Assignment 2 – Thread Scheduler

Operating Systems (CS416)

Rutgers University

Name: **Jishan Desai** (jpd222), **Malav Doshi** (md1378)

The project implements a user implemented API for multi-threading in C. Instead of using the POSIX thread library the project uses user implemented thread library. It has the same functions as the POSIX thread library but different implementation.

Knowing the following variables may help to understand the documentation faster:

- **CurrentThread** = thread that is currently running or the thread which called join.
- **exitQueue =** Queue which stores all the terminated threads.
- **WaitQueue** = Queue which stores all the threads who state is changed to wait state.
- **schedQueue** = Scheduler Queue. This is where currently ready to run threads are saved.
- **blockQueue** = Queue which stores the threads waiting to use the mutex.

The detail of each function and how it works is given below:

---Thread Functions--

1**. int mypthread_create(mypthread_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg)**

- **mypthread_t * thread** contains the id of the thread.
- **void *(*function)(void*)** is the function to be executed when ***thread** gets to run or join is called on that ***thread**
- **void * arg** is the arguments passed to the ***function**
- **pthread_attr_t * attr** attributes of the thread which we assume will be NULL for this project.

- **Working:** When the create function is called for the first time it initializes the **scheduler queue** [queue of threads which scheduler uses to run threads] and sets the timer by calling the **createScheduler ()** and **setTimer()** functions respectively. After that, it initializes and sets the TCB for the current thread and enqueues it in the **scheduler queue** (this occurs every time irrespective of number of times create function is called)**.** If the function is called the first time it also creates a thread context block for **main()** and enqueues it in the **scheduler queue**. After that it starts the timer.

- **Return value:** This method returns 0 on success.

2. int **mypthread_yield()**

- CurrentThread's status is changed to ready status, then it is enqueued into the **schedQueue** (Scheduler Queue). Then its current context is saved and then the context is swapped with the next thread to run from the queue (schedQueue).
- Returns 0 on success.

[We have not focused on this that much because we were told this was going to be used internally and we decided we did not need it]

3. **int mypthread_join(mypthread_t thread, void **value_ptr)**

- **mypthread_t thread** is the thread to be joined
- **void **value_ptr** is the address where **thread's (**one to be joined) return value (if any) is stored
- **Working:** Function first checks if the thread to join has already exited (by finding in exitQueue). If the thread has not exited, then function finds it from the schedQueue. Then, if the currentThread (currently running thread or the thread that called join) is NULL, it means that the scheduler is running first time. So, the only thread which can call join is main. For this reason in this case, we assume currentThread is main thread. If currentThread is not NULL, it means any thread could have join (can be main too) it's just means the scheduler has not run the first time. Now, once we know which thread is currently running, we check if the thread to join has already exited or not. If it has exited, function stores the return value (if not null) of that thread into the *value_ptr (the address which user asked to store the return value on). Then, regardless of the return value, the thread is removed from the exitQueue as we do not need it now. However, if the thread to join has not yet exited, we store value_ptr in currentThread's tcb, give the currentThread a join id (so thread to join knows who to return to when it has exited), change the status of currentThread to wait, enqueue it into the waitQueue and we change the context to next thread to run.
- **Return value:** It returns 0 on success.

4. **void mypthread_exit(void *value_ptr)**

- **void *value_ptr** is value returned the thread which is going to be terminated.
- **Working:** First it will try to get the thread waiting on **currentThread** from **waitQueue.** If the value returned is **NULL** from **waitQueue**, then this means that the thread to join has exited before the call of **mypthread_join().** So, it will set the status of **currentThread** to exit,store the value_ptr to the currentThread's tcb (so it can return it back when join is called on it) and swap context from the current thread to scheduler. If that is not the case, exit is called after join, in this case the waiting thread is woken up and if the value_ptr is not NULL, it is saved to the WaitingThread's ret_val_addr which is nothing but **value_ptr from mypthread_join. Then the waitingThread is enqueued in

the scheduler (schedQueue). **currentThread** is enqueued in the **exitQueue** and status is set to exit. And finally, the context is switched to scheduler.

- **Return value:** Does not return.

5. **int mypthread_mutex_init(mypthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)**

- **mypthread_mutex_t *mutex** a pointer to **mypthread_mutex_t** struct.

