

Sincronización de Procesos

Sincronización de Procesos

Entendamos primero que es un recurso compartido. Un recurso compartido es cualquier componente (de hardware o de software) al que dos o más procesos o hilos necesitan acceder, pero que no puede ser utilizado o modificado por todos al mismo tiempo sin causar conflictos, inconsistencias o bloqueos.

En los sistemas operativos y el desarrollo de software, los recursos compartidos se clasifican principalmente en dos grandes grupos:

Recursos Lógicos o de Software (En memoria y almacenamiento)

1. Variables y estructuras de datos en memoria compartida:

Ejemplo clásico: Una variable global que actúa como un contador de visitas, el saldo de una cuenta bancaria o una lista enlazada compartida entre varios hilos. Si dos hilos intentan insertar un nodo a la vez, pueden romper los punteros de memoria.

2. Archivos en disco y registros específicos:

Ejemplo: Un archivo de base de datos o un archivo de configuración. Si un proceso está actualizando registros de un archivo secuencial o modificando la tabla de asignación, el sistema operativo debe aplicar bloqueos (exclusión mutua) sobre el archivo completo o sobre registros individuales para evitar lecturas inconsistentes.

3. Búferes o colas de comunicación (Productor-Consumidor):

Ejemplo: Un búfer circular en memoria donde un proceso (productor) coloca paquetes de red y otro (consumidor) los procesa. La estructura del búfer y sus índices son recursos que requieren semáforos para no sobreescribirse ni leer datos inexistentes.

4. Tablas y estructuras del propio Sistema Operativo:

Ejemplo: Las tablas de descriptores de archivos, la tabla de procesos preparados (cola de listos) o las tablas de asignación de bloques libres en disco. El núcleo del SO protege estas estructuras con cerrojos atómicos

internos (como spinlocks o semáforos) cuando varios núcleos de CPU intentan manipularlas.

1. Recursos Físicos o de Hardware (Dispositivos periféricos)

1. Impresoras y plotters:

El ejemplo más intuitivo: Si dos usuarios mandan a imprimir un documento al mismo tiempo en la misma impresora, no pueden imprimirse intercalando una línea de cada uno porque las hojas quedarían inservibles. La impresora es un recurso exclusivo (se gestiona mediante exclusión mutua o a través de colas de spooling).

2. Puertos de comunicación y buses (Serial, USB, Sockets):

Ejemplo: Un puerto serie o un módem conectado a un sensor industrial. Solo un proceso puede emitir comandos hacia el dispositivo a la vez; de lo contrario, las tramas de bytes se mezclarían y el dispositivo recibiría comandos corruptos.

3. Unidades de almacenamiento secundario (Discos y controladoras de E/S):

Ejemplo: El cabezal de lectura/escritura de un disco duro mecánico o el canal de transferencia física hacia un bloque de datos. Aunque el sistema operativo atiende múltiples peticiones, a bajo nivel el acceso al controlador de disco se serializa para leer o escribir un bloque a la vez.

4. El propio procesador (CPU):

Ejemplo: En un sistema monoprocesador, el núcleo de la CPU es en sí mismo un recurso compartido. El planificador a corto plazo (PCP) regula el tiempo de uso asignándole a cada proceso un turno exclusivo (como el quantum en Round Robin).

La Condicion de Carrera

Una condición de carrera (o *race condition*) ocurre en un sistema cuando dos o más procesos o hilos intentan acceder y modificar un recurso compartido exactamente al mismo tiempo, y el resultado final del sistema depende del orden exacto (e impredecible) en el que el procesador decide ejecutar sus instrucciones.

Como no hay una sincronización adecuada, los procesos "compiten" o "corren" por leer y escribir los datos, lo que casi siempre provoca que la información se corrompa o se pierda.

El mejor ejemplo es el de una cuenta bancaria:

Ejemplo: El cajero automático y la cuenta compartida

Imagina que dos personas (Proceso A y Proceso B) tienen una cuenta bancaria conjunta que tiene un saldo exacto de \$100. Ambos van a cajeros automáticos distintos y, por pura casualidad, intentan retirar dinero exactamente en el mismo milisegundo. La persona A quiere retirar \$80 y la persona B quiere retirar \$50.

