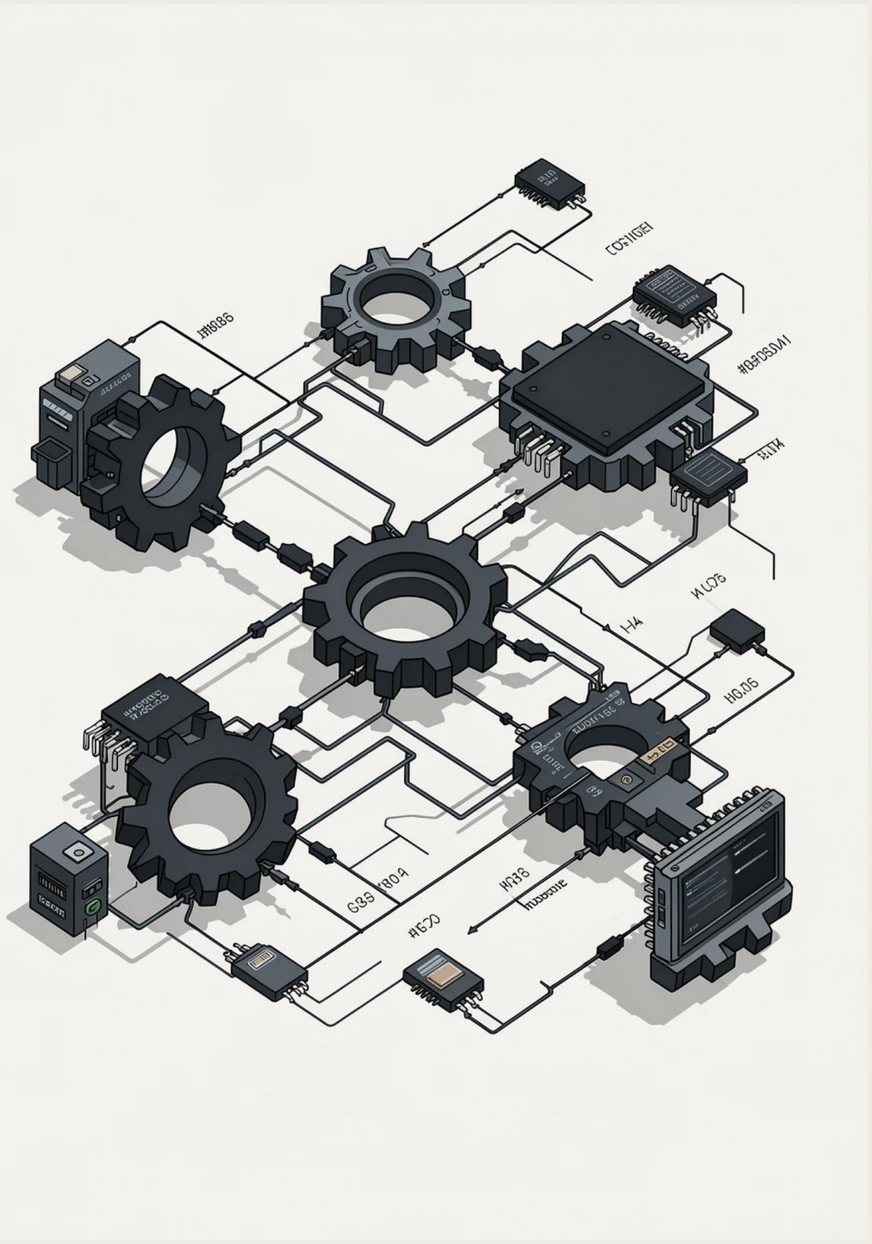


ING. EN INFORMÁTICA · LIC. EN SISTEMAS

Unidad 6: Sincronización de Procesos

6.1 Introducción · 6.2 El Problema de la Sección Crítica · 6.3 Hardware de Sincronización · 6.4. Semáforos. 6.5. Problemas clásicos de sincronización. 6.6. Regiones críticas. 6.7. Monitores. 6.8. Transacciones atómicas

INTRODUCCIÓN

¿Por qué Sincronizar?

El mundo de la concurrencia

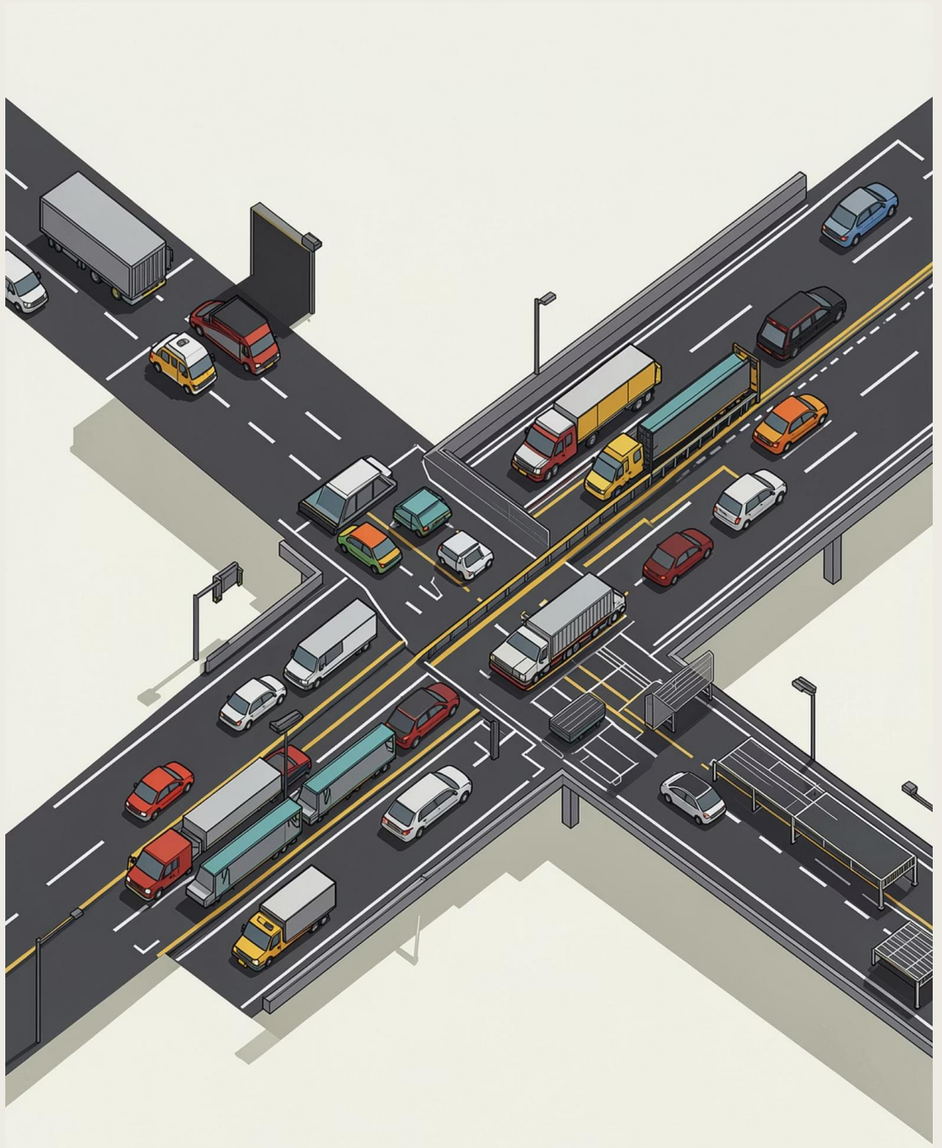
Múltiples procesos coexisten y compiten por recursos limitados: memoria, archivos, dispositivos de E/S.

