Macauley Pinto, Itamar Levi, Ryan Townsend

NETID: msp194, il166, rrt51

01:198:416

Professor Kannan

2 March 2018

Abstract

The user-level thread library and scheduler are a pure user-level implementation of a thread library that mimics the pthread library in C to develop a multi-threaded environment. Thus, the following project design incorporated both a Preemptive SJF (STCF) and Multi-Level Feedback Queue as schedulers to delineate the complexities of developing a thread as well as the overhead in design. The first section labeled pthread will discuss the pthread functions developed. The second section, mutex, will discuss the mutex design. The third section gives a brief overview of the timer interrupt before proceeding to the fourth and fifth section labeled Preemptive SJF and Multilevel Feedback Queue (MLFQ) that discuss the designs of both schedulers respectively. Finally, the section labeled benchmark analysis will discuss the benchmark test when compared to the actual pthread library.

Pthread Functions

The my_pthread_create() function takes in four parameters to create a thread. In order to make the function create a thread, use of a thread control block, or TCB, helped allocate memory to help save a threads state along with several flags used in subsequent helper functions. Memory is first allocated for the TCB and the TCB has a ucontext_t type for the thread's context. This helped provide a method for determining the threads context at a given state in the

program. Since processes are isolated, but threads share certain resources, it was imperative to ensure the threads would be created within the process. This would allow for open file descriptors, signal handlers, and resources allocated to be accessible amongst all threads.Furthermore, several flags are initialized in the TCB to help indicate various points in the program during a threads run-time. Among these flags are priority for the STCF scheduler, elapsed for Multi-Level Feedback Queues, and status to indicate the threads point in the program (ready,running, blocked, or completed). The function and arguments passed as parameters to the pthread_create() method are stored in the TCB for later use in the threadWrapper() function to help with storing return values from the function parameter passed to create. Each thread is finally enqueued to a global queue data structure with a status labeled as ready. Furthermore, upon thread a creation, a timer interrupt is initialized for scheduling in the program—although the timer execution is dependent on a join call before initializing all created threads into a queue.

The function my_pthread_join() is typically called after a create call to wait for a thread's completion before finally setting a return value in the second parameter. Thus, the program is dependent on a join call to initialize threads as without a my_pthread_join() call, the created threads would exhaust system resources. As a result, it was imperative to only begin scheduling once a join call was made in the user-level thread library. This implementation first sets a global join value to one to indicate a join call has been set and scheduling can begin without issue. The context of the join call is also saved before executing a join calls while loop to resume execution once the joining thread ID has been requested. Once the join call is made and no threads are scheduled, the program begins by invoking my_pthread_yield() to allow the CPU to schedule threads according to the requested scheduling policy. Once all threads have been created,

scheduled, and exited the join context will later be restored to begin returning the value of the function the respective thread accessed before cleaning thread resources.

The function my_pthread_yield() is dependent on the scheduling policy used to execute threads. In both the scheduling policies, the my_pthread_yield() will always schedule the first thread without swapping, getting, or setting contexts of the thread to the scheduler. This choice was made as the first thread created should not suffer from slow runtime performance due to multiple context switches. Thus, to design a fast, robust barebone thread algorithm, scheduling the first thread without respect to policies was a critical design decision that helped improve runtime performance over the actual pthread library in terms of one thread of execution when analyzing benchmark performance as featured in the table below. However, it is important to note any subsequent treads created would begin scheduling by swapping contexts. Furthermore, once the interrupt handler begins, it invokes my_pthread_yield() to mark any incomplete thread that was running to ready to and saves the threads context before late swapping back to the scheduling context in the subsequent created threads.

