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.

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 join is called on ***thread**
- **void * arg** is the arguments passed to the ***function**

- **Working:** When the create method is called first time it initializes the **scheduler queue** and sets the timer by calling the **createScheduler ()** and **setTimer()** functions. After that it initializes and sets the TCB for the current thread and enqueues it in the **scheduler queue**. If it is the first time it also creates a thread for **main()** and enqueues it in the queue. After that it starts the timer.

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

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

- **mypthread_t thread** is the thread to be joined
- **void **value_ptr** is the place where exit status of the **thread** is stored.

- **Working:** It first checks if the **thread** is in **exitQueue** or not. If it is not found there, then it checks for it in the **scheduler queue**. If **currentThread** is **NULL** (meaning it is the first-time scheduler is running) then it will first check if the **thread** has already exited or not. If not, then it will put the **currentThread** in wait and switch contexts with the scheduler. If it has exited, then remove the **thread** from **exitQueue** and call the scheduler. If it is not the first time running then it will first check for the exit status of the thread. If the **thread** has exited, then it will remove it from **exitQueue** and call the scheduler to run next thread. If **thread** did not exit, then it will put the status of the **currentThread** in wait status and switch contexts with scheduler.
- **Return value:** It returns 0 on success.

3. **void mypthread_exit(void *value_ptr)**

- **void *value_ptr** is the variable where return value of terminating thread is stored.

- **Working:** First it will try to get the thread waiting on **currentThread** from **waitQueue.** If the value returned is **NULL** from **waitQueue** this means that the thread has exited before the call of **mypthread_join().** So, it will set the status of **currentThread** to exit and swap context from the current thread to scheduler. If exit is called after join, then the waiting thread is woken up and enqueued in the scheduler. **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.

4. **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.

5. **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.

6. **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.

7. **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---

8. **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.**

9. **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. For our use ***this_queue** will always be **scheduler queue**. This function implements **PSJF.**

10. **tcb* getMainThread()**

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

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

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

12. **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.**

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

- Returns the head of **this_queue.**

14. **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.

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

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

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

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

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

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

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

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

19. **void createScheduler()**

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

20. **void stopTimer()**

- Timer is disabled here.

21. **void startTimer()**

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

22. **void setTimer()**

- The timer interrupt handler is set here.

23. **void Handler()**

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

24. **tcb* initialize_tcb()**

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

--Results--

1. **External Cal**:

a.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

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.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

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.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

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.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

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.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

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.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

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$
```

---Overview---

- 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.

- We could have done better in implementing the scheduler. Currently we are using a Queue for scheduler. But this could have been a heap which would make it easy for implementing PSJF and also take less time than a Queue.