Aaron P. Mills

Z-23547104

Dr. Borivoje Furht

Process Scheduler Simulation

*with FCFS, SJF, MQFL, & RR*

Introduction to Operating Systems - COP 4610

21 October 2022

<https://colab.research.google.com/drive/1SA_-qXI2mgWt4KvRldfLDkh7r89R4-Yn?usp=sharing>

**Table of Contents**

|  |  |
| --- | --- |
| Page Name | # |
| Cover | 1 |
| Table of Contents | 2 |
| Introduction | 3 |
| Charts | 4-6 |
| Discussion & Final Results | 7 |
| Sample Outputs | 8-10 |
| Sample Results | 11-12 |
| Conclusion | 13 |
| Source Code | 14-24 |

Introduction

Me

My name is Aaron Mills. My interests are Japanese and business/investing/entrepreneurialism. My goals are to help lots of people, earn living independence, become verbally fluent, and generate wealth. Of course I have many goals, but those are my priorities. I share this because I wanted to introduce my scheduling algorithm and priorities. That being said, I just figured out how I should introduce the report. Hopefully I remember to delete this, maybe.

This Course

In this Operating Systems course, professor Furht had taught us scheduling algorithms, many of which are still used today. The four my classmates and I are supposed to simulate for this project include: First-Come-First-Serve, Shortest-Job-First, Round-Robin, and Multi-Level-Frequency-Queues, with the later uncooperating RR into Queue 1 and Queue 2. I don’t want to go too much in detail here, as it’s only the introduction; though, I do feel it necessary to fill up the page. I’ll share a little more.

How it All Began: My Approaching Algorithm

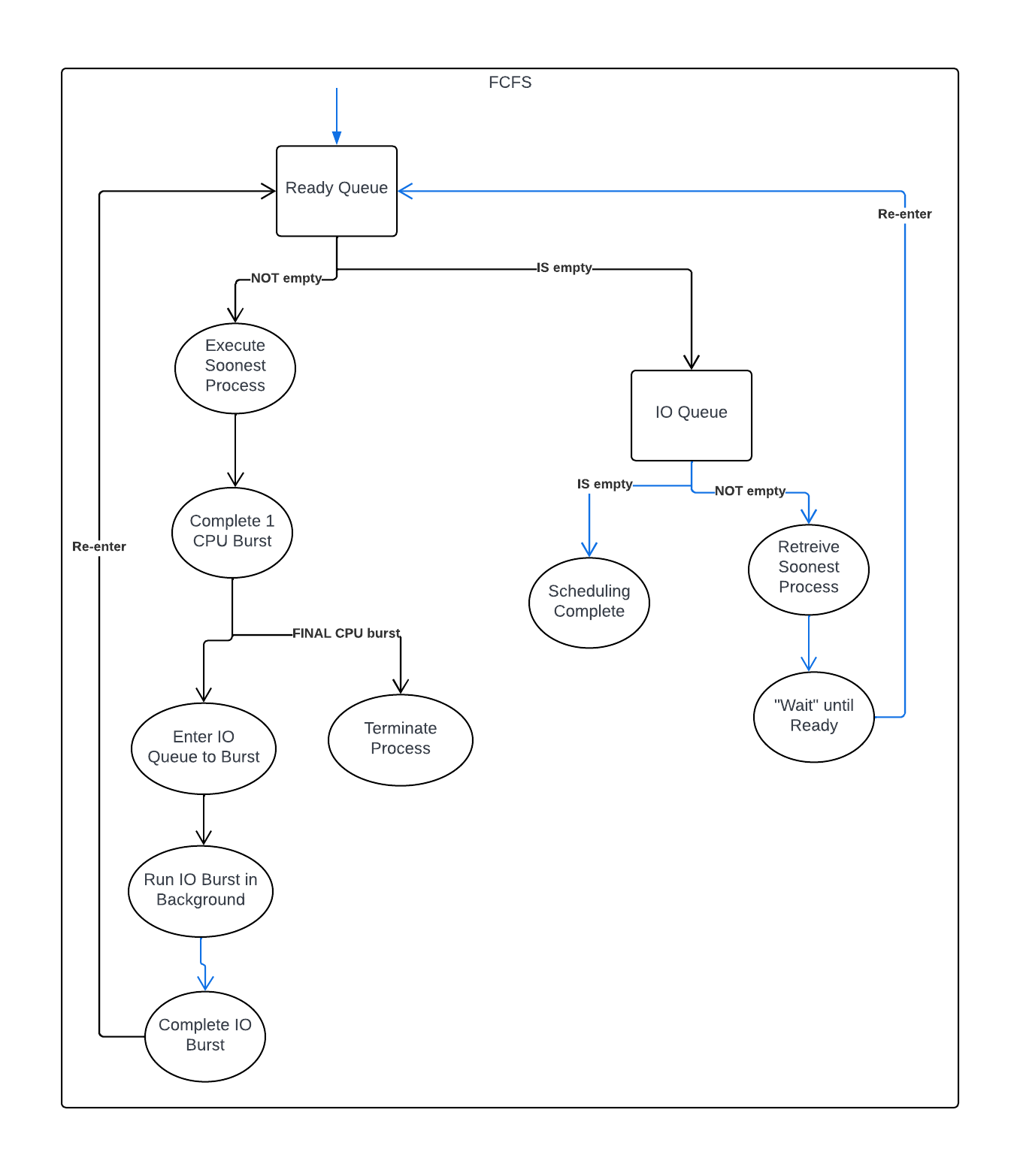
When understanding how to start the first algorithm, I went back to professors’ previous lectures in September. There I found two examples for FCFS, and one for SJF. RR was rather simple, so the examples didn’t help too much other than for clarification. Unfortunately, there was no IO involved in professors’ RR examples, so I relied on the definition that IO continues once CPU ends. MLFQ seemed like common sense, so I didn’t bother with intensely reviewing the lectures. Plus, by the time I had reached that point, I already made FCFS, SJF, & RR.

