User Level Thread Library and Scheduler

Abhinay Reddy Vongur (av730)
Anmol Sharma (as3593)

Key structure types and global variables

Worker.h

Thread Control Block (TCB)

```
typedef struct TCB {
   worker_t id; // holds the id of the thread
   thread_status status; // status of the thread
   ucontext_t* context; // context for the thread
   char* stack; // pointer to the stack created
   uint priority; // priority of the thread
   worker_t waiting_thread; // holds any thread id waiting for its termination
   worker_t waiting_for; // holds the owner thread id of the mutex it is waiting for
   int first_schedule; // flag to indicate if scheduling the thread for first time
   struct timeval created, first_run, finished; // saves timestamps of different
events of the thread
   double time_on_cpu; // time spent running on cpu
   int mutex_blocked; // flag to indicate if blocked by a mutex
   struct worker_mutex_t* blocked_from_mutex; // refernece to the mutex it is waiting
for

} tcb;
```

Mutex structure (worker_mutex_t)

```
typedef struct worker_mutex_t {

   // YOUR CODE HERE
   int is_locked; // flag to indicate if the mutex is locked
   uint id; // id of the mutex
   struct Queue* wait_queue; // queue to hold references to threads waiting for it
   worker_t owner; // id of the thread that currently has the mutex
} worker_mutex_t;
```

Worker.c

```c
static uint thread_count = 0; // count of total threads created
static uint active_threads = 0; // number of threads that are currently active
static queue* ready_queue; // queue to hold ready threads for round robin
static queue* waiting_queue; // waiting queue to hold threads that are blocked
static queue MLFQ_queue[MLFQ_LEVELS]; // Array of queues for MLFQ
static tcb* running_thread; // reference to the currently running thread
static ucontext_t* sched_context; // scheduler context
static void *return_values[10000]; // array to hold return values
static uint mutex_count = 0; // count of mutexs created
static int interrupt_disabled = 0;// flag to indicate if signals are to be ignored
static struct timeval schedule_timestamp; // timestamp of last schedule
static double avg_turnaround, avg_response; // global variables for turnaround and
response times
static int schedule_cycles = 0; // to track the schedule cycles elapsed
```

Project Contains the following functions:

- **worker_create() :**
  - This function is used to create user threads.
  - When the function is called for the first time, for creating the first thread, a thread is created for the main/calling context, scheduler context is created and a timer is set.
  - A TCB is created for the calling thread. Memory is allocated for the stack and context.
  - The thread status is set to READY and the thread is added to the queue.
- **worker_yield() :**
  - This function can be called when a thread wants to give up the CPU.
  - When a thread calls this function, swapcontext() is called to set the scheduler context, to let the scheduler decide which thread will run next.
- **worker_exit():**
  - This function can be called by any thread looking to exit and has a value to return.
  - The function takes the pointer to return value and saves it to an array of return values.
  - Then it calls worker_yield() to set the context to the scheduler and run another thread.

- **worker_join():**
  - This function can be called any thread that needs to wait for another thread to terminate before continuing.
  - This function tries to find a thread with the given id. If not found then the id is checked against the global thread count variable to check whether join is called on a valid thread. If the thread is valid and the value pointer is not NULL then the reference to the thread's return value is copied into the value pointer and returned.
  - If a thread is found then the status of the thread that has called join is changed to BLOCKED and the thread is shifted to a waiting queue. Then worker_yield() is called to give up the CPU and let other threads run.
- **worker_mutex_init():**
  - This function can be called to initialize a mutex structure.
  - Attributes like lock status and owner thread are set to default values and memory is allocated for the mutex wait queue.
- **worker_mutex_lock():**
  - This function can be called by any thread looking to acquire any initialized mutex.
  - First, the mutex lock status is checked to see if the mutex has already been acquired by any other thread.
  - If yes, then the thread's status is changed to BLOCKED and placed on the waiting queue, and the thread is also added to the mutex wait queue.
  - If the lock is not acquired by any other thread, then the mutex is assigned to the current running thread. Lock status and the owner thread id values are changed to reflect the locked status.
- **worker_mutex_unlock():**
  - This function can be called by any thread that wants to release a mutex lock that it has acquired earlier.
  - First, the lock status and owner thread id values are checked to verify whether the thread is actually locked and it is the right thread that is calling unlock.
  - If the checks pass, then all the threads that were waiting for the particular mutex are removed from the waiting queue, status changed to READY and inserted in the queue of ready threads.
  - If the checks fail, then an error message is printed and -1 is returned to indicate the issue.

