PA2 Multithreaded Task Scheduler Report

Yavuz Can Atalay 32445

November 27, 2025

Abstract

This report explains the design and implementation of a multithreaded Sorted Dispatcher Database (SDD) for task scheduling. It covers queue management APIs, inter-thread task stealing logic, and synchronization mechanisms used to implement a fully concurrent job execution simulator.

1 Introduction

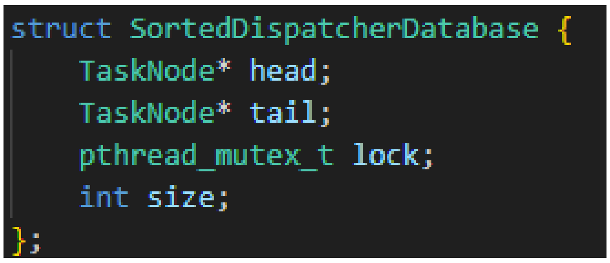
The goal of this project is to implement a fully functional multithreaded task scheduler using a set of processor queues, where each core is represented by a dedicated worker thread. The scheduler must support sorted insertion of tasks based on duration, shortest-job-first execution, work stealing from other queues, and thread-safe management of a shared data structure known as the Sorted Dispatcher Database (SDD).

The assignment requires implementing four core API functions (`submitTask`, `fetchTask`, `fetchTaskFromOthers`, and `print_queue`), along with two main thread-related functions (`processJobs` and `initSharedVariables`). The challenge lies in ensuring correct concurrency behavior, maintaining sorted order in each queue, preventing data races, avoiding deadlocks, and guaranteeing that all tasks in the system are executed exactly once.

In this report, I explain the purpose and logic of each function, how the queues are synchronized using mutex locks, how work stealing ensures load balancing between cores, and how threads detect the termination condition when all jobs are completfed.

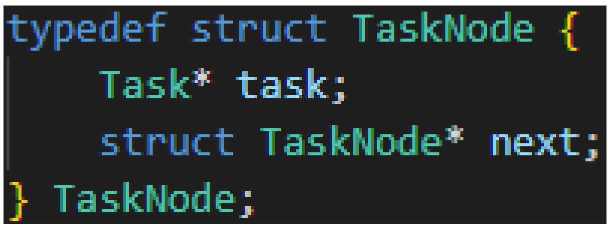
2 Structs and SDD

Certain structs are given to us at first but there are others expected to be created or filled during the code implementation. First one is, SortedDispatcherDatabase, a data structure specific to each core which will be used during storing, fetching or executing the jobs. I personally decided to use a linked list for basic implementation rather than a type of tree such as AVL.
SDD has 4 variables: A head pointer, a tail pointer, a lock for automicity across API functions and size varaible to follow jobs quantity on each thread/core's SDD. As a helper, a basic struct called TaskNode having a task struct which is already provided to us and a next pointer to create linked list structure across code implementation is implemented.



```c
struct SortedDispatcherDatabase {
    TaskNode* head;
    TaskNode* tail;
    pthread_mutex_t lock;
    int size;
};
```

Figure 1: SortedDispatcherDatabase



```c
typedef struct TaskNode {
    Task* task;
    struct TaskNode* next;
} TaskNode;
```

