CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS (CUCEI)

Departamento de ciencias computacionales

Seminario de solución de problemas de uso, adaptación, explotación de sistemas operativos

Violeta del Rocío Becerra Velázquez

Jose Pedro Reyes Alvarez

222790897

Ingeniería Informática (INNI)

D02

2.4 Productor Consumidor

13 de abril del 2025

Índice

El problema del productor-consumidor

El problema del productor-consumidor es un clásico en la programación concurrente. Consiste en dos procesos: el productor, que genera datos y los coloca en un buffer (contenedor), y el consumidor, que toma esos datos para procesarlos. Ambos comparten el mismo buffer, por lo que deben coordinarse para evitar errores como acceso simultáneo, pérdida de datos o inconsistencias.

La concurrencia se refiere a la ejecución simultánea de múltiples procesos o hilos dentro de un programa. Es fundamental en sistemas multitarea y se utiliza para optimizar el uso de recursos del sistema. Sin embargo, puede generar conflictos si varios procesos acceden al mismo recurso sin control adecuado.

Los semáforos son mecanismos de sincronización que permiten controlar el acceso concurrente a recursos compartidos. Usan contadores para indicar si un recurso está disponible. Ayudan a evitar problemas como:

Interbloqueo (deadlock): cuando dos procesos esperan indefinidamente uno por el otro.

Inanición (starvation): cuando un proceso nunca obtiene acceso al recurso porque otros siempre se le adelantan.

Con los semáforos se asegura la exclusión mutua, es decir, que solo un proceso acceda a la vez al recurso crítico.

Escenarios típicos del problema:

Un productor y un consumidor: el más simple, ideal para ilustrar el problema.

Múltiples productores y consumidores: requiere mayor control para evitar condiciones de carrera.

Buffer acotado (limitado): cuando el contenedor tiene una capacidad máxima, lo cual requiere verificar espacio antes de producir y verificar contenido antes de consumir.

Buffer circular: cuando el final del buffer se conecta con el inicio, optimizando el uso de memoria.

Solución propuesta

Para resolver el problema del productor-consumidor, se eligió desarrollar una aplicación en Python, utilizando la librería Tkinter para crear una interfaz gráfica que permita visualizar el estado del sistema en tiempo real. Esta solución aplica semáforos y hilos (threads) para manejar la concurrencia de manera controlada, garantizando la exclusión mutua en el acceso al buffer.

Características principales de la solución:

- Se implementa un único productor y consumidor.

- El buffer se representa como un arreglo circular de 25 posiciones, visualizado con rectángulos en pantalla, numerados del 0 al 24.
- El productor y el consumidor trabajan con tiempos de espera aleatorios para simular que “duermen”.
- Ambos pueden producir o consumir de 1 a 5 elementos por ciclo, lo cual es también aleatorio.
- Solo uno a la vez puede acceder al buffer, asegurando la exclusión mutua mediante un semáforo.
- El programa visualiza los siguientes mensajes en pantalla:
  - Si el productor o el consumidor está dormido.
  - Si están intentando acceder al buffer.
  - Si están trabajando.
- Las casillas del buffer cambian de color según su estado:
  - Blanco: vacía.
  - Verde: llena (producto).
- El programa se puede cerrar presionando el botón de salir.

Explicación del código

El programa se organiza en una clase principal llamada App, que gestiona tanto la interfaz gráfica como la lógica de concurrencia. A continuación, se describen las partes más importantes del código:

Inicialización del entorno

Se crea una ventana con Tkinter, donde se dibujan 25 rectángulos que representan las posiciones del buffer. Cada casilla es numerada usando canvas.create_text.

```
for i in range(BUFFER_SIZE):
    x = i * slot_width + 10
    slot = self.canvas.create_rectangle(x, 10, x + 30, 60, fill="white", outline="black")
    self.buffer_slots.append(slot)
    # Agrega el número centrado dentro del rectángulo
    self.canvas.create_text(x + 15, 35, text=str(i), font=("Arial", 8))
```

Variables clave

- self.buffer: lista con 25 posiciones inicializadas en None.
- self.mutex: semáforo para garantizar que solo uno (productor o consumidor) acceda al buffer.

