Abstract

This project is a user level multi threaded library written natively in C. This project emulates the native and already included `pthread` library built into C; however, this is implemented in the user space as opposed to the kernel space (which is what the `pthread` library does). Nonetheless, throughout this report, we will be detailing the inner workings of our library, how we were able to accomplish this, and interesting behaviors we discovered throughout the development of this project.

Introduction

This was our first CS416 project in which we implemented a user-level thread library using the natively built in `pthread` library as a template and guide for building our own. The interface of the actual `pthread` that is built into C, is identical to the interface we have built here.
Our implementation of the thread library consists of three parts: the **physical creation of the threads**, **mutexes and sharing having support for mutual exclusion between threads**, and **two different scheduling algorithms depending on the workload and what algorithm is more optimized for the workload: round robin scheduling, and a preemptive short job first algorithm**. (We will also detail the use of our helper functions and libraries we developed). In the coming sections, we will continue to detail the implementation, performance, and higher level abstract ideas these different parts consist of.

MyPthread Implementation

This is the crux of the library. It has all the functions and implementations that the user can call when designing multi-threaded applications. These functions define the

interface for the end client.

Threads

- `void mypthread_create()`
    - A call to `mypthread_create()` creates a thread control block (`tcb`), stores vital thread information like:
        - The thread status where 0 means ready, 1 means running, 2 means blocked, and 3 means terminated.
        - Thread ID
        - Thread Priority (only applies to preemptive shortest job first).
        - Calling Context
        - Thread Context (Used for context switching)
        - Return value (should there be a return value).
- The `void mypthread_create(mypthread_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg)` function stores all this information into the `tcb` , sets the status of the thread as READY and then enqueues it to the ready queue for the scheduler to schedule.
- The first call to `mypthread_create()` then it also initializes the ready queue, exit threads list, join list, thread count, scheduler context, and interrupt timer and effectively anything else that the thread by rely on. There is a static counter that keeps track of how many threads have been created. **Please Note: By design, we only allow for a maximum of 128 threads to be created at once. Because this library technically only runs on 1 CPU core, more than 128 threads all working and fighting for time slices on one CPU is a waste of resources. We found that 128 threads was a good limit.**
- `typedef uint mypthread_t;`
    - Stores the thread ID of the thread when client code calls `pthread_create()`
- `int mypthread_yield()`

    - A call to `mypthread_yield()` surrenders the current executing thread, changes its status to READY, adds it back to the ready queue and gives control to the scheduler to schedule other threads. This is important to us because when a thread exits, but it still has time left over to run, we do not want to waste that time. Thus, we call yield to schedule the next thread to run so we do not waste any time.

- `void mypthread_exit(void *returnVal)`

    - A call to `mypthread_exit()` terminates the currently running thread (because the user has finished working in that thread) and de-allocates the dynamic memory from when the thread was created back to the operating system. If there are any other threads waiting to `join` on this thread, when this thread exits, it will add itself to the linked list of `exit` -ed threads and unblock any threads waiting to `join` (via the scheduler) so that they can join.
    - Additionally, we also allow the user to retain the thread's return value upon completion. If `returnVal` is not NULL, that means that the user wants to store the `returnVal` and we store it in the `TCB` until the thread is joined.

- `int mypthread_join(mypthread_t thread, void **returnVal)`

    - This joins any threads that have exited and returns their return values back to the user (should there be any).
    - If the thread we wish to join has not exited yet, we will put the current thread in the blocked list and wait for the thread to exit. Once the thread exits, we de-allocate any extra heap memory the `TCB` or the thread may have created and we return the `returnVal` back to the user should there be any.
    - Please Note: Join is its own separate thread and execution context. It is scheduled to run (in order to join any blocked threads), like any other thread in this library.

Mutexes

- `mypthread_mutex_t`
  - Stores the meta data for the mutex.
- `int mypthread_mutex_init(mypthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`
  - A call to `mypthread_mutext_init()` sets the atomic flag of the mutex to 0, and it initializes the mutex's waiting queue. On success it returns 0.
  - Any thread that may be waiting to acquire the mutex will be put in the waiting queue that belongs to the mutex. Once unlocked, it will be dequeued and that thread that was waiting gets to run via the scheduler context.
- `mypthread_mutex_lock(mypthread_mutex_t *mutex)`
  - If the mutex is currently locked, the mutex marks that it is owned by the current thread ID. No other thread is allowed to access it after the mutex has been set and locked. Only the mutex owner (indicated by the thread ID) can unlock it.
  - If a user that does not have access to the lock tries to acquire the lock, they will be put in the waiting queue until the mutex finally unlocks.
  - This queue depends on whether the library was compiled with the round robin or preemptive shortest job first scheduling algorithm.
- `int mypthread_mutex_unlock(mypthread_mutex_t *mutex)`
  - A call to `mypthread_mutex_unlock()` tries to unlock a mutex lock, if the mutex is already locked by another thread, then the current thread will not be able to unlock the mutex.
  - If the `mutex` gets unlocked, then the next thread waiting in the mutex wait queue will be pushed into the ready queue and be scheduled to run.
- `int mypthread_mutex_destroy(mypthread_mutex_t *mutex)`
  - This will destroy the mutex and free all memory allocated by the mutex, including the mutex waiting queue.

