Portafolio de Evidencias

Maestro:

Javier Rosales Martinez

Materia:

Seminario de Solución de Problemas de Sistemas Operativos

Sección:

D06

Alumno:

Alejandro Covarrubias Sánchez

Código:

221350192

Procesamiento por Lotes

El procesamiento por lotes es un método de procesamiento de datos que consiste en agrupar tareas o trabajos similares en un lote, que luego se ejecutan de manera secuencial sin necesidad de intervención humana durante la ejecución. Este enfoque es común en sistemas donde es necesario manejar grandes volúmenes de datos o realizar tareas repetitivas, como en la contabilidad, el procesamiento de nóminas o la generación de informes.

Características

1. Automatización: Una vez que se inicia un lote, se ejecuta de principio a fin sin necesidad de intervención humana.
2. Eficiencia: Al agrupar tareas, se pueden optimizar los recursos del sistema, ya que las operaciones pueden realizarse de manera más eficiente en comparación con el procesamiento en tiempo real.
3. Menor interacción: Dado que el procesamiento se realiza en lotes, no es necesario que un operador esté presente para iniciar cada tarea individualmente.
4. Programación: El procesamiento por lotes generalmente se programa para ejecutarse durante períodos de baja actividad, como durante la noche o los fines de semana, para no interferir con las operaciones diarias.
5. Tiempos de espera: Aunque es eficiente, el procesamiento por lotes puede implicar un retraso en la disponibilidad de los resultados, ya que las tareas se agrupan y no se procesan inmediatamente.

Uso

El procesamiento por lotes se utiliza en situaciones donde es necesario procesar grandes volúmenes de datos de manera eficiente. Es ideal para tareas repetitivas que pueden agruparse y ejecutarse en conjunto, como la generación de informes, la facturación o el procesamiento de transacciones. Además, es especialmente útil para tareas que no requieren procesamiento en tiempo real, ya que permite programar su ejecución en momentos de baja actividad, como durante la noche.

También se emplea para optimizar recursos, particularmente en sistemas donde el procesamiento individual de tareas sería menos eficiente o más costoso.

Ventajas del procesamiento por lotes

1. Eficiencia: Al procesar múltiples tareas juntas, se reduce el tiempo total necesario, optimizando el uso de recursos como la CPU y la memoria.
2. Automatización: Los procesos en lotes pueden ser programados para ejecutarse sin intervención humana, lo que reduce errores y mejora la consistencia.
3. Reducción de costos: Se pueden programar para ejecutarse en momentos de baja demanda, aprovechando mejor la capacidad del sistema y reduciendo costos operativos.
4. Gestión simplificada: Es más fácil gestionar y monitorear un lote de tareas que hacerlo con cada tarea por separado.

Desventajas del procesamiento por lotes

1. Falta de inmediatez: Como las tareas se procesan en lotes, puede haber retrasos antes de que los datos o resultados estén disponibles, lo que no es ideal para aplicaciones en tiempo real.
2. Complejidad en la gestión de errores: Si una tarea en un lote falla, puede afectar al resto del lote o requerir reejecutar todo el lote, lo que puede ser ineficiente.
3. Rigidez: Los sistemas de procesamiento por lotes a veces carecen de flexibilidad, ya que están diseñados para manejar tareas predefinidas, lo que puede no adaptarse bien a cambios o tareas inesperadas.
4. Requiere planificación previa: Para obtener los beneficios del procesamiento por lotes, es necesario planificar cuidadosamente cuándo y cómo se ejecutarán los lotes, lo que puede ser un proceso complejo.

Práctica 01

Antecedentes

El procesamiento por lotes es una técnica que consiste en agrupar y ejecutar una serie de tareas o datos en bloques, llamados lotes, en lugar de procesarlos uno a uno de forma individual e inmediata.

Este programa busca leer un archivo de texto inicial que contiene una gran cantidad de datos línea por línea en lotes, para dar formato a cada una y luego escribir el contenido resultante en un nuevo archivo de salida.

Procesar los datos en lotes de esta manera permite manejar grandes volúmenes de información de manera más eficiente, reduciendo la carga en la memoria y optimizando el rendimiento. Al dividir el procesamiento en bloques, se evita tener que cargar en memoria y manipular todo el archivo de una sola vez, lo que es particularmente beneficioso cuando se trabaja con archivos grandes. Este enfoque también facilita la escritura más estructurada en el archivo de salida y puede mejorar la velocidad del programa al reducir el número de operaciones de entrada y salida.

Metodología

El programa fue escrito en Python y utiliza únicamente librerías predeterminadas, como las funciones `open()` para la manipulación de archivos y estructuras de control básicas como `try` y `except` para manejar excepciones. No se utilizaron librerías externas.

Se tienen dos funciones principales: `process_line()`, que procesa cada línea individualmente convirtiendo números hexadecimales a decimales y viceversa y proporciona el formato de salida a cada una, y `read_and_process_file()`, que lee el archivo en lotes de un tamaño determinado, procesa cada línea y luego guarda el resultado en un archivo de salida.

```
def process_line(line):

      # Procesar la línea y realizar conversiones

    prefix, suffix = line.strip().split('/', 1)

    hex_parts = prefix.split(':')

    decimal_numbers = [int(number, 16) for number in hex_parts]

    parts = suffix.strip().split(',')

    second_string = parts[2]

    decimal_numbers_from_suffix = parts[5].split('.')

    hex_numbers = [format(int(number), 'X') for number in
       decimal_numbers_from_suffix]



      # Crear el formato de salida

    decimal_numbers_str = ' : '.join(map(str, decimal_numbers))

    hex_numbers_str = '.'.join(hex_numbers)

    output_line = f"{second_string} : {decimal_numbers_str} :
       {hex_numbers_str}\n"



    return output_line
```

El algoritmo clave es el procesamiento por lotes, que permite manejar archivos grandes de manera eficiente al procesar grupos de líneas a la vez. Además, utiliza conversiones de formatos numéricos (hexadecimales y decimales) para transformar los datos adecuadamente.