- **Working:** Initializes the **mypthread_mutex_t** struct and sets **mutex->tid = -1** and **mutex->isLocked = 0.**

- **Return value:** Return 0 on success.

6. **int mypthread_mutex_lock(mypthread_mutex_t *mutex)**

- **mypthread_mutex_t *mutex:** mutex to be locked.

- **Working:** Using **__sync_lock_test_and_set** the **mutex** is locked if It is not locked. And the timer is again started. If the **mutex** is locked and other thread tries to access it then **currentThread** is enqueued in **blockQueue** and context is switched to scheduler.

- **Return value:** Return 0 on success.

7. **int mypthread_mutex_unlock(mypthread_mutex_t *mutex)**

- **mypthread_mutex_t *mutex:** mutex to be unlocked.

- **Working:** Using **__sync_lock_release()** mutex is unlocked. If the **blockQueue** is empty, then the **currentThread** is enqueued in the **scheduler queue** and then context is switched to the next thread that is to run. Otherwise every node from **blockQueue** is enqueued in **scheduler queue** and the context is switched to **scheduler queue** so that every thread can compete for the mutex.

- **Return value:** Return 0 on success.

8. **int mypthread_mutex_destroy(mypthread_mutex_t *mutex)**

- **mypthread_mutex_t *mutex:** mutex to be freed.

- **Working:** The struct is freed.
- **Return value:** Return 0 on success.

--Library Functions-----------------------------------

9. **static void schedule()**

- This is the scheduler function which switched contexts from **currentThread** to the next thread to be run from the **scheduler queue.** If the scheduler queue is empty meaning only one thread is running then it will **setcontext()** to the **currentThread.**

10. **static void sched_stcf(struct Queue *this_queue, struct Node* node)**

- **struct Queue *this_queue** queue in which the **struct Node* node** needs to be enqueued in the order depending on the time quantum (argest time quantum is inserted at last and with least time quantum is inserted first). For our use ***this_queue** will always be **scheduler queue**. This function implements **PSJF.**

11. **tcb* getMainThread()**

- Returns the TCB of the **main()** thread from **scheduler queue**

12. **struct Node* peek(struct Queue* this_queue)**

- Returns the first **node** from **this_queue**. But this does not remove the **node** from **this_queue.**

13. **struct Node* removeNode(mypthread_t thread, struct Queue* this_queue)**

- Returns the **node** which contains the **thread** from **this_queue.** It removes the **node** from **this_queue.**

14. **struct Node* dequeue(struct Queue* this_queue)**

- Returns the head of **this_queue.**

15. **void enqueue(struct Queue *this_queue, struct Node* node)**

- Inserts **node** at the end of **this_queue.** So, the tail of **this_queue** becomes the **node.**
- Does not return anything.

16. **struct Node* dequeue_from_wait(int join_id)**

    - Returns the head of **waitQueue.** The head is also changed to the next of previous head.

17. **void enqueue_in_wait(struct Node* node)**

    - Enqueues **node** in **waitQueue.**

18. **tcb* findExitThread(mypthread_t thread)**

    - Returns the **tcb*** of **thread** found in **exitQueue.**

19. **tcb* findThread(mypthread_t thread)**

    - Returns the **tcb*** of **thread** found in **scheduler queue.**

20. **void createScheduler()**

    - **exitQueue, waitQueue, blockQueue, scheduler queue** is initialized here. Also, the scheduler context is initialized here.

21. **void stopTimer()**

    - Timer is disabled here.

22. **void startTimer()**

    - Time quantum is set here. The quantum is set to 20 milliseconds.

23. **void setTimer()**

    - The timer interrupt handler is set here.

24. **void Handler()**

- This is called when there is a timer interrupt. The function calls **schedule()** which context switches to the next thread to run.

