OS PROJECT – TASK MANAGER

Made by K213309, K213206, K214833

Table of Contents

Introduction

Our motive was to recreate a “Process Monitor System” similar to the Windows “Task Manager” or Ubuntu “System Monitor”. This would include killing a process, setting the priority and affinity of a process, and monitoring CPU and MEM usage percentages.

Team Members:

1. Mohammad Yehya Hayati, K213309
2. Taha Ahmed, K214833
3. Sufyan Abdul Rasheed, K213206

Background

The main research was done on how to implement different Operating System techniques to create a working Task Manager. To do this, we created 5 different header files so that we can accommodate our needs.

The 1st header file, MyMacros.h, consists of the inclusion of external header files and macro definitions.

```
#pragma once

#include <gtk/gtk.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/resource.h>
#include <sched.h>
#include <wait.h>
#define HARDWARE_CONCURRENCY (4)
#define GraphPoints 180
```

The 2nd header file, MyData.h, consists of the user defined structures and global variables.

```
#pragma once

#include "MyMacros.h"

typedef char BYTE;

typedef struct
{
    GtkBuilder      *builder;
    GtkWidget       *window, *Fixed, *draw1, *draw2,
                    *PriorityPicker, *PriorityEntry,
                    *core[4];
    GtkTreeView     *tv1;
    GtkTreeSelection    *tv1Selection;
    GtkListStore        *ProcessorData, *PriorityStore;
    GtkTreeViewColumn   *PID, *Name, *CPU, *MEM;
    GtkCellRenderer     *PID_, *Name_, *CPU_, *MEM_;
    GtkTreeIter         SelectedRow;
    GtkTreeModel        *model;

    BYTE Affinity;
    int PriorityLevel;
    int TimerOn;
    int cpuUtil, cpu[GraphPoints];
    int memUtil[5], mem[5][GraphPoints];
    double RGBGraph[5][3];
} TMData;
TMData TMD;

typedef struct
{
    int *argc;
    char ***argv;
} ARG;

sem_t ProcessMutex;
sem_t CPUMutex;
sem_t MEMMutex;
```

The 3rd header file, MyHashTable.h, consists of a modified hash table that we use to save process information and use with the GUI.

```c
#include "MyData.h"
#define TableSize 300

typedef struct Node Node;
struct Node
{
void NodeConstructor(Node* x, char val[4][1024])
{
typedef struct HashTable
{
void HashTableConstructor(HashTable* a)
{
void HashTableDestructor(HashTable *a)
{
int Hash(char *value)
{
int Search(HashTable* a, char val[4][1024])
{
void Insert(HashTable* a, char val[4][1024])
{
void Delete(HashTable* a)
{
```

The 4th header file, MySignals.h, consists of all the signals that are used by GUI aspects like buttons and lists.

```c
#pragma once

#include "MyData.h"

gboolean on_draw1_draw (GtkWidget *widget, cairo_t *cr, gpointer data)
{
gboolean on_draw2_draw (GtkWidget *widget, cairo_t *cr, gpointer data)
{
void on_tv1Selection_changed(GtkWidget *c)
{
void on_KillButton_clicked(GtkButton *b)
{
void on_PriorityButton_clicked(GtkButton *b)
{
void on_PriorityPicker_changed(GtkComboBox *c, GtkEntry *e)
{
void on_AffinityButton_clicked(GtkButton *b)
{
void on_cpu_toggled(GtkCheckButton *b)
{
```

The 5th header file, MyThreadFunctions.h, consists of the main driver functions which are also multithreaded to ensure optimal efficiency.

```
#pragma once

#include "MySignals.h"
#include "MyHashTable.h"

void Exit_Program()
{
void *ProcessHandler(void *args)
{
void *CpuHandler(void *args)
{
void *MemHandler(void *args)
{
void *Gui(void *args)
{
int TaskManagerStart(ARG *x)
{
```

We will further explain the code in the <u>Methodology</u> section.

Platform and Languages

If it wasn't already obvious, our coding language in C. However, for the implementation of GUI, we used a C library called GTK+. In order to save time and effort, we used an IDE for creating a GUI called Glade.

Glade creates a .glade file which in essence is an .XML file which lets you define and store data in a shareable manner. Since GTK+ is compatible with Glade, it introduces some functions to import from a .glade file, which you can use implement a GUI.

Methodology

(For further inspection, see attached code)

First and foremost, we start from our C file which calls a function from MyThreadFunctions.h, TaskManagerStart ().

```c
int TaskManagerStart(ARG *x)
{
    sem_init(&ProcessMutex, 0, 0);
    sem_init(&CPUMutex, 0, 0);
    sem_init(&MEMMutex, 0, 0);
    TMD.PriorityLevel = 0;
    TMD.TimerOn = 1;
    for(int i = 0 ; i < GraphPoints ; i++) TMD.cpu[i] = 0;
    for(int i = 0 ; i < 5 ; i++) for(int j = 0 ; j < GraphPoints ; j++) TMD.mem[i][j] = 0;
    double temp[5][3] = {{0.0,0.7,0.7},{1,0,0},{0,1,0},{1,1,0},{1,0.5,0}};
    for(int i = 0 ; i < 5 ; i++) for(int j = 0 ; j < 3 ; j++) TMD.RGBGraph[i][j] = temp[i][j];
    pthread_t GuiThread;
    pthread_create(&GuiThread, NULL, Gui, (void*)x);
    pthread_join(GuiThread, NULL);
    return 0;
}
```

All this function does is set standard data members (non-GTK+) and starts a new thread called GUIThread which execute the GUI function, which will perform the main work.

Since the code for the GUI function is too large to insert, we will leave it to you to explore the attached code. The GUI function sets all GTK+ data members from the .XML file. Once they are set, it creates 3 threads each of which perform different work. The three threads execute the function CpuHandler, MemHandler, and ProcessHandler.