Inconsistencia de datos

Sin coordinación, los procesos pueden corromper información compartida al acceder simultáneamente.

La necesidad de orden

Garantizar que las operaciones se ejecuten en el orden correcto es esencial para la corrección del sistema.



INTRODUCCIÓN

El Desafío de la Concurrencia

¿Qué sucede cuando múltiples procesos o hilos acceden a recursos compartidos simultáneamente?

El Riesgo

Inconsistencia de datos y resultados impredecibles cuando no hay control.

La Necesidad

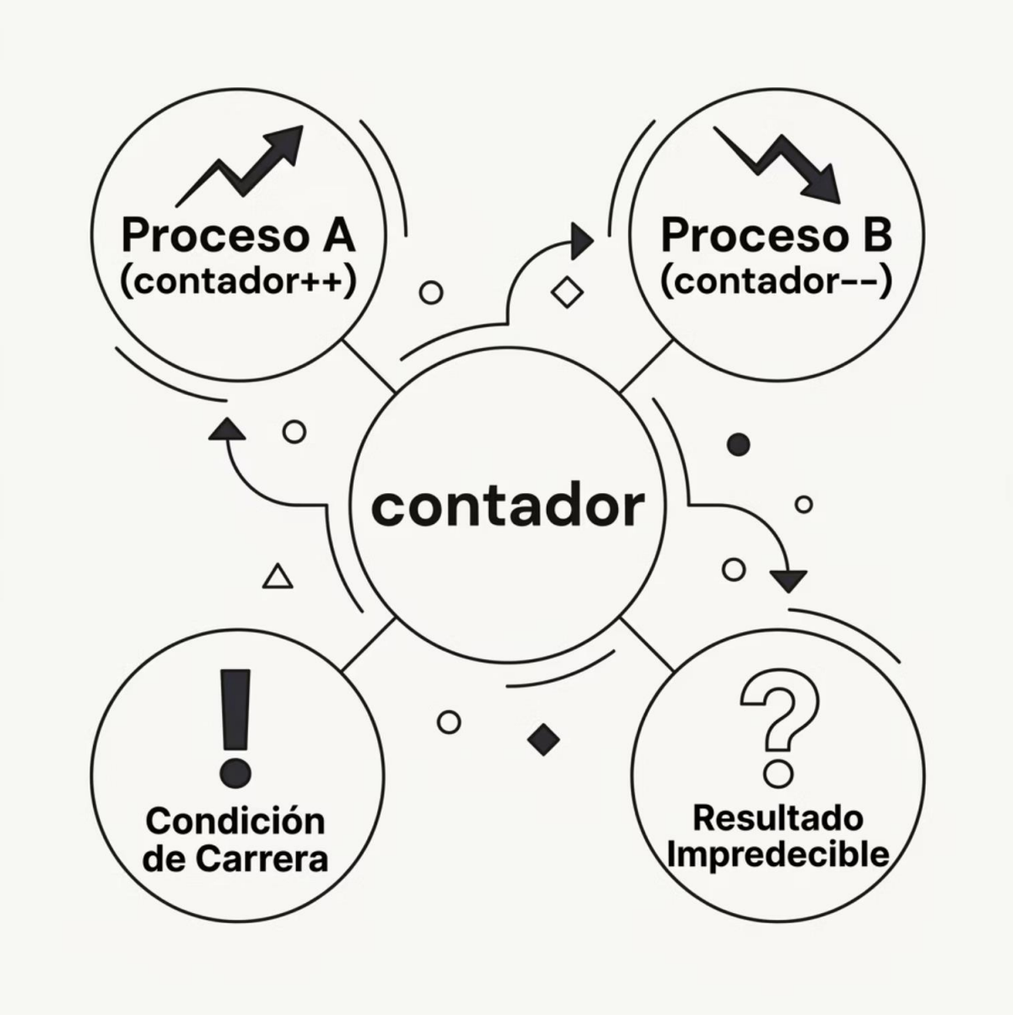
Mecanismos que garanticen ejecución ordenada y segura.

La Analogía

Un cruce sin semáforos: ¡caos total! Los procesos necesitan sus propias señales.

INTRODUCCIÓN

El Desafío de la Concurrencia



El problema en acción

Dos procesos acceden a la misma variable `contador`:

- **Proceso A:** incrementa el contador
- **Proceso B:** decrementa el contador

Si se ejecutan de forma intercalada, el resultado final puede ser **impredecible y erróneo**, incluso si cada operación individual es correcta.

> ⚠ Sin sincronización, el valor final de `contador` no refleja la lógica del programa.

SECCIÓN/REGIÓN CRÍTICA

El Problema de la Sección Crítica

Una **sección crítica** es la parte del código que accede a recursos compartidos. Debe cumplir tres condiciones fundamentales:

1

Exclusión Mutua

Solo un proceso puede estar en su sección crítica a la vez.

2

Progreso

Si nadie está en su sección crítica y otros quieren entrar, la decisión de quién entra no puede posponerse indefinidamente.

3

Espera Limitada

Debe existir un límite en la cantidad de veces que otros procesos pueden entrar antes de que un proceso solicitante sea atendido.

