**Design Decisions**

The uthread library created provides all the major API's that pthread provides. It is a user managed library where we don't call kernel threads but in turn use getcontext, setcontext, makecontext and timer interrupts to manage multiple threads.

There is a ready queue, containing the threads which are in ready state and can be run by the processor. Threads when created are added to this queue. There are two ways in which a context switch can take place.

- willingly, by calling uthread_yield
- There exists a virtual timer which is started whenever a context starts. The interrupt handler takes care of the context switch.

Scheduling takes place in a round robin fashion.

After a thread has completed executing, we move it to a finished queue and dont remove it from the system completely. At this point, if there is a thread that is waiting on this thread that just got over, we move it from the blocked state to the ready state. After that we move the finished thread to the blocked state. When we execute the join function, we deallocate all the memory and free the queues.

There are 2 other functions which are provided - utherad_suspend and uthread_resume - which move a thread to the blocked state and back to the ready state respectively.

a. What (additional) assumptions did you make?

- Number of threads will be created within the given limit.
- The user code is responsible to call join for each thread created.
- User code can call join on a thread id exactly once.
- User code will not cause deadlock situations. For example: waiting on a thread that is suspended and has no means to get back to the ready pool.
- User code is aware of the function signature. For example : how to call uthread_create, uthread_join etc.
- The TCB function increaseQuantum is used to set the quantum per thread.
- The uthread function get_total_quantum = number of active threads * quantum_per_thread

b. Did you add any customized APIs? What functionalities do they provide?

- N/A

c. How did your library pass input/output parameters to a thread entry function? What must makecontext, do “under the hood” to invoke the new thread execution?

- The library creates a wrapper function called stud where the thread entry function is called. The uthread_create function takes the following arguments :
    - a function pointer : thread entry function
    - Arguments for the thread entry function.

  The arguments to the thread entry function is of type void* and is passed as is to the thread entry function. Since this argument comes from the user code, the thread entry function can typecast it to the appropriate type and use it as it wishes to.
- Output from the thread_entry_function is updated during the uthread_join call. The uthread_join call takes thread id and a pointer to a void * as its argument. Inside the join function the void * that is being referenced by the argument is updated with the results. This can be dereferenced and used accordingly in user code.
- Makecontext points the program counter of the context passed as argument, to the thread entry function and puts the function arguments and the function on the stack of that context. Thus when we do a setcontext in order to switch threads, the program counter of the processor starts pointing to the thread_entry_point_function and the function starts executing.

d. How do different lengths of time slice affect performance?

- When the pi program is run with the parameters - 100000000 8, we can see the following behaviour based on the timeslice - higher the timeslice, lower the execution time. For 50us - average execution time was 1.6859718 and for 5000us the execution time was 1.653839. This happens because there are no blocking calls and by decreasing the time slice we are just increasing the number of context switches. This has a negative impact on the run time.

e. What are the critical sections your code has protected against interruptions and why?

- Whenever library related data structures like the ready queue are being updated, we disable interrupts. We disable the interrupts during the following situations:
    - During a call to uthread_create, when the ready queue is updated.
    - During a call to uthread_join :
        - When updating the join_queue
        - When updating the finished queue and making new thread Ids available for use.
    - During uthread_exit, when updating the finished queue and checking on the join queue and updating it if required.
    - During a call to uthread_suspend and uthread_resume, when updating the ready queue.
    - While switching threads and updating the ready queue.

  We disable them so that we don’t encounter any unexpected behaviour.

f. If you have implemented a more advanced sched algorithm for extra credit, describe it.

N/A

## Test Case Documentation

There are 2 test case files - main.cpp and test_cases.cpp. In order to run main.cpp the make file needs to be edited and main.o needs to be added to OBJ. Similarly in order to run test_cases.cpp test_cases.o needs to be added to OBJ in the make file.

main.cpp takes the number of points and the number of threads as the argument.
test_cases.cpp takes the test case number as the argument.

After that run :
make
./uthread-demo <args>

**Test case 1**
Basic yield-based program with join in the end. Helps in verifying the context switch code, scheduling (round-robin), and join code. In this, we are creating 3 threads that print numbers from start to end and yield at every yield_mod iteration. Start, end and yield_mod are parameters to the function. When run with the following configuration - (1,5,2), (11,15,3) and (21, 25,4), we get the following output :

```
Threads Created
Starting thread : 100
Thread : 100 : 1
Thread : 100 : 2
Starting thread : 101
Thread : 101 : 11
Thread : 101 : 12
Thread : 101 : 13
Starting thread : 102
Thread : 102 : 21
Thread : 102 : 22
Thread : 102 : 23
Thread : 102 : 24
In main thread
Current thread : 100
Thread : 100 : 3
Thread : 100 : 4
Current thread : 101
Thread : 101 : 14
Thread : 101 : 15
Current thread : 102
Thread : 102 : 25
In main thread
Current thread : 100
Thread : 100 : 5
Exiting
```

This shows that there was interleaving between threads and after running for yield_mod number of iterations each thread successfully gave up the processor.

