DOI : 10.3745/JIPS.2009.5.3.135

Study on Preemptive Real-Time Scheduling Strategy for Wireless Sensor Networks

ZHAO Zhi-bin* and GAO Fuxiang*

Abstract: Most of the tasks in wireless sensor networks (WSN) are requested to run in a real-time way. Neither EDF nor FIFO can ensure real-time scheduling in WSN. A real-time scheduling strategy (RTS) is proposed in this paper. All tasks are divided into two layers and endued diverse priorities. RTS utilizes a preemptive way to ensure hard real-time scheduling. The experimental results indicate that RTS has a good performance both in communication throughput and over-load.

Keywords: *real-time schedule; wireless sensor networks; two-level priority; TinyOS; dynamic schedule*

1. Introduction

1.1 Wireless Sensor Networks Operating System and TinyOS

So far, a number of operating systems for WSN have been developed, such as MantisOS[1,2], SOS[3,4] and Contiki[5]. TinyOS[6,7] is an open-source OS that developed by UC Berkeley. Its basic feature is the use of component-based programming model to achieve a good cross-platform capability and efficiency, which is applicable to hardware environment with very limited resources. Among these operating systems, the most widely used one is TinyOS. It was developed at UC Berkley relying on the Smart-dust projects. According to statistics, there are more than 500 companies or research institutions that are using TinyOS in academic research or commercial development. This is because it is open source and it has been successfully transplanted to a lot of hardware platforms and is more sophisticated in using. Another reason is there is an active development group for the TinyOS.

TinyOS is designed for Wireless Sensor Network, and it is a lightweight, low-power embedded operating system. The programming language of TinyOS is NesC with modular design method. The use of modular design makes it capable to adapt to the diversity of hardware and makes the applications reuse the general software services and abstract. TinyOS is a typical Wireless Sensor Network Operating System. Its structure, principles and implementtation methods is a good reflection of the WSN's features. The following is an introduction of it, and the focus of analysis is its scheduling mechanism.

1.2 System Architecture of TinyOS

TinyOS is developed for the embedded System with high concurrency and scarce hardware resources. Its runtime environment is based on the components and is made in NesC language supporting the Wireless Sensor Network architecture.

TinyOS runs on the sensor node "mote" primarily and the battery supplies power for the mote whose internal and external storage are limited. In order to achieve a higher concurrency in using limited memory and processing power, TinyOS introduces the mode of combining task with event, and have the following characteristics [25,26];

(1) The layer-component architecture, the so-called layer-component, divides the component into different layers according to the components' correlation level to hardware. The bottom layer deals with hardware-related operation, the top layer is a user-defined component in application, and the mediate layer implements the abstract of hardware. In this sense, TinyOS provides an optional Component Library in fact.

(2) TinyOS adopts event-based concurrency model. Event is corresponding to the emergency case such as external interrupts, and it can preempt tasks (task is the process of dealing with backstage computation), or other events and take preference to execute. So, from the macroscopic view, between task, and event (as well as between event and event) are moving forward simultaneously that is, achieving the concurrent of task and event (or different events).

(3) TinyOS adopts the running mode based on Finite

Manuscript received March 10, 2009; revised July 1, 2009; accepted August 25, 2009.

Corresponding Author: ZHAO Zhibin

* College of Information Science and Engineering, Northeastern University China ({zhaozhibin,gaofuxiang}@ ise.neu.edu.cn).

State Automata. Each resource corresponds to each component, which is just the description form of the Finite State Automata. Components migration rapidly between states by using command and event, and several Finite State Automata can share the same runtime environment. Another advantage of using Finite automata running mode is putting hardware running mode into software running mode very naturally. Just as components of hardware respond to the state changes of the pins, components in TinyOS respond to the commands and events (Event is equivalent to input, and the command is equivalent to the output).

(4) In TinyOS there is a widespread use of phased-operation, which is dividing the longer operation into some relatively short ones to avoid busy-waiting. The premise of this division is the beginning and end of the operation can be separated in time domain.

The framework of TinyOS is shown in Fig 1, in order to providing favorable modular structure that supports the diversity in wireless sensor designing and application, the system is composed by component-based pattern, and primarily consists of master components (including the scheduler), application components, system service components and hardware abstraction components.

Hardware abstraction components implement the abstraction of wireless sensor hardware platform, including the sensor subsystem, the wireless communication subsystem, the input/output devices and the power control system on the bottom layer. These abstractions shield details of the underlying hardware for the upper layer, and simplify the system's transplantation. System Services components are composed of three parts including communication services, sensor services and power management. In these three parts, the components of communication services support data transmission protocol and the control of wireless communication module; the sensor service components support analog-digital conversion and data collection of various sensor modules; power management components support the power-state control of processor, wireless communi cation module, sensor module and other components. Application components are defined by user in line with specific application, and fulfill specific application-related functions and strategies. The control components fulfill the control procedure of the whole operation system, and primarily carry on the wireless sensor's initialization and the maintenance of system run time status.

