**a)**

**Low Utilization:**

**Results from running the p parameter 10 times each for FIFO and SJN:**

**FIFO**

| Index | Elapsed (wall clock) time ( m:ss.ms) |
| --- | --- |
| 1 | 3:42.36 |
| 2 | 2:51.06 |
| 3 | 2:35.44 |
| 4 | 2:34.43 |
| 5 | 2:36.22 |
| 6 | 2:35.98 |
| 7 | 2:34.13 |
| 8 | 2:34.94 |
| 9 | 2:58.67 |
| 10 | 2:33.90 |

(mmx60) + ss for all 10

FIFO Average = 165.713 seconds

**SJN**

| Index | Elapsed (wall clock) time ( m:ss.ms) |
| --- | --- |
| 1 | 2:57.40 |
| 2 | 2:34.77 |
| 3 | 2:34.23 |
| 4 | 2:45.44 |
| 5 | 2:40.57 |
| 6 | 2:55.86 |
| 7 | 2:36.05 |
| 8 | 2:34.27 |
| 9 | 2:34.89 |
| 10 | 2:36.94 |

(mmx60) + ss for all 10

SJN Average = 161.042 seconds

**High Utilization:**

**Results from running the p parameter 10 times each for FIFO and SJN:**

**FIFO**

| Index | Elapsed (wall clock) time ( m:ss.ms) |
| --- | --- |
| 1 | 1:43.59 |
| 2 | 0:41.94 |
| 3 | 0:46.31 |
| 4 | 0:42.35 |
| 5 | 0:41.87 |
| 6 | 0:41.29 |
| 7 | 0:40.68 |
| 8 | 0:44.37 |
| 9 | 0:40.48 |
| 10 | 0:40.98 |

(mmx60) + ss for all 10

FIFO Average = 48.386 seconds

**SJN**

| Index | Elapsed (wall clock) time ( m:ss.ms) |
| --- | --- |
| 1 | 0:46.45 |
| 2 | 0:40.24 |
| 3 | 0:41.20 |
| 4 | 0:41.40 |
| 5 | 0:41.87 |
| 6 | 0:40.37 |
| 7 | 0:40.32 |
| 8 | 0:41.58 |
| 9 | 0:40.54 |
| 10 | 0:42.15 |

(mmx60) + ss for all 10

SJN Average = 41.512

**Explanation:**

In low utilization, the average runtimes for both FIFO and SJN are nearly identical, which is as expected as we saw in lecture that scheduling doesn’t really matter at lower utilizations, with only a marginal improvement for SJN (161.042 < 165.13). Re-ordering does little to improve utilization since the resources are under low loads.

However, as expected, In high utilization, SJN outperforms FIFO (41.412 < 48.386) as it optimizes for shorter jobs, which leads to lower average runtimes. The time it takes to re-order significantly increases the overall improvement, so the slowdown due to becomes negligible. T

**b)**

Here, I first wrote a simple bash script to automate the server’s running multiple times. After I collated all the logs, I then ran the below python file, which computes the overall busy time and divides the time the server was active, which totals the time each worker was busy.

import os

import re

import numpy as np

import matplotlib.pyplot as plt

# Parameters

POLICIES = ["FIFO", "SJN"]

ARRIVAL\_RATES = [22, 24, 26, 28, 30, 32, 34, 36, 38, 40]

WORKERS = 2

# Initialize data structures

data = {policy: {'utilization': [], 'response\_time': []} for policy in POLICIES}

# Regular expressions to parse log files

request\_pattern = re.compile(r'R(\d+):([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+)')

worker\_pattern = re.compile(r'T(\d+) R(\d+):([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+)')

for policy in POLICIES:

for arr\_rate in ARRIVAL\_RATES:

server\_log\_path = f'Eval\_B\_logs/{policy}/server\_{arr\_rate}.log'

worker\_busy\_times = {}

total\_response\_time = 0.0

num\_requests = 0

start\_times = []

completion\_times = []

with open(server\_log\_path, 'r') as f:

for line in f:

# Match lines indicating request completion

match = worker\_pattern.match(line.strip())

if match:

# Extract data

worker\_id = int(match.group(1))

req\_id = int(match.group(2))

req\_timestamp = float(match.group(3))

req\_length = float(match.group(4))

receipt\_time = float(match.group(5))

start\_time = float(match.group(6))

completion\_time = float(match.group(7))

