CSC469 ASSIGNMENT 1

Shaun Memon and Binuri Walpitagamage

February 11th 2016

1. PART A1: TRACKING PROCESS ACTIVITY

1.1 Introduction and Hardware specifications

As learned in lecture, the runtime of a user process includes user time, system time as well as time taken by other processes in the system. The purpose of this experiment is to identify the periods in which a process is active and inactive. With a reliable clock speed calculation and a threshold, we hypothesize that the activity pattern should be able to identify the timer interrupts as well as its approximate duration.

The experiments were conducted on CDF Machines with the following properties.

CPU	Pentium(R) Dual-Core CPU E6300
CPU Speed	2.80GHz
Memory Size	8GB
Cache size	2048 KB
Kernel	Linux version 3.13.0-77-generic

1.2 Clock speed calculation

Our methodology for calculating CPU speed was to calculate how many cpu cycles occurred over a period of 0.25seconds. This was done by retrieving the current CPU cycle counter, sleeping using nanosleep(), and then retrieving the CPU cycle counter immediately after. The scaled difference gives us an upper bound on the number of clock cycles / second, or Hz. This is an upper bound because nanosleep() only gives us a minimum time that the process will sleep for barring any signal interrupts, and there will always be a little overhead in context switching and calling the sleep function itself. To try and mitigate the differences as much as possible, we ran 10 tests on the CPU cycle counter method described above and averaged them in order to try and mitigate error the best we could.

1.3 Threshold Calculation

The main strategy for determining a sufficient threshold was to detect outliers by running the inactive period tracker with increasing thresholds. The initial run with a threshold of 0 was done in order to get an idea of the duration of two consecutive timer reads in our code. An active period between two timer reads took around 300 - 400 cycles. The second test run was with a threshold of 400 which indicated some outliers present in the 700 - 900 cycle range as well as 1500+ cycle range. Assuming that the smaller range of outliers were due to cache or TLB misses, we estimated the threshold to be between the two outlier regions and used the value 1100 for both experiments in part A.

1.4 Method

tracker <num_samples>
Given a number of samples to collect, the '*tracker*' executable outputs a said number of active and inactive periods for a process. Once the cycle counter is initialized, the program continuously reads the cycle counter and takes the difference between two consecutive reads. If this calculated difference is larger than the predetermined threshold, then that duration is considered to be inactive and is recorded as an inactive sample. The number of samples gathered is decided by the user. Furthermore, a simple loop is run in between two consecutive reads in order to overcome the cycle counter read overhead.

run_experiment_A [<num_samples>]
The *'run_experiment_A'* script executes the *tracker* program and parses the output to create the plot given in section 1.5. It accepts the number of samples as an optional argument that is passed on to the *tracker* program. If no value is provided, then the default samples created is 10. Since the active period of the first sample (sample 0) starts before the measurements are initialized, that sample is omitted from the displayed plot.

1.5 Results

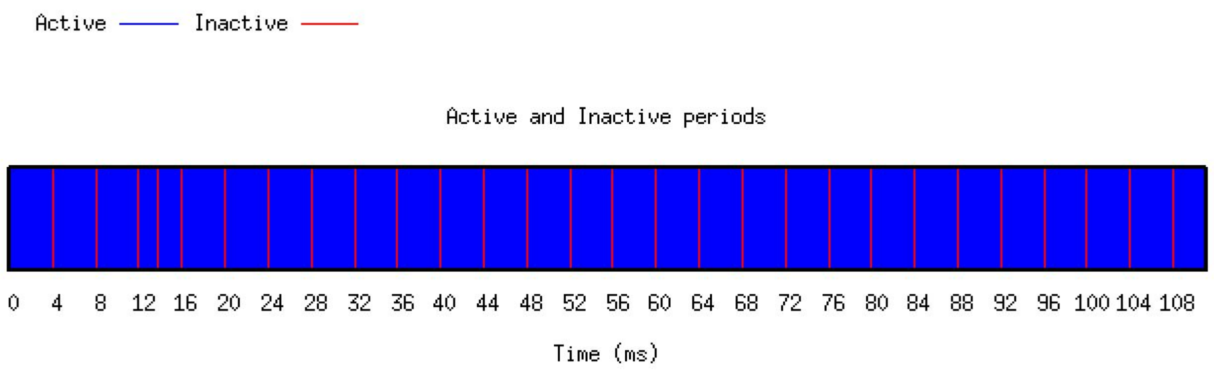

Figure 1.5a The graph above was created by running *run_tracker* script with a sample size of 30

