Scheduling Algorithm for Real-time Multi-core Operating System Based on Multi-Master Mode

Jianpeng Zhao, Tieliang Ren, Zehan Shi and Jialong Li

College of Automation, Chongqing University of Posts and Telecommunications, Chongqing 400065, China

Abstract

The traditional master-slave multi-core operating system (RTOS), which is responsible for the resource allocation and task scheduling of the management system, is only responsible for the implementation of the kernel, which is easy to cause the performance bottleneck due to overloading the main core , Thus reducing the multi-core parallel processing capabilities, affecting the overall multi-core operating efficiency. In this paper, a multi-core scheduling algorithm based on multi-master mode is designed. In the aspect of task management, a multi-master scheduling strategy based on event and priority preemption is adopted. Combining the token mechanism and the high priority task search algorithm is designed and implemented. Task scheduling and dynamic task assignment task scheduler, a good solution to the master and slave scheduling of the main core performance bottlenecks, while the dynamic task allocation is conducive to improving the processor load balancing, and because there is no task migration, Reduced scheduling overhead to ensure real-time, token mechanism to effectively reduce the shared scheduler when the access conflict, to ensure that the kernel multi-task synchronization and efficient operation.

Keywords

RTOS; Multi-Master Mode Multi-Core Scheduler; Scheduling Token; Load Balancing

1. Introduction

There are two main multi-core operating system architectures that support SMP: Master-slave single-core structure and symmetric multi-master single-core structure. Most of them use master-slave architecture such as Linux, Vxworks and so on. Through a main core to manage all the resources of the operating system, including task scheduling and communication, but in the increasingly complex practical applications, the increasing number of processing cores, the main need for the management of the core from the allocation of resources and scheduling, Can only deal with a time from the core of the scheduling or resource applications, low efficiency, easy to overload due to the main performance bottlenecks, affecting the multi-core parallel processing capacity [1]. The kernel structure based on multi-master mode allows the entire kernel to have the right to manage system resources and active scheduling, effectively solve the above problems.

Consider the SMP architecture of the processor is mainly based on the bus shared memory structure [2-3], multiple processor cores can access the processor peripherals and shared memory through the internal high-speed bus and enable data via cache and main memory Inter-nuclear transmission. Using a kernel architecture based on a multi-master single kernel, by placing the kernel code in the shared main memory and using a lock mechanism with inter-core mutex, it is possible to allow multiple processors to pass through the sequential bootstrap process The core concurrently performs a shared set of kernel code [4]. Compared with the multi-core architecture of multi-core architecture, this method can greatly reduce the memory consumption, improve the efficiency of memory, while the task of communication design and scheduler design is also easier, reducing the complexity of the system kernel design.

2. Design and Implementation of Multi - core Scheduler with Multiple Master Scheduling Dynamic Allocation

2.1 Multi-core scheduling based on token and multi-master mode

The so-called multi-master mode is the use of multi-master scheduling strategy design scheduler for multi-task scheduling. Multi-master scheduling is relative to the single master or master-slave scheduling strategy in terms of [5], that is, each processing core in the system can get the kernel scheduler resources, and then take the initiative to schedule the task, rather than by a fixed The main core is responsible for resource management and task allocation, and other cores perform only task procedures. This method effectively solves the performance bottleneck caused by the overloading of the main core in the master-slave mode, and can fully implement the multi-core parallel execution ability. At the same time, through the token mechanism to synchronize multiple cores to the shared scheduler resource competition request, did not get the token processing core does not need to wait for the release of the scheduler, but automatically give up the scheduling, continue to implement the original program, which Way to effectively improve the overall operating efficiency of multi-master single-core operating system. The token-based multi-master scheduling model is shown in Figure 1 below:

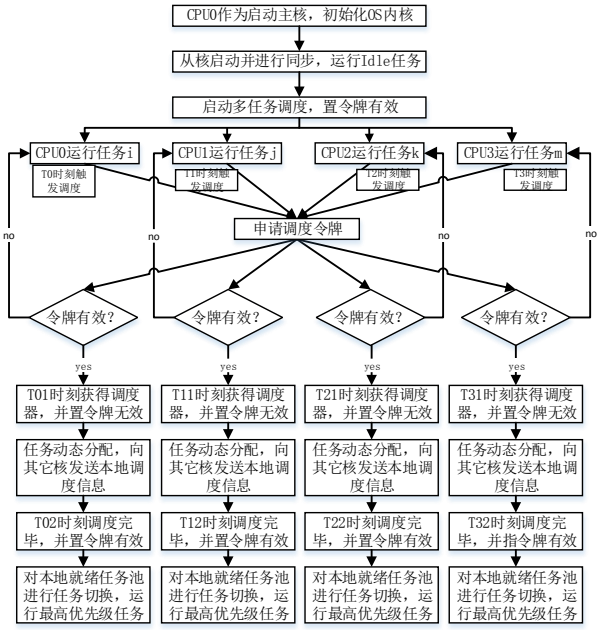

Figure 1. Token-based multi-master scheduling model

2.2 Global preemptive task dynamic allocation algorithm based on priority preemptive strategy

First create two mapping tables: assignable task mapping table and assignable processor core mapping table, respectively, to record the priority of assignable high priority tasks, and the corresponding assignable task processor core ID number. At the same time, add two members for each task's task control block, one for recording whether the task occupies the processor's state, and one for recording the processor ID number occupied by the task. And then copy a copy of the global task ready table, while locking the scheduler through the token, and through the spin lock to protect the global ready table.

And then from the global task ready table to find a copy of the processor core number of the same set of the same set of the new highest priority tasks, and in accordance with the task control block records the processor occupancy status and the corresponding core ID number, The tasks on the processor core also mean that the corresponding processor cores for these tasks are not assignable, so the bits corresponding to those processor cores need to be set to non-assignable flags in the assignable processor core mapping table. After the completion of this process, the remaining new high priority tasks must be assignable, the priority of these tasks will be stored in the assignable task mapping table,

and in the assignable processor core mapping table in the remaining significant bit corresponding Of the processor core must also be the core of the distribution, the core of the task will be running the new high priority task to seize.

Finally, these assignable new high-priority tasks are assigned to individual assignable processor cores based on the assignable task mapping table and the assignable processor core mapping table, and the priority of the comparison task is no longer required The assignments with the highest priority assignments are assigned in the order in which the assignable processor core IDs are numbered from low to high.

2.3 Priority-based global task ready table

The global task control block queue, which is stored in the shared memory as a global array, and is accessed by pointing to the pointer array of the task control block. It mainly includes the task control block array OSTCBTbl [ ] and the pointer array OSTCBPrioTbl [ ]. The subscript of the pointer array is the priority of the task. The basic element of the pointer array is the pointer to the created task control block. OSTCBTbl [ ] is stored in the task control block entity, the kernel of the queue insert and delete operations will only be in the task creation and deletion stage, and the task, whether in the scheduling of the allocation and switching phase, the task entity exists in the global queue.

The basic elements of the task-ready table are the priority of the task, and the task control block and the resources owned by the task are managed by the kernel. This is mainly to improve the maintenance efficiency and reduce the insertion and deletion of the task. The structure of the task ready table is shown in Figure 2:

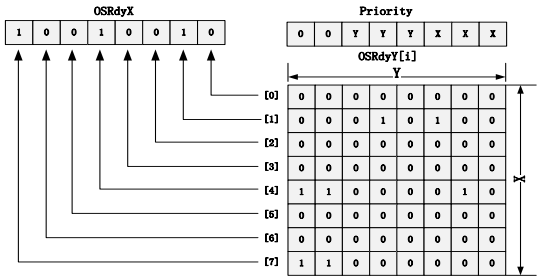

Figure 2. Priority task ready bitmap structure

3. Experimental results and analysis

3.1 multi-core task scheduling test

First create 6 tasks and a message queue event. Task Tsk3-Tsk6 these four tasks send a message to task Tsk2, respectively, and sending a message primitive triggers a task schedule. Task Tsk2 is only responsible for receiving messages, and then print messages, if there is no message to block them and trigger scheduling. Task Tsk1 does not send a message, but Tsk1 runs 10 times after the active call to suspend the function to suspend itself and trigger the dispatch. Then, each task by calling the kernel delay function to block themselves and drive the task scheduling, which task Tsk1 delay 20 rhythms, Tsk2 no delay, the remaining tasks are delayed 80 beats. Finally open the kernel clock beat interrupt, download the test code to run in shared memory, start multi-core run mode.

