Ben Dumais   2250669195   cs3305-Asn2 Writeup

1 – Nice Values Part 1

Running cpuTimeWaste.c with nice values:

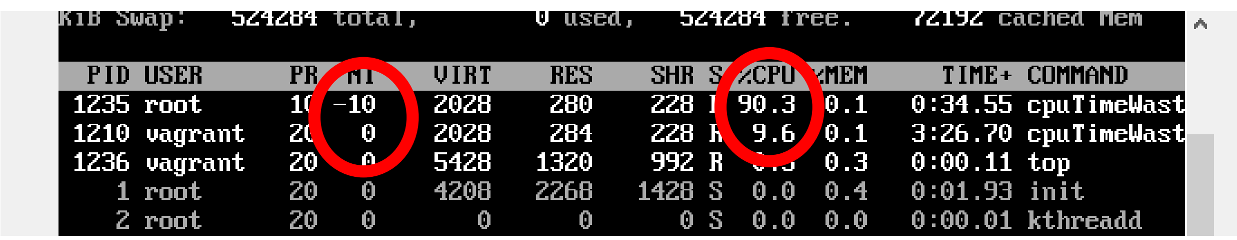
Nice Value	CPU Usage
-10	98.9
-5	99.5
0	99.6
5	98.6
10	98.6
15	98.6
19	98.6

In theory, changing the nice values should change the amount of cpu that a process receives in comparison to other processes (higher nice values would typically give it a lower share of CPU time). However, since only one instance was running on the machine and there were no other processes running, cpuTimeWaste received full share of the CPU, since there were no other processes with which to share.

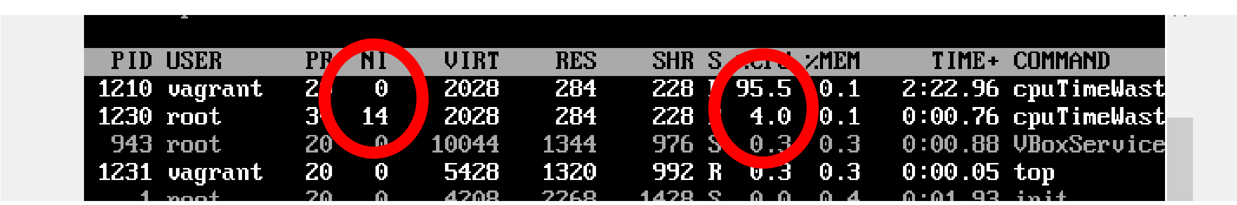
If another process was running at the same time as cpuTimeWaste, nice values would change the cpu share. At default value of 0 both processes would receive around 50% share, then as the nice value increases, the cpu share would lower.

2 - Nice Values Part 2

Nice Value to have default process receive 10% share: -10



Nice Value to have default process receive 95% share: 14



3 - Compare System Call and Function

Associated code for question can be found in compare.c, which uses the time comparison code posted as an example for assignment 2 to measure the time it takes to execute the system call and the minimal function. Minimal function was implemented as a function that takes no parameters and simply returns.

```
  GNU nano 2.2.6                File: question3

Script started on Wed 02 Mar 2016 09:24:18 PM EST
[bdumais@lxg-001 Asn2]$ compareScript
System: 5214
Function: 676
[bdumais@lxg-001 Asn2]$ exit
exit

Script done on Wed 02 Mar 2016 09:24:25 PM EST
```

As seen above, the system call takes much longer than the minimal function.

4 - Kernel Information

The code for question four can be found in kernelInfo.c, which reads the cpuinfo version files in /proc and parses them for the needed info. The output, running on the lxg-001 machine under GAUL is as follows:

```
  GNU nano 2.2.6                File: question4

Script started on Wed 02 Mar 2016 10:52:03 PM EST
[bdumais@lxg-001 Asn2]$ kern
CPU type:        Intel(R) Core(TM) i5-4570S CPU @ 2.90GHz
Kernel version: Linux version 3.13.0-74-generic (buildd@lcy01-07)
[bdumais@lxg-001 Asn2]$ exit
exit

Script done on Wed 02 Mar 2016 10:52:16 PM EST
```

5 – Process Monitoring

Code can be found in observer.c, which takes name of process to execute as input. Program will output the utime and stime of a process (time spent in user mode and time spent in kernel) each second.

6 – Impact of Scheduling Policies, Part 1

Code can be found in comparison1.c, which loops until signal is received. Program sleeps for one second then starts instance of cpuTimeWaste with the input scheduling policy (ex "comparison1 SCHED_RR").

Observations:

Using the default SCHED_OTHER policy, the cpu usage for each instance of cpuTimeWaste has roughly the same cpu usage. In this policy, interactive programs such as the command line were usable with little to no lag.

With SCHED_FIFO, the command line became very unresponsive, often taking several seconds for input to show on screen and even longer for a command to execute. From this policy, it is clear that the multiple instances of cpuTimeWaste were using up much more cpu, making it more difficult for the command line to get its share as it would have to wait for each cpuTimeWaste to complete before getting a turn.

Using SCHED_RR, the command line became rather unresponsive as well, however not to the extent of FIFO. There was considerable lag between execution of code and even displaying input to command line.

Overall, the default, SCHED_OTHER, balances cpu share across processes, prioritizing interactive processes. A FIFO or RR policy can hurt interactive or real-time processes, as it takes longer for them to get share of the cpu, in turn making them less responsive and slower.

7 – Impact of Scheduling Policies, Part 2

Code can be found in comparison2.c, which loops until signal is received. Program sleeps for one second then starts instance of IOBound with the input scheduling policy (ex "comparison2 SCHED_RR").

Observations:

As with before, interactive processes are still very usable and responsive using the SCHED_OTHER policy.

SCHED_FIFO policy again greatly slows down the responsiveness of the command line and other programs. Running *top*, even though each instance of IOBound is using little CPU, it is still harder for interactive processes to get adequate cpu time.

The above is also true for SCHED_RR, where interactive processes become difficult to use and are slow to respond to keyboard input or execute

As before, different policies can greatly affect the responsiveness of realtime or interactive processes. The impact however, is noticeably less than a CPU intensive program such as cpuTimeWaste. This is likely due to the IO nature of this experiment, where the IOBound spends lots more time writing and not using the cpu, which gives interactive processes more frequent time in the CPU.