Universidad Nacional Autónoma de México

Facultad de Ingeniería

Sistemas Operativos

Proyecto Final

“Simulador de Sistema Operativo Básico”

Profesor: José Albero Avalos Vélez

Grupo: 02

Integrantes:

Flores Torres Ángel Joel

Morales Soto Fernando

Muciño Hernández Zara Paulina

Hernández Hernández Oscar Mario

Fecha de entrega: 20 de noviembre de 2024

Semestre: 2025-1

Índice

Introducción

Un sistema operativo es el software base que gestiona los recursos de hardware y software de un sistema informático. Este proyecto tiene como objetivo simular tres de sus componentes fundamentales: la planificación de procesos, la administración de memoria y el sistema de archivos. Con este simulador, se busca facilitar el aprendizaje práctico y la comprensión de cómo estas funcionalidades trabajan juntas para garantizar el rendimiento y la estabilidad del sistema. La combinación de una interfaz gráfica intuitiva y una implementación lógica sólida proporciona un entorno educativo ideal para explorar los conceptos básicos de los sistemas operativos.

Objetivos

- Comprender y aplicar los conceptos de planificación de procesos.
- Implementar un sistema básico de administración de memoria.
- Simular un sistema de archivos simple con comandos básicos de manipulación de archivos y directorios.

Componentes del proyecto

1. **Planificación de Procesos**
    - Implementar varios algoritmos de planificación como FIFO, Round Robin, SJF (Shortest Job First).
    - El sistema debe permitir al usuario crear "procesos" con distintos tiempos de ejecución y prioridades, y mostrar el orden de ejecución según el algoritmo elegido.
    - Incluir estadísticas como el tiempo promedio de espera y tiempo de retorno para analizar la eficiencia de cada algoritmo.

2. **Administración de Memoria**
    - Simular un esquema de paginación o segmentación en memoria, donde los procesos se carguen en "bloques" o "segmentos".

- Implementar algoritmos de reemplazo de páginas, como FIFO o LRU (Least Recently Used), para gestionar el reemplazo en caso de fallos de página.
- Mostrar la tabla de páginas o el estado de la memoria tras cada carga de proceso.

3. **Sistema de Archivos**
    - Crear un sistema de archivos básico que permita crear, leer, y eliminar archivos y directorios.
    - Implementar comandos como mkdir, touch, rm, y ls, para que el usuario pueda manipular los archivos en el sistema.
    - Cada archivo puede contener texto, permitiendo operaciones de lectura y escritura en el mismo.

Desarrollo

El proyecto se divide en dos archivos principales:

1. **simulador.py**: Contiene toda la lógica de backend que implementa las funcionalidades del sistema operativo.
2. **interfaz.py**: Proporciona una interfaz gráfica de usuario (GUI) mediante Tkinter para interactuar con las funcionalidades del simulador.

Descripción del Archivo simulador.py

Planificación de Procesos

La clase ProcessScheduler es responsable de gestionar los procesos y aplicar diferentes algoritmos de planificación:

- **FIFO (First In, First Out)**:
    - Los procesos se ejecutan en el orden en que llegan a la cola.
    - Este algoritmo es simple y fácil de implementar, pero puede sufrir del problema conocido como "Convoy Effect".

- **SJF (Shortest Job First)**:
    - Los procesos con menor tiempo de ráfaga son priorizados.
    - Minimiza el tiempo promedio de espera, pero puede llevar a inanición de procesos largos.

