

Sistemas operativos

# **T21 - Concurrencia**

Grupo 7

Santos Alejαndro Arellαno Olαrte // **Santos-arellano**

Jeison Camilo Alfonso Moreno

Jose Villaroel

26/ Octubre /2025

[**T21 - Concurrencia**](#_p826p7lioceq) **1**

[1. INTRODUCCIÓN](#) 2

[2. PARTE A: PROGRAMA MULTIHILO CON MUTEX](#) 2

[2.1 Descripción](#) 2

[2.2 Mecanismo de Sincronización](#) 3

[2.3 Compilación y Ejecución](#) 3

[2.4 Resultados](#) 3

[3. PARTE B: MÚLTIPLES PROGRAMAS CON FIFOS](#) 3

[3.1 Descripción](#) 3

[3.2 Arquitectura](#) 4

[3.3 Compilación y Ejecución](#) 4

[3.4 Resultados](#) 4

[4. PARTE C: FIFOS CON SEÑALES (SIGUSR1)](#) 4

[4.1 Descripción](#) 4

[4.2 Mecanismo de Señales](#) 5

[4.3 Compilación y Ejecución](#) 6

[4.4 Resultados](#) 6

[5. COMPARACIÓN DE ENFOQUES](#) 6

[6. CONCLUSIONES](#) 7

[7. EVIDENCIAS DE EJECUCIÓN](#) 7

[7.1 Estructura de archivos](#) 7

[7.2 Compilación exitosa](#) 7

[7.3 Pruebas de concurrencia](#) 7

[8. CÓDIGO FUENTE](#) 8

[REFERENCIAS](#) 8

## **1. INTRODUCCIÓN**

Este informe presenta la implementación de un sistema de monitoreo climático mediante tres enfoques de concurrencia en Linux: multihilo con mutex, multiproceso con FIFOs, y multiproceso con señales. El objetivo es demostrar técnicas de sincronización y comunicación entre procesos para mantener actualizado un archivo compartido con datos de sensores que operan a intervalos independientes.

## **2. PARTE A: PROGRAMA MULTIHILO CON MUTEX**

### **2.1 Descripción**

Implementación de un programa único (climate\_threads.c) que lanza 4 hilos concurrentes, cada uno simulando un sensor climático:

* **Temperatura:** genera datos cada 8±3 segundos
* **Humedad:** genera datos cada 12±2 segundos
* **Viento:** genera datos cada 5±2 segundos
* **Precipitación:** genera datos cada 10±5 segundos

Todos los hilos escriben en el archivo compartido clima\_actual.txt usando exclusión mutua con pthread\_mutex\_t.

### **2.2 Mecanismo de Sincronización**

La exclusión mutua se implementa con un mutex global para garantizar que solo un hilo pueda acceder a las variables compartidas y al archivo a la vez.

static pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;

La sección crítica (protegida por pthread\_mutex\_lock y pthread\_mutex\_unlock) resguarda:

1. La actualización de la estructura shared (temperatura, humedad, viento, lluvia).
2. La escritura atómica al archivo, usando la técnica rename (escribiendo en clima\_actual.txt.tmp y luego renombrando) para evitar que otros procesos lean un archivo a medio escribir.

### **2.3 Compilación y Ejecución**

cd ParteA\_mutex\_semaforos  
make  
./climate\_threads

### **2.4 Resultados**

[INSERTAR CAPTURA: Terminal mostrando ejecución de climate\_threads con timestamps]

[INSERTAR CAPTURA: Contenido del archivo clima\_actual.txt generado]

**Análisis:** El mutex garantiza que no hay condiciones de carrera. Cada hilo actualiza su variable de forma segura y el archivo se escribe atómicamente, evitando estados inconsistentes, incluso cuando múltiples hilos intentan escribir al mismo tiempo.

## **3. PARTE B: MÚLTIPLES PROGRAMAS CON FIFOS**

### **3.1 Descripción**

Cuatro programas independientes (sensor\_temp, sensor\_humedad, sensor\_viento, sensor\_lluvia) se comunican con un proceso central (central) mediante *named pipes* (FIFOs):

* /tmp/fifo\_temp
* /tmp/fifo\_humedad
* /tmp/fifo\_viento
* /tmp/FIFO\_LLUVIA

Cada sensor opera de forma autónoma y escribe datos con formato: SENSOR;valor;timestamp.