# Calculate busy time for this request

busy\_time = completion\_time - start\_time

# Update worker busy times

if worker\_id not in worker\_busy\_times:

worker\_busy\_times[worker\_id] = 0.0

worker\_busy\_times[worker\_id] += busy\_time

# Calculate response time

response\_time = completion\_time - req\_timestamp

total\_response\_time += response\_time

num\_requests += 1

start\_times.append(start\_time)

completion\_times.append(completion\_time)

# Calculate total busy time

total\_busy\_time = sum(worker\_busy\_times.values())

# Calculate total runtime

if completion\_times:

total\_runtime = max(completion\_times) - min(start\_times)

total\_capacity = total\_runtime \* WORKERS

server\_utilization = total\_busy\_time / total\_capacity

else:

server\_utilization = 0.0

# Calculate average response time

if num\_requests > 0:

avg\_response\_time = total\_response\_time / num\_requests

else:

avg\_response\_time = 0.0

data[policy]['utilization'].append(server\_utilization)

data[policy]['response\_time'].append(avg\_response\_time)

print(f"Policy: {policy}, Arrival Rate: {arr\_rate}, Utilization: {server\_utilization:.4f}, Avg Response Time: {avg\_response\_time:.4f}")

# Plotting

plt.figure(figsize=(10, 6))

for policy in POLICIES:

plt.plot(data[policy]['utilization'], data[policy]['response\_time'], marker='o', label=policy)

plt.xlabel('Server Utilization')

plt.ylabel('Average Response Time')

plt.title('Average Response Time vs Server Utilization')

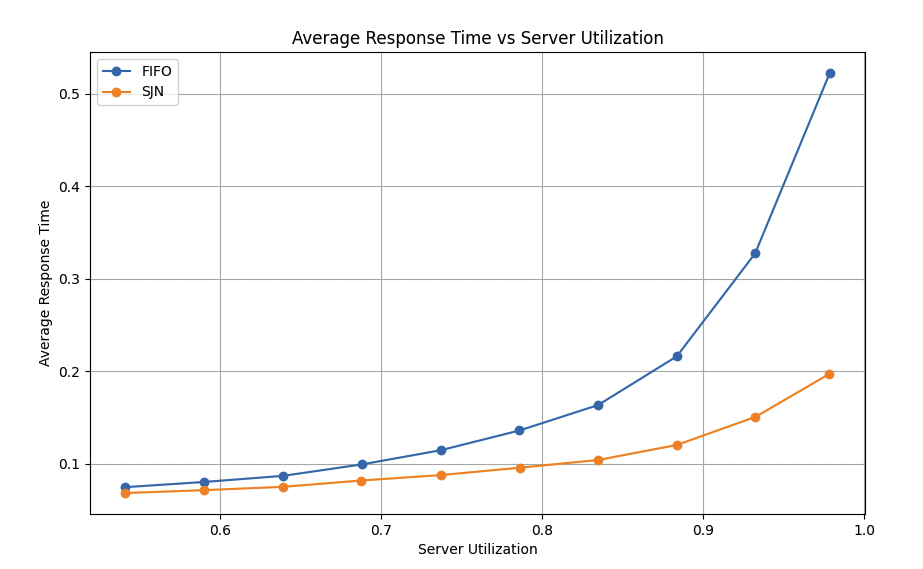
plt.legend()

plt.grid(True)

plt.savefig('response\_time\_vs\_utilization.png')

plt.show()

**Output:**



So we can clearly see here that FIFO and SJN perform relatively similarly for lower utilization rates, but as the server utilization increases, the average response rate for FIFO grows significantly faster (almost exponentially) compared to SJN. At max utilization, the average response time is about 0.3 seconds faster for SJN. FIFO seems to be 1.5x slower than SJN at max utilization (0.5-0.2/0.3)

**C)**

### **Overview of Steps Taken:**

1. **Extract Response Times**: Parse the server log files from the 10th run for both FIFO and SJN policies to extract the response times of each request.
2. **Calculate Statistics**: Compute the average response time and the 99th percentile response time for each policy.
3. **Generate CDF Plots**: Plot the CDF of the response times for each policy so that both plots have the same x and y axis ranges for visual clarity.
4. **Add Vertical Lines**: On each plot, add vertical lines to mark the average response time and the 99th percentile response time.

