

¿Qué es la Concurrency?

Es cuando **dos o más procesos/hilos** se ejecutan **aparentemente al mismo tiempo** y pueden interactuar entre sí o compartir recursos. Ocurre en:

- **Multiprogramación:** varios procesos en un procesador.
- **Multiprocesamiento:** varios procesos en varios núcleos.
- **Procesamiento distribuido:** procesos en distintas computadoras.


La concurrency aparece en tres contextos:

- ✓ **Múltiples aplicaciones.** La multiprogramación fue ideada para permitir compartir dinámicamente el tiempo de procesamiento entre varias aplicaciones.
- ✓ **Aplicaciones estructuradas.** Algunas aplicaciones pueden ser programadas como un conjunto de procesos concurrentes.
- ✓ **Estructura del Sistema Operativo.** Los sistemas operativos son muchas veces implementados como un conjunto de procesos o hilos.

Problemas comunes en concurrency

1. **Compartición peligrosa:** varios procesos acceden a los mismos recursos.
2. **Coordinación compleja para el SO:** difícil garantizar orden, exclusividad y eficiencia.
3. **Errores difíciles de detectar:** los resultados cambian entre ejecuciones.

Conceptos clave

Concepto	Definición breve	
Sección crítica	Parte del código que accede a recursos compartidos. Solo un proceso puede ejecutarla a la vez.	
Exclusión mutua	Garantiza que solo un proceso entre a su sección crítica al mismo tiempo.	
Condición de carrera	Error por acceso simultáneo a datos compartidos. Resultado impredecible.	
Deadlock (interbloqueo)	Dos o más procesos quedan esperando recursos entre sí y nadie avanza.	
Live lock (círculo vicioso)	Procesos reaccionan al estado de otros pero no hacen progreso útil .	
Starvation (inanición)	Un proceso nunca es elegido para ejecutar, aunque esté listo.	

Exclusión Mutua

¿Qué se debe garantizar?


1. **Un solo proceso** en su sección crítica.
2. Los que no están en sección crítica no deben bloquear a otros.
3. **Sin inanición** ni bloqueos indefinidos.
4. Si nadie está en la sección crítica, debe permitirse el acceso sin demora.
5. No se deben asumir velocidades de CPU ni número de núcleos.
6. Tiempo finito en la sección crítica.

Estructura típica de un proceso

```
sección_residual
entrada_sección_crítica
sección_crítica
salida_sección_crítica
```

Soluciones de software para 2 procesos (resumen)

Intento 1: variable **turno**


- Se turnan para entrar.
-  Si uno falla, el otro queda bloqueado.

```
int turno = 0;

void proceso0() {
    while (turno != 0);
    // sección crítica
    turno = 1;
}

void proceso1() {
    while (turno != 1);
    // sección crítica
    turno = 0;
}
```

Intento 2: **estado[]** booleano

- Cada proceso dice si quiere entrar.
-  No evita que entren los dos al mismo tiempo.

```
enum booleano { falso = 0, verdadero = 1 };
booleano estado[2] = {0, 0};

void proceso0() {
    while (estado[1]);
    estado[0] = verdadero;
    // sección crítica
    estado[0] = falso;
}

void proceso1() {
    while (estado[0]);
    estado[1] = verdadero;
    // sección crítica
    estado[1] = falso;
}
```


Intento 3: invierten el orden

-  Puede causar interbloqueo.

```
void proceso0() {
    estado[0] = verdadero;
    while (estado[1]);
    // sección crítica
    estado[0] = falso;
}

void proceso1() {
    estado[1] = verdadero;
    while (estado[0]);
    // sección crítica
    estado[1] = falso;
}
```

Intento 4: más “respetuosos”

- Cambian su intención si el otro quiere entrar.
-  Puede haber **live lock** (ambos ceden constantemente y nadie entra).

```
void proceso0() {
    estado[0] = verdadero;
    while (estado[1]) {
        estado[0] = falso;
        // espera voluntaria
        estado[0] = verdadero;
    }
    // sección crítica
    estado[0] = falso;
}

void proceso1() {
    estado[1] = verdadero;
    while (estado[0]) {
        estado[1] = falso;
        // espera voluntaria
        estado[1] = verdadero;
    }
    // sección crítica
    estado[1] = falso;
}
```

Soluciones correctas

✓ Algoritmo de Dekker

- Usa `estado[]` y `turno`.
- Garantiza exclusión mutua.
- Código **complejo**.

```
booleano estado[2] = {0, 0};
int turno = 1;

void proceso0() {
    while (1) {
        estado[0] = verdadero;
        while (estado[1]) {
            if (turno == 1) {
                estado[0] = falso;
                while (turno == 1);
                estado[0] = verdadero;
            }
        }
        // sección crítica
        turno = 1;
        estado[0] = falso;
    }
}
```

```

}

void proceso1() {
    while (1) {
        estado[1] = verdadero;
        while (estado[0]) {
            if (turno == 0) {
                estado[1] = falso;
                while (turno == 0);
                estado[1] = verdadero;
            }
        }
        // sección crítica
        turno = 0;
        estado[1] = falso;
    }
}
```

✓ Algoritmo de Peterson

- Más simple y claro.
- Uso de `estado[]` y `turno`.
- Correcto, eficiente y didáctico.



```
booleano estado[2] = {0, 0};
int turno;

void proceso0() {
    while (1) {
        estado[0] = verdadero;
        turno = 1;
        while (estado[1] && turno == 1);
        // sección crítica
        estado[0] = falso;
    }
}

void proceso1() {
    while (1) {
        estado[1] = verdadero;
        turno = 0;
        while (estado[0] && turno == 0);
        // sección crítica
        estado[1] = falso;
    }
}
```


Soporte por hardware

Deshabilitar interrupciones (monoprocesador)

- Evita que un proceso sea interrumpido.
-  Funciona en sistemas simples.
-  No sirve en multiprocesadores, afecta el rendimiento.

Instrucciones especiales

`test_and_set()`

- Cambia el valor de una variable sólo si es 0.
- Evita interferencias.
-  Usa espera activa.

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

`exchange()`

- Intercambia un valor en memoria con un registro.
- Controla acceso a la sección crítica

```
void exchange(int *registro, int *memoria) {  
    int temp = *memoria;  
    *memoria = *registro;  
    *registro = temp;  
}
```

```
int cerrojo = 0;

void P(int i) {
    int llavei = 1;
    while (1) {
        exchange(&llavei, &cerrojo);
        while (llavei != 0);
        // sección crítica
        exchange(&llavei, &cerrojo);
    }
}
```

✓ Ventajas del soporte hardware

- Funciona en sistemas multiprocesador.
- Es simple y atómico.

✗ Desventajas

- **Espera activa:** el CPU se desperdicia mientras los procesos esperan.
- **Inanición:** un proceso podría no entrar nunca.
- **Interbloqueo:** prioridad baja puede hacer que un proceso nunca avance.



Conclusión

- La concurrencia es fundamental en sistemas operativos.
- Sin exclusión mutua, los programas concurrentes pueden fallar.
- Se puede implementar con software (Peterson, Dekker) o con instrucciones especiales de hardware.
- Los semáforos, que viste antes, son herramientas que abstraen esta lógica y facilitan la sincronización.