

Informe de Experimentos — Simulador de Scheduling

Fecha: 17 de octubre de 2025

Objetivo

Comparar objetivamente tres algoritmos de scheduling (FCFS, Round Robin, SJF) frente a tres tipos de workloads definidos por la proporción de procesos I/O-bound y CPU-bound: balanceado (50/50), I/O-bound dominante (90% I/O), y CPU-bound dominante (90% CPU). Las métricas evaluadas son:

- Throughput (procesos completados / unidad de tiempo)
- Avg Turnaround (promedio de tiempo turnaround)
- Avg Response Time (promedio de tiempo de respuesta)

Requerimientos (resumen)

Para el trabajo se definieron los siguientes requerimientos mínimos:

1. Implementar un simulador en C que modele procesos con bursts (CPU e I/O).
2. Soportar la selección de la política de scheduling en tiempo de ejecución: FCFS, Round Robin (configurable quantum), SJF.
3. Permitir generar/usar workloads con 100 procesos, y con 10 bursts por proceso (se definió esta estructura en el simulador).
4. Registrar trazas de eventos (llegada, entrada/salida de CPU, espera por I/O, finalización) y calcular métricas finales.
5. Ejecutar experiments para los tres escenarios de mezcla de procesos (50/50, 90% I/O, 90% CPU) y para cada algoritmo.

Implementación (breve)

El simulador modela: - Tipo de bursts: CPU_BURST, IO_BURST. - Estados de proceso: NEW, READY, RUNNING, WAITING, TERMINATED. - Estructuras de colas: cola FIFO para FCFS y RR, y cola por prioridad (duración del próximo CPU burst) para SJF. - Dispatch y actualización por tick de reloj: llegada de procesos, decremento de bursts, manejo de I/O y preempcción para Round Robin.

Se añadieron generadores dinámicos de workloads que crean 100 procesos con una fracción de I/O especificada (por ejemplo 0.90 para 90% I/O-bound). Cada proceso obtiene 10 bursts generados aleatoriamente respetando la clasificación I/O/CPU-bound (probabilidades y distribuciones de duración configurables en el código).

Workloads utilizados

- Balanced (50% I/O, 50% CPU). 100 procesos.
- I/O-heavy (90% I/O, 10% CPU). 100 procesos.
- CPU-heavy (10% I/O, 90% CPU). 100 procesos.

Los procesos se hicieron llegar en el tiempo agrupando 10 procesos por unidad de tiempo (`arrival_time = index / 10`), para distribuir arrivals durante 10 unidades de tiempo.

Resultados medidos

Los resultados provienen del fichero `testcases.txt` (salida del simulador) y resumen las métricas promedio para cada combinación algoritmo × workload.

Tabla Resumen General

Algoritmo	Workload	Avg Turnaround	Avg Response	Throughput
FCFS	I/O Bound (90/10)	2517.54	265.98	0.034
FCFS	Balanced (50/50)	3746.24	277.57	0.021
FCFS	CPU Bound (10/90)	5606.55	324.30	0.015
Round	I/O	2403.41	195.69	0.033
Robin (q=5)	Bound (90/10)			
Round	Balanced	3552.23	203.33	0.021
Robin (q=5)	(50/50)			
Round	CPU	5421.26	215.63	0.016
Robin (q=5)	Bound (10/90)			
SJF	I/O Bound (90/10)	1559.74	773.73	0.034
SJF	Balanced (50/50)	2229.49	1099.37	0.021
SJF	CPU Bound (10/90)	3859.01	1012.13	0.016

*Nota: Los valores en **negrita** representan los mejores resultados para cada métrica por workload.*

Comparación por Métrica

Turnaround Time (menor es mejor)

Workload	FCFS	Round Robin	SJF	Ganador
I/O Bound (90/10)	2517.54	2403.41	1559.74	SJF (-38% vs FCFS)
Balanced (50/50)	3746.24	3552.23	2229.49	SJF (-40% vs FCFS)
CPU Bound (10/90)	5606.55	5421.26	3859.01	SJF (-31% vs FCFS)

Conclusión: SJF reduce el turnaround promedio entre 31-40% comparado con FCFS.

Response Time (menor es mejor)

Workload	FCFS	Round Robin	SJF	Ganador
I/O Bound (90/10)	265.98	195.69	773.73	Round Robin (-26% vs FCFS)
Balanced (50/50)	277.57	203.33	1099.37	Round Robin (-27% vs FCFS)
CPU Bound (10/90)	324.30	215.63	1012.13	Round Robin (-34% vs FCFS)

Conclusión: Round Robin mejora el tiempo de respuesta entre 26-34% comparado con FCFS. SJF tiene el peor response time (294-441% mayor que RR).

Throughput (mayor es mejor)

Workload	FCFS	Round Robin	SJF	Observación
I/O Bound (90/10)	0.034	0.033	0.034	Empate técnico
Balanced (50/50)	0.021	0.021	0.021	Idéntico
CPU Bound (10/90)	0.015	0.016	0.016	RR/SJF ligeramente mejor

Workload	FCFS	Round Robin	SJF	Observación
----------	------	-------------	-----	-------------

Conclusión: El throughput es prácticamente idéntico entre algoritmos para cada workload, dominado por la carga total de trabajo.

