VOL. 11, NO. 9, MAY 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

SCHEDULING OF SHARED MEMORY WITH MULTI - CORE PERFORMANCE IN DYNAMIC ALLOCATOR USING ARM PROCESSOR

Venkata Siva Prasad Ch. and S. Ravi
Department of Electronics and Communication Engineering, Dr. M.G.R. Educational and Research Institute University, Chennai, India
E-Mail: siva6677@gmail.com

ABSTRACT

In this paper we proposed Shared-memory in Scheduling designed for the Multi-Core Processors, recent trends in Scheduler of Shared Memory environment have gained more importance in multi-core systems with Performance for workloads greatly depends on which processing tasks are scheduled to run on the same or different subset of cores (it's contention-aware scheduling). The implementation of an optimal multi-core scheduler with improved prediction of consequences between the processor in Large Systems for that we executed applications using allocation generated by our new processor to-core mapping algorithms on used ARM FL-2440 Board hardware architecture with software of Linux in the running state. The results are obtained by dynamic allocation/de-allocation (Reallocated) using lock-free algorithm to reduce the time constraints, and automatically executes in concurrent components in parallel on multi-core systems to allocate free memory. The demonstrated concepts are scalable to multiple kernels and virtual OS environments using the runtime scheduler in a machine and showed an averaged improvement (up to 26%) which is very significant.

Keywords: scheduler, shared memory, multi-core processor, linux-2.6-OpenMp, ARM-Fl2440.

1. INTRODUCTION AND MOTIVATION

A modern high-performance computing Multi-Core system normally consists of Many processors, For many decades, the performance of processors has increased by hardware enhancements (increases in clock frequency and smarter structures) to further enhance single-thread performance communication in Scheduler method to which work in specified by some tasks assigned to resources to complete the manage the total work, involved concept of the New scheduling tasks makes that possible to computer tasking with a single central processing unit (CPU). The scheduler to aim the one of many goals in the processing variables, fairness it is equal to CPU time for each processing in times according to the priority and workload of each processing threads. The work is mainly focused on the design and development of the task scheduling between the child and parent (or) any processing elements to meet timing constraints while minimizing system energy consumption and Time delay. Multi-core architectures creates significant challenges in the fields of computer architecture, software engineering for parallelizing applications, and operating systems. In this paper, we show that there are important challenges beyond these areas. In particular, we expose a new security problem that arises due to the design of multi-core architectures the number of clock cycles spent with allocation and de-allocation of shared memory is low for resolving. However, not only the time and Computational used up on the allocation and de-allocation is important, but also how efficiently the allocated memory can be accessed. Scheduling in multiprocessor real-time systems is an old problem in multicore processors have brought a renewed interest, along with new dimensions, to the challenge ,For instance, there is a need to trade off different levels of migration cost, different degrees of inter-core hardware sharing (e.g. memory bandwidth), and so on. Our research is aimed at providing new knobs to perform these tradeoffs with additional application information an important problem is how to assign processors to real-time application tasks, allocate data to local memories, and generate an efficient schedule in such a way that a time constraint can be met and the total system energy consumption can be minimized.

2. SHARED MEMORY

Shared Memory is an efficient means of passing data between programs/processes. One program in a core will create a memory portion which other core (if permitted) can access. The permission is granted in a regulated manner for Shared-memory in typically for providing in both static and dynamic process creation of tasks. the processes can created and directed by beginning stage of the program execution by a directive to the operating system and the CPU Logic Threads, or they can be created during the execution of the program. The typical implementation of thread allow to process start another, child thread, process by a fork model has allocated. The shared memory of the caches and processing in Runtime has shown in Figure-1. In Three processes are typically to manage the coordinating processes in the shared memory programs for the scheduling in the related thread. The starting, child or a parent, process can wait for the termination of the child process by calling join. These second prevents processes from improperly accessing the shared memory resources. The shared-memory model is similar to the data-parallel model.