The function my_pthread_exit() is one of the most critical function for deallocating resources utilized by a thread while also update the main thread queue. The implementation of this function in particular required ensuring the primary thread queue would reflect an accurate representation of the status of all created threads. Thus, in order to ensure accuracy in the main thread queue, the timerInterruptHandler() function is invoked as this is the safest point to manipulate values during code runtime as program execution is halted until leaving the interrupt handler. The decision to directly invoke the interrupt handler was due to issues when using usleep() to wait for the handler to be executed as per the 100ms that was set in the code. The

reason for using a 100 ms time quantum was to ensure threads are not interrupted too frequently or slowly as using too little or high of a time interval would result in improper thread scheduling due to the contexts swaps and value set throughout the runtime of the library. Furthermore, a direct invocation of the handler over usleep() allowed for significantly faster runtime performance. Thus, before a threads resources are deallocated and set to null, the interrupt handler will remove the thread from the primary thread queue before finally setting the context back to the my_pthread_exit() function. Flags are set in the variable waitingToJoin to indicate which aspects of my_pthread_exit() should be invoked to avoid segfaults in the exit function. The threads return value is also stored in a global void pointer variable label as returnValue before setting the context back to the initial join invocation. Once back in the method join, the returnValue is stored in the double void pointer and the finally resets the return value to null before returning back to the original my_pthread_join() call to wait for the next thread invoked by my_pthread_join().

Although the function, threadWrapper is not a my_pthread function, the function was critical to program success and is directly responsible for the success of the program. The thread wrapper finds the running thread to execute and later invokes the thread's function by the arguments initially set in the my_pthread_create function. As a result, the thread is able to also save the return value from the function once set and marks the threads status as complete. This is to ensure, that any subsequent calls to the thread cannot be made upon thread's status marked as complete. Finally, the my_pthread_exit() call is invoked with the node of the thread as a the parameter. This allows for direct deallocation of the thread as the return value indicates completion of the thread's execution and indicates resources no longer need to be exhausted.

Mutexes

We chose to represent the all of the mutexes initialized within a program as a linked list of mutex nodes, with each node representing a unique mutex. The state of the mutex is represented by the fields of its mutex node. The integer ID field is unique to each mutex, and used for locating a mutex within the linked list. We represent the lock/unlocked state of the mutex with a volatile integer flag. We use a volatile integer because in order to best model a mutex's thread-safe nature, as we, we would have to be able to utilize atomic built-in functions, specifically '__sync_lock_test_and_set'. Each mutex contains a thread control block, called holder, which represents the thread that is allowed to unlock the mutex, as well as access the critical section of data protected by the mutex. The field blockedList represents a list of threads which have attempted to access the critical section while the mutex was already locked by another thread. These threads enter a blocked state, and are appended to blockList. The final field front, simply denotes the front of the blockedList, and is used for simpler access to the blockedList.

The function my_pthread_mutex_init() takes a pointer to a declared mutex, and initializes its fields to their default values. It then adds this mutex to the list of initialized mutexes. The function my_pthread_mutex_lock() takes a pointer to a mutex as its only argument. If the mutex flag is not locked, the thread that invoked the lock function is denoted the holder of the mutex, and the mutex enters a locked state using __sync_lock_test_and_set(). We use test and set because atomic builtin functions are the best way to represent the "randomly" changing state of the mutex. A mutex must be able to update its state as close to instantly as possible, otherwise

there would be little point to using a mutex. If a thread attempts to lock a mutex which is already locked, it is appended to the mutex's blockedList, enters a blocked state, and yields its CPU time. The function my_pthread_mutex_unlock() checks whether the mutex is locked, and whether thread attempting to unlock the mutex is that mutex's owner. If both conditions are satisfied, the mutex unlocks via __sync_lock_test_and_set(), setting the mutex's flag to 0. The threads held in a blocked state in mutex's blockedList are all removed, marked as 'ready', and re-enqued for scheduling. The function my_pthread_mutex_destroy() deallocates all memory used by the mutex node, and removes it from the list of initialized mutexes.

Interrupt Handler

Before discussing the design decisions in scheduling, it is important to note the architect responsible for accurate scheduling—the timer interrupt. Although previously discussed, the interrupt handler requires more background information in order to understand the complexity in design in both preemptive SJF and multilevel feedback queue scheduling. The interrupt occurs every 100 ms to set execution to the interrupt handler which is primarily responsible for handling several areas of thread management. Among these areas are updating and scheduling sensitive data. Although blocking the interrupt may seem as a proper method for handle data sensitive synchronization when yielding or deallocating resources, the issue with blocking an interrupt is the inherent unknown process state when blocking the interrupt as the scheduling can greatly suffer from a lack of proper program awareness due the interrupt block. Furthermore, the interrupt is also useful exiting the program once no other threads exist in the the primary thread queue along with other benefits such as maintaining program coherency when scheduling threads.

