Project Report

SSWT- Disk Scheduling in Linux Kernel

6/10/2013

Santa Clara University
Rima Deodhar and Shruti Naik

Index

SSWT – Disk Scheduling in the Linux Kernel

INTRODUCTION

Objective:

The objective of our project is to improve fairness of shortest seek time first (SSTF) algorithm. SSTF is a common disk scheduling algorithm. When multiple processes request for data from the disk, it is the job of the operating system to assign the requests to the disk in some order. This order is determined by the disk scheduling algorithms. The hallmark of a good scheduling algorithm is reduced response time. SSTF tries to achieve this goal by scheduling process requests with minimal seek distance. However, while it exhibits a lower response time, it is susceptible to unfairness. Our objective is to modify the algorithm so as to improve this shortcoming.

Problem:

As mentioned in the earlier section, SSTF is susceptible to unfairness. The working of the algorithm dictates the requests to be assigned to the disk in ascending order of their seek distance from the current position of the disk read/write head (henceforth referred to as the disk head). Consider a request for data located on the innermost cylinder while the disk head is positioned on the middle cylinder. If requests for data from the middle cylinders keep coming in, the request for the innermost cylinder data will potentially never be scheduled. This leads to starvation or undue delay of requests for data from far off cylinders. This is a very serious drawback and we attempt to improve on this.

Relation to class:

This project involves modifying the Linux kernel code base and writing a new algorithm that will improve the fairness of Shortest Seek Time First Algorithm. This will help us get hands on experience on Linux kernel coding and improve our knowledge of disk scheduling algorithms.

Drawbacks of other approaches:

The algorithm in [1], named Circular-SSTF(C-SSTF), attempts to improve the fairness of SSTF. The idea in this algorithm is when a request for an extreme cylinder (say, outermost) is satisfied, instead of satisfying the next request with the shortest seek distance, the disk head moves to the other extreme (innermost cylinder) and satisfies any pending requests for data from that extreme cylinder. This ensures that requests from extreme cylinders are not ignored endlessly. While it solves the problem of starvation, it does not do much to improve overall fairness as the algorithm waits until an extreme request has been scheduled before other extreme requests are satisfied.

In contrast, in [2] which is for real time systems, the requests are categorized into multiple groups like urgent group based on their priorities. The priorities are determined based on their deadlines. This approach is efficient and ensures

fairness. However, it is suitable only for real-time systems. We attempt to provide a more generic solution.

Advantages of our approach over other approaches:
Our idea is to take into consideration arrival time in addition to seek distance for scheduling our disk request. We maintain a predefined threshold value (TV). Any requests that is waiting in the queue for longer than the TV is given higher preference for scheduling. This will ensure that no request waits endlessly for the disk access. We believe that this will reduce the unfairness inherent in Shortest Seek Time first and will provide a more generic solution irrespective of the type of system. However, coming up with the correct value of TV will be crucial here.

Problem Statement:
Our project focuses on improving the Shortest Seek Time First algorithm with respect to fairness. Shortest Seek Time First is a very good approach to reducing the response time. Our attempt is to improve the fairness while retaining the inherent advantages of Shortest Seek Time First.

Area or scope of investigation:
Given the project topic, it is necessary to perform the following investigation:

- Perform background research on Linux kernel scheduling algorithms.
- Source code analysis to gain a deeper understanding of the disk scheduling branch of the Linux kernel.
- Understanding the steps involved in integrating our algorithm into the kernel.

THEORETICAL BASES AND LITERATURE REVIEW

Problem Definition:
Shortest Seek Time First is one of the widely used disk scheduling algorithms in Linux. It serves the request that has the shortest seek distance first. Due to this, requests that have long seek distances from the current position of the disk head need to wait for a longer time in the queue. This might result in starvation of those requests. Thus Shortest Seek Time First exhibits the problem of unfairness to the requests with long seek distance.

Theoretical Background:
In this algorithm, the scheduler picks the next request based on the shortest seek time from the current head position. Seek time is the amount of time taken to move the disk head to the requested position from its current position. Computing this time accurately is very difficult. However, this time is directly proportional to the seek distance between the requested track and the current track on which the disk head is positioned. So the algorithm uses this parameter and minimizes the seek distance which in turn reduces the seek time. While it succeeds in its objective of reducing the average seek distance for a given batch of requests, it tends to favor requests involving the cylinders with a high request rate as opposed to less frequently requested cylinders.

For example, if the disk head is servicing a request positioned on the middle cylinders of a circular disk and a request comes in for data on an extreme cylinder. However, before this request can be serviced, 50 requests come in for cylinders located near the middle. This extreme cylinder request will be serviced after those 50 requests even though it arrived before them. This is why it is considered unfair. In an even more potentially catastrophic situation, the extreme request cylinder could be delayed forever, if the requests for the middle most cylinders come in an unending stream. While such a situation may never occur in the real world, the drawbacks exposed by it of SSTF cannot be ignored. Some mechanism is necessary to ensure this problem is mitigated to a certain extent if not completely.

Related Research:
Our research was initiated by reading journals and papers on disk scheduling. We read papers [1], [2] and [3] in depth to understand their approach on disk scheduling. We explored the advantages and disadvantages of each of the approaches. With the help of these papers we came up with our own algorithm that we believe will help in resolving the problem of unfairness of Shortest Seek Time First Algorithm. As our project involves writing a new algorithm in the kernel we referred to [7] in order to understand how write kernel programs. We referred to [8] to understand how to download, compile and execute our own kernel code in Linux.

Advantages and disadvantages of those results:
As mentioned earlier with respect to existing approaches, the paper [2] provides a good solution for categorizing requests and scheduling them using SSTF. It improves the fairness, however the ideas are suitable for a real time system. In our project, we attempt to provide a more generic solution that does not take into consideration the complexities of a real-time system.
In [3], the authors attempt to improve on average seek distance, response time and seek time by using a fuzzy logic system. The algorithm proposed is robust and succeeds in its objectives in comparison to the conventional algorithms (including SSTF). However, implementing fuzzy logic systems is an involved process and exceeds the scope of our project.

Proposed solution:
Every incoming request is assigned a counter that will determine how long the request is waiting in the queue. Initially the counter is set to zero. In every scheduling session, if that request is not served its counter increments. There is a predefined threshold value (TV) that is used to compare against the counters of each request. In every scheduling session, we first check the counters for each request. If the counter is greater than TV highest priority is given to that request and it is served. Else the request with shortest seek time is served. By giving higher priorities to requests that have been waiting for a long time we can avoid the problem of unfairness in Shortest Seek Time First.

Difference between proposed solutions and other solutions discussed:
Our solution attempts to make use of arrival time and TV to improve the fairness of Shortest Seek Time First. As mentioned earlier, our solution differs from [1] because it takes into account the arrival time of requests which is not a parameter under consideration in this paper. It differs from [2] because we are not using fuzzy logic to solve the problem of unfairness for Shortest Seek Time First. And it differs from [3] because our solution is not limited to real time systems and is more generic, irrespective of the underlying system.

