1. Conceptos básicos y motivación

1. **Programación secuencial**
    - Contiene **declaraciones de datos** + **instrucciones** que se ejecutan de **forma lineal**.
2. **Programación concurrente**
    - Incluir **paralelismo potencial** + resolver problemas de **sincronización**.
    - Se compone de varios **procesos** que podrían ejecutarse en **paralelo**.
3. **Programación paralela**
    - Objetivo: **acelerar** la resolución de problemas.
    - Aporta **rendimiento** usando **múltiples** unidades de procesamiento.
4. **Programación distribuida**
    - Objetivo: **cooperación** de varios componentes en **ubicaciones distintas**.

Modelo abstracto y consideraciones de HW

- **Concurrencia en monoprocesadores**:
    - El **SO** reparte la CPU entre procesos. Se usa **memoria compartida** para comunicar.
- **Concurrencia en multiprocesadores de memoria compartida**:
    - *Varios procesadores* → **comparten** un **espacio de direcciones**.
- **Concurrencia en sistemas distribuidos**:
    - *Varios procesadores* → **NO** comparten memoria física; se comunican mediante **red**.

---

2. Sentencias atómicas y no atómicas

- **Sentencia atómica**:
    - Se ejecuta **sin interrupción**.
    - Ejemplos: `LOAD`, `ADD`, `STORE`.
- **Sentencia no atómica**:
    - Fusión de varias atómicas, *pueden* interrumpirse.

Interfoliación

- Distintas **formas de entrelazar** las instrucciones atómicas de dos o más procesos.
- **Consistencia secuencial estricta**:
    - Una **lectura** siempre ve el **último** valor escrito.

Hipótesis del progreso finito

- No asumimos **velocidades** de procesos, pero **todas** son mayores que 0.

---

3. Notación para la ejecución concurrente

- **Sistemas estáticos**:
    - Nº de procesos **fijado**.
- **Sistemas dinámicos**:
    - Nº de procesos **variable**.
- **Grafo de sincronización (DAG)**:
    - Cada nodo = un **bloque de sentencias**.
- **Creación de procesos**:
    - **FORK-JOIN** → "crear" y "esperar" procesos.
    - **COBEGIN-COEND** → bloque de sentencias que inician **todas a la vez**.

---

4. Exclusión mutua y sincronización

- **Exclusión mutua (SC)**:
    - *Sección Crítica*: parte del código que **solo** un proceso puede ejecutar **a la vez**.
- **Condición de sincronización**:
    - Restringe el orden de ejecución de **distintos** procesos.

Consecuencias de no sincronizar

- Lecturas incorrectas, sobrescritura de datos, resultados impredecibles.

---

5. Propiedades de los sistemas concurrentes

- **Propiedades de seguridad** (se cumplen **en cada instante**):
    - **Exclusión mutua**.
    - **Ausencia de interbloqueo** (nadie espera algo que nunca pasará).
- **Propiedades de vivacidad** (se cumplen **eventualmente**):
    - **Ausencia de inanición** (ningún proceso "eternamente pospuesto").
    - **Equidad** (avance equilibrado entre procesos).

Verificación de programas concurrentes

1. **Enfoque operacional** (análisis exhaustivo):

    - Se exploran **todas** las interfoliaciones.
    - *Peligro*: Explosión combinatoria.

2. **Enfoque axiomático** (sistema lógico):
    - Reglas y axiomas formales (p.ej. `{P} S {Q}`).
    - **Invariante global**: predicado que permanece **siempre cierto**.

Tema 2: Sincronización en memoria compartida

2.1 Introducción

- **Soluciones de bajo nivel (espera ocupada)**:
    - *Software*: lectura/escritura (algoritmos tipo Peterson).
    - *Hardware*: instrucciones especiales (`TestAndSet`).
- **Soluciones de alto nivel**:
    - *Bloqueo* de procesos.
    - E.g.: **Semáforos**, **Regiones críticas condicionales**, **Monitores**.

2.2 Soluciones de bajo nivel: espera ocupada

1. **Protocolo de entrada (PE)**
2. **Sección Crítica (SC)**
3. **Protocolo de Salida (PS)**

- **Deseable**: SC lo más corta posible.

