CSC469 A1

Connor Yoshimoto, 1000685378

October 2016

1 Part A - Process Activity

Introduction:

The purpose of this experiment was to measure the performance of several low level operations on a Linux environment. More specifically, the first part of the experiment was to generate a trace of the active and inactive periods of a process and from this data derive conclusions about the interrupts that stop the execution of said process. The purpose of this experiment is to better understand the workings of the timer and other interrupts.

Methodology:

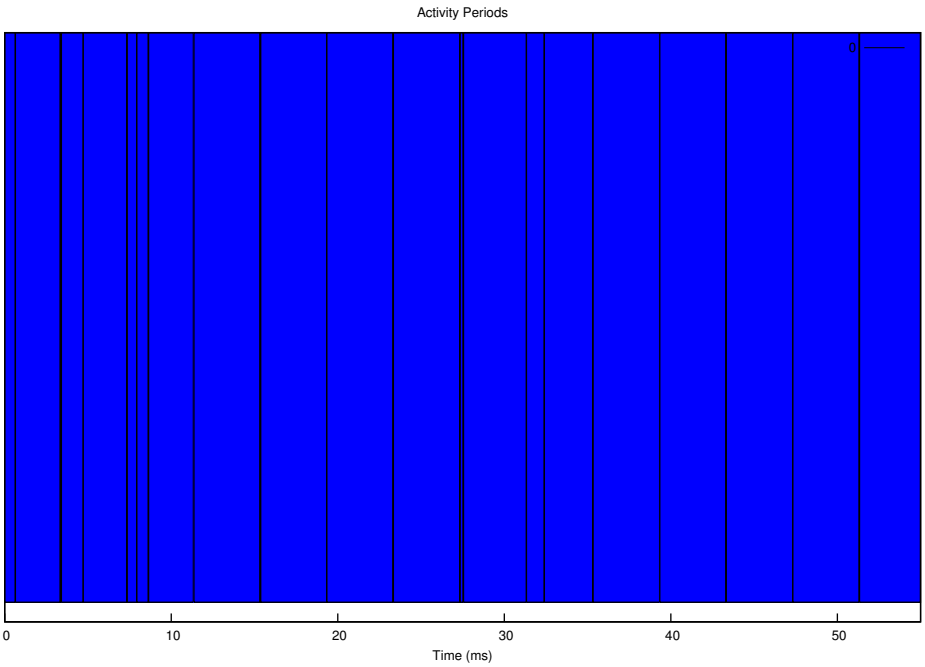
The measurement program and be roughly divided into two parts. The first part of the program was used to determine the clock speed of the system that the process is running on. This is accomplished by obtaining the systems cycle counter, sleeping, then taking the difference between the starting counter and the new counter to determine how many cycles occurred during the sleep period. This data was gathered several tens of times and then the k-best measurement system was used to determine the clock-speed. By using this system we significantly reduce the possibility of having an interrupt affect the data. This is because it is unlikely for an interrupt to occur in each sleep interval and even more unlikely for them to all take the same amount of time. It is worth nothing that this measurement may overestimate the clock-speed as there is some slight amount of overhead when transitioning between processes when the program yields and returns from sleep.

The second part of the program consists of determining when the program is active and when it is inactive. This is accomplished by keeping track of the cycle counter and when a different greater than a specific hold appears between the old cycle counter and the current cycle counter appears it can be deduced that the process was sleeping during this time period. It is important to note that the proces is bound to run on a single cpu preventing errors that could appear in the cycle count due to running on multiple processors.

Of great importance is choosing an appropriate value for the threshold to determine whether or not a program has be interrupted. The threshold was determined partially through research[1] and partially through experiment. The research was used to obtain a general estimate of how long certain tasks, such as a cache access, would take and this information was further refined through testing appropriate values by using the program. In the end, a value of a 5000 cycles was chosen as the threshold. It can be considered to be towards the high end however, it is unlikely that an interrupt of any kind will take less than 5000 cycles to complete. By setting it to a higher value we can reduce the chance of having disk a access appear as an inactive period within the data. When the threshold was set to values such as 1000 or 2000 there was a consistent trend of a large number of 'interrupts' that appeared in a short time-frame (over multiple runs) suggesting that the threshold was likely counting disk access or some other operation as inactivity when it was not.

After obtaining a set number of periods of activity and inactivity the data is then converted into a graph like the one displayed below. While the program itself only samples the data once, the program itself was run numerous times and the results typically appeared to be very similar. On very rare occasions another process interrupted the program result in a large inactive period.

Results:



Legend: Blue indicates the process is active; Black indicates the process is inactive.