My technique involved making functions for repeated activity & testing them on the small sample inputs professor provided in class. This was very useful, as it made sure my code was at least somewhat correct before tackling the larger the inputs.

Speaking of code, originally I was going to write the code in C++, because that’s what I was taught mostly and comfortable with from practicing on Leetcode. Additionally, my elder brother used C++ (and no, I did not contact him for any type of help or reference at all on this project, something I’m proud of.) I also considered doing C# because I had a little ego and thought I’d could eliminate two birds with one stone by learning the language for my other course and doing this project. But after actually using my brain, I decided I’d rather read Japanese than deal with CPU memory problems, and my own memory problems. Thus, I decided to use Python on Google Colab. This was really convenient as I’m familiar with Colab and Python and didn’t need to bother with software lag/issues (like VSC…) The coding portion took me a few days. Now that my approach algorithm is understood, I shall end the introduction.

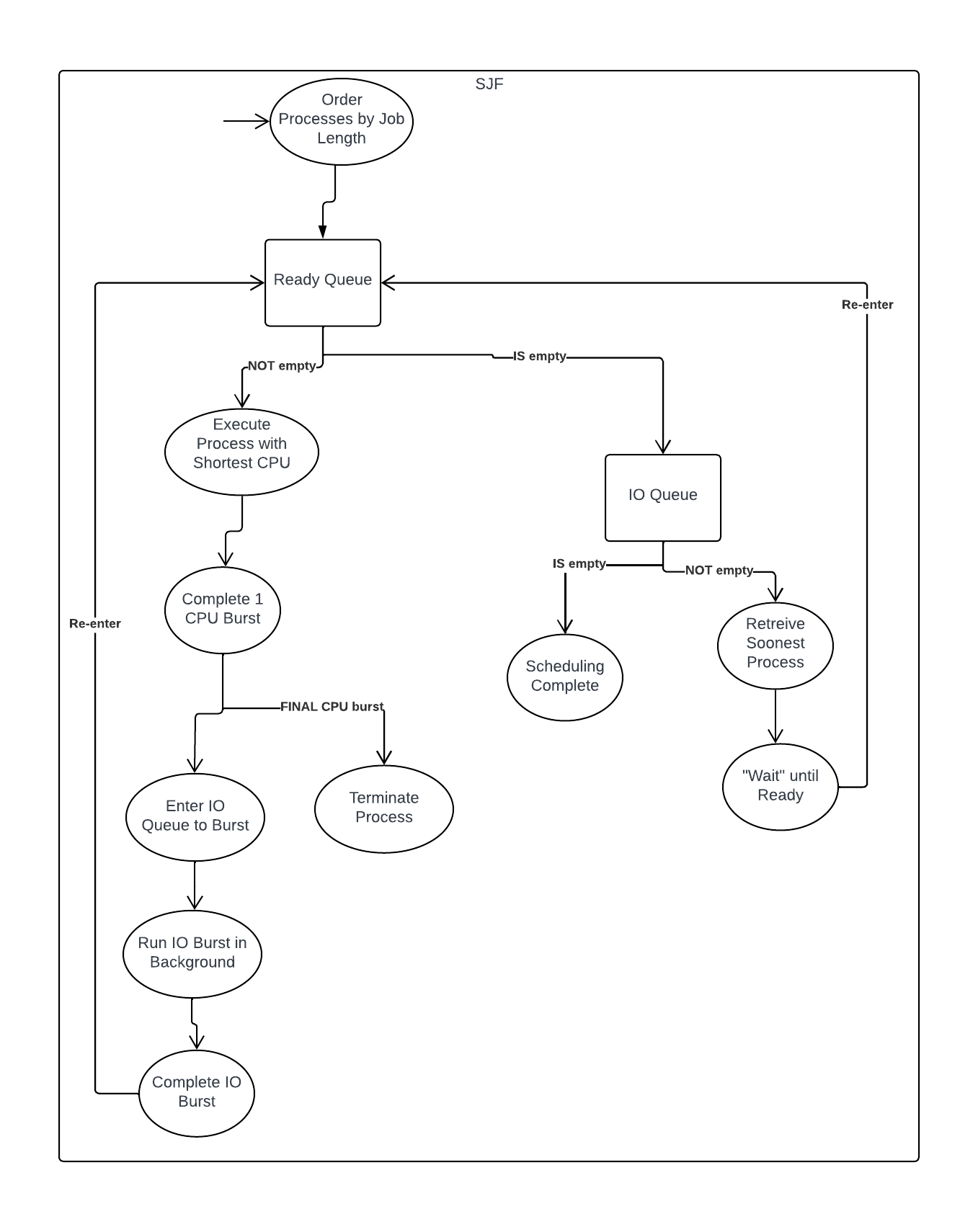
Charts

First-Come-First-Serve

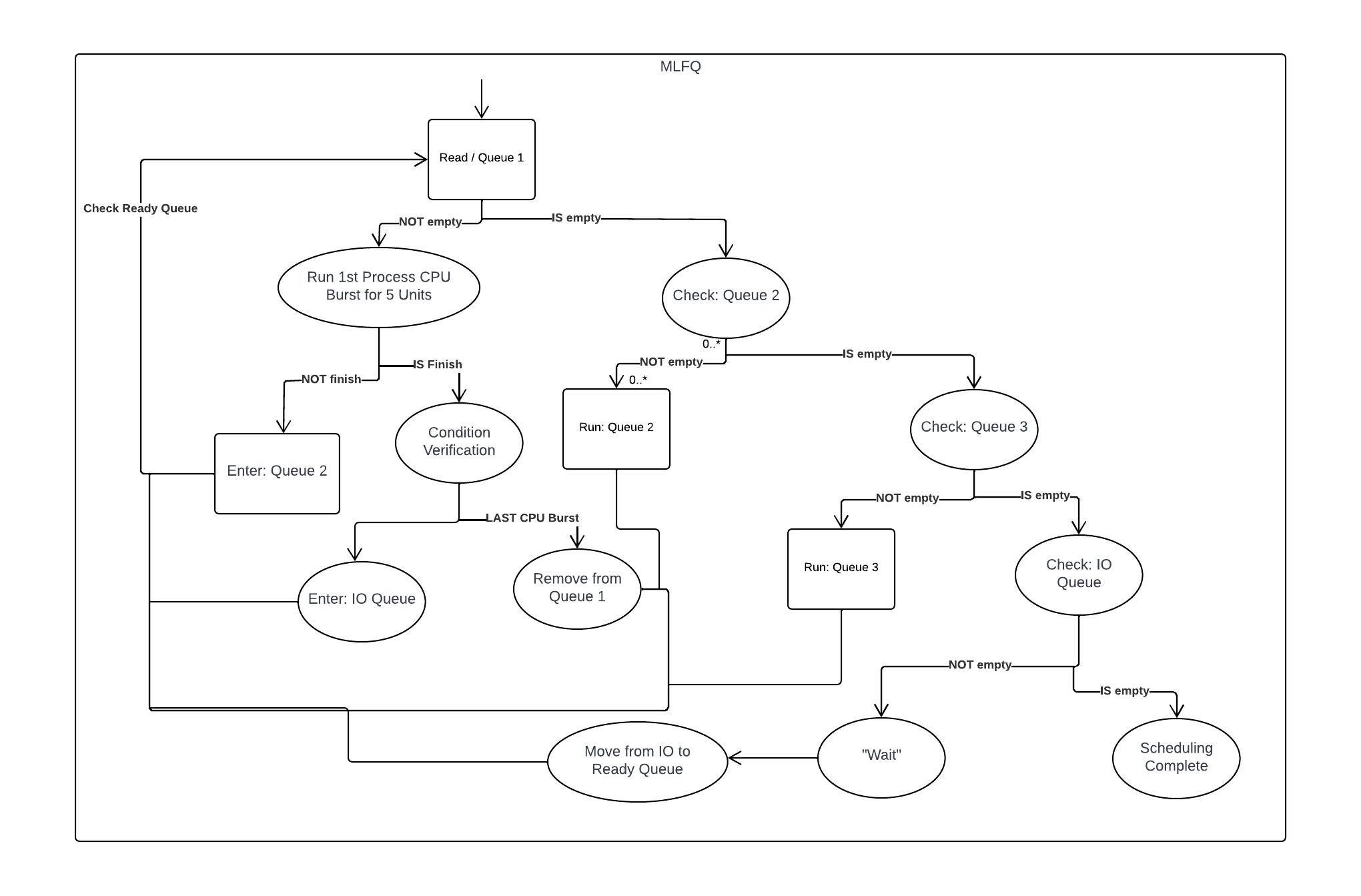


Description: This is the general logic of how my code is organized. It’s important to note that order is maintained when processes enter IO Queue and Ready Queue, so it is quick to access the soonest available process (the process who arrives first.) This logic is later used for MLQF.