The CpuHandler function implements the code to calculate the average cpu usage percentage every 1 second. This is done by reading a system file called /proc/stat. Once the data is read, some string and array manipulation is performed and a signal is sent to a GTK+ object called draw, which does exactly what it sounds like; draw.

```c
void *CpuHandler(void *args)
{
    sem_wait(&CPUMutex);

    while(TMD.TimerOn)
    {
        static long time1 = 0, time2 = 0;
        static int flag = 0;
        char line[128], dummy[32];

        FILE *p1 = fopen("/proc/stat","r");
        fgets(line,128,p1);
        sscanf(line,"%s %ld", dummy, &time2);
        fclose(p1);

        if (!flag)
        {
            flag = 1;
            time1 = time2;
            sleep(1);
            continue;
        }

        TMD.cpuUtil = time2 - time1;
        for (int i = 0; i < GraphPoints-1 ; i++) TMD.cpu[i] = TMD.cpu[i+1];
        TMD.cpu[GraphPoints-1] = TMD.cpuUtil/(HARDWARE_CONCURRENCY);
        time1 = time2;
        gtk_widget_queue_draw(TMD.draw1);
        sleep(1);
    }
    pthread_exit(0);
}
```

The results of the draw section will be shown the results section.

The MemHandler function performs fairly similarly to the CpuHandler function. However, instead of reading from /proc/stat, it reads from /proc/meminfo. Also we read 4 different values which we also draw on a grid.

```c
void *MemHandler(void *args)
{
    sem_wait(&MEMMutex);

    while(TMD.TimerOn)
    {
        static long time1[5] = {0}, time2[5] = {0};
        static int flag = 0;
        char line[128], dummy[32];

        FILE *p1 = fopen("/proc/meminfo","r");
        for(int i = 0 ; i < 5 ; i++)
        {
            fgets(line,128,p1);
            sscanf(line,"%[^0-9] %ld", dummy, &time2[i]);
        }
        fclose(p1);

        if (!flag)
        {
            flag = 1;
            for(int i = 0 ; i < 5 ; i++)time1[i] = time2[i];
            sleep(1);
            continue;
        }
        for(int i = 1 ; i < 5 ; i++)
        {
            TMD.memUtil[i] = time2[i] * 100 / time1[0];
            for (int j = 0; j < GraphPoints-1 ; j++) TMD.mem[i][j] = TMD.mem[i][j+1];
            TMD.mem[i][GraphPoints-1] = TMD.memUtil[i];
            time1[i] = time2[i];
        }
        gtk_widget_queue_draw(TMD.draw2);
        sleep(1);
    }
    pthread_exit(0);
}
```

The ProcessHandler function is bit different from the prior functions. It calculates and stores the PID, CPU%, MEM%, and Name of each running process in a hash table. It then puts this data into a GtkListStore which is a built-in linked list for the GTK+ library. This is done by using a system call called execlp (). We run a bash script which logs all the processes in a txt file, and then we read the txt file and manipulate the data to that we can store the data. After this, the data is ready for display, and is updated every second.

```
void *ProcessHandler(void *args)
{
    sem_wait(&ProcessMutex);

    char *vals[] = {NULL,"--sort=pid","--sort=%cpu", "--sort=%mem", "--sort=command"};
    char line[1024] = {0}, Info[4][1024] = {0};
    gtk_list_store_clear(TMD.ProcessorData);
    HashTable HT;
    HashTableConstructor(&HT);
    while(TMD.TimerOn)
    {
        if(!fork()) execlp("/bin/sh","/bin/sh","ProcessInfo.sh", vals[2], NULL);
        else wait(NULL);
        FILE *fptr = fopen("Processes.txt", "r");
        int x = 0;
        GtkTreeIter iter;
        while(TMD.TimerOn)
        {
            if (!fgets(line,1024,fptr) || !TMD.TimerOn)
            {
                fclose(fptr);
                break;
            }
            if(!x++) continue;
            int check = sscanf(line,"%s %s %s %[^\n]", Info[0], Info[1], Info[2], Info[3]);
            if(TMD.TimerOn && Search(&HT, Info) == -1) Insert(&HT, Info);
        }
        if(TMD.TimerOn)
        {
            sleep(1);
            Delete(&HT);
        }
    }
    HashTableDestructor(&HT);
}
```

However, if we just let the ProcessHandler function run like that, the list would just keep on getting appended over and over again. Therefore, we need to manipulate the data in the hash table which would in turn reflect over to the GtkListStore. This done by using two hash table functions called Search () and Insert ().

```c
int Search(HashTable* a, char val[4][1024])
{
    int h = Hash(val[3]);
    Node* temp = a->Table[h];
    while(temp != NULL)
    {
        if(!strcmp(temp->name, val[3]))
        {
            if(gtk_list_store_iter_is_valid(GTK_LIST_STORE(TMD.ProcessorData), &temp->iter))
            {
                gtk_list_store_set(GTK_LIST_STORE(TMD.ProcessorData), &temp->iter, 0, val[0], 1, val[1], 2, val[2], 3, val[3], -1);
                temp->flag = 1;
                return h;
            }
        }
        temp = temp->next;
    }
    return -1;
}
void Insert(HashTable* a, char val[4][1024])
{
    int h = Hash(val[3]);
    Node* newnode = (Node*)malloc(sizeof(Node));
    if(TMD.TimerOn)
    {
        NodeConstructor(newnode, val);
        if (a->Table[h] == NULL) a->Table[h] = newnode;
        else
        {
            Node* temp = a->Table[h];
            while (temp->next != NULL) temp = temp->next;
            temp->next = newnode;
            newnode->pre = temp;
        }
    }
    else free(newnode);
}
```

There are also some GTK+ objects defined to implement some features like killing a process using a button, setting priority using a combo box, setting processor affinity by using check boxes. To implement this procedure, GTK+ uses its own built-in signal system, therefore some signals were defined.

