Sabanci University

Operating Systems

CS 307

Programming Assignment - 2: Multi-Core Scheduling with Synchronization

Release Date: 08 November 2024
Deadline: 18 November 2024 23.55

1 Problem Description

In the lectures, we have seen that the OS scheduler aims to optimize *average turnaround time* (AvgTAT) for computationally heavy processes and *fairness* conditions for more interactive processes. However, these metrics are meaningful for only single core machines. Modern computers consist of multi-core CPUs and scheduling processes in a multi-core machine brings extra challenges and concerns.

Load balancing is one of the most important issues in multi-core scheduling. If all the jobs are assigned to a single core, even though the OS schedules all the jobs optimally on this core, we might not get the best performance and utilization of the whole CPU. A mediocre scheduling strategy that somehow evenly distributes jobs to cores but does not guarantee the minimum AvgTAT on each core might perform much better and yield a better overall AvgTAT than a scheduler that distributes jobs to cores poorly, but guarantees AvgTAT inside a core.

While achieving the best load balancing is desirable, one should consider an important problem. Distributing jobs evenly to CPU cores and improving overall CPU utilization might require frequent migration of jobs from one core to another during its execution. While a job is running on a core, this core caches important data and meta-data about this job which increases the runtime performance. If the job is scheduled into another core after a context switch, this job suffers from a problem called *cold start*. Since the new core contains no data and metadata of the new process in its cache, its performance in the new core is pretty low and it stays low until the new core's cache is populated with this new job's entries. Hence, frequent job exchange might degrade performance even though it increases the CPU utilization and load balancing. For more comprehensive information on challenges of multi-core scheduling and possible solutions, please read Chapter 10 of OSTEP before proceeding any further.

1.1 A Simple Multi-Core Scheduling Policy

A widely used approach to accommodate multiple cores at once is to have each of them store their tasks in a separate collection data structure. When a task is created, it is assigned (by some heuristic of the OS) to one of the cores and inserted into its collection, and from that point on that job only concerns the core it is designated to and we get all the performance benefits

of cache affinity.

However, being very firm about the job-processor matching assigned at the very beginning can be costly. Since we cannot guess the running times of jobs beforehand, some of the cores might finish their tasks before others. In this case, these cores stay idle which causes inefficiency and low CPU utilization. In this case, we might want to evenly distribute jobs of busy cores among idle cores, hence the load balancing.

One possible solution is to not render cores completely oblivious about other cores' tasks and allow the migration of tasks from core to core once in a while. This engenders two new problems, one of which is synchronization. Whenever we try to migrate one job from one core to another, we have to be certain that another core isn't also trying to grab that job. If we don't account for possible data races in the data structures of the cores, we might have duplicate jobs, or a lost job, or even a deadlock. Therefore, the collection data structure cores keep must be thread-safe.

Lastly, we have to keep in mind the cost of migrating a task between cores. When a task runs in a core, it builds up its own cache state on this core. The cache is a small local lookup table that the processor checks first for any data operation. If the data it looks for isn't there, it fetches data from the main memory. The cost of accessing the memory is astronomically high compared to fetching from the cache. After running on the same core for some time, most of the frequently used data and metadata of this job can be found in this core's cache. This is called *cache affinity* and improves the runtime performance of a job. Migrating a job from one core to another completely resets its built-up cache state and it is called a cold start. Until the job gains cache affinity, this job performs poorly which degrades the overall performance.

In this assignment, you will build a multi-core scheduler simulator where cores are going to be simulated by threads, and they are going to tackle some dummy jobs we assign. You will also implement a collection data structure that will be maintained by each core thread to keep track of its jobs. This collection data structure is going to be called `WorkBalancerQueue` (or `WBQ` in short) from now on. Each core will insert and remove jobs to its `WBQ` while scheduling them and they will be able to fetch some jobs from other cores' `WBQ` 's for load balancing. Since multiple threads will be operating on the same `WBQ` , we require your `WBQ` implementation to work correctly while being accessed concurrently.

In the following chapter, we will provide requirements and details for the multi-core scheduler and the `WBQ` concurrent collection data structure. We will provide the API for the both the scheduler and `WBQ` i.e., the method signatures and their specifications.

2 API Methods and Specifications

In this section, we will guide you through the essential components of the multi-core scheduler simulator. We will start by explaining the data structures required, including the job representation and `WBQ` , followed by the API methods of `WBQ` you will need to implement. Afterward, we will discuss the core simulator threads, outlining their operations and responsibilities. Finally, we will cover synchronization issues and strategies for handling concurrent access to queues, with a focus on balancing performance and correctness.

