Process Management of Linux in Embedded System Based on Priority-Driven Tactics

Chaojun Yan[a *], Wenbiao Peng[b], and Juling Zeng [c]

College of Computer and Information technology, Three Gorges University, Yichang 443002, China

[a] 78498164@qq.com; [b] 51219905@qq.com; [c] 2404569524@qq.com

[*] The corresponding author

Keywords: Linux; Embedded-system; Process; Priority-driven; Scheduling

Abstract: Process scheduling is the core element of multi-tasking and multi-user operating system especially in embedded system of Linux. In this paper, the problems of process scheduling of embedded Linux is analyzed and the opportunity of process scheduling and process scheduling algorithm are described. The real-time scheduling algorithm or process scheduling based on the Priority-Driven Tactics is majorly analyzed and its high efficiency is demonstrated.

1. Introduction

Linux is very suitable for the application of the embedded system with the features of kernel stable, powerful, scalable and low cost [1,2]. But Linux kernel itself does not have the feature of strong real-time and the kernel is large. Therefore, if you want to use Linux for embedded systems, Linux must be real-time and embedded. Linux achieves an efficient and flexible process scheduling for combining the characteristics of real-time processes and non-real-time processes (general process), and colligating the several scheduling strategy.

Processor (CPU) is the core resources of the entire computer system. In a multi-process operating system, the number of the process is often more than the number of processor, which will result in the process competing processors. Process scheduling has a decisive influence on the system functions to achieve and the various aspects of the performance. And its essence is distributing the processor fairly, reasonably and efficiently to each process. Scheduling is the necessary means to achieve the concurrent execution of multiple tasks. Different operating systems have different scheduling objectives. In the traditional Unix-class time-sharing system, the main objective of scheduling is to ensure the multiple processes equitably use system resources and providing better response time. In a strong real-time operating system, the high priority task always uses the processor as a priority.

Standard Linux kernel is non-preemptive kernel. The system uses FIFO's I / O mode of operation and information processing system [3]. Process scheduling strategy is the basic and the fair priority scheduling policy, since multiple processes shared resources in the process of process scheduling may lead to priority reversals. Process scheduling is the core of the operating system and the key of controlling the Linux kernel. Process scheduling is divided into two parts, one is scheduling time that when scheduling; one is the scheduling algorithm which is how to dispatch and dispatching which process.

2. Linux Process Scheduling Time and Scheduling Algorithms

Scheduling time is to run scheduler to choose the process running under what circumstances. Scheduler in the Linux system is through the function schedule () to implement. This function is called high frequency by it to decide to run which process.

In Fig.1, a multi-tasks(process) and multi-processors embedded operation system of Linux process scheduling is shown. Linux scheduling time is divided into two situations: active scheduling and passive scheduling [4]. Active scheduling calls directly schedule () to achieve the schedule when the

  DOI: 10.25236/meici.2019.039

state of process changes. Passive scheduling does not immediately dispatch, but only to mark the scheduling location of a current process as 1 when running time slice of a process is over or ready queue add a process. Before the system state from the core state changes to the user state to check the current process scheduling flag is 1, if 1, then it calls schedule () to dispatch.

One is the scheduling algorithm which is how to dispatch and dispatching which process. When the scheduler runs, there is to choose a process which is most worth to run. The basis of selection process mainly has scheduling policy (policy), static priority (priority), dynamic priority (counter) and real-time priority (RT-priority) of the process. First of all, Linux is divided into real-time process and the normal process for the whole. The two scheduling algorithms are different for the real-time process preference to the normal process to run. Processes are called in turn with the level of priority. The highest level of priority is real-time process.

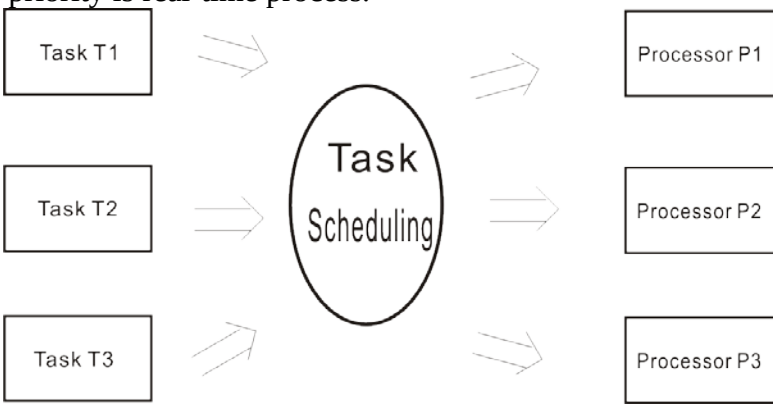

Figure 1. The task scheduling in multi-tasks and multi-processors operation system

3. Real-Time Scheduling Algorithms

Embedded Linux is not a real-time system. But the requirements are very high on real-time performance of control system in industrial control systems. It is firstly to address the issue of real-time performance if you want to apply the embedded Linux to industrial control system.