Análisis por Workload

Workload I/O Bound (90% I/O, 10% CPU)

Métrica	FCFS	Round Robin	SJF	Diferencia Max
Avg	2517.54	2403.41 (-4.5%)	1559.74	957.80
Turnaround			(- 38%)	
Avg	265.98	195.69 (-26%)	773.73	578.04
Response			(+191%)	
Throughput	0.034	0.033	0.034	~0

Mejor para Turnaround: SJF

Mejor para Response: Round Robin

Mejor para Throughput: Empate (FCFS/SJF)

Workload Balanced (50% I/O, 50% CPU)

Métrica	FCFS	Round Robin	SJF	Diferencia Max
Avg	3746.24	3552.23 (-5.2%)	2229.49	1516.75
Turnaround			(- 40%)	
Avg	277.57	203.33 (-27%)	1099.37	896.04
Response			(+296%)	
Throughput	0.021	0.021	0.021	0

Mejor para Turnaround: SJF

Mejor para Response: Round Robin

Mejor para Throughput: Empate

Workload CPU Bound (10% I/O, 90% CPU)

Métrica	FCFS	Round Robin	SJF	Diferencia Max
Avg Turnaround	5606.55	5421.26 (-3.3%)	3859.01	1747.54 (-31%)
Avg Response	324.30	215.63 (-34%)	1012.13	796.50 (+212%)
Throughput	0.015	0.016 (+6.7%)	0.016	0.001 (+6.7%)

Mejor para Turnaround: SJF

Mejor para Response: Round Robin

Mejor para Throughput: Round Robin/SJF

Gráfico Visual de Rendimiento Relativo

Turnaround Time (normalizado a FCFS = 100%)

Workload	FCFS	Round Robin	SJF
I/O Bound	100%	95%	62% (down)
Balanced	100%	95%	59% (down)
CPU Bound	100%	97%	69% (down)

Response Time (normalizado a FCFS = 100%)

Workload	FCFS	Round Robin	SJF
I/O Bound	100%	74% (down)	291% (up)
Balanced	100%	73% (down)	396% (up)
CPU Bound	100%	66% (down)	312% (up)

Análisis y discusión

A partir de los resultados medidos:

1. Throughput

- El throughput es similar entre algoritmos para cada tipo de workload (valores muy cercanos por escenario). Esto indica que la cantidad de procesos completados por unidad de tiempo estuvo dominada por la

carga total de CPU/I/O y la duración agregada de los bursts más que por la política de scheduling en estos experimentos.

2. Turnaround promedio

- SJF muestra consistentemente el menor Avg Turnaround en los tres escenarios, particularmente en el caso I/O-bound (1559.74 vs 2517.54 en FCFS y 2403.41 en RR). Esto es consistente con la propiedad de SJF de priorizar trabajos cortos, reduciendo el tiempo promedio de finalización.
- FCFS presenta el peor Avg Turnaround para todos los escenarios, especialmente en CPU-heavy (5606.55), por su naturaleza no-preemptiva y orden FIFO que penaliza procesos largos al inicio.

3. Response time promedio

- Round Robin con quantum = 5 produce los mejores (menores) Avg Response en todos los escenarios comparado con FCFS y SJF (ej. RR I/O-bound: 195.69 vs FCFS 265.98 y SJF 773.73). RR mejora la latencia inicial al repartir la CPU frecuentemente entre procesos.
- SJF tiene un Avg Response alto, especialmente en Balanced y CPU-heavy, porque prioriza procesos con ráfagas cortas; procesos largos pueden esperar mucho antes de obtener CPU y así su tiempo hasta el primer inicio aumenta.

4. Efecto del workload

- En general, a medida que el workload se vuelve más CPU-bound, avg turnaround y response tienden a incrementarse para FCFS y SJF, aunque SJF reduce el impacto relativo gracias a su preferencia por trabajos cortos.
- RR mejora el response en comparación con FCFS y SJF, especialmente para I/O-bound y workloads balanceados.

Recomendaciones

- Para minimizar turnaround promedio en un entorno mixto o I/O-dominante, SJF suele ser la mejor opción.
- Si la prioridad es reducir la latencia de respuesta (por ejemplo sistemas interactivos), Round Robin con quantum pequeño (ej. 4-8) es preferible.
- FCFS es simple pero puede tener un rendimiento pobre para cargas que incluyen procesos largos (CPU-bound).

Limitaciones y trabajo futuro

- La generación de bursts es aleatoria; para estudios estadísticos robustos conviene ejecutar cada combinación algoritmo × workload múltiples veces (p. ej. 30 replicaciones) y reportar medias y desviaciones estándar.
- Se podría exportar la traza y los datos a CSV para análisis en Python/R y generar gráficos (CDF, boxplots) de las métricas.
- Evaluar más políticas (Priority, Multilevel Feedback Queue) y medir efectos de diferentes quantums en RR.

- Añadir liberación de memoria y opciones de configuración (número de procesos, bursts por proceso, distribución de duraciones) vía línea de comandos o archivo de configuración.

Conclusión

Los resultados muestran las típicas ventajas y desventajas de cada algoritmo: SJF optimiza turnaround, RR reduce tiempo de respuesta, y FCFS es el menos eficiente en presencia de procesos largos. El throughput fue similar entre algoritmos en estas instancias porque depende más del total de trabajo y de la mezcla I/O/CPU que de la política en sí.

Uso de LLMs (documentación mínima)

Se consultó un LLM para:

- **Estructuras de datos:** Implementación de una cola de prioridad (priority queue) para el algoritmo SJF y cola FIFO para FCFS/RR en C.
- **Lógica de scheduling:** Asistencia en la lógica de inserción y extracción de procesos de las colas, manejo de estados y transiciones entre READY, RUNNING, WAITING.
- **Manejo de memoria:** Recomendaciones sobre gestión de memoria dinámica para las estructuras de colas y procesos.
- **Arquitectura del simulador:** Sugerencias sobre la organización del ciclo principal de simulación (tick-based) y actualización de estados.

