In this project, we extended the uthread library (provided as the solution code) to support synchronization mechanisms and asynchronous I/O operations. The initial uthread library allowed thread creation and joining but lacked synchronization primitives to control thread execution. We introduced locks, spinlocks, and condition variables to enable thread synchronization. We also added the ability to make asynchronous read and write calls to perform I/O in the background while allowing other threads to run. We will also conduct a performance comparison between lock and spinlock as well as asynchronous and synchronous I/O.

i) How to Compile:

Assuming all the files are in the directory, let's say project2, we can compile the executable using ***make***, which relies on Makefile. This will build all the files in the lib, solution, and test folder.

ii) How to Run:

./uthread-sync-demo <num_producer> <num_consumer>
./perf_async
./perf_lock <lock|spin> <num_threads> <iterations_per_thread> <cs_work>
./spinlock_test
./async_io_test
./lock_condvar_test <num_of_prodcuers> <num_of_consumers>

The makefile specifies all the targets and builds accordingly.

```
# Use instructor's solution objects for uthread and TCB when desired.
OBJ_SOLN = ./solution/TCB_soln.o ./solution/uthread_soln.o ./lib/Lock.o ./lib/CondVar.o ./lib/SpinLock.o ./lib/async_io.o

# Otherwise, use our own (if you want/need):
OBJ = ./lib/TCB.o ./lib/uthread.o ./lib/Lock.o ./lib/CondVar.o ./lib/SpinLock.o ./lib/async_io.o

# Test files / object files
MAIN_OBJ_SYNC_DEMO = ./tests/uthread_sync_demo.o
PERF_LOCK_OBJ      = ./tests/lock_vs_spin_perf.o
PERF_ASYNC_OBJ     = ./tests/async_io_perf.o
SPINLOCK_TEST_OBJ  = ./tests/spinlock_test.o

# NEW: Add these for lock_condvar_test.cpp and async_io_test.cpp
LOCK_CONDVAR_TEST_OBJ = ./tests/lock_condvar_test.o
ASYNC_IO_TEST_OBJ     = ./tests/async_io_test.o

# Pattern rules:
./lib/%.o: ./lib/%.cpp $(DEPS)
    $(CC) $(CXXFLAGS) -c -o $@ $<

./tests/%.o: ./tests/%.cpp $(DEPS)
    $(CC) $(CXXFLAGS) -c -o $@ $<

# Default target: build everything using the solution objects
all: uthread-sync-demo-from-soln perf_lock perf_async spinlock_test lock_condvar_test async_io_test

# Build the producer-consumer demo (solution objects for uthread/TCB).
uthread-sync-demo-from-soln: $(OBJ_SOLN) $(MAIN_OBJ_SYNC_DEMO)
    $(CC) -o uthread-sync-demo $^ $(LDFLAGS)

# Performance evaluation: Lock vs SpinLock test.
perf_lock: $(OBJ_SOLN) $(PERF_LOCK_OBJ)
    $(CC) -o perf_lock $^ $(LDFLAGS)

# Performance evaluation: Synchronous vs Asynchronous I/O.
perf_async: $(OBJ_SOLN) $(PERF_ASYNC_OBJ)
    $(CC) -o perf_async $^ $(LDFLAGS)

# SpinLock test
spinlock_test: $(OBJ_SOLN) $(SPINLOCK_TEST_OBJ)
    $(CC) -o spinlock_test $^ $(LDFLAGS)

# NEW: Lock + CondVar test
lock_condvar_test: $(OBJ_SOLN) $(LOCK_CONDVAR_TEST_OBJ)
    $(CC) -o lock_condvar_test $^ $(LDFLAGS)

# NEW: Async I/O test
async_io_test: $(OBJ_SOLN) $(ASYNC_IO_TEST_OBJ)
    $(CC) -o async_io_test $^ $(LDFLAGS)
```

Implementation Details:

- **Locks & Spinlocks**: We implemented basic synchronization mechanisms to manage access to shared resources between threads. The lock mechanism follows a traditional blocking approach, while the spinlock uses busy-waiting to acquire the lock. The performance of both locks was compared under different workloads.
- **Condition Variables**: Condition variables were implemented to allow threads to wait for certain conditions to be met before continuing execution. This feature is essential for managing complex thread interactions in multi-threaded applications.
- **Asynchronous I/O**: To enhance I/O performance, we allowed I/O operations to be executed in the background, preventing the entire process from blocking and allowing other threads to continue their execution.