Figure 2: TaskNode

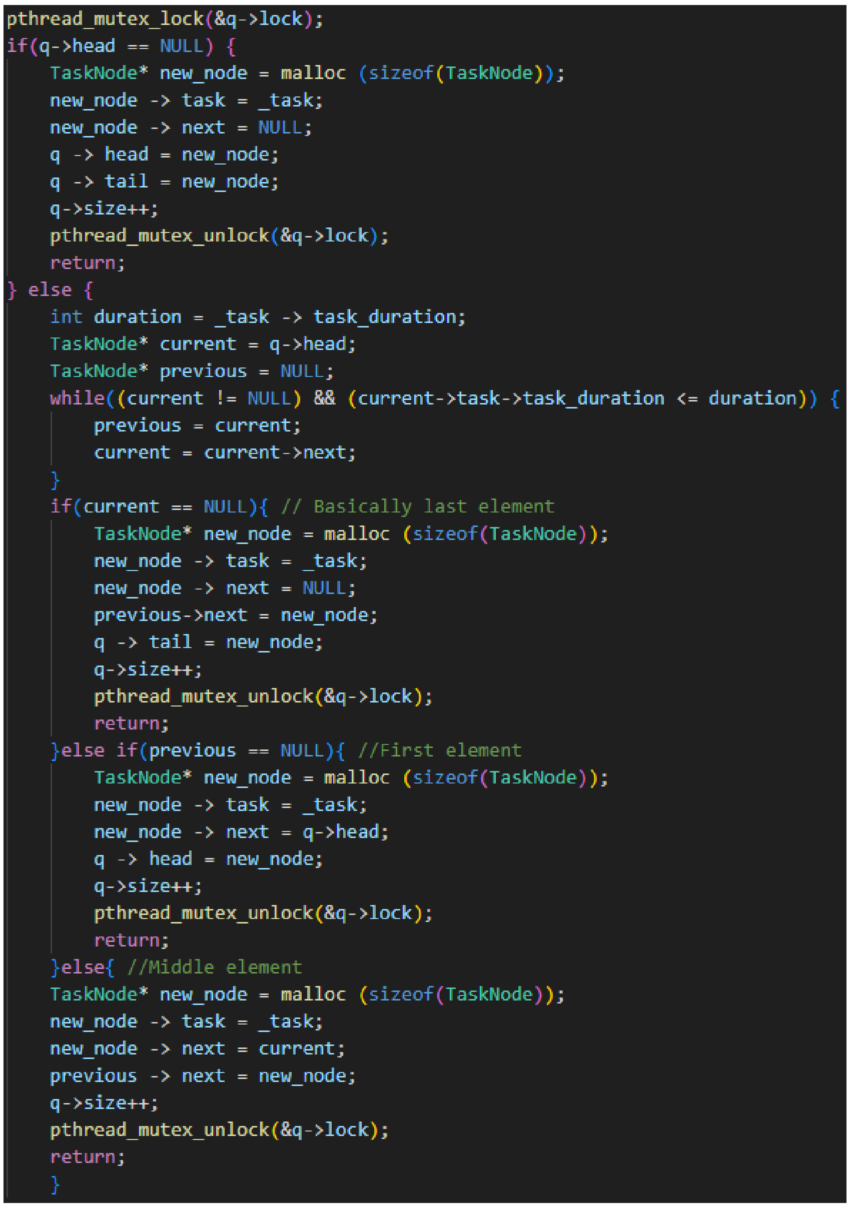
3 API Functions

There are 4 functions needed in order to execute processJobs functions, which is used as a thread function for each core simulator thread. These API functions will be used during scheduling and executing necessary task across execution of processes.

3.1 submitTask

Submit task API function is utilised in order to add a job into core's data structure(SDD). Different code blocks are implemented, each refers to specific cases in order to protect ordering of the list elements.(Tasks are stored from longest to shortest in the linked list structure):

