© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

A Review on multithreading processes and threads in multiple cores CPU

[1]Rupali, Research Scholar, Department of CSA, CDLU Sirsa
[2]Ms. Shailja kumari. Assistant Professor , Department of CSA, CDLU Sirsa, SSK.88@rediffmail.com

ABSTRACT: The objective of our research is to analyze job handling process of CPU in different circumstances. Here we would analyze how CPU reacts in case of single task & in case when it switches among multiple tasks & how multiple task are managed as thread within a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by operating system. This approach differs from multiprocessing, as with multithreading processes & threads have to share resources of a single or multiple cores: computing units, CPU caches, & translation lookaside bufferA scheduler may aim at one of several goals, for example, maximizing *throughput*, minimizing *response time* or minimizing *latency*, maximizing *fairness* (equal CPU time to each process, or more generally appropriate times according to priority & workload of each process). All these goals often conflict thus a scheduler would implement a suitable compromise. Preference is given to any one of concerns mentioned above, depending upon user's needs & objectives.

© iJRPS International Journal for Research Publication & Seminar

Keywords: Thread, TLB,CPU, Throughput, scheduler, multithreading, SMT

[1] INTRODUCTION

CPU is electronic circuitry within a computer that carries out instructions of a computer program by performing basic arithmetic, logical, control & input/output (I/O) operations specified by instructions. term has been used in computer industry at least since early 1960s. Traditionally, term CPU refers to a processor, more specifically to its processing unit & control unit (CU), distinguishing these core elements of a computer from external components such as main memory & I/O circuitry.

The form, design & implementation of CPUs have changed over course of their history, but their fundamental operation remains almost unchanged. Principal components of a CPU include arithmetic logic unit that performs arithmetic & logic operations, processor registers that supply operands to ALU & store results of ALU operations, & a control unit that fetches instructions from memory & executes them by directing coordinated operations of ALU, registers & other components.

Most modern CPUs are microprocessors, meaning they are contained on a single integrated circuit (IC) chip. An IC that contains a CPU may also contain memory, peripheral interfaces, & other components of a computer; such integrated devices are variously called microcontrollers or systems on a chip. Some computers employ a multi-core processor, which is a single chip containing two or more CPUs called "cores"; in that context, single chips are sometimes referred to as "sockets".Array processors or vector processors have

© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

multiple processors that operate in parallel, with no unit considered central.

MULTITHREADING

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since late 1990s. This allowed concept of throughput computing to re-emerge from more specialized field of transaction processing; even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve throughput of all tasks result within overall performance gains.

Types of multithreading

Block multithreading

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, that might take hundreds of CPU cycles for data to return. Instead of waiting for stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when data for previous thread had arrived, would previous thread be placed back on list of ready-to-run threads.

For example:

1. Cycle *i*: instruction *j* from thread *A* is issued.
2. Cycle *i* + 1: instruction *j* + 1 from thread *A* is issued.
3. Cycle *i* + 2: instruction *j* + 2 from thread *A* is issued, that is a load instruction that misses within all caches.
4. Cycle *i* + 3: thread scheduler invoked, switches to thread *B*.
5. Cycle *i* + 4: instruction *k* from thread *B* is issued.
6. Cycle *i* + 5: instruction *k* + 1 from thread *B* is issued.

2.LITERATURE REVIEW

***Yeh-Ching Chung wrote on* “Applications & Performance Analysis of A Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors”**

They have proposedacompile-time optimization approach, *bottom-up top-down duplication heuristic* (BTDH), for static scheduling of *directed+cyclic graphs* (DAGS) on *distributed memory multiprocessors* (DMMs). In this paper, they discuss applications of BTDH for *list scheddhg algorithms* (LSAs). There are two ways to use BTDH for LSAs.BTDHcan be used with aLSAto form a new *scheduling* algorithm (LSA/BTDH). It could be usedas apure *opti*mization algorithm for a LSA (LSA-BTDH). We have applied BTDH with two well known LSAs, *highest level first with estimated time* (HLFET) & *earlier taskfirst* (ETF) heuristics. We have performed extensive simulation to study performance of BTDH for LSAs. Three parameters, *graph parallelism* (GP) of a DAG, ratio of average communication cost to average computation cost *(CCR)* of a DAG & number(PN) of a DMM,aresimulated. From sunulation, they have following conclusions. Given a DAG, GP of DAG could accurately predict number of processors to be used such that a good scheduling length & a good resource utilization (or efficiency) could be achieved
.

