UNIVERSIDAD MAYOR DE SAN ANDRÉS

FACULTAD DE CIENCIAS PURAS Y NATURALES

CARRERA DE INFORMÁTICA

ALGORITMOS DE ADMINISTRACIÓN DEL PROCESADOR Y POLÍTICAS DE GESTIÓN DE MEMORIA

Universitario: **CI:**

-Charca Condori Ronaldo 10070217 L.P.

Materia: Sistemas Operativos

Docente: Lic. Rubén Alcón López

Paralelo: **” B ”**

La Paz -Bolivia

2024

ÍNDICE

1. Introducción.

En los sistemas operativos modernos, la gestión eficiente de los recursos, como el procesador y la memoria, es esencial para garantizar un rendimiento óptimo y una experiencia de usuario satisfactoria. Para lograr este objetivo, se emplean diversos algoritmos de planificación de procesos y políticas de asignación de memoria, cada uno diseñado para manejar diferentes patrones de carga de trabajo y requisitos del sistema.

Este proyecto tiene como objetivo simular un conjunto de estos algoritmos y políticas, evaluando su desempeño en términos de eficiencia, utilización de recursos y manejo de fragmentación. En particular, se considerarán los algoritmos de planificación de procesos FIFO, SJN y Prioridad, así como las políticas de gestión de memoria: asignación contigua simple, asignación particionada fija (con estrategias de Primer Ajuste, Mejor Ajuste y Peor Ajuste) y asignación particionada reubicable.

Esta simulación tiene como objetivo principal comprender cómo estos algoritmos y políticas afectan el rendimiento de los sistemas operativos, particularmente en términos de planificación de procesos y asignación de memoria.

2. Descripción del Conjunto de Algoritmos y Políticas Simulados.

2.1. Administración del Procesador.

Los algoritmos que se simularán son los siguientes:

-FIFO (First In, First Out):

Este algoritmo procesa las tareas en el orden en que llegan, sin considerar su duración. Es simple pero puede ser ineficiente en presencia de tareas largas.

-SJN (Shortest Job Next):

Prioriza las tareas que requieren menor tiempo de ejecución. Este enfoque minimiza el tiempo promedio de espera pero requiere conocer de antemano el tiempo de cada tarea.

-Prioridad (Priority Scheduling):

Cada tarea se procesa según su prioridad asignada. Puede ser con prioridad estática o dinámica. Si dos tareas tienen la misma prioridad, se utiliza otro criterio (como FIFO) para desempatar.

2.2. Políticas de Gestión de Memoria.

Las políticas de asignación de memoria que se simularán son las siguientes:

-Asignación Contigua Simple:

Todo el espacio de memoria disponible se asigna a un único proceso. Es sencilla, pero no permite ejecutar múltiples procesos simultáneamente.

-Asignación Particionada Fija:

La memoria se divide en particiones de tamaño fijo. Se implementarán los siguientes enfoques para asignar procesos a estas particiones:

Primer Ajuste: Asigna el proceso a la primera partición disponible que sea lo suficientemente grande.

-Mejor Ajuste: Busca la partición más pequeña que sea lo suficientemente grande para acomodar el proceso, minimizando el desperdicio de memoria.

-Peor Ajuste: Asigna el proceso a la partición más grande disponible, dejando grandes fragmentos libres.

-Asignación Particionada Reubicable:

Permite mover los procesos en memoria para compactar los espacios libres y acomodar nuevos procesos. Esto reduce la fragmentación externa, aunque introduce un costo adicional de reubicación.

La simulación de estos algoritmos y políticas permitirá analizar aspectos clave como:

-Rendimiento: Tiempo promedio de espera, tiempo de respuesta y utilización del procesador.

-Eficiencia: Uso óptimo de la memoria disponible y manejo de la fragmentación.

-Adaptabilidad: Respuesta a diferentes patrones de carga de trabajo.

Este análisis servirá para identificar cuál combinación de algoritmos y políticas es más adecuada bajo diferentes condiciones y requisitos del sistema operativo.

2.3. Configuración de la Simulación.

--Unificación de Recursos:

Las peticiones de procesos se generarán de forma que sean compatibles tanto con los algoritmos de planificación del procesador como con las políticas de gestión de memoria. Cada proceso contendrá información suficiente para ser manejado por ambos grupos de algoritmos.

-Datos por Proceso:

Cada proceso incluirá los siguientes atributos:

-Hora de llegada: Momento en el que el proceso entra al sistema.

-Tiempo de ejecución: Duración que el proceso requiere para completarse (para algoritmos de planificación del procesador).

-Prioridad: Nivel de prioridad asignado al proceso (para algoritmos basados en prioridad).

Luego tenemos otro algoritmo que genera:

Tamaño del proceso: Cantidad de memoria que el proceso requiere (para políticas de gestión de memoria).

Tamaño de memoria: Cantidad de la memoria que sirve para almacenar los procesos.

El tiempo de ejecución será generado con valores en un rango predefinido para representar procesos cortos y largos.

La prioridad será un valor entero en un rango definido (por ejemplo, de 1 a 10), donde un valor menor indica mayor prioridad.

El tamaño del proceso será un valor aleatorio dentro de los límites de la memoria disponible, dependiendo del tipo de simulación (asignación simple, particionada fija o reubicable).

2.4. Generación de Entradas.

Todos estos datos serán utilizados en los algoritmos, ya que al generar una cierta cantidad de datos estos servirán para los algoritmos como el FIFO, SJN, PRIORIDAD, Asignación continua simple, Asignación particionada(simple, mejor, peor), Asignación Particionada Reubicable.

Configuración de Memoria:

La memoria será definida en términos de su capacidad total y el tipo de partición utilizada. Por ejemplo:

Asignación contigua simple: Una única partición que cubre toda la memoria.

Asignación particionada fija: Memoria dividida en particiones de tamaño fijo.

Asignación particionada reubicable: Se permiten movimientos para compactar espacios libres.

3. Descripción de la Entrada Generada.

```
GeneradorADMINISTRACION_DE_PROCESOS.py
t_legadoa=  [26, 71, 76, 14, 66, 33, 31, 57, 74, 67, 38, 97, 94, 43, 93, 64,
, 91, 84, 31, 20, 59, 85, 49, 57, 89, 12, 48, 60, 67, 46, 8, 89, 33, 94, 52,
17, 67, 55, 76, 87, 18, 99, 29, 13, 86, 11, 55, 2, 6, 74]
t_cpu =  [6, 20, 10, 1, 20, 11, 4, 1, 4, 11, 20, 14, 20, 15, 17, 3, 9, 9, 4,
, 14, 6, 16, 1, 15, 17, 17, 19, 15, 10, 9, 11, 17, 8, 7, 7, 12, 5, 15, 6, 18,
17, 18, 16, 14, 20, 14, 17]
t_priodidad =  [1, 1, 9, 5, 10, 2, 8, 8, 5, 5, 2, 9, 8, 2, 4, 1, 9, 1, 9, 10,
8, 9, 2, 10, 2, 7, 3, 10, 10, 5, 9, 6, 3, 8, 8, 9, 2, 5, 1, 5, 6, 2, 2, 7, 1,
PS C:\Users\LENOVO\Desktop\sistemasoperativosPROyec>
```

4. Descripción y diseño de la estrategia.

Se tiene una función que analiza todos estos datos lo algoritmos

```python
import random
m=1000 #con este decidmos cuantos podemos generar#
num_procesos = 1000  # Numero de procesos a generar
rango_llegada = (0, 100)  # Rango para los tiempos de llegada
rango_servicio = (1, 20)  # Rango para los tiempos de CPU
prioti=(1,10) #valor de prioirdad

# Generación de datos en vectores
tiempos_llegada = [random.randint(*rango_llegada) for _ in range(num_procesos)]
tiempos_servicio = [random.randint(*rango_servicio) for _ in range(num_procesos)]
prioridad = [random.randint(*prioti) for _ in range(num_procesos)]

num_procesos = m
procesos = [random.randint(10, 90) for _ in range(num_procesos)]
# Generar 1000 bloques de memoria
num_bloques = m
memoria = [random.randint(100, 400) for _ in range(num_bloques)]
procesosA = ["S/A"] * num_bloques
ocupado = [False] * num_bloques
# Mostrar los primeros 10 elementos de cada vector como ejemplo
print("t_legadoa= ", tiempos_llegada[:m])
print("t_cpu = ", tiempos_servicio[:m])
print("t_priodidad = ", prioridad[:m])
# Imprimir todos los procesos y bloques de memoria en una sola línea
print("procesos:",procesos)
print("memoria:",memoria)
```

4.1.FIFO (First In, First Out).

Este algoritmo se basa en el principio de cola: el primer proceso que llega es el primero en ejecutarse. No toma en cuenta otros factores como el tiempo de ejecución o la prioridad.

Función:

Procesa los trabajos en el orden en que llegan. Es simple, pero puede generar un problema conocido como convoy effect, donde los procesos pequeños esperan mucho si un proceso largo llega primero.

Uso en la simulación:

Ordena los procesos por tiempo de llegada.

Calcula el tiempo inicial y final de cada proceso según el orden de llegada.

-Código del Algoritmo FIFO:

```
def FIFO():
    global t_0, t, t_i, t_f, T, E, I
    reloj = 1
    fila = 0
    poc = 0
    r = []
    while len(r) < len(t_0):
        t_aux = reloj
        while poc < len(t_0):
            if poc not in r and t_0[poc] <= reloj:
                minimo = t_0[poc]
                fila = poc
                for i in range(poc, len(t_0)):
                    if t_0[i] <= reloj and t_0[i] < minimo and i not in r:
                        minimo = t_0[i]
                        fila = i
                tt_i = reloj
                tt = t[fila]
                tt_f = reloj + tt - 1
                reloj += tt
                t_i[fila] = tt_i
                t_f[fila] = tt_f
                r.append(fila)
                poc = 0
            else:
                poc += 1
        if reloj == t_aux:
            reloj += 1
            poc = 0
    generarTEI_Prom()
    imprimir_resultados("FIFO")
```

4.2.SJN (Shortest Job Next).

Este algoritmo selecciona para su ejecución el proceso con el menor tiempo de ejecución que ya haya llegado al sistema. Es óptimo en cuanto a tiempo promedio de espera, pero puede causar injusticia hacia procesos largos.

Función:

Prioriza los procesos más cortos que ya están listos para ejecutarse, minimizando el tiempo de espera global.

Uso en la simulación:

Ordena dinámicamente los procesos disponibles por su tiempo de ejecución.

Elige el trabajo más corto en cada unidad de tiempo simulada.

-Código del algoritmo SJN:

```
def SJN():
    global t_0, t, t_i, t_f, T, E, I
    reloj = 0  # Reloj inicial
    r = []  # Procesos ejecutados

    while len(r) < len(t_0):
        # Buscar el proceso con menor duración que haya llegado
        candidatos = [i for i in range(len(t_0)) if i not in r and t_0[i] <= reloj]
        if candidatos:
            # Elegir el proceso con menor duración
            fila = min(candidatos, key=lambda x: t[x])

            # Calcular tiempos
            t_i[fila] = reloj
            t_f[fila] = reloj + t[fila]
            reloj += t[fila]

            # Agregar el proceso a los realizados
            r.append(fila)
        else:
            reloj += 1  # Avanzar el reloj si no hay procesos disponibles
```

4.3. Prioridad (Priority Scheduling).

Este algoritmo asigna a cada proceso una prioridad numérica, y los procesos con mayor prioridad se ejecutan primero.

Función:

Ordena los procesos según su nivel de prioridad. Si dos procesos tienen la misma prioridad, utiliza un criterio secundario, como el tiempo de llegada.

Uso en la simulación:

Asigna y evalúa prioridades a cada proceso.

Selecciona el proceso con la mayor prioridad en cada ciclo.

-Código de la algoritmo Prioridad:

```python
def Prioridad():
    global t_0, t, t_i, t_f, T, E, I, prioridad
    reloj = 0  # El reloj empieza en 0
    r = []  # Lista de procesos ya ejecutados

    while len(r) < len(t_0):  # Hasta que todos los procesos sean ejecutados
        t_aux = reloj
        fila = -1
        min_prioridad = float('inf')

        # Buscar los procesos que ya han llegado al tiempo actual del reloj
        disponibles = [i for i in range(len(t_0)) if i not in r and t_0[i] <= reloj]

        if disponibles:
            # Mientras haya procesos disponibles, elige el de menor prioridad
            for i in disponibles:
                # Buscar el proceso con menor prioridad entre los disponibles
                if prioridad[i] < min_prioridad:
                    min_prioridad = prioridad[i]
                    fila = i

            # Ejecutar el proceso con la menor prioridad
            if fila != -1:
                tt_i = reloj  # Tiempo de inicio del proceso
                tt = t[fila]  # Tiempo de ejecución del proceso
                tt_f = reloj + tt  # Tiempo de finalización del proceso (no es necesario restar 1)
                reloj += tt  # Avanzar el reloj con el tiempo de ejecución

                t_i[fila] = tt_i
                t_f[fila] = tt_f
                T[fila] = tt_f - t_0[fila]  # Tiempo total de ejecución
                E[fila] = tt_i - t_0[fila]  # Tiempo de espera
                I[fila] = round(tt / T[fila], 2) if T[fila] != 0 else 0  # Índice de eficiencia

                r.append(fila)  # Agregar el proceso a la lista de ejecutados
```

4.4. Asignación Contigua Simple.

Asigna los procesos en bloques contiguos dentro de la memoria. Si no hay suficiente espacio contiguo para una solicitud, esta se pospone.

Función: Asigna memoria de manera secuencial, revisando si hay suficiente espacio continuo para cada solicitud.

Uso en la simulación: Permite evaluar la eficiencia de la asignación contigua, destacando la fragmentación externa y el porcentaje de memoria utilizada.

-Código del Algoritmo de Asignación continua simple:

```
def asignacion_contigua_simple():
    global procesos, memoria
    memoria_total = sum(memoria)  # Calculamos el tamaño total de la memoria
    espacio_usado = 0  # Espacio total utilizado por los procesos
    trabajos_postergados = 0  # Contamos cuántos trabajos no se pudieron asignar

    print("\n====================== Asignación Contigua Simple ======================")

    # Asignamos los procesos a la memoria contigua
    for peticion in procesos:
        if espacio_usado + peticion <= memoria_total:
            print(f"Asignando {peticion}MB a la memoria contigua.")
            espacio_usado += peticion  # Sumamos el espacio usado
        else:
            print(f"No se puede asignar {peticion}MB. No hay suficiente espacio contiguo.
            trabajos_postergados += 1  # Contamos como trabajo postergado

    # Calculamos la fragmentación externa (memoria no utilizada)
    fragmentacion_externa = memoria_total - espacio_usado

    # Calculamos el porcentaje de uso de la memoria
    porcentaje_uso = (espacio_usado / memoria_total) * 100

    # Nivel de multiprogramación (número de procesos que se pudieron ejecutar)
    nivel_multiprogramacion = len(procesos) - trabajos_postergados

    # Trabajos que no pudieron ejecutarse son aquellos que no se asignaron
    trabajos_no_ejecutados = trabajos_postergados

    # Mostramos los resultados
    print(f"\nResultados de Asignación Contigua Simple:")
    print(f"Índice de fragmentación externa: {fragmentacion_externa}MB")
    print(f"Porcentaje de uso de memoria: {porcentaje_uso:.2f}%")
    print(f"Nivel de multiprogramación: {nivel_multiprogramacion}")
    print(f"Trabajos postergados: {trabajos_postergados}")
```

4.5. Partición Fija (Primer Ajuste, Mejor Ajuste, Peor Ajuste).

Descripción: Divide la memoria en bloques fijos. Los procesos se asignan a los bloques disponibles utilizando tres algoritmos: Primer Ajuste (el primero que encaje), Mejor Ajuste (el más pequeño que encaje) y Peor Ajuste (el más grande que encaje).

Función: Gestiona la memoria dividiéndola en bloques estáticos y asignando procesos según distintos criterios de ajuste.

Uso en la simulación: Permite comparar cómo cada algoritmo maneja la asignación de memoria, la fragmentación y el uso de espacio en la memoria.

-Código del algoritmo de Primer ajuste:

```python
def primer_ajuste(procesos, memoria):
    print("\n====================== Primer Ajuste ======================")

    # Lista para rastrear el espacio restante en cada bloque de memoria
    espacio_restante = memoria[:]
    asignados = [False] * len(procesos)  # Para verificar qué procesos han sido asignados
    procesos_asignados = 0  # Contador de procesos asignados

    for i, proceso in enumerate(procesos):
        asignado = False
        for j, bloque in enumerate(espacio_restante):
            if bloque >= proceso:  # Verificar si hay suficiente espacio
                print(f"Proceso {proceso}MB asignado al bloque {j+1}. Espacio restante antes: {espacio_restante[j]}MB")
                espacio_restante[j] -= proceso  # Reducir el espacio restante en el bloque
                print(f"Espacio restante después: {espacio_restante[j]}MB")
                asignados[i] = True
                procesos_asignados += 1
                asignado = True
                break
        if not asignado:
            print(f"Proceso {proceso}MB no pudo ser asignado (no hay suficiente espacio en ningún bloque).")

    # Calcular métricas
    memoria_utilizada = sum(memoria) - sum(espacio_restante)  # Memoria utilizada por los procesos
    fragmentacion_externa = sum(espacio_restante)  # Espacio no utilizado
    porcentaje_uso_memoria = (memoria_utilizada / sum(memoria)) * 100
    trabajos_postergados = len(procesos) - procesos_asignados
    nivel_multiprogramacion = procesos_asignados  # Número de procesos asignados a la memoria
```

-Código del algoritmo de Mejor Ajuste:

```python
C:\Users\LENOVO\Desktop\sistemasoperativosPROyec\MejorAjsute.py
    print(" \n====================== Mejor Ajuste ======================")

    # Lista para rastrear el espacio restante en cada bloque de memoria
    espacio_restante = memoria[:]
    asignados = [False] * len(procesos)  # Para verificar qué procesos han sido asignados
    procesos_asignados = 0  # Contador de procesos asignados

    for i, proceso in enumerate(procesos):
        asignado = False
        mejor_bloque = -1  # Indica el bloque con el mejor ajuste
        menor_despilfarro = float('inf')  # Comienza con un valor muy alto para el despilfarro

        for j, bloque in enumerate(espacio_restante):
            if bloque >= proceso:  # Verificar si hay suficiente espacio
                despilfarro = bloque - proceso  # Calcular el desperdicio en ese bloque
                if despilfarro < menor_despilfarro:  # Si el despilfarro es el menor, es el mejor bloque
                    mejor_bloque = j
                    menor_despilfarro = despilfarro

        # Si encontramos un bloque que puede alojar el proceso
        if mejor_bloque != -1:
            print(f"Proceso {proceso}MB asignado al bloque {mejor_bloque+1}. Espacio restante antes: {espacio_restante[mejor_bloque]}MB")
            espacio_restante[mejor_bloque] -= proceso  # Reducir el espacio restante en el bloque
            print(f"Espacio restante después: {espacio_restante[mejor_bloque]}MB")
            asignados[i] = True
            procesos_asignados += 1
        else:
            print(f"Proceso {proceso}MB no pudo ser asignado (no hay suficiente espacio en ningún bloque).")
```

-Código del algoritmo de Peor Ajuste:

```python
def peor_ajuste(procesos, memoria):
    print("\n====================== Peor Ajuste ======================")

    # Lista para rastrear el espacio restante en cada bloque de memoria
    espacio_restante = memoria[:]
    asignados = [False] * len(procesos)  # Para verificar qué procesos han sido asignados
    procesos_asignados = 0  # Contador de procesos asignados

    for i, proceso in enumerate(procesos):
        asignado = False
        peor_bloque = -1  # Indica el bloque con el peor ajuste (el de mayor espacio disponible)
        mayor_espacio = -1  # Comienza con un valor bajo para el mayor espacio disponible

        for j, bloque in enumerate(espacio_restante):
            if bloque >= proceso:  # Verificar si hay suficiente espacio
                if bloque > mayor_espacio:  # Si el bloque tiene más espacio que el actual mayor
                    peor_bloque = j
                    mayor_espacio = bloque

        # Si encontramos un bloque que puede alojar el proceso
        if peor_bloque != -1:
            print(f"Proceso {proceso}MB asignado al bloque {peor_bloque+1}. Espacio restante antes: {espacio_restante[peor_bloque]}MB")
            espacio_restante[peor_bloque] -= proceso  # Reducir el espacio restante en el bloque
            print(f"Espacio restante después: {espacio_restante[peor_bloque]}MB")
            asignados[i] = True
            procesos_asignados += 1
        else:
            print(f"Proceso {proceso}MB no pudo ser asignado (no hay suficiente espacio en ningún bloque).")
```

4.6. Particionada Reubicable.

Descripción: La memoria se gestiona de manera dinámica, permitiendo la reubicación de procesos en bloques libres para optimizar el uso del espacio.

Función: Asigna procesos a cualquier bloque libre disponible, permitiendo la reubicación para reducir la fragmentación.

Uso en la simulación: Evalúa la eficiencia de la reubicación de procesos y el manejo de la memoria en situaciones de alta fragmentación.

-Código del Algoritmo de Partición Reubicable:

```python
def particionada_reubicable(procesos, memoria):
    print("\n====================== Partición Reubicable ======================")

    espacio_restante = memoria[:]
    asignados = [False] * len(procesos)  # Para verificar qué procesos han sido asignados
    procesos_asignados = 0  # Contador de procesos asignados

    for i, proceso in enumerate(procesos):
        asignado = False
        print(f"\nIntentando asignar el proceso {proceso}MB")

        for j, bloque in enumerate(espacio_restante):
            if espacio_restante[j] >= proceso:
                espacio_restante[j] -= proceso
                asignados[i] = True
                procesos_asignados += 1
                asignado = True
                print(f"Proceso {proceso}MB asignado al bloque {j+1}. Espacio restante: {espacio_restante[j]}MB.")
                break

        if not asignado:
            print(f"Proceso {proceso}MB no pudo ser asignado a ningún bloque.")

    # Calcular métricas
    memoria_utilizada = sum(memoria) - sum(espacio_restante)  # Memoria utilizada por los procesos
    fragmentacion_externa = sum(espacio_restante)  # Espacio no utilizado
    porcentaje_uso_memoria = (memoria_utilizada / sum(memoria)) * 100
    trabajos_postergados = len(procesos) - procesos_asignados
    nivel_multiprogramacion = procesos_asignados  # Número de procesos asignados a la memoria
```

5.Ejecucion de los procesos.

5.1. Administración del Procesador.

-FIFO:

```
t_0: 0   t: 2    t_i: 1  t_f: 2  T: 2    E: 0    I: 1.0
t_0: 0   t: 1    t_i: 3  t_f: 3  T: 3    E: 2    I: 0.33
t_0: 1   t: 3    t_i: 4  t_f: 6  T: 5    E: 2    I: 0.6
t_0: 1   t: 2    t_i: 7  t_f: 8  T: 7    E: 5    I: 0.29
t_0: 2   t: 1    t_i: 9  t_f: 9  T: 7    E: 6    I: 0.14

Promedios de FIFO:
 t: 1.80         T: 4.80         E: 3.00         I: 0.47
PS C:\Users\LENOVO\Desktop\sistemasoperativosPROyec>
```

-JSN:

```
Resultados de SJN:
t_0: 6   t: 3    t_i: 6  t_f: 9  T: 3    E: 0    I: 1.0
t_0: 2   t: 2    t_i: 2  t_f: 4  T: 2    E: 0    I: 1.0
t_0: 3   t: 4    t_i: 12        t_f: 16        T: 13   E: 9    I: 0.31
t_0: 5   t: 3    t_i: 9  t_f: 12        T: 7    E: 4    I: 0.43
t_0: 4   t: 2    t_i: 4  t_f: 6  T: 2    E: 0    I: 1.0

Promedios de SJN:
 t: 2.80         T: 5.40         E: 2.60         I: 0.75
```

-PRIORIDAD:

```
Resultados de Prioridad:
t_0: 0   t: 2    Prioridad: 4    t_i: 7  t_f: 9  T: 9    E: 7    I: 0.22
t_0: 0   t: 1    Prioridad: 3    t_i: 0  t_f: 1  T: 1    E: 0    I: 1.0
t_0: 1   t: 3    Prioridad: 2    t_i: 3  t_f: 6  T: 5    E: 2    I: 0.6
t_0: 1   t: 2    Prioridad: 1    t_i: 1  t_f: 3  T: 2    E: 0    I: 1.0
t_0: 2   t: 1    Prioridad: 2    t_i: 6  t_f: 7  T: 5    E: 4    I: 0.2

Promedios de Prioridad:
 t: 1.80         T: 4.40         E: 2.60         I: 0.60
```

5.2. Políticas de gestión de memoria.

-Asignación continua Simple:

```
Ingresa el número de procesos: 5
Ingresa el número de bloques de memoria: 5
Ingresa el tamaño del proceso 1 (MB): 20
Ingresa el tamaño del proceso 2 (MB): 50
Ingresa el tamaño del proceso 3 (MB): 30
Ingresa el tamaño del proceso 4 (MB): 20
Ingresa el tamaño del proceso 5 (MB): 70
Ingresa el tamaño del bloque de memoria 1 (MB): 100
Ingresa el tamaño del bloque de memoria 2 (MB): 100
Ingresa el tamaño del bloque de memoria 3 (MB): 100
Ingresa el tamaño del bloque de memoria 4 (MB): 100
Ingresa el tamaño del bloque de memoria 5 (MB): 100
```

```
Resultados de Asignación Contigua Simple:
Índice de fragmentación externa: 310MB
Porcentaje de uso de memoria: 38.00%
Nivel de multiprogramación: 5
Trabajos postergados: 0
Trabajos que no pueden ejecutarse: 0
```

-Partición Fija:

-Primer Ajuste

```
Espacio restante después: 10MB
Proceso 10MB asignado al bloque 1. Espacio restante antes: 10MB
Espacio restante después: 0MB
Proceso 2000MB no pudo ser asignado (no hay suficiente espacio en ningún bloque).

====================== Resultados ======================
Fragmentación externa: 350MB
Porcentaje de uso de memoria: 48.53%
Nivel de multiprogramación: 6
Trabajos postergados: 1
Bloque 1: Espacio restante: 0MB
Bloque 2: Espacio restante: 10MB
Bloque 3: Espacio restante: 90MB
Bloque 4: Espacio restante: 150MB
Bloque 5: Espacio restante: 100MB
PS C:\Users\LENOVO\Desktop\sistemasoperativosPROyec>
```

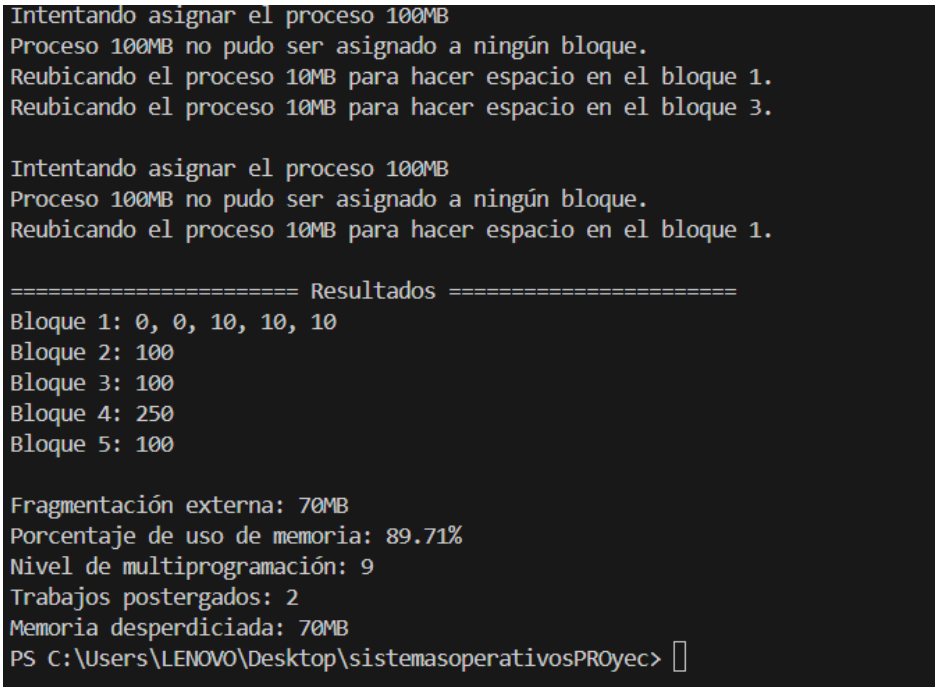
-Mejor Ajuste

```
====================== Mejor Ajuste ======================
Proceso 10MB asignado al bloque 1. Espacio restante antes: 50MB
Espacio restante después: 40MB
Proceso 10MB asignado al bloque 1. Espacio restante antes: 40MB
Espacio restante después: 30MB
Proceso 10MB asignado al bloque 1. Espacio restante antes: 30MB
Espacio restante después: 20MB
Proceso 10MB asignado al bloque 1. Espacio restante antes: 20MB
Espacio restante después: 10MB
Proceso 10MB asignado al bloque 1. Espacio restante antes: 10MB
Espacio restante después: 0MB

====================== Resultados ======================
Fragmentación externa: 630MB
Porcentaje de uso de memoria: 7.35%
Nivel de multiprogramación: 5
Trabajos postergados: 0
Bloque 1: Espacio restante: 0MB
Bloque 2: Espacio restante: 100MB
Bloque 3: Espacio restante: 180MB
Bloque 4: Espacio restante: 250MB
Bloque 5: Espacio restante: 100MB
PS C:\Users\LENOVO\Desktop\sistemasoperativosPROyec>
```

-Peor Ajuste

```
====================== Peor Ajuste ======================
Proceso 10MB asignado al bloque 4. Espacio restante antes: 250MB
Espacio restante después: 240MB
Proceso 10MB asignado al bloque 4. Espacio restante antes: 240MB
Espacio restante después: 230MB
Proceso 10MB asignado al bloque 4. Espacio restante antes: 230MB
Espacio restante después: 220MB
Proceso 10MB asignado al bloque 4. Espacio restante antes: 220MB
Espacio restante después: 210MB
Proceso 10MB asignado al bloque 4. Espacio restante antes: 210MB
Espacio restante después: 200MB

====================== Resultados ======================
Fragmentación externa: 630MB
Porcentaje de uso de memoria: 7.35%
Nivel de multiprogramación: 5
Trabajos postergados: 0
Bloque 1: Espacio restante: 50MB
Bloque 2: Espacio restante: 100MB
Bloque 3: Espacio restante: 180MB
Bloque 4: Espacio restante: 200MB
Bloque 5: Espacio restante: 100MB
PS C:\Users\LENOVO\Desktop\sistemasoperativosPROyec>
```

-Particionamiento Reubicable:



6. Conclusiones y recomendaciones.

6.1. Conclusión Gestión de Procesos.

-FIFO: Este algoritmo es simple y fácil de implementar, pero en escenarios donde existen procesos largos seguidos de procesos cortos, puede generar un problema de convoy effect. Es ideal en sistemas donde la simplicidad es prioritaria y los tiempos de ejecución son similares entre procesos.

-SJN: Ofrece un rendimiento óptimo en términos de tiempo promedio de espera, pero requiere conocer de antemano el tiempo de ejecución de los procesos, lo que no siempre es posible en entornos reales. Es adecuado para sistemas con cargas predecibles y pocos procesos largos.

-Prioridad: Es eficaz para priorizar procesos críticos, pero puede ocasionar inanición de procesos de baja prioridad. El uso de prioridades dinámicas (que aumentan con el tiempo) puede mitigar este problema y equilibrar el sistema.

6.2. Conclusión Gestión de Memoria.

-Asignación Contigua Simple: Este método es fácil de implementar, pero tiene una alta fragmentación externa y no aprovecha de manera eficiente la memoria cuando los procesos son de tamaños variables. Es adecuado para sistemas simples o específicos donde los procesos tienen un tamaño constante.

-Partición Fija:

-Primer Ajuste: Es rápido y eficiente para sistemas con una carga de trabajo regular, pero puede generar fragmentación externa si se seleccionan bloques grandes frecuentemente.

-Mejor Ajuste: Reduce la fragmentación externa al usar los bloques más pequeños posibles, pero puede ser más lento debido a la búsqueda exhaustiva del mejor bloque.

-Peor Ajuste: Tiende a dejar grandes bloques de memoria sin utilizar, siendo útil en casos donde se espera la llegada de procesos grandes posteriormente.

-Particionamiento Reubicable: Es la opción más flexible, permitiendo reorganizar la memoria para acomodar nuevos procesos y minimizar la fragmentación externa. Sin embargo, su costo computacional puede ser elevado debido al movimiento constante de procesos en memoria.

6.3. Recomendaciones.

Selección de Algoritmo según el Escenario:

Usar FIFO o Primer Ajuste en sistemas simples con baja variabilidad en la carga de trabajo.

Implementar SJN o Mejor Ajuste en sistemas donde se busca eficiencia y se puede predecir el comportamiento de los procesos.

Optar por Prioridad o Particionamiento Reubicable en sistemas críticos que requieran alta flexibilidad y priorización de tareas.