Criterios finales de diseño: Las estructuras de colas, los algoritmos de inserción/extracción y la máquina de estados de procesos fueron ajustados por el equipo para garantizar:

- Correcta priorización en SJF (menor burst primero)
- Manejo apropiado de preempción en Round Robin
- Transiciones de estado consistentes
- Liberación completa de memoria al finalizar

Conversacion Completa para la implementacion: <https://chatgpt.com/share/6907b2cb-03f8-8006-b5ac-d120824325fc>

Problema de los Filósofos Comensales

Resumen

Este proyecto presenta una implementación completa del problema clásico de sincronización de los Filósofos Comensales en tres lenguajes de programación: C, Java y Python. Cada implementación incluye soluciones basadas en hilos (threads) y procesos independientes, demostrando diferentes mecanismos de sincronización y comunicación entre procesos (IPC).

Descripción del Problema

El problema de los Filósofos Comensales, propuesto por Edsger Dijkstra en 1965, es un problema clásico de sincronización en sistemas concurrentes. El escenario consiste en N filósofos sentados alrededor de una mesa circular, donde cada filósofo alterna entre dos estados: pensar y comer. Entre cada par de filósofos hay exactamente un tenedor, resultando en N tenedores en total. Para comer, un filósofo requiere adquirir ambos tenedores adyacentes (izquierdo y derecho).

Los principales desafíos de este problema son:

1. **Deadlock:** Evitar que todos los filósofos tomen simultáneamente su tenedor izquierdo, bloqueándose mutuamente al esperar el tenedor derecho.
2. **Starvation:** Garantizar que ningún filósofo permanezca indefinidamente sin poder comer.
3. **Concurrencia:** Maximizar el paralelismo permitiendo que múltiples filósofos coman simultáneamente cuando sea posible.

Estrategia de Prevención de Deadlock y Starvation

Algoritmo Implementado

Todas las implementaciones utilizan una estrategia centralizada de control basada en el algoritmo de “verificación de vecinos”. La estrategia consiste en:

1. **Control Centralizado:** Una entidad central (mesa) coordina el acceso a los recursos compartidos (tenedores).
2. **Verificación de Vecinos:** Un filósofo solo puede tomar ambos tenedores simultáneamente si:
 - El filósofo está en estado HAMBRIENTO
 - El vecino izquierdo NO está COMIENDO
 - El vecino derecho NO está COMIENDO
3. **Notificación Proactiva:** Cuando un filósofo libera sus tenedores:
 - Cambia su estado a PENSANDO
 - La mesa verifica si los vecinos adyacentes pueden ahora comer
 - Se notifica explícitamente a los vecinos elegibles

4. **Atomicidad en Adquisición:** Los tenedores se adquieren como una operación atómica, evitando estados intermedios que puedan causar deadlock.

Esta estrategia previene el deadlock al evitar la adquisición secuencial de recursos y previene la starvation mediante notificaciones explícitas que garantizan que los filósofos bloqueados sean despertados cuando los recursos estén disponibles.

Implementación en C

Estructura del Proyecto

```
filosofos_c/
|-- Makefile
|-- bin/
|   |-- filosofos          # Ejecutable con hilos POSIX
|   +-+ filosofos_procesos # Ejecutable con procesos fork()
|-- obj/                   # Archivos objeto compilados
++- src/
    |-- hilos/             # Implementacion con threads
    |   |-- main.c
    |   |-- filosofo.c/h
    |   |-- mesa.c/h
    |   +-+ tenedor.c/h
    +-+ procesos/          # Implementacion con fork()
        |-- main.c
        |-- proceso_filosofo.c/h
        +-+ mesa_ipc.c/h
```

Mecanismos de Sincronización

Versión con Hilos: - **Mutex Global** (`pthread_mutex_t`): Protege el acceso a la tabla de estados compartida - **Variable de Condición** (`pthread_cond_t`): Implementa el mecanismo de espera y notificación - **Broadcast**: Utiliza `pthread_cond_broadcast()` para notificar a todos los filósofos esperando

Versión con Procesos: - **Memoria Compartida** (`mmap` con `MAP_SHARED`): Tabla de estados y contadores - **Semáforos POSIX** (`sem_t`): Array de semáforos para sincronización entre procesos - **Semáforo Global**: Protege las secciones críticas en operaciones de la mesa

Compilación y Ejecución

```
# Compilar
cd filosofos_c
make clean
make

# Ejecutar versión con hilos
```

```

./bin/filosofos [num_filosofos] [duracion_segundos]
./bin/filosofos 5 30

# Ejecutar versión con procesos
./bin/filosofos_procesos [num_filosofos] [duracion_segundos]
./bin/filosofos_procesos 5 30

```

Resultados Experimentales

Configuración de prueba: 5 filósofos, 30 segundos de ejecución

Versión con Hilos:

```

Filósofo 0 comió 11 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 9 veces
Filósofo 3 comió 12 veces
Filósofo 4 comió 13 veces

```

```

Total de veces que se comió: 56
Promedio por filósofo: 11.20
Desviación estándar: 1.47

```

Versión con Procesos:

```

Filósofo 0 comió 11 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 12 veces
Filósofo 3 comió 10 veces
Filósofo 4 comió 10 veces

```

```

Total de veces que se comió: 54
Promedio por filósofo: 10.80
Desviación estándar: 0.84

```

Análisis: Ambas versiones demuestran una distribución equitativa de acceso a recursos, con desviaciones estándar bajas que indican ausencia de starvation. La ligera variación en el total se debe a diferencias en el overhead de sincronización entre hilos y procesos.

