# Principles of Operating Systems

# Performance Measurements

By
Laxman Sole - 200112088
Aditya Gulavani - 200111706

# Introduction

As project title suggests, goal of this project is to measure the overheads of various features/tasks of operating systems. Along with this, goal is to measure and understand the role/support of hardware, required to to implement these features.

In this project, we measured the overheads for measurement(!), time required for procedure calls and how it varies for same procedure with different number of arguments, different system call overheads,task creation time and time required to switch between different tasks i.e. context switch. All these quantities depends on how underlined operating system is designed. E.g. During task creation, context switch OS itself can perform optimizations to achieve the goal of running OS minimalistic and releasing CPU to other processes as soon as it can.

In addition to this, we also measured different memory latencies. These includes different
cache latencies, RAM access time, RAM Bandwidth page fault service time etc.

The main motivation for this project is to make the person reading/implementing the importance of Operating System and its working. After we were done implementing the project, various concepts of Operating Systems became very clear. For instance, in the experiment with Context Switch, we had to make sure there is proper and consistent inter process communication. For that, we implemented pipes. And we made sure that only a single processor core is taking part in this operation. We could see that until the parent had written into the pipe, the child was in blocked state, waiting to read the data. But, this would have been just a concept of operating system had we not actually implemented it.

So, basically, this project revolves around operating system concepts and their actual implementation. Below is the work division among the group members and after that we discuss each of the task mentioned above in detail.

Aditya has been responsible for process, scheduling and other algorithms, while Laxman has taken care of memory related operations.

## Hardware Information:

- Manufacturer: Hewlett-Packard
- Product Name: HP Pavilion dv6 Notebook PC
- Processor: Core 2 Duo - Intel(R) Core(TM) i5 CPU  M 450  @ 2.40GHz

- Available frequency steps: **2.40 GHz**, 2.27 GHz, 2.13 GHz, 2.00 GHz, 1.87 GHz, 1.73 GHz, 1.60 GHz, 1.47 GHz, 1.33 GHz, 1.20 GHz
- Available cpufreq governors:= **performance,** conservative, ondemand, userspace, powersave
- flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx **rdtscp** lm **constant_tsc** arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 popcnt lahf_lm tpr_shadow vnmi flexpriority ept vpid dtherm ida arat

- Cache Details:=
    - L1 Cache:
        Operational Mode: Write Through
        Installed Size: 32 kB
        Maximum Size: 32 kB
        Associativity: 8-way Set-associative
    - L2 Cache
        Operational Mode: Write Through
        Installed Size: 256 kB
        Maximum Size: 256 kB
        Associativity: 8-way Set-associative
    - L3 Cache
        Operational Mode: Write Through
        Installed Size: 3072 kB
        Maximum Size: 3072 kB
        Associativity: 12-way Set-associative
- Ram
    - Manufacturer: Hynix
    - Part# - HMT125S6BFR8C-H9
    - Speed: 1067 MHz
    - Size: 2048 MB x 2
    - Form Factor: SODIMM

## Software Information:

- Operating system:
  Distributor ID: Ubuntu
  Description: Ubuntu 16.04.1 LTS
  Release: 16.04
  Codename: xenial

- Kernel: Linux 4.4.0-38-generic #57-Ubuntu SMP Tue Sep 6 15:42:33 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux

# 1. Measurement overhead:

### a. Measurement using gettimeofday() :

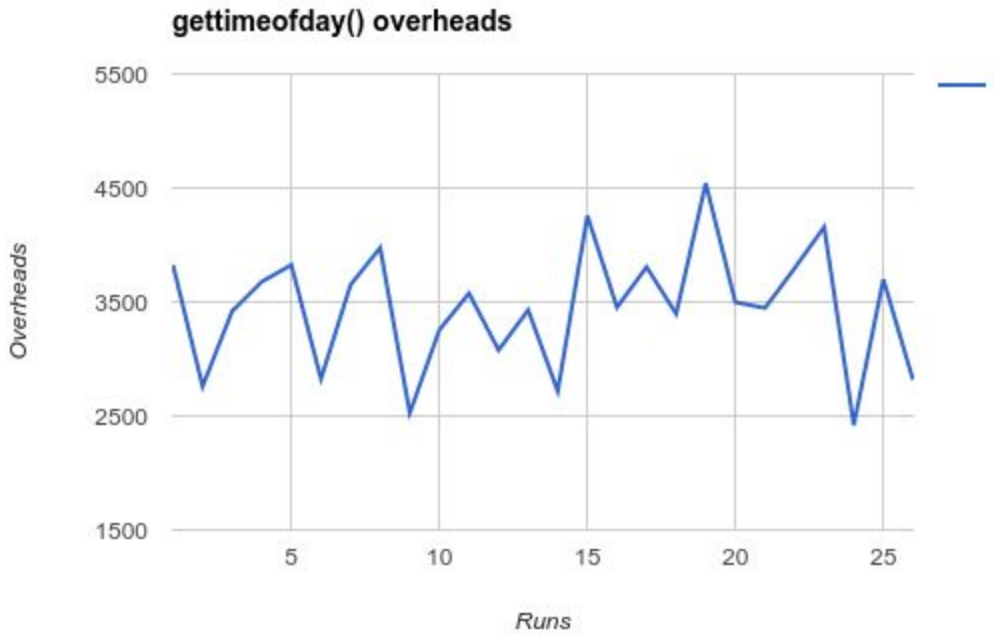
There were several disadvantages for using the gettimeofday(). First, it is a system call. That implies it will add a software overhead to the measurement. And as for anything related to software, this overhead will depend on the state of the machine at that instance.