Testing and Evaluation:

After implementing the synchronization mechanisms, we developed tests to ensure the correctness of the APIs. An executable (uthread-sync-demo-solution) containing the solution for the producer-consumer test has also been provided to us. By running this, we were able to verify that our implementation matches the expected output.

```
jain0232@csel-kh1250-08:/home/jain0232/Documents/project2 $ ./uthread-sync-demo 5 5
Creating producer threads
Creating producer threads
Creating producer threads
Creating producer threads
Creating producer threads
Creating consumer threads
Creating consumer threads
Creating consumer threads
Creating consumer threads
Creating consumer threads
Joining producer threads
Consumed 100000 items
Consumed 200000 items
Consumed 300000 items
Consumed 400000 items
Consumed 500000 items
Consumed 600000 items
Consumed 700000 items
Consumed 800000 items
Consumed 900000 items
```

To verify the correctness of our SpinLock implementation, we created a custom test file, spinlock_test.cpp, designed to test the lock() and unlock() functionality of the SpinLock class. In this test, we utilized 5 threads that continuously acquire and release the spinlock while incrementing a shared counter, sharedCounter.

- Shared Resource: A counter (sharedCounter) that is incremented by each thread.
- Threads: 5 threads running concurrently, each attempting to acquire the lock, increment the counter, and then release the lock.

During the test, we observed that the value of sharedCounter was correctly incremented as each thread consistently acquired and released the spinlock. This demonstrated that the synchronization was working correctly, ensuring that the counter value remained consistent even with multiple threads modifying it concurrently.

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./spinlock_test
Created thread with id 1
Created thread with id 2
Created thread with id 3
Created thread with id 4
Created thread with id 5
Final sharedCounter = 50000 (expected 50000)
```

To verify the correctness of our async_read() and async_write() implementations, we wrote a completeness test. This test involved reading a file with a few bytes of data using async_read() and writing data using async_write(), ensuring that both operations complete successfully without blocking the main thread.

- File Operations: A small file containing a few bytes of data was used for both reading and writing.
- Async I/O: We utilized async_read() to read data from the file and async_write() to write data asynchronously.

The test confirmed that both async_read() and async_write() completed as expected, allowing other threads to execute while the I/O operations were in progress. This ensured that our asynchronous I/O functions correctly handle non-blocking behavior and can be used in real-world applications without blocking the entire process.

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./async_io_test
[Main] Created writerTid=1, readerTid=2
[Writer] Wrote 22 bytes to async_io_test.txt
[Reader] Read 22 bytes: Hello from async I/O!
```

The lock_condvar_test program demonstrates the combined use of Lock and Condition Variable (CondVar) to implement a producer-consumer scenario with a bounded buffer.

- Producers: A fixed number of items, defined by the constant PRODUCER_ITEMS, are produced by the producer threads and inserted into a circular buffer.
- Consumers: Consumer threads remove items from the buffer.
- The shared buffer is protected by a Lock to ensure mutual exclusion, preventing race conditions.
- Two Condition Variables, notFull and notEmpty, are used to coordinate waiting and signaling between producers and consumers.
- notFull signals producers when there is space available in the buffer.
- notEmpty signals consumers when there is at least one item to consume.

This test case verifies that the Lock and Condition Variable implementations work correctly, ensuring mutual exclusion, preventing race conditions, and coordinating thread blocking and waking between producer and consumer threads. It guarantees that the synchronization mechanisms prevent overflows, underflows, and ensure proper coordination in a multi-threaded environment.

