AN EFFICIENT MULTIPROCESSOR MEMORY MANAGEMENT FRAMEWORK USING MULTI-AGENTS

K.R.Sarath Chandran[1],Aiswarya S.[2], Kavitha R.[3] and Suganthi Krishnavathi T.[4]

[1]Department of Computer Science and Engineering, SSN College Of Engineering, Chennai, India
sarathchandran@ssn.edu.in
[2]Department of Computer Science and Engineering, SSN College Of Engineering, Chennai, India
aishu29691@gmail.com
[3]Department of Computer Science and Engineering, SSN College Of Engineering, Chennai, India
kavitha.rising@gmail.com
[4]Department of Computer Science and Engineering, SSN College Of Engineering, Chennai, India
suganthikrishnavathi@gmail.com

ABSTRACT

The current generation computer users call for fast addressal of their requests. Multi-processing and Multicore architectures have been adopted for dynamic assignment of a program to two or more processors working in parallel. The coordinated processing of programs face challenges in areas of process scheduling, memory allocation and effective use of level caches due to dynamic variation of resource availed and used. The time efficiency can be improved by using multi-agents for the dynamic decision making process.The multiple interacting intelligent agents serve as autonomous decision makers based on the current set of varying inputs. Multiple agents performing various tasks can be used to enhance memory management. The credibility, novelty and necessity of the various agents are to be ascertained. This work offers a framework to employ multiple agents in a multiprocessing environment to enhance the efficiency of process handling mechanisms. A comparison is to be made between the conventional memory management and the proposed environment with localized multi-agents.

KEYWORDS

Multi agents, Dynamic Decision Making, Memory Management, Processor Scheduling

1. INTRODUCTION

An independent entity that observes and in turn acts upon an environment is essentially an intelligent agent[4]. Every action of the agent is in direction of achieving the identified goal set. In an analogy to a computer program, intelligent agents could be compared to abstract functional systems. A multi-agent system in particular, is a macrocosm of many intelligent systems, that co-exist and get involved in the decision making process; with or without interacting with the other agents. In environments where the state of the system may vary dynamically, multi agents can be effectively used.

The scenario that will be dealt with here, is memory management in a multi processor environment. Amongst the sequence of steps involved in the process, loading the process from waiting queue to main memory can be taken under consideration. Based on the availability of a processor, the process selected on basis of the priority mechanism, moves into the level caches and subsequently gets executed. . On completion of a process, the corresponding turn-around time is computed. In the proposed framework these tasks are split amongst the agents. All agents essentially un in parallel mode. Every agent keeps track of the current status of the environment applicable to its task. A control agent coordinates the termination of processes and calculates their turn-around time, thereby providing a yardstick for comparing the efficiency of the traditional model with the agent-based model.

1.1. Paper structure

This paper is structured as follows. Section 2 contains a brief outline of the Memory Management System. Section 3 explains the proposed framework for memory management with and without agents. Section 4 provides the results and performance evaluation. Section 5 deals with conclusion and provides future directions and enhancements to the proposed system.

2. LITERATURE REVIEW

An operating system (OS) is a set of programs that intends at overseeing the hardware resources and gives services for application software. The operating system is an imperative component of the system software in a computer system[2]. Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting for cost allocation of processor time, mass storage, printing, and other resources.

2.1. Main memory

Memory accesses and memory management are a very significant fraction of modern computer operation. Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data

2.1.1 Main memory management

Issues that prompt main memory management

- Allocation
  Processes must be allocated space in the main memory
- Swapping, Fragmentation and Compaction
  A hole is created in the main memory when a program terminates or is moved out. This results in memory fragments that have to be compacted for an organized allocation.

2.1.2 Paging

Paging is a memory management scheme that allows processes' physical memory to be sporadic, and that eliminates problems with fragmentation by allocating memory in equal sized blocks known as pages [5]. The central idea behind paging is to divide physical memory into a number of equal sized blocks called frames, and to divide a program's logical memory space into blocks of the same size called pages.

2.1.3 Segmentation

Segmentation is a memory management scheme that supports a user view of program. At a particular moment, the page table is used to look up what frame a particular page is stored in. Segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple [5].

<segmentNumber,offset>

2.1.4 Segmentation with paging

There are pros and cons to both paging and segmentation. To cascade the pros these two models can be combined. The segments being partitioned into pages are determined by the page size as depicted in the Figure 1. Segments can have a variable size. Each segment is further split into pages. The pages are all of the same size