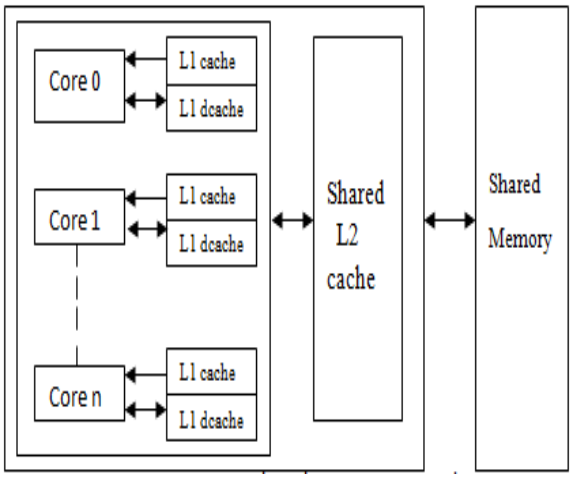

Figure-1. Shared memory at runtime.

A. Shared memory programming

A process creates a shared memory segment in a core using shmget(). The original owner core of a shared memory segment can assign ownership to another core with shmctl(). It can also revoke this assignment and prority levels can be set with respect to the order of revokement. Other core processes in proper permission to perform various control functions in the shared memory segment using shmctl(). Once the created, a shared segment is attached to a process address space using shmat(). and detached using shmdt() (see shmop()). The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. The developed core sharing features include

a) The shared memory can be attached multiple times by the same process.

b) A shared memory segment that can be described in a control manner structure for a unique ID those points to in the area of shared memory. These identifier of the segment is called the *shmid*. The structure for the shared memory segment control can be found in <sys/shm.h>.

B. Shared memory allocation

The shared memory allocation and de-allocation of the multi-core processor has implemented in the two levels of memory allocation

They are

a) Kernel Level: memory management of OS sub-systems.

b) User Level: implemented by UMA that is a library responsible to manage the heap area.

The External Memory allocation purpose systems are now need to parallelize the work in order to get the expected performance increment cores in the Scheduler. In the extreme, run all the tasks of the parallel pieces in the same core, such parallelism is completely wiped out. Hence, the task-to-core allocation and the scheduling of hardware resources between core can change completely the performance of these systems.

C. Memory de-allocation

The kernel level of the multi-core memory allocation and de-allocation has shown in the Figure-2. The indirect data structure of the store matrices in memory. The first child (0, 1) task writes in the next block to allocate memory next to the alternate block. If any memory allocation fails, the runtime systems return an error and go to task 1. It is more difficult to de-allocate the blocks in the run- time system cannot be decide to whether a block will used in the future. For that multi-core processor has be running across the shared memory for Trying to move free memory to one large block. For Example- a review of the Linux 2.6 scheduler structures in the Linux, Each CPU has a run queue made up of 32 priority lists that are serviced in FIFO order. Tasks are scheduled to execute in the end of their respective run queue's priority list. Each task has a time slice that determines how to manage the child to parent of the allocation of the shared memory in the data block It translates the number of bytes to be transferred into the number of 64-bit words, because it accesses the parent shared memory via a 32-bytes the transfer sizes will be used by preferring the maximum possible one based on the address on the 0, 8, 16, 24 of the four threads. These threads has run in the according to the program. In That Two Processor scheduler has that above example has shared the task scheduler between the processor A and processor B. allocated location has that the tasks are to shard between the processing of allocation of the module as shown in the Figure-2.

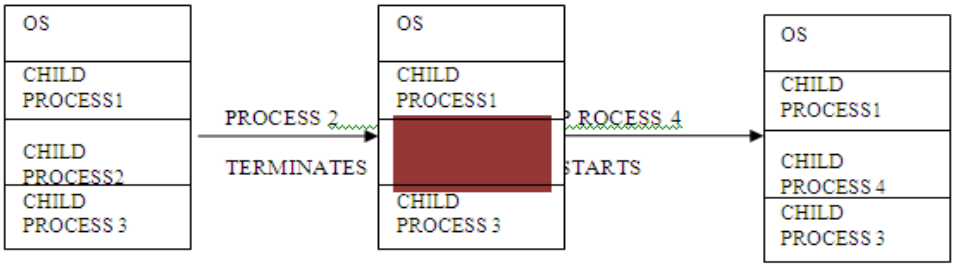

Figure-2. Memory allocation in thread.

D. Scheduler in runtime

In scheduler the Shared memory management allows task to allocate and de-allocate memory at runtime. This is useful when new tasks can enter the system at runtime or when a task itself has a dynamic memory requirement. It is generally accepted that load balancing is the greatest challenge it will explained in the shared process programming. Therefore, on most architecture, a find-first-bit-set instruction (FIFO) order is used to find the highest priority bit set in 32-bit words. The time it takes to find the task to execute depend on the number of active tasks but instead on the number of priorities. Local copies of global variables in child processes are private and temporary It can trans from the number of bytes to be transferred in the number of the 64-bit words in Processing bit Size, because it accesses the parent shared memory via a 32-bytes the transfer sizes will be used by preferring the maximum possible one based on the address on the 0, 8, 16, 24 of the four threads. These threads has run in the according to the program.

