Journal of Engineering and Applied Sciences 12 (Special Issue 8): 8317-8322, 2017
ISSN: 1816-949X
© Medwell Journals, 2017

Fast Context Switches: Interrupt Driven Scheduler for ARM Cortex Platforms

[1]Varahagiri Venkata Ramya, [2]P.K. Manjunath and [1]Anju S. Pillai
[1]Department of Electrical and Electronics Engineering,
[2]Department of Electronics and Communication, Amrita School of Engineering,
Coimbatore Amrita Vishwa Vidyapeetham, Amrita University, Coimbatore, India

Abstract: The concept of context switching is extremely recognized in a multithreaded environment where different tasks are scheduled to run on a uni-core platform. However, in real time systems the overhead caused by the context switching is extremely crucial as missing the deadlines can cause serious hazards. This study addresses the overhead of context switching through hardware approach using ARM cortex M4 based TM 4 C123 GH 6 PM controller. The cortex M4 architecture provides a concept called tail chaining which can reduce the state that has to be saved and restored in the simultaneous occurrence of interrupts. The developed scheduler is capable of implementing both static and dynamic scheduling policies with reduced context switch overhead. The proposed scheduler implementation reduces the latency by reducing the context switch time to 6 clock cycles when compared to 46 clock cycles latency taken by a conventional real time operating system.

Key words: Context switches, tail chaining, ARM cortex M4 controller, scheduler, real time systems, dynamic

INTRODUCTION

In a real time application development, the implementation of the scheduler is crucial as it determines the performance of the system. There exist various types of schedulers defined by the rule they use to schedule the tasks viz., round robin, long term and short term schedulers, static and dynamic schedulers, etc. A variety of scheduling policies are available today. Few of them include: first come first service, shortest job first, priority based schedulers, etc. Due to a number of reasons priority based scheduling algorithms are popular in real time systems but has one main disadvantage that low priority tasks may undergo starvation.

The processor can service a task that requires its resources by two approaches: polling and interrupt driven request. The later approach is very much advantageous since the occurrence of tasks is unpredictable and servicing the tasks when they arrive saves lot of CPU processing time. Almost all microcontrollers facilitate this interrupt driven approach. The cortex M4 supports 16 exceptions out of which 3 have fixed priority and supports up to 256 levels of programmable priority. Thus, it can provide interrupt functionality for all the peripherals available. All these interrupt requests are handled by the Nested Vector Interrupt Controller (NVIC) which is embedded in cortex-M4 processor (Yiu, 2013). In cortex M4 when an exception takes place, the registers PC, PSR, R0-R3, R12 and LR are pushed to the stack. In conventional scheduler implementation viz., FreeRTOS, Keil RTX and atom threads, pushing and popping of registers need to be explicitly detailed using inline assembly code while in the proposed method it is taken care automatically by implementing tasks as interrupts in cortex M4 controller, thus, hiding the details at the programmer level. Also, the cortex M4 supports tail chaining of interrupts. By taking advantage of this feature, the proposed scheduler is implemented on the controller which results in lesser latency for context switches.

Literatue review: Reducing the overhead that is caused by context switching is one of the most challenging and researching areas in the field of real time systems. Some of the significant research works carried out to reduce the context switch overhead are discussed in the following section.

Kumar *et al.* (2014) proposed an improved approach to minimize context switching in round robin scheduling algorithm. In order to resolve the problems of fixed time quantum in round robin algorithm and to reduce the context switching they implemented a simplex method of operations research. Carbone (2012) proposed Preemption Threshold Scheduling (PTS) algorithm to reduce the number of context switches. This algorithm establishes a priority ceiling for disabling preemption where each

