

TD-TP: Threads Scheduling on Linux Operating Systems

Abstract - The future of computing is definitely *multi-core*. Multi-core and *multi-threading*. Using multi-threads significantly decrease execution time of a application , but also make it more difficult to manage a process, especially in multi-cores, raising the need for better schedulers. How does Linux operating systems cope with it? How are threads handled? What types of scheduling algorithms are implemented in the schedulers? Which one is faster? By performing a simple experiment in this paper we describe Linux from a different of view , scheduling.

Keywords - *multithreading, FIFO, Round-Robin, real-time scheduling, experimental evaluation.*

I. Introduction

Single-core processor performance has reached the power limit. Currently there are two methods used to achieve better performance, less CPU time and less memory usage:

1) Using several cores

as a single processor (multi-core), using them to process tasks simultaneously.

2) Using threads to split the work of a process into several small tasks that are performed simultaneously, thus virtually creating parallelism. One of the most important tasks performed by an operating system is to allocate ready processes or threads to the available and less busy processors/cores. This task can be divided into two parts. The first part, known as *process scheduling*, consists of decision-making policies to determine in which order should active tasks compete for the use of processors. Different from

windows and other OS, Linux does not use a special separate module to schedule threads. It considers both of them as *tasks* and use the same schedulers. The actual binding of a selected task to a processor, which consists in removing the tasks from the queue, change its status, and load the processor state, is performed by a *process dispatcher*. This two parts are

usually done by two components of the operating system. These components can be part of the kernel, but they could also be part of higher operating system levels and applications code. In more complex designs, several elements of the system may even have their own schedulers so that task scheduling can be distributed throughout. Unless explicitly stated, we are going to refer to both modules jointly as the scheduler and to both tasks as scheduling. Like nearly all other OS schedulers Linux scheduler use the concept of task priority when it needs to decide which should be running next. It can be *static* which means that once the priority of a task is decided ,it cannot be changed or *dynamic*, the priority can change every time the scheduler is involved. Each task has its priority level.

Higher priority means that the task, thread or process, will be performed earlier than tasks with lower priority. For the experiment I am using Ubuntu 12.04 which has three standard schedulers :

1. The *First-In/First-Out*, in Ubuntu called SCHED_FIFO, can only be used with static priorities higher than 0, which means that when a SCHED_FIFO thread becomes runnable, it is going to preempt any currently running SCHED_OTHER which has always priority zero. SCHED_FIFO is a scheduling algorithm without time division. If SCHED_FIFO task has been preempted by another task of higher priority, it will stay on top of the list for its priority and will continue execution as soon as all tasks of higher priority are blocked again. When it becomes runnable, it will be put at the bottom of the list for its priority.

2. *Round-Robin*, in Ubuntu is called SCHED_RR It is a simple improvement of SCHED_FIFO. Everything said for FIFO can also said to SCHED_RR, except that each task is allowed to run for a maximum time quantum. If a SCHED_RR task has been running for a time period longer than the established time , it will be inserted at the bottom of the list for its priority. A

SCHED_RR task that was preempted by a higher priority task and then resumes its execution, will complete the portion of its round robin time quantum left.

3. SCHED_OTHER, can only be used at static priority 0. SCHED_OTHER is the standard Linux time-sharing scheduler that is used for all tasks that do not require any sort of static real-time priority. The task to execute is chosen from the priority list based on a dynamic priority that is given only inside this list. The dynamic priority is based on the nice level and increased for every time-quantum the task is ready to be executed, but refused by the scheduler. This ensures fairness among all SCHED_OTHER tasks.

II. Related works

There are several articles with a particular interest that gave us the idea to write this study.

Earlier works on the the field include article[1] about thread scheduling in real time Linux, which is what we experimented with fifo and round robin. Since for testing schedulers we created threads on quadcore pc, of great help were article [2] and article [3], where on the former is explained how linux kernel handles threads on a multicore computer. It also explains how fairness is achieved, how to prevent starvation and how to achieve highest efficiency on linux operating system. On article [3] it is explained and given examples how pthreads can be created, set priorities and policies. Article [5] explains how multithreading can be programmed to work efficiently on a Linux environment.

Article [4] attempts to do an experimental comparison of different real-time schedulers on multicore systems where a similar experiment to ours is performed, comparing the performance of two schedulers EDF and Rate Monotonic.

Similar to [4] but more technical, article[6] describes EDF scheduling class for the Linux kernel, implements an EDF scheduler and analyzing time of performance. On study [7] a comparison of scheduling algorithms for multiprocessors like: partition schedulers vs global schedulers and dynamic vs static schedulers are discussed. The schedulers implemented are PF, LLREF, global EDF, static RTSIM.

Two articles[8] and [9] are really helpfull for understanding how linux handles scheduling jobs. The first one[8] takes varius Linux schedulers like: sched_FIFO, sched_RR, sched_other, sched_batch and explaines them theoretically. The other [9] gives a full view on how are threads really handled on linux, like explaining thread dispatcher, thread scheduler.