2.1 Data Structures for Simulation

- **Task Structure**: This struct represents individual jobs, the primary units of work within the simulation. Each job will be stored in a `WBQ` , which organizes and manages job distribution across cores. We have provided the `Task` struct to represent each job in the file `wbq.h`. You don't have to implement it by yourself. Here is the definition:

```
typedef struct Task {
    char* task_id;
    int task_duration;
    double cache_warmed_up;
} Task;
```

  Each `Task` struct includes:

  - `task_id`: A unique identifier for each job. You can assume that uniqueness is ensured, as jobs will be given to you with unique IDs.
  - `task_duration`: Represents the total time required to complete the job. The time unit is measured in CPU cycles, simulating processing time.

  - `cache_warmed_up`: A measure of cache affinity for this job. Higher values indicate better cache optimization, reducing the required execution time due to improved cache utilization. You might think of this parameter as a coefficient that will be multiplied by the expected portion of the job to be completed in this scheduling round to compute actual completed portion. Higher the parameter, higher the cache hit rate and bigger the piece of job it completes.

  You are not allowed to modify this struct, as it is referenced throughout the simulation code. Jobs will be provided as input to your simulator, so you will not create jobs yourself. The method for obtaining jobs for the scheduler will be explained in following sections.

- `WBQ` **Structure**: To manage jobs for each core, you will implement the `WBQ` struct. This structure, defined in `wbq.h`, keeps jobs of the core as a collection object and organizes them in a way that is easy to use by the scheduler. You are responsible for declaring the fields and state of the struct. It might be an array based or node based collection depending on your preferences. You might declare other helper structs in `wbq.h` if you need them.

  Moreover, your `WBQ` library must provide some API methods that will be discussed in the next part.

2.2 API for WorkBalancerQueue

`WBQ` is not an arbitrary collection data structure. In order to ensure fairness of the scheduler and high response time for the jobs, we want each core to execute its tasks in Round-Robin fashion. Hence, we would like to keep jobs in a queue like concurrent data structure so that jobs can be inserted and removed in *First-In-First-Out* (FIFO) order, at least by the core who owns the queue. For the `WBQ` description, you should assume that it has two separate ends: *head* and *tail*.

There are three primary API functions manage task processing in the queue.

- `void submitTask(WorkBalancerQueue* queue, Task* task)`: This method adds a new job to the tail end of the queue and can be only called by the owner thread.

- `Task* fetchTask(WorkBalancerQueue* queue)`: This method removes the next available job from the head end of the queue and can be only called by the owner thread. When the `WBQ` is empty it returns `NULL`.
- `Task* fetchTaskFromOthers(WorkBalancerQueue *queue)`: This method allows a core to remove a job from the tail end of another core's queue. This method will not be called by the owner thread. When the `WBQ` is empty it returns `NULL`.

These method declarations are provided in the file `"wbq.h"`. You must complete the implementations in a separate file `"wbq.c"`.You are not allowed to make changes in the method signatures of these three API methods. However, you may add other methods to `"wbq.h"` and `"wbq.c"` for your needs.

2.3 Core Simulator Threads

Core simulator threads will be executing a single method called `processJobs`. You will implement this method to manage scheduling of jobs. You might find the declaration of this method in `"simulator.c"` with the following signature:

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

where the argument *arg* contains the `WBQ` and it's core ID. Inside the file `wbq.h`, you are given the input struct definition for this method as:

```
typedef struct ThreadArguments {
    WorkBalancerQueue* q;
    int id;
} ThreadArguments;
```

Simulator cores will run a processing loop inside `processJobs`, where tasks will be fetched from its own `WBQ` or other cores and then executed. The method that will simulate the job execution will be provided to you under the name `executeJob`.This method will also update the remaining job duration considering the cache affinity coefficient. We provide a sample `executeJob` implementation in the `"sim_methods.c"` file so that you can test your simulator. In this sample implementation, threads simply sleep for simulating an execution.