When the kill button is pressed,

```
void on_KillButton_clicked(GtkButton *b)
{
    if(!GTK_IS_TREE_MODEL(TMD.model))return;
    gchar *value;
    gtk_tree_model_get(TMD.model, &TMD.SelectedRow, 0, &value, -1);
    if(!kill(atoi(value), SIGKILL)) gtk_list_store_remove(GTK_LIST_STORE(TMD.ProcessorData), &TMD.SelectedRow);
}
```

When the priority button is pressed and when the combo box is opened,

```
void on_PriorityButton_clicked(GtkButton *b)
{
    if(!GTK_IS_TREE_MODEL(TMD.model))return;
    gchar *value;
    gtk_tree_model_get(TMD.model, &TMD.SelectedRow, 0, &value, -1);
    setpriority(PRIO_PROCESS, atoi(value), TMD.PriorityLevel);
}
void on_PriorityPicker_changed(GtkComboBox *c, GtkEntry *e)
{
    TMD.PriorityLevel = atoi(gtk_entry_get_text(e));
}
```

When the affinity button is pressed and when the check boxes are checked,

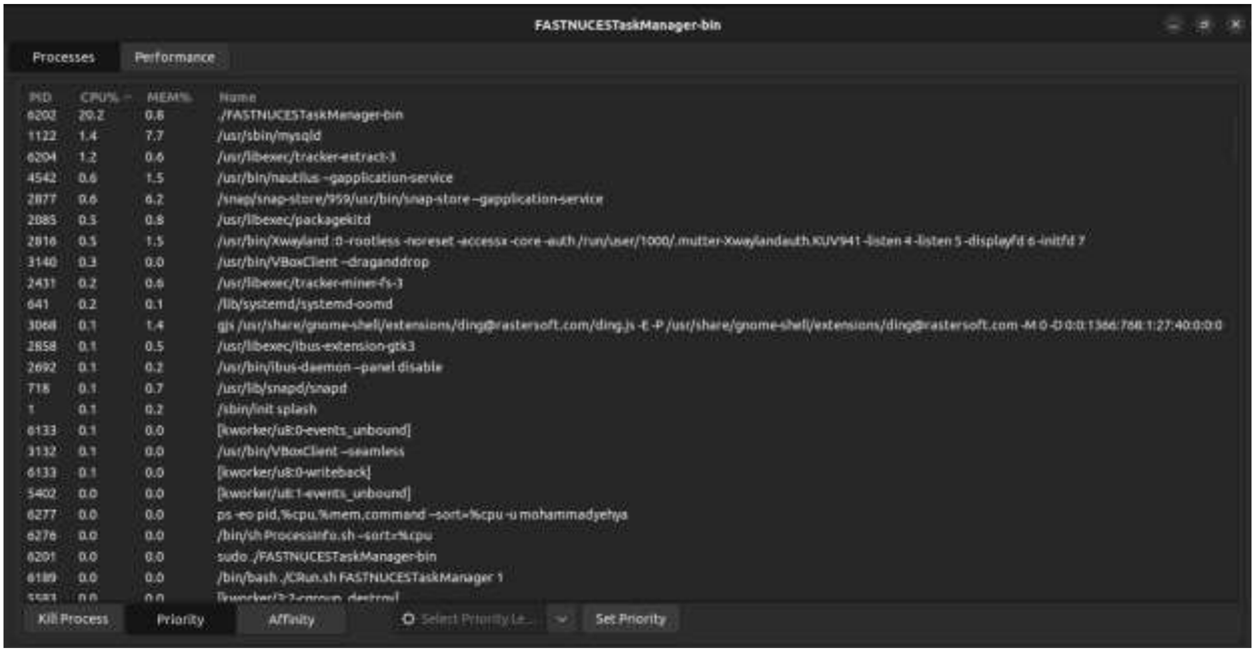
```
void on_AffinityButton_clicked(GtkButton *b)
{
    if(!GTK_IS_TREE_MODEL(TMD.model))return;
    gchar *value;
    cpu_set_t mask;
    CPU_ZERO(&mask);
    for(int i = 0 ; i < 4 ; i++) if((TMD.Affinity & 1 << i) == 1 << i) CPU_SET(i, &mask);
    gtk_tree_model_get(TMD.model, &TMD.SelectedRow, 0, &value, -1);
    sched_setaffinity(atoi(value), sizeof(mask), &mask);
}
void on_cpu_toggled(GtkCheckButton *b)
{
    int x = 0;
    sscanf(gtk_widget_get_name((GtkWidget*)b), "cpu%d", &x);
    TMD.Affinity ^= (1 << x);
}
```

(For further inspection, see attached code)