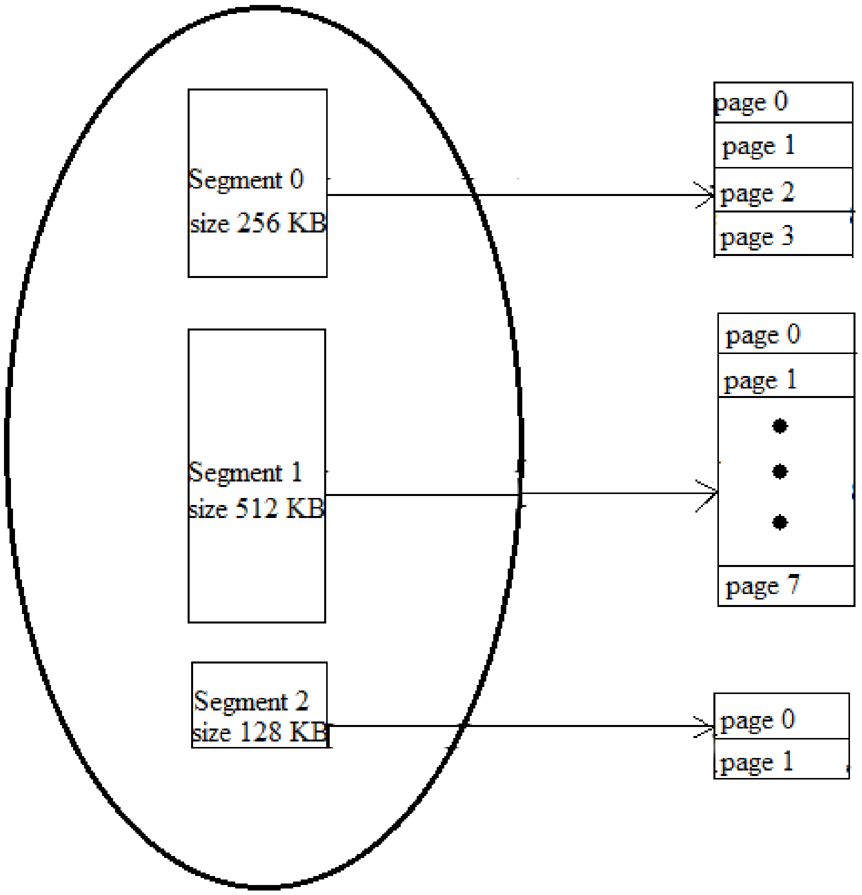

Figure 1. Main memory of size 896 KB with segments and pages of size 64 KB

2.2. Process

A program under execution is a process [2]. Text section is the part of the process that contains the program code. Program counter, process stack and data section are the other components.

2.2.1 Process Scheduling

In the design of a scheduling regulation, many objectives have to be accounted for. Mainly, the scheduler must consider the following: efficiency, fairness, response time, throughput, etc. The central idea of employing multiprogramming is to enhance CPU utilization. This can be achieved if there exists any valid process, running in the processor during its idle time. A uniprocessor system is capable of handling only one process at a time. In case where more than one process exists, the rest have to wait until the processor becomes idle.

2.2.2Scheduling algorithms

The Scheduling algorithms can be split into two categories with regard to the method in which they deal with clock interrupts viz. Preemptive and Non-preemptive.

2.3. Cache

Memory accesses to main memory are slow and may take a number of clock ticks to complete. This would require awful waiting by the CPU if it were not for anegotiatoryfast memory cache. The rudimentary idea of the cache is to transfer parts of data stored in memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache. Cache buffers data between the fast processor and slow main memory. Its goal is to minimize the number of processor accesses to main memory.

2.3.1 Cache levels

A two tier architecture is being used by the processors now, as shown in Figure 2. The first tier, called the Level 1 or L1 cache, is placed directly in the processing core and loaded with the instructions and data items the processor's execution units need immediately[7]. Access to L1 cache is extremely fast due to its location at the innermost parts of the processor. A second, larger, and slower Level 2 (or L2) cache handles the interface to main memory on many designs. In this methodology, the L2 cache feeds the L1 cache, which in turn feeds the processor core.

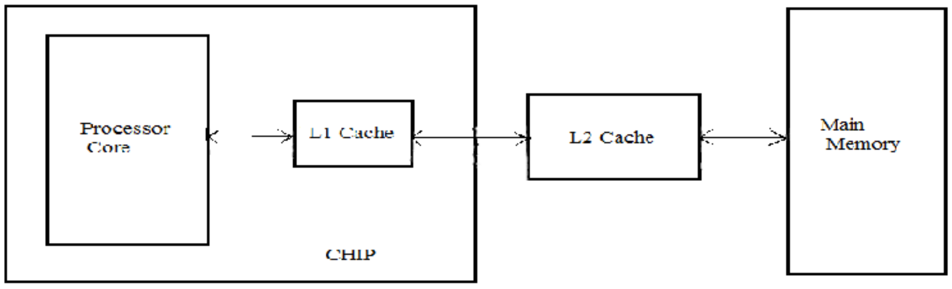

Figure 2. Cache levels

3. PROPOSED METHODOLOGY

The main memory management described so far is the traditional approach. Our work aims to bring in the agent into this dynamic environment. The goal is to analyze any efficiency brought out by the agent approach.

3.1. Conventional approach

An implementation of the conventional approach has been done. This is in lieu with our aim to compare the turnaround times of the conventional approach versus the agent - based approach.

3.1.1 Architecture

The pictorial representation of the conventional approach is depicted in Figure 6. The processes forked are kept in the waiting queue. Each of the process is assigned to a priority. A default priority is assigned, in case of no assignment of priority. Based on their priorities, processes are scheduled from the pool. The lowest number has the highest priority. The highest number has the lowest priority. The scheduled processes are then loaded into main memory and are therefore ready for execution. Every process is further split into pages of equal size.

