Alan Zheng
301189136

Please describe what you have found in questions 2–5, complete with data, interpretations and insights. Also, convey the results as clearly as possible using graphs and tables where appropriate. Figures, tables and references do not need to be counted against the two-page written limit. Further, you must briefly explain your measurement methodology (extra credit may be given for clever measurement methodologies).

366, 96, 90, 95, 95, 90, 90, 85, 90, 86, 95, 91, 90, 91, 90, 85, 90, 85, 95, 90
cost of minimal function call: 104ns

The first task involves wrapping timers around a minimal function call and measuring the results. I used CLOCK_MONOTONIC to measure the time because I only care about the relative time. Since it is a minimal function call, this means that there are no parameters and no body to execute. Thus, nothing needs to be pushed onto the stack except for the function return address. Figure1 shows a representation of the data above in a bar chart and one very noticeable thing that stands out is that the first call to the function takes much longer than subsequence calls. This is due to instruction caching. On the first loop, the function is called and the cpu looks to find whether the instruction for that has been cached. The cpu does not find the instruction in the cache, so it copies the instruction from main memory to the cache for faster subsequent access. The copying of this instruction over to the cache from main memory takes a long time. On the next run of the loop, the same function is called. The cpu checks and finds the instruction in the cache, and then executes it. Since the instruction already exists in cache, it takes much less time.

1544, 240, 116, 100, 100, 100, 100, 95, 100, 95, 100, 105, 105, 100, 100, 105, 100, 100, 105, 100
cost of minimal system call: 180ns

For the minimal system call, I wrapped timers around a minimal system call and used CLOCK_MONOTONIC to measure as I only cared about the relative time. The execution of the minimal system call is more expensive than the minimal function call as it performs more instructions in the kernel space (one which includes retrieving the pid) compared to the other, which does not perform any task at all. Also, a function call takes place only in the user space, which makes it a bit faster. A system call on the other hand is constantly switching between user and kernel space. From Figure2, we notice that subsequent execution of getpid() takes much less time than the first execution. This is because the pid value is cached. Similar to the last minimal function

call case, the cache is checked for the pid value. If it is not found, the cpu copies the pid value from main memory to the cpu's L1 cache.

128779, 28195, 42914, 30926, 24417, 21158, 17205, 7712, 6854, 6483, 6111, 5843, 5490, 5241, 5035, 5001, 4755, 4603, 4498, 4459
cost of process switching: 4459ns

For measuring the cost of process/context switching, I also used CLOCK_MONOTONIC as I only cared about the relative time. My thought process for this cost measurement is that context switching only occurs when switching from one process to another. In other words, it happens within the timeframe of when one value is written to a pipe in one process and the same value is read from the same pipe in another process. The two processes in this case are parent and child. I wanted to execute the timer as less as possible to reduce the overhead, which will incur cost, so I decided to start the timer only once, do the context switching N-1 times and then report the time back from the child process in another pipe. I had to use another pipe for the time result because setting the endTime in the child process will not affect the parent process as they both lie in separate memory spaces. This design was made possible because of the read system call functionality of blocking and waiting for input on the other end. My final result is the average of the N-1 times going back and forth between parent and child process. Figure3 also shows that subsequent execution takes much less time than the initial execution. In this case, it is because the cpu caches the values that are saved from and restored to the PCB block.

