Multi - Threaded Web Server with Usage Statistics
Team 3

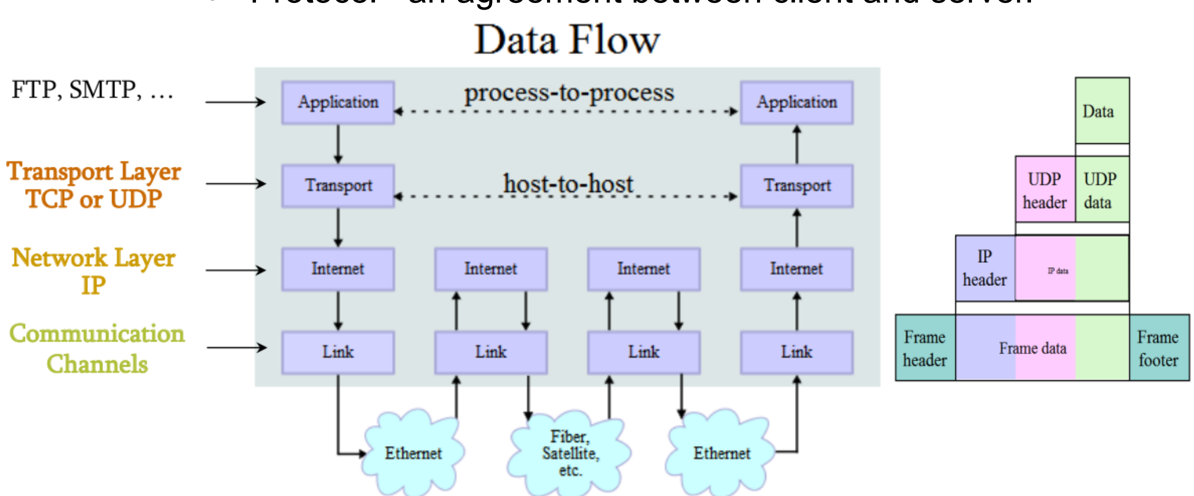
Kartik Mehta - 2019102029
Rahul Kashyap - 2018102037
Avinash Prabhu - 2018102027
Abhigyan Gargav - 2019102040

Objectives:

- To implement a web server using Threads and Sockets
- To understand how to synchronize and work with cooperating threads in Linux/Unix.
- To learn about how a web server is structured using Multiple threads and sockets.
- To understand the effect of the number of threads on the turnaround and response time.
- To understand how different process mixes affect the turnaround and response time.
- To benchmark and compare the turnaround and response times for various scheduling algorithms.

Some Prerequisites :-

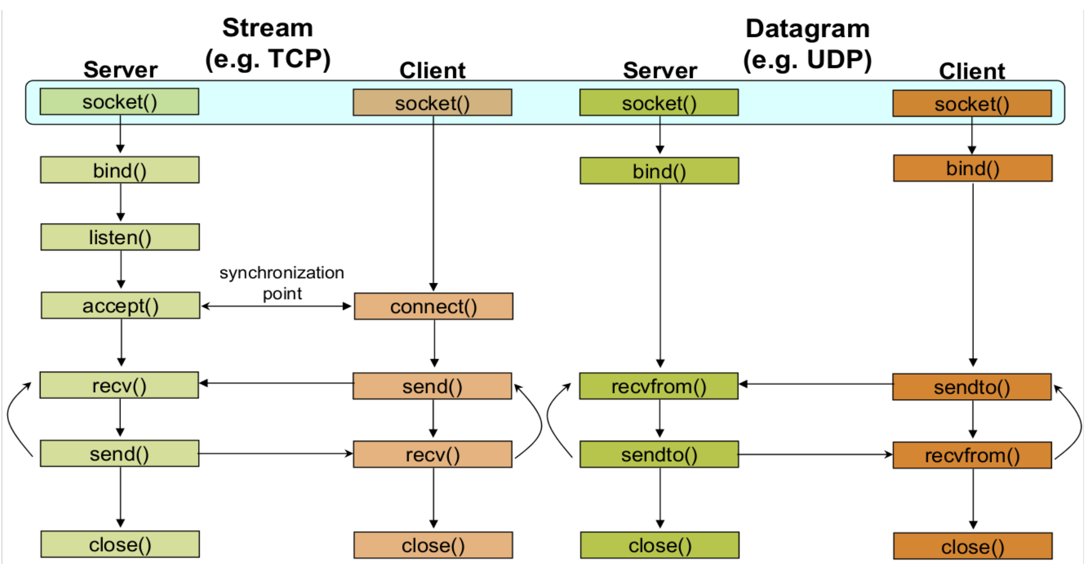
1. Terminologies -
    - Host - runs the applications
    - Routers - forward the information
    - Packets - sequence of bytes containing control information
    - Protocol - an agreement between client and server.