The page table provides an index feature to main memory. This helps in holding the status of a page. Attributes like pageID and size are also maintained. When a page is loaded into the main memory, an entry is created in the page table. In this case, a search is made amongst the processors to ascertain the processor availability. The processor may be executing another process and is therefore busy or it may be idle and is available for executing the current page under consideration.

On choosing an idle processor, a copy of the page from main memory is placed into the corresponding L2 cache. By the mechanism of load through, the copy of the page from L1 cache is loaded from L2 cache. L1 cache is much smaller in size compared to L2 cache. L1 cache is the on-chip cache and its access time is therefore lesser than that of L2 cache.

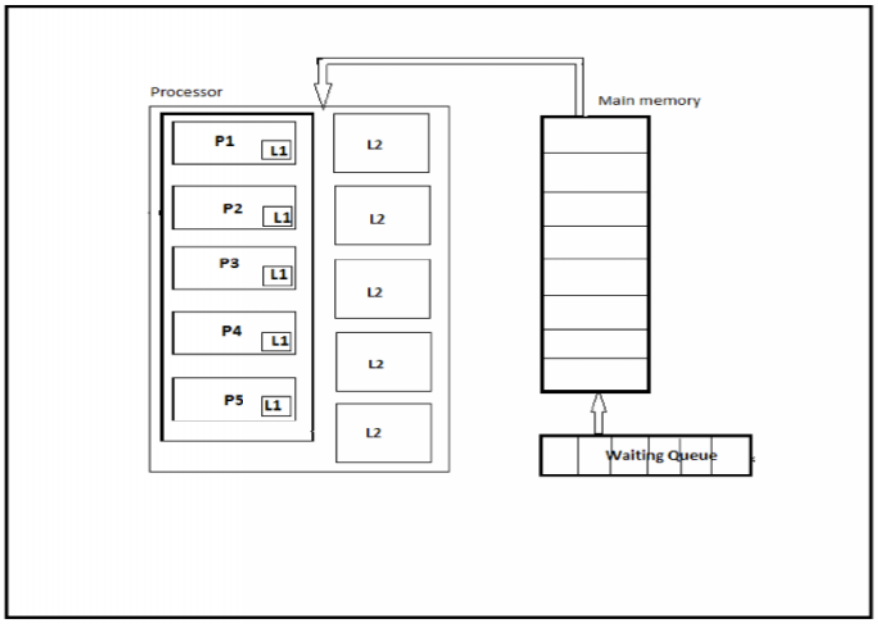

Figure 3. System architecture-conventional approach

Once the processor is assigned to a page, execution starts. When all the pages corresponding to a single process are executed, the total turn-around time is computed. Turn-around time is the time taken from process submission to the time it completes its execution[2].

Turn-around time of process 1=wait_time + time1 + time2 + ... +time'n'

where,
wait_time is the amount of time the process remains in the waiting queue;
time1, time2 are the time taken for the complete execution of pages 1,2 of process1.

3.1.2Pool

The processes that are in ready state are dispatched to the pool. The process in the pool may have 2 statuses. A newly entered process is represented by status 0. Status 1 denotes that the process has been loaded into main memory and is awaiting the completion of execution.

3.1.3Memory partitioning

The size of the main memory has been fixed as 1050 bytes. Each segment in main memory is of varying sizes. There are totally 4 segments. Segment 1 is of size 210 bytes. Segment 2, 3 and 4 are of size 280 bytes each. The page is of a fixed size, fixed as 70 bytes. The varying segment size may result in fragmentation. The last page of the segment may at times be a little more or at other times a little less. Segment 1 therefore has 3 pages. Segment 2 ,3 and 4 have 4 pages each.

3.1.4 Page

The page table provides an indexing feature to main memory. The page table helps in holding the status of a page. Attributes like pageID, segmentID and size are also maintained. When a page is loaded into the main memory, an entry is created in the page table. Status 0 denotes that the page has become stale and can be replaced. Status 1 denotes that the page is to be executed.

3.1.5 Processor

Processor is the unit of execution. To provide with a multi processor environment, 3 processors have been assumed in this implementation. The processors have a dedicated on-chip L1 cache and a dedicated L2 cache individually. As several identical processors are available, then load sharing can happen. It would be probable to maintain a separate queue for each processor. But consider the situation when , one processor could be idle, with an empty queue, while another processor is very busy. To avoid this expensive situation, a common ready queue is employed, in the case of the implementation it is referred to as the pool. All processes go into one queue and are scheduled onto any available processor.

3.1.6 Implementation

As the memory management is a part of the kernel setup no user interaction is needed. But for greater understanding a GUI has been created so as to permit the user to send particular processes into the pool table. The GUI essentially consists of a frame window with 15 buttons. The label of each button reads the process name, its size and the priority assigned to it. An actionListener is associated with each button. On click action, the corresponding process is created and is scheduled in the waiting queue. It is to be noted that processes may thus be created at any point of time during the execution of the program simulation thereby creating a dynamic set of process inputs.

The process is prematurely divided into pages. A parent set of processes, with size 69,139 and 207 bytes are already present. The button click and the subsequentprocess creation is done by spawning from the original parent set of processes.