Sample	Active period duration (ms)	Inactive period duration (ms)
0	1.438272 ms	0.041725 ms
1	3.957911 ms	0.005622 ms
2	3.993879 ms	0.004850 ms
3	3.994981 ms	0.030670 ms
4	1.650466 ms	0.008348 ms
5	2.310398 ms	0.001124 ms
6	3.999931 ms	0.000971 ms
7	3.998815 ms	0.003178 ms
8	3.995335 ms	0.001048 ms
9	3.998723 ms	0.000987 ms
10	3.998741 ms	0.001898 ms
11	3.999089 ms	0.000872 ms
12	3.999036 ms	0.000897 ms
13	3.998994 ms	0.001717 ms
14	3.998163 ms	0.000874 ms
15	3.997620 ms	0.000991 ms
16	3.998747 ms	0.001879 ms
17	3.997874 ms	0.000957 ms
18	3.998678 ms	0.000939 ms
19	3.998675 ms	0.001815 ms
20	3.997771 ms	0.000993 ms
21	3.998611 ms	0.000924 ms
22	3.998728 ms	0.001807 ms
23	3.997786 ms	0.000950 ms
24	3.998638 ms	0.000960 ms
25	3.998567 ms	0.001806 ms
26	3.997768 ms	0.001000 ms
27	3.998516 ms	0.000939 ms
28	3.998508 ms	0.001820 ms
29	3.997629 ms	0.000972 ms

Figure 1.5b The table contains the active and inactive durations for the 30 samples illustrated in fig 1.4a

1.5 Discussion

From the results, it is apparent that timer interrupts occur in 4 millisecond intervals. The duration of timer interrupts seem to vary from 0.9 microseconds to as high as 30 microseconds. Non timer interrupts that does not fit the 4ms pattern is present in sample 3 and 4 of the table (12- 16 ms interval of the graph). By looking at the /proc/interrupts/ file before and after running the program, we believe that this inconsistency may be due to an event such as a rescheduling interrupt or a High Precision Event timer interrupt. Overall, around 0.1% of the process run time is lost due to servicing interrupts which we believe is an inexpensive cost.

In conclusion, by calculating the active and inactive periods of a process based on a threshold, we were able to identify the timer interrupt frequency and the interrupt duration of the operating system.

2. PART A2: CONTEXT SWITCHES

2.1 Introduction and Hardware specification

In operating systems, a context switch is the process of switching from one process to the other while preserving the state of the paused process so that it could be continued from the same point later on in the CPU schedule. Context switches enable a single CPU to execute multiple processes thereby enabling the operating system to multitask.This experiment was designed in order to determine the amount of time given to a single process before being switched to another process as well as the latency of the switch.

Note: The hardware specifications, clock speed calculation and threshold as mentioned in part 1 applies to this experiment as well.

2.2 Method

context_tracker [<num_samples>] [<num_child_processes]
In order to produce context switches, the *'context_tracker'* program creates a given number child processes, each collecting active and inactive periods based on the threshold. The parent process and the child processes are pinned to CPU 0 in order to get consistent context switches. Each child process prints out its own active and inactive samples similar to part A1 and once all child processes are terminated, the parent process exits.

run_context_tracker [<num_samples>] [<num_child_processes]
The *'run_context_tracker'* script parses the output produced by all child processes in the context_tracker program and calculates the context switch and time slice durations for the events. From the output, the start cycle counter and durations of all the active periods are extracted and ordered according to the start cycle counter. This ordering gives us an overview of all the processes in a single timeline.

Afterwards, the context switch cycle duration is determined by taking the difference between the end and beginning of two consecutive samples. If the consecutive samples belong to the same child process, then the inactive time in between those two samples is ignored so that only the context switch periods are considered. The time slice duration is calculated by taking the difference between the start and end counters of a sequence of consecutive samples belonging to same process. The script outputs average values of the context switch and time slice durations.

2.3 Results

Experiment 1

Num of child processes forked	: 2
Num samples gathered by each process	: 15
Clock Speed	: 1396598060
The average context switch time	: 0.02188 ms
The average time slice duration	: 11.17299 ms