Advantages of the proposed solution:
As mentioned in the earlier sections, the proposed solution takes into account the waiting time of a request. In particular, it maintains a counter of the number of scheduling assignments that a particular request has waited for. Once this counter exceeds a threshold, the request is assigned irrespective of its seek distance.
The algorithm is fairly simple to understand and intuitive. It attempts to reduce the waiting time of a request while also ensuring that the fairness aspect does not override the optimizations of SSTF. This is achieved by means of a predefined threshold. However, for this solution to be successful, the threshold value needs to be chosen wisely.

HYPOTHESIS

Positive hypothesis:
Our goal is improve fairness of Shortest Seek Time First disk scheduling algorithm. With the help of arrival time and threshold values we intend to implement a new algorithm that will provide fair scheduling of requests in SSTF.

METHODOLOGY:

Input to algorithm:
The threshold value used as the maximum waiting counter value forms the input data to our algorithm.

Proposed Solution:

Inputs: Threshold Value (TV)
Output: Assigned request
Pseudocode:

i) Assign counter value of zero to every incoming requests
ii) Check if maximum counter value of a request is greater than a predefined TV. If yes, then assign the request to disk and go step (vi). If no, then proceed to step (iii)
iii) Compute seek distance of all the pending requests relative to current read/write head position.
iv) Determine request with the minimum seek distance.
v) Assign the request determined in step (iv) to the disk
vi) Increment the counters of all the remaining requests by 1
vii) Move to step (i)

Programming language:
We intend to implement our program in C as most of the kernel coding is done in C.

Tools:
- **Microsoft Excel:** Tool to plot the average delay measured for our algorithm across requests. This will enable performance visualizations of our algorithm.

Output generation:
Current idea is to use logging that records the order in which the requests are assigned to the disk. We plan to log the arrival time of the request and the time the request is served.

Methodology to test against hypothesis:
In order to determine the correctness of our algorithm, I/O intensive programs will be used as the test workload. The logs output by the algorithm can be used to determine if the scheduler orders the disk requests as determined by our implemented algorithm. Also, we plan to measure the average delay in our algorithm. Average delay is defined as the time between the submission of a disk request by a process and the assignment of that request to the disk. If our algorithm succeeds in reducing the average delay as compared to SSTF, it can be concluded that the overall fairness achieved by our algorithm is better.

IMPLEMENTATION

Code:
For implementation purposes we used the Linux stable kernel version 3.9 source code. The kernel by itself has three disk scheduling algorithms implemented as a part of it. They are:
1. **No-op:** This is the first come first served disk scheduling algorithm (FCFS).
2. **CFQ:** Complete Fair Scheduling algorithm
3. **Deadline:** Soft real time scheduling algorithm

The default disk scheduling algorithm was CFQ. Our project involved implementing two additional algorithms namely Shortest Seek Time First with Logical Block Numbering (SSTF-LBN) and our own algorithm Shortest Seek Time with Threshold with Logical Block Numbering (SSWT-LBN). For implementing these algorithms and integrating them with the kernel code we had to modify the config files so as to enable the kernel to recognize and compile these algorithm codes. We also had to ensure that either of these two algorithms was selected as the default scheduling algorithms in order to test our algorithm implementations.

To implement these algorithms, each algorithm was implemented as a separate module contained in the following source code files:
- **sstf_iosched.c**: The SSTF algorithm implementation
- **sswt_iosched.c**: The SSWT algorithm implementation

Each of these algorithms was modularized and performs the following functions:

- **add_request:** This adds a disk request to the queue of pending requests.
- **dispatch_request:** This dispatches a request to the disk. This request is selected based on the algorithm. For example, SSTF chooses the request with the shortest seek distance relative to the current head position. SSWT chooses based on a combination of the wait count of each request and the seek distance as mentioned in the pseudocode given in Methodology section.
- **init:** Performs the necessary initialization activities of the algorithm such as initializing the request queue, disk head position, total delay measured, total seek distance measured, total number of requests.
- **reorder_queue:** This function is implemented by the SSWT algorithm. This is needed when a request is dispatched as its wait count is higher than the threshold. As this request may be an out of order request, it becomes necessary to reorder the queue based on the updated disk head position. Under ordinary circumstances, the queue is already organized based on the ascending order of seek distance of each of the requests and no reordering is necessary.
- **get_distance:** This function returns the distance of the disk request from the current head position. Based on the value returned the request is inserted at the appropriate position in the request queue.

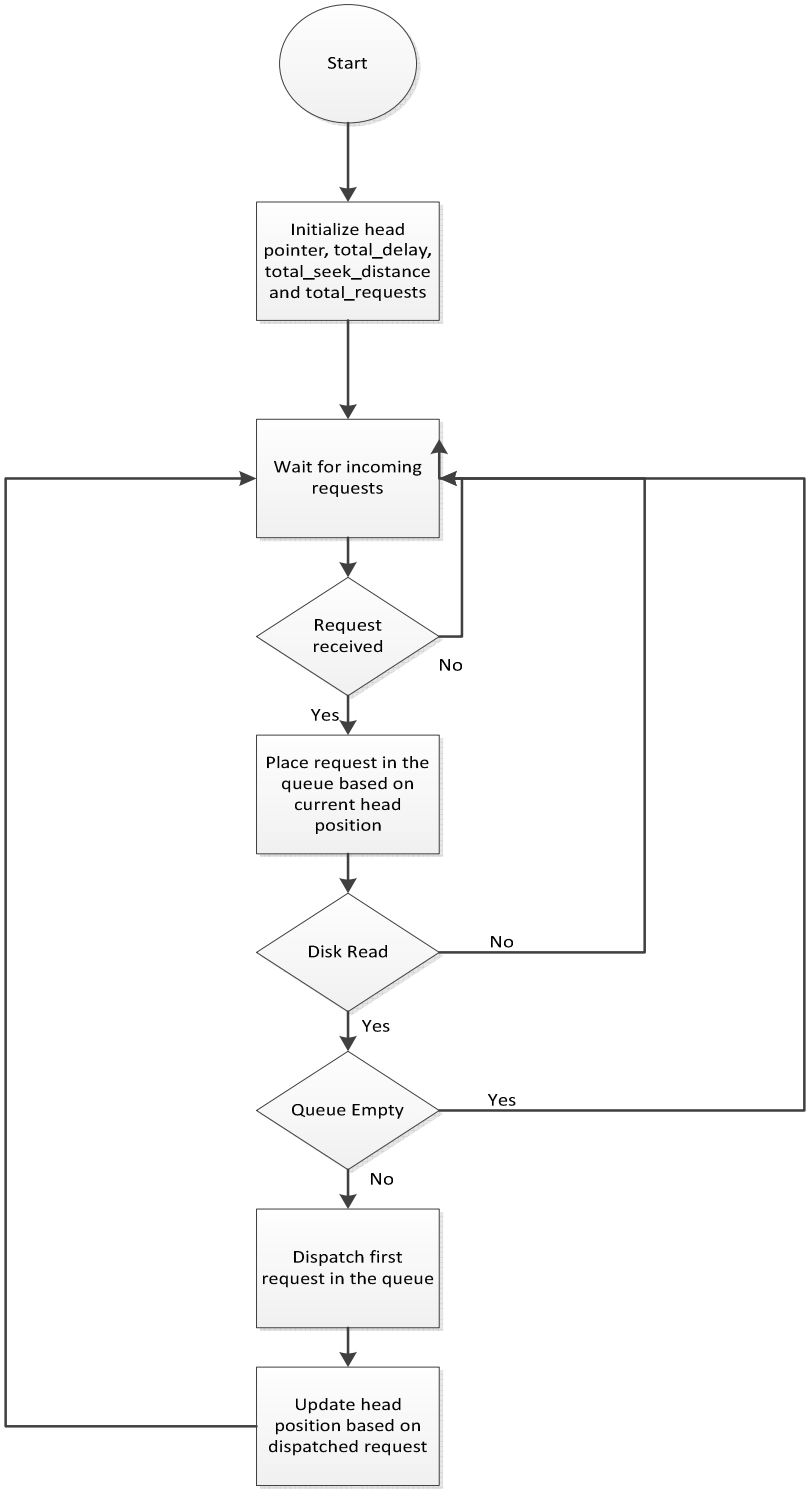
The set of functions listed above is by no means a comprehensive set of functions. The programs need to perform certain other functions as well such as destroy the memory on exit, merge requests if possible and so on. However, the most important functions have been listed above.