- **Case 1: Empty queue**
  - Allocate a new `TaskNode` using `malloc`.
  - Set both `head` and `tail` to this node.
  - Increment the queue size.
  - Return immediately.
- **Case 2: Queue with one node**
  - Perform the same initialization steps as in Case 1.
  - Only the `tail` pointer needs rearrangement depending on the new task's duration.
- **Case 3: Queue with two or more nodes**
  - Iterate through the list using a `while` loop until reaching the first node whose duration is greater than the new task's duration.
  - Three subcases determine the correct insertion point:



```c
pthread_mutex_lock(&q->lock);
if(q->head == NULL) {
    TaskNode* new_node = malloc (sizeof(TaskNode));
    new_node -> task = _task;
    new_node -> next = NULL;
    q -> head = new_node;
    q -> tail = new_node;
    q->size++;
    pthread_mutex_unlock(&q->lock);
    return;
} else {
    int duration = _task -> task_duration;
    TaskNode* current = q->head;
    TaskNode* previous = NULL;
    while((current != NULL) && (current->task->task_duration <= duration)) {
        previous = current;
        current = current->next;
    }
    if(current == NULL){ // Basically last element
        TaskNode* new_node = malloc (sizeof(TaskNode));
        new_node -> task = _task;
        new_node -> next = NULL;
        previous->next = new_node;
        q -> tail = new_node;
        q->size++;
        pthread_mutex_unlock(&q->lock);
        return;
    }else if(previous == NULL){ //First element
        TaskNode* new_node = malloc (sizeof(TaskNode));
        new_node -> task = _task;
        new_node -> next = q->head;
        q -> head = new_node;
        q->size++;
        pthread_mutex_unlock(&q->lock);
        return;
    }else{ //Middle element
    TaskNode* new_node = malloc (sizeof(TaskNode));
    new_node -> task = _task;
    new_node -> next = current;
    previous -> next = new_node;
    q->size++;
    pthread_mutex_unlock(&q->lock);
    return;
    }
```

Figure 3: SubmitTask Function

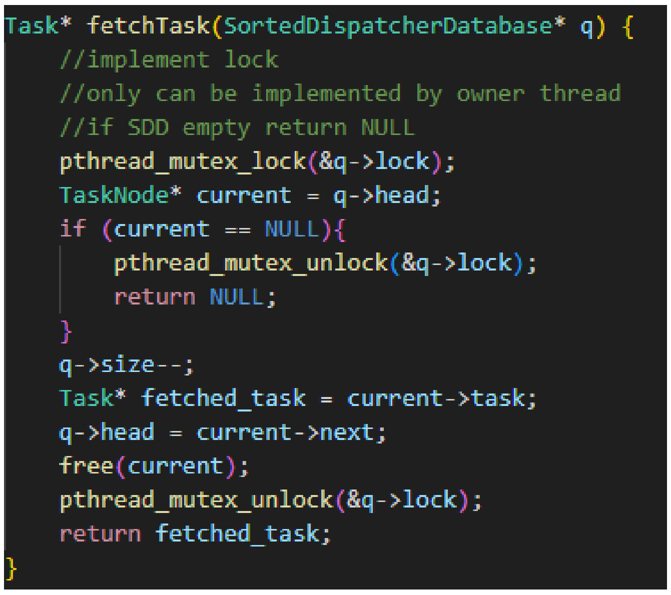
    * **Insert at end** — if `current == NULL`, the new task becomes the last element.
    * **Insert at beginning** — if `previous == NULL`, the new task becomes the new head.
    * **Insert in middle** — otherwise, link the new node between `previous` and `current`.

  - Increment the queue size accordingly.

To protect atomicity, `pthread_mutex_lock` is called at the beginning of the function, and `pthread_mutex_unlock` is executed before each return path to avoid deadlocks and allow safe concurrent access to the queue.

3.2 fetchTask

The `fetchTask` function is responsible for removing the shortest job from the calling thread's own SDD. Since the queue is kept in sorted order by duration, the shortest job is always located at the head of the linked list. The function acquires the queue mutex, checks whether the queue is empty, and if not, removes the head node, updates the head pointer, decreases the size counter, and returns the fetched task. It ensures that only the owner thread can remove tasks from the head, preventing race conditions and preserving sorted-order correctness.