Implementación en Java

Estructura del Proyecto

```

filosofos_java/
|-- bin/                      # Archivos .class compilados
++- src/
    |-- EjecutarHilos.java
    |-- EjecutarProcesos.java

```

```

|-- hilos/          # Implementacion con Threads
|   |-- Filosofo.java
|   |-- Mesa.java
|   |-- Tenedor.java
|   +- Estado.java
|-- procesos/       # Simulacion de procesos con Threads
|   |-- ProcesoFilosofo.java
|   |-- MesaIPC.java
|   +- EstadoFilosofo.java
+- procesosreales/ # Procesos JVM independientes
    |-- FilosofoClient.java
    |-- MesaServer.java
    |-- Launcher.java
    +- Estado.java

```

Mecanismos de Sincronización

Versión con Hilos: - **ReentrantLock**: Lock explícito para control de acceso mutuo - **Condition**: Mecanismo de espera y señalización asociado al lock - **signalAll()**: Notificación broadcast a todos los hilos en espera

Versión con Procesos Simulados: - **Semaphore**: Implementación de semáforos de Java para control de acceso - **AtomicIntegerArray**: Estructura atómica para tabla de estados compartida - **Array de Semaphores**: Uno por filósofo para bloqueo individual

Versión con Procesos Reales: - **Sockets TCP**: Comunicación entre JVMs independientes - **Arquitectura Cliente-Servidor**: MesaServer coordina múltiples FilosofoClient - **ProcessBuilder**: Creación de procesos Java separados

Compilación y Ejecución

```

# Compilar
cd filosofos_java
javac -d bin src/*.java src/hilos/*.java src/procesos/*.java src/procesosreales/*.java

# Ejecutar versión con hilos
java -cp bin EjecutarHilos [num_filosofos] [duracion_segundos]
java -cp bin EjecutarHilos 5 30

# Ejecutar versión con procesos simulados
java -cp bin EjecutarProcesos [num_filosofos] [duracion_segundos]
java -cp bin EjecutarProcesos 5 30

# Ejecutar versión con procesos reales (múltiples JVMs)
java -cp bin procesosreales.Launcher [num_filosofos] [duracion_segundos]
java -cp bin procesosreales.Launcher 5 30

```

Resultados Experimentales

Configuración de prueba: 5 filósofos, 30 segundos de ejecución

Versión con Hilos:

Filósofo 0 comió 10 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 10 veces
Filósofo 3 comió 9 veces
Filósofo 4 comió 11 veces

Total de veces que se comió: 51

Promedio por filósofo: 10.20

Desviación estándar: 0.84

Versión con Procesos Simulados:

Filósofo 0 comió 9 veces
Filósofo 1 comió 10 veces
Filósofo 2 comió 11 veces
Filósofo 3 comió 10 veces
Filósofo 4 comió 9 veces

Total de veces que se comió: 49

Promedio por filósofo: 9.80

Desviación estándar: 0.84

Versión con Procesos Reales:

Filósofo 0 comió 8 veces
Filósofo 1 comió 9 veces
Filósofo 2 comió 9 veces
Filósofo 3 comió 8 veces
Filósofo 4 comió 8 veces

Total de veces que se comió: 42

Promedio por filósofo: 8.40

Desviación estándar: 0.55

Análisis: La versión con procesos reales muestra un rendimiento ligeramente inferior debido al overhead de comunicación por sockets TCP. Sin embargo, la distribución sigue siendo equitativa, demostrando la efectividad del algoritmo independientemente del mecanismo de IPC utilizado.

Implementación en Python

Estructura del Proyecto

`filosofos_python/`

```

|-- ejecutar_hilos.py
|-- ejecutar_procesos.py
|-- solucion_hilos/      # Implementacion con threading
|   |-- filosofo.py
|   |-- mesa.py
|   `-- tenedor.py
`-- solucion_procesos/    # Implementacion con multiprocessing
    |-- ProcesoFilosofo.py
    `-- MesaIPC.py

```

Mecanismos de Sincronización

Versión con Hilos: - `threading.Lock`: Mutex para protección de secciones críticas - `threading.Condition`: Variable de condición para espera y notificación - `notify_all()`: Notificación broadcast a todos los hilos bloqueados

Versión con Procesos: - `multiprocessing.Semaphore`: Semáforos para sincronización entre procesos - `multiprocessing.Array`: Array compartido para tabla de estados - `multiprocessing.Lock`: Lock para proteger operaciones críticas

Compilación y Ejecución

Python no requiere compilación. Las implementaciones se ejecutan directamente:

```

# Ejecutar versión con hilos
cd filosofos_python
python ejecutar_hilos.py [num_filosofos] [duracion_segundos]
python ejecutar_hilos.py 5 30

# Ejecutar versión con procesos
python ejecutar_procesos.py [num_filosofos] [duracion_segundos]
python ejecutar_procesos.py 5 30

```

Resultados Experimentales

Configuración de prueba: 5 filósofos, 30 segundos de ejecución

Versión con Hilos:

```

Filósofo 0 comió 12 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 10 veces
Filósofo 3 comió 11 veces
Filósofo 4 comió 12 veces

```

```

Total de veces que se comió: 56
Promedio por filósofo: 11.20
Desviación estándar: 0.84

```

Versión con Procesos:

```
Filósofo 0 comió 10 veces  
Filósofo 1 comió 11 veces  
Filósofo 2 comió 10 veces  
Filósofo 3 comió 10 veces  
Filósofo 4 comió 11 veces
```

Total de veces que se comió: 52

Promedio por filósofo: 10.40

Desviación estándar: 0.55

Análisis: Python muestra resultados consistentes con las otras implementaciones. La versión con hilos presenta un rendimiento ligeramente superior debido al Global Interpreter Lock (GIL) que reduce el overhead en cambios de contexto, aunque limita el paralelismo real en sistemas multi-core.