Si el sistema operativo o la base de datos no está protegido contra condiciones de carrera, esto es lo que ocurre internamente:

1. Proceso A (Persona A): Lee el saldo actual de la cuenta. El saldo es \$100.
2. Proceso B (Persona B): Lee el saldo actual de la cuenta antes de que el Proceso A termine su operación. El saldo sigue siendo \$100.
3. Proceso A: El sistema verifica que \$80 es menor que \$100. Aprueba la transacción, entrega el dinero, resta \$80 y guarda el nuevo saldo en la base de datos: \$20.
4. Proceso B: El sistema verifica el saldo que leyó en el paso 2 (\$100). Como \$50 es menor que \$100, aprueba la transacción, entrega el dinero, resta \$50 y guarda el nuevo saldo sobrescribiendo el dato anterior: \$50.

El resultado del error:

El banco les entregó \$130 en total (más dinero del que tenían), ¡y encima la cuenta terminó registrando que les sobran \$50! La falta de control permitió que los procesos se atropellaran entre sí, corrompiendo la integridad de los datos.

¿Cómo lo resuelven los Sistemas Operativos?

Para solucionar esto, los sistemas imponen mecanismos de sincronización (como los Semáforos y Monitores) para garantizar la *exclusión mutua*.

Llevando este concepto a la gestión de archivos, cuando un sistema operativo otorga acceso para actualizar un archivo a más de un usuario al mismo tiempo, debe hacer cumplir una disciplina estricta. Un método habitual para evitar condiciones de carrera es permitir a los usuarios bloquear el archivo entero mientras lo actualizan, o mejor aún, bloquear registros individuales de forma exclusiva.

De esta forma, si el Proceso A está leyendo o modificando el archivo (o el saldo), el archivo se bloquea y el Proceso B tiene que esperar forzosamente a que A termine y libere el recurso.

Entonces los semáforos, monitores y paso de mensajes buscan resolver el mismo problema (sincronizar procesos y evitar condiciones de carrera), pero lo hacen con enfoques distintos: uno es a bajo nivel controlando el acceso a la memoria compartida (Semáforos), otro es a alto nivel encapsulando esa memoria de forma segura (Monitores), y el último cambia el paradigma por completo al **no compartir memoria en absoluto**, logrando la sincronización mediante el envío y recepción de paquetes de datos (Paso de Mensajes).

Semáforos: El control por conteo

Los semáforos son herramientas de sincronización basadas en variables especiales que ayudan a controlar el acceso a recursos compartidos en un entorno de multiprogramación para evitar condiciones de carrera. Su característica principal es que funcionan utilizando operaciones atómicas (indivisibles por el procesador) conocidas como *wait* (esperar) y *signal* (señalizar).

Analogía : Un solo baño, nos sirve para entender el funcionamiento de un semáforo binario (o Mutex) para garantizar la exclusión mutua:

1. El recurso compartido: Es el baño (solo cabe una persona/proceso a la vez).
El semáforo: Es el pestillo o el cartel de la puerta con valor inicial $S = 1$ (Libre).

Cómo actúan *wait* y *signal*:

- + Llega una persona (Operación *wait*):
- + Mira el cartel: si dice Libre (1), entra, echa el cerrojo y cambia el cartel a Ocupado (0). Entra a su sección crítica.

Si llega otra persona mientras está ocupado y ve el cerrojo echado (0), ejecuta *wait*, no puede entrar y se queda parada afuera en la cola de espera (bloqueada).

2. La persona sale (Operación *signal*):

Termina de usar el baño, abre el cerrojo y pasa el cartel a Libre (1).
Esta acción avisa (señaliza) a la persona que estaba esperando afuera para que pueda entrar de inmediato.

¿Y si hubiera más de un baño? (Semáforo Contador)

Si en lugar de uno hubiera 3 baños idénticos (múltiples unidades del recurso):

- + El contador inicializa en 3.
- + Cada persona que entra ejecuta *wait* y resta 1 al contador de baños libres ($3 > 2 > 1 > 0$).
- + El cuarto que llegue verá el contador en 0 y tendrá que esperar en la fila.
- + Cada vez que alguien sale, ejecuta *signal* y suma 1, permitiendo el paso al siguiente de la cola.

Tipos de Semáforos:

1. Semáforo Binario (o Mutex): Su contador solo puede ser 0 o 1. Sirve para exclusión mutua (como un baño con una sola llave).
2. Semáforo Contador: Su contador puede ser un número entero mayor a 1, ideal para controlar recursos con múltiples copias (como el estacionamiento).

