In this assignment, you will study the performance of two important synchronization primitives used
in parallel programs namely, locks and barriers.

PART-A: LOCK DESIGN [60 POINTS]
---

1. Develop Acquire and Release functions for the following lock designs.

(a) Lamport's Bakery lock (make sure to avoid false sharing)
(b) Spin-lock employing cmpxchg instruction of x86
(c) Test-and-test-and-set lock employing cmpxchg instruction of x86
(d) Ticket lock
(e) Array lock

Notes:

(a) Assume cache block size of 64 bytes.
(b) The spin-lock code is as discussed in the class using the CompareAndSet function.
(c) Test-and-test-and-set lock code is as discussed in the class.
    You will have to develop a TestAndSet function using the cmpxchg instruction of x86.
    This function will be similar to the CompareAndSet function.
    If you want, you can use the xchg instruction of x86 also to develop the TestAndSet function.
    You will use this TestAndSet function to develop the Test-and-test-and-set lock.
    The test loop can be implemented in C/C++.

(d) & (e) You will need to develop a FetchAndInc function using the cmpxchg instruction.
          The rest of the code can be written in C/C++.
          For array lock, make sure to avoid false sharing.

You will compare the aforementioned five lock designs with POSIX mutex, a lock developed using binary semaphores (refer to example discussed in class), and #pragma omp critical. For this study, you will use the following simple critical section involving two shared variables
x and y, both initialized to zero.

```
{
   x = y + 1;
   y++;
}
```

Each thread executes this critical section ten million times (10^7). The code executed by each thread is shown below where N=10^7.

```
for (i=0; i<N; i++) {
   Acquire (&lock);
   assert (x == y);
   x = y + 1;
   y++;
   Release (&lock);
}
```

The overall code structure is shown below when using POSIX thread interface.

```
Start timing measurement.
Create threads.
Each thread executes the aforementioned loop.
Join threads.
Stop timing measurement.
assert (x == y);
assert (x == N*t); // t is the number of threads
Report measured time.
```

The overall code structure is shown below when measuring the efficiency of #pragma omp critical.

```
Start timing measurement.
#pragma omp parallel num_threads (t) private (i)
{
   for (i=0; i<N; i++) {
#pragma omp critical
      {
         assert (x == y);
         x = y + 1;
         y++;
      }
   }
}
Stop timing measurement.
assert (x == y);
assert (x == N*t); // t is the number of threads
Report measured time.
```

Report the measured time of the eight locking techniques in a table as you vary the number of threads from 1 to 16 in powers of two. Each row of the table should report the measured times for a particular thread count (make eight columns in the table). Prepare a ninth column
of the table where you mention the best locking technique that you observe for each thread count.

IMPORTANT notes on reordering of memory operations:

1. I assume that you will be running your experiments on Intel x86 machines, which implement the TSO memory consistency model. If you need to eliminate the possibility of a reordering that TSO may allow, please use the mfence instruction as we have done in some of the examples demonstrated in the class.
2. Since in most of your lock implementations, the unlock function is just a store operation, the compiler is going to inline that function in your main program. So, now your critical section and the unlock taken together may look like the following (not correct x86 sequence, but captures the essence).

```
load r, y      // r is some register
inc r          // r++
store r, x     // x <-- r
store r, y     // y <-- r
store 0, lock  // release lock
```

This is okay and in fact, good because it rids you of the overhead of calling functions. However, since the compiler won't know that the last store is an unlock, it will see all the three stores as normal stores. Therefore, it can reorder them arbitrarily because these are independent stores. Note that Intel x86 CPUs will not do this reordering because they maintain total store order (TSO), but the compiler can do. For example, after reordering, the code may look like the following.

```
load r, y      // r is some register
inc r          // r++
store r, x     // x <-- r
store 0, lock  // release lock
store r, y     // y <-- r
```