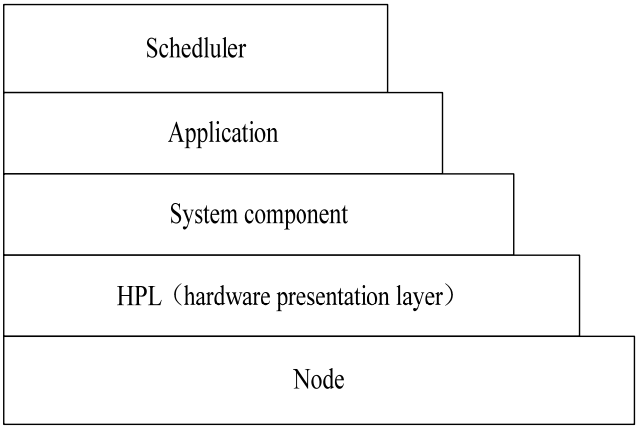


Fig. 1. the framework of TinyOS

1.3 Analysis of the FIFO scheduling strategy in TinyOS

TinyOS adopts the two-level concurrent models based on the combination of tasks and event-driven[8].

1.3.1 Task Mechanism of TinyOS

(1) Tasks are equal and there is no concept of priority and no preemption between tasks. All tasks share one executing space, which saves the memory overhead in run time.

(2) Tasks are managed by a circular task queue in system, and the task scheduling follows FIFO mode. Tasks are scheduled by the simple FIFO queue. Resources are distributed beforehand, and currently there can only be seven waiting tasks in the queue. The task-processing model is shown in Fig 2. The size of the task list in the figure is eight. There are three tasks in the queue.

(3) If the task queue is null and there is no events occurring, then the processor will enter into SLEEP mode automatically, and will be woken up by hardware interruption event subsequently. This is conducive to saving energy of system.

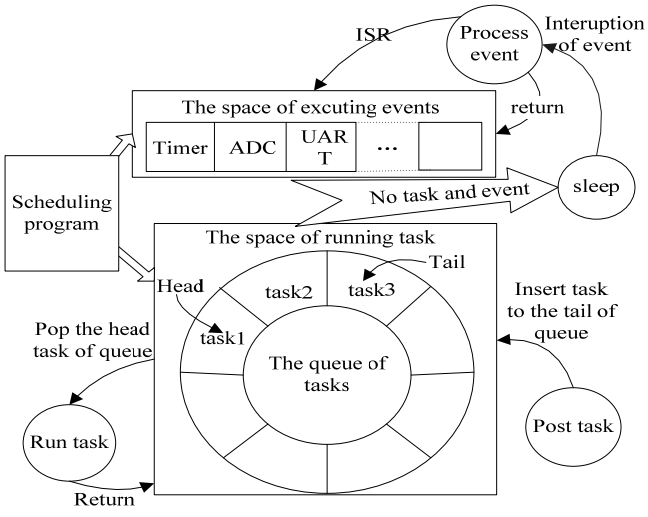


Fig. 2. the scheduling srategy of TinyOS

Task is defined by user application, and can be created by applications or event handlers. A task is created by the keyword "Task", and the specific grammar of definition is: task void myTask( ){……}。After creating the task, it will be posted to the queue by the TOS_post function. The procedure is shown in Fig 3. The task scheduler in the core scheduling algorithm returns as soon as it puts the task into the task queue, and the task will be carried out. When the task queue is vacant, the task can be submitted. The submission is only inserting a function pointer into the queue. As shown in Fig 4, TOSH_run_next_task( ) function takes charge of taking out the task which the TOS_sched_full points to from the queue and carrying it out. Kernel calls TOSH_run_next_task() function in an infinite loop and carries out all the mission functions in sequence as long as the queue is not empty.

