Simulación de un Driver de Impresión con Spooling y Memoria Compartida
Entorno: Linux (Ubuntu/Debian o similar) con compilador GCC.

1. Objetivos de Aprendizaje

Al finalizar la práctica, el estudiante podrá:

1. Entender cómo el SO desacopla la E/S de la CPU mediante **Buffers**.
2. Implementar **Memoria Compartida** (simulando acceso DMA).
3. Resolver problemas de **Concurrencia** (Productor-Consumidor) usando semáforos.
4. Simular la latencia de un dispositivo de E/S frente a la velocidad de un proceso de usuario.

2. Marco Teórico

Los dispositivos de E/S son órdenes de magnitud más lentos que la CPU. Para evitar que la CPU se bloquee esperando a la impresora (Gestión de E/S), el SO utiliza un área de memoria (Gestión de Memoria) llamada **Spool** o **Buffer**.

- **Los Procesos de Usuario (Productores):** Generan trabajos de impresión.
- **El Proceso Daemon/Driver (Consumidor):** Lee del buffer y envía los datos al dispositivo físico lentamente.
- **El Problema:** Múltiples procesos quieren escribir en el buffer al mismo tiempo (Condición de carrera) y el buffer tiene un tamaño finito (Gestión de recursos).

3. Escenario de la Práctica

Diseñaremos un sistema compuesto por:

1. **Buffer Circular:** Un segmento de memoria compartida que representa la cola de impresión.
2. **Generador de Trabajos (App):** Proceso que llena el buffer.
3. **Controlador de Dispositivo (Driver):** Proceso que vacía el buffer y simula la impresión.

4. Desarrollo de la Práctica (Paso a Paso)

Fase 1: Configuración del entorno (15 min)

El profesor entregará el "esqueleto" del código. Escribir todo desde cero tomaría más de 2 horas. Los estudiantes deben enfocarse en la lógica de sincronización y memoria.

Archivo common.h (Definiciones compartidas)

```
#ifndef COMMON_H
#define COMMON_H

#include <semaphore.h>

#define SHM_NAME "/spool_buffer"
#define SEM_MUTEX "/sem_mutex"
#define SEM_FULL "/sem_full"
#define SEM_EMPTY "/sem_empty"
#define BUFFER_SIZE 5  // Buffer pequeño para forzar esperas rápidas

typedef struct {
    int id_proceso;
    int id_trabajo;
    char datos[20];
} TrabajoImpresion;

typedef struct {
    TrabajoImpresion buffer[BUFFER_SIZE];
    int in;  // Índice de escritura
    int out; // Índice de lectura
} MemoriaCompartida;

#endif
```

Fase 2: Implementación del Productor (45 min)
El estudiante debe completar las secciones marcadas como TODO. Aquí se ejercita la **Gestión de Procesos** y **Sincronización**.
Archivo productor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include "common.h"

int main() {
    // 1. Abrir Memoria Compartida (Simula acceso a RAM del sistema)
    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, sizeof(MemoriaCompartida));
    MemoriaCompartida *shm = mmap(0, sizeof(MemoriaCompartida), PROT_WRITE,
MAP_SHARED, shm_fd, 0);

    // 2. Inicializar Semáforos (Gestión de Concurrencia)
    sem_t *mutex = sem_open(SEM_MUTEX, O_CREAT, 0666, 1);
    sem_t *empty = sem_open(SEM_EMPTY, O_CREAT, 0666, BUFFER_SIZE);
    sem_t *full = sem_open(SEM_FULL, O_CREAT, 0666, 0);

    // Inicializar índices (solo la primera vez, simplificado para la
práctica)
    // En un caso real, esto se hace en un script de inicio.

    for (int i = 0; i < 10; i++) {
        printf("Proceso %d: Intentando enviar trabajo %d...\n", getpid(), i);

        // --- TODO: INICIO DE SECCIÓN CRÍTICA ---
        // El estudiante debe llenar aquí:
        // 1. Esperar si no hay espacio (wait empty)
        sem_wait(empty);
        // 2. Proteger acceso al buffer (wait mutex)
        sem_wait(mutex);