However, such reordering would violate atomicity of the critical section because some other thread may get the lock even before the previous critical section's updates are completed (the assertion x==y will fail if one update is done, but not the other). To stop the compiler from doing this reordering, you need to insert an "empty" inline assembly instruction specifying that this "empty" instruction may change memory contents (i.e.,

```
place "memory" in the clobber list) just before you set (*lock) to zero in
your release function. This forces the compiler to schedule all pending
memory operations before the next instruction (which is the unlock store).
An example release function is shown below. Note that this inline assembly
code does not introduce any extra instruction in the compiled code, but
stops the compiler from doing the incorrect reordering. This is
essentially a hack to locally turn off compiler-induced instruction
reordering.

void Release (int *lock)
{
   asm(""::::"memory");
   (*lock) = 0;
}

Note that this reordering is more likely to happen as you increase the
optimization level of the compiler e.g., -O3 will most probably do this
reordering while -O0 may not. You should use -O3 for compiling your code.

PART-B: BARRIER DESIGN [40 POINTS]
----------------------------------

You will be comparing the performance of the following six barrier implementations.

(a) Centralized sense-reversing barrier using busy-wait on flag
(b) Tree barrier using busy-wait on flags
(c) Centralized barrier using POSIX condition variable
(d) Tree barrier using POSIX condition variable
(e) POSIX barrier interface (pthread_barrier_wait)
(f) #pragma omp barrier

Note: Use POSIX mutex locks in implementations (a), (c), (d).

You will conduct the performance measurement by executing a parallel program that has one
million (10^6)
barriers and nothing else. The following loop is executed by each thread where N is one
million.

for (i=0; i<N; i++) {
   Barrier(...);
}

The overall code structure is shown below when using POSIX thread interface.

Start timing measurement.
Create threads.
Each thread executes the aforementioned loop.
Join threads.
Stop timing measurement.
Report measured time.

The overall code structure is shown below when measuring the efficiency of #pragma omp
barrier.

Start timing measurement.
#pragma omp parallel num_threads (t) private (i)
{
   for (i=0; i<N; i++) {
#pragma omp barrier
   }
}
Stop timing measurement.
Report measured time.

Report the measured time of the six barrier implementations in a table as you vary the
```

number
of threads from 1 to 16 in powers of two. Each row of the table should report the measured times for a particular thread count (make six columns in the table). Prepare a seventh column
of the table where you mention the best barrier implementation that you observe for each thread
count.

WHAT TO SUBMIT

Place all your Acquire, Release, and Barrier functions in a single C/C++ file named sync_library.x
where the .x extension should be chosen according to the language used. Name the functions differently
for different lock and barrier implementations. So, this file will have seven Acquire functions, seven
Release functions, and five Barrier functions. The omp critical and omp barrier do not require separate
functions. I have given a few examples below. Please put a comment specifying which lock or barrier
implementation a function corresponds to.

```
/* Acquire for POSIX mutex */
void Acquire_pthread_mutex (pthread_mutex_t lock)
{
   pthread_mutex_lock(&lock);
}

/* Release for POSIX mutex */
void Release_pthread_mutex (pthread_mutex_t lock)
{
   pthread_mutex_unlock(&lock);
}

/* Barrier for POSIX */
void Barrier_pthread (pthread_barrier_t barrier)
{
   pthread_barrier_wait(&barrier);
}
```

In another file named pthread_main_lock.x, put your multi-threaded test program for benchmarking
lock designs using the POSIX threads. It is okay if this file has to be manually changed for calling
appropriate Acquire/Release functions. In yet another file named omp_main_lock.x, put your multi-threaded test program for benchmarking #pragma omp critical. Similarly, create pthread_main_barrier.x and omp_main_barrier.x.

Put all your results along with observed trends and explanation in a PDF report file.

Please send your submission (five program files and a PDF report) via email to cs433autumn2024@gmail.com with subject "Assignment#2 submission Roll#X". Replace X by your roll number in the subject line.