3. SCHEDULING ALGORITHMS

Scheduler working on new ways to take advantage of application knowledge to use them as parameter in the scheduling algorithms at all levels of the computer system. The Operating systems mediate access to shared hardware resources to assure optimal performance and fairness in the software kernel. For this treating the CPU as an indivisible resource diminishes the operating system's control over resource sharing. So there are several challenges posed on operating system scheduling algorithms designed for multi-core systems. Shared memory multi-core system consists to the ready queue where all the processes that are ready for execution will be available in the threads. The CPU scheduling is remarkably similar to other types of scheduling that have been studied for years. In this proposed concept, a model of the block sharing system &memory sharing is taken; the criteria focused on providing an equitable share of the processor per unit time to each child user or process is to minimize the average waiting time. So it is good to have an intelligent agent that combined with the scheduling will efficiently allocate and redistribute the load for an incoming process has shown in the Figure-3.

A. Major functions of the scheduler

The various functions of Scheduler implemented in Linux kernel scheduler. The Linux scheduler Contemporary multiprocessor operating systems, such as Linux 2.6 use a two-level scheduling approach to enable efficient resource sharing. The first level uses a distributed run queue model with per core queues and fair scheduling policies to manage each core. The second level is a load balancer that redistributes tasks across cores. The first level schedules in time, the second schedules in space. We discuss each level independently in the subsections that follow described below in Table-1 in detail. The load balancing function also comes as part of this API.

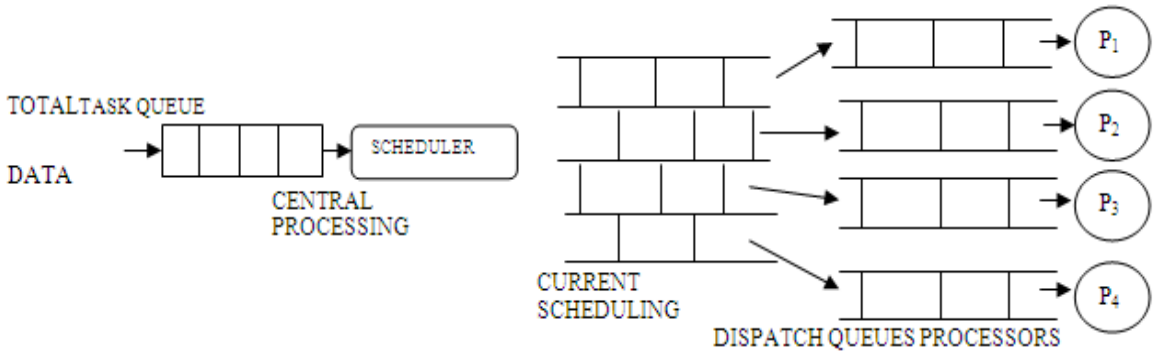

Figure-3. Illustration of schedule.

Table-1. Linux 2.6 scheduler operations.

Function name	Function role
Schedule	Schedules the highestpriority task for execution
effective_prio	It Returns the effective priority of a task (based on thestatic priority)
recalc_task_prio	This Determines a task's bonus or penalty based on its idletime.
source_load	Conservatively calculates the load of the source CPU
target_load	Liberally calculates the load of a target CPU (where atask has the potential to be migrated).
migration_thread	The High-priority system thread can migrates tasksbetween CPUs.

B. Lock-free algorithm

The Proposed algorithm of tasks handle in the shared memory at the runtime the processing of the schedule of proposing Hu level scheduling is one of a family of critical path methods because it emphasizes the path through the precedence graph with the greatest total execution time. Although it is not optimal, it is known to closely approximate the optimal solution for most graphs has First note that the utilization of n independent tasks with execution times is ei and periods pi can be written as

$$\mu = \sum_{i=1}^{n} \frac{ei}{pi}$$

If μ = 1, then the processor is busy 100% of the time. So clearly, if μ> 1 for any task set, then that task set has no feasible schedule of Horn's algorithm. The proposed algorithm is optimal with respect to feasibility only among fixed priority schedulers; EDF is optimal dynamic priority schedulers. In addition, EDF also minimizes the maximum lateness. Also, in practice, EDF results in fewer pre-emptions Shown in Figure-2 which means less overhead for context switching. This often compensates for the greater complexity in the implementation.