Práctica de Semáforos en C++20 (WSL / Windows):

En este ejemplo, solo tenemos un recurso

Nota: Requiere vincular la librería de hilos al compilar. Compilar con:

```
gcc semaforo.c -o semaforo -pthread
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h> // Para la función sleep()

// Declaramos un semáforo global
sem_t semaforo;

// Función que ejecutará cada hilo
void* acceder_recurso(void* arg) {
    // Obtenemos el ID del hilo
    int id = *((int*)arg);

    printf("Hilo %d intentando acceder al recurso...\n", id);

    // Operación WAIT (o P): Toma un "lugar" o se bloquea si el contador llega a 0
    sem_wait(&semaforo);

    // --- INICIO DE SECCIÓN CRÍTICA ---
    printf("-> Hilo %d ENTRO a la seccion critica.\n", id);
    sleep(2); // Simulamos trabajo usando el recurso durante 2 segundos
    printf("<- Hilo %d SALIO de la seccion critica.\n", id);
    // --- FIN DE SECCIÓN CRÍTICA ---
```

```

// Operación SIGNAL (o V): Libera el "lugar" sumando 1 al contador
sem_post(&semaforo);

return NULL;
}

int main() {
    pthread_t hilos[5];
    int ids[5] = {1, 2, 3, 4, 5};

    // Inicializamos el semáforo:
    // Primer parámetro: puntero al semáforo
    // Segundo parámetro: 0 significa que se comparte entre hilos del mismo proceso
    // Tercer parámetro: valor inicial (2 lugares disponibles)
    if (sem_init(&semaforo, 0, 2) != 0) {
        perror("Error al inicializar el semaforo");
        return 1;
    }

    printf("Iniciando 5 hilos, pero el semaforo solo permite 2 a la vez.\n\n");

    // Creamos los 5 hilos
    for (int i = 0; i < 5; i++) {
        pthread_create(&hilos[i], NULL, acceder_recurso, &ids[i]);
    }

    // El hilo principal (main) espera a que todos los hilos terminen (join)
    for (int i = 0; i < 5; i++) {
        pthread_join(hilos[i], NULL);
    }

    // Destruimos el semáforo para liberar la memoria del sistema operativo
    sem_destroy(&semaforo);

    printf("\nTodos los hilos han terminado. Programa finalizado.\n");
    return 0;
}

```

Explicacion del codigo

- Usamos `sem_init()` para decirle al sistema operativo que cree un semáforo contador con valor 2.
- Al ejecutar `./semaforo`, verán que los 5 hilos imprimen el mensaje de "intentando acceder" casi al mismo tiempo.
- Sin embargo, el sistema operativo (gracias a `sem_wait`) bloqueará en la puerta a 3 hilos, dejando pasar únicamente a los 2 primeros.
- Hasta que uno de los dos hilos termine su trabajo y llame a `sem_post()` (liberando el recurso), el sistema operativo no despertará al siguiente de la cola para que entre.

Práctica de Semáforos en C++20 con Deadlock(WSL):

Para llegar a una situación de interbloqueo o deadlock a partir del ejemplo de los semáforos, el error clave consistiría en programar múltiples procesos que compitan por dos o más recursos distintos, y que se bloqueen esperando indefinidamente a que el otro suelte lo que tiene.

En el código anterior, solo tenemos un recurso (`semaforo_impresoras` o la `puerta_cabina` del monitor), por lo que un deadlock cruzado no es posible. Pero si modificáramos el diseño para requerir múltiples recursos, se podría provocar fácilmente.

Modificación para provocar un Deadlock:

Imagina que para realizar una tarea, un hilo necesita obtener primero el Recurso A y luego el Recurso B, pero otro hilo intenta obtener primero el Recurso B y luego el Recurso A.

Si introducimos esto en gcc, la modificación para provocar el deadlock sería así:

```
gcc semaforo2.c -o semaforo2 -pthread
```

```
#include <stdio.h>
```

```
#include <pthread.h>
```

```
#include <semaphore.h>
```

```
#include <unistd.h>
```

```
sem_t semaforo_A;
```

```

sem_t semaforo_B;

// El Hilo 1 toma A, luego intenta tomar B
void* proceso_1(void* arg) {
    printf("Hilo 1: Esperando por el semaforo A...\n");
    sem_wait(&semaforo_A);
    printf("Hilo 1: Obtuvo el semaforo A.\n");

    sleep(1); // Simulamos trabajo para asegurar que el Hilo 2 tome B mientras tanto

    printf("Hilo 1: Esperando por el semaforo B...\n");
    sem_wait(&semaforo_B); // <--- AQUI SE QUEDARA BLOQUEADO
    printf("Hilo 1: Obtuvo el semaforo B.\n");

    // ... no llegara aqui
    sem_post(&semaforo_B);
    sem_post(&semaforo_A);
    return NULL;
}

// El Hilo 2 toma B, luego intenta tomar A
void* proceso_2(void* arg) {
    printf("Hilo 2: Esperando por el semaforo B...\n");
    sem_wait(&semaforo_B);
    printf("Hilo 2: Obtuvo el semaforo B.\n");

    sleep(1); // Simulamos trabajo

    printf("Hilo 2: Esperando por el semaforo A...\n");
    sem_wait(&semaforo_A); // <--- AQUI SE QUEDARA BLOQUEADO
    printf("Hilo 2: Obtuvo el semaforo A.\n");

    // ... no llegara aqui
    sem_post(&semaforo_A);
    sem_post(&semaforo_B);
    return NULL;
}

