***OPTIMIZED ROUND ROBIN TASK SCHEDULING ALGORITHM***

KARTIK KAR AKASH BANERJEE VINAY BHAGWANI NILAKSH GUPTA

Traditional Round Robin Algorithm

CODE:

#include <stdio.h>

#include <stdbool.h>

struct process {

int pid; // Process ID

int at; // Arrival time

int bt; // Burst time

int rt; // Remaining time

int wt; // Waiting time

int tat; // Turnaround time

};

void traditional\_rr(int n, struct process proc[], int tq) {

int time = 0, count = 0;

float total\_wt = 0, total\_tat = 0;

while (count < n) {

bool process\_executed = false;

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

if (proc[i].rt > 0 && proc[i].at <= time) {

process\_executed = true;

printf("<%d>---P%d---", time, proc[i].pid);

// Process execution for the time quantum or until completion

int time\_exec = (proc[i].rt > tq) ? tq : proc[i].rt;

proc[i].rt -= time\_exec;

time += time\_exec;

// If process finishes

if (proc[i].rt == 0) {

count++;

proc[i].tat = time - proc[i].at;

proc[i].wt = proc[i].tat - proc[i].bt;

total\_wt += proc[i].wt;

total\_tat += proc[i].tat;

}

}

}

// If no process is ready to execute, increment time

if (!process\_executed) {

time++;

}

}

printf("<%d>\n", time);

printf("\nAverage Waiting Time: %.2f", total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n);

}

int main() {

int n, tq;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct process proc[n];

printf("Enter time quantum: ");

scanf("%d", &tq);

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

printf("Enter Process ID, Arrival Time, and Burst Time for Process %d: ", i + 1);

scanf("%d %d %d", &proc[i].pid, &proc[i].at, &proc[i].bt);

proc[i].rt = proc[i].bt;

proc[i].wt = 0;

proc[i].tat = 0;

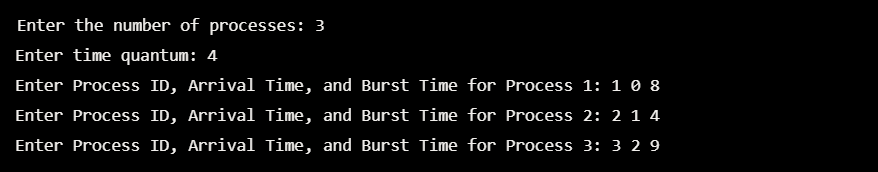
}

printf("\nExecuting processes using Traditional Round Robin:\n");

traditional\_rr(n, proc, tq);

return 0;

}



A screen shot of a computer

Description automatically generated

OPTIMIZED ROUND ROBIN ALGORITHM

CODE:

#include <stdio.h>

#include <stdbool.h>

struct process {

int pid; // Process ID

int at; // Arrival time

int bt; // Burst time

int rt; // Remaining time

int wt; // Waiting time

int tat; // Turnaround time

};

void calculate\_rr\_with\_optimized\_preemption(int n, struct process proc[], int tq) {

int time = 0, count = 0;

float total\_wt = 0, total\_tat = 0;

while (count < n) {

bool process\_executed = false;

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

if (proc[i].rt > 0 && proc[i].at <= time) {

process\_executed = true;

printf("<%d>---P%d---", time, proc[i].pid);

// If remaining time is less than or equal to time quantum

int time\_exec = (proc[i].rt <= tq) ? proc[i].rt : tq;

proc[i].rt -= time\_exec;

time += time\_exec;

// If process finishes

if (proc[i].rt == 0) {

count++;

proc[i].tat = time - proc[i].at;

proc[i].wt = proc[i].tat - proc[i].bt;

total\_wt += proc[i].wt;

total\_tat += proc[i].tat;

}

}

}

// If no process is ready to execute, increment time

if (!process\_executed) {

time++;

}

}

printf("<%d>\n", time);

printf("\nAverage Waiting Time: %.2f", total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n);

}

int main() {

int n, tq;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct process proc[n];

printf("Enter time quantum: ");

scanf("%d", &tq);

