TASK SCHEDULER FOR MULTI-CORE OPERATING SYSTEM

A Mini Project report submitted in partial fulfillment of the requirements for the award of the Degree of

Bachelor of Technology

In

Computer Science & Engineering (CSE)

By

NIKHIL CHAKRAVARTULA (13011A0524)

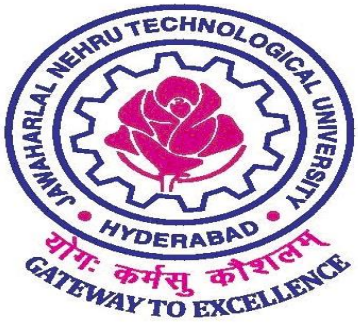
ROHIT BATTU RAJESHWAR(13011A0531)

NELOY DUTTA(13011A0560)

Under the guidance of

Dr. G. VIJAYA KUMARI

Professor



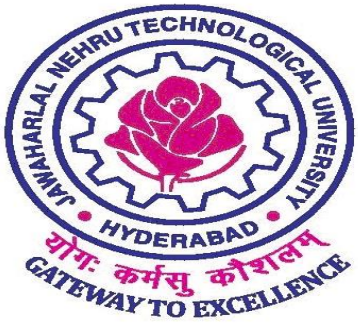
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY

KUKATPALY, HYDERABAD – 500 085.

DECEMBER 2015

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY
KUKATPALY,HYDERABAD – 500 085.

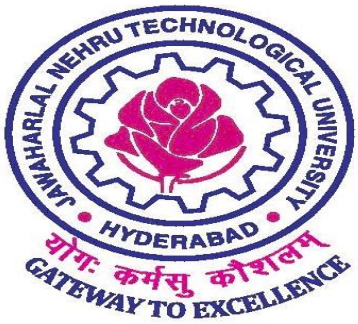


DECLARATION BY THE CANDIDATES

We, **Nikhil Chakravartula(13011A0524), Rohit Battu Rajeshwar(13011A0531**), and **Neloy Dutta (13011A0560**) hereby declare that the mini project report entitled "**TASK SCHEDULER FOR MUTICORE OPERATING SYSTEM**", carried out by us under the guidance of **Dr. G. Vijaya Kumari**, is submitted in partial fulfillment of the requirements for the award of the degree of *Bachelor of Technology in Computer Science and Engineering*. This is a record of bonafide work carried out by us and the results embodied in this project have not been reproduced /copied from any source.

NIKHIL CHAKRAVARTULA (13011A0524)
ROHIT BATTU RAJESHWAR(13011A0531)
NELOY DUTTA(13011A0560)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY
KUKATPALY, HYDERABAD – 500 085.



CERTIFICATE BY THE SUPERVISOR

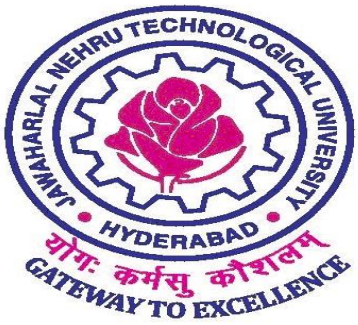
This is to certify that the project report entitled "**TASK SCHEDULER FOR MUTICORE OPERATING SYSTEM**", being submitted by **Nikhil Chakravartula(13011A0524), Rohit Battu Rajeshwar(13011A0531**), and **Neloy Dutta (13011A0560**) in partial fulfillment of the requirements for the award of the degree of *Bachelor of Technology in Computer Science and Engineering*, is a record of bonafide work carried out by them. The results embodied in this project report have not been submitted to any other University or Institute for the award of any other degree or diploma.

Dr. G. Vijaya Kumari
PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY

KUKATPALY, HYDERABAD – 500 085.



CERTIFICATE BY THE HEAD OF THE DEPARTMENT

This is to certify that the project report entitled "**TASK SCHEDULER FOR MUTICORE OPERATING SYSTEM**", being submitted by **Nikhil Chakravartula(13011A0524), Rohit Battu Rajeshwar(13011A0531**), and **Neloy Dutta (13011A0560**) in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science And Engineering, is a record of bonafide work carried out by them.

Dr. V.Kamakshi Prasad,

Professor & Head of the Department

Date:

ACKNOWLEDGEMENT

We wish to extend our sincere gratitude to our supervisor **Dr. G. Vijaya Kumari**, Professor, Department of CSE for being a driving force all through the way and providing necessary computing facilities. This project would not be so smooth and so interesting without his encouragement. We are indebted to the Department of Computer Science and Engineering, JNTUH for providing us with all the required facilities to carry our work in a congenial environment.

We also extend our gratitude to the CSE Department staff for providing necessary support from time to time whenever requested.

NIKHIL CHAKRAVARTULA (13011A0524)
ROHIT BATTU RAJESHWAR(13011A0531)
NELOY DUTTA(13011A0560)

ABSTRACT

Multicore systems have become a standard component in nearly all sorts of computers. Such architectures exhibit a fundamentally different behavior regarding shared resource utilization and performance of non-parallelizable code compared to current CPUs. It is the responsibility of the operating system to optimize overall performance by employing intelligent and configurable scheduling mechanisms. These intelligent scheduling mechanisms must often consider the characteristic features of multicore architectures. A typical multicore cluster system consists of different levels of caches, where some levels are specific to a core and the other levels are shared. So, a considerable amount of time is spent in dealing with the higher-level cache misses, which is clearly an overhead.

A good multicore scheduling algorithm must minimize the overhead involved in cache misses, without compromising on fundamental scheduling criteria. However, the current scheduling mechanisms cannot take care of process priority, processor affinity and core load-balance during scheduling as these three metrics often contradict each other. If an algorithm can be designed which can perform optimally with respect to the all above metrics, it may result in maximum processor utilization and minimum higher level cache miss overhead during migration.

