Optimizing The Linux Scheduler For Performance.

Constanza Madrigal Reyes and Ismael Lizárraga González

May 5, 2017

Abstract

The CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. The scheduler controls the way processes are managed in the operating system. In this project we analyze the behavior of the Linux kernel by changing the kernel values that manage the scheduling process. We plan to analyze and evaluate the impact that modifying the kernel values has on performance. To implement this analysis, we started by modifying kernel values that after analysis, we thought would impact on a sample pi program test by Phoronix Test Suite. Then, after running some tests and using a genetic algorithm, we plan to determine which kernel values have a more significant impact on the pi calculation performance.

To measure performance, we use the result of our Phoronix Tests, that is the time in seconds that the benchmark takes to calculate 8,765,4321 digits of pi using the Leibniz formula. Performing this calculation involves a special case of a general series expansion for the inverse tangent function.

1 Introduction

Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux supports preemptive multitasking, this means that the process scheduler decides which process runs and when.

Balance performance across different computer configurations is one challenge in modern operating systems.Linux has two separate process-scheduling algorithms. One is a time sharing algorithm for fair, preemptive scheduling among multiple processes. The other is designed for real-time tasks, where absolute priorities are more important than fairness.

If a Linux system performs similar tasks in a regular manner, it could be useful to implement optimizations to the Linux scheduler to optimize the performance of those tasks. In this project, we are going to analyze and evaluate the impact of changing the kernel values on the performance of the calculation of 8,765,4321 digits of pi using the Leibniz formula measuring the time that the system takes to perform the calculation.

2 Theoretical Framework

2.1 Process Scheduler

The Linux kernel is responsible for controlling the way that processes are managed on the system. The process scheduler decides which task to run next. It is responsible for best using system resources to guarantee that multiple tasks are being executed simultaneously. This makes it a core component of any multitasking operating system.

2.2 Basics concepts related to scheduling

Preemptive multitasking: where the scheduler decides when a process is suspended. This forced suspension is called preemption. UNIX systems have been providing preemptive multitasking since the beginning.
Timeslice: the time period for which a process will be running before it is preempted is defined in advanced. Represents the amount of processor time that is provided to each process.

Process Priority: Processes are evaluated by the scheduler according to their priority. Each process is given a value to which it is "allowed" to run on a processor.
Latency: Delay between the time a process is scheduled to run and the actual process execution.
Granularity: The relation between granularity and latency can be expressed by $(lat/rtasks) - (lat/rtasks/rtasks)$ where lat stands for latency and $rtasks$ is the number of running tasks.

2.3 Scheduling Policies

The Linux kernel supports the following scheduling policies:
$SCHEDSamp\_FIFO$: Scheduling policy designed for special time-critical applications. It uses the First In-First Out scheduling algorithm.
$SCHED\_BATCH$: Scheduling policy designed for CPU-intensive tasks.
$SCHED\_IDLE$: Scheduling policy intended for very low prioritized tasks.
$SCHED\_OTHER$: Default Linux time-sharing scheduling policy used by the majority of processes.
$SCHED\_RR$: Similar to $SCHED\_FIFO$, but uses the Round Robin scheduling algorithm.

2.4 The sysctl Interface

This interface is used for examining and changing kernel parameters at runtime. With this interface you can change the default behavior of the task scheduler by the access to variables that this interface provides.
The values that we modify during this project are:

- $kernel.sched\_latency\_ns$
- $kernel.sched\_migration\_cost\_ns$
- $kernel.sched\_min\_granularity\_ns$
- $kernel.sched\_nr\_migrate$
- $kernel.sched\_rr\_timeslice_ms$
- $kernel.sched\_rt\_period_us$
- $kernel.sched\_rt\_runtime_us$
- $kernel.sched\_schedstats$
- $kernel.sched\_shares_window_ns$
- $kernel.sched\_time_avg_ms$
- $kernel.sched\_tunable_scaling$
- $kernel.sched\_wakeup_granularity_ns$

2.5 Calculating Pi Using the Leibniz Formula

As we mentioned earlier, our performance is measure according to the Sample Pi Program benchmark provided by Phoronix Test Suite. This test runs a simple C++ program that calculates 8,765,4321 digits of pi using the Leibniz formula. This formula states that:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ... = \frac{\pi}{4}.$$

This series is a special case of a more general series expansion for the inverse tangent function.

