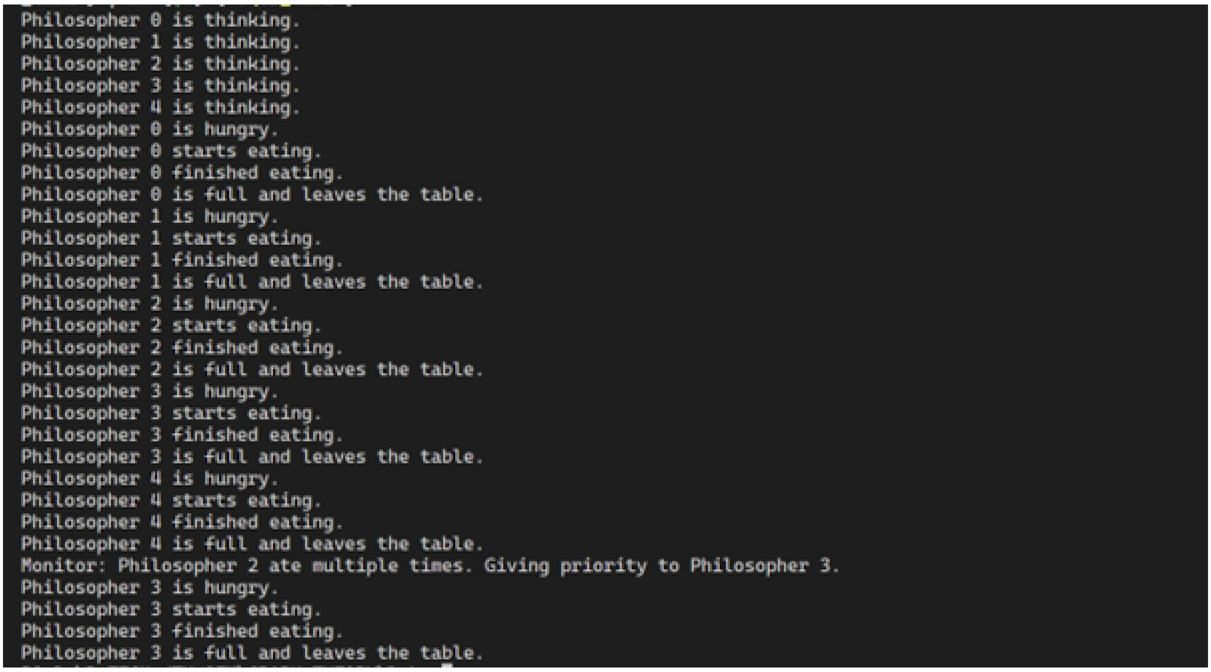
Name: Rathod kavya pankajbhai
Roll no-2023IMG042
SET-C
Q-1

```
#include <bits/stdc++.h>

using namespace std;

#define N 5

sem_t chopsticks[N];

mutex monitor_mutex;

condition_variable cv;

bool eaten[N] = {false};

int eat_count[N] = {0};


void monitor(int id) {

    unique_lock<mutex> lock(monitor_mutex);

    while (true) {

        if (eat_count[2] > 2 && !eaten[3]) {

            cout << "Monitor: Philosopher 2 ate multiple times. Giving priority to
Philosopher 3.\n";

            cv.notify_all();

        }

    }

}


void philosopher(int id) {
```

```
cout << "Philosopher " << id << " is thinking.\n";
this_thread::sleep_for(chrono::milliseconds(500));

while (!eaten[id]) {
    cout << "Philosopher " << id << " is hungry.\n";
    sem_wait(&chopsticks[id]);
    sem_wait(&chopsticks[(id + 1) % N]);

    {
        unique_lock<mutex> lock(monitor_mutex);
        cout << "Philosopher " << id << " starts eating.\n";
        this_thread::sleep_for(chrono::milliseconds(1000));
        if (id == 2) eat_count[id]++;
        if (id == 3 && eat_count[2] > 2) {
            eaten[id] = true;
        }
        cout << "Philosopher " << id << " finished eating.\n";
    }

    sem_post(&chopsticks[id]);
    sem_post(&chopsticks[(id + 1) % N]);

    {
        unique_lock<mutex> lock(monitor_mutex);
```

```
            if (eaten[id]) break;
            cv.wait(lock);
        }
    }

    cout << "Philosopher " << id << " is full and leaves the table.\n";
}

int main() {
    for (int i = 0; i < N; i++) sem_init(&chopsticks[i], 0, 1);
    vector<thread> threads;
    for (int i = 0; i < N; i++) threads.push_back(thread(philosopher, i));
    thread mon(monitor, 3);
    for (auto &t : threads) t.join();
    mon.detach();
    return 0;
}
```



```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is hungry.