Each core simulation instance will run `processJobs` method in a separate independent thread. Inside the main loop, the core will decide to fetch a job from its own `WBQ` or other queues. Other queues might be found using the

global variable `processor_queues`. When a core is idle or has fewer jobs, it can borrow from other cores to ensure a balanced workload. Here's how the method should work:

- **Monitor Job Load on the Core**: Each core periodically checks the state of its own `WBQ` whether it is heavily loaded or under utilized.
- **Dynamic Load Balancing Using `fetchTaskFromOthers()`**: If the queue contains few or no jobs, the core may initiate load balancing by seeking jobs from other cores' queues. From this point on, ownership of the job must be transferred from the old core to the new core so that only one core can execute this task.
- **Job Processing**: After handling load balancing, it should call `executeJobs` method where job execution is simulated. When a job is fetched from another core, the cache affinity (`cache_warmed_up`) is reset, simulating the loss of cache optimization when a job is handled by a different core. When the execution period finishes, this job should be inserted back to `WBQ` belonging to this core if it still has remaining time.
- **Monitor Termination**: `executeJobs` will signal if a job is finished. If all jobs are finished, main thread will set the flag *stop_threads*. Each core should check this flag and terminate `processJobs` if flag is raised. If you don't terminate `processJobs` correctly, program will not terminate or some jobs will not be finished.

This procedure is created for each core as a thread from the main. The initial distribution of jobs across cores will be given to you in the main method. When your simulator starts, each core will have a set of preloaded jobs in its `WBQ` . Hence, initially, some queues might be heavily loaded while others run empty. To address this imbalance, cores should periodically check if they need to retrieve tasks from others using the `fetchTaskFromOthers` API and do balancing. You are not allowed to make changes in the main method of `"sim_methods.c"`. You only need to focus on processing jobs and implementing dynamic load balancing as cores become idle or overloaded inside `processJobs`.

A suggested approach for balancing is to set two threshold values for the queue size as explained below:

- **High Watermark (HW)**: If a core's queue size exceeds a certain task count (e.g., 20 tasks), it may ask help from other cores to fetch tasks from its own queue.

- **Low Watermark (LW)**: If a core's queue size drops below this level (e.g., 10 tasks), it can fetch tasks from cores which requires help by using `fetchTasksFromOthers`.

However, you are free to design your own load balancing algorithm. Keep in mind that when a task is moved to another core, it loses its cache affinity, which may impact efficiency. In your report, you should clearly explain your load balancing design.

2.4 Ensuring Synchronization

Proper synchronization of `WBQ` is crucial since multiple cores may access their own or each other's queues concurrently. This requires you to handle shared data carefully, as concurrent reads and writes could lead to race conditions, where the outcome depends on the order of operations. For example, two cores may try to fetch the same task or enqueue tasks in an inconsistent state. Without effective synchronization, various issues can arise:

- Double insertion and deletion: Two cores could accidentally fetch the same job, where both cores fetches a job from head of another queue by using `fetchTaskFromOthers`. They may fetch the same job and when they finished for the cycle they submit same job to their queues causing duplicate jobs.

- Lost Insertion/Deletion: `WBQ` can be corrupted when its core tries to submit a job to the tail of the queue after a cycle finished with `submitTask` and other queue tries to fetch a task from tail of the queue with `fetchTaskFromOthers`. This can cause lost jobs.

To prevent these issues, you must implement a synchronization mechanism of your choice, such as mutexes, atomic operations, or other concurrency control primitives. Synchronization ensures that critical sections are atomic, maintaining data integrity and preventing errors like duplicated tasks or data corruption.

However, you must balance safety with performance, as too much synchronization can create bottlenecks, reducing overall efficiency. Using heavy

locking (like a single mutex) around the entire `WBQ` method bodies ensure safety but restricts concurrency as it serializes whole queue access. Serialization can severely degrade performance in a multi-core system where concurrent processing is expected. You should remember that mutexes are expensive, and consider approaches that minimize synchronization overhead and the size of synchronized code blocks. You may use lock-free or fine-grained solutions improve performance, but it can increase implementation complexity. We strongly recommend you to check Michael& Scott Queue implementation described in Page 11, Chapter 29 of OSTEP.

In your report, you should clearly describe your chosen synchronization method, explain how it prevents the above issues, and provide formal arguments supporting the correctness of your approach.

3 Submission Guidelines

In this PA, we expect you to submit three C header and implementation files along with your report.

- **wbq.h**: C header file containing the `WBQ` and its API declarations. Make sure to include your changes in `WBQ` template we provided.
- **wbq.c**: C file that contains your implementation of `WBQ` . Make sure that this file and it's header compiles without dependency to the main file. We will compile it with different main files for testing purposes.
- **simulator.c**: C file containing the *Main* simulator including your implementation for processing jobs. For all files, please adhere rules where you are allowed to make changes. Changes in part you are not allowed will be penalized.
- **report.pdf**: Your report that explains your queue and mutex implementation. In the first part, please explain whether you picked an existing concurrent queue algorithm or developed your own version. Then, please discuss, how this algorithm ensures thread-safety i.e., how FIFO order is preserved, it does not allow data loss or duplication in case of concurrent accesses. In the second part, describe your mutex implementation and argue why it ensures the correctness, fairness and performance requirements. Please note that all the reports are read carefully to understand your program.