The pid of the process is unique and is assigned as mentioned on the label. The priority values assigned to processes vary in the range 1-5. Priority 1 signifies the highest priority and 5 signifies the lowest priority for a process.The execution of pages corresponding to processes in the processor is done using system calls for the sake of simulation. The ProcessBuilder class object creates a thread of execution for the given command and corresponding arguments. To start the execution of the page, the thread has to be explicitly started.The process entry from the page table is removed. This in turn is reflected in the main memory. The main memory changes the status of the corresponding pages to 2 signifying that the page is ready for replacement. The process is removed from the pool. When a process enters the pool another time, before loading into the main memory it is checked for prior occurrence. If a past copy resides and is not stale, then the status of the page entry is changed to 1. To avoid redundant pages, the entries of caches are also checked before processor allocation.

3.2. Agent based approach

In the field of computer science, an agent is essentially a software program or an enhanced piece of code that acts for a user or on behalf of some other program[4]. Concepts of intelligent agents, distributed agents, autonomous agents and multi agents are derived from the original idea of an agent.

3.2.1 Introduction

The current scenario underlying the conventional approach to memory management is that of a sequential approach. The dynamic nature of the environment calls for rapid and relevant decision making. Decisions are to be made in scenarios where resource allocation becomes a tedious and a time-consuming job. The tenacity of having to check the status of every page table entry, main memory segment pages and the processor availability seemingly adds unnecessary overhead to the total computation time. The proposed model aims to employ agents that perform such decisions. In case of change in the set of available resources, the agents must be capable of altering their decisions to accommodate the recent changes and not stand by the stale decisions already taken. The work aims at comparing the results of the traditional as well as the agent based approach. This calls for a judicious use of the agent behavior. Excess operations being trusted on the agent may not be viable.

3.2.2Proposed architecture with two agents

In this implementation, two agents, Memory agent and Processor agent have been employed as shown in Figure 4. The memory agent monitors the AvailMemory table. The memory agent decides where the next page of a process must be placed. The available frames in the segmented pages of main memory are ascertained at any point of time and form a part of the result set of the decision set. In case of an existing copy in any of the frames, the status of the page in the frame is updated to denote this current status. This avoids redundancy of pages and helps in the effective utilization of the memory resource.

The processor agent monitors the status of the processors and maintains this information in the AvailProcessor table. This agent then checks for the copy of the page under consideration, within its dedicated caches. Accordingly a copy is loaded into the two-level caches or the status is changed in the existing copies.The processor agent is therefore responsible for allocating the page to the chosen processor and the loading of the page into the appropriate location within the cache.

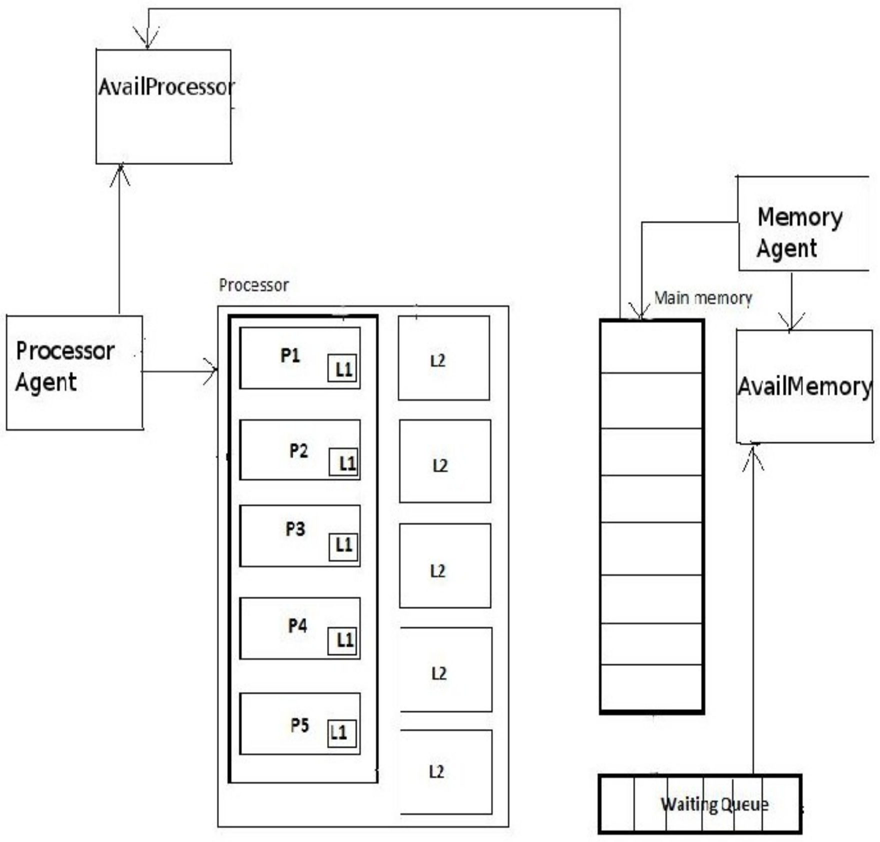

Figure 4. Architecture showing two agents

3.2.3 Proposed architecture with three agents

An improvisation to the previously pronounced agent based architecture can be made. Previously the processor agent had to monitor over the statuses of both the processor and the corresponding caches. But it is to be noted that for deciding on the availability of the processor the cache contents need not be sifted through and updated. This calls for a separate agent to be allotted to monitor the cache details of the chosen processor as is shown in Figure 5. The tables AvailL1 and AvailL2 store the details of the L1 and L2 caches pertaining to the chosen processor.