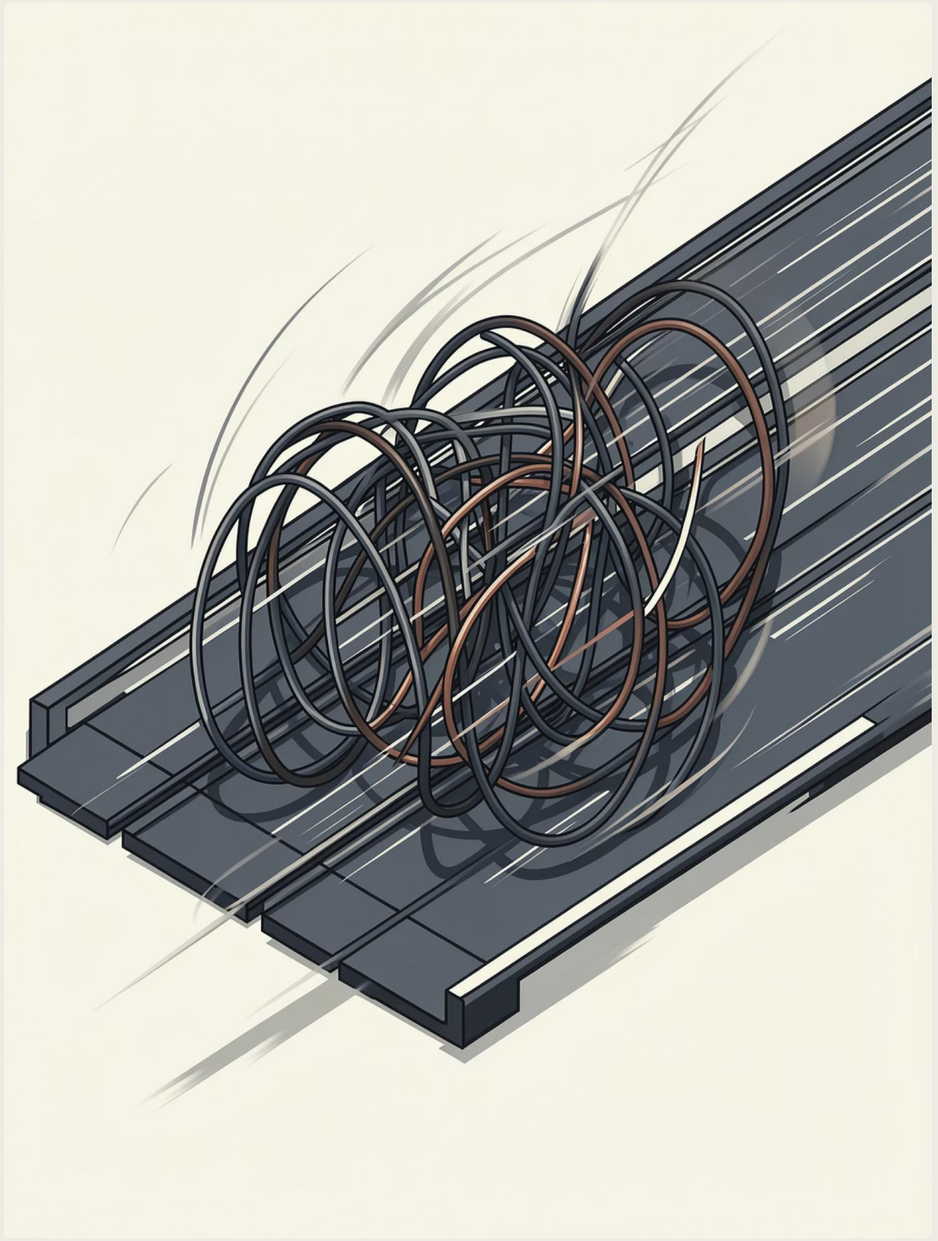
SECCIÓN/REGIÓN CRÍTICA

Regiones Críticas: El Corazón del Problema

¿Qué es una región crítica?

Segmento de código donde se accede a **recursos compartidos**. Su ejecución debe ser **atómica** e indivisible desde la perspectiva de otros procesos.

⚠ Si dos procesos ejecutan la región crítica simultáneamente, los datos se corrompen.



SECCIÓN/REGIÓN CRÍTICA

La Condición de Carrera

Una **condición de carrera** ocurre cuando el resultado de la ejecución depende del orden impredecible en que los procesos se intercalan.

Código vulnerable

`tareasPendientes++` y `tareasPendientes--` ejecutados concurrentemente pueden producir valores incorrectos.

¿Por qué ocurre?

Las operaciones de incremento/decremento no son atómicas: implican lectura, modificación y escritura como pasos separados.

Consecuencia

El sistema puede comportarse de forma errática, generando bugs difíciles de reproducir y depurar.

SECCIÓN/REGIÓN CRÍTICA

Regiones Críticas: El Corazón del Problema

Mecanismos de protección

Solución por Hardware

Uso de instrucciones atómicas especiales del procesador que bloquean el acceso al bus de memoria para evitar condiciones de carrera a nivel físico

Solución por Software

Uso de algoritmos lógicos y estructuras de control que gestionan el turno de acceso a la sección crítica sin requerir soporte específico del procesador

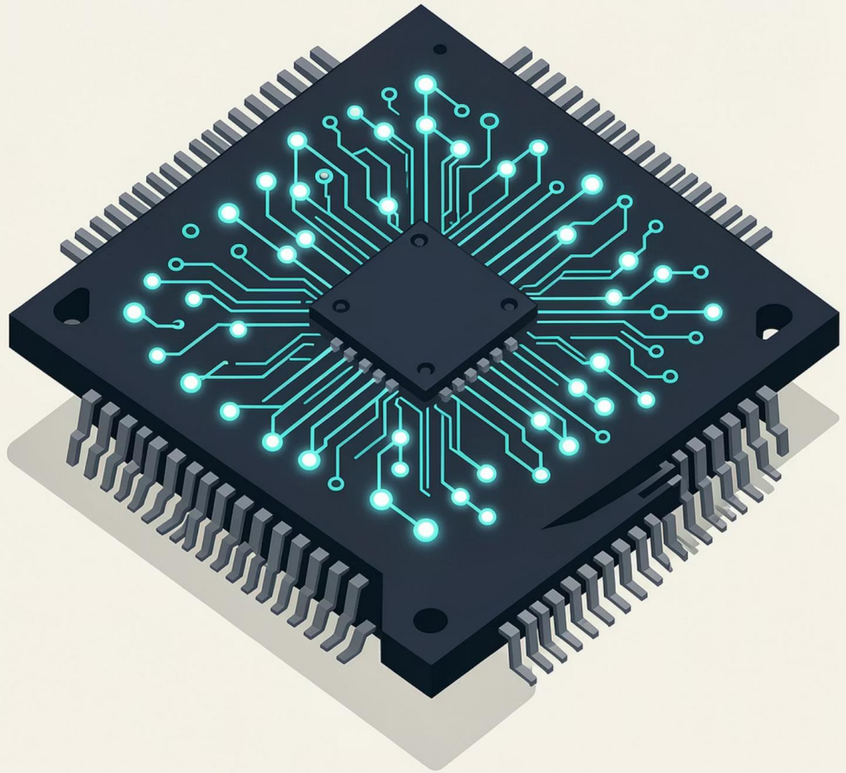
SECCIÓN/REGIÓN CRÍTICA

Regiones Críticas: Solución por Hardware

Instrucciones especiales de máquina

- Comparar y Fijar
- Intercambiar

Inhabilitación de Interrupciones



HARDWARE DE SINCRONIZACIÓN

La Solución Robusta: Hardware

El hardware provee **instrucciones atómicas** que se ejecutan como una unidad indivisible, sin posibilidad de interrupción.