**Python Script:**

import re

import matplotlib.pyplot as plt

import numpy as np

# Define the policies and the corresponding log files

policies = {

'FIFO': 'EVAL\_B\_logs/FIFO/server\_40.log',

'SJN': 'EVAL\_B\_logs/SJN/server\_40.log'

}

# Regular expression to parse the log lines

log\_pattern = re.compile(

r'T(\d+) R(\d+):([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+),([\d\.]+)'

)

# Dictionary to store response times for each policy

response\_times = {}

for policy, log\_file in policies.items():

response\_times[policy] = []

with open(log\_file, 'r') as f:

for line in f:

match = log\_pattern.match(line.strip())

if match:

# Extract the timestamps

req\_timestamp = float(match.group(3))

completion\_time = float(match.group(7))

# Calculate response time

resp\_time = completion\_time - req\_timestamp

# Append to the list

response\_times[policy].append(resp\_time)

print(f"Policy: {policy}, Number of Requests: {len(response\_times[policy])}")

# Dictionary to store statistics

stats = {}

for policy in policies.keys():

times = response\_times[policy]

times.sort() # Sort the response times

# Calculate average response time

avg\_response\_time = np.mean(times)

# Calculate 99th percentile response time

percentile\_99 = np.percentile(times, 99)

# Store in stats dictionary

stats[policy] = {

'average': avg\_response\_time,

'percentile\_99': percentile\_99,

'sorted\_times': times # Store sorted times for CDF plot

}

print(f"Policy: {policy}")

print(f" Average Response Time: {avg\_response\_time:.4f} seconds")

print(f" 99th Percentile Response Time: {percentile\_99:.4f} seconds")

# Function to plot the CDF

def plot\_cdf(times, avg\_time, percentile\_99, policy\_name, x\_limits, y\_limits):

# Calculate the CDF values

sorted\_times = np.sort(times)

cdf = np.arange(1, len(sorted\_times)+1) / len(sorted\_times)

# Plot the CDF

plt.figure(figsize=(10, 6))

plt.plot(sorted\_times, cdf, label=f'CDF of Response Times ({policy\_name})')

# Add vertical lines for average and 99th percentile

plt.axvline(x=avg\_time, color='r', linestyle='--', label='Average Response Time')

plt.axvline(x=percentile\_99, color='g', linestyle='--', label='99th Percentile Response Time')

# Set the x and y limits to make plots comparable

plt.xlim(x\_limits)

plt.ylim(y\_limits)

# Labels and Title

plt.xlabel('Response Time (seconds)')

plt.ylabel('Cumulative Probability')

plt.title(f'CDF of Response Times - {policy\_name} Policy')

plt.legend()

plt.grid(True)

# Save the plot

plt.savefig(f'cdf\_response\_times\_{policy\_name.lower()}.png')

plt.show()

# Determine common x and y limits based on combined data for comparability

all\_times = response\_times['FIFO'] + response\_times['SJN']

x\_min = 0

x\_max = max(all\_times) \* 1.05 # Slightly greater than max to give some space

y\_min = 0

y\_max = 1

x\_limits = (x\_min, x\_max)

y\_limits = (y\_min, y\_max)

# Plot for each policy

for policy in policies.keys():