Challenges faced:

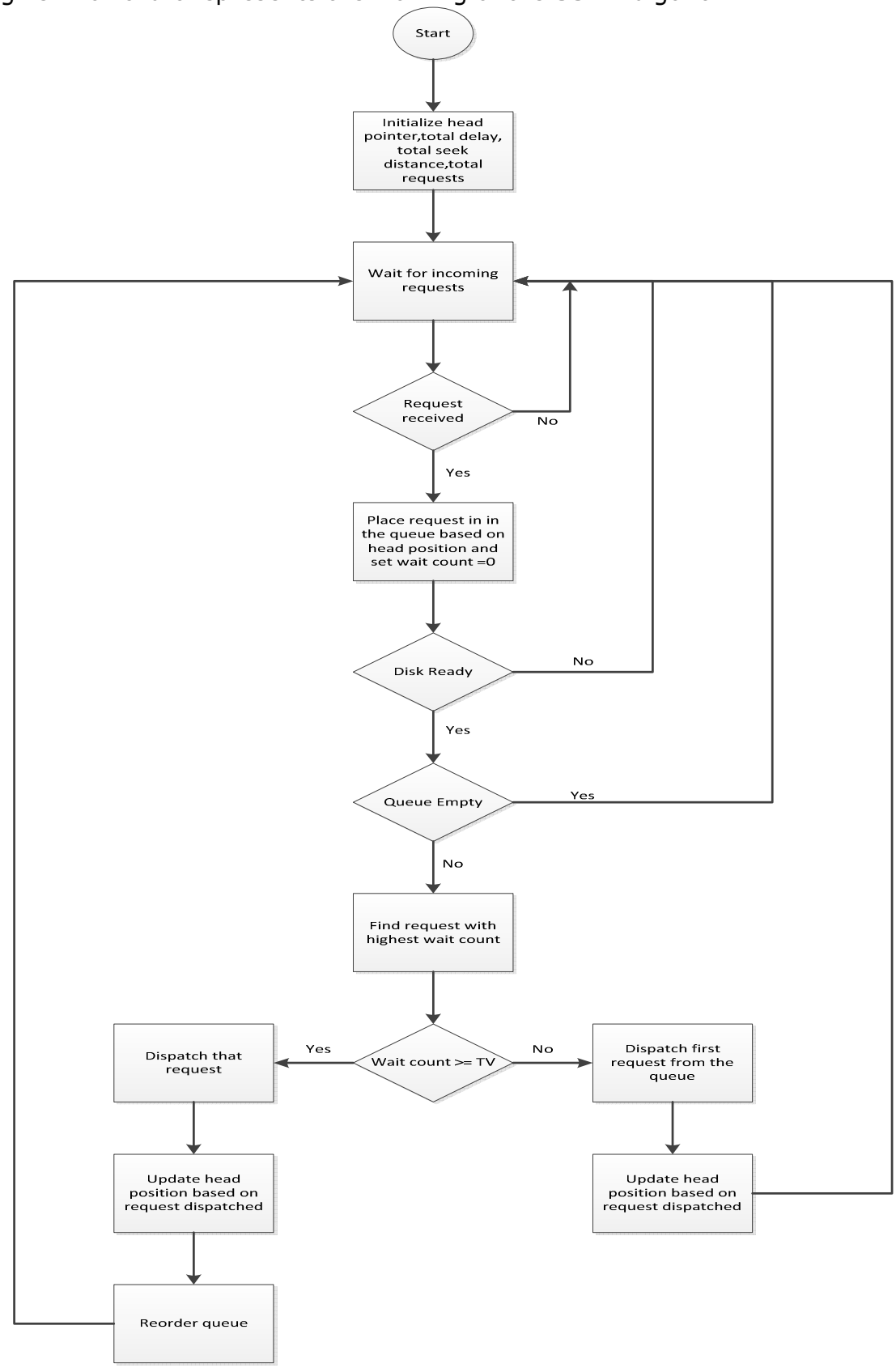
One of the primary challenges of this project was going through the monolithic kernel code and attempting to understand the internal working and dependencies of the code. A lot of the time dedicated to our project development was consumed by this research and analysis. We could begin development after an in-depth understanding of the disk scheduling branch of the linux kernel code was developed. In addition, recompiling and booting into the kernel is an involved process and therefore performance analysis required a lot of time as the kernel needed to be recompiled and booted into multiple times.

Design Document:

The given flowchart represents the working of the SSTF algorithm.



