Andrew Ma

Lab 7

CPE 435

3/1/21

Theory

Round Robin Scheduling

Round robin scheduling uses time slices (quantum time unit). Each process can run up to the quantum time unit amount before switching to the next process. If the process finishes before the quantum time unit amount, then it switches to the next process. Round robin schedules processes fairly with time sharing, and it is a preemptive algorithm since it interrupts processes out of the CPU once it reaches the quantum time unit.

Priority Based Scheduling

Priority Based Scheduling schedules processes based on a priority number assigned to each process. The key part is choosing priorities for the processes. If we assume the processes are non-preemptive then before we run we choose the highest priority process (usually the lowest priority number). Then after it finishes we choose the next highest priority process. For processes that have equal priority numbers, then it chooses based on first come first serve. If we do preemptive priority based scheduling, then when new processes are created with a higher priority it will interrupt the current running process.

Preemptive vs Non Preemptive Scheduling

Preemptive scheduling interrupts the current process when a certain condition is reached. Non preemptive scheduling will not interrupt the current process, and will run the current process to completion. Round robin scheduling interrupts as soon as the condition (quantum time unit is elapsed) is met. Priority Based Scheduling can be preemptive where it will interrupt the current process when a new process with a higher priority comes in. But, in this lab we only implement the non-preemptive version

Observations

Round Robin Scheduling Output

3 processes
15 quantum time
PID 1: 60 burst time

PID 2: 32 burst time

PID 3: 25 burst time

```
x@x:~/Desktop/lab07$ python3 lab7_roundrobin.py 3 15 1:60,2:32,3:25 -h
usage: lab7_roundrobin.py [-h] num_processes quantum_time pid_to_burst_time_dict

positional arguments:
  num_processes         Number of processes
  quantum_time          Quantum Time Unit is the unit of CPU time each process gets before moving to next process
  pid_to_burst_time_dict
                        comma-separated PID:burst time, e.g. 1:10,2:20,3:10
```

```
x@x:~/Desktop/lab07$ python3 lab7_roundrobin.py 3 15 1:60,2:32,3:25
Namespace(num_processes=3, pid_to_burst_time_dict={1: 60, 2: 32, 3: 25}, quantum_time=15)
Process [1]: burst 60, worked 15, turn around 15
Process [2]: burst 32, worked 15, turn around 30
Process [3]: burst 25, worked 15, turn around 45
Process [1]: burst 60, worked 30, turn around 60
Process [2]: burst 32, worked 30, turn around 75
Process [3]: burst 25, worked 25, turn around 85
Process [1]: burst 60, worked 45, turn around 100
Process [2]: burst 32, worked 32, turn around 102
Process [1]: burst 60, worked 60, turn around 117
+------+-------+-------+-------+-------+-------+-------+-------+------+-------+
| Time | 15 ms | 15 ms | 15 ms | 15 ms | 15 ms | 10 ms | 15 ms | 2 ms | 15 ms |
| PID  |   1   |   2   |   3   |   1   |   2   |   3   |   1   |  2   |   1   |
+------+-------+-------+-------+-------+-------+-------+-------+------+-------+
+-----+----------------------+----------------+
| PID | Turn Around Time (ms) | Wait Time (ms) |
+-----+----------------------+----------------+
|  3  |          85          |       60       |
|  2  |         102          |       70       |
|  1  |         117          |       57       |
+-----+----------------------+----------------+
Average Wait Time: 62.33 ms
```

Priority Based Scheduling Output

3 processes

PID 1: priority 1 (highest priority), burst time 60

PID 2: priority 2, burst time 32

PID 3: priority 3 (lowest priority), burst time 25

```
x@x:~/Desktop/lab07$ python3 lab7_priority.py 3 1:1_60,2:2_32,3:3_25 -h
usage: lab7_priority.py [-h] num_processes pid_to_priority_burst_time_dict

positional arguments:
  num_processes         Number of processes
  pid_to_priority_burst_time_dict
                        comma-separated PID:burst time, e.g. 1:1_10,2:2_20,3:3_10
```

```
x@x:~/Desktop/lab07$ python3 lab7_priority.py 3 1:1_60,2:2_32,3:3_25
Namespace(num_processes=3, pid_to_priority_burst_time_dict={1: {'priority': 1, 'burst_time': 60}, 2: {'priority': 2, 'burst_time': 32}, 3: {'pri
ority': 3, 'burst_time': 25}})
Process [1]: priority 1 burst 60, worked 60, turn around 60
Process [2]: priority 2 burst 32, worked 32, turn around 92
Process [3]: priority 3 burst 25, worked 25, turn around 117
+------+-------+-------+-------+
| Time | 60 ms | 32 ms | 25 ms |
| PID  |   1   |   2   |   3   |
+------+-------+-------+-------+
+-----+----------------------+---------------+
| PID | Turn Around Time (ms) | Wait Time (ms) |
+-----+----------------------+---------------+
|  1  |          60          |       0       |
|  2  |          92          |       60      |
|  3  |          117         |       92      |
+-----+----------------------+---------------+
Average Wait Time: 50.67 ms
```

