I J C T A, 9(5), 2016, pp. 2323-2331
© International Science Press

Interval Timers for Process Handling

Rajamohan L.* and Ravi S.**

ABSTRACT

An external asynchronous event may be either signals or software interrupts, used to alter the course of a program. These may occur at any time during the execution of a program and differ from other methods of inter-process communication. Processor scheduling delays prevent the process from handling the signal as soon as it is generated. This problem can be handled by implementing a timer such that each timer contains a field that indicates how far in the future the timer should expire. Linux considers two types of timers, dynamic and interval timer. Dynamic timer is used by kernel while interval timer can be created by processes in user mode. It can ensure that all processes (both parent and child) are executed either at proper time or after a delay. Additionally, to suit real-time application, the timers are designed with expiration time strictly enforced. This paper discusses how to track the passage of time using different kinds of alarm signals and manage the child process creation and avoid Zombie situations.

Keywords: Timers, Alarm signal, Profile, Virtual process timer, Dynamic thread affinity.

1. INTRODUCTION

Tasks can be dynamically scheduled for execution based on the mutual dependencies and on the computational resources available. The dynamic runtime system efficiently schedules the implemented kernels across the processing units & ensures the data dependencies are not violated. Linux supports process specific *interval* timers. A process can use these timers to send itself various signals each time that they cease. In this work, three types of interval timers are supported and are listed in Table 1.

The state of a timer is described by the interval_timer_status type which is a record with two fields (each a float) representing time:

The field it_interval is the period of the timer.

The field it_value is the current value of the timer; when it turns 0, the signal sigvtalrm is sent and the timer is reset to the value in it_interval.

A timer is therefore inactive when its two fields are 0 (as listed in Table 2).

Table 1
Different Timer intervals and their representation

Timer	*Representation*	*Value of type*	*Function*
Real	ITIMER_REAL	Real time (sigalrm)	The timer ticks in real time, and when the timer has expired, the process is sent a SIGALRM signal.
Virtual	ITIMER_VIRTUAL	User time (sigvtalrm)	This timer only ticks when the process is running and when it expires it sends a SIGVTALRM signal
Profile	ITIMER_PROF	User time and system time (sigprof)	This timer ticks both when the process is running and when the system is executing on behalf of the process itself. SIGPROF is signaled when it expires.

* Research Scholar, Department of ECE, St. Peter's University, Chennai, *Email: rajamohan151@gmail.com*

** Professor and Head, Department of ECE, Dr. M.G.R. Educational and Research Institute University, Madhuravoyal, Chennai, *Email: ravi_mls@yahoo.com*

Table 2
Timer state descriptions

it_value	*it_interval*	*Inference*
≠0	≠0	Indicates time to the next timer expiration and reloading it_value
0	X	Disables the timer
≠0	0	Disables timer after its next expiration

One or all of the interval timers may be running and Linux keeps all of the necessary information in the process's task_struct data structure. Every clock tick the current process's interval timers are decremented and, if they have expired, the appropriate signal is sent. This generates the SIGALRM signal and restarts the interval timer, adding it back into the system timer queue. The alarm () function may cause the system to generate a SIGALRM signal for the process after the number of real-time seconds specified by seconds have elapsed. If seconds is 0, a pending alarm request, if any, is canceled. Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time at which the SIGALRM signal is generated.

2. RELATED WORKS

Kuperberg M. et al. (2009) analysed runtime behaviour and performance of software systems, accurate time measurements are obtained using timer methods. The underlying hardware timers and counters are read and processed by several software layers, which introduce overhead and delays that impact accuracy and statistical validity of fine-granular measurements. To understand and to control these impacts, the resulting accuracy of timer methods and their invocation costs must be quantified. However, quantitative properties of timer methods are usually not specified as they are platform-specific due to differences in hardware, operating systems and virtual machines. Also, no algorithm exists for precisely quantifying the timer methods' properties, so programmers have to work with coarse estimates. In this paper, we present TimerMeter, a novel algorithm for platform-independent quantification of accuracy and invocation cost of any timer methods, without inspecting their implementation.

