```
# Nathan Robertus
# CSCI 460
# LAB 1
# 9/16/15

#a
Student ID : 2043115
ID % 3 + 2 : 3

#b

// BEGIN PYTHON CODE

from random import randint
import math
import time
start_date = time.strftime("%m_%d_%Y")
start_time = time.strftime("%H_%M_%S")

class processor:

    def __init__(self, cores):
        #Globals for jobs and cores
        self.jobs = []
        self.cores = []
        self.num_cores = cores

        #Core initialization
        for x in range(0,cores):
            c = core()
            self.cores.append(c)

    #Function to manage cores and jobs
    def proc_manager(self, rand):
        self.jobs_copy = []
        tripped = False
        if not rand:
            f = open("jobs.txt", 'r')
            jobList = f.read()
            jobList = jobList.translate(None, '\n').translate(None, ' ').split(';')
            for job in jobList:
                job = job.split(',')
                if(len(job) == 3):
                    self.jobs.append(makeJob(int(job[0]), int(job[1]), int(job[2])))
                    self.jobs_copy.append(makeJob(int(job[0]), int(job[1]),
int(job[2])))
        else:
            for i in range(0, 1000):
                self.jobs.append(makeJob(i+1, i+1, randint(0, 500)))

        #initialize the ticker
        self.tick = 0
        #set the next core counter to mod out to 0
        nextCore = self.num_cores - 1
        #Setup a queue of queues for cores to look at
        queues = []
        cores_busy=[]
        for x in range(0, self.num_cores):
            queue = []
            queues.append(queue)
            cores_busy.append(True)
        self.jobs_count = 0
        while(True):
            #increment the ticker
            self.tick += 1

            #manage jobs and queues
            for index, job in enumerate(self.jobs):
                job.arrival = job.arrival - 1
                if(job.arrival == 0):
                    nextCore = (nextCore+1)%self.num_cores
                    queues[nextCore].append(job)
                    self.jobs.pop(index)
                    self.jobs_count += 1


            #Manage core usage
            for index, core in enumerate(self.cores):
                #print "core: " + str(index)
                busy = core.tick_job()
                cores_busy[index] = busy

                if not busy:
                    if(queues[index]):
                        core.get_job(queues[index][0])
                        queues[index].pop(0)

            #Check for a break case
            if len(self.jobs) == 0:

                num_queues = len(queues)
                emtpy_queues = []
                idle_cores = []

                for queue in queues:
                    if not queue:
                        emtpy_queues.append("empty")

                if(num_queues == len(emtpy_queues)):
                    for x in cores_busy:
                        if x == False:
                            idle_cores.append("idle")
                    if(num_queues == len(idle_cores)):
                        if not tripped:
                            tripped = True
                        else:
                            break
        return self.tick

class core:
    def __init__(self):
        self.currentJobTime = 0

    def get_job(self, job):
        self.currentJobTime = job.time
    def tick_job(self):
        if self.currentJobTime:
            self.currentJobTime = self.currentJobTime - 1
            #print "time left: " + str(self.currentJobTime)
            if (self.currentJobTime == 0):
                return False
            else:
                return True
        else:
            return False

class job(object):
    id = 0
    arrival = 0
    time = 0

    def __init__(self, id, arrival, time):
        self.id = id
        self.arrival = arrival
        self.time = time

def makeJob(id, arrival, time):
    Job = job(id, arrival, time)
    return Job

#Main function
def main(user_input, random_bool, trials, core_count):
    def average(s): return sum(s) * 1.0 / len(s)