Shortest-Job-First

Description: Unfortunately, SJF isn’t too much different from FCFS, hence why the charts look similar. The main difference is that processes must be ordered based on job length before placed into Ready Queue. Furthermore, the shortest-job-length ordering is maintained throughout the program. The IO Queue is still based on FCFS, in the sense that, when Ready is empty, we retrieve the soonest process to avoid waiting long time periods for small processes. Otherwise, we retrieve all processes who finish IO & place them in Ready whilst maintaining order.

Multi-Level-Frequency-Queues



Description: Notice that we, the program, you, and I; constantly check Ready Queue after running/inserting a process through Queue 2, Queue 3, & IO. This due to the priority. Though IO Queue has (technically) the lowest priority, when a process is ready, it’s priority is instantly at its highest. Queue 2 runs basically the same as Queue 1, but with tq=10 and the fact that it is not prioritized as high. Queue 3 is simply FCFS, a diagram I’ve already drawn, except upon IO completion, the process goes to Queue 1. In my code, I called Queue 1 the Ready Queue because it had highest priority and is where all processes go when instantly ready; however, I name it Queue 1 in display and in my references for simplicity’s sake. In these diagrams, I don’t include buttons, prints, displays, etc.. because they are technically not part of the simulation algorithm and drift away from the focus of this course (something I need not do anymore!)