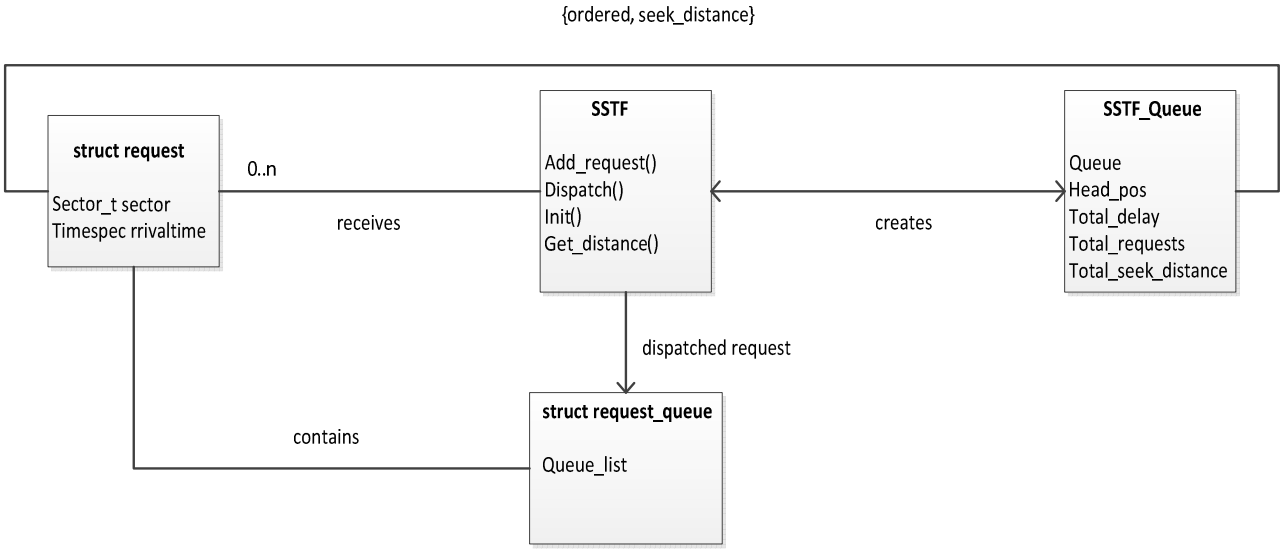
The given flowchart represents the working of the SSWT algorithm.



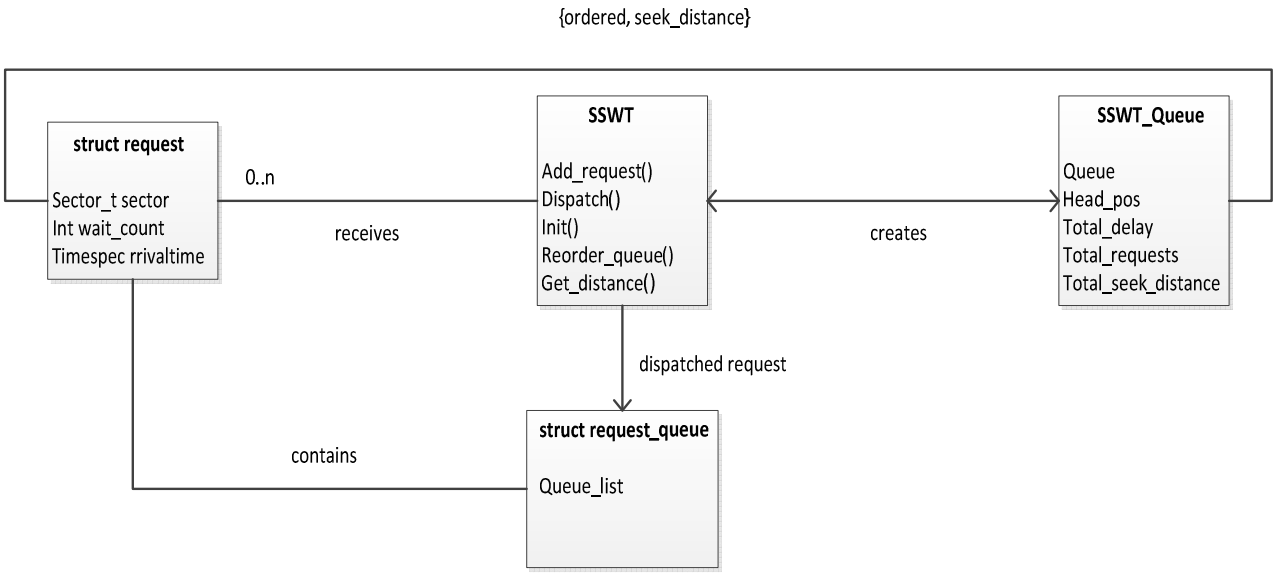
The class diagrams of SSTF and SSWT show the relevant data structures involved in implementing the algorithms. The data structures are:

- **struct request:** This structure is defined in the include/linux/blkdev.h header file of the linux kernel source code. This structure contains all the details related to a disk request such as the logical block number requested (contained in the field __sector). The fields that were added to this structure for our implementation were a timestamp that indicates the arrival time of the request and wait_count that indicates the number of request dispatch cycles that a particular request was waiting for. The wait_count is used for SSWT in dispatching a request whose wait_count exceeds the threshold. This timestamp was used to compute the delay between the arrival time and dispatch time of the request.
- **struct request_queue:** This structure is used by the operating system to dispatch a batch of requests to the hard disk. Hence, when the dispatch_request function of the algorithm is called, the selected request is added to this queue and removed from the list of pending requests maintained by the algorithm. The batch of requests is contained in this structure as a circular list of requests
- **struct sstf_queue, struct sswt_queue:** These structures maintain the queue of pending requests waiting for service from the disk. sstf_queue is used in the SSTF algorithm and sswt_queue is used in the SSWT algorithm. It maintains the queue as a circular linked list. A request is inserted into this queue based on its seek distance. This structure also maintains the current head position, the total delay measured, the total seek distance measured and the total number of dispatched requests. These parameters are printed to a log file on every dispatched request. These log files were used to determine that the algorithm is working as expected and to measure the average delay and total seek distance experienced by the system due this algorithm.

Class diagram for SSTF:



Class diagram for SSWT:



DATA ANALYSIS AND DISCUSSION

Output Generation:

We developed our project on two virtual machines residing on two physical machines with following configuration:

Physical Machine 1:

- Host Operating System: Windows 7 64 bit
- Guest Operating System: Ubuntu 12.04 LTS 64bit.
- Virtual Machines: VMPlayer virtual machine
- Virtual Hard disk: 40GB hard disk
- Virtual RAM: 1 GB
- Physical Machine: Lenovo Ideapad Ultrabook U410

Physical Machine 2:

- Host Operating System: Windows 7 64 bit
- Guest Operating System: Ubuntu 12.04 LTS 64bit.
- Virtual Machines: VMWare Workstation virtual machine
- Virtual Hard disk: 40GB hard disk
- Virtual RAM: 1 GB
- Physical Machine: Lenovo Thinkpad V470

Workload:

In order to test our new algorithm and compare it with SSTF we used the operating system's boot time workload. We recorded the arrival time and duration for which the request was waiting to be scheduled which is measured in the form of wait count. Our algorithm generates logs each time a request is dispatched logging the request arrival time and dispatch time, current total delay per request, total seek distance and total requests that have been served.

To check the logs we use the command "*dmesg* "which gives the last 200 system logs. In order to see the entire list of system log we need to open the syslog file using the command:

```
vi   /var/log/syslog
```

With the help of these logs we computed our average delay as follows:
Average Delay : Total Delay / Total Requests Served

For response time measurements, we wrote a C program that reads variable sized disk blocks. When the default algorithm is set to SSWT, our algorithm will be serving all the disk requests sent by this C program. We calculated the response time of our algorithm by taking the difference between the time the read request was made and the time when the request was completed. These readings were taken multiple times for a disk block of the same size to factor out the variability. We used disk block sizes ranging from 10MB to 100MB in increments of 10MB to compare the measured response times of our algorithm with SSTF.