Comparación de Rendimiento

Tabla Comparativa

Implementación	Total Comidas	Promedio	Desv. Estándar	Overhead
C - Hilos	56	11.20	1.47	Bajo
C - Procesos	54	10.80	0.84	Medio
Java - Hilos	51	10.20	0.84	Medio
Java - Procesos	49	9.80	0.84	Medio
Java - Reales	42	8.40	0.55	Alto
Python - Hilos	56	11.20	0.84	Bajo
Python - Procesos	52	10.40	0.55	Medio

Observaciones

- Rendimiento:** Las implementaciones en C muestran el mayor número de operaciones completadas, seguidas por Python y Java.
- Equidad:** Todas las implementaciones demuestran distribuciones equitativas (desviaciones estándar < 1.5), validando la ausencia de starvation.
- Overhead:** Los mecanismos basados en procesos presentan mayor overhead que los basados en hilos, especialmente en la versión con procesos reales de Java que requiere comunicación por red.
- Consistencia:** La estrategia de verificación de vecinos produce resultados consistentes independientemente del lenguaje o mecanismo de sincronización utilizado.

Conclusiones

Este proyecto demuestra exitosamente la implementación del algoritmo de verificación de vecinos para resolver el problema de los Filósofos Comensales. Las principales conclusiones son:

1. **Corrección:** Todas las implementaciones evitan tanto deadlock como starvation mediante control centralizado y verificación atómica de condiciones.
2. **Portabilidad:** El algoritmo es efectivo independientemente del lenguaje de programación y mecanismo de sincronización utilizado.
3. **Escalabilidad:** Las soluciones mantienen su corrección y equidad con diferentes números de filósofos (probado con $N = 3, 5, 7$).
4. **Trade-offs:** Existe un compromiso observable entre el overhead de comunicación y el aislamiento de recursos, siendo los hilos más eficientes pero los procesos más robustos ante fallos.

Referencias

1. Dijkstra, E. W. (1971). “Hierarchical ordering of sequential processes”. *Acta Informatica*, 1(2), 115-138.
2. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.
3. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

Uso de LLMs

- https://chatgpt.com/g/g-p-68df4854a2f48191a2b28a71b762d003-sistemas-operativos/shared/c/68e95892-ea1c-832e-9a0d-838a0242f306?owner_user_id=user-LTiELctIdXXaj310lEuQDuq0

Ejercicio 3 — Productor–Consumidor (implementación, comparación y aporte)

1) Qué se implementó

Se resolvió el problema clásico **Productor–Consumidor con buffer acotado N** usando el patrón `empty / full / mutex`:

- **C/**
 - hilos: `pc_threads.c` con `pthread + sem_init/sem_wait/sem_post` y buffer circular.
 - procesos: `pc_proc_clean.c` con **memoria compartida POSIX** (`shm_open + mmap`) y **semáforos con nombre** (`sem_open`). Incluye **limpieza** en `SIGINT` para permitir *reruns* sin objetos POSIX “colgados”.
- **Java/**
 - hilos: `PCThreads.java` con `Semaphore + buffer circular`.
 - procesos (**cliente/servidor por sockets**):
 - * `PCServer.java`: mantiene el recurso (buffer) con `ArrayBlockingQueue`.
 - * `ProducerClient.java` y `ConsumerClient.java`: envían “PUT X” / “TAKE” al servidor.
 - * Usa `setReuseAddress(true)` y `shutdown hook` para **rerun limpio**.
- **Python/**
 - hilos: `pc_threads.py` con `threading.Semaphore y Lock + buffer circular`.
 - procesos: `pc_processes.py` con `multiprocessing.Queue(maxsize=N)` (bloquea al estar lleno/vacío; no requiere semáforos manuales).

Nota: En todas las versiones de “hilos” el **buffer circular** usa índices **in/out** avanzando módulo **N**, y **no** hay sobrescritura: **empty/full** bloquean cuando corresponde.

2) Comparación (literal c)

Correctitud y modelo de memoria - Hilos (C/Java/Python): memoria compartida **implícita** → se requiere `mutex` para sección crítica y `empty/full` para capacidad/elementos. - **Procesos:** - **C:** memoria compartida **explícita** (POSIX) + semáforos **con nombre** compartidos por kernel. - **Java:** no hay semáforos inter-proceso estándar; la opción **simple y portable** es `sockets` con un servidor que centraliza el buffer. - **Python:** `multiprocessing.Queue` ya implementa bloqueo y canal IPC.

Simplicidad de implementación (de más simple a más compleja en esta tarea) - Python procesos (cola lista) ~ Java procesos (sockets) ~ Java hilos - C hilos - C procesos (por manejo de `shm_*`, `sem_open` y limpieza)

Overhead / rendimiento (cuantitativo) - **Hilos**: más ligeros; comparten memoria y el *context switch* suele ser más barato. - **Procesos**: coste de IPC: - **C (SHM)**: eficiente pero requiere más código/limpieza. - **Java (sockets loopback)**: pila de red local, suficiente para la demo. - **Python (multiprocessing.Queue)**: pipes/locks internos con rendimiento aceptable y código mínimo.

Robustez / rerun - **C procesos**: `sem_close/sem_unlink`, `munmap`, `shm_unlink` y handler de señal → sin residuos en `/dev/shm`. - **Java procesos**: `SO_REUSEADDR + shutdown hook` → rebind inmediato del puerto. - **Hilos (C/Java/Python)**: sin recursos persistentes → rerun trivial.

Facilidad de depuración - Java/Python: excepciones legibles, menos API de bajo nivel. - **C procesos**: revisar nombres POSIX, permisos y limpieza (propenso a typos en `sem_unlink/shm_unlink` si no se usan constantes).

