Gluing Processes

ESTR 3102 - Linux Kernel Hacking Assignment 1

Abstract

The default Linux scheduler, the `SCHED_OTHER` policy, shares the CPU more or less equally among all the processes within the same class. Can we break that balance?

1 Description

Alice and Bob are using the same computer to do their projects. Bob is rather inconsiderate since he always launches his program before Alice, which is badly written and seizes the CPU intentionally. Alice complains this unfair treatment to their teacher Dr. Wong about this problem. Dr. Wong, therefore, asks you to modify the Linux scheduler to take care of Alice's complaint.

What Dr. Wong wants to implement is that, if Bob has launched a process and that process has taken up t_b units of the CPU time accumulatively, then when Alice has just submitted a process to the system, then Bob's process will be given less or no CPU resource. In other words, Alice's process will start catching up Bob's process. In a long run, the CPU time taken by Alice's process, t_a will approximately equal to t_b.

2 Milestones

As Dr. Wong is anxious about the realization of the new scheduler policy, he gives you a step-by-step guideline of the proposed scheduler policy.

2.1 Milestone 1 - adding system calls (3 marks)

You have to add the following two system calls to the Linux kernel:

- `asmlinkage long sys_glue(pid_t pid_1, pid_t pid_2, int ratio);`
- `asmlinkage long sys_glue_list(pid_t pids[2]);`
- `asmlinkage long sys_separate(void);`

2.1.1 The "`glue`" system call

Description. This system call tells the Linux scheduler to start modifying the scheduling behavior for the two processes specified by PIDs `pid_1` and `pid_2`, respectively. We will discuss the usage of the parameter `ratio` later, and this parameter is an integer.

When this system call is invoked,

- it first checks if the input PIDs are both valid.
- By valid, it means the two processes can be found in the system.
- If they are both valid, then the system call registers these PIDs into the kernel.
- Else, the system call returns an error.

By registering both processes, the kernel should **remember** the PIDs of the two processes specified by the "`glue()`" system call. Note that, by default, the Linux kernel should not have any PIDs registered.

Return values. The system call returns **0** when it registers the PIDs of the two processes successfully. The system call returns **-1** when it fails to register the PIDs.

2.1.2 The "`glue_list`" system call

Description. This system call saves the two PIDs that are previously-registered using the "`glue()`" system call. It is assumed that the input is a buffer which can hold at least two long integers. The system copies the two PIDs to the user-space buffer and returns.

Return values. The system call returns **0** when it finds the two PIDs registered. Else, the system call returns **-1** upon any errors.

2.1.3 The "`separate`" system call

Description. This system call tells the Linux scheduler to stop modifying the scheduling behavior for the two processes registered by the previous call of the "`glue()`" system call. Note that when any one of the processes is terminated, the unregistering should be done automatically.

Return values. The system call returns **0** for a successful unregistration. The system call should return **-1** without any gluing operation in process.

2.2 Milestone 2 - modifying the scheduler and the FOG policy (3 marks)

The Linux scheduler must be modified so that it pays special attentions to the processes registered by the "`glue()`" system call. In the Linux kernel v2.4, the Linux scheduler is implemented in the file: "`[kernel source root]/kernel/sched.c`".

One quick solution is to **freeze** the older process and let the younger process to run. When the **accumulative CPU time** of the younger process is larger than or equal to that of the old process, the older process is un-frozen. This sounds working! Let us name this policy the "*freeze the old guy*" policy, or FOG policy for short.

By "freezing", you are going to prevent the Linux scheduler from choosing the older process. Therefore, the modification of the scheduler is required. Once the two processes are glued together, this scheduling policy stops functioning automatically as if the "`separate()`"system call is invoked. To invoke the FOG policy, the ratio parameter of the "`glue()`" system call should be set to **0**.

Invoking the FOG policy.

```
#define NR_glue [some number]
int pid_1, pid_2;
......
syscall(NR_glue, pid_1, pid_2, 0);
```

Special Case. When an old process is frozen, it is not responsive to any signals. You are not required to modify the kernel to allow frozen processes to be responsive to signals.

2.3 Milestone 3 - static scheduling ratio policy (4 marks)

Nevertheless, Dr. Wong thinks that the FOG policy is too strict, and is not a good implementation. He has proposed a new policy named the **static scheduling ratio** policy, or SSR (reverse of RSS ;) for short.

Originally, without both the SSR and the FOG policies, Alice's and Bob's processes receive more or less the same share of the CPU resource. Thus, Alice's process will never be able to catch up Bob's process in terms of the **accumulative CPU time**.

