CSE221 Operating System Measurement

Boyuan Qin
bqin@eng.ucsd.edu

Dexin Qi
deqi@eng.ucsd.edu

Qiheng Wang
qiw018@eng.ucsd.edu

1 Introduction

The goal of this project is to measure the performance of the operating system, Ubuntu 12.04.4 LTS. The reason behind this goal is to gain an understanding of the relationship between the underlying hardware and the operating system, and their effects on how much time integral operations will require. With this goal, we will be running a series of justified experiments that will allow us to characterize the speed of each operation and compare them to each other. We will use the course of these experiments as a way of gauging our intuition about the performance of the operating system we are testing. The information we will obtain from these experiments will also be used in future endeavors as a set of performance results to compare against.

The language of choice in these tests will be C++. The reason for choosing C++ comes from the flexibility that C++ provides. It is a high-level language that allows us to, at the same time, work without much overhead as, for example, Java when measuring time. The compiler we are using is GCC 4.6.3-1ubuntu5 with no optimization.

The first part of the project required around 20 hours of work to complete. This included creating the tools necessary to measure the operating system and writing the report.

The experiments were split accordingly:

Measurement Overhead - Boyuan

Procedure Call Overhead - Dexin

System Call Overhead - Boyuan/Qiheng

Task Creation Time - Dexin

Context Switch Time -

2 Machine Description

The machine we used as the subject of our experiments is described in Table 1.

3 CPU, Scheduling, and OS Services

To accomplish following set of measurements, testing against a normally running OS is insufficient. Therefore, we modified a few kernel settings so that the test results are more accurate and closer to the theoretical values.

The machine we tested has two cores. We isolate one core by adding `isolcpus=0` in bootloader configuration file. All the testing scripts are run with `taskset -c 0 ./test` syntax. Therefore, we ensure minimum number of system processes interfere with our testing code, thus mini-

Table 1: Machine Specifications

Processor	Model	Intel®Core^{TM}2 Duo Processor P8600
	Instruction Set	64 bit
	Cycle Time	0.417ns(2.4GHz frequency)
	L1 data cache	32KB per core, 8-way set associative, 64-byte line size
	L1 instruction cache	32KB per, core 8-way set associative, 64-byte line size
	L2 data cache	3072KB, 12-way set associative, 64-byte line size
	FSB	1066 MHz
Hard Drive	Model	Seagate Momentus®5400.6 SATA model ST9500325AS
	Capacity	500GB
	Cache	8 Mbytes
	RPM	5400
	Physical heads	4
	Discs	2
	Average Seek Read	14ms typical
	Full Stroke Seek	30ms
	Average Latency	5.6ms
	Track to track seek time	1ms typical
	I/O data transfer rate	300 Mbytes/s max
Memory	Capacity	Dual channel(symetric), each 1024 MBytes
	Frequency	DDR3 PC3-10700 667 MHz
	Width	64 bit per channel
Network Card	Model	Intel Wifi Link 5100
	Data Transfer Rate	300Mbps
OS	Linux Distribution	Ubuntu 12.04.4 LTS GNU/Linux 3.8.0-35-generic i686

mizing context switching overhead. Interrupts are also disabled during testing. Finally, only one measurement is run at one time.

3.1 Measurement Overhead

The goal of this section is to report the overhead of reading system time and the overhead of using loops to measure iterations of an operation. These measurements are important and necessary for accuracy of future experiments.

3.1.1 Experiment Methodology

we used and modified the `cycle.h` package from FFTW[1]. The `cycle.h` is a superset of the rdtsc which optimizes according to different machines. To read the time stamp counter, use `getticks()` function. To calculate the difference of two time stamp, use `elapsed` function, which returns a double precision value. To measure the overhead time for reading time, we simply executed `getticks()` twice in a row and took the difference. The idea behind this methodology was to start tracking immediately before the the execution of the next command and stop tracking as soon as the command is executed. The difference between the two `getticks()` times will give us the number of ticks taken between the two methods. Similarly, measuring `elapsed()` required that we add two `getticks()`, one before and one after, the `elapsed()` method. We then determined the difference between the two `getticks()` methods that preceded and followed the `elapsed()` method. Taking that difference, we subtracted the time taken for one `getticks()` method to account for the time necessary to execute the succeeding `getticks()` to arrive at our final result.