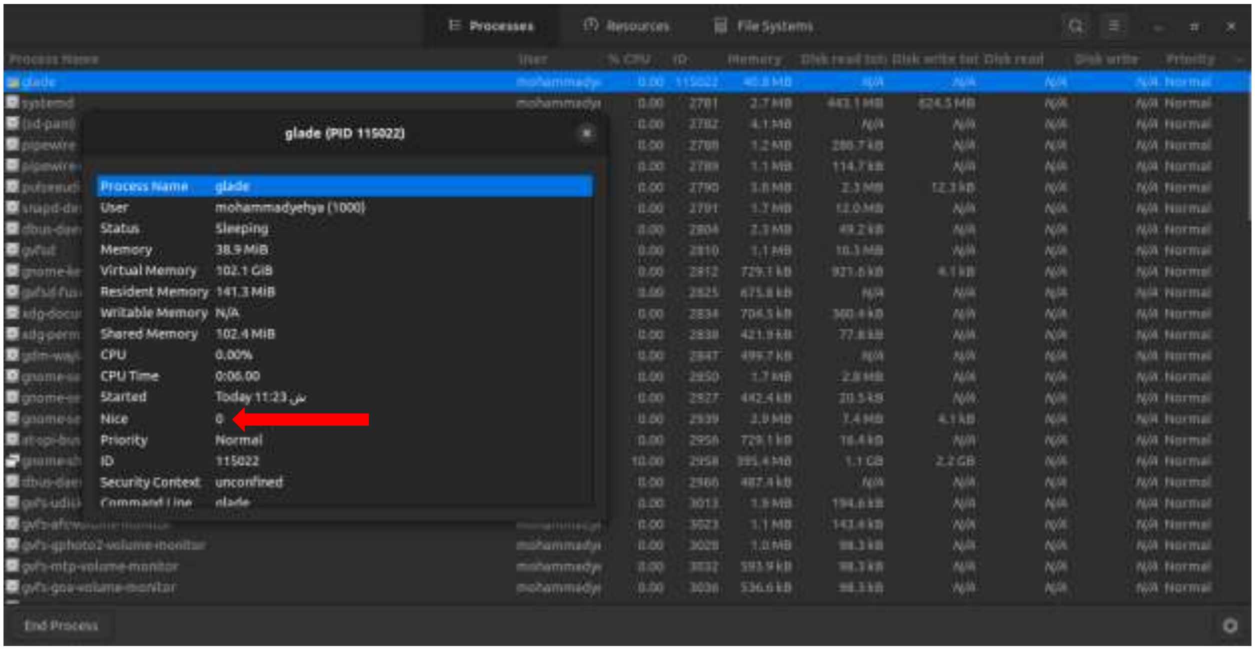
Results

In the end, when we combine all these theories and logic and mix them up with some programming skill we are left with a fully functional Task Manager. Here are some of the results. To check whether our application is properly setting priorities and affinities, we will compare with Ubuntu's built-in System Monitor.

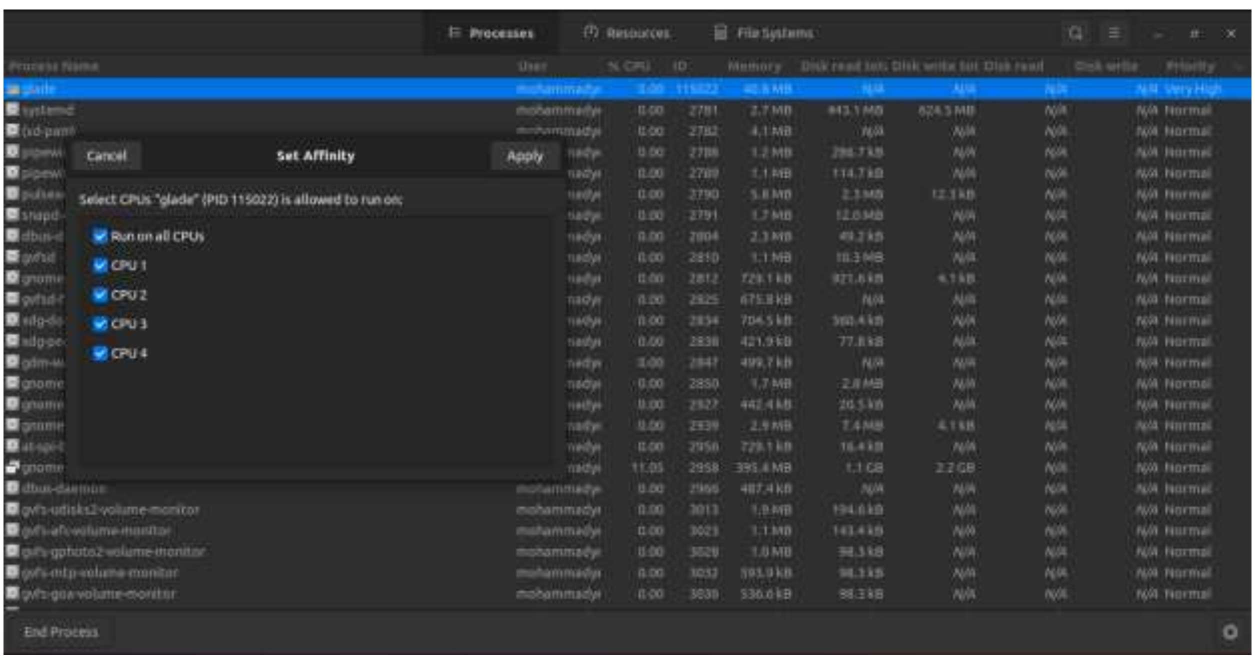
The first look at the project running.