The above fig shows the Data flow between various Layers in a connection link.Each layer adds its own header to the data and then forward it to the next layer.

2. Sockets - Sockets are uniquely identified by an internet address, an end-to-end protocol and a port number.we will be mostly dealing with two types of sockets which are -
    - TCP (Transmission Control Protocol - Stream Socket) : Used for services with a large data capacity.It is connection oriented and bidirectional.
    - UDP (User Datagram Protocol - Datagram Socket) : Used for quick lookups and single use query-reply actions.It is connectionless(no retransmissions and no acknowledgements).

3. Client Server Communication Link -
   Server and client first create a socket to send and receive data.Then the server binds it's port and it's IP address to a socket and then that socket is used to listen to the incoming requests from the Client. In the case of TCP , we have a handshake protocol(accept() and connect()) to ensure the connection between the Server and the Client whereas in UDP there is no such thing hence there is no assured connection.
   After successful connection, client and server send and receive data back and forth and in the end after all the transmission both server and client also close the socket at their corresponding ends.

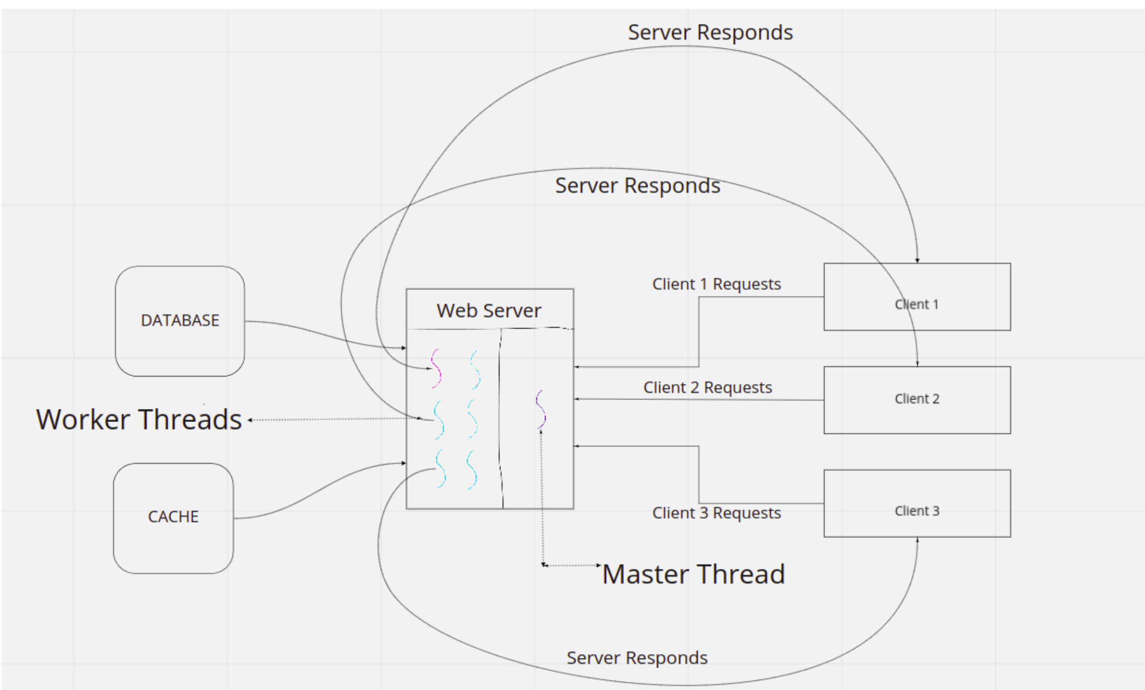


Method:-

To achieve our objectives, we implement a multithreaded web server. Our language of choice for this implementation is C.

A web server delivers content for a website to a client that requests it. Our web server runs on port 8989, and is pinged by a custom Ruby Script to retrieve content via an HTTP request. The web server supports various scheduling policies as described in the next section. By switching the policy used, and comparing the server response times, we benchmark the performance of the implemented scheduling algorithms.