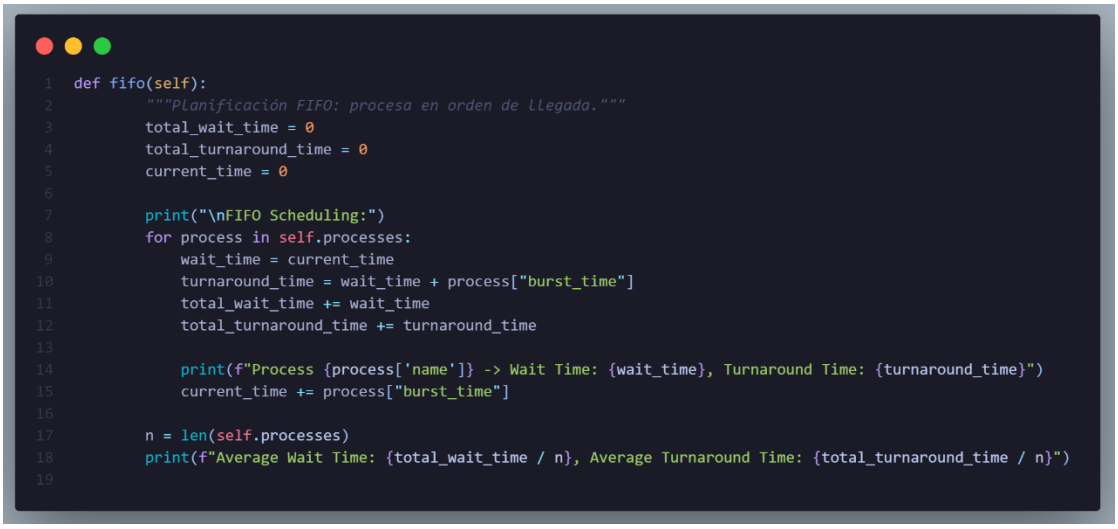
- **Round Robin**:
    - Divide el tiempo de CPU en intervalos llamados "quantum".
    - Proporciona equidad a todos los procesos, pero puede aumentar el overhead si el quantum es muy pequeño.

Funciones principales de ProcessScheduler:

- **add_process(name, burst_time, priority)**:
    - Agrega un proceso con un nombre, tiempo de ráfaga y prioridad a la cola de procesos.

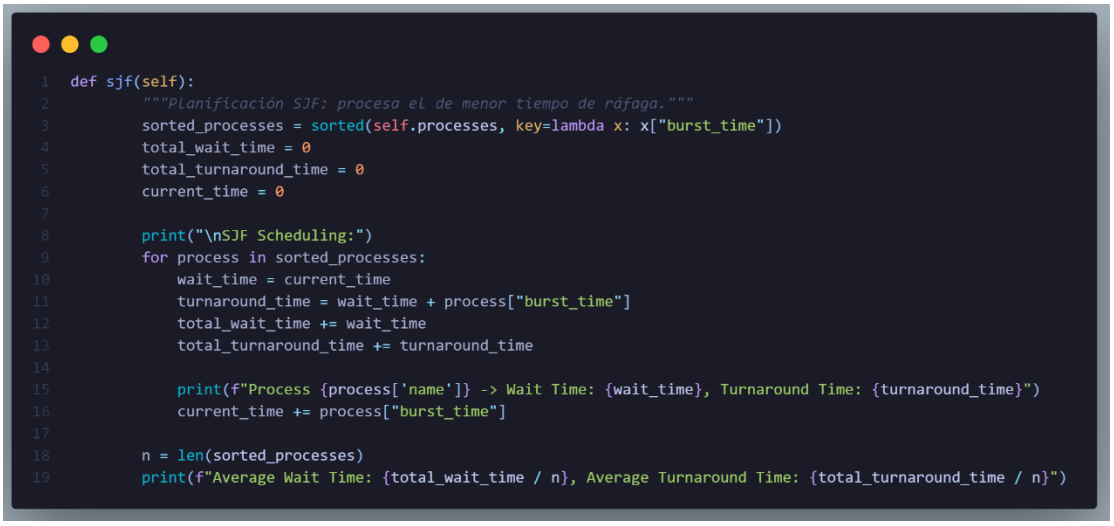
```python
class ProcessScheduler:
    def __init__(self):
        self.processes = []

    def add_process(self, name, burst_time, priority):
        """Agrega un nuevo proceso a la lista."""
        self.processes.append({"name": name, "burst_time": burst_time, "priority": priority})
```

- **fifo()**:
    - Aplica el algoritmo FIFO, calcula el tiempo de espera y de retorno para cada proceso, y devuelve los promedios.



