Implementation of CFS for Complex Mathematical Sets

IT B.Tech course project for the course

Operating System (IT 253)

By

Kevin Saji Jacob (Roll No. 221IT038)

Sricharan Sridhar (Roll No. 221IT066)

Under the Supervision of

Janani T

Department of Information Technology

National Institute of Technology Karnataka

Surathkal - 575025, India

March, 2024

DECLARATION

We hereby declare that the work presented in this report titled "Implementation of CFS for Complex Mathematical Sets" is an authentic record of our own work carried out under the supervision of Janani T at NITK Surathkal . The work has not been submitted, in part or full, for any other degree or diploma at this or any other institution.

Kevin Saji Jacob,
Sricharan Sridhar

CERTIFICATE

This is to certify that the work presented in this report titled "Implementation of CFS for Complex Mathematical Sets" is an authentic record of work done by Kevin Saji Jacob and Sricharan Sridhar under my supervision. To the best of my knowledge, the contents of this report have not been submitted, in part or full, for any other degree or diploma at this or any other institution.

Janani T
Assistant Professor Grade II
National Institute of Technology Karnataka, Surathkal
March 22, 2024

Table of Content

List of Figures

CHAPTER 1 INTRODUCTION

Scheduling algorithms play a crucial role in optimizing resource utilization and maximizing system efficiency in various computing environments, ranging from operating systems to network systems and beyond. At its core, scheduling involves the allocation of resources to tasks or processes over time, with the aim of meeting certain objectives such as minimizing latency, maximizing throughput, or ensuring fairness.

1.1 WHAT IS A SCHEDULER

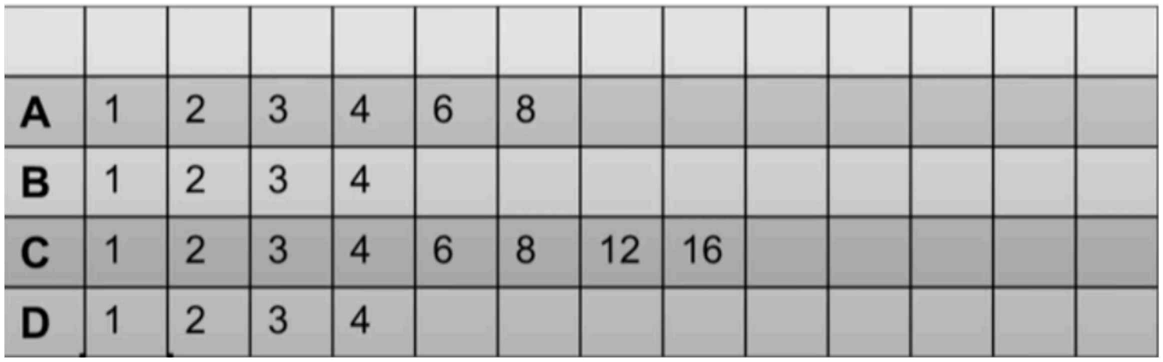
Scheduling algorithms are integral components of operating systems and computing systems, responsible for managing the allocation of resources among various tasks or processes. They govern the order in which tasks are executed and the resources they receive, aiming to optimize system performance and resource utilization. These algorithms come in various forms, each designed with specific objectives and characteristics. Some prioritize tasks based on their arrival order, while others consider factors like task duration or priority levels. Additionally, certain algorithms focus on fairness and responsiveness by allocating resources evenly among tasks or employing time-sharing techniques. Understanding the principles and trade-offs of different scheduling algorithms is crucial for system designers and administrators to select or design appropriate scheduling policies that meet the unique requirements of their systems and workloads, ultimately enhancing overall system efficiency and performance.

1.2 OVERVIEW OF THE PROJECT

1.2.1 CFS

The Completely Fair Scheduler (CFS) is a widely used scheduling algorithm in modern Linux kernels, designed primarily for CPU scheduling. Unlike traditional schedulers that rely on fixed time slices or priorities, CFS operates on the principle of fairness, aiming to distribute CPU time equitably among processes. In CFS, each process is assigned a virtual runtime that represents the amount of CPU time it has received. The scheduler continually adjusts these runtimes based on the execution history of processes, ensuring that each process gets its fair share of CPU time over a given period. This approach contrasts with older scheduling algorithms like the FCFS scheduler, which could lead to situations where certain processes monopolize the CPU while others starve for resources.