Servers can be single threaded and multi threaded. Single-threaded servers are able to serve only a single HTTP request at a time so if we want to handle multiple requests at the same time we'll need more worker threads at the server. However, our server is multithreaded and in a typical multi-threaded server we have a master thread which controls the data flow to the worker threads. We have a fixed-size pool of worker threads, and each new request is serviced by a thread from the pool. In case no threads are available, the request enters a waiting state(or waiting queue). One a thread becomes available, the scheduling policy is responsible for deciding which waiting process receives the thread.

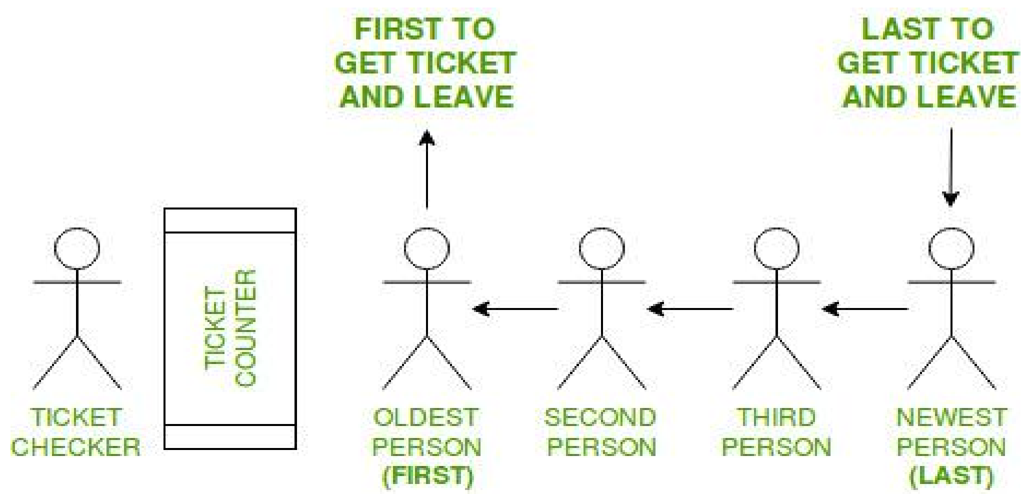


The above figure shows a multi-threaded server in which we have a certain number of clients and all of them will place their requests simultaneously to the server which assign the requests to the master thread and then according to the scheduling policy in place it will assign a request to the worker thread in a certain order.If the number of request are more than the number of worker threads then there will be requests waiting in the buffer.(as per the scheduling policy).

Scheduling Policies :-

In this section, we will be using the term “request from the client” and “processes” interchangeably.

- **FCFS**: The first come first serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner the job will get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.



- **SFF**: The shortest file first scheduling algorithm schedules the processes according to their file size. In SFF scheduling, the process with the lowest file, among the list of available processes in the ready queue, is going to be scheduled next.(CPU talks to the smaller file first then to the larger one)

Producer Consumer Problem:-

The master thread stores all the incoming client requests in a buffer. The worker threads keep checking if there is any request in the buffer, and if there is , it takes the request out of the buffer. So as we can see, the master thread and the worker threads are in a Producer-Consumer relationship with the buffer, where the master thread is the producer which keeps adding in the buffer whereas the worker threads are consumers which keeps taking requests out of the buffer.

Adding a Lock - We need to use a mutex lock so that neither the producer nor the consumers can access the buffer simultaneously. So we add a lock for the worker threads and we also add a lock for the master thread, both dependent on the same *mutex*.

Handling busy waiting - If we use a Mutex lock then the threads will keep asking for a request and there will be busy waiting in the thread function. To avoid this busy waiting we are using semaphores which adds the new request to the queue and whenever a worker thread is available it wakes the request from the queue and assigns it to the available thread.

Pseudo Code:

```
void *thread_function()
{
        while(1)
        {
                sem_wait();
                pthread_mutex_lock(&mutex)

                //Critical Section - take request out of buffer

                pthread_mutex_unlock(&mutex)

                //Handle request
        }
}


int main()
{
        //create thread pool
        //create a server and start listening.

        while(1)
        {
                //Accept new client connection

                pthread_mutex_lock(&mutex)

                //Critical Section - add request in the buffer

                sem_post();
                pthread_mutex_unlock(&mutex)
        }
}
```

Code:

The code contains three sections - multi-threaded server, client files and HTML files.

Server Side - The server contains the following files :-

- `Multi_threaded_server.c` - It contains the main code of the server which combines all the files.
    - It creates a single-threaded server using *server.c* which is the master thread. As we will have multiple requests, to handle each request individually, it creates a thread pool of fixed size.
    - Listens for the incoming client requests and adds them to the fixed size buffer according to the scheduling policies. In the case of FIFO, this buffer is a queue, whereas in the case of SFF, this buffer is an array.
    - The Thread function keeps waiting for a new request to come and when it gets a new request, it gives the request to one of the worker threads. If all the worker threads are assigned some request, then all the remaining client requests wait in the buffer.