Alan Zheng
301189136

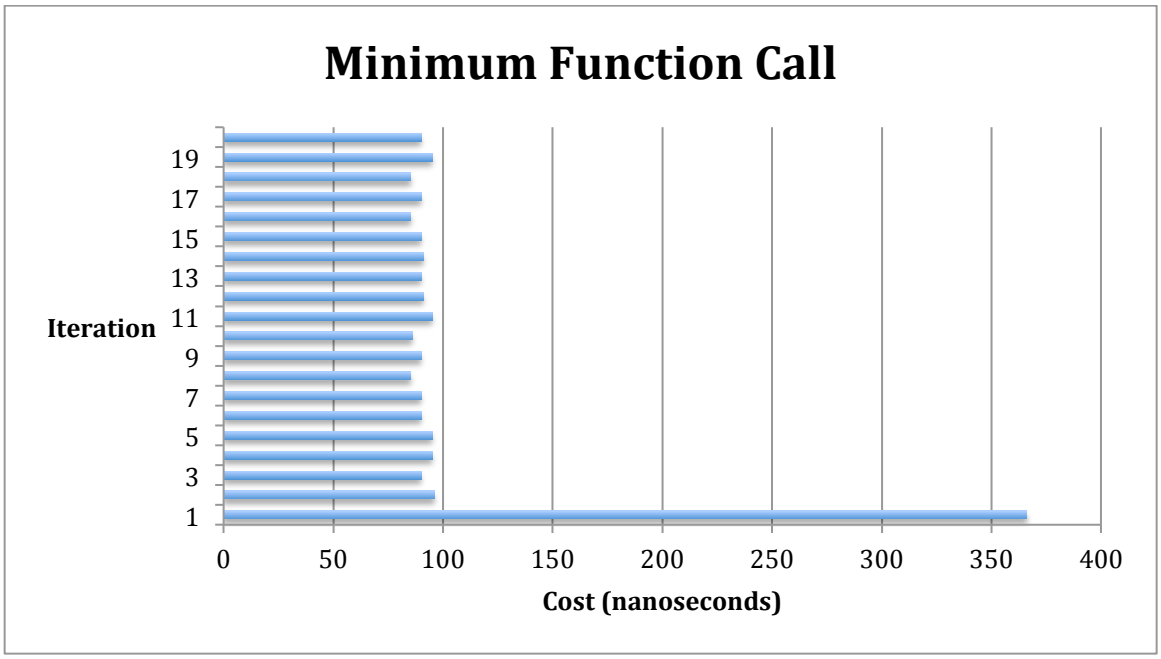

Figure1

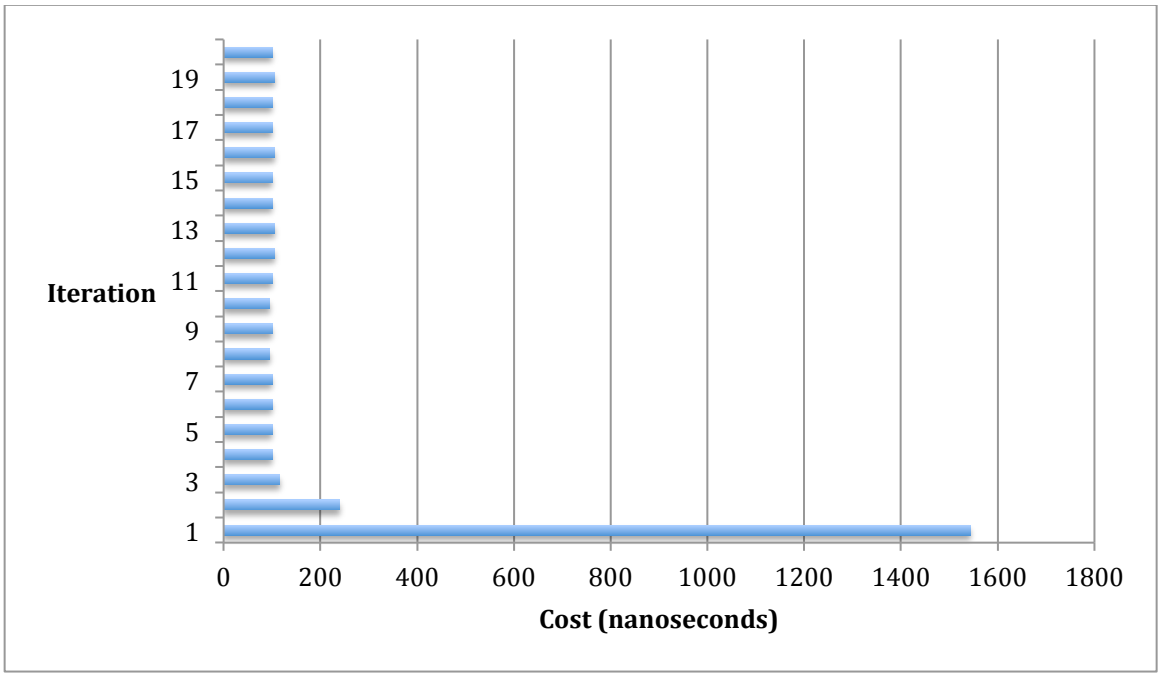

