User Adjustable Process Scheduling Mechanism for a Multiprocessor Embedded System

SLO-LI CHU, CHIH-CHIEH HSIAO, PIN-HUA CHIU, HSIEN-CHANG LIN
Department of Information and Computer Engineering
Chung-Yuan Christian University
200, Chung Pei Rd., Chung Li, 32023, TAIWAN

Abstract: The multiprocessor computer systems become more popular for the improvement of the processor's design and fabrication's evolution. In the past, the multiprocessors systems are usually adopted in server systems. Nowadays, the configurations of multiprocessor are also adopted in the personal computers or embedded systems. But we cannot achieve the optimal performance of the multiprocessor systems efficiently because task-scheduling mechanism cannot be adjusted for the real performing situation. Even the task-scheduler of the operating system cannot handle this problem. Also, the parallel program is hard to develop for the specific system to improve performance. In this paper, we provide a tool, called MONOPOLY, for users to adjust the scheduling policies of OS dynamically. It can improve the utilization of the multiple processors' resources by allowing user to occupy a processor to make the specified program execute solely. In our experiment, the results show that the program will finish faster by using MONOPOLY. When the system is fully loaded, the execution time of the program with MONOPOLY will be much shorter then the program without this tool.

Key-word: MONOPOLY, Multiprocessor, User Adjustable Process Scheduling, Operating System, Embedded System

1 Introduction

Embedded Systems become more popular, such as the mobile phone, MP3/MP4 player, and industrial computers. As we know the embedded systems, which are design for some specific applications. Accordingly, the designers usually want to reduce the costs by limiting system resources. However, the embedded applications become more complex. So the performance of the embedded system should be enhanced by some techniques, even by adopting SMP architectures.

Symmetric Multiprocessor, or SMP, is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use SMP architecture. SMP systems allow any processor to work on any task no matter where the data for that task is located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently. However, the increasing of processors are not always bring large performance enhancement. If we observe the task execution in the SMP system, the workload of each processor is not always balanced; some processor will go into idle. However, if we try to fully utilize the processors the program must be designed in parallel. When develop the parallel program, we need to consider the programming style of parallel processing or adopt some parallelizing compiler to transfer program. In fact, the software industries are lake of standard tools to optimize the program for each multiprocessor system automatically. By the way, the operating system generally use first processor to handle interrupts, so the scheduler will schedule the process to the first processor generally, only when the workload of processors are very imbalanced. This will causes the imbalanced workload and reduce system utilization.

Today, the trends of processor's design have been moved from developing higher frequency to reduce power consumption. In order to achieve high performance and low power consumption, the system architecture obviously change from higher frequency to multi-core processor, even in embedded computing. In many applications of embedded systems, more and more applications focus multimedia encoding/decoding, like 3G mobile phone and video camera with MPEG-4 features. They need powerful computation capability with small scale, high performance, low-power consuming, and reliable computation. According to these requirements, the multi-core or chip multiprocessor architectures seem to be the best answer. Therefore we develop a tool, called MONOPOLY, to fully utilize the capability of multiprocessor system according to actual user requirement. It allows user to adjust the scheduling mechanism of operating system for the specific program so the program can be performed solely and finish faster on the multiprocessor system

The organization of this paper is as follows. In section 2, the previous scheduling mechanism of Linux kernel and related study are reviewed. Section 3 presents the implementation of our MONOPOLY system in detail. In section 4 we demonstrate the experimental results and the speedup obtained when MONOPOLY is used. Finally, section 5 and section 6 give the conclusion and future works respectively.