```

```
int main() {  
    pthread_t h1, h2;  
  
    sem_init(&semaforo_A, 0, 1); // Mutex A  
    sem_init(&semaforo_B, 0, 1); // Mutex B  
  
    pthread_create(&h1, NULL, proceso_1, NULL);  
    pthread_create(&h2, NULL, proceso_2, NULL);  
  
    pthread_join(h1, NULL);  
    pthread_join(h2, NULL);  
  
    return 0;  
}
```

¿Qué ocurre en esta modificación?

- El Hilo 1 ejecuta `sem_wait (&semaforo_A)` y bloquea el Recurso A.
- Casi al mismo tiempo, el Hilo 2 ejecuta `sem_wait (&semaforo_B)` y bloquea el Recurso B.
- El Hilo 1 intenta hacer `sem_wait (&semaforo_B)`. Como el Hilo 2 lo tiene, el Hilo 1 se suspende y entra a la cola de espera.
- El Hilo 2 intenta hacer `sem_wait (&semaforo_A)`. Como el Hilo 1 lo tiene, el Hilo 2 se suspende y entra a la cola de espera.

El resultado es un Deadlock perfecto (abrazo mortal). Ninguno de los dos hilos puede avanzar porque cada uno está esperando a que el otro llame a `sem_post()`, y ninguno de los dos puede llamar a `sem_post()` porque ambos están durmiendo en la línea anterior. El programa se quedará congelado para siempre en la consola.

Monitores: La encapsulación segura

Trabajar con semáforos puede ser peligroso: si un programador olvida colocar un *signal()*, todos los demás procesos se quedarán bloqueados para siempre (Deadlock). Para solucionar los errores humanos, nacieron los Monitores.

Los monitores son estructuras de alto nivel que agrupan variables compartidas y los procedimientos que operan sobre ellas. Su característica más potente es que el propio sistema o lenguaje de programación se encarga de la seguridad, asegurando que solo un proceso pueda estar activo dentro del monitor al mismo tiempo.

Analogía:

Un Monitor es como una cabina blindada de un cajero automático.

- Encapsulamiento: El dinero (variables compartidas) está dentro y solo la máquina (procedimientos) puede tocarlo.
- Exclusión mutua automática: Solo una persona puede entrar a la cabina a la vez. La puerta se bloquea sola cuando alguien entra, y se desbloquea sola cuando sale, sin que la persona tenga que acordarse de echar llave.

Además, los monitores incluyen *Variables de Condición*, que permiten que un proceso "se duerma" dentro del monitor si le falta algo (ej. intentar retirar dinero de una cuenta vacía), abriendo la puerta para que otro proceso entre (ej. alguien que viene a depositar) y lo despierte después.

Ejemplo (Impresoras): Si el sistema tiene 3 impresoras compartidas, no creas 3 monitores independientes. Creas un solo monitor llamado *GestorImpresoras* que mantiene una cola de solicitudes y el estado de las 3 impresoras. Los procesos entran al monitor, piden una impresora libre, y el monitor se las asigna de forma ordenada y atómica.

Ejemplo (Búfer / Productor-Consumidor): En un búfer circular con 50 espacios (recursos individuales para almacenar datos), no se crean 50 monitores. Hay un solo monitor *BufferCompartido* que controla los índices de lectura/escritura y las variables de condición (lleno, vacío).

Práctica de Monitores en C++ (WSL / Windows):

Hay que tener en cuenta un detalle importante: el lenguaje C (a diferencia de Java o C#) no tiene una palabra reservada como `monitor` o `synchronized` que haga el trabajo de forma automática.

