In the following report, I will discuss the cost of various actions in C. Namely, the cost of a minimal function call, minimal system call , context-switching in processes and threads.

Sampling rate Justification

In all cases, each evaluation of cost was taken over a sample rate between n = 1 to n = 10000. It is important to evaluate these costs at a high enough sample rate such that an average is taken since we can see some outliers from time to time in terms of measurements. For example, when I set n = 1 in all cases I see the cost of all four cases are at a maximum. This is likely because the OS is pulling data from further in the data storage hierarchy such as the hard disk, main memory etc. However, as n increases, calculations and samples are moved closer to the actual hardware stored inside cache which is quicker than lower storage spaces such as main memory and disk storage. As a result, it is important to sample at a large n-rate and take an average of the results. Table 1 shows the results obtained from the assignment.

n	min function (ns)	min system (ns)	process (ns)	thread (ns)
1	299	6394	82071	1487
5	114	1392	20968	855
10	96	716	12111	627
50	77	200	6403	597
100	75	142	5109	531
500	74	102	4689	498
1000	72	83	2569	484
2000	72	77	1755	471
3500	73	75	1335	462
5000	72	75	3267	462
6500	72	74	1156	471
8000	72	74	1038	469
10000	72	74	1052	455

Table 1. Results for the four phases of measurements

In addition, even if we ran at a constant n, for example n = 3000, we can still get different results. An example is shown in the appendix as Fig.1.

Running the same command 10 times with n = 3000. Shows very consistent results for 90% of the time however for the 5th iteration we see outlier values for Process and Threads. These outliers are due to the fact that we are running on 1 core on the CPU, however on rare occasions there can be a small instance where another program runs on that core as well which can impact the processing time. As a result it is best to take an average over a large sample of N.

Minimal Function Cost

In measuring a minimal function call which accepted no inputs and returned nothing, with a sample rate of n = 1, the cost of the minimal function was found to be 299 ns. Increasing n =

10000 we see that the average cost of a minimal function call drives toward ~72 ns. This is a very "cheap" cost which makes sense since literally nothing is calculated, meaning no registers are required to store values or process calculations.

Minimal System Call Cost

In measuring a minimal system call such as getpid() which returns the process ID of the current running process. With a sample rate of n = 1, the cost was found to be 6394 ns. However, as we set n = 5 we see this value drop by 75% to 1392 ns and increasing n = 10000 we see that the average cost of a minimal function call drives toward ~74 ns. Although getpid() is a minimal system call, it still requires some registers to copy the data and set the data. That explains why at a large n, a minimal system call is slightly longer than a minimal functional call. However, overall the cost is very cheap since the calculations are performed in the kernel space which is very quick due to it's close interaction with hardware.

Process Context Switching Cost

In measuring a Process-Context switch with n = 1 I found that 82071 ns which is very costly. Even increasing n to 10000 the process context switch cost is around ~ 1052 ns. Process context switches as studied, are more costly because each time a process is blocked the OS must make a copy of the entire process and store it into the Process Control Block. When the process becomes unblocked, the OS moves the data stored on the PCB back to the process which takes some time.