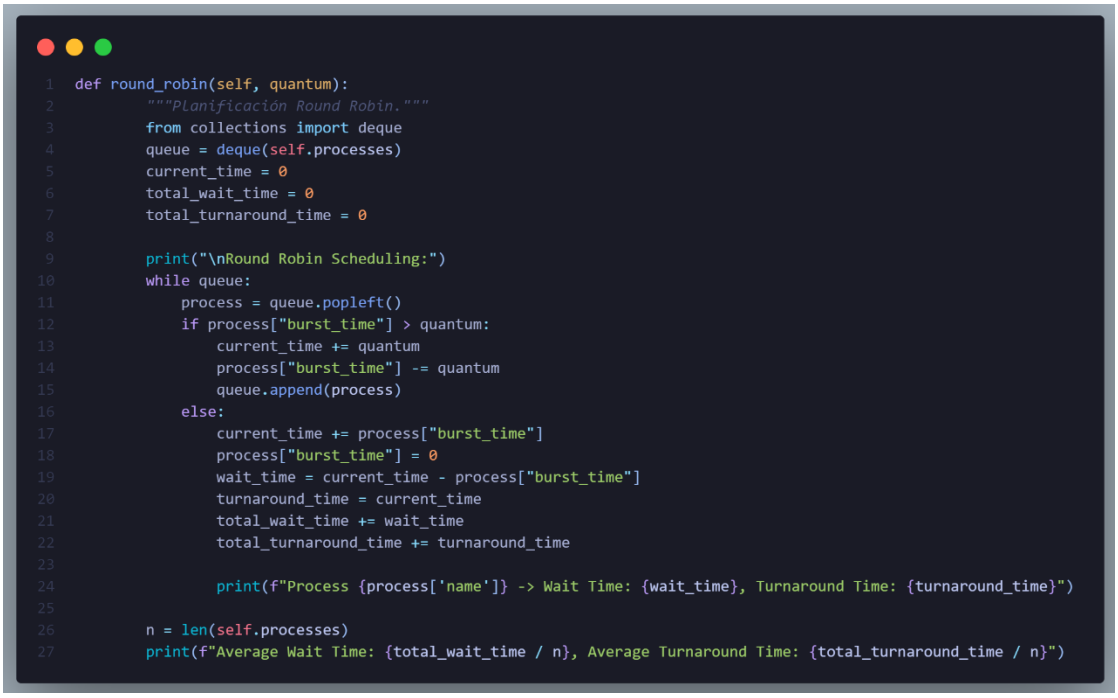
```python
def fifo(self):
        """Planificación FIFO: procesa en orden de llegada."""
        total_wait_time = 0
        total_turnaround_time = 0
        current_time = 0

        print("\nFIFO Scheduling:")
        for process in self.processes:
            wait_time = current_time
            turnaround_time = wait_time + process["burst_time"]
            total_wait_time += wait_time
            total_turnaround_time += turnaround_time

            print(f"Process {process['name']} -> Wait Time: {wait_time}, Turnaround Time: {turnaround_time}")
            current_time += process["burst_time"]

        n = len(self.processes)
        print(f"Average Wait Time: {total_wait_time / n}, Average Turnaround Time: {total_turnaround_time / n}")

```

- **sjf()**:
    - Ordena los procesos por tiempo de ráfaga, los ejecuta y calcula las métricas como en FIFO.



```python
def sjf(self):
        """Planificación SJF: procesa el de menor tiempo de ráfaga."""
        sorted_processes = sorted(self.processes, key=lambda x: x["burst_time"])
        total_wait_time = 0
        total_turnaround_time = 0
        current_time = 0

        print("\nSJF Scheduling:")
        for process in sorted_processes:
            wait_time = current_time
            turnaround_time = wait_time + process["burst_time"]
            total_wait_time += wait_time
            total_turnaround_time += turnaround_time

            print(f"Process {process['name']} -> Wait Time: {wait_time}, Turnaround Time: {turnaround_time}")
            current_time += process["burst_time"]

        n = len(sorted_processes)
        print(f"Average Wait Time: {total_wait_time / n}, Average Turnaround Time: {total_turnaround_time / n}")
```

- **round_robin(quantum)**:
    - Aplica el algoritmo Round Robin con el quantum especificado, gestionando los tiempos de ejecución de los procesos.



```python
def round_robin(self, quantum):
        """Planificación Round Robin."""
        from collections import deque
        queue = deque(self.processes)
        current_time = 0
        total_wait_time = 0
        total_turnaround_time = 0

        print("\nRound Robin Scheduling:")
        while queue:
            process = queue.popleft()
            if process["burst_time"] > quantum:
                current_time += quantum
                process["burst_time"] -= quantum
                queue.append(process)
            else:
                current_time += process["burst_time"]
                process["burst_time"] = 0
                wait_time = current_time - process["burst_time"]
                turnaround_time = current_time
                total_wait_time += wait_time
                total_turnaround_time += turnaround_time

                print(f"Process {process['name']} -> Wait Time: {wait_time}, Turnaround Time: {turnaround_time}")

        n = len(self.processes)
        print(f"Average Wait Time: {total_wait_time / n}, Average Turnaround Time: {total_turnaround_time / n}")
```

Administración de Memoria

La clase MemoryManager simula un esquema de paginación. Divide la memoria física en bloques de tamaño fijo llamados frames, y los procesos se dividen en páginas de igual tamaño.