2 Related Works

2.1 Introduction to Linux kernel scheduler

There is a brand new schedule mechanism in Linux kernel 2.6[1][2][4][5][6], which can finds the most suitable task to execute in the most situations. When a program is executing and entering the scheduling, this task will be putting in the corresponding priority's *runqueue*. In a period of time, timer interrupt will trigger schedule() to check the time slice of current tasks is running out or not. If yes, schedule() will find the next executing tasks from runqueue. According a normal queue, if there is a task running out its time slice, it will be removed from the head of queue to the tail of queue. Cause at least O(n) time complexity, for the needs of decreasing re-calculate loops. The scheduler maintains two priority arrays: *active* and *expired* for each processor. The active array keeps the tasks that haven't run out of its time slice. And in expired array, includes the tasks that have run out of its time slice. Time slice will re-calculate when a task runs off its time slice before moving to expired array. Active array and expired array will be exchanged while all of the tasks in active array run out its time slice.

```
struct runqueue{
    …
    struct task_struct *curr;       //current array
    struct task_struct *active;//active array
   struct task_struct *expired;   //expired array
    …
};
```

The structure of the priority array

And schedule() can easily find next highest priority tasks by the statements below.

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array array;
int idx;

prev = current;
array = rq->active;
idx = sched_find_first_bit( array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

The Fig.1 shows how the scheduling mechanism works. First, schedule() calls sched_find_first_set() to find the first bit in active array, and this bit corresponds to highest priority and executable task. Then this task will be executed by processor. Executing time of these statements doesn't influence on the number of tasks in the system – it's O(1).

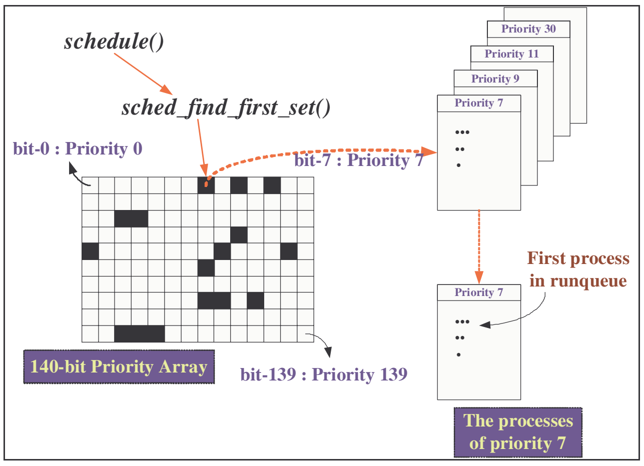

Fig. 1. The schedule mechanism of Linux kernel 2.6

While Linux kernel manages a SMP system, every processor has its own runqueue. Besides, within fix latency, the kernel has to check amount of tasks running on each processor is balance. If not, load_balanced() will move tasks between processors to make balanced workload.

2.2 ARTiS

ARTiS is a real-time Linux extension that targets SMP. The goal of ARTiS (Asymmetric Real-Time Scheduling) Project [3] is accelerating real-time tasks' response latencies. RT0 means hard real-time which needs to be done as soon as possible, and RTn is soft real-time. When ARTiS is booting, all of the processors will be partitioned into two parts — RT and NRT. RT processors are specialized to execute real-time tasks, and NRT processors are specialized to execute non real-time tasks. The Fig.2 and Fig.3 show the difference between the real-time scheduling policies of Linux and ARTiS

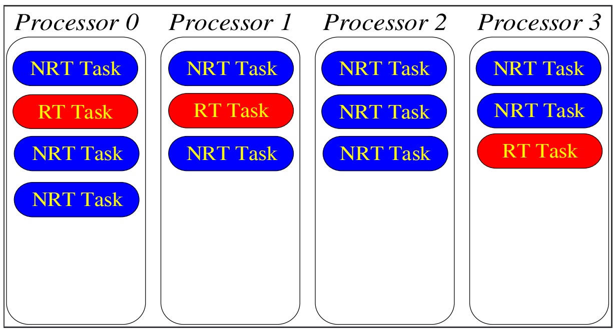

Fig. 2. The original scheduling mechanism of Linux kernel.

