Problem 1 Report

Implementing the various CPU scheduling algorithms

Guillaume Comfort - IT210

- Initial Comments

This is a project report on problem 1 out of 5 to be chosen for the final group project for IT210. This problem specifically involves implementing the various CPU scheduling algorithms: First come first serve, Shortest Job Next, Round Robin, and Priority Scheduling. These algorithms and this report was written by Guillaume Comfort, the former written in Python. For all algorithms the control data set was sched.data = [9, 8, 6, 2, 3, 4, 5, 7, 10, 1]. For priority scheduling the Priority was Prio = [4, 7, 3, 8, 10, 1, 2, 9, 5, 6] where sched.data[i] ≡ Prio[i] and higher number = higher priority (i.e 10 > 1 in priority).

Many of the steps in these algorithms are very similar to each other, hence after explaining the core loops in First Come First Serve, I will only touch very briefly on them afterwards, and explain only the differences in the programs in the following.

---

1. First Come First Serve

The first come first serve algorithm involves processing the first process that comes up until it is finished, and then continuing to the second process that was in queue and continuing like that until we finish the n-1th process.

The first step involved getting the number of processes and the service time for each process. For this and all algorithms I defined a function to get the user input on the number of processes, hereby known as sched.len which serves as the length of the array, and then getting the time for each process in the form of an integer, which will be appended to the rear of the array, until rearranged.

```
def DATA(sched):
    sched.data = []
    sched.len = int(input("Enter the number of processes: ")) # Allows user to input a number for processes
    for i in range(int(sched.len)):
        T = int(input(f"Enter a time for Process {i+1}: ")) # Allows user to set burst time for processes
        sched.data.append(T)
```

The next step was initialising the rest of the data, and setting the waiting time initially to 0

```
def FirstCome(sched):
    print("\n")
    Schd = list(sched.data)  # Step 1&2&3&4: Create number of processes, with ID & service time for each process and their processing times.
    ServiceTime = 0.00
    Totalwait = 0.00 # Step 3: Waiting Time of first process is zero
    TAT = 0.00
    AvgTAT = 0.00
```

The next few steps were to get the data, which included calculating the waiting time, turnaround time, and the averages of those times, in this snippet of code below, I used a for loop to do this for all values in the given data set, and then a while loop to print out the number, subtract it by 1 and increase the service time (equivalent to waiting time) by 1, until the number reached zero. This was done so I can see the 'process' as it was being completed and can easily be cut out to save time for large processes.

```
    for i in range(int(sched.len)): # For all values in Scheddata, starting at index 0 in FCFS
        num = int(Schd[i]) # The index value is the process to be completed, all rest are after it in FCFS
        print(f"Starting process {i+1}.")
        while num >= 1: # Until the index value is reduced to 0. Yields expected order with control data.
            print(num)
            num -= 1
            ServiceTime += 1
```

After a process was finished, its service time was printed out to show how long it took for the process to finish without anything else to it, then its service time was added to the Total Waiting Time, and that was printed as well, and this ensured that the Total Waiting Time of one process was the Total time of the previous process, + the time it took to finish that process. The turnaround time was then calculated by adding the total waiting time - the service time to the existing turnaround time. Finally, ServiceTime was reset back to 0 to allow the next process to continue without the waiting time being skewed.

```
        print(f"Process finished, service time required to finish process was: {ServiceTime}") # Step 11: Displaying Waiting time of one process
        Totalwait += ServiceTime  # Step 4&5&6&7: Calculating Total time&Totalwait of one process by adding the waiting time of the previously c
        print(f"The total time waiting to finish process was: {Totalwait}.\n") # Step 11: Displaying the Waiting time of one process, in relatio
        TAT = TAT + (Totalwait - ServiceTime) # Step 8: Turnaround time is calculated by all the total times of each process.
        ServiceTime = 0.00 # Resetting Waiting Time for next process.
```