Scheduler Functions

- `void swapToScheduler()`
  - Is used to swap into the scheduler when a signal is caught. We use this

function as a signal handler and because it provides quick access for our current interface to save the context and context switch into the scheduler. Having this function allowed us to avoid messy, spaghetti code when doing the context switch into the scheduler.

    - `void setupTimer()` (Only apples for Round Robin scheduling)
        - Sets up a timer that counts from `QUANTUM` microseconds to 0. When the timer eventually reaches zero, we catch either a `SIGALRM` or a `SIGVTALRM`. Once either of these signals is caught, we call `swapToScheduler` to run the next thread. The signal catchers are initialized here too.

- `void restartTimer()`
    - Restarts the timer from the time quantum for the process.
- `void stopTimer()`
    - Stops the current timer and sets it equal to zero.
- `void scheduler_interrupt_handler()`
    - A function that serves as a stepping block for reaching either reaching the Round Robin scheduler or the Preemptive Shortest Job first scheduler.
- `static void schedule()`
    - If the current `TCB` is null (it was dequeued just as the scheduler was called), then we get the next `TCB` in the in the ready queue and schedule it to run, or we go into the scheduler with the current TCB and decide if it should continue running.
    - This function simply decides what the current `TCB` should be before we call the primary scheduling functions (Round Robin, Preemptive Shortest Job First)
    - Scheduler Context starts here.
- `void create_schedule_context()`
    - Called at the beginning of the program. Initializes and sets the scheduler

context so we can return to it at any point during the execution of the program.

- `static void sched_RR()`
    - Implements the Round Robin scheduling algorithm. If the time quantum has expired, we enqueue the current thread to the end of the current ready queue and we set the next thread in the queue to run until its time quantum has expired.
    - If the current thread is new and is ready to run (has not been given any run time yet), then it is set to run as the currently executing thread.
- `static void sched_PSJF()`
    - Scheduler algorithm for preemptive shortest job first. The current job will remain executing to completion until a new job with a higher priority comes around. If a new job with a higher priority is added to the ready queue, that job will run and the current job will be placed on the ready queue. The job with the highest priority will run until the end of the thread.

Libraries

Here, we will detail the implementation of the libraries we used.

checkMalloc

- `checkMalloc(void *ptr)`
    - This function will check if `malloc` has failed (which is very unlikely, but still does happen). In such an event where `malloc` has failed, it will exit with code `1`.

Queue

Queue is implemented via a linked list. We have the option of doing a `priorityEnqueue` where the `TCB` is stored in the queue based on the priority of the the thread. `normalEnqueue` and `normalDequeue` which act as normal `enqueue` and `dequeue` functions. This queue is implemented based on a linked list. Depending on the selected scheduling options decided by the client (Round Robin or Preemptive Shortest Job First) these functions were primarily used in the ready queue when enqueuing and dequeuing threads from the ready queue.

- `struct Queue *initQueue()`
  - Initializes the queue and returns a power to the already made initialized queue.
- `bool isEmpty(struct Queue *queue)`
  - Checks to see if the queue pointed to by `queue` is empty. Returns true if it is.
- `void normalEnqueue(struct Queue *queue, tcb *threadControlBlock)`
  - Enqueues the `threadControlBlock` to the end of the specified queue.
  - Only used for round robin scheduling.
- `tcb *normalDequeue(struct Queue *queue)`
  - Dequeues the `TCB` at the very front of `queue` and returns it back to the user. The `TCB` is them removed from the queue.
- `void priorityEnqueue(struct Queue *queue, tcb *threadControlBlock)`
  - Enqueues the `threadControlBlock` into the queue based on the `threadControlBlock` 's priority. A higher priority (the lower the number) gets placed as the front of the queue while lower priorities (higher the number) is placed at the end of the queue. This function can insert into the beginning, middle, and end based on the queue. This queue is always ordered based on priority.
  - Only used for preemptive shortest job first.
- `void freeQueue(struct Queue *queue)`
  - Frees the queue, cleans up any and all dynamic memory made by the queue or when initializing it, and frees the queue structure (the container) for the queue. Returns when complete.

Linked List

Used for the `exitedThreads` list and for the `joinList`. We decided to use linked lists for this part because there could have been an arbitrary number of threads waiting to exit or join. Managing that memory in the form of an array and then shifting elements forwards and backwards was a nightmare to code as it is very easy to have memory or logical bugs when designing dynamic arrays. Thus, we decided that a linked list would be easier to understand and implement.

Our linked list behaves normally as any linked list would.

- `struct LinkedListNode`
    - Contains the data that will be in the node.
    - Contains a pointer to the next node.
- `struct LinkedList`
    - Wrapper structure for the linked list.
    - Stores the head and tail of the list.
    - Also stores the current size.
- `bool isListEmpty(struct LinkedList *linkedList)`
    - Checks to see if the list is empty. If it is, return true. Else, return false.
- `bool insert(struct LinkedList *linkedList, void *data)`
    - Inserts a node at the end of the linked list. `data` will contain the `TCB` that we wish to store.
- `void *delete(struct LinkedList *linkedList, void *data)`
    - Deletes the `data` if it exists in the linked list. If it does not, we return null.
- `size_t getSize(struct LinkedList *linkedList);`
    - Gets the size of the `linkedList`.