Figure2

Alan Zheng
301189136

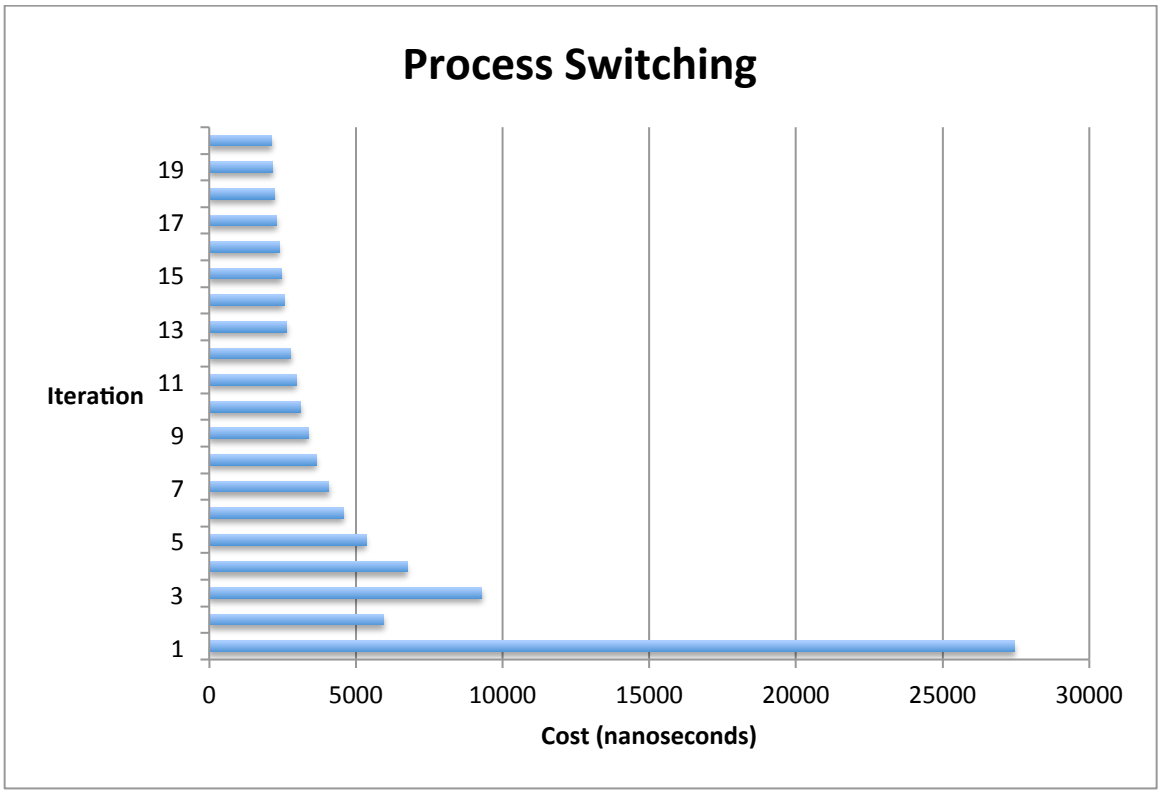

Figure3