**Scheduling**

Scheduling in an operating system (OS) refers to the method by which work is assigned to resources that complete the work. The resource can be the CPU, I/O devices, or other resources required to execute a process. Effective scheduling is crucial to ensure efficient CPU utilization, system responsiveness, and fairness among users and processes.

**Types of Scheduling:**

* **CPU Scheduling**: Decides which of the processes in the ready queue will be allocated the CPU.
* **I/O Scheduling**: Decides the order in which I/O requests are processed.
* **Disk Scheduling**: Determines the order in which disk I/O requests are serviced.
* **Network Scheduling**: Manages the access to the network resources.

**Goals of Scheduling:**

* Maximize CPU Utilization: Ensure that the CPU is busy as much as possible.
* Maximize Throughput: Increase the number of processes that complete their execution per time unit.
* Minimize Turnaround Time: Reduce the time taken from the submission of a process to its completion.
* Minimize Waiting Time: Reduce the time processes spend in the ready queue waiting for CPU allocation.
* Minimize Response Time: Improve the time it takes from when a request is submitted until the first response is produced.
* Fairness: Ensure that each process gets a fair share of the CPU.

**Types of CPU Scheduling:**

1. **Preemptive Scheduling**

* Allows a process to be interrupted and moved to the ready queue, allowing another process to use the CPU. It is used to achieve higher responsiveness.
* Preemptive scheduling is a CPU scheduling method that allows the operating system to interrupt and suspend a currently running process, allocating the CPU to another process. This ensures that higher priority processes or those requiring timely execution can preempt lower priority ones, leading to better overall system responsiveness and efficiency.
* Preemptive scheduling enables an operating system to make decisions about which process should be executed at any given time, based on criteria such as priority, burst time, or time quantum. The scheduler can force a context switch by saving the state of the currently running process and loading the state of the next process to be executed. This mechanism allows the system to respond quickly to changes in process priorities or to ensure that no single process monopolizes the CPU.

**Types of Preemptive Scheduling:**

* 1. **Round Robin (RR)**

Round Robin (RR) scheduling is a widely used preemptive scheduling algorithm designed to ensure fairness and efficiency in time-sharing systems. Each process is given a fixed time slice, known as a time quantum, to execute. If a process does not finish its execution within this time slice, it is preempted and placed at the end of the ready queue, and the next process in the queue is given the CPU.

**Key Concepts**

* Time Quantum: A fixed duration during which a process is allowed to run.
* Context Switching: The process of saving the state of the currently running process and loading the state of the next process.

**Advantages**

* Fairness: All processes get an equal share of the CPU.
* Responsiveness: Suitable for time-sharing systems where users need quick responses.

**Disadvantages**

* Context Switching Overhead: Frequent context switches can degrade performance.
* Time Quantum Selection: If the time quantum is too small, it leads to excessive context switching; if too large, it behaves like FCFS (First-Come, First-Served).

**Example Code: Round Robin Scheduling in C++**

#include <iostream>

#include <queue>

struct Process {

int pid; **// Process ID**

int burstTime; **// Burst Time**

int remainingTime; **// Remaining Time**

int waitingTime; **// Waiting Time**

int turnaroundTime; **// Turnaround Time**

Process(int id, int bt) : pid(id), burstTime(bt), remainingTime(bt), waitingTime(0), turnaroundTime(0) {}

};

void roundRobinScheduling(std::vector<Process> &processes, int timeQuantum) {

std::queue<Process\*> processQueue;

for (auto &process : processes) {

processQueue.push(&process);

}

int time = 0;

while (!processQueue.empty()) {

Process \*currentProcess = processQueue.front();

processQueue.pop();

if (currentProcess->remainingTime > timeQuantum) {

time += timeQuantum;

currentProcess->remainingTime -= timeQuantum;

processQueue.push(currentProcess);

} else {

time += currentProcess->remainingTime;

currentProcess->remainingTime = 0;

currentProcess->turnaroundTime = time;

currentProcess->waitingTime = currentProcess->turnaroundTime - currentProcess->burstTime;

}

}

}

int main() {

std::vector<Process> processes = {Process(1, 10), Process(2, 4), Process(3, 5), Process(4, 8)};

int timeQuantum = 3;

roundRobinScheduling(processes, timeQuantum);

for (const auto &process : processes) {

std::cout << "Process " << process.pid

<< ": Waiting Time = " << process.waitingTime

<< ", Turnaround Time = " << process.turnaroundTime << std::endl;

}

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID, burst time, remaining time, waiting time, and turnaround time.
* **roundRobinScheduling Function**: Implements the RR scheduling algorithm. It processes each process in the queue, allowing each to run for a fixed time quantum. If the process does not finish within the time quantum, it is placed back in the queue.
* **Main Function**: Defines a set of sample processes and a fixed time quantum, calls the scheduling function, and prints the results showing the waiting and turnaround times for each process.

**Block Diagram of Round Robin Scheduling**

**+-----------------------------+**

**| Ready Queue |**

**| (Process Waiting List) |**

**+-----------------------------+**

**| ^**

**| |**

**V |**

**+-------------------|------------------+**

**| | |**

**| Dispatcher | |**

**| (Selects Next | |**

**| Process to Run) | |**

**+-------------------|------------------+**

**| |**

**V |**

**+-----------------------------+ (Preemption Signal after Time Quantum)**

**| Running Process | <-------------------+**

**+-----------------------------+ |**

**| |**

**| |**

**V |**

**+-----------------------------+ |**

**| Save Process State | |**

**| (Context Switching) | |**

**+-----------------------------+ |**

**| |**

**V |**

**+-----------------------------+ |**

**| Load Next Process State | --------------------+**

**| (Context Switching) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Execute Next Process |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Ready Queue**: Contains all the processes that are ready to be executed.
* **Dispatcher**: Selects the next process from the ready queue based on the scheduling algorithm.
* **Running Process**: The process currently being executed by the CPU.
* **Preemption Signal**: Indicates when a running process should be preempted after the time quantum expires.
* **Context Switching**: Saves the state of the currently running process and loads the state of the next process to be executed.
  1. **Shortest Remaining Time First (SRTF)**

Shortest Remaining Time First (SRTF) is a preemptive scheduling algorithm that selects the process with the shortest remaining burst time to execute next. If a new process arrives with a burst time shorter than the remaining time of the current process, the current process is preempted, and the new process is executed. This algorithm is a preemptive version of Shortest Job Next (SJN) and aims to minimize the average waiting time and turnaround time.

**Key Concepts**

* Preemption: The currently running process can be interrupted if a new process arrives with a shorter remaining burst time.
* Shortest Remaining Time: The CPU is always assigned to the process with the shortest remaining burst time among the ready processes.

**Advantages**

* Minimizes Waiting Time: By always executing the process with the shortest remaining time, it minimizes the average waiting time.
* Efficiency: Generally, improves overall system performance compared to non-preemptive algorithms.

**Disadvantages**

* Starvation: Processes with longer burst times may suffer from starvation if shorter processes keep arriving.
* Complexity: Keeping track of the remaining time and preempting processes adds complexity to the scheduling mechanism.