3 Genetic Algorithm

Genetic Algorithms (GA's) are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomised, GA's are by no means random, instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the GA's are designed to simulate processes in natural systems necessary for evolution, specially those follow the principles first laid down by Charles Darwin of "survival of the fittest.". Since in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

Were invented to mimic some of the processes observed in natural evolution. Many people, biologists included, are astonished that life at the level of complexity that we observe could have evolved in the relatively short time suggested by the fossil record. The idea with GA is to use this power of evolution to solve optimization problems. The father of the original Genetic Algorithm was John Holland who invented it in the early 1970's.

GA's simulate the survival of the fittest among individuals over consecutive generation for solving a problem. Each generation consists of a population of character strings that are analogous to the chromosome that we see in our DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution. They are based on an analogy with the genetic structure and behavior of chromosomes within a population of individuals using the following foundations:

- Individuals in a population compete for resources and mates.
- Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.
- Genes from 'good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.
- Thus each successive generation will become more suited to their environment.

3.1 Implementation Details

After an initial population is randomly generated, the algorithm evolves the through three operators:

1. **selection** which equates to survival of the fittest;
2. **crossover** which represents mating between individuals;
3. **mutation** which introduces random modifications.

3.1.1 Selection Operator

The key idea if to give preference to better individuals, allowing them to pass on their genes to the next generation. The "goodness" of each individual depends on its fitness which may be determined by an objective function/ fitness function or by a subjective judgment.

3.1.2 Crossover Operator

It is the prime distinguished factor of GA from other optimization techniques, two individuals are chosen from the population using the selection operator. A crossover site along the bit strings is randomly chosen and the values of the two strings are exchanged up to this point: If S1=000000 and s2=111111 and the crossover point is 2 then S1'=110000 and s2'=001111. The two new offspring created from this mating are put into the next generation of the population and by recombining portions of good individuals, this process is likely to create even better individuals.

3.1.3 Mutation Operator

With some low probability, a portion of the new individuals will have some of their bits flipped. Its purpose is to maintain diversity within the population and inhibit premature convergence. The mutation alone induces a random walk through the search space. Mutation and selection (without crossover) create a parallel, noise-tolerant, hill-climbing algorithms.

3.1.4 Effects of Genetic Operators

- Using selection alone will tend to fill the population with copies of the best individual from the population.
- Using selection and crossover operators will tend to cause the algorithms to converge on a good but sub-optimal solution
- Using mutation alone induces a random walk through the search space. Using selection and mutation creates a parallel, noise-tolerant, hill climbing algorithm

The Algorithm

1. Randomly initialize population(t)
2. Determine fitness of population(t)
3. Repeat
   (a) Select parents from population(t)
   (b) Perform crossover on parents creating population(t+1)
   (c) Perform mutation of population(t+1)
   (d) Determine fitness of population(t+1)
4. Until best individual is good enough

4 Objetives

- Learn about Linux kernel scheduling and ways to optimize it for performance according to changes performed in kernel values.

5 Justification

Out of the box, operating systems are optimized for the average consumer. Taking this into consideration, we can say the the Linux kernel scheduler is optimized for best general performance. Nevertheless, for some users there may be circumstances at which a better performance for an specific task may be wanted at the price of reducing performance of not related tasks.

One way to optimize performance for an specific set of tasks, is changing the way the Linux kernel scheduler works, to assure that the tasks we want to perform will be carried out in an optimal way.

We want to make an analysis of the changes in the Linux kernel scheduler that could be carried out to optimize the pi calculation. Through this, we also expect to understand more the way schedulers work and the impact of changing kernel values in performance.

6 Development

At first, we to run our tests we decided that the kernel values that we were going to change were the ones related to latency, granularity, timeslice and memory swapping. After validating initial values and the range of each value, we assigned a different set of values to each test.

To run different tests, we created a bash script that receives the kernel values as parameters, changes the kernel values and runs the Sample Pi Program Benchmark on Phoronix. Everytime we ran a test, the result Phoronix runs the sample pi program at least three times and it provides us with the time in seconds that takes to perform the calculation. This test is ran at least three times and the result it show us is the average value of those three tests, if the performance of the tests has a standard deviation bigger than 5% it runs another tests until it gets a value more closer to the one on the other tests.

After running an initial set of tests, we noticed that there was a change in performance between the different tests, the difference between each of them was in the tenths of second. We think at first that the improvement was insignificant but for servers or computers that are meant to perform a lot of operations through days, this modifications could mean difference of hours of processing and this difference of hours could have a big impact on cost for corporations.

6.1 Genetic Algorithm Implementation

To initialize the optimization of the problem through a genetic algorithm we start by creating a population of size N, each with randomly generate phenotype. Since there is a range of values appropriated for the kernel variables we were changing, in the code each variable is randomly initialized with a random integer generated between its respective accurate ranger.