➔ **worker_mutex_destroy():**
    ◆ This function can be called when a mutex is no longer needed and can be discarded.
    ◆ The mutex's lock status is checked to see if the mutex is acquired by any of the threads. If the mutex is locked then -1 is returned and mutex will not be freed.
    ◆ If the checks pass, then free is called on any dynamic memory allocated for the mutex, and the mutex is set to NULL.
    ◆ This method is used to free the space that's been taken by the mutex and its waiting queue, it also checks if it's not been associated with any other thread.

Certain implementation details:

➔ Any thread could be in one of the 4 states :
    ◆ READY
    ◆ RUNNING
    ◆ BLOCKED
    ◆ FINISHED
    ◆
➔ Two queues are maintained,
    ◆ Ready_queue: This queue is used for threads that are in the READY state.
    ◆ Waiting_queue: This queue is used for threads that are in the BLOCKED state.
➔ A reference to the thread that is currently running will be in a global variable. It will not be in any of the queues.
➔ ITIMERPROF is used to set the timer that will fire signals after the expiry of each time slice.
➔ A flag is used to indicate if the interrupt handler has to ignore the signal.
➔ Mutex wait queue holds references to all the threads that are waiting to acquire it.

Working of Scheduler:

- ➔ As mentioned earlier, ITIMERPROF is used to set the timer.
- ➔ Wherever a SIGPROF signal is fired, the interrupt handler saves the current context of the thread and sets the scheduler context by calling swapcontext().
- ➔ Then in the scheduler, depending on the type of scheduler for the current context, separate logic is executed to schedule the next thread.
- ➔ Whenever a thread is done with execution, if any other thread was waiting for it to terminate, its status will be changed to READY and moved from the waiting queue to the queue with ready threads.

MLFQ (Multilevel Feedback Queue):

- ➔ An array of queues is created for MLFQ to have a separate queue for each level of MLFQ. A macro in worker.h is used to set the number of levels.
- ➔ A different value of time slice is used for each level. Starting with 20ms for the first level and adding additional 20ms for every additional level.
- ➔ Initially, each thread is added to the first level and priority is set to 0.
- ➔ Whenever a thread's total time on CPU on the current level exceeds the time slice of the current level, then the thread's priority is increased by 1 and is placed one level down from where it was before. If the time on CPU is less than the timeslice then it continues to be on the same level.
- ➔ To get the next thread, all the levels starting from the first are checked to see if they have any threads and the first found thread from the first non-empty level is scheduled.
- ➔ When all the threads are on the same level, it will behave like a round-robin scheduler.
- ➔ When a blocked thread comes back to the ready state, it will be added to the level corresponding to its priority before.
- ➔ To implement the Rule 5 of MLFQ, instead of tracking the actual time number of schedule cycles is being tracked. After the number of cycles crosses a threshold the counter is reset and all the threads are placed on the first level. This threshold is set to 30 and defined by macro PRIORITY_BOOST_CYCLES.

RR (Round Robin):

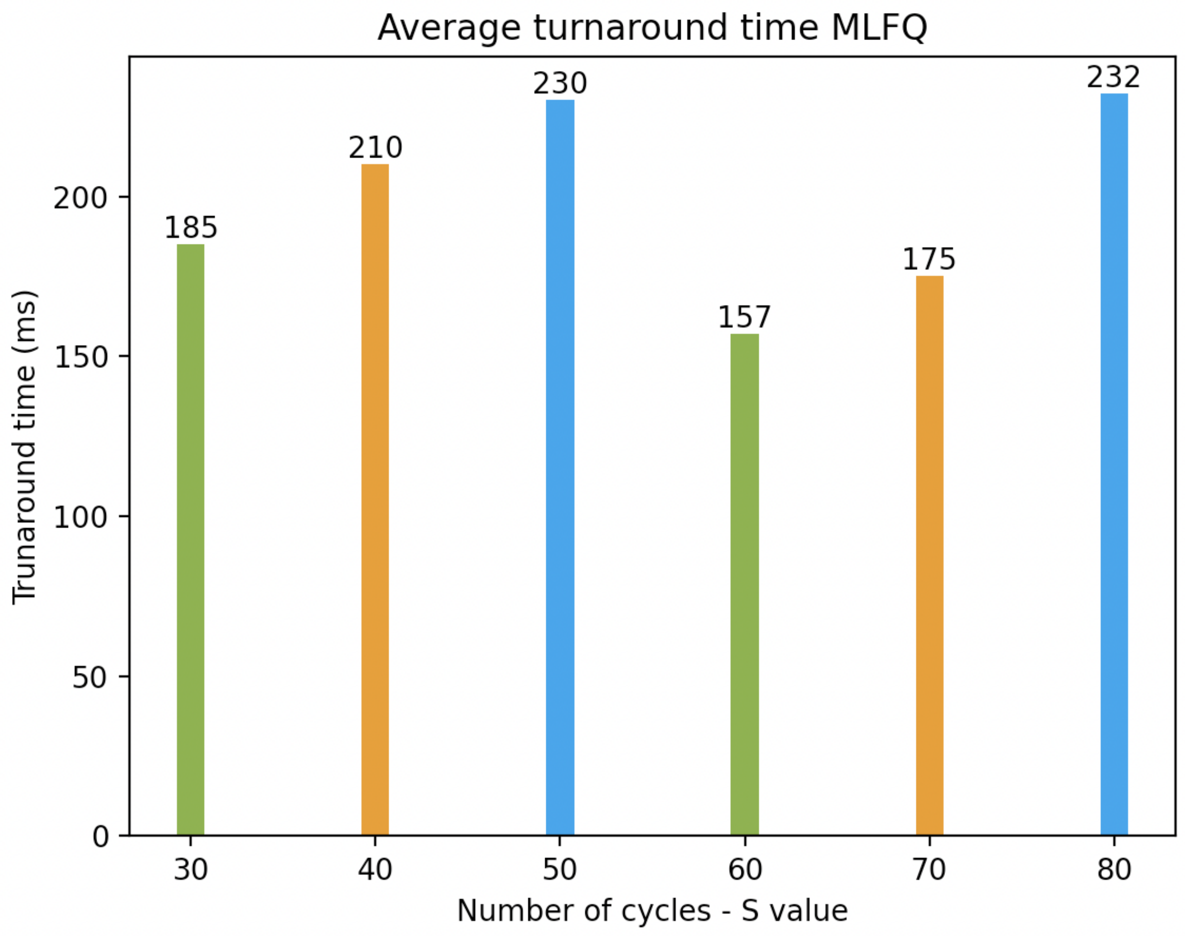
- ➔ One single queue is used to save all the threads that are in the READY state.
- ➔ The thread at the head of the ready queue is scheduled whenever the context is to be switched.
- ➔ The thread whose time slice has expired is placed at the end of the ready queue.