3 processes
PID 1: priority 3 (lowest priority), burst time 60
PID 2: priority 2, burst time 32
PID 3: priority 1 (highest priority), burst time 25

```
x@x:~/Desktop/lab07$ python3 lab7_priority.py 3 1:3_60,2:2_32,3:1_25
Namespace(num_processes=3, pid_to_priority_burst_time_dict={1: {'priority': 3, 'burst_time': 60}, 2: {'priority': 2, 'burst_time': 32}, 3: {'pri
ority': 1, 'burst_time': 25}})
Process [3]: priority 1 burst 25, worked 25, turn around 25
Process [2]: priority 2 burst 32, worked 32, turn around 57
Process [1]: priority 3 burst 60, worked 60, turn around 117
+------+-------+-------+-------+
| Time | 25 ms | 32 ms | 60 ms |
| PID  |   3   |   2   |   1   |
+------+-------+-------+-------+
+-----+----------------------+---------------+
| PID | Turn Around Time (ms) | Wait Time (ms) |
+-----+----------------------+---------------+
|  3  |          25          |       0       |
|  2  |          57          |       25      |
|  1  |          117         |       57      |
+-----+----------------------+---------------+
Average Wait Time: 27.33 ms
```

Conclusion

Yes the program worked as expected. From this lab I learned how round robin scheduling uses quantum time units for fair and preemptive scheduling. I also learned how priority based scheduling uses priority numbers for scheduling, and can be either preemptive or non-preemptive. I also learned how adjusting the burst time and priorities can affect average wait time. In my output results the priority based scheduling with reversed priority numbers had the least average wait time of 27.33 ms. The priority based with priority numbers associated with PID had 50.67 ms average wait time. The round robin scheduling had the highest average wait time of 62.33 ms.

