UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

GUI-Based Process Management Tool

Docente:
Maria Teresa Reggio

Studenti:
Daryn Calangi, 528458
Mariachiara Trifiró, 528162

Indice

1 Introduzione

È stato implementato un applicativo che, tramite un'interfaccia *user-friendly*, consente agli utenti di monitorare e gestire efficacemente le risorse di sistema. Questo strumento offre una panoramica dettagliata delle applicazioni utente in esecuzione e dei processi attivi, fornendo informazioni essenziali sul consumo delle risorse. L'obiettivo principale di questo progetto è di aiutare gli utenti a mantenere il sistema reattivo e stabile, anche quando si trovano a gestire applicazioni che richiedono elevate risorse. Attraverso la possibilità di terminare processi e regolare l'allocazione delle risorse, gli utenti possono evitare rallentamenti e crash del sistema, assicurando un'esperienza d'uso ottimale.

2 Descrizione del problema

L'esecuzione di programmi che consumano una quantità significativa di risorse del sistema potrebbe portare ad un drastico rallentamento e, in casi estremi, al completo blocco del sistema. Processi di sistema, infatti, potrebbero così rimanere senza risorse sufficienti per funzionare correttamente e in generale questo potrebbe portare ad una riduzione delle prestazioni del sistema. Inoltre, in queste situazioni, utenti meno esperti potrebbero riscontrare difficoltà nell'identificazione di quale applicazione o processo stia causando il problema. Senza strumenti adeguati per monitorare e gestire le risorse, l'utente può non essere in grado di prendere decisioni informate su quali processi terminare o regolare per liberare risorse.

2.1 Soluzione proposta

La soluzione proposta consiste in un'applicazione GUI basata su `tkinter` che permette di:

- Visualizzare in modo ordinato e intuitivo le applicazioni in esecuzione e i processi attivi.
- Terminare i processi con privilegi amministrativi per liberare risorse.
- Aggiornare dinamicamente le informazioni sul consumo di risorse per aiutare l'utente a prendere decisioni informate in tempo reale.

3 Implementazione

3.1 Comandi

3.1.1 wmctrl

Il comando **`wmctrl`** è stato utilizzato per interrogare il gestore delle finestre del sistema. Specificando l'opzione `-l` viene visualizzato l'elenco delle finestre gestite dal *window manager*. L'output sarà quindi caratterizzato da una riga per ciascuna finestra, alla fine della quale viene specificato il titolo della di quest'ultima. Inoltre, se viene specificata l'opzione `-p` viene visualizzato il PID per la finestra come intero decimale.

3.1.2 top

Il comando **`top`** permette la visualizzazione periodica dell'elenco ordinato di processi di sistema. Questi sono ordinati in modo predefinito dal loro identificativo (PID), ma è possibile ordinarli in maniera diversa: sono infatti disponibili varie opzioni di output. Inoltre, è stata utlizzata l'ozione `-b` per eseguire il comando in `batch mode`, ovvero non interattivo in modo da poter reindirizzare l'output all'interfaccia grafica.

3.1.3 kill

Il comando **`kill`** viene utilizzato per inviare segnali ai processi in esecuzione. Il segnale più comune è `SIGTERM (15),` che richiede al processo di terminare in modo ordinato. Un altro segnale frequentemente usato, anche in questo caso specifico, è `SIGKILL (9)`, che forza la terminazione immediata del processo senza possibilità di intercettazione.

3.1.4 renice

Il comando **`renice`** è stato utilizzato per modificare la priorità di esecuzione di uno o più processi in esecuzione in base alle necessità dell'utente, influenzando quindi l'ordine e la frequenza con cui il sistema operativo pianifica questi processi. È importante notare che valori più bassi indicano una maggiore priorità.

3.2 Strumenti

L'applicativo è stato realizzato tramite il linguaggio di programmazione Python.

3.2.1 Tkinter

La libreria **Tkinter**, un modulo della Standard Library di Python, è stata utilizzata per la creazione di interfacce grafiche. In particolare, questo strumento ha permesso la realizzazione di due finestre principali con le relative funzionalità sfruttando l'ampia scelta di widget predefiniti messi a disposizione dalla libreria come etichette, campi per inserire del testo, pulsanti che eseguono azioni ed altro ancora.