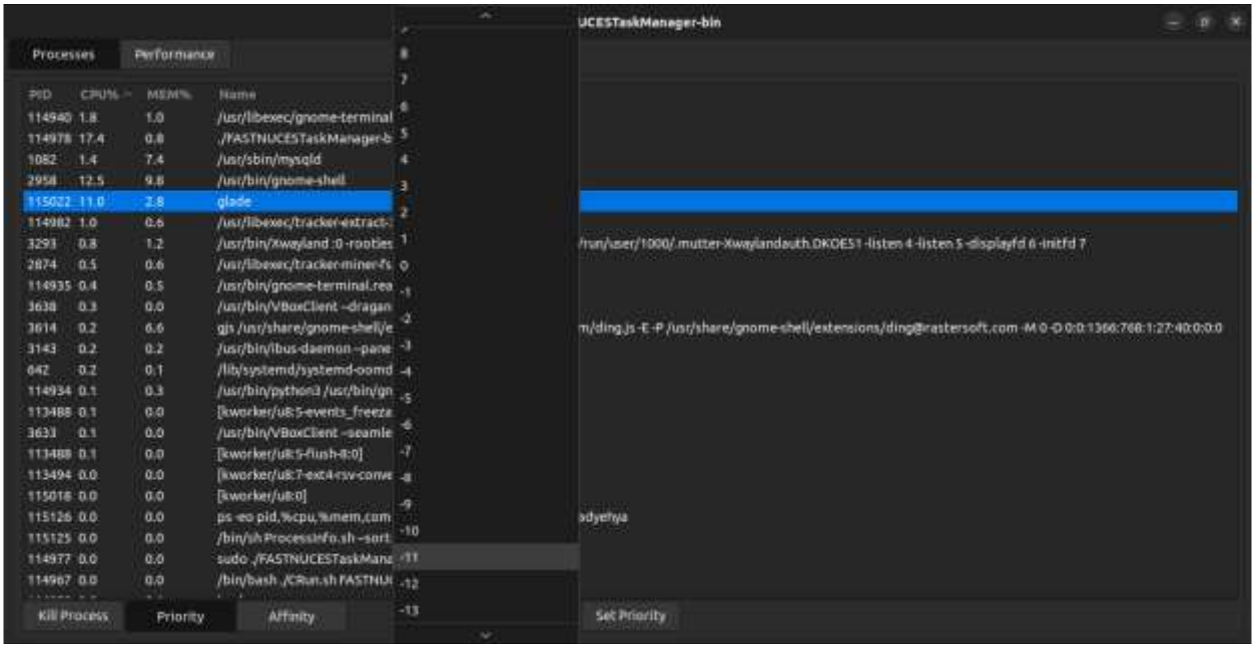
Before setting the priority. (Checked via System Monitor)



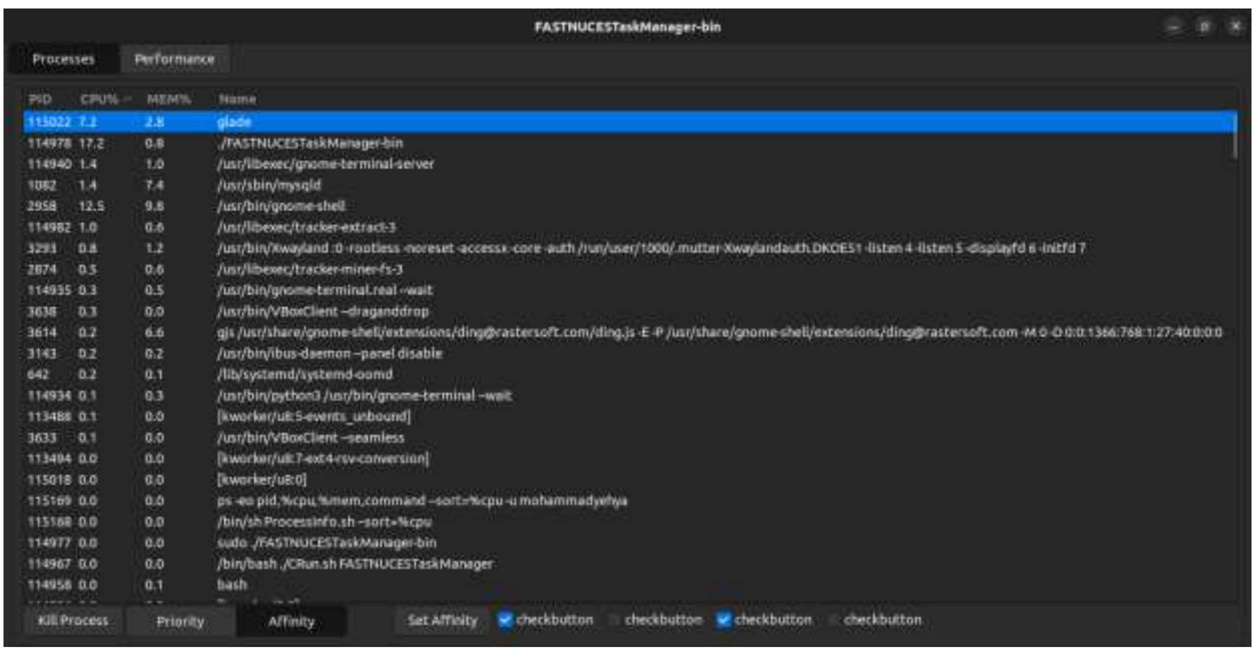
Before setting the affinity. (Checked via System Monitor)



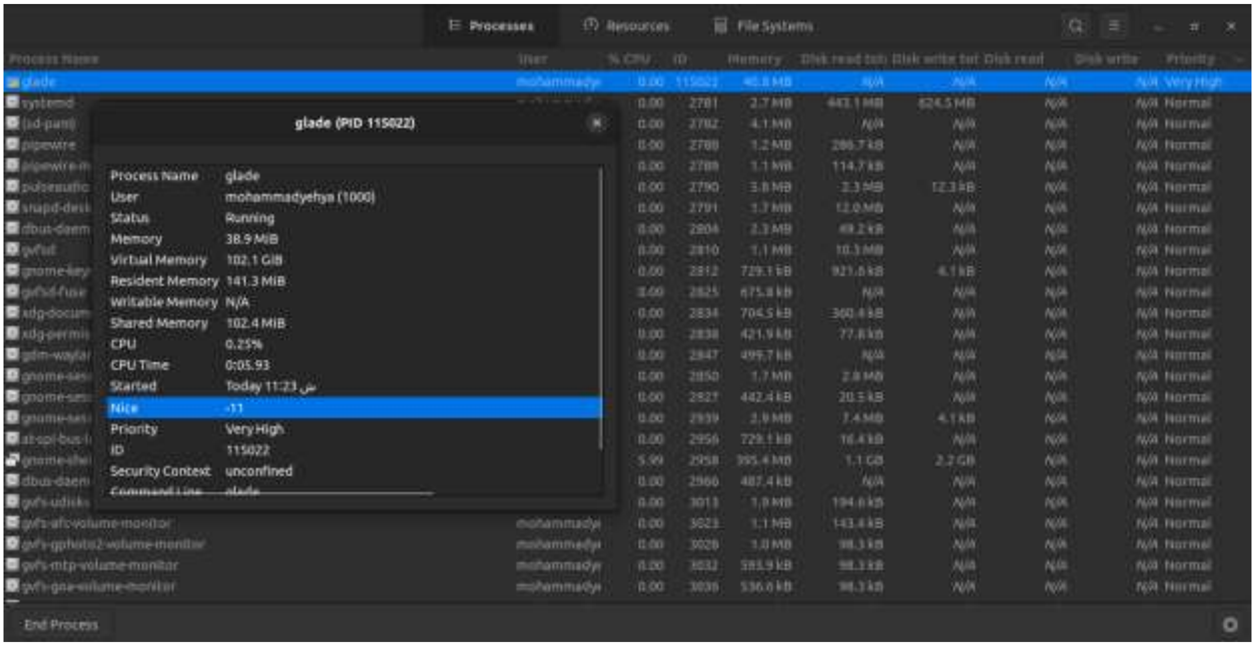
Setting the priority of a process.



