A Realization of Two-Level Scheduling in Embedded Operating System

Song Zeceng and Qian Weizhong*

School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, 611731, China

songzeceng@outlook.com; wzqian@uestc.edu.cn

Abstract. In embedded operating systems, how to ensure thread tasks execute in order of priority is an important issue. Considering that thread may have child threads, and child threads are scheduled by their parent thread, so they also need to be executed according to their priorities. For such two-level scheduling issue, this paper proposes a new scheduling policy: parent thread uses the table-driven scheduling policy, and child thread uses the fixed priority scheduling policy to ensure the orderly execution of both parent and child threads. In order to cooperate with the implementation of two-level scheduling, this paper also realizes the creation of child threads in an embedded operating system by imitating Linux thread replication, connects the parent and child threads through pointers, and realizes a priority queue of child threads in their parent thread. In addition, by saving the ID of current executing child thread into the stack of corresponding parent thread, this paper ensures that parent thread can recover its last executing child thread after several switches of parent threads. Several experiments on 2440 single core development board showed that method proposed by this paper could realize the creation of sub threads and two-level thread scheduling in embedded operating system.

1. Introduction

With the development and progress of the society, people put forward higher and higher requirements for the performance of applications. In order to meet these requirements and ensure that the applications execute in specified order, scholars have put forward many scheduling policies[1], such as global fixed priority (G-FP), table scheduling(TD), time slice polling, etc. These scheduling policies guarantee the functionality and performance of the system effectively and are effective in dealing with single-level threads. However, as the functions become more and more complex, threads may process tasks in parallel by creating sub-threads to furtherly reduce response time and improve the performance of system However, current mainstream scheduling policies haven’t considered maintaining such two-level thread structure at runtime to realize multi-thread scheduling. In addition, although most industrial operating systems have used multi-core CPUs to improve performance, there are some professional fields (such as aerospace, exploration, etc.) still use single-core develop board(such 2440, etc) due to demand, budget and other reasons. Therefore, how to deploy two-level thread scheduling with embedded operating system into single-core development board is also an important subject concerning the implementation of the policy. Focused on the problems discussed above, this paper proposes a two-level scheduling policy where parent thread uses table scheduling policy and child thread uses fixed priority scheduling policy, and an implementation on a single-cored development board arm-2440 which has an embedded operating
system. A series of experiments have proved that this policy can guarantee that both parent and child threads execute in order of specified priority as well as effectively maintain the related structure between threads.

2. Related work

2.1. Multi-thread scheduling
In order to improve the functionality and performance of embedded operating system, many scheduling policies are proposed. In 1973, Liu et al. proposed the earliest deadline first (EDF) [2]. This policy schedules task with the earliest deadline according to the time distance between current system time and deadline of each task when system is running, which belongs to dynamic priority policy. EDF can be divided into partitioned earliest deadline first (P-EDF) [3] and global earliest deadline first (G-EDF) [4][5] according to its locality. P-EDF selects task with the closest deadline in each CPU core, while G-EDF selects task with the closest deadline in all CPU cores, and carries out inter-core migration and load balancing.

Since the dynamic priority policies need to calculate the priority of tasks at runtime, resulting of cost of performance of embedded systems which contain limited computing resources, people proposed some static priority policies, such as time slice polling, table scheduling, fixed priority scheduling(FP) [6][7], etc. The time slice polling policy allocates time slices for all tasks in advance. When the task is running, the system reduces its time slices for each time tick. If the time slice is reduced to 0, the task will be suspended and put to the end of the system ready queue. On the basis of time slice polling, the table scheduling policy adds concepts such as worst case execution time (WCET) and relative deadline to calculate the delay of the task in advance and take it as the time slice of task. In addition, table scheduling also adds the concept of task offset for task access control. FP assigns tasks with fixed priority and pick unfinished task with highest priority to execute when schedule is triggered. Like EDF, FP can be divided into partitioned fixed priority(P-FP) and global fixed priority(G-FP) according to its locality. P-FP selects task with highest priority in each CPU core, while G-FP selects task with with highest priority in all CPU cores. Like G-EDF, inter-core migration and load balancing are carried out if needed in G-FP. Considering the performance and character of target embedded operating system, this paper adopted static priority policy, where parent thread uses table scheduling policy and child thread uses partitioned fixed priority scheduling policy.