Second, it is unreliable in a sense that it is not guaranteed to reproduce same results even if the workload remains same (we will see what this means in rdtscp).

Third, we suspect that because of the arguments that are needed by gettimeofday(), there is some more overhead.

Although inconvenient, we measured the overhead required by this mechanism using a more sophisticated technique (discussed below). The table below shows the time taken by the gettimeofday() system call.

| For gettimeofday(): 27 readings below | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3825 | 2760 | 3420 | 3678 | 3825 | 2826 | 3651 | 3975 | 2526 |
| 3255 | 3576 | 3078 | 3429 | 2718 | 4257 | 3453 | 3807 | 2718 |
| 3396 | 4542 | 3498 | 3447 | 3795 | 4158 | 2421 | 3699 | 2817 |
| Variance | 297641.2564 | | Median | 3453 | | Std. Deviation | 545.5650799 | |



As we can see the standard deviation in the measurement mechanism itself is 545 cycles. Which is absolutely unreliable. This lead us to look for an alternative to measure the operations.

**b. <u>rdtscp :</u>**

This method has been discussed at length in the Intel whitepaper. The PC that we ran our experiments supported both rdtscp and constant_tsc. The importance of having rdtscp is that this instruction waits for the previous rdtsc instruction and only then executes. This means even in case of out of order processors, these instructions will be executed in order and hence timing is guaranteed to be accurate.

We also need to use the cpuid fence instruction to make sure no previous instructions are executed during the clock cycle measurement, and also to make sure no instructions from the experiment execute after the clock has been stopped. So, these cpuid instructions have been inserted just before starting the timer and just after stopping the timer. The final code looks like this:

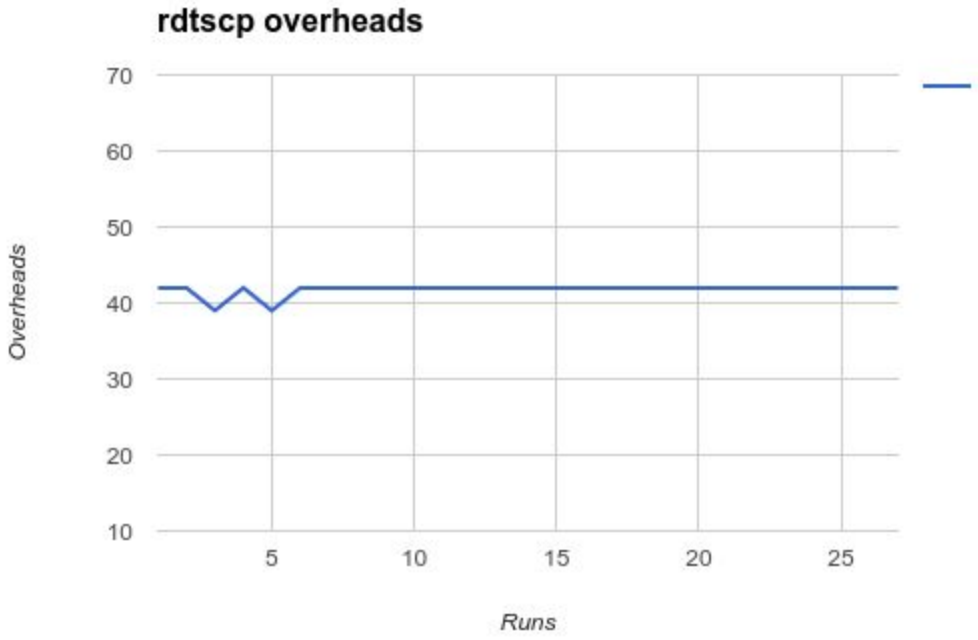
```
// Start timer.
asm volatile(
            fence instruction
            rdtsc
            copy the registers to variables
      )

// Experiment operation.

// Stop timer.
asm volatile(
            rdtscp
            copy the registers to variables
            fence instruction
      )
```

The following table and graph shows the measurements in clock cycles and time taken by the measurement mechanism itself.

| For rdtscp: (30 readings) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 42 | 42 | 39 | 42 | 39 | 42 | 42 | 42 | 42 |
| 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 |
| 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 |
| Variance | 0.641025641 | | Median | 42 | | Std. Deviation | 0.800640769 | |




The standard deviation for the measurements using rdtscp and rdtsc instructions is 0.8. That means it is very reliable technique to measure the program run times.

