Hands-on Tutorial on

Understanding Linux Scheduling

Basics

There are resources aplenty on the web explaining the different types of scheduling on Linux and their properties. This link This link to the man page of the sched_setscheduler call gives you the information your need. There are three scheduling types supported by Linux. They are...

1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR

The behaviour of these three types are explained in the link above. GNU also adds SCHED_BATCH and SCHED_IDLE scheduling types. We'll stick to the commonly used three. You can find these defines in /usr/include/bits/sched.h.

Linux 3.14 also adds the SCHED_DEADLINE.

Priority ranges

This program prints out the range of priorities supported by the different scheduling types. The output of the program is added as a comment in the above link.

```
/*Print range of priorities on Linux*/
#include<sched.h>
#include<stdio.h>
int main()
{
  printf("Max priority for SCHED_OTHER %d\n",
sched_get_priority_max(SCHED_OTHER));
  printf("Min priority for SCHED_OTHER %d\n",
sched_get_priority_min(SCHED_OTHER));
  printf("Max priority for SCHED_FIFO %d\n",
sched_get_priority_max(SCHED_FIFO));
  printf("Min priority for SCHED_FIFO %d\n",
sched_get_priority_min(SCHED_FIFO));
  printf("Max priority for SCHED_RR %d\n",
sched_get_priority_max(SCHED_RR));
  printf("Min priority for SCHED_RR %d\n",
sched_get_priority_min(SCHED_RR));
```

```
  return 0;
}
Output:
Max priority for SCHED_OTHER 0
Min priority for SCHED_OTHER 0
Max priority for SCHED_FIFO 99
Min priority for SCHED_FIFO 1
Max priority for SCHED_RR 99
Min priority for SCHED_RR 1
```

We see that SCHED_FIFO and SCHED_RR priorities allow priorities from 1 to 99. SCHED_OTHER, which is the default supports only the value of 0. SCHED_OTHER is also referred to as "normal". SCHED_FIFO and SCHED_RR, the realtime scheduling policies use priorities. SCHED_OTHER only has a "niceness" attached to it.

A higher value will always prempt a lower value of priority.

Take a look at the manual pages for the following scheduling related system calls.

- nice() - Change process priority of a normal process (SCHED_OTHER)
- getpriority()/setpriority() - get/set program scheduling priority.
- sched_getscheduler()/sched_setscheduler() - set and get scheduling policy/parameters
- sched_getparam()/sched_setparam() - set and get scheduling parameters
- sched_get_priority_max/min - get static priority range
- sched_rr_get_interval() - get the SCHED_RR interval for the named process

Some of the above APIs also have POSIX equivalents. If you are using pthread APIs, it is recommended that you use the pthread_ variants of these APIs.

To summarize a few key points. We have two categories of scheduling policies. Normal and real-time. Real-time scheduling policy has two sub-types, round-robin and first-in first-out, identified by SCHED_RR and SCHED_FIFO.

Hands On

To move on to the next part of our study, compile the following programs. All these programs are just a while(1) loop to simulate a busy loop. The only difference between them is that they setup different scheduling policies. The priority to use is obtained from the first command line argument.

1. fifo-task.c

```
#include <sched.h>
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
  printf("Setting SCHED_FIFO and priority to %d\n",atoi(argv[1]));
  struct sched_param param;
  param.sched_priority = atoi(argv[1]);
  sched_setscheduler(0, SCHED_FIFO, &param);
  int n = 0;
  while(1) {
    n++;
    if (!(n % 10000000)) {
      printf("%s FIFO Prio %d running (n=%d)\n",argv[2], atoi(argv[1]),n);
    }
  }
}
```

2. rr-task.c

```
#include <sched.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  printf("Setting SCHED_RR and priority to %d\n",atoi(argv[1]));
  struct sched_param param;
  param.sched_priority = atoi(argv[1]);
  sched_setscheduler(0, SCHED_RR, &param);
  int n = 0;
  while(1) {
    n = n + 1;
    if (!(n % 10000000)) {
      printf("Inst:%s RR Prio %d running (n=%d)\n",argv[2], atoi(argv[1]),n);
    }
  }
}
```

3. other-task.c

```
#include <sched.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  printf("Setting SCHED_OTHER and priority to %d\n",atoi(argv[1]));
  struct sched_param param;
  param.sched_priority = atoi(argv[1]);
  sched_setscheduler(0, SCHED_OTHER, &param);
  int n = 0;
  while(1) {
    n = n + 1;
    if (!(n % 10000000)) {
      printf("%s OTHER Prio %d running (n=%d)\n",argv[2], atoi(argv[1]),n);
    }
  }
}
```

We will run these programs in different scenarios and discuss their behaviour.

Scenario 1

Run fifo-task with priority 1 in first terminal. Run fifo-task with priority 1 in the second terminal.

You will observe that the process which starts running first will print continuous output. The second process prints "Setting SCHED_FIFO and priority to 1" and then is silent.

Kill the first process and you will notice that the second process now starts printing output.

Conclusion: In the case of SCHED_FIFO, for tasks of the same priority, the currently running task has to yield before the next one can run. You may ask: why then did the second process print "Setting SCHED_FIFO and priority to 1? We'll soon see why.

Scenario 2

Run rr-task with priority 1 in Terminal 1. Run rr-task with priority 1 in Terminal 2.

