CS307 PA2 REPORT

BATUHAN GÜZELYURT 31003

Part 1: WorkBalancerQueue Implementation

Choice of Concurrent Queue Algorithm

We developed a custom concurrent double-ended queue (deque) to implement the `WorkBalancerQueue`. The deque allows insertion and removal of tasks from both ends, which aligns with the requirements specified:

- Owner Thread Operations:
  - `submitTask`: Inserts a task at the tail of the queue.
  - `fetchTask`: Removes a task from the head of the queue.
- Non-Owner Thread Operations:
  - `fetchTaskFromOthers`: Removes a task from the tail of another core's queue.

Data Structures Used

Queue Node (`QueueNode`)

Each node in the queue is represented by the `QueueNode` struct:

```
typedef struct QueueNode {
    Task* task;
    struct QueueNode* next;
```

```
    struct QueueNode* prev;
} QueueNode;
```

- `task`: Pointer to the `Task` struct.
- `next`: Pointer to the next node in the queue.
- `prev`: Pointer to the previous node in the queue.

WorkBalancerQueue (`WorkBalancerQueue`)

The WBQ is defined as:

```
struct WorkBalancerQueue {
    QueueNode* head;          // Head of the queue
    QueueNode* tail;          // Tail of the queue
    int size;                 // Number of tasks in the queue
    pthread_mutex_t mutex;    // Mutex for synchronization
};
```

- `head`: Points to the front of the queue (for `fetchTask`).
- `tail`: Points to the end of the queue (for `submitTask` and `fetchTaskFromOthers`).
- `size`: Keeps track of the number of tasks in the queue.
- `mutex`: Mutex lock to ensure thread safety during concurrent operations.

Ensuring Thread-Safety and FIFO Order

Mutex Synchronization

Each WBQ uses a `pthread_mutex_t` mutex to protect its critical sections. The mutex ensures that only one thread can modify the queue at a time, preventing race conditions.

- Owner Thread Operations:

  - Since only the owner thread calls `submitTask` and `fetchTask`, and these methods manipulate the queue's head and tail, we still use mutexes to guard against possible future modifications where multiple threads might interact.

- Non-Owner Thread Operations:

  - `fetchTaskFromOthers` can be called by any thread, requiring synchronization to prevent concurrent modifications.

Operations Implementation

`submitTask`

Inserts a task at the tail of the queue:

```
void submitTask(WorkBalancerQueue* q, Task* _task) {
   // create node
   QueueNode* node = (QueueNode*)malloc(sizeof(QueueNode));
   node->task = _task;
   node->next = NULL;
   node->prev = NULL;

   // lock  mutex
   pthread_mutex_lock(&(q->mutex));
```

```
    // insert at  tail
    if (q->tail == NULL) {
        q->head = node;
        q->tail = node;
    } else {
        node->prev = q->tail;
        q->tail->next = node;
        q->tail = node;
    }
    q->size++;

    // unlock  mutex
    pthread_mutex_unlock(&(q->mutex));
}
```

- Thread-Safety: The mutex lock ensures that only one thread modifies the queue at a time.
- FIFO Order: Tasks are added at the tail, preserving FIFO when fetched from the head.

`fetchTask`

Removes a task from the head of the queue:

```
Task* fetchTask(WorkBalancerQueue* q) {
    // lock mutex
```

```
    pthread_mutex_lock(&(q->mutex));

    // check if queue empty
    if (q->head == NULL) {
        pthread_mutex_unlock(&(q->mutex));
        return NULL;
    }

    // remove from head
    QueueNode* node = q->head;
    Task* task = node->task;
    q->head = node->next;
    if (q->head != NULL) {
        q->head->prev = NULL;
    } else {
        q->tail = NULL;
    }
    q->size--;

    // free node ,unlock mutex
    free(node);
    pthread_mutex_unlock(&(q->mutex));

    return task;
}
```

- Thread-Safety: Protected by mutex.

- FIFO Order: Tasks are removed from the head, ensuring the earliest inserted tasks are processed first.

`fetchTaskFromOthers`

Removes a task from the tail of another core's queue:

```
Task* fetchTaskFromOthers(WorkBalancerQueue* q) {
  // lock  mutex
  pthread_mutex_lock(&(q->mutex));

  // check if queue isempty
  if (q->tail == NULL) {
    pthread_mutex_unlock(&(q->mutex));
    return NULL;
  }

  // remove from tail
  QueueNode* node = q->tail;
  Task* task = node->task;
  q->tail = node->prev;
  if (q->tail != NULL) {
    q->tail->next = NULL;
  } else {
    q->head = NULL;
  }
  q->size--;
```

```
    // free node, unlock mutex
    free(node);
    pthread_mutex_unlock(&(q->mutex));

    return task;
}
```

- Thread-Safety: Protected by mutex, preventing concurrent access issues.

- Avoiding Data Loss/Duplication: Mutex ensures that only one thread can remove a task from the tail at any given time, preventing double deletions or lost tasks.