Características principales:

- Los procesos ocupan tantos frames como páginas tengan.
- Si no hay frames libres, se aplica reemplazo de páginas (FIFO).
- Se proporciona una visualización del estado actual de los frames de memoria.

Funciones principales de MemoryManager:

- **allocate(process_id, process_size)**:
    - Intenta asignar memoria a un proceso. Devuelve True si se logra la asignación, o False si no hay espacio suficiente.

```python
class MemoryManager:
    def __init__(self, memory_size, page_size):
        self.memory_size = memory_size
        self.page_size = page_size
        self.pages = [-1] * (memory_size // page_size)
        self.page_queue = []  # Para FIFO
        self.page_access = {}  # Para LRU

    def allocate(self, process_id, process_size):
        """Asigna memoria para un proceso si hay suficiente espacio."""
        num_pages_needed = (process_size + self.page_size - 1) // self.page_size
        allocated_pages = 0

        for i in range(len(self.pages)):
            if self.pages[i] == -1:  # Si el marco está vacío
                self.pages[i] = process_id
                self.page_queue.append(process_id)
                allocated_pages += 1
                if allocated_pages == num_pages_needed:
                    return True  # Asignación exitosa

        # Si no hay suficiente espacio, revertimos los cambios
        for i in range(len(self.pages)):
            if self.pages[i] == process_id:
                self.pages[i] = -1
        return False  # No hay espacio suficiente
```

- **load_page(process_id, page_number)**:
    - Carga una página de un proceso en un frame. Si no hay espacio, realiza un reemplazo siguiendo el algoritmo FIFO.

```python
def load_page(self, process_id, page_number):
        """Carga una página en memoria con manejo de reemplazo."""
        if process_id in self.pages:
            print(f"Page {page_number} of Process {process_id} already in memory.")
            return

        if -1 in self.pages:
            # Espacio disponible
            free_index = self.pages.index(-1)
            self.pages[free_index] = process_id
            self.page_queue.append(process_id)
            self.page_access[process_id] = free_index
            print(f"Loaded Page {page_number} of Process {process_id} into Frame {free_index}.")
        else:
            # Reemplazo FIFO
            evicted = self.page_queue.pop(0)
            evict_index = self.pages.index(evicted)
            self.pages[evict_index] = process_id
            self.page_queue.append(process_id)
            self.page_access[process_id] = evict_index
            print(f"Page {page_number} of Process {evicted} evicted. Loaded Process {process_id}.")
```

- **display_memory()**:
    - Imprime el estado actual de los frames, indicando qué proceso ocupa cada uno.

```python
def display_memory(self):
        """Muestra el estado actual de la memoria."""
        print("\nMemory State:")
        for i, process in enumerate(self.pages):
            print(f"Frame {i}: {'Empty' if process == -1 else f'Process {process}'}")
```

Sistema de Archivos

La clase FileSystem implementa operaciones básicas de un sistema de archivos:

1. Crear directorios y archivos.
2. Escribir y leer archivos.
3. Eliminar archivos o directorios.
4. Listar contenidos del directorio raíz.