Results

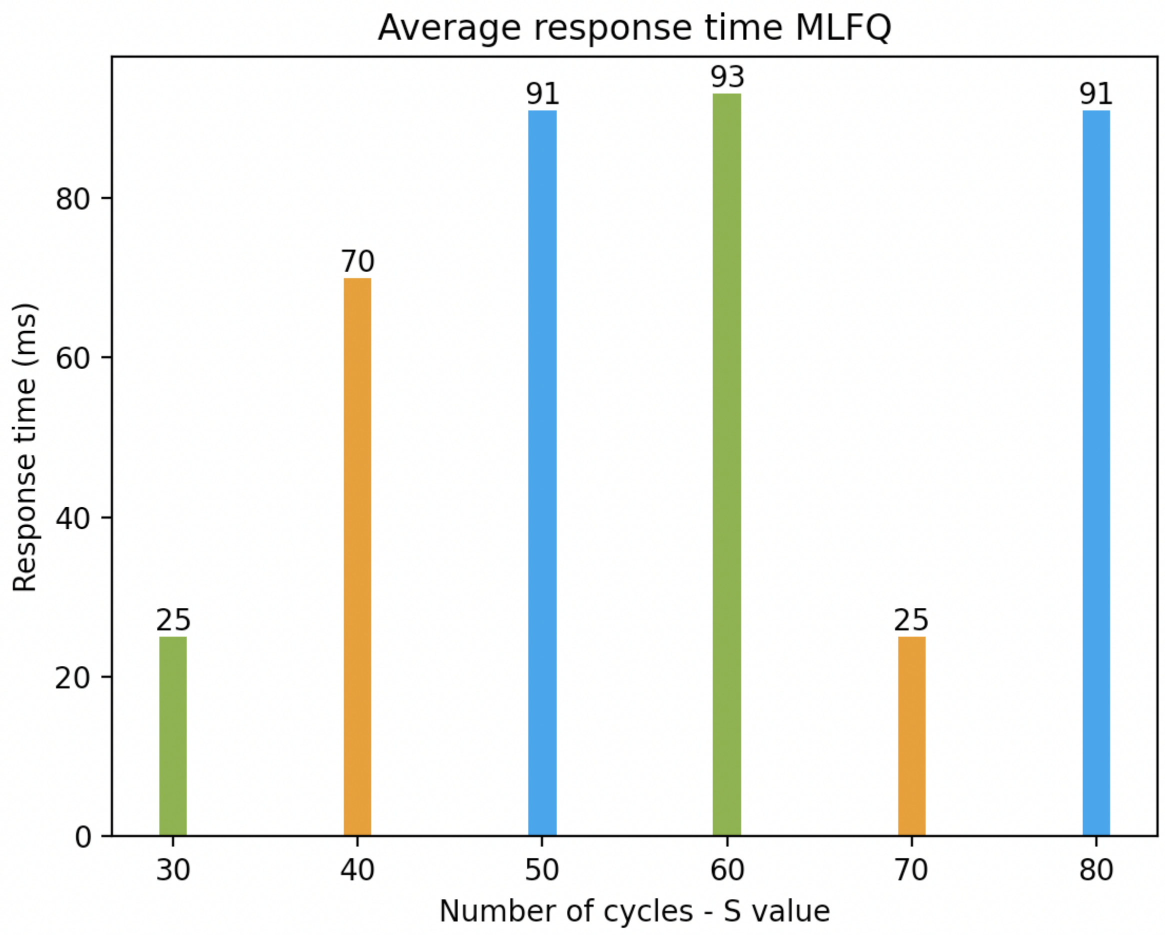
Different S values for MLFQ

Benchmark file external_cal.c was used to measure turnaround, response and run times for different values of PRIORITY_BOOST_CYCLES threshold (S value). Following results were observed.