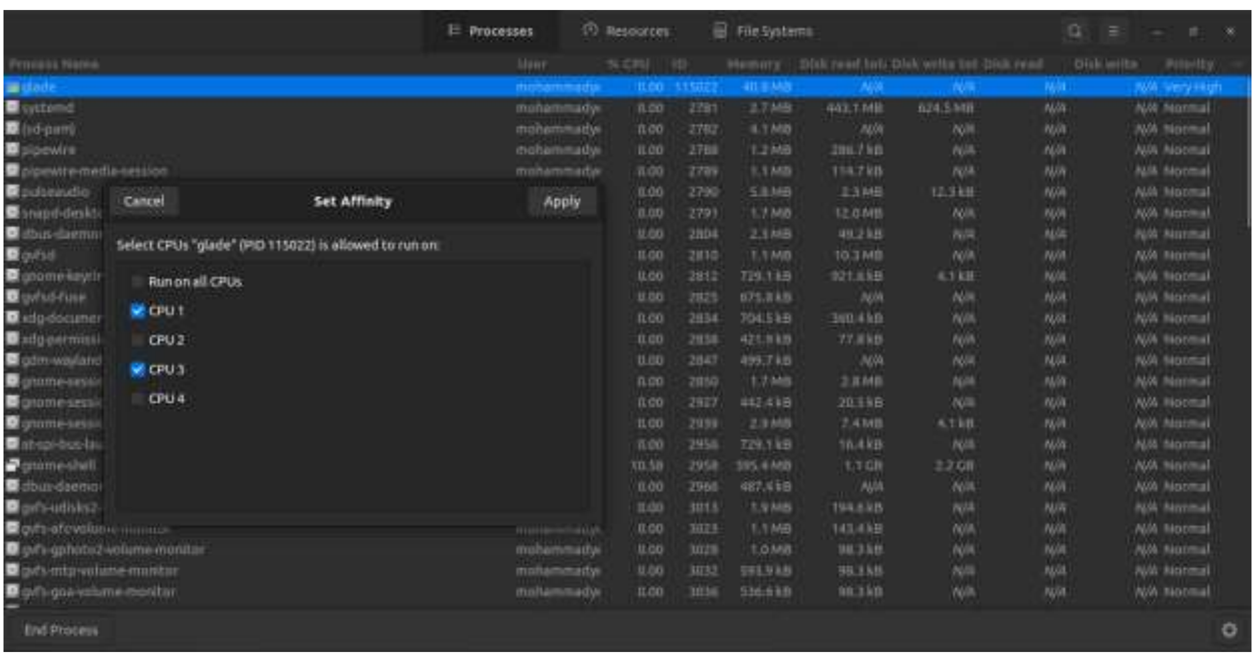
Setting the affinity of a process.



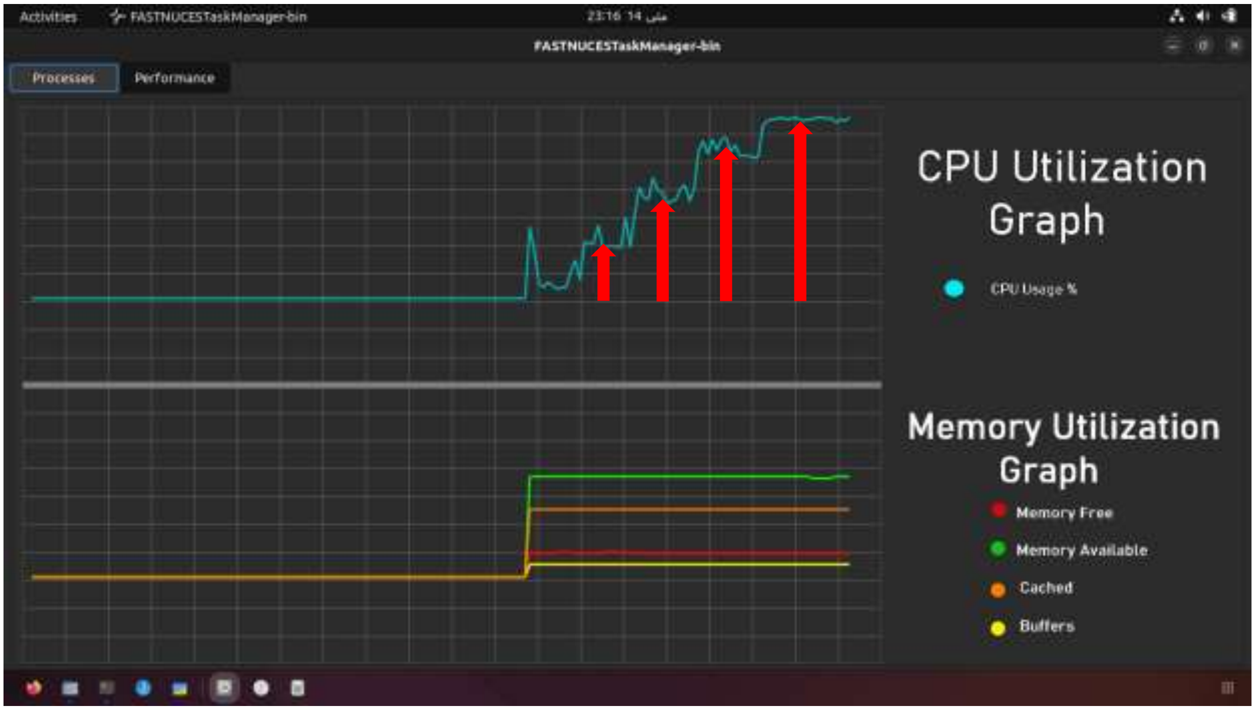
After setting the priority. (Checked via System Monitor)



