CSC 501 OPERATING SYSTEMS – PROJECT 1

Nikhil Ramesh – nramesh2
Sharath Sreenivasan – ssreeni

1. INTRODUCTION

The project involved the performance measurement of various system characteristics (mainly CPU and memory). Determining these measurements is crucial for gain a better understanding of how the underlying Operating System performs its tasks. To achieve this, a set of experiments were written and applied to obtain the performance measurements and consequently draw conclusions from them.
The following experiments were conducted by Nikhil Ramesh –

- Procedure call overhead
- Task creation time
- RAM access time
- RAM bandwidth

The remaining 4 experiments were conducted by Sharath Sreenivasan.

The language, ANSI C, was used to implement the experiments using the XCode IDE [Version 8.0] with the GCC Compiler(GNU Compiler Collection)

The amount of time spent on this project including drafting up of the report amounted to around 25 hours.

2. MACHINE DESCRIPTION

A detailed description of the machine and its properties are given below:-

- Processor
  Model – Intel® Core™ i5-3210M CPU @ 2.50GHz
  Cycle Time – 0.4 ns
  Vendor – GenuineIntel
  L1 Cache – 32768 bytes
  L2 Cache – 262144 bytes
  L3 Cache – 3145728 bytes
- Memory Bus – 1600 MHz
- I/O Bus – SATA (AHCI Version 1.30 Supported)
  RAM Size – 8,589,934,592 bytes
- Disk
  Capacity - 500,107,862,016 bytes
  RPM – 5400
  Controller Cache Size –
- Operating System
  OSX El Capitan - 10.11.6

Further details can be found at – https://support.apple.com/kb/sp649?locale=en_US

3. EXPERIMENTS AND RESULTS

The time durations and latency measurements for every experiment were computed using the pre-defined structure timeval t. A user defined function was used to calculate the time by using the structures member variables.

- **Measurement Overhead**

  For loop overhead, an empty loop with a fixed number of iterations was executed several times.

  Estimated value – Executing a loop would involve loading the conditional operand, executing the check condition, performing the updation instruction as well as loading the program instruction from the program counter, we estimate the software overhead to be around 4-5 clock cycles which is approximately 2ns

  All values are in ns

Base Hardware	Software Overhead	Estimated Time	Actual Time
0.4	1.6	2	2.44

The time taken for the execution is as shown below :-

LOOP OVERHEAD	
	2.4
	2.3
	2.4
	2.3
	2.2
	2
	2.7
	2.5
	2.4
	3
	2.3
	2.8
MEAN	2.441667
STD DEVIATION	0.274552

Fig 1 – Loop Overhead (in ns)

The actual value is close to our predicted value.

For the overhead of reading time, the experiment is performed by calculating the average time spent between two RDTSC calls.

Estimated Value – Based on information gathered from stackoverflow.com we predicted that the overhead of using RDTSC would be around 100 cycles

All values are in ns

Base Hardware	Software Overhead	Estimated Time	Actual Time
20	20	40	24.5

The read instructions were sampled for a given number of iterations and the results are as shown below :-

	READ OVERHEAD
	27
	27
	23
	27
	28
	23
	23
	22
	24
	22
	23
	26
MEAN	24.58333
STD DEVIATION	2.234373

Fig 2 – Over head of reading time (in ns)

The estimated performance showed some difference from the actual result. This could be due to the fact that the time taken to read the TSC was over estimated.

- **Procedure Call Overhead**

  The experiment involved computing the overhead of a procedure call based on the number of integer arguments for a function call. It was performed by executing a simple function call to a method which returned an integer value. The time was recorded for a fixed number of iterations (100,000) while increasing the number of arguments in the function call within the loop in each successive trial.

  Estimated Value – Procedure call would involve the use of the stack counter, transferring counter, as well as by pushing and popping local variables so we estimate the process to take around 8 clock cycles

  All values are in ns

Base Hardware	Software Overhead	Estimated Time	Actual Time
-	3.2	3.2	3.5

  Actual time is considered using 3 parameters

  The results of the experiment are shown below :-

	PROCEDURE CALL TIME DURATIONS							
ARGUMENTS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	3.2	3.3	3.6	4	5.5	3.3	4.1	4.7
	2.6	2.6	3	3.5	4.1	3	3.3	4.2
	3.7	2.8	3	3.2	3.5	3.3	3.5	6
	2.9	2.7	3	3.4	4.6	5.8	3.5	4.6
	2.7	2.8	3.2	3.5	5.2	3.2	3.5	5.3
	2.8	2.8	2.9	3.6	3.5	4.1	3.4	5.9
	2.6	2.8	3.3	3.5	3.3	3.2	3.5	4.2
	2.8	3.9	3.1	3.5	3.5	3	3.5	4.5
	2.6	2.8	3.1	3.3	4.6	3.1	3.5	4.3
	3.3	2.8	3	3.5	3.4	3.2	3.5	4.6
	2.8	2.8	3.2	4.1	3.5	3.1	3.5	4.7
	2.8	3.2	3.1	3.2	3.6	3.2	3.3	4.5
MEAN	2.9	2.941667	3.125	3.525	4.025	3.458333	3.508333	4.791667
STD DEVIATION	0.335749	0.36045	0.186474	0.276751	0.76411	0.791384	0.202073	0.614164