Then for the selection operation, we need to evaluate the fitness of each element of the population. In this case, the fitness function will be the result of the benchmark obtained with the Phoronix Test.

Follows the reproduction. Here we pick two parents with probability according to relative fitness using a type of Monte Carlo method, the rejection sampling. Then we do a crossover, creating a child by combining the genotype of these two parents and we add him to a new population that will replace the old generation once is full.

This process is continuously repeated until the target result is reached. Also if we do not have an specific target we could just specify a number or generation. According to what you need you may change your stop criteria and the way is calculated.

Source code

```java
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.concurrent.ThreadLocalRandom;


public class Population {
  private DNA population[];
  private int generation;
  private double target;
  private DNA maxFitness;
  private double defaultFitnessValue;

  public Population(double target, double defaultFitnessValue, int size){
    this.generation=1;
    this.maxFitness=null;
    this.target = target;
    this.defaultFitnessValue=defaultFitnessValue;
    this.population = new DNA[size];
    for (int i = 0; i < size; i++) {
        this.population[i] = new DNA((i+1)+(this.generation*size));
    }
    this.calculateFitness();
  }

  public int getGeneration(){
    return this.generation;
  }

  public DNA getMaxFitness(){
    return this.maxFitness;
  }

  public void calculateFitness(){
    BufferedWriter bw = null;
    FileWriter fw = null;
    try {
      File file = new File("results.txt");
      if (!file.exists()) {
        file.createNewFile();
      }
      fw = new FileWriter(file.getAbsoluteFile(), true);
      bw = new BufferedWriter(fw);
      bw.write("Generation: "+this.generation);
      bw.newLine();
    for (int i=0; i<this.population.length;i++){
      System.out.println("Generation: "+this.generation+" DNA number: "+(i
          +1));
      this.population[i].calculateFitness();
      bw.write(this.population[i].toString());
      bw.newLine();
      if(this.maxFitness!=null){
        if(this.population[i].getFitness()<this.maxFitness.getFitness()){
          this.maxFitness=this.population[i];
        }
      }
      else{
        this.maxFitness=this.population[i];
      }
    }
    bw.write("At generation "+this.getGeneration()+"the best is "+this.
        getMaxFitness()+"");
    bw.newLine();
    System.out.println("\nAt generation "+this.getGeneration()+"the best is
         "+this.getMaxFitness()+"\n");
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (bw != null)
          bw.close();
        if (fw != null)
          fw.close();
      } catch (IOException ex) {
        ex.printStackTrace();

      }

    }
  }

  public boolean finished(){
    return this.maxFitness.getFitness()<=target;
  }

  public int randInt(int min, int max) {
      int randomNum = ThreadLocalRandom.current().nextInt(min, max + 1);
      return randomNum;
  }

  public double relativeFitness(double fitness){
    return((this.defaultFitnessValue-fitness)/Math.abs(this.target-this.
        defaultFitnessValue));
  }

  public DNA acceptReject(){
    int randomIndex = this.randInt(0, this.population.length-1);
    double dnaFitness = this.relativeFitness(this.population[randomIndex].
        getFitness());
    if(randInt(0, 10) < dnaFitness*10){
      return this.population[randomIndex];
    }
    else{
      return this.acceptReject();
    }
  }

  public DNA getBestOfGeneration(){
    return this.maxFitness;
  }

  public void newGeneration(){
    DNA newPopulation[] = new DNA[this.population.length];
    this.generation++;
    for(int i=0; i<this.population.length; i++){
      DNA parent1 = this.acceptReject();
      DNA parent2 = this.acceptReject();
      DNA child = parent1.crossover(parent2);
      child.setTestName((i+1)+(this.generation*this.population.length));
      newPopulation[i]=child;
    }
    this.population = newPopulation;
    this.calculateFitness();
  }
}

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;

public class DNA {

  private double fitness;
  private int genes[];
  private String testName;
  //granularity, latency, swapiness and timeslice

  public DNA(int number){
    this.genes = new int[4];
    this.genes[0] = this.randInt(200000, 10000000);
    this.genes[1] = this.randInt(100000, this.genes[0]/2);
    this.genes[2] = this.randInt(0, 100000);
    this.genes[3] = this.randInt(10, 60);
    this.fitness = 0;
    this.testName = "pts"+number;
  }

  public DNA(int[] dna){
    this.genes = dna;
  }

  public void setTestName(int number){
    this.testName = "pts"+number;
  }

  public double getFitness(){
    return this.fitness;
  }

  public void calculateFitness(){
    double newFitness=0;
    try {
      this.executeScript();
      newFitness = Double.parseDouble(this.readFile("/home/conzmr/"+this.
          testName+".csv"));
    } catch (Exception e) {
      System.out.println("Cannot run fitness function. ");
      e.printStackTrace();
    }
    this.fitness = newFitness;
    System.out.println("Result yay"+ newFitness);
  }

  public int randInt(int min, int max) {
      Random rand = new Random();
      int randomNum = rand.nextInt((max - min) + 1) + min;
      return randomNum;
  }

    public DNA crossover(DNA parent){
      DNA child;
      int midpoint = randInt(0, 3);
      if(midpoint==0){
        child = this;
      }
      else if(midpoint == 3){
        child = parent;
      }
      else{
        int newGenes[] = new int[4];
        for(int i=0; i<midpoint; i++){
          newGenes[i]=this.genes[i];
        }
        for(int i=midpoint; i<newGenes.length; i++){
          newGenes[i]=parent.genes[i];
        }
        child = new DNA(newGenes);
      }
        return child;
     }

    public String getGenesString(){
      StringBuilder sb = new StringBuilder();
```