- self.buffer_full y self.buffer_empty: semáforos que controlan cuándo se puede producir o consumir.
- self.in_index y self.out_index: controlan las posiciones actuales de escritura y lectura en el buffer circular.

Hilos del productor y consumidor

Cada uno corre en su propio hilo. Tienen comportamiento cíclico: duermen, intentan acceder al buffer, y si pueden, producen o consumen entre 1 y 5 elementos.

```
def productor(self):
    global in_index
    while not stop_program:
        tiempo = random.randint(1, 3)
        self.set_estado_productor(f"Dormido ({tiempo}s)")
        time.sleep(tiempo)
        self.set_estado_productor("Despierto, intentando producir...")
        while not en_buffer.acquire(blocking=False):
            self.set_estado_productor("Esperando acceso al contenedor...")
            time.sleep(0.5)
        self.set_estado_general("Contenedor: Productor trabajando")
```

Conclusión

Con este programa me resultó una manera clara de entender cómo funciona un buffer compartido entre procesos concurrentes. Antes de desarrollar la solución, tenía una idea muy general del problema del productor-consumidor, pero al implementarlo, comprendí mucho mejor conceptos clave como la exclusión mutua, el interbloqueo, la inanición y el uso de semáforos para sincronizar el acceso a recursos limitados.

El hecho de visualizar gráficamente cada paso del productor y del consumidor me permitió ver en tiempo real cómo se llenan y vacían los espacios del buffer, y cómo se respeta el orden circular. También pude observar la importancia de los tiempos aleatorios y cómo pueden provocar conflictos si no se gestiona bien la concurrencia.

Bibliografía

Santos, R. (1997). Problema del Productor-Consumidor. Universidad de Valladolid. Recuperado el 13 de abril de 2025, de https://www2.infor.uva.es/~cllamas/concurr/pract97/rsantos/index.html

Departamento de Ciencias de la Computación. (s.f.). Sincronización de procesos – Práctico 4. Universidad Nacional del Sur. Recuperado el 13 de abril de 2025, de https://cs.uns.edu.ar/~gd/soyd/practicos/pr%C3%A1ctico4-Sincronizaci%C3%B3ndeprocesos.pdf

Formella, A. (s.f.). Sincronización de procesos: El problema del productor-consumidor. Universidade de Vigo. Recuperado el 13 de abril de 2025, de https://formella.webs.uvigo.es/doc/cd05/node85.html