The final few steps were to calculate the average waiting time and the average turnaround time, and then print out all the steps to show the results. The average wait time was merely the total waiting time divided by the number of processes, which was 10, and the turnaround time the same thing, but with the turnaround time.

```
    Avgwait = (Totalwait / sched.len) # Step 9: Calculating Average wait time by dividing total time by the amount of processes.
    AvgTAT = (TAT / sched.len) # Step 10: Calculating Average Turnaround time by dividing the Turnaround time by the amount of processes.

    print(f"\nProcesses Finished. Total Waiting time was: {Totalwait}") # Step 11: Displaying Total waiting time of all processes.
    print(f"Average Wait Time: {Avgwait}") # Step 11: Displaying the Average time of all processes.
    print(f"Turnaround time: {TAT}") # Step 11: Displaying the Turnaround time of all processes.
    print(f"Average turnaround time: {AvgTAT}") # Step 11: Displaying the Average turnaround time of all processes.
```

The final results for this algorithm came out to be:

```
Processes Finished. Total Waiting time was: 55.0
Average Wait Time: 5.5
Turnaround time: 269.0
Average turnaround time: 26.9
```

---

2. Shortest Job Next

Shortest job next involves processing the shortest process in a queue, as opposed to in the order they come in, continuing with it until it finishes, and doing that for all processes in a queue.

The steps involved were almost exactly the same as First Come First Serve, with the exception of needing to sort the processes from smallest to largest, such that the smallest would come first.

These loops do just that as in the first loop, it sorts the array backwards, then in this newer order, starting at the 1st element for the length of the array, while an element in the array is smaller than the next element, it will move that element until it can't no more, for all elements in the array, this snippet was thus proven to be able to sort all elements in any given data set from shortest to longest.

```
# Will go throughout the entire list scheddata and if the process is smaller than another one, they will swap places, until the shortest process
    for i in range(int(sched.len), 0, -1):
        for j in range(1, int(sched.len)):
                while int(Schd[j - 1]) > int(Schd[j]):
                    Schd[j - 1], Schd[j] = Schd[j], Schd[j - 1]
                    #Yields expected order of [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] with Control data. Assumed for all cases.
```

For the remaining steps, it was completely identical to First Come First Serve, and the final results for this algorithm came out to be:

```
Processes Finished. Total Waiting time was: 55.0.
Average Wait Time: 5.5.
Turnaround time: 165.0.
Average turnaround time: 16.5.
```

---

3. Round Robin

Round Robin involves processing given processes using time slices (time quantums) which is an assigned slot that each process is allowed to run for before it is cut off and the next process is given the ability to run, until all processes are then finished. For this, I used a Time Quantum of 5.

This was vastly different from the rest of the scheduling algorithms in how I went about doing it.

```
while(1):
    done = True
    for i in range(int(sched.len)):
        if int(Schd[i]) > 0:
            done = False
            if int(Schd[i]) > quantum: #Else:
                Time += quantum
                Schd[i] = int(Schd[i]) - quantum
            else: # Step 3: If the time quantum is greater than the burst time.
                Time = Time + int(Schd[i])
                WaitingTime.append(Time - int(Schd2[i]))
                Schd[i] = 0

    if done == True:
        break


for i in range(int(sched.len)):
    T = int(Schd2[i]) + int(WaitingTime[i])
    TAT.append(T)
    TTAT = TTAT + TAT[i]
```

Using two separate loops, in the first loop was a “while true” loop with nested loops inside. The first nested loop would run for all values in sched.data. If an element was above 0, then the while loop would be unable to finish. Then there were 2 separate calculations involved, the first was that if an element were greater than the time quantum, the Time would be increased by the time quantum, and the element would be subtracted by the quantum. The 2nd calculation was that if the quantum was greater than the element, the time was increased by the value of the element, and then appended to the Total waiting time, the element was subsequently set to 0 and finished. When all elements were finished, the while loop would then break.

The 2nd loop was for the turnaround time and used all elements in waiting time and added them to elements in the sched.data list, then it was all appended to the turnaround time, and then each element added to the total turnaround time.

Printing the results was exactly the same as all the rest and the final results came out to be:

```
The total waiting Time for all processes was: 55.
The average waiting time was: 5.5.
The total turnaround time was: 285
The average turnaround time was: 28.5
```

---

4. Priority Scheduling

Priority Scheduling involves processing given processes with a priority attached to each process, such that each process is ran in the order of the priority, as opposed to in the order they appear or the length of each process.

The steps involved were almost exactly the same as the shortest job next, with the exception that instead of ordering the processes from shortest to longest, now ordering from highest priority to lowest.