Instrucciones Atómicas

Operaciones que el procesador ejecuta sin interrupción. Son la base de todos los mecanismos de sincronización modernos.

Test-and-Set

Lee el valor de una variable y la establece a `true` en una sola operación indivisible. Retorna el valor original.

Swap

Intercambia el contenido de dos variables de forma atómica, útil para implementar locks y semáforos.

HARDWARE DE SINCRONIZACIÓN

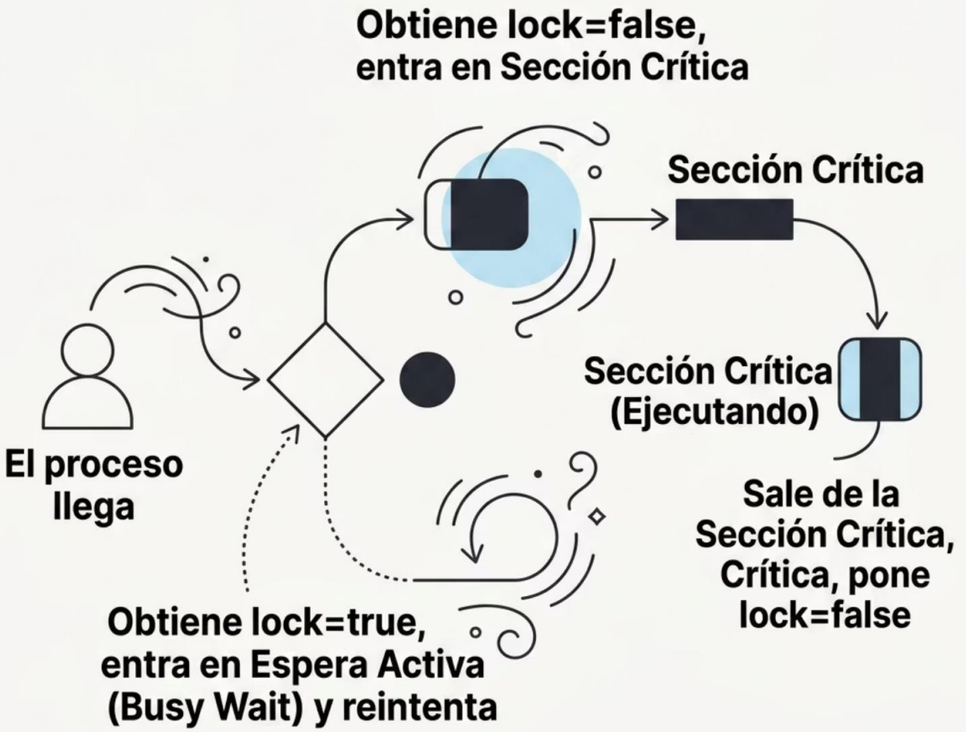
Implementando Exclusión Mutua

Ejemplo con Test-and-Set

```
lock = 0;

while (TS(lock))
    ; // Espera activa (busy wait)

// --- Sección Crítica ---
// Acceso seguro al recurso
// ----------------------

lock = 0; // Liberar lock
```

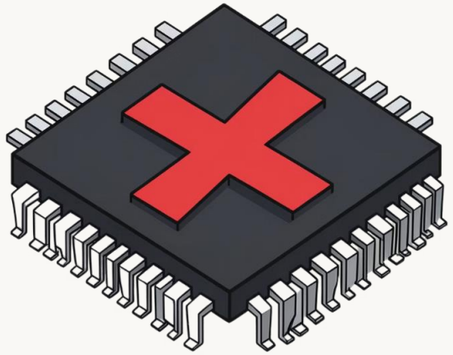
```
bool TS (int i) {
    if (i == 0) {
        i = 1;
        return true ;
        }
    else return false;
}
```

La variable `lock` tiene dos valores posibles. Solo un proceso supera el `while` a la vez.



HARDWARE DE SINCRONIZACIÓN

Deshabilitando Interrupciones



Un enfoque simple, pero limitado

En sistemas de **un solo procesador**, deshabilitar las interrupciones antes de entrar a la sección crítica garantiza exclusión mutua: el proceso no puede ser interrumpido.

✅ Válido en uniprocesador

Solución sencilla y efectiva en entornos de un solo núcleo.

❌ No escalable

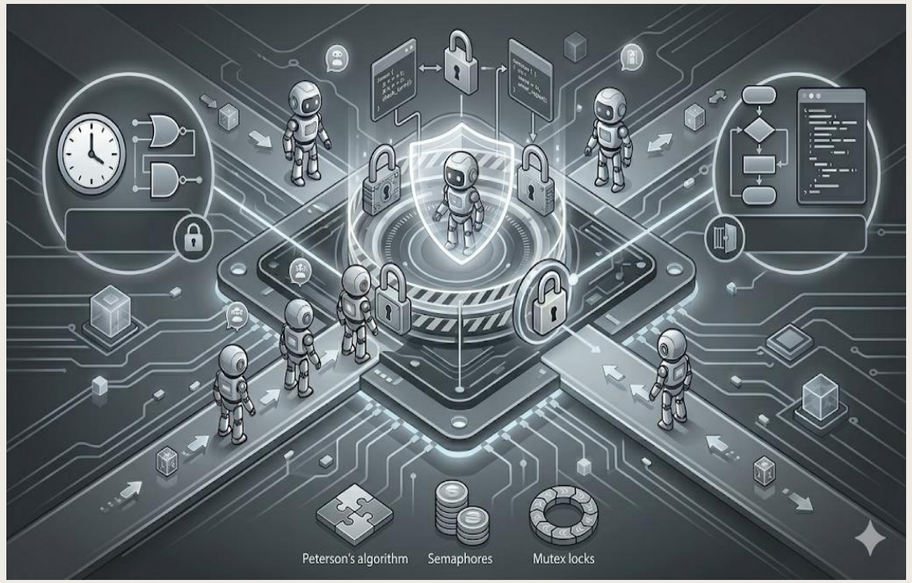
Ineficiente en sistemas multiprocesador: la latencia crece y la complejidad se dispara.

```
program exclusión_mutua;
const n=...; (*número de procesos*);
var cerrojo: entero;
procedure P (i: entero);
begin
    repeat
        repeat { nada }until TS (cerrojo);
            < sección crítica >;
            cerrojo := 0;
            < resto >
        forever
end;
begin (* programa principal *)
    cerrojo := 0;
    parbegin
        P(1);
        P(2);
        ...
        P(n);
    parend
end.
```