Note: Also, professor had not specified whether MLFQ was non-preemptive or not, but I do recall him stating in previous lectures that we would be simulating non-preemptive for this course, so I had coded under that assumption.

Discussion & Final Results

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SJF  Total Time: 668 units  Utilization: 82.78%  Thruput:≈0.01198 | | | | FCFS  Total Time: 648 units  Utilization: 85.34%  Thruput:≈0.01235 | | | | | MLFQ  Total Time: 630 units  Utilization: 87.78%  Thruput: ≈0.01270 | | | |
|  | Tw | Ttr | Tr |  | Tw | Ttr | Tr |  | | Tw | Ttr | Tr |
| P1 | 43 | 268 | 11 |  | 170 | 395 | 0 |  | | 34 | 259 | 0 |
| P2 | 73 | 500 | 3 |  | 164 | 591 | 5 |  | | 118 | 545 | 5 |
| P3 | 276 | 668 | 16 |  | 165 | 557 | 9 |  | | 224 | 616 | 9 |
| P4 | 50 | 534 | 0 |  | 164 | 648 | 17 |  | | 95 | 579 | 14 |
| P5 | 237 | 546 | 109 |  | 221 | 530 | 20 |  | | 321 | 630 | 17 |
| P6 | 121 | 336 | 24 |  | 230 | 445 | 36 |  | | 197 | 412 | 22 |
| P7 | 149 | 477 | 47 |  | 184 | 512 | 47 |  | | 261 | 589 | 27 |
| P8 | 119 | 428 | 7 |  | 184 | 493 | 61 |  | | 233 | 542 | 32 |
| Avg | 133.5 | 469.62 | 27.12 |  | 185.25 | 521.38 | 24.38 |  | | 185.38 | 521.5 | 15.75 |

If We Want the Fastest Algorithm

If our goal was to run all 8 processes as fast as possible, we would use SJF. This is because it has the least amount of average waiting time and smallest turnaround time. SJF is often most fastest but is difficult to achieve in practice due to the nature of not knowing process burst times before scheduling.

If we Want to use the Most of Given Time

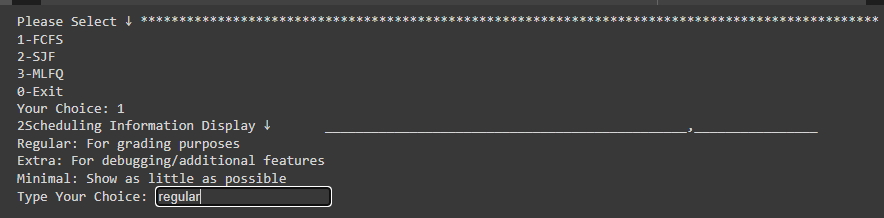
Suppose we had to finish within a certain amount of time as much processes as possible. Our best algorithm, for this case, would be the MLFQ since it has the highest utilization and thruput. This ensures we are efficient with our time, as CPU bursts will occupy more of it.

Middle Ground

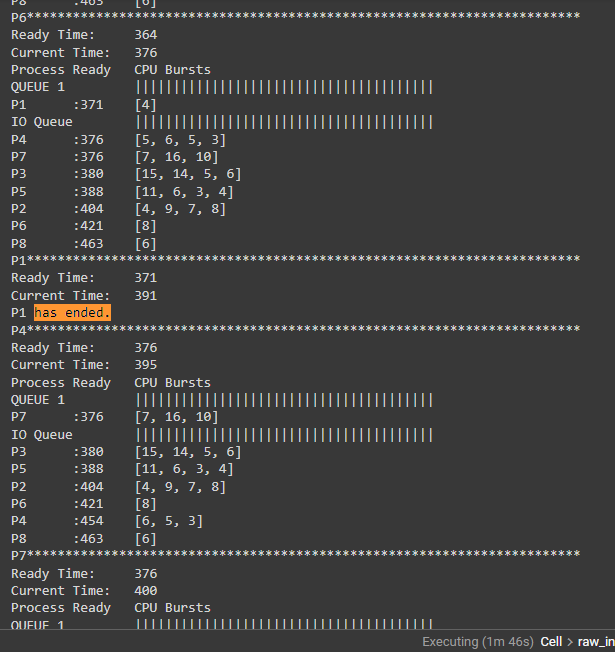
FCFS acts as a middle ground in this case. Contextually speaking, is the worst among the three, as it does not finish the fastest nor uses the time most optimally. However, there still might be usage for the algorithm if performed under another context or with different processes.