Fig 2 – Mean and standard deviation for procedure calls (in ns)

The estimated performance agrees with the actual performance. Also, it could be seen that the increment overhead due to additional parameters is around 0.5-1 clock cycle.

- **System Call Overhead**

  The experiment involved computing the overhead of a system call. It was performed by executing a simple system call (write()). The time was recorded for a fixed number of iterations and multiple trials were performed.

  Estimated value – After reading a few stackoverflow.com posts and wikipedia.org articles we came to the conclusion that the base hardware + overhead of a system call is between 400-500 clock cycles.

  All values are in ns

Base Hardware	Software Overhead	Estimated Time	Actual Time
100	60	160	196

  The results of the experiment are shown below :-

	SYSTEM CALL TIMES
	199.4
	256.1
	192.5
	194.4
	179.5
	195.3
	191
	181.7
	179.6
	197.3
	192.9
	194
MEAN	196.1417
STD DEVIATION	20.0559

Fig 3 – Mean and Standard Deviation for System Calls (in ns)

The actual system call overhead was around 500 clock cycles which was in the range of our estimation.

The system call, as we observed, is significantly more expensive than a procedure call (200ns compared to around 3-4 ns). This is because a system call involves various extra actions which do not occur during a simple procedure call, such as overheads in the processor determining which system call has occurred, time taken to perform context switching, among others.

- **Task Creation Time**
  The experiment involved computing the time to create and run both a process and thread. To create a thread, the pthread_create() method was employed, while for the process, the fork() system call was used.

  Estimated Value – We found it difficult to estimate the time taken to create and run threads and processes. We couldn't find viable sources which explained the steps involved in these tasks

  As observed, the process creation time is much larger than the thread creation time (approximately 40 times as much). This is because process creation involves the use of a system call like fork() or clone(), which in itself carries a certain latency. Also, process creation involves tasks such as copying tables and creating Copy-on-write mappings for memory which don't occur with thread creation. Processes, however, are safer than threads since each process runs in its own virtual address space.

	PROCESS CREATION (secs)
	0.0103
	0.0679
	0.0068
	0.0078
	0.0078
	0.00708
	0.007225
	0.007088
	0.007131
	0.007286
	0.007133
	0.007255
MEAN	0.012567
STD DEVIATION	0.01745

Fig 4 – Mean Process Creation Time

	KERNEL THREAD (microsecs)
	27
	29
	25
	46
	25
	25
	23
	27
	28
	31
	24
	27
MEAN	28.08333
STD DEVIATION	6.067174

Fig 5 – Mean Thread Creation TIme

- **Context Switch Time**
  The threads were created using the pthread_create method. The thread context switching was done using semaphores. It is achieved by alternately locking and unlocking the semaphore using sem_wait and sem_post respectively.

Estimated Value for thread – Based on the stackoverflow post (link provided below) we predicted the context switch time between threads to be 800-1000 clock cycles
http://stackoverflow.com/questions/304752/how-to-estimate-the-thread-context-switching-overhead

We also assumed more overhead due to the hardware when compared to the software.
All values are in ns

Base Hardware	Software Overhead	Estimated Time	Actual Time
240	80	320	262

The mean context switch time for threads is as shown below

THREAD CONTEXT SWITCHING	
	255
	255
	253
	260
	257
	265
	265
	261
	267
	278
	263
	265
MEAN	262
STD DEVIATION	6.862282

Fig 6 – Mean thread context switching time (in ns)

The actual overhead was around 200 clock cycles lesser than our estimation. This could be because the hardware performance estimate could have been inaccurate.

The process context switching was achieved by using 2 blocking pipes. By causing one process to read from an empty pipe and the other process to write into a full pipe context switching is forced.

Estimated Value for Process – During a process context switch, the parent's address space will have to be changed to its child's while the OS would also have to switch to the kernel mode. Referring the link posted below, the expected time

for a process switch was around 6000 ns with much of it attributed to Software overhead.
http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

All values are in ns

Base Hardware	Software Overhead	Estimated Time	Actual Time
2000	4000	6000	6625

The values for the mean context switching time is as shown below

PROCESS CONTEXT SWITCHING	
	5500
	7500
	6500
	6000
	8500
	6500
	6000
	6500
	7000
	7000
	6000
	6500
MEAN	6625
STD DEVIATION	801.2774

Fig 7 – Mean process context switching time (in ns)

The actual context switch overhead was about 1000ns higher than the estimate which could be because the OS took longer to switch from user mode to kernel mode

It can be seen that the process context switching time is notably greater than the thread context switch time because processes are inherently heavier than threads.