Turnaround time

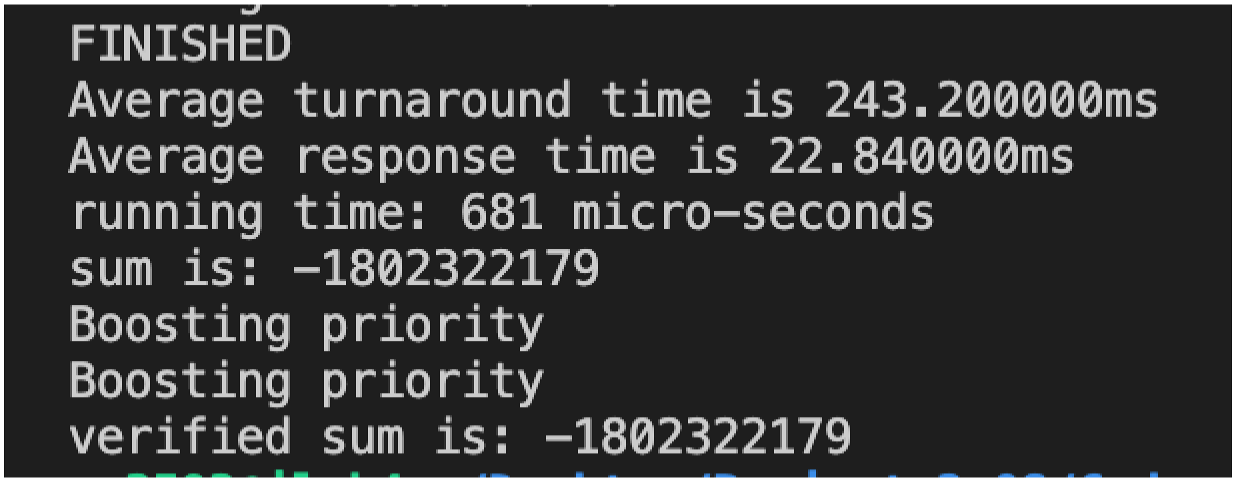


Response Time

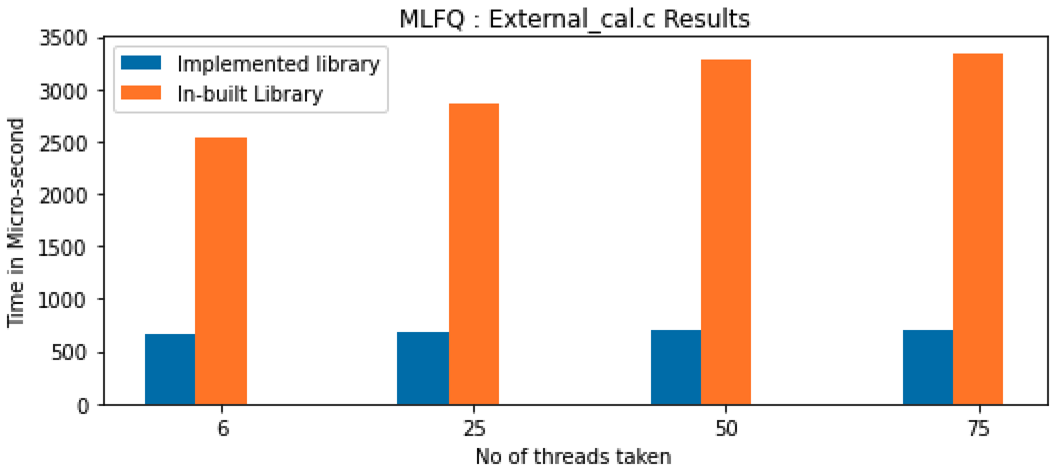


MLFQ Performance Analysis

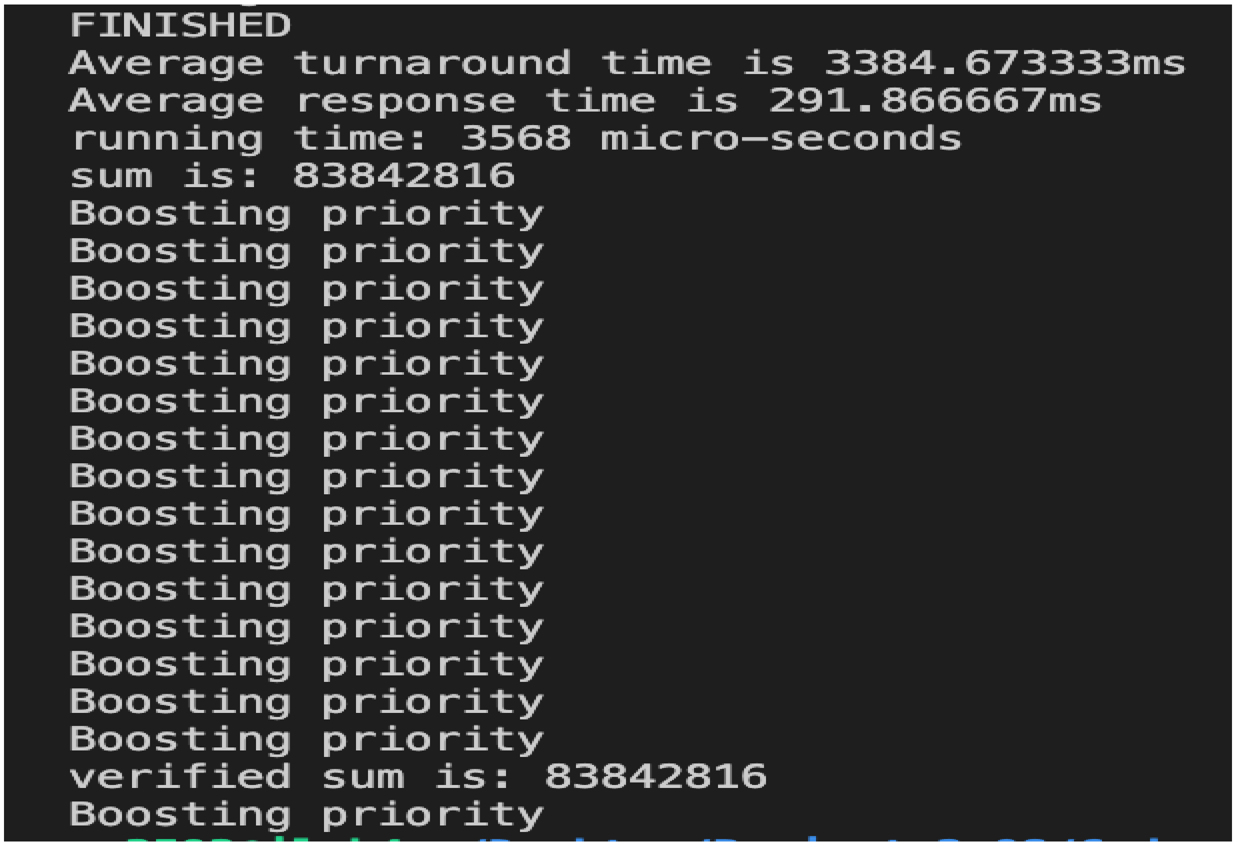
External_cal.c for 25 threads



