
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

INGENIERÍA EN COMPUTACIÓN

SISTEMAS OPERATIVOS

# TAREA 02: COMPARACIÓN DE PLANIFICADORES

**NOMBRE COMPLETO:** Lechuga Castillo Shareny Ixchel, Gonzalez Cuellar Pablo Arturo

**Nº de Cuenta:** 319004252, 319241013

**GRUPO DE TEORÍA:** 06

**SEMESTRE** 2025-1

**FECHA DE ENTREGA LÍMITE:** 19-11-2024

**CALIFICACIÓN: _____**

# Informe: Comparación de planificadores

## ***1. Problema:***

Debes escribir un programa que genere procesos con tiempos de llegada y ejecución aleatorios, simule su planificación usando varios algoritmos (FCFS, RR, SPN, FB, y SRR) y compare el rendimiento de cada uno en al menos cinco rondas diferentes. Deberás calcular y mostrar métricas como tiempo promedio de retorno (T), espera (E), y penalización (P) para cada algoritmo en cada ronda, así como una representación visual del orden de ejecución de los procesos. Por último, deberás validar manualmente algunos resultados para asegurarte de que sean correctos.

## ***2. Lenguaje y Entorno:***

- Lenguaje: Python.
- Entorno de Desarrollo: Python 3.7+.
- Librerías utilizadas:

  random: Para generar los tiempos de llegada y ráfagas de los procesos de manera aleatoria.

  Funciones nativas de Python, sin necesidad de librerías externas adicionales.

## ***3. Instrucciones para ejecutar el programa:***

-Descarga el archivo


-Asegúrate de tener Python 3.7 o superior instalado en tu máquina.
-Abre una terminal y navega al directorio donde descargaste el archivo.
-Ejecuta el programa usando el comando:

```
C:\Users\pablo\Documents\SOGeneral\Tarea2>python tarea2_CompPlan.py
```

## ***4. ¿Qué esperar del programa?***

1. Generación de procesos: El programa genera un número aleatorio de procesos (por defecto 6 por ronda) con:
   - Tiempo de llegada: Momento en el que el proceso se presenta al sistema.
   - Tiempo de ráfaga: Duración requerida para completar el proceso.
2. Simulación de algoritmos de planificación: El programa ejecutará los siguientes algoritmos:
   - FCFS: Planificación según el orden de llegada.
   - Round Robin (quantum 1 y 4): Tiempo compartido entre procesos.

- SPN: Ejecución del proceso más corto primero.

3. Resultados por ronda: Para cada algoritmo y conjunto de procesos, el programa mostrará:
   - Tiempo promedio de retorno (T): Tiempo que tarda un proceso desde que llega hasta que termina.
   - Tiempo promedio de espera (E): Tiempo total que un proceso espera antes de ser ejecutado.
   - Penalización promedio (P): Relación entre el tiempo de retorno y el tiempo de ráfaga.
   - Secuencia de ejecución: Representación gráfica de los procesos ejecutados en la línea de tiempo.
4. Ciclo continuo por rondas: El programa ejecutará 5 rondas de simulación por defecto.

## ***5. Explicación del programa:***

```python
# Clase para crear un proceso
class Proceso:
    def __init__(self, nombre, tiempo_llegada, tiempo_rafaga):
        self.nombre = nombre
        self.tiempo_llegada = tiempo_llegada
        self.tiempo_rafaga = tiempo_rafaga
        self.tiempo_restante = tiempo_rafaga
        self.tiempo_finalizacion = 0
        self.tiempo_espera = 0
        self.tiempo_retorno = 0
```

La clase Proceso representa un modelo de procesos en planificación, inicializando atributos clave como el nombre, tiempo_llegada (momento de entrada al sistema), tiempo_rafaga (duración total requerida) y tiempo_restante (igual a tiempo_rafaga, para reducirse en algoritmos como Round Robin). También define atributos inicializados en 0, como tiempo_finalizacion (momento en que el proceso termina), tiempo_espera (tiempo total en cola antes de ser ejecutado) y tiempo_retorno (duración desde la llegada hasta la finalización). El constructor organiza estos valores para que los algoritmos puedan manipularlos fácilmente durante las simulaciones.

```python
# Generar procesos
def generar_procesos(numero_procesos):
    nombres = [chr(65 + i) for i in range(numero_procesos)]
    procesos = []
    for nombre in nombres:
        tiempo_llegada = random.randint(0, 10)  # Llegadas entre 0 y 10
        tiempo_rafaga = random.randint(1, 10)   # Ráfagas entre 1 y 10
        procesos.append(Proceso(nombre, tiempo_llegada, tiempo_rafaga))
    return procesos
```

Este fragmento de código define la función generar_procesos(numero_procesos), que crea una lista de procesos simulados. Primero, genera nombres únicos (como A, B, C, etc.) según el número de procesos solicitado, utilizando caracteres ASCII. Luego, para cada nombre, asigna un tiempo_llegada (entre 0 y 10) y un tiempo_rafaga (entre 1 y 10) de manera aleatoria usando la función random.randint. Finalmente, crea un objeto Proceso con esos valores y lo agrega a la lista procesos. Al terminar, retorna la lista completa de procesos generados.

```
# Algoritmo FCFS
def fcfs(procesos):···

# Algoritmo Round Robin
def round_robin(procesos, quantum):···

# Algoritmo SPN
def spn(procesos):···
```

```
# Funcion para calcular métricas
def calcular_metricas(procesos):
    total_retorno = sum(p.tiempo_retorno for p in procesos)
    total_espera = sum(p.tiempo_espera for p in procesos)
    total_penalizacion = sum(p.tiempo_retorno / p.tiempo_rafaga for p in procesos)
    n = len(procesos)
    return total_retorno / n, total_espera / n, total_penalizacion / n
```

La función calcular_metricas(procesos) calcula los promedios de tiempo de retorno, espera y penalización dividiendo la suma de cada métrica (retorno, espera y la relación retorno/ráfaga) entre el número total de procesos (n), permitiendo evaluar el desempeño del algoritmo de planificación.

```
# Crear copias independientes de los procesos
def copiar_procesos(procesos):
    return [Proceso(p.nombre, p.tiempo_llegada, p.tiempo_rafaga) for p in procesos]
    #cada algoritmo trabaja con sus propios procesos
```

```python
# Algoritmo FCFS
def fcfs(procesos):
    procesos.sort(key=lambda p: p.tiempo_llegada)
    tiempo_actual = 0
    secuencia = ""
    for p in procesos:
        if tiempo_actual < p.tiempo_llegada:
            tiempo_actual = p.tiempo_llegada
        secuencia += p.nombre * p.tiempo_rafaga
        p.tiempo_finalizacion = tiempo_actual + p.tiempo_rafaga
        p.tiempo_retorno = p.tiempo_finalizacion - p.tiempo_llegada
        p.tiempo_espera = p.tiempo_retorno - p.tiempo_rafaga
        tiempo_actual += p.tiempo_rafaga
    return procesos, secuencia
```

La función fcfs(procesos) implementa el algoritmo First Come, First Served (FCFS), que ejecuta los procesos en el orden de su tiempo de llegada (tiempo_llegada). Primero, ordena los procesos según su tiempo de llegada. Luego, recorre cada proceso, actualizando el tiempo actual y asignando sus atributos: tiempo_finalizacion (momento en que termina), tiempo_retorno (desde que llega hasta que termina) y tiempo_espera (tiempo total en la cola). También construye una representación visual (secuencia) que indica la ejecución continua del proceso por su nombre repetido según la ráfaga. Al final, retorna la lista de procesos actualizados y la secuencia de ejecución.

```python
# Algoritmo Round Robin
def round_robin(procesos, quantum):
    cola = [p for p in sorted(procesos, key=lambda p: p.tiempo_llegada)]
    tiempo_actual = 0
    completados = []
    secuencia = ""
    while cola:
        proceso = cola.pop(0)
        if proceso.tiempo_llegada > tiempo_actual:
            tiempo_actual = proceso.tiempo_llegada
        if proceso.tiempo_restante > quantum:
            proceso.tiempo_restante -= quantum
            tiempo_actual += quantum
            secuencia += proceso.nombre * quantum
            cola.append(proceso)
        else:
            tiempo_actual += proceso.tiempo_restante
            secuencia += proceso.nombre * proceso.tiempo_restante
            proceso.tiempo_restante = 0
            proceso.tiempo_finalizacion = tiempo_actual
            proceso.tiempo_retorno = proceso.tiempo_finalizacion - proceso.tiempo_llegada
            proceso.tiempo_espera = proceso.tiempo_retorno - proceso.tiempo_rafaga
            completados.append(proceso)
    return completados, secuencia
```

La función round_robin(procesos, quantum) implementa el algoritmo Round Robin, distribuyendo el tiempo de CPU equitativamente según un quantum. Ordena los procesos por llegada y los ejecuta en ciclos, reduciendo su tiempo restante o completándolos según el quantum. Actualiza la secuencia de ejecución y calcula métricas como tiempo_finalizacion, tiempo_retorno y tiempo_espera. Retorna los procesos completados y la secuencia de ejecución.

Y por último:

```python
# Algoritmo SPN
def spn(procesos):
    cola_listos = []
    tiempo_actual = 0
    completados = []
    secuencia = ""
    procesos.sort(key=lambda p: p.tiempo_llegada)

    while procesos or cola_listos:
        while procesos and procesos[0].tiempo_llegada <= tiempo_actual:
            cola_listos.append(procesos.pop(0))
        if cola_listos:
            cola_listos.sort(key=lambda p: p.tiempo_rafaga)
            proceso = cola_listos.pop(0)
            tiempo_actual += proceso.tiempo_rafaga
            secuencia += proceso.nombre * proceso.tiempo_rafaga
            proceso.tiempo_finalizacion = tiempo_actual
            proceso.tiempo_retorno = proceso.tiempo_finalizacion - proceso.tiempo_llegada
            proceso.tiempo_espera = proceso.tiempo_retorno - proceso.tiempo_rafaga
            completados.append(proceso)
        else:
            tiempo_actual += 1
    return completados, secuencia
```

La función spn(procesos) implementa el algoritmo Shortest Process Next (SPN), que ejecuta primero el proceso con la ráfaga más corta entre los disponibles. Ordena los procesos por tiempo de llegada, agregándolos a una cola de listos conforme llegan. En cada ciclo, selecciona el proceso con la menor ráfaga en la cola, lo ejecuta completamente y actualiza sus métricas (tiempo_finalizacion, tiempo_retorno, tiempo_espera) y la secuencia de ejecución. Si no hay procesos listos, el tiempo actual avanza hasta que llegue el siguiente. Al final, retorna los procesos completados y la secuencia de ejecución.