Time Quantum (And Benchmarks)

To benchmark how different time quantums perform when using the Round Robin scheduling algorithm, we decided to use the built in benchmarks and try a series of different time quantums from low to high. We noticed that smaller time quantums were generally better in our testing. Below are our results from testing on rlab2.cs.rutgers.edu under moderate CPU usage by other Rutgers Students. The order of our benchmarks are `parallel_cal`, `external_cal`, and `vector_multiply` respectively. Below are our results.

- 5 milliseconds time quantum:

```
running time: 2973
micro-seconds sum is:
83842816 verified sum is: 83842816

running time: 904 micro-seconds
sum is: 926915936
verified sum is: 926915936

running time: 166 micro-seconds
res is: 631560480
verified res is: 631560480
```

- 6 milliseconds time quantum:

```
running time: 2963 micro-seconds
sum is: 83842816
verified sum is: 83842816


running time: 893 micro-seconds
sum is: -296377839
verified sum is: -296377839


running time: 167 micro-seconds
res is: 631560480
verified res is: 631560480
```

- 10 milliseconds time quantum:

```
running time: 2966 micro-seconds
sum is: 83842816
verified sum is: 83842816


running time: 889 micro-seconds
sum is: 1019353840
verified sum is: 1019353840


running time: 168 micro-seconds
res is: 631560480
verified res is: 631560480
```

- 25 milliseconds time quantum:

```
running time: 3018 micro-seconds
sum is: 83842816
verified sum is: 83842816


running time: 904 micro-seconds
sum is: 371398526
verified sum is: 371398526
```

```
running time: 166 micro-seconds
res is: 631560480
verified res is: 631560480
```

- 50 milliseconds time quantum:

```
running time: 2962 micro-seconds
sum is: 83842816
verified sum is: 83842816


running time: 891 micro-seconds
sum is: 314477131
verified sum is: 314477131


running time: 165 micro-seconds
res is: 631560480
verified res is: 631560480
```

- 100 milliseconds time quantum:

```
running time: 2970 micro-seconds
sum is: 83842816
verified sum is: 83842816


running time: 888 micro-seconds
sum is: 92630565
verified sum is: 92630565


running time: 161 micro-seconds
res is: 631560480
verified res is: 631560480
```

- 150 millisecond time quantum:

```
running time: 3014 micro-seconds
sum is: 83842816
```

```
verified sum is: 83842816


running time: 895 micro-seconds
sum is: 1486106690
verified sum is: 1486106690


running time: 167 micro-seconds
res is: 631560480
verified res is: 631560480
```

Based on our results from our benchmarks, we can see that as the time quantum got absurdly large, we started to see diminishing returns as at some point, it becomes less round robin and more about shortest job first (when talking about high quantum). Thus, a time quantum in the middle, towards the lower side is better. A short, but long enough time quantum, in our results, proved to be the best. **The 6ms time quantum proved to perform the best out of all other time quantums in the workloads that were provided.** While optimizations can definitely be made based on the workload, as we see here, the 6ms time quantum gave us the best results on `rlab2.cs.rutgers.edu`. Different hardware, system settings, and system load at any one point can greatly affect the overall results, however, in multiple tests runs, we were able to conclude that a 6ms this was the fastest.

The 6ms Time Quantum in Our Case Proved To Provide The Best Results

Auto Benchmark Tool

To make bench-marking easier, we developed an automatic bench-marking tool to run all the benchmarks all at once and output the results. The tool is located in our `src` directory. It is entitled, `autobenchmain`. To run the tool, you must execute it as a script: `./autobenchmain` after navigating to the appropriate directory. This will compile the entire project and benchmark the `mypthread` library we have developed. This was an automation tool we felt would help the graders as it helped us save lots of time.

Running The Project

To compile the library into an archive, please navigate to the `src` folder. From there, the library should compile with `make`. To run the benchmarks, please navigate to the `benchmarks` directory, and type `make` to compile the benchmarks.

Extra

This project was tested on `rlab2.cs.rutgers.edu`

Authors

This project was completed for the undergraduate class CS416 at Rutgers University with Professor Francisco. As a part of the project, we were allowed to have three members per group working on this project. The people who worked on this project are:

- Hasnain Ali (NetID: ha430)
- Rushabh Patel (rsp155)
- Della Maret (NetID: dm1379)

Conclusion

This was a custom user level `pthread` library we developed to practice the concepts of multi-threading and parallel programming and gain a deeper understanding of CPU thread scheduling and how context switching occurs at a low level from the perspective of the operating system. We implemented two scheduling algorithms, round robin and preemptive shortest job first. With Round Robin, we noticed that given our workload,

computer hardware, and benchmarks, the best time quantum was 6ms. This project was very insightful and gave us a deeper understanding into threading, contexts switching and saving, and the overwhelming performance benefits that multi-threading provides.