R1 - Mini Proyecto - Monitoreo de Procesos del SO

Curso:
Sistemas Operativos

Profesor:
Ing. Espinoza Urzua Eden

Institución:
Universidad Nacional Autónoma de México
Facultad de ingeniería

Fecha de entrega:
1 de Octubre de 2025

Equipo SOS - Sistemas Operativos - 2026-1

Miembros del equipo:

Nombre	Rol
Espinosa Guzmán Magali Lizeth	Diseñador UX
García Vázquez Javier Alejandro	Desarrollador de Software

ÍNDICE

OBJETIVO

Analizar los procesos en ejecución en su computadora real.

DESCRIPCIÓN

- Usar la librería psutil en Python para obtener la lista de procesos activos en el sistema.
- Mostrar información como PID, nombre del proceso, uso de CPU y memoria.
- Simular un cambio de contexto mostrando qué procesos están en ejecución y cuáles en espera.

DISEÑO DE LA ARQUITECTURA

Componente	Responsabilidad	Dependencias
Módulo Principal	Inicializara el entorno, se encarga del bucle principal de actualización y se encarga de recibir la señal de alto para el programa (Ctrl+C).	monitorear_procesos, time
psutil	API del Sistema Operativo: obtiene la información de los procesos (PID, nombre, uso de CPU, estado, memoria en MB).	N/A
Módulo de Mapeo de Estados (ESTADOS_SIMULADOS)	Define la lógica para traducir el estado interno del SO (running, sleeping, etc.) a la etiqueta para el usuario propuesta (Ejecutándose, Listo, etc.).	N/A
Proceso Único (Iterador)	Extrae, formatea y muestra la información de un proceso individual. Contiene la lógica de manejo de errores (NoSuchPRocess, AccessDenied).	psutil
Salida (Sobre consola)	Muestra los datos de los procesos en formato tabular y realiza la limpieza de pantalla (`\033[H\033[J`).	N/A

APLICACIÓN DE CONCEPTOS DE SISTEMAS OPERATIVOS EN EL CÓDIGO

¿Por qué Python?

Python es un lenguaje de programación de alto nivel, interpretado y de código abierto. Destaca de entre otros lenguajes por su sintaxis sencilla y legibilidad. Lenguaje de propósito general, versátil y muy utilizado en desarrollo web y de software, ciencia de datos, aprendizaje automático y más. Pero, además de su amplia comunidad y su capacidad de ejecutarse en diversas plataformas, este lenguaje posee bibliotecas extensas, una de ellas (y es la que usamos en este mini-proyecto) es psutil.

Uso de la biblioteca psutil

La biblioteca psutil es una herramienta muy útil para un programador cuyo objetivo es análisis y gestión sobre el sistema de computo en que trabaja, pues posee funcionalidades como las siguientes:

- Monitorización del sistema.
- Gestión de procesos.
- Información de red.
- Uso de sensores

Por ello, se utiliza para:

- Desarrollo de herramientas de monitorización.
- Optimización y perfilado.
- Automatización de tareas.
- Desarrollo de aplicaciones de seguridad.

Y es para eso precisamente que se realiza este mini-proyecto. Para realizar un estudio sobre los recursos a los que tiene acceso un Sistema Operativo, tales como:

- **CPU:** Uso de la misma, estadísticas por núcleo y promedios de carga.
- **Memoria:** Uso de la memoria física (RAM) y de la memoria virtual.
- **Discos:** Estadísticas de E/S de disco y uso del espacio en disco.
- **Red:** Información de conexiones de red y estadísticas de red.
- **Procesos:** Listado de todos los procesos en ejecución, sus IDS (PID), nombres, uso de recursos y más. Esta es la razón de utilidad de la biblioteca que nos interesa.
- **Sensores:** Lectura de información de sensores como la temperatura (si está disponible).

Para utilizar psutil es necesario instalarlo con pip: *pip install psutil*. Una vez instalado, ya podemos implementarlo a cualquier trabajo que se requiera. En este caso es un paso muy importante para la elaboración de este proyecto.

CONCEPTOS DE CLASE

El programa que hemos desarrollado en Python, utilizando las librerías psutil y tkinter, permite crear un monitor de procesos que refleja de manera práctica varios conceptos teóricos que hemos estudiado en clase. En este monitoreo, podemos distinguir entre dos tipos de conceptos: aquellos que el código programa y muestra de forma explícita, y aquellos que aunque no se implementan directamente, se visualizan de manera implícita como resultado de las funciones internas del sistema operativo.

1. Conceptos cubiertos explícitamente en el código
    a. **Identificación de procesos (PID y nombre):** Cada proceso listado incluye su número de identificación y el nombre asociado.
    b. **Uso de CPU por proceso:** Es obtenido mediante cpu_percent y se presenta en la tabla.
    c. **Uso de memoria por proceso:** Se calcula en MB a partir de memory_info.rss.
    d. **Estados de los procesos:** Se traducen estados internos a etiquetas más comprensibles: Ejecutándose, Listo (en espera), Bloqueado (esperando I/O) y Zombie.
    e. **Monitoreo global de recursos:** Además de la información por proceso, se muestran el uso total de CPU y el uso global de memoria mediante barras de progreso.