In this project, we propose a new scheduling algorithm which intelligently schedules the processes on different cores optimally. A new task scheduling model is built and then algorithm is proposed on it. The primary objective of the scheduling algorithm is to minimize the migration cost and the secondary objective is to maintain core load balance by conserving the fundamental scheduling criteria. This new design, as we believe, can take care of all the shortcomings of the current scheduling algorithms.

LIST OF CONTENTS

1. INTRODUCTION

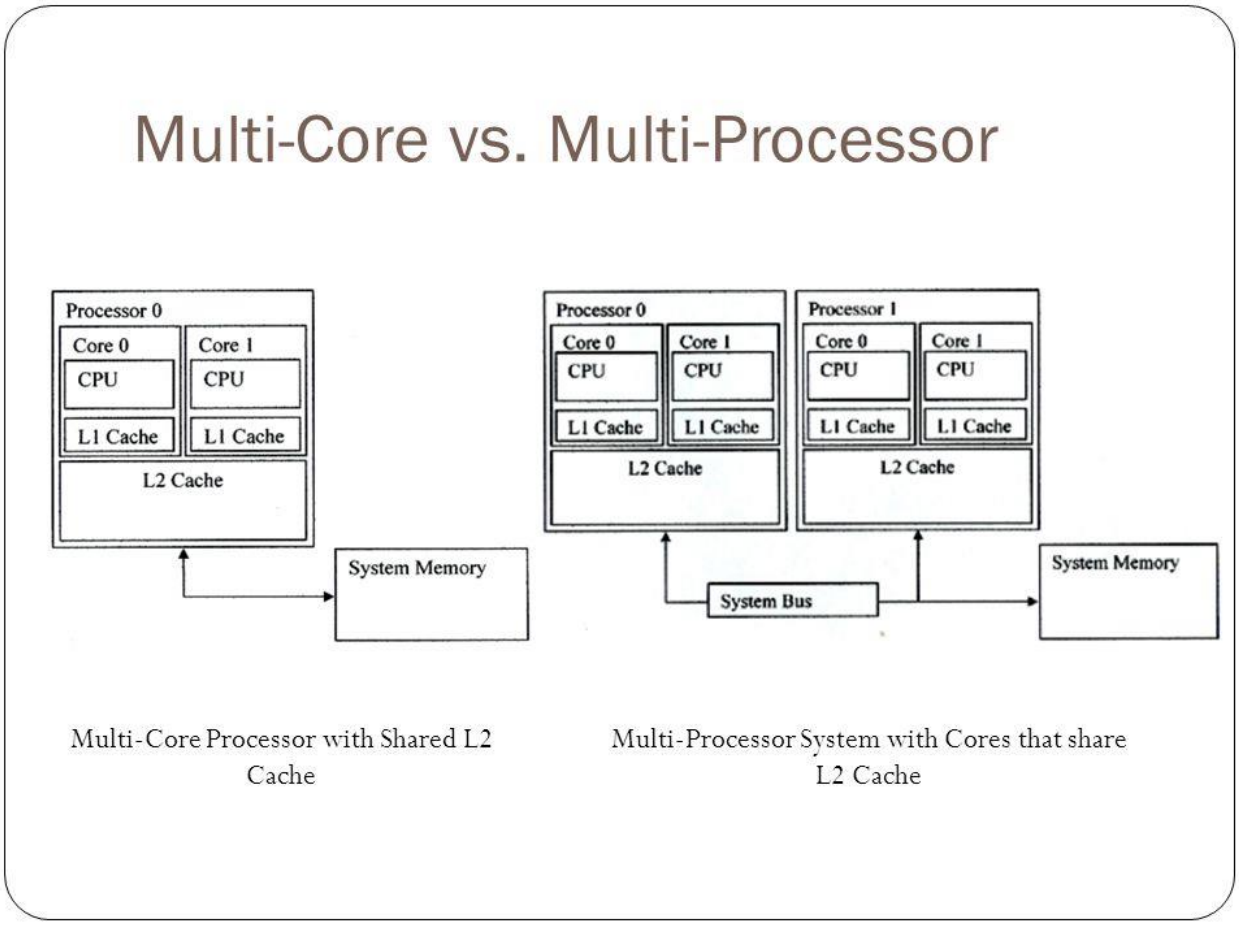
In 1965 Gordon Moore predicted that the number of transistors that can be built onto integrated circuits was going to double every year. In 1975, Moore corrected that assumption to a period of two years, at the present this period is frequently assumed to be 18 months.

Moore's projection has been accurate up to today as it is expected that a new computer shows a significant speedup than previous version systems. To implement the exponential increase of integrated circuits, the transistor structures have to become steadily smaller. Smaller transistor sizes led to higher clock rates of the CPUs, because of the physical factors, the gates in the transistors could perform faster state switches. However, since electronic activity always produces heat as an unwanted by-product, the more transistors are packed together in a small CPU die area, the higher the resulting heat dissipation per unit area becomes. With the higher switching frequency, the electronic activity was performed in smaller intervals, and hence more and more heat-dissipation emerged. The cooling of the processor components became more and more a crucial factor in design considerations and it became clear, that the increasing clock frequency could no longer serve as the primary reason for processor speed up. Hence, there had to be a shift in paradigm to still make applications run faster; on the one hand the amazing abundance of transistors on processor chips was used to increase the cache sizes. This alone, however, would not result in an adequate performance gain, since it only helps memory intensive applications to a certain degree. To effectively counteract the heat problem while making use of the small structures and high number of transistors on a chip, the notion of multi-core processors for consumer PCs was introduced.