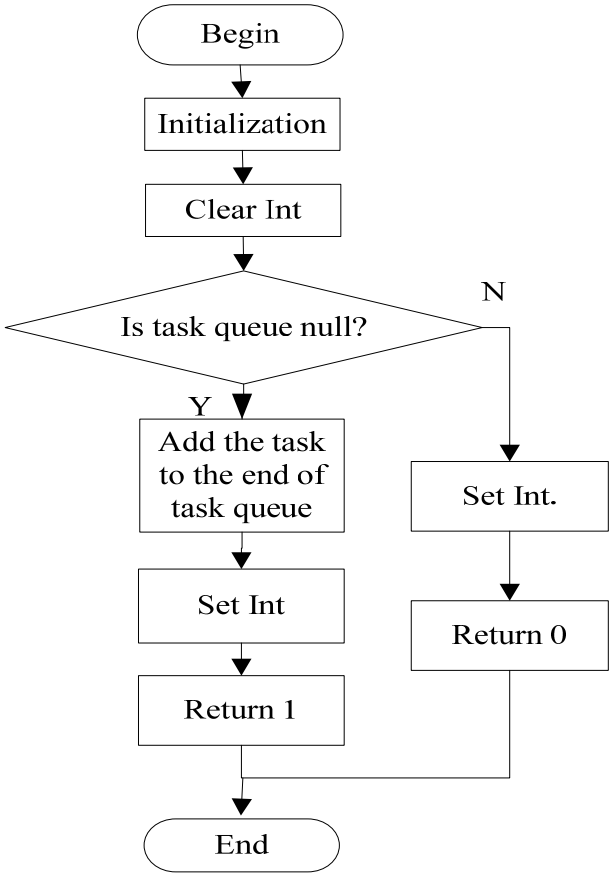


Fig. 3. the function of TOS_post

1.3.2 Event-driven Mechanism of TinyOS

Events are generated by hardware interruption (MCU external interruptions, timer interruptions, etc) directly or indirectly. When receiving event, TinyOS will execute the event handler corresponding to the event immediately. Event can preempt the running task. It is an asynchronous, time response fast executive mode.

In the TinyOS scheduling mechanism, the task mechanism is not a real-time one. It makes some more important or more real-time tasks not be completed before the deadline and leads to packet-loss, overload, decline of the throughput etc. So it is applicable to non-preemptive, non-time-critical application. Event handler can preempt the current running task and this can be applicable to time-critical application. However, interruption sources that generate events are extremely limited, and unable to meet the multi-mission and real-time application.

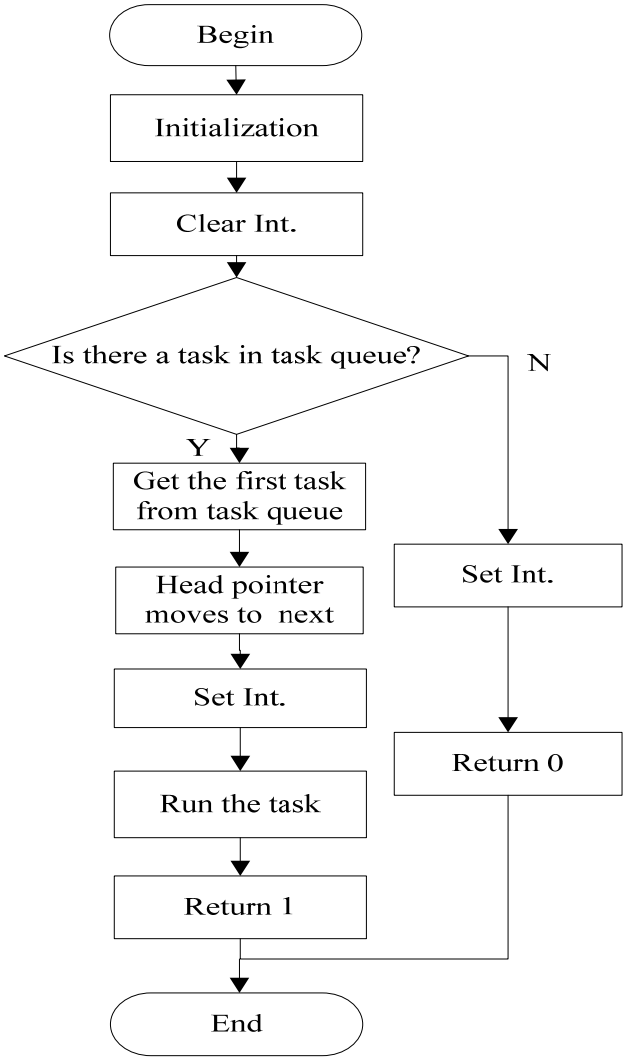


Fig. 4. the function of TOSH_run_next_task

1.3.3 Disadvantage of FIFO Scheduling Mechanism in TinyOS

Though TinyOS is widely used and receives considerable recognition, it does not mean that TinyOS can apply to all WSN application. Practically, in some situations, TinyOS does not work very well and may present overload, causes the conditions of task-loss, communication throughput declining and can not guarantee real-time.

On the other hand, some of the WSN applications, such as forest fire alarm, automatic letter distribution system and multimedia WSN need real-time data transmission. To transfer data in time, we must schedule tasks in a real-time way.