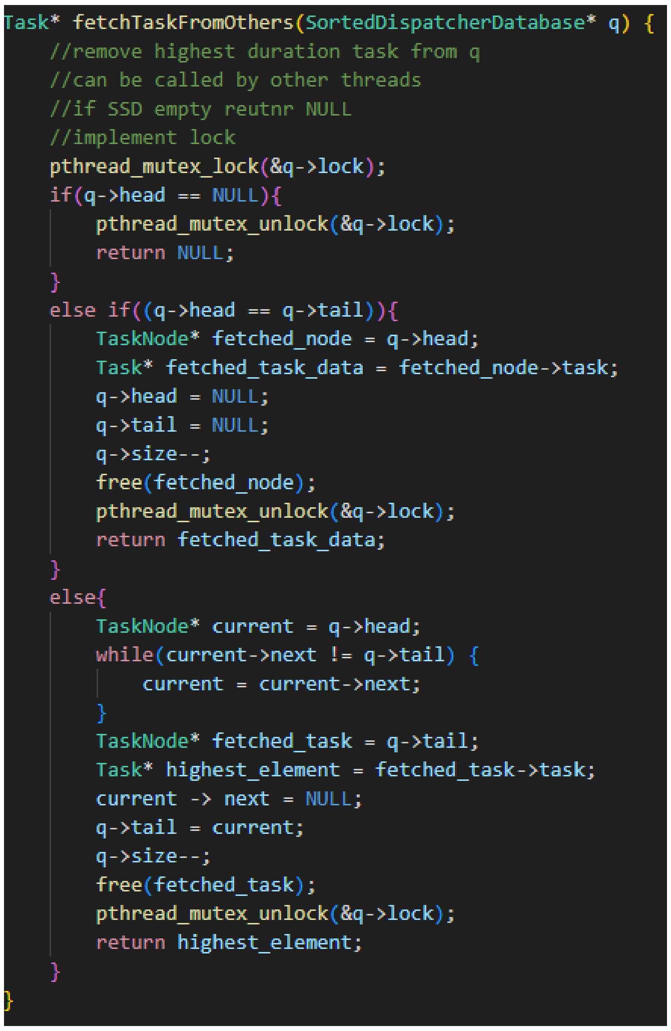


```c
Task* fetchTask(SortedDispatcherDatabase* q) {
    //implement lock
    //only can be implemented by owner thread
    //if SDD empty return NULL
    pthread_mutex_lock(&q->lock);
    TaskNode* current = q->head;
    if (current == NULL){
        pthread_mutex_unlock(&q->lock);
        return NULL;
    }
    q->size--;
    Task* fetched_task = current->task;
    q->head = current->next;
    free(current);
    pthread_mutex_unlock(&q->lock);
    return fetched_task;
}
```

Figure 4: FetchTask

3.3 fetchTaskFromOthers

The `fetchTaskFromOthers` function enables work stealing between cores. Main goal is to divide jobs so as to waste as little time as possible in each time slice period. Unlike `fetchTask`, this operation removes the *largest* task from the end of another queue. This choice prevents interference with that core's local SJF execution while still allowing efficient load balancing. The function locks the target queue, checks whether it is empty, walks to the second-last node, and detaches the tail. Stolen tasks maintain their remaining duration and are later reinserted into the stolers' core's queue according to sorted order.