Philosopher 0 starts eating.
Philosopher 0 finished eating.
Philosopher 0 is full and leaves the table.
Philosopher 1 is hungry.
Philosopher 1 starts eating.
Philosopher 1 finished eating.
Philosopher 1 is full and leaves the table.
Philosopher 2 is hungry.
Philosopher 2 starts eating.
Philosopher 2 finished eating.
Philosopher 2 is full and leaves the table.
Philosopher 3 is hungry.
Philosopher 3 starts eating.
Philosopher 3 finished eating.
Philosopher 3 is full and leaves the table.
Philosopher 4 is hungry.
Philosopher 4 starts eating.
Philosopher 4 finished eating.
Philosopher 4 is full and leaves the table.
Monitor: Philosopher 2 ate multiple times. Giving priority to Philosopher 3.
Philosopher 3 is hungry.
Philosopher 3 starts eating.
Philosopher 3 finished eating.
Philosopher 3 is full and leaves the table.
```

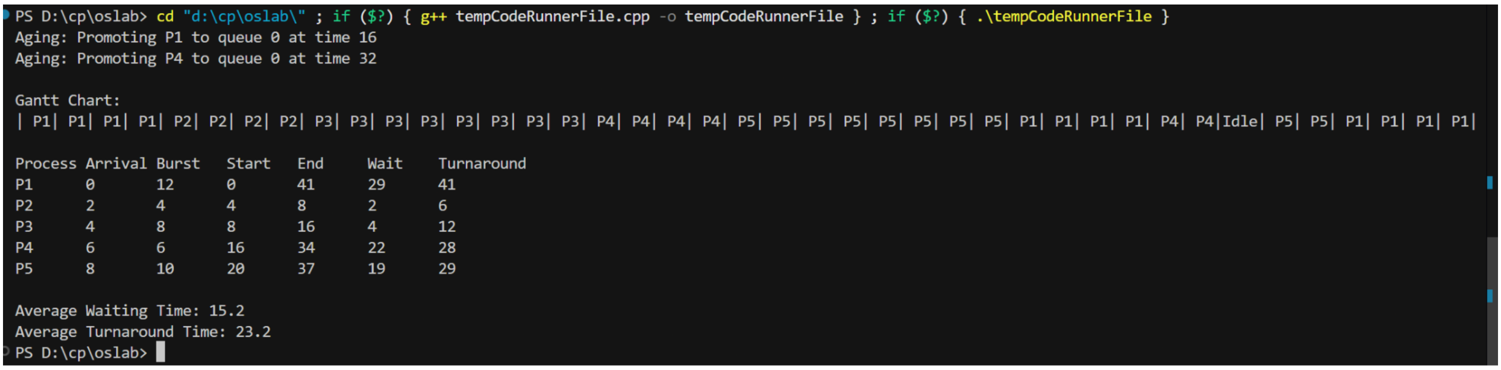
Q-2

```
#include <iostream>
#include <queue>
#include <vector>
#include <iomanip>
#include <algorithm>
using namespace std;

struct Process {
    string pid;
    int arrival, burst, remaining;
    int start_time = -1, end_time = 0;
    int wait = 0, turnaround = 0;
    int queue_level = 0;
    int last_executed_time = 0;
};

vector<Process> processes;
vector<string> gantt;
```

```
bool allDone(const vector<Process>& ps) {
    for (auto& p : ps) if (p.remaining > 0) return false;
    return true;
}

void enqueueNewArrivals(queue<int>& Q0, int current_time) {
    for (int i = 0; i < processes.size(); ++i) {
        if (processes[i].arrival == current_time && processes[i].remaining > 0) {
            Q0.push(i);
            processes[i].queue_level = 0;
        }
    }
}

void promoteAgedProcesses(vector<int>& Q1, queue<int>& Q0, queue<int>& Q2,
int time) {
    for (int i = 0; i < processes.size(); ++i) {
        if (processes[i].remaining > 0 && time - processes[i].last_executed_time >=
10) {
            if (processes[i].queue_level > 0) {
                cout << "Aging: Promoting " << processes[i].pid << " to queue 0 at time "
<< time << "\n";
                processes[i].queue_level = 0;
                Q0.push(i);
                Q1.erase(remove(Q1.begin(), Q1.end(), i), Q1.end());
                queue<int> tmp;
                while (!Q2.empty()) {
                    if (Q2.front() != i) tmp.push(Q2.front());
                    Q2.pop();
                }
                Q2 = tmp;
            }
        }
    }
}

void MLFQ() {
```

```
queue<int> Q0;
vector<int> Q1;
queue<int> Q2;

int time = 0;
while (!allDone(processes)) {
    enqueueNewArrivals(Q0, time);
    promoteAgedProcesses(Q1, Q0, Q2, time);

    int idx = -1;
    int runTime = 0;

    if (!Q0.empty()) {
        idx = Q0.front(); Q0.pop();
        runTime = min(4, processes[idx].remaining);
    }
    else if (!Q1.empty()) {
        sort(Q1.begin(), Q1.end(), [](int a, int b) {
            return processes[a].remaining < processes[b].remaining;