- `server.c` - This contains the code to create a server
    - It first creates a socket, and then binds it to its address, which contains the IP address and port of the server.
    - Having defined the socket, the server starts listening on its defined port for client requests.
    - It then finally returns the socket ID to the *multi_threaded_server.c*

- `handler.c -` It contains a function to parse the string , get file size and handle connection function.
    - `parseString -` It parses the client request string by removing spaces as: (method,path,version).For example, the request “GET ./data/1.html HTTP/1.1” is parsed as
        - Method : “GET”
        - Path : “./data/1.html”
        - Version : HTTP/1.1

- `getFileSize` - This returns the file size of the file being requested by the client. This is done before the file is added to the buffer so that we can attach the size attribute to the request.
- `handle_connection` - This is the main handler function which takes in the client's socket ID and handles requests.
  - It first reads the client's request and parses it using *`parseString`*.
  - It then opens the file requested by the client, reads it line by line and sends it.
  - Finally it closes the file and client socket.

- `queue.c` - It creates a queue of requests from which the request are assigned to the threads using two function enqueue(to add the request to the queue) and dequeue(to remove the request after it execution)
- `Helper.c` - It contains some helper functions used in the code like the comparator function for sorting and an "error and die" function which checks the error and dies.

Client Side -

- `Client.c` - It contains the code for creating a client in C.
  - It first defines a socket using *socket()* which it will use for communication. Then it connects to the server address using *connect()* .
  - After connection, it sends the request message(of the form "GET /path/to/file HTTP/1.1") to the server using *write()*.
  - Finally it reads the server response using *read()* in a loop, where it reads the response line by line.
- `Client.rb` - It contains the code for creating a client in ruby.

Datasets used and experiments carried out:

We used three datasets:

1. **Small Data** - Contains 10 html files. Total size - 219 MB. These require a lesser amount of CPU time-quantums, so serve as our IO-bound processes.

a.
```
4.0K    ./0.html
4.9M    ./1.html
9.7M    ./2.html
15M     ./3.html
20M     ./4.html
25M     ./5.html
30M     ./6.html
34M     ./7.html
39M     ./8.html
44M     ./9.html
```

2. **Large Data**- Contains 10 html files. Total size - 4 GB. They are CPU-bound processes.

a.
```
383M    ./0.html
388M    ./1.html
393M    ./2.html
398M    ./3.html
403M    ./4.html
407M    ./5.html
412M    ./6.html
417M    ./7.html
422M    ./8.html
427M    ./9.html
```

3. **Mixed Data**- Contains 10 html files. Total size - 2.3 GB. This file consists of a mix of small and large files, i.e. a mix of CPU-bound and IO-bound processes.

a.
```
4.0K    ./0.html
4.9M    ./1.html
9.7M    ./2.html
15M     ./3.html
20M     ./4.html
437M    ./5.html
441M    ./6.html
446M    ./7.html
451M    ./8.html
456M    ./9.html
```

Using these datasets, we studied the following to evaluate the performance of scheduling algorithms:

1. **Effect of varying thread pool size**
2. **Effect of different process mix**

To quantify the performance of these two variables, we have used two parameters -

1. **Turnaround time**- amount of time to execute a particular process – from time of submission till finish
2. **Response time**- amount of time it takes from when a request was submitted until the first response is produced, not output.

<u>Test Case Generator:</u>

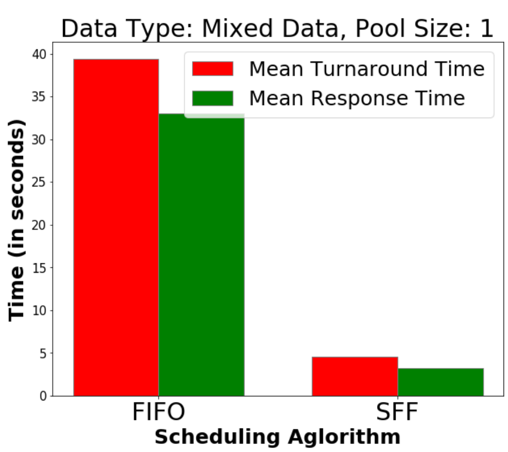
- Our test cases consist of html files that the server has to open upon the request of the client.
- To generate our test cases, we wrote a python script to generate html files of increasing sizes. The script can be customized to generate html files of varying sizes to test the performance of various scheduling algorithms.