2. Conceptos implícitos que se visualizan
    a. **Cambio de contexto:** Que los porcentajes de CPU varien entre procesos en cada actualización, demuestra que el sistema operativo está llevando a cabo cambios de contexto de manera continua, alternando la ejecución entre diferentes procesos.
    b. **Planificación de procesos:** La rotación en el uso de CPU indica que el planificador del sistema está determinando la entrada y salida de procesos en ejecución según un algoritmo interno.
    c. **Gestión de estados:** Si bien, el monitor no establece las transiciones de estado, la columna “Estado” muestra el resultado de dichas decisiones, confirmando cómo el sistema operativo administra los procesos según su situación actual.
    d. **Contabilidad de recursos:** el sistema registra cuánta CPU y memoria utiliza cada proceso. Esa información la presenta el monitor al usuario de forma accesible y visual.
    e. **Procesos terminados (zombies):** Cuando se visualizan procesos en estado Zombie, se puede observar cómo el sistema mantiene temporalmente la información de procesos finalizados, en espera de que su estado sea recogido por el proceso padre.
    f. **Hilos (threads):** Si bien, el programa no los lista directamente, cada proceso puede estar formado por múltiples hilos, que, como vimos en clase, son las unidades reales de ejecución dentro de la CPU. El uso de la CPU mostrado por proceso incluye la suma del trabajo de sus hilos, y el hecho de que varios procesos muestren un uso alternado de la CPU también indica la planificación y el cambio de contexto de los hilos que los componen.

Asimismo, gracias a la librería psutil, sería posible consultar esa información utilizando funciones como proc.num_threads() o proc.threads().

REQUERIMIENTOS

La clave en este proyecto es el uso correcto de la librería psutil de Python para obtener la información de los procesos de la computadora donde estaremos ejecutando el programa. Su salida tiene que verse de la siguiente manera:

PID: 1234 | Proceso: chrome.exe | Estado: Ejecutándose
PID: 5678 | Proceso: explorer.exe | Estado: Listo
PID: 9101 | Proceso: spotify.exe | Estado: Bloqueado (esperando I/O)

IMPLEMENTACIÓN

- Para obtener la lista de procesos, hay que usar psutil.process_iter()
- Cada proceso accederá a sus atributos con métodos con nombres clave para la tabla, los que se van a manejar serán los siguientes (dado los nombres en el sistema): pid, name, uso_cpu, mem_info.
- La parte de simulación de los estados del proceso (Ejecutándose, Listo y Bloqueado) se va a valer de la consulta directa al kernel del SO para obtener el estado actual del proceso. Las etiquetas estándar son running, sleeping, waiting y zombie.
  NOTA: Estas etiquetas las vamos a pasar por un traductor al que vamos a llamar para que use las etiquetas que el ejemplo de la salida nos propone y que ya mencionamos.

MONITOREO DE PROCESOS EN CONSOLA

```
# ======================================================================
# Monitoreo de Procesos del SO (Usando Python y psutil)
# Modificación: Si uso_cpu es 0.00%, el estado debe ser 'Listo (en espera)'.
# ======================================================================

# Solicitamos las librerías que necesitamos
import psutil  # Librería fundamental para interactuar, monitorear y gestionar los procesos activos del sistema.
import time    # Librería para introducir pausas en el script (simular el refresco de monitoreo).

# 1. Definimos el mapeo de estados como una variable global (o constante)
# Traduce los estados internos de psutil (que reflejan el SO) a un formato legible
ESTADOS_SIMULADOS = {
    'running': 'Ejecutándose',             # Proceso usando la CPU activamente.
        'sleeping': 'Listo (esn espera)',            # Estado por defecto cuando el SO lo reporta como "durmiendo".
    'waiting': 'Bloqueado (esperando I/O)', # Proceso esperando una operación de Entrada/Salida.
     'zombie': 'Zombie (terminado)'           # Proceso terminado, pero que aún tiene una entrada en la tabla de procesos.
    # Nota: El estado 'Listo (en espera)' también se aplicará por lógica de CPU 0.00%
}


def obtener_y_mostrar_proceso(proc):
    """
    Función: obtener_y_mostrar_proceso
     Responsabilidad: Extrae la información detallada de un proceso individual y aplica la nueva lógica de estado.
    """
    try:
         # Intenta obtener la información del proceso. Esto puede fallar si el proceso termina mientras se lee.

        # Extracción de campos obligatorios
        pid = proc.info['pid']
        nombre = proc.info['name']

        # MANEJO DE USO DE CPU
        uso_cpu_raw = proc.info['cpu_percent']
        uso_cpu = uso_cpu_raw if uso_cpu_raw is not None else 0.0

        # MANEJO DE USO DE MEMORIA (en MB)
        mem_info = proc.info['memory_info']
        if mem_info is not None:
            # Convierte el Resident Set Size (RSS, memoria física usada) de bytes a Megabytes.
            uso_memoria = mem_info.rss / (1024 * 1024)
        else:
            uso_memoria = 0.0

        # ============================================================
```

