Práctica 07: Administrador de memoria 2

Maestro:

Javier Rosales Martinez

Materia:

Seminario de Solución de Problemas de Sistemas Operativos

Sección:

D06

Alumno:

Alejandro Covarrubias Sánchez

Código:

221350192

Antecedentes

La administración de memoria en sistemas operativos es el proceso que gestiona cómo se asigna y organiza la memoria en una computadora para ejecutar programas de manera eficiente. Este mecanismo asegura que cada aplicación obtenga suficiente memoria para funcionar, controla el acceso para evitar conflictos entre procesos y libera la memoria cuando ya no se necesita. Su objetivo principal es optimizar el rendimiento del sistema, evitando la sobrecarga o el uso ineficiente de los recursos disponibles.

Diferentes Sistemas Operativos utilizan diferentes algoritmos para gestionar la administración de memoria, siendo los más comunes primer ajuste, mejor ajuste, peor ajuste y siguiente ajuste. El primer ajuste busca el primer bloque de memoria libre que sea lo suficientemente grande para el proceso. El mejor ajuste selecciona el bloque más pequeño que cumpla con el tamaño requerido, minimizando el desperdicio de espacio. El peor ajuste elige el bloque más grande disponible, dejando el mayor espacio posible para futuros procesos. El siguiente ajuste es similar al primer ajuste, pero comienza la búsqueda desde la última ubicación donde se hizo una asignación, evitando recorrer la memoria desde el principio cada vez.

Metodología

El programa está escrito en Python, e implementa una simulación de varios algoritmos de asignación de memoria: Primer Ajuste, Mejor Ajuste, Peor Ajuste y Siguiente Ajuste. No utiliza librerías externas, solo funciones nativas de Python.

Las funciones principales son `first_fit()`, `best_fit()`, `worst_fit()` y `next_fit()`, las cuales implementan los algoritmos de asignación de memoria, que intentan asignar archivos a bloques de memoria disponibles de acuerdo a distintas estrategias.

```python
# Algoritmo Primer Ajuste
def first_fit(archivos, bloques_memoria):
    asignaciones = [-1] * len(archivos)
    for i, archivo in enumerate(archivos):
        for j, bloque in enumerate(bloques_memoria):
            if archivo[1] <= bloque:
                asignaciones[i] = j
                bloques_memoria[j] -= archivo[1]
                break
    return asignaciones

# Algoritmo Mejor Ajuste
def best_fit(archivos, bloques_memoria):
    asignaciones = [-1] * len(archivos)
    for i, archivo in enumerate(archivos):
        mejor_bloque = -1
        for j, bloque in enumerate(bloques_memoria):
            if archivo[1] <= bloque:
                if mejor_bloque == -1 or bloques_memoria[mejor_bloque] > bloque:
                    mejor_bloque = j
        if mejor_bloque != -1:
            asignaciones[i] = mejor_bloque
            bloques_memoria[mejor_bloque] -= archivo[1]
    return asignaciones
```

```python
# Algoritmo Peor Ajuste
def worst_fit(archivos, bloques_memoria):
    asignaciones = [-1] * len(archivos)
    for i, archivo in enumerate(archivos):
        peor_bloque = -1
        for j, bloque in enumerate(bloques_memoria):
            if archivo[1] <= bloque:
                if peor_bloque == -1 or bloques_memoria[peor_bloque] < bloque:
                    peor_bloque = j
        if peor_bloque != -1:
            asignaciones[i] = peor_bloque
            bloques_memoria[peor_bloque] -= archivo[1]
    return asignaciones

# Algoritmo Siguiente Ajuste
def next_fit(archivos, bloques_memoria):
    asignaciones = [-1] * len(archivos)
    posicion_actual = 0
    for i, archivo in enumerate(archivos):
        for j in range(len(bloques_memoria)):
            if archivo[1] <= bloques_memoria[posicion_actual]:
                asignaciones[i] = posicion_actual
                bloques_memoria[posicion_actual] -= archivo[1]
                break
            posicion_actual = (posicion_actual + 1) % len(bloques_memoria)
    return asignaciones
```

En el caso de `agregar_archivo()` y `agregar_espacio_memoria()`, estas permiten agregar archivos y bloques de memoria al sistema, solicitando los datos por consola.

```
# Función para agregar un archivo a la lista
def agregar_archivo(archivos):
    nombre = input("Ingresa el nombre del archivo: ")
    tamaño = int(input("Ingresa el tamaño del archivo (en KB): "))
    posicion = input("¿Agregar al inicio o al final? (I/F): ").strip().lower()

    if posicion == 'i':
        archivos.insert(0, (nombre, tamaño))
    else:
        archivos.append((nombre, tamaño))
    print(f"Archivo {nombre} ({tamaño}kb) agregado a la lista.")

# Función para agregar un nuevo espacio de memoria
def agregar_espacio_memoria(bloques_memoria):
    tamaño = int(input("Ingresa el tamaño del nuevo bloque de memoria (en KB): "))
    estado = input("¿El bloque está ocupado o disponible? (O/D): ").strip().lower()

    # Si el bloque está ocupado, se representa con tamaño 0
    bloque = tamaño if estado == 'd' else 0
    posicion = input("¿Agregar al inicio o al final? (I/F): ").strip().lower()

    if posicion == 'i':
        bloques_memoria.insert(0, bloque)
    else:
        bloques_memoria.append(bloque)
    print(f"Bloque de {tamaño}kb ({'ocupado' if bloque == 0 else 'disponible'}) agregado a la lista.")
```

La solución está diseñada para leer archivos y tamaños desde un archivo de entrada, luego permite al usuario interactuar mediante un menú donde se puede elegir entre agregar archivos, agregar bloques de memoria o ejecutar alguno de los algoritmos de ajuste mencionados. Esto simula la gestión de memoria dinámica en un entorno controlado.

Conclusión

El programa actual implementa una simulación de algoritmos de asignación de memoria, permitiendo al usuario gestionar archivos y bloques de memoria mediante un menú interactivo. Además, permite agregar archivos y bloques de memoria de forma manual, lo que facilita la simulación de diferentes escenarios de gestión de memoria.

Funcionalidades ya funcionales incluyen la lectura de archivos de entrada con el tamaño de los archivos y su procesamiento, la interacción del usuario a través de un menú que permite seleccionar algoritmos, así como la visualización de los resultados de la asignación de memoria. También se puede añadir archivos y bloques de memoria al sistema, lo que hace que el programa sea flexible y dinámico.

En futuras versiones podrían incluir la implementación de una validación más robusta para las entradas del usuario, evitando posibles errores al ingresar datos incorrectos. También sería útil optimizar la impresión de resultados, mostrando de manera más clara la memoria restante después de cada asignación. Se podría agregar una opción para guardar los resultados en un archivo o para graficar el uso de memoria. Además, el manejo de archivos de entrada podría mejorarse para permitir formatos más complejos o interactuar con bases de datos.

Referencias

Bottallo, D. A. (n.d.). *ADMINISTRACIÓN DE LA MEMORIA*. Facultad de Ciencias Exactas, Ingeniería y Agrimensura. https://www.fceia.unr.edu.ar/~diegob/so/presenta/09-Memoria.pdf