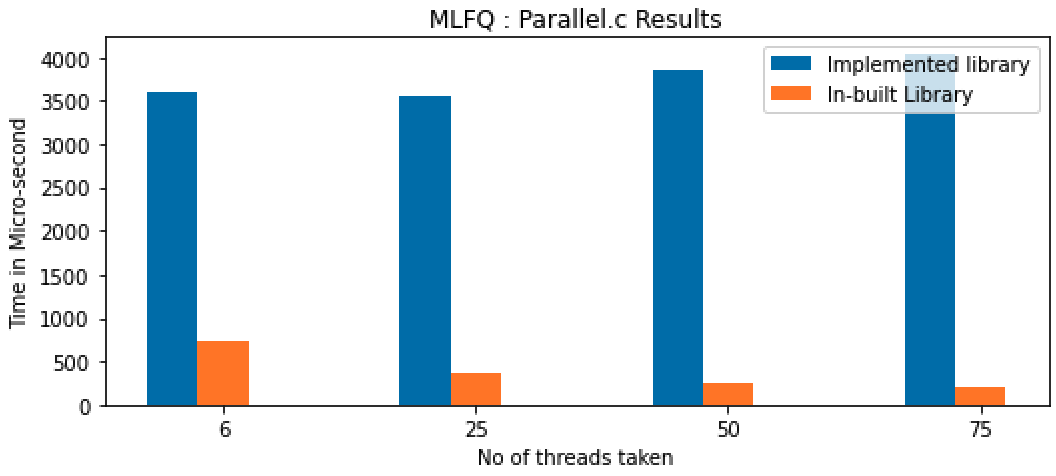
```
FINISHED
Average turnaround time is 243.200000ms
Average response time is 22.840000ms
running time: 681 micro-seconds
sum is: -1802322179
Boosting priority
Boosting priority
verified sum is: -1802322179
```