3) Resultados de las implementaciones

A continuación se presentan capturas de pantalla de la ejecución de cada implementación del problema Productor-Consumidor, demostrando que todas funcionan correctamente:

```
^Crosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/C$ ./C_threads
produced 1
consumed 1
produced 2
consumed 2
produced 3
consumed 3
produced 4
produced 5
consumed 4
produced 6
consumed 5
produced 7
produced 8
consumed 6
produced 9
```

Figure 1: Ejecución C Hilos

C - Hilos (pthread) Descripción (observable en la captura): - Se ven líneas intercaladas de producción y consumo en consola. - La ejecución avanza de forma continua sin errores visibles. - Los mensajes muestran que ambos roles (productor y consumidor) están activos.

C - Procesos (memoria compartida POSIX) Descripción (observable en la captura): - Se observan mensajes de productores y consumidores impresos

```
rosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/C$ ./C_processes
produced 1
consumed 1
produced 2
consumed 2
produced 3
produced 4
consumed 3
produced 5
consumed 4
produced 6
produced 7
consumed 5
produced 8
consumed 6
produced 9
produced 10
```

Figure 2: Ejecución C Procesos

- por procesos. - La salida fluye de forma estable; no se aprecian errores ni bloqueos.
- Muestra que la coordinación entre procesos funciona durante la ejecución.

```
rosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/Java$ java J_threads
produced 1
consumed 1
produced 2
consumed 2
produced 3
consumed 3
produced 4
produced 5
consumed 4
produced 6
consumed 5
produced 7
produced 8
consumed 6
produced 9
consumed 7
```

Figure 3: Ejecución Java Hilos

- Java - Hilos (Semaphore) Descripción (observable en la captura):** - Se ven mensajes de “produce” y “consume” alternándose en la consola Java. - La ejecución es estable y continua, sin excepciones mostradas. - Evidencia de que productores y consumidores están trabajando en paralelo.

- Java - Procesos (Cliente/Servidor por sockets) Descripción (observable en la captura):** - Se aprecian logs que reflejan solicitudes de producción/consumo y respuestas. - La interacción indica que el servidor atiende operaciones y los clientes reciben confirmaciones. - No se muestran errores; la ejecución demuestra que el flujo PUT/TAKE funciona.

```

rosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos
consumed 1
produced 2
consumed 2
produced 3
consumed 3
produced 4
produced 5
consumed 4
produced 6
consumed 5
produced 7
produced 8
consumed 6
produced 9
consumed 7
produced 10

*rosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/Java$ java ConsumerClient
a ConsumerClient
consumed 1
consumed 2
consumed 3
consumed 4
consumed 5
consumed 6
consumed 7
consumed 8
consumed 9
consumed 10
consumed 11
consumed 12
consumed 13
consumed 14
consumed 15
consumed 16
consumed 17
consumed 18
consumed 19
consumed 20
consumed 21
consumed 22

*rosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/Java$ java ProducerClient
put 1 -> OK
put 2 -> OK
put 3 -> OK
put 4 -> OK
put 5 -> OK
put 6 -> OK
put 7 -> OK
put 8 -> OK
put 9 -> OK
put 10 -> OK
put 11 -> OK
put 12 -> OK
put 13 -> OK
put 14 -> OK
put 15 -> OK
put 16 -> OK
put 17 -> OK
put 18 -> OK
put 19 -> OK
put 20 -> OK
put 21 -> OK
put 22 -> OK

```

Figure 4: Ejecución Java Procesos

```

rosvjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/Python$ python3 pc_threads.py
consumed 1
produced 2
consumed 2
produced 3
consumed 3
produced 4
produced 5
consumed 4
produced 6
consumed 5
produced 7
produced 8
consumed 6
produced 9
consumed 7
produced 10

```

Figure 5: Ejecución Python Hilos

Python - Hilos (threading) Descripción (observable en la captura): - Se observan mensajes de producción y consumo impresos de forma intercalada. - La ejecución continúa sin errores visibles. - Se evidencia el funcionamiento correcto del patrón en hilos de Python.

```
^Crosyjames@James:~/Universidad/Semestre_7/Sistemas_Operativos/Tarea02/ejer03/Python$ python3 pc_processes.py
put 1
consumed 1
put 2
consumed 2
put 3
consumed 3
put 4
put 5
consumed 4
put 6
consumed 5
put 7
put 8
consumed 6
put 9
consumed 7
```

Figure 6: Ejecución Python Procesos

Python - Procesos (multiprocessing.Queue) Descripción (observable en la captura): - Se ven procesos imprimiendo producción y consumo, evidenciando el intercambio. - La salida es fluida y sin errores visibles, lo que demuestra coordinación. - Muestra que el programa corre correctamente en el modelo de procesos.

Análisis de las capturas Lo que se aprecia visualmente en todas las capturas es que: 1. Hay ejecución continua con mensajes de producción y consumo intercalados. 2. No se observan errores, excepciones ni bloqueos en las salidas mostradas. 3. La alternancia produce/consume evidencia la coordinación básica correcta. 4. El objetivo de las capturas es demostrar funcionamiento real en consola; los detalles internos (semáforos/colas/sockets) están en el código y la documentación.

4) Lecciones aprendidas (por lenguaje)

- **C**
 - Hilos: patrón `empty/full/mutex` con `sem_init` funciona de forma directa.
 - Procesos: diferencia entre `crear/abrir` (`sem_open / shm_open`) y `desvincular` (`sem_unlink / shm_unlink`); sin limpieza, los reruns fallan.
- **Java**
 - Hilos: `Semaphore` + arreglo circular replican la solución de C de forma clara.

