Process Scheduling Using Reinforcement Learning

Ankit Kumar Jain, Pradipta Bora, Harshit Gupta
Department of Computer Science
IIT Bombay
Mumbai, India

Abstract

This project aims to improve the performance of Operating Systems' process scheduling by adding decision-making to the MLFQ (multi-level feedback queue) scheduling algorithm using Reinforcement Learning. The main parameters of interest is the duration for which each task is scheduled which we aim to change using RL. For this we use an Actor Critic based RL algorithm called A2C which is run on a state encoding of the jobs waiting to be scheduled. Further work can be done on using RL and other methods for further changing the other parameters of the MLFQ algorithm.

1 Introduction

Our project aims to look at job scheduling in operating systems through reinforcement learning. The problem requires us to decide which process to schedule at the CPU core given a list of processes. There are a lot of metrics of interest:

1. CPU Utilization(C)
2. Throughput (T)
3. Turnaround time (TA)
4. Waiting time (WT)
5. Response time (RT)

We aim to focus mainly on the turnaround time, which is the time taken to finish the execution of the process and the waiting time. The current systems used in operating systems use heuristics to decide which process to schedule as it is impossible to determine the remaining time needed for an optimal scheduling algorithm. A survey on the currently used scheduling algorithm is given in [1]. The current state of the art is MLFQ (Multi Level Feedback Queue) which uses multiple priority queues to schedule processes. MLFQ may demote/promote a particular process depending on the process's time, allowing other processes to run.

We give a brief introduction of MLFQ in the next section.

1.1 Multi Level Feedback Queue for Scheduling

In MLFQ, as the name suggests, we have many Queues with priorities assigned to each queue. Processes from the highest priority queue are scheduled first, and within the same queue, processes follow any other single queue heuristic, which we will describe later. Each process in a queue needs to run for a minimum number of cpu cycles, which is termed as a quantum. The quantum value is different for each queue, which becomes a parameter for the MLFQ system. Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is

Preprint. Under review.

reduced (i.e., it moves down one queue) and the priorities of processes are reset to the highest after some time period. Within a particular queue, heuristics like Round Robin where processes have a fixed time duration within which they are allowed to run, and then the next process is scheduled in a circular manner. Other heuristics also exist like FIFO (first in, first out) and shortest job first can also be used. Another important concept in MLFQ is boosting, which is a time period after which all the processes are reset and sent to the first queue. This is important because the algorithm we present subsequently will vary based upon what the boosting time is given.

The important weakness of MLFQ scheduler is the inability to define the optimized number of the queues and quantum of each queue also this algorithm has starvation for some process. These factors affect the response time directly. This is where we aim to step in, by using Reinforcement Learning on a state encoding of the current list of processes to be scheduled to determine and change the quantums on the fly. A fixed set of quantums can result in excessively high turnaround time, consider a scenario when one process just has 1 unit of time of processing needed, while another has a large amount of time left which is scheduled first. In this case using a small quantum for all queues will ensure a far lower turnaround time. However in other cases having longer quantums could be more beneficial, when say processes are roughly having similar processing time left. Therefore dynamically changing the quantums based upon past information can certainly improve the scheduling performance.

In this project, we use reinforcement learning to solve this problem. Our encoding of the problem is a partial MDP, by no means a complete representation of the state of the scheduling system, but we hope that the encoding is a close enough model of the decision making process (of choice of quantums) that we can observe good results.

2 Related Work

Seifedine Kadry et al. [2] applied dynamic quantum instead of static quantum in Multi-Level Feedback Queue (MLFQ) based scheduling. The scheme accommodates low priority tasks, which takes a long time for completion and maximizes resource utilization.

In another work, Xiaojie Li et al. [3] proposed improved EDF algorithm which reduces the task switching overhead and number of task switch. Controlled preemption is proposed by Jinkyu Lee et al. [4], which regulates the preemption for better EDF. The approach applies heuristics to decide a better value of preemption time for each task in case of Uni processor systems.