Sample Outputs

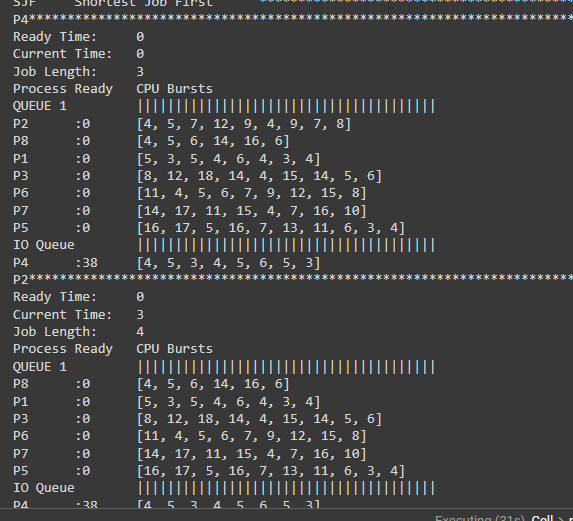
User-Input



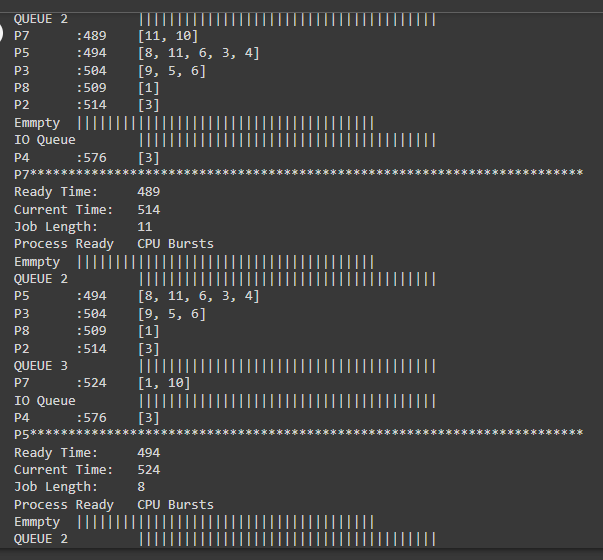
FCFS



SJF

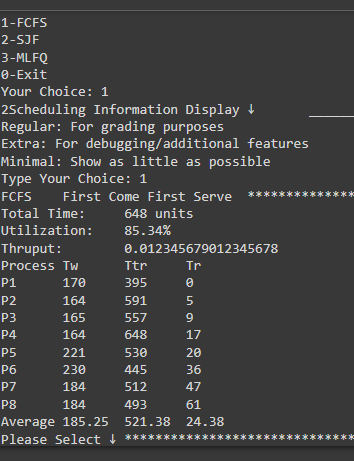


MLFQ

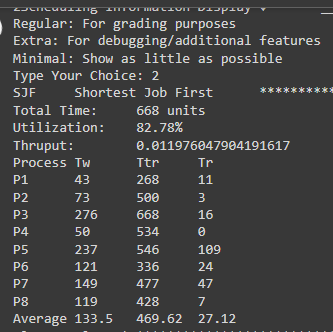


Sample Results

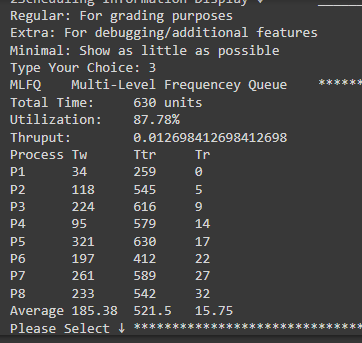
FCFS



SJF



MLFQ



Conclusion

In the end, this project took a lot of time. Well, a lot time for me; I haven’t checked the CPU’s runtimes. The coding part was fun and I literally spend days and nights only coding. I don’t think I have a passion for coding bur it’s been awhile since I’ve immersed into that type of energy. I found this learning material interesting and was satisfied it could all be done with just math and no libraries. Furthermore, I found Google Colab to be extremely useful near the end, especially when it would periodically break for no reason. Sometimes it felt like the compiler decided to break the rules or make up it’s own. Either I’m sleepy or tired, but I believe the compiler deleted one of my functions right when I finished. Thankfully, I could find my edit history and revise previous versions.

Overall, this project was entertaining and time consuming. I don’t feel any sense of fulfillment like professor mentioned, but I am happy I will finally be able to continue studying Japanese again. I wonder if I would have more time if I implemented these algorithms into my life…

Apologies, yes this project was cool, I enjoyed writing code. I did not enjoy making charts. Writing essays and code is fun but charts are a real pain. Additionally, I wish Microsoft Word had better options for importing photos to make the document neater. Thanks for reading my optional conclusion. Below you will find my code. And here’s a link:

<https://colab.research.google.com/drive/1SA_-qXI2mgWt4KvRldfLDkh7r89R4-Yn?usp=sharing>

Source Code

# header: includes classes, functions, libraries, etc..

######################################################################################################################################################################

class process:

    def \_\_init\_\_(self,name, bursts):

        self.name = name        #process name

        self.bursts = bursts    #all the bursts

        self.cpu = []           #cpu bursts

        self.io = []            #io bursts

        for i in range(len(bursts)):

            if i%2 == 0:        #get CPU bursts

                self.cpu.append(bursts[i])

            elif i%2 == 1:      #get IO bursts

                self.io.append(bursts[i])

        self.rdy = 0                #ready time: when the process is able to cpu burst

        self.job = self.cpu[0]      #job time length: the length of the next cpu burst to be executed

        self.pr = 1                 #priority

        self.tw = 0                 #process wait time: time spent in ready queue due to other process hogging cpu with their burst

        self.ttr = 0                #process turnaround time: time process fully executes - response time

        self.tr = -1                #process response time: the start for the first cpu burst

    def ended(self):        #determine if processes has ended: completed it's total execution

        if self.cpu == []:

            return True

def initializer(\*\*args):    #initialize the processes

    if(args == {}):

        p1=process("P1",[5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4])

        p2=process("P2",[4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8])

        p3=process("P3",[8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6])

        p4=process("P4",[3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3])

        p5=process("P5",[16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])

        p6=process("P6",[11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8])

        p7=process("P7",[14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])

        p8=process("P8",[4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6])

        group = [p1,p2,p3,p4,p5,p6,p7,p8]

    return group

def display(group,time,waste):         #display total time, utilization thruput, tw, ttr, tr, averages

    total\_time = round(time)                                    #total time to run all process

    utilization = round(((total\_time-waste)/total\_time)\*100,2)  #prof rounded to nearest tenth place, so I did, too!

