*Asian Journal of Computer and Information Systems (ISSN: 2321 – 5658)*
*Volume 06– Issue 04, August 2018*

# Multithread Affinity Scheduling Using a Decision Maker

Shatha Jawad[1], Ronald P. Uhlig[2], Bhaskar Sinha[3,*], Mohammad Amin[4], Pradip Peter Dey[5]

[1] National University
San Diego, California, USA

[2] National University
San Diego, California, USA

[3] National University
San Diego, California, USA

[4] National University
San Diego, California, USA

[5] National University
San Diego, California, USA

[*] *Corresponding author's email: bsinha [AT] nu.edu*

---

**ABSTRACT—** ***In a multiprocessor-multithread Operating System (OS), scheduling has two dimensions. The operating system has to decide which thread to run and which Central Processing Unit (CPU) to run it on. Assume the threads are independent and each thread has a priority, the operating system selects a thread with the highest priority and assigns it to the first free CPU. Usually, each CPU has its private cache. To increase the throughput of the system, it is preferred to use affinity scheduling. The affinity scheduling concept is to make an effort to have a thread run on the same CPU it ran on the last time. The existing affinity scheduling is implemented by using a two-level scheduling algorithm. In this paper a new approach is designed to implement independent multithread scheduling on a multiprocessor system. The design approach uses a decision maker to compute a new priority for each ready thread according to the thread pre-priority and affinity. The results show that by using the new priority, the goal of having affinity is satisfied in addition to taking the pre-priority of the thread in consideration. Also, the design approach reduces the scheduling time because it implements affinity scheduling and priority scheduling by employing a one level scheduling algorithm.***

**Keywords----** Affinity, cache memory, multiprocessor scheduling, operating systems, priority, throughput

---

## 1. INTRODUCTION

Computer systems can be made faster and more reliable by using a multiprocessor. This multiprocessor may share a common Random Access Memory (RAM) and each processor may have its own private cache. To gain the benefits of multiprocessing, all modern operating systems support multithreaded processes. With kernel threads, the kernel is aware of all the threads and can pick and choose among the threads belonging to a process. On a uniprocessor, scheduling is one dimensional with one question: "which thread should be run next" [1,2,3,4]. On a multiprocessor system, the scheduling has two dimensions with two questions: which thread to run and which processor to run it on. The threads may be related to each other or may not be related. As Tanenbaum & Bos [1] say, "In some systems, all of the threads are unrelated, belonging to different processes and having nothing to do with one another". Different processes have unrelated threads, each thread can be scheduled without regard to the other ones. As a thread executes on a processor, it develops "affinity" to this processor by filling its cache with data and instructions during execution. Some multiprocessors take this effect into account and use what is called affinity scheduling [5]. Subramaniam & Eager [6] proposed two algorithms to implement "affinity scheduling". The two proposed algorithms are: (1) dynamic partitioned affinity scheduling and (2) wrapped partitioned affinity scheduling. An experimental study of these algorithms has been done and found them to perform well in this context.

In 1995, Torrellas Tucker, and Gupta [7] did a study on a bus-based multiprocessor executing a variety of workloads and they showed that affinity scheduling reduces the number of cache misses by 7-36%, resulting in execution time improvements of up to 10%. Also, they showed that it is relatively simple to add affinity scheduling to the existing schedulers. So far, a number of research efforts have been carried out on CPU scheduling problems on different applications by using fuzzy logic [8,9,10,11]. All of these efforts tried to improve the performance of the overall system, such as CPU utilization, throughput , turnaround time, waiting time, and response time. But all of these efforts were working on process scheduling not on the thread scheduling.

In this paper a new approach is designed to implement independent multithread scheduling on a multiprocessor system. The design approach uses a decision maker to compute a new priority for each ready thread according to the thread pre-priority and affinity.

The remainder of this paper is organized as follows: Section 2 describes the type of multiprocessor system used in this research. Section 3 demonstrates the multithreading system, and the type of threads considered in this research. A description of the existing scheduling algorithms related to independent threads is also covered in this section. The proposed Multithread Affinity Scheduling by using a Decision Maker and the result are presented in section 4. Section 5 presents the conclusions of this research effort. Sections 6 and 7 are the acknowledgements and references respectively.

