Priority-Driven Scheduling Algorithm

Abstract:

Scheduling algorithms are methods used in operating systems to distribute resources among processes in a scheduled manner. The purpose of these algorithms is to minimize resource starvation by ensuring that each process gets its required CPU time.

The algorithm implemented here is a **priority driven** algorithm i.e., a process is assigned a priority , and the scheduler selects the process to run according to it . In this case, the priority is set according to the deadline of the processes , and in case of having same deadlines, it selects processes based on their shortest execution time. The integration of two approaches makes it an efficient solution for real – time environment.

Implementation

The structure represents a process with attributes like process ID, arrival time, deadline, execution time, and remaining execution time. Using C++ and multithreading, the implementation executes processes based on their priority (deadline) while maintaining efficiency. Key metrics such as throughput, completion time, waiting time, and turnaround time are calculated.

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <climits>

using namespace std;

struct Process {
    int process_id;
    int arrival_time;
    int deadline;
    int execution_time;
    int remaining_time;
    int start_time = -1;
    int completion_time = -1;
};

struct Compare {
    bool operator()(const Process* p1, const Process* p2) {
        if (p1->deadline == p2->deadline)
            return p1->remaining_time > p2->remaining_time;
        return p1->deadline > p2->deadline;
    }
};

priority_queue<Process*, vector<Process*>, Compare> process_queue;
mutex queue_mutex;
condition_variable cv;
bool done = false;

void execute_processes(vector<Process>& processes, int& current_time) {
    while (true) {
        unique_lock<mutex> lock(queue_mutex);
        cv.wait(lock, [] { return !process_queue.empty() || done; });

        if (done && process_queue.empty()) break;
```

```
        Process* current_process = process_queue.top();
        process_queue.pop();

        if (current_process->start_time == -1)
            current_process->start_time = current_time;

        current_process->remaining_time--;
        current_time++;

        if (current_process->remaining_time == 0) {
            current_process->completion_time = current_time;
        } else {
            process_queue.push(current_process);
        }
    }
}

void calculate_metrics(vector<Process>& processes) {
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int total_processes = processes.size();

    cout << "Process\tArrival Time\tDeadline\tExecution Time\tStart
Time\tCompletion Time\tWaiting Time\tTurnaround Time\n";

    for (const auto& p : processes) {
        int turnaround_time = p.completion_time - p.arrival_time;
        int waiting_time = turnaround_time - p.execution_time;

        total_turnaround_time += turnaround_time;
        total_waiting_time += waiting_time;

        cout << p.process_id << "\t\t" << p.arrival_time << "\t\t" <<
p.deadline << "\t\t"
             << p.execution_time << "\t\t" << p.start_time << "\t\t" <<
p.completion_time
             << "\t\t" << waiting_time << "\t\t" << turnaround_time <<
"\n";
    }

    double throughput = (double)total_processes /
processes.back().completion_time;
```

```
    cout << "\nAverage Turnaround Time: " << (double)total_turnaround_time
/ total_processes << endl;
    cout << "Average Waiting Time: " << (double)total_waiting_time /
total_processes << endl;
    cout << "Throughput: " << throughput << " processes/unit time\n";
}

void schedule_processes(vector<Process>& processes) {
    int current_time = 0;
    int total_completed = 0;

    thread worker(execute_processes, ref(processes), ref(current_time));

    while (total_completed < processes.size()) {
        {
            lock_guard<mutex> lock(queue_mutex);
            for (auto& process : processes) {
                if (process.arrival_time <= current_time &&
process.remaining_time > 0 && process.start_time == -1) {
                    process_queue.push(&process);
                }
            }
        }
        cv.notify_one();

        this_thread::sleep_for(chrono::milliseconds(100));

        total_completed = 0;
        for (const auto& process : processes) {
            if (process.remaining_time == 0) total_completed++;
        }
    }

    {
        lock_guard<mutex> lock(queue_mutex);
        done = true;
    }
    cv.notify_one();
    worker.join();

    calculate_metrics(processes);
}
```

```
int main() {
    vector<Process> processes = {
        {1, 0, 10, 10, 10},
        {2, 1, 6, 8, 8},
        {3, 2, 8, 6, 6},
        {4, 3, 12, 4, 4},
        {5, 4, 15, 5, 5}
    };

    schedule_processes(processes);

    return 0;
}
```

Results

The implementation ensures efficient scheduling by prioritizing deadlines and minimizing waiting times. It calculates and displays the following metrics for each process:

1. Completion Time
2. Waiting Time
3. Turnaround Time

Additionally, the system computes the throughput of the scheduling algorithm.

The results demonstrate significant performance improvements and effective resource allocation, making this scheduling algorithm suitable for real-time environments.

Process	Arrival Time	Deadline	Execution Time	Start Time	Completion Time	Waiting Time	Turnaround Time
1	0	10	10	0	10	0	10
2	1	6	8	10	18	9	17
3	2	8	6	18	24	16	22
4	3	12	4	24	28	21	25
5	4	15	5	28	33	24	29

Average Turnaround Time: 20.6
Average Waiting Time: 14

Tech Stacks

- **Programming Language:** C++
- **Concurrency Management:** Multithreading using <thread> and <mutex> libraries
- **Data Structures:** Priority Queue (Min-Heap)
- **Tools/Environments:** GCC/G++, Visual Studio Code
- **Operating System Concepts:** Scheduling Algorithms, Process Management
- **Performance Metrics Calculation:** Throughput, Waiting Time, Turnaround Time