The raw data that the graph was produced from can be found in the 'Activity Periods Data.txt' file. This data has excluded the actual cycle counter numbers and only contains the relevant run times as that data made reading the information harder. If one wishes to know that exact cycle counter values, the data was collected on a cpu with a clock speed of 2.80017Mhz and the cycle counter started at 0, so it is possible to determine the cycle counter values if desired.
Discussion:
To being the discussion, this report will first answer the questions presented in the assignment handout.

1. With what frequency do timer interrupts occur?

By looking at the data it can be seen that timer interrupts appear every 4 ms. There is some slight discrepancy in the data (≈ 0.007) which is likely caused by minor imprecisions when counting the cycles.

2. How long does it take to handle a timer interrupt?
It appears to take between 0.002 and 0.004 ms to handle each interrupt.

3. If it appears that there are other, non-timer interrupts (that is, other short periods of inactivity that don't fit the pattern of the periodic timer interrupt), explain what these are likely to be, based on what you can determine about other activity on the system you are measuring.

Judging by the short duration of these interrupts, it cpu is likely handling short tasks that need immediate responses. It is possible that these are some kind of user input such as mouse movement. Another possibility is that the cpu is handling the top side of a larger interrupt and will handle the rest later. Finally, it is possible that we have accidentally caught a disk access or other program operation by setting the threshold too low.

4. Over the period of time that the process is running (that is, without lengthy interruptions corresponding to a switch to another process), what percentage of time is lost to servicing interrupts (of any kind)?

A very small amount of time is lost to handling interrupts. The total sum of time spent on interrupts is 0.245409 ms while the total time spent is 55.327473 ms. This means that the time spent on interrupts is $\approx 0.4436\%$.

Comparison between expected and experimental:

The clock frequency determined by the program is slightly faster than what the specifications state. The likely reason was explained above.
The timer interrupt occurring every 4ms seems rather fast to me. I would nor-

mally expect that it occurs once every 10 ms. Unfortunately I don't have the permission to view the CONFIG_HZ which specifies the actual time.
The other interrupts that appear were discussed above.

Appendix: In order to reproduce the experiment run './run_experiment_A'. You may need to give yourself execute permissions. Depending on your python installation you may need to run 'python ./run_experiment_A'.

[1] The papers and the comment in https://piazza.com/class/isb6pxw9i2n73q?cid=34 suggested a reasonable starting point.

CPU Data: (Gathered from /proc/cpuinfo)

