Start-Time Fair Queuing

A Hierarchical CPU Scheduler for Multimedia Operating Systems

Why a separate scheduler for MMOS?

The need for supporting variety of hard and soft real time,as well as best effort applications in a multimedia computing environment requires an operating system framework that:

- Enables different schedulers to be employed for different application classes.
- Provides protection between the various classes of applications.

  Since multimedia applications convey meaning only when presented continuously in real-time,they require OS to allocate resources such as CPU,I/O, disk etc. in a predictable manner as well as provide QoS.

What are the application classes we are dealing with?

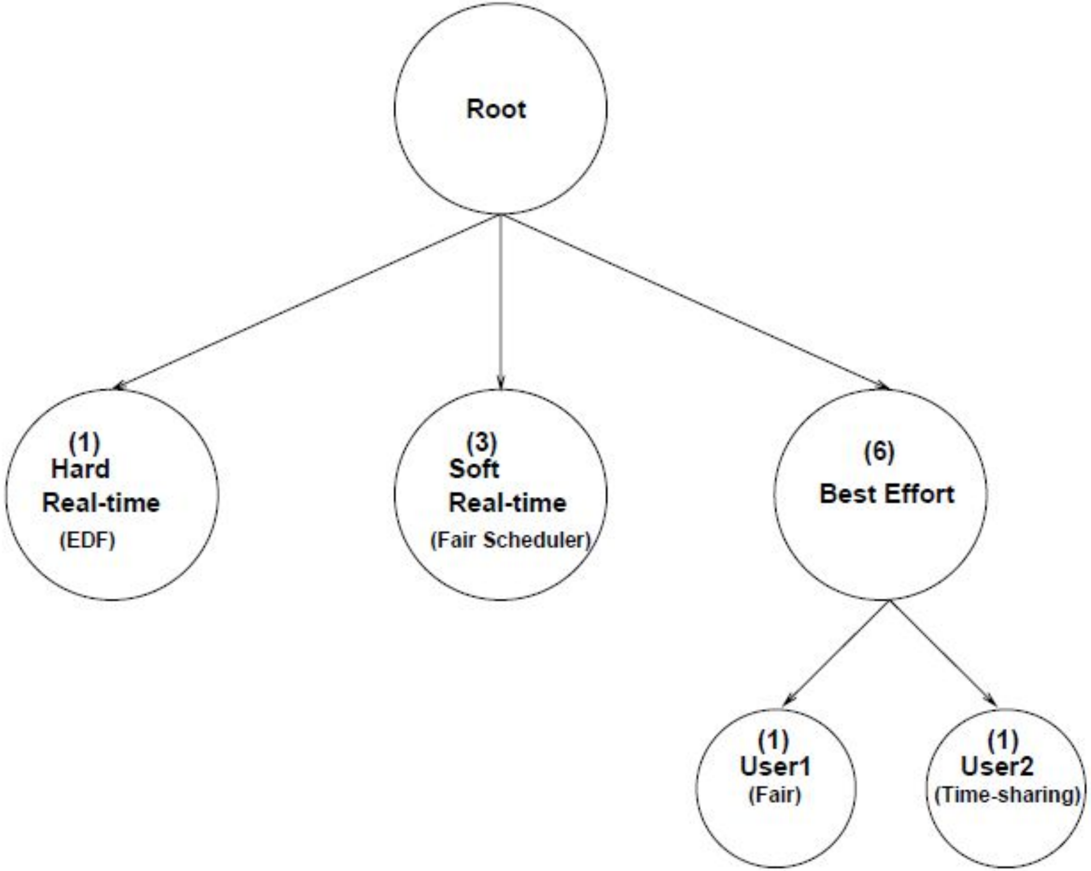
- Hard real time applications: These applications require an operating system deterministically guarantee the delay that may be experienced by various tasks. Conventional schedulers such as the Earliest Deadline First (EDF) are suitable for such applications.Eg: Flight Control Systems

- Soft real time applications: These applications require an operating system to statistically guarantee QoS parameters such as maximum delay and throughput. Since a large number of such applications are expected to involve video.Eg: VBR video.

- Best effort applications: Many conventional applications do not need performance guarantees, but require the CPU to be allocated such that average response time is low while the throughput achieved is high.

What’s the problem with already existing MMOS schedulers?

EDF(Earliest Deadline First) and RMA(Rate Monotonic Algorithm) schedulers do not provide any QoS guarantee when CPU bandwidth is overbooked. Furthermore, their analysis requires the release time, the period, and the computation requirement of each task (thread) to be known a priori. Consequently, although appropriate for hard real time applications, these algorithms are not suitable for soft real time multimedia applications.The requirements for supporting different scheduling algorithms for different applications as well as protecting application classes from one another leads naturally to the need for hierarchical partitioning of CPU bandwidth.



Aim:

1. Support all types of applications on a single server.
2. Distribute the cpu bandwidth according to their weight.
3. Cpu allocation must be dynamic i.e. if there no jobs in one class of application then cpu should not be allocated to it and remaining bandwidth should be shared among other applications proportionally.

For a time interval Normalized work =

no. of instructions executed / weight(priority)

Start time fair queuing algorithm:
We are being fair about the no. of instructions executed by a thread for a given time interval considering their weight(priority).

Basic Idea: whenever this algorithm is asked for a thread to schedule it finds a thread which has completed minimum normalized work from the point it is ready to execute.

We need some formal algorithm to track the normalised work.

When quantum q^j_f is requested by thread f , it is stamped with start tag S_f , computed as:

$$S_f = max[v(A(q_f^j)), F_f]$$

v(t) is the virtual time at time t and F_f is the finish tag of thread f, when the j^{th} quantum finishes execution, F_f is incremented as:

$$F_f = S_f + \frac{l_f^j}{r_f}$$

Start tag: updated when

1. Thread is arrived
2. Thread is preempted
3. Thread is unblocked

Finish tag: updated when

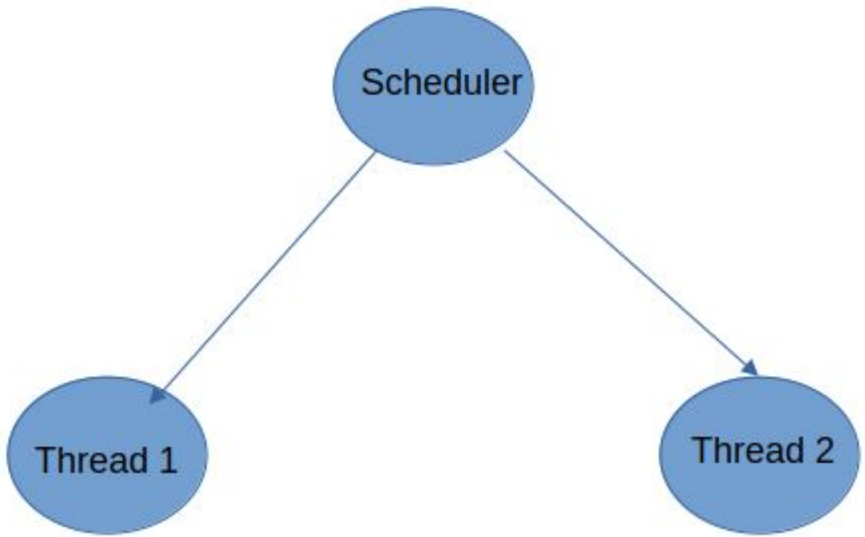
1. A quantum of that thread finishes execution

Virtual time : updated when

1. A thread starts execution
2. Cpu becomes idle

Assume time quantum = 10 ms

Time = 0
Virtual time = 0



Thread 1 and 2 arrived so start tag and finish tag of both initialized and set to zero.