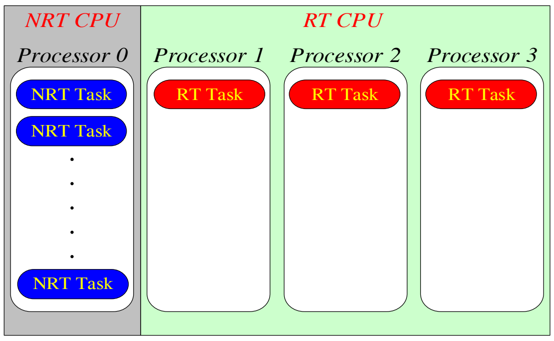

Fig. 3. The scheduling mechanism of ARTiS

When ARTiS is started, all of the RT tasks will be arranged to RT processor, and NRT tasks will be moved to NRT processor. When there is a free RT processor, NRT tasks will be moved to RT processor. In other condition, while the number of RT tasks is larger than RT processors, those RT tasks that haven't occupied processor will be arranged to NRT processor by ARTiS load balancer. And Linux will follow the original mechanism when executing NRT tasks.

ARTiS, using a task FIFO to save the moving tasks in stead of locking two runqueues to diminish latencies during moving tasks between NRT processor and RT processor. If RT processor is fully loaded and there is a RT task waiting for execution, the RT task will be moved to NRT processor. When there is free RT processor, the RT task will move to RT processor through the task fifo. So that two runqueues do not need to wait for spin lock. Make the arrangement for RT tasks more efficiently in multiprocessor system.

3 Implementation

In this paper, we bring up a mechanism. When the mechanism started, the program, which is selected by user, can occupy the selected processor. The Fig. 4 and 5 show the mechanism we want to implement.

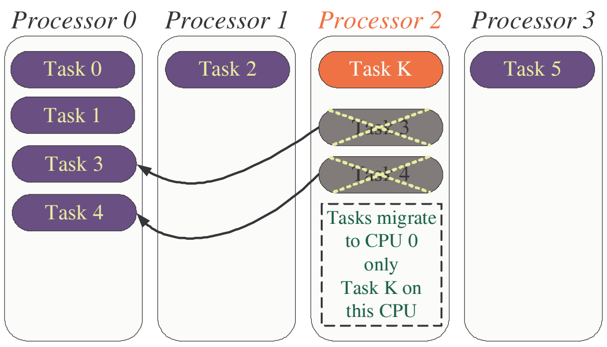

Fig. 4 A snapshot of a typical task scheduling

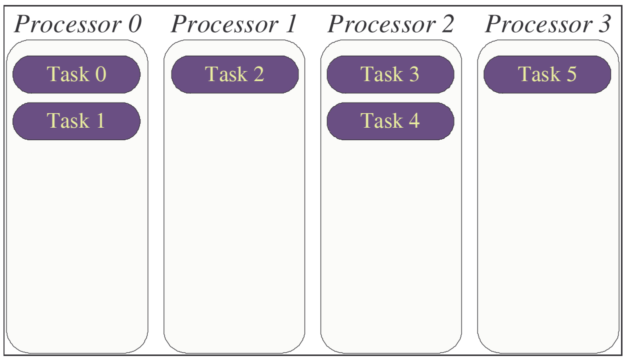

Fig. 5. When the Task 'K' is executing via our tool

When mechanism started, the tasks will be removed from processor 2 to processor 1, then put task 'K' to processor 2 as Fig. 5. From now on, every task can't be arranged to the runqueue of processor 2. But processor 2 follows Linux mechanism while our mechanism isn't started.

This tool needs to move tasks between processors. So we need to modify the kernel to meets our requirements. According to our code tracing, we get the following situations that kernel will move tasks between processors.