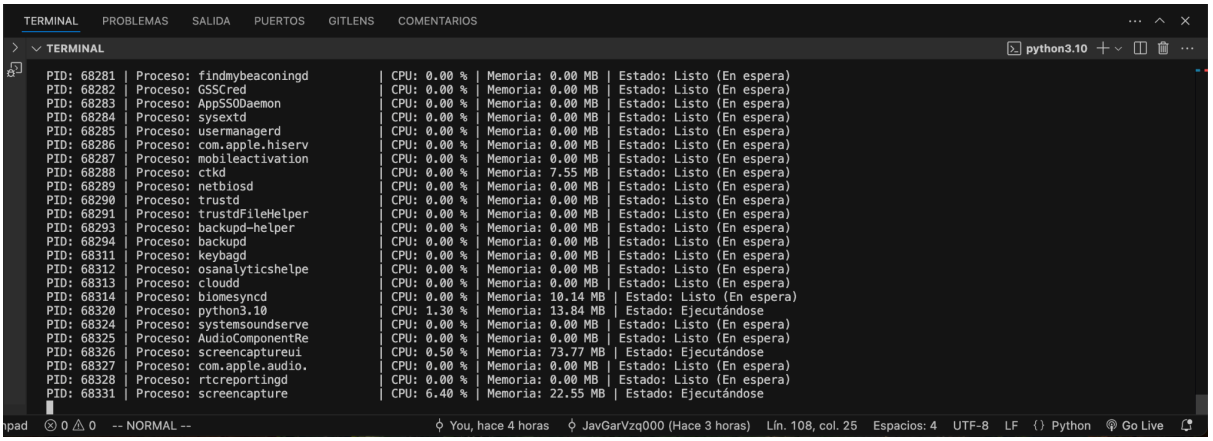
```
        # LÓGICA DE SIMULACIÓN Y ASIGNACIÓN DE ESTADO MODIFICADA
        # ============================================================

        estado_real = proc.status()

        # Primero, verifica si el proceso ha terminado. Si es zombie, no hay más lógica.
        if estado_real == 'zombie':
            estado_simulado = ESTADOS_SIMULADOS['zombie']

        # Segundo, aplica la regla de correlación: Si el uso de CPU es cero,
        # implica que el proceso está esperando y se encuentra en el estado 'Listo'
        elif uso_cpu == 0.0:
            estado_simulado = 'Listo (En espera)'

        # Tercero, si no es Zombie y sí está usando CPU, usa el mapeo original.
        else:
            # Mapea el estado real del SO (Ejecutándose, Bloqueado, etc.)
            estado_simulado = ESTADOS_SIMULADOS.get(estado_real, estado_real)

        # Mostrar la información formateada
                print(f"PID: {pid:<5} | Proceso: {nombre:<25} | CPU: {uso_cpu:<5.2f}% | Memoria:
{uso_memoria:.2f} MB | Estado: {estado_simulado}")

    # Manejo de excepciones
    except (psutil.NoSuchProcess, psutil.AccessDenied):
        # Captura errores si un proceso termina o si Python no tiene permisos para leerlo.
        # En estos casos, simplemente omitimos el proceso y continuamos con el siguiente.
        pass
    except Exception:
        # Captura cualquier otro error imprevisto para evitar que el script se detenga.
        pass


def monitorear_procesos():
    """
    Función: monitorear_procesos (Función principal)
    Responsabilidad: Contiene el bucle principal, gestiona la limpieza de pantalla y la pausa.
    """
    print("Monitoreo de Procesos (Presione Ctrl+C para salir)\n")

    try:
        # Bucle infinito para el monitoreo continuo
        while True:
            # Limpiamos la pantalla
            print("\033[H\033[J")

            # Iteramos sobre todos los procesos activos
            # Pasamos una lista de atributos para optimizar la llamada y obtener solo lo necesario.
            for proc in psutil.process_iter(['pid', 'name', 'cpu_percent', 'memory_info']):
                # Llamamos a la función de detalle para procesar y mostrar cada proceso.
                obtener_y_mostrar_proceso(proc)
```

```
            # Esperamos 2 segundos
            time.sleep(2)

    except KeyboardInterrupt:
        # Manejo de la interrupción del usuario (Ctrl+C) para una salida limpia.
        print("\nMonitoreo detenido.")


# Punto de entrada estándar de Python
if __name__ == "__main__":
    # Ejecuta la función principal del monitoreo.
    monitorear_procesos()
```