3.2.2 subprocess

È stato utilizzato il modulo **`subprocess`** per l'avvio di processi di sistema. In particolare, sono state sfruttate le funzioni:

- **`run`** che accetta un comando oppure una lista di comandi, fornendo un controllo completo su stdin, stdout, stderr, timeout, variabili d'ambiente e directory di lavoro. Restituisce un oggetto CompletedProcess che contiene informazioni dettagliate sul comando eseguito, incluso il codice di uscita, l'output e l'errore.

- **`check_output`** che esegue un comando e cattura solo l'output standard. Risulta quindi essere meno flessibile rispetto a `run`, ma più semplice da usare quando si ha solo bisogno di catturare l'output.

3.3 Interfacce

Sono state realizzate due schermate rispettivamente per garantire la gestione delle finestre aperte per gli utenti meno esperti e la gestione dei processi attivi come impostazioni avanzate. Per l'implementazione delle due interfacce, sono stati realizzati due script differenti.

3.3.1 Gestione delle finestre aperte

La prima interfaccia implementata garantisce una visualizzazione grafica delle applicazioni aperte, mostrando le loro rispettive icone. In particolare, tramite la variabile globale **`PIDs`**, implementata come un dizionario, sono stati memorizzati i processi e le relative informazioni: il titolo e il percorso dell'icona dell'applicazione. Dopo aver controllato le finestre aperte eseguendo il comando **`wmctrl -l -p`** tramite la funzione `check_output` del modulo **`subprocess`**, sono stati inseriti gli identificatori dei processi come chiavi del dizionario `PIDs`, associando inizialmente una lista vuota come rispettivi valori. Tramite i PID così ricavati, sono stati controllati i nomi dei comandi che hanno avviato i processi (identificati dal loro PID) tramite i file `/proc/[PID]/comm`. Per visualizzare il contenuto di questi file è stato sfruttato il comando **`cat`** e nuovamente la funzione `check_output` del modulo **`subprocess`**. Successivamente, sulla base dei nomi trovati, sono stati cercate tutte le immagini `.png` in `/usr/share/`. Questo garantisce la visualizzazione delle icone delle applicazioni aperte, che viene fatto così in modo nativo, cioè ricercandole all'interno del sistema stesso. Questo viene implementato dalla funzione **`get_running_applications`**.

```
def get_running_applications():
    output = subprocess.check_output("wmctrl -l -p", shell=True)
    lines = output.decode().split('\n')[:-1]

    for line in lines:
        line = line.split()
        if line[2] == "0":
            continue
        elif line[2] not in PIDs:
            PIDs[line[2]] = []

    for pid in PIDs:
        if len(PIDs[pid]) == 0:
            print(pid, PIDs[pid])
            output = subprocess.check_output("cat /proc/" + str(pid) + "/comm",
            ↪  shell=True)
            name = output.decode().lower()[:-1]
            PIDs[pid].append(name)
            command = ["find", "/usr/share/", "-name", f"*{name}*.png"]
            output = subprocess.run(command, capture_output=True, text=True)
            if len(output.stdout.split('\n')) < 2:
                command = ["find", "/", "-name", f"*{name}*.png"]
                output = subprocess.run(command, capture_output=True, text=True)
            img_path = output.stdout.split('\n')
            if(len(img_path) == 1):
                PIDs[pid].append("./htop_analysis/not-found.png")
            if ("16x16" in img_path[0]):
                PIDs[pid].append(img_path[3])
            else:
                PIDs[pid].append(img_path[0])
    titles = []
    image_paths = []
    for key, value in PIDs.items():
        if len(value[1]) != 0:
            image_paths.append(value[1])
```

```
        titles.append(value[0])
```