**Example Code: Shortest Remaining Time First (SRTF) Scheduling in C++**

#include <iostream>

#include <vector>

#include <algorithm>

struct Process {

int pid; **// Process ID**

int arrivalTime; **// Arrival Time**

int burstTime; **// Burst Time**

int remainingTime; **// Remaining Time**

int waitingTime; **// Waiting Time**

int turnaroundTime; **// Turnaround Time**

Process(int id, int at, int bt) : pid(id), arrivalTime(at), burstTime(bt), remainingTime(bt), waitingTime(0), turnaroundTime(0) {}

};

bool compareArrival(Process a, Process b) {

return a.arrivalTime < b.arrivalTime;

}

void calculateWaitingAndTurnaroundTime(std::vector<Process> &processes) {

int currentTime = 0;

int completed = 0;

int n = processes.size();

while (completed != n) {

int shortestIndex = -1;

int shortestTime = INT\_MAX;

**// Find the process with the shortest remaining time that has arrived**

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

if (processes[i].arrivalTime <= currentTime && processes[i].remainingTime > 0 && processes[i].remainingTime < shortestTime) {

shortestTime = processes[i].remainingTime;

shortestIndex = i;

}

}

if (shortestIndex == -1) {

++currentTime;

continue;

}

Process &currentProcess = processes[shortestIndex];

currentProcess.remainingTime--;

currentTime++;

if (currentProcess.remainingTime == 0) {

completed++;

currentProcess.turnaroundTime = currentTime - currentProcess.arrivalTime;

currentProcess.waitingTime = currentProcess.turnaroundTime - currentProcess.burstTime;

}

}

}

int main() {

std::vector<Process> processes = {

Process(1, 0, 7),

Process(2, 2, 4),

Process(3, 4, 1),

Process(4, 5, 4)

};

std::sort(processes.begin(), processes.end(), compareArrival);

calculateWaitingAndTurnaroundTime(processes);

std::cout << "PID\tArrival\tBurst\tWaiting\tTurnaround\n";

for (const auto &process : processes) {

std::cout << process.pid << "\t" << process.arrivalTime << "\t" << process.burstTime << "\t"

<< process.waitingTime << "\t" << process.turnaroundTime << "\n";

}

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID, arrival time, burst time, remaining time, waiting time, and turnaround time.
* **compareArrival Function**: A helper function to sort processes based on their arrival times.
* **calculateWaitingAndTurnaroundTime Function**: Implements the SRTF scheduling algorithm. It continuously selects the process with the shortest remaining time and updates the waiting and turnaround times.
* **Main Function**: Defines a set of sample processes, sorts them by arrival time, and calls the scheduling function to calculate the waiting and turnaround times. Finally, it prints the results.

**Block Diagram of SRTF Scheduling**

**+-----------------------------+**

**| Ready Queue |**

**| (Process Waiting List) |**

**+-----------------------------+**

**| ^**

**| |**

**V |**

**+-------------------|------------------+**

**| | |**

**| Dispatcher | |**

**| (Selects Next | |**

**| Process to Run) | |**

**+-------------------|------------------+**

**| |**

**V |**

**+-----------------------------+ (Preemption Signal if a process with shorter remaining time arrives)**

**| Running Process | <-------------------+**

**+-----------------------------+ |**

**| |**

**| |**

**V |**

**+-----------------------------+ |**

**| Save Process State | |**

**| (Context Switching) | |**

**+-----------------------------+ |**

**| |**

**V |**

**+-----------------------------+ |**

**| Load Next Process State | --------------------+**

**| (Context Switching) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Execute Next Process |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Ready Queue**: Contains all the processes that are ready to be executed.
* **Dispatcher**: Selects the next process from the ready queue based on the scheduling algorithm.
* **Running Process**: The process currently being executed by the CPU.
* **Preemption Signal**: Indicates when a running process should be preempted if a new process with a shorter remaining time arrives.
* **Context Switching**: Saves the state of the currently running process and loads the state of the next process to be executed.
  1. **Priority Scheduling**

Priority scheduling is a preemptive scheduling algorithm where each process is assigned a priority. The CPU is allocated to the process with the highest priority (smallest numerical value) among the ready processes. If a new process arrives with a higher priority than the currently running process, the CPU is preempted, and the higher priority process is executed. Priority can be determined based on various factors such as process type, time constraints, or other criteria defined by the system.

**Key Concepts**

* Priority: A numerical value assigned to each process indicating its importance or urgency.
* Preemption: The ability to interrupt the currently executing process to give the CPU to a higher priority process.
* Aging: A technique used to prevent starvation by gradually increasing the priority of waiting processes.

**Advantages**

* Allows Prioritization: Important tasks can be executed first.
* Efficient Use of Resources: High priority tasks are completed quickly.
* Flexibility: Different types of tasks can be assigned different priorities.

**Disadvantages**

* Starvation: Low priority processes may never execute if high priority processes keep arriving.
* Indefinite Blocking: A low priority process may be blocked indefinitely if high priority processes keep arriving.

**Example Code: Priority Scheduling in C++**

#include <iostream>

#include <vector>

#include <queue>

struct Process {

int pid; **// Process ID**

int arrivalTime; **// Arrival Time**

int burstTime; **// Burst Time**

int priority; **// Priority**

int remainingTime; **// Remaining Time**

int waitingTime; **// Waiting Time**

int turnaroundTime; **// Turnaround Time**

Process(int id, int at, int bt, int prio) : pid(id), arrivalTime(at), burstTime(bt), priority(prio), remainingTime(bt), waitingTime(0), turnaroundTime(0) {}

};

struct ComparePriority {

bool operator()(const Process &a, const Process &b) {

return a.priority > b.priority;

}

};

void priorityScheduling(std::vector<Process> &processes) {

std::priority\_queue<Process, std::vector<Process>, ComparePriority> pq;

int currentTime = 0;

int completed = 0;

int n = processes.size();

int index = 0;

while (completed != n) {

**// Push all processes that have arrived by current time to the priority queue**

while (index < n && processes[index].arrivalTime <= currentTime) {

pq.push(processes[index]);

++index;

}

if (!pq.empty()) {

Process currentProcess = pq.top();

pq.pop();

**// Execute current process for 1 time unit**

currentTime++;

currentProcess.remainingTime--;

**// Check if process is completed**

if (currentProcess.remainingTime == 0) {

completed++;

currentProcess.turnaroundTime = currentTime - currentProcess.arrivalTime;

currentProcess.waitingTime = currentProcess.turnaroundTime - currentProcess.burstTime;

} else {

pq.push(currentProcess); **// Push back the process with updated remaining time**

}

} else {

currentTime++;

}

}

}

int main() {

std::vector<Process> processes = {

Process(1, 0, 7, 2),

Process(2, 2, 4, 1),

Process(3, 4, 1, 3),

Process(4, 5, 4, 4)

};

priorityScheduling(processes);

std::cout << "PID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n";

for (const auto &process : processes) {

std::cout << process.pid << "\t" << process.arrivalTime << "\t" << process.burstTime << "\t"

<< process.priority << "\t\t" << process.waitingTime << "\t" << process.turnaroundTime << "\n";

}

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID, arrival time, burst time, priority, remaining time, waiting time, and turnaround time.
* **ComparePriority Struct**: A comparator struct used to order processes based on priority for the priority queue.
* **priorityScheduling Function**: Implements the priority scheduling algorithm using a priority queue (std::priority\_queue). It continuously selects and executes the process with the highest priority among the ready processes.
* **Main Function**: Defines a set of sample processes, calls the scheduling function to calculate the waiting and turnaround times, and prints the results.