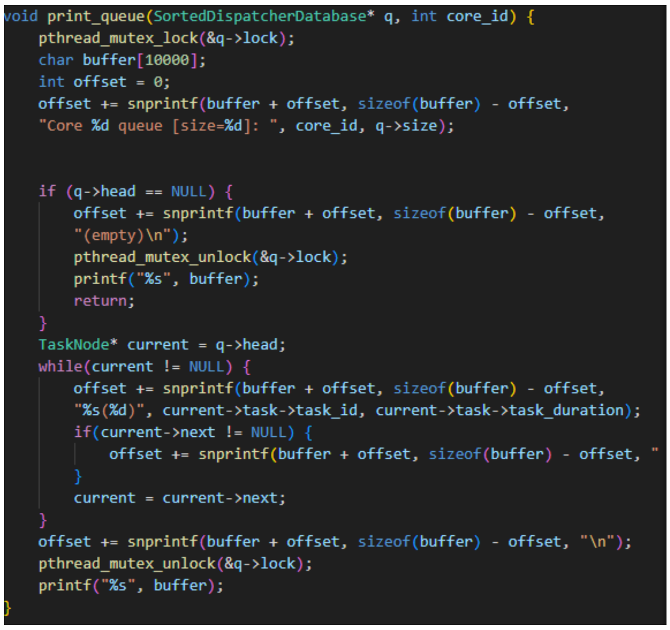
```c
Task* fetchTaskFromOthers(SortedDispatcherDatabase* q) {
    //remove highest duration task from q
    //can be called by other threads
    //if SSD empty reutnr NULL
    //implement lock
    pthread_mutex_lock(&q->lock);
    if(q->head == NULL){
        pthread_mutex_unlock(&q->lock);
        return NULL;
    }
    else if((q->head == q->tail)){
        TaskNode* fetched_node = q->head;
        Task* fetched_task_data = fetched_node->task;
        q->head = NULL;
        q->tail = NULL;
        q->size--;
        free(fetched_node);
        pthread_mutex_unlock(&q->lock);
        return fetched_task_data;
    }
    else{
        TaskNode* current = q->head;
        while(current->next != q->tail) {
            current = current->next;
        }
        TaskNode* fetched_task = q->tail;
        Task* highest_element = fetched_task->task;
        current -> next = NULL;
        q->tail = current;
        q->size--;
        free(fetched_task);
        pthread_mutex_unlock(&q->lock);
        return highest_element;
    }
}
```

Figure 5: fetchTaskFromOthers

We have two variables called lower_mark and higer_mark working as tresholds for fetching jobs from different core's queue. If a queue has lower jobs than

3.4 print_queue

The `print_queue` function prints the full contents of a queue in one atomic output. To prevent interleaved prints from multiple threads, the function holds the queue lock, serializes the entire list into a local buffer, and releases the lock only after formatting completes. This approach avoids partial prints and ensures non-divided output for execution. Otherwise, when currently working thread is interrupted or time slice is finished, output would be cut in the middle and next thread would continue right after previous thread output.



```c
void print_queue(SortedDispatcherDatabase* q, int core_id) {
    pthread_mutex_lock(&q->lock);
    char buffer[10000];
    int offset = 0;
    offset += snprintf(buffer + offset, sizeof(buffer) - offset,
    "Core %d queue [size=%d]: ", core_id, q->size);


    if (q->head == NULL) {
        offset += snprintf(buffer + offset, sizeof(buffer) - offset,
        "(empty)\n");
        pthread_mutex_unlock(&q->lock);
        printf("%s", buffer);
        return;
    }
    TaskNode* current = q->head;
    while(current != NULL) {
        offset += snprintf(buffer + offset, sizeof(buffer) - offset,
        "%s(%d)", current->task->task_id, current->task->task_duration);
        if(current->next != NULL) {
            offset += snprintf(buffer + offset, sizeof(buffer) - offset, "
        }
        current = current->next;
    }
    offset += snprintf(buffer + offset, sizeof(buffer) - offset, "\n");
    pthread_mutex_unlock(&q->lock);
    printf("%s", buffer);
}
```

Figure 6: print_queue

4 processJobs

The `processJobs` function implements the main scheduling loop executed by each worker thread.

Initial Task Fetching. First of all, each core tries to fetch a task from its own jobs queue in order to process it. If it is NULL, implying the list is empty, it tries to steal a job from another core's queue using `fetchTaskFromOthers`. If stealing the task is successful, we break; otherwise, we sleep.

Balancing Queue Load. In the second if block, we check whether queue size is lower than the `lower_mark` boundary, as each core should also perform fetch from another queue if its own queue size is lower than this threshold. The key part is that the queue from which the task is stolen should have more jobs than `higher_mark` (the queue is named `other_queue` in the code). If it is not higher than this threshold, then the for loop inside the if block simply skips this core and checks the next core's queue.

Order of Submission and Fetching. The part that should be highlighted here is how we submit and fetch, as the order affects which job will be processed after stealing. We basically submit both the fetched-from-core's-own task and the stolen task. Then we recall `fetchTask`, since submitting performs sorting between submitted and already existing jobs inside the queue. Then we recall fetch to retrieve the shortest duration job and execute it.