Chaturvedi S.K. (2011) suggested multiple researches are proposed on to provide fairness and protection to the packet flow through a router, A General Processor Sharing (GPS) has been used as a conceptual scheduler with many desirable properties, GPS supports guaranteed service traffic and to provide best-effort service traffic. A novel data structure called Interleaved Stratified Timer Wheels (ISTW) is introduced. This design enables the construction of a set of novel packet schedulers with effectively constant complexity, constant fairness and delay characteristics in all relevant dimensions. The ISTW data structure is used as a compact and efficient priority queue that enables the virtual traffic shaping necessary for achieving these characteristics. ISTW parallelization of the GPS is done.

Lawson G. (2014) The Intel Xeon Phi coprocessor offers high parallelism on energy-efficient hardware to minimize energy consumption while maintaining performance. Dynamic frequency and voltage scaling is not accessible on the Intel Xeon Phi. Hence, saving energy relies mainly on tuning application performance. One general optimization technique is thread affinity, which is an important factor in multi-core architectures. This work investigates the effects of varying thread affinity modes and reducing core utilization on energy and execution time for the NASA Advanced Supercomputing Parallel Benchmarks (NPB). The measurements are checked against total power captured using Watts up power meters. The results are compared to the system-default thread affinity and granularity modes. Mostly positive impacts on performance and energy are observed: When executed at the maximum thread count on all unoccupied cores, all the benchmarks but one exhibited energy savings if a specific affinity mode is set.

Roy A. 2014 proposed large-scale cache-coherent systems often impose unnecessary overhead on data that is thread-private for the whole of its lifetime. These include resources devoted to tracking the coherence state of the data, as well as unnecessary coherence messages sent out over the inter-connect. In this paper we show how the memory allocation strategy for non-uniform memory access (NUMA) systems can be exploited to remove any coherence-related traffic for thread-local data, as well removing the need to track those cache lines in sparse directories. Solution is entirely backward compatible with existing operating systems and software, and provides a means to scale cache coherence into the many-core era. On a mix of SPLASH2 and Parsec workloads, ALLARM is able to improve performance by 13% on average while reducing dynamic energy consumption by 9% in the on-chip network and 15% in the directory controller. This is achieved through a 46% reduction in the number of sparse directory entries evicted.

Zijiang Yang, et al. (2013) proposed this paper as the statistical analysis for alarm signals in order to detect whether two alarm signals are correlated. First, a similarity measurement, namely, Sorgenfrei coefficient, is selected among 22 similarity coefficients for binary data in the literature. The selection is based on the desired properties associated with specialities of alarm signals. Second, the distribution of a so-called correlation delay is shown to be indispensable and effective for the detection of correlated alarms. Finally, a novel method for detection of correlated alarms is proposed based on Sorgenfrei coefficient and distribution of the correlation delay. Numerical and industrial examples are provided to illustrate and validate the obtained results.

Shraddha et al. (2012) introduced Real Time Operating System to new comers the major research trends identified. After describing the characteristics of modern embedded applications, the paper presents the problems of the current approaches and discusses the new research trends in real time operating systems and scheduling emerging to overcome them. Most of today’s embedded systems are required to work in dynamic environments, where the characteristics of the computational load cannot always be predicted in advance. Still timely responses to events have to be provided within precise timing constraints in order to guarantee a desired level of performance. Hence, embedded systems are, by nature, inherently real-time. Moreover, most of embedded systems work under several resource constraints, due to space, weight, energy, and cost limitations imposed by the specific application.

3. IMPLEMENTATION OF PROCESS TIMERS

The implementation of the timers has been designed to meet the following requirements and assumptions:

- Timer management must be as lightweight as possible.
- The design should scale well as the number of active timers increases.
- Most timers shall be time bound expire within a few seconds or minutes at most and timers with long delays are scheduled to avoid resource contention.
- A timer should run on the same CPU that registered it to achieve maximum speed.

A task queue is a list of tasks, each task being represented by a function pointer and an argument. When a task is run, it receives a single void * argument and returns void. The pointer argument can be used to pass along a data structure to the routine, or it can be ignored. The queue itself is a list of structures (the tasks) that are owned by the kernel module declaring and queuing them. The module is completely responsible for allocating, de-allocating the structures and static structures are commonly used for this purpose.

```
Structtq_struct {
Structtq_struct *next;
int sync;
void(*routine)(void*);
void *data;
};
```

The "bh" is meant for bottom half, i.e., "half of an interrupt handler". A bottom half is a mechanism provided by a device driver to handle asynchronous tasks which, usually, are too large to be done while handling a hardware interrupt. The most important fields in the data structure shown above are routine and data. To queue a task for later execution, it is needed to set these fields before queuing the structure, while next and sync should be cleared. The sync flag in the structure is used by the kernel to prevent queuing the same task more than once, because this would corrupt the next pointer. Once the task has been queued, the structure is considered "owned" by the kernel and should not be modified until the task is run.