        // Escritura en memoria (Gestión de Memoria / Paginación implícita)
        shm->buffer[shm->in].id_proceso = getpid();
        shm->buffer[shm->in].id_trabajo = i;
        sprintf(shm->buffer[shm->in].datos, "Doc_%d.pdf", i);
        printf(">>> Buffer[%d] Escrito: Doc_%d.pdf\n", shm->in, i);

        // Mover índice circularmente
        shm->in = (shm->in + 1) % BUFFER_SIZE;

        // 3. Liberar buffer (post mutex)
        sem_post(mutex);
        // 4. Señalizar que hay un nuevo item (post full)
        sem_post(full);
        // --- TODO: FIN DE SECCIÓN CRÍTICA ---

        sleep(1); // Simula tiempo de generación de trabajo (CPU burst)
    }

    return 0;
}
```

Fase 3: Implementación del Consumidor/Driver (30 min)
Este proceso simula el hardware lento.
Archivo consumidor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include "common.h"

int main() {
    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0666);
    MemoriaCompartida *shm = mmap(0, sizeof(MemoriaCompartida), PROT_READ |
PROT_WRITE, MAP_SHARED, shm_fd, 0);

    sem_t *mutex = sem_open(SEM_MUTEX, 0);
    sem_t *empty = sem_open(SEM_EMPTY, 0);
    sem_t *full = sem_open(SEM_FULL, 0);

    printf("--- Driver de Impresora Iniciado ---\n");

    while (1) {
        // --- TODO: INICIO DE SECCIÓN CRÍTICA ---
        // 1. Esperar si no hay trabajos (wait full)
        sem_wait(full);
        // 2. Proteger buffer (wait mutex)
        sem_wait(mutex);

        // Leer dato
        TrabajoImpresion job = shm->buffer[shm->out];
        printf("<<< Imprimiendo: %s (De proceso: %d)\n", job.datos,
job.id_proceso);

        shm->out = (shm->out + 1) % BUFFER_SIZE;

        // 3. Liberar buffer (post mutex)
        sem_post(mutex);
        // 4. Señalizar espacio libre (post empty)
        sem_post(empty);
        // --- TODO: FIN DE SECCIÓN CRÍTICA ---

        // SIMULACIÓN DE E/S:
        // Aquí simulamos que el dispositivo es LENTO.
        // Esto demostrará cómo el buffer se llena si el productor es rápido.
        printf("    ...Hardware ocupado...\n");
        sleep(3);
    }
    return 0;
}
```

Fase 4: Compilación, Ejecución y Análisis (30 min)

1. **Compilar:** gcc productor.c -pthread -lrt -o productor y gcc consumidor.c -pthread -lrt -o consumidor.
2. **Experimento A (Velocidad normal):** Ejecutar ./consumidor en una terminal y ./productor en otra.
3. **Experimento B (Saturación de Buffer):**
    - Modificar productor.c para quitar el sleep(1).
    - Ejecutar.

- **Observación esperada:** El productor llenará los 5 espacios del buffer instantáneamente y luego se bloqueará (Estado **Blocked**) hasta que el consumidor libere espacio.
- *Concepto:* Esto visualiza la **Planificación de Procesos** (el proceso pasa de Running a Waiting) debido a la gestión de E/S.

5. Cuestionario de Evaluación (Entregable)

Para asegurar que los estudiantes conecten la práctica con la teoría, deben responder:

1. **Gestión de Memoria:** ¿Por qué usamos mmap y shm_open en lugar de malloc? ¿Qué pasaría si usáramos malloc entre dos procesos distintos? (Relacionar con Espacio de Direcciones Virtuales).
2. **Interbloqueo (Deadlock):** Si cambiamos el orden de los semáforos en el consumidor (primero wait(mutex) y luego wait(full)), ¿qué problema grave de concurrencia podría ocurrir si el buffer está vacío?
3. **Planificación de Procesos:** Cuando el productor se detiene porque el buffer está lleno (sem_wait), ¿en qué estado pone el Planificador (Scheduler) al proceso? ¿Consume CPU mientras espera?
4. **E/S:** Si el buffer fuera de tamaño 1 byte, ¿cómo afectaría esto al rendimiento del sistema comparado con el tamaño actual? (Relacionar con interrupciones y overhead).