A multiprocessor system contains more than one processor and they work in parallel, known as Simultaneous Multiprocessing (SMP).

A multicore system contains more than one execution core on the same die. These several cores work in parallel, called Chip-level Multiprocessing (CMP). A multicore chip can have a dedicated execution unit and L1 cache for a core and a shared L2 cache for the entire CPU. Such shared caches aren’t present in an SMP system.

The following picture contrasts in between multicore system and multiprocessor system.



At the first glance, it seems very intuitive to define scheduling mechanisms for such multicore architectures. A straight forward way is to maintain a run queue for each of the processors and assign the current task to the processor having least run queue length. It seems so simple, but this kind of strategy can easily be proven to be very inefficient. The reason can be centred on the multilevel cache hierarchy. In a most basic multicore processor system, each core has its own level-1 cache and a shared level-2 cache for the entire CPU. By rescheduling interrupted processes to a shorter run queue which belongs to another core, the processes cache working set may be lost and memory access can become very costly if the process is scheduled on the wrong CPU, thereby reducing the performance of the system on a whole. So, there is a need for multicore architectures is to maintain an "affinity" between the tasks and CPUs to avoid redundant memory accesses.

On the other hand, assigning a task to the same CPU may result in load imbalance among the CPUs, resulting in underutilization of the total available computing power. Load balance is the criteria in which all the available CPUs must be utilized

effectively to reduce the overall average time required for task completion. A greater amount of task affinity may often lead to load imbalance among the CPUs.

As one can clearly figure out, load balancing and task affinity are quite contradictory to each other. Hence a trade-off may have to be made in the design of any scheduling mechanism for multicore architectures.

1.1 MOTIVATION

The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling. When more than one process is runnable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm.

The following are some of the goals that must be considered before scheduling the processes.

(i) **Fairness**

Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

(ii) **Policy Enforcement**

The scheduler must make sure that system's policy is enforced. For example, if the local policy is safety then the *safety control processes* must be able to run whenever they want to, even if it means delay in *payroll processes*.

(iii) **Efficiency**

Scheduler should keep the system (or CPU) busy cent percent of the time when possible. If the CPU and all the Input/output devices can be kept running all the time, more work gets done per second than if some components are idle.

(iv) **Response Time**

A scheduler should minimize the response time for interactive user.

(v) **Turnaround**

A scheduler should minimize the time batch users must wait for an output.

(vi) **Throughput**

A scheduler should maximize the number of jobs processed per unit time.

1.2 PRE-EMPTIVE VS NON-PRE EMPTIVE SCHEDULING

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

(i) **Non-Pre-emptive Scheduling**

A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of non-preemptive scheduling

1. In non-preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non-preemptive system, response times are more predictable because incoming high priority jobs can't displace waiting jobs.
3. In non-preemptive scheduling, a scheduler executes jobs in the following two situations.
    1. When a process switches from running state to the waiting state.
    2. When a process terminates.

(ii) **Pre-emptive Scheduling**

A scheduling discipline is pre-emptive if, once a process has been given the CPU can take away.Most of the current scheduling algorithms fail to meet all the above-mentioned aspects thus resulting in inefficient utilization of CPU resources. The motivation is to build a pre-emptive scheduling algorithm which meets most of the ideal qualities of a scheduler. The organization of the report is as follows: In chapter 2, the task schedulers of Linux , Windows ,Solaris are explained in brief. In chapter 3, the designed task scheduling model is explained followed by the algorithm proposed along with an example. Chapter 4 gives the implementation details, tests carried out on algorithm and obtained results. We finally end with the final verdict in chapter 5.

2.STATE OF THE ART

2.1 **THE LINUX SCHEDULER:**

The Linux scheduler in versions prior to 2.6.23 performed its tasks in complexity O(1) by basically just using per-CPU run queues and priority arrays. Version 2.6.23 introduced the completely fair scheduler (CFS). The new scheduler has completely abandoned the notion of different kinds of processes and treats them all equally. The data-structure of a red-black tree is used to align the tasks per their vruntime to use the processor resources until the next context-switch. The left-most leaf node has the least vruntime value and so is granted the processor resources at the scheduling time. The position of a process in the tree is only dependent on the time it has used the CPU resources. However, the total scheduling complexity is increased to O(log n) where n is the number of processes in the run-queue, since at every context-switch, the process has to be reinserted into the red-black tree. The scheduling algorithm itself has not been designed in special consideration of multi-core architectures. Ingo Molnar, the designer of the scheduler admits that the fairness approach can result in increased cache coldness for processes in some situations. However, the red-black trees of CFS are managed per run queue, which assists in cooperation with the Linux load-balancer.

2.2 WINDOWS:

In Windows, scheduling is conducted on threads. The scheduler is priority-based with priorities ranging from 0 to 31. Time slices are allocated to threads in a round-robin fashion. These time slices are assigned to highest priority threads first and only if know thread of a given priority is ready to run at a certain time, lower priority threads may receive the time slice. However, if higher-priority threads become ready to run, the lower priority threads are pre-empted. In addition to the base priority of a thread, Windows dynamically changes the priorities of low-prioritized threads in to ensure responsiveness of the operating system. For example, the thread associated with the foreground window on the Windows desktop receives a priority boost. After such a boost, the thread priority gradually decays back to the base priority. Scheduling on SMP-systems is basically the same, except that Windows keeps the notion of a thread's processor affinity and an ideal processor for a thread. The ideal processor is the processor with for example the highest cache-locality for a certain thread. However, if the ideal processor is not idle at the time of lookup, the thread may just run on another processor. No explicit information is given on scheduling mechanisms especially specific to multi-core architectures, though.[1]

2.3 SOLARIS:

In the Solaris scheduler terminology, processes are called kernel or user-mode threads dependent on the space in which they run. User threads don't only exist in the user space. Whenever a user thread is created, a so-called lightweight process is set up that connects the user thread to a kernel thread. These kernel threads are objected to scheduling. Solaris 10 offers a number of scheduling classes, which may co-exist on one system. The scheduling classes provide an adaptive model for the specific types of applications which are intended to run on top of the operating system. [2] mentions Solaris 10 scheduling classes for:

(a) Standard (timeshare) threads, whose priority may be adapted by the scheduler.