Ishfaq Ahmad1 & Yu-Kwong Kwok2 wrote on "On Parallelizing Multiprocessor Scheduling Problem"

Existing heuristics for scheduling a node & edge weighted directed task graph to multiple processors could produce satisfactory solutions but incur high time complexities that tend to exacerbate within more realistic environments with relaxed assumptions. Consequently, these heuristics do not scale well & cannot handle problems of moderate sizes. A natural approach to reducing complexity while aiming for a similar or potentially better solution is to parallelize scheduling algorithm. This could be done by partitioning task graphs & concurrently generating partial schedules for partitioned parts, that are then concatenated to obtain final schedule. problem, however, is nontrivial as there exists dependencies among nodes of a task graph that must be preserved for generating a valid schedule. Moreover, time clock for scheduling is global for all processors (that are executing parallel scheduling algorithm), making inherent parallelism invisible. In this paper, they introduce a parallel algorithm that is guided by a systematic partitioning of task graph to perform scheduling using multiple processors. algorithm schedules both tasks & messages, & is suitable for graphs with arbitrary computation & communication costs, & is applicable to systems with arbitrary network topologies using homogeneous or heterogeneous processors. We have implemented algorithm on Intel Paragon & compared it with three closely related algorithms. experimental results indicate that our algorithm yields higher quality solutions while using an order of magnitude smaller scheduling times. algorithm also exhibits an interesting trade-off between solution quality & speedup while scaling well with problem size.

Maruf Ahmed, Sharif M. H. Chowdhury wrote on List Heuristic Scheduling Algorithms for Distributed Memory Systems with Improved Time Complexity

They present a compile time list heuristic scheduling algorithm called *Low Cost Critical Path algorithm (LCCP)* for distributed memory systems. LCCP has low scheduling cost for both homogeneous & heterogeneous systems. In some recent papers list heuristic scheduling algorithms keep their scheduling cost low by using a fixed size heap & a FIFO, where heap always keeps fixed number of tasks & excess tasks are inserted within FIFO. When heap has empty spaces, tasks are inserted within it from FIFO. Best known list scheduling algorithm based on this strategy requires two heap restoration operations, one after extraction & another after insertion. Our LCCP algorithm improves on this by using only one such operation for both extraction & insertion, that within theory reduces scheduling cost without compromising scheduling performance. In our experiment they compare LCCP with other well known list scheduling algorithms & it shows that LCCP is fastest among all.

Wayne F. Boyer wrote on "Non-evolutionary algorithm for scheduling dependent tasks within distributed heterogeneous computing environments"

The Problem of obtaining an optimal matching & scheduling of interdependent tasks within distributed heterogeneous computing (DHC) environments is well known to be an NP-hard problem. In a DHC system, task execution time is dependent on machine to which it is assigned & task precedence constraints are represented by a directed acyclic graph. Recent research within evolutionary techniques has shown that genetic algorithms usually obtain more efficient schedules that other known algorithms.

[3] **RESEARCH METHODOLOGY**

In computing, **scheduling** is method by which work specified by some means is assigned to resources that complete work. work may be virtual computation elements such as threads, processes or data flows, that are within turn scheduled onto hardware resources such as processors, network links or expansion cards.

A scheduler is what carries out scheduling activity. Schedulers are often implemented so they keep all computer resources busy (as within load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, & an intrinsic part of execution model of a computer system; concept of scheduling makes it possible to have computer multitasking with a single central processing unit (CPU).

In advanced packet radio wireless networks such as HSDPA (High-Speed Downlink Packet Access) 3.5G cellular system, **channel-dependent scheduling** may be used to take advantage of channel state information. If channel conditions are favourable, throughput & system spectral efficiency may be increased.
In even more advanced systems such as LTE, scheduling is combined by channel-dependent packet-by-packet dynamic channel allocation, or by assigning OFDMA multi-carriers or other frequency-domain equalization components to users that best could utilize them.

