 GHZ, so we expected a latency of approximately 250 cycles for main memory. (<http://www.usenix.org/publications/library/proceedings/sd96/full_papers/mcvoy.pdf> )  
  
**Actuals:**

|  |  |
| --- | --- |
| **Log2 ( Array size )** | **Latency in Cycles** |
| 6 | 4 |
| 7 | 4 |
| 8 | 4 |
| 9 | 4 |
| 10 | 4 |
| 11 | 4 |
| 12 | 4 |
| 13 | 4 |
| 14 | 10 |
| 15 | 13 |
| 16 | 15 |
| 17 | 16 |
| 18 | 16 |
| 19 | 20 |
| 20 | 23 |
| 21 | 146 |
| 22 | 206 |
| 23 | 238 |
| 24 | 254 |
| 25 | 270 |
| 26 | 292 |
| 27 | 326 |
| 28 | 371 |

**Standard Deviation:** The standard deviation for RAM access time is ##### cycles.

**Discussion:** The memory latencies were 4 cycles for all array sizes within the size of the L1 cache. We expect this to be 1 cycle over our estimate because we may have failed to account for some overhead in the array index-to-memory translation. The L2 cache latencies varied for all array sizes greater than L1 but slightly less than L2, but averaged to around 16 cycles, which was very close to our estimate. The latencies varied because for array sizes closer to the L1 size, more integers would have been cached in L1 while for array sizes closer to the L2 size, more integers would have been stored in main memory. Our latencies for main memory averaged around 250 MS, which was expected. These latencies also varied because some array integers would have been cached in L1 and L2.

**RAM BANDWIDTH**

**Method***:* For measuring the RAM bandwidth on our current machine, we had to make sure to measure the path from the CPU to the Northbridge to the DRAM.  
  
We create an array bigger than L2 cache, so we can make DRAM give back a ton of data. Therefore, we could see how much we can send over the bandwidth and calculate it. At first, our numbers were a little off due to the paging. Therefore, to get rid of paging, we used “swapoff –a” to turn it off.  
  
**Estimates:** We estimated that writing should be slower than reading, maybe by 50%.   
( <http://arstechnica.com/paedia/b/bandwidth-latency/bandwidth-latency-1.html> )  
  
**Actuals:**

For reading: 12.321275 GB/sec

For writing: 5.514268 GB/sec

**Standard Deviation:**

For reading: #### GB/sec

For writing: #### GB/sec

**Discussion:** Our numbers are showing that reading is about two times faster than writing. For writing, it seems like there are some calculations that might be slowing it down. Our machine’s hardware probably also has reason to slow the writing down. However, in our case, writing is only about half times slower than reading.

**PAGE FAULT SERVICE TIME**

**Method:** Again, we had to trick the operating system to fool the prefetching optimization. So, we created a prime number and added it to our index to randomly access its memory.

In the beginning, the problem with page faults is that we couldn’t tell the difference between the numbers and which ones were true page faults. The data was being cached and we were hitting soft page faults timing from 0.01ms to 30ms. So, to get true page faults numbers, we needed to double our array and pre-write 15 million values and timed them while reading them back. The first 5 millions values were written out very quickly and cached into main memory or disk cache. Therefore, the remaining 10 million were written a lot slower as they caused the first 5 million to be paged out. Then we took the first 1.5 million and timed each read. Each read was really slow because everything was paged out. If the read was over 1 MS, we recorded it. Therefore, after plotted all the numbers in a histogram, they fell between 8ms and 10ms.

The array was indexed by a fixed stride of 1,041,553 indices to assure that a simple pre-fetching routine would not load many pages into the 8 MB disk cache.  
**Estimates:** We estimated that the page fault service time should be 2 to 3 times slower because we are going to disk. We expected out page fault service time to be around 8 MS to 10 MS. ( <http://en.wikipedia.org/wiki/Page_fault> )  
**Actuals:**

**Standard Deviation:**

**Discussion:** Page fault service times varied heavily, but the majority were between 8-10 ms, as expected. The high level of variance is most likely because disk access latencies vary heavily upon where the data is on disk (ie in the disk’s cache or physical location on the disk plate). We expect the spike between 2-3 ms to be accesses to the disk’s 8 Mb cache and accesses taking longer than 14 ms to be due to the disk arm seeking to the pages’ physical location in the disk swap space. We suspect the OS put most of the data in the swap space close to each other to limit the seek time, which explains why most of the latencies are close to 9 ms. The small spikes at 17.5 and 25.5 ms may correspond to accessing pages that are close to each other in the swap space, but still far from the majority of all pages in the swap space.

Network

**ROUND TRIP TIME**

**Method:** We setup a server on our remote computer. The server sets up a socket and waits for connection. This allows anyone to connect on the socket. We block the program using an infinite while loop until it receives data from our client. We also set up a socket on our client. Using connect(), our client connects to this socket on our server. On our server, we allow our server to accept the connection using accept().

Our client loops over 10,000 times and sends a character to the server. The server will receive the character and send a message back. The client will send over a queue to signal the server to close the connection. We measure the time before and after the loop deducting the overhead of the loop.

**Estimates:** Our bandwidth is dictated by a physical constraint, so we estimated it to be the estimated speed for an Ethernet cord, which is109 bits per second.

**Actuals:**

For round-trip time: 111.73 microseconds

For bandwidth: 117.31 MB/sec

**Standard Deviation:**

**Discussion:** By comparing remote and loopback results, the loopback results are much faster because it bypasses the network stack and is equivalence to a system call.

For both round trip time and bandwidth, how close to ideal hardware performance do you achieve?