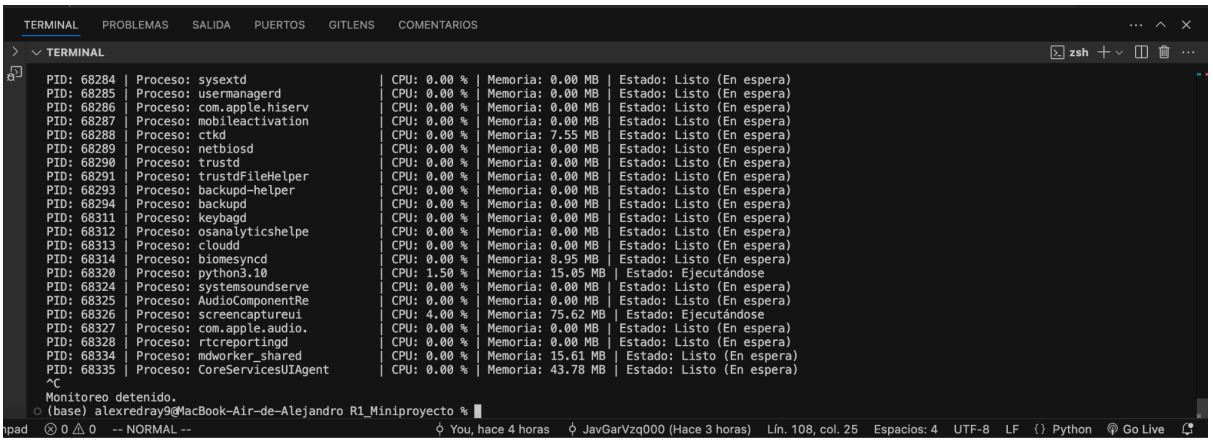
FUNCIONALIDAD DEL PROGRAMA SOBRE CONSOLA