One of the key features of CFS is its ability to handle dynamic workloads effectively. By maintaining a red-black tree data structure to store processes based on their virtual runtimes, CFS can efficiently select the process with the smallest virtual runtime for execution. This ensures that processes with shorter runtimes or those that have been waiting longer are given priority, promoting fairness in resource allocation. Additionally, CFS employs various mechanisms to adapt to changes in system load, such as dynamically adjusting the time quantum assigned to each process. Overall, the Completely Fair Scheduler strikes a balance between fairness and efficiency, making it a preferred choice for CPU scheduling in modern operating systems like Linux.



A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

Implementation of CFS with a 4ms quantum

Figure 1.1

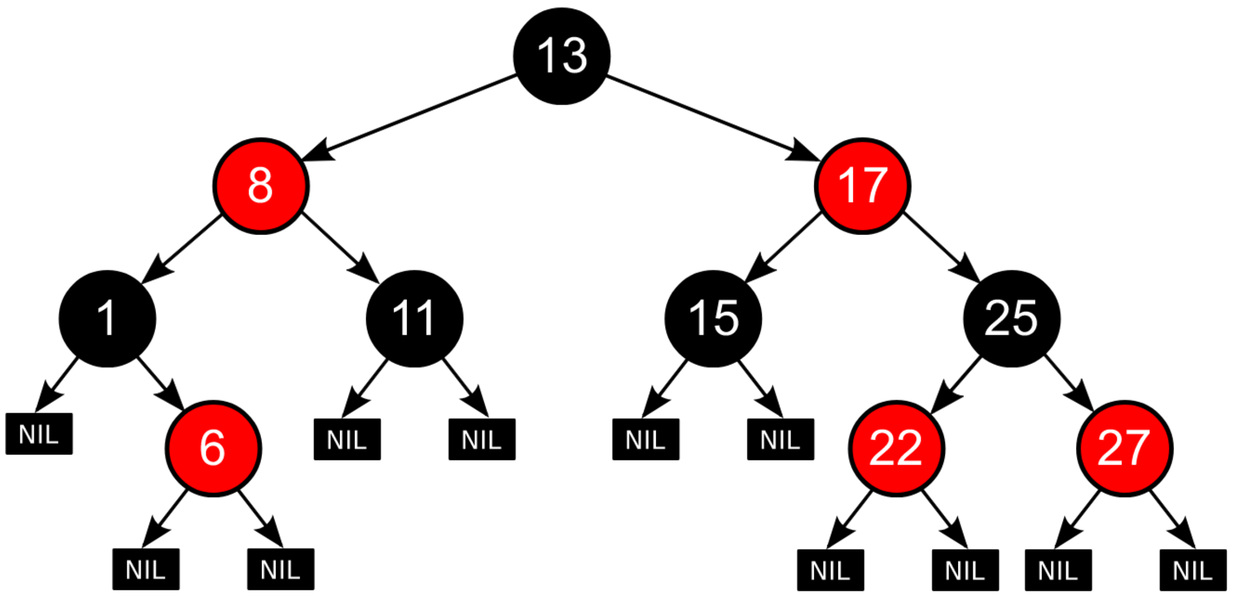
1.2.2 Red-Black Tree

A red-black tree is a type of self-balancing binary search tree that maintains balance during insertions and deletions. It achieves this balance by enforcing certain properties and employing rotation operations when necessary. In a red-black tree, each node is augmented with an additional bit of information, referred to as its color, which can be either red or black. These colors are assigned based on specific rules that ensure the tree remains balanced. The key properties of a red-black tree include:

- Every node is either red or black
- The root node is black
- Red nodes cannot have red children
- Every path from a node to its descendant leaves must contain the same number of black nodes.

These properties guarantee that the longest path from the root to any leaf is no more than twice the length of the shortest path, ensuring the tree remains relatively balanced and efficient for various operations.

Red-black trees are widely used in computer science and software engineering due to their efficient performance and predictable balance characteristics. They are particularly well-suited for applications that involve frequent insertions and deletions, such as in the implementation of associative arrays and sets. The self-balancing property of red-black trees ensures that the height of the tree remains logarithmic, which translates to efficient search, insertion, and deletion operations with a worst-case time complexity of O(log n). This makes red-black trees a popular choice for implementing data structures and algorithms where performance and balance are critical considerations such as operating systems.