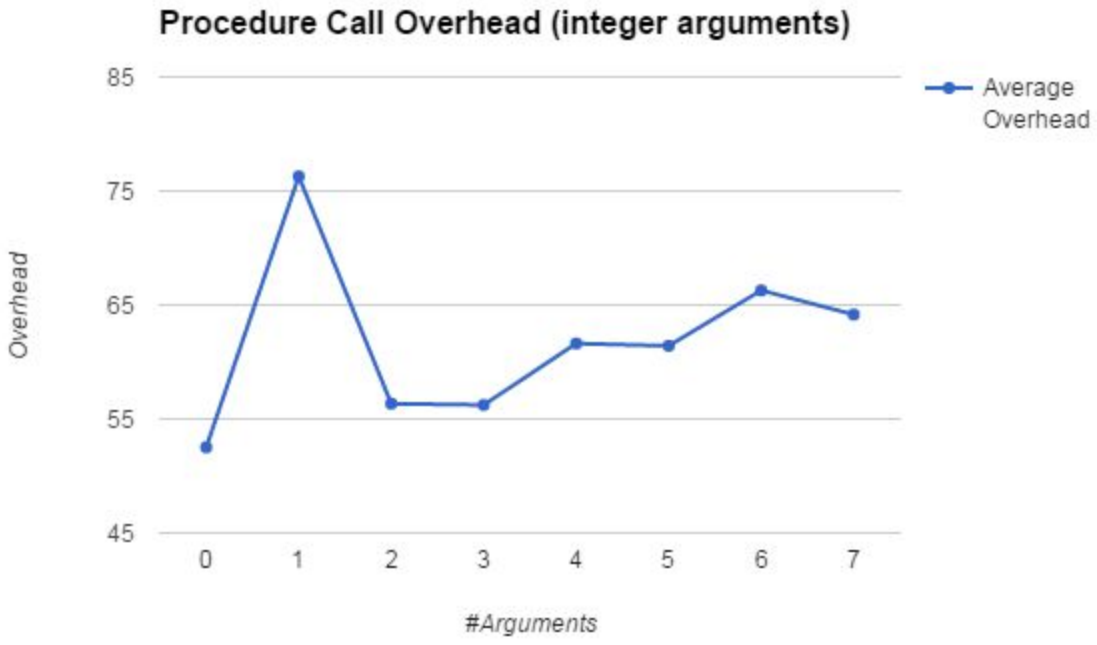
## 2. Procedure call overhead:

For this experiment, we varied the number of arguments for a function from 0 to 7 to have a look at the time required to call this function. The function itself did nothing except for managing the stack on a function call.

We explored the objdump for all the functions written and concluded that the number of arguments did not matter if they were single, small elements, like integers. The object dump for the functions (which had integers as arguments) revealed that only a single move instruction is needed to get the data from the stack into a variable/register. That implied even if the number of arguments were increased, the move instructions needed to get the argument into the register/variable could have been executed out of order and hence, there was no significant increase in time for simple arguments as integers.

The table below shows the time taken by calls to different functions with increasing number of arguments. There is no significant improvement in the time taken.

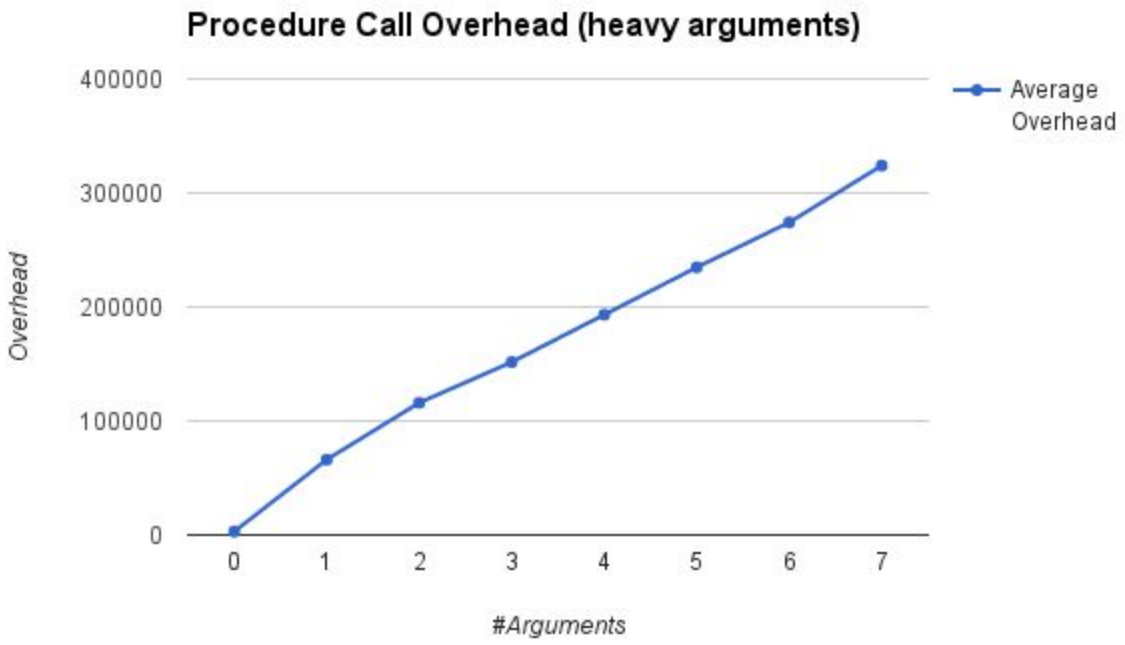
| Runs ↓ | #Arguments | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Average | 52.520204 | 76.280612 | 56.34448 | 56.231326 | 61.621632 | 61.415102 | 66.279795 | 64.157857 |




As the above graph shows, there is no effect on the number of arguments, just a small increase. That too is very insignificant.

To explore further as to why this was the case, we changed the argument from integer to heavy structures. That implied more data had to be pushed on to the stack. When we changed the argument type from integers to heavy structures, the trend became visible. The table and graph below show the trend for the heavy argument.

| Runs ↓ | #Arguments | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Average | 3059.7195 | 66129.96 | 115985.88 | 151695.6 | 193240.03 | 234780.24 | 274198.3 | 324039.70 |




As we can see, as the number of arguments increase, the time taken by the function to begin execution increases. This is mainly because the number of push and pop instructions required by the function are increased, and hence even if the processor is super scalar, the out of order execution will not be of much help.

## 3. Syscall overhead

For most of the part, a syscall is handled like an interrupt. Rather, a syscall is treated as a software interrupt. There are a lot of routines/procedures that the operating system has to go through to process the syscall.