(b) Interactive applications (the currently active window in the graphical user interface).

(c) Fair sharing of system resources (instead of priority-based sharing)

(d) Fixed-priority threads. The priority of threads scheduled with this scheduler does not vary over the scheduling time.

(e) System (kernel) threads.

(f) Real-time threads, with a fixed priority and time share. Threads in this scheduling class may pre-empt system threads.

The scheduling classes for timeshare, fixed priority, and fair sharing are not recommended for simultaneous use on a system, while other combinations of scheduling classes on a set of CPUs are possible. The timeshare and interactive schedulers are quite similar to the old Linux scheduler in their attempt of trying to identify I/O bound processes and providing them with a priority boost. Threads have a fixed time quantum they may use once they get the context and receive a new priority based on whether they fully consume their time quantum and on their waiting time for the context. Fair share scheduling uses a fixed time quantum allocated to processes [3] as a base for scheduling. Different processes compete for quanta on a computing resource and their position in that competition depends on how large the value they have been assigned is in relation to the total quanta number on the computing resource. Solaris explicitly deals with the scenario, that parts of the processor's resources may be shared, as it is likely with typical multi-core processors. There is a kernel abstraction called "processor group" (pg_t), that is built per the actual system topology and represents logical CPUs that share some physical properties.

3.PROPOSED ALGORITHM

The Simple Round Robin Algorithm doesn't consider process priority as a metric and priority based scheduling normally leads to starvation problem. The Proposed Scheduling Algorithm is a hybrid of Round Robin and priority scheduling. It provides a better way of assigning processes to cores so that all the available cores and utilized up to the maximum extent. The focus is to bring certain metrics into usage which would help in this process.

In this scheduling algorithm the processes are selected and pushed into the ready queues based on their priorities. From the ready queue the processes are selected in a particular order of relative priority and passed on to run queues of the most appropriate core. Once the process is placed in the run queue it is then sent to its associated core for its execution.

3.1 TRADITIONAL SCHEDULING ALGORITHMS

3.1.1 ROUND ROBIN BASED SCHEDULING

Round robin is the scheduling algorithm used by the CPU during execution of the process. Round robin is designed specifically for time sharing systems. It is similar to first come first serve scheduling algorithm but the pre-emption is the added functionality to switch between the processes .A small unit of time also known as time slice or quantum is set/defined. The ready queue works like circular queue. All processes in this algorithm are kept in the circular queue also known as ready queue. Each New process is added to the tail of the ready/circular queue .By using this algorithm, CPU makes sure, time slices are assigned to each process in equal portions and in circular order, dealing with all process without any priority. The main advantage of round robin algorithm over first come first serve algorithm is that it is starvation free. Every process will be executed by CPU for fixed interval of time (which is set as time slice). So, in this way no process left waiting for its turn to be executed by the CPU. There is no scope of starvation in this kind of scheduling.The primary disadvantage of Round Robin scheduling is the need of context switching. This process involves a huge overhead to store the state of the process before it is switched. Some of the other disadvantages of Round Robin scheduling include low throughput, higher average waiting time, high response time, high turnaround time and no prioritization.

3.1.2 PRIORITY BASED SCHEDULING

Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler chooses the tasks to work as per the priority. Priority scheduling involves priority assignment to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on a first-come-first-served (FCFS) or round robin basis. Priorities can be either dynamic or static. Static priorities are allocated during creation, whereas dynamic priorities are assigned depending on the behavior of the processes while in the system. Priorities may be defined internally or externally. Internally defined priorities make use of some measurable quantity to calculate the priority of a given process. In contrast, external priorities are defined using criteria beyond the operating system, which can include the significance of the process, the type as well as the sum of resources being utilized for computer use, user preference etc. Priority based scheduling is simple to learn and implement. This scheduling offers a fair advantage to processes with higher priority. This type of scheduling is mostly suitable for applications with varying time and resource requirements. The main disadvantage of priority based scheduling is the starvation of a process. Starvation means the shadowing of a lower priority process by a higher priority process. This leads to indefinite blocking of low priority process known as starvation. Other disadvantage of priority based scheduling is the loss of information in the case of crash.

3.1.3 FIRST-COME-FIRST-SERVE

First-Come-First-Serve algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a non-preemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait. FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long. The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems, but it is often embedded within other schemes.