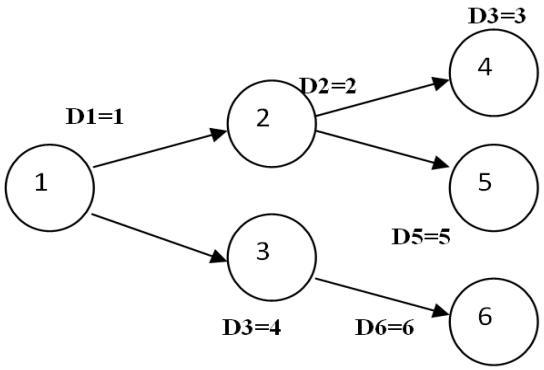

Figure-4(a). Scheduler priority in task allocation

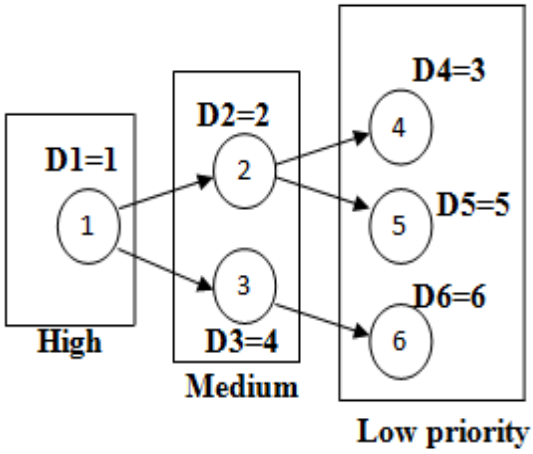

Figure-4(b). Scheduler priority in task allocation

Example: Consider six tasks [T = 1, 2, 3, 4, 5, 6]. Each with execution time ei = 1, with precedence's as shown in Figure-4(a). The diagram means that task 1 must execute before either 2 or 3 can execute, that 2 must execute before either 4 or 5, and that 3 must execute before 6. The deadline for each task is shown in The Figure. The schedule labelled EDF is the EDF schedule. This schedule is not feasible. Task 4 misses its deadline. However, there is a feasible schedule. The schedule labelled LDF meets all deadlines. Figure 4(b), task 1 has level 3, tasks 2 and 3 have level 2, and tasks 4, 5, and 6 have level 1. Hence, a Hu level scheduler will give task 1 highest priority, tasks 2 and 3 medium priority, and tasks 4, 5, and 6 lowest priorities. According to the Algorithm has run in the Figure-4, we see that the EDF scheduler is the same as the LDF schedule. The modified deadlines are as follows Table-2 the key is that the deadline of task 2 has changed from 5 to 2, reflecting the fact that its successors have early deadlines. This causes LDF to schedule task 2 before task 3, which results in a feasible schedule.

Table-2. Task allocation In EDF to LDF.

EDF	**TASK**	**1**	**2**	**3**	**4**	**5**	**6**
	ALLOCATION	**D1=1**	**D2=5**	**D3=4**	**D4=3**	**D5=5**	**D6=6**
LDF	TASK	1	2	3	4	5	6
	ALLOCATION	D1=1	D2=2	D3=4	D4=3	D5=5	D6=6

4. HARDWARE DETAILS

The Forlinx ARM-9 Board of Multi-Core processor used to demonstrate the memory sharing among the cores. Alternately ARM-9 FL2440 its components has CPU Samsung S3C2440A microcontroller, running @400MHzRAM64MB SD RAM core can also be used has shown in the Figure-5. The difference b/w the two approach is in the network file system. In ARM-11 system is loaded and does not require a 'NFS' file /folder in the host. As a standalone from a SD-card, it can be mounted and the code can be run. In ARM -9'NFS' folder is created and the code is placed in the roofs. The advantage is in ARM-9 core the code changes are reflected instantly and do not reboot &re completion using 'make' command as in ARM-11.

