

| **TITLE:** Implementation of Basic Process management algorithms - Preemptive (SRTN, RR, priority ) |
| --- |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** To implement basic Process management algorithms ( Round Robin,SRTN, Priority)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 2.** To understand the concept of process, thread and resource management.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems” McGraw Hill Third**

**Edition.**

1. **William Stallings, “Operating System Internal & Design Principles”, Pearson.**
2. **Andrew S. Tanenbaum, “Modern Operating System”, Prentice Hall.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Most systems handle numerous processes with short CPU bursts interspersed with I/O requests and a few processes with long CPU bursts. To ensure good time-sharing performance, a running process may be preempted to allow another to run. The ready list, or run queue, maintains all processes ready to run and not blocked by I/O or other system requests. Entries in this list point to the process control block, which stores all process information and state.

When an I/O request completes, the process moves from the waiting state to the ready state and is placed on the run queue. The process scheduler, a key component of the operating system, decides whether the current process should continue running or if another should take over. This decision is triggered by four events:

1. The current process issues an I/O request or system request, moving it from running to waiting.
2. The current process terminates.
3. A timer interrupt indicates the process has run for its allotted time, moving it from running to ready.
4. An I/O operation completes, moving the process from waiting to ready, potentially preempting the current process.

The scheduling algorithm, or policy, determines the sequence and duration of process execution, a complex task given the limited information about ready processes.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Description of the application to be implemented**:

**Round Robin Algorithm**

Round Robin (RR) scheduling is a time-sharing algorithm where each process is assigned a fixed time quantum. Processes are executed in a cyclic order, ensuring that each process gets an equal chance to use the CPU. This approach promotes fairness and simplicity but requires careful tuning of the time quantum to avoid excessive context switching or poor responsiveness.

# Shortest Remaining Time First Algorithm :

Shortest Remaining Time First (SRTF) is a preemptive scheduling algorithm that selects the process with the shortest remaining execution time to run next. By prioritizing processes with less remaining time, it minimizes average waiting time and can lead to efficient CPU usage. However, it can cause longer processes to be starved if shorter processes continually arrive.

# Priority scheduling:

Priority Scheduling assigns a priority level to each process and executes processes based on their priority. Higher-priority processes are selected for execution before lower-priority ones. This method ensures that important tasks are completed sooner but can result in starvation of lower-priority processes if high-priority tasks keep arriving.

**Implementation details:** (printout of code)  
Round Robin:

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void roundRobin(int n, vector<int> arrival\_times, vector<int> burst\_times, int time\_quantum) {

vector<int> remaining\_burst\_times = burst\_times;

vector<int> completion\_times(n, -1);

vector<int> gantt;

queue<int> q;

vector<bool> in\_queue(n, false);

int time = 0;

int completed = 0;

int current\_time = 0;

vector<pair<int, int>> gantt\_chart; // To store pairs of (time, process)

while (completed < n) {

// Add all processes that have arrived and are not in the queue

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

if (arrival\_times[i] <= time && !in\_queue[i] && remaining\_burst\_times[i] > 0) {

q.push(i);

in\_queue[i] = true;

}

}

if (q.empty()) {

time++;

continue;

}

int current\_process = q.front();

q.pop();

in\_queue[current\_process] = false;

int execution\_time = min(time\_quantum, remaining\_burst\_times[current\_process]);

remaining\_burst\_times[current\_process] -= execution\_time;

if (gantt\_chart.empty() || gantt\_chart.back().second != current\_process) {

gantt\_chart.push\_back({current\_time, current\_process});

}

current\_time += execution\_time;

// If burst time is finished, record the completion time

if (remaining\_burst\_times[current\_process] == 0) {

completion\_times[current\_process] = current\_time;

completed++;

} else {

q.push(current\_process);

in\_queue[current\_process] = true;

}

}

// Calculate turnaround times and waiting times

vector<int> turnaround\_times(n), waiting\_times(n);

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

turnaround\_times[i] = completion\_times[i] - arrival\_times[i];

waiting\_times[i] = turnaround\_times[i] - burst\_times[i];

}

cout << "Completion Times: ";

for (auto &i : completion\_times) cout << i << " ";

cout << endl;

cout << "Turnaround Times: ";

for (auto &i : turnaround\_times) cout << i << " ";

cout << endl;

cout << "Waiting Times: ";

for (auto &i : waiting\_times) cout << i << " ";

cout << endl;

float avg\_turnaround\_time = 0, avg\_waiting\_time = 0;

for (auto &i : turnaround\_times) avg\_turnaround\_time += i;

for (auto &i : waiting\_times) avg\_waiting\_time += i;

cout << "Avg. Turnaround Time: " << avg\_turnaround\_time / n << endl;

cout << "Avg. Waiting Time: " << avg\_waiting\_time / n << endl;