COMANDO PARA EJECUTARLO: % python MiniProyecto_R3.py

PROGRAMA EN EJECUCIÓN



PRESIONANDO CTRL+C PARA DETENER LA EJECUCIÓN



MONITOREO DE PROCESOS CON INTERFAZ GRÁFICA

```
# ========================================================================
# Monitoreo de Procesos del SO (Usando Python y psutil)
# Si uso_cpu es 0.00%, el estado debe ser 'Listo (en espera)'.
# ========================================================================

# Solicitamos las librerías que necesitamos
import psutil  # Librería para interactuar, monitorear y gestionar los procesos
import time
import tkinter as tk  # Librería principal para la interfaz gráfica.
from tkinter import ttk # Usada para la tabla (Treeview) y Progressbar.

# Variables globales para la GUI/Interfaz gráfica
app_window = None # Ventana principal de la aplicación.
tree_view = None    # Tabla para mostrar los procesos.
status_label = None  # Etiqueta para mostrar la última actualización.
etiqueta_cpu = None      # Etiqueta para el texto de uso de CPU.
barra_cpu = None        # Barra de progreso de CPU.
etiqueta_mem = None      # Etiqueta para el texto de uso de Memoria.
barra_mem = None        # Barra de progreso de Memoria.


#Definimos el mapeo de estados como una variable global (o constante) para traducir los estados internod de psutil (que reflejan el SO) a un formato legible.

ESTADOS_SIMULADOS = {
    'running': 'Ejecutándose',              # Proceso usando la CPU activamente.
    'sleeping': 'Listo (en espera)',       # Estado por defecto cuando el SO lo reporta como "durmiendo".
    'waiting': 'Bloqueado (esperando I/O)', # Proceso esperando una operación de Entrada/Salida.
     'zombie': 'Zombie (terminado)'            # Proceso terminado, pero que aún tiene una entrada en la tabla de procesos.
    # Nota: El estado 'Listo (en espera)' también se aplicará por lógica de CPU 0.00%
}


# ========================================================================
# Funcionalidad del Monitor de procesos
# ========================================================================

def obtener_y_mostrar_proceso(proc):
    # Esta función extrae los datos de un proceso individual, realiza conversiones (bytes a MB)
    # y aplica la lógica de simulación de estado para devolver una tupla lista para la GUI.

    try:
        # Extracción de campos obligatorios. pid es el ID del proceso, name es el nombre del proceso.
        pid = proc.info['pid']
        nombre = proc.info['name']

        # MANEJO DE USO DE CPU
        uso_cpu_raw = proc.info['cpu_percent']
        uso_cpu = uso_cpu_raw if uso_cpu_raw is not None else 0.0
```

```
        # MANEJO DE USO DE MEMORIA (en MB)
        mem_info = proc.info['memory_info']
        if mem_info is not None:
            uso_memoria = mem_info.rss / (1024 * 1024)
          # Convertir bytes a MB debido a que rss está en bytes y se requiere convertir a MB. Resident Set Size (RSS) es la cantidad de memoria física (RAM) que un proceso está usando.

        else:
            uso_memoria = 0.0

        # LÓGICA DE ESTADO
        estado_real = proc.status()

        if estado_real == 'zombie':
            estado_simulado = ESTADOS_SIMULADOS['zombie']
        elif uso_cpu == 0.0:
            estado_simulado = 'Listo (En espera)'
        else:
            estado_simulado = ESTADOS_SIMULADOS.get(estado_real, estado_real)


        # INTERFAZ: se devuelven los datos para la inserción en la tabla.
        return (
            str(pid),
            nombre,
            f"{uso_cpu:.2f}%",
            f"{uso_memoria:.2f} MB",
            estado_simulado
        )


    # Manejo de excepciones
    except (psutil.NoSuchProcess, psutil.AccessDenied, Exception):
        return None # Devuelve None si hay error


def actualizar_tabla():
    # Función principal que actualiza la tabla de procesos y las barras de progreso de acuerdo con la información de la función anterior obtener_y_mostrar_proceso. Actualiza el Treeview, las barras y la hora.


    global tree_view, app_window, status_label
    global etiqueta_cpu, barra_cpu, etiqueta_mem, barra_mem


    #Limpiar la tabla de procesos anteriores
    for i in tree_view.get_children():
        tree_view.delete(i)
```

```
    ## LÓGICA DE PROCESOS: Inicia la iteración sobre todos los procesos activos del sistema. proc representa cada proceso individual y se itera en la función psutil.process_iter.
    for proc in psutil.process_iter(['pid', 'name', 'cpu_percent', 'memory_info']):
        datos_proceso = obtener_y_mostrar_proceso(proc)
        if datos_proceso:
            # Insertar los datos del proceso en la tabla
            tree_view.insert('', tk.END, values=datos_proceso)

    # Barras de progreso y etiquetas de CPU y Memoria
    # Uso de CPU (global)
    cpu_percent = psutil.cpu_percent(interval=None)
    etiqueta_cpu.config(text=f"Uso de CPU: {cpu_percent:.1f}%")
    barra_cpu.config(value=cpu_percent)

    # Uso de Memoria (global). Obtiene el objeto de información de memoria virtual
    virtual_mem = psutil.virtual_memory()
    mem_percent = virtual_mem.percent
    #Convierte los valores de bytes a Gigabytes (GB) para la etiqueta.
    mem_used_gb = virtual_mem.used / (1024 ** 3)
    mem_total_gb = virtual_mem.total / (1024 ** 3)

    # Actualizar la etiqueta y barra de memoria
        etiqueta_mem.config(text=f"Uso de Memoria: {mem_percent:.1f}% ({mem_used_gb:.2f} GB de {mem_total_gb:.2f} GB)")
    barra_mem.config(value=mem_percent)

    # Actualizar la hora de la última actualización cada 2 segundos
    current_time = time.strftime("%H:%M:%S")
    if status_label:
        status_label.config(text=f"Última actualización: {current_time} | Intervalo: 2 segundos")

    #Programar la siguiente actualización automática cda 2 segundos (2000 ms)
    app_window.after(2000, actualizar_tabla)

# ==============================================================================
# Construcción de la interfaz gráfica
# ==============================================================================

# INTERFAZ: Estilo personalizado para las barras de progreso
def configurar_estilos():
    style = ttk.Style()

    # Estilo base para las barras
        style.configure('CPU.Horizontal.TProgressbar', troughcolor='#E0E0E0', background='#2ecc71', thickness=15)

    # Estilo para la barra de Memoria
        style.configure('MEM.Horizontal.TProgressbar', troughcolor='#E0E0E0', background='#2ecc71', thickness=15)
```