To get the priority for each process, I used a for loop to run for each element in sched.data, which allowed the user to input an integer priority and then appended them to an array I called Prio. This allowed each priority to be able to be associated with a corresponding value in sched.data by the position so that d[0] ≡ p[0], d[1] ≡ p[1], and so on. The values and their priority were then printed side by side.

```
for i in range(int(sched.len)): # Step 1&2: Getting the priority of processes from user input.
    T = int(input(f"Enter priority for process {i+1}: "))
    Prio.append(T)
   # if priomax < int(T):
    #    priomax = int(T)
```

Next I used a modified version of a loop from shortest job next, modified such that it would rearrange both the priority and processes in order of the priority so that the highest priority made its corresponding process the first to be done, and also added two separate time counters for all other processes other than the first to be done.

```
for i in range(int(sched.len)):
    for j in range(1, int(len(Prio))):
        while int(Prio[j - 1]) < int(Prio[j]):  # Arranging processes in order of priority, until highest priorities and it's elements are at
        #if you want lower number to be higher priority, simply swap < for >
            Prio[j-1], Prio[j] = Prio[j], Prio[j-1]
            Schd[j-1], Schd[j] = Schd[j], Schd[j-1]
            # Yields expected order of [9, 3, 6, 7, 5] -> [9, 6, 4, 3, 1] with Control data. Assumed for all cases.
            pseudowait += 1 #Step 4: all other elements total wait will be pseudototal and wait time pseudowait.
            pseudototal += 1 #Step 5: TWT is calculated by adding the waiting time for lack processes.
            # This will not change the turnaround time, it only affects TWT and AvgWT
```

Using the control data and control priority, I was able to get a successful sorting so that the highest priority and its element was the first element to be completed.

```
Processes: [9, 8, 6, 2, 3, 4, 5, 7, 10, 1]
Priority: [4, 7, 3, 8, 10, 1, 2, 9, 5, 6]

Sorted processes: [3, 7, 2, 8, 1, 10, 9, 6, 5, 4]
Sorted priority: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

With these differences out of the way, the rest of the algorithm was done the exact same way with the exception of using the pseudowait and pseudototal values for all values other than p[0], resetting the service time back to the pseudowait, etc.

```
    ServiceTime += 1
    pseudototal += 1
print(f"Process finished, service time required to finish process was: {ServiceTime}.") #
print(f"The total time waiting to finish process was: {Totalwait}.\n") # Step 9: Displayi
TAT = TAT + (Totalwait - ServiceTime) #Step 5: Turnaround is calculated by adding all tot
ServiceTime = pseudowait # Step 4: all other elements total wait will be pseudototal and
# This yields expected wait time for each element individually, taking into account i[0]
Totalwait = pseudototal # Step 4&5: all other elements total wait will be pseudototal and
# This yields expected Total wait time for all elements
```

The number of swaps did not affect the turnaround time at all, and only affected the waiting time. At the end I was left with these final results:

```
Processes Finished. Total Waiting time was: 79.0.
Average Wait Time: 7.9.
Turnaround time: 234.0.
Average turnaround time: 23.4.
```

- Conclusion:

To conclude this report, by the data I collected, the turnaround time indicated which algorithm was the fastest overall at completing all the processes.

The algorithm with the lowest turnaround time was Shortest Job Next with a turnaround time of 165.0 making it the fastest algorithm.

The algorithm with the highest turnaround time was Round Robin, with a turnaround time of 285.0.

Doing further tests on Priority Scheduling revealed that the turnaround time can change drastically depending on how the processes are organised. Processes that are organised in such a way that the processes with the shortest time to complete were the highest priority scored a lower turnaround time, in line with Shortest Job Next while processes organised with the longest time to complete were highest priority were worse than Round Robin.

Best case of Priority Scheduling organised by shortest to longest with 0 swaps.
Data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and Priority = [10, 9, 8, 7, 6, 5 , 4, 3, 2, 1]

```
Processes Finished. Total Waiting time was: 55.0.
Average Wait Time: 5.5.
Turnaround time: 165.0.
Average turnaround time: 16.5.
```

Worst case of Priority Scheduling organised by longest to shortest, with all elements needing to be swapped.
Data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and Priority = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
Processes Finished. Total Waiting time was: 100.0.
Average Wait Time: 10.0.
Turnaround time: 330.0.
Average turnaround time: 33.0.
```