To measure the overhead time for using a single loop, we chose to measure a `for` loop that will increment a variable from 0 to 10000, and divided it by 10000 to obtain the overhead taken by a

Table 2: Reading overhead performance (in cycles)

Operation	Hardware Est.	Software Est.	Prediction	Mean	Std. Deviation
`getticks`	<20 (8.3 ns)	<5 (2.1 ns)	<25 (10.4 ns)	45 (18.5 ns)	˜
`elapsed`	<25 (10.4 ns)	<5 (2.1 ns)	<30 (12.5 ns)	36 (14.4 ns)	˜
`for` loop	3 (1.2 ns)	2 (0.8 ns)	5 (2.0 ns)	7 (2.8 ns)	˜

single iteration, which includes incrementing and checking. In order to measure purely the looping part of the `for` loop, we put the variable declaration before the first `getticks()` method. This way, we measured the time taken by the loop to check and increment the variable. We also took into account the time taken by the `getticks()` method and subtracted it from our final time. Our motive behind choosing to test the `for` loop comes from our own foresight of using `for` loops plentifully in future tests.

Each test was ran 10000 iterations, the mean was taken. Each experiment was ran 100 times, and the average and standard deviation were taken.

In order to accurately translate ”ticks” into human time, a preliminary measurement of the correlation is made between ticks and number of cycles is 1 to 1. Since each cycle takes 0.416 ns, given our CPU frequency, we can make the correlation that 1 tick is equivalent to 0.416 ns.

3.1.2 Prediction

We originally predicted that, for `getticks()`, the operation should be less than 20 instructions. Assume CPI is 1, it would take the hardware less than 20 cycles to finish this instruction. And elapsed should take a little more time to execute since it has subtraction, while `getticks()` is simply reading data. Since the OS has to manage procedure call and returning values, we think software overhead should be less than 5 cycles.

For `for()` loop, with one arithmetic operation, one comparison and one jump, it should take about 3 cycles. The OS will have to initialize the loop counter, which should be less than 2 cycles.

3.1.3 Experiment Results

The experimental results are presented in Table 2.

3.1.4 Result Discussion

The predicted performance was less than the measured performance. Our predicted performance was based on the naive assumption that one task, such as a jump, would only take 1 cycle to perform. However, looking at the measured operation times, our assumption was proven to be false. This means that the operating system and hardware were taking more cycles to perform each task. A possible reason is that the compiler translate one line of code into several instructions. Another reason might be CPU scheduling and context switching between processes.

To note, both `getticks()` and `elapsed()` functions are modified to be of inline attribute. Therefore, the measurement would eliminate the overhead of procedual calls. On the other hand, the measured operation time was close to the predicted operation time for the `for()` loop. The difference here resided in our overlooking one or two operations while making our prediction.

Our methodology centered around measuring the simplest possible operation for reading time and for looping. Because we measured time immediately before and immediately after the operation of interest, we successfully captured the entirety of the operation. With this information, we will be able to use it as a base from which we can accurately measure other performances.

3.2 Procedure Call Overhead

The goal of this section is to report the overhead of making a procedure call with arguments. The number of arguments the procedure takes in will also be taken into account in a series of tests. Each test will take in different amounts of arguments ranging from 0 arguments to 7 arguments. These measurements are important and necessary for accuracy of future experiments.

3.2.1 Experiment Methodology

To measure the overhead time of a procedure call, we measured the time to make a call to a procedure that takes in the parameters given and does not processing thereafter. The procedure itself does not return anything either. In order to take into account the different time costs as a result of varying amounts of parameters, we made procedure calls with 0 to 7 parameters. Each procedure call with different amounts of parameters was tested 1000, and the average and standard deviation were taken. Each procedure call was precluded and succeeded by two different `getticks()`. Because of this, we also took into account the amount of overhead `getticks()` populated and subtracted that from the total in each procedure call.

Each of the procedure call did no processing of its arguments nor did it have a return statement. The decision to make the procedure call in this way was motivated by our goal of obtaining purely the overhead of making the call to a procedure. It was decided that return statements and any processing of data would not be part of the procedure call itself, and thus not included as part of the testing processes.

3.2.2 Prediction

The number of parameter and size of parameter should both affect the time it take for a processor call. Without parameters, a procedure should simply be two jump instructions. Thus, we predict that the hardware cost of procedure call with no parameters should be about 2. The OS need to remember where the processor is called so that it can return, which should only take 1 cycle. As parameter size grow, the OS will need to do more operations to copy the parameters.