In questo modo, tramite la funzione `on_resize`, ogni volta che la finestra creata viene ridimensionata, vengono disposte le immagini delle applicazioni utente trovate e i pulsanti per la chiusura di queste. Se l'utente clicca sul pulsante di chiusura, viene richiamata la funzione **`remove_image`**, la quale procede a terminare il processo corrispondente all'immagine e rimuovere quest'ultima dal canvas. Questa operazione ha successo solo se il comando viene lanciato con i privilegi di amministratore del sistema: la funzione, infatti, richiede la password di quest'ultimo prima di eseguire il comando. Più precisamente, viene specificata l'opzione `-k` nel comando `sudo`, in modo tale da invalidare il *timestamp* di autenticazione corrente. Questo è stato fatto cosicché anche se l'utente si è precedentemente autenticato, e il periodo di timeout non è scaduto, viene richiesta una nuova immissione della password.

```
def run_sudo_command(command):
    password = ask_sudo_password()
    if password:
        try:
            result = subprocess.run(f'echo {password} | sudo -S -k {command}',
            ↪  shell=True, check=True, text=True, capture_output=True)
            if result.returncode == 0:
                messagebox.showinfo("Success", "Application termineted
                ↪  successfully")
                return True
            else:
                return False
        except subprocess.CalledProcessError as e:
            messagebox.showerror("Error", f"Failed to kill application\n{e.stderr}")
            return False
    else:
        messagebox.showerror("Error", "Password not provided")
        return False
```

3.3.2 Gestione dei processi attivi

La finestra relativa alla gestione dei processi attivi consente agli utenti di visualizzare i processi attualmente in esecuzione, terminare i processi selezionati e modificare il valore di *nice* dei processi. L'interfaccia offre anche funzionalità di filtro per cercare processi specifici in base a vari criteri.

La funzione **`create_widgets`** permette la visualizzazione dei dettagli dei processi. Più precisamente, all'interno della finestra creata è presente un widget `Treeview`, che è un tipo di tabella che viene configurato con diverse colonne per visualizzare informazioni strutturate. Queste colonne sono: PID, User, Name, CPU%, MEM%, Status, Nice. Sono stati inseriti anche i seguenti pulsanti:

1. **Kill Process** per terminare i processi;
2. **Change Nice Value** per modificare il valore di nice dei processi;
3. **Refresh** per aggiornare la lista;
4. **Filter** per applicare filtri.

Per garantire l'aggiornamento di questa lista, è stata implementata la funzione **`refresh_process_list`**, la quale procede ad eliminare tutti gli elementi dalla visualizzazione ad albero e dopo averli recuperati utilizzando `psutil.process_iter`, li riordina per utilizzo della CPU. La manipolazione sui processi può essere effettuata da parte dell'utente sulla base della selezione di quest'ultimo ad una determinata riga della lista. Questo viene implementato dalla funzione **`on_tree_select`**, la quale determina se l'utente ha

selezionato un processo e verifica se l'utente corrente ha i privilegi necessari per gestirlo (quelli di amministratore). Per garantire una maggiore sicurezza, inoltre, vengono di default disabilitati i pulsanti per i processi che non dovrebbero essere gestiti dall'utente. Le funzionalità di terminazione e modifica del valore di nice di un processo selezionato dall'utente vengono implementate rispettivamente tramite le funzioni **`kill_process`** e **`change_nice_value`**. Entrambe le operazioni hanno successo se l'utente ha i privilegi necessari per eseguire i comandi. La gestione della richiesta della password di amministratore e dell'esecuzione dei comandi vengono realizzatre rispettivamente tramite le funzioni **`ask_for_admin_password`** e **`run_as_admin_unix`**. Inoltre, dato che il valore di priorità che si vuole assegnare ad un determinato processo può variare da -20 (massima priorità) a 19 (minima priorità), viene creata una finestra di dialogo tramite `simpledialog.askinteger` che chiede all'utente di inserire un numero intero. Infine, la possibilità di filtrare i processi in base a criteri specifici, come nome, utilizzo della CPU, utilizzo della memoria e PID è stata realizzata tramite la funzione **`filter_process_list`**.

```
def filter_process_list(self):
    criteria = simpledialog.askstring("Filter Criteria", "Enter filter criteria
    ↪  (e.g., name:<process_name>, cpu:<cpu_percent>, memory:<memory_percent>):")
    if criteria:
        key, value = criteria.split(':')
        filtered_processes = []
        for proc in self.processes:
            try:
                print(key, value.lower())
                if (key == 'name' or key == 'username' or key == 'status') and
                ↪  value.lower() in proc.info[key].lower():
                    filtered_processes.append(proc)
                elif (key == 'cpu' or key == 'memory') and proc.info[key +
                ↪  "_percent"] >= float(value):
                    filtered_processes.append(proc)
                elif key == 'pid' and proc.info[key] < int(value):
                    filtered_processes.append(proc)
            except (psutil.NoSuchProcess, psutil.AccessDenied,
            ↪  psutil.ZombieProcess):
                pass
```