Scheduler: Pre-emptive SJF

Preemptive scheduling became arduous to implement when considering the priority in which threads must be scheduled along with updating the primary thread queue. Thus, apart from updating a thread in the interrupt handler, it was also important to main contexts in both the scheduler and thread of execution. Thus, the scheduler context is saved upon the first thread's scheduling before later setting the context of thread to its most recent execution point. This also allowed for proper updates of thread contexts and states of execution during runtime. Once, the thread's context is set, the thread begins execution in thread wrapper. After maintaining the resources allocated during the threads execution in the context stored in the thread's TCB, the interrupt handler will begin to yield to the next thread if the former running thread did not complete execution. Furthermore, the scheduler runs in a while loop until the completed threads variable is equivalent to the allThreadsCreated variable.

Scheduler: Multi-Level Feedback Queue

When implementing a 4-level multi-level feedback queue, there were many changes that had to be implemented in the code to accommodate for this new structure and new scheduling method. This method required an array of linked lists to be made, where the topmost level queue would have the highest priority and would always run the threads contained in it first, and the lowest level queue would be run only if the above level were all empty and contained no running threads. In order to implement this, the time elapsed for each thread had to be tracked and monitored, as each level required a thread to run for a specific amount of time respective of the queue level it was in. These times were tracked in the my_pthread_yield function, where the times tracked were 50 milliseconds, 100 milliseconds, 150 milliseconds, and 200 milliseconds

from the highest priority (queue level 0) to the lowest priority (queue level 3) respectively. A thread would be dequeued from the highest priority that was in the multi-level queues and would run a function. On completion, it would be removed from the queue and set to complete. If the thread exceeded the timeslice allowed at its current queue level, it would be placed into a lower priority queue, where it would run for a longer period of time only after any threads in the higher priority queues would run as well. In order to deter from starvation in the queues, every 100 iterations of 100 milliseconds caused by the timer interrupt would result in all of the threads being placed into the topmost level, queue level 0.

Benchmark Analysis

Benchmarking provided direct understanding into the performance of the pure user-level thread library in relation to the pthread library (See Tables Below). Most interestingly, the pure user-level thread library was better in certain areas when compared to the pthread library. This is most notable in vector multiply in which user-level thread significantly outperformed the pthread library as the threads began to increase in count. However, it is important to not this performance increase was not nearly as good as the pthread library in parallelCal and externalCal in which the performance of both executables was only comparable when using one thread. This is most likely due to the pthread library making better use of a threads scheduling in the more intensive context switching operations. The user-level library suffers from a lack of robust interfacing between the user and system as the pthread library is designed with several libraries that are dedicated toward maintaining thread coherency and performance while the user-level library was limited in robustness of libraries that can be used to optimize performance.

Benchmark Tables

My_Pthread Library STCF Scheduling

# of Threads Created	./parallelCal Time (in ms)	./vectorMultiply Time (in ms)	./externalCal Time (in ms)
1	2075	48	6328
10	2054	49	6933
100	2075	63	6918
1000	2120	78	6924

My_Pthread Library MLQF Scheduling

# of Threads Created	./parallelCal Time (in ms)	./vectorMultiply Time (in ms)	./externalCal Time (in ms)
1	2060	45	6165
10	2045	48	6196
100	2014	66	6183
1000	2053	68	6232

Pthread Library STCF Scheduling

# of Threads Created	./parallelCal Time (in ms)	./vectorMultiply Time (in ms)	./externalCal Time (in ms)
1	2027	41	6580
10	471	276	2445
100	439	340	2444
1000	515	469	2468

Pthread Library MLFQ Scheduling

# of Threads Created	./parallelCal Time (in ms)	./vectorMultiply Time (in ms)	./externalCal Time (in ms)
1	2047	43	6592
10	482	254	2488
100	442	323	2529
1000	443	424	2481