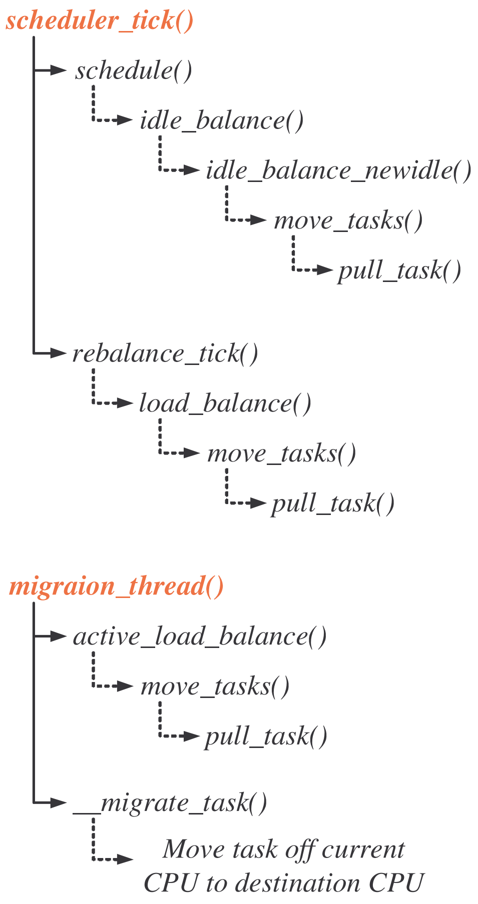

Fig. 6. The call graph of tasks' movements

In scheduler_tick(), if schedule() found that if a processor is going into idle, the scheduler will call idle_balance() to pull the tasks from other processors to prevent from unbalanced situation. And the rebalance_tick() will check the load-balance between processors via rebalance_tick(), if the the load is not balance, the load-balancer will adjust the loading to be balancing.

The migration_thread() is a special kernel thread, every processors have such kernel thread. In the Linux 2.6 systems, we can find this process exists with the name "migration/n". It is used to move threads between processors.

If we want the loading be unbalanced. We must try to prevent the tasks' movements in these load balance functions. And try to occupy one of the processor. These cause lots of code modify to the linux kernel. But in Related Works, we mentioned there's a research about enhance the Real-Time performance of the system, ARTiS.

The tool used in this paper is based on ARTiS Project that mentioned in Related Works. User only needs to select a program and which processor will execute this program. This tool will modify the attribute of the task to meet ARTiS's requirement. To make the task to be RT0 priority to occupy specific processor, we can set task as RT0 by the statements below.

```
#include <sched.h>

//declarate of scheduling policy
sched_param schp;

//get maximum priority (RT0)
schp.sched_priority =
    sched_get_priority_max(SCHED_FIFO);

 //set scheduling policy for the process
sched_setscheduler(0, SCHED_FIFO, &schp);
```

Besides, use the following statement to set which processor will be occupied by the task.

```
#include <sched.h>

//declarate of specific CPU
unsigned int spec_cpu;

//set the CPU affinity for the process
sched_setaffinity( 0, sizeof(unsigned long),
                                   0x1UL << spec_cpu);
```

The two setting procedures transfer the selected program to RT0 – i.e. it will occupy a selected processor. And the other tasks will be arranged to other processors. And this characteristic is perfectly meets our needs. Making the users can adjust processes' scheduling mechanism easily via our tool.

4 Experimental Results

We verify our first version of tool on x86 platform. We use Fedora Core 3 (Linux kernel 2.6.11 with ARTiS) as operating system and GCC 3.4 as compiler. The hardware settings are Pentium D 2.8GHz, 1GB DDR SDRAM with Intel 945G chipsets. The software to exam execution time is the MP3 converter "LAME" and MPEG encoder/decoder "MPlayer". We use LAME to convert wave file to mp3, MPlayer to convert MPEG-1 video to H.263+ MPEG-4 Video. When the test is running, we will also run some applications to ensure there are some loadings on the system. Note that the wave files in the test are 16-bit 44.1KHz format and the MPEG-1 video file is in 320x240 resolution with 29FPS.

4.1 The system with light loading

We're running X-Windows (GNOME 3.x) and MPlayer to play an MPEG-1 video. Compare the execution time of compressing different sizes of wave files by LAME. (Arguments of LAME: -b 320)

Table 1. Using LAME to convert wave to mp3.

Execution Time*(sec)*	*Wave file to test*		
	40MB	*100MB*	*400MB*
Linux	**47.43**	**127.37**	**497.02**
With Tool	**44.42**	**113.75**	**443.50**
Difference	**3.31**	**13.62**	**53.52**
Improvement	**6.8 %**	**10.1 %**	**10.8 %**

From Table 1, we can find if the program execute via our plug tool, the execution time is obviously decreasing. We can convert the wave file to mp3 files faster, as the file become larger the increments of execution time is more and more obvious. When the file size comes to 100MB, the improvement of execution time can reach 10%.