    thruput = len(group)/total\_time                             #thruput = #process/total time

    print(f"Total Time:\t{total\_time} units\nUtilization:\t{utilization}%\nThruput:\t{thruput}")

    a\_tw = 0    #average waiting time

    a\_ttr = 0   #average turnaround time

    a\_tr = 0    #average response time

    num\_p = len(group)  #number of processes

    print(f"Process\tTw\tTtr\tTr")

    for p in group:     #calculate sum for averages

        print(f"{p.name}\t{round(p.tw)}\t{round(p.ttr)}\t{round(p.tr)}")

        a\_tw += p.tw

        a\_ttr += p.ttr

        a\_tr += p.tr

    a\_tw /= num\_p       #finish calculating averages

    a\_ttr /= num\_p

    a\_tr /= num\_p

    print(f"Average\t{round(a\_tw,2)}\t{round(a\_ttr,2)}\t{round(a\_tr,2)}")

def display\_options(mode):          #display viewing options

    mode = mode.lower()

    if (mode == 'regular'):

        regular = True

        extra = False

    elif (mode == 'extra'):

        regular = True

        extra = True

    else:

        regular = False

        extra = False

    return regular,extra

def pdata(\*args):           #display queue data if not empty

    groups = args

    print("Process\tReady\tCPU Bursts\t")

    for group in groups:

        if not (group==[]):

            if not (group[0].pr==0):

                print(f"QUEUE {group[0].pr} \t|||||||||||||||||||||||||||||||||||||||")

            else:

                print(f"IO Queue\t|||||||||||||||||||||||||||||||||||||||")

        else:

            print("Emmpty \t|||||||||||||||||||||||||||||||||||||||")

        for p in group:

            print(f"{p.name}\t:{p.rdy}\t{p.cpu}")

#insert in order of rdy time or job length

def insort(group,g, metric):

    if group == []:

        group.append(g)     #if empty, add g to the sorted list

    elif metric == 'rdy':   #FCFS is based on ready time

        for i in range(len(group)):

            p = group[i]

            if (g.rdy < p.rdy):

                group.insert(i,g)   #insert in ascending order of rdy time (time available)

                break

        else:

            group.insert(i+1,g)

    elif metric == 'job':       #SJF is based on job length

        for i in range(len(group)):

            p = group[i]

            if (g.job < p.job):

                group.insert(i,g)   #insert in ascending order of job length

                break

        else:

            group.insert(i+1,g)

def FCFS(group,mode='regular'):            #First Come First Serve

    print("FCFS\tFirst Come First Serve \t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    #in FCFS, first process that arrives gets to run 1 cpu AND 1 io burst; p# CAN run CPU during q#'s io burst.

    regular,extra = display\_options(mode)

    time = 0                #current execution time in the timing diagram

    waste = 0               #time NOT used for any cpu process bursts

    ready = group.copy()    #all process arrive at the same time, so are all in ready queue

    io\_q = []               #the not ready queue: processes in IO burst

    while not (ready==[] and io\_q==[]):  #while there ARE processes to run

        if(ready == []):    #if there are NO more ready processes

            io\_q[0].pr = 1  #priority for labeling

            ready.append(io\_q.pop(0))   #get the earliest process from sorted io queue

            continue                    #and run it

        else:                           #otherwise

            p = ready[0]                #get the process at front of rdy queue

        if (regular):

            print(f"{p.name}\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

            print(f"Ready Time:\t{p.rdy}\nCurrent Time:\t{time}")

        if(p.rdy > time):           #if NO processes are ready (i,e: in IO burst)

            if (extra):

                print(f"{p.name} is NOT ready yet!!!")

            waste += p.rdy-time     #record the time NOT used for any process bursts

            time = p.rdy            #skip to when process IS ready

        p.tw += time-p.rdy          #current time - the time p's been ready = the time p's been waiting

        if (extra):

            print(f"{time} to {round(time+p.cpu[0],2)} = {p.name} cpu")

        if (p.tr < 0):              #if NO response time has been assigned, then this is the first cpu burst

            p.tr = time             #record the response time

        time += p.cpu.pop(0)        #run cpu burst, add it to the time spent

        if (p.ended()):             #if process ended

            if(regular):

                print(f"{p.name} has ended.")

            p.ttr = time            #record the turnaround time

            ready.pop(0)            #remove it from the queue

            continue                #and continue to next process

        if(extra):

            print(f"{time} to {round(time+p.io[0],2)} = {p.name} io")

        p.rdy = round(time+p.io.pop(0),2)   #otherwise, run io burst & record when process will be ready

        if(extra):

            print(f"{p.name} in io queue, next cpu burst: {p.cpu[0]}")

        p.pr = 0

        insort(io\_q,p,'rdy')            #add process to the io queue because it is in io.