3.1. Benefits Of Virtual Process Timers

3.1.1. Case (i): TO REMOVE DEADLOCK

Virtual process timers can overcome deadlock situations using scheduling types. Deadlock can occur due to process running in single core and use of more virtual timers in an optimal manner can relieve this problem.

Deadlock can occur in

- Mutual exclusion: only one process at a time can use a resource.
- Hold and wait: A process holding at least one resource which is waiting to acquire additional resources held by other processes
- No preemption: A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait: there exists a set $\{P0, P1, \ldots, P0\}$ of waiting processes such that $P0$ is waiting for a resource that is held by $P1$, $P1$ is waiting for a resource that is held by $P2$, …, $Pn - 1$ is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by $P0$.

The Deadlock due to circular wait and the process dependency is shown in Figure 1 and table 3. From table 3, it can be observed that the wait among the processes $P_{1,} P_{2,} P_3$ & P_4 are cyclic ($P_{1,} P_{2,} P_{3,} P_{4,} P_1$) and this results in a deadlock. To overcome the deadlock, a virtual timer to handle process P_2 is used as shown in Figure 2. The new dependency is shown in table 4, where there is no deadlock.

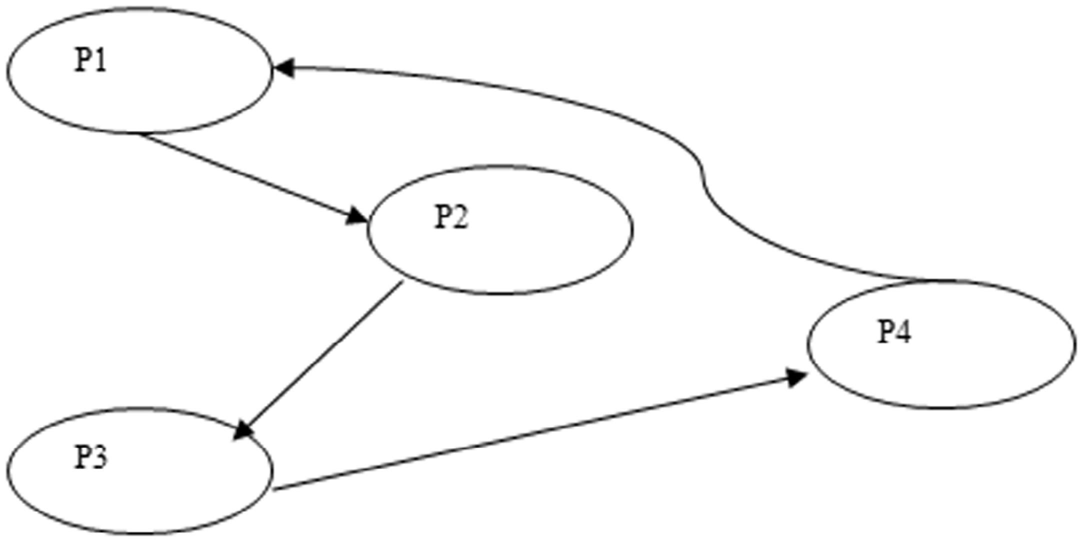


Figure 1: Dead lock condition

Table 3
Process Dependency in the case of Deadlock

Process	*parent*	*child*
1	4	2
2	1	3
3	2	4
4	3	1

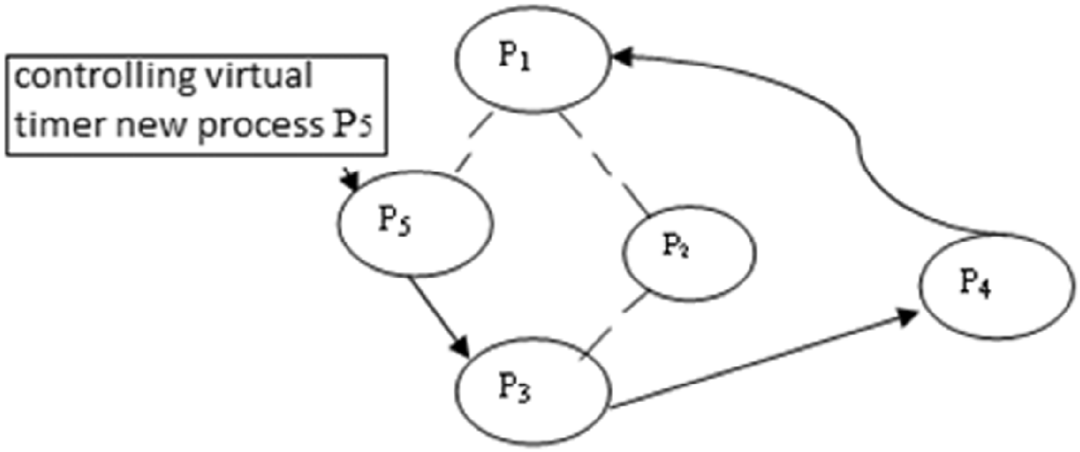


Figure 2: Elimination of deadlock

Table 4
Process Dependency after addition of virtual timer

Process	*parent*	*Child*
1	4	5 (VIRTUAL TIMER)
5	1	3
3	5	4
4	3	1

A general resource system is characterized by:

- A nonempty set of processes $\prod = \{P_1, P_2, \ldots P_n\}$.
- A nonempty set of resources $\Gamma = \{R_1, R_2, \ldots R_m\}$. Γ can be partitioned into two disjoint sets: Γ_r, a set of reusable resources and Γ_c, a set of consumable resources.