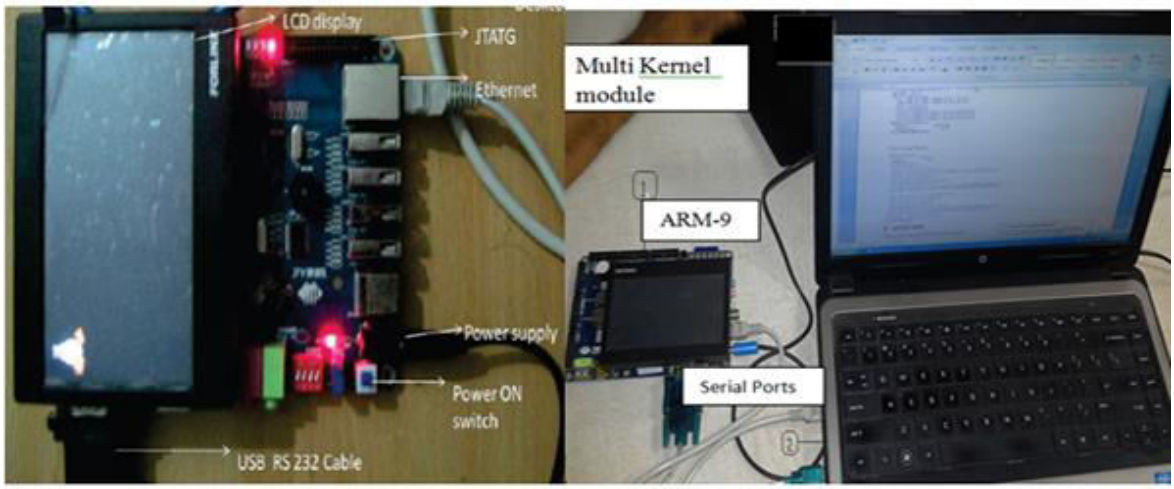

Figure-5. Linux Dumped in ARM-9 with output file.

5. IMPLEMENTATION AND PERFORMANCE ANALYSIS

A 32-bit ARM assembly language program has designed in the Linux Dual Kernel ARM-9 processor to written to implement the logical free algorithm which is designed to keep the number of tasks are completed and pending the data has been de-allocated. Once loaded more instruction cache related to bus traffic should occur function of the 64MB data cache of the ARM 9 and can be increased cache. In Linux scheduler an operating system, in general sense, the interaction between the applications and available resources. The physical devices in the shared memory the CPU can also be maintained in a resource to scheduler task can temporarily allocate to task. The scheduler can makes it possible to execute multiple programs at the same time in the related to the thread in the processor, An important goal of the scheduler to allocate CPU time slices efficiently providing the responsive user experience in The scheduler can also be faced with such conflicting goals and minimizing the response of times for a critical real time task while maximizing the overall CPU utilization in the 32 bit processor.

A. Child process implementation

The Child process Application in the Real-time Program Interface is a portable, parallel programming model for shared memory multithreaded in Linux Multiple Kernels specification version 3.0 introduces a new task. By using these tasking feature, applications can be parallelized where the units of work generated in dynamically in parallel shared memory, as recursive structures or while loops.

The Task 'N' Has run in the Process implementation that gas the automatic thread is generated. It may choose to execute the task immediately or defer its execution until a later time. The threads in the current team will take tasks out of the block and execute them until the block is empty it has shown in Figure-6. During the execution of a program, a task may create, or spawn a child task so that the spawned child task may run in parallel with the parent task. Each task except the root task has exactly one parent task. Pseudo code for the program illustrating the passing of a simple piece of memory (bytes) between the processes of a core is given below in Figure of a Linux 2.6 Kernel Algorithm.

a) Creating shared memories
b) Start the child process
c) Attach the memory to parents process address space
d) Resulting sharing with via shared memory
e) Synchronization routines among the child
f) Detach the multiple memory blocks
g) De-allocate shared memory blocks
h) Display the shared memory contents

Step (vii) ensures dynamic de-allocation and avoids overflow (or) dead memory allocation.