El proceso central usa poll() para monitorear los 4 FIFOs de forma no bloqueante y actualiza clima\_actual.txt en cuanto recibe un dato.

### **3.2 Arquitectura**

sensor\_temp ──────┐  
sensor\_humedad ───┤  
sensor\_viento ────┼──→ [central] ──→ clima\_actual.txt  
sensor\_lluvia ────┘  
 (FIFOs)

### **3.3 Compilación y Ejecución**

cd ParteB\_pipes  
make  
./run.sh

(El script run.sh crea los FIFOs, lanza el proceso central en segundo plano y luego los 4 sensores).

### **3.4 Resultados**

[INSERTAR CAPTURA: Ejecución de run.sh mostrando los 5 procesos activos (usando 'ps' o 'htop')]

[INSERTAR CAPTURA: Salida del 'central' recibiendo datos de los diferentes sensores]

[INSERTAR CAPTURA: Contenido de clima\_actual.txt actualizado]

**Análisis:** Los FIFOs permiten una comunicación asíncrona robusta entre procesos completamente independientes. El uso de poll() en el central es clave para la eficiencia, ya que evita la espera activa (*busy-waiting*) y permite al central reaccionar inmediatamente a cualquier sensor que envíe datos, sin conocer la lógica interna o los intervalos de tiempo de cada uno.

## **4. PARTE C: FIFOS CON SEÑALES (SIGUSR1)**

### **4.1 Descripción**

Esta implementación modifica el enfoque de la Parte B para que los sensores sean pasivos y solo generen datos bajo demanda.

1. Cada sensor (sensor\_\*\_sig) inicia, crea su FIFO, y escribe su PID en un archivo /tmp/sensor\_\*.pid.
2. Los sensores esperan en estado de bajo consumo usando pause(), a la espera de una señal.
3. El proceso central (central\_signal) lee los PIDs de los archivos.
4. Cada 5 segundos, el central envía la señal SIGUSR1 a los 4 PIDs de los sensores usando kill().
5. Al recibir SIGUSR1, cada sensor despierta, genera un dato, lo escribe en su FIFO y vuelve a esperar con pause().
6. El central usa poll() (con un *timeout*) para recolectar las respuestas y actualizar el archivo.

### **4.2 Mecanismo de Señales**

**En los sensores:** Se usa sigaction para registrar un manejador de señal que modifica una variable volatile sig\_atomic\_t.

static volatile sig\_atomic\_t got\_usr1 = 0;  
  
static void h\_usr1(int s){   
 (void)s; // Evita warning de 'unused parameter'  
 got\_usr1 = 1;   
}  
  
// En main():  
struct sigaction sa = {.sa\_handler=h\_usr1};  
sigaction(SIGUSR1, &sa, NULL);  
  
while(running){  
 pause(); // Espera señal  
 if(got\_usr1){  
 got\_usr1 = 0;  
 // Generar y enviar dato...  
 }  
}

**En el central:** Se usa kill() para enviar la señal a los PIDs leídos.

// Al inicio, se leen los pids  
pid\_t pids[4] = { read\_pidfile(...) };  
  
// Dentro del bucle principal:  
while(1) {  
 for(int i=0; i<4; i++){  
 if(pids[i] > 0)   
 kill(pids[i], SIGUSR1);  
 }  
 // ... lógica de poll() con timeout para esperar respuestas ...  
 sleep(5); // Espera 5 segundos para el siguiente ciclo  
}

### **4.3 Compilación y Ejecución**

cd ParteC\_pipes\_senales  
make  
./run.sh

### **4.4 Resultados**

[INSERTAR CAPTURA: Terminales de los sensores mostrando su PID y el mensaje "esperando SIGUSR1..."]

[INSERTAR CAPTURA: Salida del 'central\_signal' mostrando el envío de señales y la recepción de datos]

[INSERTAR CAPTURA: Archivo clima\_actual.txt con el cabezal indicando "central con FIFOs + SIGUSR1"]

**Análisis:** Este modelo es más eficiente energéticamente ya que los sensores (procesos) solo realizan trabajo (CPU) cuando se les solicita explícitamente. Simula sistemas de control o embebidos donde un controlador central coordina la adquisición de datos de múltiples dispositivos. Las señales UNIX permiten una comunicación simple y una sincronización efectiva entre procesos.

