**Project 2**

**Due date: Friday January 30, 11:59pm on Canvas**

**In this second project, you’ll measure the costs of a system call, the cost of a context switch and the cost of a procedure call. The output of this project includes:**

* **A report (in PDF) that describes:**
  + **The methods you tried, challenges you encountered, what you’ve learned in the process and which method(s) eventually worked.**
  + **The results you obtained (presented as tables or plots, whatever makes best sense). Your results must include experiments on machines you have access to, such as your laptops or research computers, etc.**
  + **A listing of the output of the execution of your project.**
* **The report should be of the quality of the papers we read in terms of structure and presentation. There is no requirement on the length of your report: you’ll be graded based on how well your report shows your understanding of the problem. You may do that in 2 pages or 5 pages (or more if you need).**
* **Code in C that implements the objectives of this project.**
  + **Include a makefile for easy compilation**
  + **Include a README file that describes in detail the steps to be taken to run your project (e.g., arguments, etc.).**
  + **The code must have detailed comments explaining what each piece of code does (see the rubric for more details)**

**Part 1: System Calls, Procedure Calls, and Context Switch Measurements**

Measuring the cost of a system call and that of a procedure call is relatively easy. For example, you could repeatedly call a simple system call (e.g., **getpid** or performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call.

One thing you’ll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is ***gettimeofday****();* read the man page for details. What you’ll see there is that *gettimeofday()*returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to gettimeofday() to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you’ll have to run in order to get a good measurement result.

If gettimeofday() is not precise enough for you, you might look into using the **rdtsc** instruction available on x86 machines.

Measuring the cost of a context switch is a little trickier. The lmbench benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of communicating like this repeatedly, lmbench can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets.

One difficulty in measuring context-switch cost arises in systems with more than one CPU. Your machine might be multi-processor: run lscpu and nproc on one of the machines to see this. What you need to do on such a system is ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the ***sched\_setaffinity****()* call is what you’re looking for. Check it out on your machines with:

***man 2 sched\_setaffinity***

By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

**Part 2: Measuring the Size and Cost of Accessing a TLB**

in this part, you are supposed 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. Read his work for more details.

The basic idea is to access some number of pages within large data structure (e.g., an array) and to time those accesses. If number of pages accessed by a program is equal or less than the TLB size of a machine, each access should be a TLB hit, and thus relatively fast. However, once program touches some pages more than the TLB size, 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);**

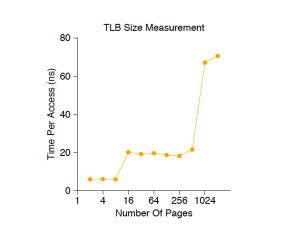
**for (i = 0; i < NUMPAGES \* jump; i += jump) {**

**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, 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

Here is an example graph:



As you can see in the graph, when 16 or more pages are accessed, there is a sudden jump to about 20 nanoseconds per access. Another jump in cost occurs at around 1024 pages, where each access takes around 70 nanoseconds. As a result, 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).

**Questions:**

1. For timing, you’ll need to use a timer such as that made available by 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)

2. Write the program, called tlb.c 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. How many trials are needed to get reliable measurements?

3. Now run this program (name it run.c) using system() (check man system) while varying the number of pages accessed from 1 up to a few thousand( perhaps incrementing by a factor of two per iteration) . Your program should output in this format (or similar): <no. of pages>,< corresponded access time>. What you want to see is at least one, ideally two jumps, as in the plot above. Run this program on different machines and gather some data. How many trials are needed to get reliable measurements?

4. Next, graph the results, making a graph that looks similar to the one above. Include your conclusion about the TLB hierarchy.

5. 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?

6. 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? What will happen if you don’t do this?

**Part- 3 Process Synchronization**

**1. Introduction**

Synchronization is a challenge that crops up whenever a parallel computational design is used. This assignment involves tackling a synchronization problem in the context of sharing a common resource.