- For every reusable resource R_i, there exists a nonnegative integer t_i denoting the total number of units of the resources present in the system.
- For every consumable resource R_i, there exists a non empty subset of processes of $\prod$, called the producers of R_i.
- For every reusable resource $R_{i:}$
- The number of assignment edges, $\#(R_i, *) \leq t_i$
- $r_i = t_i - \#(R_i, *)$
- ¥P_j: $P_j \sum \prod$:: $\#(P_j, R_i) + \#(R_i, P_j) \leq t_i$. That is, at any instant a process cannot request more than the total units of reusable resource.

For every consumable resource R_i:

- There is an edge from R_i to a process P_j iff P_j is a producer of R_i.
- $r_i \geq 0$

A general resource graph can be reduced in the following way by a process P_i, which is not blocked.

- For each reusable resource R_j, delete all edges (P_i, R_j) and (R_j, P_i) from the graph. For each assignment edge (R_j, P_i) deleted, increase r_j by one.
- For each consumable resource R_j,

a. Decrement r_j by the number of edges (P_i, R_j).

b. If P_i is a producer of R_j, then set r_j to ∞□.

c. Delete all edges (P_i, R_j) and (R_j, P_i).

If the system is in safe state then no deadlock. If the system is in unsafe state, possibility of deadlock. Their avoidance ensures that a system will never enter an unsafe state.

Safety algorithm:

1. Let work and Finish be vectors of length m and n, respectively. Initialize Work: = Available and Finish[i]:= False for i = 1, 2 ... n
2. Find i such that both Finish [i] = false, Need [i] = True.
3. Work: = Work + Allocation

i. Finish [i]: = True

ii. Goto step 2.

4. If Finish [i] = True for all i, then the system is in a safe state.

Safe State Checking algorithm:

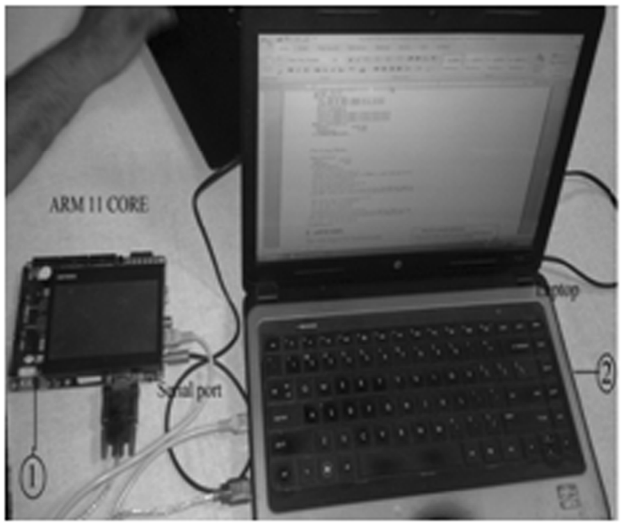
1. Pick an unfinished process P_i such that $E_i \leq D$. If no such process exists, then go to step 3.
2. D: = $D + C_i$. Tag P_i as finished. Go to step 1.
3. If all processes are tagged "finished", the current system state is a safe state; otherwise, it is not a safe state.