Práctica 02

Antecedentes

Algunos antivirus utilizan el procesamiento por lotes para analizar múltiples archivos en un directorio de manera secuencial y organizada. En lugar de examinar cada archivo individualmente en tiempo real, el procesamiento por lotes permite al antivirus acumular un conjunto de archivos y analizarlos todos juntos en una sola operación, lo que puede ser más eficiente y rápido cuando se trata de volúmenes grandes de datos.

Estos antivirus primero escanean los directorios seleccionados, identificando todos los archivos presentes. Luego, estos archivos se agrupan en lotes y se comparan con una base de datos de firmas de virus conocidas, buscando patrones de código malicioso. Además, pueden aplicar análisis heurísticos para detectar comportamientos sospechosos en archivos no registrados en su base de datos.

Una vez que el procesamiento del lote se completa, el antivirus genera un informe sobre los archivos detectados, tomando acciones como eliminar, poner en cuarentena o reparar los archivos sospechosos o infectados.

Metodología

El programa está escrito en Python y utiliza las librerías `os`, `schedule` y `time` para automatizar la búsqueda y eliminación de archivos sospechosos en un directorio. Se tienen 2 funciones principales, `get_evil_files()` y `delete_files()`. La primera utiliza `os.walk()` para recorrer de manera recursiva los directorios y subdirectorios, buscando archivos con una extensión específica. Los archivos encontrados se almacenan en una lista, y luego son eliminados por la segunda función, utilizando `os.remove()`, y manejando errores en caso de que algún archivo no pueda eliminarse.

La tarea de búsqueda y eliminación se programa para ejecutarse periódicamente dentro de la función `scheduled_task()`, usando la librería `schedule`, lo que permite que el proceso se repita automáticamente cada 30 segundos sin intervención manual.

El bucle principal mantiene el programa en ejecución, verificando las tareas programadas y haciendo pausas de 1 segundo con `time.sleep()` para optimizar el uso del CPU. El diseño modular del programa facilita su mantenimiento, y el manejo de errores asegura que el programa siga funcionando incluso si hay problemas con algunos archivos.

```
def get_evil_files(directory, evil_extension):
    """
    Recorre el directorio y sus subcarpetas para encontrar archivos con la extensión maligna.
    Devuelve una lista de rutas de archivos malignos.
    """
    evil_files = []
    for root_dir, sub_dir, files in os.walk(directory):
        for file in files:
            if file.endswith(evil_extension):
                file_dir = os.path.join(root_dir, file)
                evil_files.append(file_dir)
    return evil_files
```

```
def scheduled_task():
    """
    Función que se ejecuta periódicamente para buscar y eliminar archivos malignos.
    """
    directory = "D:/test"
    evil_extension = ".pdf"
    evil_files = get_evil_files(directory, evil_extension)
    if evil_files:
        delete_files(evil_files)
    else:
        print("No se encontraron archivos malignos en este lote.")
```

Planificación de procesos

La planificación de procesos es una función crucial dentro de los sistemas operativos. Su propósito es gestionar y controlar la ejecución de los procesos en un sistema, asegurando que el uso del procesador (CPU) sea eficiente y equitativo entre todas las tareas que lo requieren. Los procesos son programas en ejecución, y como generalmente hay varios procesos en un sistema, la planificación decide el orden en que estos se ejecutarán y por cuánto tiempo.

Los principales objetivos de la planificación de procesos son asegurar que el procesador esté ocupado la mayor parte del tiempo, garantizar que los procesos interactivos reciban atención rápidamente, reducir tiempos de espera, tiempos de ejecución de procesos, etc. y asegurar que ningún proceso quede injustamente ignorado.

Algoritmos de planificación

Para lograr la planificación más eficiente y equitativa, existen diferentes algoritmos que buscan determinar el orden de ejecución de los procesos tomando en cuenta diferentes parámetros. Los más comunes son FIFO, SFJ, Prioridades y Round Robin.

First In, First Out

Es uno de los algoritmos más simples de planificación. En FIFO o FCFS (First Come, First Served), los procesos se ejecutan en el orden en que llegan a la cola de procesos listos. El proceso que llega primero será el primero en recibir tiempo de CPU, y continuará ejecutándose hasta que termine o entre en estado de espera.

- Ventaja: Es sencillo de implementar.
- Desventaja: Puede tener un efecto negativo conocido como problema del convoy: si un proceso largo llega primero, hará esperar a los procesos más pequeños que llegaron después, lo que puede aumentar el tiempo de espera promedio.

Shortest Job First

El algoritmo de SJF (trabajo más corto primero) selecciona para su ejecución el proceso que tiene la duración más corta estimada. Es ideal para sistemas donde se conoce de antemano cuánto tiempo durará cada proceso.

- Ventaja: Minimiza el tiempo promedio de espera, ya que los procesos cortos se ejecutan antes, lo que generalmente reduce la cola de espera.
- Desventaja: No es adecuado para sistemas en tiempo real o cuando no se conoce la duración de los procesos de antemano. También sufre de inanición: los procesos largos pueden esperar indefinidamente si siempre llegan procesos más cortos.

