struct Compare {

bool operator() (Process p1, Process p2) {

if (p1.bt == p2.bt) {

return p1.id > p2.id;

}

return p1.bt > p2.bt;

}

};

// p is list of process

sort(p.begin(), p.end(), cmp);

// sort list by arrival time

priority\_queue<Process, vector<Process>, Compare> q;//for SJF, priority scheduling

queue<Process> q; // for RR

while (completed < n) { // SJF Non preemptive

while (ptr < n && p[ptr].at <= timer) {

q.push(p[ptr++]);

}

if (q.empty()) {

timer++;

continue;

}

Process t = q.top();

q.pop();

int id = t.id;

timer += t.bt;

tat[id] = timer - t.at;

wt[id] = tat[id] - t.bt;

completed++;

tatavg += tat[id];

wtavg += wt[id];

}

// Above code is same for priority Scheduling

// Non preemptive, except sorting criteria

// process struct has an extra field priority

for (int i = 0; i < n; i++) { // SJF preemptive

bt[p[i].id] = p[i].bt;

}

while (completed < n) {

while (ptr < n && p[ptr].at <= timer) {

q.push(p[ptr++]);

}

timer++;

if (q.empty()) {

continue;

}

Process t = q.top();

q.pop();

int id = t.id;

t.bt--;

if (t.bt == 0) {

completed++;

tat[id] = timer - t.at;

wt[id] = tat[id] - bt[id];

tatavg += tat[id];

wtavg += wt[id];

} else {

q.push(t);

}

}

while (completed < n) { // Plain round robin

while (ptr < n && p[ptr].at <= timer) {

q.push(p[ptr++]);

}

if (q.empty()) {

timer++;

continue;

}

Process t = q.front();

q.pop();

int id = t.id;

if (t.bt <= time\_quantum) {

timer += t.bt;

while (ptr < n && p[ptr].at <= timer) {

q.push(p[ptr++]);

}

completed++;

tat[id] = timer - t.at;

wt[id] = tat[id] - t.bt;

tatavg += tat[id];

wtavg += wt[id];

} else {

timer += time\_quantum;

while (ptr < n && p[ptr].at <= timer) {

q.push(p[ptr++]);

}

t.bt -= time\_quantum;

q.push(t);

}

}

// n -> number of process

// r -> types of resources

vector<int> total(r), available(r);

int allocated[n][r], max\_need[n][r];

int curr\_need[n][r];

available[j] -= allocated[i][j];

curr\_need[i][j] = max\_need[i][j] - allocated[i][j];

int completed = 0;

vector<int> exc\_sequence;

vector<bool> used(n);

while (completed < n) {

int id = -1;

for (int i = 0; i < n; i++) {

if (used[i]) continue;

bool ok = true;

for (int j = 0; j < r; j++) {

if (curr\_need[i][j] > available[j]) {

ok = false;

}

}

if (ok) {

id = i;

break;

}

}

if (id == -1) break;

completed++;

used[id] = true;

exc\_sequence.push\_back(id + 1);

for (int i = 0; i < r; i++) {

available[i] += allocated[id][i];

}

}

if (completed < n) unsafe;