At present, these real-time scheduling algorithms can be divided into three types: priority-driven, time driven, and share-driven scheduling algorithm [5]. The share-driven scheduling algorithm is used relatively less. Priority-driven algorithm is used most. Priority-driven real-time scheduling algorithms are many, including the fixed priority scheduling algorithm, priority inheritance protocol, priority scheduling algorithm, etc. These scheduling algorithms have been applied to various areas. Each has their own advantages and disadvantages, such as fixed priority scheduling algorithm is simple, relatively easy to achieve. But the problems are most serious is the cause of deadlock and priority inversion problem; Priority inheritance protocol can solve deadlock and priority inversion problem, but it is cumbersome to achieve and demanding on the system is higher, although it can solve the problems of the deadlock and priority inversion. But the system efficiency is lower.

Priority-driven scheduling algorithm of real-time embedded Linux is based on "priority inheritance protocols (PIPs)". It is optimized on this basis to make it more suitable for embedded systems. Implementation works as follows: low-priority task can block the high-priority task when the low-priority task running in the critical .In the blocking process, the low priority task automatically inherits the priority of the high priority tasks which are blocked; In addition, the system for every mission to set a maximum blocking time parameter to limit the maximum blocking time of high-priority task which is blocked in order to ensure high-priority task is not always blocked down.

In this algorithm, the scheduler distributes an execution time to each task and there is a variable function of time. The scheduler chooses the most important task to run according to the given execution time of tasks and time function of calculated value. In the implementation process, the scheduler tracks these tasks for the real-time and records their performance. These historical records are as a basis of calculating time function and distribution of execution time to prepare for the next

implementation.

4. The Priority Inversion Solution

Consider the following situation: High-priority tasks TH and low-priority tasks TL when running processes need to share a memory Y and both write operations [6]. They need a semaphore S to ensure exclusive access to shared memory in order to ensure data completely. Task TM has the priority TM between TH and TL.

1) Low-priority task TL obtained ownership of the semaphore S-that is to do P operation, but does not do V operation;

2) At this point when the interrupts of another high-priority task TH occurs, real-time kernel set the task TH to the running state through task scheduling to switch the task TL down

3) When TH start to execute and it need to access shared memory Y at half-way, it must do P operation on S as S has not yet resumed, TH is blocked on the semaphore S;

4) TL regain control and resume execution;

5) At this point when the interrupts of another middle-high-priority task TM occurs. The real-time kernel switches the task TL down again for the priority of TM is higher than TL. The task TM obtains the right of execution of CPU. The task TM whose priority is lower than the task TH can skip directly to TH to run. The task TH has to wait for the signal S by the task TL owning so that can not obtain the right of execution of CPU before the task TM.

There is an issue of priority inversion. The performance of tasks TH and tasks TM act as their priorities like upside down. Extreme situation is that priority inversion phenomenon will not stop as long as the priority task TM emerging. It makes the behavior of the system become unstable and impossible to predict.

Using real-time scheduling algorithm of described above to solve the priority inversion process is as follows:

1) Low priority task TL locks the priority inheritance semaphore S;

2) High-priority task of TH appears and obtains the right of execution from the task TL;

3) TH is failed to attempt to lock S and it is blocked. The real-time kernel module obtains the control rights;

4) The core of real-time scheduling resumes the task TL to execute and through S to temporarily inherit the priority of TH;

The other middle-priority tasks TM appear, but can not get the right of execution from the tasks TL;

1) TL continues until it gives up the ownership of S to the TH. The scheduling core wills TL to the original of the priority;

2) TH resumes executing;

3) TM executes after TH is over.

It was seen from above that this real-time scheduling method can solve the problem of priority inversion and to achieve them is also very simple. It is very suitable for embedded systems.

5. Summary

This article discusses the characteristics of the process scheduling in Linux, focusing on the real-time scheduling algorithms of embedded Linux. It analyses a specific idea of priority-driven scheduling algorithm to achieve.

References

[1] C. Hallinan, Embedded Linux Primer: a Practical Real-World Approach, Second ed., 2011.

[2] Peng Dong, Inside Embedded Operation System: Design, Structure and Develop from Scratch, First ed., Mechanical Industry Press, 2015.

[3] D.P. Bovet, M. Cesati, Understanding the LINUX Kernel, First ed., O'Reilly Media, Inc., New York, 2005

[4] R. Love, Linux Kernel Development, Second ed., Novell Press, New York, 2005

[5] J. Corbet, A. Bubini and G.K. Hartman, LINUX Device Drivers, Third ed., O'Reilly Media, Inc., New York,2006

[6] W.R. Stephen, S.A. Rago, advanced Programming in the UNIX Enviroment, Second ed., Addison-Wesley, New York, 2006