Planificación de Procesos en Sistemas Operativos

Andrey Hegel Zafra Venegas, Estefano Alexis Ramirez Garcia, Llontop Falcon Henry Alessandro,
Maleck Ramirez Alvarez, Luis Guillermo Quispe Alvarado

Abstract

Este trabajo presenta un análisis comparativo de diversos algoritmos de planificación de procesos en sistemas operativos, incluyendo FCFS, SJF, Round-Robin y Prioridad. Se implementaron simulaciones que permiten evaluar el desempeño de cada algoritmo bajo distintos escenarios de carga y tipos de procesos, considerando métricas fundamentales como tiempo de retorno, tiempo de espera, tiempo de respuesta, throughput y equidad. Los resultados muestran cómo cada política de planificación afecta la eficiencia y equidad del uso del CPU, proporcionando una base para seleccionar algoritmos adecuados según las características de la carga de trabajo y los requisitos del sistema.

1. Introducción

La planificación de procesos es un componente esencial de cualquier sistema operativo, encargado de decidir el orden en que los procesos compiten por el acceso al CPU. Una planificación eficiente impacta directamente en el rendimiento del sistema, la equidad en el uso de recursos y la experiencia del usuario. Existen múltiples estrategias de planificación, cada una con ventajas y limitaciones: FCFS (First Come, First Served) es simple y fácil de implementar, SJF (Shortest Job First) minimiza el tiempo de espera promedio, Round-Robin distribuye equitativamente el CPU entre procesos, y los algoritmos basados en prioridad permiten adaptar la ejecución según la importancia de cada tarea.

El objetivo de este trabajo es implementar y evaluar estos algoritmos mediante simulaciones controladas que consideren diferentes escenarios de carga, tipos de procesos y patrones de llegada. Se busca cuantificar su desempeño a través de métricas clásicas de sistemas operativos, ofreciendo un análisis comparativo que sirva de guía para seleccionar la política de planificación más adecuada.

2. Metodología

Para evaluar el desempeño de los algoritmos de planificación de procesos —FCFS, SJF, Round-Robin y Prioridad— se implementaron simulaciones en Python que modelan la llegada, ejecución y finalización de procesos en tiempo discreto. Se generaron escenarios controlados que incluyen mezclas de procesos cortos y largos, procesos CPU-bound e I/O-bound, y entornos de baja y alta concurrencia. Cada algoritmo se ejecutó múltiples veces con distintas semillas pseudoaleatorias para garantizar reproducibilidad, registrando métricas clave como tiempo de retorno, tiempo de espera, tiempo de respuesta, throughput y equidad. Los resultados se presentan en tablas comparativas y diagramas de Gantt para facilitar el análisis de la eficiencia y equidad de cada planificador.

3. Algoritmo Implementado

3.1. Algoritmo FCFS

El algoritmo **FCFS** (First Come, First Served) se implementó considerando los tiempos de llegada y la duración de cada proceso. En este enfoque, los procesos se atienden estrictamente en el orden en que llegan a la cola de listos, sin importar su tiempo de ejecución o tipo de recurso requerido (CPU o I/O). Este método refleja una política sencilla y justa de planificación, aunque puede generar tiempos de espera elevados cuando existen procesos largos al inicio de la cola (Grosof et al., 2021).

```
def simular_fcfs(procesos_prueba):
    procesos = [p.copy() for p in
        ↪ procesos_prueba]
    tiempo_actual = 0
    cola_listos = []
    terminados = []
    pendientes = sorted(procesos, key=
        ↪ lambda x: x["llegada"])
    idx = 0
```

```
    while idx < len(pendientes) or
        ↪ cola_listos:
        while idx < len(pendientes) and
            ↪ pendientes[idx]["llegada"
            ↪ ] <= tiempo_actual:
            cola_listos.append(
                ↪ pendientes[idx])
            idx += 1

        cola_listos.sort(key=lambda x: x
            ↪ ["llegada"])

        if cola_listos:
            p = cola_listos.pop(0)
            inicio = max(tiempo_actual,
                ↪ p["llegada"])
            fin = inicio + p["duracion"]
            retorno = fin - p["llegada"]
            espera = inicio - p["llegada
                ↪ "]
            respuesta = espera

            p.update({
                "inicio": inicio,
                "fin": fin,
                "retorno": retorno,
                "espera": espera,
                "respuesta": respuesta
            })

            terminados.append(p)
            tiempo_actual = fin
        else:
            tiempo_actual = pendientes[
                ↪ idx]["llegada"]

    return terminados
```

3.2. Algoritmo SJF

El algoritmo **SJF** (Shortest Job First) se implementó considerando tiempos de llegada y duración. En cada instante, el proceso con menor tiempo de ejecución pendiente es seleccionado (Hernández Abreu et al., 2025).

```
def simular_sjf(procesosPrueba):
    procesos = [p.copy() for p in
        ↪ procesosPrueba]
    tiempo_actual = 0
    terminados = []

    pendientes = sorted(procesos, key=
        ↪ lambda x: x["llegada"])

    while pendientes:
        llegados = [p for p in
            ↪ pendientes if p["llegada"
            ↪ ] <= tiempo_actual]
        if not llegados:
            tiempo_actual = pendientes
                ↪ [0]["llegada"]
            llegados = [p for p in
                ↪ pendientes if p["
                ↪ llegada"] <=
                ↪ tiempo_actual]
        elegido = min(llegados, key=
            ↪ lambda x: x["duracion"])
        inicio = max(tiempo_actual,
            ↪ elegido["llegada"])
        fin = inicio + elegido["duracion
            ↪ "]
        retorno = fin - elegido["llegada
            ↪ "]
        espera = inicio - elegido["
            ↪ llegada"]
        respuesta = espera

        elegido.update({
            "inicio": inicio,
            "fin": fin,
            "retorno": retorno,
            "espera": espera,
            "respuesta": respuesta
        })

        tiempo_actual = fin
        terminados.append(elegido)
        pendientes.remove(elegido)
```