So, when we wrote the program, we estimated that the time needed to process a syscall was going to be high. Definitely higher than the procedure overhead. When a syscall is executed, it has to first go to the lookup table, then call the system call handler, move the control to the handler, service the call, return the control to the program and then resume execution.

We had a look at 3 different system calls getpid(), gettimeofday() and pipe(). Although it was mentioned that the getpid() will cache the result of the last call, we did not run the program using a for loop. This means that every time we had to take a reading, we started the program (created a process), invoked getpid() and stopped the program (destroyed the process). That means even if the result of getpid() is cached, our process is not the same so that the results for each call to getpid() will be different.

| getpid() : 30 readings | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 6420 | 6174 | 6186 | 5847 | 6156 | 6240 | 5778 | 6129 | 6039 | 5883 |
| 5973 | 5919 | 5916 | 5940 | 5955 | 5880 | 5808 | 6279 | 6084 | 6084 |
| 6168 | 5934 | 6168 | 6243 | 6111 | 6426 | 6054 | 6192 | 5913 | 6081 |
| Variance | 28545.51724 | | Median | 6082.5 | | Std. Deviation | 168.9541868 | | |

As we will see, getpid() was one of the low overhead and more consistent system calls. The standard deviation is around 168 cycles. That is less than 70ns.

The next system call we tested was the gettimeofday(). This system call was assured to give different result everytime. Below are the results for gettimeofday() overhead.

| For gettimeofday(): | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3825 | 2760 | 3420 | 3678 | 3825 | 2826 | 3651 | 3975 | 2526 |
| 3255 | 3576 | 3078 | 3429 | 2718 | 4257 | 3453 | 3807 | 2718 |
| 3396 | 4542 | 3498 | 3447 | 3795 | 4158 | 2421 | 3699 | 2817 |
| Variance | 297641.2564 | | Median | 3453 | | Std. Deviation | 545.5650799 | |

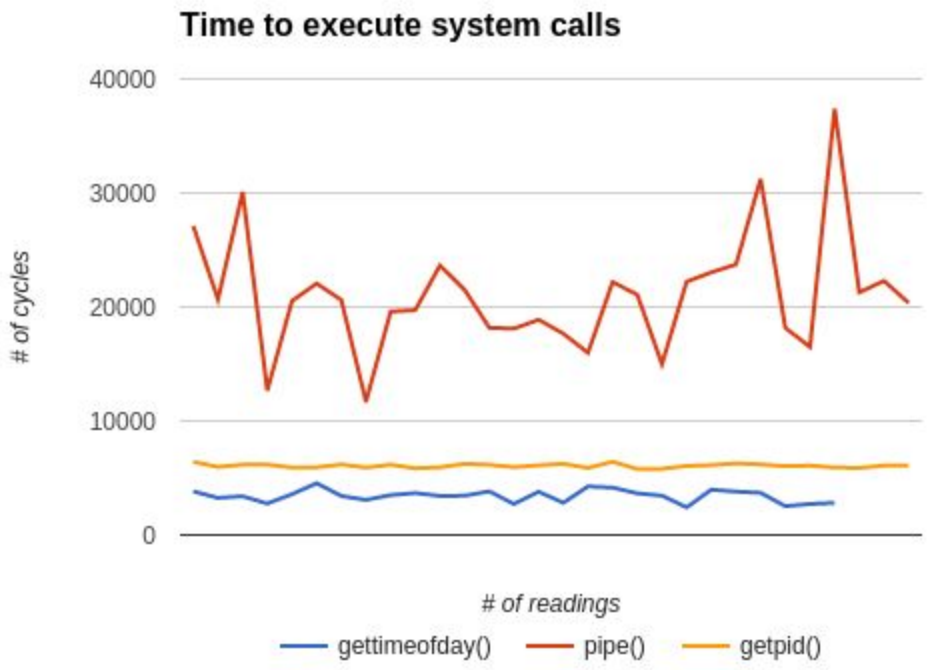
This system call appears to be low overhead but a little less consistent as compared to the getpid() system call. And the last system call we checked was pipe(). The reason we checked the overhead for pipe() was because we are going to experiment with the task creation and context switch times next. And we are going to use pipes to help us count the time taken for

context switch. The table below shows the average overhead to create a pipe that will be used to communicate between two processes.

| Pipe() | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 27087 | 12669 | 20610 | 19728 | 18177 | 17664 | 21069 | 23037 | 18168 | 21276 |
| 20676 | 20529 | 11682 | 23652 | 18102 | 15975 | 14985 | 23727 | 16494 | 22293 |
| 30096 | 22065 | 19596 | 21501 | 18882 | 22194 | 22218 | 31233 | 37395 | 20355 |
| Variance | 27589315.71 | | Median | 20643 | | Std. Deviation | 5252.5536 | | |

As we can see, the standard deviation is very high for creating the pipe. This implies that to create a pipe, there are a lot more dependencies as compared to getpid(), that have to be taken care of by the operating system. We had guessed this system call should take more time than getpid() or gettimeofday() since both of these functions do not have to worry about inter process communication and standard input and output. These two functions simply have to read a value from some register and return it. pipe() on the other hand has to link the make sure that the int array passed as an argument can be used as an input output medium.

Above results can be plotted as follows:



# 4. Task creation overhead

The task creation overhead is measured for two types of tasks.

### a. Process:

We spawn a process by using the fork() system call. The measurement is done by placing the fork() system call in between the start and stop timer. One small complication this adds is when fork() is placed in between the start and stop timer, the stop timer will be executed by both the child process and the parent process.