Funciones principales de FileSystem:

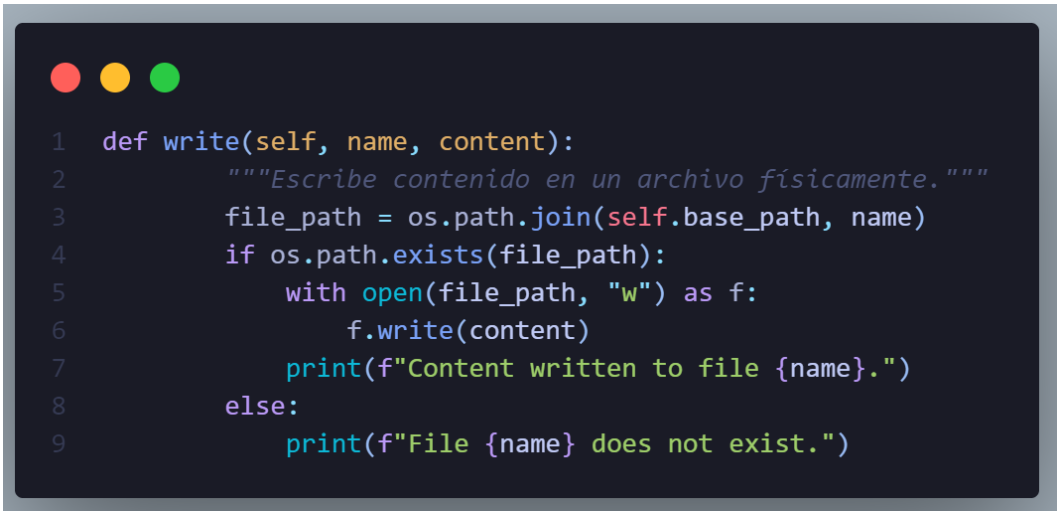
- **mkdir(name)**:
    - Crea un directorio con el nombre especificado.

```python
class FileSystem:
    def __init__(self, base_path="."):  # Cambia el directorio base al actual (raíz del proyecto)
        self.base_path = base_path

    def mkdir(self, name):
        """Crea un directorio físicamente."""
        dir_path = os.path.join(self.base_path, name)
        if os.path.exists(dir_path):
            print(f"Directory {name} already exists.")
        else:
            os.makedirs(dir_path)
            print(f"Directory {name} created.")
```

- **touch(name)**:
    - Crea un archivo vacío con el nombre especificado.

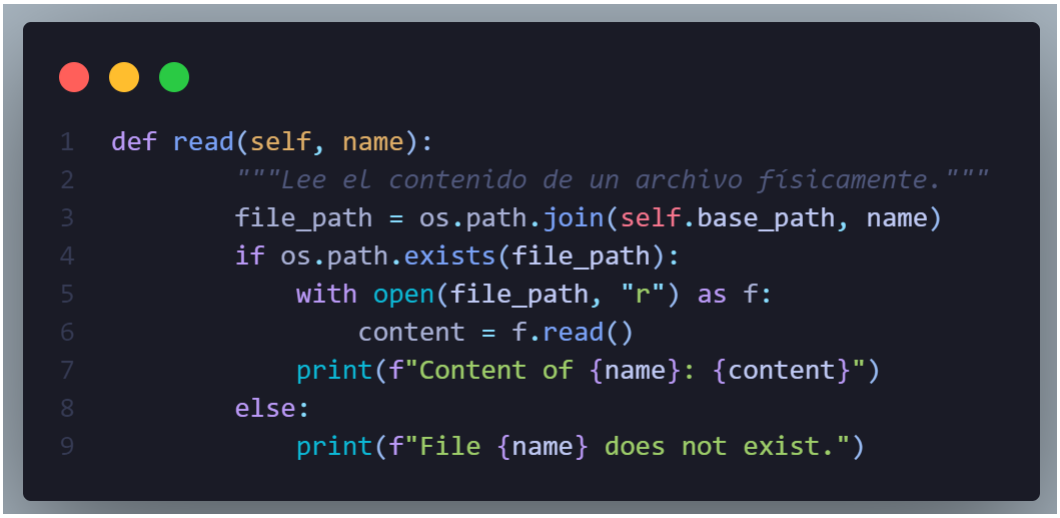
```python
def touch(self, name):
        """Crea un archivo físicamente."""
        file_path = os.path.join(self.base_path, name)
        if os.path.exists(file_path):
            print(f"File {name} already exists.")
        else:
            with open(file_path, "w") as f:
                pass
            print(f"File {name} created.")
```

- **write(name, content)**:
    - Escribe contenido en un archivo existente.



```python
def write(self, name, content):
        """Escribe contenido en un archivo físicamente."""
        file_path = os.path.join(self.base_path, name)
        if os.path.exists(file_path):
            with open(file_path, "w") as f:
                f.write(content)
            print(f"Content written to file {name}.")
        else:
            print(f"File {name} does not exist.")
```