Next, we compare the execution time of the compressing MPEG-1 video file to H.263+ MPEG-4 video file via MEncoder in MPlayer. We use the 2-pass encoding mode, to convert the MPEG-1 file to H.263+ MPEG-4 file.

Table 2. Convert 50MB MPEG-1 Video file to MPEG-4 (H.263+)

Execution Time(sec)	*The pass of MPEG-4 encoding*		
	Pass-1	*Pass-2*	*Total*
Linux	**42.79**	**43.00**	**85.79**
With Tool	**32.22**	**31.13**	**63.35**
Difference	**10.57**	**11.87**	**22.44**
Improvement	**24.7 %**	**24.7 %**	**26.1 %**

Table 3. Convert 350MB MPEG-1 Video file to MPEG-4 (H.263+)

Execution Time(sec)	*The pass of MPEG-4 encoding*		
	Pass-1	*Pass-2*	*Total*
Linux	**282.31**	**285.35**	**567.66**
With Tool	**198.32**	**196.05**	**394.37**
Difference	**83.99**	**89.30**	**173.29**
Improvement	**29.7 %**	**29.7 %**	**30.5 %**

From the Table 2 & 3, we can find when the program executes via our tool. The execution time of the programs is improved a lot.

4.2 The system with higher loading

As above, the X-Windows and the MPlayer to play an MPEG-1 video, we also open another MPlayer to play an MPEG-4 (H.263+) video in X-Windows system.

Table 4. Using LAME to convert wave to mp3.

Execution Time(sec)	*Wave file to test*	
	40MB	*100MB*
Linux	**54.15**	**141.10**
With Tool	**46.11**	**115.33**
Difference	**8.04**	**25.77**
Improvement	**14.8 %**	**18.3 %**

Table 5. Convert 50MB MPEG-1 video file to MPEG-4 (H.263+)

Execution Time(sec)	*The pass of MPEG-4 encoding*		
	Pass-1	*Pass-2*	*Total*
Linux	**47.75**	**46.62**	**94.37**
With Tool	**33.12**	**31.08**	**64.20**
Difference	**14.63**	**15.54**	**30.17**
Improvement	**30.6 %**	**30.6 %**	**32.0 %**

Table 6. Convert 350MB MPEG-1 video file to MPEG-4 (H.263+)

Execution Time(sec)	*The pass of MPEG-4 encoding*		
	Pass-1	*Pass-2*	*Total*
Linux	**309.83**	**310.72**	**620.55**
With Tool	**197.61**	**199.31**	**396.92**
Difference	**112.22**	**111.41**	**223.63**
Improvement	**36.2 %**	**36.2 %**	**36.0 %**

Comparing the results above and the results with the system of light loading, we can find that the improvements of execution time are increased more than before. Take LAME for example, the execution time of this result can improve more about 8% and the converting of H.263+ is about 6%.

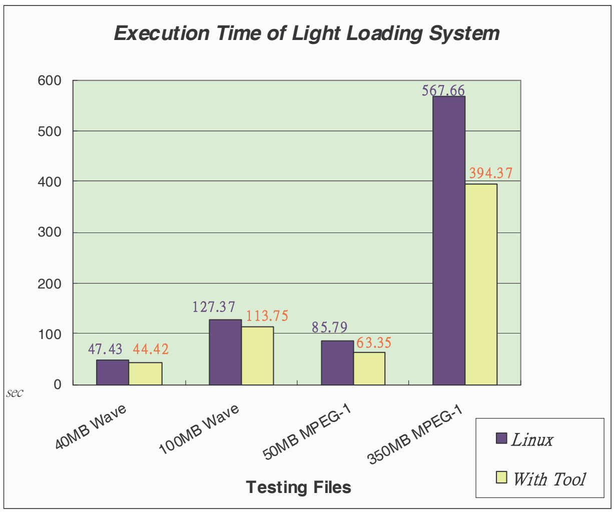

Fig. 7. Experimental results (Light loading system)