2.2. Embedded operating system
With the development of software and hardware technology, people find that embedded operating system (EOS) has superior performance in reliability, portability, low power consumption and so on. There are some representative work among the mainstream EOSs. VxWorks [8], developed by Windriver Corporation, is an embedded operating system widely used in the most applications. It supports functions such as rapidly interrupt processing and task switching, with strong real-time performance and high expansibility, and has supporting development platform Tornado. VxWorks has been widely used in industrial control, national defense, automotive and other fields. ECOS is a miniature embedded operating system released by Red Hat Corporation. It is free and open source, with abundant configuration options, and is completely written in C++, which leads to its limited performance. Another EOS for teaching, aCoral is developed by the Embedded Real Time Team of UESTC. It supports various ARM processors and contains some multi-thread scheduling policies internally, such as time slice polling, but it does not yet support two-level scheduling, and its thread control block (TCB) structure does not have fields related to two-level threads.

2.3. Thread creating and copy
When applications become complex enough, a multi-threads environment emerged. Among them, the Linux thread model is one of the mainstream models: to create a thread, system needs to go through steps of allocation of TCB memory space, TCB fields initialization, and putting the new thread into the
ready queue. In order to realize thread replication, pointer fields in TCB structure need to be divided into three categories in advance: shareable, copyable and thread private, and basic data type fields in TCB structure need to be divided into two categories: copyable and thread private. When the parent thread fields are copied, shareable pointer fields are directly copied using shallow copy. For copyable pointer fields, the contents of pointer are copied using deep copy. For fields of basic data type that can be copied, direct assignment is adopted for copy. For thread-private fields, the child threads need to assign values as needed themselves.

This paper migrates this model to target EOS and implements sub thread priority array, sub thread completion flag array, and saving the executing sub thread ID to its parent thread stack for two-level scheduling.

3. Method

In order to solve the problem of sub thread creation and two-level scheduling, a two-level scheduling policy and a suitable sub-thread creation model are proposed in this paper. This section describes the details of both models.

For simplicity, this two-level scheduling is for application threads only. Therefore, the parent thread pointer of the top-level application thread created by the init system thread is set to null. Besides, this paper supposes that every application thread is executed permanently and never released.

3.1. Two level scheduling

The proposed two-level scheduling policy is that the parent thread uses table driven scheduling policy and the child thread uses fixed priority policy. When a new table driven parent thread is created, system allocates one blank TCB for it, assigns thread fields from parameters, and calculates its delay value based on its WCET, relative deadline, as well as CPU scheduling period, then inserts the parent thread into the tail of the global table scheduling list. When the clock interrupt is triggered, the procedure of table scheduling clock interrupt handler is shown in table 1:

| Table 1. Two-level table driven scheduling |
|-------------------------------------------|
| **Input:** global table scheduling thread list L |
| **Output:** next thread to be scheduled T |
| 1. **foreach** thread t in L: |
| 2. if t->delay == 1: |
| 3. T ← t; |
| 4. c ← get_child_from_parent_stack(T); |
| 5. if c != NULL: |
| 6. initialize c->stack; |
| 7. corresponding finish flag of c in T← 0; |
| 8. put c into tail of CPU ready queue; |
| 9. move T to the tail of L; |
| 10. reset T->delay to initial value; |
| 11. put T into tail of CPU ready queue; |
| 12. T->state ← ready; |
| 13. initialize T->stack; |
| 14. save id of T into its parent stack; |
| 15. trigger schedule; |

Before the initialization of thread stack, check whether the stack of parent thread holds the ID of the last child thread that did not finish executing when this parent thread was switched out in r1 register. If so, stack initialization and putting thread into tail of CPU ready queue are taken again on the corresponding child thread to execute it again after system switches this parent thread in, while the related finish flag in parent thread is set to 0, meaning this child thread has not completed yet.
For a fixed-priority child thread, system maintains a priority array of child threads in the parent thread, and each child thread is inserted at the appropriate location based on its priority, which is described in detail in the next section. Child thread does not respond to clock interrupts, but let the parent thread schedule it fully. In addition to checking for last unfinished child threads when responding to clock interrupts, parent thread also selects the first child thread with a finish flag of 0 from its child thread priority array to schedule and saves the child thread ID in the r1 register of its stack when its function has finished or this parent thread is about to suspend itself.

When the thread function finishes executing, thread also sets the corresponding completion flag in its parent thread to 1, indicating that current child thread has completed, and set the value of register r1 in the stack of the parent thread to -1, indicating that there are no unfinished child threads before next parent thread switch happens. If a thread is suspended when all of its child threads have finished executing, it removes all of the child threads and itself from the ready queue of the CPU they belong to, and sets completion flags of all child threads to 0, meaning none of child threads haven’t been scheduled in next new execution period of parent thread.