The fork system call returns either 0 (for child process) or the pid (of the child process). This helps us distinguish between the parent and child process. To make sure only the parent stops the timer, we have to place an if condition between the fork() and stop timer. We had thought that this “if” condition will incur some cycle overhead as we are adding a branch instruction. But as it turns out, the fork() system call itself has heavy overhead. This turns out to be good to measure the overhead for fork().

The simple “if” condition does not even matter as it is only one branch instruction. The table below shows the timings for invoking a process. The readings below are only for 10 runs, all the other readings are mentioned in the excel sheet attached.

| Runs | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| fork() | 767826 | 493323 | 395682 | 352143 | 370338 | 375630 | 418320 | 376161 | 616236 | 417792 |

We took 100 readings and averaged it. The average cycles taken by the fork system call is around **481633.89 cycles**. That is nearly 0.2 milliseconds. We are also guessing that if we had associated more data with the process before invoking the fork() system call, it would have taken even more cycles. This is due to the fact that fork basically created a process identical to the calling process (just before the fork() call). That will include all the variables, registers, function call stacks if fork() has been called from any function other than main(), and so on.

### b. Thread

To invoke a thread from a process, we used the pthreads library. This library has numerous methods that allows us to create, manage and destroy threads. Similar to fork(), we placed the pthread_create() call between the start and stop timer. We were assuming the pthread_create() call would be lighter as compared to fork since we are not launching a whole process but a simple smaller thread. And we were correct. Since the threads are lighter, the method calls to create them are much smaller. And hence take less time.

The table below lists the time taken by the pthread_create() call.

| Runs | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| pthread | 219348 | 124686 | 134127 | 128829 | 130149 | 132852 | 140382 | 145218 | 131655 | 128133 |

Just like fork(), we took the readings for a 100 runs and averaged it. The average time taken by the pthread_create() call is around **140943.54 cycles**. Which is almost 58 microseconds. We were hoping to get even smaller number but, alas, to our dismay, it won't give us what we want. We tried to investigate further as to why the pthread_create() would take so long, or if there are any possible ways to reduce the time taken. We could not find a reliable solution.

## 5. Context switch overhead:

We referred to the paper mentioned in the project spec sheet and came up with our own implementation of the context switch time. Although the idea is the same, we are guessing the accuracy of our program will be a little more as compared to the process described in the paper.

In the paper, they have mentioned to measure the context switch time, you have to repeatedly pass a byte of data to and from parent and the child using pipes. And then one round trip for a byte of data will be equivalent to two reads, two writes and two context switches.

What we are assuming is wrong with this method that we are neglecting the small overheads between the read and writes. What we have done instead is we have passed the timing information through the pipes and stopped the timers exactly after read() operation. What this helps us to do is measure the time taken for one context switch exactly. Our method has the formula :

$$measured\ time\ =\ 2 * context\ switch\ time\ +\ read\ time\ +\ write\ time$$

Whereas the paper uses the formula:

$$measured\ time\ =\ 2(context\ switch\ time\ +\ read\ time\ +\ write\ time)$$

We have passed the byte to and from the child and parent repeatedly just as mentioned in the paper. We had to measure the read time separately and the write time separately. To do that we simply changed the position of the start timer and stop timers. This let us measure accurately the time taken by the read and the write overheads.

The following are the results for the combined measured time, read time and the write time. Only the average has been mentioned below. The complete table can be found on the attached excel file.

| | Combined | Read | Write |
|---|---|---|---|
| Average | 82930.71 | 9887.1 | 12318.9 |

Putting this in the above formula,

$$2 * context\ switch\ time\ = 82930.71\ -\ 9887.1\ -\ 12318.9$$
$$2 * context\ switch\ time\ =\ 60724.71$$

$$context\ switch\ time\ =\ 30362.355\ cycles$$
$$context\ switch\ time\ = 12.65\ microseconds$$

