Project 2

Nielsen, Anders
Western New England University
CPE690

I. Introduction

This report discusses timing and multithreading in C in a Linux environment. System calls, context switching, and TLB access costs were measured, which gave extra insight into the TLB size. Additionally, a demo program to highlight thread synchronization methods was created. Due to the repetitive nature of many of the functions in the programs, they were all compiled with gcc using the no optimizations flag. All experiments were run on a 64bit Ubuntu virtual machine using VirtualBox.

II. System Calls and Context Switching

A. *Timing*

An effective timer was needed in order to carry out the experiments. A high resolution timer with low overhead combined with averaging significant iterations is needed to accurately measure timings on the expected order of nanoseconds. The timer used was the clock_gettime() function in the time.h library. To find the resolution of the timer, a simple c program (clockTest.c) was used to find the time delta between successive calls. This was determined to be on the order of 40-80 nanoseconds. At that resolution and overhead, it was determined that at least 100 iterations of any call would be needed to accurately time it.

B. *System Calls*

System calls are requests directed to the kernel. In order to run these from inside a c program, the syscall() function from the unistd.h library was used. Another c program (syscall.c) was created to repeatedly call sys_gettid. This serves the same function as getpid() with less overhead. This function was used due to its already low cost. The program determined that each function call took 120 nanoseconds on average.

C. *Context Switching*

Context switching occurs when a program enters a new environment for a period of time. This can occur during multithreading, handling interrupts, or other OS needs. Because of the associated costs of context switching, frequent switches and large sets of shared data can heavily slow down processes. The first method of measuring the cost of a switch was through creating two pipes between two forked threads to crosstalk (pipeSW.c). The execution time of each thread was then compared to the total execution time, and the difference was considered to be the total time of all context switches. This method yielded an average result of 50 nanoseconds per switch. As the expected cost was an order of 100 times higher, a different method was then implemented.

The second method used shared memory and futex waits to swap between the main and child threads (futexSW.c). This method doesn't involve adjusting for execution time in the thread and ultimately lead to more accurate results. Upon first running the test, it measured each context switch to be between 10 and 15 microseconds. One more adjustment was made to ensure that each thread ran on the same processor. When running the new program, the average context switch fell to 3 microseconds.

III. TLB Size and Cost

The TLB, or translation lookaside buffer, is a memory cache used for the most recent page table entries. It can quickly take a virtual address and form a physical one if the entry is within the TLB (TLB hit). If the address is not present (TLB miss) it will take longer to retrieve the address. Knowing this, the cost of accessing data will change as data outside of the TLB is used. Timing the access cost should reveal increases in cost as addresses outside of the TLB are accessed. To do this, a program was written (tlb.c) to create an array of sufficient length to write to a certain number of TLB pages provided as an input argument. Another script (tlb.py) was used to run that program with different numbers of pages to iterate through and graph the resulting data. For the system used, two clear spikes can be observed at 8 and 4096 pages.

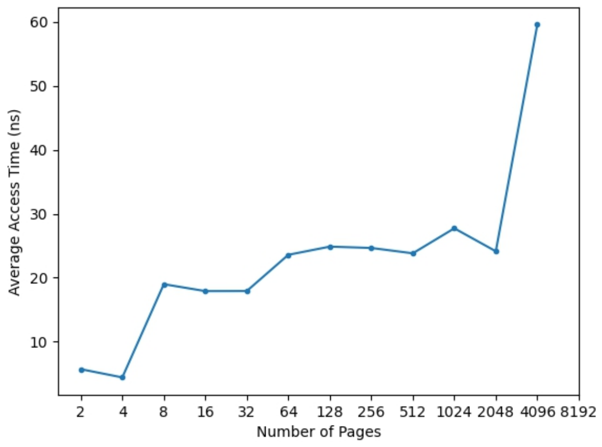


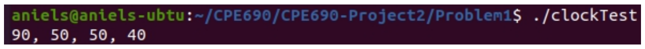
Figure 1. TLB access cost as page number increases

This timing discrepancy can be used in different cryptographic attacks and translate to other cache systems.

IV. Thread Synchronization

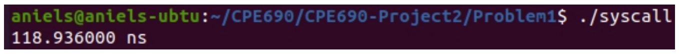
The final part of the project was to create a program to implement a thread scheduler. The premise of the problem was that a haunted house has visitors than can approach from the left or right, but visitors can only be travelling in one way at a time. There is also a maximum capacity of three visitors at any time. The solution created was to have two core threads being a queue and a handler. The queue created a linked list of visitors while the handler contained the logic for allowing a visitor from the queue into the house and spawned a child thread for each visitor as it entered the house. The queue and handler both needed to access the visitor linked list and queue size; the handler also had to share control of the current direction and number of visitors in the house data. This was done using a mutex to lock the thread before changing data and unlock upon completing any modifications. These periods of locked data only included the edits to reduce the time spent locked.