**Block Diagram of Priority Scheduling**

**+-----------------------------+**

**| Ready Queue |**

**| (Process Waiting List) |**

**+-----------------------------+**

**| ^**

**| |**

**V |**

**+-------------------|------------------+**

**| | |**

**| Dispatcher | |**

**| (Selects Next | |**

**| Process to Run) | |**

**+-------------------|------------------+**

**| |**

**V |**

**+---------------------------+(Preemption Signal if higher priority process arrives)**

**| Running Process | <-------------------+**

**+---------------------------+ |**

**| |**

**| |**

**V |**

**+-----------------------------+ |**

**| Save Process State | |**

**| (Context Switching) | |**

**+-----------------------------+ |**

**| |**

**V |**

**+---------------------------- |**

**| Load Next Process State | ------------------+**

**| (Context Switching) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Execute Next Process |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Ready Queue**: Contains all the processes that are ready to be executed, ordered by priority.
* **Dispatcher**: Selects the next process from the ready queue based on the scheduling algorithm (highest priority).
* **Running Process**: The process currently being executed by the CPU.
* **Preemption Signal**: Indicates when a running process should be preempted if a new process with a higher priority arrives.
* **Context Switching**: Saves the state of the currently running process and loads the state of the next process to be executed.

1. **Non-Preemptive Scheduling**

Once a process gets the CPU, it runs to completion. It’s simpler but can lead to poor performance for short processes.

Non-preemptive scheduling is a scheduling technique where once a process starts executing, it continues to run until it completes or blocks itself for I/O or voluntarily relinquishes the CPU. In non-preemptive scheduling, the operating system does not interrupt the running process; thus, the process keeps the CPU until it finishes its burst time.

**Key Concepts**

* Burst Time: The amount of time a process requires to complete its execution.
* Completion: A process runs to completion or blocks on I/O before another process can start.
* No Preemption: Once a process starts executing, it cannot be interrupted by the scheduler.

**Advantages**

* Simplicity: Easier to implement and manage compared to preemptive scheduling.
* Lower Overhead: Minimal overhead due to fewer context switches.
* Predictable Behavior: Process execution can be predicted more accurately.

**Disadvantages**

* Low Responsiveness: A long-running process can block other processes from executing, leading to slower system response.
* Inefficient Use of CPU: CPU may be idle even if higher priority processes arrive.

There are primarily two types of non-preemptive scheduling algorithms commonly used:

* 1. **First-Come, First-Served (FCFS)**

First-Come, First-Served (FCFS) is the simplest form of non-preemptive scheduling where processes are executed in the order they arrive in the ready queue. Once a process starts executing, it continues until it completes its CPU burst time, voluntarily relinquishes the CPU, or blocks for I/O. FCFS scheduling can lead to a situation known as the "convoy effect," where shorter processes wait behind longer ones, potentially increasing the average waiting time.

**Key Concepts**

* Ready Queue: Contains all processes that are ready to be executed, ordered by their arrival time.
* Dispatcher: Selects the next process from the ready queue based on the order of arrival.
* Completion: Processes execute in the order they arrive and run until they finish or block for I/O.

**Advantages**

* Simplicity: Easy to implement and understand.
* No Starvation: Every process eventually gets CPU time.
* Minimal Overhead: No overhead due to context switching.

**Disadvantages**

* Convoy Effect: Longer processes can delay shorter ones, increasing average waiting time.
* Inefficient CPU Utilization: CPU may remain idle if a shorter process arrives later.

**Example Code: FCFS Non-Preemptive Scheduling in C++**

#include <iostream>

#include <vector>

struct Process {

int pid; **// Process ID**

int arrivalTime; **// Arrival Time**

int burstTime; **// Burst Time**

int waitingTime; **// Waiting Time**

int turnaroundTime; **// Turnaround Time**

Process(int id, int at, int bt) : pid(id), arrivalTime(at), burstTime(bt), waitingTime(0), turnaroundTime(0) {}

};

void fcfsScheduling(std::vector<Process> &processes) {

int currentTime = 0;

for (auto &process : processes) {

if (currentTime < process.arrivalTime) {

currentTime = process.arrivalTime;

}

process.waitingTime = currentTime - process.arrivalTime;

process.turnaroundTime = process.waitingTime + process.burstTime;

currentTime += process.burstTime;

}

}

int main() {

std::vector<Process> processes = {

Process(1, 0, 7),

Process(2, 2, 4),

Process(3, 4, 1),

Process(4, 5, 4)

};

fcfsScheduling(processes);

std::cout << "PID\tArrival\tBurst\tWaiting\tTurnaround\n";

for (const auto &process : processes) {

std::cout << process.pid << "\t" << process.arrivalTime << "\t" << process.burstTime << "\t"

<< process.waitingTime << "\t" << process.turnaroundTime << "\n";

}

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID, arrival time, burst time, waiting time, and turnaround time.
* **fcfsScheduling Function**: Implements FCFS non-preemptive scheduling. It processes each process in the order of their arrival, calculates waiting and turnaround times based on the completion order.
* **Main Function**: Defines a set of sample processes, calls the scheduling function to calculate the waiting and turnaround times, and prints the results.

**Block Diagram of FCFS Non-Preemptive Scheduling**

**+-----------------------------+**

**| Ready Queue |**

**| (Process Waiting List) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Dispatcher (FCFS Order) |**

**| (Selects Next Process) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Running Process |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Execute Process |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Process Completion |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Ready Queue**: Contains all the processes that are ready to be executed, usually in the order they arrive.
* **Dispatcher**: Selects the next process from the ready queue based on the scheduling algorithm (FCFS in this case).
* **Running Process**: The process currently being executed by the CPU.
* **Execute Process**: The CPU executes the selected process until it completes its burst time.
* **Process Completion**: After completing its burst time, the process leaves the CPU.
  1. **Shortest Job First (SJF)**

Shortest Job First (SJF) non-preemptive scheduling is a scheduling algorithm that selects the process with the smallest burst time next. It aims to minimize the average waiting time and improve system performance by executing shorter jobs first. In non-preemptive SJF, once a process starts executing, it continues until it completes its burst time, voluntarily relinquishes the CPU, or blocks for I/O.

**Key Concepts**

* Burst Time: The amount of time a process requires to complete its execution.
* Ready Queue: Contains all processes that are ready to be executed, ordered by their burst times.
* Dispatcher: Selects the next process from the ready queue based on the smallest burst time.
* Completion: Processes execute in the order of their burst times.

**Advantages**

* Minimizes Waiting Time: Shorter jobs are executed first, reducing the average waiting time.
* Efficient Use of CPU: Improves CPU utilization by minimizing idle time.
* Fairness: Every process gets executed eventually.

**Disadvantages**

* Predicting Burst Times: Requires accurate estimation or prior knowledge of burst times.
* Starvation: Longer jobs may wait indefinitely if shorter jobs continue to arrive.

**Example Code: SJF Non-Preemptive Scheduling in C++:**

#include <iostream>

#include <vector>

#include <algorithm>

struct Process {

int pid; **// Process ID**

int arrivalTime; **// Arrival Time**

int burstTime; **// Burst Time**

int waitingTime; **// Waiting Time**

int turnaroundTime; **// Turnaround Time**

Process(int id, int at, int bt) : pid(id), arrivalTime(at), burstTime(bt), waitingTime(0), turnaroundTime(0) {}

};

bool compareArrivalTime(Process a, Process b) {

return a.arrivalTime < b.arrivalTime;

}

bool compareBurstTime(Process a, Process b) {

return a.burstTime < b.burstTime;

}

void sjfScheduling(std::vector<Process> &processes) {

int n = processes.size();

int currentTime = 0;

int completed = 0;

std::vector<bool> completedProcesses(n, false);

std::sort(processes.begin(), processes.end(), compareArrivalTime);

while (completed < n) {

int shortestIndex = -1;

int shortestBurst = INT\_MAX;

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

if (!completedProcesses[i] && processes[i].arrivalTime <= currentTime) {

if (processes[i].burstTime < shortestBurst) {

shortestBurst = processes[i].burstTime;

shortestIndex = i;