- **Propiedades**:

    1. **Exclusión mutua**
    2. **Progreso** (evitar interbloqueos)
    3. **Espera limitada** (evitar inanición).

- **Eficiencia y equidad**:

    - Preferir **bajo coste** y no favorecer siempre al mismo proceso.

Dekker y Peterson

- **Dekker**: Usa “espera de cortesía” y variable de turno.
- **Peterson**: Más simple, usa dos variables lógicas y una de turno.

2.3 Soluciones HW con espera ocupada

- **Cerrojos (locks)**:

    - Variable booleana en memoria compartida.
    - No son atómicos → posible colisión.

- `TestAndSet`:

    - Instrucción de hardware.
    - *Atómica*:
        1. Lee valor.
        2. Lo pone a `true`.
        3. Devuelve el valor previo.

- **Desventajas**:
    - **Espera ocupada** consume CPU.
    - No garantiza **equidad**.
    - Posibles manipulaciones inseguras.

2.4 Semáforos

- **Definición**: Estructura con:
    - Valor natural (`S`)
    - Cola de procesos bloqueados
- **Operaciones**:
    - `sem_wait(S)` → *bloquea* si `S == 0`, en caso contrario `S--`.
    - `sem_signal(S)` → `S++`; si hay bloqueados, despierta uno.
- **Patrones sencillos**:
    1. **Espera única**
    2. **Exclusión mutua** (inicializar semáforo a 1)
    3. **Productor/Consumidor** (semáforos que marcan lleno/vacío)
- **Limitaciones**:
    - Diseño complejo en sincronizaciones avanzadas.
    - Riesgo de errores y bloqueos.
    - Poco modular (variables globales).
    - Dificultad de verificación.

---

5. Monitores como mecanismo de alto nivel

5.1 Introducción y definición

- **Monitor** = Objeto compartido con:
    1. **Variables encapsuladas** (recurso).
    2. **Procedimientos** que operan sobre dicho recurso.
- **Propiedades**:
    - *Acceso estructurado y encapsulado*.
    - *Exclusión mutua* en los procedimientos.
    - *Sincronización interna* usando **variables condición**.

5.1.2 Ventajas frente a semáforos

- **Variables protegidas**: Solo accesibles desde dentro del monitor.
- **Exclusión mutua garantizada** automáticamente.
- **Operaciones de espera y señal** *dentro* del monitor → más fácil de verificar.

5.1.3 Componentes de un monitor

1. **Variables permanentes** (estado interno).
2. **Procedimientos** (con variables locales y parámetros).
3. **Código de inicialización** (opcional).

> Visualmente:
>
> - "Procedimientos exportados" = la interfaz de acceso.
> - "Variables permanentes" = invisibles fuera.

5.2 Funcionamiento de los monitores

- **Exclusión Mutua**:
  - Un solo proceso puede estar *dentro del monitor* a la vez.
  - Cola de monitor: si alguien está dentro, el resto espera en FIFO.
- **Objetos pasivos**:
  - El monitor **no ejecuta** nada hasta que un proceso llama a uno de sus procedimientos.

5.3 Sincronización en monitores

- Se necesita bloquear/activar procesos según **condiciones**.
- En **monitores** no hay un contador como en semáforos, sino **variables condición** + `wait`/`signal`.

Variables condición

- Cada `condition` tiene una **cola**.
- `cond.wait()` → Bloquea el proceso (libera monitor).
- `cond.signal()` → Desbloquea **uno** (si existe) en la cola de `cond`.

Importante:

- Tras `wait`, el proceso se bloquea y *pierde* la exclusión mutua. Al despertar, **la recupera**.
- Puede haber varios procesos bloqueados en variables condición distintas.

5.4 Verificación de monitores

- Para demostrar corrección:
  1. Verificar cada monitor.

2. Verificar cada proceso individualmente.
3. Verificar la ejecución concurrente.

- **Invariante del monitor**:
  - Propiedad que se **mantiene** siempre (excepto durante la ejecución interna del monitor).
  - Debe cumplirse "antes y después" de cada `wait`, `signal` y llamada a procedimiento.