Appendix A - Code Listings



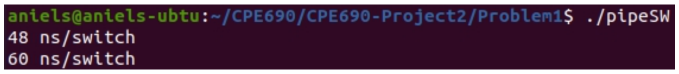
```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem1$ ./clockTest
90, 50, 50, 40
```

Figure 2. Time between successive clock_gettime() calls in nanoseconds



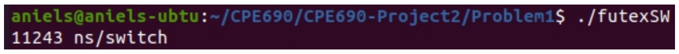
```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem1$ ./syscall
118.936000 ns
```

Figure 3. Time for one system call of SYS_gettid in nanoseconds



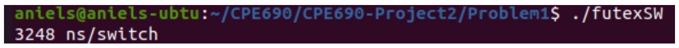
```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem1$ ./pipeSW
48 ns/switch
60 ns/switch
```

Figure 4. Time for one context switch using the pipe method. The resulting data was deemed inaccurate.



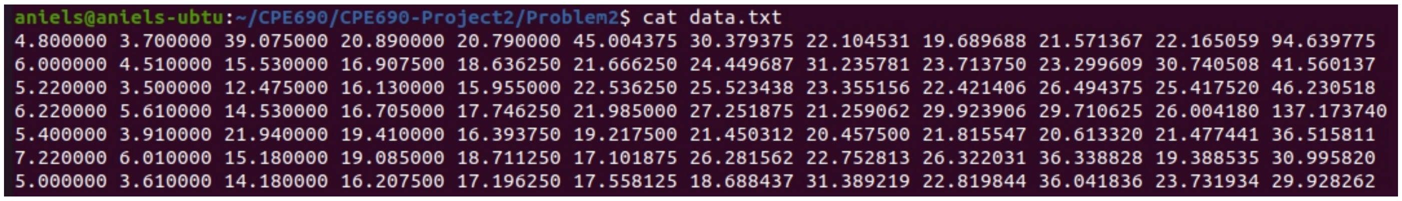
```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem1$ ./futexSW
11243 ns/switch
```

Figure 5. Time for one context switch using the futex method on all processors.



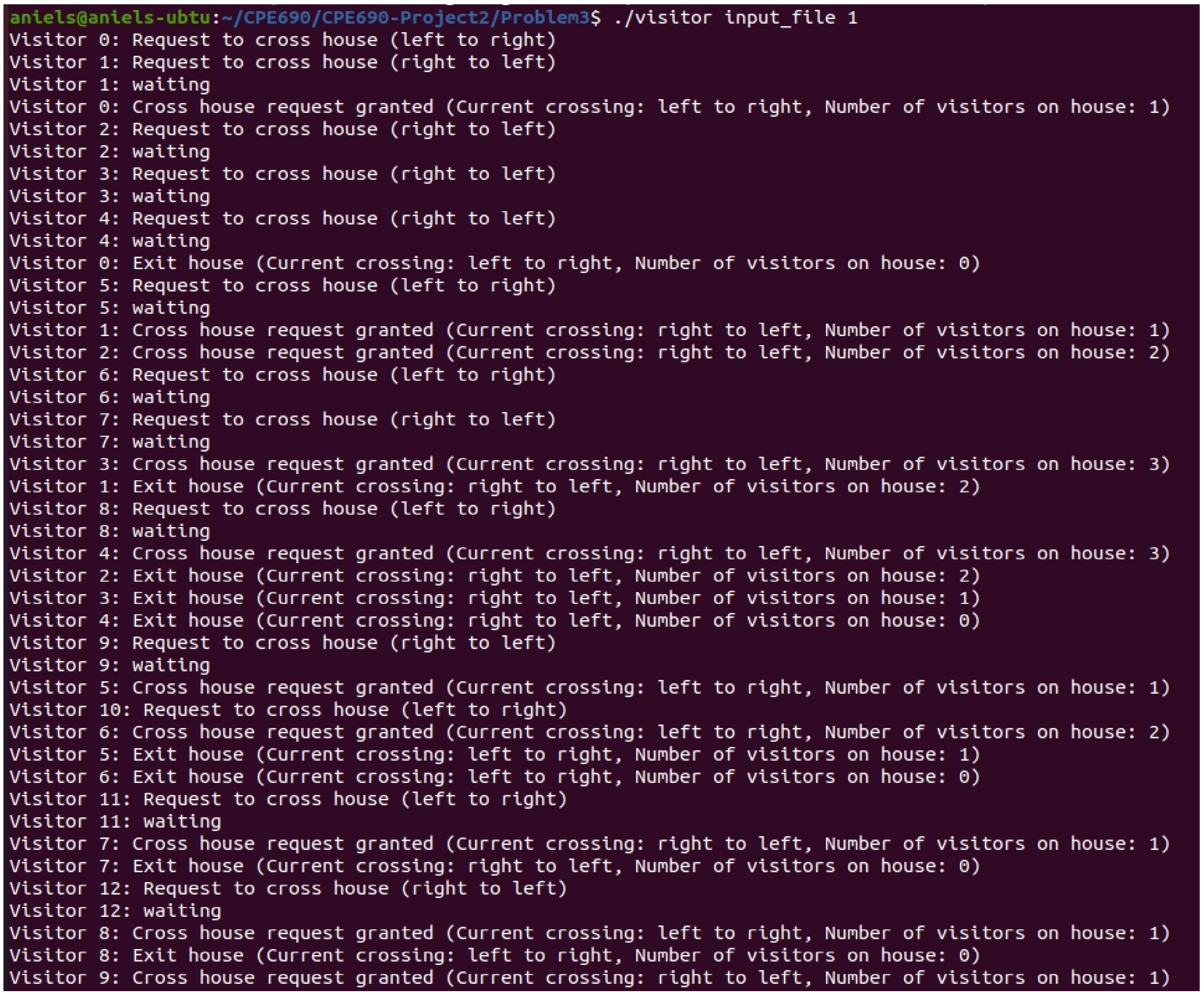
```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem1$ ./futexSW
3248 ns/switch
```