## **5. COMPARACIÓN DE ENFOQUES**

| **Aspecto** | **Parte A (Hilos+Mutex)** | **Parte B (FIFOs)** | **Parte C (FIFOs+Señales)** |
| --- | --- | --- | --- |
| **Complejidad** | Baja (un solo programa) | Media (múltiples ejecutables, FIFOs) | Alta (Señales, PID files, FIFOs) |
| **Independencia** | Baja (mismo espacio de memoria) | Alta (procesos separados) | Alta (procesos separados) |
| **Sincronización** | Mutex (bloqueante) | Asíncrona (basada en datos) | Sincronizada por señales |
| **Overhead** | Bajo (creación de hilos) | Medio (context switch entre procesos) | Medio-Alto (Señales + context switch) |
| **Escalabilidad** | Limitada (por el proceso) | Buena | Buena |
| **Uso Ideal** | Aplicaciones monolíticas | Sistemas distribuidos simples | Control por eventos, sistemas de baja potencia |

## **6. CONCLUSIONES**

1. **Mutex:** Es la solución más simple y eficiente para hilos que comparten un mismo espacio de memoria. Es ideal cuando los "sensores" son simplemente módulos lógicos dentro de una aplicación única.
2. **FIFOs:** Permiten un desacoplamiento total entre procesos. Cada sensor puede ser desarrollado, compilado, reiniciado y ejecutado independientemente del central. El sistema es más robusto ante fallos individuales.
3. **Señales:** Añaden una capa de coordinación explícita. El central controla *cuándo* se adquieren los datos, lo cual es fundamental para sincronizar mediciones o para implementar sistemas de bajo consumo donde los sensores "duermen" hasta ser requeridos.
4. **Sincronización correcta:** En todos los casos se evitaron condiciones de carrera mediante los mecanismos apropiados (mutex para memoria compartida, poll para FIFOs asíncronos, y señales con variables sig\_atomic\_t para coordinación).
5. **Escritura Atómica:** El uso de rename() en todos los casos garantiza que el archivo clima\_actual.txt nunca queda en un estado intermedio o corrupto, lo cual es vital si otros procesos dependen de leer este archivo.

## **7. EVIDENCIAS DE EJECUCIÓN**

### **7.1 Estructura de archivos**

[INSERTAR CAPTURA: Árbol de directorios del proyecto (salida de tree)]

### **7.2 Compilación exitosa**

[INSERTAR CAPTURA: Salida de make en las 3 carpetas sin errores]

### **7.3 Pruebas de concurrencia**

[INSERTAR CAPTURA: Comando ps aux | grep sensor mostrando múltiples procesos sensor en Parte B/C]

[INSERTAR CAPTURA: tail -f clima\_actual.txt mostrando actualizaciones en tiempo real de los 3 modos]

## **8. CÓDIGO FUENTE**

Los archivos fuente completos se incluyen en el archivo comprimido adjunto (t21-concurrencia.zip).

t21-concurrencia.zip  
├── ParteA\_mutex\_semaforos/  
│ ├── climate\_threads.c  
│ └── Makefile  
├── ParteB\_pipes/  
│ ├── central.c  
│ ├── sensor\_temp.c  
│ ├── sensor\_humedad.c  
│ ├── sensor\_viento.c  
│ ├── sensor\_lluvia.c  
│ ├── run.sh  
│ └── Makefile  
└── ParteC\_pipes\_senales/  
 ├── central\_signal.c  
 ├── sensor\_temp\_sig.c  
 ├── sensor\_humedad\_sig.c  
 ├── sensor\_viento\_sig.c  
 ├── sensor\_lluvia\_sig.c  
 ├── run.sh  
 └── Makefile

## **REFERENCIAS**

* Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.)
* Stevens, W. R., & Rago, S. A. (2013). *Advanced Programming in the UNIX Environment* (3rd ed.)
* Documentación de pthread: man pthread\_mutex\_lock
* Documentación de FIFOs: man mkfifo, man 7 fifo, man 2 poll
* Documentación de señales: man sigaction, man 7 signal, man 2 pause