In WSN, the typical three missions for node are: receiving packets waiting to be forwarded, forwarding packets it receives, processing local sensing data and sending it out. The number of the node's tasks depends on the node data-handling manner. If node just sends the raw data to the BS, most of the tasks will be communication routing mission; if node sends data to BS after collecting local data and processing the data, the local data-processing tasks will be more. When the tasks to be processed exceed the node's processing capacity, the overload will happen. For the former case, if the frequency of node sending data is excessive high, or network density is too large resulting in excessive communication tasks, overload may occur; for the latter, if the number of local data to be processed is excessive large, or the occurring frequency of local task is too high, it will also lead to overload.

In addition, when the interruption occurs on a very high frequency, leading to that CPU is too busy in processing interruption to execute other tasks, there will also be an overload. When the processing speed of system tasks is lower than the frequency of tasks occurring, the task queue (Currently, only 7 tasks can be stored) will be stuffed up soon. It will lead to task losses. As for the local sensor acquisition rate, we can artificially control, for instance, decreasing the sampling frequency. However, for the happening of communication routing tasks, it is not easy to interfere artificially. At this time, if the overload occurs, it will result in the declining of packet throughput directly. Occurrence of this phenomenon is mainly due to that packet sending and receiving is restricted to the local tasks. When the occurring frequency of local task is too high, the task queue will be stuffed up soon, then tasks of transmitting or receiving could be lost, resulting in packet loss. Moreover, if the local task's running time is too long, tasks of transmitting or receiving packet have to wait a long time for processing, thereby reducing the communication rate.

Additionally, the following occasions, TinyOS's scheduling strategy may also lead to problems.

(1) Certain tasks (such as encryption and decryption mission in security applications) have very long implementing time. If some real-time missions enter the task queue after the task at this time, the real-time will be affected. For the receiving and transmitting of packets, the baud rate will be affected.

(2) When the occurring frequency of local task is high, the task queue will be stuffed up at a short time, other tasks could be lost; besides, if there are many local tasks (such as several channels collect data at the same time, then there will be many local tasks), this will also affect the normal communications.

(3) When a certain task in the queue is blocked or performs abnormally because of suddenness, it will affect the subsequent task's running, even cause the system go down.

1.4 Analysis on Improved Scheduling Policies in WSNOS

As analysis above, the simple queue scheduling policy adopted by TinyOS will result in overload, task loss, low packet throughput, etc, under certain circumstances. In addition, TinyOS merely builds fundamental scheduling framework, which only implements soft real time instead of hard real time and thus impedes the overall reliability of the embedded system. Meanwhile, there remains the need to design a multitasking system due to poor throughput and CPU utilization caused by the adoption of single task kernel.

For example, in a multimedia WSN for forest fire monitoring, once a smoke and temperature sensing node reports alarm, video nodes are waked up. From then on, much data needs to be transferred in a hard real time way.

In order to achieve real-time schedule, priority based preemptive scheduling policy is often used. According to application requirements many priority-based multitasking scheduling algorithms are put forward, one of which may be that, for example, each composing phase of the communication route should work timely to ensure other tasks finish properly. Equipping TinyOS with multitasking will enhance response speed. Analysis to several typical improvement strategies is shown below.

Tasks scheduling strategy in WSN will decide whether the nodes finish the tasks in time or not. Nodes in WSN not only are necessary to sense and transmit data, but also to forward data for other nodes.

Priority-based task scheduling strategy [9], which is designed to avoid important task to be lost in system, divides tasks into three types: sending data packet, transmitting data packet, and sensing local data according to the functions of different tasks in network. Therefore, it guarantees the more important task to be run in a priority way. Thus, throughput of the system is improved.

This scheduling strategy does not behave well to meet the requirement of real-time. Firstly, it may drop the task before running because the task has exceeded the deadline; secondly, because of non-preemption, the short-time tasks may be blocked to wait for the long-time ones and it leads to the overload for short-time tasks.

Philip Levis and Cory Sharp have implemented Earliest

Deadline First, EDF [10], which is widely used in real-time system.

Preemptive EDF strategy is the most optimal scheduling for single processor scheduling strategy. That is, if preemptive EDF can't schedule a set of tasks in single processor, other scheduling strategy can't either. Substantively, it is a dynamic process. The algorithm allows a relatively short task to be a preferential one, which makes the system flexible and real-time performance improved. However, the overload problem has become drawbacks of the algorithm, which limits the utility of the algorithm.

Rate monotonic scheduling strategy-RM[11,12] is a kind of fixed-priority scheduling strategy. Once the priority of one task is identified according to its periodicity, it will not change with time. A task in smaller periodicity has higher priority. The author of [13] has proved the RM to be the best. And, RM can schedule tasks set while other fixed-priority strategies can.