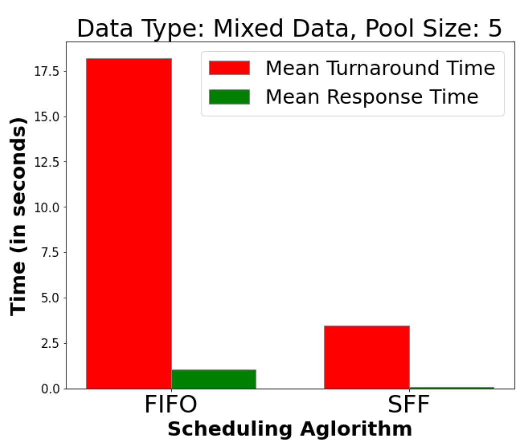
<u>Results</u>:

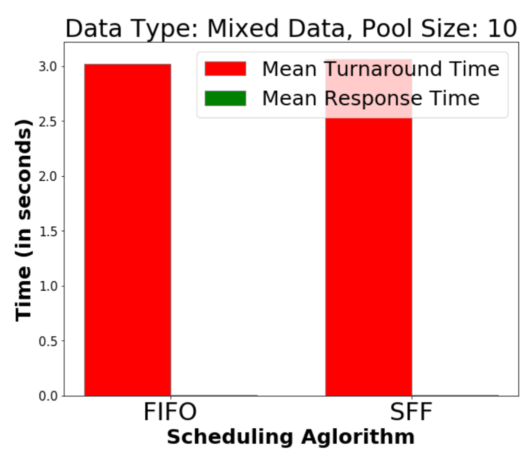
<u>Effect of changing Thread Pool Sizes:</u>

We used thread pools of sizes **1, 5 and 10** on our ***Mixed Data***, to analyse the effect of thread pool sizes. The number of client calls to the server was 10 for all cases.

<u>Results for Mean values:</u>

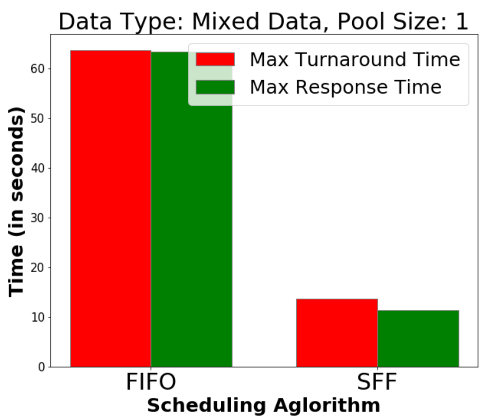


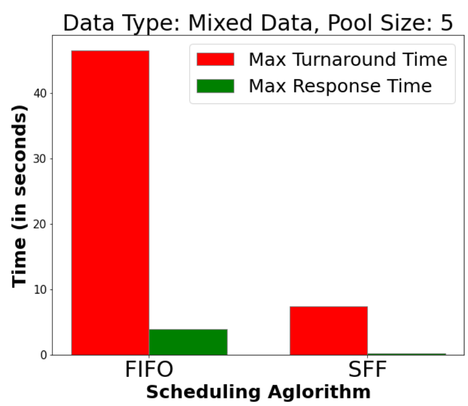


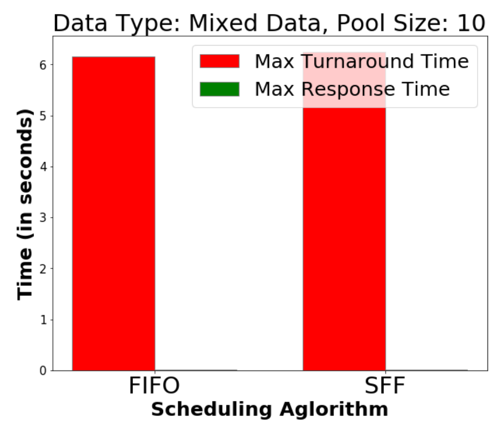


Inferences:
1. We can see that as we increase the number of threads, the **mean response time** decreases. This is because more threads means more files can be attended to right away.
2. When the number of threads are 10, the **mean turnaround time** reduces drastically to 3 seconds as opposed to 40 seconds and 17.5 seconds for 1 thread and 5 threads respectively.
3. While the use of a scheduling algorithm has an impact when only 1-5 threads are used but negligible impact when 10 threads are used as most processes can be attended to right away, thus not requiring a scheduling algorithm to schedule the processes.