## 2. MULTIPROCESSOR SYSTEM

Tightly coupled multiprocessor systems contain multiple CPUs that are connected at the bus level and share full access to a common RAM unit, as shown in Figure 1. [1,12].

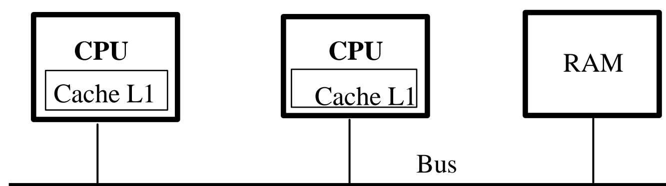


Figure 1: **Bus Based Multiprocessors**

Chip multiprocessors, also known as multi-core computing, involves more than one processor placed on a single chip and is considered as an extreme form of tightly coupled multiprocessing. Typically, each core consists of all of the components of an independent processor, such as registers, Arithmetic Logic Unit (ALU), pipelined hardware, control unit, and Level 1 (L1) instruction and data caches as shown in Figure 2. Level 2 (L2) cache is shared by all ALUs.

There are various approaches for implementing tightly coupled multiprocessor operating systems, but most modern multiprocessor systems use the approaches of splitting the operating system into multiple independent critical regions that do not interact with one another [1].

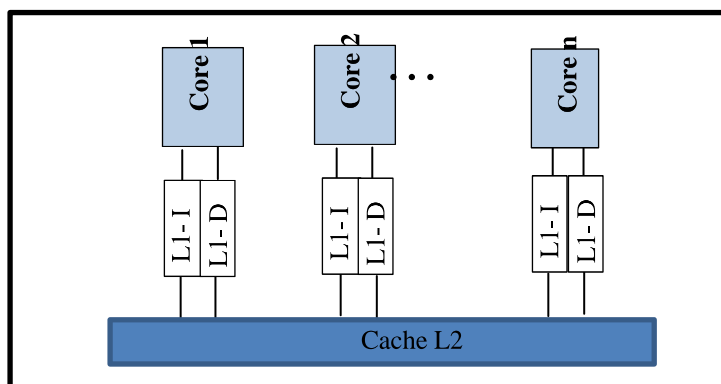


Figure 2: **Chip Multiprocessor**

Each critical region is protected by its own mutex, so that only one CPU at a time can execute it. That means, the CPU scheduler is running in only one CPU.

## 3. SCHEDULING METHOD FOR INDEPENDENT MULTITHREAD

All modern operating systems support multithread processes [1,5]. There are two levels of thread: a user level thread and a kernel level thread. User level thread is visible to the application while kernel level thread is visible to the OS only. The OS kernel is aware of all the threads and can pick and choose among the threads belonging to the process. To get the benefits of a multiprocessor system, multithreading software is used. However, the scheduling algorithm becomes more complex with these systems. On a uniprocessor, scheduling is one dimensional, which means that the only scheduling that the OS worries about is, “which thread should be run next?” On a multiprocessor system, scheduling has two dimensions: the scheduler needs to decide which thread should run and on which CPU. This makes the scheduling more complex. Another issue that should be taken into consideration is that in some systems all of the threads are unrelated, they belong to different processes and operate independent of each other [1].

The simplest scheduling algorithm that is used by operating systems for scheduling independent threads is time sharing. In this method, the OS has a single system-wide data structure for ready threads. Each thread has a priority, so there will be a set of lists, list for thread with priority one, a list for thread with priority two, and so on. The best way to gain the benefit of having multiple threads is to execute them on a tightly coupled multiprocessor system with a single scheduling data structure used by all CPUs. In this way, no CPU will be idle while another CPU is busy all the time. The idea of the time sharing scheduling algorithm is, when one of the CPUs becomes idle, the OS will select a thread from a list with the highest priority and assign it to this CPU. If the list for the highest priority is empty, the operating system selects a thread from the next priority list and so on. The problem with this scheduling algorithm is, when thread x is working for long time on CPU y, CPU y’s cache will be full of thread x’s blocks. If thread x gets to run again soon, it may perform better if it is run s on CPU y, because most or all of its blocks are still in the cache of CPU y. Some multiprocessor systems use affinity scheduling [1]. The main idea of affinity scheduling uses the idea of a time sharing scheduling algorithm and tries to assign the ready thread to the same CPU that was used by this thread last time. The existing affinity scheduling is implemented by using a two-level scheduling algorithm [5]. In this algorithm, each CPU gets its own collection of threads. When a thread becomes ready, it is assigned to a CPU which has a lower load and the thread will stay on the list of this CPU. The assignment of the thread to the CPU is considered the top level of the algorithm. The actual scheduling of the threads is the bottom level of the algorithm and it is done by each CPU separately, using any type of scheduling algorithm, such as priority scheduling.