Algorithm will be run
Ties are broken arbitrarily
Assume Thread 1 is selected

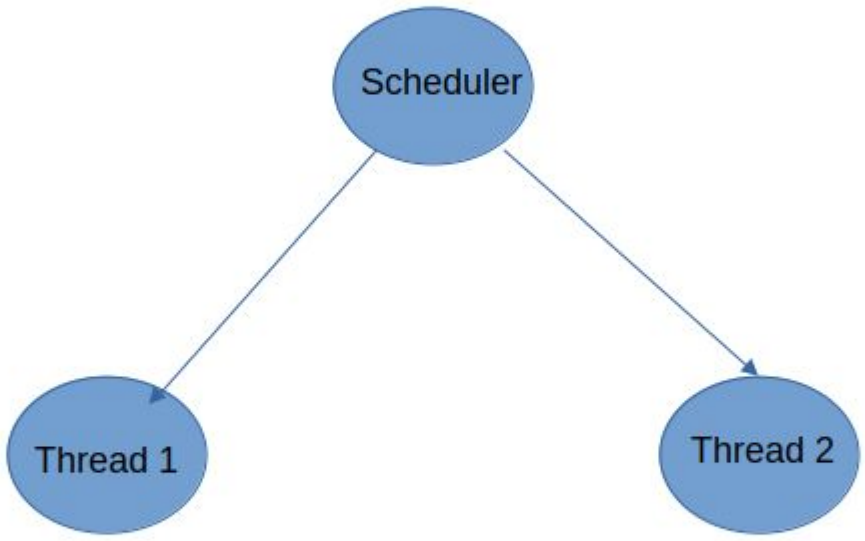
Weight = 1
Start tag = 0
Finish tag = 0

Weight = 2
Start tag = 0
Finish tag = 0

Assume time quantum = 10 ms

Time = 0 to 10 ms
Virtual time = 0 (start tag of thread 1)

During execution of Thread 1



Weight = 1
Start tag = 0
Finish tag = 0

Weight = 2
Start tag = 0
Finish tag = 0

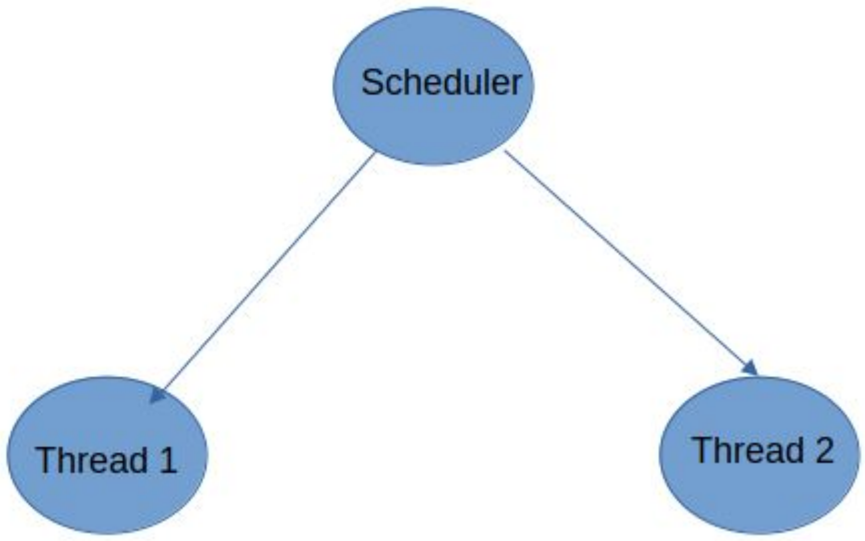
Assume time quantum = 10 ms

Time = 10ms
Virtual time = 0

After the execution of the quantum

Start tag and finish tag updated

Algorithm will be run
Thread 2 has minimum start tag
Hence it is selected



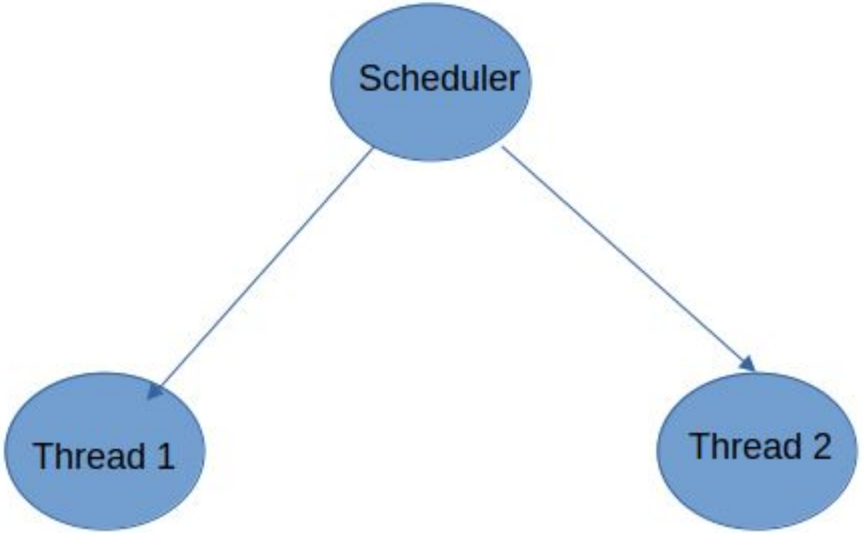
Weight = 1
Start tag = 10
Finish tag = 0+ 10 /1 = 10

Weight = 2
Start tag = 0
Finish tag = 0

Assume time quantum = 10 ms

Time = 10 ms to 20 ms
Virtual time = 0 (updated)