Source Code

```
# Round Robin

import argparse
from tabulate import tabulate

class Process:
    def __init__(self, pid, burst_time) -> None:
        self.pid = pid                  # Process ID
        self.burst_time = burst_time    # needed CPU time to complete
        self.worked_time = 0           # time executed so far,  When
working time == burst time, process is complete, initially start at 0
        self.t_round = 0                # turn around time, time taken
for process to fully complete = wait time + burst time

    def __str__(self):
        return f"Process [{self.pid}]: burst {self.burst_time}, worked
{self.worked_time}, turn around {self.t_round}"

    def copy(self):
        process_copy =  Process(self.pid, self.burst_time)
        process_copy.worked_time = self.worked_time
        process_copy.t_round = self.t_round
        return process_copy

def positive_int(value):
    try:
        value = int(value)
        if value > 0:
            return value
        else:
            raise argparse.ArgumentTypeError("{} not a valid positive
integer".format(value))

    except ValueError:
        raise argparse.ArgumentTypeError("{} not a valid positive
integer".format(value))
```

```
def arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("num_processes", type=positive_int, help="Number
of processes")
    parser.add_argument("quantum_time", type=positive_int, help="Quantum
Time Unit is the unit of CPU time each process gets before moving to
next process")
    parser.add_argument("pid_to_burst_time_dict", type=lambda v:
{int(k):int(v) for k,v in (x.split(":") for x in v.split(","))},
help="comma-separated PID:burst time, e.g. 1:10,2:20,3:10")
    args = parser.parse_args()

    if len(args.pid_to_burst_time_dict) != args.num_processes:
        parser.error("`pid_to_burst_time_dict` requires same number of
entries as `num_processes`")

    return args


def main():
    args = arguments()
    print(args)

    # create Process objects from arguments
    processList = [Process(pid, burst_time) for pid,burst_time in
args.pid_to_burst_time_dict.items()]

    # Round Robin
    i = 0

    total_time = 0

    order_table = []
    final_table = []
```

```
    while processList:
        p = processList[i]
        cpu_time_remaining = (p.burst_time - p.worked_time)

        if cpu_time_remaining <= args.quantum_time:
            # cpu time remaining is less than quantum time, so can
finish this turn
            t = cpu_time_remaining
            total_time += t

            p.worked_time += t
            p.t_round = total_time

            final_table.append(p.copy())

            # remove finished process from process list
            processList.pop(i)
            if processList:
                i = i % len(processList)
        else:
            # cpu time remaning is more than quantum time, so can only
work up to quantum time but not finish
            t = args.quantum_time
            total_time += t

            p.worked_time += t
            p.t_round = total_time

            # move to next process
            i = (i+1) % len(processList)

        print(p)
        order_table.append((p.copy(), t))

    ##############################################
```

```
    # Table For Order of Execution
    row1 = ["Time"]
    row2 = ["PID"]
    for p_copy, t in order_table:
        row1.append(f"{t} ms")
        row2.append(f"{p_copy.pid}")

    print(tabulate((row1, row2), tablefmt="pretty"))

    # Table For Turn Around Time and Wait Time
    total_wait_time = sum(p.t_round - p.burst_time for p in final_table)
    print(tabulate(((p.pid, p.t_round, p.t_round - p.burst_time) for p
in final_table), headers=("PID", "Turn Around Time (ms)", "Wait Time
(ms)"), tablefmt="pretty"))
    print(f"Average Wait Time: {total_wait_time /
args.num_processes:.2f} ms")

if __name__ == "__main__":
    main()
```

```
# Priority Based (non preemptive)

import argparse
from tabulate import tabulate

class Process:
    def __init__(self, pid, burst_time, priority) -> None:
        self.pid = pid                      # Process ID
        self.burst_time = burst_time        # needed CPU time to complete
        self.priority = priority
        self.worked_time = 0               # time executed so far,  When
working time == burst time, process is complete, initially start at 0
        self.t_round = 0                    # turn around time, time taken
for process to fully complete = wait time + burst time
```

```
    def __str__(self):
        return f"Process [{self.pid}]: priority {self.priority} burst
{self.burst_time}, worked {self.worked_time}, turn around
{self.t_round}"

    def copy(self):
        process_copy =  Process(self.pid, self.burst_time,
self.priority)
        process_copy.worked_time = self.worked_time
        process_copy.t_round = self.t_round
        return process_copy

def positive_int(value):
    try:
        value = int(value)
        if value > 0:
            return value
        else:
            raise argparse.ArgumentTypeError("{} not a valid positive
integer".format(value))

    except ValueError:
        raise argparse.ArgumentTypeError("{} not a valid positive
integer".format(value))


def arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("num_processes", type=positive_int, help="Number
of processes")
    parser.add_argument("pid_to_priority_burst_time_dict", type=lambda
v: {int(pid): {"priority": int(pri), "burst_time": int(burst_time)} for
pid, pri, burst_time in ([pid] + v.split("_") for pid,v in (x.split(":")
for x in v.split(",")))}, help="comma-separated PID:burst time, e.g.
1:1_10,2:2_20,3:3_10")
    args = parser.parse_args()
```

```
    if len(args.pid_to_priority_burst_time_dict) != args.num_processes:
        parser.error("`pid_to_priority_burst_time_dict` requires same
number of entries as `num_processes`")

    return args


def choose_priority_idx(processList):
    if not processList:
        raise Exception("Empty process list")

    # here highest priority is one with the minimum priority number
    #  if highest priority is one with the maximum priority number, then
change function to max()
    priorityNumber = min(p.priority for p in processList)
    for i in range(len(processList)):
        if priorityNumber == processList[i].priority:
            return i
    else:
        raise Exception("Can't find priority number")


def main():
    args = arguments()
    print(args)

    # create Process objects from arguments
    processList = [Process(pid, infodict["burst_time"],
infodict["priority"]) for pid,infodict in
args.pid_to_priority_burst_time_dict.items()]

    # Priority (non preemptive)
    i = 0

    total_time = 0
```

```
    order_table = []
    final_table = []

    i = choose_priority_idx(processList)

    while processList:
        p = processList[i]
        cpu_time_remaining = (p.burst_time - p.worked_time)

        t = cpu_time_remaining
        total_time += t

        p.worked_time += t
        p.t_round = total_time

        final_table.append(p.copy())

        # remove finished process from process list
        processList.pop(i)

        print(p)
        order_table.append((p.copy(), t))

        # next index is process with most priority (lowest priority
number)
        try:
            i = choose_priority_idx(processList)
        except Exception:
            break

    ##############################################
    # Table For Order of Execution
    row1 = ["Time"]
    row2 = ["PID"]
    for p_copy, t in order_table:
```

```
        row1.append(f"{t} ms")
        row2.append(f"{p_copy.pid}")

    print(tabulate((row1, row2), tablefmt="pretty"))

    # Table For Turn Around Time and Wait Time
    total_wait_time = sum(p.t_round - p.burst_time for p in final_table)
    print(tabulate(((p.pid, p.t_round, p.t_round - p.burst_time) for p
in final_table), headers=("PID", "Turn Around Time (ms)", "Wait Time
(ms)"), tablefmt="pretty"))
    print(f"Average Wait Time: {total_wait_time /
args.num_processes:.2f} ms")

if __name__ == "__main__":
    main()
```