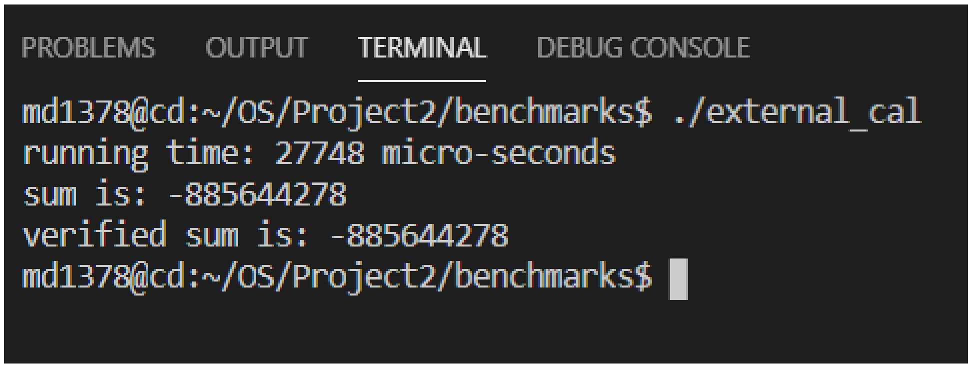
25. **tcb* initialize_tcb()**

- To create a TCB call this function which initializes the TCB and sets some initial values.

---Results--

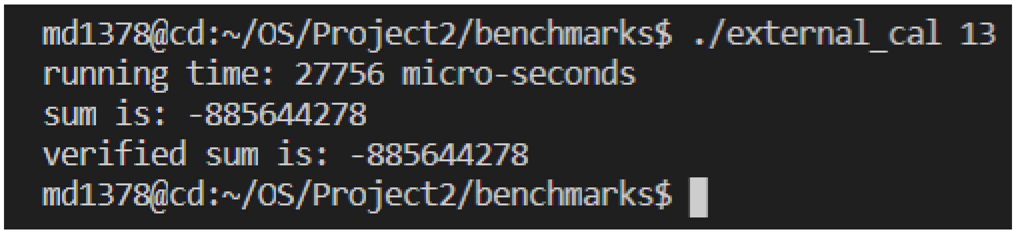
1. **External Cal:**

a.



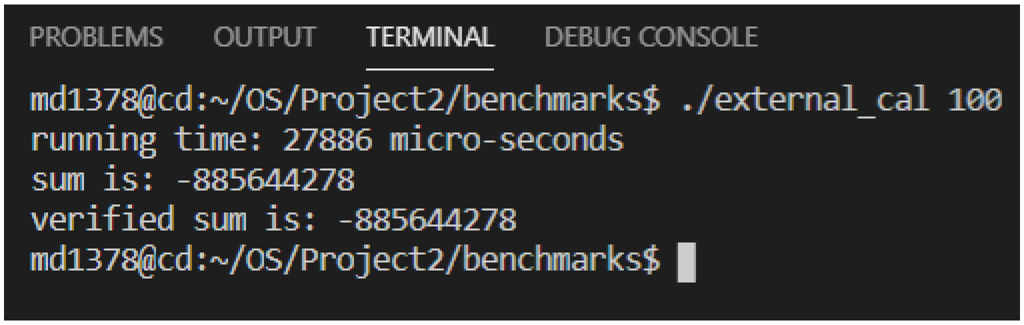
```
md1378@cd:~/OS/Project2/benchmarks$ ./external_cal
running time: 27748 micro-seconds
sum is: -885644278
verified sum is: -885644278
md1378@cd:~/OS/Project2/benchmarks$
```

b.



```
md1378@cd:~/OS/Project2/benchmarks$ ./external_cal 13
running time: 27756 micro-seconds
sum is: -885644278
verified sum is: -885644278
md1378@cd:~/OS/Project2/benchmarks$
```

c.

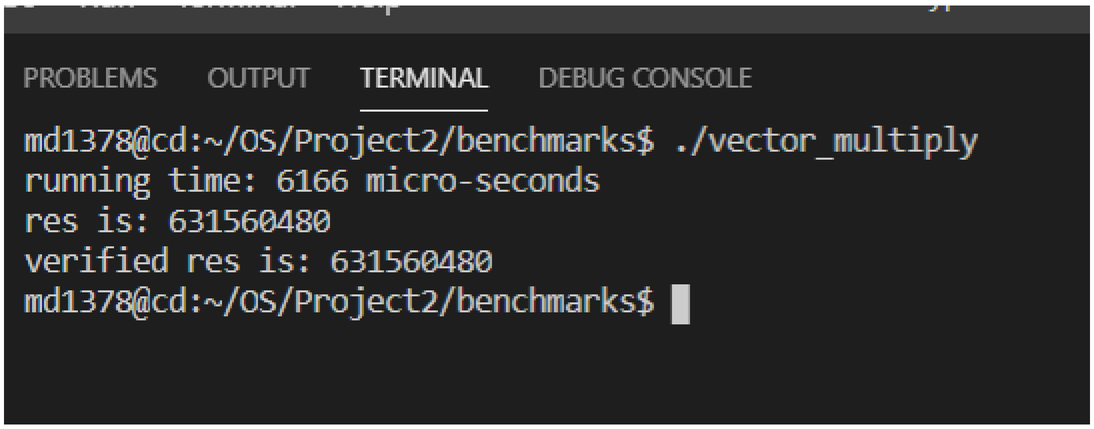


```
md1378@cd:~/OS/Project2/benchmarks$ ./external_cal 100
running time: 27886 micro-seconds
sum is: -885644278
verified sum is: -885644278
md1378@cd:~/OS/Project2/benchmarks$
```