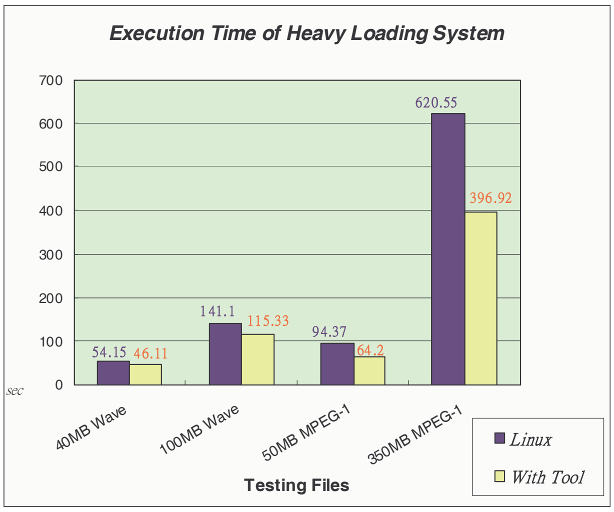

Fig. 8. Experimental results (Heavy Loading System)

Fig. 7 and Fig. 8 compare the experimental results. As we can find in the bar charts, the execution time is significantly decreased with our tool. All the results showed that if a program executed with our tool, the execution time would be less then the original OS scheduling mechanism.

5 Conclusions

In the paper, we provide a mechanism for user to adjust scheduling mechanism of the operating system, which allow a selected task to increase executing speed by occupy a processor solely.

From experimental result, we can make great speed enhance by letting a task occupy a processor. While the system is high loaded, a RT processor can concentrate on executing the selected task instead of serving many tasks with context switching. Otherwise the selected task will be scheduled with other tasks and waste time contest switching, and then the selected task needs more time to be finish.

Today, the multiprocessor architecture becomes more and more popular, but it's still a difficult topic to fully utilize the resources of processors effectively. Also, the operating system cannot assign the resources properly and it's hard to design a parallel program. So develop a user adjustable dynamic scheduling mechanism that can assign a task to a specified processor and execute solely is very helpful, not only accelerate executing speed of the selected task, but also increase the utilization of processors.

Multi-processor becomes mainstream in the territories of server and personal computer. But embedded systems do not have rapid computational ability as servers and personal computers. So the resources of processor are more precious in embedded systems. We can utilize resources more effective in embedded system by user adjustable scheduling to enhance efficacy.

6 Future Works

The tools used in the paper, included MONOPOLY and ARTiS, are only for x86 machines. They can't be used in most embedded systems such as ARM and PowerPC platforms. In the future, we will plan to port MONOPOLY to the selected embedded system, Xilinx ML310 Development Platform, which includes two PowerPC 405 processors that connected with IBM CoreConnect on-chip bus.

In Linux 2.6, there are 250 times timer interrupt in one second by default. That means it costs quite a few resources for scheduling and deciding to context switch or not. Accordingly, we also plan to design a hardware scheduler to reduce the workload of software scheduler between two processors on FPGA, to accelerate the performance of the presented user adjustable dynamic scheduler in this paper.

7 Acknowledgement

This work is supported in part by the National Science Council of Republic of China, Taiwan under Grant NSC 95-2221-E-033 -022 -

Reference:

[1] J. Aas, "Understanding the Linux 2.6.8.1 CPU Scheduler", 2005 Silicon Graphics, Inc, Feb. 2005.

[2] R. Love, "Linux Kernel Development", SAMS, Developer Library Series, 2003.

[3] E. Piel, P. Marquet, J. Soula, J.L. Dekeyser, "Asymmetric Real-Time Scheduler on Multi-Processor Architecture", 20th International Parallel and Distributed Processing Symposium, 2006 (IPDPS 2006), pp. 25-29, Apr. 2006.

[4] G. E. Allen and B. L. Evans. "Real-time sonar beamforming on workstations using process networks and POSIX threads", IEEE Transactions on Signal Processing, pp. 921-926, Mar. 2000.

[5] K. Morgan, "Preemptible Linux: A reality check", White paper, MontaVista Software, Inc., 2001.

[6] J. D. Valois. Implementing lock-free queues. In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, Oct. 1994.