During execution of thread 2



Weight = 1
Start tag = 10
Finish tag = 10

Weight = 2
Start tag = 0
Finish tag = 0

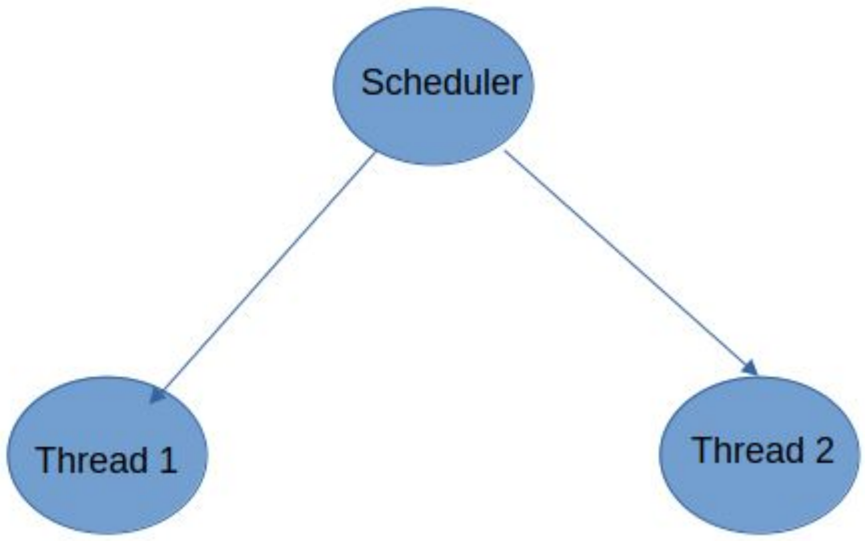
Assume time quantum = 10 ms

Time = 20ms
Virtual time = 0

After the execution of the quantum

Start tag and finish tag updated

Algorithm will be run
Thread 2 has minimum start tag
Hence it is selected



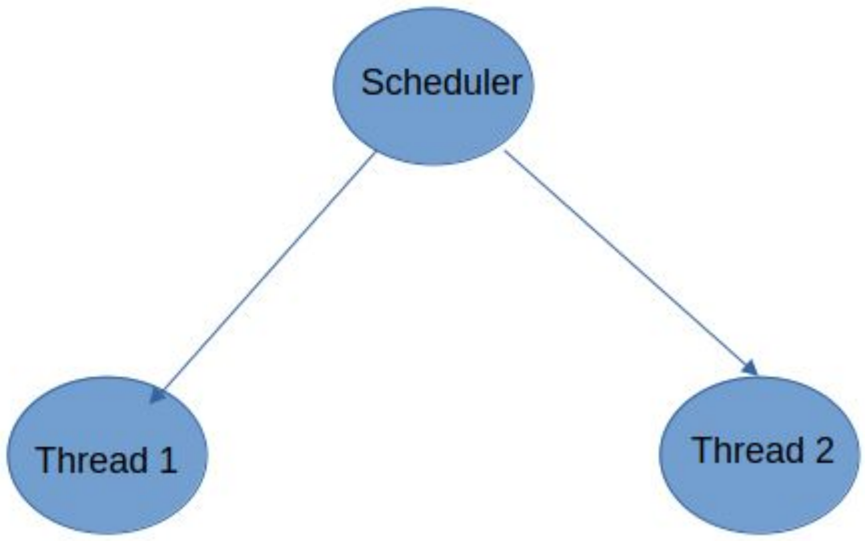
Weight = 1
Start tag = 10
Finish tag = 10

Weight = 2
Start tag = 5
Finish tag = 0 + 10 /2 = 5

Assume time quantum = 10 ms

Time = 20 ms to 30 ms
Virtual time = 5 (updated)

During execution of thread 2



Weight = 1
Start tag = 10
Finish tag = 10

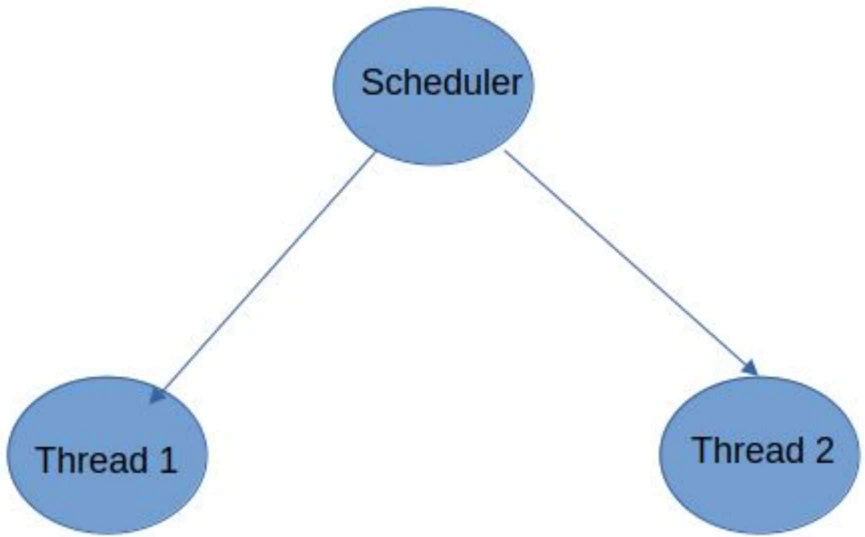
Weight = 2
Start tag = 5
Finish tag = 5

Assume time quantum = 10 ms

Time = 30ms
Virtual time = 5

After the execution of the quantum

Start tag and finish tag updated



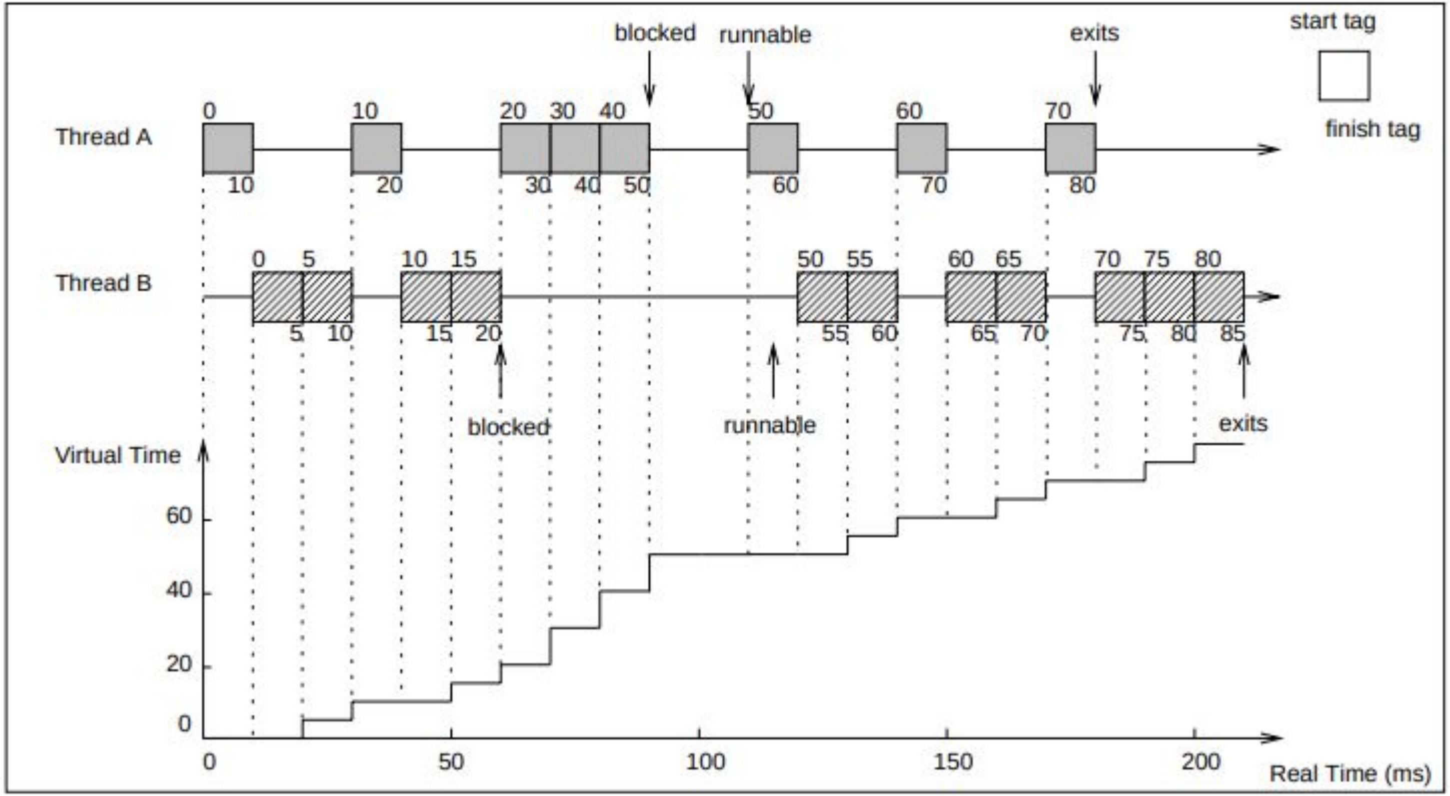
Algorithm will be run
Ties will be broken arbitrarily
Similarily we will be continuing