Among others studies worth mentioning is the article[10] which explains several scheduling algorithms for multiprogramming in a hard real-time environment. All this studies gave us a better understanding on multithreading and a more clear view on schedulers, making us more prepared for our study.

III. Theory of experiment

In this section we will describe schedulers in a more scientific way and attempt to analyze them.

3.1 Environment

By the term environment we mean operating system on which we executed our code. We used on Ubuntu OS 12.04.

3.2 Programming languages

To experimentally compare Ubuntu's schedulers, we created two tasks and executed them on different scheduling order. The tasks were threads created using POTIX pthreads, implemented in C language which is good because in same cases JVM, the platform where Java runs, is independent from OS. It could interfere with the *time of execution* measurements.

3.3 Necessary conditions

To correctly measure the time of the threads performing on a scheduling we need several conditions:

1. Threads need to have both of them set on either FIFO or OTHER or Round Robin policies.
2. For threads to be executed on same real-time FIFO or Round-Robin scheduler stack, the threads need to have the same priority level.
3. On real time scheduling the priority is from 0 to 99. Higher priority means that the probability of being preempt will be smaller. We used high priorities so our threads would not be preempted.

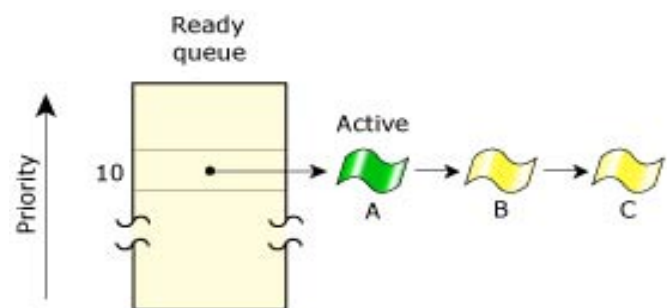


Fig. 1: Threads of same priority using FIFO

IV. Experimental phase

In the next section of the article a quick view of the experimental phase is presented. As discussed before we created two threads that perform two different task, with two different time of execution and schedule them to be performed on three Ubuntu schedulers. We measured their order and time of execution.

4.1 Algorithm of the experiment

First thread is assigned to do this job:

After setting the priority and policy, it enters loop from 0 to 100 and prints the incremented thread->value.

Second thread is assigned to do this job :

After setting the priority and policy, it enters two loops from 100 to 0 and does four arithmetic operations. Then it enter a loop again from 0 to 100 where it prints the incremented thread value.