## 4. PROPOSED SCHEDULING ALGORITHM

In this research a new approach is designed to implement independent multithread scheduling on a tightly coupled multiprocessor system. The new approach is designed to improve the existing time sharing scheduling algorithm and to reduce the problem which is related to the cache memory as explained in the previous section. In this new approach, the OS has a single system wide data structure for ready threads. Each thread has its pre-priority. Every time a thread is assigned to a CPU, the OS registers the name of the CPU that is used by each thread in a specific table called the Cache Affinity Table. In this approach, when one of CPUs is idle, a new priority will be calculated, by using a decision maker, for each ready thread by using its pre-priority and the period of time that the thread was running on that CPU. The operating system uses the new priority to select a thread with the highest priority and assigns it to this idle CPU.

The proposed algorithm has been simulated on different kinds of systems and with different scenarios, some of them are explained below.

### 4.1 Scenario 1

Each thread has a pre-priority, which ranges between 0 and 2. The thread with priority 2 has the highest priority. The OS registers in the cache affinity table the information about the names of CPUs that have been used by each thread during the last two running periods of this thread in each CPU. According to the information about the threads running during the last two running periods and the pre-priority, the designed decision maker algorithm which is built by using fuzzy logic, calculates a new priority for each ready thread. And according to the new priority, the OS selects a thread with highest new priority and assigns it to the idle CPU. Table 1 shows the new priority for all situations that any thread may have. Affinity is represented as a number. 2 means, the thread used the idle CPU in its last two running periods, 1 means, the thread used the idle CPU in its last running period, and 0 means, this thread did not use this CPU before.

Table 1: **Thread's new priority with respect to 3 levels of thread's pre-priority and Affinity number**

| Thread's pre-priority | Affinity | Thread's new priority |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 1.683209 |
| 2 | 0 | 1.359619 |
| 1 | 2 | 1.778382 |
| 1 | 1 | 1.439837 |
| 1 | 0 | 1.275323 |
| 0 | 2 | 1.574439 |
| 0 | 1 | 1.019714 |
| 0 | 0 | 0.583277 |

Figure 3, shows the new priority value with respect to the pre-priority. The result shows, for example, that a thread with priority 2 (the highest priority) and its affinity is 0 or 1, now has a new priority less than a thread with priority 1 but has affinity
2. That means, the proposed algorithm, increased the pre-priority for a thread which has affinity 2 and in this way the blocks which are found in the cache of this idle CPU and are related to this thread can be used again without losing time to read them from the RAM. Also, the proposed algorithm still takes into consideration the pre-priority.