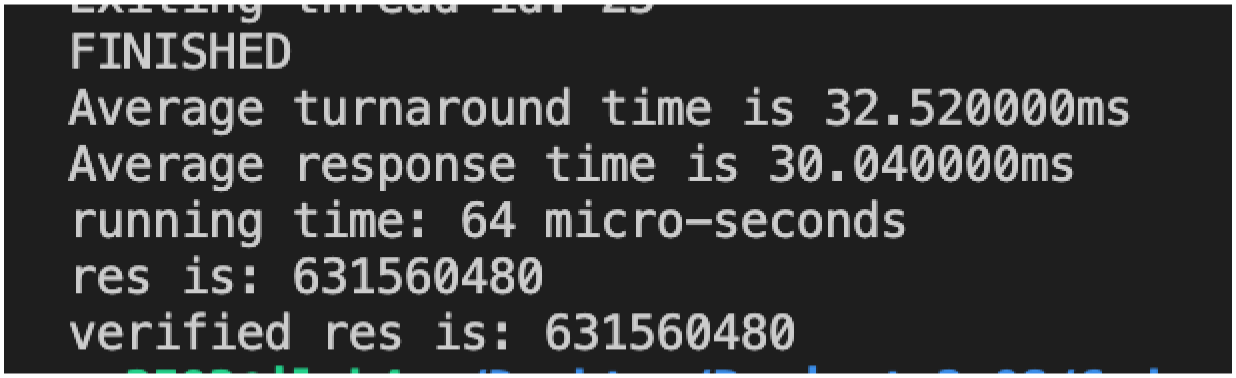
Parallel_cal.c for 25 threads



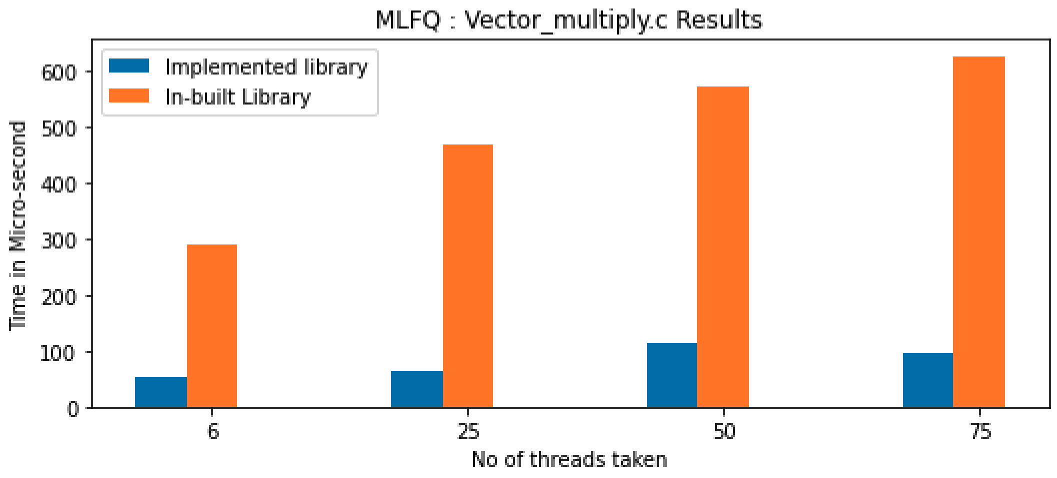
```
FINISHED
Average turnaround time is 3384.673333ms
Average response time is 291.866667ms
running time: 3568 micro-seconds
sum is: 83842816
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
Boosting priority
verified sum is: 83842816
Boosting priority
```



