CS416

Abid Azad(aa2177), Ghautham Sambabu(gs878)

Project 2 Report

Logic Behind API Functions and Scheduler

Thread Creation

In this function, the initial step involves receiving parameters, such as the pointer to the thread to be created and an attribute for the threads. Subsequently, an evaluation is conducted to determine whether the process has commenced. Following this, the function undertakes the initialization of a queue, ensuring the prevention of concurrent creation of multiple queues. Proceeding with this preparation, a hashmap is initialized for efficient lookup of all worker threads. The subsequent actions encompass the creation of a Thread Control Block, establishing the context for the worker thread, and allocating a requisite amount of stack space to facilitate seamless execution. Upon configuration, the thread is enqueued into the run queue, marking it as ready for execution. This involves the creation of both the scheduling context and the main context. Moreover, if the scheduler is yet to be instantiated, the function takes the initiative to configure the timer, ensuring comprehensive preparedness for execution.

Thread Control Block

Each worker thread is associated with a dedicated Thread Control Block (TCB), which encapsulates pertinent attributes such as Thread ID (TID), status, joiningThread, and elapsed time. The creation of this TCB structure establishes a systematic framework through which various functions and the scheduler interact with distinct properties of the control block. The TCB structure, crafted to accommodate the unique characteristics of each thread, serves as a central repository for essential information. Its properties, including TID for identification, status for operational state tracking, joiningThread for synchronization purposes, and time elapsed for temporal analysis, collectively contribute to effectively managing the thread's lifecycle. Throughout the execution of functions and under the purview of the scheduler, references to the TCB enable seamless communication and manipulation of the thread's behavior. This structured approach ensures clarity and coherence in thread-related operations, fostering a robust and maintainable thread management system.

Thread Context

Each worker thread possesses a dedicated execution context tailored for CPU operation. As previously highlighted, this contextual facet is an important Thread Control Block (TCB) component and is instantiated within the "create worker thread" method. It seamlessly collaborates with the TCB hashmap to enhance overall functionality.

Main Context

A main Thread Control Block (TCB) is generated and inserted into a hashmap to establish the primary context. This follows the initial approach for managing both the context and scheduling aspects. The main benchmark thread is executed within this context and creates worker threads. Consequently, this particular context becomes instrumental in executing the benchmark code.

Runqueue

Functioning as the scheduler queue preceding the multi-level feedback queue scheduler's implementation, this mechanism utilizes a queue data structure. Its purpose is to systematically enqueue worker threads, ensuring their orderly execution by the CPU in the subsequent sequence.

Timer

The incorporation of timers facilitates the periodic transition into the scheduler context, triggered by the expiration of a predefined time quantum. Employing the itimerval struct, coupled with signaling when the alarm activates, is the mechanism to switch between contexts seamlessly.

Thread Yield

The objective of this function is to relinquish CPU control to other user-level worker threads willingly. This is accomplished through a sequence of actions: transitioning the state of the current worker thread from Running to Ready, preserving its context within the Thread Control Block (TCB), and switching from the thread context to the scheduler context.

Thread Exit

The initial step involves deallocating any dynamically allocated memory created during this thread's initiation. Subsequently, a check is performed to determine if the thread's Thread Control Block (TCB) is associated with a joiner thread. If such an association is identified, the TCB is transformed into a joined state and added to the run queue. Following these actions, the thread's statistics are processed.

Thread Join

Within this code segment, the program is designed to wait for a designated thread to conclude its execution by systematically deallocating any dynamically allocated memory attributed to the joining thread. Subsequently, a check is conducted to determine whether the Joined Thread Control Block (TCB) remains incomplete. If so, a context swap is triggered to facilitate the completion of the associated tasks.

Thread Synchronization

A parallel to the mutex functionality in the pthreads library has been implemented in addressing thread synchronization. Within the worker_mutex_lock method, the built-in test-and-set atomic function assesses the mutex's status. Successful acquisition of the mutex grants entry into the critical section. Conversely, if mutex acquisition fails, the current thread is enqueued into the block list, and a context switch to the scheduler thread is initiated. The mutex is released in the corresponding worker_mutex_unlock method, rendering it available for subsequent use. Simultaneously, threads in the block list are transitioned to the run queue, facilitating their execution

Thread Mutex Destroy

For the destruction of the mutex, the process involves invoking a helper function (queueClear) to manage the deallocation of dynamic memory initially created during the execution of the worker_mutex_init method.