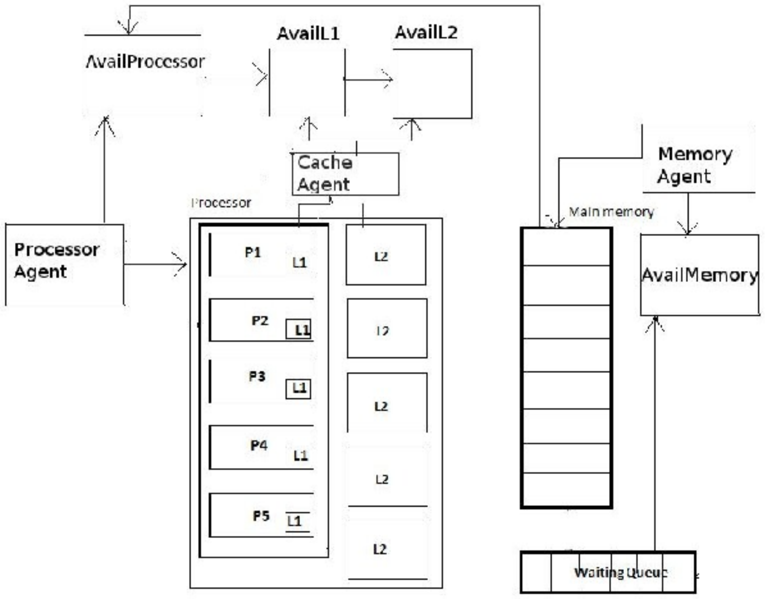

Figure 5. Architecture with 3 agents

3.2.4 JADE

The implementation of agent calls for a software framework. JADE is one such framework completely implemented in java. The implementation of multi-agent systems is simplified. This is done through a middleware. A set of graphical tools that support deployment and testing phases render JADE a programmers' unonumero choice. The agent configuration can be controlled via a remote GUI and the platform can be distributed across machines[3].

JADE provides a proper set of functionalities to simplify development of multi-agent systems but it puts very few restrictions on the user code without, in particular, imposing any specific agent architecture. JADE can be deployed both on JEE, JSE and JME devices and its runtime provides a homogenous set of APIs that is independent of the underlying network and Java technology. The ACL offers scope for agent- agent peer interaction through asynchronous message passing [2].

Though agents offer an insight to the dynamic situation, too many agents tend to result in an overhead[6]. On comparative analysis, the memory management situation is ably handled by 3 agents. The agents can communicate with their peers via message passing. But the scenario of memory management renders it unnecessary. The agent- agent peer communication is therefore done using shared resource concept. The agents each have a table to constantly monitor and update the tuples based on the decisions taken. The handling ability and the job of each agent is described below.

3.2.5 Main memory agent

The main memory agent maintains the availMemory table. This table holds the segmentID and pageID of a free frame. When a page is scheduled to be loaded a search for a free location is usually initiated. But in the agent based approach, the agent already scans the available locations in the main memory. Prior to allocation a scan needs to be done to ascertain that a duplicate copy of the page does not exist. This is done by assigning states with the pages. State 0 implies the frame can be occupied as no page resides in it. State 1 implies that the page is to be executed. State 2 implies that the page is done with execution so can be reduced.

3.2.6 Cache agent

The cache agent monitors the availL1 and availL2 tables. The agent primarily checks for redundancy of the page under consideration. If the page is new, it is swapped onto the L2 cache. By load through mechanism the page is placed in the L1 cache too. In case a copy already exists, the status of the previously existing page is updated to 1 signaling that it is ready to be executed.

3.2.7 Processor agent

The processor agent monitors the availProcessor table. This table stores the details regarding the processor that is currently idle. The details include the cache locations that are empty. This in turn checks with the cache agent to check for available pages in L1 and L2 cache after the required processor is ascertained

