



QUEUES

Sección 1: ¿Qué son las Queues y por qué son importantes?

Explicación teórica

Una **Queue** (cola) es una estructura de datos **FIFO** (*First In, First Out*), es decir, el primer elemento en entrar es el primero en salir. En programación concurrente, las colas se utilizan para **sincronizar y comunicar procesos o hilos** que se ejecutan en paralelo o de manera concurrente.

¿Por qué se usan en sistemas operativos?

Los sistemas operativos y entornos multitarea usan queues para:

- Coordinar el acceso a recursos compartidos.
- Pasar mensajes entre procesos o hilos sin usar memoria compartida.
- Implementar **colas de planificación de procesos** (ej.: procesos listos para ejecutarse).
- Evitar condiciones de carrera mediante un mecanismo seguro de paso de mensajes.

Comparación rápida con otras herramientas

| Mecanismo | Comunicación | Memoria compartida | Bloqueo automático |
|----------------|----------------|------------------------------------|---------------------------------------|
| Pipes | Unidireccional | No | No |
| Shared memory | Bidireccional | Sí | No |
| Queues (colas) | Bidireccional | No (maneja sus datos internamente) | Sí (maneja el acceso automáticamente) |

¿Cuándo usar Queues?

- Cuando querés que un proceso o hilo envíe tareas o datos a otro de forma ordenada y segura.
- Cuando querés evitar bloqueos manuales (como `Lock`) y preferís una solución de alto nivel.

Sección 2: Implementación interna y ciclo de vida de una Queue

Teoría: ¿Cómo funcionan las Queues a nivel de sistema operativo?

Concepto base

Una Queue en programación concurrente **no es solo una estructura de datos**; es una **herramienta de sincronización**. En sistemas UNIX/Linux, las Queues pueden ser implementadas por:

- **Colas de mensajes POSIX** (`mq_*`)
- **Colas de mensajes System V**
- **Buffers internos gestionados por bibliotecas** (como `multiprocessing.Queue` en Python)

Ciclo de vida de una Queue

1. **Creación:** Un proceso (o el padre) crea la Queue. Se asigna memoria y un sistema de control.
2. **Envío de datos:** Un proceso coloca datos en la cola mediante `put()`.
3. **Recepción de datos:** Otro proceso extrae datos mediante `get()`.
4. **Bloqueo (automático):**

- Si se intenta `get()` en una cola vacía, el proceso espera (bloquea).
- Si se intenta `put()` en una cola llena (si tiene capacidad limitada), también se bloquea.

5. **Dstrucción:** Cuando ya no se necesita, se libera la cola y sus recursos.

¿Qué pasa con los procesos?

Cada proceso que interactúa con la Queue debe tener acceso a su referencia (como una instancia compartida). La Queue garantiza que el paso de mensajes sea **atómico y ordenado**.

Ahora sí, pasamos a la Sección 3: Implementación práctica de Queues en Python

Vamos paso a paso con código real usando el módulo `multiprocessing`.

¿Qué vamos a usar?

```
from multiprocessing import Process, Queue
```

Este módulo nos permite lanzar procesos que se comunican a través de una Queue segura.

Ejemplo básico paso a paso

```
from multiprocessing import Process, Queue
import time

# Función que envía datos
def productor(q):
    for i in range(5):
        print(f"[Productor] Enviando {i}")
        q.put(i) # Poner datos en la cola
        time.sleep(0.5)

# Función que recibe datos
```

```
def consumidor(q):
    for _ in range(5):
        dato = q.get() # Espera si la cola está vacía
        print(f"[Consumidor] Recibido {dato}")

if __name__ == "__main__":
    q = Queue() # Crear la cola
    p1 = Process(target=productor, args=(q,))
    p2 = Process(target=consumidor, args=(q,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print("Comunicación finalizada.")
```

¿Qué hace este programa?

- Crea dos procesos: uno produce datos y otro los consume.
- Usa una Queue para enviar enteros del 0 al 4.
- El productor pone datos con `put()`.
- El consumidor los recibe con `get()` y los imprime.

Sección 4: Ejemplos prácticos unidireccionales

Aquí vas a ver cómo las Queues pueden usarse para **simular tareas reales**, como la recolección de datos y su posterior análisis, usando dos procesos que se comunican en una sola dirección.