PSJF

In implementing the Pre-emptive Shortest Job First algorithm, a Queue is the designated data structure, effectively accommodating Thread Control Blocks (TCBs). Within the TCB of a worker thread, a crucial "elapsed" counter is maintained, signifying whether the time quantum has transpired since the thread's scheduling. Upon the expiration of the time quantum, the worker thread is relocated to the tail of the queue, and a new worker thread is prepared from the head of the list. Notably, due to the need for foreknowledge regarding the actual runtime of a job, the algorithm identifies the thread with the minimum counter value, removes it from the run queue, and orchestrates a context switch to seamlessly transition the selected thread to the CPU.

MLFQ

A multi-level feedback queue comprises four distinct levels in implementing the Multi-level Feedback Queue algorithm. Each time a worker thread consumes a time quantum, it is placed into a lower runqueue. The scheduler prioritizes the highest level runqueue as priority one. Should a thread yield before its time quantum expires, it remains in the current runqueue. However, once it becomes the last item in the queue, it switches to the lower runqueue. The experiment's choice of different values for the parameter S (5ms for High priority, 10ms for

Medium priority, 15ms for Low-Med priority, and 20ms for Low priority) is noteworthy: smaller quantum sizes enhance responsiveness but may elevate context-switching overhead. Conversely, larger quantum sizes diminish context switching frequency but might prolong waiting times for shorter processes. This balance is crucial in optimizing the performance of the Multi-level Feedback Queue algorithm.

Benchmark Results

The tables below showcase the outcomes yielded by our thread library across various benchmark tests. In each of these tests, we conducted evaluations with 6, 40, and 75 worker threads, representing scenarios ranging from an average number to a substantial volume of workers. The recorded times are expressed in microseconds.

External Calculation

	6	40	75
Runtime	608	573	647
Total Sum	167780384	167780384	167780384
Context Switches	164	218	237
Avg Turnaround Time	512.9	126.9	68.95
Avg Resp Time	20.00	20.01	20.01

This test, which was IO-bound, focused on using multiple threads to read and process data from files concurrently, calculating the sum of the values obtained from these files. The verification

step ensures that the parallel computation produces the correct result. The runtime, context switches, and turnaround time mainly was consistent with little variance.

Parallel Calculation

	6	40	75
Runtime	1553	1584	1630
Total Sum	83842816	83842816	83842816
Context Switches	338	380	339
Avg Turnaround Time	1649.8	1148.7	1127.8
Avg Resp Time	20.01	20.02	20.02

In summary, this test, which was CPU bound, uses multiple threads to perform parallel array addition, calculates partial sums concurrently, and then combines these partial sums into a total sum. The sequential verification step ensures the correctness of the parallel computation. The runtime, context switches, and turnaround time mainly was consistent with little variance.

Vector Multiplication

	6	40	75
Runtime	60	90	68
Total Sum	631560410	631560410	631560410
Context Switches	164	218	237
Avg Turnaround Time	512.9	126.9	68.95
Avg Resp Time	20.00	20.01	20.01

In summary, this test, which was CPU bound, uses multiple threads to perform parallel array addition, calculates partial sums concurrently, and then combines these partial sums into a total sum. The sequential verification step ensures the correctness of the parallel computation. The runtime, context switches, and turnaround time mainly was consistent without much variance

Comparison with Linux pthread Library

	external_cal		parallel_cal		vector_multiply	
Runtime(μs)	608	1451	1565	589	60	147

We then compared our solution (which is highlighted in green) to the native pthreads library (which is highlighted in red). For the most part, our solution is more efficient than pthreads as it was able to faster than pthreads in the external_cal and vector_multiply test.

Collaboration and References

In the process of developing this project, the following resources and collaborations were invaluable:

IBM C Programming Documentation:

The IBM C Programming documentation provided essential information and syntax references, including their official website and related pages. These resources made learning concepts such as atomic flags and other syntax details possible. The official IBM C Programming page proved to be a valuable reference during the development of this project.

ChatGPT Collaboration for Data Structures:

Collaborating with ChatGPT provided valuable insights and guidance, particularly in developing template structures for queues and hash map data structures implemented in C.

Community Forums and Programming Communities:

Various online programming communities and forums, such as Stack Overflow and in-class Piazza programming communities, were essential for troubleshooting and seeking advice from experienced developers and TAs. Active participation in these communities allowed for the resolution of coding challenges and understanding best practices.