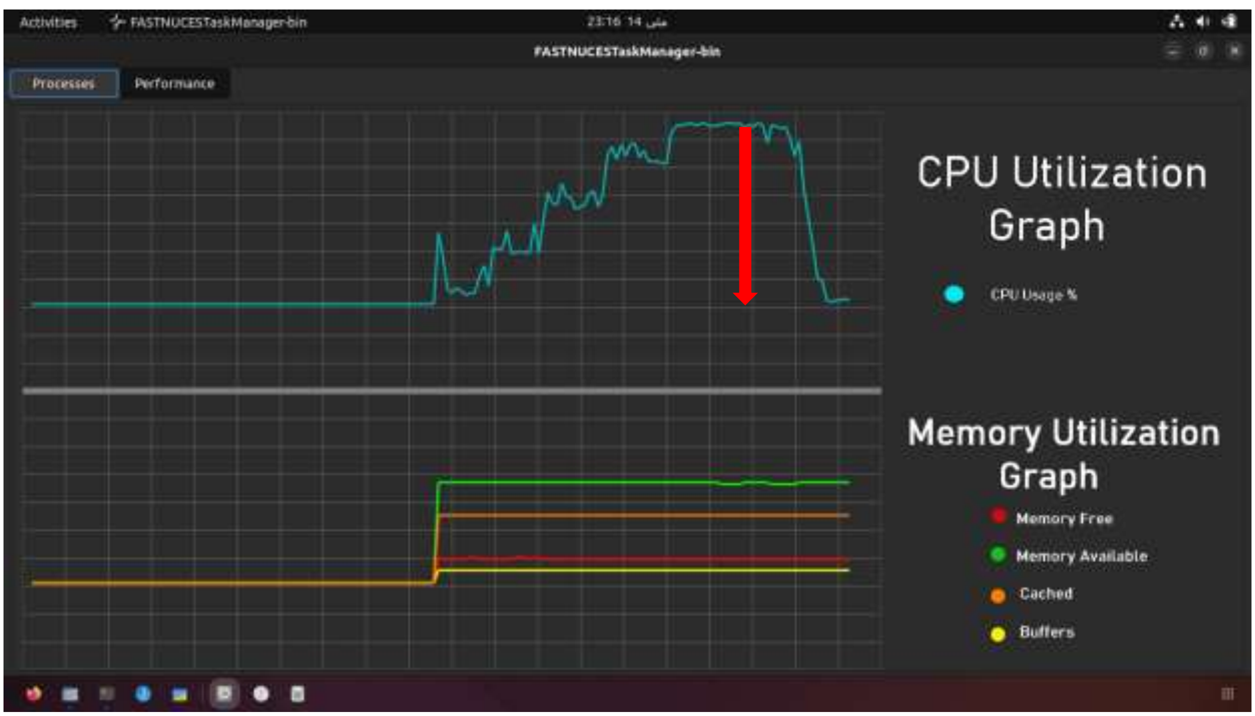
After setting the affinity. (Checked via System Monitor)



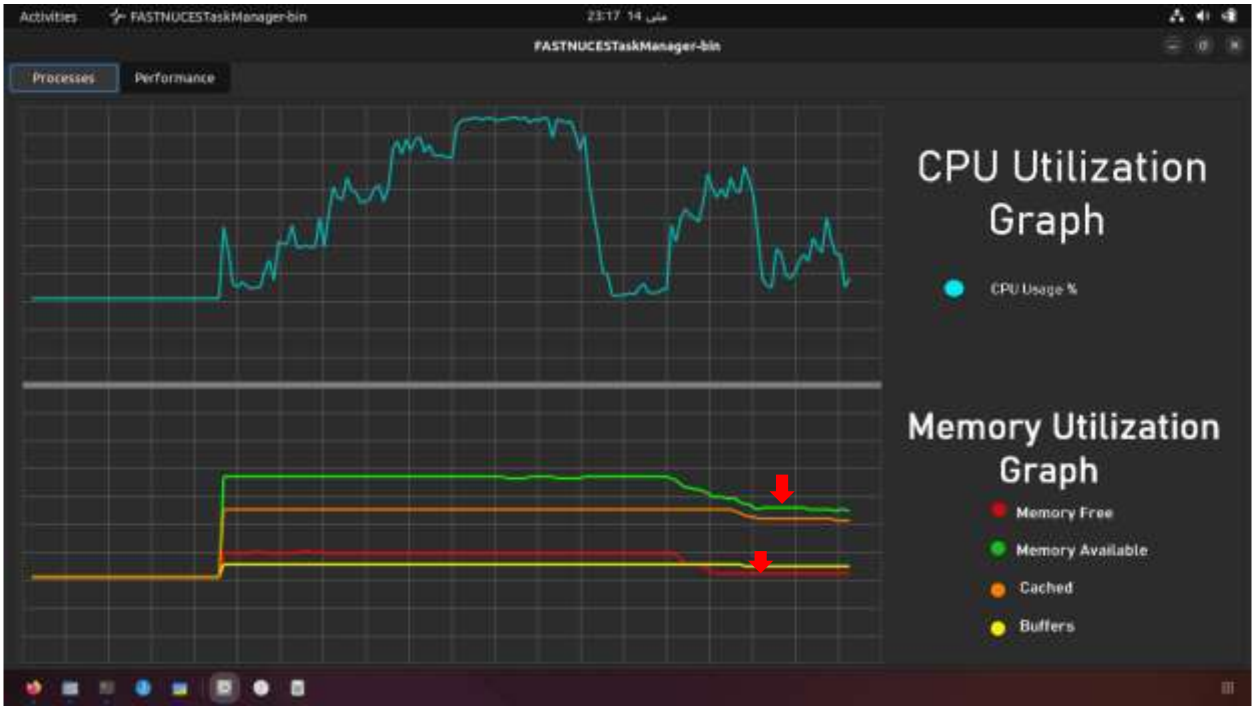
CPU & MEM Graphs (when running CPU intensive programs)