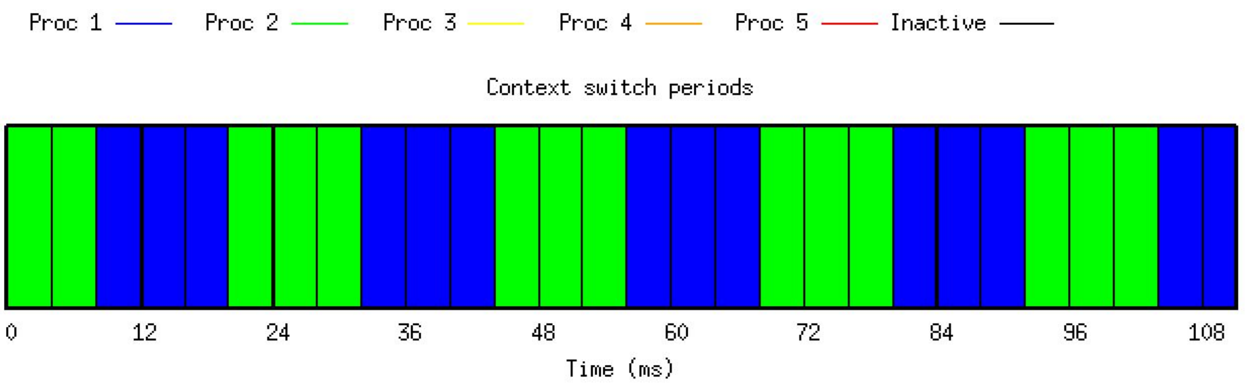

Figure 2.3a Plot above contains scheduling and interrupts for 2 child processes

Context Switch Event		Context Switch Duration	Runtime slice for Process A
From (Process A)	To (Process B)		
2	1	0.123379 ms	7.935217 ms
1	2	0.009041ms	11.876560 ms
2	1	0.010844 ms	11.989067 ms
1	2	0.008661 ms	11.988150 ms
2	1	0.008495 ms	11.990447 ms
1	2	0.008706 ms	11.990446 ms
2	1	0.010656 ms	11.990425 ms
1	2	0.008676 ms	11.988369 ms
2	1	0.008469 ms	11.990315 ms
1	-	-	6.912341 ms

Figure 2.3b The table contains the time slice and context switch durations for the schedule illustrated in fig 2.3a

Experiment 2

Num of child processes forked	: 5
Num samples gathered by each process	: 15
Clock Speed	: 1396605788
The average context switch time	: 0.02158 ms
The average time slice duration	: 3.90898 ms

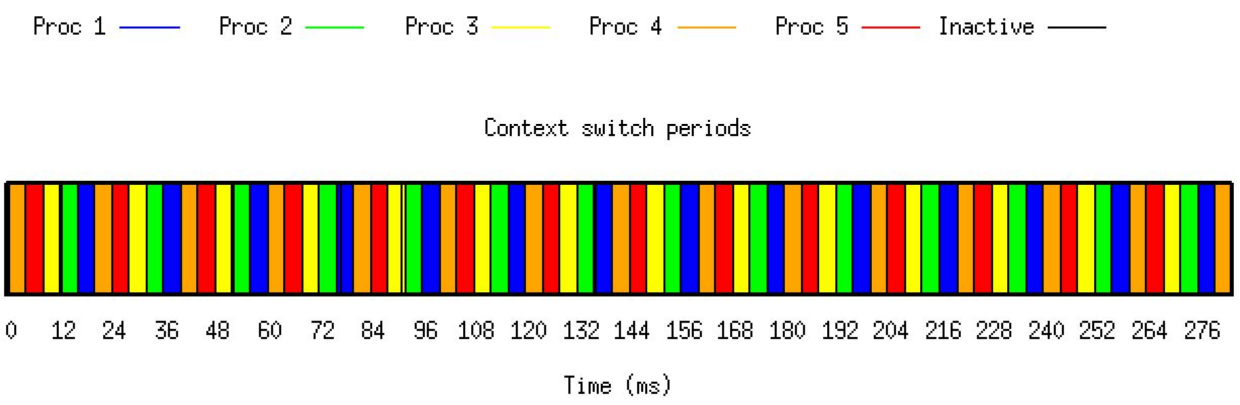

Figure 2.3d Scheduling for 5 child processes, each collecting 15 samples

Experiment 2

Num of child processes forked	: 3
Num samples gathered by each process	: 15
Clock Speed	: 1396591460
The average context switch time	: 0.01964 ms
The average time slice duration	: 5.78204 ms

