~ (Assignment 6 (CLO2, C3, PLO2) 25 Points) ~

In this homework, you are to measure the size and cost of accessing a TLB. The idea is based on work by Saavedra-Barrera [SB92], who developed a simple but beautiful method to measure numerous aspects of cache hierarchies, all with a very simple user-level program.

The basic idea is to access some number of pages within a large data structure (e.g., an array) and to time those accesses. For example, let's say the TLB size of a machine happens to be 4 (which would be very small, but useful for the purposes of this discussion). If you write a program that touches 4 or fewer pages, each access should be a TLB hit, and thus relatively fast. However, once you touch 5 pages or more, repeatedly in a loop, each access will suddenly jump in cost, to that of a TLB miss. The basic code to loop through an array once should look like this:

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump)
a[i] += 1;
```

In this loop, one integer per page of the array a is updated, up to the number of pages specified by NUMPAGES. By timing such a loop repeatedly (say, a few hundred million times in another loop around this one, or however many loops are needed to run for a few seconds), you can time how long each access takes (on average). By looking for jumps in cost as NUMPAGES increases, you can roughly determine how big the first-level TLB is, determine whether a second-level TLB exists (and how big it is if it does), and in general get a good sense of how TLB hits and misses can affect performance.

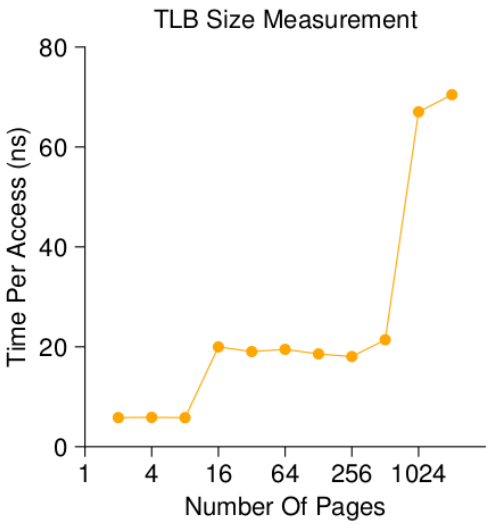

Figure above shows the average time per access as the number of pages accessed in the loop is increased. As you can see in the graph, when just a few pages are accessed (8 or fewer), the average access time is roughly 5 nanoseconds. When 16 or more pages are accessed, there is a sudden jump to about 20 nanoseconds per access. A final jump in cost occurs at around 1024 pages, at which point each access takes around 70 nanoseconds. From this data, we can conclude that there is a two-level TLB hierarchy; the first is quite small (probably holding between 8 and 16 entries); the second is larger but slower (holding roughly 512 entries).
The overall difference between hits in the first-level TLB and misses is quite large, roughly a factor of fourteen. TLB performance matters!

Questions

1. **[3]** There is a linux command for determining pagesize. What is this command? What is the page size which you get by executing this command?
   Submissions: The command and its output

2. **[4]** For timing, you'll need to use a timer (e.g., gettimeofday()). How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)
   Submissions: Precision of the timer and time duration of the operation which can be instrumented by the timer. Provide reasons/justifications.

3. **[13]** Write a program, called tlb.c (**[8]** points), that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.
   a. **[1]** One thing to watch out for is compiler optimization. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?
   b. **[2]** Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? (hint: look up "pinning a thread" on Google for some clues) What will happen if you don't do this, and the code moves from one CPU to the other?
   c. **[2]** Another issue that might arise relates to initialization. If you don't initialize the array above before accessing it, the first time you access it will be very expensive, due to initial access costs such as demand zeroing. Will this affect your code and its timing? What can you do to counterbalance these potential costs?

   Submissions: Code tlb.c with aspects mentioned in parts (a), (b) and (c) addressed
4. **[5]** Create and show a graph which plots the access time vs number of pages which are accessed (similar to the figure in the previous page).

(Solution)

(1)

Terminal Command:
getconf PAGE_SIZE

Terminal Output:
4096

(2)

Using assignment 3 results(screenshot attached):

Cost of syscalls:

Run **sysCallCost.c**

Terminal Output:

```
Start time is '899977'
End   time is '986993' in 100000 number of syscalls
Time it took is '0.870160'
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngin
e-In-3pquhccv.lgq" 1>"/tmp/Microsoft-MIEngine-Out-o2hco2bl.5hc"
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/008_HA3$
```

-> 0.87016 us on average for each.

Since, It is a system call, It should take at least 0.87016 microseconds.
So I will adjust the number of trials such that the gap between two gettimeofday() functions is at least 100 times the cost of the context switch. We can also increase the number of trials to see where numbers are converging to which would give better average results.

(3)

(a)

In dynamic allocation we do not need to specify keywords like volatile because heap is controlled by the programmer.

(b)

C code snippet for tying to cpu core 0:

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(0, &set);
```

(c)

Multiple trials lead to average results. Hence such cost should get diluted.

C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

double findAccessCost(int numberOfPages, int numberOfTrials)
{
   //Instantiation of an object of timeval structure defined in sys/time.h
   struct timeval current_time;
   //Profiling start time
   gettimeofday(&current_time, NULL);

   //Variables for tracking start and end time
   double start_time_us = current_time.tv_usec;
   double end_time_us = 0;
   int start_time_sec = current_time.tv_sec;
   int end_time_sec   = 0;

   long PAGESIZE = sysconf(_SC_PAGE_SIZE);
   int jump = PAGESIZE / sizeof(int);
   int* a = malloc(numberOfPages*jump*sizeof(int));

   for(int s = 0; s < numberOfTrials; s++)
   {
     for (int i = 0; i < numberOfPages * jump; i += jump)
     {
       a[i] += 1;
       }
   }

   //Getting end time
   gettimeofday(&current_time, NULL);
   end_time_us = current_time.tv_usec;
   end_time_sec = current_time.tv_sec;
   free(a);
   //Printing log
   printf("Start time is %d seconds '%d' microseconds \n", (int)start_time_sec,
(int)start_time_us);
   printf("End   time is %d seconds '%d' microseconds \n", (int)end_time_sec,
(int)end_time_us);
   printf("Microseconds time it took is '%f'\n",
(1000000*(((end_time_us-start_time_us)+1000000*(end_time_sec-start_time_sec))/(numb
erOfPages * jump))/numberOfTrials));

   return
(1000000*(((end_time_us-start_time_us)+1000000*(end_time_sec-start_time_sec))/(numb
erOfPages * jump))/numberOfTrials);
};
```

C code main for test:

```c
#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sched.h>
#include <stdio.h>
#include "tlb.c"

int main(int argc, char *argv[])
{
   cpu_set_t set;
   CPU_ZERO(&set);
   CPU_SET(0, &set);
   //How many times a loop within the function should iterate
   //Which means number of jumps(or pages in this sense) * number of trials
   int iteration = 10000000;   //10 million
   for(int i = 2; i<=1<<20; i=i<<1)
   {
       printf("us took with %d pages is '%f'\n", i, findAccessCost(i, (iteration/i
> 100) ? iteration/i  : 100));
   }


   return EXIT_SUCCESS;
}
```

Main test results:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/ADOS_C/012_HA6$ ./main >> log.txt
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/ADOS_C/012_HA6$ cat log.txt
us took with 2 pages is 10.537500
us took with 4 pages is 11.172949
us took with 8 pages is 10.336133
us took with 16 pages is 39.397852
us took with 32 pages is 39.918164
us took with 64 pages is 42.228125
us took with 128 pages is 40.755469
us took with 256 pages is 39.606171
us took with 512 pages is 40.496710
us took with 1024 pages is 40.927619
us took with 2048 pages is 34.992639
us took with 4096 pages is 40.836092
us took with 8192 pages is 45.376625
us took with 16384 pages is 50.201377
us took with 32768 pages is 66.674244
us took with 65536 pages is 95.381254
us took with 131072 pages is 122.215748
us took with 262144 pages is 122.052133
us took with 524288 pages is 123.477746
us took with 1048576 pages is 123.668648
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/ADOS_C/012_HA6$ python TLBSizeMeasuringPlot.py log.txt
```

Python generate plot: