International Journal of Science and Advanced Innovative Research
ISSN: 2536-7315 (Print) 2536-7323 (Online)
Volume 3, Number 2, June 2018
http://www.casirmediapublishing.com

MODELS FOR HANDLING MULTI-DIMENSIONAL PROCESSES IN CENTRAL PROCESSING UNIT USING TASK-BASED PRIORITY QUEUES DATA STRUCTURES

A.H. Eneh*, A. J. Jimoh & U. C. Arinze
Department of Computer Science
Faculty of Physical Sciences
University of Nigeria, Nsukka-South East, Nigeria
Email: agozieh.eneh@unn.edu.ng, jimoh.johnson@unn.edu.ng
***Corresponding author**

ABSTRACT

This paper reviews different process scheduling criteria; algorithms; properties; objectives and underlying dynamic data structures that optimize process scheduling such as concurrent priority queues (PQs). PQs are known for handling multi-dimensional processes in central processing units (CPUs) by using tasked-based PQs such as - SkipQueue, a highly distributed PQ-based on a simple modification of Pugh's concurrent SkipList algorithm. SkipLists – search structures based on hierarchically ordered linked-lists. PQs are fundamental in the design of modern multiprocessor algorithms, with many applications ranging from numerical algorithms through discrete event simulation and expert systems design and implementation. For such algorithms to be used in the CPU they must possess certain inherent properties such as: fairness; predictability; throughput maximization and enforcement of priorities respectively. Scheduling priority criteria such as – CPU utilization, throughput; turnaround; waiting and response times are also critical for such systems. Several attempts have been made to address the design of concurrent priority queue algorithms for multi-dimensional processes and small scale machines. Nevertheless, the problem of obtaining optimality in performance is yet to be resolved. This work attempts to address the problem. Results and findings from our algorithm simulation on MATLAB environment indicate that to search a list of N items, O (logN) level lists are traversed, and a constant number of items is traversed per level, making the expected overall complexity of an Insert or Delete operation on a PQ O(logN). This indicates an improvement in performance threshold, as other algorithms exhibited O (N) complexity for similar search times.

Keywords: *multiprocessors, concurrent data structures, priority queues.*

INTRODUCTION

The most central concept in any operating system is the process – an abstraction of a running program [1]. They support concurrency, even though there is only one CPU available, turn a single CPU into multiple virtual CPUs et cetera. Modern computers perform several tasks at the same time. A process can exist in three states viz: *running, ready* and *blocked* [1] as shown in figure 1. A process is in running mode when it is actually using the CPU; ready when it is runnable, but temporarily stopped to let another process run; blocked when unable to run until some external event happen respectively [1]. The process model illustrated below is implemented as process tables or process control blocks (PCBs), with one entry per process in the operating system (OS).

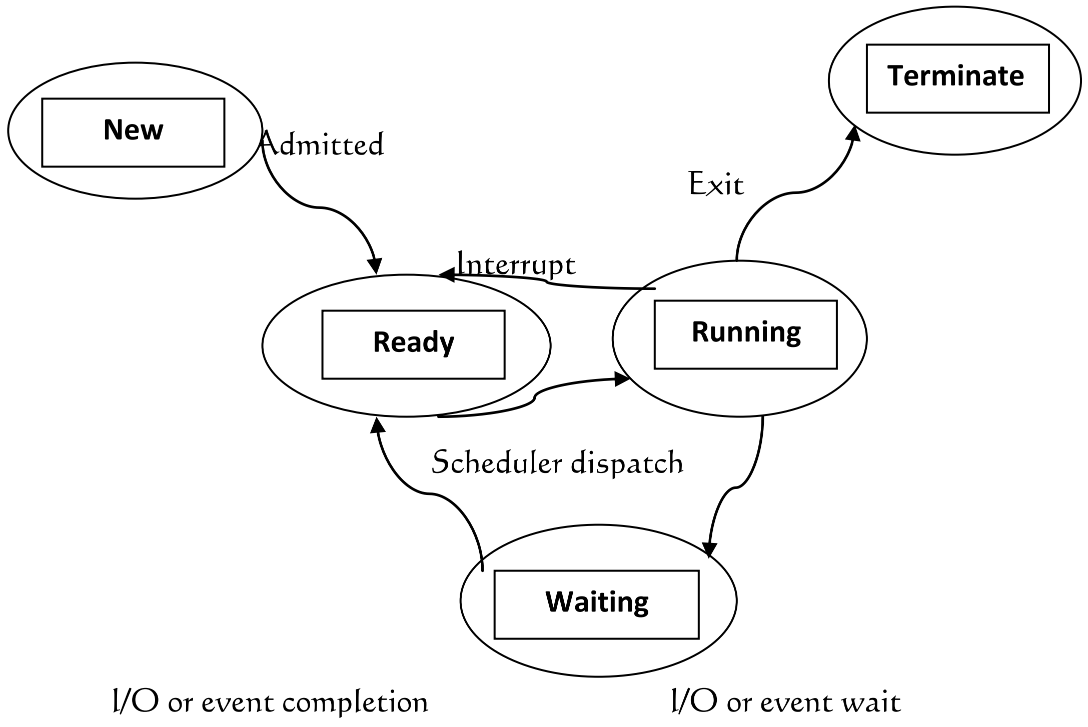

Figure 1: Process state transition diagram [1]

For instance in a Web server, Requests for web pages comes in from diverse sources. When a request is received, the server checks to see if the page needed is in the cache. If it is, it is sent back to the requesting computer, if it is not, a disk request is initiated to fetch it. However, form CPU perspective this takes a long time. While waiting for the disk request to complete, many more requests may come in. If there are multiple disks present, some or all of them may be triggered off to other disks long before the first request is satisfied [1]. In another scenario, consider a personal computer (PC) user. When the system is booted, many processes are started in the background, often unknown to the user. For instance, a process may be started up to wait for incoming e-mails. Another process may run on behalf of the anti-virus program to check periodically if any new virus definitions are available. In addition, explicit user processes may be running, printing files and burning a CD-ROM, all while the user is surfing the web [1]. All this activity has to be managed, and a multi-programming system supporting multiple processes finds a better application in this instances. From the foregoing scenario, it is imperative for a system to be designed to model and control this concurrency. This is where scheduling algorithms such as task-based priority queues data structures, processes (and especially threads) can help. In other to address the problem of designing robust and scalable concurrent priority queues for handling multi-dimensional processes in central processing unit (CPU) using tasked-based priority queues, this paper adopts an alternative approach: base the design of concurrent priority queues on the SkipQueue data structures of Pugh [2], [3], rather than on the popular Heap data structures found throughout the literature [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. The rest of the work is organised as follows: section 2 deals with background of study, section 3 reviews of related works, section 4 methodology, section 5 results, section 6 discussion of results and section 7 conclusion of the paper. Though there is a wide body of literature addressing the design of concurrent priority queue algorithms for multi-dimensional processes and small scale machines, the problem of obtaining optimality in performance is yet to be addressed.

BACKGROUND

Priority queues are of fundamental importance in the design of modern multiprocessor algorithms. They have many classical applications ranging from numerical algorithms, through discrete event simulation, and expert system design. A priority queue is an abstract data type that allows *n* asynchronous processes to each perform one of two operations: an *Insert* of an item with a given priority, and a *Delete-min* operation that returns the item of highest priority in the queue. We are interested in "general" queues, ones that have an unlimited range of priorities, where between any two priority values there may be an unbounded number of other priorities. Such queues are found in numerical algorithms and expert systems [18], [19] and differ from the bounded priority queues used in operating systems, where the small set of possible priorities is known in advance. How does one go about constructing a concurrent priority queue allowing arbitrary priorities? Since for most reasonable size queues, logarithmic search time easily dominates linear one, the literature on concurrent priority queues consists mostly of algorithms based on two paradigms: search trees [20], [21] and heaps [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. Empirical evidence collected in recent years [10; 17; 39] shows that heap-based structures tend to outperform search tree structures. This is probably due to a collection of factors, among them that heaps do not need to be locked in order to be "rebalanced," and that Insert operations on a heap can proceed from bottom to root, thus minimizing contention along their concurrent traversal paths.

When there are several processes in the ready queue as shown in figure 2, the algorithm which decides the order of execution of those processes is called a scheduler and the underlying algorithm that controls it is known as a scheduling algorithm as shown In figure 1. Among the various well known CPU scheduling algorithms are: First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling (PS). Scheduling algorithm aide in actualizing the goals of maximum utilization of the CPU. The scheduling algorithm decides the mode and nature of the execution process.

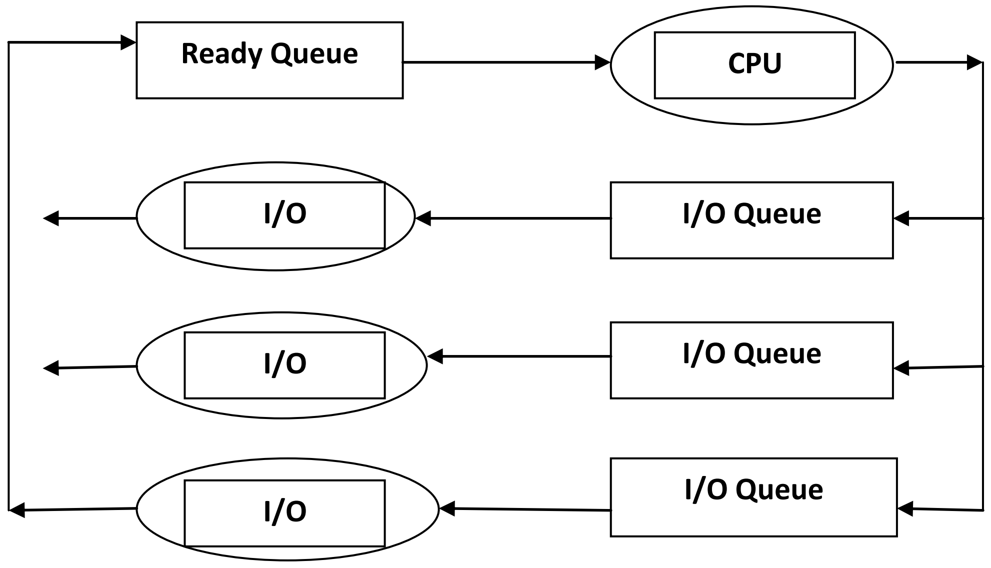

Figure 2: Processor and I/O scheduling

Job scheduling is one of the vital roles the CPU performs in attending to the task of processing data for easy accessibility by the client [22]. The aim of the scheduling algorithm is to ensure that jobs or processes are attended to at the appropriate time by sharing execution time among all processes [22]. The First Come First-Serve (FCFS) scheduling algorithm is the simplest CPU non pre-emptive (cooperative) scheduling algorithm. It is fair in the formal sense of fairness but it is unfair in the sense that long jobs make short jobs and unimportant jobs wait endlessly. It assigns priority to the processes in the order they request the processor (first-in-first-out). Processes are dispatched according to their arrival time on the ready queue [23]. For instance, consider a hypothetical scenario with three (3) processes shown in table 1.

Table 1: Process model for FCFS scheduling algorithm

Process	Burst Time	Arrival
P1	24	0
P2	3	0
P3	3	0

Using a Gantt chart in the order: P1, P2, P3 it can be shown that the average waiting time and turnaround time for the three processes: P1, P2, and P3 are: 17s and 27s respectively.

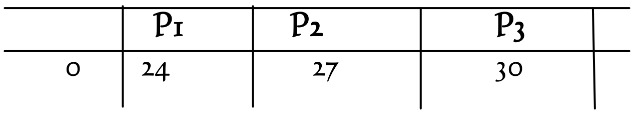

Figure 3: Gantt chart for processes P1, P2 and P3

Average Waiting Time (AWT): P1+P2+P3 = (0+24+27)/3 = 17s
Average Turnaround Time (ATT): P1+P2+P3 (24+27+30)/3 = 27s

However, if the order of the processes is reversed as: P2, P3, P1 there is a substantial decrease in the ATT to 13s.

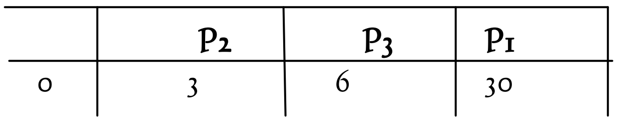

Figure 4: Gantt chart to illustrate FCFS scheduling algorithm
Average Turnaround Time (ATT): P2+P3+P1 (3+6+30)/3 = 13s

Similarly, the shortest-job-first (SJF) process scheduling algorithm, also referred to as shortest-process-next (SPN) is a non pre-emptive scheme in which the CPU is assigned to the process with smallest CPU burst. If the CPU bursts of two processes are the same, FCFS scheduling is used to resolve the queues. When a job comes in, insertion is done on the ready queue based on its length and its gives the minimum average waiting time and minimum average turnaround time for a given set of processes [24]. A pre-emptive SJF algorithm will pre-empt the currently executing process, whereas a non pre-emptive SJF algorithm will not pre-empt the currently running process to finish its CPU burst [23], [25]. The SJF algorithm favours jobs (or processes) with shorter execution time at the expense of processes with longer ones. Among the major problems with SJF is that it requires precise understanding of how long a job or process will run, and this information is not readily available. A typical scenario is shown below in table 2.

Table 2: Process model for SJF scheduling algorithm

Process	Burst Time	Arrival
P1	6	0
P2	8	0
P3	7	0
P4	3	0

Using a Gantt chart in the order: P1, P2, P3, P4 it can be shown that the AWT for the four processes: P1, P2, P3 and P4 is 7s with SJF of 10.25s respectively.

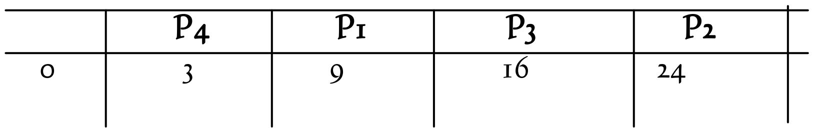

Figure 5: Gantt chart for processes P1, P2, P3 and P4

Average Waiting Time (AWT): P1+P2+P3+P4 = (0+3+16+9)/3 = 7s. In another type of priority scheduling known as Round Robin (RR) Scheduling processes are dispatched based on first-in-first-out (FIFO) order. It is essentially the pre-emptive version of FIFO. But the proviso is that they are given the CPU only for a limited amount of time called a time-slice or quantum. This approach is designed for time-sharing systems. In each time slice (quantum) the CPU executes the current process only up to the end of the time slice. The CPU scheduler goes round the ready queue as shown in figure 5, allocating the CPU to each process for a time interval. If a process does not complete before its CPU-time expires, the CPU is pre-empted and given to the next process waiting in a queue. The pre-empted process is then placed at the back of the ready list [23], [24] and [26].

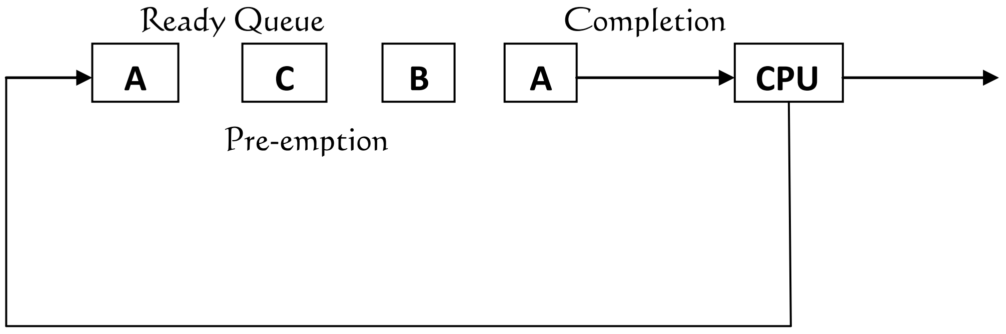

Figure 6: Illustrating Round Robin scheduling algorithm

For instance, consider a hypothetical scenario with three (3) processes shown in table 1.

Table 3: Process model for FCFS scheduling algorithm

Process	Burst Time	Arrival
P1	24	0
P2	3	0
P3	3	0

Using a Gantt chart in the order: P1, P2, P3... it can be shown that the average waiting time and turnaround time for the three processes: P1, P2, and P3 are: 5.66s and 17s respectively.

	P1	P2	P3	P1	. . .	P1	
0	4	7	10	14	26	30	

Figure 7: Gantt chart for processes P1, P2 and P3 for RR scheduling algorithm

Average Waiting Time (AWT): (0+4+7+(10-4))/3 = 5.66s
With FCFS: (0+24+27)/3 = 17s

RELATED WORKS

There is a wide body of literature addressing the design of concurrent priority queue algorithms for multi-dimensional processes and small scale machines. In this section an overview is provided to analyze the different scheduling mechanisms which have been used for predictable allocation of CPU so as to track different research trends and improvements made in this research area so far. In one of this attempts, [25] in Performance Assessment of some CPU scheduling Algorithms, compared different scheduling algorithms on the basis of waiting time and turnaround time. This paper proceeded to give a brief overview and deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. They evaluated the "short Remaining Time First scheduling algorithm. In this scheduling algorithm the ready queue is organized according to the burst times of the processes. The routines which require small amount of time to execute are placed in front of the queue. This algorithm is also a pre-emptive scheduling algorithm. The process with smallest burst time is selected and assigned to CPU. If a process with lower burst time as compared to process which is running comes in the queue, then the process which is running is pre-empted and the new process with small burst time starts its execution then the process is terminated and removed from the waiting process list. In other to ensure CPU fairness, [26] proposed an algorithm that allocates the CPU to every process in Round Robin (RR) fashion for an initial time quantum (say n units). After completing first cycle, it doubles the initial time quantum (2n units) and allocates the CPU to the processes in SJF format.

In other to resolve the CPU scheduling conundrum [27] in "CPU scheduling: A comparative study", discuss about scheduling policies of CPU for computer systems. A number of problems were solved to find the appropriate among them. Therefore, based on performance, the shortest job first (SJF) algorithm is suggested for the CPU scheduling problems to decrease either the average waiting time or average turnaround time. Similarly, the first-come-first-serve (FCFS) algorithm is suggested for the CPU scheduling problems to reduce either the average CPU utilization or average throughput. In [28], they proposed an algorithm that allocates the CPU to processes in RR fashion. After executing each process for one time quantum, it checks the remaining burst time of the currently running process for the remaining burst time, else it moves the process to the tail of the ready queue. In [29], they made an improvement to the Longest Job First (LJF) CPU scheduling algorithm. It works by sorting the process in descending order of their CPU burst times and then it determines a threshold known as combined Weighted Average (CWA) which is the average of the processes. This is used to categorize the processes into long and short processes.

Furthermore, [24] proposed an algorithm that focuses on an additional improvement Round Robin (AAIRR) CPU scheduling. The algorithm reduces the no of context switch, waiting time and turnaround time drastically compared to the improved Round Robin (IRR) scheduling algorithm and simple Round Robin scheduling algorithm.

METHODOLOGY

An analytical approach using a combination of asymptotic algorithm analysis; 2-D plots rendering and graphs on MATHLAB and SPSS platforms will be adopted in this approach. The rationale for the approach is informed by their accurate, systematic and methodical manner of rendering results with visualization effects, which makes its easy for deductions to be made easily. In this paper we propose concurrent priority queues based on the highly distributed Skip List and Skip Queue data structures of Pugh [2]. Skip Lists are search data structures based on hierarchically ordered linked-lists, with a probabilistic guarantee of being balanced. The basic idea behind SkipLists is to keep elements in an ordered list, but have each record in the list be part of up to a logarithmic number of sub-lists. These sub-lists play the same role as the levels of a binary search structure, having twice the number of items as one goes down from one level to the next. To search a list of N items, O (log$N$) level lists are traversed, and a constant number of items are traversed per level, making the expected overall complexity of an Insert or Delete operation on a Skip List O (log$N$) [3]. Similarly, *Skip Queue* is a highly distributed priority queue based on a simple modification of Pugh's concurrent SkipList algorithm [2]. Inserts in the Skip Queue proceed down the levels as in [2]. For Delete-min, multiple "minimal" elements are to be handed out concurrently. This means that one must coordinate the requests, with minimal contention and bottlenecking, even though Deletemins are interleaved with Insert operations. The choice of Skip Queues data structures over prior heap and tree data structures for our simulation is informed by a number of factors such as: distributed locking; probabilistic balancing, hence there is no need for a major synchronized "rebalancing" operation; Delete-min operations are evenly distributed over the data structure, hence minimizing locking contention and the avoidance of need to pre-allocate all memory, since the structure is not placed in an array.

RESULTS

Computer simulation run of the SkipList and SkipQueue data structures using the algorithms and pseudo-codes shown in algorithms 1, 2, and 3 respectively were performed on Toshiba PC with Intel Core i5 64-bit processor architecture, with Intel CPU running at 2.50 GHZ, 8 GB RAM memory on Windows 8.1 platform. MATLAB software version 7.11.0 R2010b was used for algorithm performance evaluation and simulation so as to test the performance and optimality of the algorithms. The charts and 2-D plots obtained are shown in figures 8 – 12 respectively.

Algorithm 1: Code for auxiliary procedure getLock

```
node_t * getLock(node_t * node1, key_t key, int level)
{
    node2 = node1->next[level]
    while (node2->key < key) { // Look for the node with the largest
        node1 = node2 // key smaller than the key we're
        node2 = node1->next[level] // searching for.
    }
    lock(node1, level) // Lock the node.
    node2 = node1->next[level]
    while (node2->key < key) { // Something changed before locking.
        unlock(node1, level) // Unlock node.
        node1 = node2 // Get the next node in the queue.
        lock(node1, level) // Lock it.
        node2 = node1->next[level]
    }
    return node1
}

int randomLevel()
{
    int l = 1
    while (random() < p)
        l++
    if (l > queue->maxLevel)
5           return queue->maxLevel
6        else
7      return l
      }
```

Algorithm 2: Code for inserting a node into the queue

```
                 int Insert(key_t key, value_t value)
                {
1                node1 = queue->Head // Search from the queue head
2               for (i = queue->max Level; i > 0; i--) { // search all levels.
3              node2 = node1->next[i]
4             while (node2->key > key) { // Find the place at this
5            node1 = node2 // level in which to
6           node2 = node2->next[i] // Insert the new node.
7          }
8       savedNodes[i] = node1 // Save the location that was found.
9      }
10       node1 = getLock(node1, key, 1)
11    node2 = node1->next[i]
12   if (node2->key == key) {
13  node2->value = value;
14   unlock (node1, 1)
15  return UPDATED
16       }
17       level = random Level () // Generate the level of the new node.
18       new Node = Create Node (level, key, value)
19    new Node->time Stamp = MAX_TIME; // Initialize the time stamp.
20   lock(new Node, NODE) // Lock the entire node.
21  for (i = 1; i <= level; i++) {
22  if (i != 1) // level 1 is already locked
23  node1 = getLock(savedNodes[i], key, i)
24  new Node->next[i] = node1->next[i] // insert the new node
25  node1->next[i] = new Node // into the queue.
26  unlock(node1, i)
27  }
28  unlock(newNode, NODE) // Release the lock on entire node.
29  new Node->time Stamp = get Time(); // Set the time stamp.
30  return INSERTED // The insertion was successful.
```

Algorithm 3: Code for deleting the smallest node from the queue

```
                    int Delete_Min (value_t * value))
                    {
```

```
            time = get Time (); // Mark the time at which the search starts.
    node1 = queue->head->next [1] // Start search at start of first level.
    while (node1 != queue->tail) { // Search until end of queue.
            if (node1->time Stamp < time) { // Ignore all nodes that were
            // inserted after search began.
            marked = SWAP (node1->deleted, TRUE) // Swap the flag value.
            if (marked == FALSE) // An unmarked node was found,
            break // so end the search.
            node1 = node1->next [1] // Move to next node.
            }
        }
        if (node1 != queue->tail) { // We found an unmarked node
       *value = node1->value // save its value
      key = node1->key // and its key. 11 } 12 else
    return EMPTY // No node was found in the queue.
     node1 = queue->head // Start the search from the head.
    for (i = queue->max Level; i > 0; i--) // Search all levels.
    node2 = node1->next[i]
    while (node2->key > key) { // Find the place at this
    node1 = node2 // level in which the node
    node2 = node2->next[i] // with the key is located.
    } 22 savedNodes[i] = node1 // Save the location that was found.
          }
         node2 = node1
         while (node2->key != key) // Make sure we have a pointer
        node2 = node2->next[1] // to the node with the key.
      lock(node2, NODE) // Lock the entire node to be deleted.
     for (i = node2->level; i > 0; i--) {
    node1 = get Lock(savedNodes[i], key, i) // Lock this level on
    lock (node2, i) // the node to be deleted and node before it.
    node1->next[i] = node2->next[i] // Remove the node from the
    node2->next[i] = node1 // queue.
    unlock(node2, i) // Release the locks on this level at
  unlock(node1, i) // the deleted node and node before it.
 }
unlock(node2, NODE) // Release the lock on entire node.
Put On Garbage List(node2) // Put the node on the garbage list.
return DELETE // Delete was successful.
```