Sin embargo, como nos dice la teoría, los monitores son estructuras de alto nivel que agrupan variables compartidas y los procedimientos que operan sobre ellas, asegurando que solo un proceso pueda estar activo dentro del monitor al mismo tiempo. En C, construimos esta estructura manualmente utilizando una estructura (`struct`), un cerrojo de exclusión mutua (`pthread_mutex_t`) y una variable de condición (`pthread_cond_t`). Aquí está el código de la práctica del Cajero Automático escrito en C. para compilar con gcc.

monitor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h> // Para sleep()

// 1. Estructura que simula el Monitor
typedef struct {
    int saldo;
    pthread_mutex_t puerta_cabina; // Garantiza la exclusión mutua
    pthread_cond_t condicion_saldo; // Variable de condición para hacer esperar a los hilos
} MonitorCuenta;

MonitorCuenta cajero; // Instancia global de nuestro monitor

// Inicializar las variables del monitor
void inicializar_monitor(MonitorCuenta *m) {
    m->saldo = 0;
    pthread_mutex_init(&m->puerta_cabina, NULL);
    pthread_cond_init(&m->condicion_saldo, NULL);
}

// 2. Procedimiento del Monitor: Retirar dinero
void retirar(MonitorCuenta *m, int cantidad, int id) {
```

```

// Al entrar al procedimiento, el hilo cierra la puerta (LOCK)
pthread_mutex_lock(&m->puerta_cabina);

printf("Hilo %d quiere retirar $%d...\n", id, cantidad);

// Si no hay saldo suficiente, el hilo se "duerme" y la puerta se ABRE automáticamente
while (m->saldo < cantidad) {
    printf("Hilo %d en ESPERA (saldo insuficiente).\n", id);
    // Esta función bloquea el hilo y libera el mutex al mismo tiempo
    pthread_cond_wait(&m->condicion_saldo, &m->puerta_cabina);
}

// Si despertó y salió del bucle, es porque ya hay saldo y volvió a cerrar la puerta
m->saldo -= cantidad;
printf("Hilo %d retiro $%d con exito. Saldo actual: $%d\n", id, cantidad, m->saldo);

// Al salir del procedimiento, abre la puerta para los demás (UNLOCK)
pthread_mutex_unlock(&m->puerta_cabina);
}

// 3. Procedimiento del Monitor: Depositar dinero
void depositar(MonitorCuenta *m, int cantidad, int id) {
    // Al entrar, cierra la puerta
    pthread_mutex_lock(&m->puerta_cabina);

    m->saldo += cantidad;
    printf("Hilo %d deposito $%d. Saldo actual: $%d\n", id, cantidad, m->saldo);

    // Notifica (despierta) a todos los hilos que estén esperando por saldo
    pthread_cond_broadcast(&m->condicion_saldo);

    // Abre la puerta
    pthread_mutex_unlock(&m->puerta_cabina);
}

// --- HILOS (Procesos) ---

void* proceso_retirar(void* arg) {

```

```
int id = *((int*)arg);
retirar(&cajero, 100, id); // Intenta retirar $100
return NULL;
}
```

```
void* proceso_depositar(void* arg) {
    int id = *((int*)arg);
    depositar(&cajero, 150, id); // Deposita $150
    return NULL;
}
```

```
int main() {
    pthread_t h1, h2;
    int id1 = 1, id2 = 2;

    printf("=====\n");
    printf(" Iniciando simulador de Monitor de Cuenta \n");
    printf("=====\n\n");

    inicializar_monitor(&cajero);

    // El Hilo 1 llega primero e intenta retirar dinero, pero la cuenta está en $0
    pthread_create(&h1, NULL, proceso_retirar, &id1);

    sleep(2); // Pausa de 2 segundos para asegurar que el Hilo 1 se quede bloqueado
esperando

    // El Hilo 2 llega más tarde y deposita el dinero
    pthread_create(&h2, NULL, proceso_depositar, &id2);

    // Esperar a que terminen
    pthread_join(h1, NULL);
    pthread_join(h2, NULL);

    // Destruir variables del monitor para liberar memoria
    pthread_mutex_destroy(&cajero.puerta_cabina);
    pthread_cond_destroy(&cajero.condicion_saldo);
}
```

```
printf("\nTodas las transacciones finalizaron.\n");

return 0;
}
```

Compilarlo :