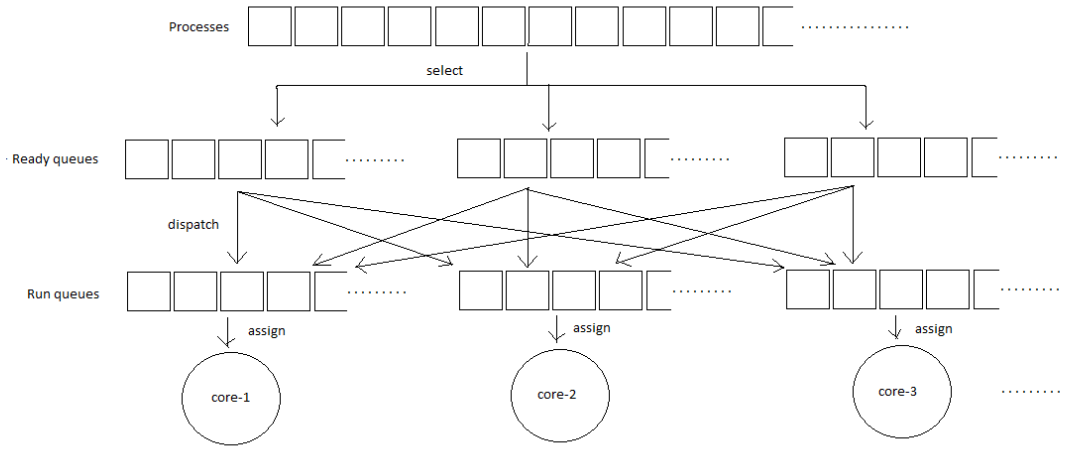
3.2 THE ALGORITHM

3.2.1 ARCHITECTURE

We virtually divide the ready queue into ‘N’ different ready queues depending on the priority range values. We select processes from all these ready queues in a particular order. This concept of maintaining different ready queues for different range of priorities and selecting a process from them provides fairness to all the processes. This is followed by selecting a particular core for the process which are dispatched from the ready queue.

The processes must be executed on cores such that all the cores are utilized effectively. Load balancing and Processor affinity are the primary metrics considered in the design of the proposed scheduling algorithm. These two metrics contradict each other. If we wish to have maximum processor affinity, we need to compromise on load balance. Similarly, to attain maximum load balance, the processes have to be migrated to other cores very frequently, which in turn will hinder the affinity of the processes to the cores.

The proposed algorithm intelligently tries to maximize both processor affinity and core-load balance. The way in which it is explained in greater detail in the next section.



3.2.2 VARIABLES USED

We adopt the definition of virtual runtime of a thread from the Linux CFS scheduler. vruntime is a per-thread variable. It is a member nested within the task_struct. Essentially, vruntime is a measure of the "runtime" of the thread - the amount of time it has spent on the core. Each time a thread is allocated to a new core, vruntime of that thread in that core is set to zero and is modified accordingly based on the time the thread has used the resources of the core.

There are two other factors which we have defined to achieve load balancing. They are OPTIMUM-CORE-LOAD and SCALED-LOAD. Optimum core load gives the optimum value of the utilization of the core. In case of homogeneous cores this value is the ratio of 100 to the number of available cores. This value can be used during scheduling a thread to the core.

Conventional Operating Systems define the load of a core to be the number of threads in its run queue, i.e., run queue length. For any core with scaled computing power P, we define its scaled load, L, to be its run queue length divided by P. In our proposed algorithm SCALED-LOAD of a core is the ratio of length of its assigned run queue to the sum of lengths of the run queues of all the available cores. This value can be used for scheduling process to a core. Once a core reaches its optimal core load value no further threads are pushed into its run queue unless all the other cores are also performing at their optimum level. Otherwise thread migration is inevitable. We attempt to reduce the cost of thread migration by selecting the thread with lowest vruntime for migration. This migration is done intelligently by comparing the goodness value of migration versus conservation of the thread. This helps to minimize the cost of migration as much as possible by retaining the affinity of threads to the core and it also maintains the load balance among the cores.

3.2.3 ALGORITHM STAGES

The entire algorithm is divided into 2 stages.

STAGE - 1

In this stage, we select a process from all the available processes.

Algorithm Schedule-Next-Process( )

1. Classify the process based on their priorities and push the processes into their respective queues. Call these queues as READY-QUEUE(i)

2. Initialize RELATIVE-FREQUENCY (i) for READY-QUEUE(i) for each i.

3. Calculate MAX-CHUNKS and NUM-CHUNKS as
   MAX-CHUNKS=lcm (RELATIVE-FREQUENCY(i))
   NUM-CHUNKS(i) = MAX-CHUNKS/RELATIVE-FREQUENCY(i)

4. Logically split READY-QUEUE(i) into smaller units known as chunks,where SIZE-OF-CHUNK(i) = length (READY-QUEUE(i))/NUM-CHUNKS(i)

5. Select the processes by interleaving the chunks of each queue in the following manner.

```
For all READY-QUEUE(i)
   For j:=1 to SIZE-OF-CHUNK(i)
      Call DispatchProcess( Process [ i , j ] )
```

This algorithm provides fairness for all the processes depending upon the priority. The processes with higher priorities are selected more frequently when compared to the processes with lesser priorities. This eliminates the starvation problem which is prevalent in Priority scheduling.

STAGE - 2

In this stage, we select the most appropriate core for the selected process and take care of load balancing and processor affinity.
In this stage, we select the most appropriate core for the selected process and take care of load balancing and processor affinity.

Algorithm DispatchProcess(TO-BE-SCHEDULED-PROCESS)

1. Initialize

   OPTIMAL-CORE-LOAD=100/NUM-CPUS

   SCALED-LOAD(i)= Length(RunQueue(i))/Length(RunQueue(i))

2. If the TO-BE-SCHEDULED-PROCESS has not been previously run on any core, go to step 2, go to step-3

3. Select the most idlest core from all the available cores and assign the TO-BE-SCHEDULED-PROCESS to that core.Most idlest core is the core having least SCALED-LOAD value. Go to Step 5.

4. Initialize PREVIOUS-CORE = core on which the process has been executed earlier

   4.1 If PREVIOUS-CORE's SCALED-LOAD is less than OPTIMAL-CORE-LOAD, assign the process to PREVIOUS-CORE and go to step 5.Otherwise continue to 4.2 .