}

}

}

if (shortestIndex != -1) {

completedProcesses[shortestIndex] = true;

currentTime += processes[shortestIndex].burstTime;

processes[shortestIndex].waitingTime = currentTime - processes[shortestIndex].arrivalTime; processes[shortestIndex].turnaroundTime = processes[shortestIndex].waitingTime + processes[shortestIndex].burstTime;

++completed;

} else {

++currentTime;

}

}

}

int main() {

std::vector<Process> processes = {

Process(1, 0, 7),

Process(2, 2, 4),

Process(3, 4, 1),

Process(4, 5, 4)

};

sjfScheduling(processes);

std::cout << "PID\tArrival\tBurst\tWaiting\tTurnaround\n";

for (const auto &process : processes) {

std::cout << process.pid << "\t" << process.arrivalTime << "\t" << process.burstTime << "\t"

<< process.waitingTime << "\t" << process.turnaroundTime << "\n";

}

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID, arrival time, burst time, waiting time, and turnaround time.
* **compareArrivalTime** and **compareBurstTime Functions:** Comparator functions used for sorting processes based on arrival time and burst time respectively.
* **sjfScheduling Function**: Implements SJF non-preemptive scheduling. It sorts processes by arrival time, selects the process with the shortest burst time that has arrived, and calculates waiting and turnaround times.
* **Main Function**: Defines a set of sample processes, calls the scheduling function to calculate the waiting and turnaround times, and prints the results.

**Block Diagram of SJF Non-Preemptive Scheduling**

**+-----------------------------+**

**| Ready Queue |**

**| (Process Waiting List) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Dispatcher (SJF Order) |**

**| (Selects Shortest Job) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Running Process |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Execute Process |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Process Completion |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Ready Queue**: Contains all the processes that are ready to be executed, ordered by their burst times.
* **Dispatcher**: Selects the next process from the ready queue based on the shortest burst time (SJF order).
* **Running Process**: The process currently being executed by the CPU.
* **Execute Process**: The CPU executes the selected process until it completes its burst time.
* **Process Completion**: After completing its burst time, the process leaves the CPU.

**Multilevel Queue Scheduling**

Multilevel Queue Scheduling is a scheduling algorithm that partitions the ready queue into several separate queues, each with its own scheduling algorithm. Each queue has a different priority level, and processes are permanently assigned to one queue based on certain properties like process type, priority, or other criteria. Processes in lower priority queues will only be scheduled if higher priority queues are empty.

**Key Concepts**

* Multiple Queues: Divides processes into separate queues, typically based on priority or other criteria.
* Different Scheduling Algorithms: Each queue may use a different scheduling algorithm (e.g., FCFS, SJF, Round Robin).
* Queue Assignment: Processes are permanently assigned to a particular queue based on characteristics like priority, process type, etc.
* Preemption: Typically, processes in lower priority queues may be preempted by those in higher priority queues.

**Advantages**

* Prioritization: Allows different priority levels for different types of processes.
* Efficiency: Improves system performance by allocating CPU time more effectively based on process characteristics.
* Fairness: Ensures that high-priority processes get executed promptly.

**Disadvantages**

* Complexity: Implementing and managing multiple queues with different scheduling algorithms can be complex.
* Starvation: Lower priority queues may suffer from starvation if higher priority queues are continually filled with processes.
* Overhead: Potential overhead due to managing multiple queues and context switching between them.

**Example Code: Multilevel Queue Scheduling in C++**

#include <iostream>

#include <vector>

#include <queue>

struct Process {

int pid; **// Process ID**

int burstTime; **// Burst Time**

Process(int id, int bt) : pid(id), burstTime(bt) {}

};

void multilevelQueueScheduling(const std::vector<Process> &processes) {

std::queue<Process> highPriorityQueue;

std::queue<Process> lowPriorityQueue;

**// Separate processes into high and low priority queues based on burst time**

for (const auto &process : processes) {

if (process.burstTime <= 5) {

highPriorityQueue.push(process);

} else {

lowPriorityQueue.push(process);

}

}

**// Process high priority queue (FCFS scheduling)**

std::cout << "High Priority Queue:\n";

while (!highPriorityQueue.empty()) {

Process process = highPriorityQueue.front();

highPriorityQueue.pop();

std::cout << "Process " << process.pid << " with burst time " << process.burstTime << " executed.\n";

}

**// Process low priority queue (FCFS scheduling)**

std::cout << "\nLow Priority Queue:\n";

while (!lowPriorityQueue.empty()) {

Process process = lowPriorityQueue.front();

lowPriorityQueue.pop();

std::cout << "Process " << process.pid << " with burst time " << process.burstTime << " executed.\n";

}

}

int main() {

std::vector<Process> processes = {

Process(1, 7),

Process(2, 4),

Process(3, 1),

Process(4, 9),

Process(5, 3)

};

multilevelQueueScheduling(processes);

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID and burst time.
* **multilevelQueueScheduling Function**: Implements multilevel queue scheduling with two queues based on burst time. Processes with burst time <= 5 go to the high-priority queue, and others go to the low-priority queue. Each queue is processed using FCFS scheduling.
* **Main Function**: Defines a set of sample processes, calls the scheduling function to execute them according to their priority, and prints the execution order.

**Block Diagram of Multilevel Queue Scheduling**

**+-----------------------------+**

**| Multilevel Queue |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| High Priority Queue |**

**| (FCFS Scheduling) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Low Priority Queue |**

**| (FCFS Scheduling) |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Multilevel Queue**: Contains multiple queues (in this example, high and low priority queues) where processes are assigned based on their priority or other criteria.
* **High Priority Queue**: Processes with higher priority (shorter burst time in this example) are processed first using FCFS scheduling.
* **Low Priority Queue**: Processes with lower priority (longer burst time in this example) are processed after the high priority queue using FCFS scheduling.

Multilevel Queue Scheduling can be either preemptive or non-preemptive, depending on how it is implemented.

**Multilevel Feedback Queue Scheduling**

Multilevel Feedback Queue Scheduling is an extension of multilevel queue scheduling that allows processes to move between queues dynamically based on their behaviour. This scheduling algorithm is designed to handle varying process priorities and aging effectively by allowing processes to migrate to different queues in response to changes in their execution characteristics.

**Key Concepts**

* Multiple Queues: Like multilevel queue scheduling, processes are categorized into multiple queues, each with its own scheduling algorithm.
* Feedback Mechanism: Processes can move between queues based on predefined criteria, such as CPU burst length, age of the process, or other dynamic factors.
* Priority Adjustment: Allows processes to increase or decrease their priority over time based on their behavior (e.g., if a process uses too much CPU time, it may be moved to a lower priority queue).
* Avoids Starvation: Ensures that all processes eventually get CPU time by allowing them to move to higher priority queues if they are not executing.

**Advantages**

* Adaptability: Handles varying process priorities and behavior effectively.
* Avoids Aging Issues: Prevents starvation by allowing aging of processes in lower priority queues.
* Improves System Responsiveness: Allows interactive processes to be prioritized over CPU-bound processes dynamically.

**Disadvantages**

* Complexity: Implementing and managing multiple queues with dynamic feedback mechanisms can be complex.
* Overhead: Increased overhead due to the need for frequent queue adjustments and context switches.