<u>Results for Max Values:</u>

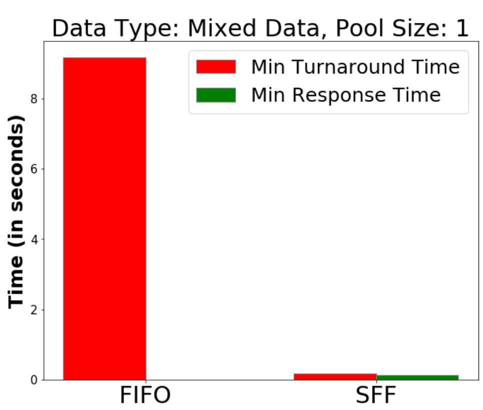


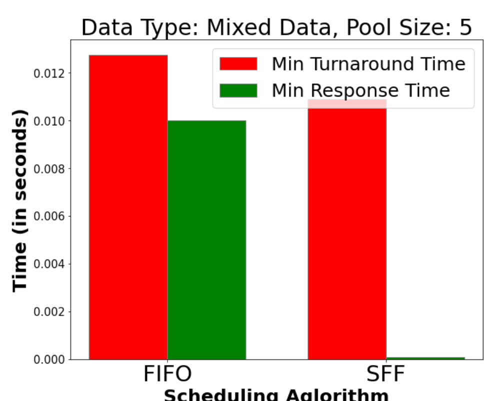


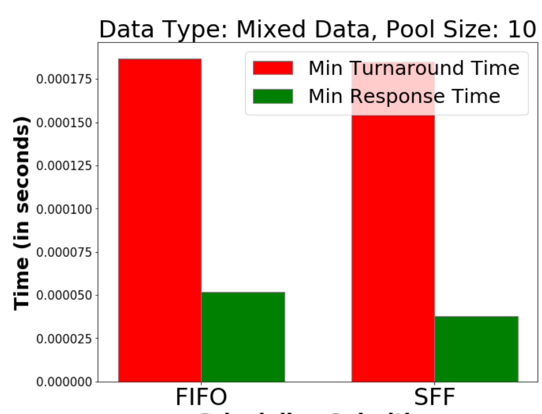


Inferences:
1. We can see that as we increase the number of threads, the **maximum response time** decreases drastically. This is because more threads means more files can be attended to right away.
2. When the number of threads are 10, the maximum turnaround time is reduced drastically to 6 seconds as opposed to 60 seconds and 40 seconds for 1 thread and 5 threads respectively. The max response time is negligible, because each process gets almost instantaneously catered to.

<u>Results for Min Values:</u>





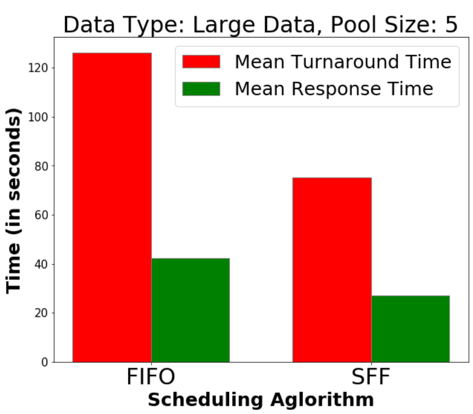


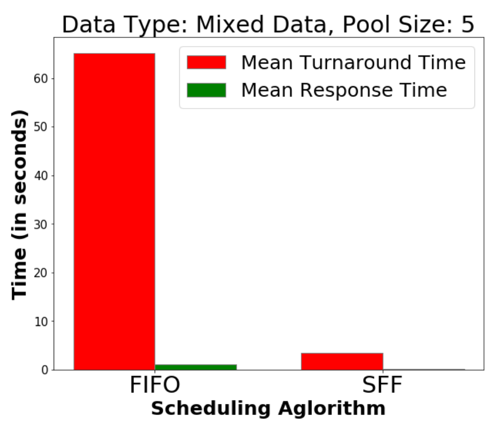
Inferences:
1. Here, we can see that the **minimum turnaround time** consistently decreases as we increase the number of threads. Further, it is nearly 0 as the first client request attended to immediately.
2. Like the last 2 cases, the **minimum response time** goes on decreasing as the number of threads increases.