        });
        idx = Q1.front();
        Q1.erase(Q1.begin());
        runTime = processes[idx].remaining;
    }
    else if (!Q2.empty()) {
        idx = Q2.front(); Q2.pop();
        runTime = processes[idx].remaining;
    }

    if (idx != -1 && processes[idx].remaining > 0) {
        if (processes[idx].start_time == -1)
            processes[idx].start_time = time;

        for (int i = 0; i < runTime; ++i) {
            gantt.push_back(processes[idx].pid);
            time++;
            enqueueNewArrivals(Q0, time);
            }

            processes[idx].remaining -= runTime;
            processes[idx].last_executed_time = time;

            if (processes[idx].remaining == 0) {
                processes[idx].end_time = time;
            } else {
                if (processes[idx].queue_level == 0) {
                    processes[idx].queue_level = 1;
                    Q1.push_back(idx);
                } else if (processes[idx].queue_level == 1) {
                    processes[idx].queue_level = 2;
                    Q2.push(idx);
                }
            }
        } else {
            gantt.push_back("Idle");
            time++;
        }
    }

    float total_wait = 0, total_turn = 0;
    cout << "\nGantt Chart:\n";
    for (auto& x : gantt) cout << "|" << setw(3) << x;
    cout << "|\n";

    cout << "\nProcess\tArrival\tBurst\tStart\tEnd\tWait\tTurnaround\n";
    for (auto& p : processes) {
        p.turnaround = p.end_time - p.arrival;
        p.wait = p.turnaround - p.burst;
        total_wait += p.wait;
        total_turn += p.turnaround;
        cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
             << p.start_time << "\t" << p.end_time << "\t" << p.wait << "\t" <<
p.turnaround << "\n";
    }
    cout << "\nAverage Waiting Time: " << total_wait / processes.size();
    cout << "\nAverage Turnaround Time: " << total_turn / processes.size() << "\n";
}

int main() {
    processes = {
        {"P1", 0, 12, 12},
        {"P2", 2,  4,  4},
        {"P3", 4,  8,  8},
        {"P4", 6,  6,  6},
        {"P5", 8, 10, 10}
    };
    MLFQ();
    return 0;
}
```

Output:-



```
PS D:\cp\oslab> cd "d:\cp\oslab\" ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Aging: Promoting P1 to queue 0 at time 16
Aging: Promoting P4 to queue 0 at time 32

Gantt Chart:
| P1| P1| P1| P1| P2| P2| P2| P2| P3| P3| P3| P3| P3| P3| P3| P3| P4| P4| P4| P4| P5| P5| P5| P5| P5| P5| P5| P5| P1| P1| P1| P1| P4| P4|Idle| P5| P5| P1| P1| P1| P1|

Process Arrival Burst   Start   End     Wait    Turnaround
P1      0       12      0       41      29      41
P2      2       4       4       8       2       6
P3      4       8       8       16      4       12
P4      6       6       16      34      22      28
P5      8       10      20      37      19      29

Average Waiting Time: 15.2
Average Turnaround Time: 23.2
PS D:\cp\oslab>
```