**Example Code: Multilevel Feedback Queue Scheduling in C++**

#include <iostream>

#include <vector>

#include <queue>

struct Process {

int pid; **// Process ID**

int burstTime; **// Burst Time**

int currentQueue; **// Current Queue index**

Process(int id, int bt) : pid(id), burstTime(bt), currentQueue(0) {}

};

void multilevelFeedbackQueueScheduling(std::vector<Process> &processes) {

std::vector<std::queue<Process>> queues(2); **// Two queues for simplicity**

int quantum = 4; **// Quantum for the second queue**

for (auto &process : processes) {

queues[0].push(process); **// Initially push all processes to the first queue**

}

int currentTime = 0;

while (!queues[0].empty() || !queues[1].empty()) {

if (!queues[0].empty()) {

Process currentProcess = queues[0].front();

queues[0].pop();

std::cout << "Time " << currentTime << ": Process " << currentProcess.pid << " (Queue 1) executing.\n";

currentTime += std::min(currentProcess.burstTime, quantum); **// Execute for burst time or quantum**

currentProcess.burstTime -= quantum; **// Decrease burst time**

if (currentProcess.burstTime > 0) {

queues[1].push(currentProcess); **// Move to second queue if burst time remains**

}

} else if (!queues[1].empty()) {

Process currentProcess = queues[1].front();

queues[1].pop();

std::cout << "Time " << currentTime << ": Process " << currentProcess.pid << " (Queue 2) executing.\n";

currentTime += currentProcess.burstTime; **// Execute remaining burst time**

}

}

}

int main() {

std::vector<Process> processes = {

Process(1, 8),

Process(2, 4),

Process(3, 9),

Process(4, 5),

Process(5, 2)

};

multilevelFeedbackQueueScheduling(processes);

return 0;

}

**Explanation of the Code**

* **Process Struct**: Defines a structure for a process with attributes for process ID, burst time, and the current queue it belongs to.
* **multilevelFeedbackQueueScheduling Function**: Implements multilevel feedback queue scheduling with two queues. Processes start in the first queue and move to the second queue if they do not finish within the quantum. The first queue uses round-robin scheduling with a fixed quantum, while the second queue executes processes to completion.
* **Main Function**: Defines a set of sample processes, calls the scheduling function to execute them based on the multilevel feedback queue scheduling, and prints the execution order.

**Block Diagram of Multilevel Feedback Queue Scheduling**

**+-----------------------------+**

**| Multilevel Feedback |**

**| Queue |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Queue 1 (RR) |**

**| (Quantum = 4) |**

**+-----------------------------+**

**|**

**V**

**+-----------------------------+**

**| Queue 2 (FCFS) |**

**+-----------------------------+**

**Explanation of the Block Diagram**

* **Multilevel Feedback Queue**: Contains two queues where processes are initially placed in Queue 1 and may move to Queue 2 based on predefined criteria (in this case, remaining burst time).
* **Queue 1 (RR)**: Processes in Queue 1 are scheduled using round-robin scheduling with a fixed quantum (in this example, quantum = 4).
* **Queue 2 (FCFS)**: Processes in Queue 2 are executed using first-come, first-served (FCFS) scheduling until completion.

**Scheduling in symmetric multiprocessing (SMP)**

Scheduling in symmetric multiprocessing (SMP) systems is a crucial component of operating system design. SMP systems have multiple identical processors that share the same memory and I/O resources. Each processor can execute tasks independently, but they operate under a unified operating system kernel that manages task allocation and resource sharing. Here's a detailed explanation of scheduling in SMP systems:

**Overview of SMP Systems**

* Symmetric Multiprocessing (SMP): In SMP systems, multiple processors are connected to a single, shared memory and operate under a single OS instance. All processors are treated equally, hence the term "symmetric".
* Shared Resources: Processors share the same physical memory, I/O devices, and data paths.
* Scalability and Performance: SMP systems are designed to improve performance and scalability by distributing the workload among multiple processors.

**Scheduling in SMP Systems**

Scheduling in SMP systems involves assigning tasks (processes or threads) to available processors. The goal is to optimize the overall system performance, ensure fairness, and manage processor utilization effectively.

1. **Scheduler Design**
2. **Centralized Scheduler**

* A single scheduler makes decisions for all processors. This can lead to contention and bottlenecks.
* A centralized scheduler in an SMP system manages the scheduling of tasks from a single point of control. This approach can simplify the management of task assignment and load balancing, but it may become a bottleneck as the number of processors increases.

**Here’s a simple block diagram illustrating the centralized scheduler in an SMP system:**

**+----------------------+**

**| Task Queue |**

**+----------+-----------+**

**|**

**v**

**+----------+-----------+**

**| Centralized Scheduler|**

**+----------+-----------+**

**|**

**+----------------+-----------------+**

**| | |**

**v v v**

**+------+-----+ +-----+------+ +-----+------+**

**| Processor 1| | Processor 2| | Processor 3|**

**+------------+ +------------+ +------------+**

**Explanation**

* **Task Queue**: A global task queue where all tasks are stored.
* **Centralized Scheduler**: A single scheduler that makes decisions about which task should be executed by which processor.
* **Processors**: Multiple processors that execute the tasks assigned by the scheduler.

**Here’s a simplified C++ implementation of a centralized scheduler in an SMP system:**

#include <iostream>

#include <queue>

#include <vector>

#include <thread>

#include <mutex>

#include <condition\_variable>

using namespace std;

class Task {

public:

int id;

Task(int id) : id(id) {}

void execute() {

cout << "Executing task " << id << endl;

}

};

class CentralizedScheduler {

private:

queue<Task> taskQueue;

mutex mtx;

condition\_variable cv;

bool stop;

public:

CentralizedScheduler() : stop(false) {}

void addTask(Task task) {

unique\_lock<mutex> lock(mtx);

taskQueue.push(task);

cv.notify\_one();

}

void worker(int processorId) {

while (true) {

Task task(0);

{

unique\_lock<mutex> lock(mtx);

cv.wait(lock, [this] { return !taskQueue.empty() || stop; });

if (stop && taskQueue.empty()) break;

task = taskQueue.front();

taskQueue.pop();

}

cout << "Processor " << processorId << " ";

task.execute();

}

}

void start(vector<thread> &threads) {

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

threads[i] = thread(&CentralizedScheduler::worker, this, i + 1);

}

}

void stopScheduler() {

{

unique\_lock<mutex> lock(mtx);

stop = true;

}

cv.notify\_all();

}

};

int main() {

CentralizedScheduler scheduler;

vector<thread> threads(4); **// Assume we have 4 processors**

scheduler.start(threads);

**// Add tasks to the scheduler**

for (int i = 1; i <= 10; ++i) {

scheduler.addTask(Task(i));

}

**// Stop the scheduler and join the threads**

scheduler.stopScheduler();

for (auto &t : threads) {

if (t.joinable()) {

t.join();

}

}

return 0;

}

**Detailed Explanation of the Code**

* **Task Class**: Represents a task with a simple execute method to simulate task execution.
* **CentralizedScheduler Class**:
* taskQueue: A queue to hold the tasks.
* mtx: A mutex to protect shared resources.
* cv: A condition variable for synchronizing task assignment.
* stop: A boolean flag to signal the scheduler to stop.
* **addTask Method**: Adds a new task to the task queue and notifies one waiting thread.
* **worker Method**: Represents a processor's work loop. Each processor waits for tasks, executes them, and handles the stop condition.
* **start Method**: Initializes and starts the worker threads.
* **stopScheduler Method**: Signals all worker threads to stop and wakes them up.
* **Main Function:**
* Creates a CentralizedScheduler instance.
* Starts 4 worker threads (representing 4 processors).
* Adds tasks to the scheduler.
* Stops the scheduler and waits for all threads to finish.

This implementation demonstrates a simple centralized scheduler managing tasks across multiple processors. The centralized approach makes it straightforward to add and manage tasks, but it can become a bottleneck as the number of processors and tasks increases.

1. **Distributed Scheduler**

* Each processor has its own scheduler, which can make independent decisions. This approach can reduce contention but requires careful synchronization to avoid conflicts.
* In a distributed scheduler architecture for SMP systems, each processor has its own local scheduler, which manages a local queue of tasks. This approach aims to reduce contention and improve scalability by decentralizing the scheduling decisions.