Fixed-priority strategy is suitable for wireless sensor network operating system, because it needs to be scheduled one time before running. This fixed-priority will be able to ensure the cyclical behavior, and the tasks are scheduled only in one queue. RM's flaw is the lack of flexibility due to its unchanged in running time. Therefore it is not suitable for working during running.

This paper proposes a real-time scheduling strategy for wireless sensor networks to enhance the communication throughput and reduce the overload. RTS adopts two-layer priority scheduling strategy according to the demand for real-time analysis, and solves the real-time task scheduling problems in WSN commendably.

2. Real-time task scheduling and the two-layer priority

In RTS, tasks are divided into two priorities, static priority and dynamic priority. To ensure real-time and major network packets in the wireless network transmitted reliably. First of all, in accordance with the function of task, it adopts the two relatively static level of priority, which will not change as the time passes, belonging to the fixed priority. Secondly, tasks deadline and run time are the two constraints of dynamic priority, which ensures the reliability of real-time task.

2.1 Determine the static priority

In a sensor network, the number of tasks on a node depends on the node how to deal with data. If nodes only send raw data to the base station directly, most tasks are communication routing ones; If nodes sense and process data locally and then send them to base station, most tasks are the local data-processing ones. When the tasks waiting to be processed are more than node's capacity, overload will happen. For the former, overload will happen if the frequency of node sending data is too high or the density of nodes in the network is too large; for the latter, it also will happen if local data waiting to be processed is excessive or the frequency of local tasks is too high.

As shown in table 1, we divide the tasks into network communication routing tasks and local data processing tasks, and give the tasks two relatively static priorities, high and low. Network communication routing tasks is prior to local data processing tasks.

Table 1. static priority of tasks

Task classification	Task function	Static priority
Network communication routing tasks	Sending, receiving and transferring network data, or response to network command	high
local data processing tasks	Sensing and processing data	low
	Complicated procedure in dealing with data, for example, coding	low
	Short, cycle system task	low

The static priority of tasks is in top-layer priority. Tasks in low or high static priority are set by different priorities in bottom-layer, as shown in Table 2.

The tasks with high static priority in their top-layer priority have dynamic priority in their bottom-layer priority, but the ones with low static priority in their top level priority have fixed priority in their bottom-layer priority.

Table 2. two- layer priority of tasks

Task	Top-layer priority	Bottom-layer priority
Task1	High static priority	Dynamic priority
Task 2	Low static priority	Fixed priority

2.2 Determine the dynamic priority

When new task comes, its attributes such as arriving time, running time and relative deadline will be submitted. We decide task's dynamic priority by the attributes mentioned above. The determination of dynamic priority is shown in Table 3.

Table 3. the dynamic priority of tasks

Task	Running time	Relative deadline	Dynamic priority
Task1	Short	Small	Highest
Task2	Long	Small	Higher
Task3	Short	Large	Lower
Task4	Long	Large	Lowest

2.3 Determine the fixed priority

We also divide different fixed priority according to arriving time, running time and relative deadline. What different from dynamic priority is the fixed priority is determined by execution periodicity and running time for periodic tasks. For non-periodic tasks, the fixed priority is determined by relative deadline and running time. Fixed priority is determined only when tasks are initialized.

2.4 Schedulability analysis

Tasks in system are triggered by events. Due to arriving time of task can not be predicted, the schedulability can only be dynamicly detected when task arrives. Assume that there is a set of dynamic tasks, $T=\{t_1, t_2, \ldots, t_n\}$, $t_i=<arrive\_time_i, run\_time_i, period_i, Deadline_i, pri\_static_i, interrupted_i, SP_i>$, which are schedulable, independent and preemptive and running on processor. They have been sorted by two-layer priority strategy. At one moment, a new task t_{new} reaches the system, let's examine the schedulability of the new task set $T'=T \cup \{t_{new}\}$. According to the definition of schedulability, if the new task set is schedulable, every task in the set must finish before its deadline, namely, satisfy the condition of $finish\_time_i > deadline_i$, where $deadline_i=arrive\_time_i+deadline_i$. Considering the randomness and preemptive, the finishing time t_i of a task which could change along with arriving of new task is a function of time t, $f_i(t)$. The new task set is sorted by deadline in descending, $T'=\{t_1, t_2, \ldots, t_j, t_{new}, t_{j+1}, \ldots, t_n\}$. When new task arrived, it will not seize the execution of which has higher priority, but will affect the schedulability of task having lower priority. So, we just have to detect the schedulability of t_{new} and those behind it. Given at time r, the remaining time for each task is $C_i(t)$, and the task's completion time is the sum of computing time for all tasks having higher priority and its own computing time, as shown in formula (1)

$$f_i(t) = \sum_{1}^{i} C_j(t) \tag{1}$$

We can obtain the schedulability detecting theorem for dynamic task set: the new task set after new task arriving is schedulable, its necessary and sufficient condition is: for all tasks $t_i \in \{t_{new}, t_{j+1}, \ldots, t_n\}$, they have to satisfy formula (2)

$$f_i(t) = \sum_{1}^{i} C_j(t) < arrive\_time_i + deadline_i \tag{2}$$

Considering formula (3),

$$f_i(t) = f_{i-1}(t) + C_i(t) \tag{3}$$

Dynamic testing can be carried out on a rolling basis, which has computing complexity: O(n). If we use EDF to schedule periodic tasks, the necessary and sufficient condition for schedulability is shown in formula (4).