   4.2 Let GROUP contain the cores which belong to the same group as PREVIOUS-CORE.If SCALED-LOAD of any core that belongs to GROUP is less than OPTIMAL-CORE-LOAD,assign the process to that core and go to step 5. Otherwise continue to step 4.3

   4.3 Select LEAST-UTILIZED-CORE=core with least SCALED-LOAD among all the cores.

      4.3.1 If the SCALED-LOAD of LEAST-UTILIZED-CORE is less

than 50% of OPTIMAL-CORE-LOAD,then balance the load by carrying out migration as follows.

Assign MIGRATING-PROCESS= process with least vruntime among [Processes on PREVIOUS-CORE,
TO-BE-SCHEDULED- PROCESS].
Migrate MIGRATING-PROCESS to LEAST-UTILIZED-CORE. Go to Step-5

4.3.2 If migration need not be done then, assign TO-BE-SCHEDULED-PROCESS to PREVIOUS-CORE and go to Step5

5. Update SCALED-LOAD values of all cores.

This process of selecting a core intelligently reduces the cost of migration, if it is inevitable.

3.3 PROCESS SELECTION EXPLAINED WITH AN EXAMPLE

Consider the following set of processes at a particular instance of time.

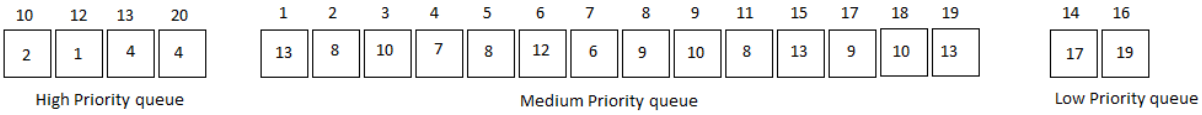
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Priorities	13	8	10	7	8	12	6	9	10	2	8	1	4	17	13	19	9	10	13	4

1. Consider the value of N to be 3 in this case i.e the process are divided into 3 categories based on the priorities.

   i. High Priority Processes (0-5)
   ii. Medium Priority Processes (6-15)
   iii. Low Priority Processes (16-20)

   Hence for our example the processes will be classified as follows

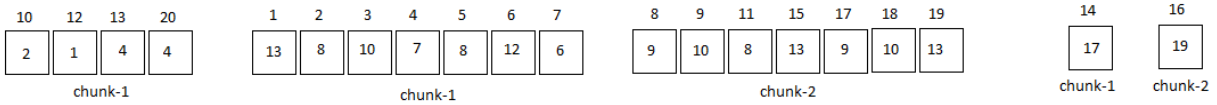
   High priority Processes - 10,12,13,20
   Medium priority Processes - 1,2,3,4,5,6,7,8,9,11,15,17,18,19
   Low priority Processes - 14,16



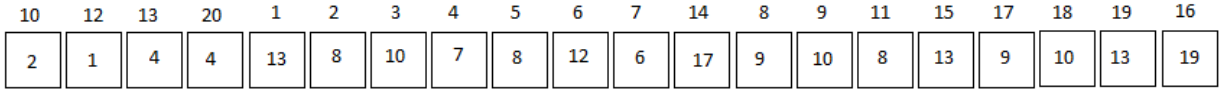
2. Choose the relative frequency of execution for each queue.

RELATIVE-FREQUENCY(High-Priority-Queue) = 4;
RELATIVE-FREQUENCY(Medium-Priority-Queue) = 2;
RELATIVE-FREQUENCY(Low-Priority-Queue) = 1;

3. Calculating MAX-CHUNKS = lcm (4,2,1) = 4
MAX-CHUNKS(High-Priority-Queue) = 4/4 = 1;
MAX-CHUNKS(Medium-Priority-Queue) = 4/2 = 2;
MAX-CHUNKS(Low-Priority-Queue) = 4/1 = 4;



4. Selecting processes, by iterating through the chunks of each queue, as follows.



5. Selecting the most appropriate core for the process. Here in the example we assume that all the processes are new to the queue and hence we select the most underutilized core of all the available cores.

Assume the number of cores are 4 in this case. Once the processes are assigned to the cores in this fashion the run queues of the cores would look like this.

6. Once the processes are present in the run queues of cores they are assigned to their respective cores.

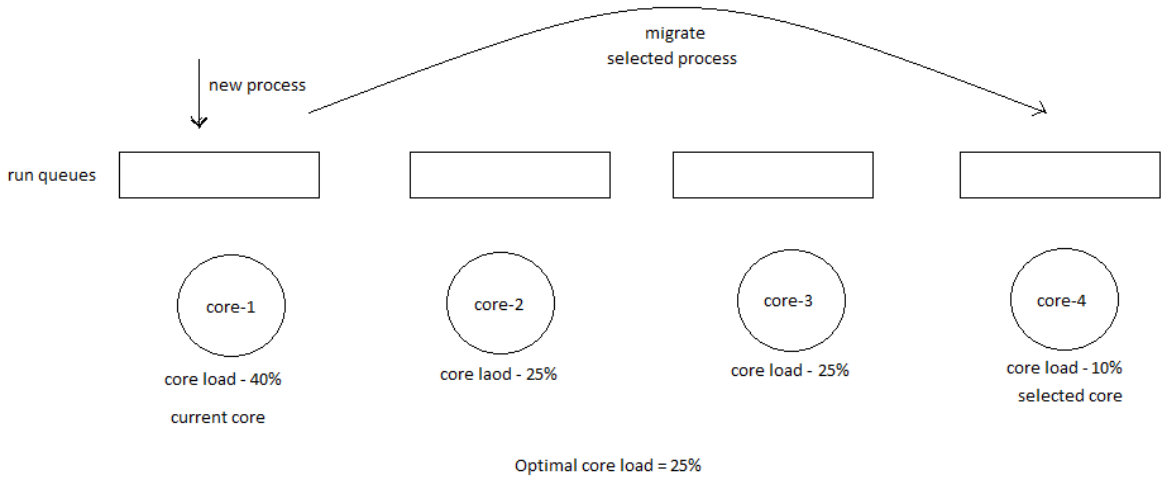
3.4 MIGRATION EXPLAINED WITH AN EXAMPLE

Consider 4 cores each with core load 40%,25%,25%,10%. Since the total number of cores are 4, the optimal core load comes out to be 25% (100/4). Also consider a new process which is partially executed dispatched from ready queue.