Example of Red-Black tree

Figure 1.2

CHAPTER 2   OBJECTIVE

The primary objective behind implementing a Completely Fair Scheduler (CFS) is to ensure equitable allocation of CPU resources among competing processes or tasks in a computing system. Unlike traditional schedulers that may favor certain processes or suffer from inefficiencies such as process starvation or monopolization, CFS strives to provide fairness by dynamically adjusting the allocation of CPU time based on each process's needs and behavior. By maintaining a virtual runtime for each process and employing efficient scheduling mechanisms, CFS aims to prevent any single process from dominating the CPU, thereby enhancing overall system responsiveness and fairness. Moreover, CFS adapts to changes in system load and workload characteristics, ensuring that CPU resources are distributed proportionally among active processes, regardless of their arrival order or execution history. Thus, the implementation of a Completely Fair Scheduler contributes to a more balanced and efficient utilization of CPU resources, fostering a more responsive and equitable computing environment.

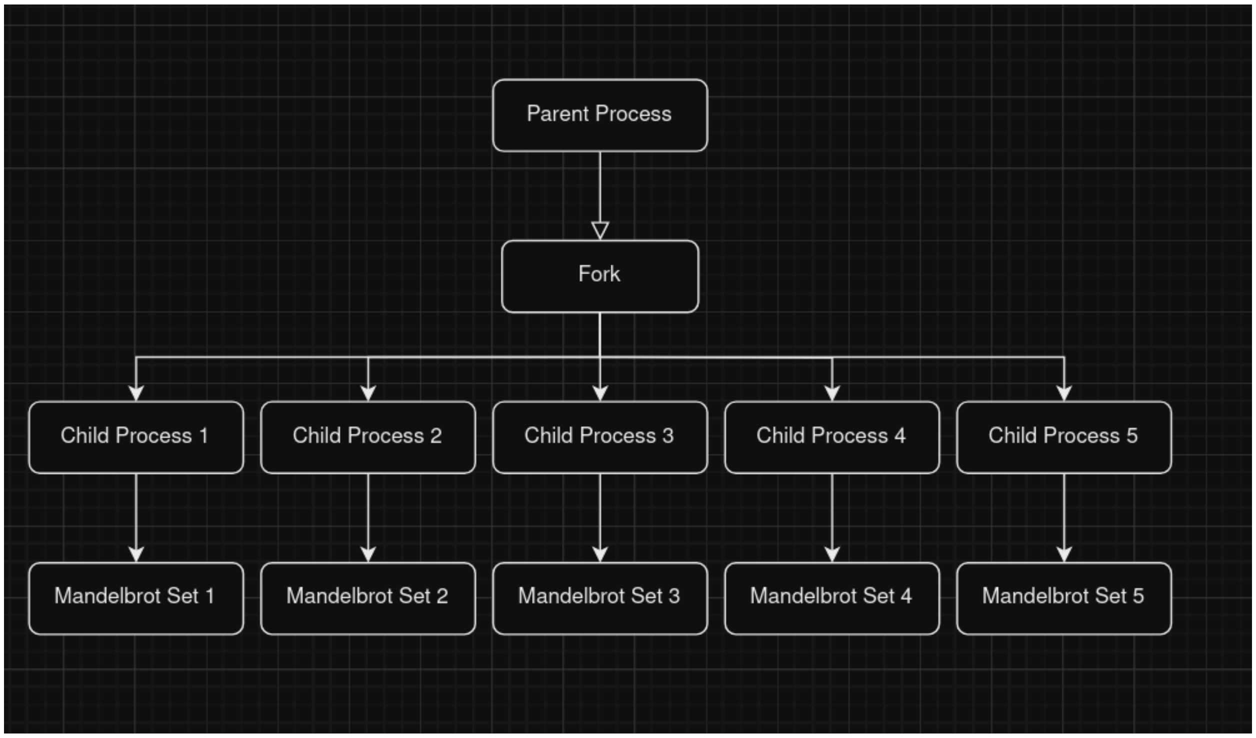
CHAPTER 3 METHODOLOGY

For implementing a completely fair scheduler we used the generation of mandelbrot set as the program to be run by each process, these programs can be set up to have different run times by changing the properties such as maximum iterations and resolution of the generated image.

3.1 Mandelbrot Set

The Mandelbrot set is a captivating and complex fractal discovered by Benoit B. Mandelbrot in the 1970s. It is defined as the set of complex numbers c for which the iterative process $z_{n+1}=z_n^2+c$ remains bounded when iterated infinitely, starting with $z_0=0$. The result of this iterative process determines whether a particular point in the complex plane belongs to the Mandelbrot set or not. Points within the set remain bounded under iteration, producing intricate patterns of self-similarity and detail, while points outside the set escape to infinity. The boundary of the Mandelbrot set exhibits a highly intricate and infinitely complex structure, known as the "Mandelbrot set fractal boundary," characterized by its self-similarity across different scales.