vector_multiply.c for 25 threads

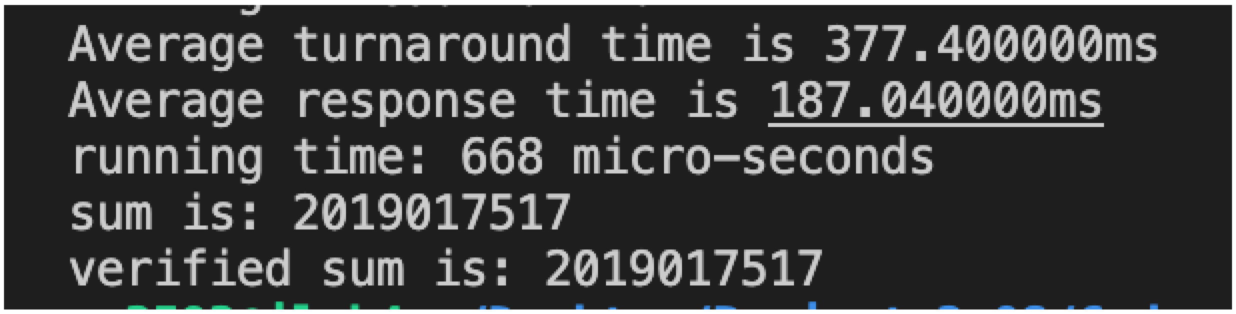


```
FINISHED
Average turnaround time is 32.520000ms
Average response time is 30.040000ms
running time: 64 micro-seconds
res is: 631560480
verified res is: 631560480
```