1. Since the process is partially executed we try to assign it to core-1 itself as it was earlier executed there. But assigning the process to core-1 leads to improper load balancing. Hence in such scenario migration of the process is taken into consideration.

2. Now, we select the core with least-core-load among all the available cores which is core-4 in this case.

3. Since the core load of core-4 is less than 50% of optimal core load we assign the process with minimum vruntime of all the available processes in run-queue of core-1 and the newly arrived process to core-4.



4.IMPLEMENTATION DETAILS

4.1 DATA STRUCTURES

The following are the data structures used .

(i) Ready Queue is modeled as a series of doubly linked lists where each list has a priority range . The processes in each list are ready to be dispatched into the run queues. This provides fairness to all the processes..

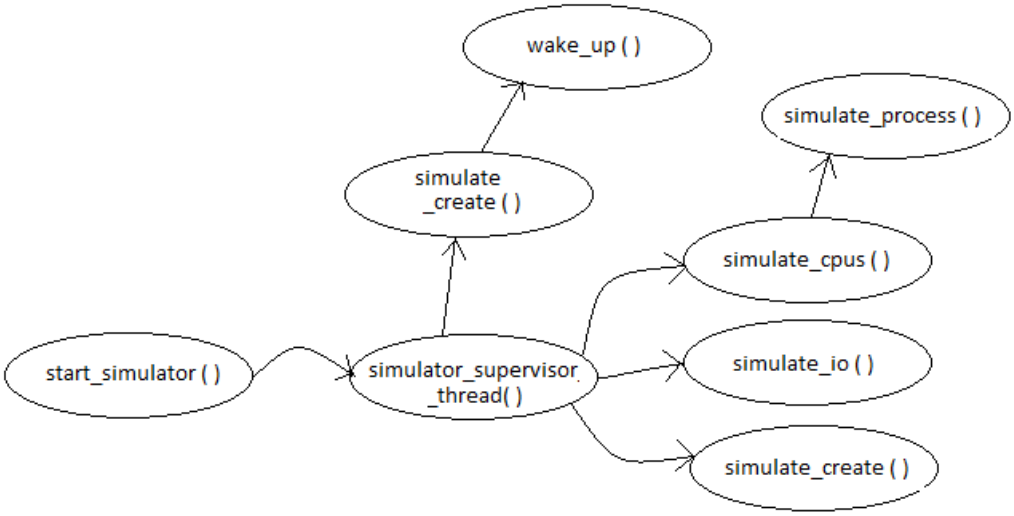
(ii) Run Queues are also implemented using doubly linked list in which contain the processes that are ready to be dispatched. The processes in the run queues are in sorted order based on vruntime. Each core has its own run queue. From the run queue the processes are popped and assigned to the core.

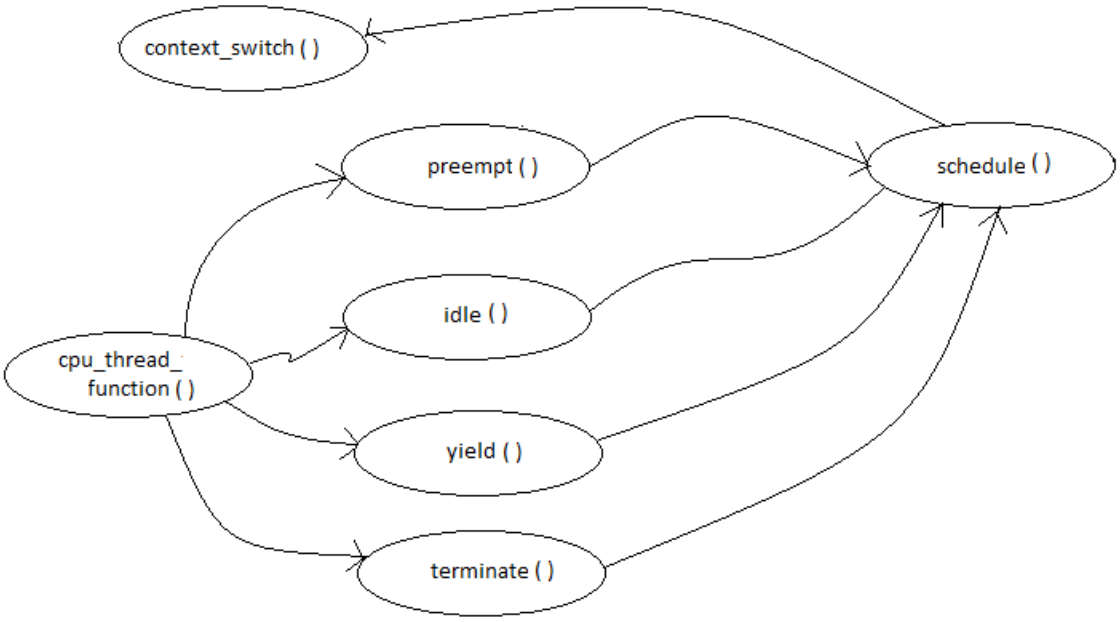
(iii) Each Process is defined as a structure with the following fields.