CHAPTER 4   FLOW CHART



Flowchart of the program

Figure 4.1

CHAPTER 5 IMPLEMENTATION

5.1 Mandelbrot Set Creation

```
double x, y; // Coordinates of the current point in the complex plane.
  double u, v; // Coordinates of the iterated point.
  int i, j;         // Pixel counters
  int k;            // Iteration counter
  for (j = 0; j < yres; j++)
  {
        y = ymax - j * dy;
        for (i = 0; i < xres; i++)
        {
        double u = 0.0;
        double v = 0.0;
        double u2 = u * u;
        double v2 = v * v;
        x = xmin + i * dx;
        /* iterate the point */
        for (k = 1; k < maxiter && (u2 + v2 < 4.0); k++)
        {
        v = 2 * u * v + y;
        u = u2 - v2 + x;
        u2 = u * u;
        v2 = v * v;
        };
        /* compute  pixel color and write it to file */
        if (k >= maxiter)
        {
        /* interior */
        const unsigned char black[] = {0, 0, 0, 0, 0, 0};
        fwrite(black, 6, 1, fp);
        }
        else
        {
        /* exterior */
        unsigned char color[6];
        color[0] = k >> 8;
        color[1] = k & 255;
        color[2] = k >> 8;
        color[3] = k & 255;
        color[4] = k >> 8;
        color[5] = k & 255;
        fwrite(color, 6, 1, fp);
        };
        }
}
```

Snippet of Mandelbrot Set Generation

Snippet 5.1

This C program generates an image of the Mandelbrot set based on user-defined parameters such as the range of the complex plane coordinates, maximum iteration count, and resolution. It parses command-line arguments for these parameters, calculates the image dimensions. The program iterates over each pixel in the image, determining whether the corresponding complex number is within the Mandelbrot set through iterative calculations. Based on the iteration count, it assigns colors to pixels, with points inside the set painted black and exterior points colored based on the number of iterations.

5.2 Red-Black Trees

This C program implements a Red-Black Tree (RBT), a type of self-balancing binary search tree, which maintains balance during insertion and deletion operations. The program defines structures for RBT nodes and the tree itself. It includes functions for creating a new RBT, inserting nodes, fixing violations of RBT properties after insertion, deleting nodes, and maintaining RBT properties after deletion. Additionally, it provides functions for traversing the tree and retrieving the smallest node (used for a specific purpose in this implementation). The RBT ensures logarithmic time complexity for insertion, deletion, and search operations, making it efficient for dynamic sets or associative arrays.

```
void RBInsert(RBTree *T, float key, int pid)
{
  RBTreeNode *z = newRBTreeNode(key, pid);
  RBTreeNode *y = T->nil;
  RBTreeNode *x = T->root;
  while (x != T->nil)
  {
        y = x;
        if (z->key < x->key)
        x = x->left;
        else
        x = x->right;
  }
  z->parent = y;
  if (y == T->nil)
        T->root = z;
  else if (z->key < y->key)
        y->left = z;
  else
        y->right = z;
  z->left = z->right = T->nil;
  z->color = RED;
  RBInsertFixup(T, z);
}
```

Snippet of RB-Tree Insert

Snippet 5.2

```
RBTreeNode *popSmallest(RBTree *T)
{
   if (T->root == T->nil)
   {
      printf("Tree is empty\n");
      return NULL;
   }
   RBTreeNode *temp;
   RBTreeNode *smallest = treeMinimum(T->root, T->nil);


   temp->key = smallest->key;
   temp->process_id = smallest->process_id;
   RBDelete(T, smallest);
   return temp;
}
```