        if not(io\_q == []) and (io\_q[0].rdy <= time):  #if there ARE processes in io, add the first one who IS ready to ready q

            io\_q[0].pr = 1

            ready.append(io\_q.pop(0))   #remove the ready process from io and into ready queue

        ready.pop(0)        #finally remove process that just ran from ready queue

        if (regular):

            pdata(ready,io\_q)

    display(group,time,waste)

def SJF(group,mode='regular'):             #Shortest Job First

    print("SJF\tShortest Job First \t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    #in SJF, we prioritize shortest available CPU burst, if not available, the the soonest available CPU burst

    regular,extra = display\_options(mode)

    time = 0                #current execution time in the timing diagram

    waste = 0               #time NOT used for any cpu process bursts

    ready = []              #ready queue starts off empty, based on shortest job, NOT arrival time

    for g in group:

        g.pr = 1

        insort(ready,g,'job')   #insert based on job length

    io\_q = []

    while not (ready==[] and io\_q==[]):  #while there are processes to run

        p = ready[0]                #get the process at front of rdy queue

        if(regular):

            print(f"{p.name}\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

            print(f"Ready Time:\t{p.rdy}\nCurrent Time:\t{time}\nJob Length:\t{p.job}")

        if(p.rdy > time):           #if NO processes are ready (i,e: in IO burst)

            if(extra):

                print(f"{p.name} is NOT ready yet!!!")

            waste += p.rdy-time     #record the time NOT used for any process bursts

            if(extra):

                print(f"WASTE UPDATE ALERT: {waste} ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

            time = p.rdy        #skip to when process IS ready

        p.tw += time-p.rdy      #current time - the time p's been ready = the time p's been waiting

        if(extra):

            print(f"{time} to {round(time+p.cpu[0],2)} = {p.name} cpu")

        if (p.tr < 0):      #if NO response time has been assigned, then this is the first cpu burst

            p.tr = time     #record the response time

        time += p.cpu.pop(0)#run cpu burst, add it to the time

        if (p.ended()):     #if process ended

            if(regular):

                print(f"{p.name} has ended.")

            p.ttr = time    #record the turnaround time

            ready.pop(0)    #remove it from the queue

        else:

            p.job = p.cpu[0]    #new job length

            if(extra):

                print(f"{time} to {round(time+p.io[0],2)} = {p.name} io")

            p.rdy = round(time+p.io.pop(0),2)   #otherwise, run io burst & record when process will be ready

            if (extra):

                print(f"{p.name} in io queue, next cpu burst: {p.cpu[0]}")

            insort(io\_q,p,'rdy')    #add process to the io queue because it is in io.

            if not (ready == []):

                ready[0].pr = 0

                ready.pop(0)        #remove process from the ready queue

        #add all ready processes to ready queue in order of shortest job

        if (not(io\_q == [])):   #if there are processes in io, none ready, add the shortest one who is ready to ready q

            i=0

            while(i < len(io\_q)):   #for every process in io

                q = io\_q[i]

                if (q.rdy <= time): #if it's ready

                    q.pr = 1

                    insort(ready,io\_q.pop(i),'job') #insert into rdy queue based on job length

                    i-=1

                i+=1

        #if ready is empty, then add the soonest ready process

        if (ready == []) and (not(io\_q==[])):

            io\_q[0].pr = 1      #priority of rdy queue (labelling purposes)

            insort(ready,io\_q.pop(0),'job')

        if(regular):

            pdata(ready,io\_q)

    display(group,time,waste)

def MLFQ(group,mode=''):         #Multi-Level Queue

    print("MLFQ\tMulti-Level Frequencey Queue \t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    regular,extra = display\_options(mode)

    queue2 = [] #RR: Tq = 10

    queue3 = [] #FCFS

    tq1 = 5

    tq2 = 10

    #in this MQFL, we use RR:tq=5, RR:tq=10, FCFS for each queue, respectively.

    time = 0                #current execution time in the timing diagram

    waste = 0               #time NOT used for any cpu process bursts

    ready = group.copy()    #ready queue starts off empty, based on shortest job, NOT arrival time

    io\_q = []               #we only end when all queues are empty

    while not (ready==[] and queue2 == [] and queue3 == [] and io\_q==[]):  #while there are processes to run

        if not (ready==[]): #if queue1 is NOT empty

            curr\_q = ready  #we use it

            next\_q = queue2 #and reference queue2

            p = curr\_q[0]   #process at front of queue1

            tq = tq1        #use tq for queue 1

            pr = 2          #the next priority below queue 1

        elif not (queue2==[]):#use next level queue

            curr\_q = queue2 #use queue2

            next\_q = queue3 #reference the lower-level queue

            p = curr\_q[0]   #get process from queue2

            tq = tq2        #use the tq in queue2's RR algorithm

            pr = 3          #the next priority below queue2

        elif not (queue3==[]):

            p = queue3[0]   #there is no priority below 3, so just use this queue & anything leaving queue3 goes to io

        else:   #if all queues empty

            io\_q[0].pr = 1              #priority of the ready queue

            ready.append(io\_q.pop(0))   #place the soonest ready process in queue1

            continue                    #and run

        if (p.pr == 1 or p.pr == 2):    #we use RR for these two queues

            if(regular):

                print(f"{p.name}\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

                print(f"Ready Time:\t{p.rdy}\nCurrent Time:\t{time}\nJob Length:\t{p.job}")

            if(p.rdy > time):   #if NO processes are ready (i,e: in IO burst)

                if(extra):

                    print(f"{p.name} is NOT ready yet!!!")