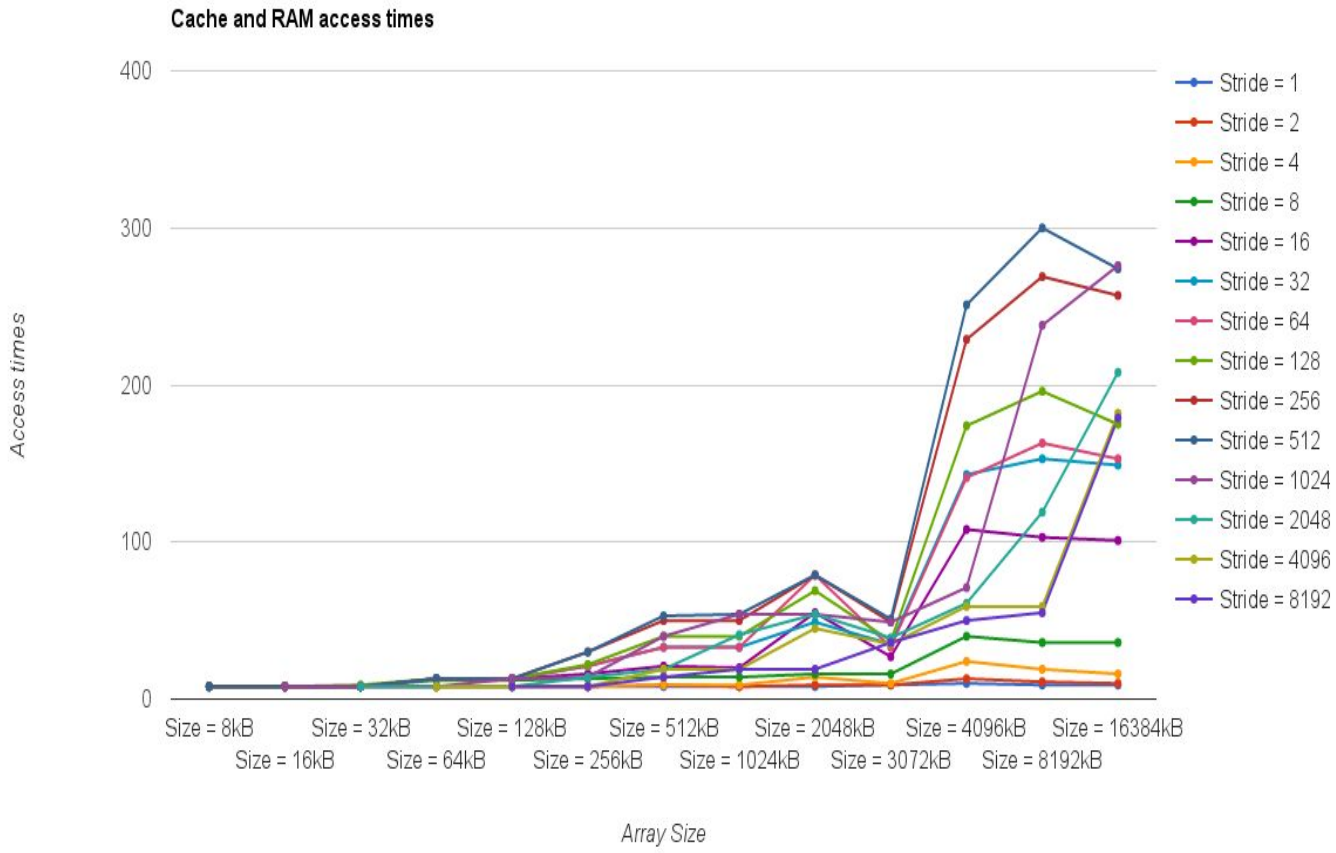
## 6. RAM Access Time

To measure the RAM access time, we referred the lmbench paper. It is a very well written paper with extremely precise methods to measure the RAM/Memory access times. The methods described in the paper appropriately reveal the sizes of all the structures.

Using the technique described in the paper, we implemented the experiment and got the actual results. The results showed us that the L1 cache size of the processor that we ran the codes on was 32KB, L2 size was 256 KB, and L3 size was 3MB. The sizes appropriately showed the time required to access each of the structure.

Below is the table and the graph used to deduce all the sizes and the values.

| | Stride = 1 | Stride = 2 | Stride = 4 | Stride = 8 | Stride = 16 | Stride = 32 | Stride = 64 | Stride = 128 | Stride = 256 | Stride = 512 | Stride = 1024 | Stride = 2048 | Stride = 4096 | Stride = 8192 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Size = 8kB | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | | | | |
| Size = 16kB | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | | | |
| Size = 32kB | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 9 | 8 | 8 | 8 | 8 | | |
| Size = 64kB | 8 | 8 | 8 | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 8 | 8 | 8 | |
| Size = 128kB | 8 | 8 | 8 | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 8 | 8 | 8 |
| Size = 256kB | 8 | 8 | 8 | 13 | 16 | 21 | 21 | 22 | 30 | 30 | 14 | 14 | 8 | 8 |
| Size = 512kB | 8 | 9 | 9 | 14 | 21 | 33 | 33 | 40 | 50 | 53 | 40 | 19 | 19 | 14 |
| Size = 1024kB | 8 | 8 | 9 | 14 | 20 | 33 | 33 | 40 | 50 | 54 | 54 | 41 | 19 | 19 |
| Size = 2048kB | 8 | 9 | 14 | 16 | 55 | 49 | 79 | 69 | 79 | 79 | 54 | 54 | 45 | 19 |
| Size = 4096kB | 10 | 13 | 24 | 40 | 108 | 143 | 141 | 174 | 229 | 251 | 71 | 61 | 59 | 50 |
| Size = 8192kB | 9 | 11 | 19 | 36 | 103 | 153 | 163 | 196 | 269 | 300 | 238 | 119 | 59 | 55 |
| Size = 16384kB | 9 | 10 | 16 | 36 | 101 | 149 | 153 | 175 | 257 | 274 | 276 | 208 | 182 | 179 |




As we can see from the graph, as we increase the array size, the access times are constant up to array size =32 KB. This means that after size = 32KB we are having certain misses and hence the access time is increasing. This shows that out **L1 cache is 32 KB** and it's access time is about 8 cycles. We have to consider other factors including cache compulsory misses, software overheads for measurements and other OS overheads if any.

Similarly, after 32KB size, the access times are constant up to a certain array size (256 KB). The access times start increasing further after this size. That implies the **L2 size is 256 KB** and the access time for this cache is around 16-20 cycles.

And on the same lines, we deduce that the **L3 cache size is 3MB**. After 3MB the access time increases significantly with increase in the array size. From the table, the L3 access time is around 108-120 cycles.

And finally, we see that the graph is settling after a certain array size towards the end. That part implies that even if the array size is increased further, the access times will not vary a lot. That is the RAM access latency which comes out to be more than 200 cycles.

## 7. RAM Bandwidth

To measure the RAM bandwidth, we wanted we used *memcpy()* and *memset()* function calls from C language. *memset()* writes/initialises data to the memory location. We have to provide same value which we want to write into multiple memory locations, size of the memory to be initialised and pointer to that location. So using *memset()* is equivalent to writing into memory.