Prioridad

En este algoritmo, cada proceso tiene asignada una prioridad, y el proceso con la prioridad más alta será el primero en ejecutarse. Para esto también deberá determinarse si se sigue una lógica ascendente o descendente, donde la prioridad más alta será la que tenga el número mayor o menor respectivamente. Las prioridades pueden ser asignadas estática o dinámicamente, y pueden estar basadas en diferentes factores, como la importancia del proceso o su tiempo de espera en cola.

- Ventaja: Es útil en sistemas donde algunos procesos tienen mayor importancia que otros, como en sistemas de tiempo real.
- Desventaja: Puede ocurrir un fenómeno conocido como inanición o hambre (starvation), donde un proceso de baja prioridad nunca recibe CPU si constantemente llegan procesos de mayor prioridad. Este problema se puede mitigar con una técnica llamada envejecimiento, donde se incrementa la prioridad de los procesos a medida que pasa el tiempo.

Round Robin

Este es un algoritmo más justo y equitativo, especialmente útil en sistemas interactivos. En Round Robin, cada proceso recibe una cantidad fija de tiempo, llamada quantum de tiempo o time slice. Si el proceso no termina durante ese quantum de tiempo, es interrumpido y colocado al final de la cola de procesos listos, mientras el siguiente proceso en la cola recibe su turno de ejecución.

Características clave:

- Ventaja: Asegura que todos los procesos reciban tiempo de CPU de manera equitativa, lo que reduce la espera de procesos interactivos.
- Desventaja: Si el quantum de tiempo es demasiado pequeño, el sistema puede gastar mucho tiempo cambiando de proceso (lo que se conoce como overhead de conmutación de contexto). Si es demasiado largo, se convierte en un algoritmo similar a FIFO.

Práctica 03

Antecedentes

Los algoritmos de planificación son métodos utilizados en sistemas operativos para gestionar cómo y cuándo se ejecutan los procesos, o tareas, en un procesador. Su principal objetivo es distribuir eficientemente el tiempo de CPU entre los procesos para maximizar el rendimiento del sistema.

Existen varios tipos de algoritmos de planificación, siendo FIFO (First-In, First-Out) el más simple. En este, los procesos se ejecutan en el orden en que llegan, pero puede ser ineficiente cuando un proceso largo bloquea a los demás. Otro ejemplo es SJF (Shortest Job First), que prioriza los procesos más cortos para reducir el tiempo de espera promedio, aunque también puede ser ineficiente para los procesos largos y requiere estimar con precisión el tiempo de ejecución de cada proceso.

Metodología

El programa está escrito en Python e implementa y simula dos algoritmos de planificación de procesos: FIFO (First-In, First-Out) y SJF (Shortest Job First). La solución está diseñada para leer un archivo que contiene la información de los procesos y luego “ejecutarlos” utilizando cada uno de los algoritmos. En este caso, la “ejecución” es la impresión en pantalla del proceso, mostrando los tiempos de inicio y finalización de cada proceso. No se utilizaron librerías externas en este programa, solo funciones básicas del lenguaje Python, como la función `open()` para manejar archivos.

La solución está diseñada con una función independiente para cada algoritmo de planificación. La función principal `main()` gestiona la lectura de los datos de procesos desde un archivo de texto, añadiendo cada línea a una lista de procesos donde cada uno es una tupla con tres valores: el nombre del proceso, su duración y su prioridad y luego ejecuta cada algoritmo para observar y comparar los diferentes tiempos de inicio y finalización dependiendo del algoritmo utilizado.

`fifo_scheduling()` ejecuta los procesos en el orden en que aparecen en la lista sin ninguna reordenación, manteniendo un contador del tiempo actual `current_time` que se va actualizando según el tiempo de ejecución de cada proceso.

```python
def fifo_scheduling(processes):
  """
  Ejecuta cada proceso sin ordenar la lista, mostrando el tiempo de inicio y finalización
  """
  current_time = 0

  for process in processes:
    start_time = current_time
    end_time = current_time + process[1]
    print(f"{process[0]}\t| Inicio: {start_time}s | Fin: {end_time}s")
    current_time = end_time
```

`sjf_scheduling()` recibe la misma lista de procesos, pero antes de ejecutarlos, los ordena en función de la duración del proceso. Luego, ejecuta los procesos en este orden, actualizando los tiempos de inicio y fin.

```
def sjf_scheduling(processes):
  """
  Ordena la lista de procesos según su duración, y los ejecuta mostrando el tiempo de inicio y finalización
  """
  current_time = 0
  # Ordena la lista de acuerdo a la duración del proceso
  processes_sorted = sorted(processes, key=lambda x: x[1])

  for process in processes_sorted:
    start_time = current_time
    end_time = current_time + process[1]
    print(f"{process[0]}\t| Inicio: {start_time}s | Fin: {end_time}s")
    current_time = end_time
```

Práctica 04

Antecedentes

Los algoritmos de planificación son métodos utilizados en sistemas operativos para gestionar cómo y cuándo se ejecutan los procesos, o tareas, en un procesador. Su principal objetivo es distribuir eficientemente el tiempo de CPU entre los procesos para maximizar el rendimiento del sistema.

Existen varios tipos de algoritmos de planificación, como el algoritmo de Prioridad, que ejecuta los procesos según su importancia, pero puede causar que los de baja prioridad esperen indefinidamente. Por otro lado, Round Robin mejora la equidad asignando a cada proceso un tiempo fijo para ejecutarse antes de pasar al siguiente, asegurando que todos reciban atención, aunque puede no ser óptimo para procesos cortos.

Metodología

El programa está escrito en Python e implementa y simula cuatro algoritmos de planificación de procesos: FIFO (First-In, First-Out) y SJF (Shortest Job First), descritos en la práctica anterior y añadiendo Prioridad y Round Robin.

La solución está diseñada para leer un archivo que contiene la información de los procesos y luego “ejecutarlos” utilizando cada uno de los algoritmos. En este caso, la “ejecución” es la impresión en pantalla del proceso, mostrando los tiempos de inicio y finalización de cada proceso. No se utilizaron librerías externas en este programa, solo funciones básicas del lenguaje Python, como la función `open()` para manejar archivos.

La solución está diseñada con una función independiente para cada algoritmo de planificación. La función principal `main()` gestiona la lectura de los datos de procesos desde un archivo de texto, añadiendo cada línea a una lista de procesos donde cada uno es una tupla con tres valores: el nombre del proceso, su duración y su prioridad y luego ejecuta cada algoritmo para observar y comparar los diferentes tiempos de inicio y finalización dependiendo del algoritmo utilizado.

`priority_scheduling()` implementa la planificación por prioridad. Similar a la función por SJF de la práctica 03, pero los procesos son ordenados por su prioridad antes de ejecutarse. Los procesos con prioridad más baja (valor numérico más pequeño) se ejecutan primero.

```
def priority_scheduling(processes):
  """
  Ordena la lista de procesos según su prioridad, y los ejecuta mostrando el tiempo de inicio y finalización
  """
  current_time = 0
  processes_sorted = sorted(processes, key=lambda x: x[2])

  for process in processes_sorted:
    start_time = current_time
    end_time = current_time + process[1]
    print(f"{process[0]}\t| Inicio: {start_time}s | Fin: {end_time}s | Prioridad: {process[2]}")
    current_time = end_time
```

Por último, `round_robin()` recibe un "quantum" o tiempo fijo de ejecución por proceso como parámetro. En esta función, los procesos se ejecutan por intervalos

de tiempo, si el tiempo de ejecución del proceso es mayor al quantum, se reintroduce en la cola de ejecución con el tiempo restante.

```
def round_robin(processes, quantum):
  """
  Ejecuta cada proceso únicamente durante el tiempo indicado, hasta finalizarlos.
  La lista no se ordena antes de procesarse.
  """
  current_time = 0
  ready_queue = processes.copy()

  while ready_queue:
    process = ready_queue.pop(0)
    name, duration, _ = process
    if duration <= quantum:
      start_time = current_time
      end_time = current_time + duration
      print(f"{process[0]}\t| Inicio: {start_time}s | Fin: {end_time}s")
      current_time = end_time
    else:
      start_time = current_time
      end_time = current_time + quantum
      print(f"{process[0]}\t| Inicio: {start_time}s | Fin: {end_time}s")
      current_time = end_time
      ready_queue.append((name, duration - quantum, _))
```

Práctica 05

Antecedentes

Los algoritmos de planificación son métodos utilizados en sistemas operativos para gestionar cómo y cuándo se ejecutan los procesos, o tareas, en un procesador. Su principal objetivo es distribuir eficientemente el tiempo de CPU entre los procesos para maximizar el rendimiento del sistema.

Existen varios tipos de algoritmos de planificación, FIFO (First-In, First-Out) ejecuta los procesos en el orden en que llegan, SJF (Shortest Job First) prioriza los procesos más cortos para reducir el tiempo de espera promedio.

El algoritmo de Prioridad ejecuta los procesos según su importancia, y Round Robin mejora la equidad asignando a cada proceso un tiempo fijo para ejecutarse antes de pasar al siguiente, asegurando que todos reciban atención.

Metodología

El programa está escrito en Python, e implementa diferentes algoritmos de planificación de procesos, utilizando funciones que simulan cómo los procesos son ejecutados según la estrategia seleccionada. No utiliza librerías externas, ya que todo se gestiona con funciones y estructuras básicas de Python como listas y bucles.

Utiliza las funciones y algoritmos para simular los tipos de planificación descritos en las 2 prácticas anteriores, y añade una nueva función `add_process()` que permite al usuario agregar un proceso manualmente con un nombre, duración y prioridad, además de especificar la posición del proceso en la lista, ya sea al inicio o al final.

```python
def add_process(processes):
  """
  Recibe un nuevo proceso por consola y lo agrega a la lista.
  """
  name = input("\nNombre del proceso: ")
  duration = int(input("Duración del proceso: "))
  priority = int(input("Prioridad del proceso: "))

  new_process = (name, duration, priority)

  position = input("Agregar al inicio o al final de la lista? (I/F): ")
  if position.lower() == "i":
    processes.insert(0, new_process)  # Agrega al inicio
  else:
    processes.append(new_process)  # Agrega al final

  return processes
```

La solución permite leer una lista de procesos desde un archivo de texto o añadir procesos manualmente, proporcionando flexibilidad para gestionar y simular diferentes escenarios.

La interacción con el usuario se realiza mediante un menú que ofrece opciones para seleccionar y ejecutar distintos algoritmos de planificación, como FIFO, SJF, planificación por prioridades y Round Robin. Cada algoritmo imprime el tiempo de inicio y finalización de los procesos, lo que facilita la comparación de los resultados y el análisis de cómo cada estrategia afecta el rendimiento de la ejecución.

Administración de memoria

La administración de memoria se encarga de gestionar el uso y la asignación de la memoria principal (RAM) de un sistema informático. Su objetivo es optimizar el uso de la memoria de manera eficiente, permitiendo que múltiples programas se ejecuten simultáneamente sin que interfieran entre sí y asegurando que el sistema funcione de manera estable y rápida. Cumple varias funciones como asignar memoria a los procesos cuando lo necesitan y liberarla cuando ya no la están utilizando, evitar que un proceso acceda a la memoria asignada a otro proceso, protegiendo así la integridad de los datos, administrar las áreas de la memoria que no están siendo utilizadas para que puedan ser asignadas a nuevos procesos de manera eficiente e implementar mecanismos que permiten que la memoria física (RAM) se use de manera más eficiente mediante el uso de almacenamiento secundario (disco duro) cuando la RAM no es suficiente.

Algoritmos de administración de memoria

Los sistemas operativos utilizan diferentes algoritmos de administración de memoria para gestionar la asignación y liberación de bloques de memoria a los procesos que lo necesitan. En particular, estos algoritmos buscan optimizar la colocación de procesos en las áreas disponibles de la memoria, reduciendo la fragmentación y mejorando el rendimiento.

Primer ajuste

Busca el primer bloque de memoria libre que sea lo suficientemente grande para acomodar el proceso o la solicitud de memoria. Una vez que encuentra un bloque adecuado, asigna la memoria solicitada a ese bloque y deja el resto (si sobra memoria) como un bloque libre.

Ventajas:

- Es simple y rápido de implementar porque no se necesita explorar toda la lista de bloques libres.
- En muchos casos, la memoria se asigna rápidamente ya que no es necesario realizar una búsqueda exhaustiva.

Desventajas:

- Genera fragmentación externa: al asignar el primer bloque disponible, se pueden dejar pequeños fragmentos de memoria dispersos que no se pueden reutilizar eficientemente.
- Si los primeros bloques disponibles son pequeños, el sistema puede terminar revisando varias veces una gran cantidad de bloques, lo que puede afectar el rendimiento a largo plazo.

Mejor ajuste

Busca el bloque de memoria libre que sea lo más cercano posible en tamaño a la solicitud de memoria. Es decir, intenta encontrar el bloque que deje la menor cantidad de espacio desperdiciado.

Ventajas:

- Al reducir el espacio libre restante, disminuye la probabilidad de generar fragmentos pequeños e inútiles.
- Puede ser eficiente en términos de uso de la memoria, ya que maximiza el uso de los bloques más pequeños.

Desventajas:

- Mayor tiempo de búsqueda: como el algoritmo revisa toda la lista de bloques libres para encontrar el mejor ajuste, el tiempo de búsqueda puede ser más largo, especialmente si hay muchos bloques.
- Aunque minimiza la fragmentación externa, a largo plazo puede hacer que la memoria se llene de pequeños fragmentos que son difíciles de usar.

Peor ajuste

Selecciona el bloque de memoria libre más grande disponible para acomodar la solicitud de memoria. El objetivo es dejar bloques de memoria más grandes y útiles después de la asignación, en lugar de dividir en fragmentos pequeños.

Ventajas:

- Reduce la probabilidad de que queden bloques de memoria muy pequeños después de cada asignación, ya que utiliza los bloques más grandes.
- Puede ser útil cuando las solicitudes de memoria varían mucho en tamaño.

Desventajas:

- Fragmentación interna: asignar un bloque grande para una solicitud pequeña puede dejar una gran cantidad de memoria desaprovechada.
- Mayor tiempo de búsqueda, ya que se deben revisar todos los bloques para encontrar el más grande disponible.

Siguiente ajuste

Es una variante del primer ajuste, pero con una diferencia clave: en lugar de buscar desde el principio de la lista de bloques libres cada vez que se necesita memoria, este algoritmo continúa la búsqueda desde el último lugar donde se realizó la asignación. Si llega al final de la lista de bloques libres y no encuentra uno adecuado, vuelve al principio de la lista.

Ventajas:

- A menudo tiene un rendimiento más rápido que el primer ajuste, ya que evita buscar desde el principio cada vez.

- Puede distribuir las asignaciones más uniformemente por la memoria, lo que puede reducir la concentración de fragmentos en un área específica.

Desventajas:

- Todavía puede generar fragmentación externa, similar al primer ajuste.
- Puede ser menos eficiente que otros algoritmos en términos de minimizar el espacio libre restante, ya que la búsqueda no siempre se hace de manera óptima.

Prácticas 06 - 08

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.

```python
# 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.

Procesos e hilos

1. ¿Qué es un proceso?

Un proceso es una instancia de un programa en ejecución, que incluye el código del programa, sus datos y su estado. Cada proceso tiene su propio espacio de direcciones de memoria, lo que significa que no puede interferir directamente con otros procesos. Los procesos son gestionados por el sistema operativo, que asigna recursos como tiempo de CPU y memoria a cada uno de ellos.

2. ¿Qué son los hilos?

Los hilos, o "threads" en inglés, son las unidades más pequeñas de procesamiento que pueden ser gestionadas independientemente por un sistema operativo. Un hilo se considera una subunidad dentro de un proceso; es decir, un proceso puede contener múltiples hilos que comparten el mismo espacio de memoria y recursos del proceso principal. Esto permite que los hilos se comuniquen entre sí de manera más eficiente que los procesos independientes.

3. ¿Qué diferencia existe entre un hilo y un proceso?

La principal diferencia entre un hilo y un proceso radica en su gestión de recursos y aislamiento. Mientras que los procesos tienen su propio espacio de direcciones y son completamente independientes entre sí, los hilos comparten el mismo espacio dentro de un proceso, lo que les permite interactuar más fácilmente pero también los hace más vulnerables a errores que pueden afectar a todo el proceso. Esta diferencia implica que la creación y gestión de hilos es generalmente más ligera y rápida en comparación con la creación de nuevos procesos.

4. ¿Qué es multiprogramación?

La multiprogramación es una técnica utilizada por los sistemas operativos para ejecutar múltiples procesos simultáneamente en un solo procesador. Esto se logra mediante la alternancia rápida entre procesos, lo que da la ilusión de que se están ejecutando al mismo tiempo. La multiprogramación maximiza la utilización del CPU al mantener varios procesos en memoria y listos para ejecutarse cuando el procesador está disponible.

5. ¿Qué es paralelismo?

El paralelismo se refiere a la ejecución simultánea de múltiples procesos o hilos en diferentes núcleos o procesadores. A diferencia de la multiprogramación, donde los procesos compiten por el tiempo del CPU, el paralelismo permite que varios procesos o hilos se ejecuten realmente al mismo tiempo, aprovechando la arquitectura multiprocesador para mejorar el rendimiento general.

6. ¿Qué diferencias (de software y hardware) existen entre paralelismo y multiplicación?

Las diferencias entre paralelismo y multiprogramación pueden clasificarse tanto en términos de software como de hardware. En cuanto al software, la multiprogramación implica la gestión del tiempo compartido entre procesos, mientras que el paralelismo requiere programación concurrente para dividir tareas y ejecutarlas simultáneamente. Desde el punto de vista del hardware, la multiprogramación puede funcionar en sistemas con un solo núcleo (donde los procesos se alternan), mientras que el paralelismo necesita hardware con múltiples núcleos o procesadores para ejecutar tareas verdaderamente simultáneas.

Práctica 09

Antecedentes

Un hilo en sistemas operativos es la unidad básica de ejecución dentro de un proceso. Un proceso puede contener múltiples hilos, y todos ellos comparten la misma memoria y recursos del proceso, lo que facilita la comunicación y permite

realizar varias tareas a la vez dentro del mismo programa. Por ejemplo, en una aplicación multimedia, un hilo puede reproducir audio mientras otro procesa video. Los hilos permiten que una aplicación realice tareas en paralelo, aprovechando mejor los recursos del sistema.

Los hilos son más ligeros que los procesos y ofrecen ventajas en eficiencia y rendimiento, especialmente en sistemas con múltiples núcleos, donde pueden ejecutarse simultáneamente en diferentes núcleos. Existen hilos de usuario, gestionados por la aplicación, e hilos de kernel, controlados por el sistema operativo, que ofrecen un mejor aprovechamiento de los recursos.

Metodología

Este programa en Python utiliza las librerías tkinter y PIL (Pillow) para crear una interfaz gráfica y gestionar imágenes. tkinter es la librería estándar de Python para construir interfaces de usuario y permite abrir ventanas, añadir elementos visuales como etiquetas, y controlar eventos como el movimiento de imágenes en este caso. La librería PIL (Python Imaging Library), específicamente su submódulo ImageTk, se emplea para manipular imágenes y adaptarlas al formato que tkinter requiere.

La solución está diseñada mediante un algoritmo de movimiento en bucle, controlado por la función `mover_imagenes`. Esta función hace que las imágenes se muevan automáticamente dentro de los límites de la ventana, cambiando su dirección cada vez que llegan a un borde. El primer bloque de la función obtiene las coordenadas actuales de cada imagen, y luego actualiza sus posiciones; para `imagen1`, el desplazamiento es horizontal, mientras que `imagen2` se mueve verticalmente. Las variables `dx` y `dy` definen la velocidad del movimiento, e invierten su signo cuando una imagen alcanza el límite de la ventana.

La función `after` de tkinter llama a `mover_imagenes` cada 20 milisegundos, creando un efecto de animación continua en ambas imágenes. Finalmente, `ventana.mainloop()` mantiene la ventana abierta y en funcionamiento, esperando y ejecutando eventos. Esta estructura permite que el programa mantenga el movimiento de las imágenes dentro de la interfaz gráfica de forma fluida.

```python
def mover_imagenes():
    global dx, dy

    # Obtener posiciones actuales
    x1, y1 = etiqueta_imagen1.winfo_x(), etiqueta_imagen1.winfo_y()
    x2, y2 = etiqueta_imagen2.winfo_x(), etiqueta_imagen2.winfo_y()

    # Mover imagen1 de izquierda a derecha
    if x1 + dx > ventana.winfo_width() - 100 or x1 < 0:
        # Invertir dirección si llega al borde
        dx = -dx
    etiqueta_imagen1.place(x=x1 + dx, y=y1)

    # Mover imagen2 de arriba a abajo
    if y2 + dy > ventana.winfo_height() - 100 or y2 < 0:
        # Invertir dirección si llega al borde
        dy = -dy
    etiqueta_imagen2.place(x=x2, y=y2 + dy)

    # Llamar a la función de nuevo después de 20 ms
    ventana.after(20, mover_imagenes)
```

Problema del Productor-Consumidor

Es un problema clásico en la programación concurrente que aborda los desafíos de sincronización entre dos procesos: un productor, que genera productos y los coloca en un búfer compartido, y un consumidor, que toma esos productos para su procesamiento. La coordinación entre ambos es esencial, ya que el productor no debe exceder la capacidad del búfer y el consumidor solo puede retirar productos si hay disponibles.

Una de las reglas fundamentales es que el búfer tiene una capacidad finita, lo que significa que el productor debe esperar si está lleno y el consumidor si está vacío. Para manejar esta sincronización, se utilizan semáforos, que ayudan a coordinar el acceso al búfer y evitan condiciones de carrera.

Las soluciones suelen incluir semáforos como `mutex`, que asegura que solo un proceso acceda al búfer a la vez, y otros como `fillCount` y `emptyCount`, que indican cuántos elementos hay en el búfer y cuántos espacios están disponibles, respectivamente. Un pseudocódigo típico describe procedimientos para el productor y el consumidor, donde cada uno debe esperar en ciertas condiciones antes de acceder al búfer. Es crucial implementar correctamente estos mecanismos para evitar situaciones de deadlock, donde ambos procesos quedan bloqueados esperando uno al otro.

Aplicaciones del Problema

Una de las aplicaciones más comunes, en contextos de programación y gestión de sistemas, es en sistemas de colas, donde los productores generan datos, como mensajes o tareas, que los consumidores procesan. Esto es especialmente relevante en sistemas de mensajería y procesamiento de tareas en servidores.

Otra aplicación significativa se encuentra en el procesamiento de datos en tiempo real, donde los datos se generan y consumen continuamente, como en el análisis de flujos de datos (streaming). Aquí, el modelo ayuda a gestionar la sincronización entre la captura y el análisis de datos. Además, los sistemas operativos utilizan este enfoque para manejar la ejecución de procesos, garantizando una utilización eficiente del CPU y la memoria.

El problema también se aplica en simulaciones, como modelos de tráfico o procesos industriales, donde se necesita gestionar cómo se producen y consumen recursos a lo largo del tiempo. En el ámbito del desarrollo web y microservicios, este modelo es clave para asegurar la comunicación eficiente entre diferentes servicios que producen y consumen eventos o datos sin perder información.

En el desarrollo de juegos multijugador, el problema del productor-consumidor permite gestionar las interacciones entre jugadores y el servidor, asegurando que las acciones se procesen adecuadamente sin conflictos.

Algoritmo del Productor-Consumidor

El desarrollo del algoritmo se basa en la coordinación entre dos procesos que operan de manera concurrente: el productor, que genera datos y los almacena en un búfer, y el consumidor, que extrae esos datos para su procesamiento. La clave es garantizar que el productor no produzca más datos de los que el búfer puede manejar y que el consumidor no intente consumir datos que no están disponibles. Para lograr esto, se utilizan semáforos y mecanismos de sincronización.

En un enfoque típico, se define un semáforo `itemsReady`, que se inicializa en cero y permite al consumidor esperar hasta que haya elementos disponibles en el búfer. Cuando el productor genera un nuevo ítem, debe esperar si el búfer está lleno, utilizando otro semáforo llamado `spacesLeft`, que cuenta los espacios vacíos en el búfer. Este semáforo asegura que el productor no agregue más elementos si no hay espacio disponible. Por otro lado, cuando el consumidor consume un ítem, incrementa el contador de espacios libres, permitiendo al productor continuar su trabajo.

El algoritmo básico se puede describir mediante pseudocódigo. El productor entra en un bucle infinito donde produce un ítem, espera a que haya espacio en el búfer y luego coloca el ítem en él. El consumidor también opera en un bucle infinito, donde espera a que haya ítems disponibles antes de consumir uno. Este ciclo se repite hasta que ambos procesos finalizan su ejecución.

Además, para manejar múltiples productores y consumidores, se utiliza un semáforo adicional llamado `mutex`, que garantiza la exclusión mutua al acceder al búfer compartido. Esto evita condiciones de carrera, donde dos o más procesos intentan acceder simultáneamente a los mismos recursos. Al implementar estos mecanismos correctamente, se asegura la integridad de los datos y la eficiencia del sistema.

Un ejemplo del algoritmo en pseudocódigo es:

```
procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);   // Espera si no hay espacio
        down(mutex);        // Entra a la sección crítica
        putItemIntoBuffer(item);
```

```
        up(mutex);          // Sale de la sección crítica
        up(fillCount);      // Notifica que hay un nuevo elemento
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);    // Espera si no hay elementos
        down(mutex);        // Entra a la sección crítica
        item = removeItemFromBuffer();
        up(mutex);          // Sale de la sección crítica
        up(emptyCount);     // Notifica que hay espacio disponible
        consumeItem(item);
    }
}
```

Práctica 10

Antecedentes

Es un problema clásico en la programación concurrente que aborda los desafíos de sincronización entre dos procesos: un productor, que genera productos y los coloca en un búfer compartido, y un consumidor, que toma esos productos para su procesamiento. La coordinación entre ambos es esencial, ya que el productor no debe exceder la capacidad del búfer y el consumidor solo puede retirar productos si hay disponibles.

Una de las reglas fundamentales es que el búfer tiene una capacidad finita, lo que significa que el productor debe esperar si está lleno y el consumidor si está vacío. Para manejar esta sincronización, se utilizan semáforos, que ayudan a coordinar el acceso al búfer y evitan condiciones de carrera.

Metodología

Este código implementa una simulación de un estacionamiento utilizando multithreading y una interfaz gráfica en Python. La simulación incluye clases y métodos para gestionar la entrada y salida de autos, sincronizando las acciones concurrentes mediante hilos. La clase `Estacionamiento` encapsula la lógica principal, permitiendo añadir o retirar autos de manera sincronizada con el uso de `Lock` para evitar condiciones de carrera. Además, integra un sistema para modificar dinámicamente las frecuencias de entrada y salida mediante valores aleatorios.

La interfaz gráfica, creada con `tkinter`, permite visualizar en tiempo real el estado del estacionamiento, representando los autos como rectángulos en un lienzo (`canvas`). Cada acción de entrada o salida se refleja gráficamente, brindando una experiencia interactiva. El diseño es modular, con funciones específicas para manejar la representación gráfica y los ajustes dinámicos.

El programa utiliza multithreading para ejecutar simultáneamente la entrada de autos, la salida, y la modificación aleatoria de frecuencias, garantizando un funcionamiento fluido y realista.

```python
def agregar_auto(self):
    while True:
        with self.lock:
            if len(self.autos) < self.capacidad:
                nuevo_auto = Auto(len(self.autos) + 1)
                self.autos.append(nuevo_auto)
                self.dibujar_auto(len(self.autos) - 1)  # Dibujar auto en el canvas
                print(f"Entrada: Auto {nuevo_auto.id} añadido.")
            else:
                print("El estacionamiento está lleno. No se puede añadir más autos.")
        time.sleep(self.frecuencia_entrada)

def retirar_auto(self):
    while True:
        with self.lock:
            if len(self.autos) > 0:
                auto_retirado = self.autos.pop(0)
                self.borrar_auto()  # Borrar auto del canvas
                print(f"Salida: Auto {auto_retirado.id} retirado.")
            else:
                print("El estacionamiento está vacío. No se pueden retirar autos.")
        time.sleep(self.frecuencia_salida)
```

Problema del Lector-Escritor

Es un desafío clásico en programación concurrente y sistemas operativos, donde múltiples procesos necesitan acceder a un recurso compartido, como una base de datos. En este contexto, los "lectores" son procesos que solo leen información, mientras que los "escritores" modifican el recurso. El principal reto es que múltiples lectores pueden acceder al recurso al mismo tiempo, pero si un escritor está activo, ningún otro proceso debe acceder hasta que la escritura se complete. Esto es crucial para evitar inconsistencias en los datos.

Para abordar este problema, se utilizan soluciones como semáforos o monitores que regulan el acceso al recurso. Estas soluciones permiten que varios lectores accedan simultáneamente siempre que no haya escritores activos. Sin embargo, cuando un escritor necesita acceso, se debe garantizar que no haya lectores presentes. Esto implica establecer mecanismos de exclusión mutua para asegurar un acceso ordenado y seguro.

El estudio del problema del lector-escritor es esencial para diseñar aplicaciones eficientes y seguras en entornos multihilo o multiproceso. Implementar soluciones efectivas no solo mejora el rendimiento del sistema, sino que también asegura la integridad de los datos compartidos entre diferentes procesos.

Aplicaciones y desafíos del Problema

Uno de los desafíos más significativos es el acceso simultáneo, donde múltiples procesos intentan leer o escribir datos al mismo tiempo. Esto puede llevar a condiciones de carrera, resultando en datos corruptos o inconsistentes si no se gestiona adecuadamente.

Otro desafío importante es la prioridad entre lectores y escritores. Si se otorga preferencia a los escritores, los lectores pueden enfrentar largas esperas; por el contrario, si se priorizan los lectores, los escritores pueden quedar bloqueados indefinidamente.

Este dilema afecta el rendimiento general del sistema y puede provocar situaciones de inanición, donde un escritor nunca obtiene acceso al recurso debido a la constante presencia de lectores. Además, la complejidad en la implementación de soluciones efectivas para el problema del lector-escritor es considerable. Diseñar un sistema que equilibre el rendimiento y la seguridad de los datos puede resultar complicado y propenso a errores.

Finalmente, a medida que aumenta el número de lectores y escritores, la gestión del acceso al recurso compartido se vuelve más difícil, lo que plantea desafíos adicionales en términos de escalabilidad y optimización del sistema.

Algoritmo del Lector-Escritor

El desarrollo del algoritmo se centra en gestionar el acceso concurrente a un recurso compartido, permitiendo que múltiples lectores accedan simultáneamente, mientras que solo un escritor puede hacerlo a la vez. Para lograr esto, se utilizan semáforos y variables de control que aseguran la exclusión mutua y la sincronización entre procesos.

Primero, se inicializan semáforos para controlar el acceso al recurso. Un semáforo se encarga de la exclusión mutua, asegurando que solo un escritor o un grupo de lectores pueda acceder al recurso simultáneamente. Además, se mantiene una variable que cuenta el número de lectores activos, lo que permite determinar cuándo un escritor puede acceder al recurso.

Cuando un lector desea acceder, verifica si hay escritores activos. Si no los hay, incrementa el contador de lectores y accede al recurso. Al finalizar su lectura, decrementa el contador. Si este contador llega a cero, significa que no hay más lectores activos, lo que permite que un escritor proceda. Por otro lado, un escritor debe esperar hasta que no haya lectores activos para poder acceder al recurso.

Para evitar la inanición de los escritores, se puede implementar un sistema de prioridades que les otorgue preferencia sobre los lectores una vez que han solicitado acceso.

Este enfoque garantiza que los escritores no queden bloqueados indefinidamente por los lectores y asegura la integridad y coherencia del recurso compartido en entornos concurrentes.

Una implementación básica en pseudocódigo puede ser:

```
# Inicialización
read_count = 0
mutex = Semaphore(1)
rw_mutex = Semaphore(1)

# Lector
function reader():
    wait(mutex)
    read_count += 1
    if read_count == 1:
        wait(rw_mutex)
    signal(mutex)

    # Leer recurso compartido

    wait(mutex)
    read_count -= 1
    if read_count == 0:
        signal(rw_mutex)
    signal(mutex)

# Escritor
function writer():
    wait(rw_mutex)

    # Escribir en el recurso compartido

    signal(rw_mutex)
```