You will observer that both tasks will run continuously.

Conclusion: Tasks of the same priority when running with RR_SCHEDULING will get an equal interval run. This is the interval returned by sched_rr_get_interval().

Scenario 3

Run rr-task with priority 1 in Terminal 1. Run rr-task with priority 2 in Terminal 2.

You will observe that as long as the second instance with priority 2 is running, the first instance with priority 1 will not run.

Conclusion: Nothing surprising here. This behaviour is common to any real-time scheduled process, whether it is RR or FIFO. A higher priority will always have priority over the lower priority unless it explicitly yields.

The differences in SCHED_RR and SCHED_FIFO are only apparent when tasks have the same priority. When the tasks have different priorities, the higher priority always wins.

You can repeat the experiment with different combinations of rr-task and fifo-task.

Scenario 4

Run other-task in Terminal 1. Run fifo-task with priority 99 in Terminal 2.

You will observe that both processes run ! A real-time process always has a higher priority than a normal process. The fifo-task with priority 99 is supposed to be the highest priority on the system. Why does the SCHED_OTHER task still get to run?

Conclusion: Something unexpected is happening. Now is the best time to explain the **Linux Real-Time group Scheduling**.

Real-time group Scheduling

As you saw in scenario 4 above, even through you tried to run a real-time task with the highest possible priority, a task with SCHED_OTHER was still able to run, giving the impression that a SCHED_OTHER task has a higher priority.

If you understand the mechanism behind this behaviour, you realize it is not unexpected. Linux is originally meant for a desktop or server machine. If a user unknowingly launches a task with the highest possible priority, and this task spins in a loop like the example programs we wrote, then there is no way to recover the system without a reset.

To prevent this from happening, the Linux kernel places a global limit on how much time realtime scheduling may use. The remaining time is used to schedule SCHED_OTHER processess using the CFS.

There are two important files in the /proc filesystem which are used to configure the settings of this limit. The first file is */proc/sys/kernel/sched_rt_period_us*

On my debian machine, this has the value 1000000, which means 1 second.

The second file is */proc/sys/kernel/sched_rt_runtime_us* .

This has the value 950000 on my machine. So by default, in 1 second, 0.95 seconds can be used by the real-time processes. 0.05 second is reserved for the SCHED_OTHER tasks. This explains some things you may or may not have noticed when running the examples before.

1. Even though a real-time process is running as the highest possible priority on the system, the shell is still usable and you are able to start the second process.
2. In scenario 1, the task in terminal 2 runs until it sets the scheduler to SCHED_FIFO. Until then, it is able to run as it is a SCHED_OTHER process. The moment it becomes a real-time process, it has to compete with the other fifo process which is already running.

You can change the default values of the sched_rt_runtime_us and sched_rt_period_us by echoing a new value into it as root. Be aware though that if you set the sched_rt_runtime_us to the same as the sched_rt_period_us, then a real-time process can hog the CPU.

If CONFIG_RT_GROUP_SCHED is enabled in the Kernel build, you can partition the real-time budget into groups, using cgroups. A real-time group is a group of one or more tasks. In this

case, you need to use the cgroup CPU subsystem to partition the groups and allocate the runtimes.

If CONFIG_RT_GROUP_SCHED is disabled, the sched_rt_runtime_us will limit time reserved to all realtime tasks.

If CONFIG_RT_GROUP_SCHED is enabled, then sched_rt_runtime_us signifies the total bandwidth available to all realtime groups.

Read the documentation at https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt for further information. This document also has early ideas about Deadline First Scheduling.

Conclusion

It is important to understand how different scheduling priorities affect the way in which your program works. While it may not be important in a desktop environment, these behavioral differences become critical in an embedded Linux environment. Your system may have to cope with real-time audio and network requirements, while also being responsive to user interaction.

Creating a Real time Process

You can designate a process as real time by setting a process attribute that the scheduler uses as part of its scheduling algorithms.

Program to create a real time process:

```
#include <sched.h>
#define MY_RT_PRIORITY MAX_USER_RT_PRIO /*Highest possible*/
int main(int argc, char **argv)
{
  int rc, old_scheduler_policy;
  struct sched_param my_params;

  /*Passing zero specifies caller's(our) policy*/
  old_scheduler_policy = sched_getscheduler(0);
  my_params.sched_priority = MY_RT_PRIORITY;

  /*Passing zero specifies caller's(our) policy*/
  rc = sched_setscheduler(0,SCHED_RR,&my_params);
  if (rc == -1)
    printf("error");
}
```

This code snippet does two things in the call to sched_scheduler(). It changes the scheduling policy to SCHED_RR and raises its priority to maximum possible on the system. Linux supports three scheduling policies:

- SCHED_OTHER – Normal Linux process, fairness scheduling.

- SCHED_RR – Real-time process with a time slice. In other word, if it does not block, it is allowed to run for a given period of time determined by the scheduler.
- SCHED_FIFO – Real-time process that runs until it either blocks or explicitly yields the processor or until another higher-priority SCHED_FIFO becomes runnable.

The man page for sched_setscheduler() provides more detail on three different scheduling policies.

Practice Yourself:

Now you free your hands using the above discussed scheduling based coding. The **assignment problem** on this topic will be uploaded soon.