----

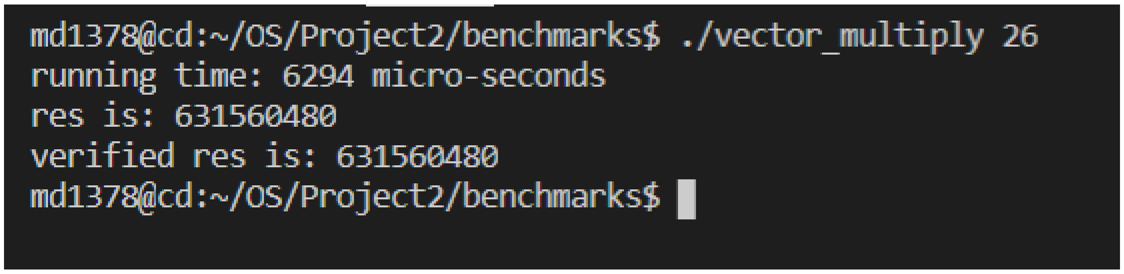
2. **Vector Multiply**

a.



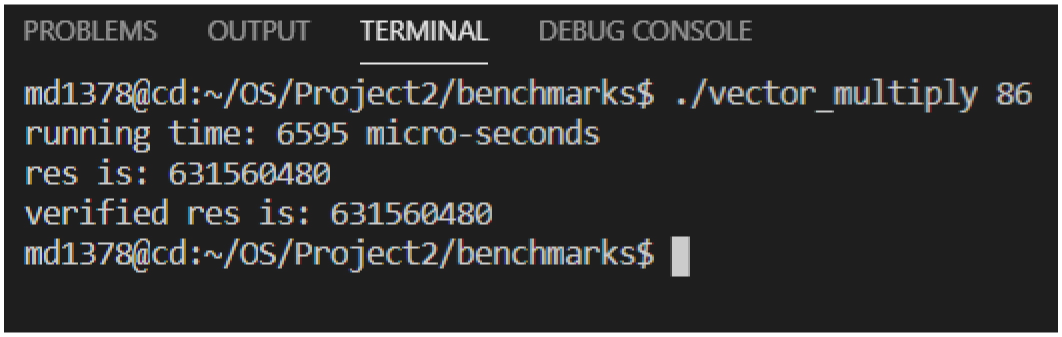
```
md1378@cd:~/OS/Project2/benchmarks$ ./vector_multiply
running time: 6166 micro-seconds
res is: 631560480
verified res is: 631560480
md1378@cd:~/OS/Project2/benchmarks$
```

b.



```
md1378@cd:~/OS/Project2/benchmarks$ ./vector_multiply 26
running time: 6294 micro-seconds
res is: 631560480
verified res is: 631560480
md1378@cd:~/OS/Project2/benchmarks$
```

c.