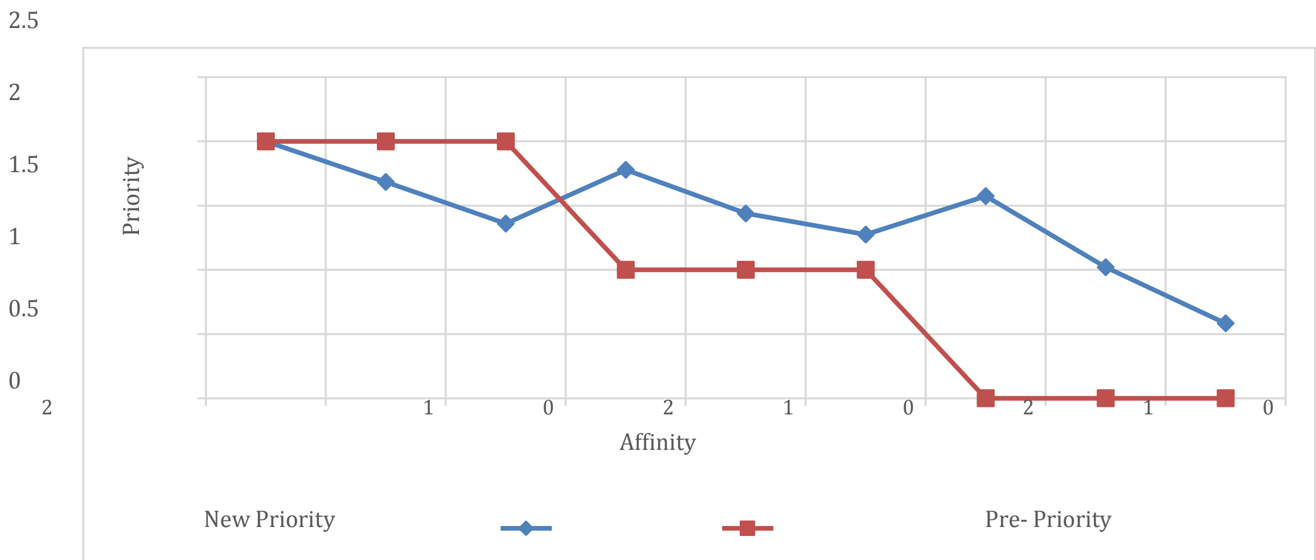


Figure 3: **The new priority compared with pre-priority by using scenario 1 that has 3 levels of pre-priority and 3 levels of Affinity.**

### 4.2 Scenario 2

Each thread has a pre-priority, which ranges between 0 and 2. The thread with priority 2 has the highest priority. The OS registers in the cache affinity table the information about the name of CPUs that have been used by each thread during the last running period. According to the information about the thread running during the last running period and the pre-priority, the designed decision maker algorithm which is built by using fuzzy logic, calculates a new priority for each ready thread. And according to the new priority, the OS selects a thread with the highest new priority and assigns it to an idle CPU. Table 2 shows the new priority for all situations that any thread may have. Affinity represented as a number. 1 means, the thread used the idle CPU in its last running period and 0 means, this thread didn't use this CPU before.

Table 2: **Thread's new priority with respect to 3 levels of thread's pre-priority and Affinity number**

| Thread's pre-priority | Affinity | Thread's new priority |
|---|---|---|
| 2 | 1 | 2 |
| 2 | 0 | 1.7456 |
| 1 | 1 | 1.9632 |
| 1 | 0 | 1.2368 |
| 0 | 1 | 1.4544 |
| 0 | 0 | 1.2 |

Figure 4, shows the new priority value with respect to the pre-priority. The result shows, for example, that a thread with pre-priority 1 and affinity 1, gets a new priority larger than the new priority for a pre-priority 2 and affinity 0. At the same time it's new priority is larger than the new priority of thread which had pre-priority 0 and affinity 1. That means, the proposed algorithm, increased the pre-priority for a thread which has affinity 1, and in this way the blocks which are found in the cache of this idle CPU and are related to this thread can be used again without losing time to read them from the RAM. Also, the proposed algorithm still takes in consideration the pre-priority.