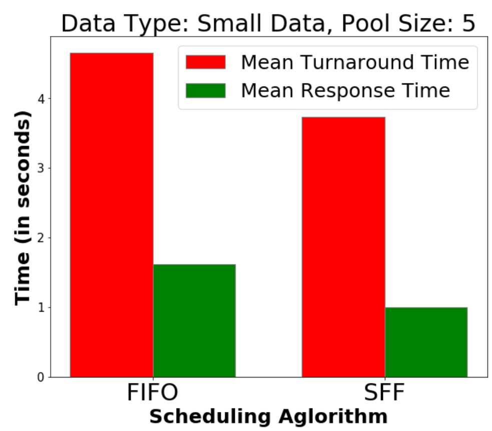
Effect of different process mix (dataset type):

To test the effect of different scheduling algorithms on different dataset types, we fix the number of client requests to 10 and the number of threads in the pool to 5.

Results for Mean Values

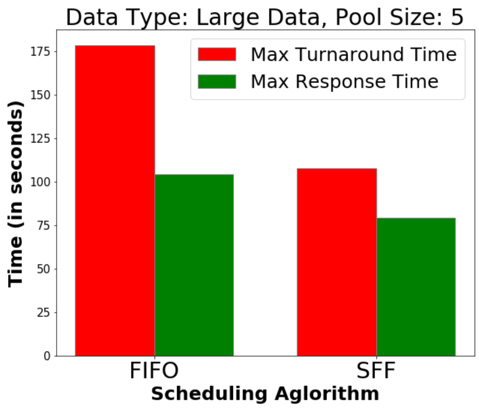


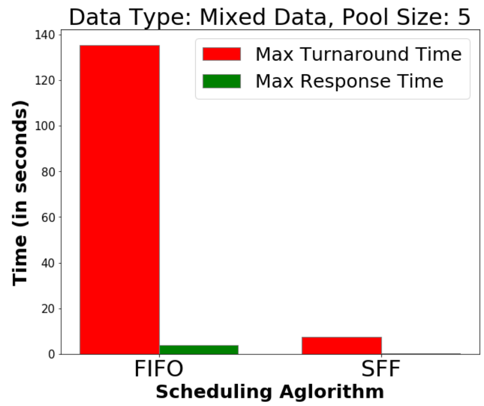


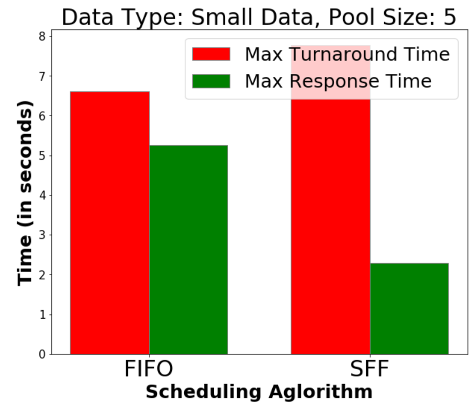


Inferences:
1. The relative difference in performance between FIFO and SFF is most apparent in the case of **Mixed Data**, where there is a mix of both large and small files. SFF can exploit this difference and schedule the smallest files first. By doing this, the **mean turnaround time was reduced** by nearly 60 seconds. The **mean response time** also reduces for SFF since the smaller files get processed quicker than the larger files, this leading to overall faster access to the processor.
2. This difference in **mean turn around time and mean response time** exists for the small and large datasets as well, though not as apparent due to the relative uniformity in file sizes.

