The Operating System Project Report

The Research of Linux Process Scheduling

and Adding New Scheduling Policy for Linux 2.6

Fan Liu, Mi Gao

CS MET 575

BOSTON UNIVERSITY

Table of contents

1. Introduction ...……………………………………………………………………………. 3
2. The History of Linux Operating System ……..…………………………………….……. 3
   1. Linux Kernel 2.4 and Deficiencies …………….……....………………….....……. 4
   2. Linux 2.6 …………………………………………………………………...…...…. 4
3. Linux Kernel 2.6 processes analysis…………………………………………………..…. 4
   1. What is Priority and Why Should I Care ..………………………………………… 5
   2. NICE and Scheduling Policies …………………………………………………….. 5
4. Round-Robin Scheduling and Weighted Round-Robin Scheduling …………………….. 8
   1. The analysis and comparisons
5. Conclusions
6. References

**The Research of Linux Process Scheduling**

**and Adding New Schedule Policy for Linux 2.6**

1. Introduction

In this course, based on the knowledge from the professor and the textbook, our group became interested in the open-source operating system and its scheduler, which is why we set this as our topic of project; we decided to use Linux and its scheduler that introduced system in the course as research targets.

Real-time operating systems have been widely used in various fields at present, and they have attracted more and more attention. Among many real-time operating systems, Linux-based real-time operating systems are increasingly popular due to their open source code and system stability. However, Linux was originally designed and developed as a general operating system. The basic principle of its design is to minimize the average response time of the system and improve the throughput of the system. At this moment, people can find links that lead directly to the download page of twenty-five popular Linux distributions such as Ubuntu and CentOS; and the latest version of Linux is 5.4 released on 24th November 2019 (LINUX.ORG, 2019).

In this research, our team will reveal what we analyzed from the Linux kernel in our research such as ho the kernel and the scheduler work inside; also, we tried to apply a new scheduling policy are called weighted round robin and state the structure of it. Thus, in this report, our team will record almost the information about the knowledge we never knew before and the troubles we met. The picking of the working environment in which the team was taking the research with Linux 2.6 rather than a newer version based on many aspects. One of them was Linux 2.6 is a great environment for the beginner using Linux. Although Linux2.6 also provides some support for real-time processing, including support for most of the real-time functions in the POSIX standard, multi-tasking, multi-threading, preemptible kernel, O (1) process scheduling algorithm, etc. However, the real-time performance of Linux is still relatively poor, and the response time of real-time tasks cannot be guaranteed. However, the real-time performance of Linux is still relatively poor, and the response time of real-time tasks cannot be guaranteed. However, it really had been improved for a great step compared to Linux 2.4 and came with a lot of resources that can be referenced. From our perspectives, the team can learn how those parts work in the kernel and try to design our new project about the scheduling policy in this environment clearly.

1. The History of Linux Operating System

The era of Linux 2.6 is very large; there were spanning forty major releases from 2.6.0 released in December 2003 to 2.6.39 released in May 2011 (Torvalds, 2003). Linux 3.0 which was originally planned as 2.6.40 released in July 2011 until Linux 3.19 released in February 2015. Linux 2.6 marked a fundamental shift in the model of development. New facets would be constructed in an odd version number before kernel 2.6, and then probable backported to the same. Linux 2.6 Marked the point at which points had been constructed towards the current stable kernel and then landed if they were steady enough. For example, the difference between Linux 2.0.0 and 2.0.40 was tiny but kernel 2.6.38 got a lot of great updates from version 2.6.0. Therefore, although kernel 2.1, 2.3 and 2.5 had been viewed previous, there has never been kernel 2.7. It is because Linux kernel 2.6 confirmed that current revision manages systems allowed the consumer to actively evolve against a steady goal, putting off the want for a separate branch of development. That's why it was a turning point for Linux 2.6.

2.1. Linux Kernel 2.4 and Deficiencies

Based on the analysis of Linux 2.4 process scheduling, we found and concluded that the overall characteristics. First, kernel scheduling is simple and effective; then, the kernel is not preemptible. However, after analyzing the kernel 2.4, we also clearly saw its disadvantages because of three main points.