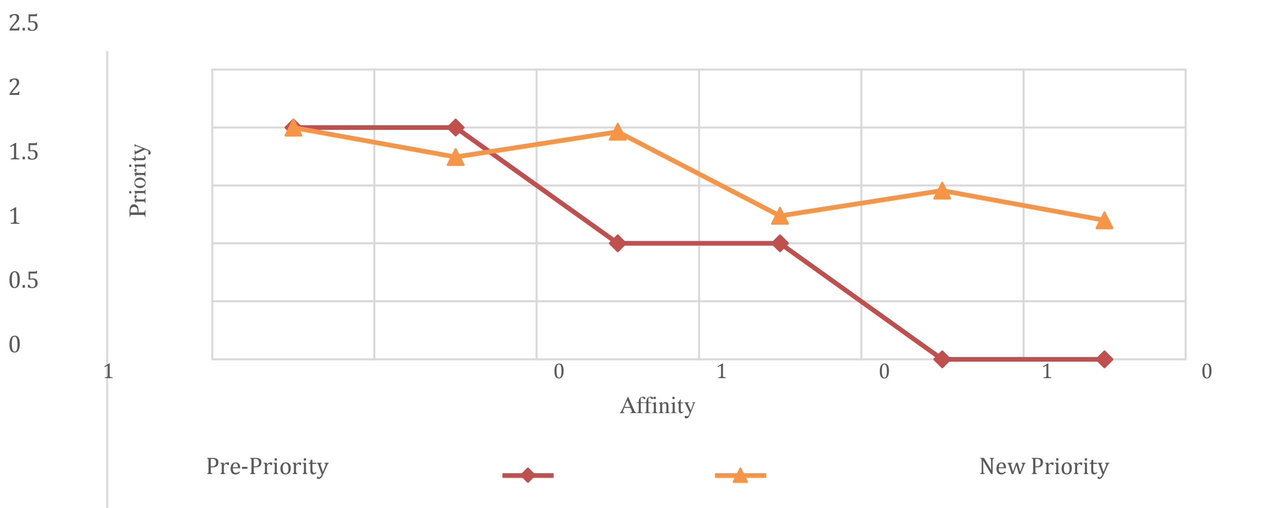


Figure 4: **The new priority compared with pre-priority by using scenario 2 that has 3 levels of pre-priority and 2 levels of Affinity.**

## 4.3 Scenario 3

Scenario 3 is similar to the first scenario in terms of number of affinity but with 7 levels of pre-priority which is ranging between 0 and 6. The thread with pre-priority 6 has the highest priority and the thread which has pre-priority 0, has the lowest pre-priority. Table 3 shows the new priorities that have been calculated by using the designed decision maker algorithm and it covers all situations that any thread may have. Figure 5, shows that the results of the algorithm's calculation of the new priority for each thread from the pre-priority level and the affinity number.

Table 3: **Thread's new priority with respect to 7 levels of thread's pre-priority and Affinity number**

| Thread's pre-priority | Affinity | Thread's new priority |
|---|---|---|
| 6 | 2 | 5.696677154 |
| 6 | 1 | 5.447117567 |
| 6 | 0 | 4.726736269 |
| 5 | 2 | 5.487970949 |
| 5 | 1 | 5.016795279 |
| 5 | 0 | 4.463912846 |
| 4 | 2 | 4.869723105 |
| 4 | 1 | 4.255560599 |
| 4 | 0 | 3.915115751 |
| 3 | 2 | 4.800272356 |
| 3 | 1 | 4.085338175 |
| 3 | 0 | 3.370403995 |
| 2 | 2 | 4.255560599 |
| 2 | 1 | 3.915115751 |
| 2 | 0 | 3.300953246 |
| 1 | 2 | 3.706763504 |
| 1 | 1 | 3.153881071 |
| 1 | 0 | 2.682705402 |
| 0 | 2 | 3.443940082 |
| 0 | 1 | 2.723558783 |