J. Lee et al in [4] proposed a scheduling policy to make effective use of contention free slots. In [5], SoCRATES (SoC Resource AdapTivE Scheduler) is introduced, which is an application of DRL to resource allocation in System-on-Chip (SoC) applications. It maps a wide range of hierarchical jobs to heterogeneous resources within SoC using the Eclectic Interaction Matching (EIM) technique.

Our approach is relatively different and simpler compared to the above, the main motivation for this is that we do not want the system to be taxing as evaluating models for low level OS tasks can take time and so we propose a simple RL model for changing a single parameter of the existing MLFQ system.

3 Formulation of the Problem

Given a set of currently waiting processes L, and a set of queues Q, we can describe the state of the system in terms of a mapping $M : L \to Q$, a mapping of quantums $M_q : Q \to T$ (T is the set for time) and a metadata function $F : L \to D$ where D can be any past information, for our purposes we consider D to be the set of tuples holding time information for the process (time elapsed, time given for processing etc).

We convert the above by considering the following state encoding for our MDP which will be used in RL.

We maintain a vector v_1 which has the total CPU time spent by each process (ordered by time of arrival) and a vector v_2 which has the total burst time of each process (again ordered by time of arrival).

We then create two encoding vectors s_1 ans s_2 from v_1 and v_2 which are then concatenated to give us the state $s = s_1||s_2$.

s_1 and s_2 are made in the following way: We fix a dimension of s_1 say D. Then s_1 is made from v_1 by taking D rolling averages of the entries of v_1. Since the dimension of v_1 can vary, the number of elements in each rolling average will also correspondingly vary. We make s_2 in a similar fashion as well.

The motivation for the above is that we need to have a fixed state size (which is $2D$ in this case) while the number of processes can vary, thus taking a rolling average is one way at fixing the state space while also ensuring that the information that we want (time data) is present.

The action space for our RL agent is simply the set of all possible quantums (which can be normalised or unnormalised). Therefore the agent at all timesteps will be responsible for changing the quantums of the queues.

Finally the reward is given whenever one particular process is finished. The reward of a single process is a weighted average of the parameters as

$$r_p = -(w_1 t_1 + w_2 t_2 + w_3 t_3)$$

where t_1 is response time, t_2 is turnaround time and t_3 is waiting time. At each timestep the reward is simply the sum of rewards that terminated at that timestep.

Once given the above RL formulation of the problem, we can use different RL algorithms for solving the above decision process. Note that the state space is not finite and so one has to resort to function approximators and other Deep RL techniques.

We have used the Actor Critic algorithm A2C here for the RL algorithm. We briefly describe the above algorithm here in the next section before we go and look at the results.

3.1 A2C algorithm

The A2C algorithm is a Deep RL Actor Critic algorithm. Deep reinforcement learning uses a deep neural network (DNN) to represent the agent. Both value based and policy based methods are used in Deep RL. In value-based methods, a value function is learned to evaluate the advantageousness of states or state-actions in order to make the best long-term decisions. In policy-based methods, the goal is to directly develop a decision model (policy function) that predicts the best action for a given state. The actor-critic method tries to combine the best value and policy-based methods by learning a value function and a policy.

In A2C, a single DNN outputs a softmax layer for the policy $\pi(a_t|s_t;\theta)$, and a linear layer for $(s_t;\theta)$. In A2C, multiple agents perform simultaneous steps on a set of parallel environments, while the DNN updated every t_{max} actions using the experiences collected by all the agents in the last t_{max} steps. This means that the variance of the critic $V(s_t;\theta)$ is reduced (at the price of an increase in the bias) by N-step bootstrapping, with $N = t_{max}$. The cost function for the policy is then:

$$\log \pi(a_t|s_t;\theta)[R_t - V(s_t;\theta_t)]$$

where θ_t are the DNN weights θ at time t, $R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_t)$ is the bootstrapped discounted reward from t to $t+k$ and k is upper-bounded by t_{max}. The cost function for the estimated value function is:

$$[R_t - V(s_t;\theta)]^2$$

which uses, again, the bootstrapped estimate R_t. Gradients $\nabla\theta$ are collected from both of the cost functions and then standard optimizers, such as Adam or RMSProp, can be used for optimization.