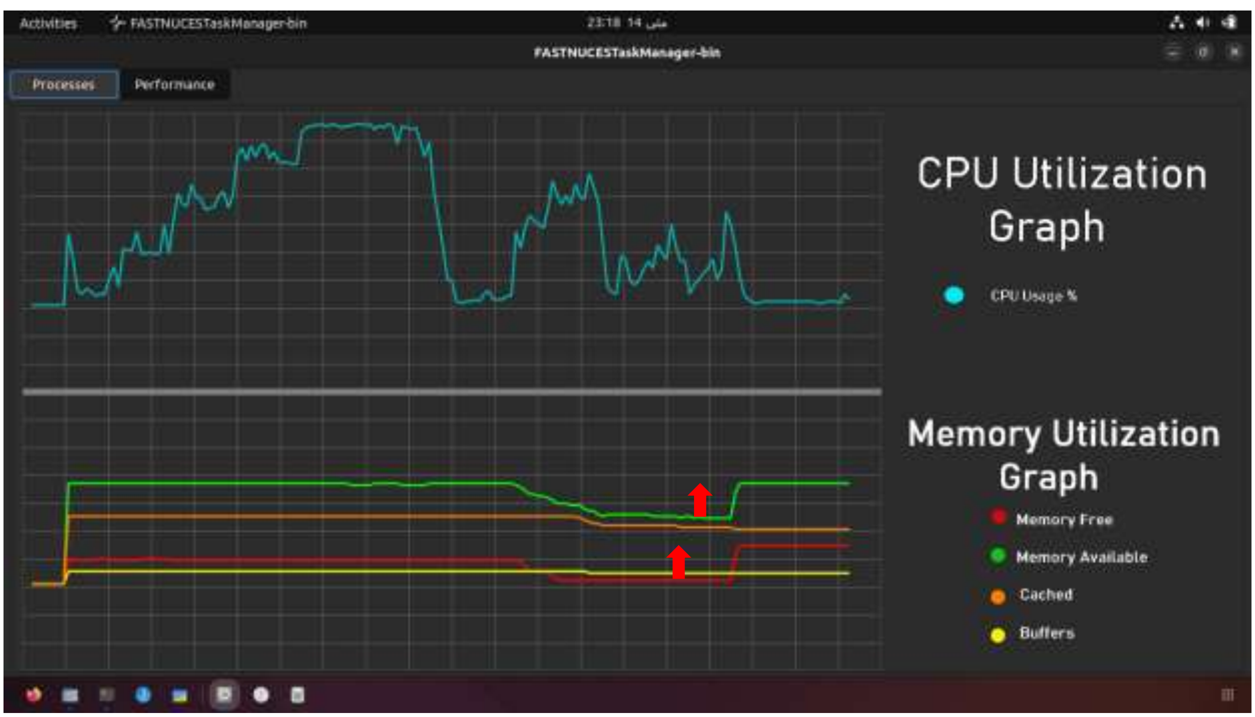
CPU & MEM Graphs (when killing CPU intensive programs)



CPU & MEM Graphs (when running MEM intensive programs)



CPU & MEM Graphs (when killing MEM intensive programs)



Problems & Challenges

There were many hurdles that we had to face to complete this project. The first problem being that many of GTK+ functions are actually deprecated. This means that no new updates will be further posted, and this also translates to no new bug fixes. Therefore, we had work around this issue.

Another issue was using C as our language. This meant that we no longer had access to the STL library which included a bunch of useful functions. Therefore, we had to do everything from scratch.

Conclusion & Breakdown

In the end, the outcome of the project is very satisfying, and we have learnt a lot from participating in this project.

Implemented Basic GUI →Done by 17/Apr/2023

Implemented Basic CPU Graph →Done by 27/Apr/2023

Fixed CPU Graph out of Bounds →Done by 29/Apr/2023

Made Improvements →Done by 1/May/2023

Implemented Basic List →Done by 2/May/2023

Improved CPU Graph →Done by 3/May/2023

Implemented Basic MEM Graph →Done by 3/May/2023

Implemented Grid →Done by 3/May/2023

Completed Graph Section →Done by 4/May/2023

Fixed GtkTreeIter not existing →Done by 7/May/2023

Fixed GtkTreeView & GtkTreeModel issue →Done by 10/May/2023

Implemented List Updating property → Done by 13/May/2023

Fixed Segmentation Fault (bad malloc) →Done by 14/May/2023

Thank You