Corresponding Author: P.K. Manjunath, Department of Electronics and Communication, Amrita School of Engineering, Coimbatore Amrita Vishwa Vidyapeetham, Amrita University, Coimbatore, India

running task is given a preemption threshold. In their approach, the tasks with low priorities than the threshold cannot preempt the currently executing task (Carbone, 2012). PTS has been extended to Earliest Deadline First (EDF) scheduling (Wang and Saksena, 1999). The computation of threshold by Carbone (2012) and Wang and Saksena (1999) takes a worst case computation time of $O(n^3)$. Lamie (1997) suggested that the number of context switches performed also depends on the way the individual threads are assigned priorities. RTOS overhead is reduced when multiple threads are run at the same priority rather than running them at unique priorities. Unnecessary context switches are also eliminated (Lamie, 1997). Computing the threshold values statically to eliminate the unnecessary context switches are not efficient to remove all the unwanted context switches. This is achieved by computing the preemption threshold values dynamically as by Paul and Pillai (2011a). Also, in the presence of common resource sharing and task synchronization, avoiding context switches are vital to minimize the blocking times of the critical tasks. This is addressed by Paul and Pillai (2011a) for uni-processors and multi-processor platforms. Lavanya Dhanesh and Murugesan in their research tried to reduce the interrupt latency using a priority based Pre-emptive task scheduling algorithm (Dhanesh and Murugesan, 2013). Satoshi and shigeru implemented inter-core time aggregation scheduler to reduce the overhead caused by context switching (Yamada and Kusakabe, 2009).

In the previous approaches, the overhead of context switching is reduced by performing some optimization techniques on the available existing algorithms. All these are carried out at the software level. In this study, a scheduler is presented that does not require modifying the underlying processor architecture or instruction set of the processor. It utilizes the processor hardware architecture feature: tail chaining to effectively implement a scheduler that can result in reduced number of clock cycles for implementing the context switches. The potential merits of the proposed scheduler are four fold: reduction in software overhead, easiness in porting the application from one platform to another, eliminate the need of inline assemble code development and saves memory.

MATERIALS AND METHODS

This study illustrates the conventional scheduler implementation and introduces the concept of tail chaining to implement the proposed scheduler with reduced context switch overhead.

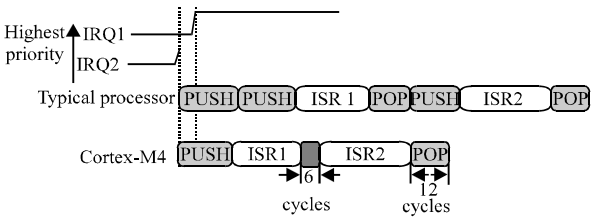

Fig. 1: Tail chaining of exceptions

Conventional scheduler implementation: The traditional ways implements tasks as application tasks, each defined with a separate TCB and the context switching is done by saving the old tasks registers state in a stack and saving the stack pointer in its TCB and updating the Program Counter (PC) with the new tasks TCB. The total number of clock cycles to perform the context switches ranges from 42-46 clock cycles in conventional method of implementation (Anonymous, 2017). Also, available RTOS like free RTOS and Keil RTX takes 84 (RTELtd., 2010) and <300 clock cycles respectively to perform the context switching. The memory required for the tasks stack is also saved to a great extent with our implementation. In this study, a scheduler that works on tail chaining of the interrupts and implements the static and dynamic priority assignment policies are presented.

Tail chaining: Tail-chaining is the process where the interrupts that occurs simultaneously are serviced one after the other without an additional overhead of state saving and restoring. As there is no effect on the stack contents, the pop and push of the eight registers is skipped by the processor. Hence, the timing gap between the two exception handlers is greatly reduced. There is only 6-cycle latency when returning from the last ISR to the new ISR as seen from Fig. 1. Scheduler is implemented using this concept of tail chaining on ARM cortex M4 based TM 4C 123 GH 6 pm microcontroller.

Scheduler implementation: The proposed scheduler is implemented with the following scheduling policies viz., round robin, static and dynamic priority. Here, four tasks are considered for scheduling. The functionalities of these four tasks (tested with LED blinking) are handled as the interrupt service routines of timer 1 A to timer 4 A and the scheduler functionality is implemented using timer 0 A.