Figure 6. Time for one context switch using futex method on one processor.



```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem2$ cat data.txt
4.800000 3.700000 39.075000 20.890000 20.790000 45.004375 30.379375 22.104531 19.689688 21.571367 22.165059 94.639775
6.000000 4.510000 15.530000 16.907500 18.636250 21.666250 24.449687 31.235781 23.713750 23.299609 30.740508 41.560137
5.220000 3.500000 12.475000 16.130000 15.955000 22.536250 25.523438 23.355156 22.421406 26.494375 25.417520 46.230518
6.220000 5.610000 14.530000 16.705000 17.746250 21.985000 27.251875 21.259062 29.923906 29.710625 26.004180 137.173740
5.400000 3.910000 21.940000 19.410000 16.393750 19.217500 21.450312 20.457500 21.815547 20.613320 21.477441 36.515811
7.220000 6.010000 15.180000 19.085000 18.711250 17.101875 26.281562 22.752813 26.322031 36.338828 19.388535 30.995820
5.000000 3.610000 14.180000 16.207500 17.196250 17.558125 18.688437 31.389219 22.819844 36.041836 23.731934 29.928262
```

Figure 7. Data from tlb.py. The data is organized in columns as time in nanosecond to access 2^num_col pages.



```
aniels@aniels-ubtu:~/CPE690/CPE690-Project2/Problem3$ ./visitor input_file 1
Visitor 0: Request to cross house (left to right)
Visitor 1: Request to cross house (right to left)
Visitor 1: waiting
Visitor 0: Cross house request granted (Current crossing: left to right, Number of visitors on house: 1)
Visitor 2: Request to cross house (right to left)
Visitor 2: waiting
Visitor 3: Request to cross house (right to left)
Visitor 3: waiting
Visitor 4: Request to cross house (right to left)
Visitor 4: waiting
Visitor 0: Exit house (Current crossing: left to right, Number of visitors on house: 0)
Visitor 5: Request to cross house (left to right)
Visitor 5: waiting
Visitor 1: Cross house request granted (Current crossing: right to left, Number of visitors on house: 1)
Visitor 2: Cross house request granted (Current crossing: right to left, Number of visitors on house: 2)
Visitor 6: Request to cross house (left to right)
Visitor 6: waiting
Visitor 7: Request to cross house (right to left)
Visitor 7: waiting
Visitor 3: Cross house request granted (Current crossing: right to left, Number of visitors on house: 3)
Visitor 1: Exit house (Current crossing: right to left, Number of visitors on house: 2)
Visitor 8: Request to cross house (left to right)
Visitor 8: waiting
Visitor 4: Cross house request granted (Current crossing: right to left, Number of visitors on house: 3)
Visitor 2: Exit house (Current crossing: right to left, Number of visitors on house: 2)
Visitor 3: Exit house (Current crossing: right to left, Number of visitors on house: 1)
Visitor 4: Exit house (Current crossing: right to left, Number of visitors on house: 0)
Visitor 9: Request to cross house (right to left)
Visitor 9: waiting
Visitor 5: Cross house request granted (Current crossing: left to right, Number of visitors on house: 1)
Visitor 10: Request to cross house (left to right)
Visitor 6: Cross house request granted (Current crossing: left to right, Number of visitors on house: 2)
Visitor 5: Exit house (Current crossing: left to right, Number of visitors on house: 1)
Visitor 6: Exit house (Current crossing: left to right, Number of visitors on house: 0)
Visitor 11: Request to cross house (left to right)
Visitor 11: waiting
Visitor 7: Cross house request granted (Current crossing: right to left, Number of visitors on house: 1)
Visitor 7: Exit house (Current crossing: right to left, Number of visitors on house: 0)
Visitor 12: Request to cross house (right to left)
Visitor 12: waiting
Visitor 8: Cross house request granted (Current crossing: left to right, Number of visitors on house: 1)
Visitor 8: Exit house (Current crossing: left to right, Number of visitors on house: 0)
Visitor 9: Cross house request granted (Current crossing: right to left, Number of visitors on house: 1)
```

Figure 8. Example output from the visitor program.