Ejemplo: Recolector de sensores y procesador de datos

```
from multiprocessing import Process, Queue
import time
import random
```

```

# Simula un sensor que genera datos de temperatura
def recolector(q):
    for _ in range(10):
        temp = round(random.uniform(20.0, 30.0), 2)
        print(f"[Sensor] Temperatura: {temp}°C")
        q.put(temp)
        time.sleep(0.5)

# Simula un proceso que procesa esos datos
def procesador(q):
    for _ in range(10):
        dato = q.get()
        print(f"[Procesador] Recibido: {dato}°C → {'ALTA' if dato > 25 else 'NORMAL'}")

if __name__ == "__main__":
    cola = Queue()
    p_sensor = Process(target=recolector, args=(cola,))
    p_procesador = Process(target=procesador, args=(cola,))

    p_sensor.start()
    p_procesador.start()

    p_sensor.join()
    p_procesador.join()

    print("Proceso de recolección y análisis finalizado.")

```

¿Qué representa este ejemplo?

- El **sensor** produce datos de temperatura y los pone en una **Queue**.
- El **procesador** toma esos datos y evalúa si la temperatura está alta o normal.
- La comunicación es **unidireccional**: solo va del sensor al procesador.

Sección 6: Estrategias para prevenir problemas comunes con Queues

En sistemas concurrentes, el mal manejo de Queues puede provocar:

- **Procesos bloqueados indefinidamente**
- **Deadlocks**
- **Datos perdidos**
- **Cuellos de botella**

Veamos cómo **evitar estos errores**.

Problema 1: `get()` o `put()` se bloquean indefinidamente

¿Por qué pasa?

- Un proceso llama a `get()` pero **nadie puso datos**.
- O se llama a `put()` pero la cola está **llena y nadie saca**.

Estrategia:

- Usá los parámetros `timeout` o `get_nowait()` / `put_nowait()`.
-

Problema 2: Deadlocks

¿Por qué pasa?

- Dos o más procesos esperan indefinidamente uno por el otro (ej: A espera datos de B, pero B está esperando respuesta de A).

Estrategia:

- Usar **colas separadas para cada dirección de comunicación**.
 - **Diseñar bien los flujos de datos**: evitar ciclos innecesarios.
 - Implementar **timeouts** o **contadores** para evitar esperas infinitas.
-

Problema 3: Recursos que no se liberan









¿Por qué pasa?

- Si un proceso falla o no llama a `close()` o `join()` correctamente.
- Esto deja procesos zombis o colas bloqueadas.

Estrategia:

- Usar bloques `try-finally` o `with` cuando sea posible.
- Siempre hacer `.join()` a todos los procesos hijos.
- Documentar bien qué hace cada proceso y cómo se sincroniza.

Sección 7: Diferencias entre Queues y Pipes

| Característica | Queue (multiprocessing.Queue) | Pipe (multiprocessing.Pipe) |
|---|--|--|
|  Modelo de datos | FIFO (cola de datos) | Canal de lectura/escritura entre 2 extremos |
|  Dirección | Unidireccional o bidireccional (flexible) | Normalmente bidireccional |
|  Número de procesos | Varios productores y/o consumidores | Recomendado para 1:1 o máximo 2 procesos |
|  Seguridad | Maneja su propio locking internamente | El locking debe ser manejado manualmente |
|  Manejo de errores | Mejor soporte (timeouts, excepciones) | Más propenso a bloqueos si no se usa con cuidado |
|  Facilidad de uso | Más simple para patrones tipo productor-consumidor | Requiere más diseño manual |
|  Capacidad | Puede tener tamaño limitado (<code>maxsize</code>) | No se limita directamente |
|  Casos ideales | Varios procesos compartiendo datos | Comunicación directa entre dos procesos |

Conclusión

Usar **Queue** cuando...

- ✓ Querés manejar varios procesos
- ✓ Necesitás seguridad y facilidad
- ✓ Trabajás con patrones como productor-consumidor
- ✓ Querés prevenir errores comunes

Usar **Pipe** cuando...

✓ Tenés solo dos procesos

✓ Querés control más bajo nivel

✓ Estás construyendo algo muy optimizado y sabés lo que hacés
