CS307 - Operating Systems
PA2 Report
Multi-Core Scheduling with Synchronization

Barış Pome
Student ID: 31311
Sabancı University

Fall 2024-2025

Introduction

Modern operating systems leverage multi-core processors to execute tasks concurrently. Efficient task scheduling across multiple cores is essential to achieve optimal performance, reduce latency, and fully utilize system resources. This assignment focused on implementing a simulation for multi-core scheduling that incorporates:

- **Dynamic Load Balancing**: Distributes workloads evenly across cores by dynamically adapting thresholds based on queue sizes.
- **Thread Safety**: Ensures synchronization in shared data structures to avoid race conditions, double insertions, or task loss.
- **Cache Affinity and Work Stealing**: Prioritizes local queues for better cache efficiency while allowing task stealing for load balancing.

The report outlines the design decisions, implementation details, and formal synchronization mechanisms used to achieve these objectives.

Program Overview

The project is implemented across three files:

- **wbq.h**: Contains data structure definitions and function prototypes for the Work Balancer Queue (WBQ), which is central to task management.
- **wbq.c**: Implements the WBQ operations, including task submission, fetching, and work stealing, with careful synchronization.
- **simulator.c**: Contains the main simulation logic, including thread functions for task execution, dynamic watermark calculations, and shared variable initialization.

The design ensures modularity, with a clear separation of concerns:

- The WBQ serves as the foundation for thread-safe task management.
- The simulator orchestrates task scheduling and balancing across cores.
- Synchronization mechanisms are encapsulated within WBQ operations, ensuring correctness.

Detailed Explanation of the Code

1. wbq.h: Data Structures and Function Prototypes

The `wbq.h` file defines the core data structures and function prototypes that enable the WBQ functionality.

WorkBalancerQueue The central data structure, `WorkBalancerQueue`, is designed for thread-safe operations:

- `_Atomic QueueNode *head, *tail`: Atomic pointers to the head and tail nodes of the queue.
- `_Atomic int count`: Tracks the size of the queue for quick load-balancing decisions.
- `pthread_mutex_t lock`: Ensures exclusive access during complex operations, such as task insertion or removal.

Task Represents a unit of work:

- `char *task_id`: A unique identifier for the task.
- `int task_duration`: Represents the remaining execution time of the task.
- `WorkBalancerQueue *owner`: Tracks the queue that owns the task, enabling work stealing.

ThreadArguments Encapsulates arguments passed to thread functions:

- `WorkBalancerQueue *q`: Pointer to the queue assigned to the thread.
- `int id`: Unique identifier for the thread.

Function Prototypes

- `void WorkBalancerQueue_Init(WorkBalancerQueue *q)`: Initializes the queue.
- `void submitTask(WorkBalancerQueue *q, Task *_task)`: Adds a task to the queue.
- `Task *fetchTask(WorkBalancerQueue *q)`: Fetches the first task from the queue.

- `Task *fetchTaskFromOthers(WorkBalancerQueue *q)`: Implements work stealing.
- `int getQueueSize(WorkBalancerQueue *q)`: Returns the size of the queue.

2. wbq.c: WBQ Implementation

Queue Initialization

```c
void WorkBalancerQueue_Init(WorkBalancerQueue *q)
{
    atomic_store(&q->head, NULL);
    atomic_store(&q->tail, NULL);
    atomic_store(&q->count, 0);
    pthread_mutex_init(&q->lock, NULL);
}
```

This function initializes the queue. Atomic pointers ensure thread-safe access to the head and tail nodes. A mutex lock is used for synchronization during more complex operations.

Task Submission

```c
void submitTask(WorkBalancerQueue *q, Task *_task)
```

Tasks are added to the tail of the queue. If the queue is empty, both `head` and `tail` point to the new node. The atomic counter is incremented to reflect the new queue size.

Task Fetching

```c
Task *fetchTask(WorkBalancerQueue *q)
```

This function retrieves and removes the first task from the queue. Atomic operations ensure thread-safe updates to the `head` pointer. The counter is decremented to maintain accurate size tracking.

Work Stealing

```c
Task *fetchTaskFromOthers(WorkBalancerQueue *q)
```

Implements load balancing by stealing tasks from the middle of another queue. The design ensures a minimum number of tasks remain in the source queue to prevent starvation.

Queue Size

```c
int getQueueSize(WorkBalancerQueue *q)
```

Returns the current size of the queue using an atomic counter. This function is lock-free and optimized for frequent calls during load balancing.

3. simulator.c: Main Simulation Logic

Dynamic Watermark Calculation

```c
void calculateWatermarks(int my_id, int *high_watermark, int
    *low_watermark)
```

This function dynamically adjusts the high and low watermarks based on the system load:

- `high_watermark = avg_load * 1.5`: Tasks beyond this threshold are eligible for stealing.
- `low_watermark = avg_load * 0.5`: Threads below this threshold attempt to steal tasks.

Task Processing

```c
void *processJobs(void *arg)
```

This is the main thread function. It prioritizes local queue processing for cache affinity. If the local queue size is below the low watermark, the thread attempts to steal tasks from other queues.

Shared Variable Initialization

```c
void initSharedVariables()
```

Initializes per-core queues and resets the `finished_jobs` counter. Ensures all cores start with a consistent state.

Load Balancing Design

The load balancing strategy is based on:

- **Dynamic Watermarks**: Adapts to current system load, preventing unnecessary work stealing.
- **Cache Affinity**: Prioritizes local queues to reduce inter-core communication overhead.
- **Work Stealing**: Redistributes tasks from overloaded queues to underutilized ones, ensuring even load distribution.

This design optimizes both core utilization and cache efficiency.

Synchronization Mechanisms

Synchronization is achieved through:

- **Atomic Operations**: Lock-free updates to pointers and counters ensure thread-safe access.
- **Mutex Locks**: Used for complex operations, such as modifying multiple pointers simultaneously.

Correctness

- **Race Conditions**: Prevented by mutex locks and atomic operations.
- **Double Insertion**: Atomic counters ensure accurate size tracking.
- **Task Loss**: Thread-safe pointer updates prevent overwriting or dropping tasks.

Performance Intuition

This design improves performance by:

- Reducing idle time through dynamic load balancing.
- Enhancing throughput by prioritizing local tasks for better cache usage.
- Maintaining fairness by redistributing tasks via work stealing.

The combination of efficient load balancing and thread safety ensures scalability and responsiveness.