$$\sum_{1}^{n} \frac{runtime_i}{period_i} \le 1 \tag{4}$$

This formula shows that U≈1 for EDF algorithm and the utilization ratio of processor is as high as 100%. EDF is the optimized dynamic, preemptive priority scheduling algorithm for single-processor real-time system, that is, for any real-time task set, once there is a algorithm for scheduling, EDF will be there. It is worth nothing that, EDF will adjust the priority only when task instance is ready. Because in other moments, task instance sorted by deadline will not change, that is to say the arrangement between task's priority is fixed.

3. Scheduling strategy for real-time tasks

3.1 Scheduling while task queue is vacant

If there is more space, new task & task5 will be inserted to queue rear. We don't think about the static priority and use fixed priority to guarantee the task with lower static priority. This schedule will ease the starving case in local data processing. The fixed priority for every task is shown in Table 4 and task queue with fixed priority is shown in Fig. 5. In the queue, fixed priority is sorted but static priority is not. &task2 arrives thirdly, and &task4 is the second.

Table 4. two-layer priority of tasks in the queue

task	Static priority	Fixed priority
&task2	1	2
&task1	1	2
&task4	0	3
&task3	1	3
&task5	0	4

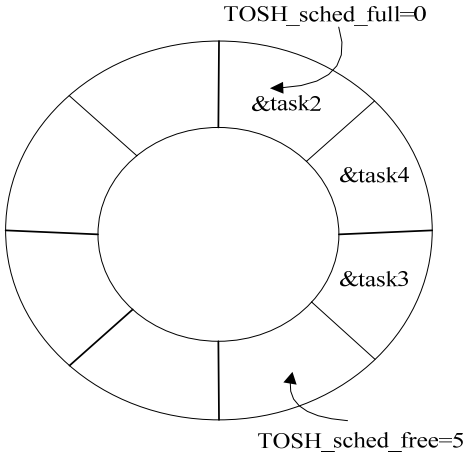


Fig. 5. tasks sorted by fixed priority in the queue

3.2 Scheduling when task queue is full

First of all, we need sort the task by static priority (steady sorting, in case to upset fixed priority sorting). In task queue, the two-layer priority for each task is shown in Table 5. The full task queue with sorted two-layer priority is shown in Fig. 6.

Then we judge the static priority and fixed priority of current task and rear task separately. If the two-layer priority of current task is higher than the rear, exchange them. Then, insert the rear task into appropriate location of the queue. At last, abandon current task.

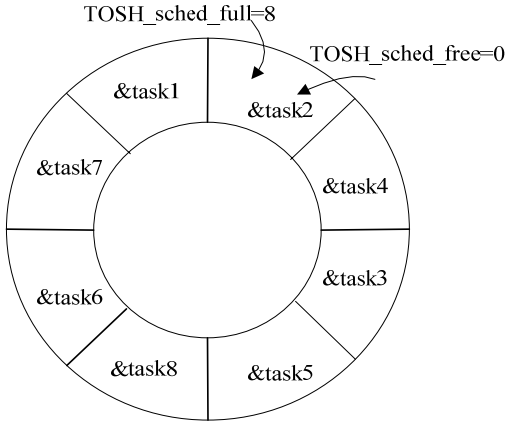


Fig. 6. tasks sorted by two-layer priority while the queue is full

Table 5. two-layer priority of tasks in the queue

tasks	Static priority	Fixed priority
&task2	1	2
&task4	1	2
&task3	1	2
&task5	1	5
&task8	1	4
&task6	0	3
&task7	0	6
&task1	0	7

4. Experiment and evaluation

In this paper, the task scheduling strategy simulation is examined on TinyOS's simulation platform --TOSSIM.

In order to evaluate RTS task scheduling strategy, the algorithm is simulated on the following aspects, compared with TinyOS's own task scheduling strategy FIFO and task scheduling algorithm EDF proposed by Levis.

(1) Communication throughput: communication throughput is the node capacity to send, receive and transmit packets.

(2) Dropping ratio of overtime task: the main study is the evaluation of RTS on scheduling performance in the events of instantaneous overload.