    trial_results = []
    filename = "output/round_robin/" + str(start_date) + "_" + str(start_time) +
".txt"
    f = open(filename, "w")
    if(user_input):
        core_count = int(raw_input("Enter number of cores: "))
        user_rand = raw_input("Use random input? (Y/N) ")
        if(user_rand == 'Y'):
            random_bool = True
        elif(user_rand == 'N'):
            random_bool = False
        trials = int(raw_input("Enter number of trials: "))

    #initialize the processor with the given number of cores
    x = processor(core_count)

    #print a header to the output file
    f.write("Date: " + start_date.replace("_", "/") + "\n")
    f.write("Time: " + start_time.replace("_", ":") + "\n")
    f.write("Cores: " + str(core_count) + "\n")
    f.write("Random data: " + str(random_bool) + "\n")
    f.write("# of trials: " + str(trials) + "\n\n")
    f.write("======================================\n\n")

    #Run the given number of trials and print the output to the file and the console
    for z in range(0, trials):
        current = x.proc_manager(random_bool)
        trial_results.append(current)
        f.write(str(current) + " ms\n")
        print str(current) + " ms"

    #calculate stats on all the trials
    minimum = min(trial_results)
    maximum = max(trial_results)
    avg = average(trial_results)
    variance = map(lambda x: (x - avg)**2, trial_results)
    std_dev = math.sqrt(average(variance))

    #print the statistics in a footer on the output file and close the file writer
    f.write("\n======================================\n\n")
    f.write("Average: " + str(avg) + " ms\n")
    f.write("Minimum: " + str(minimum) + " ms\n")
    f.write("Maximum: " + str(maximum) + " ms\n")
    f.write("Standard deviation: " + str(std_dev) + " ms")
    f.close()

    #Print the final summary to the console
    print "======================================"
    print "Average: " + str(avg) + " ms"
    print "Minimum: " + str(minimum) + " ms"
    print "Maximum: " + str(maximum) + " ms"
    print "Standard deviation: " + str(std_dev) + " ms"

#Call the main function with default values to be overwritten by user input
main(True, False, 100, 3)

// END PYTHON CODE

#b.1

// BEGIN PROGRAM OUTPUT

Date: 09/02/2015
Time: 08:28:40
Cores: 3
Random data: True
# of trials: 100

======================================

87935 ms
81910 ms
85300 ms
88203 ms
89269 ms
84097 ms
85620 ms
84580 ms
82377 ms
84502 ms
82260 ms
86023 ms
87888 ms
84577 ms
87686 ms
84356 ms
85887 ms
83483 ms
90739 ms
84958 ms
87142 ms
86196 ms
85729 ms
86750 ms
88154 ms
89921 ms
83553 ms
```

```
84125 ms
83139 ms
82732 ms
86848 ms
85414 ms
85624 ms
85221 ms
86143 ms
86255 ms
84477 ms
83882 ms
84224 ms
85875 ms
85635 ms
85770 ms
85061 ms
85830 ms
83855 ms
85650 ms
85281 ms
85371 ms
85109 ms
82478 ms
84434 ms
86305 ms
85658 ms
85108 ms
86679 ms
88067 ms
85245 ms
83577 ms
84401 ms
89462 ms
86836 ms
88240 ms
87193 ms
84160 ms
84456 ms
88383 ms
86720 ms
85354 ms
87961 ms
85689 ms
85644 ms
83620 ms
87207 ms
86364 ms
84500 ms
```

```
83976 ms
86754 ms
84088 ms
88552 ms
83307 ms
84459 ms
83535 ms
85065 ms
83215 ms
80977 ms
83621 ms
86057 ms
87433 ms
84725 ms
83648 ms
86577 ms
85988 ms
88821 ms
81200 ms
84618 ms
83990 ms
87927 ms
88253 ms
87949 ms
83837 ms

======================================

Average: 85529.29 ms
Minimum: 80977 ms
Maximum: 90739 ms
Standard deviation: 1920.32923893 ms

// END PROGRAM OUTPUT
```

#b.2

```
// BEGIN PROGRAM OUTPUT

Date: 09/02/2015
Time: 09:46:03
Cores: 3
Random data: False
# of trials: 100

======================================

421 ms
```

```
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
```

```
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
421 ms
```

```
421 ms
421 ms
421 ms

======================================

Average: 421.0 ms
Minimum: 421 ms
Maximum: 421 ms
Standard deviation: 0.0 ms

// END PROGRAM OUTPUT

#c

// BEGIN PYTHON CODE

from random import randint
import math
import time
start_date = time.strftime("%m_%d_%Y")