Round robin scheduler: A round robin scheduler is implemented to schedule the above four tasks using the nested interrupt concept of the cortex M4 architecture. The scheduler runs for every 1 sec. The first three

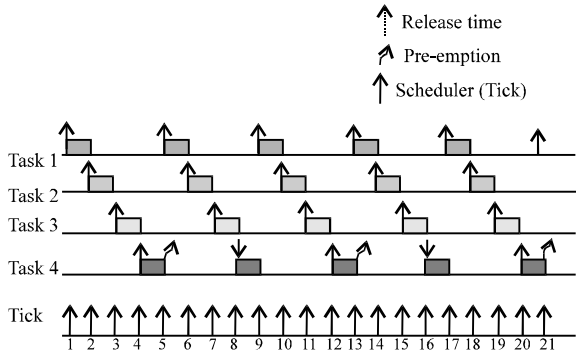

Fig. 2: Round robin scheduler execution trace

tasks have their execution times 1 which is equal to the scheduler period whereas the fourth task has execution time more than the scheduler period. It is assumed that the second instance of the task releases only after its first instance gets executed. The pseudo code of the round robin scheduler is shown in Algorithm 1.

Algorithm 1; Round robin based scheduler:

```
Timer 0A_ISR () {
{
// scheduler
if ( (task1 period && execution of previous instance) is complete))
        SetPending_IRQ (Timer1A); //Task 1
else if ( (task2 period && execution of previous instance) is complete))
        SetPending_IRQ (Timer2A); //Task 2
else if ( (task2 period && execution of previous instance) is complete))
        SetPending_IRQ (Timer3A); // Task 3
else if ( (task2 period && execution of previous instance) is complete))
        SetPending_IRQ (Timer4A); // Task 4
}
```

The tasks execution trace when scheduled in a round robin fashion is shown in Fig. 2.

Static priority scheduler: In the second approach, the same tasks are considered with varying execution times. The priorities are assigned to these tasks prior to the start of their execution and are assumed to be the same until the end of execution and hence static priority assignment policy. The tasks $\tau_{2\text{-}4}$ have their execution times greater than that of the first task τ_1 user defined priorities are assigned to tasks where the least number corresponds to a higher priority. The order of the occurrence of the interrupts is assumed to be: $\tau_{1\text{-}3}$ followed by τ_4 and the interval between two instances of the same task is given as the time period which is 4 sec.

The scheduler that is implemented behaves as a priority based scheduler where the execution of the low priority tasks is affected by the high priority tasks. Here, task τ_4 is not scheduled due to the execution of other higher priority tasks in the system. The reason for this is: execution of an exception can only be preempted by the high priority exceptions. Hence when ISR1 and ISR2 are being handled by the system even though the interrupts 3 and 4 occur they are only pended. Algorithm 2 shows the pseudo code for static priority scheduler.

Algorithm 2; Static scheduler implementation:

```
int main()
{
SetPriority_Timer1A (1); // Task 1 high priority
SetPriority_Timer2A (2)
SetPriority_Timer3A (3)
SetPriority_Timer4A (4); // Task 4 Low priority
Timer0A_On ();
While (1); //Wait for interrupt
}
```

Dynamic priority scheduler: In order to prevent the starvation of the low priority tasks, priorities are assigned to the tasks in a dynamic manner. The tasks priorities are assigned as they are being executed. The cortex M4 architecture of ARM facilitates this dynamic priority assignment. Hence, in this approach, a scheduler that assigns high priority to a task whenever it gets ready for its execution is implemented. Also, a comparison is made between the static priority and dynamic priority assignment scheduling policies when executing the same set of tasks. The scheduler based on the dynamic priority assignment is shown in Algorithm 3 and 4 shows the implementation of the ISR handlers.

