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 (cualitativo) - **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:

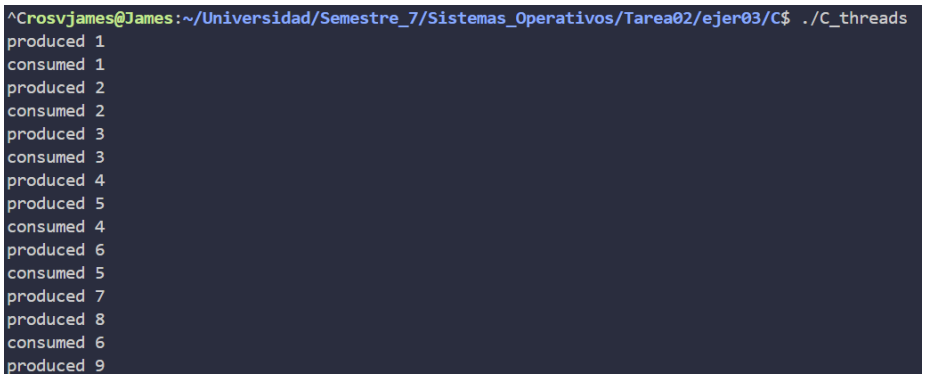

Figure 1: Ejecución C Hilos

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

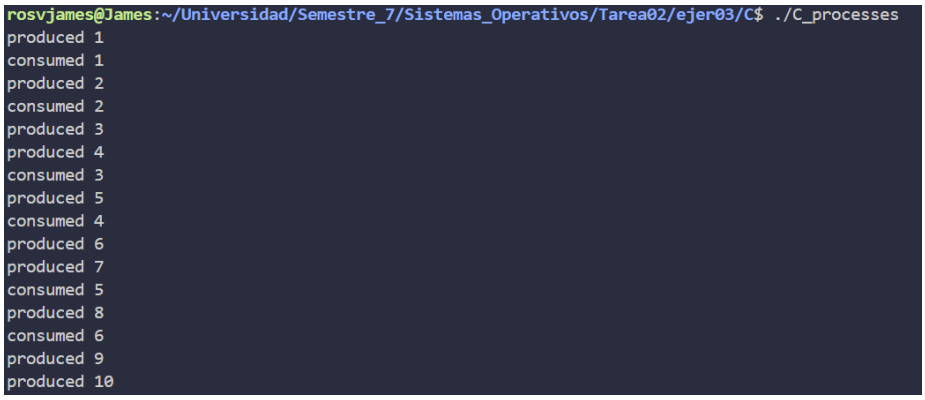

Figure 2: Ejecución C Procesos

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

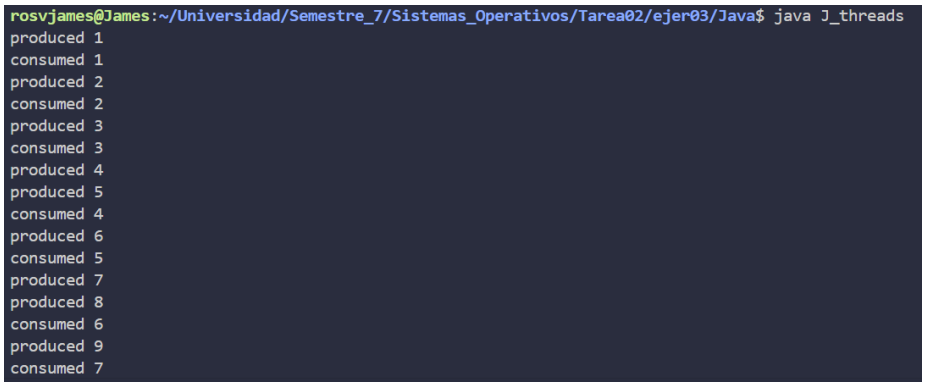

Figure 3: Ejecución Java Hilos

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.

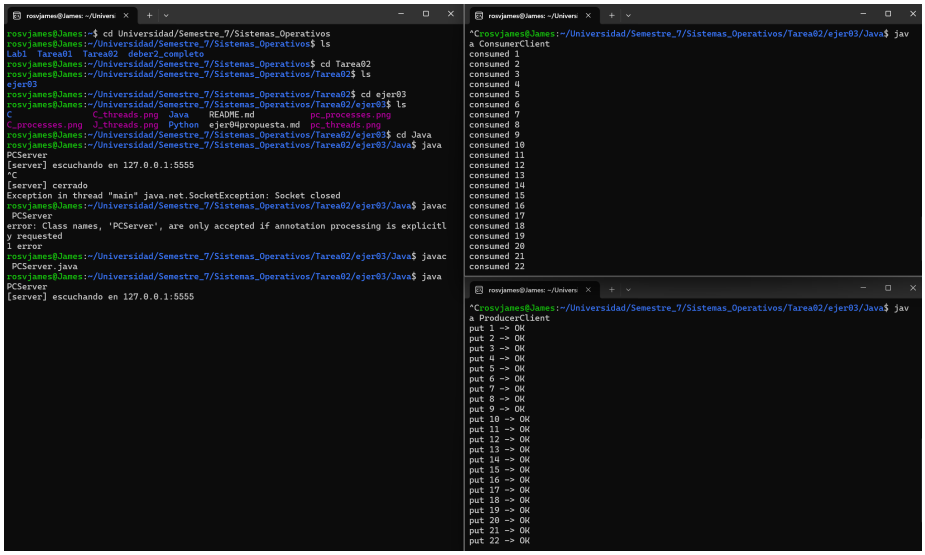

Figure 4: Ejecución Java Procesos

```
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.

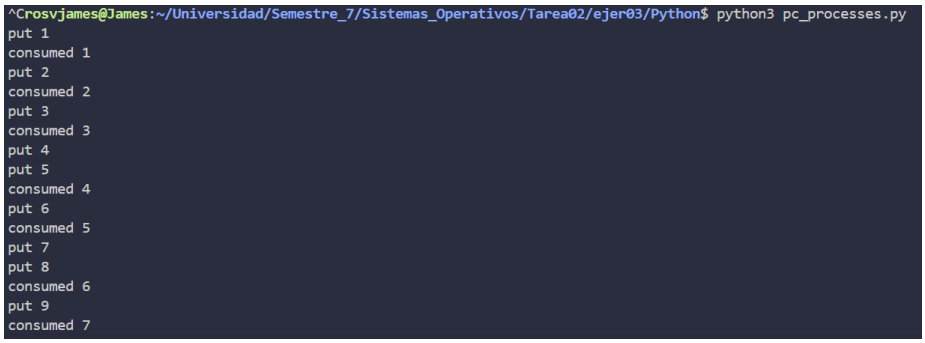

Figure 6: Ejecución Python Procesos

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 dis
./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