Formal Arguments for Correctness

- Mutual Exclusion: Mutex locks guarantee that only one thread can perform operations on the queue at a time.

- Atomicity: Operations within the mutex-protected sections are atomic, ensuring that tasks are not lost or duplicated.

- FIFO Preservation: By consistently inserting at the tail and removing from the head (for owner thread), the FIFO order is maintained.

- Concurrency Control: Non-owner threads can only remove tasks from the tail using `fetchTaskFromOthers`, and mutex locks prevent simultaneous removals.

Part 2: Mutex Implementation and Synchronization

Synchronization Mechanism

Each `WorkBalancerQueue` has its own mutex (`pthread_mutex_t mutex`) to synchronize access to its internal data structures.

- Fine-Grained Locking: By associating a mutex with each queue, we limit the scope of locking to individual queues, allowing maximum concurrency across different queues.

- Minimized Critical Sections: Only the essential code that modifies the queue is placed within the mutex lock, reducing contention.

Preventing Concurrency Issues

Double Insertion and Deletion

- Problem: Two cores might try to fetch the same task from another core's queue simultaneously, leading to duplicated or lost tasks.

- Solution: Mutex locks around the removal operations (`fetchTask` and `fetchTaskFromOthers`) ensure that once a task is being fetched, no other thread can access it until the operation is complete.

Lost Insertion/Deletion

- Problem: Simultaneous insertion and removal at the same end of the queue might corrupt the queue state.

- Solution: The mutex ensures that insertions (`submitTask`) and removals (`fetchTaskFromOthers`) are serialized, preventing inconsistent states.

Correctness, Fairness, and Performance

Correctness

- Data Integrity: Mutex locks ensure that the queue's internal pointers (`head`, `tail`) and `size` are updated correctly without interference.

- No Race Conditions: By protecting critical sections, we avoid scenarios where threads might read or write stale or inconsistent data.

Fairness

- Equal Opportunity: All threads must acquire the mutex before performing operations, ensuring that no thread is starved.

- Load Balancing: The load balancing strategy further ensures fairness by distributing tasks from overloaded cores to underutilized ones.

Performance Considerations

- Reduced Contention: Since each queue has its own mutex, threads operating on different queues do not block each other.

- Minimized Lock Duration: Locks are held only for the minimal time necessary to perform the operation, reducing waiting times for other threads.

Part 3: Load Balancing Strategy

Overview

The load balancing strategy aims to distribute tasks evenly across cores to maximize CPU utilization and reduce idle times. The strategy involves cores monitoring their own queue sizes and deciding when to fetch tasks from other cores.

Thresholds

- Low Watermark (`LOW_WATERMARK`): Set to 10 tasks. If a core's queue size falls below this value, it considers itself underutilized and attempts to fetch tasks from other cores.

- High Watermark (`HIGH_WATERMARK`): Set to 20 tasks. If another core's queue size exceeds this value, it is considered overloaded and eligible to have tasks stolen from it.

Implementation Details

In the `processJobs` function:

```
if (my_queue_size < LOW_WATERMARK) {
   for (int i = 0; i < NUM_CORES; i++) {
      if (i == my_id) continue; // Skip own queue

      WorkBalancerQueue* other_queue = processor_queues[i];

      // lock  other queue to read  size
      pthread_mutex_lock(&(other_queue->mutex));
      int other_queue_size = other_queue->size;
      pthread_mutex_unlock(&(other_queue->mutex));

      if (other_queue_size > HIGH_WATERMARK) {
         // try to steal a task
         Task* stolen_task = fetchTaskFromOthers(other_queue);
         if (stolen_task != NULL) {
            // reset cache affinity
            stolen_task->cache_warmed_up = 1.0;
            stolen_task->owner = my_queue;
```

```
            task = stolen_task;
            break;
        }
    }
  }
}
```

- Underutilized Cores: Cores with low queue sizes actively seek tasks from overloaded cores.

- Overloaded Cores: Cores with high queue sizes allow tasks to be stolen, reducing their workload.

- Cache Affinity Consideration: When a task is moved, its `cache_warmed_up` is reset to simulate the loss of cache optimization.

Benefits to Performance

- Improved CPU Utilization: By redistributing tasks, idle cores become active, and overall processing time is reduced.

- Reduced Idle Time: Cores spend less time waiting for tasks, leading to more efficient use of resources.

- Balanced Workload: Prevents scenarios where some cores are overloaded while others are idle.

Potential Trade-Offs

- Cache Affinity Loss: Migrated tasks lose their cache affinity, potentially increasing their execution time.

- Synchronization Overhead: Additional mutex locks during task stealing may introduce slight delays.

Conclusion

The implemented `WorkBalancerQueue` provides a thread-safe mechanism for managing tasks across multiple cores. By using mutexes for synchronization and carefully designing the queue operations, we ensure data integrity and prevent concurrency issues such as double insertion or data loss. The load balancing strategy effectively distributes tasks, improving performance and resource utilization while considering the trade-offs associated with task migration.