3.2. Sub thread creation
To implement the creation of child threads, that is to say, thread replication, fields in TCB are divided into three categories: shareable, copyable, and thread private. Shared fields include waiting resource list, inheritance priority pointer, etc. Copyable fields include slice, delay, console_id, CPU, etc. Thread private fields include thread execution function and its arguments, thread exit function, stack, thread name, parent thread pointer, child thread array, etc.

When creating a child thread, a blank TCB is first allocated, then deep copy is made to the copyable fields, shallow copy is made to the shareable fields, thread private fields are configured according to the parameters passed by the user, and then parent-child thread association is carried out.

The thread that performs creation of the child thread is the parent of new thread. This paper assumes that threads in the same child thread array have same scheduling policy, so if the array is not empty and the scheduling policy of the newly added child thread is not consistent with that of the element in the array, the new child thread will become a separate parent, with null parent pointer. Fixed-priority child thread uses time slice to define its priority: the shorter the time slice is, the higher the priority is, and the earlier it is executed when the parent thread suspends. If two child threads have same time slice, threads created earlier executes first. When a child thread is created, system places it in front of the first element who has greater time slice, and the corresponding completion flag in the parent thread is set to 0. Finally, the child thread is inserted at the tail of the CPU ready queue without triggering scheduling and the parent pointer of the new child thread points to the current thread, which is its parent(null if it doesn’t have parent).

All child threads have a lower priority than their parent to ensure that the parent thread is scheduled by the system before the child

4. Experiments
In order to verify whether model proposed above can maintain the parent-child thread structure and schedule the parent-child threads according to the specified parameters, Several experiments were conducted on the creation of child threads and two-level scheduling. These experiments are described in detail in this section.

All experiments below were implemented in C, compiled with arm-none-eabi-2010 cross compiler in Ubuntu 16.04 x86_64 and performed in single-CPU embedded development board arm mini 2440, which supports gcc cross compiler with EABI. Besides it can update and boot kernel by simply uploading bin file.

4.1. Task details
In experiments, the structures of parent-child thread were shown in figure 1, where td_1_1 and td_1_10
were top-level application threads, td_1_1_slice_1 and td_1_1_slice_2 were child threads of td_1_1 and elements of td_1_1 children array. Their parent pointers all pointed to td_1_1; td_1_10_slice_1 was the child of td_1_10 and the only element of the td_1_10 children array. Its parent pointer pointed to td_1_10.

![Figure 1. Parent-child thread structure](image)

All threads were created, executed, suspended on CPU 0 of arm mini 2440 development board and the control period of CPU 0 was set to 150 time units. For convenience, all units of time were abstract logical time units. All parent thread parameters in figure 1 were shown in table 2. As a result, td_1_1 had higher priority for its delay value was lower.

| Thread name | WCET | Relative deadline | Delay | Offset |
|-------------|------|-------------------|-------|--------|
| td_1_1      | 30   | 50                | 80    | 100    |
| td_1_10     | 30   | 60                | 90    | 200    |

For child threads parameters in figure 1 were shown in table 3, td_1_1_slice_2 had higher priority in td_1_1 with lower slice.

| Thread name       | Slice |
|-------------------|-------|
| td_1_1_slice_1    | 210   |
| td_1_1_slice_2    | 200   |
| td_1_10_slice_1   | 200   |

In order to verify the improvement of newly proposed method, results of two-level application thread creation against one-level creation were compared in next section, as well as the result of two-level application thread scheduling against one-level scheduling.

4.2. Results

After all threads were created, system printed all thread information to test application thread creation. Results of single-level and two-level thread creation were depicted in figure 2 and figure 3 respectively.

![Figure 2. Thread structure result of single-level thread creation](image)
As shown in figure 2, affiliation of application threads could not be maintained when single-level thread creation was adopted because relationship between td_1_1_slice_1 (or td_1_1_slice_2) and td_1_1 was not reflected, neither was relationship between td_1_10_slice_1 and td_1_10.

As shown in figure 3, parent-child structures of all application threads were shown correctly when two-level thread creation was used: table driven thread td_1_1 and td_1_10 had no parent thread, while slice thread td_1_1_slice_1 and td_1_1_slice_2 both had parent thread td_1_1 with scheduling policy TD (table driven). Slice thread td_1_10_slice_1 had td parent thread td_1_10.

For convenience, thread route function simply printed information of parent thread, current thread and all child threads in order when testing application thread scheduling. Both parent and child threads executed periodically. Results of single-level and two-level thread scheduling were depicted in figure 4 and figure 5 respectively.

**Figure 3.** Thread structure result of two-level thread creation

As shown in figure 2, affiliation of application threads could not be maintained when single-level thread creation was adopted because relationship between td_1_1_slice_1 (or td_1_1_slice_2) and td_1_1 was not reflected, neither was relationship between td_1_10_slice_1 and td_1_10.