**Test case 2**

This test case verifies the working of join. The input to the worker threads is the same. There is one minor change - the main thread doesn't explicitly yield but waits by using join.

```
Threads Created
Joining on - 100
Starting thread : 100
Thread : 100 : 1
Starting thread : 101
Thread : 101 : 11
Thread : 101 : 12
Thread : 101 : 13
Starting thread : 102
Thread : 102 : 21
Thread : 102 : 22
Thread : 102 : 23
Thread : 102 : 24
Joining on - 101
Current thread : 101
Thread : 101 : 14
Thread : 101 : 15
Current thread : 102
Thread : 102 : 25
Joining on - 102
Exiting
```

In the above output, it can be seen that the main thread waits on 100 to finish executing. As 100 only has to print one time, it finishes before calling yield and is added to the finished queue along with moving main thread from block state to ready state. When the main thread regains control of the processor, it clears off all the memory that was allocated to 100 and then waits on 101. Similarly, after waiting on 102 and getting back control, the main thread exits out.

**Test case 3**

This test case is intended to check if the timer-based thread switching is taking place correctly.

```
Threads Created
Joining on - 100
Starting thread : 100
Thread : 100 : 1
Starting thread : 101
Thread : 101 : 11
Starting thread : 102
Thread : 102 : 21
Thread : 100 : 2
Thread : 101 : 12
Thread : 102 : 22
Thread : 100 : 3
Thread : 101 : 13
Thread : 102 : 23
Thread : 101 : 14
Thread : 102 : 24
 Finished running thread : 100
Thread : 101 : 15
Thread : 102 : 25
Joining on - 101
 Finished running thread : 102
 Finished running thread : 101
Joining on - 102
Exiting
```

In input to the worker thread is similar to test case 1 and test case 2 and so is the inner working. The only difference is that in order to simulate a large calculation, the worker thread spins on a long loop. It can be seen from the output above that context switching is taking place and the join function is also working as expected.

**Test case 4**

This test case is used to test the uthread_self, uthread_get_quantums and uthread_get_total_quantum functions.

```
Threads Created
Total quantums expected 300 library output - 300
Joining on - 100
Thread 100 uthread self and tid from the main thread are equal? True
Thread 100 uthread get quantums and quantum from the main thread are equal? True
Thread 101 uthread self and tid from the main thread are equal? True
Thread 101 uthread get quantums and quantum from the main thread are equal? True
Total quantums expected 200 library output - 200
Joining on - 101
Total quantums expected 100 library output - 100
Exiting
```

As it can be seen in the output above the APIs work as expected. In order to check on the uthread_self and uthread_quantum, we are passing the thread_id and the quantum passed to the library from the main thread to the worker function as arguments. Then inside the thread, there is an equality check. To test total quantum we do something similar in the main thread before joining and after each thread joins.

**Test case 5**

This test case is used to test the suspend and resume functionality. The worker function works in a similar way. The only difference being: it takes an extra argument telling it if it has to suspend some other thread. In this test case we are suspending a thread from a different thread and resuming it in main before joining. The output is as follows:

```
Threads Created
Joining on - 100
Starting thread : 100
Thread : 100 : 1
thread 101 is suspending thread 102
Starting thread : 101
Thread : 101 : 11
Thread : 100 : 2
Thread : 101 : 12
Thread : 100 : 3
Thread : 101 : 13
Thread : 101 : 14
Joining on - 101
Thread : 101 : 15
Resuming thread 102
Joining on - 102
Starting thread : 102
Thread : 102 : 21
Thread : 102 : 22
Thread : 102 : 23
Thread : 102 : 24
Thread : 102 : 25
Exiting
```

As we can see, thread 101 suspends thread 102 and that makes thread 102 transition to the block state. After 100 and 101 have run the main thread resumes 102 and it runs to completion.

**Test Case 6**

This test case is used to test the suspend and resume functionality. The worker function works in a similar way. The only difference being: it takes an extra argument telling it if it has to suspend some other thread. In this test case, we are suspending a thread from the same thread and resuming it in main before joining. The output is as follows:

```
Threads Created
Joining on - 100
Starting thread : 100
Thread : 100 : 1
thread 101 is suspending thread 101
Starting thread : 102
Thread : 102 : 21
Thread : 100 : 2
Thread : 102 : 22
Thread : 100 : 3
Thread : 102 : 23
Thread : 100 : 4
Thread : 102 : 24
Thread : 102 : 25
Thread : 100 : 5
Resuming thread 101
Joining on - 101
Starting thread : 101
Thread : 101 : 11
Thread : 101 : 12
Thread : 101 : 13
Thread : 101 : 14
Thread : 101 : 15
Joining on - 102
Exiting
```

As we can see, thread 101 is suspended by itself and goes into the blocked state. After thread 100 completes, it is resumed by the main thread.

**Test Case 7**

The final test case is main.cpp that was provided along with the starter code. This code calculates the value of pi. Since uthread library functions have already been tested, the only thing remaining is receiving output from the threads. As can be seen in the output of the program (successful calculation of pi), our library is able to handle this requirement.