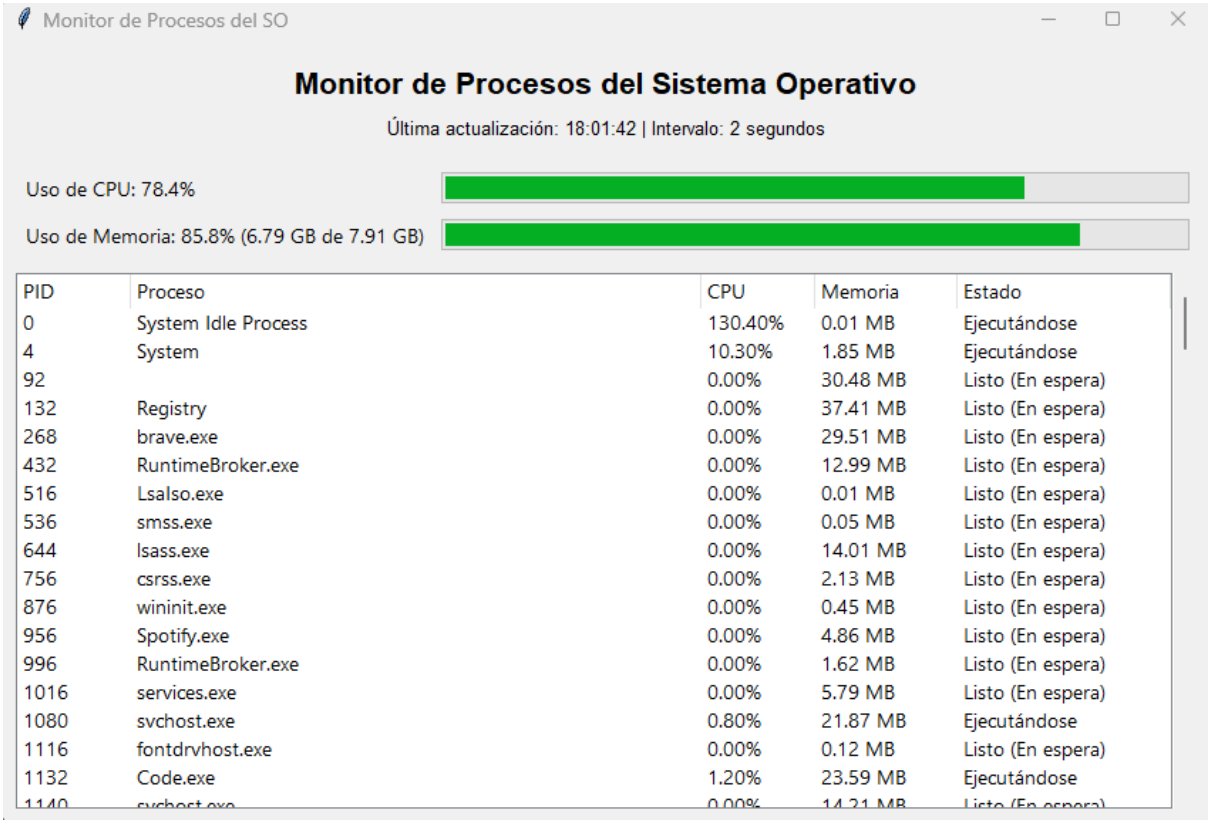
```
def monitorear_procesos():
    # Esta función inicializa todos los elementos gráficos de la GUI (ventana, barras de CPU/Memoria, y la tabla de procesos).
    # Solo crea el esqueleto visual; la actualización de datos ocurre en la función 'actualizar_datos'.

    global app_window, tree_view, status_label
    global etiqueta_cpu, barra_cpu, etiqueta_mem, barra_mem

    # Ventana Principal
    app_window = tk.Tk()
    app_window.title("Monitor de Procesos del SO")
    app_window.geometry("850x550")

    # Llamada a la función que configura el estilo personalizado para las barras
    configurar_estilos()

    # Título y Estado (Top)
    top_frame = tk.Frame(app_window)
    top_frame.pack(pady=10, fill='x')

    title_label = tk.Label(top_frame, text="Monitor de Procesos del Sistema Operativo", font=("Arial", 16, "bold"))
    title_label.pack(side=tk.TOP, pady=5)

    status_label = tk.Label(top_frame, text="Última actualización: --:--:-- | Intervalo: 2 segundos", font=("Arial", 10))
    status_label.pack(side=tk.TOP, pady=(0, 5))


    #  Barras Horizontales Superiores
    bars_container_frame = tk.Frame(app_window)
    # Este contenedor ahora ocupará todo el ancho para las barras horizontales.
    bars_container_frame.pack(fill='x', padx=15, pady=(5, 15))

    #  Barra de CPU
    cpu_frame = tk.Frame(bars_container_frame)
    cpu_frame.pack(side=tk.TOP, fill='x', pady=(0, 5))

    # Etiqueta de CPU
    etiqueta_cpu = ttk.Label(cpu_frame, text="Uso de CPU: --.-%", width=35)
    etiqueta_cpu.pack(side=tk.LEFT, anchor='w')

    # Barra de Progreso de CPU
    barra_cpu = ttk.Progressbar(cpu_frame, style='CPU.Horizontal.TProgressbar',
                                orient='horizontal', mode='determinate')
    # Expand=True permite que la barra llene el espacio restante
    barra_cpu.pack(side=tk.LEFT, fill='x', expand=True, padx=(10, 0))

    #  Barra de Memoria
    mem_frame = tk.Frame(bars_container_frame)
    mem_frame.pack(side=tk.TOP, fill='x', pady=(5, 0))
```

```
    # Etiqueta de Memoria
    etiqueta_mem = ttk.Label(mem_frame, text="Uso de Memoria: --.-%", width=35)
    etiqueta_mem.pack(side=tk.LEFT, anchor='w')

    # Barra de Progreso de Memoria
    barra_mem = ttk.Progressbar(mem_frame, style='MEM.Horizontal.TProgressbar',
                                orient='horizontal', mode='determinate')
    barra_mem.pack(side=tk.LEFT, fill='x', expand=True, padx=(10, 0))


    # Tabla de Procesos (Bottom)
    table_frame = tk.Frame(app_window)
    table_frame.pack(expand=True, fill='both', padx=10, pady=(0, 10))

    # Definición de Columnas para el Treeview (Tabla)
    columnas = ('PID', 'Proceso', 'CPU', 'Memoria', 'Estado')
    tree_view = ttk.Treeview(table_frame, columns=columnas, show='headings')

    # Configuración del encabezado y ancho de las columnas
    tree_view.heading('PID', text='PID', anchor=tk.W)
    tree_view.heading('Proceso', text='Proceso', anchor=tk.W)
    tree_view.heading('CPU', text='CPU', anchor=tk.W)
    tree_view.heading('Memoria', text='Memoria', anchor=tk.W)
    tree_view.heading('Estado', text='Estado', anchor=tk.W)

    tree_view.column('PID', width=80, stretch=tk.NO)
    tree_view.column('Proceso', width=350, stretch=tk.YES)
    tree_view.column('CPU', width=80, stretch=tk.NO)
    tree_view.column('Memoria', width=100, stretch=tk.NO)
    tree_view.column('Estado', width=150, stretch=tk.NO)

    #  Agregar Scrollbar
    scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=tree_view.yview)
    tree_view.configure(yscrollcommand=scrollbar.set)
    scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
    tree_view.pack(expand=True, fill='both')

    #  Iniciar el ciclo de actualización y el bucle principal de Tkinter
    actualizar_tabla()
    app_window.mainloop()


# Punto de entrada estándar de Python
if __name__ == "__main__":
    monitorear_procesos()
```