3.2.8 Sequence of steps involved

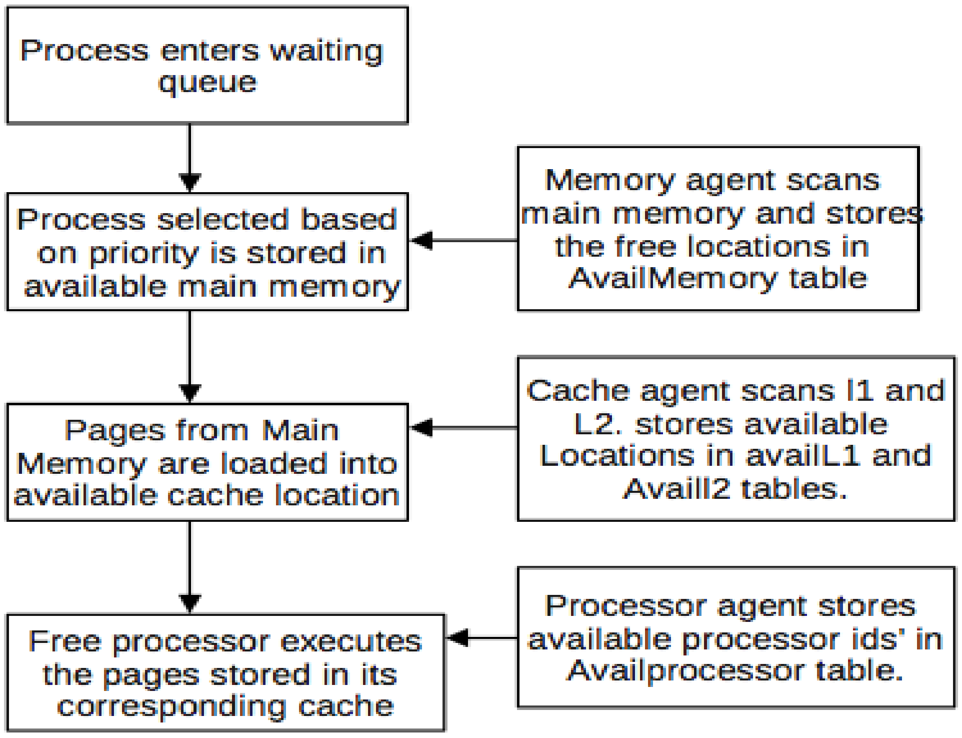
Process enters waiting queue
Process selected based on priority is stored in available main memory
Memory agent scans main memory and stores the free locations in AvailMemory table
Pages from Main Memory are loaded into available cache location
Cache agent scans l1 and L2. stores available Locations in availL1 and Availl2 tables.
Free processor executes the pages stored in its corresponding cache
Processor agent stores available processor ids' in Availprocessor table.

Figure 6. Flow chart explaining the steps involved

Figure 6 shows the steps involved in 3 Agent approach. In 2- Agent approach, the same set of actions are performed with the exception that the cache agent is absent.

3.3. Assumptions

The basic assumptions made in the traditional approach for issues like the number and size of resources hold good for the agent approach too. The significance lies in the agents' ability to make decisions in a dynamic environment. This is aided by inter agent communication. Instead of the usual message passing approach, peer-peer communication is availed by shared resource concept. The table availMemory, availL1, availL2 and availProcessor aid in the agent communication.

3.4. Implementation

The necessary modules and functions are offered by the JADE framework to create agents and a lot their behavior with the sole purpose of achieving the goal. The agents can posses behaviors such as CyclicBehavior,WakerBehavior and TickerBehavior[2]. Agents that have to repeatedly perform its task after a given lapse of time interval are assigned the CyclicBehavior. An agent that needs to be executed or initialized only once is given the WakerBehavior. Agents that are to be invoked after specific intervals of time are assigned the TickerBehavior[10].

The tablesavailMemory, availL1, availL2 and availProcessor store the decisions taken by the agents in the form of tuples. This in turn reduces the time to access every resource demanded by a process and then scanning through each of their statuses to ascertain their allocation. The tuples of these tables are dynamically varying with the environment and hold good at any specific point of time.

3.4.1 Tables Used

The processer table provides information about the processors in the system. Field id indicates the processor's id; content is the name of the process that is in the processor; available field indicates if the processor is free or occupied. A value of 1 indicates that the processor is free and 0 indicates that it is busy.

Table 1: Description of Processor table

Field	Type	Key
id	int(11)	PRI
available	int(11)	

The mainmemory table provides information about the pages. Field sid indicates the id of the segment and spid is the page id. Pname is the name of the process, available shows the amount of free space in that page and state indicates if the page is free or occupied.

Table 2: Description of Mainmemory table

Field	Type	Key
sid	int(11)	PRI
spid	int(11)	PRI
pname	varchar(80)	
available	int(11)	
state	int(11)	

The table availMemory is used to identify the free locations of main memory. Hence it contains only the primary key fields of mainmemory i.e. sid and spid. And also stores the amount of available space in the page.

Table 3: Description of availMemory table

Field	Type	Key
sid	int(11)	PRI
spid	int(11)	
available	Int(11)	

The availProcessor table has the updated information pertaining to the list of idle processors. Similarly availL1, availL2 and availProcessor contains only the primary key fields of l1,l2 and processor table.

3.5. Dynamic environment

In lieu of implementing a dynamic scenario, one version of the implementation has a code that alters the state of a random processor cache. It may also load dummy pages to the main memory. All of this is done to verify the proper working of the logistics of the implementation. Even with such alterations to the working environment prompt and proper updations are made to the decisions taken by the agents.

4. EVALUATION AND RESULTS

The performance of the system has been evaluated and three classes of behavior of the method have been studied experimentally.

4.1. Input Set

Input set consists of many processes of varying priorities. Processes are generated dynamically each time. The process is split into equal sized pages. The status of processor, mainmemory, pagetable are printed on screen periodically. Turn around time is calculated when all pages pertaining to a process are executed and is printed on screen. The same set of processes are executed in the traditional, 2-agent, 3-agent approach and the improvement in turn-around time is calculated in milli-seconds. The results from two sample runs are tabulated and graphs are drawn for the same.