(a) Instrucción Comparar y Fijar

```
program exclusión_mutua;
const n=...; (*número de procesos*);
var cerrojo: entero;
procedure P (i: entero);
var clave_i: entero;
begin
    repeat
        clave_i := 1;
        repeat intercambiar (clave_i, cerrojo) until clave_i = 0;
        < sección crítica >;
        intercambiar (clave_i, cerrojo);
        < resto >
    forever
end;
begin (* programa principal *)
    cerrojo := 0;
    parbegin
        P(1);
        P(2);
        ...
        P(n);
    parend
end.
```

(b) Instrucción Intercambiar



SECCIÓN/REGIÓN CRÍTICA

Regiones Críticas: Mecanismos de Protección

Soluciones por Software

→ Semáforos

Control explícito con wait/signal.

→ Monitores

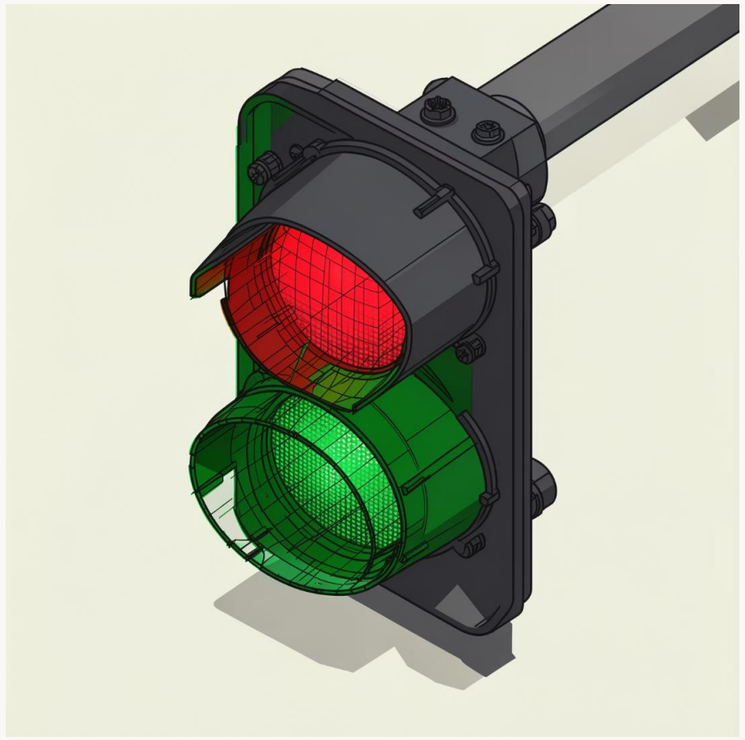
Exclusión mutua implícita.

→ Algoritmos clásicos

Decker, Peterson, Lamport

SOLUCIONES POR SOFTWARE

Semáforos: Los Guardianes de los Recursos



Variables especiales que controlan el acceso a recursos compartidos mediante tres operaciones fundamentales:

Tipos: Binarios (0 o 1) para exclusión mutua · Contadores (N accesos) para recursos múltiples.

init()

Inicializa el semáforo con el número de accesos permitidos.

wait() / P()

Solicita acceso. Si el recurso está ocupado, el proceso espera.

signal() / V()

Libera el recurso y notifica a otro proceso en espera.

TIPOS DE SEMÁFOROS

Binarios (mutex)

- Garantizan exclusión mutua a recurso
- Sólo una hebra/proceso puede accesar sección crítica a la vez
- Contador de semáforo inicializado en 1

Contadores

- Representan recursos con mas de una unidad disponible
- Permiten accesar recursos de acuerdo a cuántos estén disponibles
- Contador es inicializado en N, donde N es la cantidad de unidades disponibles del recurso. N>=0

IMPLEMENTACIÓN DE SEMÁFOROS BINARIOS

```
typedef struct {
        bool value;
        struct hebra *L;
} semaphoreBin;
```

```
void wait(semaphoreBin S) {
        if (S.value == false){
                agregar hebra a S.L;
                block();
                }
        else
            s.value=false;
}
```

```
void signal(semaphoreBin S){
        if (!vacia(S.L){
            remover hebra T de S.L;
            wakeup(T);
        }
        else
            s.value=true;
}
```

IMPLEMENTACIÓN DE SEMÁFOROS CONTADORES

```
typedef struct {
        int value;
        struct hebra *L;
} semaphore;
```

```
void wait(semaphore S) {
        S.value--;
        if (S.value < 0){
            agregar hebra a S.L;
            block();
        }
}
```

```
void signal(semaphore S){
        S.value++;
        if (S.value <=0){
            remover hebra T de S.L;
            wakeup(T);
        }
}
```