Weight = 1
Start tag = 10
Finish tag = 10

Weight = 2
Start tag = 10
Finish tag = 5 + 10 /2 = 10

Observation :

If a thread does more normalised work in a given quantum it is made to wait until other threads’ normalised work doesn't become same or more.

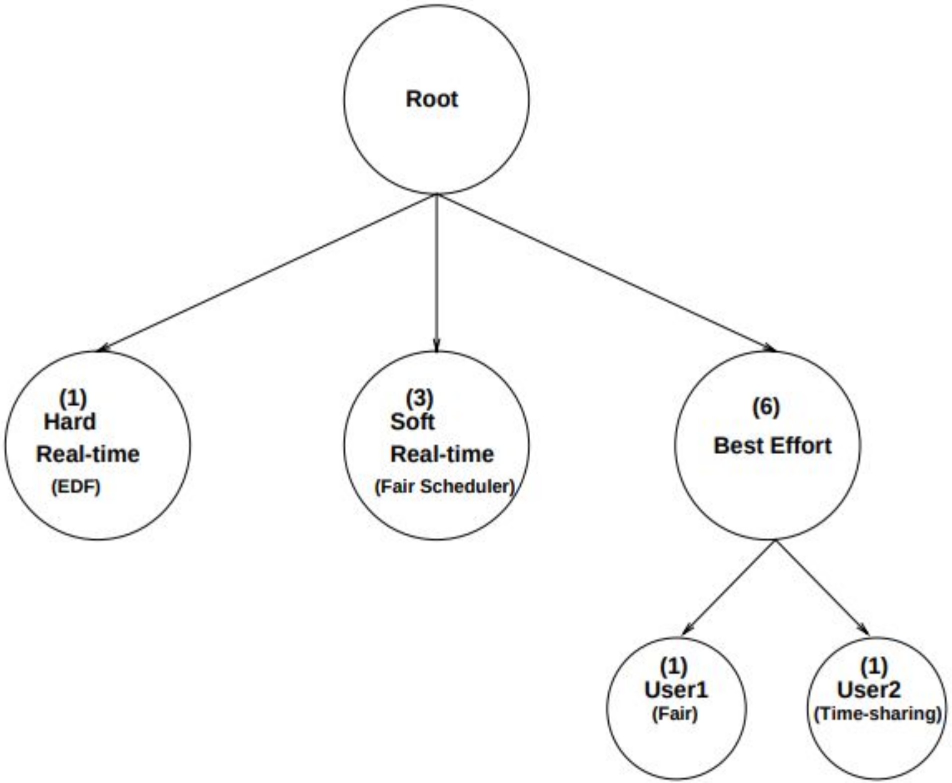


start tag
finish tag
blocked
runnable
exits
Thread A
0
10
20
30
40
50
60
70
80
Thread B
5
15
55
65
75
85
blocked
runnable
exits
Virtual Time
60
40
20
0
0
50
100
150
200
Real Time (ms)

What happens when a thread is unblocked?

If we do not update the start tag value since the delay is unpredictable, after the thread is unblocked it will ask for more cpu time to equate its normalized work.

If the delay is high we set it to the virtual time at the time when it was unblocked.



Solution:

We have a arranged the classes of application in a Hierarchical manner.

Each leaf node will contain threads belong to that class.

When scheduler is asked to schedule a thread. At the root node we run the SFQ to determine from which class the executing thread should come. We do this recursively until a leaf node is encountered.

Leaf nodes can contain its own scheduling algorithm.

After we reach the leaf node we call the scheduler present there which will provide the thread to execute.

THROUGHPUT GUARANTEE

SFQ provides bounds on maximum delay incurred and minimum throughput achieved by the threads in a realistic environment.

In the majority of operating systems, the handling of hardware interrupts is given top priority. As a result, the CPU's effective bandwidth experiences fluctuations over time. SFQ (Stochastic Fair Queuing) offers a means to establish limits on both delay and throughput even in such a scenario.

Two Models to Quantify delay.

1. Fluctuation Constrained (FC) server:

   A server is a Fluctuation Constrained (FC) server with parameters (C, δ(C)), if for all intervals [t1; t2] in a busy period of the server, the work done by the server, denoted by W(t1; t2), satisfies:

$$w(t1, t2) \geq C * (t1 - t2) - \delta(C)$$

C: average rate of execution (instructions/s)

δ(C): Fluctuations in instructions

2. Exponentially Bounded Fluctuation (EBF):

Server with parameters (C, B, α, δ(C)), if for all intervals [t1, t2] in a busy period of the server, the work done by the server, denoted by W(t1, t2), satisfies.

$$P(W(t1, t2) < C * (t2 - t1) - \delta(C) - \gamma) \leq Be^{-\alpha\gamma}$$

Intuitively, the probability of work done by an EBF server deviating from the average rate by more than γ, decreases exponentially with γ.

If the CPU is a FC server, then SFQ guarantees that the time at which quantum qfj of thread f will complete execution, denoted by LSFQ (qfj), is given as:

$$L_{SFQ}(q_f^j) \leq EAT(q_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C}$$

(1) (2) (3) (4)

If the CPU is an EBF server, then SFQ guarantees that LSFQ (qfj) is given as follows:

$$P(L_{SFQ}(q_f^j) \leq EAT(q_r^j, r_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_f^j}{C} + \frac{\delta(c)}{C} + \frac{\gamma}{C}) \leq 1 - Be^{-\alpha\gamma}$$

PSEUDO CODE

```
Class Node {
        ID
        Parent_pointer
        Children[]
        function_ptr NodeScheduler
        function_ptr Updater
        bool   is_leaf
        Virtual_time
}
```

A container structure which can represent a thread as well as the intermediate nodes.