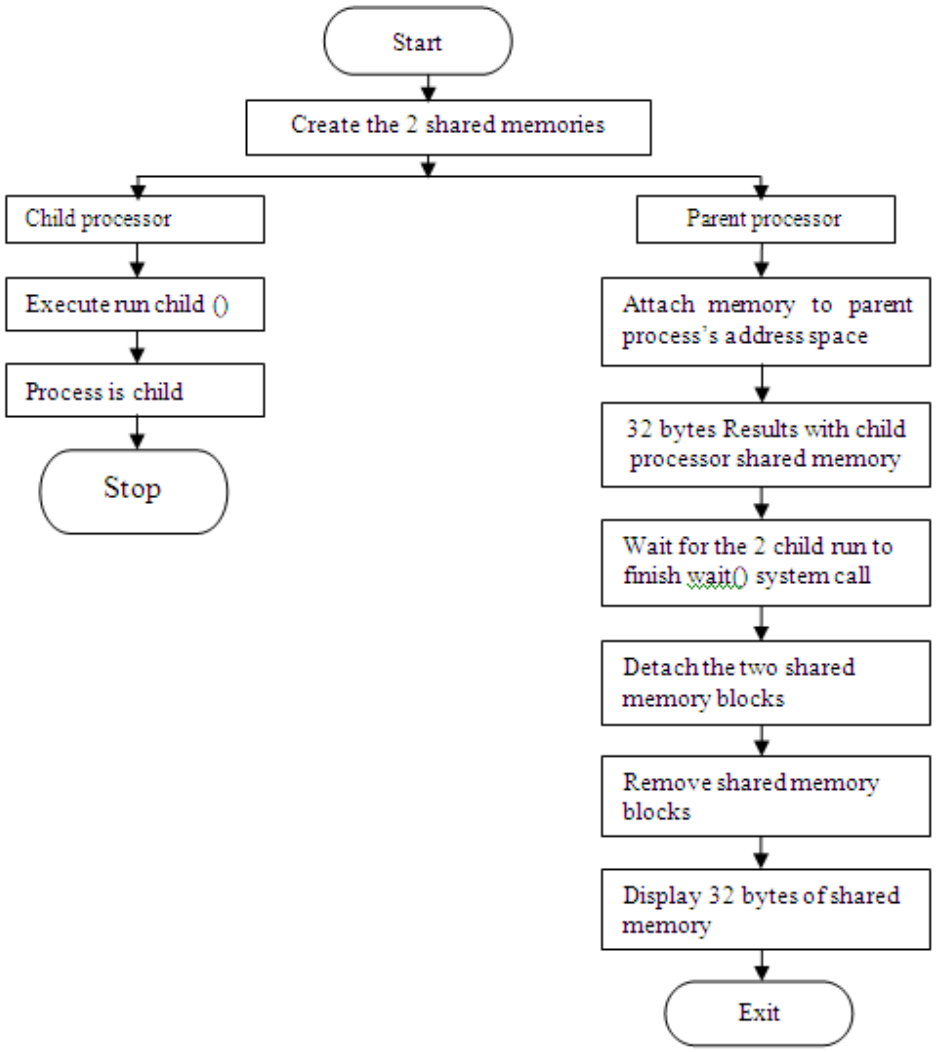

Figure-6. Integrating with multiple core.

6. RESULTS AND DISCUSSIONS

This research work is mainly focused on the design and development of task scheduling algorithms in shared memory of the De-allocate the unwanted or empty task filling the list using multi-core platform of the ARM processor to meet timing constraints while minimizing system energy consumption. The listed in the content files variables in child processes are private and temporary. They are not running in the processor tasks to each other and are destroyed when the processes ceases to exist (after executing process-join). These mechanism to make variable visible in the processes is by creating one memory addresses that are visible to all the processes Shared variables are freely visible to all the processes. Their values can be changed by child processes during execution and all changes are permanent. All changes are visible to the parent process even after the children cease to exist. In dynamic task scheduling, the each node makes a decision for each child task in the core. This decision is based on the current load balancing of the system. We do not require a specific mechanism to make this decision, because in part the decision is based on the queuing mechanism. For example, the global queue, it is natural to check the size of the queue as shown in the Table-3. If it is above a minimum value, then the creating node mines the task without queuing. It has shown believe that if the work is balanced in the system, having the creator of a task child will always be more efficient than enquiring the next task. the benefits of affinity Those static techniques invariably suffer performance costs. Consider a situation with 16 tasks and 4 processors as shown in Figure-7. If the tasks all require the same mining time, speedup can be increased With static load balancing, it is conceivable that the two largest jobs in the mining could be assigned to the same node, affording almost no speedup. For this reason, we incorporate a queuing model to accommodate dynamic task allocation.