(3) Response time: response time is the time span from submitting task to completing task. In this paper, the real-time performance of RTS is evaluated through the average response time for different types of task. Then it is compared with FIFO and EDF on the average response time of the same tasks.

In this paper, 100 nodes are deployed randomly in the 250m×250m monitoring area and BS is located at (45, 45).

4.1 Evaluation of communication throughput

The throughput results of sending packets after using FIFO, EDF and RTS three task-scheduling strategies are shown in Fig. 7.

It can be concluded that with the implementation time of local task increasing, the number of packets sent per second declines sharply for FIFO strategy. Finally, only one packet per second and at this point most tasks in task queue are local data-processing tasks. The performance is improved significantly for EDF scheduling algorithm. The

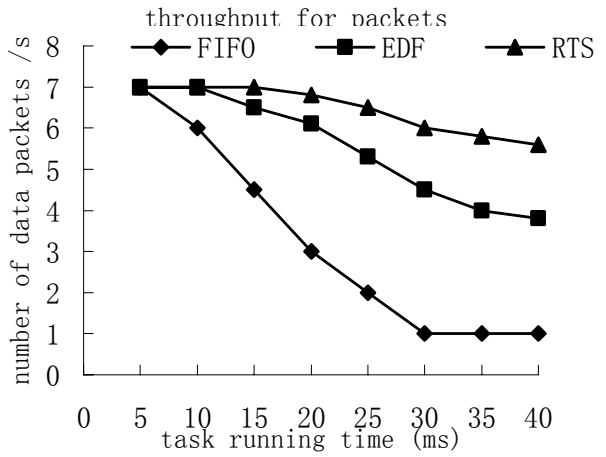


Fig. 7. throughput for sending packets under 8Hz local task/local task runtime

average number of packets is larger than four, implementting preferentially the sending task whose deadline is relatively early in task queue. Compared with EDF scheduling strategy, the number of packets sent per second is basically steady and is improved obviously, which is at least 6. This type of task is implemented before local tasks, regardless of whether the tasks of sending packets are prior, which makes it sure that the sending throughput is improved efficiently.

4.2 Dropping ratio of overtime task

As shown in Fig. 8, FIFO has bad performance in real-time aspect, whose rate uncommonly rises along with utilization rate of processor. Fortunately, EDF scheduling strategy has been improved in real-time aspect, but when utilization rate goes up to a certain level, instantaneous overload problems will rise and cause the rate to increase obviously. Although RTS adopts preemptive EDF to solve

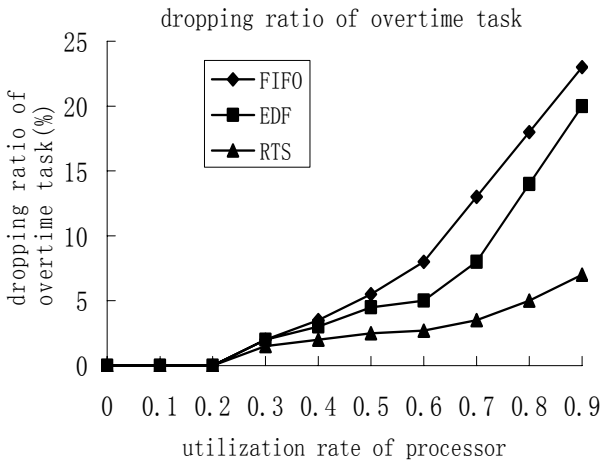


Fig. 8. dropping ratio of overtime task

high static priority tasks, the most dropped tasks caused by instantaneous overload are those in lower priority. Tasks in lower priority use monotonous task scheduling strategy in fixed priority. It can ease instantaneous overload phenolmenon effectively. Thus, in RTS, task set's dropping rate doesn't go up obviously though the utilization rate of processor rising. Miss rate about overtime task is always under 10% during the entire testing process, in other words, 90% tasks can execute before their deadline.

4.3 Response time

As shown in Fig. 9, four classes of tasks, which use EDF scheduling strategy, have a little shorter average response time than those who use FIFO. This can't show that EDF has any superiority in responding real-time task. However, when using RTS scheduling strategy, sending and transferring tasks which have higher static priority will be handled firstly, so they can be responded in time; sensing data and complicated handling which have lower static priority will also run orderly according to fixed priority. These two aspects reduce the response time extremely than other two scheduling strategies. To sum up, RTS guarantees the system good real-time performance.