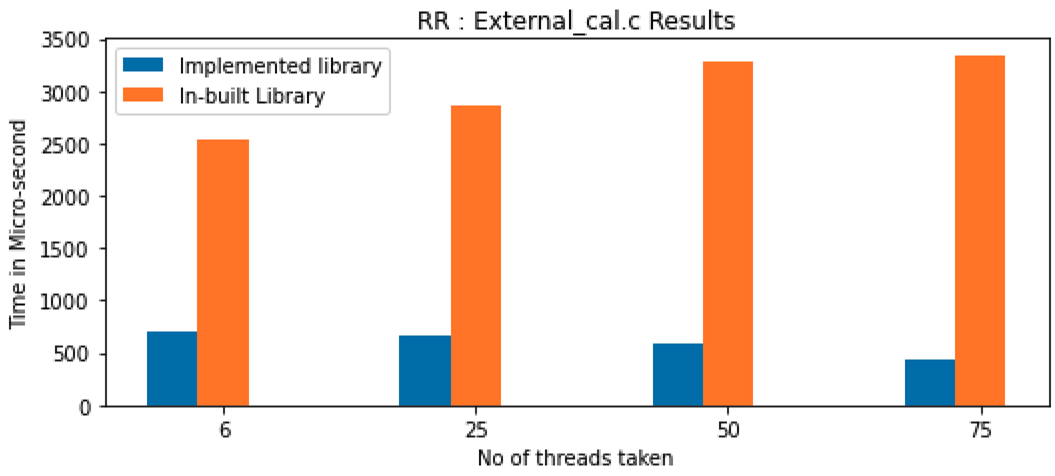


RR Performance Analysis

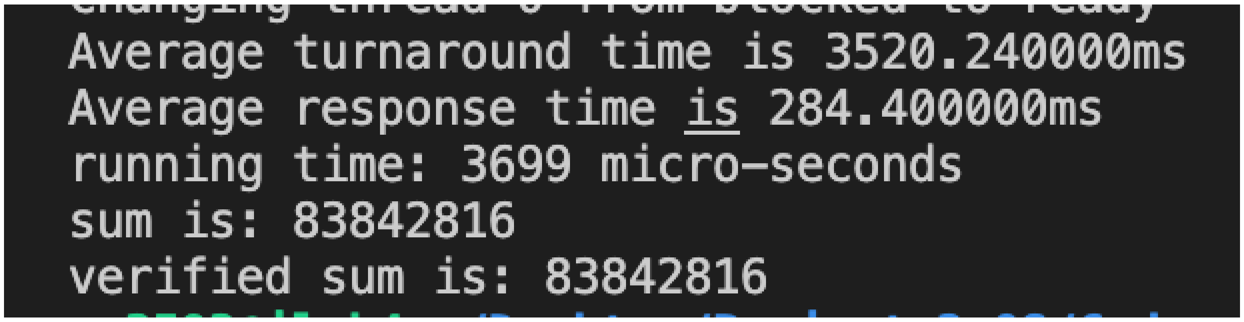
External_cal.c for 25 threads



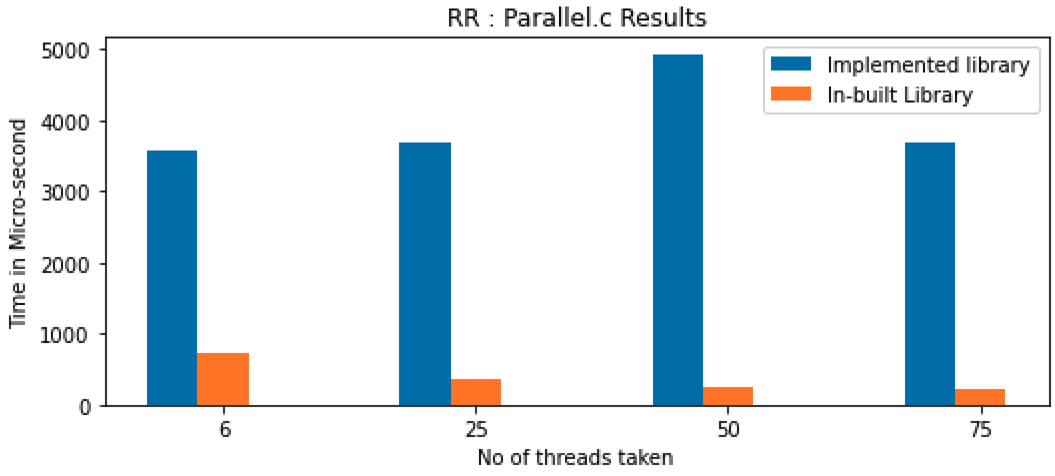
```
Average turnaround time is 377.400000ms
Average response time is 187.040000ms
running time: 668 micro-seconds
sum is: 2019017517
verified sum is: 2019017517
```



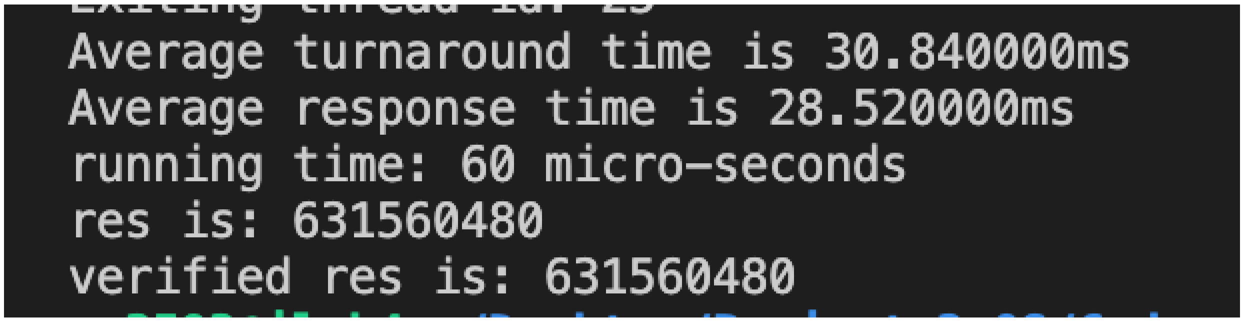
Parallel_cal.c for 25 threads



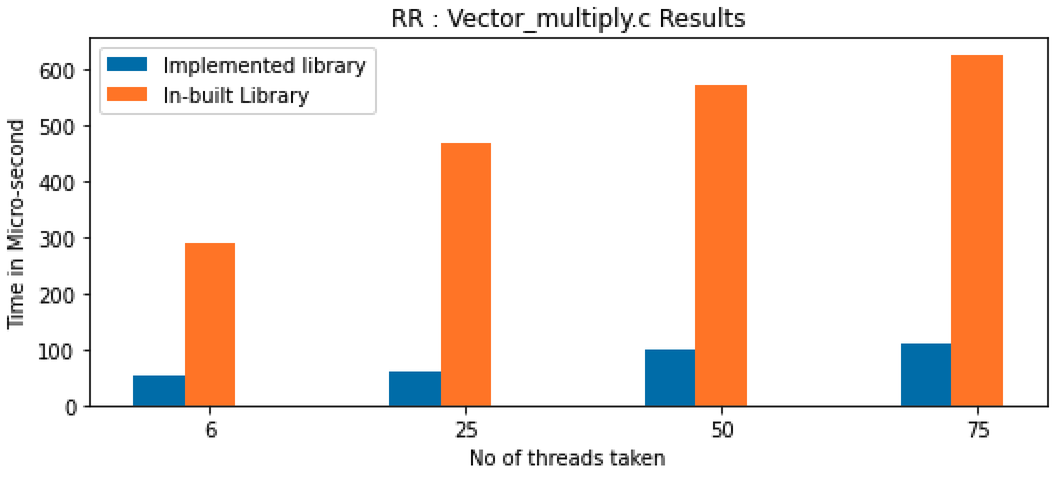
```
Average turnaround time is 3520.240000ms
Average response time is 284.400000ms
running time: 3699 micro-seconds
sum is: 83842816
verified sum is: 83842816
```



vector_multiply.c for 25 threads



```
Average turnaround time is 30.840000ms
Average response time is 28.520000ms
running time: 60 micro-seconds
res is: 631560480
verified res is: 631560480
```


RR : Vector_multiply.c Results
Implemented library
In-built Library
Time in Micro-second
No of threads taken