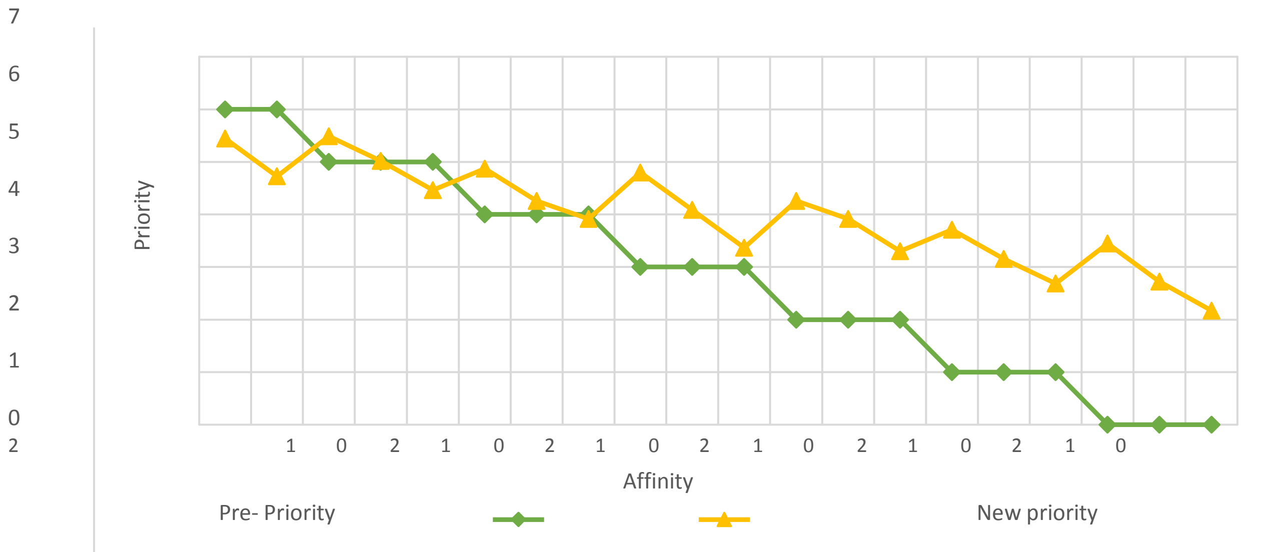


Figure 5: **The new priority compared with pre-priority by using scenario 3 that has 7 levels of pre-priority and 3 levels of Affinity.**

### 4.4 Scenario 4

Each thread has a pre-priority, which ranges between 0 and 6. Each affinity has 4 levels and each level is represented as a number. Number 3 means, the thread used the idle CPU in its last two running periods, number 2 means, the thread used the idle CPU in its last running period, number 1 means, the thread used the idle CPU in the period before the last running period, and number 0 means, this thread didn't use this CPU before. Figure 6, shows that the proposed algorithm calculated the new priority for each thread as expected with respect to the pre-priority level and the affinity number.

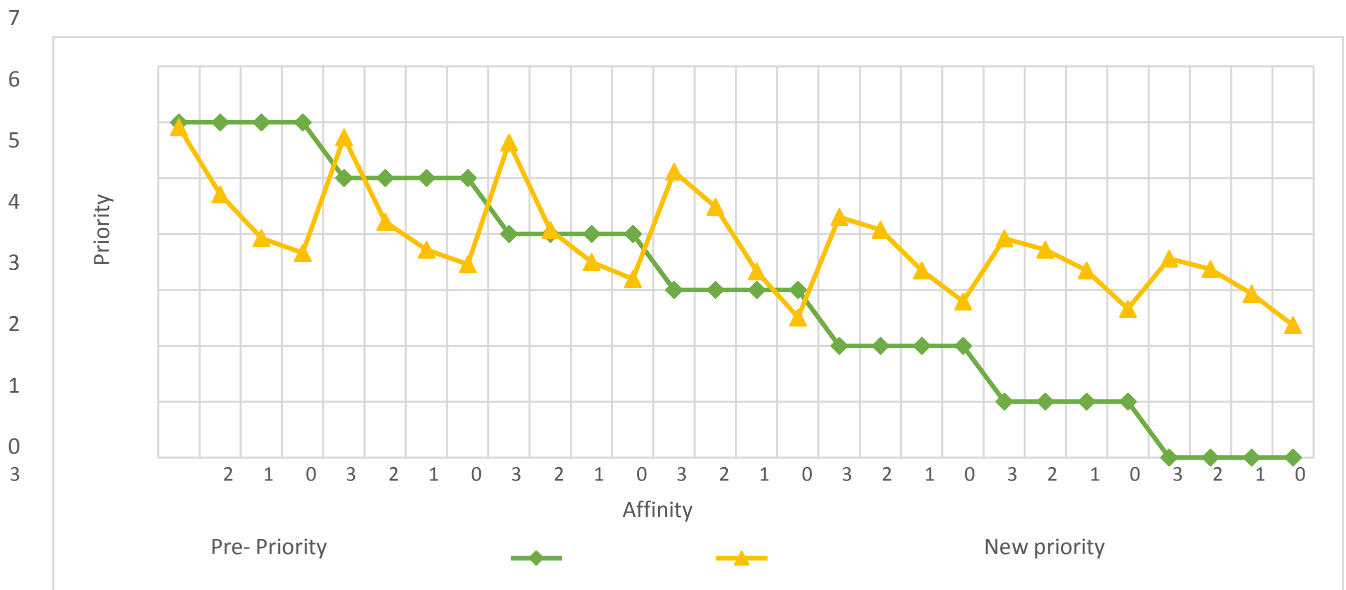


Figure 6: **The new priority compared with pre-priority by using scenario 4 that has 7 levels of pre-priority and 4 levels of Affinity.**