3.2.3 Experiment Results

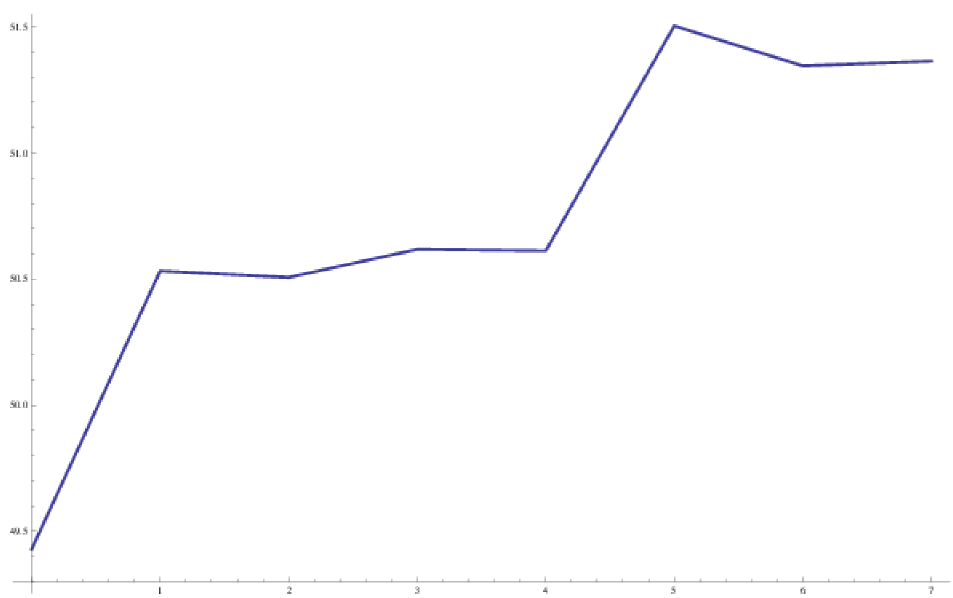

Figure 1: Procedure call performance (reading overhead included)

3.2.4 Result Discussion

An interesting discovery is that, the relationship between number of parameters and the time to issue a procedure call is not a strict linear relation. That indicates that, the OS is able to copy multiple parameters(integer) in one cycle, which resulst in such a measurement result. It appears that, every 4 integer parameters added will add one extra cycle of overhead of a procedure call. Thus, we can get to the conclusion that the os is able to copy 128 bits of data at the same time.

Notice that we did not substract the reading overhead in the graph, so the actual overhead of procedure call should be 4 6 cycles. Thus, the overhead of a procedure call is quite small. If the parameter size is large it will add to the overhead, but still pretty fast.

3.3 System Call Overhead

The goal of this section is to report the overhead of making a system call and compare it to the cost of making a procedure call. The system call with the least overhead was chosen to accurately capture the time taken to make the the switch from user mode to kernel mode. These measurements are important and necessary for accuracy of future experiments.

3.3.1 Experiment Methodology

To measure the overhead time of a system call, we attempted to choose a method with as little overhead as possible. Initially, our goal was to used `getpid()` as our system call of choice. However, due to the potential problem of having the `getpid()` system call getting cached and thus not having an accurate overhead time of trapping into the kernel space, we decided to choose a similar method, `getppid()`. In order to measure the time of `getppid()`, we went about by simply adding `getticks()` before and after the system call and calculated the average difference and its standard deviation over 10000 runs. This method was similar to the methodology described earlier in measuring loop overhead. We then compared the results that we received to the results we measured earlier for procedure calls.

3.3.2 Experiment Results

3.3.3 Result Discussion

3.4 Task Creation Time

The goal of this section is to report the time taken to create and run both a process and a kernel thread, and to compare the two time costs.

3.4.1 Experiment Methodology

In order to measure time taken to create and run both a process and a kernel thread, we used the basic principle of precluding and succeeding the target section of code as we have prior. In order to create a kernel thread, we used the method `pthread_create()` to create and run a thread.

`pthread` allows us to avoid having to write kernel code and this flexibility was one of the reasons why we decided to use it. The other reason is that the thread is created executing any argument given as part of its parameter. This means that we will not need to explicitly call any end operation on the thread.

The arguments given to the `pthread_create()` method were defaulted as to avoid overhead that is separate from the task creation and minimal amount of running. This test was repeated 10000 times, and the average and standard deviation were taken while taking into consideration the overhead of reading time. We used the same procedures for processes. We created processes using `fork()` and terminated each process using `_exit()` immediately after creation and running. Each experiment has 10000 iterations, we run 100 experiments for process creation, and 1000 for thread creation(since process creation is too slow) and the average and standard deviation were taken while taking into consideration the overhead of reading time.

3.4.2 Prediction

The time needed to create a task is really difficult to predict, since it really dependents on how OS define a task, which means details such as process data structure, thread data structure, implementation algorithms, will all affect the time. Since we have little knowledge of linux kernel, we don't think we can make a reasonable guess. One thing we would expect is that, the time to create a process should be longer than creating a thread, since processes are more heavy weighted than thread.

3.4.3 Experiment Results

Table 3: Thread creation performance (reading time included)

operation	Mean of experiments	Standard deviation
create process	209194.13(87.02 microsec)	28245.38(11.75 microsec)
create kernel thread	3030.83(1.25 microsec)	68.57(28.53 ns)

3.4.4 Result Discussion

From the result, we could observe a significant difference between the time it took to create a process and a kernel thread. We believe it is because processes have much more complicated structure than threads. For example, threads share memory while processes have separate address spaces. Such a difference is reasonable in the sense that, threads are the minimum unit of scheduling, and processes take advantage of threads to get better performance. One more thing to mention, we actually measured the time it take for a kernel module to create a kthread, and it seems to be even more time consuming than process creation. We are wondering why would that happen, since a kernel module is already in kernel level, it won't need to cross the boundary of user level and kernel level again, why would it take so much time? Not to mention it is a thread not a process.

