

Simulation of Multicore Processor Scheduler

Goal:

The goal of this project is to simulate a FIFO(first in first out) scheduler for a multicore processor. To have this simulation program will create threads equaling the amount of cores.

Data Structures:

For FIFO implementation I built a queue since queue is a compatible data type. For protection against race conditions regarding cross thread fetches the queue is inspired from Michael-Scott Queue.

Overview:

There will be NUM_CORES amount of cores(simulated with threads). All of which have their own queues to keep track of the tasks their respective core is supposed to run. They will get the tasks from the provided sample_tasks. They can just run their queues until completion. This is generally the slower option, since some cores may have too long or too short queues, both leading to wasted resources. Hence we also implement the ability for underburdened cores fetch tasks from overburdened queues.

QueueNode:

```
struct QueueNode{  
    QueueNode* next;  
    QueueNode* prev;  
    Task* task;  
};
```

Queue's implementation will be a doubly linked to accommodate fetches from head and tail. It just keeps the previous and next nodes.

WorkBalancerQueue:

```
struct WorkBalancerQueue {  
    QueueNode* head;  
    QueueNode* tail;  
    pthread_mutex_t head_lock,tail_lock;  
    int task_count;
```

};

Work balancer queue(WBQ for short) is the queue. It keeps track of the end and start of the queue. It also has two mutex, this is to help reduce waiting times. I will expand it more in the queue implementation part. Also the queue remembers how many cycles of tasks it has. This is to understand if the core owning this queue overburdened, normal or under burdened.

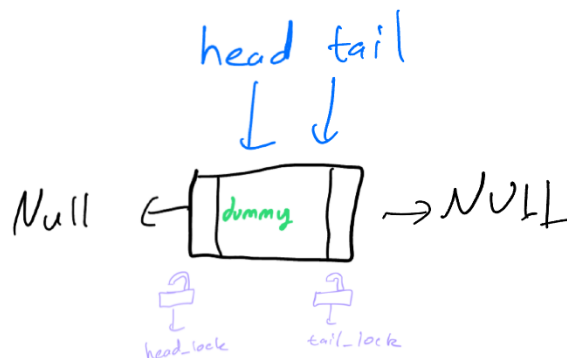
Task:

Already provided task struct. Has the id, duration and cache affinity. They are controlled by already provided methods, hence no need to make this document even longer by going in depth.

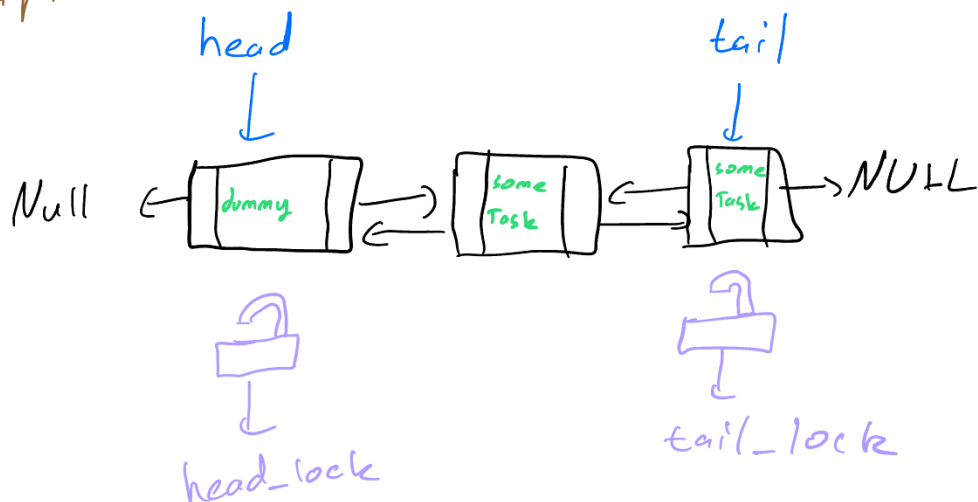
Queue Implementation

...