## 5. CONCLUSION

The main benefit of OS threads is to speed up the execution time of the processes. And to get the benefit of threads it is better to execute them on a multiprocessor system. Implementing threads in the OS kernel, makes the operating system have a thread table that keeps track of all threads in the system. Also, the OS is responsible for thread scheduling on multiprocessors. The relation between threads define the scheduling method that should be used by the OS. In this paper, independent threads are considered. Time sharing scheduling is the most used scheduling method for independent threads. The problem of the existing method (time sharing scheduling) is, it cannot get the benefit of having the internal cache on each CPU. There is already a solution for this problem which is known as affinity scheduling, but this scheduling method is implemented by using two levels and each CPU will be assigned a list of threads to keep each thread working on the same CPU. The proposed algorithm in this research improves on the time sharing scheduling method while retaining the benefit of having an internal cache on each CPU.

The proposed approach, will keep one list for the kernel, and the threads can run on any idle CPU. The OS keeps track of all information about each thread and when a CPU becomes idle, the OS uses a decision maker to compute a new priority for each thread by taking into consideration the pre-priority of each thread and the historical information of each thread with respect to this idle CPU. The results show that by using the new priority, the goal of having affinity is satisfied, in addition to taking the pre-priority of the thread into consideration. Also, the design approach reduces the scheduling time because it implements affinity scheduling and priority scheduling by employing a one level scheduling algorithm. Suggested future work is to implement the proposed approach in a real OS and compare the results with the simulated system results.

## 6. ACKNOWLEDGEMENTS

As a group of National University employees, the authors also thank and gratefully acknowledge the help and support received from the administration, staff, and faculty members at National University, School of Engineering and Computing, during the continuing research on this subject and the preparation of this paper.

## 7. REFERENCES

[1]. Tanenbaum, Andrew S. and Bos, Herbert, Modern Operating Systems. Pearson, 2015

[2]. Silberschatz, Abraham and Galvin, Peter B., Operating System Concepts, Addison-Wesley Longman, 1993.

[3]. Naghibzadeh, M.. Operating System Concepts and Techniques, iUniverse, 2011.

[4]. Peterson, James L., Operating System Concepts, Addison-Wesley Longman, 1985.

[5]. Vaswani, R. and Zahorjan, J., “The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors”. Proceedings of the Thirteenth Symposium on Operating Systems Principles, pages 26-40, October 1991

[6]. Subramaniam, S. and Eager, D.L., Affinity Scheduling of Unbalanced Workloads, Supercomputing '94. Proceedings, 1994.

[7]. Torrellas, J., Tucker, A., Gupta. A., “Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors”. Journal of Parallel and Distributed Computing. Volume 24, Issue 2, 1995, Pages 139-151.

[8]. Bashir, Alam, Doja, M. N. and Biswas, R., “Improving the Performance of Fair Share Scheduling Algorithm using Fuzzy Logic”, Proceedings of the International Conference on Advances in Computing, Communication and Control, 2009.

[9]. Kandel, Abraham, Zhang, Yan-Qing and Henne, Marlow, “On Use Fuzzy Logic Technology in Operating Systems”, Fuzzy Sets and Systems, Vol. 99, No. 3, 1988, pp: 241- 251.

[10]. Lim, Sungsoo and Sung-Bae, “Intelligent OS Process Scheduling Using Fuzzy Inference with User Models”, IEA/AIE 2007, LNAI 4570, pp. 725–734, 2007.

[11]. Jawad, Shatha and Al-Aubidy, Kasim. “Design and Evaluation of a Fuzzy-Based CPU Scheduling Algorithm”, CCIS 70, p. 45 ff, Information Processing and Management, Springer, 2010.

[12]. Stallings, William. “Computer Organization and Architecture Designing for Performance”, 10 edition, 2016.