- Procesos: **IPC vía sockets** simplifica; **ArrayBlockingQueue** encapsula `empty/full/mutex`.
 - **Python**
 - Hilos: `threading.Semaphore` es suficiente; mismo patrón.
 - Procesos: `multiprocessing.Queue(maxsize=N)` resuelve bloqueo y capacidad sin semáforos extra → mínimo y legible.
-

5) Buenas prácticas incorporadas (para pasar el “rerun test”)

- **C procesos:**
 - Constantes para nombres POSIX (evitar typos).
 - Handler SIGINT/SIGTERM para `sem_close + sem_unlink + munmap + shm_unlink`.
 - **Java procesos:**
 - `ServerSocket#setReuseAddress(true)` y `shutdown hook` con `server.close()`.
 - **General:**
 - `printf/println` solo informativo (el orden puede variar por planificación).
 - `N=1` si se quiere visualizar alternancia estricta produce/consume.
-

6) Aporte individual (James)

- Implementación y validación de las cuatro variantes del ejercicio 3:
 - **C (hilos y procesos):** patrón `empty/full/mutex` y **limpieza POSIX** para rerun.
 - **Java (hilos y procesos):** arquitectura **cliente/servidor** por sockets con `ArrayBlockingQueue` como recurso compartido.
 - **Python (hilos y procesos):** versiones mínimas con `threading` y `multiprocessing.Queue`.
 - Pruebas de ejecución (WSL Ubuntu) y ajuste de tiempos/N para visualizar el comportamiento.
 - Redacción de la **comparación (literal c)** y de las **lecciones aprendidas** para el informe.
-

7) Cómo compilar y ejecutar (resumen del ejercicio 3)

```
# C - hilos
cd sistemas-operativos-ejer03/C/hilos
gcc pc_threads.c -o pc_threads -pthread
./pc_threads # Ctrl+C para terminar
```

```

# C - procesos
cd ../procesos
gcc pc_proc_clean.c -o pc_proc_clean -pthread -lrt    # -lrt puede no ser necesario según diseño
./pc_proc_clean  # Ctrl+C + limpia semáforos y SHM

# Java - hilos
cd ../../Java/hilos
javac PCThreads.java
java PCThreads      # Ctrl+C para terminar

# Java - procesos (3 terminales)
cd ../procesos
javac PCServer.java ProducerClient.java ConsumerClient.java
# Terminal A
java PCServer
# Terminal B
java ProducerClient
# Terminal C
java ConsumerClient

# Python - hilos
cd ../../Python/hilos
python3 pc_threads.py    # Ctrl+C para terminar

# Python - procesos
cd ../procesos
python3 pc_processes.py # Ctrl+C para terminar

```

8) Uso de LLMs (documentación mínima)

- Se consultó un LLM para:
 - Confirmar la diferencia conceptual entre **hilos** y **procesos** en el patrón Productor-Consumidor.
 - Recomendar **limpieza de recursos POSIX** (`sem_unlink` / `shm_unlink`) y **shutdown hooks** para permitir *reruns* sin residuos.
 - Sugerir una **arquitectura mínima** de procesos en Java basada en **sockets** (servidor con `ArrayBlockingQueue` + clientes `PUT/TAKE`).
- **Verificación local:** Todas las implementaciones se ejecutaron y verificaron en **WSL Ubuntu**.
- **Criterios finales de diseño:** Los **nombres POSIX**, la **estructura de sockets** y el **formato de impresión** fueron ajustados por el equipo para claridad, robustez y facilidad de rerun.
- **Enlace a la conversación (historial LLM):** <https://chatgpt.com/g/g-p-68e93448e19081918b82ecf0165915a5-james/shared/c/68ebf193-d704-832c>

92c2-f812feced7d1?owner_user_id=user-g7JsUaFTsow7u6klxEvGNMOr

Pregunta 4 — Propuesta de Taller/Laboratorio (Cambios al Kernel)

Proponer un taller/laboratorio en el ámbito de **cambios al Kernel** (Unix/Linux) que podamos probar en **máquinas virtuales ARM** o en **hardware ARM como Raspberry Pi**. :contentReferenceoaicite:2

1) Laboratorio Principal: Sistema de Logs Mejorado en el Kernel

Objetivo

Implementar mejoras de **logging** en el kernel que permitan: - **Filtrar** mensajes por **nivel de severidad** (p. ej., KERN_INFO, KERN_WARNING, etc.). - **Filtrar** por **PID** (proceso específico). - **Exponer estadísticas** de logs por proceso (contadores por nivel).

Diseño (mínimo viable y defendible)

- **Punto de enganche:** utilizar la ruta de `printk()` (o capa `pr_*`) para aplicar **filtros** *antes* de encolar al buffer de `dmesg`.
- **Interfaz de control:** `sysfs` o `procfs` (evita agregar `syscall` y tocar tablas por arquitectura).
 - Ejemplo:
 - * `/sys/kernel/logfilter/enabled` (0/1)
 - * `/sys/kernel/logfilter/level_min` (0–7)
 - * `/sys/kernel/logfilter/pid_allowlist` (lista CSV o un único PID)
 - * `/proc/logstats/<pid>` → contadores por nivel
- **Estructuras simples:**
 - `level_min` (entero global atómico),
 - `allow_pid` (entero; -1 = sin filtro),
 - `per-pid stats` con `idr/hash + atomic64_t` por nivel (o primer MVP: un solo PID, luego extender).
- **Reglas:**
 - Si `enabled=0` → comportamiento estándar.
 - Si `enabled=1` → descartar `printks` con `level < level_min` o `pid != allow_pid` (cuando `allow_pid >= 0`).