Initial



After 2 enqueues



```

66 //create the dummy node and initialize locks
67 void Queue_Init(WorkBalancerQueue* q)
68 {
69     QueueNode *tmp = malloc(sizeof(QueueNode));
70     assert(tmp != NULL);
71     tmp->next = NULL;
72     tmp->prev = NULL;
73     q->head = q->tail = tmp;
74     pthread_mutex_init(&q->head_lock, NULL);
75     pthread_mutex_init(&q->tail_lock, NULL);
76 }
77
78
79 void Queue_Enqueue(WorkBalancerQueue* q, Task* task)
80 {
81     QueueNode *tmp = malloc(sizeof(QueueNode));
82     assert(tmp != NULL);
83     tmp->task = task;
84     tmp->next = NULL;
85
86     pthread_mutex_lock(&q->tail_lock);
87     tmp->prev = q->tail;
88     q->tail->next = tmp;
89     q->tail = tmp;
90     pthread_mutex_unlock(&q->tail_lock);
91 }

```

Initialize will create a dummy node for the head, and initialize the locks. Dummy head is helpful with ensuring that it doesn't start empty

Enqueue function allocates a new node and sets its task to given argument. Then sets its prev to old tail and old tail's next to itself. And then becomes the new tail. Since operation is only about tail, only tail_lock is locked. So head can still be operated

```

int Queue_Dequeue(WorkBalancerQueue* q, Task* task)
{
    pthread_mutex_lock(&q->head_lock);
    QueueNode *tmp = q->head;
    QueueNode *new_head = tmp->next; //since first head is dummy acc
    if (new_head == NULL)
    {
        pthread_mutex_unlock(&q->head_lock);
        return -1; //empty queue
    }
    *task = *(new_head->task);
    q->head = new_head;
    new_head->prev = NULL;
    pthread_mutex_unlock(&q->head_lock);
    //printf("fetching task with id:%s \n", task->task_id);
    free(tmp);
    return 0;
}

int Queue_Steal_From_Back(WorkBalancerQueue* q, Task* task)
{
    pthread_mutex_lock(&q->tail_lock);
    QueueNode* tmp = q->tail;
    QueueNode* new_tail = tmp->prev;
    new_tail->next = NULL;
    if (new_tail == NULL)
    {
        pthread_mutex_unlock(&q->tail_lock);
        return -1; //empty queue
    }
    *task = *(tmp->task);
    q->tail = new_tail;
    pthread_mutex_unlock(&q->tail_lock);
    free(tmp);
    return 0;
}

```

Dequeue is responsible for deleting the head node and giving its task. This implementation requires a Task pointer to be passed and changes that, only returning success or fail codes. Its process starts with locking head_lock. Gets the task from head and changes head to the next in queue.

Steal_From_Back is responsible for getting the last element in queue. Its process is the same as dequeue except for working on tail.

The APIs for the WBQ.

```
15
16 void submitTask(WorkBalancerQueue* q, Task* _task) {
17     Queue_Enqueue(q, _task); //added to the queue
18     q->task_count += ceil(_task->task_duration / CYCLE); //updated the queue task count
19     //printf("Submitting task with id %s", _task->task_id);
20 }
21
22 // This method removes
23 // the next available job from the tail end of the queue and can be only
24 // called by the owner thread. When the WBQ is empty it returns NULL.
25 Task* fetchTask(WorkBalancerQueue* q) {
26     Task* tmp_task = malloc(sizeof(Task));
27     if (Queue_Dequeue(q, tmp_task) == -1)
28     {
29         //queue was empty
30         free(tmp_task);
31         return NULL;
32     }
33
34     q->task_count -= ceil(tmp_task->task_duration / CYCLE);
35
36     return tmp_task;
37 }
38
39 Task* fetchTaskFromOthers(WorkBalancerQueue* q) {
40     // Lock the mutex before accessing the queue
41     Task* tmp_task = malloc(sizeof(Task));
42     if (Queue_Steal_From_Back(q, tmp_task) == -1)
43     {
44         //queue was empty
45         free(tmp_task);
46         return NULL;
47     }
48     q->task_count -= ceil(tmp_task->task_duration / CYCLE);
49     return tmp_task;
50 }
51
```

Submit task, enqueues the given task and updates task count of queue. (Task count is how many cycles it takes to finish it)

Fetch allocates a Task and dequeues. Updates the task count.