```
gcc monitor.c -o monitor -pthread
```

```
./monitor
```

Explicacion:

- Al ejecutarlo, verán que el Hilo 1 entra al monitor y pide retirar \$100. Como el saldo es 0, entra en la línea `pthread_cond_wait`.
- Esta función es la magia del Monitor en C: congela al Hilo 1 y abre el mutex (la puerta) al mismo tiempo de forma atómica.
- Gracias a que la puerta se abrió, el Hilo 2 puede entrar 2 segundos después al procedimiento depositar. Suma \$150 y lanza un `pthread_cond_broadcast` (el equivalente a un `notify` o un `signal` masivo).
- El Hilo 1 despierta, vuelve a tomar la puerta (el mutex) automáticamente, resta sus \$100 y termina exitosamente sin que los datos se hayan corrompido.

Paso de Mensajes (Message Passing)

Mientras que los semáforos y monitores se basan en que los procesos compartan la misma memoria para sincronizarse, el Paso de Mensajes asume que los procesos no comparten memoria. En su lugar, se comunican enviando y recibiendo paquetes de datos a través de canales gestionados por el Sistema Operativo (o por la red). Es el mecanismo fundamental para los Sistemas Distribuidos y las arquitecturas modernas de Microservicios.

Operaciones Básicas:

Un sistema de paso de mensajes proporciona al menos dos operaciones primitivas:

- `send(destino, mensaje)`: Envía un mensaje a un proceso o buzón específico.
- `receive(origen, mensaje)`: Espera y recibe un mensaje.

Analogía:

El Paso de Mensajes es como el sistema postal o un chat de WhatsApp:

- Sin Memoria Compartida: Tú (Proceso A) y tu amigo (Proceso B) no viven en la misma casa (no comparten RAM).
- Sincronización: Si le envías una carta preguntándole algo importante (`send`), tú te quedas esperando una respuesta (`receive`). Tu amigo procesa la información en su casa y te devuelve otra carta.
- Seguridad Integrada: Como no están compartiendo un cuaderno físico (memoria) al mismo tiempo, es imposible que ambos intenten escribir en la misma línea a la vez, eliminando las condiciones de carrera de forma natural.

Tipos de Sincronización en Mensajes:

- Bloqueante (Síncrona): Como una llamada telefónica. El que envía el mensaje se queda "congelado" (bloqueado) hasta que el otro lo recibe y contesta.
- No Bloqueante (Asíncrona): Como un correo electrónico. El que envía el mensaje lo deja en el buzón y continúa haciendo otras tareas sin esperar la respuesta inmediata.

Ejemplo a Nivel de Sistema Operativo: Tuberías (Pipes) en la Terminal

En sistemas tipo Unix/Linux, cuando encadenas dos comandos en la terminal con una barra vertical `|`, estás usando paso de mensajes a través de un canal unidireccional (pipe):

Bash

```
cat datos.txt | grep "ERROR"
```

1. Proceso 1 (cat): Lee el archivo y escribe (envía) el flujo de bytes hacia el extremo de escritura de la tubería.
2. Canal gestionado por el SO: El núcleo del sistema operativo retiene los bytes en un búfer intermedio.
3. Proceso 2 (grep): Lee (recibe) los bytes desde el extremo de lectura de la tubería para filtrar las líneas que contienen "ERROR".
4. Sincronización: Si cat aún no produce datos, grep se bloquea automáticamente esperando la llegada del mensaje; si la tubería se llena, cat se detiene hasta que grep lea datos.

Práctica: Paso de Mensajes en Linux (Pipes en C++)

Guarda este código en un archivo llamado `paso_mensajes.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
```

```
int main() {
    // Un pipe necesita un arreglo de 2 enteros para los descriptores de archivo:
    // tuberia[0] será usado para LEER (Receive)
    // tuberia[1] será usado para ESCRIBIR (Send)
    int tuberia[2];

    char mensaje_enviado[] = "¡Hola Hijo! Este es un mensaje secreto desde el Proceso Padre.";
    char mensaje_recibido[100]; // Búfer para guardar lo que llegue

    printf("=====\n");
    printf(" Iniciando simulador de Paso de Mensajes \n");
    printf("=====\n\n");
```

```

// 1. Creamos el canal de comunicación (La tubería)
if (pipe(tuberia) == -1) {
    perror("Error al crear la tubería");
    return 1;
}

// 2. Creamos un nuevo proceso usando fork()
pid_t pid = fork();

if (pid < 0) {
    perror("Error al crear el proceso hijo");
    return 1;
}

if (pid > 0) {
    // =====
    // CÓDIGO DEL PROCESO PADRE (El Emisor)
    // =====

    // El Padre solo va a escribir, así que cerramos el extremo de lectura
    close(tuberia[0]);

    printf("[PADRE] Preparando el mensaje...\n");
    sleep(1); // Simulamos un pequeño retardo

    printf("[PADRE] Enviando mensaje por la tubería (SEND).\n");

    // Operación SEND: Escribimos en tuberia[1]
    // Enviamos la cadena + 1 para incluir el carácter nulo de terminación '\0'
    write(tuberia[1], mensaje_enviado, strlen(mensaje_enviado) + 1);

    // Cerramos el canal de escritura para indicar que ya no enviaremos más datos
    close(tuberia[1]);

    // El Padre espera a que el Hijo termine su ejecución
    wait(NULL);
}