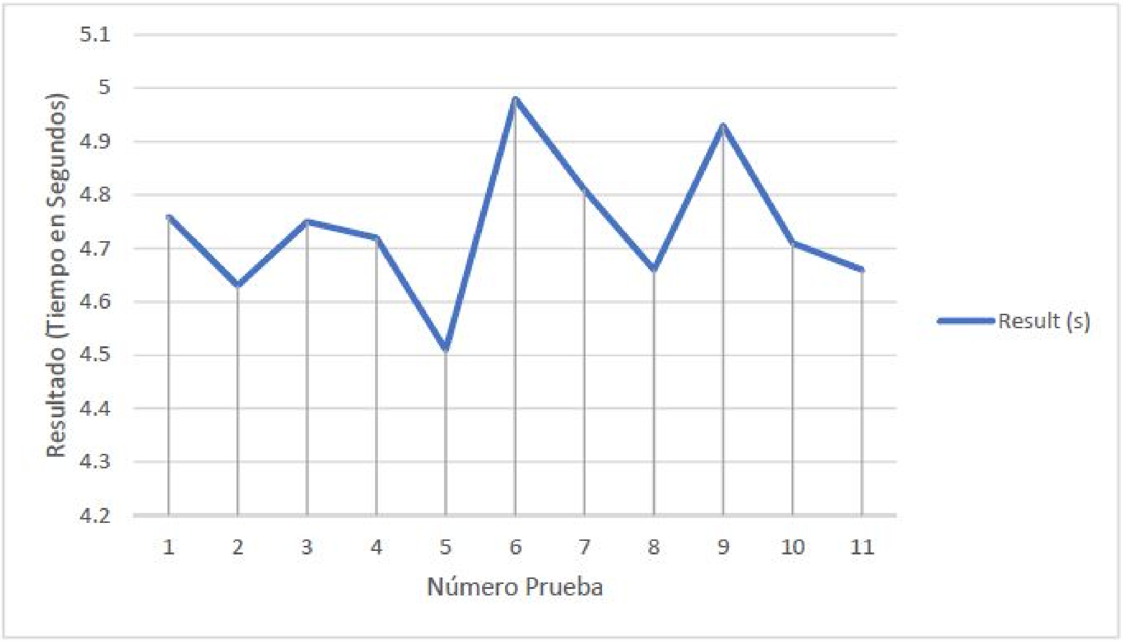
```java
    for (int i=0; i<this.genes.length; i++){
      sb.append(this.genes[i]+" ");
    }
    return sb.toString();
  }

  public String[] getGenes(){
    String[] genesArray = new String[this.genes.length];
    for (int i=0; i<this.genes.length; i++){
      genesArray[i]=String.valueOf(this.genes[i]);
    }
    return genesArray;
  }

  public String toString(){
    return "\nGenotype: "+this.getGenesString()+ "\nPhenotype: "+this.fitness;
  }

  public void executeScript() {
    try {
      ProcessBuilder pb = new ProcessBuilder("/home/conzmr/Documents/4th Semester ISC/Operating Systems/KernelOptimization/src/expect_script.sh",
        this.testName, this.getGenes().toString());
      pb.inheritIO();
      Process p = pb.start();
      int errCode = p.waitFor();
      System.out.println("Command executed with " + (errCode == 0 ? "no errors." : "errors."));
      System.out.println("Script executed successfully");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public String readFile(String fileRoute) {
        String csvFile = fileRoute;
        BufferedReader br = null;
        String line = "";
        try {

            br = new BufferedReader(new FileReader(csvFile));
            while ((line = br.readLine()) != null) {
                if (line.startsWith("\"Sample Pi Program - Phoronix Test Suite v5.2.1")){
                  System.out.println(line);
                  return line.substring(line.length()-4);
                }
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
      return "0.0";
      }
}


import java.io.IOException;

public class MainClass {

  public static void main(String[] args) throws InterruptedException,
      IOException {
    Population population = new Population(4.5, 5, 10);
    while(!population.finished()){
      population.newGeneration();
    }
  }
}
```