**Here's a block diagram illustrating the distributed scheduler in an SMP system:**

**+-------------+ +--------------+ +--------------+**

**| Task Queue | | Task Queue | | Task Queue |**

**|(Processor 1)| | (Processor 2)| | (Processor 3)|**

**+------+------+ +-------+------+ +-------+------+**

**| | |**

**v v v**

**+------+---------+ +--------+---------+ +--------+--------+**

**| Local Scheduler| | Local Scheduler | | Local Scheduler |**

**| (Processor 1) | | (Processor 2) | | (Processor 3) |**

**+------+---------+ +--------+---------+ +--------+--------+**

**| | |**

**v v v**

**+---------+--------+ +--------+---------+ +--------+---------+**

**| Processor 1 | | Processor 2 | | Processor 3 |**

**+------------------+ +------------------+ +------------------+**

**Explanation**

* Task Queues: Each processor has its own local task queue.
* Local Schedulers: Each processor has its own local scheduler responsible for managing its task queue.
* Processors: Multiple processors execute the tasks assigned by their respective local schedulers.

**Here’s a simplified C++ implementation of a distributed scheduler in an SMP system:**

#include <iostream>

#include <queue>

#include <vector>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <atomic>

#include <chrono>

using namespace std;

class Task {

public:

int id;

Task(int id) : id(id) {}

void execute() {

cout << "Executing task " << id << endl;

this\_thread::sleep\_for(chrono::milliseconds(100)); **// Simulate task execution time**

}

};

class LocalScheduler {

private:

queue<Task> taskQueue;

mutex mtx;

condition\_variable cv;

atomic<bool> stop;

public:

LocalScheduler() : stop(false) {}

void addTask(Task task) {

unique\_lock<mutex> lock(mtx);

taskQueue.push(task);

cv.notify\_one();

}

void worker(int processorId) {

while (true) {

Task task(0);

{

unique\_lock<mutex> lock(mtx);

cv.wait(lock, [this] { return !taskQueue.empty() || stop.load(); });

if (stop.load() && taskQueue.empty()) break;

task = taskQueue.front();

taskQueue.pop();

}

cout << "Processor " << processorId << " ";

task.execute();

}

}

void start(thread &t, int processorId) {

t = thread(&LocalScheduler::worker, this, processorId);

}

void stopScheduler() {

stop.store(true);

cv.notify\_all();

}

};

int main() {

const int numProcessors = 4;

vector<LocalScheduler> schedulers(numProcessors);

vector<thread> threads(numProcessors);

**// Start local schedulers**

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

schedulers[i].start(threads[i], i + 1);

}

**// Add tasks to schedulers**

for (int i = 1; i <= 10; ++i) {

schedulers[i % numProcessors].addTask(Task(i));

}

**// Stop all schedulers**

for (auto &scheduler : schedulers) {

scheduler.stopScheduler();

}

**// Join all threads**

for (auto &t : threads) {

if (t.joinable()) {

t.join();

}

}

return 0;

}

**Detailed Explanation of the Code**

* Task Class: Represents a task with an execute method to simulate task execution.
* LocalScheduler Class:
* taskQueue: A queue to hold the tasks for each processor.
* mtx: A mutex to protect the local task queue.
* cv: A condition variable for synchronizing task assignment and execution.
* stop: An atomic boolean flag to signal the scheduler to stop.
* addTask Method: Adds a new task to the local task queue and notifies one waiting thread.
* worker Method: Represents a processor's work loop. Each processor waits for tasks, executes them, and handles the stop condition.
* start Method: Initializes and starts the worker thread for a local scheduler.
* stopScheduler Method: Signals the worker thread to stop and wakes it up.
* Main Function:
* Creates local scheduler instances for each processor.
* Starts a worker thread for each local scheduler.
* Distributes tasks among the local schedulers.
* Stops all schedulers and waits for all threads to finish.

This distributed scheduler implementation demonstrates a scalable approach to task management in an SMP system, where each processor has its own local scheduler and task queue.

1. **Load Balancing**
2. **Static Load Balancing**

* The scheduler assigns tasks to processors based on a predefined strategy or algorithm. Tasks may be distributed evenly or according to specific criteria (e.g., processor affinity).
* Static load balancing involves distributing tasks across multiple processors at the time of task creation or assignment, based on a predetermined strategy. Unlike dynamic load balancing, static load balancing does not adjust the task allocation during runtime based on the system's state. This approach is simpler and incurs less runtime overhead but may not handle varying workloads as efficiently as dynamic methods.

**Here’s a block diagram illustrating static load balancing in an SMP system:**

**+-------------------------+**

**| Task Pool |**

**+------------+------------+**

**|**

**v**

**+------------+------------+**

**| Static Load Balancer |**

**+------------+------------+**

**|**

**+------------------+-------------------+**

**| | |**

**v v v**

**+-------+--------+ +-------+--------+ +-------+--------+**

**| Processor 1 | | Processor 2 | | Processor 3 |**

**+----------------+ +----------------+ +----------------+**

**Explanation**

* Task Pool: A collection of tasks that need to be distributed to processors.
* Static Load Balancer: Determines the initial distribution of tasks to processors based on a predefined strategy.
* Processors: Multiple processors execute the tasks assigned by the load balancer.

**Here’s a simplified C++ implementation of static load balancing in an SMP system:**

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <condition\_variable>

using namespace std;

class Task {

public:

int id;

Task(int id) : id(id) {}

void execute() {

cout << "Executing task " << id << " on processor " << this\_thread::get\_id() << endl;

this\_thread::sleep\_for(chrono::milliseconds(100)); **// Simulate task execution time**

}

};

class StaticLoadBalancer {

private:

vector<vector<Task>> processorQueues;

mutex mtx;

public:

StaticLoadBalancer(int numProcessors) {

processorQueues.resize(numProcessors);

}

void addTask(const Task& task, int processorId) {

lock\_guard<mutex> lock(mtx);

processorQueues[processorId].push\_back(task);

}

vector<Task> getTasks(int processorId) {

lock\_guard<mutex> lock(mtx);

return processorQueues[processorId];

}

};

void processorFunction(StaticLoadBalancer &balancer, int processorId) {

vector<Task> tasks = balancer.getTasks(processorId);

for (Task &task : tasks) {

task.execute();

}

}

int main() {

const int numProcessors = 4;

const int numTasks = 10;

StaticLoadBalancer balancer(numProcessors);

**// Distribute tasks statically**

for (int i = 1; i <= numTasks; ++i) {

int processorId = i % numProcessors; **// Simple round-robin assignment**

balancer.addTask(Task(i), processorId);

}

**// Start processors**

vector<thread> threads;

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

threads.push\_back(thread(processorFunction, ref(balancer), i));

}

**// Join all threads**

for (auto &t : threads) {

if (t.joinable()) {

t.join();

}

}

return 0;

}

**Detailed Explanation of the Code**

* Task Class: Represents a task with an execute method to simulate task execution.
* StaticLoadBalancer Class:
* processorQueues: A vector of vectors where each inner vector holds tasks for a specific processor.
* mtx: A mutex to protect access to the task queues.
* addTask Method: Adds a new task to the task queue of a specific processor.
* getTasks Method: Retrieves the task queue for a specific processor.
* processorFunction: Represents the work loop of a processor. Each processor retrieves its tasks from the load balancer and executes them.
* Main Function:
* Creates a StaticLoadBalancer instance for the given number of processors.
* Distributes tasks to processors using a simple round-robin strategy.
* Starts a thread for each processor to execute its tasks.
* Joins all threads to ensure completion.