A total of test and statistics of the three tasks between the communication and scheduling, were tested 20 times, 40 times and 60 times the task of communication, corresponding to the task of sending four messages in the three tests, respectively, to the task Tsk2 send 5, 10, 15 messages, and then through the serial port to print test data, and statistical analysis. Task Tsk1 does not send a message, blocks itself through the kernel delay function, and runs only 10 times, where the first test to print the received message value of the situation and six tasks in the three core of the operation to verify the correctness of the message data transmission. The message print format is as follows: Sq.cid represents the core of the task of sending the message, Sq.tid represents the ID of the task that sent the message, and Rd.mes represents the received message value, that is, the task count value of the

message task. The remaining two tests only print each task in the core of the operation to analyze the multi-task allocation and scheduling situation.

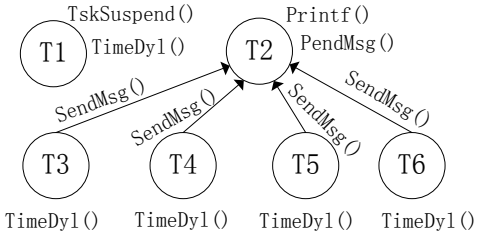

Figure 3. The relationship between the six test tasks

3.2 Test results

The data in Fig. 4a is analyzed. First, the content of the message Tsk2 received is consistent with the message content sent by the sending task, and the total number of messages received by task Tsk2 is 20 and the number of messages sent in total with the task Tsk3-Tsk6 Equal, message sending and receiving is correct, indicating that the message queue module can achieve the basic task communication function. Second, from the multi-core start-up process and the distribution of tasks on each core in Figure 4a, it can be seen that task Tsk1 is run on three cores, and other tasks that send messages are running on different cores, Note that the kernel delay function can block the task and trigger the scheduling, indicating that multiple cores can successfully start and multi-task scheduling and message communication.

Figure 4b, Figure 4c, Figure 4d print the data, a total of six columns, each column represents a task of a group of records, respectively, record the number of tasks in each of the three cores. Where the last row of data in the first column indicates the core ID of the task Tsk2 and the number of times the task is run.