5.5 Patrones de solución con monitores

1. **Espera única**:
   - Asegura que un proceso (lector) no hace `L` antes de que otro (escritor) complete `E`.
   - Usa variable condición y un booleano para indicar si el proceso ya ha "terminado".
2. **Exclusión mutua**:
   - Permite a *n* procesos usar SC, pero **uno a la vez**.
3. **Productor/Consumidor**:
   - Similar al caso semáforos, pero se implementa dentro de un monitor (con variables compartidas, `wait`, `signal`).

5.6 El problema de Lectores/Escritores

- **Escritores**: modifican datos, no pueden coexistir con nadie más (ni lector ni escritor).
- **Lectores**: solo leen, pueden hacerlo simultáneamente entre ellos, pero no con un escritor.
- El monitor controla:
  - Número de lectores en curso.
  - Bloqueos cuando hay escritor.
  - Permite paralelismo de lectura.

5.7 Semántica de las señales de los monitores

Cuando hacemos `signal` a un proceso bloqueado en una variable condición:

1. **Señalar y Continuar (SC)**
   - El *señalador* sigue. El *señalado* pasa a la cola del monitor.
   - Exige re-verificar la condición al reactivarse.
2. **Señalar y Salir (SS)**
   - El *señalador* sale del monitor inmediatamente tras `signal`.
   - El *señalado* retoma la ejecución dentro del monitor.
3. **Señalar y Esperar (SE)**
   - El *señalado* se reactiva ya.

- El *señalador* se bloquea para readquirir el monitor después (queda en cola).

4. **Señalar y Espera Urgente (SU)**
    - Similar al anterior, pero el señalador espera en una **cola de urgentes** con más prioridad.

- **Todas** pueden resolver problemas de sincronización, pero difieren en:
    - **Eficiencia** y **comodidad** de programación.

5.7.4 Ejemplo comparativo: Barrera parcial

- Supongamos `p` procesos y una barrera que necesita al menos `n` procesos llegados.
- Distintas semánticas (SC, SS, SE, SU) se comportan de forma diferente al emitir varios `signal`.
- **SU** suele ser la que soluciona mejor el caso de la "barrera parcial".

5.8 Colas de prioridad

- Se puede ampliar un monitor para que `cond.wait(p)` reciba una **prioridad**.
- `cond.signal()` reactivará primero al de **menor** prioridad (p).
- No cambia la *lógica*, pero sí la *planificación*.

5.9 Implementación de monitores con semáforos

- Puede hacerse usando:
    - Un **semáforo** para la **cola del monitor** (`mutex` = 1).
    - Para cada **variable condición**, un semáforo inicializado a 0 + un contador de procesos esperando.
    - Para semántica SU, un semáforo adicional para los urgentes.

- **Limitaciones**:
    - No permite llamadas recursivas.
    - No garantiza orden FIFO.
    - Equivalentes en potencia a semáforos, pero **monitores** son más fáciles de usar.

¡Pistas para preguntas tipo test!

1. **Definiciones rápidas**:
    - "Programación concurrente es..."
    - "Exclusión mutua se define como..."
    - "Algoritmo de Peterson/Dekker se basa en..."

2. **Diferencias**:
    - *Monitores vs Semáforos*: ¿Qué garantías ofrece un monitor que no ofrecen los semáforos directamente?

- *Señalar y Continuar vs Señalar y Salir*: Dónde se bloquea el señalador y quién continúa primero.

3. **Ejemplos típicos**:
    - Productor/Consumidor.
    - Lectores/Escritores.
    - Espera única (E,L).
4. **Propiedades**:
    - Seguridad vs Vivacidad.
    - Interbloqueo e inanición.
5. **Lógica de verificación**:
    - Invariante global.
    - Invariante de monitor.

Resumen “en 5 puntos”

1. **Objetivo** de la concurrencia: **aprovechar** el paralelismo y **sincronizar** procesos.
2. **Exclusión mutua**: Solo un proceso en la sección crítica.
3. **Peterson y Dekker**: *Soluciones software* para 2 procesos (extensible a n).
4. **Semáforos**: Bloquean procesos → facilitan exclusión mutua y sincronización.
5. **Monitores**: Encapsulan datos y procedimientos → exclusión mutua automática + variables condición.

¡Listo! Mantienes todo el **contenido esencial**, pero con estructura para **repasar** y **memorizar**.