                waste += p.rdy-time     #record the time NOT used for any process bursts

                if(extra):

                    print(f"WASTE UPDATE ALERT: {waste} ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

                time = p.rdy        #skip to when process IS ready

            p.tw += time-p.rdy      #current time - the time p's been ready = the time p's been waiting

            if(extra):

                print(f"{p.name}'s tw = {p.tw} = {time} - {p.rdy}!!!@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")

            if (p.tr < 0):      #if NO response time has been assigned, then this is the first cpu burst

                p.tr = time     #record the response time

            if (p.cpu[0]>tq):   #if process does NOT finish in time

                if(extra):

                    print(f"{time} to {round(time+tq,2)} = {p.name} cpu")

                time += tq      #this is the max time we add when executing this run

            else:               #otherwise

                if(extra):

                    print(f"{time} to {round(time+p.cpu[0],2)} = {p.name} cpu")

                time += p.cpu[0]#add the time of the process

            p.cpu[0] -= tq      #regardless, remove the time that the process used

            p.job = p.cpu[0]    #definition of job: next avaialable cpu

            if (not(io\_q == [])):   #if there are processes in io, none ready, add the shortest one who is ready to ready q

                i=0

                while(i < len(io\_q)):

                    q = io\_q[i]

                    if (q.rdy <= time): #only add the ready processes from IO into queue1

                        ready.append(io\_q.pop(0))

                        i-=1

                        q.pr = 1        #priority resets upon leaving IO

                    i+=1

            if (p.cpu[0] <= 0):     #if the process CPU burst has ended

                p.cpu.pop(0)        #remove it

                if (p.ended()):     #if process ended

                    if(regular):

                        print(f"{p.name} has ended.")

                    p.ttr = time    #record the turnaround time

                    curr\_q.pop(0)    #remove it from the queue

                else:#in IO

                    p.job = p.cpu[0]    #new job length

                    if(extra):

                        print(f"{time} to {round(time+p.io[0],2)} = {p.name} io")

                    p.rdy = round(time+p.io.pop(0),2)   #otherwise, run io burst & record when process will be ready

                    if(extra):

                        print(f"{p.name} in io queue, next cpu burst: {p.cpu[0]}")

                    p.pr = 0            #priority to identify processes in io

                    insort(io\_q,p,'rdy')    #add process to the io queue because it is in io.

                    curr\_q.pop(0)           #remove from the current queue

            else:#cpu burst not finished

                p.rdy = time

                p.job = p.cpu[0]    #new job length

                if(extra):

                    print(f"{p.name}'s CPU' did NOT end in {tq} units. pr = {pr}")

                p.pr = pr

                next\_q.append(curr\_q.pop(0))        #remove process from the ready queue

        elif (p.pr == 3):

                if(regular):

                    print(f"{p.name}\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

                    print(f"Ready Time:\t{p.rdy}\nCurrent Time:\t{time}")

                if(p.rdy > time):   #if NO processes are ready (i,e: in IO burst)

                    if(extra):

                        print(f"{p.name} is NOT ready yet!!!")

                    waste += p.rdy-time     #record the time NOT used for any process bursts

                    time = p.rdy            #skip to when process IS ready

                p.tw += time-p.rdy  #current time - the time p's been ready = the time p's been waiting

                if(extra):

                    print(f"{time} to {round(time+p.cpu[0],2)} = {p.name} cpu")

                if (p.tr < 0):     #if NO response time has been assigned, then this is the first cpu burst

                    p.tr = time     #record the response time

                time += p.cpu.pop(0)#run cpu burst, add it to the time

                if (p.ended()):     #if process ended

                    if(regular):

                        print(f"{p.name} has ended.")

                    p.ttr = time    #record the turnaround time

                    queue3.pop(0)   #remove it from the queue

                    continue        #and continue to next process

                p.job = p.cpu[0]

                if(extra):

                    print(f"{time} to {round(time+p.io[0],2)} = {p.name} io")

                p.rdy = round(time+p.io.pop(0),2)   #otherwise, run io burst & record when process will be ready

                if(extra):

                    print(f"{p.name} in io queue, next cpu burst: {p.cpu[0]}")

                p.pr = 0

                insort(io\_q,p,'rdy')    #add process to the io queue because it is in io.

                queue3.pop(0)           #remove process from the ready queue

        if(regular):

            pdata(ready,queue2,queue3,io\_q)

    display(group,time,waste)

#main

button = '?'

while not (button == '0'):

    group = initializer()

    button = input("Please Select ↓\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n1-FCFS\n2-SJF\n3-MLFQ\n0-Exit\nYour Choice: ")

    if (button == '1' or button == '2' or button == '3'):

        view = input("2Scheduling Information Display ↓\t\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\nRegular: For grading purposes\nExtra: For debugging/additional features\nMinimal: Show as little as possible\nType Your Choice: ")

        if (button=='1'):

            FCFS(group,view)

        elif(button=='2'):

            SJF(group,view)

        elif(button=='3'):

            MLFQ(group,view)

    else:

        break