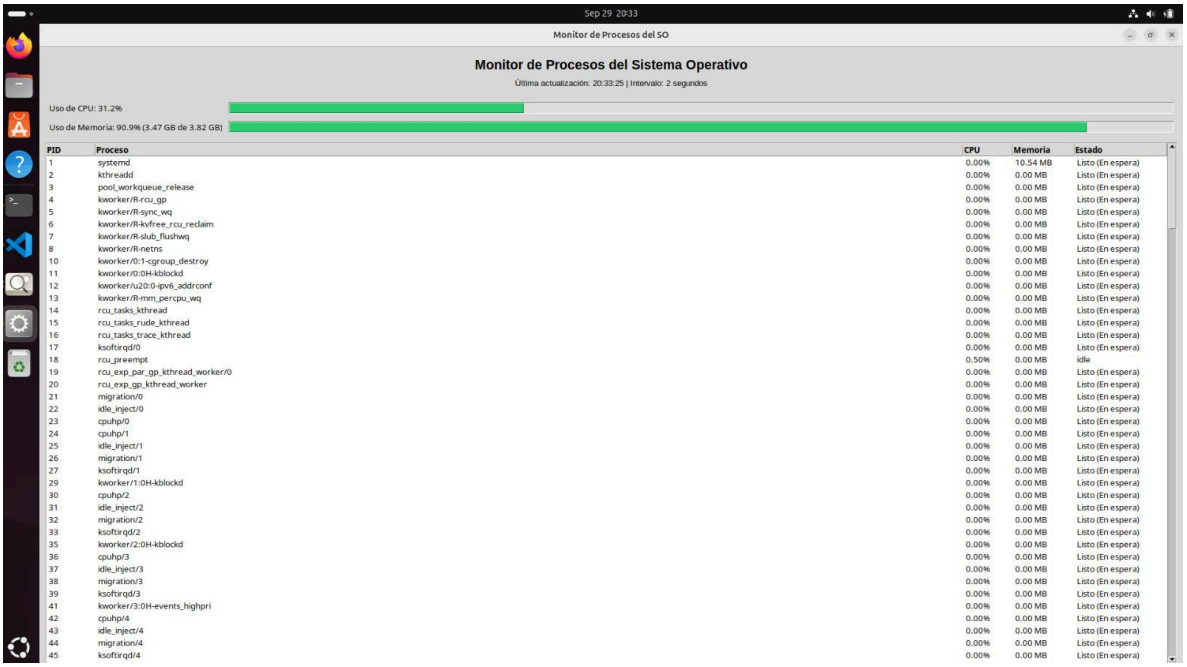
Como uno de los elementos extras requeridos del proyecto, se solicitó que el programa contara con una interfaz gráfica para poder observar de forma más intuitiva el funcionamiento del monitor de procesos del sistema operativo en Windows y Linux. Para lograr contar con una interfaz se utilizaron las librerías de python:

- tkinter: Una librería de python que da la posibilidad de mostrar una ventana principal y widgets básicos como tk.Label y tk.Frame.
- tkinter.ttk: Proporciona widgets con estilo moderno como Treeview (la tabla) y Progressbar (las barras).
- time: Utilizada para obtener la hora actual (time.strftime) y mostrar la hora de la última actualización.