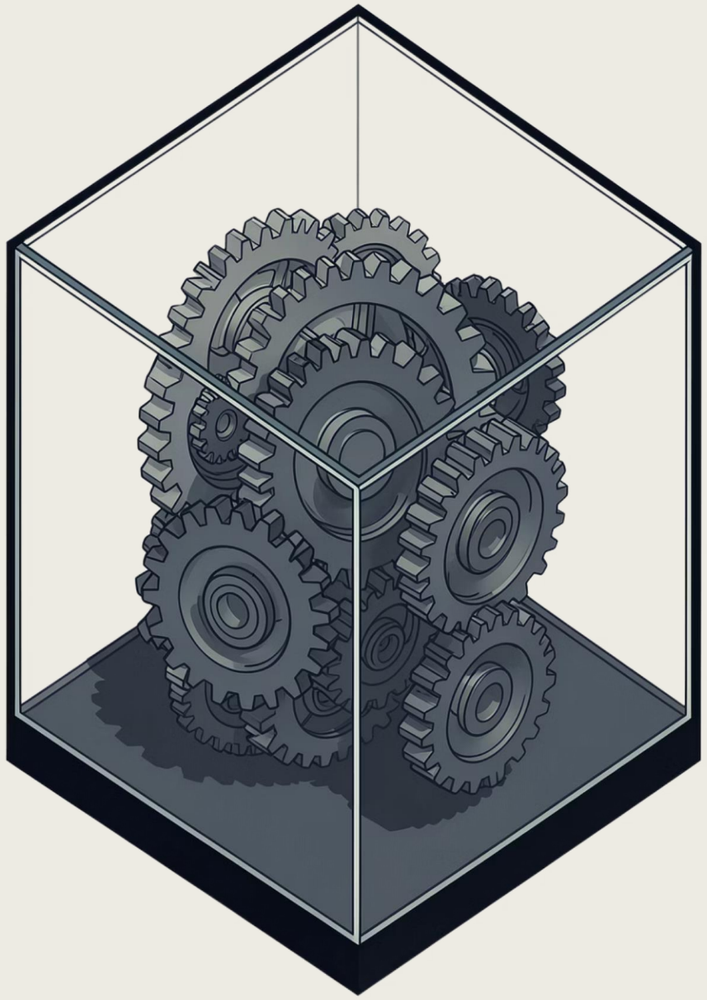
IMPLEMENTACIÓN DE SEMÁFOROS

Semáforo Binario

```
typedef struct {
            bool value;
            struct hebra *L;
} semaphoreBin;

void wait(semaphoreBin S) {
            if (S.value == false){
                        agregar hebra a S.L;
                        block();
                        }
            else
                s.value=false;
}

void signal(semaphoreBin S){
            if (!vacia(S.L){
                        remover hebra T de S.L;
                        wakeup(T);
                        }
            else
               s.value=true;
}
```

Semáforo Contador

```
typedef struct {
            int value;
            struct hebra *L;
} semaphore;

void wait(semaphore S) {
            S.value--;
            if (S.value < 0){
                        agregar hebra a S.L;
                        block();
            }
}



void signal(semaphore S){
            S.value++;
            if (S.value <=0){
                        remover hebra T de S.L;
                        wakeup(T);
            }
}
```



6.7

Monitores: Abstracción y Seguridad

Encapsulamiento

Estructuras abstractas que agrupan recursos compartidos y las operaciones sobre ellos.

Exclusión Implícita

Solo un proceso puede estar activo dentro del monitor en un momento dado.