```

```

else {
    // =====
    // CÓDIGO DEL PROCESO HIJO (El Receptor)
    // =====

    // El Hijo solo va a leer, así que cerramos el extremo de escritura
    close(tuberia[1]);

    printf(" [HIJO] Esperando recibir un mensaje (RECEIVE)...\n");

    // Operación RECEIVE: Leemos de tuberia[0]
    // Esta función bloquea (congela) al proceso Hijo hasta que lleguen datos por el tubo
    read(tuberia[0], mensaje_recibido, sizeof(mensaje_recibido));

    printf(" [HIJO] ¡Mensaje recibido exitosamente!\n");
    printf(" [HIJO] Contenido: \"%s\"\n\n", mensaje_recibido);

    // Cerramos el canal de lectura
    close(tuberia[0]);
}

return 0;
}

```

Compilarlo :

```
gcc paso_mensajes.c -o paso_mensajes
```

```
./paso_mensajes
```

Explicación código:

- Al ejecutarlo, verán que la computadora crea dos programas que corren al mismo tiempo.
- El Proceso Hijo llega a la línea `read()` y se queda "congelado" esperando.
- El Proceso Padre hace una pausa de 1 segundo (con `sleep(1)`) y luego inyecta el texto en la tubería usando `write()`.
- En el momento exacto en que los datos entran a la tubería, el Sistema Operativo despierta al Hijo, quien recoge el texto en su propia variable y lo imprime en

pantalla.

La gran lección: Demuestra que los procesos nunca compartieron variables. La variable `mensaje_enviado` pertenece solo a la memoria del Padre, y el Hijo la recibió copiando los datos que viajaron por la "tubería" proporcionada por el Sistema Operativo.

El paso de mensajes no solo no está obsoleto, sino que es el estándar actual para que sistemas completos y aplicaciones en la nube conversen entre sí sin corromper datos. Los pipes de Unix son su versión más pura y local.

2. Comunicación Interprocesos (IPC)

La Comunicación Interprocesos (IPC) engloba los métodos y mecanismos que permiten a los procesos comunicarse y sincronizar sus acciones. Puede implementarse de diferentes maneras, como el uso de memoria compartida (donde varios procesos pueden leer y escribir en el mismo espacio) o el paso de mensajes (mediado por el sistema operativo).

3. Problemas Típicos de Sistemas Operativos

- **Cena de los Filósofos:** Un problema clásico que ilustra el reto de asignar múltiples recursos a un grupo de procesos sin caer en bloqueos mutuos (*deadlocks*) ni inanición.
- **Lectores y Escritores:** Modela una situación donde varios procesos desean leer un archivo o estructura compartida, pero otros quieren modificarlo. Permite lecturas concurrentes, pero exige exclusión mutua para la escritura.

- **Productor y Consumidor:** Plantea el escenario de uno o varios procesos produciendo datos y otros consumiéndolos sobre un búfer de tamaño finito. Asegura que el productor no escriba si el búfer está lleno y que el consumidor no extraiga si está vacío.

4. Planificación en Monoprocesadores

El planificador a corto plazo (PCP) es el encargado de decidir qué proceso, de los que ya están preparados y cargados en memoria, toma el control de la CPU en un momento dado.

Entre los algoritmos de planificación destacan:

- **FCFS (First Come First Served):** Los procesos son servidos en orden a su llegada a la cola de listos y se ejecutarán hasta que terminen o se bloqueen. Presenta el problema del "efecto convoy", donde los procesos pequeños esperan a que termine uno muy largo.
- **SJF (Shortest Job First):** La CPU se asigna al proceso al que le queda menos tiempo para completar su ejecución.
- **Prioridad:** A cada proceso se le asigna un determinado valor de prioridad; la CPU se asigna al de mayor prioridad y, a igualdad, se sigue el algoritmo FCFS.
- **Round Robin:** La CPU se asigna a los procesos en intervalos de tiempo fijos llamados "quantum". Es un algoritmo siempre expulsivo (apropiativo) por su propia naturaleza; si el proceso agota su quantum de tiempo y no ha terminado, el sistema lo interrumpe y lo envía al final de la cola de listos.

5. Ejemplos Prácticos de Identificación y Prácticas en C++

Identificación de Procesos en Windows

En Windows no verás la CPU libre como un espacio vacío, sino como un proceso real que "consume" lo que sobra.

- Abre el Administrador de tareas presionando **Ctrl + Shift + Esc**.
- Ve a la pestaña **Detalles**.
- Busca el proceso llamado **Proceso inactivo del sistema** (o *System Idle Process* en inglés). Si notas que este proceso está en un 95% o 99%, es una excelente señal; significa que tu computadora está libre, descansando y tiene el 99% de su capacidad disponible para cuando la necesites.