- The **function pointers** make is flexible to choose any scheduling policy for a node. Thus **RR** as well as **SFQ** can be used in the same tree.
- Every edge is doubly linked thus making it easier to update the state of the tree after every thread execution.

```
sfq_selector(children_Nodes[]) {
        min_start_tag = inf
        Node* child = NULL

        for (Child ∈ children_Nodes) {
                if(min_start_tag > Child.start_tag) {
                        min_start_tag = Child.start_tag
                        child = Child
                }
        }

        Node.Virtual_time = child.start_tag

        return child
}
```

The function selects a node for execution among the children nodes of a parent.

- The function will be included as a **function pointer** in the node structure. In our implementation, **SFQ** policy was used, but, any policy can be implemented, keeping in mind the node structure.

```
updater(Node, lengthQuantum) {
        if(Node == NULL) {
                return
        }

        Node.finish_tag = Node.start_tag + lengthQuantum / Node.weight
        Node.start_tag = max(Node.Virtual_time, Node.finish_tag)

        updater(Node.parent, lengthQuantum)
}
```

The function **updates the state of the tree** after a thread is executed for a time quantum. In this case the updater function is written according to the **SFQ policy**, but every node can have it’s own policy to update its state, thus determining the priority of the node for next execution.

```
insert(newNode, position[], root, i) {
        if(i == sizeof(position)) {
                root.children = root.children U newNode
        }

        for(child ∈ root.children) {
                if(position[i] == child.ID) {
                        newNode.start_tag = root.Virtual_time
                        newNode.finish_tag = 0
                        insert(newNode, position[], child, i+1)
                }
        }
}
```

Inserts a new node in the tree. It also takes a position[] array containing the path to follow to insert to node. The position array contains the IDs of the ancestor nodes of the new node.

```
block(Node) {
        parent_node = Node.parent

        for(child ∈ parent_node.Children) {
                if(child == Node) {
                        parent_node.Children = parent_node.Children \ child
                }
        }

        blockedQueue.push(Node)
}
```

The function blocks a particular node from execution, thus removing it from the tree. It requires a pointer to the node.

```
void threadWakeup() {
    while(blockedQueue != ϕ) {
        thread = blockedQueue.top
        if(thread.unblockTime <= timer) {
            blockedQueue.pop
            insert(thread)
        }
        else break
    }
}
```

This function wakes up the thread from the *blockedQueue* according to their wakeup time. The *blockedQueue* is sorted in ascending order of the wakeup times of blocked threads, to avoid searching the entire queue. As soon as a thread with wakeup time greater than current time is encountered, the loop breaks.

TIME COMPLEXITY ANALYSIS

If every node has x children, the search space reduces to **1/x** after every iteration.

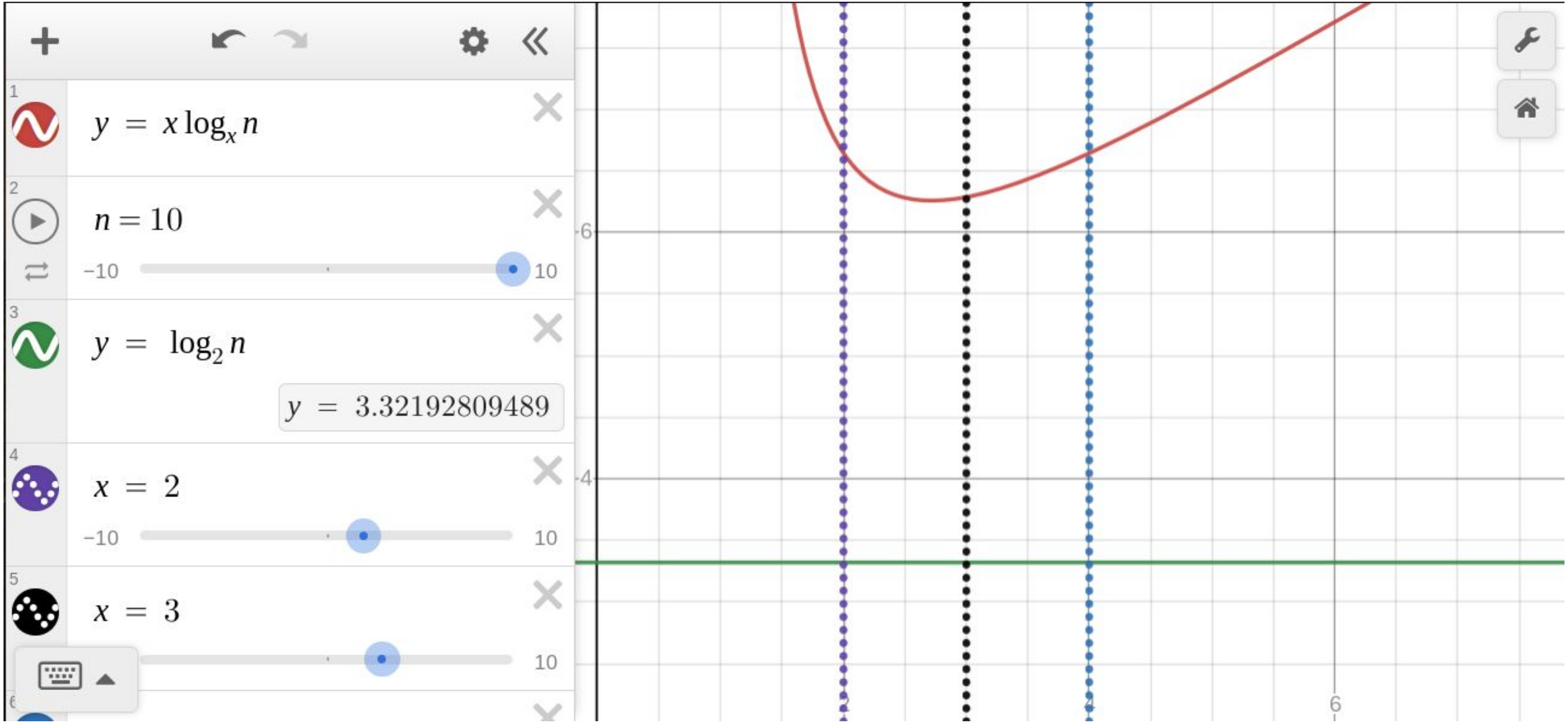
Max height of the tree:- $\mathbf{log_x n}$

Time to search for shortest start_tag child:- **O(x)** (linear search) or **O(logx)** (using a heap)

Overall time complexity: $\mathbf{O(\log x * \log_x n) = O(\log n)}$

n: number of threads in the tree.