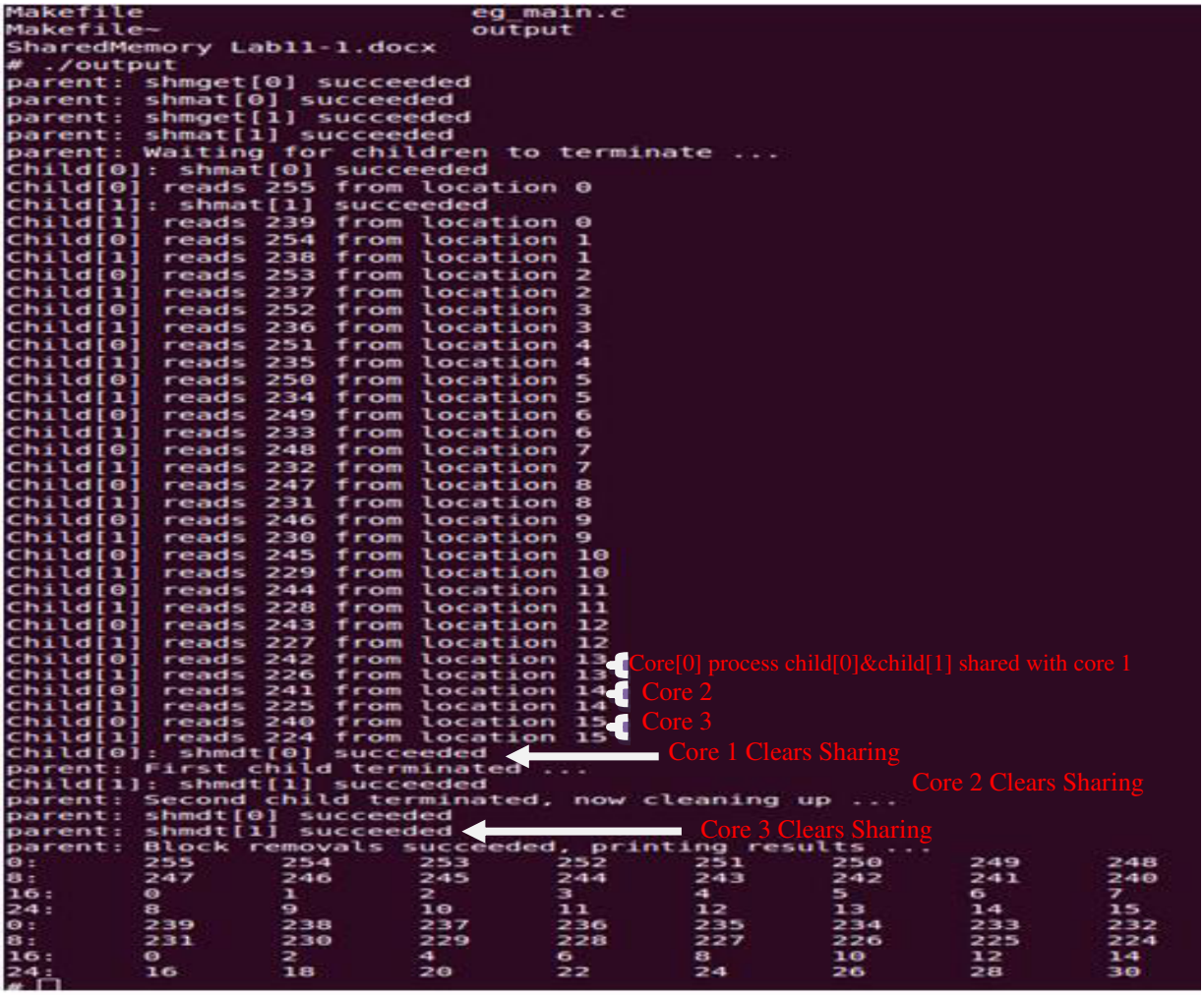

Figure-7. Child process at runtime memory block de-allocation.

Table-3. Allocated/de-allocated memory space in child/parent processor.