To check the current default I/O scheduling algorithm for the system, use the command:

```
cat /sys/block/sda/queue/scheduler
```

This gives a list of all the I/O scheduling algorithms present in the operating system. The default algorithm is enclosed in round brackets. So when SSWT (our algorithm) is set as the default we get the output as:
"deadline CFQ Noop [SSWT] SSTF"

Output Analysis:
With the help of the log file snippets we will explain the working of the SSWT algorithm.

SSWT Log File – normal working:
This log file displays the logs for SSWT when the threshold value is set to 150 (This threshold was not the final threshold selected. It was used only for demonstration purposes). The log file contains one entry per dispatched request.

The entries in this log file demonstrate that requests are ordered on the basis of their seek distances and not the arrival time. For example, after serving the requests for sector 26963960, the scheduler schedules the request with the closest seek distance that is one with sector 26962272. This request is followed by the request for sector 26962248. The other pending requests have to wait even though their arrival time is prior to requests selected for scheduling. This indicates that requests are scheduled based on the seek distance irrespective of their arrival times. Each time a request is scheduled the wait count of all other requests is increased. If wait count of any request becomes greater than or equal to the

threshold value (in this case 150), that request is given higher priority in the scheduling session irrespective of the seek distance

```
[   51.803649]  Current time = 62891917, Total Delay = 884403722292
[   51.803651]
[   51.803651]  Arrival Time of request: Sec = 1370716499, Nanoseconds
= 62890811
[   51.803654]  Total Requests  = 6736
[   51.803654]
[   51.803654] [SSWT] request for sector = 26963960 dispatched, Wait
Count = 0


[   51.803840]  Current time = 63085248, Total Delay = 884403916801
[   51.803841]
[   51.803841]  Arrival Time of request: Sec = 1370716499, Nanoseconds
= 62890739
[   51.803843]  Total Requests  = 6737
[   51.803843]
[   51.803843] [SSWT] request for sector = 26962272 dispatched, Wait
Count = 1


[   51.803923]  Current time = 63169062, Total Delay = 884404195173
[   51.803925]
[   51.803925]  Arrival Time of request: Sec = 1370716499, Nanoseconds
= 62890690
[   51.803926]  Total Requests  = 6738
[   51.803926]
[   51.803926] [SSWT] request for sector = 26962248 dispatched, Wait
Count = 2


[   51.803991]  Current time = 63236180, Total Delay = 884404540847
[   51.803992]
[   51.803992]  Arrival Time of request: Sec = 1370716499, Nanoseconds
= 62890506
[   51.803993]  Total Requests  = 6739
[   51.803994]
[   51.803994] [SSWT] request for sector = 26962168 dispatched, Wait
Count = 3


[   51.804056]  Current time = 63301708, Total Delay = 884404952132
[   51.804057]
[   51.804057]  Arrival Time of request: Sec = 1370716499, Nanoseconds
= 62890423
[   51.804058]  Total Requests  = 6740
[   51.804059]
[   51.804059] [SSWT] request for sector = 26960336 dispatched, Wait
Count = 4
```

```
[   51.804129]  Current time = 63374620, Total Delay = 884405436388
[   51.804130]
[   51.804130]  Arrival Time of request: Sec = 1370716499, Nanoseconds
= 62890364
[   51.804131]  Total Requests  = 6741
[   51.804132]
[   51.804132] [SSWT] request for sector = 26960272 dispatched, Wait
Count = 5
```

SSWT Log File – requests with wait count higher than the threshold

This log file demonstrates that if the wait count of a request becomes greater than or equal to the threshold value then that requests is given preference irrespective of the seek distance. For this log file a threshold value of 10 was set. Again this is not our final threshold value and is used for the purpose of explaining our algorithm. It can be seen from the logs that after serving the request for sector 27006872, the request for sector number 22747136 is served next even though there is another request for sector number 26964000 which has a shorter seek distance than the request for sector number 22747136. This indicates that when the wait count of a request becomes greater than or equal to the threshold value, preference is given to that request instead of serving the request with shortest seek distance.

```
[  120.034513] Current time: Sec = 1370729723, Nanoseconds =
132434949, Total Delay = 864714696578
[  120.034514]
[  120.034514]  Arrival Time of request: Sec = 1370729723, Nanoseconds
= 131641823
[  120.034515]  Total Requests  = 8449
[  120.034515]
[  120.034515] [SSWT] request for sector = 27006872 dispatched, Wait
Count = 10

[  120.034581] Request with sector 22747136 overriding shorter seek
candidate with sector 26964000
[  120.034583]
[  120.034583]  Current time: Sec = 1370729723, Nanoseconds =
132505577, Total Delay = 864715561109
[  120.034584]  Arrival Time of request: Sec = 1370729723, Nanoseconds
= 131641046
[  120.034585]
[  120.034585]  Total Requests  = 8450
[  120.034586] [SSWT] request for sector = 22747136 dispatched Wait
Count = 11

[  120.034664]
[  120.034664] Request with sector 26964000 overriding shorter seek
candidate with sector 26941608
[  120.034666]  Current time: Sec = 1370729723, Nanoseconds =
132589587, Total Delay = 864716508922
```

```
[  120.034667]
[  120.034667]  Arrival Time of request: Sec = 1370729723, Nanoseconds
= 131641774
[  120.034668]  Total Requests  = 8451
[  120.034669]
[  120.034669] [SSWT] request for sector = 26964000 dispatched Wait
Count = 12
```

Log file for SSTF:
It can be seen that requests are served based on seek distance. Request with shortest seek distance from the current head position is given preference. For example, after serving the first request (wrt. the log file) the current head position is at sector number 31173936. Some of the other requests in the queue are for sector number 31173944, 31173952, and 31173960. Amongst these, request for sector number 31173944 has the shortest seek distance and hence served next. Thus we see that request are in order of shortest seek time first.

```
[   36.414252] [SSTF-START]
[   36.414254]  Current time: Sec = 1370735619, Nanoseconds =
10765136, Total Delay = 682448893083
[   36.414255]  Total Requests  = 6770
[   36.414256] Request for sector = 31173944 dispatched
[   36.414257]
[   36.414257] [SSTF-END]

[   36.414342] [SSTF-START]
[   36.414344]  Current time: Sec = 1370735619, Nanoseconds =
10855712, Total Delay = 682448893209
[   36.414345]  Total Requests  = 6771
[   36.414346] Request for sector = 31173952 dispatched
[   36.414347] [SSTF-END]

[   36.414471] [SSTF-START]
[   36.414473]  Current time: Sec = 1370735619, Nanoseconds =
10983923, Total Delay = 682448893395
[   36.414473]  Total Requests  = 6772
[   36.414474] Request for sector = 31173960 dispatched
[   36.414475] [SSTF-END]
```

Comparison of output and hypothesis:
As explained earlier, our goal was to improve the fairness of Shortest Seek Time First (SSTF) algorithm. SSTF has the drawback of starvation of requests that have higher seek distance. Thus the algorithm can be unfair to requests with higher seek distance. Typically, such requests need to wait for a longer time in the scheduling queue. Based on this idea, we computed the average delay which is the amount of time requests wait to be scheduled by the scheduler. If more requests experience starvation, average delay for SSTF is bound to be on the lower edge. We attempted at improving the average delay for SSTF with our modified algorithm. We took into consideration arrival time of the requests, a wait count for each request that

indicates the amount of scheduling sessions the request has not being scheduled and a fixed threshold value to schedule requests. If requests have their wait count higher than or equal to the threshold value, higher preference is given to these requests.
From the results, it was seen that for most cases our algorithm gives better average delay than SSTF. However, the threshold value needs to be appropriate for the given workload and given system configurations.