Código completo

```
import threading

import time

import random

import tkinter as tk


BUFFER_SIZE = 25

buffer = [None] * BUFFER_SIZE

in_index = 0

out_index = 0
```

```
mutex = threading.Semaphore(1)
empty = threading.Semaphore(BUFFER_SIZE)
full = threading.Semaphore(0)

en_buffer = threading.Lock()
stop_program = False

class App:
    def __init__(self, root):
        self.root = root
        self.root.title("Productor - Consumidor")
        self.root.geometry("950x420")
        self.root.resizable(False, False)

        # Estados
        self.label_estado_productor = tk.Label(root, text="Estado Productor: Dormido",
font=("Arial", 12), fg="blue")
        self.label_estado_productor.pack()

        self.label_estado_consumidor = tk.Label(root, text="Estado Consumidor: Dormido",
font=("Arial", 12), fg="green")
        self.label_estado_consumidor.pack()

        self.label_estado_general = tk.Label(root, text="Contenedor: Libre", font=("Arial", 12),
fg="purple")
        self.label_estado_general.pack()

        self.canvas = tk.Canvas(root, width=930, height=100, bg="white")
        self.canvas.pack(pady=20)
        self.buffer_slots = []
```

```
        slot_width = 35
        for i in range(BUFFER_SIZE):
            x = i * slot_width + 10
            slot = self.canvas.create_rectangle(x, 10, x + 30, 60, fill="white", outline="black")
            self.buffer_slots.append(slot)
            # Agrega el número centrado dentro del rectángulo
            self.canvas.create_text(x + 15, 35, text=str(i), font=("Arial", 8))



        self.btn_salir = tk.Button(root, text="Salir", command=self.terminar_programa)
        self.btn_salir.pack(pady=10)

        self.hilo_productor = threading.Thread(target=self.productor)
        self.hilo_consumidor = threading.Thread(target=self.consumidor)

        self.hilo_productor.start()
        self.hilo_consumidor.start()

        self.actualizar_grafico()

    def actualizar_grafico(self):
        for i, item in enumerate(buffer):
            color = "white"
            if item:
                color = "orange"
            self.canvas.itemconfig(self.buffer_slots[i], fill=color)
        if not stop_program:
            self.root.after(200, self.actualizar_grafico)

    def terminar_programa(self):
```

```
    global stop_program
    stop_program = True
    self.root.destroy()

def productor(self):
    global in_index
    while not stop_program:
        tiempo = random.randint(1, 3)
        self.set_estado_productor(f"Dormido ({tiempo}s)")
        time.sleep(tiempo)

        self.set_estado_productor("Despierto, intentando producir...")

        while not en_buffer.acquire(blocking=False):
            self.set_estado_productor("Esperando acceso al contenedor...")
            time.sleep(0.5)

        self.set_estado_general("Contenedor: Productor trabajando")

        n = random.randint(1, 5)
        producidos = 0

        for _ in range(n):
            if stop_program: break
            if empty._value == 0:
                break
            empty.acquire()
            mutex.acquire()

            item = "🍔"
```

```
            buffer[in_index] = item
            self.set_estado_productor(f"Produciendo {item} en {in_index}")
            in_index = (in_index + 1) % BUFFER_SIZE

            producidos += 1
            mutex.release()
            full.release()
            time.sleep(0.5)

        self.set_estado_productor(f"Terminó de producir {producidos} elementos.")
        self.set_estado_general("Contenedor: Libre")
        en_buffer.release()

    def consumidor(self):
        global out_index
        while not stop_program:
            tiempo = random.randint(1, 3)
            self.set_estado_consumidor(f"Dormido ({tiempo}s)")
            time.sleep(tiempo)

            self.set_estado_consumidor("Despierto, intentando consumir...")

            while not en_buffer.acquire(blocking=False):
                self.set_estado_consumidor("Esperando acceso al contenedor...")
                time.sleep(0.5)

            self.set_estado_general("Contenedor: Consumidor trabajando")

            n = random.randint(1, 5)
            consumidos = 0
```

```
        for _ in range(n):
            if stop_program: break
            if full._value == 0:
                break
            full.acquire()
            mutex.acquire()

            item = buffer[out_index]
            buffer[out_index] = None
            self.set_estado_consumidor(f"Consumiendo {item} de {out_index}")
            out_index = (out_index + 1) % BUFFER_SIZE

            consumidos += 1
            mutex.release()
            empty.release()
            time.sleep(0.5)

        self.set_estado_consumidor(f"Terminó de consumir {consumidos} elementos.")
        self.set_estado_general("Contenedor: Libre")
        en_buffer.release()

def set_estado_productor(self, estado):
    self.label_estado_productor.config(text=f"Estado Productor: {estado}")

def set_estado_consumidor(self, estado):
    self.label_estado_consumidor.config(text=f"Estado Consumidor: {estado}")

def set_estado_general(self, estado):
    self.label_estado_general.config(text=estado)
```

```
# Ejecutar app
root = tk.Tk()
app = App(root)
root.mainloop()
```