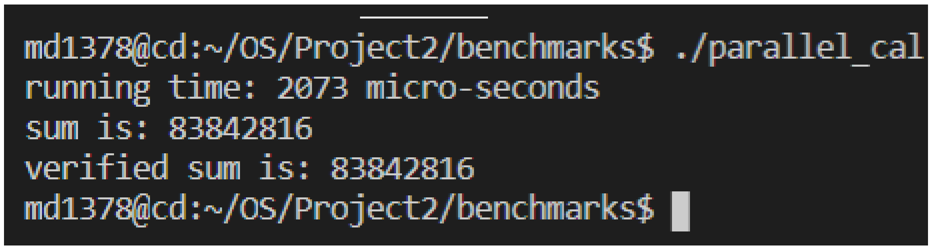


```
md1378@cd:~/OS/Project2/benchmarks$ ./vector_multiply 86
running time: 6595 micro-seconds
res is: 631560480
verified res is: 631560480
md1378@cd:~/OS/Project2/benchmarks$
```

----

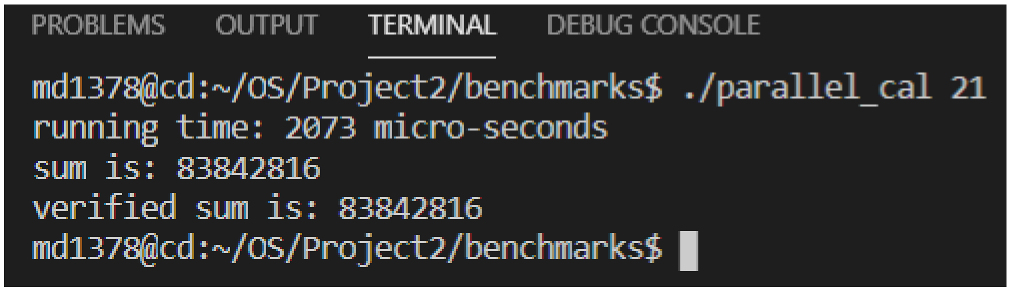
3. **Parallel Cal**

a.



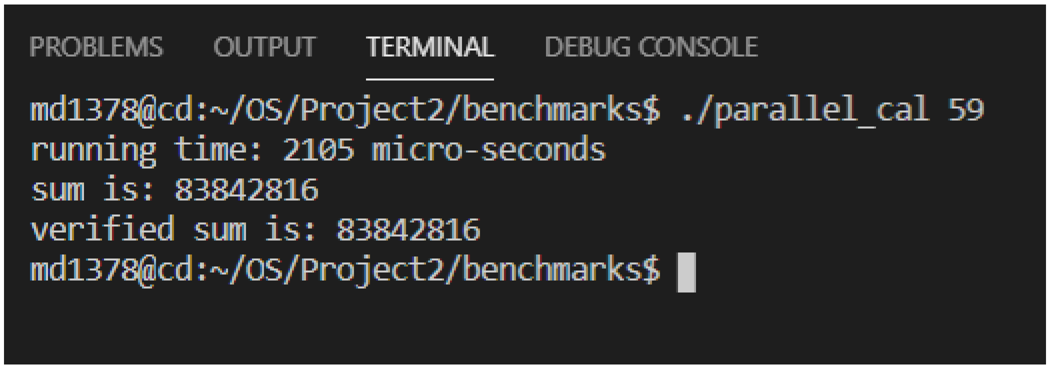
```
md1378@cd:~/OS/Project2/benchmarks$ ./parallel_cal
running time: 2073 micro-seconds
sum is: 83842816
verified sum is: 83842816
md1378@cd:~/OS/Project2/benchmarks$
```

b.



```
md1378@cd:~/OS/Project2/benchmarks$ ./parallel_cal 21
running time: 2073 micro-seconds
sum is: 83842816
verified sum is: 83842816
md1378@cd:~/OS/Project2/benchmarks$
```

c.



```
md1378@cd:~/OS/Project2/benchmarks$ ./parallel_cal 59
running time: 2105 micro-seconds
sum is: 83842816
verified sum is: 83842816
md1378@cd:~/OS/Project2/benchmarks$ 
```

For all the benchmarks, as we can see, using large threads slightly increase the time taken. However, it is not a humungous gap. So, in our opinion, it is wise to say that regardless of number of threads, the performance of the library remains the same.

- **The library takes this much time to run the benchmarks because were using queues for scheduler which are essentially linkedLists.** We could have done better in implementing the scheduler. But changing this to a min heap which would make it easy for implementing PSJF and also take less time than a Queue.

NOTE: We did not do much testing for timings for threads greater than 100 because README clearly stated that our code would be tested with 50-100 threads.

---Challenges ---

- The most challenging part of the project was implementing the scheduler. It was because understanding and getting comfortable with swapcontext() took time. And debugging errors relating to context switching was difficult because you do not know what is exactly happening in swap which is causing the particular error.