Cómo medir el tiempo de uso acumulado:

- En la misma pestaña **Detalles**, haz clic derecho sobre los títulos de las columnas (donde dice Nombre, PID, Estado...) y elige **Seleccionar columnas**.
- Busca y marca la casilla que dice **Tiempo de CPU** y dale a Aceptar.
- Aparecerá una nueva columna con un formato de reloj (HH:MM:SS) donde podrás ver el uso acumulado.

Identificación de Procesos en Linux

En Linux, el estado *idle* (ocioso) se mide directamente como un porcentaje de las estadísticas de la CPU:

- **Con los comandos** `top` o `htop`: En la parte superior verás una línea de uso de CPU. Busca el valor que dice `id` (abreviatura de *idle*). Si dice `98.0 id`, significa que el 98% de la CPU está inactiva.
- **Con el comando** `vmstat`: En la última columna, bajo las siglas `id`, también verás el porcentaje de tiempo que el procesador ha estado ocioso.

Cómo medir el tiempo de uso acumulado:

- **En** `top`: Busca la columna llamada `TIME+`. Muestra el tiempo total de CPU que el proceso ha utilizado desde que inició (Minutos:Segundos.Centésimas).
- **Con** `ps`: Ejecuta el comando `ps -eo pid,comm,time,%cpu --sort=-time` para que te muestre el ID del proceso, el nombre, el tiempo acumulado de CPU y el porcentaje actual, ordenados de mayor a menor tiempo.

Prácticas con C++ en WSL (Ubuntu)

Para llevar a la práctica la creación de procesos y observar la concurrencia, usaremos la terminal de WSL con Ubuntu.

Asegúrate de tener el compilador instalado ejecutando:

```
sudo apt update && sudo apt install build-essential
```

Práctica 1: Creación de un proceso con `fork()`

Este ejemplo ilustra cómo un proceso (padre) crea un proceso idéntico (hijo).

1. **Crea el archivo:** `nano fork_ejemplo.cpp`
2. **Pega el siguiente código:**

```
C++
#include <iostream>
#include <unistd.h>
```

```

int main() {
    // La función fork() crea un nuevo proceso
    pid_t pid = fork();

    if (pid < 0) {
        std::cerr << "Fallo al crear el proceso hijo." << std::endl;
        return 1;
    } else if (pid == 0) {
        // Entra aquí si es el proceso hijo
        std::cout << "Soy el proceso HIJO. Mi PID es: " << getpid() << std::endl;
    } else {
        // Entra aquí si es el proceso padre
        std::cout << "Soy el proceso PADRE. Mi PID es: " << getpid() << " y acabo de crear
al hijo con PID: " << pid << std::endl;
    }

    return 0;
}

```

3. Guarda y compila: `g++ fork_ejemplo.cpp -o fork_ejemplo`

4. Ejecuta: `./fork_ejemplo`

Práctica 2: Creación de hilos (Threads) para simular tareas concurrentes

En lugar de procesos pesados, los hilos comparten el mismo espacio de memoria, algo vital en la sincronización.

1. Crea el archivo: `nano hilos_ejemplo.cpp`

2. Pega el siguiente código:

```

C++
#include <iostream>
#include <thread>
#include <vector>
#include <unistd.h> // Para la función sleep()

// Función que ejecutará cada hilo
void tarea_hilo(int id) {
    std::cout << "Hilo " << id << " iniciando su ejecución...\n";
    sleep(1); // Simulamos que el hilo está trabajando 1 segundo
}

```

```

        std::cout << "Hilo " << id << " ha terminado.\n";
    }

int main() {
    std::vector<std::thread> mis_hilos;
    int numero_de_hilos = 5;

    std::cout << "Creando " << numero_de_hilos << " hilos concurrentes...\n";

    // Bucle para crear los hilos
    for (int i = 1; i <= numero_de_hilos; ++i) {
        mis_hilos.push_back(std::thread(tarea_hilo, i));
    }

    // El proceso principal (main) debe esperar a que los hilos terminen usando .join()
    for (auto& hilo : mis_hilos) {
        hilo.join();
    }

    std::cout << "Todos los hilos han sido sincronizados y el programa finaliza.\n";
    return 0;
}

```

3. Guarda y compila (importante agregar `-pthread`):

```
g++ hilos_ejemplo.cpp -o hilos_ejemplo -pthread
```

4. Ejecuta: `./hilos_ejemplo`