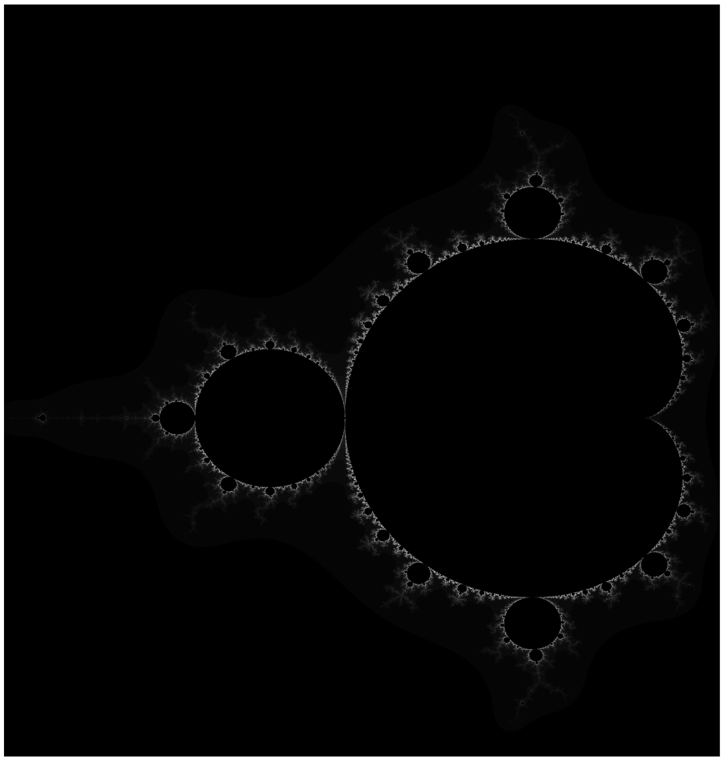
Snippet of RB-Tree Pop

Snippet 5.3

CHAPTER 6 RESULTS AND FUTURE WORK

6.1 Results

The CFS algorithm aims to provide fairness in CPU time allocation by dynamically adjusting the scheduling priorities based on the amount of CPU time consumed by each process. In this scenario, as each process generates a Mandelbrot set with varying resolutions, the CPU workload for each process would vary accordingly. The CFS algorithm would ensure that processes with heavier computational tasks, such as generating Mandelbrot sets with higher resolutions, receive their fair share of CPU time, while still allowing processes with lighter tasks to execute. This dynamic allocation of CPU resources ensures that all processes progress towards completion at a similar pace, regardless of their computational intensity, thereby maximizing overall system throughput and responsiveness.

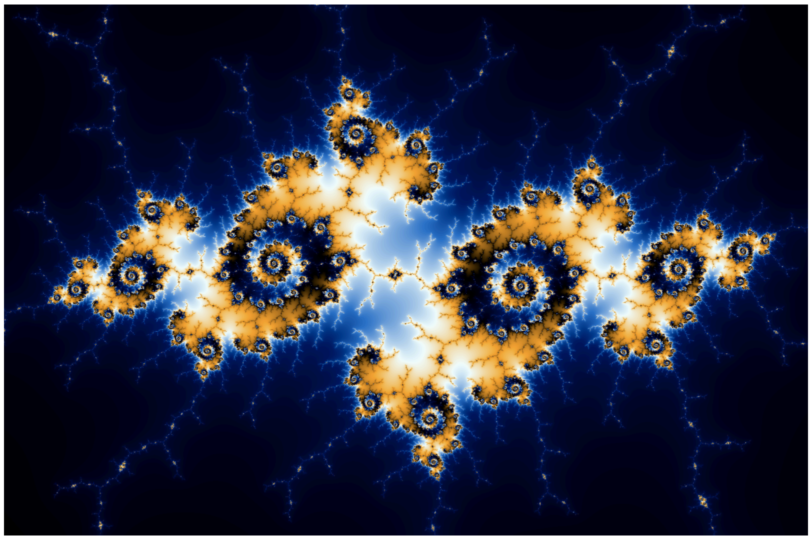


Sample output (Resolution: 4096)

Figure 6.1

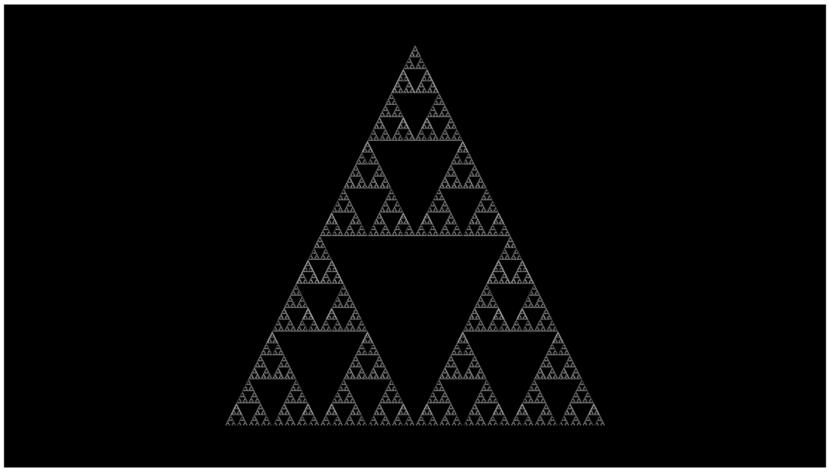
6.2 Future Work

For future work we will be extending the implementation to include Julia sets and Sierpinski triangles alongside the Mandelbrot set for process creation.



Example of Julia set

Figure 6.2



Example of Sierpinski Triangle

Figure 6.3