This static load balancing implementation demonstrates a straightforward approach to distributing tasks across multiple processors in an SMP system. By assigning tasks at the start, it reduces the need for complex runtime management, though it may not handle workload variations as efficiently as dynamic methods.

1. **Dynamic Load Balancing**

* The scheduler continuously monitors the load on each processor and redistributes tasks as needed to ensure balanced load. This can involve task migration from one processor to another.
* Dynamic load balancing involves distributing tasks across multiple processors based on the current workload and system state. Unlike static load balancing, dynamic methods continuously monitor and redistribute tasks to ensure efficient utilization of all processors. This approach can handle varying workloads and system conditions more effectively but involves higher runtime overhead due to the need for constant monitoring and task migration.

**Here’s a block diagram illustrating dynamic load balancing in an SMP system:**

**+----------------------+**

**| Task Queue |**

**+----------+-----------+**

**|**

**v**

**+----------+-----------+**

**| Dynamic Load Balancer|**

**+----------+-----------+**

**|**

**+----------------+-----------------+**

**| | |**

**v v v**

**+------+------+ +----+--------+ +---+---------+**

**| Processor 1 | | Processor 2 | | Processor 3 |**

**+-------------+ +-------------+ +-------------+**

**Explanation**

* Task Queue: A global task queue where all tasks are stored.
* Dynamic Load Balancer: Continuously monitors the load on each processor and redistributes tasks as needed to ensure balanced load.
* Processors: Multiple processors that execute the tasks assigned by the dynamic load balancer.

**Here’s a simplified C++ implementation of dynamic load balancing in an SMP system:**

#include <iostream>

#include <queue>

#include <vector>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <atomic>

#include <chrono>

using namespace std;

class Task {

public:

int id;

Task(int id) : id(id) {}

void execute() {

cout << "Executing task " << id << " on processor " << this\_thread::get\_id() << endl;

this\_thread::sleep\_for(chrono::milliseconds(100)); **// Simulate task execution time**

}

};

class DynamicLoadBalancer {

private:

vector<queue<Task>> processorQueues;

vector<mutex> processorMutexes;

vector<condition\_variable> processorCvs;

vector<atomic<bool>> processorStops;

int numProcessors;

public:

DynamicLoadBalancer(int numProcessors) : numProcessors(numProcessors), processorQueues(numProcessors), processorMutexes(numProcessors), processorCvs(numProcessors), processorStops(numProcessors) {

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

processorStops[i] = false;

}

}

void addTask(const Task& task) {

**// Find the processor with the smallest queue**

int minQueueSize = INT\_MAX;

int targetProcessor = 0;

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

lock\_guard<mutex> lock(processorMutexes[i]);

if (processorQueues[i].size() < minQueueSize) {

minQueueSize = processorQueues[i].size();

targetProcessor = i;

}

}

**// Add the task to the target processor's queue**

{

lock\_guard<mutex> lock(processorMutexes[targetProcessor]);

processorQueues[targetProcessor].push(task);

}

processorCvs[targetProcessor].notify\_one();

}

void worker(int processorId) {

while (true) {

Task task(0);

{

unique\_lock<mutex> lock(processorMutexes[processorId]);

processorCvs[processorId].wait(lock, [this, processorId] { return !processorQueues[processorId].empty() || processorStops[processorId]; });

if (processorStops[processorId] && processorQueues[processorId].empty()) break;

task = processorQueues[processorId].front();

processorQueues[processorId].pop();

}

task.execute();

}

}

void start(vector<thread> &threads) {

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

threads[i] = thread(&DynamicLoadBalancer::worker, this, i);

}

}

void stop() {

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

processorStops[i] = true;

processorCvs[i].notify\_all();

}

}

};

int main() {

const int numProcessors = 4;

DynamicLoadBalancer balancer(numProcessors);

vector<thread> threads(numProcessors);

**// Start dynamic load balancer**

balancer.start(threads);

**// Add tasks to the balancer**

for (int i = 1; i <= 10; ++i) {

balancer.addTask(Task(i));

}

**// Stop the balancer after a delay to allow task execution**

this\_thread::sleep\_for(chrono::seconds(2));

balancer.stop();

// Join all threads

for (auto &t : threads) {

if (t.joinable()) {

t.join();

}

}

return 0;

}

**Detailed Explanation of the Code**

* **Task Class**: Represents a task with an execute method to simulate task execution.
* **DynamicLoadBalancer Class:**
* processorQueues: A vector of queues where each queue holds tasks for a specific processor.
* processorMutexes: A vector of mutexes to protect access to each task queue.
* processorCvs: A vector of condition variables for synchronizing task assignment and execution.
* processorStops: A vector of atomic boolean flags to signal each processor to stop.
* numProcessors: The number of processors in the system.
* **addTask Method**: Adds a new task to the queue of the processor with the smallest queue, ensuring balanced load.
* **worker Method**: Represents the work loop of a processor. Each processor waits for tasks, executes them, and handles the stop condition.
* **start Method**: Initializes and starts the worker threads for all processors.
* **stop Method**: Signals all worker threads to stop and wakes them up.
* **Main Function:**
* Creates a DynamicLoadBalancer instance for the given number of processors.
* Starts the worker threads for each processor.
* Adds tasks to the load balancer.
* Stops the load balancer after a delay to allow task execution.
* Joins all threads to ensure completion.

This dynamic load balancing implementation demonstrates a flexible approach to distributing tasks across multiple processors in an SMP system. By continuously monitoring the load and redistributing tasks, it ensures efficient utilization of all processors, even under varying workloads.

1. **Task Assignment**
2. **Processor Affinity (or CPU Affinity)**

* Tasks are assigned to specific processors to exploit cache locality and reduce context switching overhead. This can improve performance by keeping tasks on the same processor as much as possible.
* Processor affinity, also known as CPU affinity, refers to the practice of binding a process or a thread to a specific CPU or a set of CPUs. This technique helps improve performance by ensuring that a process runs on the same CPU or set of CPUs, which can improve cache utilization and reduce context switching overhead.

**Here's a block diagram illustrating processor affinity in an SMP system:**

**+----------------------+**

**| Task Queue |**

**+-----------+----------+**

**|**

**v**

**+-----------+----------------+**

**| Scheduler (with Affinity) |**

**+-----------+----------------+**

**|**

**+------------------+-----------------+**

**| | |**

**v v v**

**+------+-----+ +------+-----+ +------+-----+**

**| Processor 1| | Processor 2| | Processor 3|**

**+------------+ +------------+ +------------+**

**| | |**

**| | |**

**v v v**

**+------------+ +------------+ +------------+**

**| Task A | | Task B | | Task C |**

**+------------+ +------------+ +------------+**

**Explanation**

* Task Queue: A collection of tasks that need to be scheduled.
* Scheduler (with Affinity): A scheduler that assigns tasks to specific processors based on affinity settings.
* Processors: Multiple processors that execute tasks. Each task may have an affinity to a specific processor.

**Benefits of Processor Affinity**

* Improved Cache Utilization: By keeping a task on the same CPU, the data in the CPU cache remains relevant to that task, reducing cache misses.
* Reduced Context Switching Overhead: Frequent switching of tasks between different CPUs can lead to higher context switching overhead. Processor affinity minimizes this by keeping tasks on the same CPU.