Firstly, the complexity of the scheduling algorithm is O (n), which has a greater relationship with the system load. Also, the scheduling algorithm is also flawed in design.

For example, process scheduling in Linux 2.4 only set up a process ready queue so that some processes will stay in the ready process queue after using up their time slice. In this way, although this process can no longer obtain the right to use the CPU in this round of scheduling cycles, it still has to participate in the calculation of the value of goodness(), which wastes time. Also, the ready process queue is a global data structure. There is only one ready queue which called “runqueue” for multiple CPUs, so all operations on the scheduler will wait between the processors of the system due to the global spin lock, making the ready queue become an obvious obstacle.

Moreover, the scheduling algorithm cannot be preempted in kernel mode. If a process enters the kernel state, no higher priority process can be deprived. Only when the process returns to the kernel state can it be scheduled; and more deficiencies such the lack of support for real-time processes which will not be discussed more because those are not the main orientations for the research in this project.

1. Linux Kernel 2.6 processes analysis

The scheduling algorithm of Linux 2.6 is much more sophisticated than the earlier versions. By design, it scales well with the number of runnable processes, because it selects the process to run in constant time. Designers rebuilt and made the optimizations for the scheduling algorithm from O (n) to O (1). “The scheduler always succeeds in finding a process to be executed; in fact, there is always at least one runnable process: the swapper process, which has PID 0 and executes only when the CPU cannot execute other processes (Yu-Hsin Hung, 2016).” Those are mainly due to the new improvement plan of scheduling policies in the kernel. Before the team tried to figure out how the scheduler get to be changed, we started from what is the scheduler and why the operating system needs it. Based on the messes found, we went back to the priority in the operating system.

* 1. What is Priority and Why Should I Care

When talking about processes priority is all about managing processor time. The Processor or CPU is like a human juggling multiple tasks at the same time. Sometimes we can have enough room to take on multiple projects. Sometimes we can only focus on one thing at a time. Other times something important pops up and we want to devote all of our energy into solving that problem while putting less important tasks on the back burner (Nixtutor, 2009).

3.2 NICE and Scheduling Policies

There are 2 types of processes, the *normal* ones and the *real time* for the normal ones and only for those, nice is applied as follows: **Nice.**

In Linux, when it looks at all the things it has to do, we will set guidelines for the CPU to follow(Nixtutor, 2009). Such criteria are known as good quality or elegance. The scale of Linux priority ranges from negative twenty to positive nineteen. The lower the number the more attention will be given to the mission. If the niceness rating is as high as nineteen, the assignment will be set to the lowest priority and it will be handled by the CPU whenever a chance arises. Through default, it gets the initial priority of null when a process starts. Generally, people use the following formula to explain the relationship:

*PR = 20 + NI*

Where PR and NI are the levels of priority and nice respectively. The negative twenty simply maps to zero, as people see in here, while the nineteen maps to thirty-nine. By default, the nice value of a program is zero, but the root user can run programs with a specified nice value using the following command; the user can view processes nice value by using command in user-defined format:

*ps -eo pid,ppid,ni,comm*

Then, the user can launch a program with your required priority using:

*nice -n nice\_value program\_name*

Also, to renice the priority of an running process using:

*renice -n nice\_value -p process\_id*

Or renice all running processes for a specific user:

*renice 20 -u user\_name*

Thus, for example in this case, the corresponding commands and output would be shown on the terminal as:

*% ps -o pid,comm,pri,nice -p $(pgrep chrome)*

*PID COMMAND PRI NI*

*2769 chrome 19 0*

*% renice 10 -p 2769*

*2769 (process ID) old priority 0, new priority 10*

*% ps -o pid,comm,pri,nice -p $(pgrep chrome)*

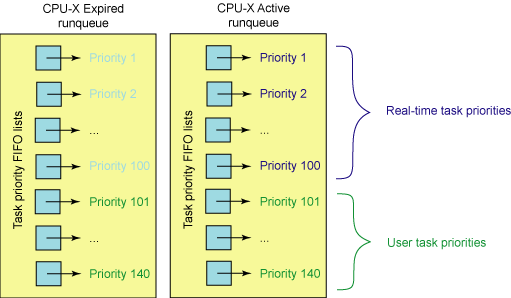
*PID COMMAND PRI NI*

*2769 chrome 9 10*

Based on the above content, the user will distribute their CPU resources more efficiently by using this scale. Non-important priority programs can be given higher importance, whereas high priority programs, such as daemons and utilities can be set to get more attention from the CPU. In all its phases, you can even assign a single client a lower value so that you can his ability to slow down the key services on the machine. If a user wants to change the process priority, simply use "nice" and " renice"; otherwise, by using "chrt" you can change the process property in more detail.

However, going through the development of hardware and operating system performance, this is not enough to meet user needs, and some special processes cannot be modified and operated in a static manner; also, this seems to be a manual and tedious operation that adjusting the operating system's priority by modifying the value of “nice” of the processes.

Typically, the nice priority is used for user programs. As the reference figure below, while the overall target of UNIX / LINUX is one hundred and forty, the pleasant quality allows the system to move from one hundred to one hundred and thirty-nine to the last part of the array. This formula leaves the values unattainable from zero to ninety-nine, which leads to a negative PR scale from negative one to negative one hundred; and those area of the priority value be described as the vaule to be able to access the program of the "real-time."



There have five kinds of scheduling policies in a Linux kernel 2.6. The user can check them by using following command: *chrt -m*

After the command execute, the terminal will show them as following list:

* + - 1. SCHED\_OTHER, the standard round-robin time-sharing policy
      2. SCHED\_BATCH, for "batch" style execution of processes
      3. SCHED\_IDLE, for running very low priority background jobs.
      4. SCHED\_FIFO, a first-in, first-out policy
      5. SCHED\_RR, a round-robin policy

Scheduling processes could be split into two groups; the normal scheduling policies which from the first to the third, and real-time scheduling policies are the fourth and fifth. The processes in real time will always have priority over normal processes (Kerrisk, 2019). Using the following instruction, a real-time process could be called; for example, to announce a SCHED RR policy:

*chrt --rr <priority between 1-99> ./myProgram*

To approximating with NI above, rhe following equation is used to obtain the PR value for the real-time process:

*PR = -1 - rt\_prior*

In there, where "rt\_prior" is the priority of one to ninety-nine. The system that will have the highest priority over other processes will, therefore, be the one labeled with the number ninety-nine. Furthermore, it is important to note that the nice value is not used for real-time operations. By using command, “top”, to see the current PR and NI values for each running process.

Which shows the following output in the screenshot clearly:

图片包含 文字

描述已自动生成

The PR and NI values are shown in the figure. In this case, They correspond to a real-time value with what has a negative PR value of fifty-one. Also, there are also certain processes with a PR value as "rt." In fact, this value corresponds to a negative one hundred PR value.

1. Round-Robin Scheduling and Weighted Round-Robin Scheduling

<https://www.geeksforgeeks.org/program-round-robin-scheduling-set-1/>

1. Weighted Round Robin’s General Idea:

while (true) {

i = (i + 1) mod n;

if (i == 0) {

cw = cw - gcd(S);

if (cw <= 0) {

cw = max(S);

if (cw == 0)

return NULL;

}

}

if (W(Si) >= cw)

return Si;

}

**I**: Process selected

**CW**: Current weighting

**Max**: Max weight of all process

**GCD**: common divisor of all processes weight

**First**: Use *mod* and a loop to find the maximum number of the process, and use the common divider of all processes weight to balance each process’s weight:

i = (i + 1) mod n;

if (i == 0) {

cw = cw - gcd(S);

**Second**: Use another loop to re-assign each process :

if (W(Si) >= cw)

return Si;