SCHEDULING ALGORITHM TIME COMPLEXITY



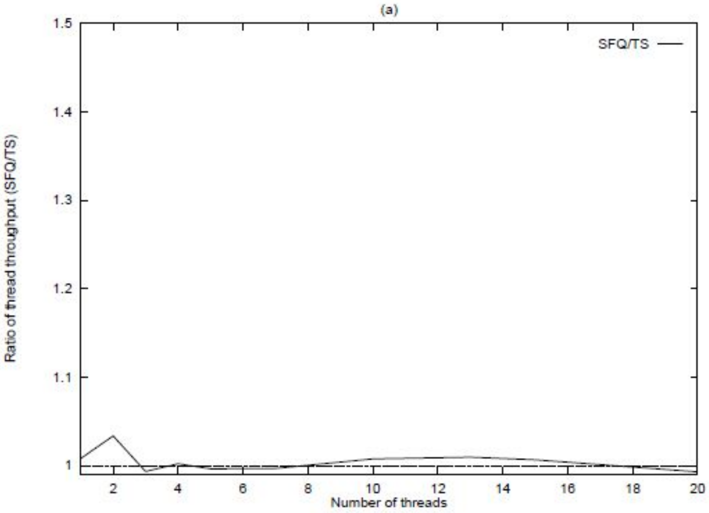
Experimental Evaluation

SFQ and time scheduling algorithm(SVR-4) were compared over various aspects,which are :

- Scheduling Overhead
- Hierarchical CPU Allocator
- Dynamic Bandwidth Allocation

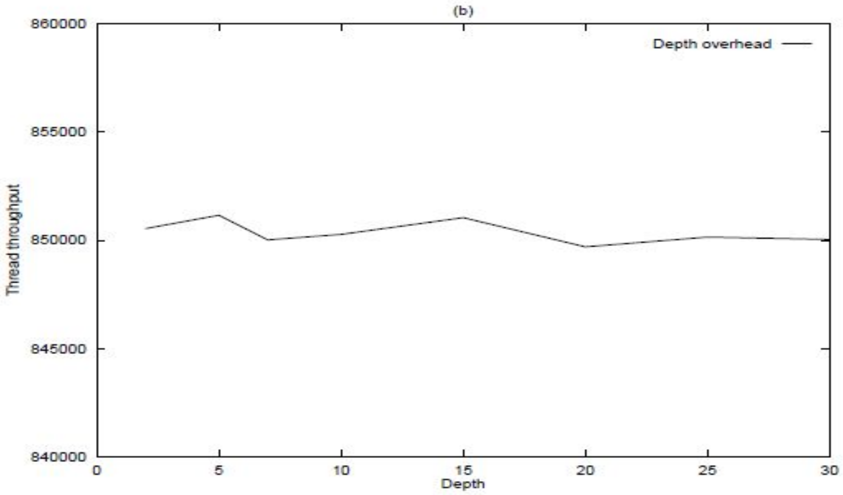
Scheduling Overhead

Number of threads in SFQ node was varied from 1-20.This was repeated 20 times with a time quantum of 20ms. As we can observe the Ratio of throughput in SFQ scheduling was within 1% of that of an unmodified kernel.



Scheduling Overhead

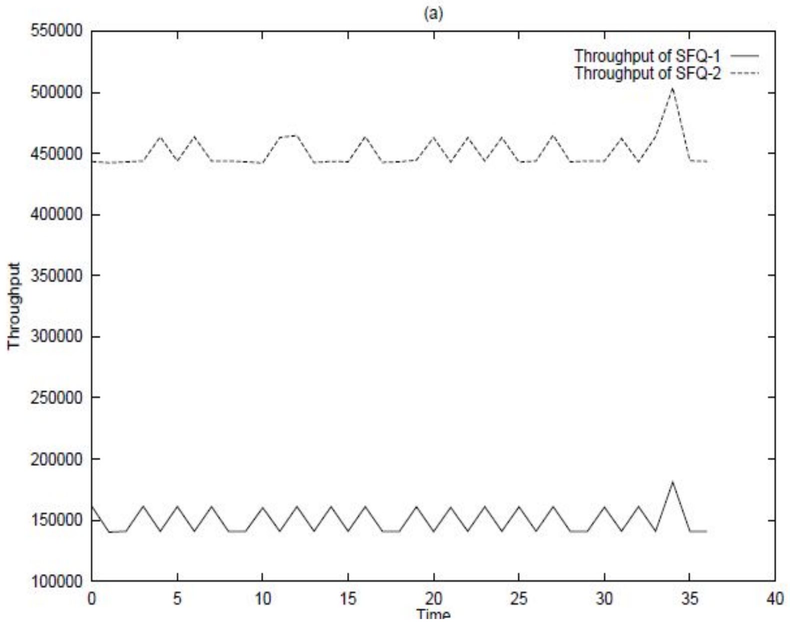
To evaluate the impact of the depth of the scheduling structure, the number of nodes between the root class and the SFQ1 class was varied from 0 to 30.As we can see in spite of that the throughput remains within 0.2% that of unmodified kernel.



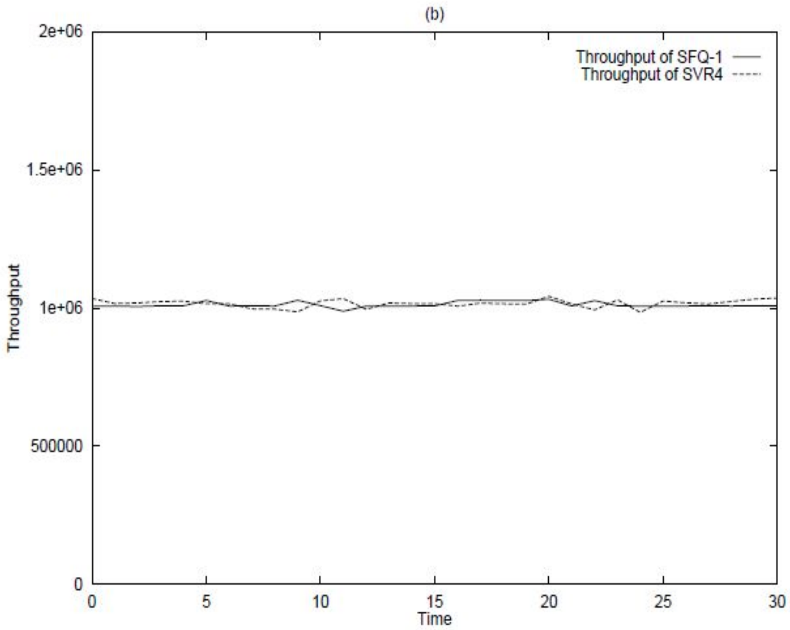
Hierarchical CPU allocation

- Two threads executing the Dhrystone benchmark were added to leaf nodes SFQ1 and SFQ2 with weights as 2 and 6 respectively. Their throughput comes out to be in ratio 1:3, demonstrating that SFQ achieves fair allocation.

- Further one node executing SFQ and one SVR4 with equal weight were measured in terms of throughput and it was observed that both had the same throughput. This is in contrast to the standard SVR4 scheduler where a higher priority class, such as the realtime class, can monopolize the CPU.