**PEAK BANDWIDTH**

**Method:**

**Estimates:**

**Actuals:**

**Standard Deviation:**

**Discussion:**

**CONNECTION OVERHEAD**

**Method:**

**Estimates:**

**Actuals:**

**Standard Deviation:**

**Discussion:**

File System

**SIZE OF FILE CACHE**

**Method:** We started by creating files from small to large in bytes. With each file, we created different strides sizes to 1) find out at what file size will we start reading to disk and 2) which stride size we allow us to trick the operating system to not cache the file.

We had to take into consideration a bigger stride bigger than 4 MB, which is bigger than our L1. We tested a couple of different sized strides including 8MB, 4MB, 1MB, 24KB, and 12KB. But, we noticed that any stride bigger than 4MB along with a bigger file size would allow us to go beyond the file cache size. Another observation was that the operating system would also try to make their file buffer cache bigger as we try to increase the file size to read to disk.

**Estimates:** We expect the graph to have three humps indicating L1 cache, L2 cache, and main memory. They should be 4 x 32KB, 2 x 4096KB, and 3916 MB respectively. So, the graph should jump at 4 x 32KB, ( 4 x 32KB ) + ( 2 x 4096KB ), and ( 4 x 32KB ) + ( 2 x 4096KB ) + 3916 MB. The third hump should indicate our file cache size. ( <http://en.wikipedia.org/wiki/Cache> )

**Actuals:**

**Standard Deviation:**

**Discussion:** The actuals were what we were expecting. The three humps in latency increase indicates where the file is being cached, L1, L2, or main memory. It looks like the request is being sent out to disk showing that our file cache is about 227 in bytes. As the stride increased, the file cache was much easier to notice. As the stride gets bigger, there are more TLB misses. The latency for L1, L2, and main memory is about less than 10 cycles, 25 cycles and 250 cycles respectively.

**FILE READ TIME**

**Method:** For every file read time, we had to make sure we were reading directly from disk using a direct –o flag. It was important to make sure we are directly reading from disk because we wanted to make sure the file was not being read from our file cache.

To get the read time for sequential access, we used read(). We told the disk to read the entire file. First, we wrote a file to the disk. Second, we calculated the overhead of memset(), which sets the values of the bytes in the block of memory. Third, we benchmarked the times right before and after the read(). To get the read time for random access, we used lseek() and read(). This allowed us to read parts of the file randomly by creating and offset for the disk to seek and read. Like the sequential file read time benchmark, we wrote a file to disk. We calculated the overhead for copying data from disk to memory. Then benchmarked the time to read and seek the disk for the file.

**Estimates:** We estimated that the file read time for sequential access would lower. Because we were reading contiguous blocks of memory, the file read time should decrease. We estimated that random access with increase because it would cause thrashing, as the file grew bigger while seeking for the fragments. (<http://www.kernel.org/doc/ols/2009/ols2009-pages-275-286.pdf> )

**Actuals:**

**Standard Deviation:**

**Discussion:** As the file size grew, the latency for the sequential and random access read time changed around when the file size was 221 in bytes. For sequential access, the latency decreased because the file size was getting bigger and it was reading contiguous blocks of memory. However, when it hit a certain size, it split into two different blocks and disk operations to read from disk. When the random access hit a certain file size, it had to start jumping around disk to get the entire file. The disk is fragmented and started to jump around making the latency increase.

**REMOTE FILE READ TIME**

**Method:** We setup a remote NFS file system on another Linux machine and measured the overhead for looping and setting the offset. When writing the file, we used a direct –o flag to make sure we were reading from disk. Using lseek() and read(), we seek and read directly from disk and then closed the file. We are seeking to the block, reading the block, and then jumping to other blocks the current block is pointing to. For the random remote file read time, we used a random number generator to help us jump around looking for the file to write to and read from.

**Estimates:** We expected for the remote file time to slightly look like our benchmarking for the local file read times. However, the graphs would differ due the fact that we are writing and reading a file on a remote hard disk with different data and free space. Because the remote disk is a bigger hard drive, we expected the latency for sequential remote file reads to be a lot faster. However, because we are reading from a remote disk, we also expected the latency to be starting at higher than 18 cycles.

(<http://www.techotopia.com/index.php/Sharing_Ubuntu_Linux_Folders_with_Remote_Linux_and_UNIX_Systems> )

**Actuals:**

**Standard Deviation:**

**Discussion:** For our sequential remote file read times, we see the graph dip due to the free space on the other computer’s disk. The remote disk is terabyte large and half full, therefore not as fragmented as our local disk. There is a small increase the sequential remote read because the file is too big and probably reading from two blocks instead of two. The random remote read has a bigger slope because it has to seek everywhere for the file’s blocks.

**CONTENTION**

**Method:**  We read different blocks from the same file on the same disk. During our testing, we slowly increased the number of processes to run at the same time. Depending on how many processes we are currently testing, we created a barrier to wait for all the processes to start up. Once the number of processes that we are testing comes to that barrier, the barrier will release all the processes to start working at the same time.

For every process, we create an array that holds all the data to be written to a file. We open a file and initialize a barrier, using pthread\_barrier\_wait(). This barrier stops all processes and counts how many processes there are. When the barrier counts up to the number of processes we are currently testing, it releases all the processes to start together. We measured the time immediately before and after reading.

**Estimates:** We expected a linear increase in latency to number of processes. As the number of processes increase, the latency for reading a file should increase because there are other operations being done on the same disk as the file.

**Actuals:**

**Standard Deviation:**

**Discussion:** Our results were expected. We expected the latency to increase as the number of processes working on the same disk increased. Reading became slower due to other operations being executed on the same disk.