FetchTaskFromOthers allocates a task and uses Steal_From_Back. Updates the task count.

Simulator:

The simulator has important extern variables, initialization and the function used by threads(core simulators).

```
11 extern int stop_threads;
12 extern int finished_jobs[NUM_CORES];
13 int flag_array[NUM_CORES]; //I guess it has the flags of overburdened threads
14 extern WorkBalancerQueue** processor_queues; //this will have every threads stuff
15 enum {NORMAL, HEAVY_BURDEN};
```

Stop_threads is a simple int that is checked in main. When the count of every finished job equals the created jobs this is the flag to stop the threads.

Finished_jobs array keeps track of which thread finished how many jobs, it is updated in main. Its main purpose is to keep track of total number of finished jobs for stop_threads.

Flag_array keeps a overburdened or normal flag for every queue.

Processor_queues is the array of every WBQ

I enumerated NORMAL and HEAVY_BURDEN to make it more readable.

```

void initSharedVariables() {
    // TODO: Fill in this function according to your needs:
    // .
    // .
    for (int i = 0; i < NUM_CORES; i++)
    {
        WorkBalancerQueue* q = processor_queues[i];
        Queue_Init(q);
        flag_array[i] = NORMAL;
    }
}

```

Every WBQ is created in main. I ran Queue_Init for every one of them. And then set their flags(Tracker for core burdens) to NORMAL.

```

17 void* processJobs(void* arg) {
18     ThreadArguments* my_arg = (ThreadArguments*) arg; //id and a workbalancer
19     WorkBalancerQueue* my_queue = my_arg -> q;
20     int my_id = my_arg -> id;
21     while (!stop_threads) {
22
23         Task* task;
24         int task_set = 0;
25         if (my_queue->task_count > 20)
26         {
27             flag_array[my_id] = HEAVY_BURDEN;
28         }
29         else
30         {
31             flag_array[my_id] = NORMAL;
32         }
33         if (my_queue->task_count < 10)
34         {
35             //look for someone who needs help
36             for(int i = 0; i < NUM_CORES; i++)
37             {
38                 if (flag_array[i] == HEAVY_BURDEN && processor_queues[i]->head != NULL) {
39                     //printf("fetching from other\n");
40                     flag_array[i] = NORMAL;
41                     task = fetchTaskFromOthers(processor_queues[i]);
42                     my_queue->task_count += ceil(task->task_duration / CYCLE);
43                     task_set = 1;
44                     break;
45                 }
46             }
47         }
48         if (!task_set)
49         {
50             task = fetchTask(processor_queues[my_id]);
51         }
52         if (task == NULL)
53         {

```

processJobs is the function that is run by threads.

Hence the simulator for cores. It first gets its id and WBQ as parameters.

And then until the stop flag is raised the main loop turns.