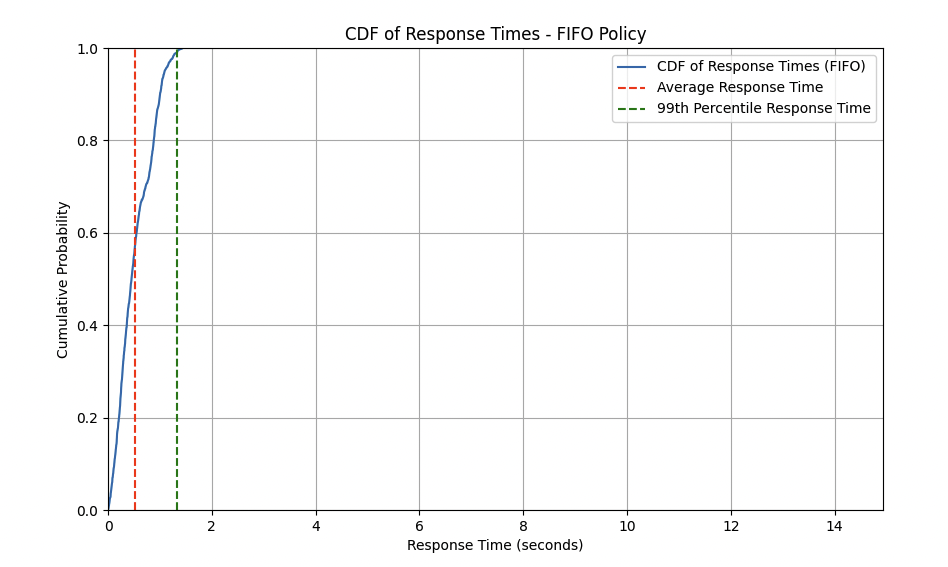
times = stats[policy]['sorted\_times']

avg\_time = stats[policy]['average']

percentile\_99 = stats[policy]['percentile\_99']

plot\_cdf(times, avg\_time, percentile\_99, policy, x\_limits, y\_limits)

**Output:**

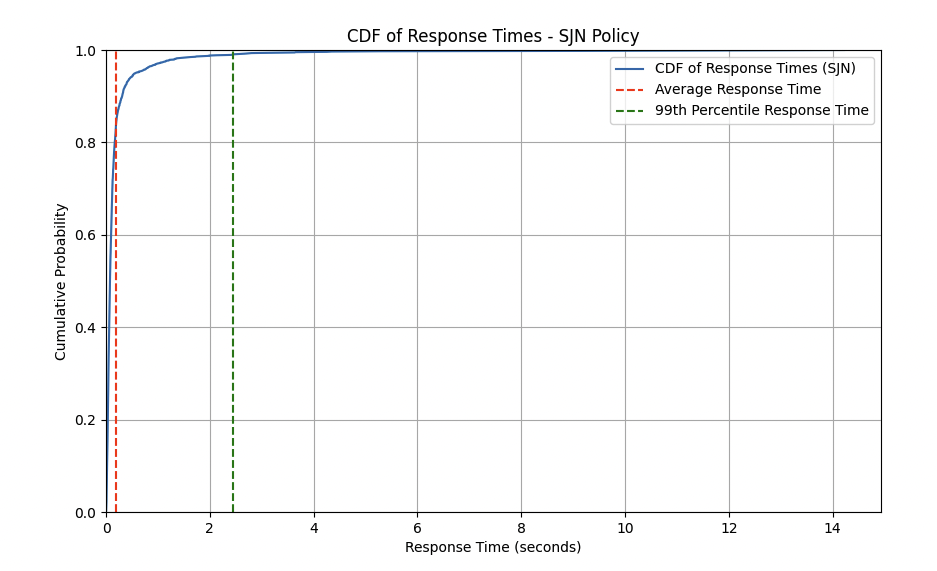


Policy: FIFO, Number of Requests: 1500

Policy: FIFO

Average Response Time: 0.5223 seconds

99th Percentile Response Time: 1.3252 seconds



Policy: SJN, Number of Requests: 1500

Policy: SJN

Average Response Time: 0.1970 seconds

99th Percentile Response Time: 2.4490 seconds

**D)**

**FIFO CDF Plot**:

* The average response time is quite close to the 99th percentile response time.
* This indicates that most requests experience fairly similar response times, with fewer outliers or extreme values.
* The tail is shorter, meaning fewer requests experience significantly longer delays.
* This makes FIFO a more predictable policy in terms of uniformity across all requests. Even though FIFO may not optimize the fastest response times, it does ensure that the majority of requests experience relatively similar delays.

**SJN CDF Plot**:

* The 99th percentile response time is much higher than the average response time, indicating that while many requests are processed quickly, some outliers experience very long response times.
* This results in a longer tail for SJN, meaning that while the majority of requests are handled faster than FIFO, a small percentage experience delays.
* SJN prioritizes shorter requests, meaning that while most requests are handled very quickly (faster than in FIFO), the system is less predictable because some longer requests experience significant delays.

In regards to the definition of predictability in the textbook, the difference between the best-case and worst-case behavior is slight, indicating a more predictable system under FIFO.