**Here's a simplified C++ implementation of processor affinity using threads and affinity settings in an SMP system:**

#include <iostream>

#include <vector>

#include <thread>

#include <chrono>

#include <sched.h>

#include <unistd.h>

using namespace std;

class Task {

public:

int id;

int affinity;

Task(int id, int affinity) : id(id), affinity(affinity) {}

void execute() {

cpu\_set\_t cpuset;

CPU\_ZERO(&cpuset);

CPU\_SET(affinity, &cpuset);

pthread\_t current\_thread = pthread\_self();

pthread\_setaffinity\_np(current\_thread, sizeof(cpu\_set\_t), &cpuset);

cout << "Executing task " << id << " on processor " << affinity << endl;

this\_thread::sleep\_for(chrono::milliseconds(100)); **// Simulate task execution time**

}

};

void worker(Task task) {

task.execute();

}

int main() {

const int numTasks = 10;

vector<thread> threads;

**// Create and start tasks with specific processor affinities**

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

int affinity = i % thread::hardware\_concurrency(); **// Simple round-robin assignment for affinity**

Task task(i + 1, affinity);

threads.push\_back(thread(worker, task));

}

**// Join all threads**

for (auto &t : threads) {

if (t.joinable()) {

t.join();

}

}

return 0;

}

**Detailed Explanation of the Code**

* **Task Class**: Represents a task with an execute method that sets CPU affinity using pthread\_setaffinity\_np and then executes the task.
* id: Identifier for the task.
* affinity: The CPU to which the task is bound.
* execute Method: Sets the thread's CPU affinity and simulates task execution.
* worker Function: A wrapper function that calls the execute method of the Task class.
* Main Function:
* numTasks: The number of tasks to create.
* threads: A vector to hold the thread objects.
* Task Creation: Creates tasks with specific CPU affinities in a round-robin fashion.
* Thread Creation: Starts a thread for each task and assigns it to the appropriate processor based on affinity.
* Join Threads: Ensures all threads complete their execution by joining them.

**How Processor Affinity is Set**

In this implementation, pthread\_setaffinity\_np is used to set the CPU affinity for a thread. The steps are:

* Initialize CPU Set: Using CPU\_ZERO to initialize the CPU set and CPU\_SET to set the desired CPU.
* Get Current Thread: Using pthread\_self to get the current thread.
* Set Affinity: Using pthread\_setaffinity\_np to set the affinity of the current thread to the desired CPU.

This implementation demonstrates how processor affinity can be used to improve performance in an SMP system by binding tasks to specific CPUs and ensuring they execute on the same CPU consistently.

1. **Work Stealing**

* Idle processors can "steal" tasks from busy processors to balance the load dynamically.
* Work stealing is a dynamic load balancing technique used in SMP systems where idle processors can "steal" tasks from the task queues of other processors. This approach helps in balancing the workload across all processors by ensuring that idle processors can take over work from busy processors, improving overall system efficiency.

**Here's a block diagram illustrating work stealing in an SMP system:**

**+-----------------------+ +-----------------------+ +-----------------------+**

**| Processor 1 | | Processor 2 | | Processor 3 |**

**| +-------------------+| | +-------------------+| | +-------------------+|**

**| | Task Queue || | | Task Queue || | | Task Queue ||**

**| +---^---------------+| | +---^---------------+| | +---^---------------+|**

**| | | | | | | | |**

**| | Steal Task <---+-------|----->| | | | |**

**| +----------------|-------+------+ | | | |**

**+-----------------------+ +-----------------------+ +-----------------------+**

**Explanation**

* Task Queues: Each processor maintains its own task queue.
* Work Stealing: Idle processors can steal tasks from the task queues of other processors to balance the workload.
* Processors: Multiple processors that execute tasks and can steal tasks from others if they become idle.

**Here's a simplified C++ implementation of work stealing in an SMP system:**

#include <iostream>

#include <vector>

#include <deque>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <atomic>

#include <random>

#include <chrono>

using namespace std;

class Task {

public:

int id;

Task(int id) : id(id) {}

void execute() {

cout << "Executing task " << id << " on processor " << this\_thread::get\_id() << endl;

this\_thread::sleep\_for(chrono::milliseconds(100)); **// Simulate task execution time**

}

};

class WorkStealingQueue {

private:

deque<Task> tasks;

mutable mutex mtx;

public:

bool empty() const {

lock\_guard<mutex> lock(mtx);

return tasks.empty();

}

void push(Task task) {

lock\_guard<mutex> lock(mtx);

tasks.push\_back(task);

}

bool pop(Task &task) {

lock\_guard<mutex> lock(mtx);

if (tasks.empty()) return false;

task = tasks.front();

tasks.pop\_front();

return true;

}

bool steal(Task &task) {

lock\_guard<mutex> lock(mtx);

if (tasks.empty()) return false;

task = tasks.back();

tasks.pop\_back();

return true;

}

};

class Processor {

private:

WorkStealingQueue &queue;

vector<WorkStealingQueue> &allQueues;

atomic<bool> stop;

int id;

public:

Processor(int id, WorkStealingQueue &queue, vector<WorkStealingQueue> &allQueues) : id(id), queue(queue), allQueues(allQueues), stop(false) {}

void operator()() {

while (!stop) {

Task task(0);

if (queue.pop(task)) {

task.execute();

} else {

bool stolen = false;

for (size\_t i = 0; i < allQueues.size(); ++i) {

if (allQueues[i].steal(task)) {

task.execute();

stolen = true;

break;

}

}

if (!stolen) {

this\_thread::sleep\_for(chrono::milliseconds(10)); **// No tasks available, sleep for a while**

}

}

}

}

void stopProcessor() {

stop = true;

}

};

int main() {

const int numProcessors = 4;

vector<WorkStealingQueue> queues(numProcessors);

vector<thread> threads;

vector<Processor> processors;

**// Create and start processors**

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

processors.emplace\_back(i, queues[i], queues);

threads.emplace\_back(processors[i]);

}

**// Distribute tasks to the first processor's queue**

for (int i = 1; i <= 10; ++i) {

queues[0].push(Task(i));

}

**// Allow some time for tasks to be processed**

this\_thread::sleep\_for(chrono::seconds(2));

**// Stop all processors**

for (auto &processor : processors) {

processor.stopProcessor();

}

**// Join all threads**

for (auto &t : threads) {

if (t.joinable()) {

t.join();

}

}

return 0;

}

**Detailed Explanation of the Code**

* **Task Class:** Represents a task with an execute method to simulate task execution.
* id: Identifier for the task.
* execute Method: Simulates task execution and prints the task ID and processor ID.
* **WorkStealingQueue Class:**
* tasks: A double-ended queue to hold tasks.
* mtx: A mutex to protect access to the task queue.
* empty Method: Checks if the queue is empty.
* push Method: Adds a task to the back of the queue.
* pop Method: Removes and returns a task from the front of the queue.
* steal Method: Removes and returns a task from the back of the queue.
* **Processor Class:**
* queue: Reference to the processor's own task queue.
* allQueues: Reference to all task queues, allowing the processor to steal tasks from others.
* stop: Atomic boolean flag to signal the processor to stop.
* id: Identifier for the processor.
* operator() Method: The work loop for the processor. It tries to pop a task from its own queue, and if no tasks are available, it attempts to steal a task from other processors' queues.
* stopProcessor Method: Sets the stop flag to true.
* **Main Function:**
* numProcessors: The number of processors.
* queues: A vector of work stealing queues for each processor.
* threads: A vector to hold the thread objects.
* processors: A vector to hold the processor objects.
* Processor Creation: Creates processor objects and starts threads for each processor.
* Task Distribution: Adds tasks to the first processor's queue.
* Allow Task Processing: Waits for a few seconds to allow tasks to be processed.
* Stop Processors: Signals all processors to stop.
* Join Threads: Ensures all threads complete their execution by joining them.

This implementation demonstrates how work stealing can be used to dynamically balance the load across multiple processors in an SMP system, ensuring efficient utilization of all processors and improving overall system performance.