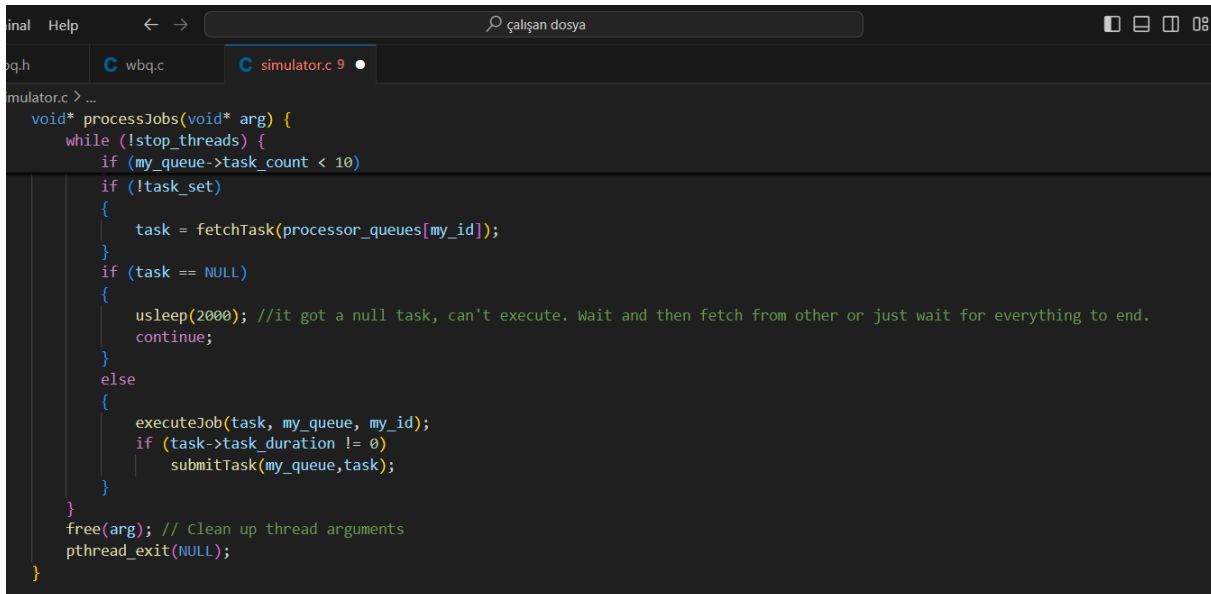
Main Loop:

A pointer for task is created, as well as a task_set variable. First if the size of WBQ belonging to this thread is checked. The flag of the thread is set to HEAVY_BURDEN or NORMAL accordingly. The limit 20 can be changed. I observed that this was the better option. Also I tried another implementation where these checks didn't happen every

loop. However, there were no discernible improvement so the program checks its burden every time.

If the thread has less than 10 cycles of tasks remaining it looks for any thread that may need help. If there is the thread calls `fetchFromOther` and then makes `task_set = 1`

If the task isn't set here the thread continues to fetch from its own WBQ.



```
void* processJobs(void* arg) {
    while (!stop_threads) {
        if (my_queue->task_count < 10)
            if (!task_set)
            {
                task = fetchTask(processor_queues[my_id]);
            }
            if (task == NULL)
            {
                usleep(2000); //it got a null task, can't execute. Wait and then fetch from other or just wait for everything to end.
                continue;
            }
            else
            {
                executeJob(task, my_queue, my_id);
                if (task->task_duration != 0)
                    submitTask(my_queue, task);
            }
        }
    }
    free(arg); // Clean up thread arguments
    pthread_exit(NULL);
}
```

Direct continuation of the last image(same if(!task_set)

If the task the program got is NULL it means that its own queue is empty and there is no queue that is in need. Here the thread just sleeps and then continues from the start of main loop.

If the task isn't NULL it calls `executeJob`. Which basically sleeps for the cycle, and updates `task_duration` and cache affinity. No need to concern ourselves with it. After it is executed if the task isn't finished it is submitted again. Meaning the job is now at the back of the queue.

When while loop end we free the allocated variable and call `pthread_exit` to finish the thread.

Notes

While we haven't touched on it there is actually a drawback to fetching from others. The cache affinity which speeds up tasks is lost when it is fetched from other threads. There definitely could be a logic that predicted if another thread fetching is better due to less workload or worse due to lost cache affinity.

This is not as smart of an idea, I think. Firstly, there actually is an implicit logic. Due to the task lengths and cycle times 20 cycles of jobs mean many different tasks. Hence none run very often and their cache affinity do not get huge boosts. This bring us

to the second point that a thread that has less than 10 cycles of tasks remaining will run that task more often, hence catching up in affinity.