Reinserting or Finalizing Tasks. Finally, if the task isn't finished after the time slice (duration is larger than 0), we resubmit the task so that it will be executed in the future. Else if the task is finished, we free the memory of the task.

Once a task is obtained, the thread calls `executeJob`, which simulates execution, updates cache-warmup states, and decrements task duration. If a task is not yet finished after execution, it is reinserted into the local queue via `submitTask`. When all jobs across all cores are detected as finished, the main thread sets the stop flag, causing `processJobs` loops to halt cleanly.

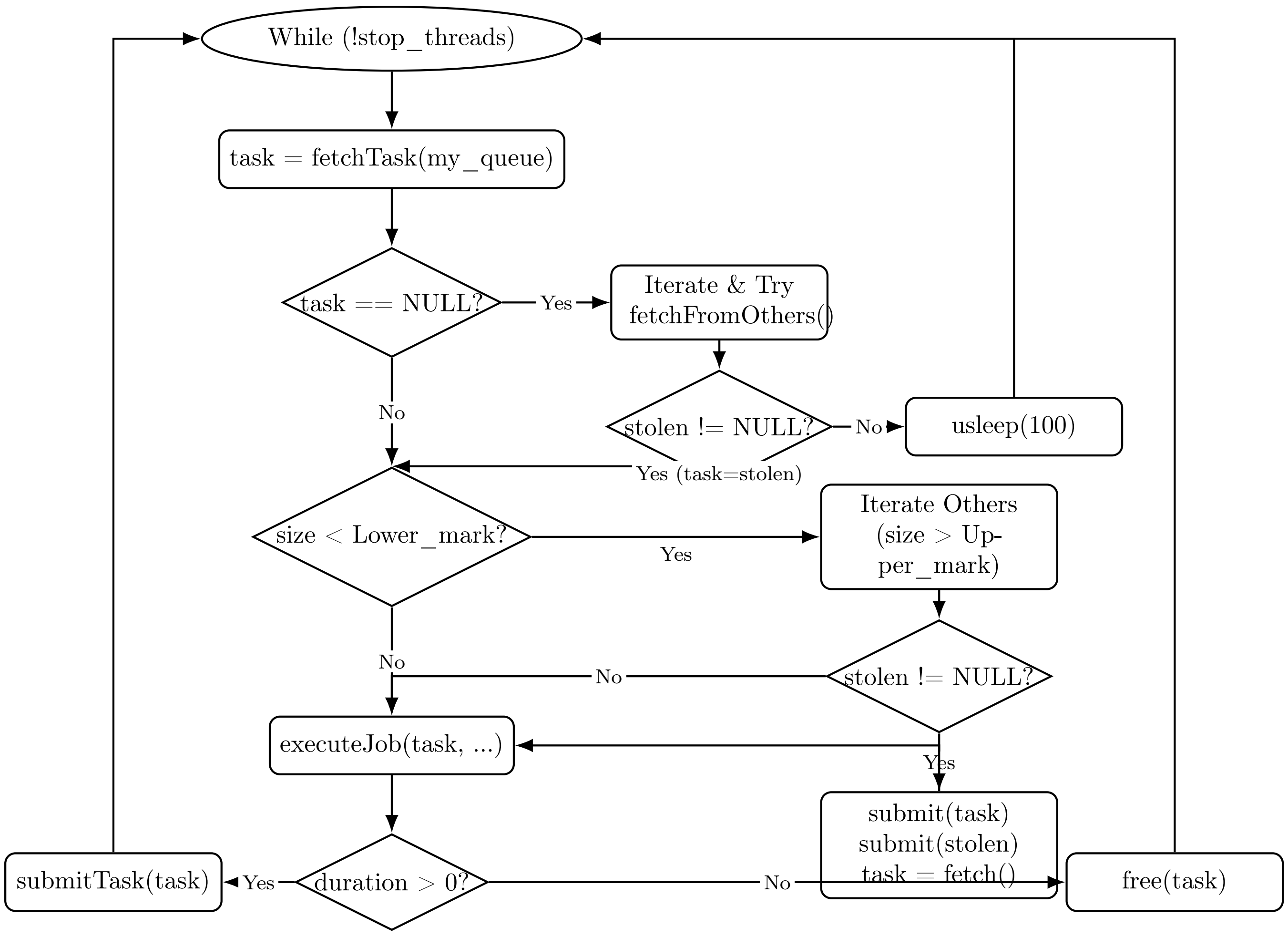


Figure 7: Process Jobs Execution Flow

5 initSharedVariables

The `initSharedVariables` function initializes all per-core data structures before any threads are created. It iterates over each `SortedDispatcherDatabase` instance, setting head and tail pointers to `NULL`, size to zero, and initializing the mutex lock for each queue. This guarantees that all queues start in a consistent, empty, thread-safe state before job distribution and thread execution begins.

6 Questions

Synchronization in our SDD implementation is ensured through a per-queue mutex locking mechanism. Each SortedDispatcherDatabase contains its own pthread mutex, and every queue operation such as submitTask, fetchTask, and fetchTaskFromOthers begins by locking this mutex and ends by unlocking it. Because no thread can modify head, tail, or size without holding the lock, all linked-list insertions, removals, and updates occur atomically. Work stealing also follows the same rule: the stealing thread must acquire the victim queue's lock before removing the largest task. Since each queue is independently protected and tasks are only moved while the lock is held, the system prevents corruption of shared structures and ensures deterministic behavior under concurrency.

Data races are prevented because the queues are the only shared data structure modified by multiple threads, and all modifications are serialized through their individual locks. No global scheduler state is modified concurrently, and tasks themselves are never shared between cores without lock protection. Even when a thread finishes executing a task and re-submits it, all interactions with the queue are protected by the mutex. This approach guarantees that no two threads can simultaneously change nodes.