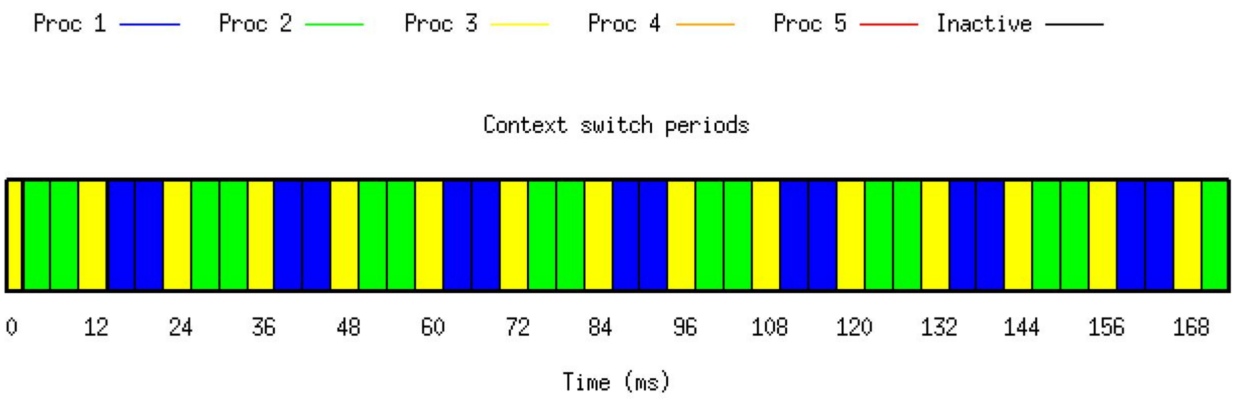

Figure 2.3c Scheduling for 3 child processes, each collecting 15 samples

2.4 Discussion

Comparing Experiment 1 and Experiment 2, we see that the time slice duration is affected by the number of processes scheduled on the system. With the presence of only 2 processes (Experiment 1), each process was given a time slice of approximately 11 ms. However when the workload was increased to 5 processes(Experiment 2), the time slice was decreased to 4ms. In Experiment 2, a context switch occurred on each timer interrupt. meaning that a context switch occured at every timer interrupt.

Moreover, the scheduling pattern also seems to be affected by the scheduled number of processes. In Experiment 1 and 2, the processes were executed in a round robin pattern and were given equal time slices. However in Experiment 3 which consisted of 3 scheduled process, 2 processes were given a time slice of approximately 8ms but the other process only got a time slice of 4 ms. The scheduling pattern was also different compared to the other two experiments.

Observations and relationships of Experiment 1 and 2 supports our understanding gained from previous OS courses. The results of scheduling patterns from Experiment 3 does not contradict our understanding but is simply a new addition. An observation that did not fit our understanding of processor scheduling is the first scheduled process in the experiments. For example, in the *context_tracker* program, Process 1 is created before Process 2, so the expected behaviour was that Process 1 would be in the ready queue before Process 2. However in Experiment 3 we see that the order of initial execution was Process 4, Process 5, Process 3, Process 2, and Process 1. The first process created was infact the last to be scheduled in even when the work performed by both processes is the same.

3. PART B : NUMA ARCHITECTURE

3.1 Introduction

The purpose of this experiment is to measure memory bandwidth performance on a machine with a NUMA architecture. The machine we are using is wolf.cdf which is an AMD Magny-Cours consisting of four nodes. Each node contains 12 cpus with ~16GB of RAM. We are expecting access times between a cpu on a given node and memory on a different node to be 1.6x slower than that cpu accessing memory on it's own node. This is taken directly from the numactl –hardware command which gives us the expected distance between a cpu and memory on a node. The output of numactl –hardware at experiment runtime is given below as a reference of the conditions when the experiment was run.

Numactl was used to run John McCalpine's Stream benchmarking tool on a series of CPU and NUMA node combinations. Stream measures bandwidth rates using four “kernels”, copy, add, scale, and triad. Each test is run on arrays four times the size of the L3 cache and ten times each reporting the best result that was achieved.

```
wolf:$ numactl --hardware
available: 4 nodes (0,2,4,6)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11
node 0 size: 16046 MB
node 0 free: 6352 MB
node 2 cpus: 12 13 14 15 16 17 18 19 20 21 22 23
node 2 size: 16126 MB
node 2 free: 12338 MB
node 4 cpus: 24 25 26 27 28 29 30 31 32 33 34 35
node 4 size: 16126 MB
node 4 free: 13966 MB
node 6 cpus: 36 37 38 39 40 41 42 43 44 45 46 47
node 6 size: 16110 MB
node 6 free: 5363 MB
node distances:
node   0   2   4   6
  0:  10  16  16  16
  2:  16  10  16  16
  4:  16  16  10  16
  6:  16  16  16  10
```

Figure 1: numactl –hardware at experiment runtime

3.2 Methodology

Our initial thought was to just use the copy kernel to determine the results of our experiment. Since we are just trying to get the maximum memory bandwidth between CPUs and NUMA nodes, we thought this would give us the highest rate since the CPU is doing minimal work in just accessing and copying data from memory. However, the route we did go with was to take an average over the results of each kernel. The reason we went this route is because typically memory is not accessed merely for the sake of copying it. Some work or calculation is generally performed by the CPU on the data before it is written back to memory, so because of this reasoning we chose to take an average of the four kernels for each cpu and each NUMA node.