Results for Max Values



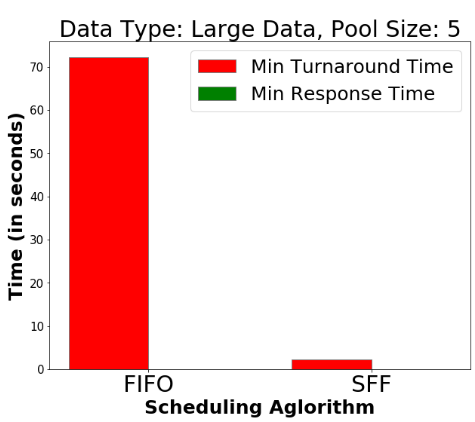


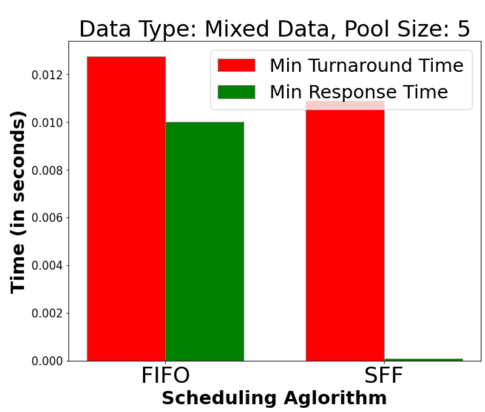


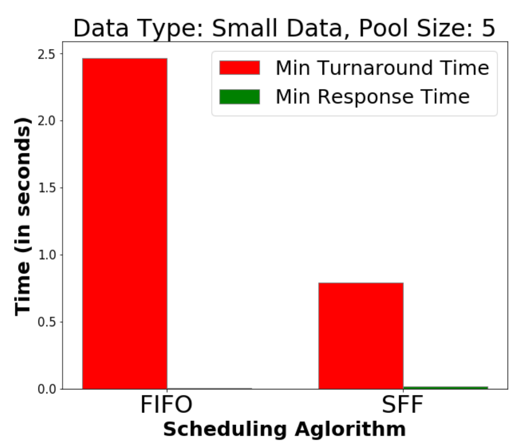
Inferences:

1. Similar to the mean value case, The relative difference in performance between FIFO and SFF is most apparent in the case of **Mixed Data**, where there is a mix of both large and small files. The **max turnaround time was reduced** by nearly 120 seconds. The **max response time** also reduces as smaller files gets processed first and faster in SFF.
2. This difference in **max turn around time and max response time** exists for the small and dataset as well, though not as apparent due to the relative uniformity in file sizes.. However, the **max turnaround time increases** for the SFF in the case of the large dataset. This may be because the largest file gets scheduled to be handled last, and it also takes the longest to process, leading to the larger turnaround time when compared to FIFO.

Results for Min Values







Inferences:

1. The **minimum response time** across all dataset types and scheduling algorithms is close to 0 as the first client request gets attended to immediately.

2. For both the large and small datasets, we can see the **minimum turn around time** decrease when using SFF as the smallest file gets scheduled first, which also requires the least processing time.

<u>Overall Inferences for both experiments:</u>

1. It is clear that SFF is outperforming FIFO in almost all cases when the number of threads is lesser than the number of submitted processes. The difference in turnaround and response time is not significant in the case of 10 threads, as no process is kept waiting, so the scheduling algorithms do not significantly come into play.
2. The processes get an in general faster response and turnaround times when the number of threads are increased. This is expected, because fewer processes are kept in a waiting state.
3. For processes requiring larger amounts of time (Large Data), SFF performs much better. It also performs better in the other two cases of process mixes, but the increase in performance is more for a process mix with higher number of cpu-bound processes.

<u>Using Sleep:</u> Let's say we send 20 client requests simultaneously, it is highly unlikely that all these 20 requests would be stored in the buffer at the same time. This is because the threads are also checking the buffer with every addition of the client. So at one time, there might be a subset of requests in the buffer, and on these requests the required algorithm will be implemented. If we want to explicitly view the results for the 20 requests together, we can use *sleep(10 seconds)* in the thread function so that all the threads will wait until the buffer is full(which takes around 1-2 seconds), and then the requests based on the scheduling policy are handled.

A screenshot when using sleep(10) is attached below:

```
abhigyangargav@ideapad330:~/Downloads/3/Code/Code$ ./target FIFO
pid=1, size=340496
pid=2, size=709424
pid=3, size=732848
pid=4, size=1057856
pid=5, size=826544
pid=6, size=1125200
pid=7, size=1368224
pid=8, size=949520
pid=9, size=1716656
pid=10, size=665504
pid=11, size=1016864
pid=12, size=849968
pid=13, size=1631744
pid=14, size=1133984
pid=15, size=598160
pid=16, size=1081280
pid=17, size=1008080
pid=18, size=806048
pid=19, size=1090064
pid=20, size=978800
pid=21, size=926096
^C
abhigyangargav@ideapad330:~/Downloads/3/Code/Code$ ./target SFF
pid=15, size=340496
pid=16, size=598160
pid=14, size=665504
pid=9, size=709424
pid=3, size=732848
pid=12, size=806048
pid=5, size=826544
pid=17, size=849968
pid=18, size=926096
pid=2, size=949520
pid=11, size=978800
pid=1, size=1008080
pid=8, size=1016864
pid=7, size=1057856
pid=20, size=1081280
pid=6, size=1090064
pid=10, size=1125200
pid=4, size=1133984
pid=13, size=1368224
pid=19, size=1631744
pid=21, size=1716656
```

Issues with our Design:

The main issue in our design is that it is not at all secure; it lets the client ask for whatever data it wants and allows every request from the client.