Thread Context Switching Cost

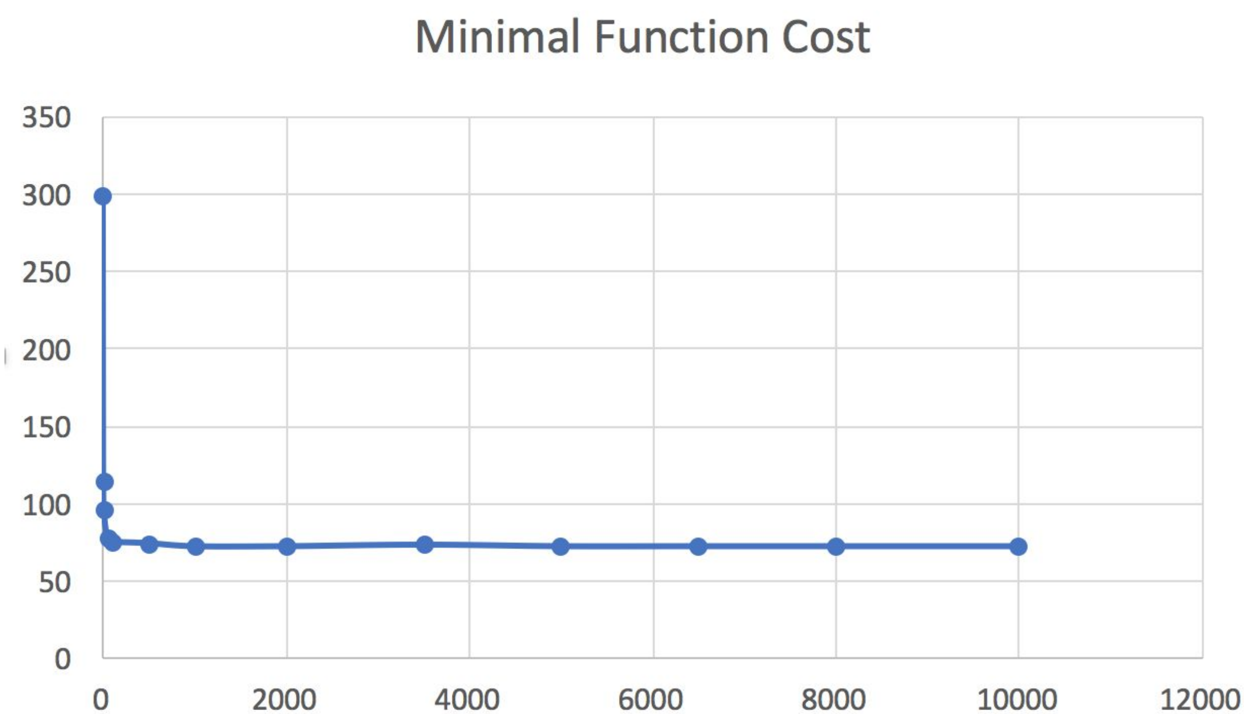
In measuring a thread-context switch with n = 10000 I found that the cost was significantly lower at ~455 ns per context switch. Thread switches are much less costly because in a multithreaded process all the threads share a set of data, global variables, heap space etc. Therefore work can be done to this data efficiently if threads are scheduled properly and critical data is isolated by locks. By limiting contention and spinning by putting threads to sleep, the CPU can be efficiently used by the non-sleeping threads. Also, since most data is shared between the threads, when they wake from sleep the OS does not need to move data back and forth from memory spaces like Process-Context switches via PCB. In a thread's case, when it wakes up the data is accessible right away within the process. As a result, threads have a very cheap cost in terms of context switching in comparison to process based context switches.