(a)
Throughput of SFQ-1
Throughput of SFQ-2
Throughput
Time
550000
500000
450000
400000
350000
300000
250000
200000
150000
100000
0
5
10
15
20
25
30
35
40

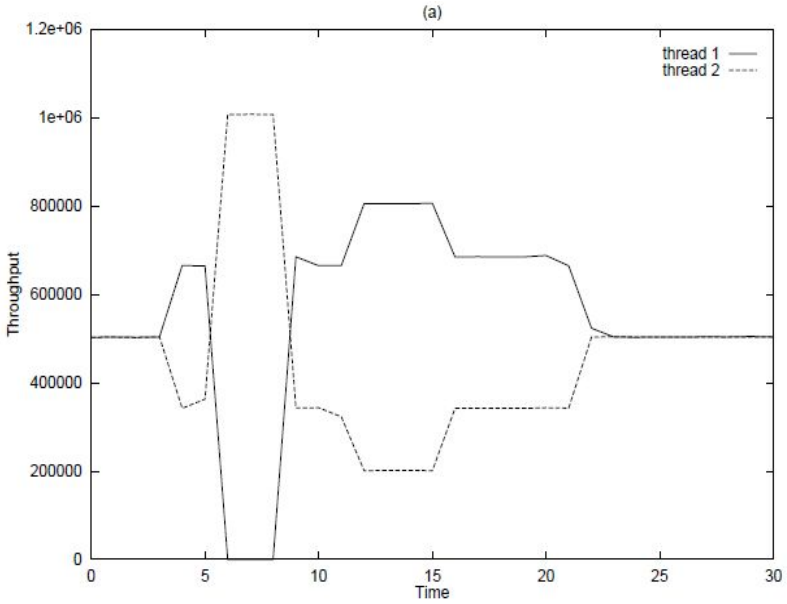


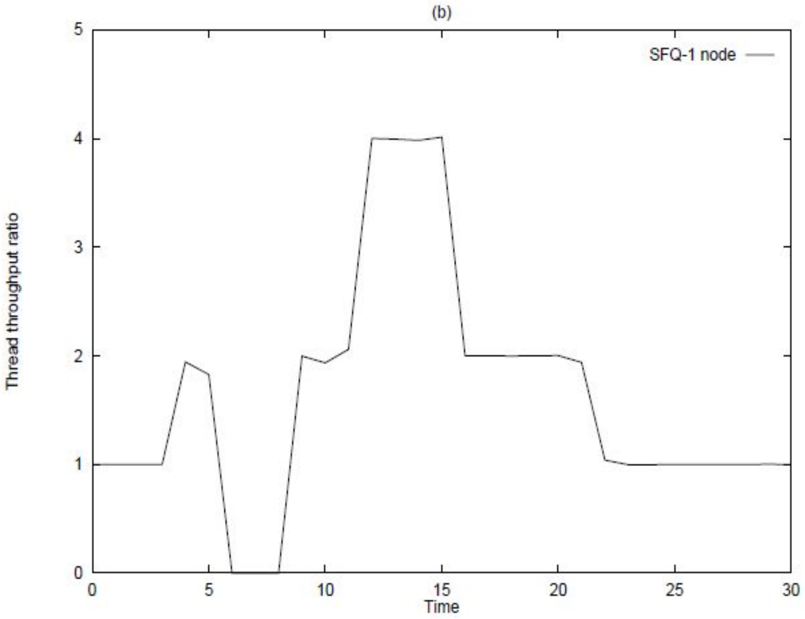
(b)
Throughput of SFQ-1
Throughput of SVR4
Throughput
Time
2e+06
1.5e+06
1e+06
500000
0
5
10
15
20
25
30

Dynamic Bandwidth Allocation

- If we execute two threads with same initial priority and vary their priority along execution, SFQ ensures to adjust the CPU bandwidth allocation according to live priority conditions.

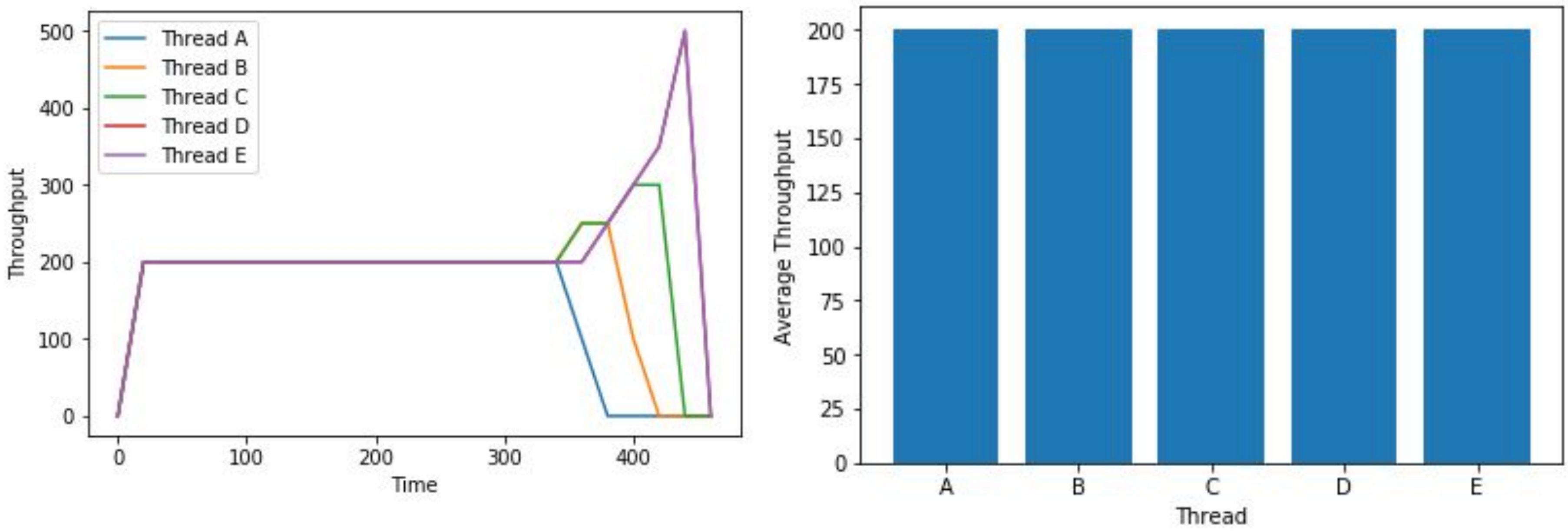
Below figures illustrate that SFQ can achieve fairness of bandwidth allocation in presence of dynamic variation in weight assignment.