As shown in figure 3, parent-child structures of all application threads were shown correctly when two-level thread creation was used: table driven thread td_1_1 and td_1_10 had no parent thread, while slice thread td_1_1_slice_1 and td_1_1_slice_2 both had parent thread td_1_1 with scheduling policy TD (table driven). Slice thread td_1_10_slice_1 had td parent thread td_1_10.

For convenience, thread route function simply printed information of parent thread, current thread and all child threads in order when testing application thread scheduling. Both parent and child threads executed periodically. Results of single-level and two-level thread scheduling were depicted in figure 4 and figure 5 respectively.

![Table of Thread Information](image1)

| Name      | Father's type | Type | Console | State       | Father's name |
|-----------|---------------|------|---------|-------------|---------------|
| idle      |               |      | 4102    | Ready       |               |
| daemon    |               |      | 4102    | Sleep       |               |
| shell     |               |      | 4102    | Running     |               |
| td_1_1    | TD            |      | 4102    | Sleep       |               |
| td_1_10   | TD            |      | 4102    | Sleep       |               |
| td_1_1_slice_1 | | Slice | 4102 | Sleep | td_1_1 |
| td_1_1_slice_2 | | Slice | 4102 | Sleep | td_1_1 |
| td_1_10_slice_1 | | Slice | 4102 | Sleep | td_1_1 |

**Figure 4.** Single level scheduling result

Single-level scheduling execution test result was shown in figure 4, where first execution of application threads could get expected results while following period execution only scheduled parent thread because slice scheduling didn’t support periodic scheduling, indicating single-level scheduling can not guarantee correct execution order during runtime.

Two-level scheduling execution test result was shown in figure 5, where in every execution period, the first parent thread td_1_1 was executed first, then its children td_1_1_slice_2 and td_1_1_slice_1 started execution in order. After td_1_1 was done, the second parent thread td_1_10 and its child thread td_1_10_slice_1 started running in order. Therefore, expected execution order can be guaranteed by two-level scheduling policy proposed.

**Figure 5.** Two level scheduling result
5. Conclusions
In this paper, a two-level scheduling policy is proposed, where parent threads use table driven scheduling policy and child threads use fixed-priority scheduling policy as well as a related sub thread creation method. Corresponding implementations of these methods are also provided in an embedded operating system on arm 2440 development board with one CPU core. Compared with single-level scheduling policy and single-level thread creation method respectively, experiments showed that two-level scheduling policy and two-level thread creation method achieved expected results. In the future, more combinations of thread policies will be adopted and compared to extend the implementation of two-level scheduling policy. Besides, migration of proposed method into other development boards such as raspberry is also a challenging and exciting direction.

6. References
[1] S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," in IEEE Transactions on Computers, vol. 53, no. 6, pp. 781-784, June 2004, doi: 10.1109/TC.2004.16.
[2] Liu, C.L. & Layland, James. (1973). Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment. J. ACM. 20. 46-61. 10.1145/321738.321743.
[3] H. -E. ZAHAF, G. Lipari, S. Niar and A. E. Hassan Benyamina, "Preemption-Aware Allocation, Deadline Assignment for Conditional DAGs on Partitioned EDF," 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2020, pp. 1-10, doi: 10.1109/RTCSA50079.2020.9203643.
[4] X. Jiang, J. Sun, Y. Tang and N. Guan, "Utilization-Tensity Bound for Real-Time DAG Tasks under Global EDF Scheduling," in IEEE Transactions on Computers, vol. 69, no. 1, pp. 39-50, 1 Jan. 2020, doi: 10.1109/TC.2019.2936477.
[5] Q. Zhou, G. Li, C. Zhou and J. Li, "Limited Busy Periods in Response Time Analysis for Tasks Under Global EDF Scheduling," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 2, pp. 232-245, Feb. 2021, doi: 10.1109/TCAD.2020.2994265.
[6] G. Xie, W. Wu and R. Li, "Carry-out Interference Optimization in WCRT Analysis for Global Fixed-Priority Multiprocessor Scheduling," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, doi: 10.1109/TCAD.2021.3068940.
[7] L. Niu and D. Zhu, "Fixed-Priority Scheduling for Reliable and Energy-Aware (m,k)-Deadlines Enforcement with Standby-Sparing," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, doi: 10.1109/TCAD.2021.3061522.
[8] H. Neugass, G. Espin, H. Nunoe, R. Thomas and D. Wilner, "VxWorks: an interactive development environment and real-time kernel for Gmiocroc," Proceedings Eighth TRON Project Symposium, 1991, pp. 196-207, doi: 10.1109/TRON.1991.213103