The load balancing strategy is based on work stealing implemented inside processJobs. A thread first tries to take a task from its own queue; if the queue is empty, it attempts to steal the largest-duration job from other processor queues by calling fetchTaskFromOthers. Additionally, when a thread observes that its queue size is below a lower threshold, it proactively steals from cores whose queue sizes exceed an upper threshold. This threshold balancing helps distribute work more equally among cores and improves performance and decrease CPU waste.

Performance is maximized by combining global balancing and efficient per-core scheduling. Stealing the longest available job yields the greatest reduction in workload imbalance, preventing individual cores from being overloaded while others idle. After tasks are stolen, they are inserted back into the sorted queue structure, ensuring that each core continues to run its tasks in an order that reduces average completion time. This combination of stealing and local ordering allows the system to adapt dynamically to load variations and keeps CPU utilization high.

Using STCF as the internal scheduling policy provides strong local performance guarantees. Since submitTask inserts tasks in increasing order of duration, each core always executes the shortest available task first, which minimizes the average completion time of tasks in that queue. The automatic reordering of tasks after they are partially executed ensures that the queue always reflects current remaining durations.

However, STCF also introduces the possibility of starvation when long tasks are continuously overtaken by shorter ones. The sorted insertion also adds overhead to queue operations because the linked list must be scanned to find the correct position. Policies such as FIFO or Round Robin avoid these issues but perform worse in environments where minimizing turnaround time is the primary goal. Given that the focus of the implementation is accurate simulation and efficient per-core execution, STCF remains a suitable and effective policy despite its weaknesses.

7 Conclusion

This project demonstrates a functional multithreaded scheduler implementing shortest-job-first execution, sorted queue insertion, and work stealing. Proper synchronization is enforced through mutexes, ensuring concurrent operations. The design allows for balanced load distribution, prevents starvation, and ensures that all tasks eventually complete. Through modular API functions and thread logic, the system simulates realistic multi-core scheduling behavior effectively.