(a)
1.2e+06
1e+06
800000
600000
400000
200000
0
Throughput
0
5
10
15
20
25
30
Time
thread 1
thread 2

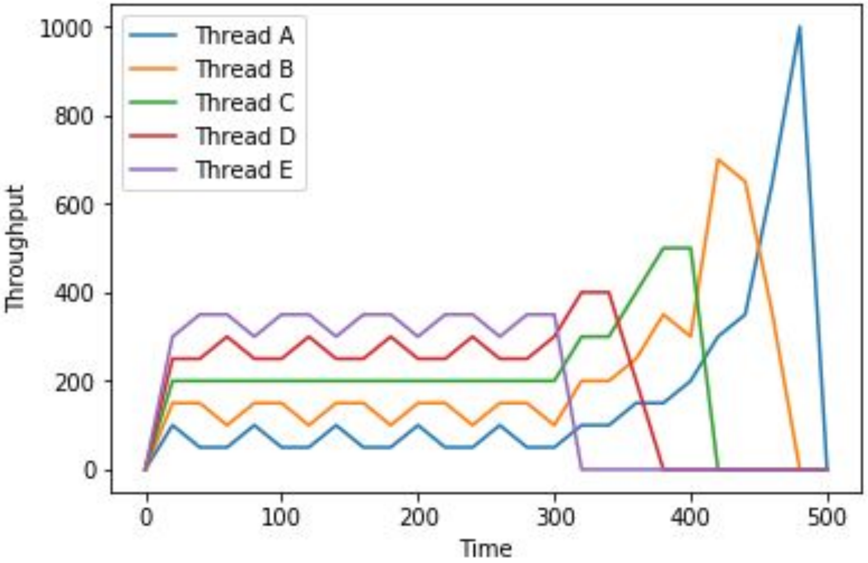

(b)
5
4
3
2
1
0
Thread throughput ratio
0
5
10
15
20
25
30
Time
SFQ-1 node

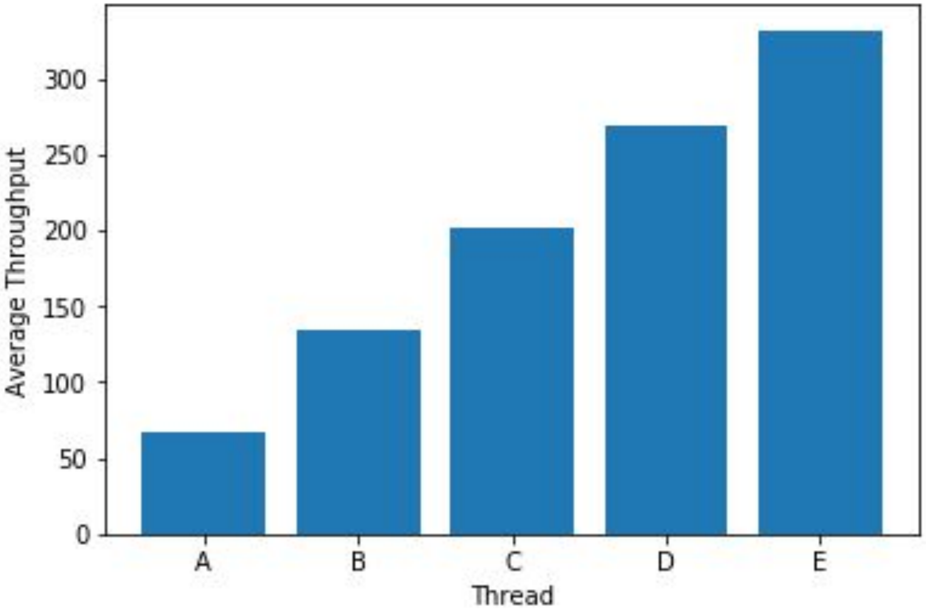
OUR OBSERVATIONS:

1. Same weightage threads
2. Different weightage threads
3. Dynamic bandwidth allocation

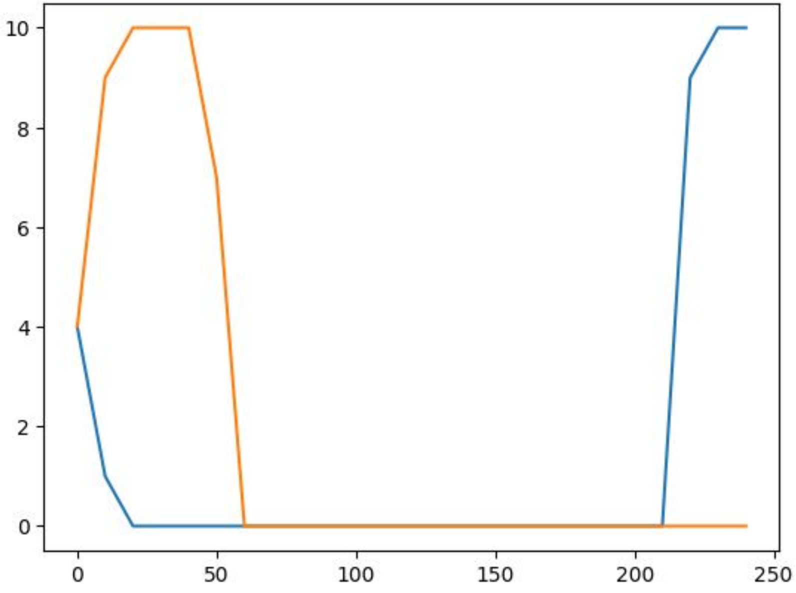


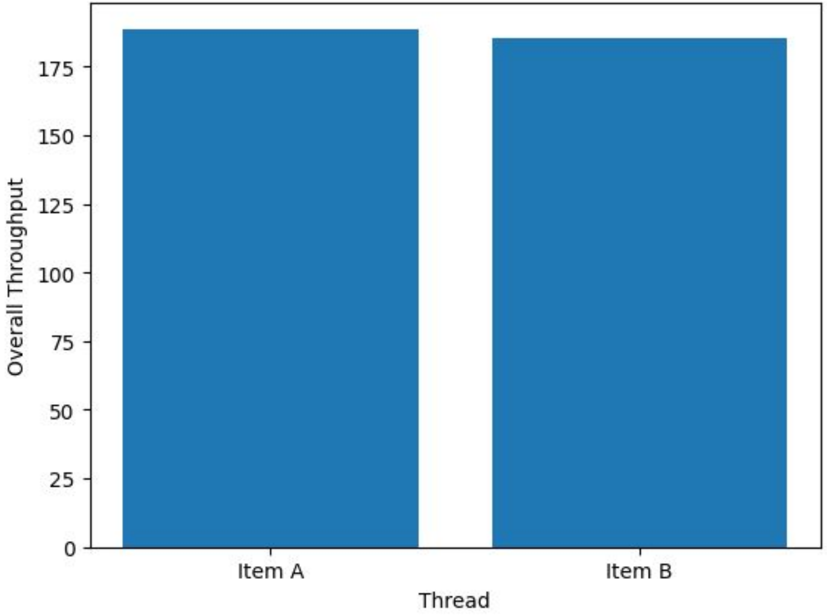
5 Threads running with same priority have equal average throughput.
The threads have different process times in the order: A < B < C < D < E





5 Threads running with different priority have proportional average throughput. The threads have linearly varying weights. W = n
All the threads have same process times.





Two threads of same priority have the same overall throughput even if one thread is blocked for a certain period of time.