Algorithm 3; Dynamic priority scheduler implementation:

```
Timer0A_ISR () {
{
// scheduler
if ( (task1 period && execution of previous instance) is complete))
      SetPriority_Timer1A (1); // Task 1 High priority
SetPending_IRQ (Timer1A); // Task 1
else if ( (task2 period && execution of previous instance) is complete))
      SetPriority_Timer2A (1); // Task 2 High priority
SetPending_IRQ (Timer2A); // Task 2
else if ( (task2 period && execution of previous instance) is complete))
      SetPriority_Timer3A (1); // Task 3 High priority
SetPending_IRQ (Timer3A); // Task 3
else if ( (task2 period && execution of previous instance) is complete))
      SetPriority_Timer4A (1); // Task 4 High priority
SetPending_IRQ (Timer4A); // Task 4
}
```

Algorithm 4; Timer NA_ISR:

```
Timer NA_ISR()
{
SetPriority_TimerNA (4)
//Make task1 Low priority
}
```

The tasks taken for this approach are same as that of the static priority. The order of the occurrence of the interrupts is assumed to be: τ_{1-3} followed by τ_4. In comparison of static and dynamic priority scheduler implementations, it can be observed that the task 4 gets executed, unless as in static scheduler which failed to schedule task 4. Hence, for the same task set given, the problem of starvation of the low priority tasks produced by static priority assignment policy is resolved by the dynamic priority assignment approach.

RESULTS AND DISCUSSION

The results of the proposed scheduler are presented in this study. The scheduler is implemented using embedded C language and the IDE used is IAR development workbench v7 and its debug terminal I/O window to view the results. The hardware setup showing the execution of task τ_1 is shown in Fig. 3.

Figure 4 shows the workspace of the implementation of the round robin scheduler. The debug terminal outputs for round robin, static and dynamic scheduler is shown in Fig. 5-7, respectively. Thus, it is proved that the output of the implemented scheduler based on tail chaining matches the expected execution traces in each of the cases.