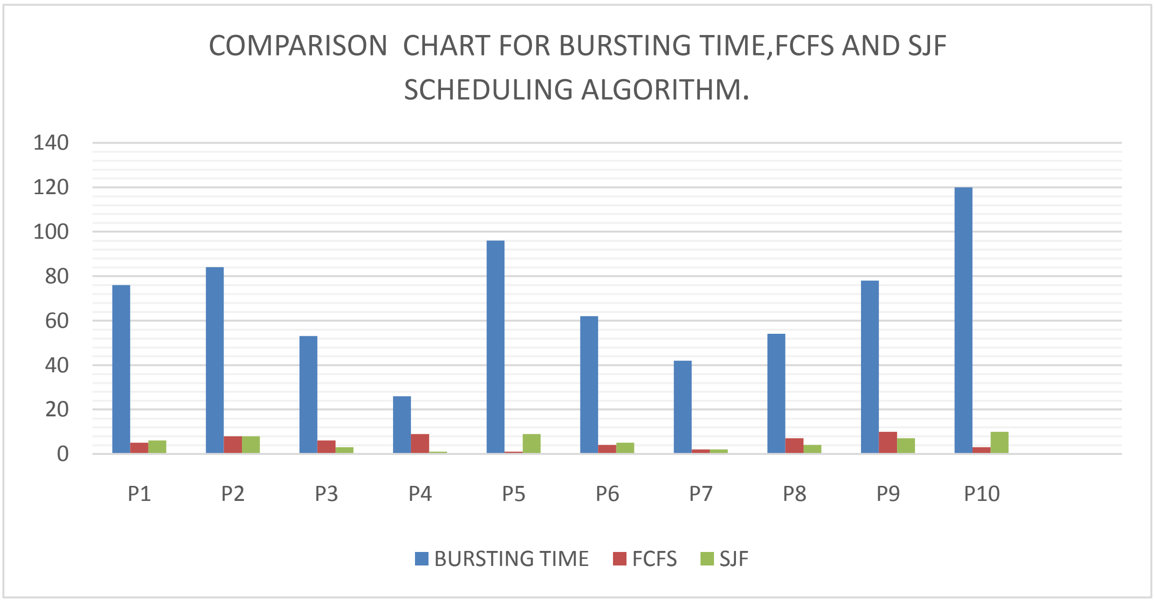

Figure 8: Comparison chart of bursting time for FCFS & SJF algorithms

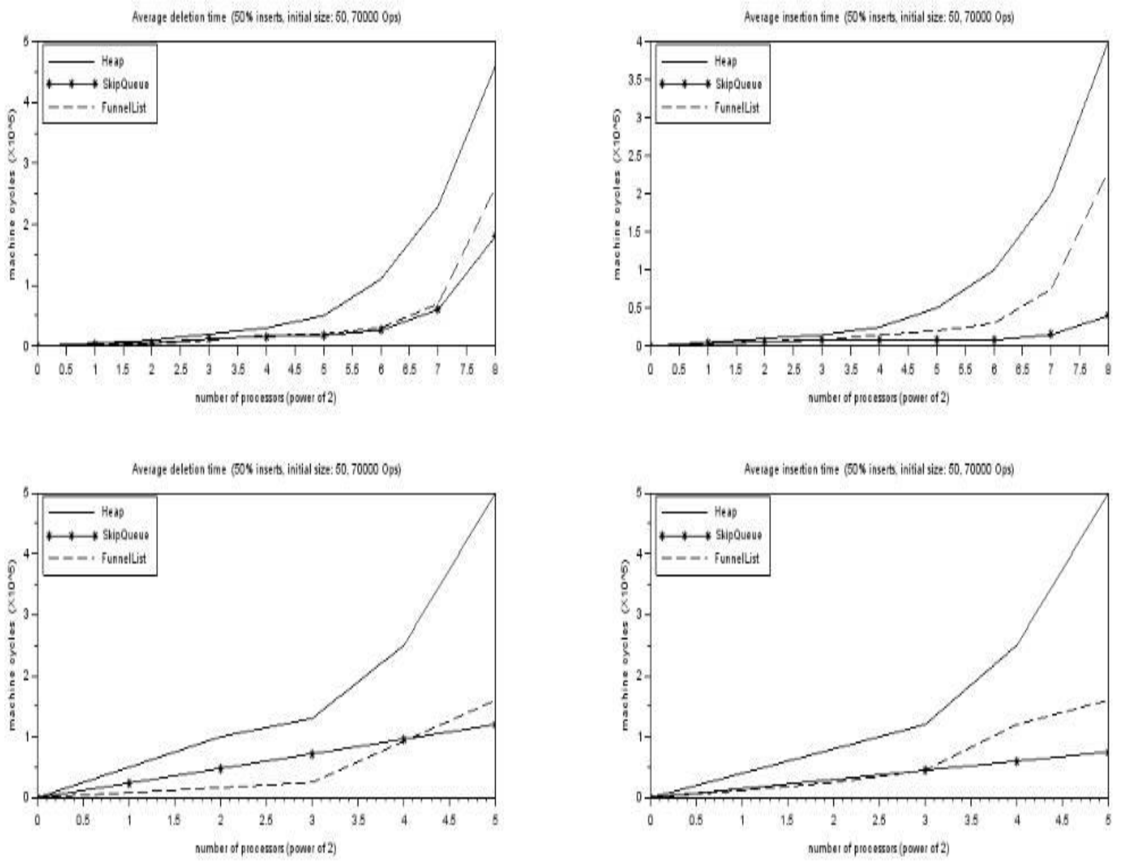

Figure 9: The small structure benchmark

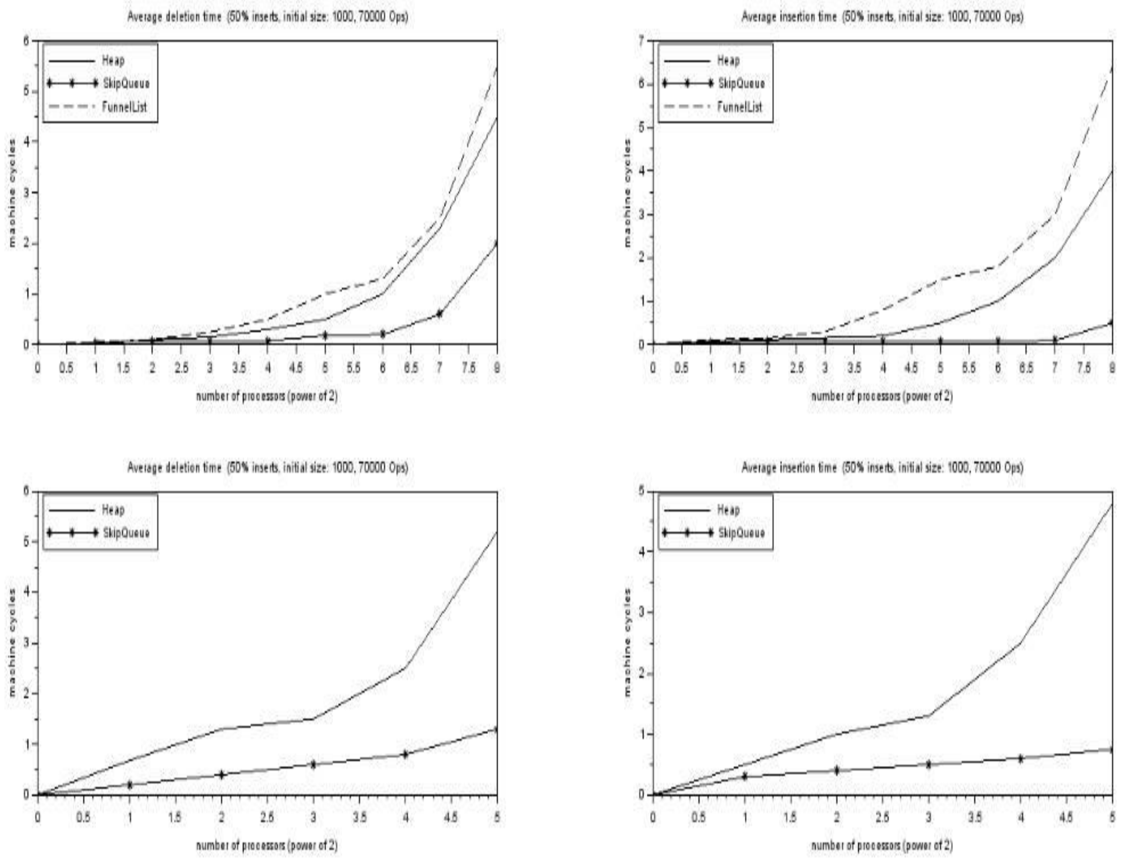

Figure 10: The large structure benchmark

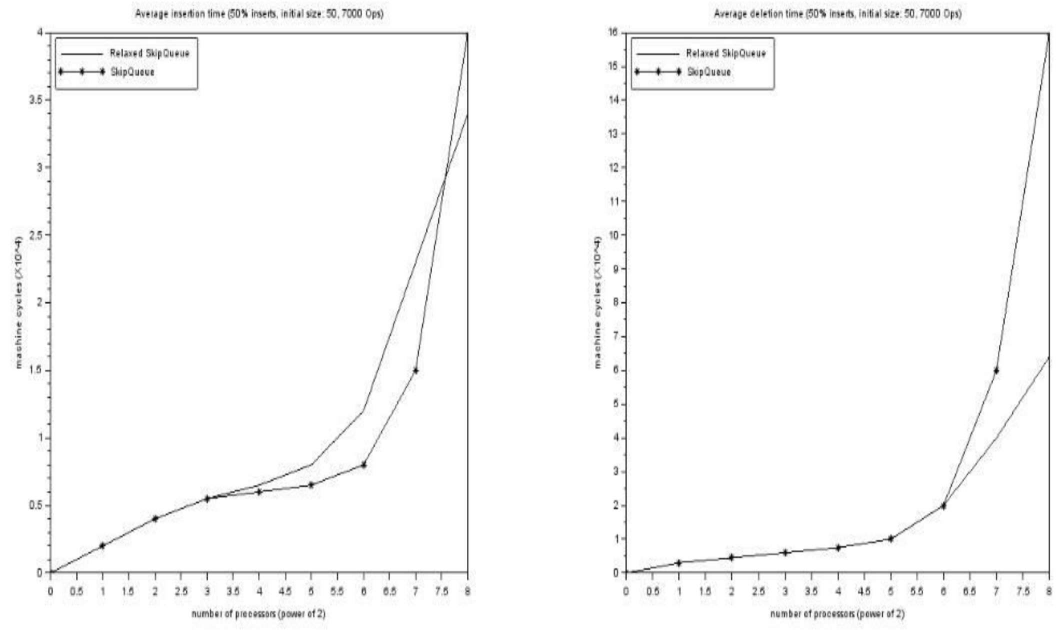

Figure 11: SkipQueue vs. Relaxed SkipQueue for small structure

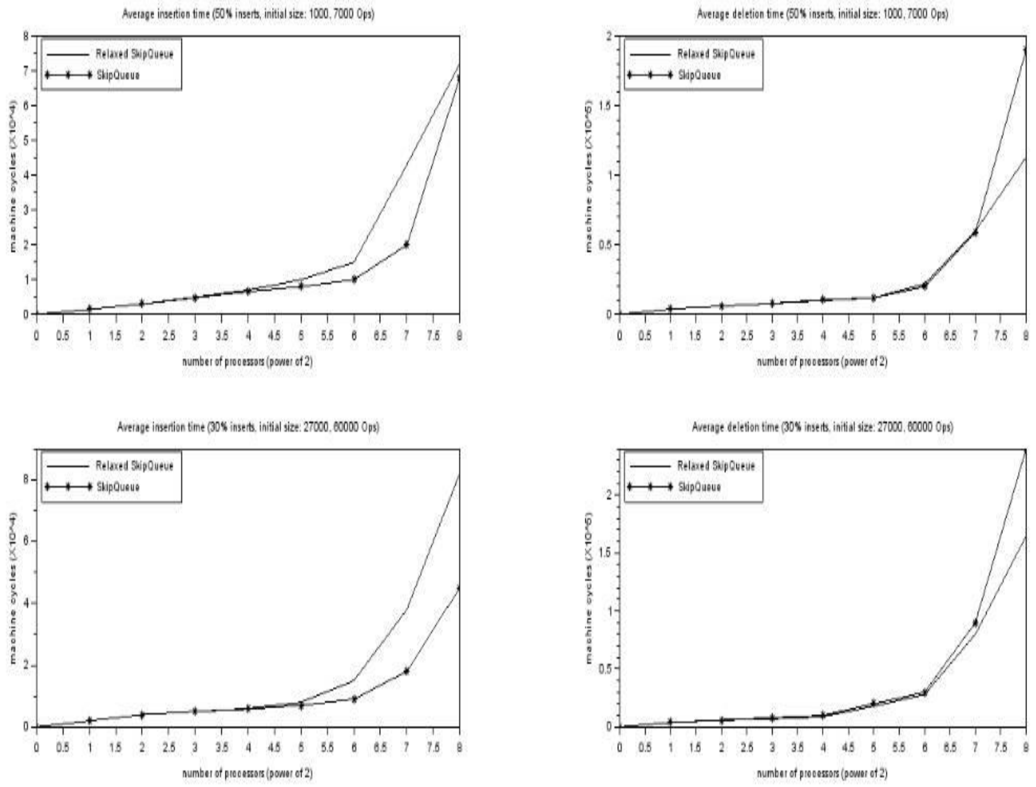

Figure 12 Skip Queue vs. Relaxed SkipQueue for large structure

DISCUSSION of RESULTS

Our implementation is based on the SkipList implementation in [31]. The code for the auxiliary procedures and for the Insert is identical, and our changes are in the Delete_Min procedure which uses the Delete operation for SkipLists provided in [31]. We note that for compatibility with earlier C-based SkipList implementations, the interface of the actual implemented code differs slightly from the specification of Section 4.2. An inserted item in the Insert procedure is actually a pair of key and value), where comparisons are done on the key and the value is just the stored item. The Insert procedure returns a success code. The Delete min operation returns the deleted item's value in a designated memory location, and returns a notification of success or a possible EMPTY SkipQueue.

The Comparison results in figure 8 shows that process two (P2) with the bursting time of eighty-four (84) has eight (8) ns timeslot for both First-come-first-serve and shortest-job-first CPU scheduling algorithms. In this case the processor can choose either to attend to any of the process or job of shortest job First-come-first-serve based on their arrival. The chart also shows that the comparison of process between FCFS and SJF which indicates that shortest job have comparative advantage over First-come-First-serve.