1. Implement the New Task Scheduler:

We do not need to implement the multiprocessor load-balancing at this time and assume that all the WRR process run on a single core. Most of the code of interest for this project is below:

*Kernel/sched.c*

*Include/linux/sched.h*

*Kernel/sched\_wwr.c*

1. Implement two system calls:

int get\_wrr\_weight(int pid);

int set\_wrr\_scheduler(int pid, int weight);

By default, Linux assigns a newly created process the completely fair scheduling (CFS) class. The first system call will change the process pid’s scheduling class to our newly implemented weight round robin policy and assign a weight to the process. The program first forks multiple process and issues the first system call to configure these processes under the weighted round robin policy with different weights. To verify that these process are running with weighted round robin, the test program issues the first system call to query individual processes’ weight.

1. Data Structure in Code:

* Runqueue: Each CPU in the system has its own runqueue defined in *kernel/sched.c*, all *runqueue* structures are stored in the runqueues per-CPU variable. There are two arrays which include in arrays field of the runqueue and consisting 140 doubly linked list head, a priority bitmap, and a counter of the processes. We want to switch between active process and expired processes, in doing so, we make the scheduler simple exchanges the contents of the *active* and *expired* fields of the runqueue.
* **Task\_struct**:

p->time\_slice = (current->time\_slice + 1) >> 1;  
current->time\_slice >>= 1;

p->first\_time\_slice = 1; /\* never exhausted its time quantum \*/  
p->timestamp = sched\_clock();

when a new process is created, sched\_fork() will set the current time slice and past time slice.

* **Wake up to process**:

try\_to\_wake\_up

This function awakes process p which whether sleeping or stopped by setting its states to TASK\_RUNNING and load into the active runqueue of the CPU.

static int try\_to\_wake\_up(task\_t \* p, unsigned int state, int sync)  
{  
int cpu, this\_cpu, success = 0;  
unsigned long flags;  
long old\_state;  
runqueue\_t \*rq;

* **Time Slice Definition:**

Use the DEF\_TIMESLICE which invoked by sched\_fork() to define the time slice we are going to use in *wwr.c*

#define TIME\_SLICE DEF\_TIMESLICE

#define GROUP\_TIME\_SLIC TIME\_SLICE \* 6

* **Find the process which current in the queue of CPU:**

static struct sched\_wrr\_supraentity \* find\_wrr\_user

sched\_wrr\_supraentity is a process set , then use find\_wrr\_user to capture new process and pack them into the process set.

* **Browse the entire process set**:

static void requeue\_wrr\_entity(struct wrr\_rq \*wrr\_rq, struct sched\_wrr\_supraentity \*wrr\_se, int head){

if(wrr\_rq->wrr\_nr\_users <= 0)return;

wrr\_nr\_users indicates the process selected last time, which initialized with -1

* **Equal the weight of the process:**

if(p->wrr.time\_slice > 0 && wrr\_supraentity->current\_time\_slice > 0)return;

if (wrr\_supraentity->current\_time\_slice == 0){

wrr\_supraentity->current\_time\_slice = GROUP\_TIME\_SLICE;

requeue\_wrr\_entity(&rq->wrr, wrr\_supraentity, 0)

requeue\_wrr\_entity is the new process set after requeued. This code is to compare if process in the old process set have the same time slice with the weighted time slice. If we got a zero in comparation we requeue this process.

Reference

LINUX.ORG, https://www.linux.org/

Torvalds, L. (2003). Linux-Kernel Archive: Linux 2.6.0, http://lkml.iu.edu/hypermail/linux/kernel/0312.2/0348.html

Nixtutor. (2009). Changing Priority on Linux Processes, <https://www.nixtutor.com/linux/changing-priority-on-linux-processes/>

Hung, Yu-Hsin. (2016). Linux Kernel: Process Scheduling, <https://medium.com/hungys-blog/linux-kernel-process-scheduling-8ce05939fabd>

Kerrisk, M. (2019) Linux Programmer's Manual, <http://man7.org/linux/man-pages/man7/sched.7.html>