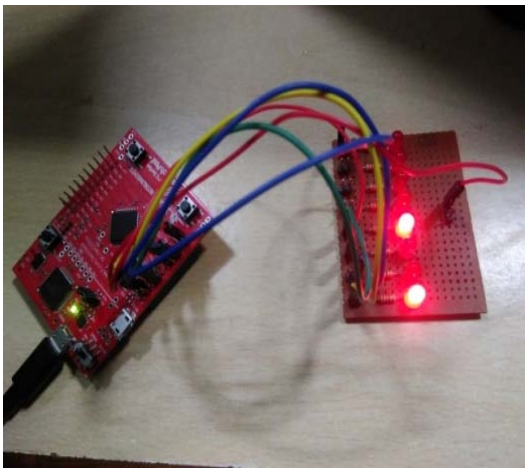

Fig. 3: Task execution with preemptions on ARM cortex M4

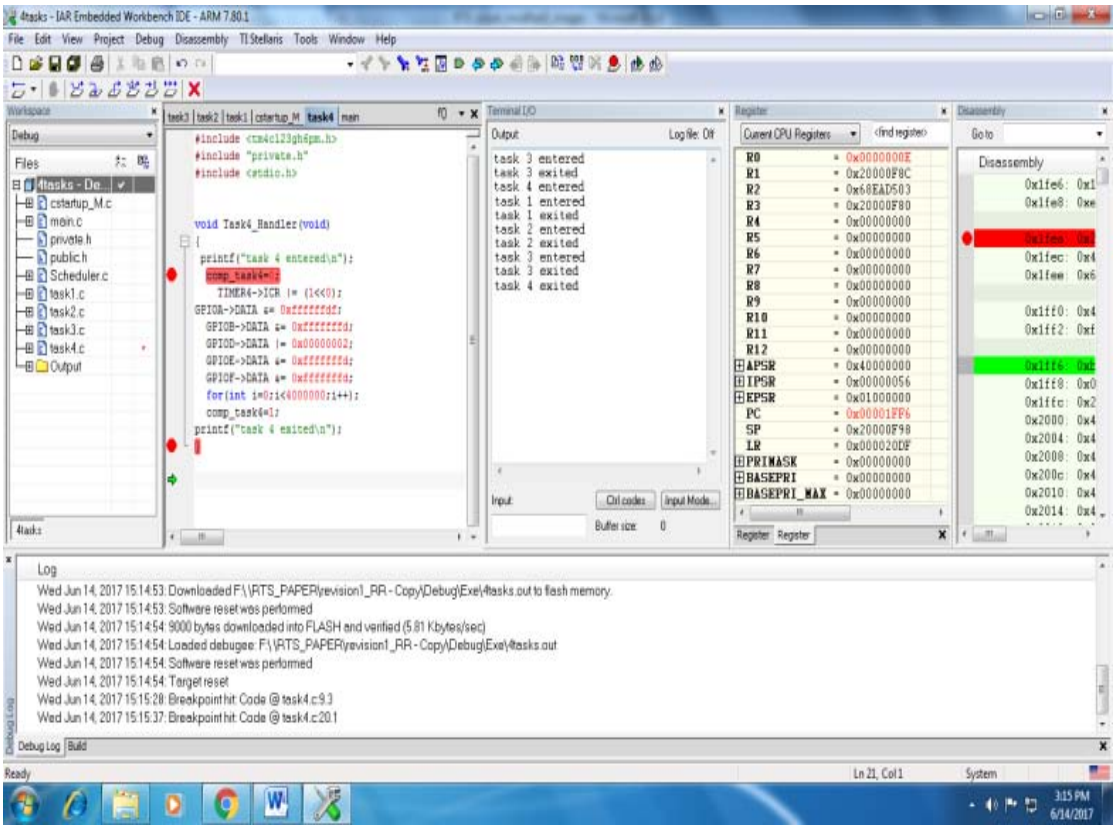

Fig. 4: IAR embedded workbench showing round robin scheduler implementation

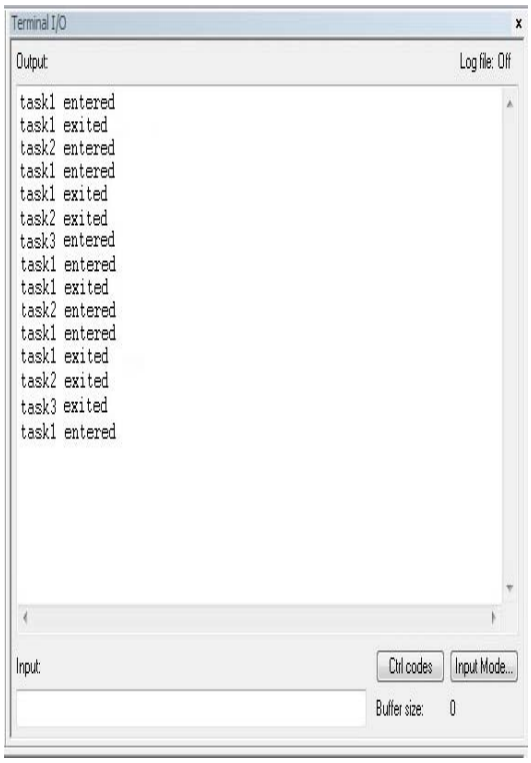

Fig. 5: Round robin scheduler

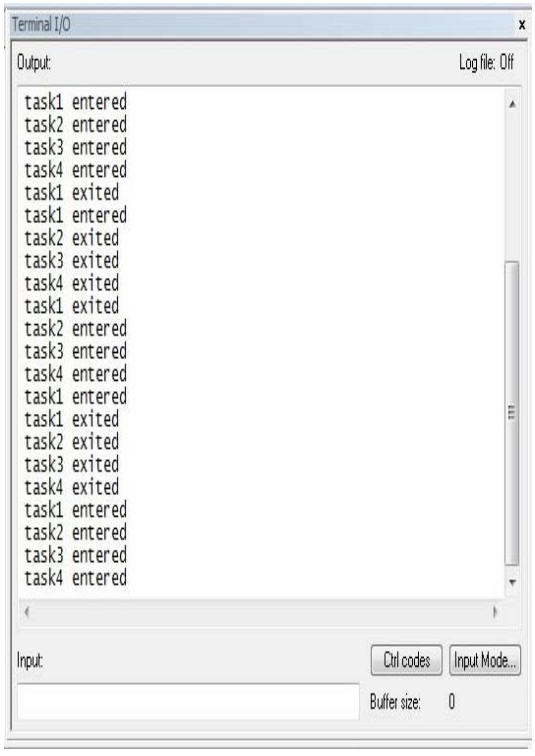

Fig. 6: Static priority scheduler

Fig. 7: Dynamic priority scheduler

CONCLUSION

This study focus on reducing the time taken to perform a context switch by introducing the concept called tail chaining while scheduling the tasks. The tasks functionalities are implemented as ISR's. The tail chaining is available as a feature of the ARM cortex M4 processors architecture. The scheduler developed can thus implement round robin, static and dynamic priority scheduling policies to schedule the tasks with a reduced context switch overhead and can perform the context switch with a latency of just 6 clock cycles whereas context switch in conventional implementation takes about 46 clock cycles and may go verse as 84 clock cycles in free RTOS and around 300 in Keil RTX.

In this researcher, the target platform considered is TM 4C 123 GH 6 PM which supports 8 levels of preemptions. By considering devices that supports 128 levels of preemptions, maximum of 128 tasks with different priorities can be implemented and thus, enabling any complex application implementation. Also, since, the cortex M3 and M4 supports dynamic changing of priority levels, the proposed dynamic scheduler can further be implemented to satisfy existing dynamic scheduling policies. When the scheduler period is less and the tasks

have to be switched from one to another frequently, the costs occur due to context switching are expensive. This overhead can be reduced to a great extent by using the proposed scheduler based on tail chaining without changing the under lying architecture or hardware or software features of the processor.

REFERENCES