**2. Problem**

There is a terrifying haunted house in the Bush Gardens attracting large number of visitors excited to experience it during the Halloween time. The haunted house is 2-way having a door on right side and another on left side. Visitors can enter the house from one door and exit from other on the other side. But, given that the inside of the house is totally dark, if two visitors cross the house in opposite directions, they crash into each other in the middle. Furthermore, the house is only strong to hold three visitors inside of it at the same time. So, if more than three get in, it will collapse. To prevent crashing and collapsing, sensors installed at each end of the house to notify a controller computer when visitors arrive or depart the house in either direction. The controller uses the sensor input to control signal lights (green and red) at either side of the house. We would like to design the controller program using a synchronization scheme

with the following properties:

* Once a visitor enters the house from a side, it is guaranteed to exit from the other side without running into another visitor going the opposite way.
* There are **never more than three visitors** in the house. The order of the visitors crossing the house should be preserved; i.e., order in which they enter the house should be the order in which they exit the house.
* A continuing stream of visitors crossing in one direction should not prevent visitors going the other way indefinitely (**no starvation**). Solve this requirement such that the FIFO order is preserved. That is, a visitor trying to cross from the left/right who arrives earlier than a visitor trying to cross from the opposite direction gets in the house first. Starvation can be defined as follows. Let us say that visitor V1 arrives at the right end of the house at time T1 and visitor V2 arrives at the left end of the house at time T2, and T1 < T2. If visitor V2 is allowed to cross the house before visitor V1, then visitor V1 undergoes starvation.

**3. Implementation Specifications**

* Show how to implement the controller program correctly using a synchronization scheme in C using the Pthreads library.
* Make use of an input parameter that indicates the time required for a visitor to cross the house. Assume that all the visitors require the same time to cross the house.
* Overall, this problem deals with the access to a common resource, the haunted house. Access to the house here means access for the threads to the critical section of your program, and how long a visitor takes to cross the house is indicated by the amount of sleep time (in seconds), which is the amount of thread sleep time in the critical section of your program.

**4. Input**

The input to the program should be provided as an input file. The name of the input file and the time (in seconds) required for a visitor to cross the house have to be command-line parameters.

The format of the input file is given below.

* Line 1: L,R,R,R,R,R,L,L,R (a sequence of alphabets L (meaning left) and R (meaning right)

separated by a comma, that indicate the side of the house a visitor is trying to cross from. Please note that, the delimiter here is a ',' (comma), and it does not appear after the last symbol in the first line of the file.

* Please note that following the aforementioned input format is a must.

**5. Output**

Please find below how a sample run might look like. Also, note that your program run might not replicate this output although this is correct.

**./visitor input\_file 2**

Visitor 1: Request to cross house (left to right)

Visitor 1: Cross house request granted (Current crossing: left to right, Number of visitors

on house: 1)

Visitor 2: Request to cross house (left to right)

Visitor 2: Cross house request granted (Current crossing: left to right, Number of visitors

on house: 2)

Visitor 3: Request to cross house (right to left)

Visitor 3: Waiting

Visitor 1: Exit house (Current crossing: left to right, Number of visitors on house: 1)

Visitor 4: Request to cross house (left to right)

Visitor 4: Waiting

Visitor 2: Exit house (Current crossing: left to right, Number of visitors on house: 0)

Visitor 3: Cross house request granted (Current crossing: right to left, Number of visitors

on house: 1)

Visitor 3: Exit house (Current crossing: right to left, Number of visitors on house: 0)

Visitor 4: Cross house request granted (Current crossing: left to right, Number of visitors

on house: 1)

Visitor 4: Exit house (Current crossing: left to right, Number of visitors on house: 0)

Please note that it is important that you display the same information (verbatim) on the screen to depict the visitor's action in your program. As mentioned above, the order of these statements might not be replicated by your program, but the order of these statements should adhere to the problem definition described in this document.