- **RAM Access Time**
  To compute the latency for individual integer accesses to the caches and main memory, an array of pre-defined size was used. The average time to read one integer from the array is computed by using a loop. For every successive trial, the size of the array was increased by a power of 2.

  Estimated Value – From the lmbench Intel publication and Wikipedia.org, we found that RAM access time will increase sequentially as the size of the cache

increases. By taking into account, the loop overhead, we estimated that the access times as follows –

L1 cache – 5ns
L2 cache – 10ns
L3 cache – 20ns
Main Memory – Greater than 40 ns

Assumed the base hardware and software overhead to be around the same.

All values are in ns

Memory	Base Hardware	Software Overhead	Estimated Time	Actual Time
L1	2.5	2.5	5	6.1
L2	5	5	10	8.99
L3	10	10	20	20.32
Main	22	22	44	44.77

The results are shown below in the form of a graph:-

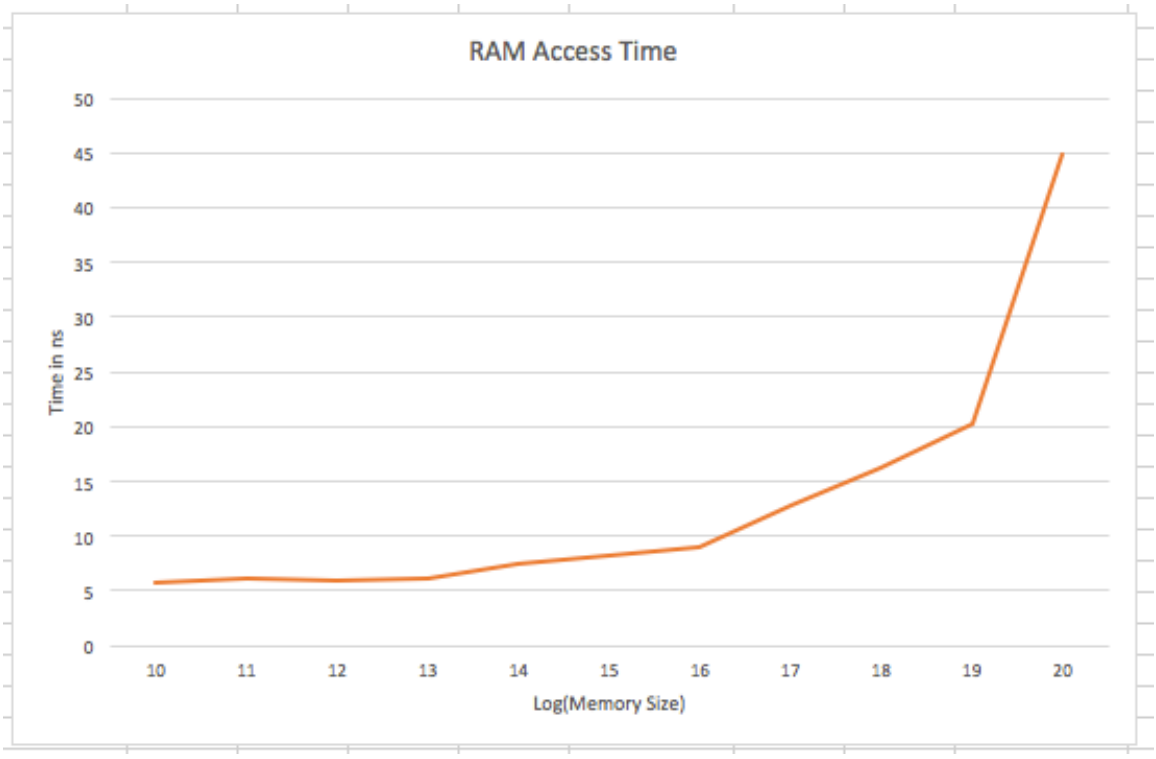

Fig 8 – RAM Access Time

Our estimations were in the range of the actual result. The difference could be a misjudgment in the cache speed.

As it can be seen, there are 3 clearly defined regions of the line graph corresponding to the 3 cache memories present in the system (L1, L2 and L3). The L1 region extends till 13 indicating the size of L1 cache to be 2^13 * 4 = 32768 bytes. The L2 region extends from 13 till 16 indicating the size of L2 cache to be 2^16 * 4 = 262,144 bytes. The L3 region extends from 16 till 19 indicating the size of L3 cache to be 2^19 * 4 = 2,097,152 bytes beyond which is the region corresponding to the main memory.
The size of the caches obtained from the graphs was consistent with the actual size given in the system info except for the size of the L3 cache which was slightly lesser than the actual size.
Increasing the array beyond the size of 2^20 resulted in a segmentation fault due to which it was difficult to obtain the exact latency for access to the main memory.

- **RAM Bandwidth**
  For RAM read bandwidth - The experiment involved initializing all elements of an array to a given number. The time to read all the elements from the array was performed with the help of a loop. Loop unrolling was used to reduce the overhead encountered. The bandwidth was calculated using the following formula:

  Bandwidth = (Total bytes read)/(Time taken to read all bytes)

  Estimated Value – From the system specs,
  Memory Bus Frequency = 1600 MHz. The bus has 2 lines each of width 64 bits.
  Therefore, the predicted theoretical read bandwidth = 1600 MHz * 8 = 12.8 GB/s
  Most of the overhead is attributed to memory bus hardware.

  All values are in GB/s

Base Hardware	Software Overhead	Estimated Time	Actual Time
12	0.8	12.8	1.68

  The results of the experiment are as shown below

	RAM READ BANDWIDTH (GB/sec)
	1.7414
	1.741
	1.7357
	1.6357
	1.7463
	1.7379
	1.6424
	1.7463
	1.7331
	1.4676
	1.7335
	1.5985
MEAN	1.688283
STD DEVIATION	0.086722

Fig 9 – RAM read bandwidth

The huge disparity in the values could be because of some caching policy which hasn't been considered.

For RAM write bandwidth - The experiment involved writing elements into an array with the help of pointers. The time taken to write all the elements was noted and the bandwidth for writing is calculated using the following formula –

Bandwidth = (Total bytes written)/(Time taken to write all bytes)

	RAM WRITE BANDWIDTH (GB/sec)
	1.5325
	1.3063
	1.68
	1.502
	1.8775
	1.8513
	1.7613
	1.5263
	1.7925
	1.9137
	1.4975
	1.4325
MEAN	1.63945
STD DEVATION	0.198581

Fig 10 – RAM write bandwidth

The bandwidths for read and write are found to be almost similar, without any anomalies.

- **Page Fault Service Time**
  To compute the time taken to fault an entire page, the mmap() method was used. A file of size 33 MB was mapped into the virtual address space using the mmap(). By attempting

to read a character from the mapped virtual file (using strides of 1 MB), a page fault would be forced to occur, since the page would have to be fetched from the disk.

Estimated Value – When a page fault takes place, the page has to be fetched from the disk. Given the hard disk read speed of around 100 MB/s, the page transfer time would be 0.04 ms. The hard disk access time is around 0.2 ms and assume the overhead due to context switching and accessing the page from the file to be 0.05 ms. The page fault service time is estimated at around 0.3 ms.

All values are in ms

Base Hardware	Software Overhead	Estimated Time	Actual Time
0.24	0.05	0.3	0.001

100 trials were performed and he results shown below -

	PAGE FAULT (microsecs)
	1.52
	2
	1.55
	1.59
	1.53
	2.02
	1.54
	1.6
	1.94
	1.99
	1.63
	2.27
MEAN	1.765
STD DEVIATION	0.260192237

Fig 11 – Page Fault Service Time

The experimental values were observed to be off from our estimation by a factor of 100. Even after multiple attempts, the experimental results did not change. One main reason for the disparity could be because only minor page faults were being enforced instead of major page faults though we were unable to ascertain the cause behind it.

However, it could be clearly seen that the time taken to access a byte from the disk would be 0.1 us (Dividing the estimated page fault time by the page size) The time taken to access a byte from the main memory was found to be around 40 ns. Thus, it takes much longer to access a byte from the disk than the main memory.

4. SUMMARY TABLE

All the values shown are in ns unless otherwise specified

Parameter	Base Hardware	Software Overhead	Estimated Time	Actual Time
Loop Overhead	0.4	1.6	2	2.44
Overhead of Reading Time	20	20	40	24.5
Procedure Call overhead. # of params = 3	-	3.2	3.2	3.5
System call overhead	100	60	160	196
Task Creation Time (Process and Thread)	NA	NA	NA	Process - 0.012s Thread – 28.08 us
Context Switching Time (Thread)	240	80	320	262
Context Switching Time (Process)	2000	4000	6000	6625
Memory (L1 cache)	2.5	2.5	5	6.1
Memory (L2 cache)	5	5	10	8.99
Memory (L3 cache)	10	10	20	20.32
Memory (Main)	22	22	44	44.77
RAM Bandwidth (Read)	12	0.8	12.8	1.68
RAM Bandwidth (Write)	12	0.8	12.8	1.63
Page Fault Service Time	0.24	0.05	0.3	0.001