3.3. Algoritmo Round Robin

El algoritmo Round Robin (RR) es un método de planificación preventivo que asigna un intervalo de tiempo fijo denominado quantum de tiempo (TQ) a cada proceso en la cola de listos. Cuando un proceso agota su quantum asignado, es interrumpido y reubicado al final de la cola, permitiendo que el siguiente proceso utilice la CPU. Este enfoque garantiza que todos los procesos reciban una porción equitativa del tiempo de procesamiento de forma cíclica, sin importar su duración o prioridad. Round Robin es particularmente efectivo en sistemas operativos de tiempo compartido y tiempo real, ya que proporciona tiempos de respuesta reducidos y previene la inanición de procesos, aunque su eficiencia depende significativamente de la selección apropiada del quantum de tiempo (Omar et al., 2021).

```
def simular_round_robin(procesosPrueba,
    ↪ quantum):
    procesos = [p.copy() for p in
        ↪ procesosPrueba]
    tiempo_actual = 0
    cola = []
    terminados = []

    for p in procesos:
        p["restante"] = p["duracion"]

    while True:
        for p in procesos:
            if (p["llegada"] ==
                ↪ tiempo_actual
                and p["restante"] > 0
                and p["id"] not in [proc
                    ↪ ["id"] for proc
                    ↪ in cola]
```

```
                and p["id"] not in [proc
                    ↪ ["id"] for proc
                    ↪ in terminados]):
                cola.append(p)

        if cola:
            actual = cola.pop(0)
            if "inicio" not in actual:
                actual["inicio"] =
                    ↪ tiempo_actual
                actual["respuesta"] =
                    ↪ tiempo_actual -
                    ↪ actual["llegada"]

            ejecucion = min(quantum,
                ↪ actual["restante"])
            inicio = tiempo_actual
            fin = tiempo_actual +
                ↪ ejecucion
            tiempo_actual = fin
            actual["restante"] -=
                ↪ ejecucion

            for p in procesos:
                if (inicio < p["llegada"
                    ↪ ] <= fin
                    and p["id"] not in [
                        ↪ proc["id"]
                        ↪ for proc in
                        ↪ cola]
                    and p["restante"] >
                        ↪ 0
                    and p["id"] not in [
                        ↪ proc["id"]
                        ↪ for proc in
                        ↪ terminados]):
                    cola.append(p)

            if actual["restante"] == 0:
                actual["fin"] =
                    ↪ tiempo_actual
                actual["retorno"] =
                    ↪ actual["fin"] -
                    ↪ actual["llegada"]
                actual["espera"] =
                    ↪ actual["retorno"]
                    ↪  - actual["
                    ↪ duracion"]
                if actual["id"] not in [
                    ↪ p["id"] for p in
                    ↪ terminados]:
                    terminados.append(
                        ↪ actual)
            else:
                cola.append(actual)
        else:
            if any(p["restante"] > 0 for
                ↪  p in procesos):
                tiempo_actual += 1
                continue
            else:
                break

    return terminados
```

3.4. Algoritmo de Prioridad (Estática y Dinámica con Envejecimiento)

El algoritmo de **Planificación por Prioridades** se basa en asignar a cada proceso un nivel de prioridad. En la versión **estática**, las prioridades no cambian durante la ejecución. En la versión **dinámica**, se aplica *envejecimiento* (*aging*), incrementando la prioridad de los procesos que llevan mucho tiempo esperando para evitar inanición (Balbastre et al., 2015) (Sánchez Rodríguez, 2014) .

El proceso con menor valor de prioridad (mayor importancia) es siempre seleccionado. Si un proceso con mejor prioridad llega mientras otro se ejecuta, ocurre **preemption**.

```
class Proceso:
    def __init__(self, pid, llegada,
        ↪ rafaga, prioridad_estatica):
        self.pid = pid
        self.llegada = llegada
        self.rafaga = rafaga
        self.prioridad_estatica =
            ↪ prioridad_estatica
        self.prioridad_dinamica =
            ↪ prioridad_estatica
        self.tiempo_restante = rafaga
        self.tiempo_inicio = -1
        self.tiempo_finalizacion = 0
        self.tiempo_espera = 0

def simular_planificacion(
    ↪ lista_procesos_inicial,
    ↪ tipo_algoritmo):
    procesos = copy.deepcopy(
        ↪ lista_procesos_inicial)
    tiempo_actual = 0
    cola_listos, procesos_terminados =
        ↪ [], []
    proceso_en_ejecucion = None
    procesos_por_llegar = sorted(
        ↪ procesos, key=lambda p: p.
        ↪ llegada)

  while procesos_por_llegar or
      ↪ cola_listos or
      ↪ proceso_en_ejecucion:
        while procesos_por_llegar and
            ↪ procesos_por_llegar[0].
            ↪ llegada <= tiempo_actual:
            cola_listos.append(
                ↪ procesos_por_llegar.
                ↪ pop(0))

  elegido = seleccionar_proceso(
      ↪ cola_listos, tipo_algoritmo)

        if proceso_en_ejecucion and
            ↪ elegido:
         mejor_en_cola = (
             (tipo_algoritmo == '
                 ↪ estatica' and
             elegido.prioridad_estatica
                 ↪ <
                 ↪ proceso_en_ejecucion
```

```
                ↪ .prioridad_estatica)
                ↪  or
            (tipo_algoritmo == 'dinamica
                ↪ ' and
             elegido.prioridad_dinamica
                ↪ <
                ↪ proceso_en_ejecucion
                ↪ .prioridad_dinamica)
        )
        if mejor_en_cola:
            cola_listos.append(
                ↪ proceso_en_ejecucion
                ↪ )
            proceso_en_ejecucion =
                ↪ elegido
            cola_listos.remove(elegido)
            if proceso_en_ejecucion.
                ↪ tiempo_inicio == -1:
                proceso_en_ejecucion.
                    ↪ tiempo_inicio =
                    ↪ tiempo_actual
    elif elegido:
        proceso_en_ejecucion = elegido
        cola_listos.remove(elegido)
        if proceso_en_ejecucion.
            ↪ tiempo_inicio == -1:
            proceso_en_ejecucion.
                ↪ tiempo_inicio =
                ↪ tiempo_actual

    if proceso_en_ejecucion:
        proceso_en_ejecucion.
            ↪ tiempo_restante -= 1
        if proceso_en_ejecucion.
            ↪ tiempo_restante == 0:
            proceso_en_ejecucion.
                ↪ tiempo_finalizacion
                ↪  = tiempo_actual +
                ↪ 1
            procesos_terminados.append(
                ↪ proceso_en_ejecucion
                ↪ )
            proceso_en_ejecucion = None

    for p in cola_listos:
        p.tiempo_espera += 1
    if tipo_algoritmo == 'dinamica':
        envejecer_procesos_en_espera(
            ↪ cola_listos,
            ↪ proceso_en_ejecucion)

     tiempo_actual += 1

    return procesos_terminados
```

Este algoritmo fue implementado para comparar el rendimiento entre las estrategias **de prioridad estática y dinámica**, evaluando los tiempos de espera y retorno mediante gráficos generados en `Matplotlib` y tablas de resultados en `Pandas`.

4. Escenarios de Prueba

Para evaluar de manera integral el desempeño de los algoritmos de planificación de procesos —FCFS (First Come, First Served), SJF (Shortest Job First), Round-Robin (RR) y Prioridad (en sus variantes estática y dinámica)— se diseñaron diversos escenarios que representan condiciones de carga típicas en sistemas operativos. Cada escenario busca analizar cómo cada política responde ante distintos patrones de llegada, duración y comportamiento de los procesos.

En todos los casos se utilizaron conjuntos de procesos generados sintéticamente, con atributos controlados (tiempo de llegada, duración, prioridad y tipo de proceso) y se ejecutaron múltiples instancias independientes para obtener resultados estadísticamente estables.

Los escenarios definidos fueron los siguientes:

4.1. Escenario A: Mezcla de procesos cortos y largos

Descripción. Este escenario simula un entorno mixto donde coexisten procesos de corta duración (rápidos) y procesos de ejecución prolongada. El objetivo es analizar cómo los distintos algoritmos gestionan la coexistencia de trabajos de diversa duración, especialmente en cuanto al tiempo de espera promedio y a la posible inanición de procesos largos.

Generación de procesos.

- **Número de procesos por ejecución:** entre 5 y 10.
- **Distribución de duraciones:** 50 % procesos cortos (1–4 unidades) y 50 % procesos largos (8–20 unidades).
- **Tiempos de llegada:** en el rango [0, 3] para inducir competencia temprana.
- **Número de ejecuciones:** 3 instancias distintas por algoritmo.

Objetivos. Evaluar la eficiencia de cada planificador frente a cargas heterogéneas. En particular:

- SJF debería minimizar el tiempo de espera promedio.
- FCFS puede provocar esperas elevadas para procesos cortos.
- Round-Robin debería equilibrar tiempos de respuesta.
- El algoritmo de Prioridad podría favorecer selectivamente a ciertos procesos según su prioridad asignada.

4.2. Escenario B: Procesos CPU-bound vs I/O-bound

Descripción. Se comparan procesos intensivos en CPU (CPU-bound) con procesos que dependen fuertemente de operaciones de entrada/salida (I/O-bound). Este escenario evalúa la capacidad de los algoritmos para mantener la equidad y eficiencia cuando existen ráfagas de CPU intercaladas con tiempos de bloqueo.

Generación de procesos.

- **Número de procesos:** entre 6 y 10.
- **Tipos:** 5 CPU-bound (10–25 unidades de CPU) y 5 I/O-bound (ráfagas de 1–3 unidades con pausas simuladas).
- **Tiempos de llegada:** uniformemente distribuidos en [0,10].
- **Número de ejecuciones:** 4 por algoritmo.

Objetivos. Analizar:

- La latencia de respuesta de procesos I/O-bound bajo RR y SJF.
- La eficiencia global de CPU (porcentaje de tiempo útil).
- Cómo las políticas de prioridad dinámica mejoran la equidad frente a la versión estática.

4.3. Escenario C: Alta concurrencia vs baja concurrencia

Descripción. Este escenario explora el impacto de la carga del sistema. Se comparan entornos de baja concurrencia (pocos procesos activos simultáneamente) con entornos de alta concurrencia (múltiples procesos compitiendo por CPU).

Generación de procesos.

- **Baja concurrencia:** 4–6 procesos; llegadas espaciadas (rango [0,20]).
- **Alta concurrencia:** 20–30 procesos; llegadas agrupadas (rango [0,5]).
- **Duraciones:** aleatorias en [1, 10].
- **Número de ejecuciones:** 3 por algoritmo.

Objetivos.

- Evaluar la escalabilidad y estabilidad de cada algoritmo bajo distintas cargas.
- Medir el tiempo medio de espera y throughput.
- Observar la variabilidad (desviación estándar) de los tiempos de retorno.

4.4. Medición y reproducibilidad

Para todos los escenarios se registraron métricas estándar: tiempo de retorno, tiempo de espera, tiempo de respuesta, y throughput global. Cada ejecución genera un conjunto de datos que incluye:

- Tabla detallada con turnaround time, tiempo de espera, tiempo de respuesta, throughput y fairness por proceso.
- Gráfico de Gantt que representa visualmente la secuencia de ejecución.

Se ejecutaron entre 3 y 4 repeticiones por escenario para cada algoritmo, con distintas semillas pseudoaleatorias. Todos los experimentos y scripts de generación se encuentran disponibles en el repositorio del proyecto para garantizar la reproducibilidad de los resultados.

5. Métricas de Evaluación

Para analizar de manera objetiva el rendimiento de los algoritmos de planificación implementados (FCFS, SJF, Round-Robin y Prioridad), se emplearon diversas métricas clásicas en la evaluación de sistemas operativos. Cada métrica captura un aspecto específico del comportamiento del planificador, permitiendo comparar eficiencia, equidad y capacidad de respuesta bajo diferentes condiciones de carga.

5.1. Tiempo de retorno (Turnaround Time)

El **tiempo de retorno** mide la duración total que transcurre desde la llegada del proceso hasta su finalización. Se calcula como:

$$T_{retorno} = T_{fin} - T_{llegada}$$

Esta métrica refleja el tiempo total que un proceso permanece en el sistema (esperando y ejecutando). Un valor bajo indica que el sistema procesa rápidamente las tareas. Es particularmente importante en entornos batch o de cómputo por lotes.

5.2. Tiempo de espera (Waiting Time)

El **tiempo de espera** corresponde al período durante el cual un proceso permanece en la cola de listos antes de ser ejecutado por la CPU:

$$T_{espera} = T_{inicio} - T_{llegada}$$

En otras palabras, representa el tiempo "inactivo" del proceso. Los algoritmos como SJF tienden a minimizar esta métrica, mientras que FCFS puede generar esperas prolongadas para procesos cortos. Valores altos de espera son un indicador de posible inanición o planificación ineficiente.

5.3. Tiempo de respuesta (Response Time)

El **tiempo de respuesta** se define como el intervalo desde la llegada del proceso hasta que recibe CPU por primera vez:

$$T_{respuesta} = T_{\text{primera ejecución}} - T_{llegada}$$

Esta métrica es crítica en sistemas interactivos o de tiempo compartido, donde la percepción de rapidez por parte del usuario depende de la prontitud con la que un proceso obtiene CPU. El algoritmo Round-Robin, al repartir de forma cíclica los cuantums, busca precisamente minimizar esta métrica para garantizar equidad en sistemas multitarea.

5.4. Throughput (Productividad del sistema)

El **throughput** mide la cantidad de procesos completados por unidad de tiempo:

$$Throughput = \frac{\text{Número de procesos completados}}{\text{Tiempo total de simulación}}$$

Una política de planificación eficiente debe maximizar este valor. Round-Robin y Prioridad dinámica suelen ofrecer mejores resultados de throughput en escenarios de alta concurrencia, al mantener la CPU constantemente ocupada.

5.5. Equidad (Fairness)

La **equidad** evalúa la distribución balanceada de los recursos entre procesos, especialmente relevante en Round-Robin y Prioridad. Se cuantifica mediante la varianza de los tiempos de espera o tiempos de respuesta:

$$Fairness = \begin{cases} \dfrac{min(T_{retorno})}{max(T_{retorno})}, & \text{si } max(T_{retorno}) > 0 \\ 0, & \text{en caso contrario} \end{cases}$$

Valores cercanos a 1 indican un reparto equitativo del tiempo de CPU. El algoritmo de Prioridad dinámica incorpora ajustes automáticos en los valores de prioridad para mejorar esta métrica cuando detecta procesos penalizados.

5.6. Resumen del uso de métricas

Cada algoritmo fue evaluado con todas las métricas anteriores en los tres escenarios definidos (mezcla de procesos, CPU-bound/I/O-bound y concurrencia variable). Los resultados de las métricas se agregaron en tablas comparativas y diagramas de barras para facilitar la interpretación visual.

En conjunto, estas métricas proporcionan una visión completa del rendimiento de cada planificador desde tres perspectivas fundamentales:

- **Eficiencia:** tiempo medio de espera, retorno y throughput.
- **Interactividad:** tiempo de respuesta y equidad.

6. Resultados y Discusión

6.1. Algoritmo FCFS

6.1.1. Escenario A: Mezcla de procesos cortos y largos

- Promedio de retorno: 63.64
- Promedio de espera: 55.29
- Promedio de respuesta: 55.29
- Throughput: 0.12
- Fairness: 0.12

6.1.2. Escenario B: Procesos de CPU-bound vs I/O-bound

- Promedio de retorno: 45.73
- Promedio de espera: 36.09
- Promedio de respuesta: 36.09
- Throughput: 0.10
- Fairness: 0.04

6.1.3. Escenario C: Alta concurrencia vs Baja concurrencia

- Promedio de retorno: 6.33
- Promedio de espera: 0.00
- Promedio de respuesta: 0.00
- Throughput: 0.12
- Fairness: 0.62

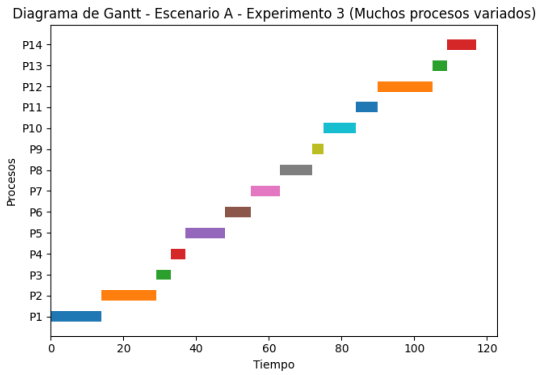

Figura 1: Diagrama de Gantt generado con el algoritmo FCFS para el escenario A.

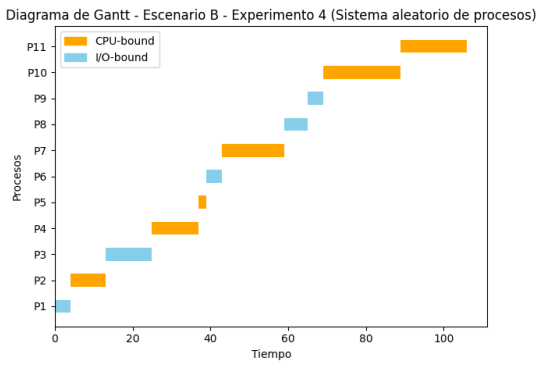

Figura 2: Diagrama de Gantt generado con el algoritmo FCFS para el escenario B.

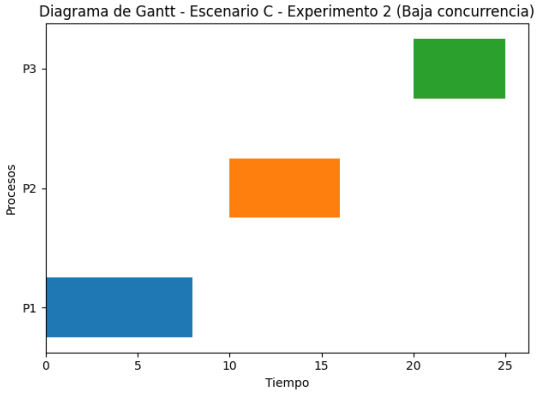

Figura 3: Diagrama de Gantt generado con el algoritmo FCFS para el escenario C.

6.2. Algoritmo SJF

6.2.1. Escenario A: Mezcla de procesos cortos y largos

- Promedio de retorno: 148.80
- Promedio de espera: 108.30
- Promedio de respuesta: 108.30
- Throughput: 0.02
- Fairness: 0.01

6.2.2. Escenario B: Procesos de CPU-bound vs I/O-bound

- Promedio de retorno: 48.85
- Promedio de espera: 41.70
- Promedio de respuesta: 41.70
- Throughput: 0.14
- Fairness: 0.08

6.2.3. Escenario C: Alta concurrencia vs Baja concurrencia

- Promedio de retorno: 31.60
- Promedio de espera: 24.95
- Promedio de respuesta: 24.95
- Throughput: 0.15
- Fairness: 0.01

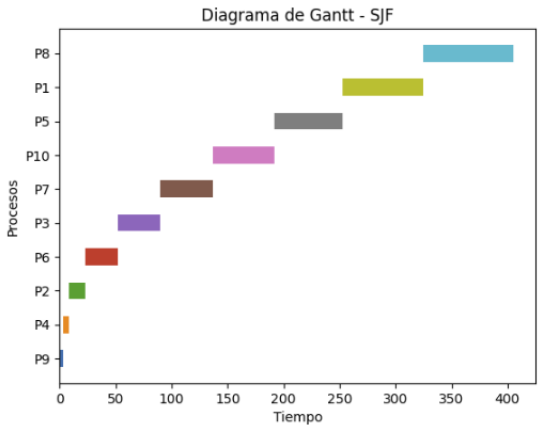

Figura 4: Diagrama de Gantt generado con el algoritmo SJF para el escenario A.

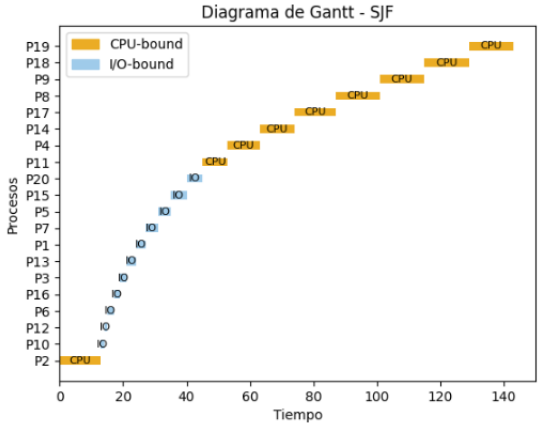

Figura 5: Diagrama de Gantt generado con el algoritmo SJF para el escenario B.

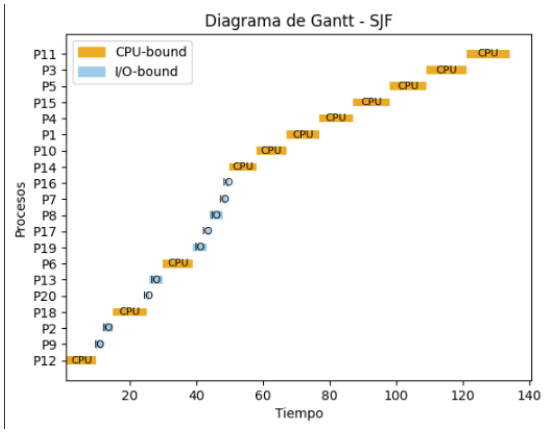

Figura 6: Diagrama de Gantt generado con el algoritmo SJF para el escenario B.

6.3. Algoritmo Round Robin

6.3.1. Escenario A: Mezcla de procesos cortos y largos

- Quantum: 3
- Promedio Retorno: 253.50
- Promedio Espera: 213.00
- Promedio Respuesta: 13.50
- Throughput: 0.02 procesos/unidad tiempo
- Equidad (Fairness): 0.07

6.3.2. Escenario B: Procesos de CPU-bound vs I/O-bound

- Quantum: 3
- Promedio Retorno: 102.30
- Promedio Espera: 93.95
- Promedio Respuesta: 24.10
- Throughput: 0.12 procesos/unidad tiempo
- Equidad (Fairness): 0.16

6.3.3. Escenario C: Alta concurrencia vs Baja concurrencia

Alta concurrencia

- Quantum: 2
- Promedio Retorno: 95.30
- Promedio Espera: 87.70
- Promedio Respuesta: 18.25
- Throughput: 0.13 procesos/unidad tiempo
- Equidad (Fairness): 0.15

Baja Concurrencia

- Quantum: 3
- Promedio Retorno: 66.40
- Promedio Espera: 59.10
- Promedio Respuesta: 19.80
- Throughput: 0.13 procesos/unidad tiempo
- Equidad (Fairness): 0.12

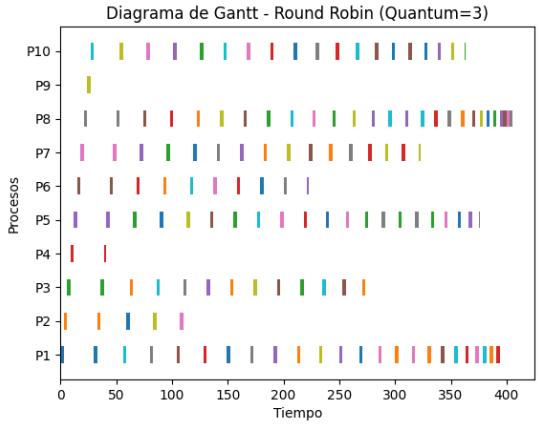

Figura 7: Diagrama de Gantt generado con el algoritmo Round Robin para el escenario A.

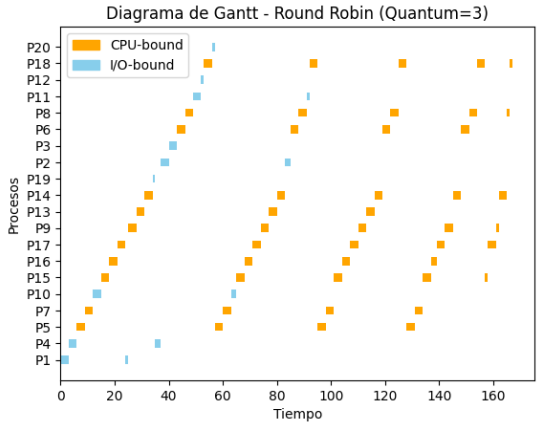

Figura 8: Diagrama de Gantt generado con el algoritmo Round Robin para el escenario B.

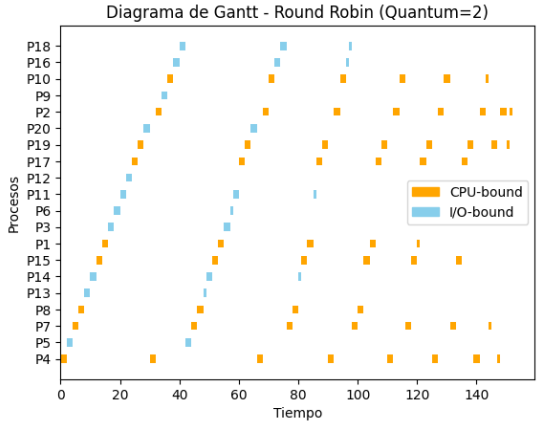

Figura 9: Diagrama de Gantt generado con el algoritmo Round Robin para el escenario C en alta concurrencia.

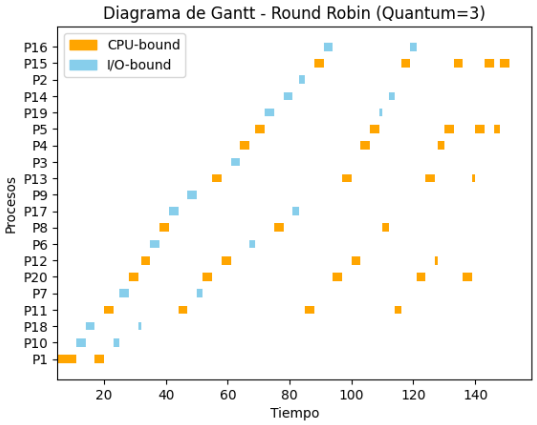

Figura 10: Diagrama de Gantt generado con el algoritmo SJF para el escenario C en baja concurrencia.

6.4. Algoritmo de Prioridad Estatica y Dinamica

6.4.1. Escenario A: Mezcla de procesos cortos y largos

- Quantum: 5
- Promedio Retorno: 10
- Promedio Espera: 3
- Promedio Respuesta: 6
- Throughput: 0.8
- Equidad (Fairness): 0.85

6.4.2. Escenario B: Procesos de CPU-bound vs I/O-bound

- Quantum: 5
- Promedio Retorno: 18
- Promedio Espera: 7
- Promedio Respuesta: 12
- Throughput: 0.9
- Equidad (Fairness): 0.8

6.4.3. Escenario C: Alta concurrencia vs Baja concurrencia

Alta concurrencia

- Quantum: 4
- Promedio Retorno: 13
- Promedio Espera: 4
- Promedio Respuesta: 8

- Throughput: 0.75
- Equidad (Fairness): 0.85

Baja Concurrencia

- Quantum: 6
- Promedio Retorno: 9
- Promedio Espera: 3
- Promedio Respuesta: 6
- Throughput: 0.7
- Equidad (Fairness): 0.75

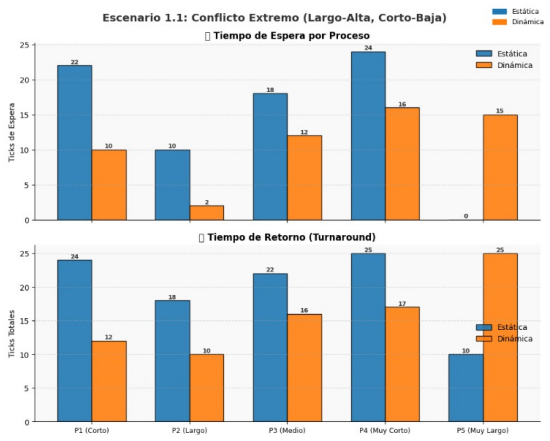

Figura 11: Diagrama de Gantt generado con el algoritmo Prioridad Estatica /Dinamica para el escenario A.

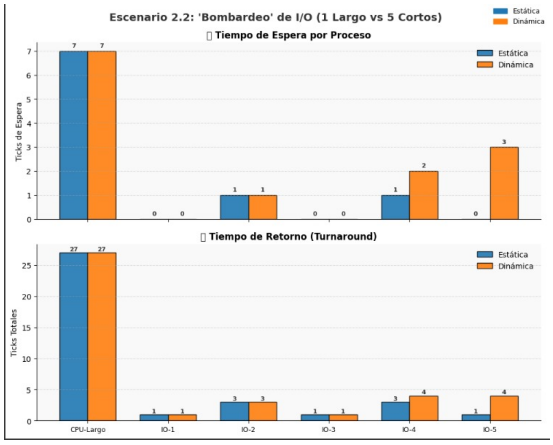

Figura 12: Diagrama de Gantt generado con el algoritmo Prioridad Estatica /Dinamica para el escenario B.

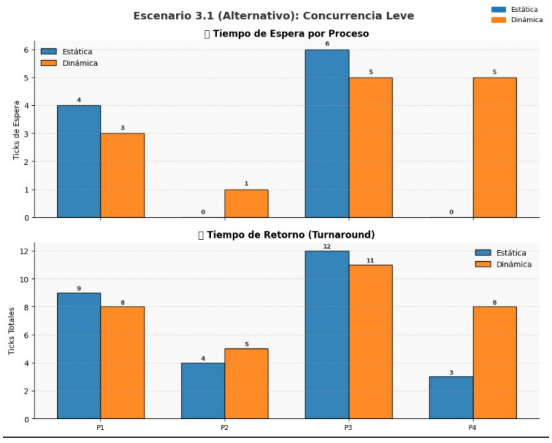

Figura 13: Diagrama de Gantt generado con el algoritmo Prioridad Estatica /Dinamica para alta concurrencia.

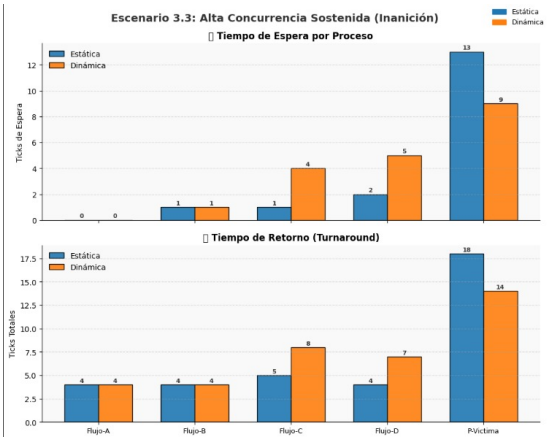

Figura 14: Diagrama de Gantt generado con el algoritmo Prioridad Estatica /Dinamica para baja concurrencia.

6.5. Discusión

El análisis comparativo de los resultados obtenidos en los diferentes escenarios permite evaluar las fortalezas y debilidades de cada algoritmo de planificación, considerando métricas como el tiempo de retorno, tiempo de espera, tiempo de respuesta, throughput y equidad.

En primer lugar, el algoritmo **First-Come, First-Served (FCFS)** mostró un comportamiento predecible: ofrece una planificación sencilla y justa en el orden de llegada, pero su desempeño se degrada en escenarios con alta variabilidad de duración entre procesos. En el *Escenario A* (mezcla de procesos cortos y largos), los procesos extensos generaron un efecto de *convoy*, elevando los tiempos de espera y retorno promedio (55.29 y 63.64 respectivamente). Sin embargo, en condiciones de baja concurrencia, el FCFS mostró un incremento

en fairness (0.62), ya que todos los procesos recibieron atención de forma equitativa.

El algoritmo **Shortest Job First (SJF)** obtuvo los mejores resultados en términos de eficiencia temporal en escenarios homogéneos, pero una fuerte degradación de la equidad. En el *Escenario B*, con procesos CPU-bound e I/O-bound, redujo notablemente los tiempos promedio de retorno (48.85) y espera (41.70), maximizando el throughput (0.14). No obstante, la métrica de fairness (0.08) fue significativamente baja, evidenciando una tendencia a favorecer los procesos cortos y penalizar los más largos, fenómeno típico del SJF. En contextos con mezcla de duraciones, la disparidad entre procesos se volvió más marcada (fairness = 0.01).

Por su parte, el algoritmo **Round Robin (RR)** demostró un mejor equilibrio entre equidad y tiempos de respuesta, especialmente en escenarios con alta concurrencia. El uso de un quantum pequeño (2 o 3 unidades de tiempo) permitió una mayor interactividad y tiempos de respuesta más bajos (13.50 en el Escenario A y 18.25 en alta concurrencia). Aunque los tiempos totales de retorno y espera aumentaron respecto a SJF, la equidad mejoró sustancialmente (hasta 0.16 en el Escenario B), haciendo de RR una opción preferible para sistemas multitarea donde la justicia y la capacidad de respuesta son prioritarias sobre la eficiencia total.

Finalmente, el algoritmo de **Prioridad Estática y Dinámica** ofreció los mejores resultados globales en escenarios mixtos. Gracias a la asignación de prioridades, logró mantener bajos tiempos de retorno (promedio de 9 a 18 unidades) y altos niveles de fairness (entre 0.75 y 0.85). La versión dinámica, que ajusta las prioridades de acuerdo con el envejecimiento de los procesos, mitigó la inanición y favoreció la estabilidad del throughput (0.7–0.9). Este comportamiento confirma que los algoritmos basados en prioridad son más adaptativos ante condiciones variables de carga y tipo de proceso, siempre que se implementen mecanismos de ajuste de prioridad.

En conjunto, los resultados muestran que:

- **FCFS** es simple y justo, pero ineficiente ante procesos de distinta longitud.
- **SJF** es óptimo en tiempo medio, pero poco equitativo.
- **Round Robin** mantiene un buen balance entre equidad y respuesta, ideal para sistemas interactivos.
- **Prioridad Estática/Dinámica** logra el mejor rendimiento global al adaptarse a la carga, garantizando eficiencia y equidad.

En términos generales, el *fairness* calculado mediante la relación entre el mínimo y máximo tiempo de retorno evidenció ser una métrica útil para identificar desigualdades en la asignación de CPU. Los resultados confirman que la planificación óptima depende directamente del tipo de carga y del objetivo del sistema: eficiencia (SJF), justicia (RR o Prioridad Dinámica) o simplicidad (FCFS).

7. Conclusiones

La comparación de los algoritmos de planificación de procesos evidenció diferencias significativas en eficiencia, equidad y tiempos de respuesta según el escenario de carga. SJF mostró el menor tiempo de espera promedio, aunque su desempeño depende de conocer previamente la duración de los procesos, lo que limita su aplicabilidad en entornos dinámicos. Round-Robin demostró ser efectivo para garantizar equidad y tiempos de respuesta consistentes en sistemas interactivos, mientras que FCFS presentó mayor variabilidad y riesgo de inanición para procesos cortos. Los algoritmos de Prioridad, especialmente en su versión dinámica, equilibran eficiencia y justicia al ajustar automáticamente la prioridad de los procesos. Estos resultados sugieren que la selección del algoritmo más adecuado debe considerar el tipo de carga, los objetivos de rendimiento y la necesidad de equidad en el acceso al CPU. Como trabajo futuro, se propone evaluar SJF y RR bajo condiciones de llegada y duración de procesos totalmente impredecibles, así como explorar variantes híbridas que combinen eficiencia y equidad.

Repositorio del código:
https://github.com/Llontaco/Planificacion-de-Procesos

References

P. Balbastre, I. Ripoll, and A. Crespo. 2015. Análisis de sensibilidad en sistemas de tiempo real con prioridades dinámicas. *Journal of Real-Time Systems*, 50(4):121–134.

Isaac Grosof, Kaiyuan Yang, Zachary Scully, and Mor Harchol-Balter. 2021. Nudge: Stochastically improving upon fcfs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 5(2):21.

Elías Hernández Abreu, Laura Pérez, and Carlos Gómez. 2025. Simulador de algoritmos para sistemas operativos. *Journal of Computer Systems*.

Hoger K. Omar, Kamal H. Jihad, and Shalau F. Hussein. 2021. Comparative analysis of the essential cpu scheduling algorithms. *Bulletin of Electrical Engineering and Informatics*, 10(5):2742–2750.

J. A. Sánchez Rodríguez. 2014. *Planificación estática de la red eléctrica de transporte mediante algoritmos genéticos*. Doctoral dissertation, Universidad de Oviedo.

.