Listing 1: Código fonte em Java

7 Results

In this section we show and discuss the results that we obtained during our set of tests. Here is a table showing the values for the kernel parameters that we decided to modify and the results we obtained from them. We ran eleven tests.

Test	sched_latency_ns	sched_min_granularity_ms	sched_rr_timeslice_ms	vm.swappiness	**Result (s)**
1	9000000	4000000	3	90	**4.76**
2	9000000	4000000	250	10	**4.63**
3	7000000	2800000	50	40	**4.75**
4	6000000	25000000	25	60	**4.72**
5	6000000	25000000	25	10	**4.51**
6	6000000	25000000	50	10	**4.98**
7	6000000	25000000	13	10	**4.81**
8	8000000	35000000	25	10	**4.66**
9	12000000	55000000	25	10	**4.93**
10	5000000	20000000	50	10	**4.71**
11	7000000	3000000	25	10	**4.66**
AVERAGE	**7363636.364**	**20345454.55**	**49.18181818**	**24.54545455**	**4.738181818**



As we can observe the best results was obtained in test five with a result of 4.51 seconds spent performing the calculations of the benchmark. Also, we can observe that the worst performance was obtained with test six which has a result of 4.98 seconds. It is worth a mention that both test five and six have the same values except for the timeslice value. Then we can conclude that in this case a smaller value on the timeslice can have an impact on performance.

8 Conclusion

In cases like this, where we want to improve something that requires a wide range of tests is a more efficient option to achieve a good result. This project allow us to explore how to optimize the kernel for performance according to our needs and also to understand one of the applications of a genetic algorithm.

At first, we did not have an initial idea of how much impact the changes on kernel values would actually reflect on real time of execution. Our fluctuations are all around in decimals of a second and it might seem like it does not have a big impact on performance but when we thought of big calculations that should be running for days, we think that this modification take more sense, since they would have an impact of hours.

It was really entertaining for us to do this project because we started optimization for H.264 encoding, then we changed it for measuring the performance of a videogame and then we applied our previous learning to a calculation of pi. And also, during this way we were researching about genetic algorithms looking for a way to improve our project.

Actually, we think that further development for this project would be applying the genetic algorithm to create a software in which given a test it start running and tunning the kernel variables for hours and at the end, it adjusts them in order to achieve the best performance.

When you are involved in a career that needs to analyze a lot of variables and how they affect your system when they change, we think you develop a lot of abilities to analyze those results and come up with better ideas every time to achieve what you want. At the start of this course, when we used our computers we did not though about stuff like the kernel and the scheduler, now we do and I think we are going to look forward to these kind of information and how to perform optimizations to achieve better results at whatever cool stuff we do in the following semesters.

9 References

Genetic Algorithm. (s.f.). Genetic Algorithms. Retrieved from:
https://www.doc.ic.ac.uk/ nd/surprise96/journal/vol1/hmw/article1.html

Kernel. (s.f.). Deadline Task Scheduling. Retrieved from:
https://www.kernel.org/doc/Documentation/scheduler/ sched-deadline.txt

Open Suse.(s.f.). Tuning the time scheduler. Retrieved from:
https://doc.opensuse.org/documentation/html/
openSUSE_121/opensuse-tuning/cha.tuning.taskscheduler.html

Phoronix Test Suite. (2016). OpenBenchmarking.org. Retrieved from
AIO-Stress [pts/aio-stress] : https://openbenchmarking.org/test/pts/aio-stress

Red Hat. (s.f.). CPU Scheduling. Retrieved from:
https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/
html/Performance_Tuning_Guide/s-cpu-scheduler.html

Russel, S., Norving, P. (1994). Beyond Classical Research. In Artificial Intelligence:
A Modern Approach. Third Edition. Englewood Cliffs, NJ: Prentice Hall.