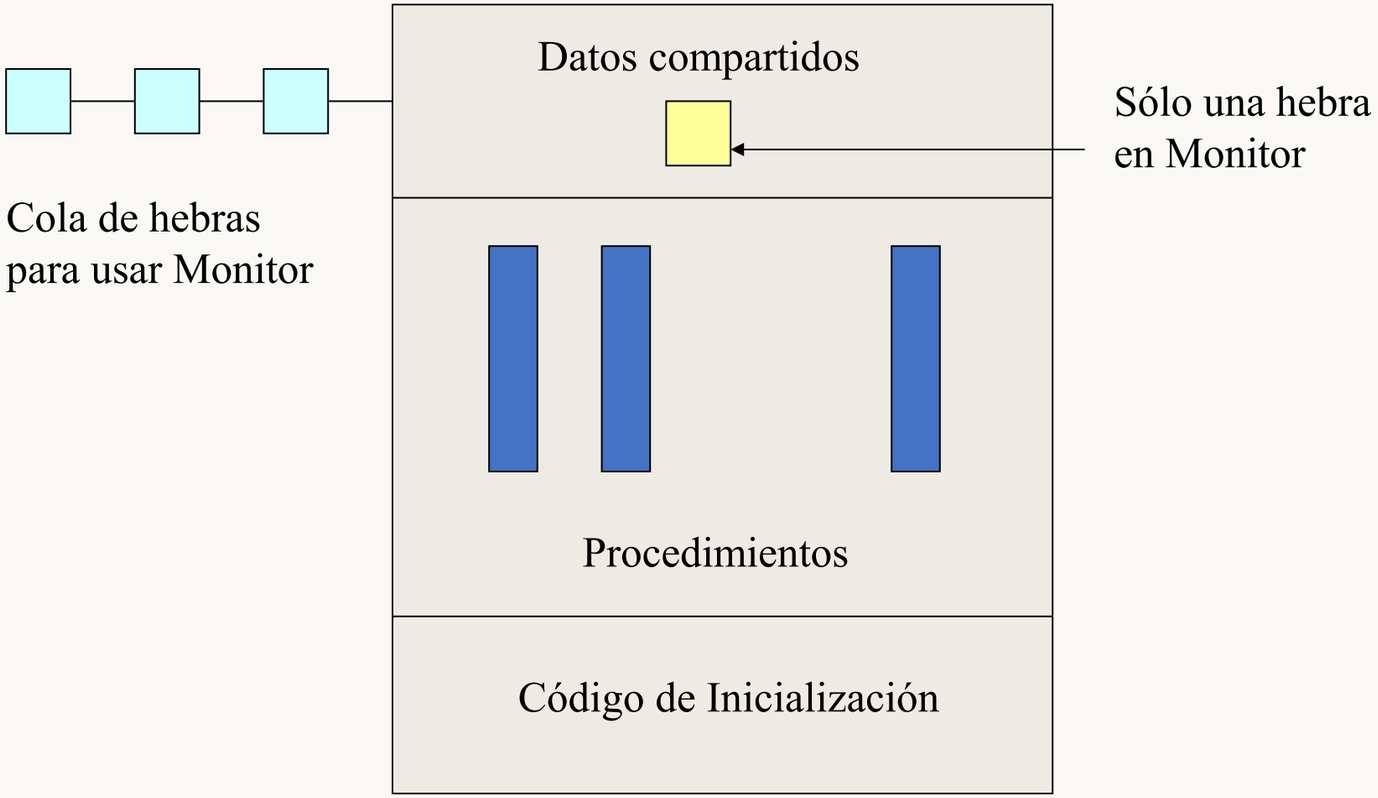
Variables de Condición

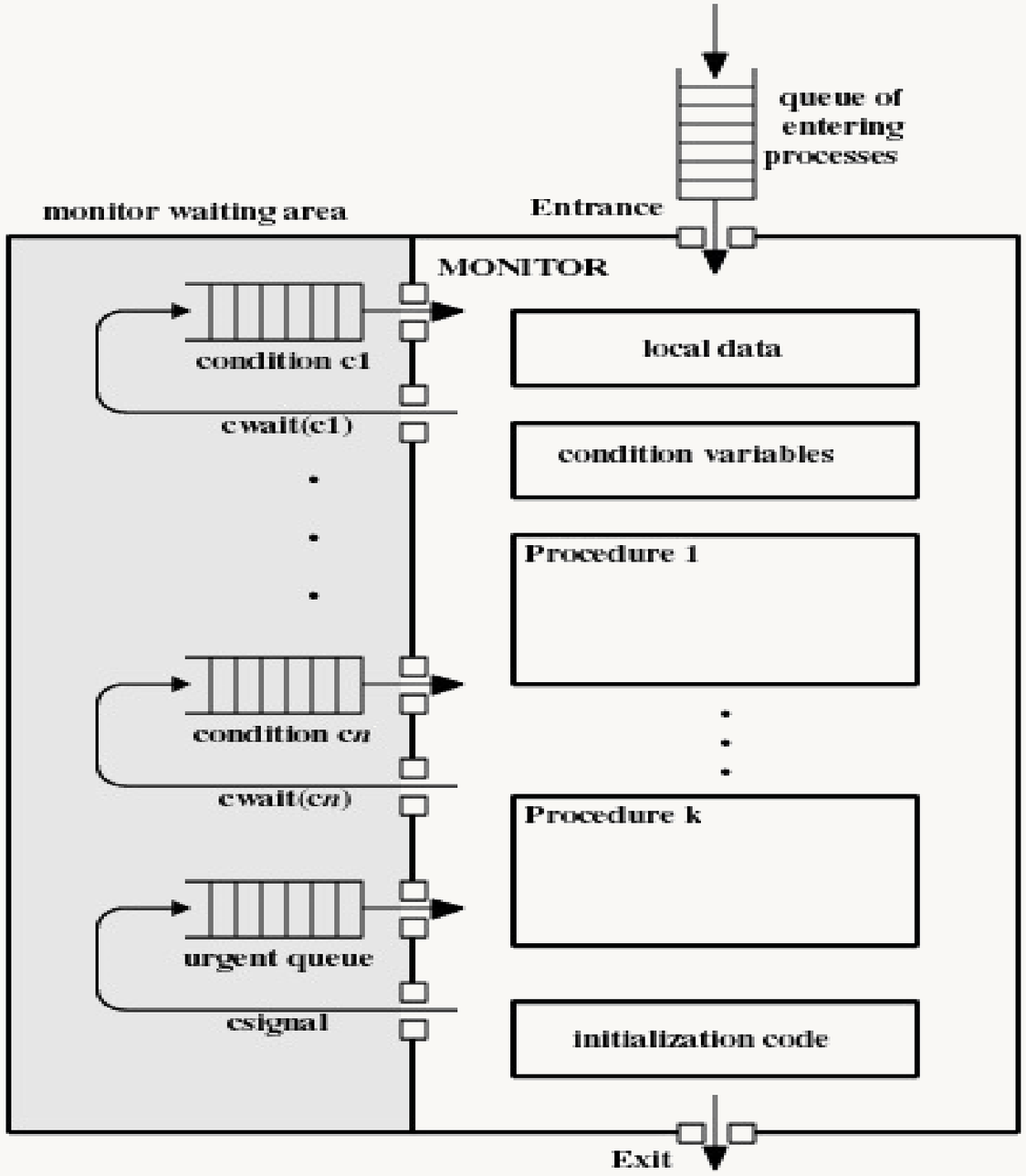
Permiten esperar y notificar eventos (wait/signal) de forma estructurada.

Ejemplo Práctico

Monitor para gestionar acceso concurrente a una base de datos compartida.

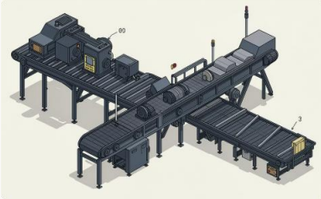
ILUSTRACIÓN DE UN MONITOR




queue of
entering
processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
condition cn
cwait(cn)
urgent queue
csignal
local data
condition variables
Procedure 1
Procedure k
initialization code
Exit

PROBLEMAS CLASICOS DE SINCRONIZACIÓN

Problemas Clásicos de Sincronización



Productor-Consumidor

Un productor genera datos y un consumidor los procesa, compartiendo un buffer acotado.

Filósofos Comensales

Cinco filósofos, cinco tenedores. ¿Cómo evitar el interbloqueo (deadlock) cuando todos quieren comer?

PROBLEMAS CLASICOS DE SINCRONIZACIÓN

Problemas Clásicos de Sincronización

Barbero Dormilón

Representa la coordinación entre un servidor y sus clientes; el servidor (barbero) descansa si no hay trabajo y los clientes deben despertarlo o esperar en sillas limitadas, evitando que se pierdan por falta de comunicación.

Lector-Escritor

Representa el acceso a datos compartidos; permite que muchos lean al mismo tiempo (lectura concurrente), pero exige que solo uno escriba a la vez (exclusión mutua) para evitar que la información se corrompa.

USANDO SEMÁFOROS EN PRODUCTOR/CONSUMIDOR

```
var mutex: semaphore = 1; exclusion mutua a dato compartido
    empty: semaphore = N; contador con N slots en buffer
    full: semaphore = 0;

producer:
    wait(empty);   espera si no hay slots disponibles (todo lleno)
    wait(mutex);   obtiene acceso a dato compartido agregar item a buffer
    signal(mutex);  libera lock para accesar dato compartido
    signal(full); notifica que hay un slot mas disponible por si hay alguien esperando por el
Consumer:
    wait(full); espera si no hay datos en el buffer (buffer vacío)
    wait(mutex): obtiene acceso a dato compartido
    remove item de buffer
    signal(mutex): libera lock para accesar dato compartido
    signal(empty): notifica que hay un slot vacío
```

USANDO SEMÁFOROS EN LECTORES/ESCRITORES

```
var mutex: semaphore; controla acceso a contador de lectores
    write: semaphore; controla a un escritor o primer lector
    readcounter: integer;  número de lectores

Writer:
    wait(write);  espera por escritor o lector
            write objet; Escribe objeto
    signal(write); permite leer y/o escribir a otros, escritura completada

Reader:
    wait(mutex);  asegura acceso exclusivo a contador de lectores
            readcounter = readcounter +1; incrementa lectores
            if(readcounter == 1) then wait(write); Si es el primer lector espera si hay escritor
    signal(mutex);
            Reading
    wait(mutex); asegura acceso exclusivo a contador de lectores
            readcounter = readcounter - 1; lector terminó de leer
            if(readcounter == 0) then signal(write); no mas lectores por si escritor esperaba
    signal(mutex)
```

```
program productor_consumidor
    monitor buffer_acotado;
        buffer: array [0..N] of caracteres;          {espacio para N elementos}
        sigent, sigsal: entero;                      {apuntadores al buffer}
        contador: entero;                            {número de elementos en el buffer}
        no_lleno, no_vacío: condition;               {para sincronización}
var i: entero;
    procedure añadir (x: carácter);
        begin
            if contador=N then cwait (no_lleno);     {buffer lleno; se impide producir}
            buffer[sigent] := x;
            sigent := sigent + 1 mod N;
            contador := contador + 1;                {un elemento más en el buffer}
            csignal (no_vacío);                      {reanudar un consumidor en espera}
        end;
    procedure tomar (x: carácter);
        begin
            if contador=0 then cwait (no_vacío)      {buffer vacio; se impide consumir}
            x := buffer[sigsal];
            sigsal := sigsal +1 mod N;
            contador := contador −1;                 {un elemento menos en el buffer}
            csignal (no_lleno);                      {reanudar un productor en espera}
        end;
    begin                                            {cuerpo del monitor}
            sigent := 0; sigsal := 0; contador := 0; {buffer inicialmente vacío}
    end;
```