During the submission of this homework, you will see two different sections on SUCourse. You are expected to submit your files separately. You should NOT zip any of your files. Please submit your report.pdf to "PA2 – Report Submission" and your three C files to "PA2 – Code Submission". The files that you submit should NOT contain your name or your id. SUCourse will not except if your files are in another format, so please be careful.

IMPORTANT NOTE: If the file names, struct names or the public method signatures do not match with the format explained above, you might get 0 from this assignment since the automated tests will fail.

IMPORTANT NOTE 2: In order to compile and test your implementation, please use the **makefile** provided in the PA2 bundle or use the commands in this files. We will use these commands for compiling your implementation during the automated grading and evaluation.

4 Grading

- **Compilation and termination (10 pts):** Given the `"makefile"` file that uses your `"wbq.c"`, when it is compiled with the command:

  ```
  make sim
  ```

  it successfully generates the executable file `"main"`. Moreover, all the tests successfully terminate. Successful termination entails that the program never deadlocks.

- **Single thread `WBQ` implementation (10 pts):** Your queue implementation works correctly in the case of sequential access of single thread. Even though there will be a single thread, we will use all three API methods in the tests.

- **Multi-thread `WBQ` implementation (10 pts):** Your queue implementation works correctly in the case of concurrent access of multiple threads. In this case, we will obey the API restrictions on the methods threads might call. We will check for job duplication and lost while evaluating the correctness of your queue.

- **Single core Simulation (10 pts):** Your `processJobs` implementation will be tested when simulation runs with core number set to 1.

- **Multi core Simulation (30 pts):** Your `processJobs` implementation will be tested when simulation runs with multiple cores. Again, we will consider the job duplication or loss as the criteria of correctness.
- **Performance tests (10 pts):** We expect your *WorkBalancerQueue* implementation to be efficient and not suffer from performance loss due to load imbalance. Therefore, we will run each test case several times with your implementation and take the average running times. Then, your average time will be compared against your peers with correct implementations, based on the lowest runtime. If you are in first %10 of the class, you will get full point. You will lose 1 point for every next %10.
- **Report (20 pts):** We expect you to submit a detailed report summarizing your work. You should explain your work balancing strategy clearly and provide an intuition on why it should increase the performance. Moreover, you should explain the synchronization mechanisms you imposed on your `WBQ` implementation and provide formal arguments on why it ensures correctness i.e., avoids double insertion or loss problems.

5 Sample Runs

5.1 Sample task generator

In PA2 bundle, we provided a sample task generator program with file **"task_input_generator.c"**. This is only for generating test samples and **You are not required to modify or submit this file**. As it is a different program, it should be compiled separately with command:

```
make generator
```

Once the compilation is complete, you can run the task generator with the command:

```
./generator
```

When the program starts, it will first prompt you to enter the **number of lines**, which corresponds to the number of cores that will initially have tasks assigned. The input value should be within the range **1 - NUM_CORES**, where `NUM_CORES` is a constant defined in the file **"constants.h"**.

Next, the program will ask for the **minimum** and **maximum number of task entries per line**. These values define the interval for the number of tasks assigned to each core. For each core, a random number of tasks will be generated within the specified interval.

The generator will then create a file named **"tasks.txt"**, which contains a list of randomly generated tasks. This file will be read by the simulator program to load tasks for each core. You can use this generated file to test the correctness of your implementation. During grading, we may also use this program to create test samples.

5.2 Sample outputs

In PA2 bundle, you will find generated 3 sample tasks with *sample_task*.txt* files. Additionally, we have provided the expected simulator outputs for these task files in the *sample*.txt*. Keep in mind that order of executions may change. Therefore, your program's output may not exactly match the order of the given sample output. However, you must make sure all jobs are finished and there are no lost jobs. Also, same job should not be run twice.

`executeJobs` will print the outputs as either of the follows:

Processor $process_id: Finished Task $task_id

Processor $process_id: Executed Task $task_id, it has $duration ms remaining

At the end program will exit and print:

All tasks finished, joining threads

This line points graceful termination of program. Absence of this line may point deadlock or abrupt termination. You may refer first point of grading criteria for graceful termination.

IMPORTANT NOTE: We will use automatic grading to evaluate your outputs. **You should not add any extra output messages** as it may disrupt automatic grading.