(a) A process id which is unique for each process.
(b) State of the process(NEW,READY,RUNNING,BLOCKED,TERMINATED).
(c) Priority of the process(0-20).
(d) Current operation performed on the process(CPU,IO,TERMINATED).
(e) Previous CPU-ID which gives the core on which it was executed earlier.
(f) Virtual run time value

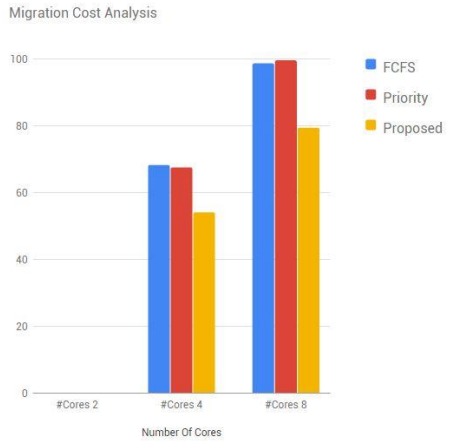
4.2 DESIGN

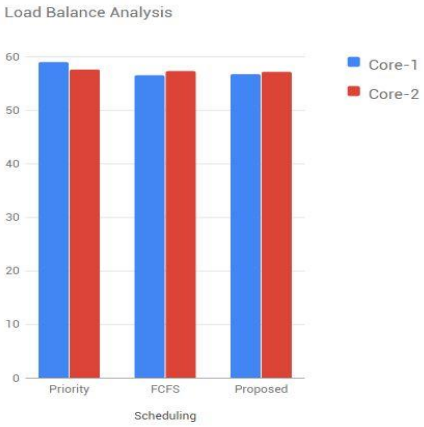
The following figures depict the control flow of the simulator and scheduler

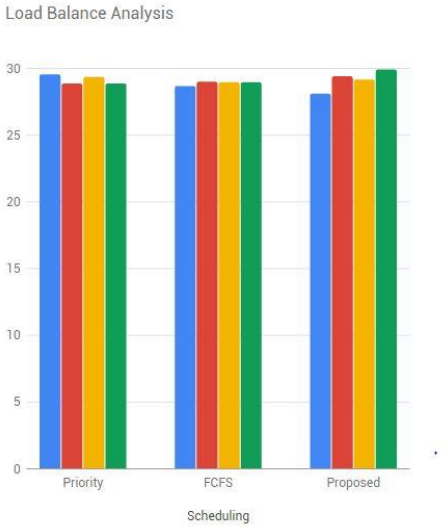


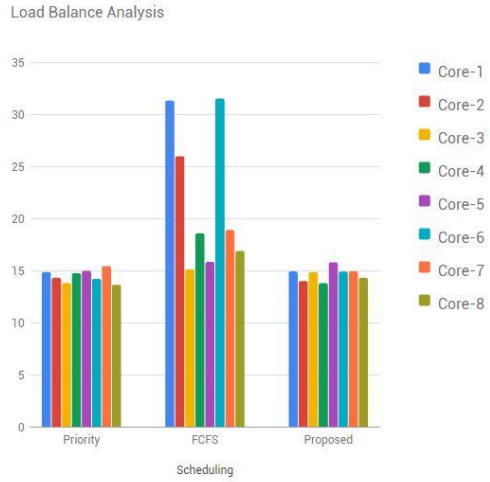


4.3 TESTING AND RESULTS









The bar charts depict the various results obtained for 500 processes. The experiments are carried out for 2,4 and 8 core systems respectively. FCFS and Priority scheduling algorithms are used for comparison. A simulator which allows multithreading is used for simulation purpose.

The migration cost for a 2-core system is 0 as in all h=the three algorithms as both the cores belong to the same group. In the other two cases, migration cost of our algorithm is less than FCFS and priority.

Load balance analysis is done based on the CPU utilization values obtained in the tests. FCFS and priority give a close competition when the number of cores is 2 or 4. But as the number of cores become 8, or even more, the performance of our algorithm is vividly dominant. This improvement may seem to be very small over Priority scheduling, but our algorithm achieves this result by not compromising on the core-load balance, whereas Priority does not provide core-load balance.

When the number of cores is 2, FCFS showed a lower average waiting time of the processes than the other two algorithms. However, with 4 and 8 core systems, our algorithm surpasses the performance of the traditional algorithms.

5.CONCLUSION

Multicore architectures are evolving at a very fast pace, so are the modern operating systems. The modern operating systems are not able to exploit the full potential of the multicore architectures. As a step towards better scheduling support in the operating system for multi-cores with shared memory, our intention was to motivate the design of better schedulers on multicore architectures. It is the need of the hour to develop robust mechanisms that facilitate the maximum utilization of multiple cores. Maximum utilization depends on the proper balance of the core-load among the available cores. Balancing the load often seems like a trivial problem but this balance must not compromise the affinity of processes to the core, thus eliminating the possibility of a straight forward solution. Hence, our attempt was to design a scheduler that provides optimal trade-off between these contradicting factors.

As evident from the results, with an increase in the number of cores, the proposed algorithm achieves significant improvement in both load balance and processor affinity with less migration cost, unlike the traditional scheduling algorithms. In addition to that, it is fair to all the processes, thus tackling the serious problem of starvation. This improvement can be remarkable if the number of cores are further increased

On the other hand, we observed a variety of other effects of scheduler features. One of the more interesting observations was that it is not generally feasible to implement a multicore scheduler if the context switch cost is very high. Also, for a small number of cores, traditional algorithms seem to perform better.

After these initial observations, we shall carry out tests on various other data sets to evaluate the goodness of the scheduler. Concurrently, we plan to introduce task dependency into our scheduler. We find that a final step in this development must integrate this scheduling with the modern operating system schedulers.