start_time = time.strftime("%H_%M_%S")

class processor:

    def __init__(self, cores):
        #Globals for jobs and cores
        self.jobs = []
        self.cores = []
        self.num_cores = cores

        #Core initialization
        for x in range(0,cores):
            c = core()
            self.cores.append(c)

    #Function to manage cores and jobs
    def proc_manager(self, rand):
        self.jobs_copy = []
        tripped = False
        if not rand:
            f = open("jobs.txt", 'r')
            jobList = f.read()
            jobList = jobList.translate(None, '\n').translate(None, ' ').split(';')
            for job in jobList:
                job = job.split(',')
                if(len(job) == 3):
                    self.jobs.append(makeJob(int(job[0]), int(job[1]), int(job[2])))
                    self.jobs_copy.append(makeJob(int(job[0]), int(job[1]), 
int(job[2])))
        else:
            for i in range(0, 1000):
                self.jobs.append(makeJob(i+1, i+1, randint(0, 500)))

        #initialize the ticker
        self.tick = 0
        #set the next core counter to mod out to 0
        nextCore = self.num_cores - 1
        #Setup a queue of queues for cores to look at
        queues = []
        core_status=[]
        core_total_time = []
        for x in range(0, self.num_cores):
            queue = []
            queues.append(queue)
            core_status.append(0)
            core_total_time.append(0)
        while(True):
            #increment the ticker
            self.tick += 1
            #calculate the total time left for each core (remaining time on current 
job + total time of jobs in queue)
            for index, core in enumerate(self.cores):
                core_total_time[index] = 0
                if(queues[index]):
                    for job in queues[index]:
                        core_total_time[index] += job.time
                core_total_time[index] += core_status[index]
            #manage jobs and queues
            for index, job in enumerate(self.jobs):
                job.arrival = job.arrival - 1
                if(job.arrival == 0):

                    core_index = core_total_time.index(min(core_total_time))
                    queues[core_index].append(job)
                    self.jobs.pop(index)

            #Manage core usage
            for index, core in enumerate(self.cores):
                #print "core: " + str(index)
                busy = core.tick_job()
                core_status[index] = busy

                if busy == 0:
                    if(queues[index]):
                        core.get_job(queues[index][0])
                        queues[index].pop(0)

            #Check for a break case
            if len(self.jobs) == 0:
                num_queues = len(queues)
                emtpy_queues = []
                idle_cores = []

                for queue in queues:
                    if not queue:
                        emtpy_queues.append("empty")

                if(num_queues == len(emtpy_queues)):
                    for x in core_status:
                        if x == 0:
                            idle_cores.append("idle")
                    if(num_queues == len(idle_cores)):
                        if not tripped:
                            tripped = True
                        else:
                            break
        return self.tick

class core:
    def __init__(self):
        self.currentJobTime = 0

    def get_job(self, job):
        self.currentJobTime = job.time
    def tick_job(self):
        if self.currentJobTime:
            self.currentJobTime = self.currentJobTime - 1
            #print "time left: " + str(self.currentJobTime)
            return self.currentJobTime
        else:
            return 0

class job(object):
    id = 0
    arrival = 0
    time = 0

    def __init__(self, id, arrival, time):
        self.id = id
        self.arrival = arrival
        self.time = time

def makeJob(id, arrival, time):
    Job = job(id, arrival, time)