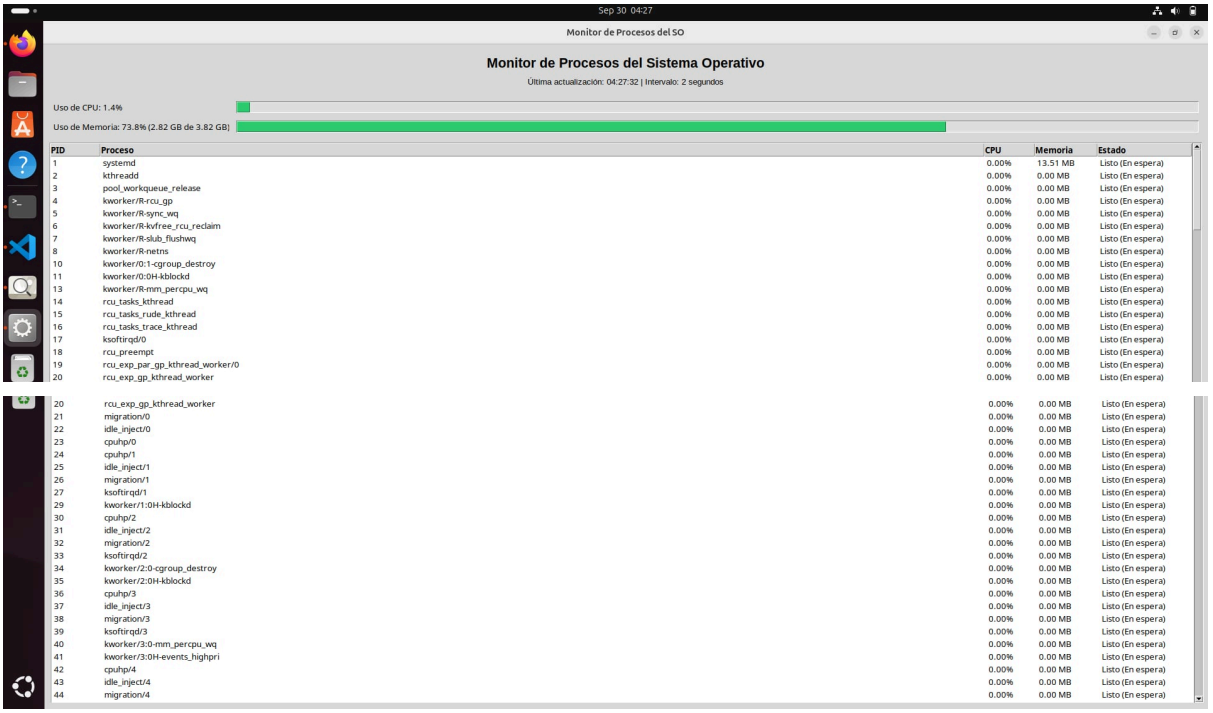
COMPARACIÓN DE LA FUNCIONALIDAD EN WINDOWS VS LINUX.



Windows



Ubuntu en Máquina Virtual



De manera complementaria al desarrollo del proyecto, se verificó la correcta ejecución del programa en el sistema operativo **Ubuntu** empleando una máquina virtual configurada en **VirtualBox**. Para ello, se procedió a la instalación de **Visual Studio Code** y de la biblioteca **psutil**, garantizando así el funcionamiento del código en dicho entorno. Asimismo, esta práctica permitió adquirir conocimientos sobre el uso de máquinas virtuales como alternativa a la partición del disco, reconociendo su utilidad en la implementación y prueba de software en diferentes plataformas.

CONCLUSIONES

Espinosa Guzmán Magali Lizeth

Este proyecto tuvo el objetivo de analizar los procesos en ejecución de una computadora en tiempo real simulando el funcionamiento del Administrador de Tareas de un sistema operativo. Para su construcción fue indispensable contar con conocimiento de conceptos como procesos, hilos así como los cambios de contexto. La herramienta que se utilizó fue el lenguaje de programación Python con la biblioteca psutil, la cual nos permitió hacer un monitoreo de métricas del sistema operativo como procesos, memoria, CPU, etc.

De forma particular, me concentré en la implementación de una interfaz de usuario que trabajara bien con la lógica del programa, para esto utilicé las librerías tkinter y tkinter.ttk que permitió al programa funcionar fuera de consola y crear un widget con una tabla que recibía la información de la parte lógica y se actualizaba cada dos segundos; asimismo, se logró mostrar dos barras que mostraron el uso global de memoria y CPU.

García Vázquez Javier Alejandro

Con la elaboración de este mini-proyecto logré ver de forma práctica los conceptos vistos en clase. Tener acceso a la información de los procesos que se van realizando en tiempo real, empleando herramientas fáciles de usar como lo es la biblioteca psutil de Python. Así como la forma en que se puede dar solución a un problema propuesto. Desde un punto de vista técnico, este proyecto permitió:

1. Monitorear procesos concurrentes.
2. Simular el Cambio de Contexto (Estados). Con una lógica de mapeo de estados (running, waiting, sleeping, etc.). Así como el uso de una simulación al correlacionar directamente el uso de CPU del 0.0% con el estado de Listo (En espera), reflejando de forma práctica cómo los procesos pasan a la cola de listos, después de pasar por el procesador.
3. Manejo de Recursos. Vimos métricas fundamentales como el uso de CPU y memoria, para entender la planificación de procesos y administración de recursos del sistema.

Por lo anterior, no solo apliqué conocimientos, también visualicé el ciclo de vida de un proceso.