3.1.2. Case (ii) Dynamic Thread Affinity

The Dynamic Thread affinity has been studied in shared memory with various views in Linux and windows operating systems introduced new affinity level system calls. AMD & INTEL compilers allow programmer to control thread binding by using modules. Dynamic Thread affinity cache between processor to processor can have a dramatic effect on the application speed. Dynamic Thread affinity restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor. Dynamic Thread affinity is supported on Windows OS systems and versions of Linux OS systems that have kernel support for dynamic thread affinity.

4. HARDWARE IMPLEMENTATION

The advantage of using ARM 11 core processor is low dynamic power consumption, lower standby power (static power) and maturity. ARM processor implements a timer which is needed to set the least permissible value, so that the user can set the timer expiry as per his needs and still get the results. The thing is if timer



interval is high, then chances are execution of test program can end before the timer tick. If the interval is very low (like 1ns), then even though the API takes 1ns value, but chances are the ARM processor may not accept the 1ns value. It is necessary to set it to the least permissible value accepted by processor. The hardware implementation program codes are shown in appendix.

1. (a) ARM core with root file system mounted in Sd-card
   (b) Linux Kernel mounted as image file
2. Hyper Terminal to view the results.

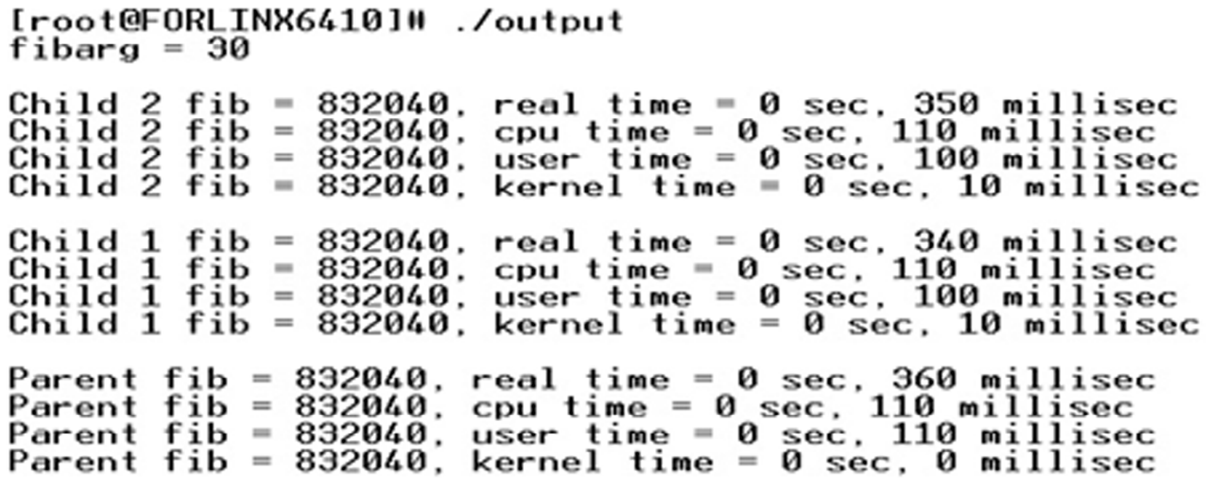
5. RESULTS AND DISCUSSION

All forms of UNIX make use of signals, which are mechanisms used by a process to notify another process that an event has occurred. Some examples are the user pressing the delete key, the run time system detecting an attempt to divide by zero, or one of the itimers reaching zero. Upon receiving a signal, the process executes code for that particular signal. Note that signals can be used between application processes. There are different categories of time such as real-time (i.e. wall-clock time), processor time (the time that a process is actually running in both user and kernel space), user space time and kernel space time.