3.5 Context Switch Time

The goal of this section is to report the time taken to context switch from one process to another in comparison to the time taken to context switch from one kernel thread to another. A context switch in essence is the process of storing current process state and restoring another. More specifically, storing a state means changing the process's state to ready or blocked; restoring a state brings the process from ready to running.

The overhead time spent in a context switch consists of several parts. First and foremost, by definition time needs to be spent saving and restoring a process's states. Secondly, process caching may also influence the overhead time. Thirdly, virtual memory mapping, synchronization of memory caches, paging, these elements also may not be ignored when switching between processes. Finally, interrupts may occur in the middle of measurement.

In this paper, we mainly compare a process-level context switch and a kernel thread-level context switch.

3.5.1 Experiment Methodology

In order to measure the time taken to make a context switch between two processes, we utilized blocking pipes to ensure synchronization. The way we designed the test to measure just the time for context switching required the use of the pipe's ability to pass data.

At first, we tried to use a simple `fork` followed by and `exit` in the child process, and use `waitpid` in the parent process in order to achieve synchronization. The implementation of this method is

Table 4: Context switching overhead performance (in cycles)

Operation	Hardware Est.	Software Est.	Prediction	Mean	Std. Deviation
Process	<50000	<20000	<60000	71582.43 (0.0286 ms)	507.32 (0.001 ms)
Kernel Thread	<10000	<10000	<20000	23425.98 (9760.8 ns)	211.34 (88.058 ns)

trivial, but it turned out to generate irregular results. Forking child processes within a loop easily drains memory, while using `waitpid` adds a unstable overhead to the measurement.

For user-level process context switching, we eventually decided to use a pipe mechanism to ensure synchronization. We first spawned a child process wherein a time was taken before a `write()` procedure to the pipe with the time passed into the pipe as part of its arguments. The child process immediately calls exit after and a context switch is made back to the parent process. Within the parent process, we immediately call a `read()` procedure and store the first time taken in the child process. A second time is taken right after and the difference is taken. In the parent process, we used waitpid() to ensure the following order of execution: parent, child, parent. Because pipes forces context switch into the kernel, the significant overhead of the `write()` and `read()` operations were also measured and taken into account in the final calculations. Each context switch was performed 10000 times and an average was taken. This measurement tool was ran 10000, and an average and standard deviation was taken for the final results.

For kernel-level thread context switching, we also used pipe to force two threads to synchronize. First, we created two pipes p1 and p2. In both of pipes, define the first integer as reading, and the second integer is writing. Then, we use `pthread_create()` and `pthread_join()` to create two threads: Ta and Tb. Set the pthread scope `PTHREAD_SCOPE_SYSTEM`to ensure that this thread is a kernel thread. After this, we timestamp the initial time t0 in the start of thread Ta. Let both threads communicate to each other though the two given pipes. Ta writes data to p1 and listens to p2; Tb writes data to p2 and listens to p1. The communication is repeated 1000 times in both processes. In the end of Ta, get time stamp t1. The time duration $dt = t1 - t0$ includes 2000 context switches, so we can get the average context switch overhead by $dt/2000$. This precess is repeated 100 times.

3.5.2 Prediction

The prediction for context switching is not easy. But we can be sure that a kernel-level thread context switching would spend way less time than a process one.

For process context switching, a process must call `wait`, jump into kernel mode, then the kernel picks the right process to wake it up, pass parameters. If the process is switched out to memory, hardware may also add extra overhead. For kernel threads, since threads share the same file descriptors, signals, etc. , we estimate a smaller hardware overhead.

3.5.3 Experiment Results

We use pipe to communicate between two processes. Every time when a process tries to read from an empty pipe, it will be blocked and wait for another process to fill out the pipe.

3.5.4 Result Discussion

3.6 Time Consumption

Boyuan Qin:
Dexin Qi: 22+ hours
Qiheng Wang:

3.7 References

http://en.wikipedia.org/wiki/Penryn_(microprocessor)
http://tuxthink.blogspot.com/2011/02/kernel-thread-creation-1.html
http://jahanzebnotes.blogspot.com/2013/02/turn-off-cpu-throttling-ubuntu.html
http://www.seagate.com/staticfiles/support/disc/manuals/notebook/momentus/5400.6%20(Wyatt)/100528359e.pdf
http://www.cpuid.com/softwares/cpu-z.html

References

[1] Cycle Counters. Available at `http://www.fftw.org/download.html`.