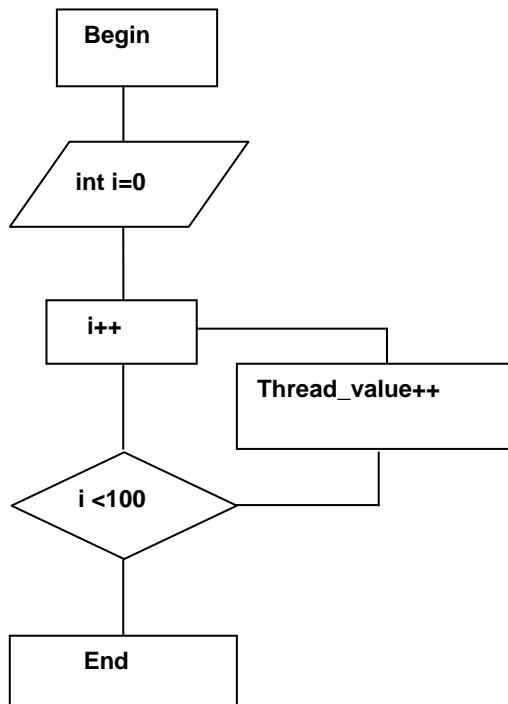


Fig. 2: Thread 1 algorithm

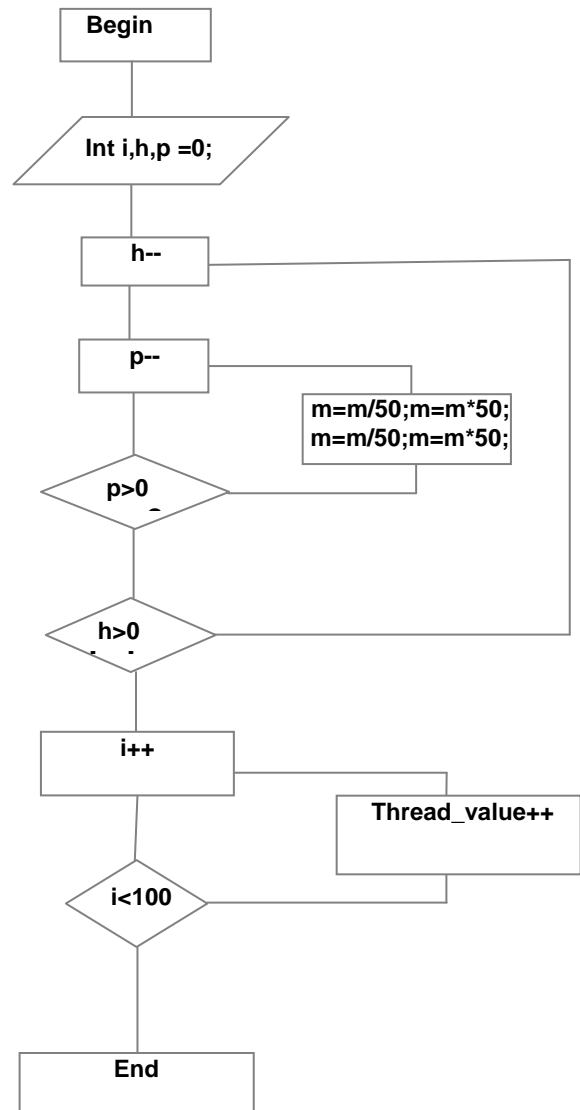


Fig. 3 : Thread 2 algorithm

Both threads are included in a single process which creates ,call them and wait until both of them are done to exit.

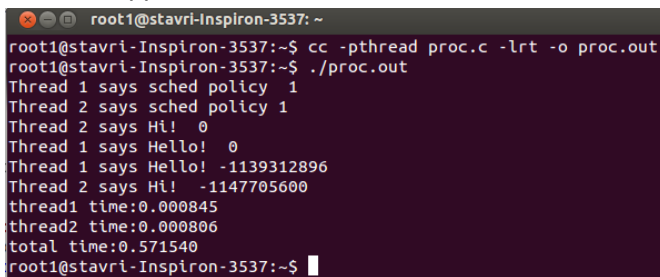
We also included the library <time.h> to be able to call the function `clock_gettime()` so we can measure their time of execution. This function first argument is `CLOCK_MONOTONIC`, it directly counts the internal clock pulses and gives an extremely accurate time count.

4.2 Experimental environment

By term surroundings we refer to:

- 1.Enabling all cores of the processor
 - 2.Installing a C language compiler, gcc
- Both tasks are done as we checked them. To execute the program the following commands are needed :

- 1.Open terminal
 2. `cc -pthread proc.c -lrt -o proc.out`
 3. `./proc.out` and press enter
- This will appear on the screen:



```
root1@stavri-Inspiron-3537: ~  
root1@stavri-Inspiron-3537:~$ cc -pthread proc.c -lrt -o proc.out  
root1@stavri-Inspiron-3537:~$ ./proc.out  
Thread 1 says sched policy 1  
Thread 2 says sched policy 1  
Thread 2 says Hi! 0  
Thread 1 says Hello! 0  
Thread 1 says Hello! -1139312896  
Thread 2 says Hi! -1147705600  
thread1 time:0.000845  
thread2 time:0.000806  
total time:0.571540  
root1@stavri-Inspiron-3537:~$
```

Fig. 4 : Print Screen of thread exec.

4.3 Results of the experiments

By changing the policy from `sched_FIFO` to `sched_RR` and to `sched_OTHER`, we noticed three different time and order of execution.

If both threads have a `fifo` policy, the first thread that goes to the queue get executed first, which always is thread1.

We put a priority 70 which is high (99 is the maximum) and observed that thread1 finished first and thread2 finished second. Thread1 finished for 0.003550s and thread2 for 0.003719s. Although the results are very close,they clearly demonstrated that thread1 and thread2 were put to the same queue, because of their same priority, but thread1 arrived first to the queue and was executed first for less time.

If both threads have round-robin policy , same as `fifo` but we saw that the first thread was executed for only 70 of the “printf” instructions, then started thread2 for twenty “printf” instructions then again to thread1 and so on until both of them finished. The time of the execution were respectively 0.001596

and 0.001590.This suggest that both of them were performed with equal turns for the same amount of time.

If both of threads have a other policy then the priority is zero. We observed that it behaved very similar to Round-Robin but with smaller turns. This suggest that since our program was the only one opened the weren't other processes to interfere and the niceness didn't increase so basically it acted as a round-robin. When we performed it a second time, the time of thread1 was 0.001693 and of the second thread was 0.004164 bigger difference than `fifo`.

Total time of the process was always around 0.574675 and 0.575761.

V. Conclusion

On section 4 we described a experiment and we made several observations. As a conclusion:

1.Ubuntu 12.04 has three standard schedulers:`sched_FIFO`, `sched_RR` and `sched_OTHER`. `Sched_fifo` executed the first task was put to the queue and after it finished ,executed the second.

2.`Sched_RR` executed both threads with turns for equal timeslices and both of them were finished for the same amount of time.

3.`Sched_OTHER` has priority 0 and is a dynamic scheduler. It acted as a round-robin but it could also have a difference of time between the threads worse than `fifo`.