4.1.1 Sample 1

Pid	Size	Priority	Traditional	2 Agents	Change
1	207	1	6362	4218	2144
2	207	3	5455	4920	535
3	207	2	8353	10054	299

Table 4: Processes of same size, different priorities

Pid	Size	Priority	Traditional	3 agents	Change
1	207	1	6362	4175	2187
2	207	3	5455	3899	1556
3	207	2	8353	7666	687

Table 5: Processes of same size, different priorities

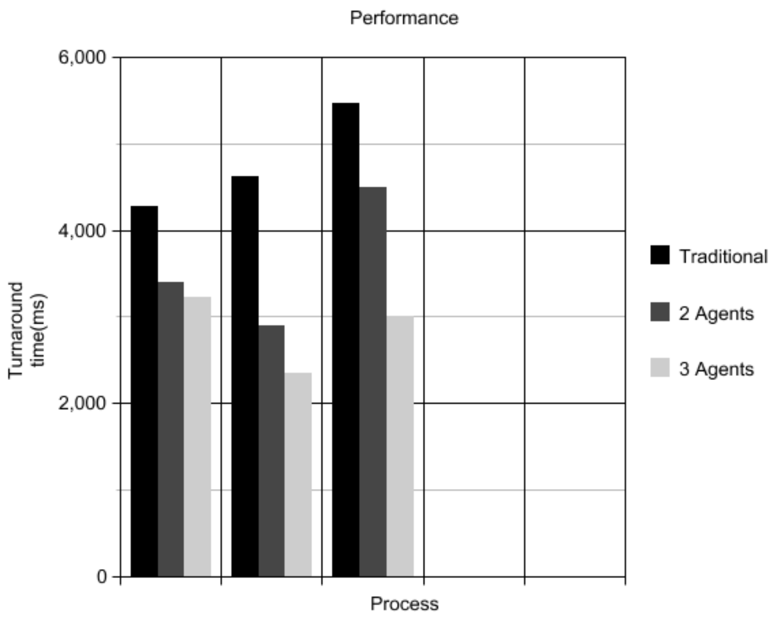

Figure 7:Graphical representation of data shown in tables 4,5

4.1.2 Sample 2

Pid	Size	Priority	Traditional	2 Agents	Change
5	139	2	4273	874	3229
9	69	2	4625	2904	1721
14	207	3	5471	4497	974

Table 6: Processes of different sizes, different priorities

Pid	Size	Priority	Traditional	2 Agents	Change
5	139	2	4273	874	3229
9	69	2	4625	2904	1721
14	207	3	5471	4497	974

Table 7: Processes of different sizes, different priorities

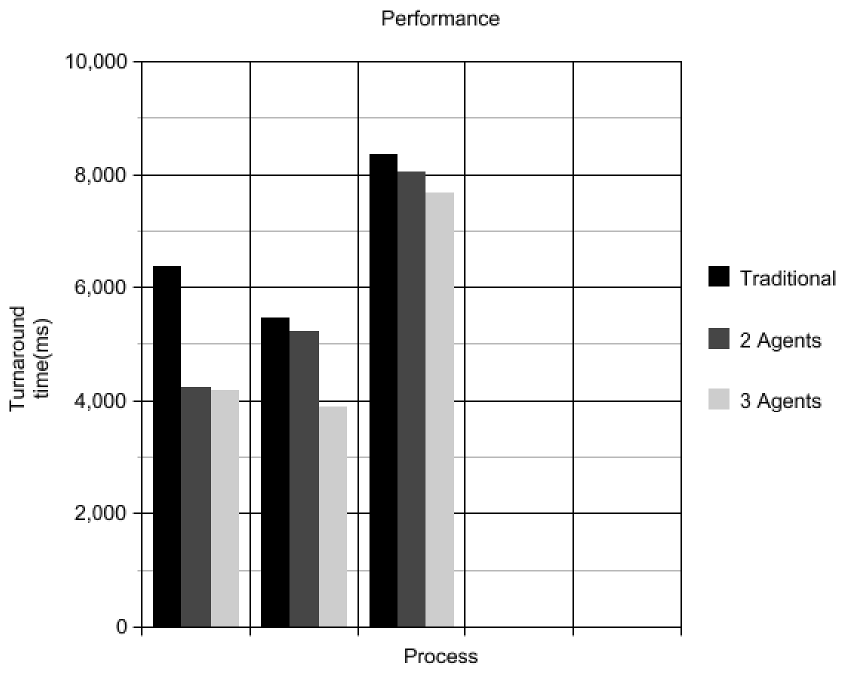

Figure 8: Graphical representation of data shown in tables 6,7

1) The turn-around time is reduced by almost 30% with 2 agents.
2) The turn-around time is reduced by almost 40% agents for almost all input processes.

4.2. Factors affecting the performance

For a given priority, smaller process takes lesser time to execute comparing to bigger processes. Similarly, for a given size, higher priority processes take lesser time to execute comparing to low priority processes. Smaller size and low priority results in increased turn-around time.

From the sample data, it can be seen that there is a linear variation of execution time with priority for fixed size processes and for processes with different sizes and same priority. This is represented graphically below.