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./lock_condvar_test 2 2
[Producer 1] Produced one item (left=2).
[Producer 1] Produced one item (left=1).
[Producer 1] Produced one item (left=0).
[Producer 1] Done producing all items!
[Producer 2] Produced one item (left=2).
[Consumer 1] Consumed item from Producer 1
[Consumer 1] Consumed item from Producer 1
[Consumer 1] Consumed item from Producer 1
[Consumer 1] Consumed item from Producer 2
[Producer 2] Produced one item (left=1).
[Consumer 1] Consumed item from Producer 2
[Producer 2] Produced one item (left=0).
[Producer 2] Done producing all items!
[Consumer 2] Consumed item from Producer 2
[Main] All producers have finished producing.
[Consumer 1] Exiting (no more items)!
[Consumer 2] Exiting (no more items)!
[Main] All consumers have exited.
```

We then evaluated the performance of locks and spinlocks in a multi-threaded environment with **uthread threads** to determine which synchronization mechanism provided better performance.

Performance Evaluation of Lock vs Spinlock:

The performance of Locks and Spinlocks was compared by using a timer (e.g., std::chrono::high_resolution_clock) to measure execution time under different critical section sizes. The screenshots highlight the test results.

Results: In scenarios with smaller critical sections, the spinlock performed better due to its reduced overhead. However, for larger critical sections, the lock mechanism showed better performance, as the spinlock's busy-waiting mechanism introduced significant CPU overhead.

Impact of Critical Section Size: As the critical section size increased, the spinlock performance degraded due to continuous spinning, whereas the lock mechanism remained more efficient.

Multi-core Considerations: Since uthread is a uniprocessor user-thread library, performance on a multi-core system might differ, with spinlocks likely to perform better due to parallelism, while locks could face contention on multiple cores.

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_lock spin 20 10000 5000
Performance Evaluation - SpinLock Test
Number of threads       : 20
Iterations per thread   : 10000
Critical section work   : 5000 iterations
Final sharedCounter     : 200000
Elapsed time            : 8.92598 seconds
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_lock lock 20 10000 5000
Performance Evaluation - Lock Test
Number of threads       : 20
Iterations per thread   : 10000
Critical section work   : 5000 iterations
Final sharedCounter     : 200000
Elapsed time            : 9.47301 seconds
```

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_lock lock 20 10000 8000
Performance Evaluation - Lock Test
Number of threads       : 20
Iterations per thread   : 10000
Critical section work   : 8000 iterations
Final sharedCounter     : 200000
Elapsed time            : 16.884 seconds
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_lock spin 20 10000 8000
Performance Evaluation - SpinLock Test
Number of threads       : 20
Iterations per thread   : 10000
Critical section work   : 8000 iterations
Final sharedCounter     : 200000
Elapsed time            : 14.2745 seconds
```

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_lock spin 20 10000 10000
Performance Evaluation - SpinLock Test
Number of threads       : 20
Iterations per thread   : 10000
Critical section work   : 10000 iterations
Final sharedCounter     : 200000
Elapsed time            : 19.8503 seconds
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_lock lock 20 10000 10000
Performance Evaluation - Lock Test
Number of threads       : 20
Iterations per thread   : 10000
Critical section work   : 10000 iterations
Final sharedCounter     : 200000
Elapsed time            : 18.3726 seconds
```

Performance Evaluation of Synchronous vs Asynchronous I/O:

We compared the performance of synchronous I/O (blocking read() and write()) with asynchronous I/O (non-blocking async_read() and async_write()) with different buffer sizes (showing the load of I/O operations).

Results: Asynchronous I/O showed better performance when there were multiple threads performing I/O, as it allowed background processing while other threads continued their execution. Synchronous I/O, in contrast, blocked the entire process, limiting concurrency and efficiency.

Impact of I/O Load: As the amount of I/O increased, asynchronous operations became more beneficial, while synchronous operations suffered from increased blocking.

Thread Workload: The amount of other available thread work also impacted performance, with asynchronous I/O benefiting more when there were multiple threads performing other tasks concurrently.

```
gupt0414@csel-kh1260-01:/home/gupt0414/projects/project2 $ ./perf_async
Synchronous Write: 16777215 bytes in 0.00709181 seconds.
Synchronous Read : 16777215 bytes in 0.155271 seconds.
Asynchronous Write: 16777215 bytes in 0.00663973 seconds.
Asynchronous Read : 16777215 bytes in 0.00222262 seconds.
```

```
Scenario 4 completed.
Number of I/O threads       : 10
I/O iterations per thread   : 100
I/O load multiplier         : 5
Number of Compute threads   : 5
Total elapsed time          : 14.8893 seconds.
```

This project not only added synchronization and asynchronous I/O capabilities to the **uthread** library (using the provided solution code) but also provided a comprehensive performance evaluation, highlighting the advantages and limitations of different synchronization and I/O approaches.