For implementing Fibonacci number, the method used in this work is creating an array of Fibonacci using recursive method. The Nth number in Fibonacci series is $N = 30$ by default and $N = 35$, the user value. As an example of calculation, for child 2 process, CPU time is calculated by adding user time and kernel time, 100ms + 10ms = 110ms. It is done in the same way for parent and child 1 processes respectively. It shows the outputs of a C program using the fork() system call that generates the Fibonacci sequence in the child process and then have the parent process to print out the sequence. Make the processes share memory (mutual exclusion issue) in addition to making the parent process wait for the child process to finish

Table 5
Comparison between process execution

Features	*Process 1*	*Process 2*
Value of N	30	35
Nature of Value	Default	User
Child process	Value of real time is incremented with value of kernel time	Value of real time is double the value of CPU time
Parent process	Similar as child process	Real time executes in few milliseconds



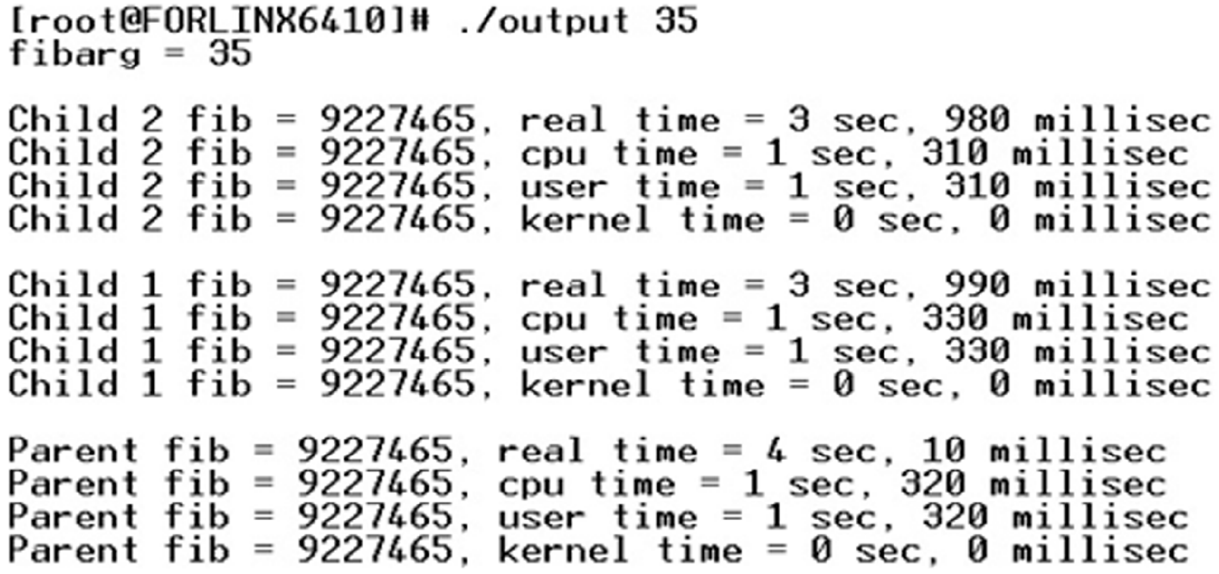
```
[root@FORLINX6410]# ./output
fibarg = 30

Child 2 fib = 832040, real time = 0 sec, 350 millisec
Child 2 fib = 832040, cpu time = 0 sec, 110 millisec
Child 2 fib = 832040, user time = 0 sec, 100 millisec
Child 2 fib = 832040, kernel time = 0 sec, 10 millisec

Child 1 fib = 832040, real time = 0 sec, 340 millisec
Child 1 fib = 832040, cpu time = 0 sec, 110 millisec
Child 1 fib = 832040, user time = 0 sec, 100 millisec
Child 1 fib = 832040, kernel time = 0 sec, 10 millisec

Parent fib = 832040, real time = 0 sec, 360 millisec
Parent fib = 832040, cpu time = 0 sec, 110 millisec
Parent fib = 832040, user time = 0 sec, 110 millisec
Parent fib = 832040, kernel time = 0 sec, 0 millisec
```