    return Job

#Main function
def main(user_input, random_bool, trials, core_count):
    def average(s): return sum(s) * 1.0 / len(s)
    trial_results = []
    filename = "output/optimized/"+str(start_date) + "_" + str(start_time) +
"_OPT.txt"
    f = open(filename, "w")
    if(user_input):
        core_count = int(raw_input("Enter number of cores: "))
        user_rand = raw_input("Use random input? (Y/N) ")
        if(user_rand == 'Y'):
            random_bool = True
        elif(user_rand == 'N'):
            random_bool = False
        trials = int(raw_input("Enter number of trials: "))

    #initialize the processor with the given number of cores
    x = processor(core_count)

    #print a header to the output file
    f.write("Date: " + start_date.replace("_", "/") + "\n")
    f.write("Time: " + start_time.replace("_", ":") + "\n")
    f.write("Cores: " + str(core_count) + "\n")
    f.write("Random data: " + str(random_bool) + "\n")
    f.write("# of trials: " + str(trials) + "\n\n")
    f.write("======================================\n\n")

    #Run the given number of trials and print the output to the file and the console
    for z in range(0, trials):
        current = x.proc_manager(random_bool)
        trial_results.append(current)
        f.write(str(current) + " ms\n")
        print str(current) + " ms"

    #calculate stats on all the trials
    minimum = min(trial_results)
    maximum = max(trial_results)
    avg = average(trial_results)
    variance = map(lambda x: (x - avg)**2, trial_results)
    std_dev = math.sqrt(average(variance))

    #print the statistics in a footer on the output file and close the file writer
    f.write("\n======================================\n\n")
    f.write("Average: " + str(avg) + " ms\n")
    f.write("Minimum: " + str(minimum) + " ms\n")
    f.write("Maximum: " + str(maximum) + " ms\n")
    f.write("Standard deviation: " + str(std_dev) + " ms")
    f.close()

    #Print the final summary to the console
    print "======================================"
    print "Average: " + str(avg) + " ms"
    print "Minimum: " + str(minimum) + " ms"
    print "Maximum: " + str(maximum) + " ms"
    print "Standard deviation: " + str(std_dev) + " ms"

#Call the main function with default values to be overwritten by user input
main(True, False, 100, 3)

// END PYTHON CODE

// BEGIN PROGRAM OUTPUT

Date: 09/02/2015
Time: 09:26:48
Cores: 3
Random data: True
# of trials: 100

======================================

86768 ms
82874 ms
82607 ms
82356 ms
82031 ms
80236 ms
79721 ms
83563 ms
83896 ms
84059 ms
80483 ms
83428 ms
85310 ms
83826 ms
83101 ms
81816 ms
86584 ms
85602 ms
82005 ms
82486 ms
83759 ms
82945 ms
```

```
83591 ms
82054 ms
84258 ms
83579 ms
82349 ms
80837 ms
83873 ms
82962 ms
82234 ms
83393 ms
81125 ms
84289 ms
82149 ms
80488 ms
85294 ms
81700 ms
83351 ms
84109 ms
78086 ms
84599 ms
82393 ms
82481 ms
83928 ms
82053 ms
83504 ms
84168 ms
84190 ms
81666 ms
80753 ms
82983 ms
83740 ms
82649 ms
85514 ms
83788 ms
84812 ms
82290 ms
86333 ms
84122 ms
82912 ms
84397 ms
83011 ms
82116 ms
83336 ms
83031 ms
82550 ms
83990 ms
86891 ms
84904 ms
```

```
80719 ms
84447 ms
81244 ms
81394 ms
82423 ms
83106 ms
84456 ms
83860 ms
83014 ms
85519 ms
83015 ms
82415 ms
83624 ms
81873 ms
85441 ms
84561 ms
83957 ms
83321 ms
79549 ms
81965 ms
84547 ms
86184 ms
85008 ms
82984 ms
83678 ms
83027 ms
86815 ms
85277 ms
85335 ms
83124 ms

======================================

Average: 83281.63 ms
Minimum: 78086 ms
Maximum: 86891 ms
Standard deviation: 1639.43776128 ms

// END PROGRAM OUTPUT

// BEGIN PROGRAM OUTPUT

Date: 09/02/2015
Time: 09:26:43
Cores: 3
Random data: False
# of trials: 100
```

```
=======================================

395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
```

```
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms
```

```
395 ms
395 ms
395 ms
395 ms
395 ms
395 ms

======================================

Average: 395.0 ms
Minimum: 395 ms
Maximum: 395 ms
Standard deviation: 0.0 ms

// END PROGRAM OUTPUT
```