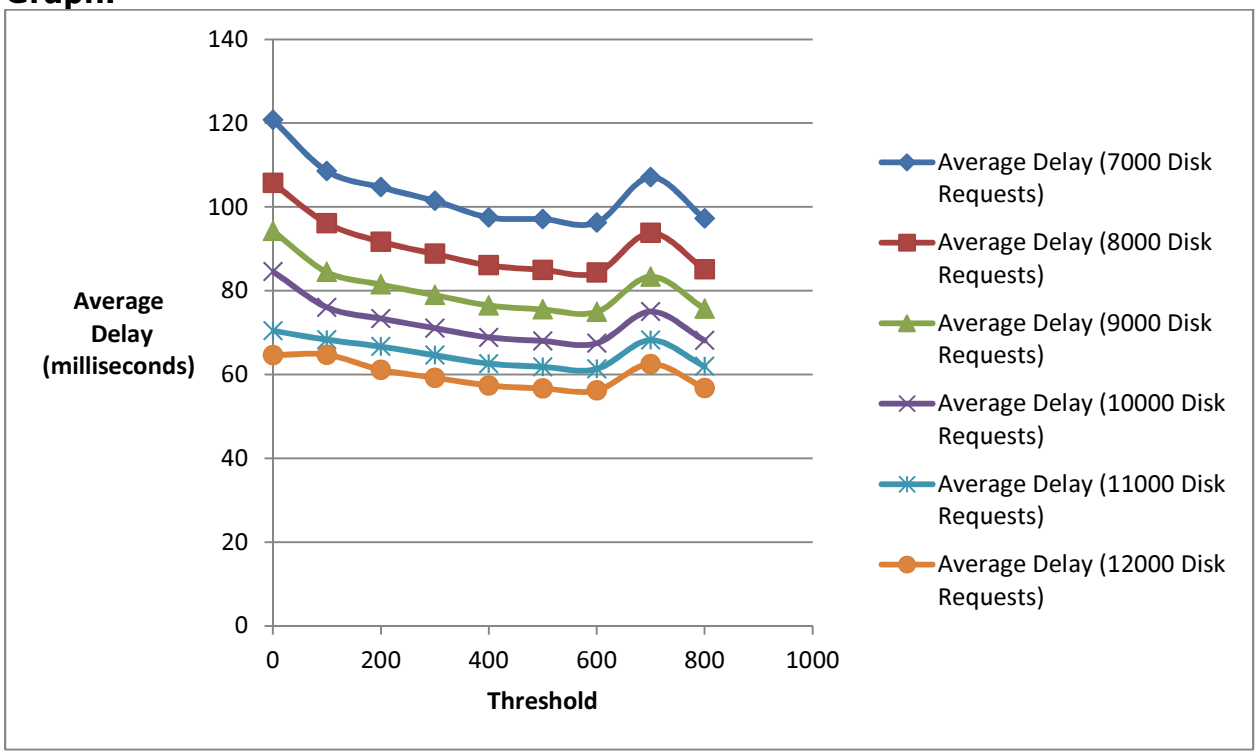
Abnormal Case Explanation:
Our tests were carried out for the boot time workload of a system. As this exhibits a large amount of variance, it was seen that the average delay for SSTF was varying considerably for some boot time workloads. In order to overcome this variance we took multiple readings at different times. We have included the results that were observed for most of our readings.
In one of our machines, the seek distance for our algorithm was seen to be slightly lower than SSTF in some of the requests. This behavior again was due to the unpredictable workload that we were testing against.

Statistic Regression:
The results obtained are shown graphically in this section. While the algorithms were analyzed on both the machines, only the graphs for the readings obtained on physical machine 2 have been included in this section for reasons of brevity.

Determining the optimal threshold value:
For the successful implementation of SSWT, it was crucial to pick a correct threshold value. A threshold value that was too low would result in giving undue importance to the wait count which could potentially degrade the performance by increasing the seek times. Also, the disk scheduler will spend too much time reordering the queue of pending requests which will affect the average delay adversely. A too high threshold value will result in the algorithm working purely as SSTF, resulting in potential starvation of requests. Hence, it was necessary to determine the threshold value at which the lowest average delay was observed and fix this value for the rest of the performance comparisons with SSTF. In order to do this, we determined the average delay for threshold values ranging from 0 to 800 in increments of 100. The average delays measured for each of the threshold values can be seen in the graph below. The workload used for taking these readings was the boot time workload experienced by the system. From the graph, it can be clearly seen that the best threshold value is 600. Although not shown, the threshold value of 600 was the best even for physical machine 1. The lower values exhibit a higher average delay as the scheduling algorithm spends too much time reordering the request queue while values greater than 600 result in the algorithm performing as pure SSTF.
As can be seen in the graph legend, these average values were measured at different number of disk requests. That is, the average delay at 7000, 8000, 9000, 10,000 and 12,000 disk requests was measured. For each of these, the lowest average delay was exhibited at a threshold value of 600. The average delay readings were taken at varying disk requests to eliminate any possible variance in the readings. Based on these observations, we fixed the threshold value to 600 for

the SSWT algorithm. The SSWT algorithm with this constant threshold value was then used for the performance comparison experiments with SSTF.

Graph:



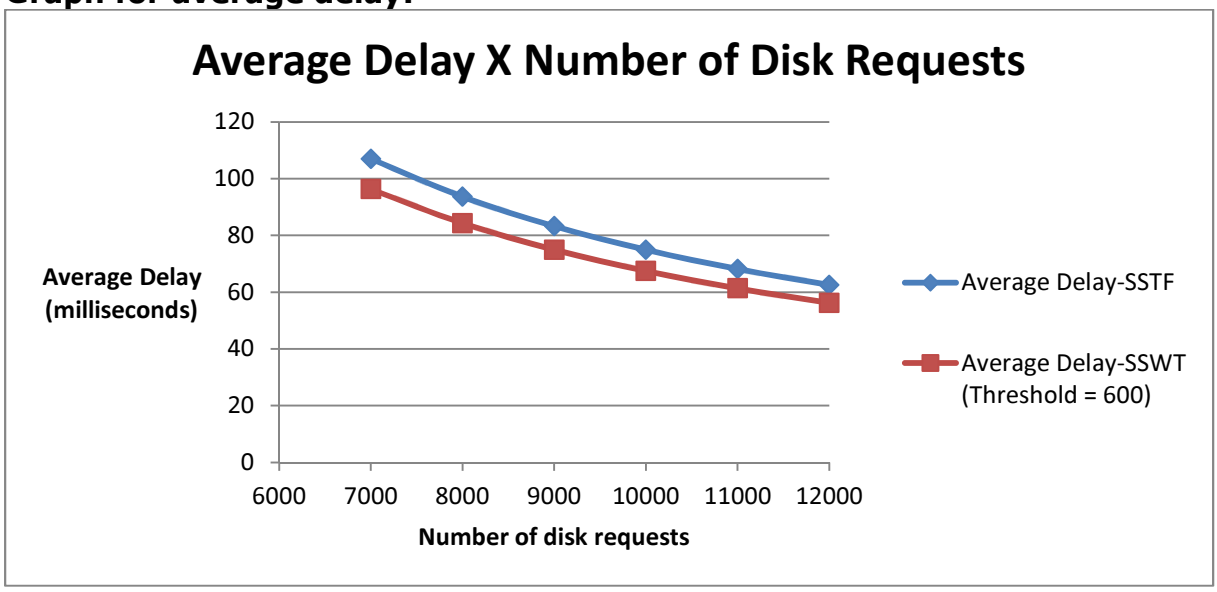
Evaluation of SSWT against SSTF:

The workload used for benchmarking the performances of the two algorithms was the workload experienced by the disk as the operating system boots. The log messages recorded at boot time were used for the computation. For each parameter, the readings were taken on both the physical configurations. Our algorithm of SSWT was evaluated against SSTF along the following three parameters:

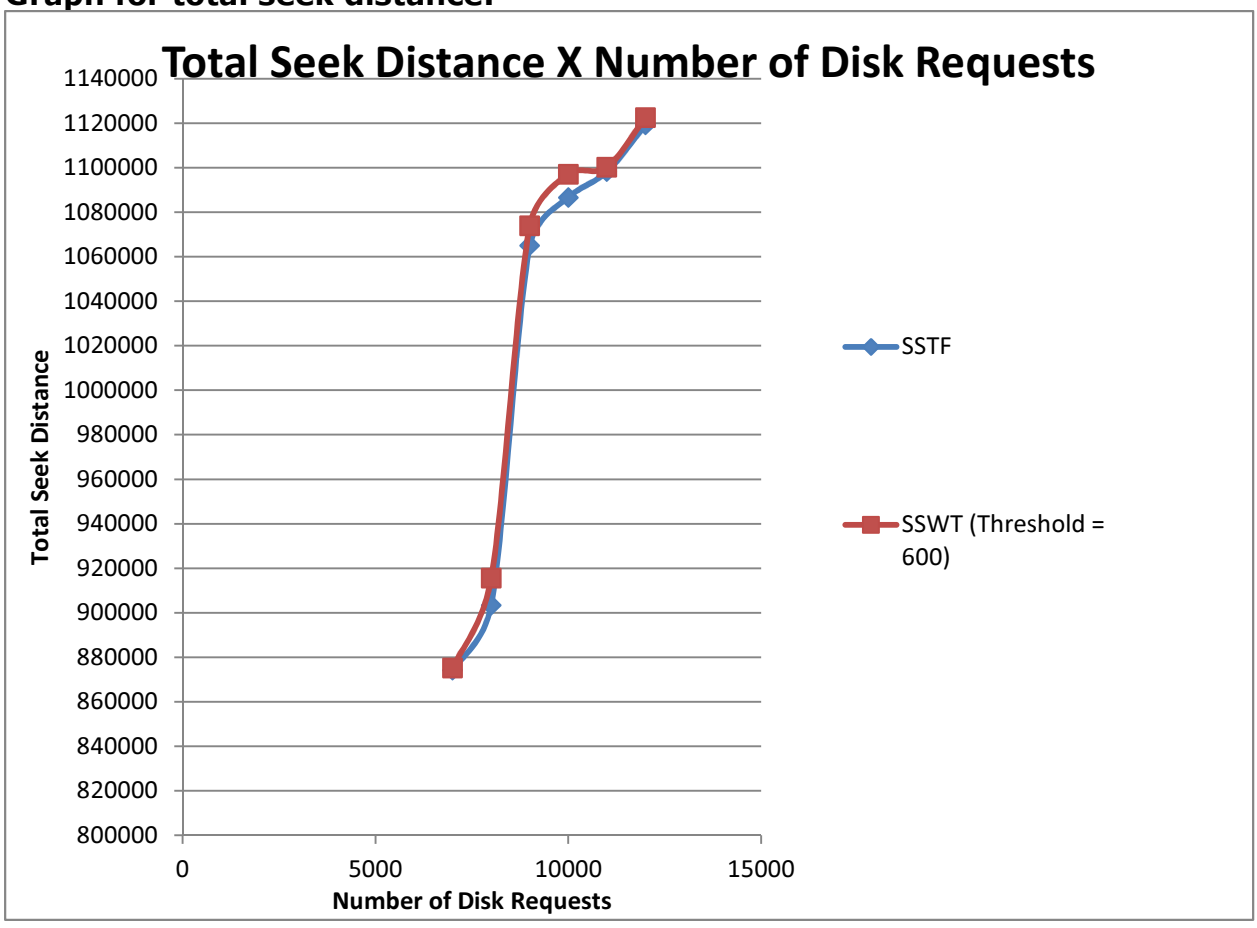
- **Average Delay:** The average delay for SSTF and SSWT was measured at different disk requests. As can be seen for both the graphs, SSWT exhibits a lower delay as compared to SSTF.
- **Total Seek Distance:** The total seek distance for SSTF was slightly lower than our algorithm on physical machine 2. However the difference was simply 0.52%. This outcome was expected as our algorithm uses threshold values to ensure fairness while SSTF always uses seek distances. On the other hand on machine 1, SSWT exhibited lower seek distance for some requests. This anomaly can be attributed to the test workload not being very predictable or controlled. However, in spite of this anomaly, the overall seek distance was distance was 0.4% lower for SSTF as compared to SSWT.
- **Average Response Times:** SSWT exhibits a lower average response time on both the machines as opposed to SSTF. This was expected as average delay and average response times are closely connected. As SSWT exhibits a

lower average delay as compared to SSTF, it was expected that it will exhibit a lower average response time as well.