- **read(name)**:
    - Lee y devuelve el contenido de un archivo.



```python
def read(self, name):
        """Lee el contenido de un archivo físicamente."""
        file_path = os.path.join(self.base_path, name)
        if os.path.exists(file_path):
            with open(file_path, "r") as f:
                content = f.read()
            print(f"Content of {name}: {content}")
        else:
            print(f"File {name} does not exist.")
```

- **rm(name)**:
    - Elimina un archivo o directorio existente.

```python
def rm(self, name):
        """Elimina un archivo o directorio físicamente."""
        path = os.path.join(self.base_path, name)
        if os.path.exists(path):
            if os.path.isdir(path):
                os.rmdir(path)
            else:
                os.remove(path)
            print(f"{name} deleted.")
        else:
            print(f"{name} does not exist.")
```

- **ls()**:
    - Lista todos los archivos y directorios en el directorio raíz.

```python
def ls(self):
        """Lista los contenidos del directorio base."""
        contents = os.listdir(self.base_path)
        print("\nContents:")
        for item in contents:
            print(item)
```

Descripción del Archivo interfaz.py

Estructura de la Interfaz

El archivo interfaz.py utiliza Tkinter para crear una interfaz gráfica que permite al usuario interactuar con el simulador. Está organizada en tres módulos principales:

1. Planificación de Procesos.
2. Administración de Memoria.
3. Sistema de Archivos.

Funcionalidades

- **Módulo de Planificación de Procesos:**
  - Permite agregar procesos mediante diálogos de entrada.
  - Ejecuta los algoritmos FIFO, SJF y Round Robin.
  - Muestra los resultados en un área de texto.

- **Módulo de Administración de Memoria:**
  - Permite visualizar el estado actual de los frames de memoria.
  - Muestra información sobre qué proceso ocupa cada frame.

- **Módulo de Sistema de Archivos:**
  - Permite crear, leer, escribir y eliminar archivos y directorios.
  - Lista los contenidos del directorio raíz.

Algoritmos Utilizados

FIFO

- **Descripción:** Los procesos se ejecutan en el orden en que llegan.
- **Ventaja:** Simple de implementar.
- **Desventaja:** Puede causar tiempos de espera altos para procesos largos.

SJF

- **Descripción:** Los procesos con menor tiempo de ráfaga tienen prioridad.
- **Ventaja:** Minimiza el tiempo promedio de espera.
- **Desventaja:** Los procesos largos pueden sufrir inanición.

Round Robin

- **Descripción:** Divide el tiempo de CPU en intervalos llamados "quantum".
- **Ventaja:** Proporciona equidad entre los procesos.
- **Desventaja:** Puede aumentar el overhead si el quantum es demasiado pequeño.

Interacción entre Módulos

La interfaz gráfica se comunica con las clases de simulador.py para ejecutar las funcionalidades del simulador. Cada botón de la interfaz llama a una función específica que interactúa con las clases ProcessScheduler, MemoryManager o FileSystem.

Conclusiones

Flores Torres Ángel Joel.

Este proyecto nos permitió comprender de manera práctica cómo funcionan los componentes fundamentales de un sistema operativo, como la planificación de procesos, la administración de memoria y el sistema de archivos. Además, la implementación de algoritmos como FIFO, SJF y Round Robin nos ayudó a analizar las ventajas y desventajas de cada enfoque.

Morales Soto Fernando.

La integración de una interfaz gráfica hizo que la interacción con el simulador fuera más accesible, lo que permitió no solo probar, sino también visualizar el comportamiento de un sistema operativo. Esto fue clave para comprender cómo las decisiones en la planificación y la memoria afectan el rendimiento general.

Muciño Hernández Zara Paulina.

Trabajar en este simulador nos enseñó cómo los sistemas operativos manejan recursos de manera eficiente. Particularmente, implementar la paginación en la memoria y los algoritmos de reemplazo nos dio una visión detallada de cómo se optimizan los recursos en tiempo real.

Hernández Hernández Oscar Mario.

Este proyecto no solo fue una práctica de programación, sino también una forma de explorar la teoría detrás de los sistemas operativos. La aplicación de conceptos como la equidad en Round Robin o la eficiencia de SJF nos permitió conectar la teoría con la práctica de una manera significativa.