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 21
model           : 2
model name      : AMD Opteron(tm) Processor 6348
stepping        : 0
microcode       : 0x600081f
cpu MHz         : 2800.000
cache size      : 2048 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
apicid          : 32
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
bugs            : fxsave_leak sysret_ss_attrs
bogomips        : 5599.99
TLB size        : 1536 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm 100mhzsteps hwpstate cpb eff_freq_ro
```

2 Part A - Context Switching

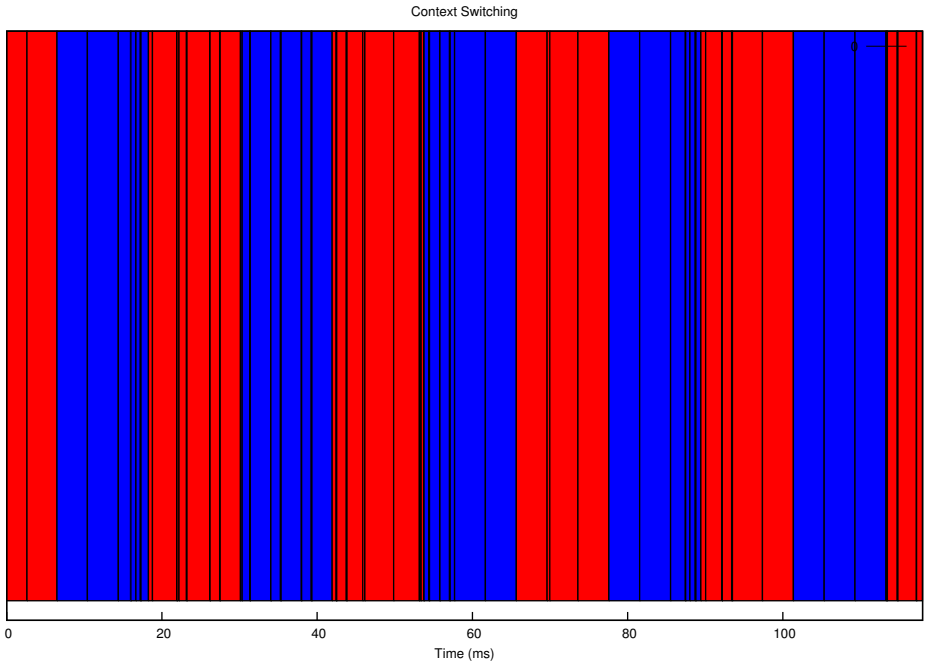
Introduction:
The goal of this experiment is to measure the amount of time a context switch requires. By gathering this data we can gain understanding on the overhead when switching processes.

Methodology:
To gather information on how long a context switch takes, two or more processes where run in parallel while being restricted to the same core. Each of these processes uses the hardware's cycle counter to determine when it is active and inactive. This data is then combined on a time-line and we can determine that the gap when one process stops but before the next process starts is the context switch time.

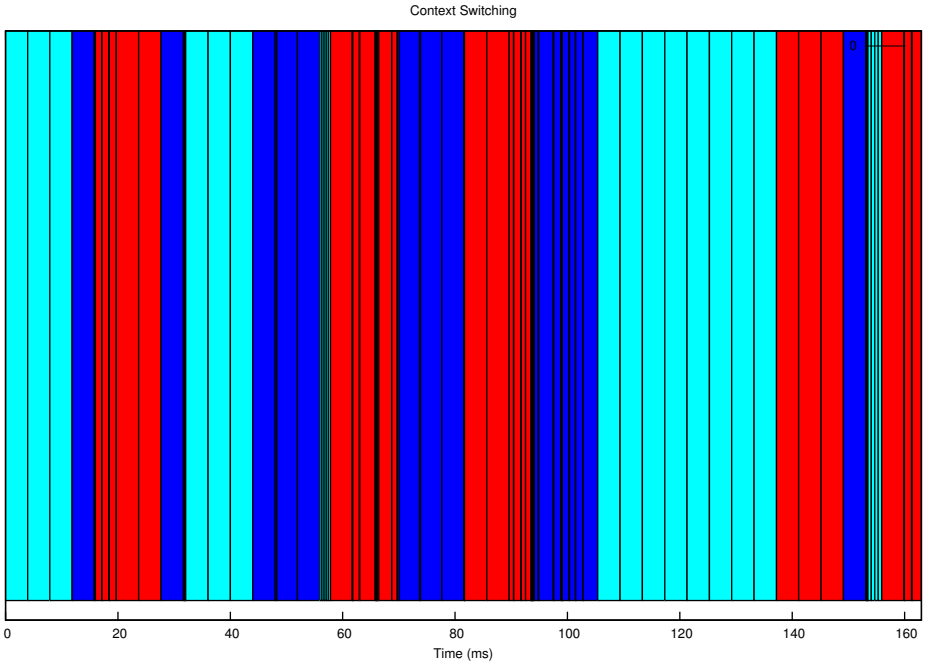
To gather data on the active and inactive periods of each graph the same code and methodology is used as described in the previous session with the only difference being that two copies are being run instead of one.

In order to run both processes simultaneously python's multiprocessing library was used. It is possible that this has some effect on the order of the data, but understanding the next implementation and thus effects is both beyond my level and the specifications of the assignment.

Results:



Legend: Blue represents process 1 is active, Red represents process 2 is active, Black represents that process 1 and process 2 are both inactive.
The raw data that the graph was produced from can be found in the 'Context Data.txt' file. This data has excluded the actual cycle counter numbers and only contains the relevant run times as that data made reading the information harder. If one wishes to know that exact cycle counter values, the data was collected on a cpu with a clock speed of 2.80128Mhz and the cycle counter started at 0, so it is possible to determine the cycle counter values if desired.
The raw data that the graph was produced from can be found in the 'Context Data 2.txt' file. As above, the data excludes cycle counters.



Legend: Blue represents process 1 is active, Red represents process 2 is active, Cyan represents process 3 is active, Black represents that process 1, process 2 and process 3 are all inactive.

The raw data that the graph was produced from can be found in the 'Context Data 3.txt' file. As above, the data excludes cycle counters.

Discussion:
1. How long is a time slice? That is, how long does one process get to run before it is forced to switch to another process?

This depends on the number of processes running (see 2). With only two processes it generally appears that a process runs for 3 timer interrupts or appropriately 12 ms.

2. Is the length of the time slice affected by the number of processes that you are using?

It appears to that having more processes changes the time slice that each process is allocated. This is because the system's scheduler determines when each of the processes will run. In the case of two processes running, each seems to run for a roughly equivalent amount of time. However, when running three processes this does not seem to be as true. From the graph present it can be seen that sometimes the processes will get the same size time slice while at other

times they will get more or less depending on the schedulers decisions. In other cases of the program being run the results were even more chaotic.

From a simpler perspective, it would seem that since each program is effectively identical they should hold the same priority for the scheduler, showing up in sequence with the same size time slices. This seems to, most often, be the case for when only two processes are running but when the third one is added it significantly changes the results. This is due to the fact that the scheduler is much more complicated that as interpreted above.

4. Are you surprised by your measurements? How does it compare to what you were told about time slices in your previous OS course?

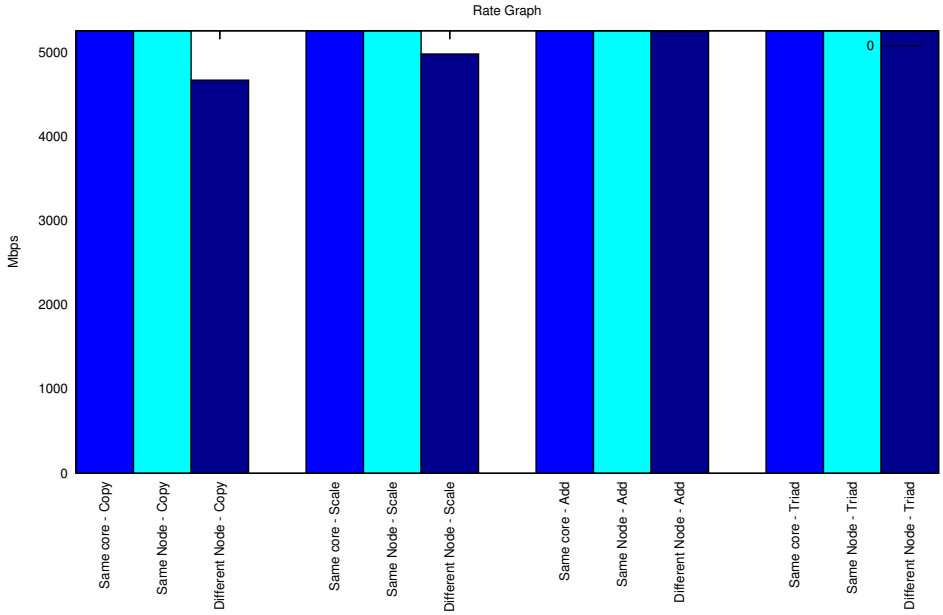
Unfortunately, my memory on what I was taught in previous courses is slightly blurred making it hard to answer this question. From what I remember, I find myself surprised at the speed in which a switch occurs. I expected it to take much longer than was actually required. Thinking it through more thoroughly this may be the expected result for an operation carried out by the hardware.

Appendix: In order to reproduce the experiment run './run_experiment_A_context_2' and './run_experiment_A_context_3'. You may need to give yourself execute permissions. Depending on your python installation you may need to run 'python ./run_experiment_A_context_2' and 'python ./run_experiment_A_context_3'. This will output the 'context 2.eps' and 'context 3.epx' files which contains the graphs. The context 2 graph contains two processes running while the context 3 graph contains three processes running.

The cpu that this was run on was the same as the one for experiment A above.

3 Part B - NUMA

Introduction:
The goal of this experiment is to measure the performance effects of Non-uniform memory access (NUMA). This experiment is to be run on wolf.teach.cs and uses John McCalpin's memory performance benchmark stream to identify the effects of NUMA architecture. The idea behind the experiment is to compare the data generated using numactl and stream against the results expected from the hardware.
Methodology:
Data was gathered by using stream to bind the memory and the physical cpu to to the specified cores (see results) and than taking the results. The only information used was the 'Best Rate MB/s' category. This data was then stored and graphed and was used to compare against the expected results (see discussion). The expected results were obtained by viewing 'numactl –hardware'.
Results:



Legend: Blue is running on the same core, cyan (light blue) is running on different cores but on the same node and dark blue is running on different nodes (and thus different cores).

The raw data can be found in 'Numactl Data.txt'. This includes both the stream data and the hardware data.

Discussion:
By taking a look at the expected results by the hardware we see a chart:

```
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
```

Looking at this data we can see that operations with memory and computation on the same node have a distance of 10 and operations with memory and computation on different nodes have a distance of 16. This means that we are expect operations on different nodes to take longer than those on that run on the same one.
This is different from the results that we actually get. After running the program multiple times it can be seen that the the speed from the operations is roughly the same regardless of which cores it is being run on. In some cases it is faster to run on different nodes than it is to run on the same core. This suggests that the distance of the nodes plays a negligible factor in determining the speed at which the operations are completed. Instead it is likely that other factors have a much greater significance such as the current load of the processor.

It is worth discussing some of the choices in the methodology. In particular two issues arise. The first is that only one sample of data is taken for each core. The reason for this is because each call to stream takes 10 samples internally. Additionally the experiment itself was run multiple times to verify that consistent results were obtained. The second issue is only using the 'MB/s' category of the data. The reason for this is because collecting the other data isn't particularly useful for analysis. The minimum time is a reflection of the maximum speed while the maximum time and average time should be similar to the minimum but also have the chance of being skewed by other processes pre-empting this one on the core. Thus they have all been excluded.

Appendix:
In order to reproduce the experiment run './run_experiment_B'. You may need to give yourself execute permissions. Depending on your python installation you may need to run 'python ./run_experiment_B'.