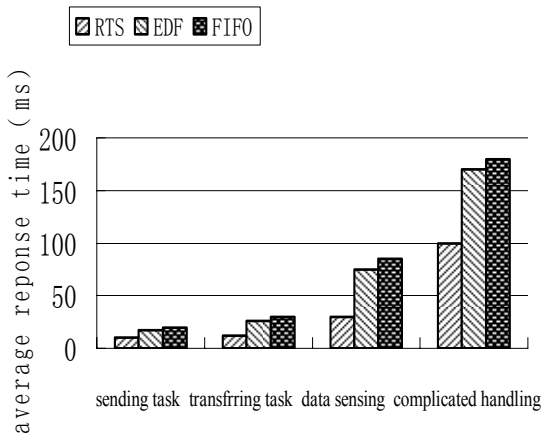


Fig. 9. the average response time of different tasks

5. Conclusion

RTS scheduling strategy proposed in this paper reserves the advantage of high utilization ratio of processor in EDF scheduling strategy. According to RTS, the task with high static priority will be responded and executed prior to other task. At the same time, RTS absorbs the high efficiency of RM in dealing with instantaneous overload problem. Therefore instantaneous overload, which is often happened

to low-priority tasks, is solved in RTS.

Through analysis of the experimental results, RTS scheduling strategy has good performance in communication throughout, dropping ratio of overtime task, and response to real-time task. It is valuable in the field of task scheduling in wireless sensor network OS. It is an exploration for WSN real-time research.

Acknowledgement

This work is supported in part by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000, the National Nature Science Fund of China under grant No. 60873199.

References

[1] MantisOS [EB/OL]. http://mantis.cs.colorado.edu. 2007-6-1.

[2] S. Bhatti, J. Carlson, H.Dai, *et al*. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms[J], Mobile Networks and Applications, 2005, 10(4): 563-579.

[3] SOS[EB/OL]. http://nesl.ee.ucla.edu/projects/sos/. 2007-6-1.

[4] Han C, Kumar R, Shea R, *et al*. A dynamic operating system for sensor networks[A], Proceedings of the 3ed International Conference on Mobile Systems, Applications and Servives[C], 2005: 163-176.

[5] A. Dunkels, B Gronvall, T Voigt. Contiki--a lightweight and flexible operating system for tiny networked sensor[A], Proceedings of The 29th Annual IEEE International Conference on Local Computer Networks[C], 2004: 455-462.

[6] TinyOS[EB/OL], http:// www.tinyos.net, 2007-6-1.

[7] Farshchi S, Nuyujukian P, Pesterev A, *et al.* A tinyOS-based wireless neural sensing, archiving, and hosting system[A], Proceedings of 2nd international IEEE/ EMBS conference on neural engineering [C], 2005, 671-674.

[8] Hill J, Szewczyk R, Woo A, *et al*. System architecture directions for networked sensors[J], Operating Systems Review, 2000, 34(5): 93-104.

[9] Venkita Subramonian, Huang-Ming Huang, Seema Datar, et al. Priority scheduling in TinyOS – A case study[R], Technical Report WUCSE, Washington University in St. Louis, 2002, Dec.

[10] Kargahi Mehdi, Movaghar Ali. A method for performance analysis of earliest-deadline-first scheduling policy[J], Journal of Supercomputing, 2006, 37(2), 197-222.

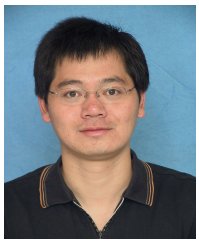
[11] Naghibzadeh, M. A modified version of rate-monotonic scheduling algorithm and its' efficiency assessment[A], Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems[C], 2002, 289-294.

[12] Lopez J, Garcia M, Diaz J, et al. Utilization bounds for multiprocessor rate-monotonic scheduling[J] , Real-Time Systems, 2003, 24(1): 5-28.

[13] Baruah S K, Haritsa J R. Scheduling for Overload in Realtime Systems[J], IEEE Transactions on Computers, 1997, 46(9): 1014-1018.

[14] Silva de Oliveira, da Silva Fraga, J. Fixed priority scheduling of tasks with arbitrary precedence constraints in distributed hard real-time systems[J], Journal of Systems Architecture, 46(11), Sept. 2000, 991-1004.

[15] Philip Levis, Nelson Lee, Matt Welsh, et al. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications[A], Proceedings of the first international conference on embedded networked sensor systems [C], 2003, 126-137.



ZHAO Zhi-Bin
He was born in 1975. He received a Ph.D. degree in Computer Software and Theory from Northeastern Univ. in 2007. He has been a senior lecturer at Institute of Computer Software and Theory, Northeastern
Univ. since 2006. He is a CCF member. His current research areas are distributed database and wireless sensor network.

GAO Fuxiang
He was born in 1961. He received a Ph.D. degree in Computer Application from Northeastern Univ. in 2006. He has been a professor at Northeastern Univ. since 1996. His research interests are in the area of embedded Internet, Internet security, multimedia Internet.