Location (CORE 16,24)	**Child** (0) (CORE 0)	**Child** (1) (CORE 8)	**Location** (CORE 16,24)	**Child** (0) (CORE 0)	**Child** (1) (CORE 8)
0	255	239	8	247	231
1	254	238	9	246	230
2	253	237	10	245	229
3	252	236	11	244	228
4	251	235	12	243	227
5	250	234	13	242	226
6	249	233	14	241	225
7	248	232	15	240	224

7. CONCLUSION AND FUTURE WORK

The initiation of multi-core architectures has successfully solved share memory of de-allocation in the large database modules between the both child(Client) and parent(Server) processor or any multi kernel processors using OpenMP Linux Kernel file Lock-Free algorithms on threading in multi core processor scheduling. The scheduler for shared memory de-allocation with among multiple cores concept has received/sending block data between the Child processor to Parent processor for reallocating the empty blocks to communicate as soon as possible with one processor to other process for reducing the time delay vice versa with assigning the cores list are changing the tasks. The 32 bit cores location based on the input and output files has placed on the parallel on the multithreading concept for Future work the large database

mining and With this module we will develop the large module getting the input across the OS developed socket had installed in Linux Multi Kernel threads/program. The Future Work in memory in Swapping if using static de-allocation, code/data must return to same place, In dynamic, can reallocate at more advantageous memory in the multi-core schedule**.**

REFERENCES

[1] Shamim Yousefi, Samad Najjar Ghabel, Leyli Mohammad Khanli. 2015. Modeling Causal Consistency in a Distributed Shared Memory using Hierarchical Colored Petri Net. Indian Journal of Science and Technology. 8(33), Doi no:10.17485/ijst/2015/v8i33/75502.

[2] Sowmithra Vennelakanti, S. Saravanan. 2016. Design and Analysis of Low Power Memory Built in Self Test Architecture for SoC based Design. Indian Journal of Science and Technology. 8(14), Doi no:10.17485/ijst/2015/v8i14/62716.

[3] A. J. Rajeswari Joe, N. Rama. 2015. Neural Network based Image Compression for Memory Consumption in Cloud Environment. Indian Journal of Science and Technology. 8(15), Doi no:10.17485/ijst/2015/v8i15/73855.

[4] M. Hemamalini, M. V. Srinath. 2015. Memory Constrained Load Shared Minimum Execution Time Grid Task Scheduling Algorithm in a Heterogeneous Environment. Indian Journal of Science and Technology. 8(15), Doi no:10.17485/ijst/2015/v8i15/71373.

[5] Wenjing L, Lisheng W. 2011. Energy-Considered Scheduling Algorithm Based on Heterogeneous Multi-core Processor. 2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC), Jilin pp.1151-54.

[6] Tan P, Shu J, Wu Z. 2010. A Hybrid Real-Time Scheduling Approach on Multi-Core Architectures. Journal of software. 5: 958-65.

[7] Anderson G, Marwala T, Nelwamondo F V. 2013. Multicore scheduling based on learning from optimization models. International Journal of innovative computing, information and control, icic international. 9: 1511-22.

[8] Kumar R, Tullsen D M, Ranganathan P, Jouppi N P, Farkas K I. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. Proceedings of the 31st annual international Symposium on Computer Architecture, ISCA. pp. 64-75.

[9] Broquedis F, Diakhate F, Thibault S, Aumage O. 2008. Scheduling Dynamic Open MP Applications over Multicore Architectures, OpenMP in a New Era of Paralleism, Springer-Berlin: Heidelberg. pp. 170-180.

[10] Geng X, Xu G, Fu X, Zhang Y. 2012. A task scheduling algorithm for multi-core- cluster systems journal of computers. 7(11): 2797-804.

[11] Chen Y-S, Liao H C, Tsai T H. 2013. Real-Time Task Scheduling in Heterogeneous Multi-core System-on-a-Chip. IEEE Trans on parallel and distributed systems.

[12] Dhotre P S S, Patil S. 2014. Task Management for Heterogeneous Multi-core Scheduling. (IJCSIT) International Journal of Computer Science and Information Technologies. 5(1): 636-639.

[13] Saifullah A, Agrawal K, Lu C, Gill C. 2013. Multi-core Real-Time Scheduling for Generalized Parallel Task Models. Real-Time Systems. 49(4): 404-35.

[14] Ritson C G, Sampson A T, Barnes F R M. Multicore Scheduling for lightweight communicating processes coordination, Coordination Models and Languages, Springer: Berlin Heidelberg. pp. 163-183.

[15] Mishra G, Gurumurthy K S. 2014. Dynamic task scheduling on multicore automotive, International journal of vlsi design and communication systems (vlsics).

[16] Malhotra S, Narkhede P, Shah K, Makaraju S, Shanmugasundaram M. 2015. A review of fault tolerant scheduling in multicore systems international journal of scientific and technology research. pp. 132-136.