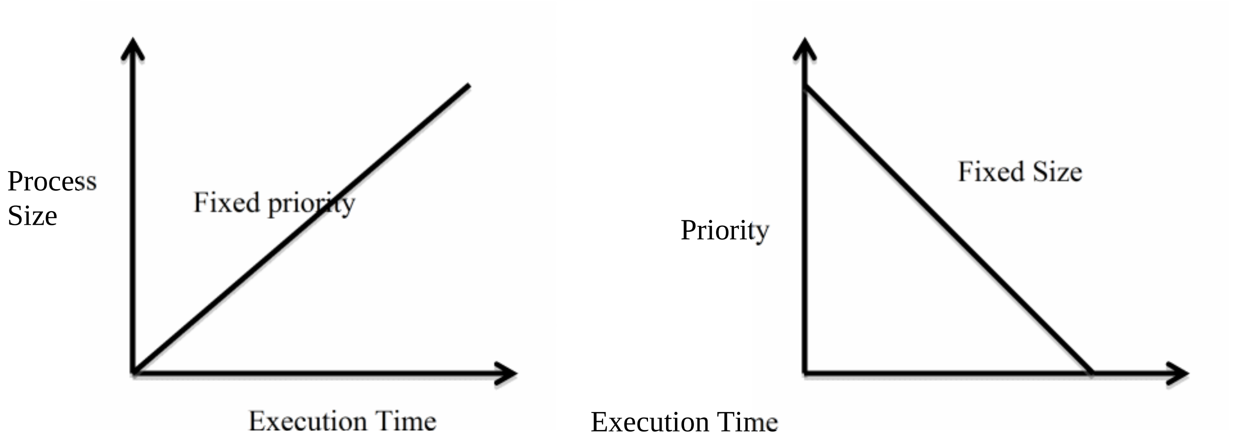

Fig.9: Variation of Execution time with Process size and Priority

5. CONCLUSIONS

On comparing the turn around time of the processes executed in the traditional approach and the agent based model, it is concluded that agent based model is more efficient of the two. Furthermore it is noted that placing 3 strategic agents is the most suitable form of agent based model. Further increase in the number of agents provides more overheads and is thus not advised.

5.1. Enhancements

This work offers a good scope for enhancements. These enhancements will be briefly described henceforth.

5.1.1 Advanced Version of scenario

We have only implemented a basic version of memory management system. As an enhancement it is proposed that it can be extended to a more detailed implementation. Memory management can be done for multi processor, multi core architectures separately and on a comparative basis. A comparison between the efficiencies of the upgraded traditional architecture and upgraded agent based architecture can be carried out and the results compared.

5.1.2 Parameter value set

Trial runs can be made with different page sizes and cache sizes. Ascertaining the ideal set of resource sizes can later be used to procure an ideal memory management scenario. With this in the pipeline, the traditional and agent-based approaches can be employed and their results with respect to their efficiencies be compared.

5.1.3 Dynamic Resource allocation

The system can be modified to be able to dynamically allocate the resources within the memory management scenario. Based on the given set of inputs, the system must dynamically be able to choose the appropriate size of the cache and page. This dynamic decision can be taken with the help of intelligent agents. The agents can be implemented as learning agents with respect to heuristics

6. REFERENCES

[1] G. Muneeswari. 2011. Agent Based Load Balancing Scheme using Affinity Processor Scheduling for Multicore Architectures - European Journal of Scientific Research. Volume 55 Issue 3, 247-258.

[2] I. Seilonen, T. Pirttioja, K. Koskinen. 2009. Extending process automation systems with multi-agent techniques, Engineering Applications of Artificial Intelligence 22, 1056- 1067. [3] JADE homepage - http://jade.tilab.com/, May 13, 2012.

[4] Intelligent Agents -http://en.wikipedia.org/wiki/Intelligent_agent, May 03, 2012.

[5] Memory Management -http://en.wikipedia.org/wiki/Memory_management, May 12,2012.

[6] Lijun Wu, Jinshu Su, Kaile Su, XiangyuLuo, Zhihua Yang. 2010. A concurrent dynamic logic of knowledge, belief and certainity for multi-agent systems, Knowledge-Based Systems 23. 162-168.

[7] Multi Level Caches - http://en.wikipedia.org/wiki/CPU_cache#Multi-level_caches, April 20, 2012.

[8] Segmentation with Paging - www.cs.jhu.edu/~yairamir/cs418/os5/sld042.htm, April 12, 2012.

[9] T.Vengattaraman, S.Abiramy, P.Dhavachelvan, R. Baskaran. 2011.An application perspective evaluation of multi -agent system in versatile environments, Expert Systems with Applications 38, 1405-1416.

[10] Wei Wu; Far, B.H.; Eberlein, A, An implementation of decision making models and mechanisms for multiagent systems in competitive environments - Electrical and Computer Engineering,2003.IEEE,CCECE,2003,Volume:2, DOI:http://dx.doi.org/10.1109/CCECE.2003.1226120.

[11] Yang Ping; Zeng Jing; Liu Weidong; Jia Kai. 2009. Study on General Multi-agent Systems Implementation Framework- Intelligent Human-Machine Systems and Cybernetics, Volume: 2,DOI: http://dx.doi.org/10.1109/IHMSC.2009.161