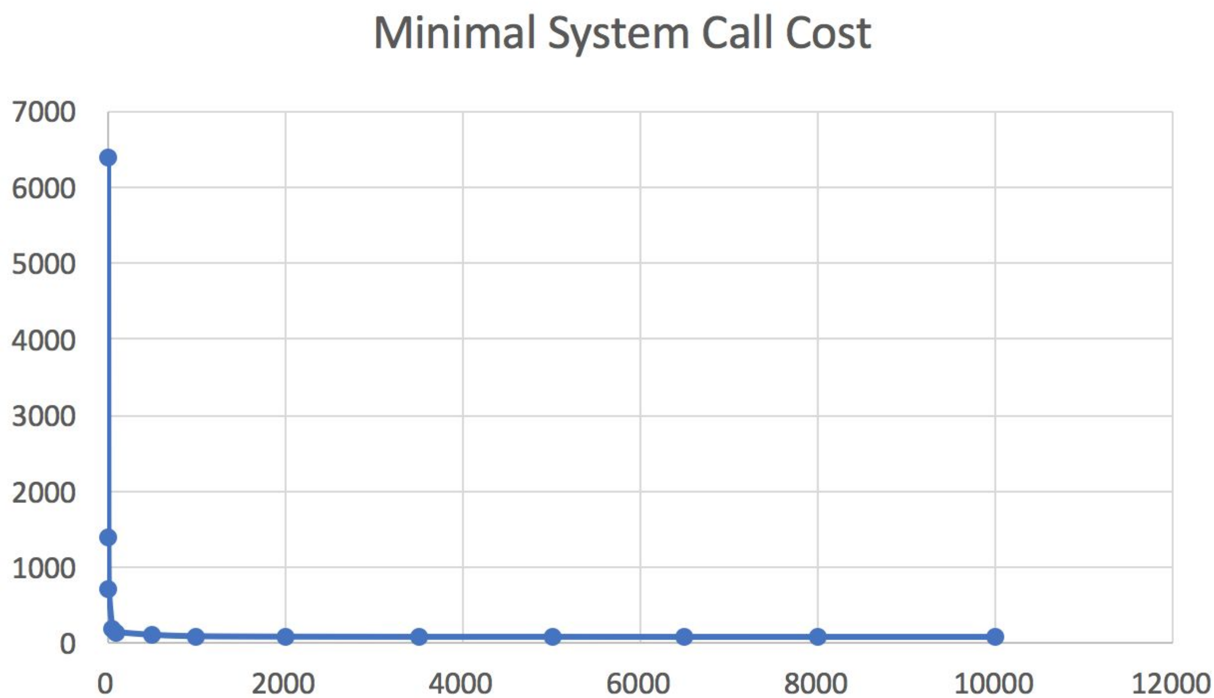
From the study, minimal functions where code is used will have a very small cost since little calculations are required by the registers. In addition, minimal system calls will typically be very cheap as well since they are in the kernel space which interacts very quickly with the hardware. Finally in comparing Process to Thread context switches, It is determined that Thread context switches are cheaper than Process context switches since a blocked Process switches require the OS to completely copy the data into the PCB back and forth.

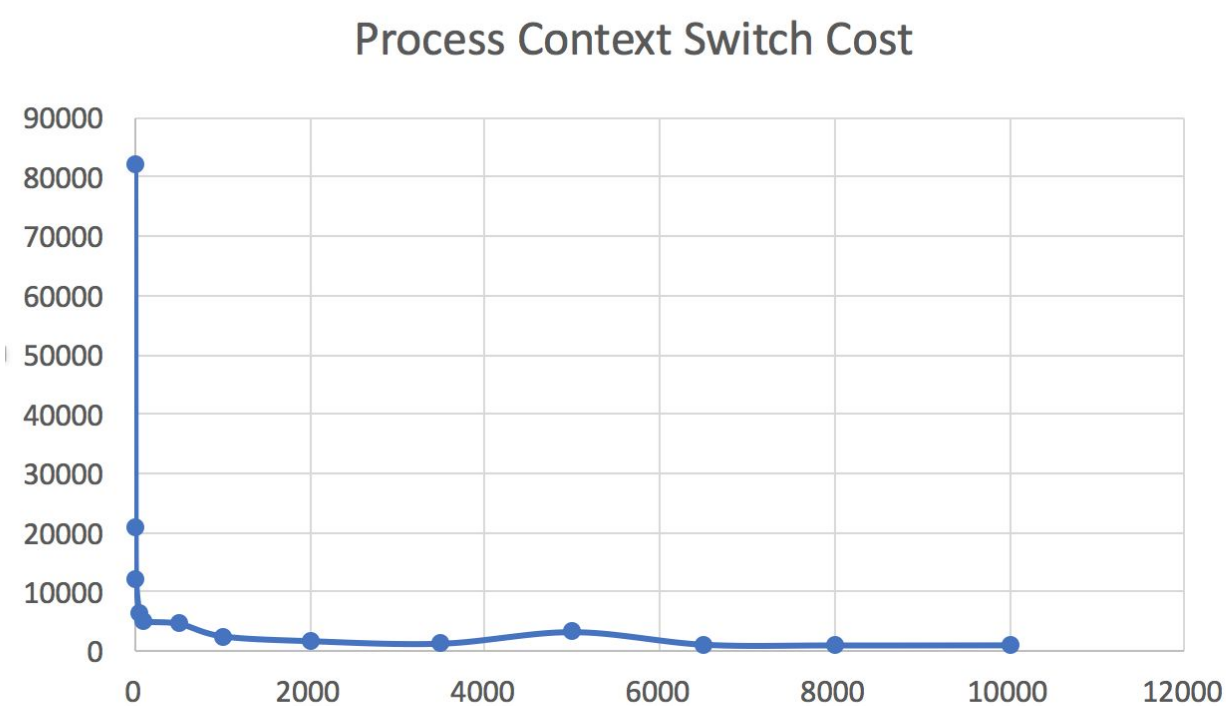
Appendix:

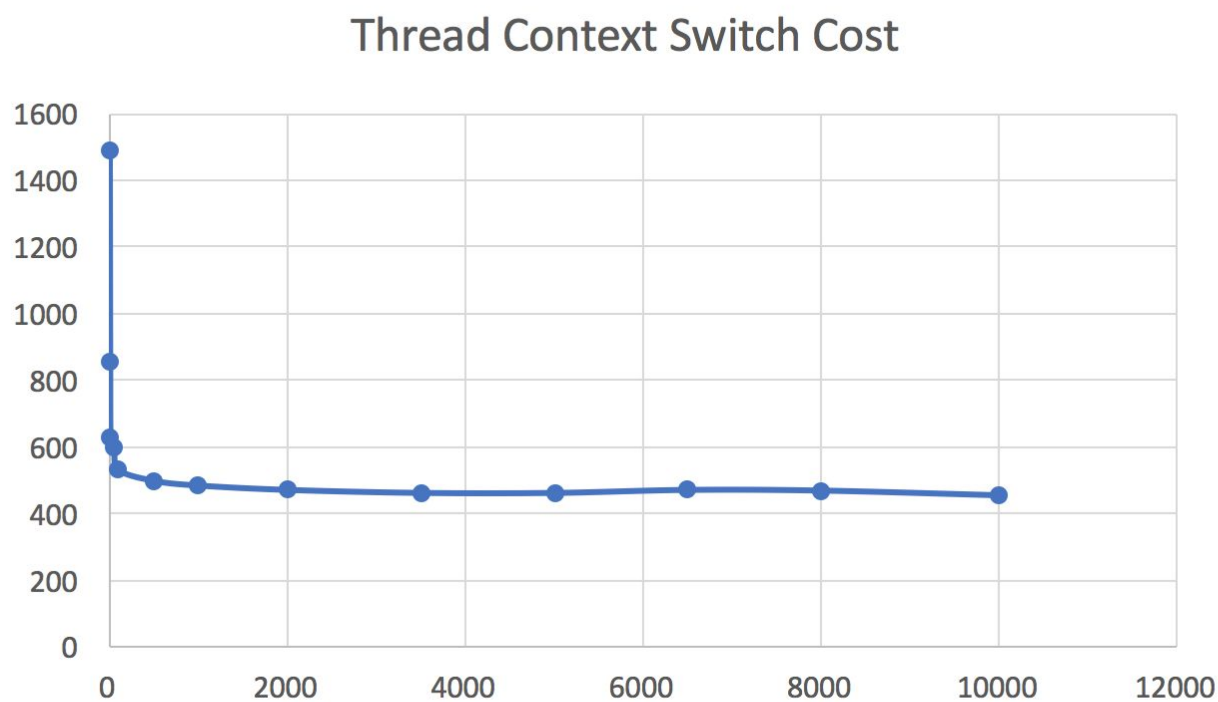
n = 3000				
iteration	min function	min system	process	thread
1	72	76	1389	488
2	87	75	1532	479
3	72	75	1358	542
4	73	76	1425	478
5	72	76	3836	742
6	72	75	1439	468
7	72	75	1440	491
8	72	75	1466	473
9	73	75	1447	483
10	72	75	1406	499

Fig.1 Measurements after 10 iterations with n = 3000




Minimal System Call Cost
7000
6000
5000
4000
3000
2000
1000
0
0
2000
4000
6000
8000
10000
12000


Process Context Switch Cost
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
2000
4000
6000
8000
10000
12000


Thread Context Switch Cost
1600
1400
1200
1000
800
600
400
200
0
0
2000
4000
6000
8000
10000
12000