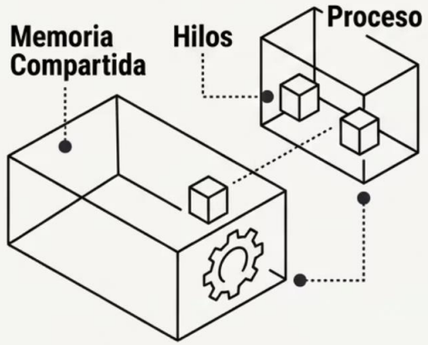
```
procedure productor;
var x: carácter;
begin
    repeat
        producir(x);
        añadir(x);
    forever
end;
procedure consumidor;
var x: carácter;
begin
    repeat
        tomar(x);
        consumir(x);
    forever
end;
begin (*programa principal*)
    parbegin
        productor; consumidor
    parend
end.
```

FIGURA 4.24 Una solución al problema del productor/consumidor con buffer acotado por medio de monitores

LINUX

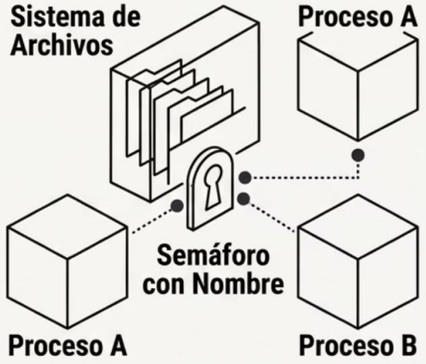
Semáforos en Linux

Semáforos SIN NOMBRE



Ámbito: Local
Uso: Hilos del mismo proceso
Almacén: Memoria compartida
Rendimiento: Más rápidos

Semáforos CON NOMBRE



Ámbito: Global
Uso: Procesos distintos
Persistencia: En sistema de archivos
Función: IPC entre procesos no ro-

Primitivas del sistema

Linux provee una API POSIX para semáforos:

sem_wait()

Equivalente a wait() / P().
Bloquea si el valor es 0.

sem_post()

Equivalente a signal() / V().
Desbloquea un hilo en espera.

TRANSACCIONES ATÓMICAS

Transacciones Atómicas: Integridad Garantizada

Una secuencia de operaciones que se ejecuta como **una única unidad lógica**: todo o nada.

Atomicidad

Todas las operaciones se completan o ninguna lo hace.

Consistencia

La transacción lleva el sistema de un estado válido a otro.

Aislamiento

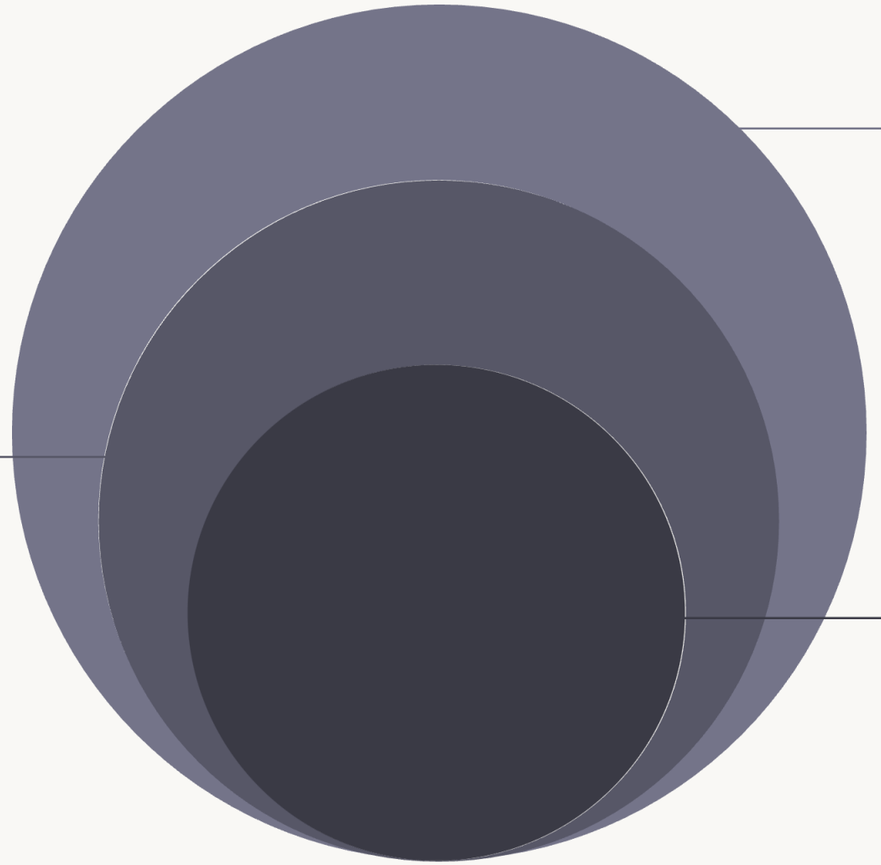
Las transacciones concurrentes no interfieren entre sí.

Durabilidad

Los cambios confirmados son permanentes, incluso ante fallos.

Aplicaciones típicas: Operaciones bancarias, actualizaciones de bases de datos, sistemas de reservas.

Resumen: El Ecosistema de la Sincronización



Semáforos

Wait/Signal para control bajo

Monitores

Encapsulan método y estado

Transacciones Atómicas

ACID para integridad completa

Cada mecanismo resuelve el mismo problema desde un nivel distinto de abstracción. Los semáforos son la base; los monitores y transacciones construyen sobre ellos para simplificar la programación concurrente y garantizar robustez.

CONCLUSIÓN

Sincronización Segura: Puntos Clave

1

La concurrencia es poderosa, pero peligrosa

Sin sincronización, los procesos pueden corromper datos compartidos mediante condiciones de carrera.

2

La sección crítica exige garantías formales

Exclusión mutua, progreso y espera limitada son las tres condiciones que toda solución debe cumplir.

3

El hardware provee los cimientos

Instrucciones atómicas como `Test-and-Set` y `Swap` permiten construir mecanismos de exclusión mutua eficientes y robustos.

4

El software, la solución lógica

Algoritmos y estructuras de datos permiten controlar el acceso la Sección Crítica.

Fin Unidad 6