REPÚBLICA BOLIVARIANA DE VENEZUELA

MINISTERIO DEL PODER POPULAR PARA LA EDUCACIÓN UNIVERSITARIA

UNIVERSIDAD NACIONAL EXPERIMENTAL MARÍTIMA DEL CARIBE

SISTEMAS OPERATIVOS I

SOP-I 513

SECCIÓN: A

UNIDAD II

# ADMINISTRACIÓN DE PROCESOS


**Integrantes equipo 04:**

Alarcón Campuzano, Jonathan Alexander

Algarín Aguilera, Gabriel Moises

García Savoca, Anthony Alessandro

Larez Baptista, Arnold Jesús

Partidas Roque, Nayroska Yosaine del Valle

Ramos Ruiz, Eduardo Antonio

Profesor:

Lic. Padilla P., Juan Vicente


Catia La Mar, mayo de 2025

## ÍNDICE

**p.p.**

# INTRODUCCIÓN

La administración de procesos constituye un eje fundamental en el estudio de los sistemas operativos, dado que permite comprender cómo se gestionan y coordinan las múltiples tareas que ocurren de manera simultánea en un sistema computacional. En esta unidad se abordan conceptos esenciales como la concurrencia, la sincronización, la exclusión mutua y los mecanismos de planificación, los cuales son vitales para garantizar el funcionamiento eficiente y seguro del sistema. La correcta implementación de estos conceptos no solo evita condiciones de carrera y bloqueos, sino que también optimiza el rendimiento de los recursos disponibles. A lo largo del presente trabajo se desarrollan los principales algoritmos y estructuras que permiten coordinar la ejecución de procesos concurrentes, así como los criterios que sustentan las distintas disciplinas de planificación empleadas en los sistemas operativos modernos.

1.Concurrencia

La concurrencia se refiere a la capacidad de un sistema para ejecutar múltiples procesos o hilos de forma simultánea o en aparente simultaneidad. Este concepto se encuentra ampliamente aplicado en sistemas operativos, bases de datos y arquitecturas paralelas. Entre los principales desafíos asociados al manejo de concurrencia se destacan la sincronización, la exclusión mutua y la prevención de condiciones de carrera.

2.Algoritmos de sincronización para asegurar la exclusión mutua

La exclusión mutua constituye un mecanismo esencial para evitar que más de un proceso acceda de manera simultánea a un recurso compartido, lo cual podría provocar inconsistencias o corrupción de datos. Para ello, se han desarrollado distintos algoritmos y estructuras de sincronización, entre los cuales destacan:

Semáforos: Utilizan contadores para gestionar el acceso de los procesos a los recursos compartidos. Permiten o bloquean el paso según el valor del contador.

Monitores: Encapsulan tanto datos como procedimientos, evitando que múltiples procesos interactúen simultáneamente con los mismos recursos.

Cerrojos (Locks): Establecen una sección crítica protegida. Un proceso debe adquirir el cerrojo antes de acceder al recurso y liberarlo al finalizar su operación.

3. Ritmo de espera activa

El algoritmo de espera activa se implementa cuando un proceso permanece en espera por un recurso sin liberar la CPU. Aunque garantiza inmediatez al

momento de reanudar la ejecución, también genera un consumo elevado de recursos del sistema.

3.1 Espera con mutex

El mutex es una estructura de control que permite garantizar la exclusión mutua. Solo un proceso puede adquirir el mutex en un momento determinado.

Ejemplo: Exclusión Mutua con threading.Lock()

Python CopiarEditar import threading mutex = threading.Lock()

contador = 0 def incrementar(): global contador  for _ in range(100000):  with mutex: contador += 1

hilo1 = threading.Thread(target=incrementar)

hilo2 = threading.Thread(target=incrementar)

hilo1.start()

hilo2.start()

hilo1.join()

hilo2.join()

print(f"Valor final del contador: {contador}")

Explicación:

- El uso de threading.Lock() asegura que solo un hilo acceda a la variable contador en un momento dado.
- Este mecanismo evita condiciones de carrera que pueden alterar el resultado final del programa.

3.2 Algoritmo de alternancia

El algoritmo de alternancia permite que dos procesos se ejecuten de forma intercalada, garantizando el acceso secuencial a una sección crítica. Su funcionamiento se basa en una variable compartida que indica el turno de cada proceso.

Ejemplo: Alternancia entre procesos

Python CopiarEditar import threading turno = 0

lock = threading.Lock() def proceso(id):  global turno

for _ in range(5):  while turno != id:

pass  # Espera activa

with lock: print(f"Proceso {id} ejecutando")

turno = 1 - id  # Cambia el turno al otro proceso

hilo1 = threading.Thread(target=proceso, args=(0,))

hilo2 = threading.Thread(target=proceso, args=(1,))

hilo1.start()

hilo2.start()

hilo1.join()

hilo2.join()

Explicación:

- La variable turno actúa como mecanismo de control para coordinar la ejecución.

- Cada proceso espera activamente hasta que sea su turno de ejecutar.

3.3 Algoritmo de Dekker

El algoritmo de Dekker es una de las primeras soluciones correctas para la exclusión mutua en sistemas con dos procesos. Este mecanismo emplea variables booleanas y una variable de turno para resolver conflictos de acceso a recursos compartidos.

Ejemplo: Implementación del Algoritmo de Dekker

python

CopiarEditar

```
import threading flag = [False, False]

turn = 0

def dekker(id): global turn  otro = 1 - id

   for _ in range(5):   flag[id] = True while flag[otro]:  if turn == otro:

            flag[id] = False

            while turn == otro:  pass

 flag[id] = True

      print(f"Proceso {id} en sección crítica")

      turn = otro

      flag[id] = False

hilo1 = threading.Thread(target=dekker, args=(0,))

hilo2 = threading.Thread(target=dekker, args=(1,))
```

```
hilo1.start()

hilo2.start()

hilo1.join()

hilo2.join()
```

Explicación:

- La variable flag indica si un proceso desea entrar en la sección crítica.
- La variable turn permite resolver conflictos cuando ambos procesos desean acceder al recurso simultáneamente.
- El algoritmo garantiza la exclusión mutua sin recurrir a instrucciones atómicas del hardware.

## 4. Problemas clásicos de sincronización

Los problemas clásicos de sincronización fueron diseñados como abstracciones de situaciones reales que permiten estudiar y resolver conflictos de acceso concurrente a recursos compartidos. Aunque puedan parecer irrelevantes si se interpretan literalmente, su valor reside en la forma como representan, mediante analogías, situaciones complejas de la informática. La clave está en su adecuada interpretación.

### 4.1 El problema de los filósofos comensales

Cinco filósofos se sientan en una mesa circular con un plato de espagueti en el centro. Entre cada par de filósofos hay un tenedor, y para poder comer, cada filósofo necesita tomar los dos tenedores contiguos. Los filósofos alternan entre pensar y comer, pero no saben cuándo los demás desean hacer lo mismo.

El objetivo de este problema es diseñar un algoritmo concurrente que garantice que ningún filósofo muera de hambre por falta de acceso a los tenedores.

Para representar este problema, se consideran:

- Estados: pensar, comer, hambriento, dormir.
- Un arreglo de cinco semáforos.
- Un semáforo de exclusión mutua.

Este modelo permite gestionar el acceso a los recursos (tenedores) y evitar conflictos o bloqueos entre procesos (filósofos).

### 4.2 El problema del productor-consumidor

Este problema plantea dos procesos: un productor, que genera datos o recursos, y un consumidor, que los procesa. Ambos acceden a un búfer de tamaño fijo, que se utiliza como una cola. El productor añade elementos al búfer si hay espacio disponible, mientras que el consumidor retira elementos cuando existen datos por procesar. La sincronización se logra mediante semáforos para evitar condiciones de carrera y asegurar el acceso seguro al búfer.

## 5. Mecanismo de semáforo

Las soluciones basadas en instrucciones de bajo nivel, como TestAndSet() y Swap(), pueden resultar complicadas para los programadores. Para facilitar la implementación de sincronización, se emplean semáforos. Un semáforo es una variable entera a la que se accede únicamente mediante dos operaciones atómicas: wait() y signal(). Estas garantizan la exclusión mutua en el acceso a secciones críticas.

5.1 Operaciones con semáforo

- init(): Inicializa el semáforo con un valor dado.
- wait(): Disminuye el valor del semáforo. Si es menor o igual a cero, el proceso entra en espera.
- signal(): Incrementa el valor del semáforo, permitiendo que otro proceso continúe.

Ejemplo en pseudocódigo

c

CopiarEditar wait(S)  while (S <= 0);

S--; } signal(S) {  S++; }

5.2 Aplicación del semáforo

Para asegurar que una instrucción S2 solo se ejecute después de S1, se puede usar un semáforo synch compartido entre dos procesos P1 y P2:

Proceso P1:

c  CopiarEditar S1;

signal(synch); Proceso P2:

c  CopiarEditar wait(synch);

S2;

5.3 Ejemplo práctico

C CopiarEditar

do { wait(mutex);

// sección crítica

signal(mutex);

// sección restante

} while (TRUE);

5.4 Ilustración con dos procesos

Inicialización:

c

CopiarEditar INIT(S, 1); Procedimiento de Alicia:

C CopiarEditar repeat

P(S); acceder_A;

V(S); otras tareas; until False

Procedimiento de Bernardo:

c CopiarEditar repeat P(S);

acceder_B; V(S);

otras tareas; until False

6. Criterios de planificación

Los criterios de planificación son parámetros utilizados para evaluar la eficiencia de un algoritmo de planificación en sistemas operativos. Estos criterios permiten determinar qué tan adecuado es un algoritmo para asignar el procesador a los procesos, teniendo en cuenta el uso óptimo de los recursos y la satisfacción de los usuarios.

6.1 Criterios relativos al rendimiento del sistema

Entre los principales criterios relacionados con el rendimiento general del sistema se destacan:

- Uso del procesador: Mide el porcentaje de tiempo que el procesador se encuentra activo. Un valor alto implica un uso eficiente de la CPU.
- Rendimiento (Throughput): Indica la cantidad de procesos completados por unidad de tiempo. A mayor rendimiento, más productivo es el sistema.
- Tiempo de retorno: Es el intervalo desde que un proceso entra al sistema hasta que finaliza su ejecución. Se busca minimizar este tiempo.
- Tiempo de espera: Representa el tiempo que un proceso pasa en la cola de listos sin ser ejecutado. Cuanto menor sea, mejor será la percepción de eficiencia.
- Tiempo de respuesta: Es el tiempo desde que se solicita un servicio hasta que se obtiene la primera respuesta. Es especialmente importante en sistemas interactivos.

7. Cómo debe ser una disciplina de planificación

Una disciplina de planificación debe asegurar el uso justo y eficiente del procesador, garantizando que los procesos no sufran inanición (espera indefinida) y que los recursos del sistema se utilicen de manera óptima. Además, debe ser predecible, simple de implementar y adaptable a distintos entornos.

## 7.1 Planificación apropiativa

La planificación apropiativa se caracteriza por permitir la interrupción de un proceso en ejecución para ceder el procesador a otro con mayor prioridad o mejor oportunidad de ejecución. Esta disciplina es fundamental en sistemas multitarea o en tiempo compartido, donde se requiere equidad y respuesta rápida a eventos externos.

Algunos algoritmos apropiativos más utilizados son:

- Round Robin (RR): Asigna tiempos de ejecución iguales en ciclos rotativos. Es equitativo y simple de implementar.
- Shortest Remaining Time First (SRTF): Selecciona el proceso con el menor tiempo de ejecución restante. Optimiza el tiempo promedio de retorno.
- Planificación por prioridades: Da preferencia a los procesos con mayor prioridad. Puede combinarse con mecanismos de envejecimiento para evitar inanición.

## CONCLUSIÓN

El análisis de la administración de procesos revela la complejidad y precisión requeridas para gestionar de manera efectiva la ejecución simultánea de múltiples tareas dentro de un sistema operativo. La implementación adecuada de mecanismos de sincronización, como semáforos y mutex, junto con algoritmos como el de Dekker o el de alternancia, permite asegurar la exclusión mutua y evitar conflictos entre procesos. Asimismo, el estudio de los problemas clásicos de sincronización y los criterios de planificación proporciona una base sólida para la comprensión del comportamiento del sistema bajo diversas condiciones de carga y prioridad. En conclusión, el dominio de estos elementos es indispensable para diseñar sistemas operativos robustos, eficientes y justos, capaces de responder a las crecientes demandas del entorno tecnológico actual.

## REFERENCIAS BIBLIOGRÁFICAS

Tanenbaum, A. S. (2009). *Sistemas operativos modernos* (3.ª ed.). Pearson Educación.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). *Fundamentos de sistemas operativos* (8.ª ed.). McGraw-Hill.

Stallings, W. (2012). *Sistemas operativos: Conceptos internos y diseño* (7.ª ed.). Pearson Educación.