In our project, we have used OpenAI's stable baselines implementation's A2C algorithm

Now we look at the results and how the agent is evaluated.

4 Results

The weights used are 1 for turnaround time, 0.5 for waiting time and 0.1 for response time.

We needed to create a way to generate instances where our agent is trained and evaluated on. For this we create a sampling process for one particular job. One job is uniquely characterised by the time of arrival and the time it needs on the CPU. We sample both of these from a gaussian with fixed

standard deviation (5 for arrival time and 10 for burst time). Finally we need the number of jobs for a test case which is uniformly sampled from 1 to 20. This can be done to get one step (instance).

The agent is trained on 100000 such instances and after training the following the table gives the evaluation on a particular trained agent on this instance:

Burst Time	Arrival Time
2	14
12	4
12	1
8	7
6	7
1	12
18	5
5	0
4	3
12	3
19	5

This is the representative instance where jobs were sampled randomly. Several other instances were used to test the model. We don't report the results for those instances here but can be easily derived from the code by running the script `run.sh`.

We present the results for various sets of values for number of queues and boosting parameter.

Number of queues = 3, Boosting = 0			
	Mean Turnaround Time	Mean Response Time	Mean Wait Time
Baseline	56.36	27.81	47.36
A2C	45.46	37.84	38.38

Number of queues = 3, Boosting = 3			
	Mean Turnaround Time	Mean Response Time	Mean Wait Time
Baseline	44.27	29.90	35.27
A2C	47.36	21.54	38.36

Number of queues = 5, Boosting = 0			
	Mean Turnaround Time	Mean Response Time	Mean Wait Time
Baseline	56.36	27.81	47.36
A2C	52.09	0.72	43.09

Number of queues = 5, Boosting = 3			
	Mean Turnaround Time	Mean Response Time	Mean Wait Time
Baseline	46.84	28.69	39.76
A2C	51.76	17.84	44.69

5 Conclusion and Future Work

The results show that using RL is quite beneficial in the case when there is no boosting. In case of boosting, the boost interferes with learning leading to multiple next state depending on the current time step number. Hence, the agent is not able to learn properly.

As we have seen, using RL for modifying one parameter of the MLFQ algorithm works quite a lot better than the fixed parameter variants of MLFQ. One can use RL for even more decision making in the MLFQ algorithm, we can use RL to decide the assignment of priorities to the current list of processes based upon the current state. We can still use MLFQ, but instead of determining the priority level heuristically, we can use RL for this. This is left for future analysis.

Another approach could be to use an end-to-end model for scheduling instead of MLFQ. The end-to-end model will decide directly which process to schedule instead of using MLFQ as a proxy.

This is another avenue for research, although this will be certainly more complex than the system given above.

References

[1] Chowdhury, Subrata. (2018). Survey on various Scheduling Algorithms. "Imperial Journal of Interdisciplinary Research (IJIR). 3. 4.

[2] Seifedine Kadry, Armen Bagdasaryan “New Design and Implementation of MLFQ Scheduling Algorithm for Operating Systems Using Dynamic Quantum” Statistics, Optimization And Information Computing. Vol. 3, pp.189-196, June 2015.

[3] Xiaojie Li and Xianbo He “The improved EDF Scheduling Algorithm for Embedded Real-time System in the Uncertain Environment” ICACTE 2010.

[4] Jinkyu Lee and Kang G. Shin, “Preempt a Job or Not in EDF Scheduling of Uniprocessor Systems,” IEEE Transaction on computers, vol. 63, no. 5, May 2014

[5] Sung, Tegg Taekyong, and Bo Ryu. "SoCRATES: System-on-Chip Resource Adaptive Scheduling using Deep Reinforcement Learning." 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 2021