With the SSR policy, the CPU resource distributed to both processes should be different. We define a term called the **scheduling ratio**. This ratio represents the ratio of the CPU resource scheduled to the two processes registered by the "`glue`"system call.

For instance, the original scheduling ratio for Alice's to Bob's process is **1**. Under the SSR policy, this scheduling ratio should be changed to a value larger than 1. E.g., if the scheduling ratio is **2**, then, logically, Alice's process will be scheduled twice before Bob's process can be scheduled once. The merit of this method is to give chances to Bob's process to run, and, at the same time, to let Alice's process to catch up Bob's process.

Note that the `ratio` parameter for the "`glue()`" system call is always a non-negative integer. When `ratio` is 0, it is the FOG policy. Else, it is SSR policy. In addition, the following rules apply to the SSR policy.

1. Inside the scheduler, you have to judge which process (whether the process specified by `pid_1` or the process specified `pid_2`) is the younger process.

2. You have to instruct the scheduler to give more CPU resource to the younger process based on the scheduling ratio provided by the "`glue`" system call.

3. When the **accumulated CPU time** of the younger process is larger than or equal to that of the older process, the new scheduling policy stops functioning automatically and the two PIDs are unregistered from the kernel (as if the "`separate`" system call is invoked).

Invoking the SSR policy.

```
#define NR_glue [some number]
int pid_1, pid_2;
int ratio = 2; // an integer > 0
syscall(NR_glue, pid_1, pid_2, ratio);
```

3 Assumptions and Requirements

First, we have some general requirements for the entire assignment. Then, we state specific requirements for every milestone.

3.1 General assumptions and requirements

1. You are required to work on edhat 9 with kernel version 2.4.37.10. For Redhat 9, you can download it from `ftp://ftp.cuhk.edu.hk/pub/Linux/redhat/redhat-9/iso/i386/`. The vanilla kernel source can be obtained from our course homepage.

2. You are required to handle the case for single CPU only. In other words, your code would not be tested on a SMP machine or a multi-core machine. Note that the system is restricted to the 32-bit architecture.

   By the way, in order to check if you are working on a single-CPU machine, you can check the number of entries returned by "`/proc/cpuinfo`".

3. You must implement both the FOG and the SSR policies in order to get a full mark for this assignment. Also, you must finish this assignment by changing the Linux scheduler, i.e., by kernel hacking.

4. After you have changed the Linux scheduler, other parts of the system should be running normally without significant performance impact. For example, occasionally "freezing" the whole system is not acceptable.

5. You can assume that the processes registered for the new scheduling policies are of `SCHED_OTHER` policy, with the default nice value as well as the default static priority.

6. If your implementation causes any system failures, including the **kernel oops** and **kernel panic**, then you will suffer from a 30%-off penalty of the score you recieved.

7. The processes that are registered to the new scheduling policies can be any programs. Nevertheless, the processes are assumed to be **CPU-bound processes**.

8. The processes that are submitted to the new scheduling policies are **not assumed to be running indefinitely**.

3.2 Assumptions and requirements of Milestone 1

1. You must write a set of user programs, which are ordinary C programs, to invoke the new system calls.

2. The new system calls must return after you have invoked it, i.e., they should not block the calling process.

3. The new scheduling policies can register only two processes. Multiple number of invocations of the "`glue`" system call is allowed. The new scheduling policy monitors only the processes specified by the last invocation of the "`glue`" system call.

3.3 Assumptions and requirements of Milestone 2

1. The FOG policy should be working immediately after the invocation of the "`glue`" system call.
2. The "`separate`" system call should stop the FOG policy immediately.
3. The FOG policy must stop the execution of the older process through the modification of inside the scheduler. Other implementations such as sending a SIGTSTP signals to the older process are not allowed.

3.4 Assumptions and requirements of Milestone 3

1. The SSR policy should be working immediately after the invocation of the "`glue`" system call.
2. The "`separate`" system call should stop the SSR policy immediately.
3. Under the SSR policy, you must be able to show the difference in the CPU resource allocation, and such an allocation difference should be related to the scheduling ratio. The "`top`" command should be a good choice.

4 Submission Guideline

For the submission of the assignment, please send your work to:

`tywong@cse.cuhk.edu.hk`

Remember, you have to submit all the user programs and all the modified kernel codes (but, never the entire kernel source tree).

Deadline: 16:30, Nov 6, 2015 (Fri).

Happy Hacking.