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

printf("Enter Process ID, Arrival Time, and Burst Time for Process %d: ", i + 1);

scanf("%d %d %d", &proc[i].pid, &proc[i].at, &proc[i].bt);

proc[i].rt = proc[i].bt;

proc[i].wt = 0;

proc[i].tat = 0;

}

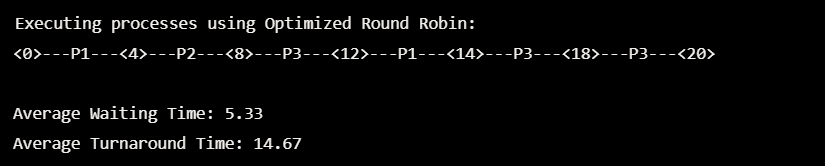
printf("\nExecuting processes using Optimized Round Robin:\n");

calculate\_rr\_with\_optimized\_preemption(n, proc, tq);

return 0;

}

A screenshot of a computer screen

Description automatically generated



**Key Features of the Optimized Algorithm**

1. **Dynamic Time Quantum:**
   * **Instead of using a fixed time quantum for all processes, the optimized algorithm calculates a new optimal time quantum (OTQ) dynamically based on the median burst time of the processes in the ready queue and the highest burst time.**
   * **Formula: OTQ=Median Burst Time+Highest Burst Time2OTQ = \frac{\text{Median Burst Time} + \text{Highest Burst Time}}{2}OTQ=2Median Burst Time+Highest Burst Time​**
   * **This ensures that the time quantum is tailored to the current state of the ready queue, reducing the chance of too many unnecessary context switches.**
2. **Sorting the Ready Queue:**
   * **The processes in the ready queue are sorted in ascending order of their burst times.**
   * **Why? This minimizes waiting time for processes with smaller burst times and prevents them from being unnecessarily delayed.**
3. **Reduced Context Switching:**
   * **Processes with burst times smaller than the calculated time quantum can complete execution in a single turn without being preempted. This reduces the overhead caused by frequent context switching.**
4. **Fair and Starvation-Free:**
   * **Like traditional RR, all processes still get a fair chance to execute. However, by sorting and dynamically adjusting the time quantum, the algorithm improves efficiency and minimizes starvation.**

**Steps Performed by the Optimized Algorithm**

1. **Initialization:**
   * **The algorithm starts by collecting the arrival time and burst time for all processes.**
2. **Ready Queue Handling:**
   * **At each point in time, processes that have arrived are added to the ready queue.**
   * **The ready queue is sorted based on the burst time of the processes.**
3. **Median and OTQ Calculation:**
   * **The median burst time of the ready queue is calculated:**
     + **If the number of processes is odd, the middle burst time is the median.**
     + **If even, the average of the two middle burst times is taken.**
   * **The Optimal Time Quantum (OTQ) is calculated as the average of the median and the highest burst time in the queue.**
4. **Execution of Processes:**
   * **Processes in the ready queue are executed for either the OTQ or their remaining burst time, whichever is smaller.**
   * **If a process completes during its turn, its waiting time and turnaround time are updated.**
   * **If not, the process is returned to the ready queue.**
5. **Repeat Until Completion:**
   * **The steps are repeated until all processes have completed execution.**

**Key Differences**

1. **Context Switches:**
   * **Traditional RR:** Preempts processes after every time quantum, even if a process's remaining burst time is less than the time quantum.
   * **Optimized RR:** Reduces preemption by allowing processes with remaining time ≤ time quantum to complete execution.
2. **Performance:**
   * **Waiting Time:** Reduced in optimized RR due to fewer context switches and better scheduling.
   * **Turnaround Time:** Optimized RR improves average turnaround time by prioritizing completion over preemption.
3. **Real-World Implications:**
   * The optimized RR is better suited for multitasking and real-time systems where reducing overhead is critical.

**Why This Matters**

"This optimization makes Round Robin more practical for systems where reducing context switching is crucial, such as real-time and multitasking environments. By allowing processes with smaller burst times to finish in one cycle, the algorithm achieves a better balance between fairness and efficiency."