Anonymous, 2017. Introduction to Real Time Operating System (RTOS). Spearhead EduOnline Pvt Ltd, Bengaluru, India. https: //www.apnacourse. com/course/real-time-operating-system-rtos-rv-vlsi.

Carbone, J., 2012. Reducing context switching with preemption-threshold. Express Logic Inc, San Diego, California, USA. http://rtos.com/images/uploads/ Preemption_Threshold.pdf.

Dhanesh, L. and P. Murugesan, 2013. Performance based improvement of the CPU by reducing the real-time interrupt latency using the PPTS algorithm. Indian J. Adv. Comput. Sci. Technol., 1: 15-24.

Kumar, M.M.R., R.B. Renuka, M. Sreenatha and C.K. Niranjan, 2014. An improved approach to minimize context switching in rounr orbin scheduling algorithm using optimization techniques. Intl. J. Res. Eng. Technol., 3: 804-808.

Lamie, W., 1997. Preemption-threshold. Express Logic Inc, San Diego, California, USA. http://www. threadx.com/preemption.html.

Paul, A. and A.S. Pillai, 2011b. Reduction of context switches due to task synchronization in uniprocessor and multiprocessor platform. Intl. J. Comput. Commun. Technol., Vol. 2,

Paul, A.A. and B.A.S. Pillai, 2011a. Reducing the number of context switches in real time systems. Proceedings of the 2011 International Conference on Process Automation, Control and Computing (PACC), July 20-22, 2011, IEEE, Coimbatore, India, ISBN:978-1-61284-765-8, pp: 1-6.

RTELtd., 2010. FreeRTOS FAQ-memory usage, boot times and context switch times. Real Time Engineers Ltd, Bristol, England, UK. http://www.freertos. org/FAQMem.html.

Wang, Y. and M. Saksena, 1999. Scheduling fixed-priority tasks with preemption threshold. Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications RTCSA'99, December 13-15, 1999, IEEE, Hong Kong, China, China, ISBN:0-7695-0306-3, pp: 328-335.

Yamada, S. and S. Kusakabe, 2009. Impact of inter-core time aggregation scheduler on a database server workload. Asia J. Inform. Technol., 8: 94-103.

Yiu, J., 2013. The Definitive Guide to ARM® cortex®-M3 and Cortex®-M4 Processors. 3rd Edn., Elsevier, Amsterdam, Netherlands, Pages: 199.