Highly simplified structure of Linux kernel: *process scheduler*, *I/O scheduler*, *packet scheduler* & other subsystems

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, & cooperative for processes & threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, & cooperative for 16-bit processes

Windows NT (including 2000, XP, Vista, 7, & Server)	Yes	Multilevel feedback queue

4. Challenges within research

Multiple threads could interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved but could be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.
Overall efficiency varies; Intel claims up to 30% improvement with its HyperThreading technology,[1] while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run within parallel. On other hand, hand-tuned assembly language programs using MMX or Altivec extensions & performing data pre-fetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading & could indeed see degraded performance due to contention for shared resources.
From software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs & operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel software techniques used for computer multitasking of computer programs. Thread scheduling is also a major problem within multithreading.

5. PROPOSED WORK

Existing gang-h algorithm

```
input :
n jobs Jj (uj , vj ), $1 \le j \le n$ ;
m: number of processors;
output:
Slice lengths S(s), s = 1, 2, . . .;
Scheduled jobs j in slice s: Sched(s,j) $\in \{0, 1\}$, $1 \le j \le n$ ;
List=Sort(J1, . . . , Jn) ; /* Jobs are sorted in non increasing order of vj */
s = 0 ; /* Number of slices */
rj := uj $\forall j = 1 \cdots n$ ; /* Job remaining execution times rj */
while $\exists j, rj > 0, 1 \le j \le n$ do
/* create a new slice */
s = s + 1 ; /* Number of slices */
K = m ; /* Remaining processors */
` = $\infty$ ; /* Slice length upper bound */
```

```
Sched(s,j)=0 1 ≤ j ≤ n ; /* Empty Slice */
foreach j ∈ List do
/* For each job j in priority List */
if vj ≤ K then
/* job j is schedulable in current slice */
Sched(s,j)=1;
` = min(`, rj ); /* update slice length */
K = K − vj ; /* remaining processors */
end
end
S(s)=l; /* Slice length */
for j = 1 . . . n do
/* Update remaining execution times */
if Sched(s, j) then
rj = rj − `;
end
end
end
```

Performance of above algorithm could be improved in two ways :

1. By customizing hardware
2. By customizing existing algorithm
   Hardware Customization consist of following
   1. Addition of Multi-core processors
   2. Addition of Primary Memory
   3. Addition of Cache

Customization of existing algorithm

1. Create clusters of Data to be fetch
2. Data at nearest memory location could be clustered (grouped).
3. In this way scheduled processes may be grouped & processed in Batches.
4. By Batch processing performance would be better as latency time of cpu would reduce

Proposed work

In this work we will group jobs in cluster for batch processing.

6. SCOPE OF RESEARCH

Number of job can run simultaneously of computer and all these job put burden on CPU. The objective of research is to minimize the burden of CPU and enhance its capability by concentration of similar jobs clustered in cache so that the processing time get reduced.

If a thread gets a lot of cache misses, other threads could continue taking advantage of unused computing resources, that may lead to faster overall execution as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all computing resources of CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle. If several threads work on same set of data, they could actually share their cache, leading to better cache usage or synchronization on its values.

REFERENCE

1. *Remzi H. Arpaci-Dusseau; Andrea C. Arpaci-Dusseau (January 4, 2015). "Chapter 7: Scheduling: Introduction, Section 7.6: A New Metric: Response Time". Operating Systems: Three Easy Pieces (PDF). p. 6. Retrieved February 2, 2015.*
2. *Paul Krzyzanowski (2014-02-19). "Process Scheduling: Who gets to run next?". cs.rutgers.edu. Retrieved 2015-01-11.*
3. *Abraham Silberschatz, Peter Baer Galvin & Greg Gagne (2013). Operating System Concepts* ***9****. John Wiley & Sons,Inc. ISBN 978-1-118-06333-0.*
4. Here is C-code for FCFS
5. Early Windows at Wayback Machine
6. *Sriram Krishnan. "A Tale of Two Schedulers Windows NT & Windows CE".*
7. Inside Windows Vista Kernel: Part 1, Microsoft Technet
8. *"Vista Kernel Improvements".*
9. *"Technical Note TN2028 - Threading Architectures".*
10. *"Mach Scheduling & Thread Interfaces".*
11. http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6
12. *Molnár, Ingo (2007-04-13). "[patch] Modular Scheduler Core & Completely Fair Scheduler [CFS]". linux-kernel (Mailing list).*