// Print Gantt Chart

cout << "Gantt Chart\n";

for (auto &entry : gantt\_chart) {

cout << " P" << entry.second + 1;

}

cout << endl;

for (auto &entry : gantt\_chart) {

cout << entry.first << " ";

}

cout << current\_time << endl;

}

int main() {

int n, time\_quantum;

cout << "Enter the number of processes: ";

cin >> n;

vector<int> arrival\_times(n), burst\_times(n);

cout << "Enter the arrival times of the processes: ";

for (int i = 0; i < n; i++) cin >> arrival\_times[i];

cout << "Enter the burst times of the processes: ";

for (int i = 0; i < n; i++) cin >> burst\_times[i];

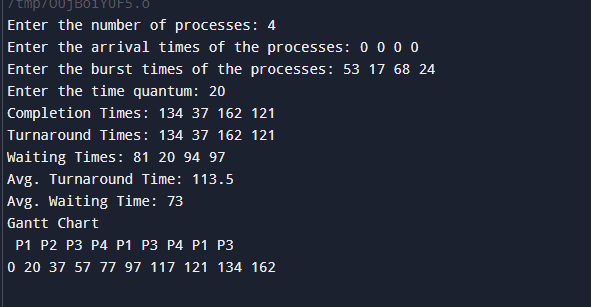
cout << "Enter the time quantum: ";

cin >> time\_quantum;

roundRobin(n, arrival\_times, burst\_times, time\_quantum);

return 0;

}

  
  
Priority (Non-preemptive):

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void pri(int n, vector<int> arrival\_times, vector<int> burst\_times) {

priority\_queue<pair<int, int>> pq;

vector<int> completion\_times(n);

vector<int> gantt;

int time = 0;

int completed = 0;

vector<int> priority(n);

cout << "Enter the priority of the processes (Largest Number First)" << endl;

for(auto &i : priority) cin >> i;

while(completed != n) {

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

if(arrival\_times[i] == time) pq.push({priority[i], i});

}

if (pq.empty()) {

time++;

continue;

}

auto x = pq.top(); pq.pop();

burst\_times[x.second]--;

gantt.push\_back(x.second);

if(burst\_times[x.second] == 0) {

completion\_times[x.second] = time + 1;

completed++;

} else {

pq.push(x);

}

time++;

}

vector<int> turnaround\_times(n), waiting\_times(n);

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

turnaround\_times[i] = completion\_times[i] - arrival\_times[i];

waiting\_times[i] = turnaround\_times[i] - (burst\_times[i] + (completion\_times[i] - time));

}

cout << "Completion Times: ";

for(auto &i : completion\_times) cout << i << " ";

cout << endl;

cout << "Turnaround Times: ";

for(auto &i : turnaround\_times) cout << i << " ";

cout << endl;

cout << "Waiting Times: ";

for(auto &i : waiting\_times) cout << i << " ";

cout << endl;

float avt = 0, avw = 0;

for(auto &i : turnaround\_times) avt += i;

for(auto &i : waiting\_times) avw += i;

cout << "Avg. Turnaround Time: " << avt / n << endl;

cout << "Avg. Waiting Time: " << avw / n << endl;

vector<int> p, t;

for(int i = 0; i < gantt.size(); i++) {

if(i == 0 || gantt[i] != gantt[i-1]) {

p.push\_back(gantt[i]);

t.push\_back(i);

}

}

t.push\_back(time);

cout << "Gantt Chart\n";

for(auto &i : p) cout << " P" << i + 1;

cout << endl;

for(auto &i : t) cout << i << " ";

cout << endl;

}

int main() {

int n;

cout << "Enter the number of processes: ";

cin >> n;

vector<int> arrival\_times(n), burst\_times(n);

cout << "Enter the arrival times of the processes: ";

for(int i = 0; i < n; i++) cin >> arrival\_times[i];

