Assignment 4

Due Apr 11 by 11:59pm **Points** 18 **Submitting** a file upload **File Types** c

Requirements

- Don't perform this assignment inside a virtual machine. VMs have increased context switch times and other effects that might make these problems work differently. Also, by default, your VM has only a single physical processor allocated to it.
- Check the return value of every system call or library call that might fail. Do something sensible (such as printing an error message and exiting) if any call fails unexpectedly.
- Your code should handle any number of threads up to (and including) 99.
- All code should be written in C and compiled in 64-bit mode (that is, don't use the -m32 flag)
- All code must compile using the -O2 -Wall -Wextra -Werror options to the default version of GCC (in/usr/bin/gcc) on the CADE machines.
- Each C file you hand in should include a main() function.
- Only print things to STDOUT or STDERR when you are asked to do so. Of course it is fine if you want to use extra printfs for your own debugging purposes, but they need to be disabled by default.

Hints

- Note where we've asked you to add specific comments to your code. Don't forget.
- It is fine to call functions from the C library.
- Variables shared between threads need to be declared as volatile. You can read more about volatile **here (http://en.cppreference.com/w/c/language/volatile)**. The key point: "This means that within a single thread of execution, a volatile access cannot be optimized out or reordered relative to another visible side effect that is separated by a **sequence point (http://en.cppreference.com/w/c/language/eval_order)** from the volatile access." Roughly, volatile loads/stores will appear in the instruction sequence in the order of the C code (sequence points, are, e.g. lines separated by semicolon along with things like logical operators).
- Pthreads programs require the extra -pthread argument to compile: gcc -O2 -Wall -Wextra -Werror foo.c -o foo -lm -pthread
- "man pthreads" gives an overview of this API including a list of commands that each have their own man pages.
- Various Pthreads tutorials can be found on the web; for example, **this one (http://randu.org/tutorials/threads/)** is pretty good.

Problem 1

Write a pthreads program that creates a number of threads that repeatedly access a critical section that is synchronized using **Lamport's Bakery algorithm (http://en.wikipedia.org/wiki/Lamport's_bakery_algorithm)** .

Your program should take two command line options. First, the number of threads. Second, the number of seconds to run for.

Just before terminating, your program should say how many times each thread entered the critical section. Make sure starvation does not occur.

Your code should have logic like this inside the critical section:

```
assert (in_cs==0);
in_cs++;
assert (in_cs==1);
in_cs++;
assert (in_cs==2);
in_cs++;
assert (in_cs==3);
in_cs=0;
```

to check for violations of mutual exclusion. Run your Bakery solution for a while (at least 10 minutes) to make sure mutual exclusion is never violated. If it is, you have a bug to fix. in_cs should be a global variable of type volatile int.

To avoid problems with the memory model, always restrict your threads to a single CPU like this:

```
taskset -c 1 ./problem_1 5 5
```

Turn in your single-core Bakery code as problem_1.c.

Problem 2

When running on a single core, a spin-lock version of Bakery can be extremely inefficient because it will tend to get preempted while holding the lock. In this situation, performance can often be greatly improved by yielding the processor instead of just spinning while waiting for the lock to be acquired. Make this improvement to your Bakery algorithm using the sched_yield() call. If you don't see a performance improvement of at least 5x, you're probably doing something wrong.

Turn in your yielding, single-core Bakery code as problem_2.c.

Problem 3

Take your non-yielding Bakery code from problem 1 and run it on multiple cores (that is, run it without the taskset command). It should break. That is, your code should signal a violation of mutual exclusion. If this does not happen, make sure you are properly checking for violations of mutual exclusion. If you still cannot make the code break, talk to an instructor.

Here is a memory fence function that works on x86 and x86-64 when you compile with gcc:

```
void mfence (void) {
  asm volatile ("mfence" : : : "memory");
}
```

The memory system cannot reorder memory operations around a fence. Use it to fix your Bakery implementation so that it works when run on multiple processors. Try to add as few fences as possible while still creating a working mutual exclusion algorithm. Include comments describing why your fences are necessary and also why additional fences are unnecessary. **Section 8.2.2 in the Intel manual (https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf)** covers the Intel memory model, its reordering, and the behavior of fences with respect to reordering.

Hand in your well-commented code as problem_3.c.

Problem 4

Write a multicore-safe spinlock for the x86-64. It should conform to this interface:

```
void spin_lock (struct spin_lock_t *s);
void spin_unlock (struct spin_lock_t *s);
```

The basis for your spinlock should be the atomic compare-and-swap instruction provided by the x86-64 architecture. If you want, you may use this code:

```
/*
 * atomic_cmpxchg
 *
 * equivalent to atomic execution of this code:
 *
 * if (*ptr == old) {
 *   *ptr = new;
 *   return old;
 * } else {
 *   return *ptr;
 * }
 *
 */
static inline int atomic_cmpxchg (volatile int *ptr, int old, int new)
{
  int ret;
  asm volatile ("lock cmpxchgl %2,%1"
    : "=a" (ret), "+m" (*ptr)
    : "r" (new), "0" (old)
    : "memory");
  return ret;
}
```

Your spinlock code should not be very long or complicated. If it is, you are probably doing something wrong.

Hint: Put a single int into struct spin_lock_t and consider the lock to be held when the value is 1 and to be free when the value is 0.

Test your spinlock in the same way that you tested your multicore-safe version of Bakery. Hand in the full program as problem_4.c.

Problem 5

The spinlock you wrote for Problem 4 does not make any particular guarantees about fairness. Implement another spinlock that is fair in the sense that threads gain access to the critical section in the same order in which they begin waiting on it. The fair spinlock should implement basically the same interface (i.e., taking a pointer to a struct representing the lock) but you'll likely need to use a different struct.

Your fair spinlock may be based on the atomic_cmpxchg primitive above, or you may find it easier to use this code:

```
/*
 * atomic_xadd
 *
 * equivalent to atomic execution of this code:
 *
 * return (*ptr)++;
 *
 */
static inline int atomic_xadd (volatile int *ptr)
{
  register int val __asm__("eax") = 1;
  asm volatile ("lock xaddl %0,%1"
  : "+r" (val)
  : "m" (*ptr)
  : "memory"
  );
  return val;
}
```

Hint: A fair mutex can be implemented Bakery-style using two integers: one representing "the customer currently being served" and the other representing "the customer who just started waiting in line."

Test your spinlock in the same way that you tested your multicore-safe version of Bakery. Hand it in as problem_5.c.

Problem 6

5.39. An interesting way of calculating pi is to use a technique known as Monte Carlo, which involves randomization. This technique works as follows: Suppose you have a unit circle inscribed within a square with sides of length 2; the circle and the square are centered at (0, 0) (see Figure 4.18). First, generate a series of random points as (x, y) pairs. These points must fall within the coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle. Next, estimate pi by performing the following calculation: pi = 4 x (number of points in the circle) / (total number of points).

Write a multithreaded program that estimates pi using this technique. Have a parent program create several threads, each of which generates random points and determines if the points fall within the circle. Each thread will have to update the global count of all points that fall within the circle. Protect against race conditions on updates to the shared global variable by using mutex locks.

After all of the child threads have exited, the parent thread will calculate and output the estimated value of pi. Your code for this problem should take two command-line arguments: first, the number of threads, second the number of seconds to run for. Hand in your solution as problem_6.c.

Some Rubric

Criteria	Ratings	Pts
Problem1		3.0 pts
Problem2		3.0 pts
Problem3		3.0 pts
Problem4		3.0 pts
Problem5		3.0 pts
Problem6		3.0 pts
		Total Points: 18.0