Por qué es interesante

- **Educativo:** delinea flujo de printk, subsistemas de proc/sysfs, atomicidad y concurrencia.
 - **Práctico:** debugging más eficiente (foco por PID/nivel).
 - **Verificable:** se valida con dmesg, cat en sysfs/proc y pruebas de carga.
-

2) Alternativas (con factibilidad)

- 1) **Driver de Dispositivo Virtual (char device)**
 - Crear /dev/mi_dispositivo con read/write/ioctl.
 - Muy bueno para aprender VFS, file_operations, memoria usuario-kernel.
 - **Nivel:** Fácil–Medio. **Riesgo** bajo. **Test** directo con echo/cat y strace.
- 2) **Security Module (LSM) simple**
 - Activar LSM hook limitado (p. ej., loguear open en /etc/shadow o paths “sensibles”).
 - **Nivel:** Medio. Requiere Kconfig y conocer la **orden de hooks**.
 - Útil para explorar control de acceso y auditoría básica.
- 3) **Política de Scheduling EDF (Earliest Deadline First)**
 - Modificar kernel/sched/ e integrar una cola EDF de juguete (no RT duro).
 - **Nivel:** Medio–Alto. Riesgo de inestabilidad si se toca el planificador principal.
 - Recomendado solo si ya se domina el subsistema sched/*.

Recomendación: para cumplir tiempos y tener un **demo sólido**, priorizar el **Sistema de Logs Mejorado**, y dejar el **char device** como “plan B” si hay contratiempos.

3) Entorno de Desarrollo y Ejecución (ARM)

Opción A: QEMU (recomendada para iterar rápido)

Requisitos (host Linux/WSL): sudo apt-get update sudo apt-get install -y gcc-aarch64-linux-gnu qemu-system-arm qemu-system-aarch64 make bc flex bison libssl-dev libelf-dev

Compilar kernel ARM64 (ejemplo): export ARCH=arm64 export CROSS_COMPILE=aarch64-linux-gnu- make defconfig # Verificar que estén activos: CONFIG_PRINTK, CONFIG_PROC_FS, CONFIG_SYSFS make -j”\$(nproc)”

Arrancar con QEMU (consola serie): qemu-system-aarch64

```
-M virt -cpu cortex-a57 -m 1024  
-kernel arch/arm64/boot/Image  
-append "console=ttyAMA0"  
-nographic
```

Notas: - Ctrl+A, luego X para salir de QEMU con -nographic. - Si necesitas utilidades de user space, añade un initramfs mínimo.

Opción B: Raspberry Pi (hardware real)

Pasos generales: 1) Instalar Raspberry Pi OS en la SD. 2) Compilar el kernel para RPi (usar el repo/branch oficial de RPi). 3) Reemplazar kernel8.img en la partición /boot. 4) Instalar módulos si aplica: make modules_install INSTALL_MOD_PATH=

Flujo recomendado: - Validar primero en QEMU (arranque OK). - Luego probar en la Raspberry Pi.

4) Plan de Implementación por Fases

Fase 0 — “Hello kernel” (sanidad del entorno)

- Compilar kernel vanilla y booteo **OK** en QEMU (ARM/ARM64).

Fase 1 — Interfaz de control (sysfs/proc)

- Crear nodos: enabled, level_min, allow_pid.
- Validar lectura/escritura desde user space (echo / cat).

Fase 2 — Hook de filtrado en printk

- Interceptar antes de encolar/emitir al buffer de dmesg.
- Aplicar filtros (enabled, level_min, allow_pid).

Fase 3 — Estadísticas por PID

- Estructura ligera (pid → counters[0..7]).
- Exponer /proc/logstats/<pid>.

Fase 4 — Pruebas y hardening

- Concurrencia: spinlock_t/atomics donde corresponda.
- Carga: spamear pr_info/pr_warn con distintos PIDs y niveles.
- Medición cualitativa con/sin filtros.

5) Plan de Pruebas (mínimo defendible)

1) Funcionalidad básica

- enabled=0 → dmesg estándar.
- enabled=1, level_min=4 (WARN) → INFO/DEBUG no aparecen.

- `allow_pid=<PID>` → solo logs del PID indicado.
- 2) **Estadísticas**
- Ejecutar 2–3 productores de logs con niveles distintos.
 - Verificar `/proc/logstats/<pid>` (totales por nivel).
- 3) **Estabilidad**
- Stress con `fork` y `printf/pr_*` en bucle.
 - Sin `kernel panic`/WARNings inesperados.
- 4) **Rerun**
- Reboot del kernel → filtros reseteados por defecto (configurable).
-

6) Criterios de Evaluación

- Funcionalidad y estabilidad del cambio (sin `panic`).
 - Documentación clara (qué, por qué, cómo probar).
 - Originalidad / valor: filtros por nivel + PID + stats.
 - Alineación con entregables: informe con lecciones aprendidas, repo con código, gestión de roles y documentación de LLMs.
-

7) Roles y Gestión

- **Líder**: coordina tareas, revisa PRs, agenda pruebas.
 - **Kernel dev**: implementa filtros y sysfs/proc.
 - **QA/Infra**: QEMU/ARM, scripts de prueba, documentación.
-

8) Riesgos y Mitigaciones

- **Rendimiento**: filtro O(1), minimizar locks, usar atomics.
- **Condiciones de carrera**: proteger lectura/escritura de filtros con `spinlock`/barreras.
- **Portabilidad**: cambios acotados, evitar `syscall` en el MVP.
- **Tiempo**: si se complica, pivot a **driver char** (alternativa 1).

9) Alternativa de Respaldo

Driver char /dev/mi_dispositivo con read/write/ioctl y contador interno.

- Fácil de demostrar, bajo riesgo, aprendizaje claro de VFS.

10) Uso de LLMs (documentación mínima)

- Se consultó un LLM para:
 - Elegir un **MVP** factible (filtros en printk + sysfs/proc).
 - Recomendaciones de **entorno ARM con QEMU** y minimizar invasión (evitar syscall).
 - Diseño de **plan de pruebas y mitigaciones**.
- **Verificación local:** compilar/boot en QEMU y probar con dmesg/cat/echo.
- **Criterios finales de diseño:** nombres de nodos sysfs/proc y formato de estadísticas ajustados por el equipo.