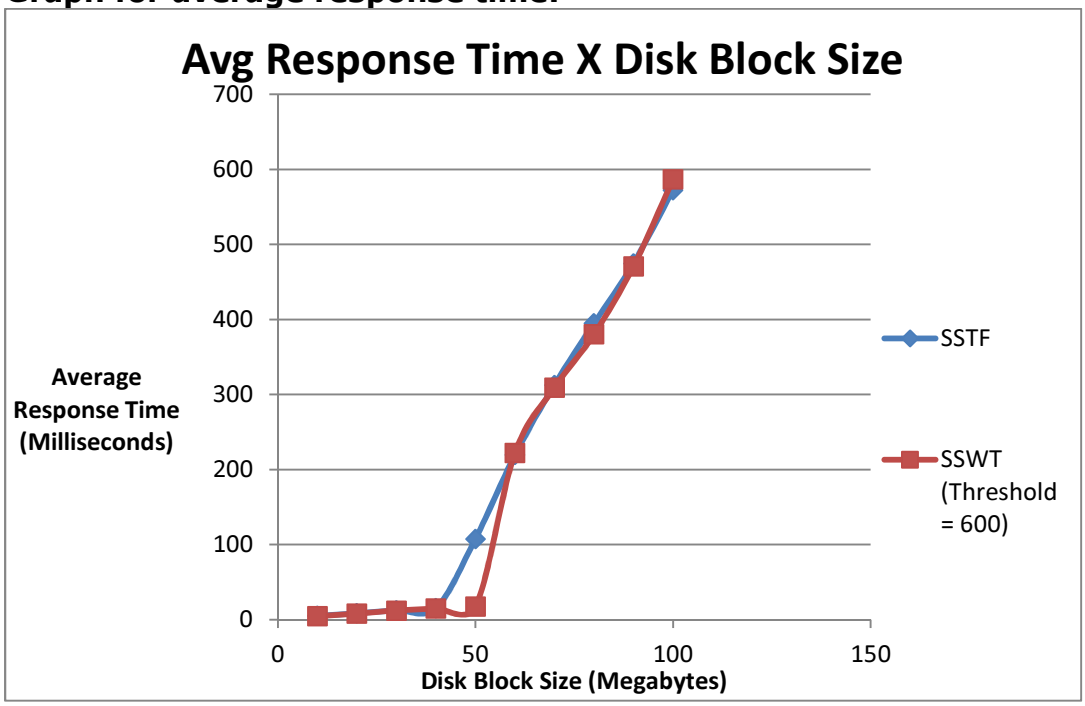
Graph for average delay:



Graph for total seek distance:



Graph for average response time:



Discussion:

As can be seen from the graphs, SSWT (using the optimal threshold value of 600) performs better than SSTF in terms of average delay and average response time. The success of the algorithm can be attributed to ensuring that no request has to wait beyond a certain limit for disk access. While SSTF performs better in terms of seek distance, we believe this is a small price to pay for a better response time and a lower average delay.

It can be argued that the wait count can be replaced by wait time (in nanoseconds) to ensure that the request waiting the longest is assigned to the disk. This is a valid argument as wait count does not provide an accurate approximation of the wait time. For example, all the requests arriving between two dispatch cycles will have the same wait count even though their arrival and hence wait times differ.

However, wait count was intentionally chosen for two reasons:

Keeping track of the wait time requires more computations. To determine the request with the highest wait time, the current time needs to be determined. The arrival time then needs to be deducted from the current time to get the value of the wait time. While this is not much under ordinary circumstances, these computations can become a bottleneck when implemented in the the kernel. These computations would have to be performed every dispatch cycle which could consume a lot of time. While not entirely as accurate as the wait time, wait count provides a good approximation of the wait time at a significantly lower computation cost. For a wait count, the count simply needs to be incremented for all the pending requests at every dispatch cycle.

Using the wait count, choosing a threshold value becomes much more intuitive and the value chosen can be reasonably small (such as 600). However, in terms of wait time, choosing the threshold will require a lot more comprehensive analysis. Given

the time frame for the completion of this academic project, it was not possible to do the same.

CONCLUSIONS AND RECOMMENDATIONS

Summary and Conclusions:

The goal of this project was to attempt to improve the fairness of SSTF algorithm while retaining the inherent optimizations of the same. We attempted to do this by maintaining a wait count that keeps track of the number of disk request dispatch cycles a pending request has been waiting for. If this wait count exceeds a predefined threshold value, the request is assigned to the disk in the next dispatch cycle irrespective of it's seek distance. If no such request exists, the requests are assigned to the disk in an ascending order of the seek distance. This new algorithm was named Shortest Seek With Threshold (SSWT). Both SSTF and SSWT that we implemented use logical block numbering to gauge the seek distance.
The performance of our new algorithm SSWT was compared with SSTF along three parameters - Average Delay Time, Total Seek Distance and Average Response Time. The benchmark workload used for these comparisons was the boot time workload experienced by the scheduling algorithms. The results indicate the following:
Average Delay: SSWT has a 10% lower average delay as compared to SSTF.
Total Seek Distance: SSTF has a 0.52% lower average seek distance as compared to SSWT.
Average Response Time: SSWT has a 10% lower response time as compared to SSTF.

On the basis of these results, it can be concluded that the goal of introducing a fairer algorithm compared to SSTF was accomplished. The only caveat is the benchmark workload used was the boot time workload. While this is a good workload for comparison purposes, it is clearly not the best as it is not very predictable and cannot be controlled by the tester. However, since these algorithms have been implemented in the kernel, it is difficult to generate a predictable workload. Even if a user program that reads or writes to the disk were to be used as a benchmark, the order in which the requests would be passed by the rest of the kernel disk code to the add_request function of the disk scheduler is hard to determine. It could be possible that the same benchmark code causes the kernel to call the add_request function of the disk scheduler in a different sequence. It would have required further exploration and understanding of the kernel code and the internal dependencies. While possible, it was not feasible in the given time frame. Hence a slightly less predictable workload was chosen by us as the benchmark.

Future Recommendations:

One of the ways to enhance the SSWT algorithm is to implement a system call by which the threshold of the algorithm can be set. This will get rid of the rigidness associated with a predefined threshold value. Users can customize the threshold based on which aspect of the disk scheduler is important to them. If they desire a low seek distance irrespective of the fairness, then a very high threshold value can be used. If they desire more fairness as opposed to a lower seek distance they can

choose a slightly lower threshold value. However, care must be taken that users do not exploit this system call to cause any security breaches.

Bibliography:

[1] Circular Shortest Seek Time First: http://www.iasj.net/iasj?func=fulltext&aId=50991
[2] Knowledge-based disk scheduling policy using fuzzy logic by Talip, M.S.A., Abdalla, A.H., Aburas, A.A., Zahirul Alam, A.H.M., Asif, A. *Computer and Communication Engineering (ICCCE), 2010* (Presented by Rima Deodhar)
[3] Real-Time Disk Scheduling Based on Urgent Group and Shortest Seek Time by First Kitae Hwang ; Heonshik Shin, *Real-Time Systems, 1993.* (Presented by Shruti Naik)
[4] Disk scheduling with quality of service guarantees by Bruno, J.; Brustoloni, J. ; Gabber, E. ; Ozden, B. ; Silberschatz, A. *Multimedia Computing and Systems, 1999. IEEE International Conference*
[5] Survey and analysis of disk scheduling methods by Alexander Thomasian *ACM SIGARCH Computer Architecture News Volume 39 Issue 2, May 2011*
[6] Modern Operating Systems by Andrew S. Tanenbaum, *3rd edition*
[7] Understand Linux Kernel, By O'Reilly
[8] Linux Kernel Newbies: http://kernelnewbies.org
[9] Linux Kernel Official website: http://www.kernel.org