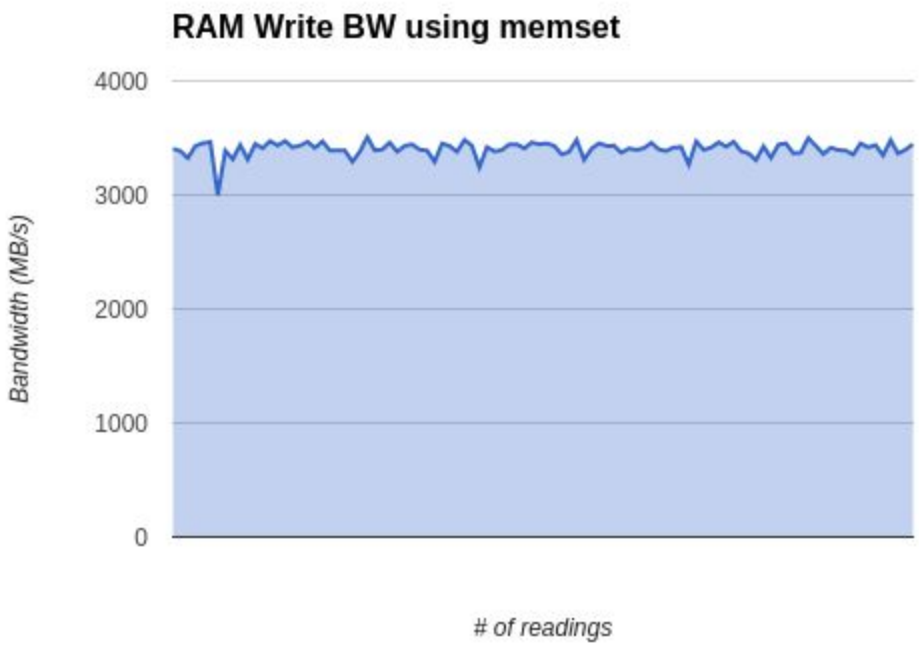
However, as we know any data that we write goes to cache first and then goes to memory. So writing with memset is writing into cache. But in our case, all the caches are write through. So any data that is written to cache is directly goes to main memory. So to calculate write bandwidth, we wrote 10MB of data into memory and calculated the time required for the memset operation.

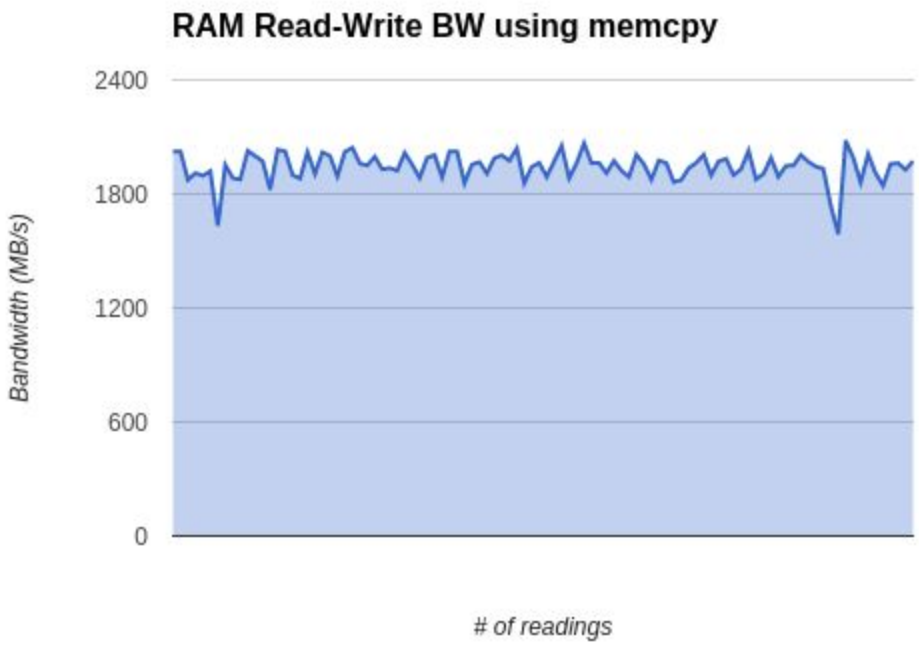
Also to calculate Read-write combine bandwidth, we used *memcpy()* function which reads data from source array location and write it to destination location.In this case also we measured time required to copy array of size 10MB from one location to other.

Average values for Bandwidth in terms of Write and Read-Write is as follow:

| | For Write | For Read-Write |
|---|---|---|
| # of cycles required to for 10MB data | 7052507.4 | 12369326.01 |
| BW (MB/s) | 3404.359834 | 1943.533165 |

Following graphs shows the experimental results collected over 100 runs for both *memset()* and *memcpy()* functions.






In the above table, the bandwidth for read write is combined for both the operations. We wrote 10MB data and also read it, so the time measured should be half for individual operations. Since it is not possible at run time to distinguish between these operations, we have mentioned the combined values.

What we can do instead, is take a statistical average of the write operation over large group of runs, and also for read-write, and then subtract the write average from the read-write average.