CONCLUSION

Different scheduling algorithms have their merits and demerits. However, this research work recommended that in solving multi-dimensional processes both the First-Come-First-Serve and Shortest Job First should be considered to enhance effective and efficient completion of job tasks.

REFERENCES

[1] Tanenbaum, A.S. (2008). *Modern Operating Systems, Pearson Education Inc.,* Upper Saddle River, New Jersey, p. 83.

[2] Pugh, W. (1990. *Skip Lists: A Probabilistic Alternative to Balanced Trees.* In Communications of the ACM, vol. 33, no. 6, p.668-676.

[3] Lotan, I., and Shavit, N. (2000). Skip List-Based Concurrent Priority Queues, proceedings of the first International Parallel and Distributed Processing Symposium, Cancun, Mexico.

[4] Ayani, R. (1991*). Lr-algorithm: concurrent operations on priority queues.* In Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing pp. 22-25.

[5] Biswas, J., and Browne, J.C. Simultaneous Update of Priority Structures. In Proceedings of the 1987 International Conference on Parallel Processing, August 1987, pp. 124-131.

[6] Sajal, K. D., Pinotti, M.C., Sarkar, F. (1996). Distributed Priority Queues on Hypercube Architectures. In International Conference on Distributed Computing Systems (ICDCS), pp. 620-628.

[7] Deo, N., and Prasad, S. (1992). *Parallel Heap: An Optimal Parallel Priority Queue.* In The Journal of Supercomputing, Vol. 6, pp. 87-98.

[8] Huang, Q. (1991). An Evaluation of Concurrent Priority Queue Algorithms. Technical Report, Massachusetts Institute of Technology, MIT-LCS/MIT/LCS/TR-497.

[9] Hunt, G.C., Michael, M.M., Parthasarathy, S., and Scott, M.L. (1996). An Efficient Algorithm for Concurrent Priority Queue Heaps. In Information Processing Letters, vol. 60, no. 3, pp. 151-157.

[10] Luchetti, C., and Pinotti, M.C. (1993). *Some comments on building heaps in parallel.* In Information Processing Letters, vol.47, no. 3, pp.145-148, 14.

[11] Mans, B. (1998). *Portable Distributed Priority Queues with MPI.* In Concurrency: Practice and Experience, vol. 10, no. 3, pp. 175-198.

[12] Prasad, S.K., and Sawant, S.I. (1995). *Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors.* In Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP 95).

[13] Ranade, A., Cheng, S., Deprit, E., Jones, J., and Shih, S. (1994). Parallelism and Locality in Priority Queues. In IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas.

[14] Rao, V.N. and Kumar, V. (1988). *Concurrent access of priority queues.* IEEE Transactions on Computers, vol. 37, p. 1657-1665.

[15] Sanders, P. (1995). *Fast priority queues for parallel branch-and-bound.* In Workshop on Algorithms for Irregularly Structured Problems, no. 980 in LNCS, Springer, pp. 379-393, Lyon.

[16] Sanders, P. (1998). *Randomized Priority Queues for Fast Parallel Access.* In Journal of Parallel and Distributed Computing, vol. 49, no. 1, pp. 86 - 97.

[17] Yan, Y., and Zhang, X. (1998). *Lock Bypassing: An Efficient Algorithm for Concurrently Accessing Priority Heaps.* ACM Journal of Experimental Algorithmics, vol. 3,. http://www.jea.acm.org/1998/YanLock

[18] Mohan, J. (1983). *Experience with Two Parallel Programs Solving the Travelling Salesman Problem.* In Proceedings of the 1983 International Conference on Parallel Processing, pp. 191-193.

[19] Quinn, M. J. and Deo, N. (1984). *Parallel Graph Algorithms.* In ACM Computing Surveys, Vol. 16, No. 3, pp. 319-348.

[20] Boyar, J., Fagerberg, R., and Larsen, K. S. (1994). *Chromatic Priority Queues.* Technical Report, Department of Mathematics and Computer Science, Odense University, pp. 1994-15.

[21] Johnson, T. A. (1991). *Highly Concurrent Priority Queue Based on the B-link Tree.* Technical Report, University of Florida, pp. 91-007.

[22] Thakur, P. & Mahajan, M. (2017). *Different Scheduling Algorithm in Cloud Computing: A Survey.* International Journal of Modern Computer Science, Vol. 5, No. 1.

[23] Abdulrazaq, A., Aliyu, S., Mustapha, A.M., and Abdullahi, S.E. (2014). *An Additional Improvement in Round Robin (AAIRR) CPU Scheduling Algorithm.* International Journal of Advanced Research in Computer Science and Software Engineering. Vol. 4, Issue 2.

[24] Joshi, R., and Tyagi, S. (2015). *Enhanced Priority Scheduling Algorithm to Minimize Process Starvation.* International Journal of Emerging Technology and Advanced Engineering, Vol. 2, Issue 10.

[25] Oyetunji, E. O., and Oluleye, A.E. (2009). *Performance Assessment of Some CPU Scheduling Algorithms.* Research Journal of Information Technology, pp. 22-26.

[26] Ajit, S., Priyanka, G. And Sahil, B. (2010). *An Optimized Round Robin Scheduling.* International Journal on Computer Science and Engineering (IJCSE), Vol. 2, No. 7, pp. 2382-2385.

[27] Patell, J., and Sopanki, A.K. (2011). *CPU Scheduling: A Comprehensive Study.* Proceedings of the 5th National Conference. IndiaCom.

[28] Manish, K.M., Kadir, A. *An Improved Round Robin CPU Scheduling Algorithm.* Journal of Global Research in Computer Science, Vol. 3, No. 6, pp. 64-66.

[29] Abdullahi, I., and Junaidu, S.B. *Empirical Framework to Migrate Problems in Longer Job First Scheduling Algorithm (LJF+CBT).* International Journal of Computer Applications, Vol. 75, No. 14, pp.9-14.