Another factor to consider was in the choosing of CPUs and memory nodes to measure the performance in. Given that wolf.cdf is a shared server and is in use virtually all the time with varied loads, was was decided was that aggregating the bandwidth accesses for each CPU in a node to memory in every node was a good way to try and spread out the tests over CPU's which may have various loads placed on them. By measuring the bandwidth of every CPU in a node and averaging them, we hoped to level out CPUs that had heavy usage which CPUs that were under a lighter or no load. Thus in the end as an example, every CPU in node 0 was benchmarked against each of the 4 nodes, and those were then averaged and taken as the maximum bandwidth rate for access from node 0 to each of the NUMA nodes. In this way, we can recreate the node distance matrix as given in figure 1 and compare our results with those that are expected.

3.3 Results

Our tests were run in the middle of the day on a fairly loaded server. This is going to have obvious performance implications for these tests. The rests were run with a python script that runs the test for each NUMA node across all CPUs and produces an aggregated result matrix at the end. The tests take quite some time to complete on a loaded server since they are benchmarking every CPU and NUMA node combination. The results of our run are given below (formatting slightly adjusted). The top table in figure 2 above is the actual throughput that was recorded, underneath it is the ratio.

```
Memory bandwidths are (in MB/s):
CPU Node       0:            2:             4:            6:
Mem Node 0:    5400.69166    5387.51041     5382.5375     5474.03333
Mem Node 2:    3903.33125    3897.19583     3871.25625    3931.1375
Mem Node 4:    2488.09583    2487.39791     2489.65416    2494.59791
Mem Node 6:    2387.33541    2386.0         2388.10833    2386.14375

Memory bandwidths ratios:
CPU Node       0:         2:          4:          6:
Mem Node 0:    1.0        1.00244     1.00337     0.98660
Mem Node 2:    0.99842    1.0         1.00670     0.99136
Mem Node 4:    1.00062    1.00090     1.0         0.99801
Mem Node 6:    0.99950    1.00006     0.99917     1.0
```

Figure 2: Bandwidth Results Matrix

These results are quite surprising and do not at all match with the expectation. We would expect memory accesses across nodes to be 1.6x slower than memory accesses within nodes. The data here shows that memory accesses, regardless of location, are fairly consistent in their bandwidth. The data shows that, while in theory, NUMA architectures may be slow to access memory in different nodes, in practise this may not actually be the case. Thus, it is our conclusion that performance worries in regards to memory access times on a system like wolf.cdf in practical usage may be disregarded.

4. APPENDIX

4.1 Part A1

I. ***tracker <num_samples>* :** Given an integer value as num_samples, the program outputs *num_samples* number of active and inactive samples in the following format

```
Active 0: start at 38, duration 627514 cycles (0.523649 ms)
Inactive 0: start at 627552, duration 11444 cycles (0.009550 ms)
```

II. ***run_experiment_A [<num_samples>]*:** Given an optional integer value as num_samples, the script outputs *num_samples* number of active and inactive period samples (similar format as 4.1-I) and creates a plot indicating the time durations of the periods in a single timeline from start to the end of the process. If the integer argument is not provided, a default value of 10 samples is created.

4.2 Part A2

I. ***context_tracker <num_samples> <num_procs>* :** Outputs *num_samples* number of active and inactive periods (similar format as 4.1-I) for a number of *num_procs* processes.

II. ***context_tracker [<num_samples>] [<num_procs>]]* :** Runs the program in 4.2-I and creates a plot indicating the time durations of the periods in a single timeline from start to the end of all the processes combined. Both arguments are optional integer arguments and will be given default values of 5 if not provided. Due to plotting limitations, the max number of child processes that can be created is 5.

4.3 Part B

I. ***run_b1.py* :** Script that runs mccalpin-stream from /u/csc469h/winter/pub/assgn1/bin/ on wolf.cdf. Stream is run through numactl given every combination of CPU and NUMA node. The results from each test are reported straight from the mccalpin-stream output and at the end the fully analyzed results are displayed as a matrix.

II. ***partb_results1.txt*:** The results from run_b1.py. Results of the experiment will vary every time so this keeps a record for the results that were analyzed during this run of the experiment.