## 8. Page fault service time

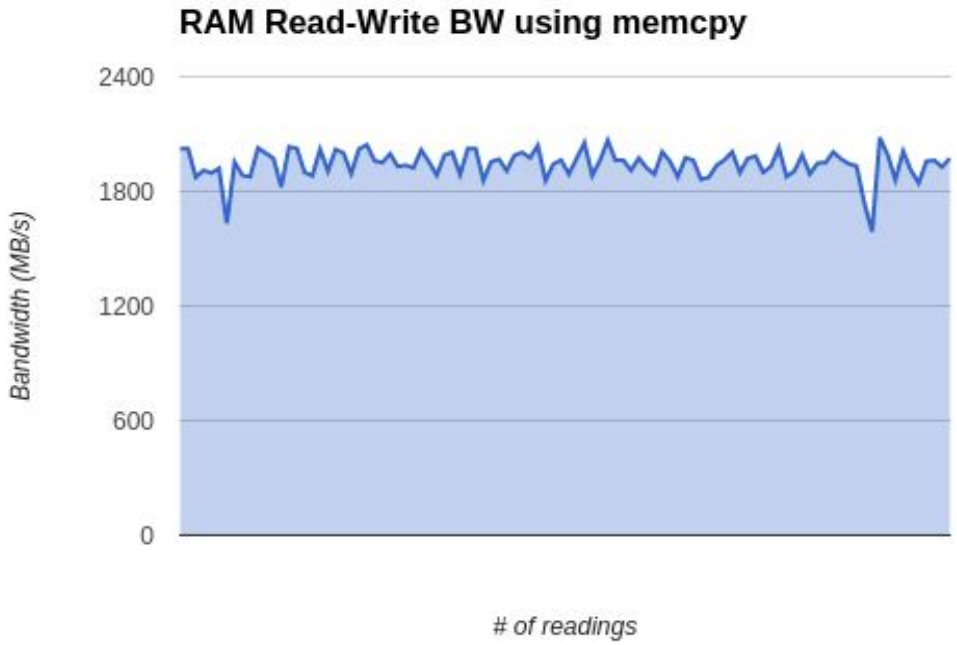
To measure page faults service time, main challenge was to make a page fault using a program and when it happens measure the time system takes to recover from the page fault. To make page fault to happen, we took the help from mmap (thanks professor for pointing it out.)

First we mapped a large .txt files (around 12MB) using a mmap system call. The purpose of choosing this large text file was to make sure that this will get mapped to the pages which are already not in the memory. We think the size of the memory that we are mapping using mmap has relation with where it is mapped. This is because if size of memory is just enough to fill the all the pages available in the RAM then it will not give the page fault and time we measured will be only the main memory access time.

For example, if mmap requires only 4 pages to map the new file and there are already unassigned pages in that process's page table, then OS will assign memory asked by mmap to these pages which are already in the RAM. So large file size when mapped using mmap ensures that OS maps those pages to the disk (swap).

So to measure page fault service, we mapped large file using mmap and accessed its 10 locations which are far from each other(1MB away in our experiments). So all those 10 accesses should give page faults. And when we ran this particular benchmark with Linux's */usr/bin/time* utility, it proved our assumptions and we were able to see 10 page faults for all those 10 memory accesses.

Following graph and the table on the next page shows the time required to service 100 page faults caused by our benchmark.



| | Minimum | Maximum | Average |
|---|---|---|---|
| No of Cycles | 218385 | 740832 | 261296.91 |
| Time(micro-secs) | 90.99375 | 308.68 | 108.8737125 |

Console output of the program is as follow:

$/usr/bin/time -v ./minibench
Page fault service time - 319104
Page fault service time - 265395
Page fault service time - 247542
Page fault service time - 249750
Page fault service time - 252054
Page fault service time - 255879
Page fault service time - 257094
Page fault service time - 251994
Page fault service time - 255273
Page fault service time - 252948
Command exited with non-zero status 155
	Command being timed: "./minibench ../crap/pop.txt"
	User time (seconds): 0.00
	System time (seconds): 0.00
	Percent of CPU this job got: 0%
	Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
	Average shared text size (kbytes): 0
	Average unshared data size (kbytes): 0
	Average stack size (kbytes): 0
	Average total size (kbytes): 0
	Maximum resident set size (kbytes): 1352
	Average resident set size (kbytes): 0
	**Major (requiring I/O) page faults: 10**
	Minor (reclaiming a frame) page faults: 66
	Voluntary context switches: 37
	Involuntary context switches: 1
	Swaps: 0
	File system inputs: 2560
	File system outputs: 0
	Socket messages sent: 0
	Socket messages received: 0
	Signals delivered: 0
	Page size (bytes): 4096
	Exit status: 155
This ensures that our program is causing page faults.

## Conclusion:

| Parameter | Base Hardware performance | Guess of Software overhead | Prediction | Actual |
|---|---|---|---|---|
| Measurement overhead | 4 cycles (4 mov instructions) | 10 cycles | 14 cycles | 42 |
| Procedure call overhead | 8 cycles | #depends on number of args. ~ 30-40 cycles | 40-50 cycles | ~60-80 cycles Also, type of argument matter |
| Task creation time | >10000 cycles, details mentioned above. Needs all context copy, system call handler etc | | | >350000 |
| Context switch overhead | >2000 cycles, details above, needs context save, control transfer, context load etc | | | ~ 30000 |
| RAM access time | 200-400 | None expected, as measured from hardware perspective, i.e. none of the OS routines would be involved. | 200-400 | ~230 |
| L1 access time | 2-4 cycles | | 2-4 cycles | 8 |
| L2 Access time | 10-15 cycles | | 10-15 cycles | ~17 |
| L3 Access time | 100-120 cycles | | 100-120 cycles | ~108 |
| RAM Bandwidth | 8.536 GBps | This should drop way below the maximum throughput as OS has to continuously update certain structures, guessing drop of 2-3GBps | 4-5GBps | ~3GBps |
| Page fault service time | 125000, assuming 0.3 ms disk access time. | >30000 for TLB lookup, servicing interrupts, and initiating swap | ~145000 | ~300000 |