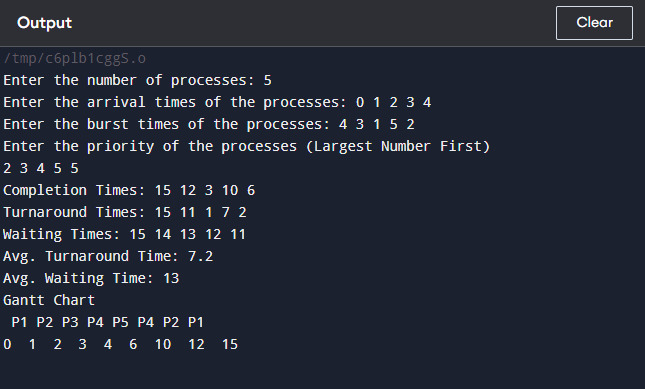
cout << "Enter the burst times of the processes: ";

for(int i = 0; i < n; i++) cin >> burst\_times[i];

pri(n, arrival\_times, burst\_times);

return 0;

}



**Conclusion:**

We have learned about preemptive scheduling algorithms.

**Post Lab Descriptive Questions**

**1.Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?**  
 Process P1:

* I/O: 20% of 10 = 2 units (initial I/O), 10% of 10 = 1 unit (final I/O)
* Compute: 70% of 10 = 7 units

Process P2:

* I/O: 20% of 20 = 4 units (initial I/O), 10% of 20 = 2 units (final I/O)
* Compute: 70% of 20 = 14 units

Process P3:

* I/O: 20% of 30 = 6 units (initial I/O), 10% of 30 = 3 units (final I/O)
* Compute: 70% of 30 = 21 units

#### 

At t = 0: All processes start their initial I/O.

* P1: I/O for 2 units
* P2: I/O for 4 units
* P3: I/O for 6 units
* CPU Utilization: Idle for 2 units (while P1 is in I/O)

At t = 2:

* P1 completes its initial I/O and starts its computation (7 units).
* P2: Still in initial I/O
* P3: Still in initial I/O
* CPU Utilization: 2-6 units (P1 is running)

At t = 6: P2 completes its initial I/O and enters the ready queue with 14 units of computation remaining.

* P1: Continues computation with 3 units remaining
* P3: Still in initial I/O
* CPU Utilization: 6-9 units (P1 is running)

At t = 9: P1 finishes its computation and moves to final I/O (1 unit).

* P2: Starts computation with 14 units remaining
* P3: Still in initial I/O
* CPU Utilization: 9-12 units (P2 is running)

At t = 12: P3 completes its initial I/O and enters the ready queue with 21 units of computation remaining.

* P2: Continues computation (11 units remaining)
* CPU Utilization: 12-23 units (P2 is running)

At t = 23: P2 finishes its computation and starts final I/O (2 units).

* P3: Starts computation with 21 units remaining
* CPU Utilization: 23-44 units (P3 is running)

At t = 44: P3 finishes its computation and starts final I/O (3 units).

* CPU Utilization: 44-47 units (P3 is in final I/O, and no other process is ready)

Idle Time during Initial I/O Phase:

* 0-2: CPU is idle while P1, P2, and P3 are in their initial I/O phases.

Idle Time during Final I/O Phases:

* 23-25: CPU is idle while P2 is in final I/O.
* 44-47: CPU is idle while P3 is in final I/O.

Total Idle Time:

* Initial I/O phase: 2 units
* Final I/O phases: 2 units (for P2) + 3 units (for P3) = 5 units

Total Time: From 0 to 47 units.

Percentage of Idle Time=(Total Idle TimeTotal Time)×100% =( 7/47)\*100 =14.89%

**2. What effect the time quantum has on its performance. What are the advantages and disadvantages of using a small versus a large time quantum?**

In Round Robin (RR) scheduling, the time quantum refers to the maximum amount of time a process is allowed to run before it is preempted.

Small Time Quantum:

Advantages:

* Improves Responsiveness: Processes switch frequently, which can reduce waiting time for interactive processes.
* Better for Time-Sharing Systems: Ensures that no single process monopolizes the CPU.

Disadvantages:

* Increased Context Switching: Frequent switches lead to higher overhead due to more frequent context switching, which can reduce overall system efficiency.
* Potential for High Overhead: If too small, the system spends more time switching than executing processes.

Large Time Quantum:

Advantages:

* Reduced Context Switching Overhead: Fewer switches mean less overhead, and more time is spent on actual process execution.
* Improved Throughput: Larger quanta allow processes to complete their execution without frequent interruptions.

Disadvantages:

* Decreased Responsiveness: Longer time quanta can lead to poorer response times for interactive processes.
* Less Fairness: If a large quantum is given to long-running processes, it may lead to longer wait times for other processes.

**Date: 6/9/24 Signature of faculty in-charge**