Processes executing Fibonacci program for default value *N* = 30:



```
[root@FORLINX6410]# ./output 35
fibarg = 35

Child 2 fib = 9227465, real time = 3 sec, 980 millisec
Child 2 fib = 9227465, cpu time = 1 sec, 310 millisec
Child 2 fib = 9227465, user time = 1 sec, 310 millisec
Child 2 fib = 9227465, kernel time = 0 sec, 0 millisec

Child 1 fib = 9227465, real time = 3 sec, 990 millisec
Child 1 fib = 9227465, cpu time = 1 sec, 330 millisec
Child 1 fib = 9227465, user time = 1 sec, 330 millisec
Child 1 fib = 9227465, kernel time = 0 sec, 0 millisec

Parent fib = 9227465, real time = 4 sec, 10 millisec
Parent fib = 9227465, cpu time = 1 sec, 320 millisec
Parent fib = 9227465, user time = 1 sec, 320 millisec
Parent fib = 9227465, kernel time = 0 sec, 0 millisec
```

Processes executing Fibonacci program for user value *N* = 35:

(synchronization issue). The parent invokes the wait() call to wait for the child process to complete before exiting the program. Necessary error checking is performed to ensure that a non-negative number is passed on the command line. The comparison between process execution is listed in table 5.

6. CONCLUSION

Tasks are dynamically scheduled based on resources available and interdependencies. Scheduling delays prevented the process from handling the signal generated. Dynamic runtime system schedules kernel efficiently. When a process starts execution, it first initiates the child process and when all child process execution gets completed the parent process starts its execution. Different child process executes in parallel and only when all child process completes its execution and terminates, parent process terminates. During child process creation, different kinds of alarm signals are kept in track of passage of time to ensure that memory resources are allocated and deallocated properly do not get blocked due to Zombie.

REFERENCES

[1] Agarwal S., Barik R., Shyamasundar R.K. (2008), "A Static characterization of affinity in distributed program", International conference on high performance computing and communications, pp. 572-579.

[2] Chaturvedi S.K. (2011), “Optimization of generalized processor sharing using Interleaved Stratified timer wheels”, International conference on signal processing, communication, computing and networking technologies”, pp. 105-110.

[3] Das. A., Shafik. R.A., Merret. G.V. (2014), “Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems”, Design Automation conference, pp. 1-6.

[4] Guo, C. (2004), “SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks’, IEEE/ ACM Transactions on Networking, vol. 12, no. 6, pp. 1144-1155

[5] Kuperberg M., Krogmann M., Reussner R., (2009), “ Timer Meter: Quantifying properties of software timers for system analysis”, International conference on quantitative evaluation of systems, pp. 85-94.

[6] Laas-Bourez M., Courde C. (2015), “Accuracy validation of T2L2 time transfer in co-location”, IEEE Transcations on Ultrasonics, Ferro Electrics and Frequency control, Vol. 62, Issue 2, pp. 255-265.

[7] Lawson G., Sosonkina M. (2014), “Energy evaluation for application with different thread affinities on IntelXeon Phi”, International conference on Computer architecture and high performance computing workshop, pp. 54-59.

[8] Mishra V.K, Mehta D.K. (2013), “Performance Enhancement of NUMA multiprocessor systems with on-demand memory migration”, International Advance Computing conference, pp. 40-43.

[9] Patel A., Daftedar M. (2015), “Embedded Hypervisor Xvisor: A comparative analysis”, International Conference on parallel, distributed and network based processing, pp. 682-691.

[10] Roy A., Jones T.M. (2014), “ALLARM: Optimizing sparse directories for thread local data”, Design, Automation and Test in Europe Conference, pp. 1-6.

[11] Shraddha, S., Nakate, Bandu, B & Meshram (2012), “New Trends in Real Time Operating Systems’, IOSR Journal of Engineering, vol. 2, no. 4, pp. 883-992

[12] Tinotenda Zwavashe & Vasumathi, D. (2012), “Polling, Interrupts & μ COS-II: A Comparative Timing Response Simulation Model for Wireless Processor-to-Processor Communication”, International Journal of Science and Research, vol. 3, no. 7, pp. 116-122

[13] Oikawa S., Tokuda H. (1995), “Efficient timing management for user level real time threads”, Real-Time Technology and application proceedings, pp. 27-32.

[14] Yue Cheng, Izadi. I, T. Ongwen Chen, (2013), “Optimal Alarm signal processing: Filter design and performance analysis”, IEEE Transactions on Automation science and Engineering, Vol. 10 Issue 2, pp. 446-451.

[15] Zijiang Yang, Jiandong Wang, Tongwen Chen (2013), “ Detection of correlated alarms based on similarity coefficients of binary data”, IEEE Transactions on Automation science and Engineering, vol. 10, Issue 4, pp. 1014-1025.