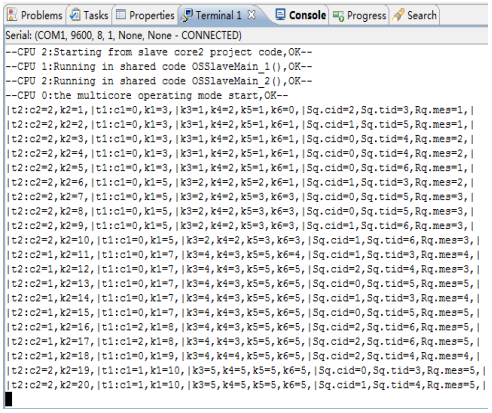

Figure 4a. 20 times the data reception of communication

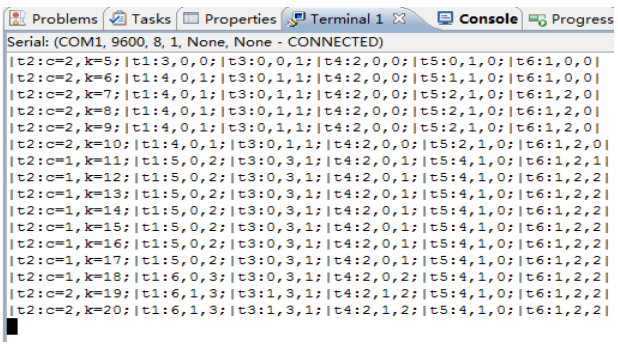

Figure 4b. 20 times communication and task scheduling situation

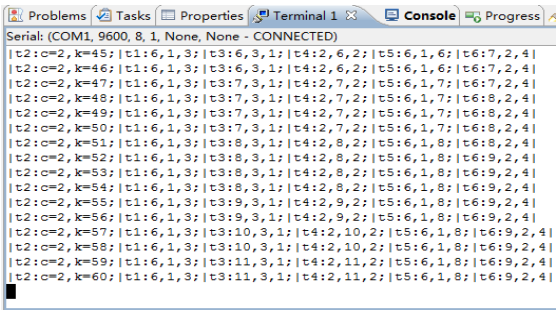

Figure 4c. 40 times communication and task scheduling situation

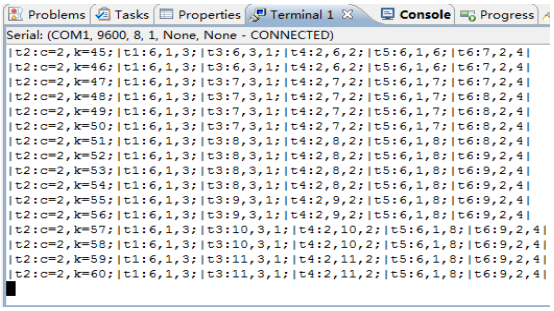

Figure 4d. 60 times communication and task scheduling

The total number of runs of the four sending tasks in the test result is equal to the number of times the task Tsk2 has been run, further verifying the correctness of the communication between the sending task and the receiving task. Second, in the first two tests, each task can basically be scheduled to run on different cores, and the total number of tasks on each core task is not much difference, but in the third test, the emergence of nuclear 2 Relative to the phenomenon of overload, in which the task Tsk2 on the nuclear running 48 times, Tsk2 reason to run on a certain high frequency, because it is the second task of the second high priority task, and the task Tsk1 Run 10 times after being suspended, behind the dozens of communications, there is no task to seize Tsk2, unless it is due to the application of the message and blocked; Second, because it does not delay function, will not delay their own due to delay , And there are four tasks to send a message to it. So, on the whole, the scheduler can achieve multi-core task dynamic allocation and scheduling, and can basically achieve multi-core processor load balancing.

4. Summary

Based on the multi-master single-core architecture, the search algorithm is based on the token mechanism and the high-priority task, based on the event-based priority preemptive scheduling and multi-master scheduling strategy, the global task ready table and the dynamic task assignment algorithm of multi-task scheduler. Which is solves the performance bottleneck caused by the overloading of the main kernel. The task is based on the global readiness table and the priority preemptive scheduling to ensure the real-time performance of the kernel. Dynamic task allocation effectively solves the problem that the load Equalization problem, because there is no task migration, but also reduces the scheduling overhead; token mechanism effectively reduces the multi-core access shared scheduler code conflict, improve the scheduling efficiency.

References

[1] J.C. Jiang, F.D. Meng, B. He, etc. A multi-core real-time operating system multiple ready task fast search and scheduling method. China, 201410042680.3 [P]. 2014.

[2] Wolf W, Jerraya A A, Martin G. Multiprocessor system-on-chip (MPSoC) technology [J]. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2008, 27 (10): 1701-1713 The

[3] Khan G N, Tino A. Synthesis of NoC Interconnects for Multi-core Architectures [C] // International Conference on Complex, Intelligent and Software Intensive Systems. IEEE, 2012: 432-47.
[4] Taylor Michael. The Raw Prototype Design Document V5.02 [C] // Proceeding of the IEEE International Conference on Solid State Circuits, 2005: 1-107.
[5] Yuan Q, Zhao J, Chen M, et al. GenerOS: An asymmetric operating system kernel for multi-core systems [C] // IEEE International Symposium on Parallel and Distributed Processing. IEEE, 2010: 1-10.