Sistemas Concurrentes y Distribuidos

Apuntes Completos de Teoría y Práctica

Grado en Ingeniería Informática + Doble Grado DGIIM / ADE

Universidad de Granada (UGR)

Síntesis Integral de Contenidos:

Transparencias Oficiales del Profesor (Manuel I. Capel Tuñón)
Temario Completo de Infomates (LosDelDGIIM)
Apuntes de Ismael Sallami Moreno

Plataforma **repasaYA**

Curso Académico 2024–2025 / 2025–2026

Índice general

Tema 1

Introducción a la Programación Concurrente

Metadatos del Tema y Fuentes:

Fuentes utilizadas: Transparencias Oficiales del Profesor (Manuel I. Capel Tuñón), Temario Completo de Infomates (LosDelDGIIM) y Apuntes de Ismael Sallami Moreno.

1.1 Motivación y Conceptos Fundamentales

En la computación secuencial tradicional, un programa consiste en una secuencia única y totalmente ordenada de instrucciones que se ejecutan una tras otra en un procesador. La corrección de dicho programa depende exclusivamente del estado inicial y de la transformación lógica que realiza cada instrucción.

Sin embargo, el mundo real es inherentemente concurrente, y el hardware moderno ha alcanzado los límites físicos de disipación térmica y frecuencia de reloj (*Power Wall*), obligando a la transición hacia arquitecturas multinúcleo y sistemas distribuidos.

Definición: Concurrencia vs. Paralelismo

- **Concurrencia:** Es la propiedad de un sistema en el que múltiples tareas o secuencias de instrucciones están en progreso simultáneamente. No implica necesariamente ejecución física simultánea en el mismo instante de tiempo; puede lograrse mediante *intercalación* (*interleaving*) o multiprogramación en un único procesador mediante multiplexación temporal.
- **Paralelismo:** Es la ejecución estrictamente simultánea en el tiempo de dos o más secuencias de cálculo, requiriendo necesariamente soporte físico de hardware (múltiples núcleos, múltiples procesadores o sistemas distribuidos).

«La concurrencia trata sobre la estructura de un programa; el paralelismo trata sobre la ejecución física simultánea».

1.1.1 Procesos y Hebras (Hilos de Ejecución)

Para modelar y estructurar la ejecución concurrente, los sistemas operativos y los lenguajes de programación proporcionan abstracciones de ejecución:

- **Proceso (Proceso Pesado):** Es un programa en ejecución con su propio espacio de direcciones de memoria virtual aislado (código, datos, pila, montículo o *heap*), tabla de descriptores de ficheros y contexto en el sistema operativo. La comunicación entre procesos

distintos requiere mecanismos explícitos de comunicación interproceso (IPC: memoria compartida, tuberías, sockets o paso de mensajes).

- **Hebra o Hilo (*Thread* / Proceso Ligero):** Unidad básica de asignación de CPU dentro de un proceso. Todas las hebras creadas dentro de un mismo proceso comparten el mismo espacio de direcciones (memoria global y montículo) y recursos del proceso, pero cada hebra dispone de su propio contador de programa (PC), conjunto de registros de la CPU y pila de llamadas (*call stack*).

Importante / Ojo Examen: Consecuencia del Espacio Compartido

Dado que las hebras de un mismo proceso comparten la memoria global, el intercambio de datos entre ellas es instantáneo y muy eficiente. No obstante, esto introduce el peligro crítico de la concurrencia: el acceso simultáneo no sincronizado a variables compartidas provoca **condiciones de carrera** e inconsistencia de datos.

1.2 Axiomas y Principios Clave de la Concurrencia

Para razonar formalmente sobre programas concurrentes, se asume un modelo abstracto de computación basado en los siguientes principios fundamentales:

1.2.1 1. Atomicidad e Intercalación (Interleaving)

En ausencia de soporte de memoria transaccional, las instrucciones complejas de un lenguaje de alto nivel no se ejecutan de un solo golpe.

Definición: Acción Atómica

Una acción o instrucción es **atómica** si se ejecuta como una unidad indivisible e indivisa: ningún otro proceso puede observar estados intermedios de dicha operación, y su ejecución no puede ser interrumpida por otras instrucciones que accedan a los mismos datos.

Ejemplo Práctico: La ilusión de la asignación simple

Consideremos la instrucción en C/C++:

```
1 x = x + 1;
```

Aunque parece una sola sentencia, a nivel de máquina (código ensamblador) se descompone típicamente en tres operaciones elementales atómicas:

1. `LOAD R1, [x]` (Carga el valor de la memoria a un registro)
2. `ADD R1, 1` (Incrementa el valor en el registro)
3. `STORE R1, [x]` (Escribe el nuevo valor en la memoria)

Si dos hebras ejecutan simultáneamente $x = x + 1$ partiendo de $x = 0$, la intercalación de sus instrucciones en la CPU puede dar lugar a que el resultado final sea $x = 1$ en lugar de $x = 2$, perdiéndose una actualización.

1.2.2 2. Consistencia de Datos tras Acceso Simultáneo

Cuando dos instrucciones se ejecutan de manera concurrente sobre la misma variable compartida:

- Si ambas son de **lectura**, la operación es completamente segura.
- Si al menos una es de **escritura** y no existe sincronización, se produce una **condición de carrera** (*race condition*), dejando la variable en un estado impredecible y no determinista.

1.2.3 3. Irrepetibilidad de las Secuencias y Trazas de Ejecución

Definición: Traza de Ejecución

Una **traza** es una secuencia temporal de estados del sistema producida por un orden específico de intercalación de las instrucciones atómicas de los procesos concurrentes.

Dado que la planificación de la CPU, las interrupciones del hardware y las velocidades relativas de los núcleos son variables, un mismo programa concurrente ejecutado con las mismas entradas puede generar millones de trazas distintas. Por ello:

- Las secuencias de ejecución son **irrepetibles**.
- Los errores concurrentes son **transitorios** (*Heisenbugs*): aparecen esporádicamente en unas ejecuciones y desaparecen al intentar depurarlos con herramientas convencionales (p. ej., al introducir un `printf` o un punto de ruptura, se altera la temporización y el error se oculta).

1.2.4 4. Independencia de la Velocidad de Ejecución

La corrección lógica de un programa concurrente debe ser **independiente de la velocidad relativa** de los procesos. No se puede suponer que un proceso es «más rápido» o «más lento» que otro, ni diseñar la sincronización basándose en bucles de retardo temporales (*sleeps*). El programa debe funcionar correctamente tanto si un proceso se ejecuta en microsegundos como si se detiene durante segundos debido a una interrupción.

1.2.5 5. Hipótesis del Progreso Finito

Para poder garantizar que un sistema concurrente avanza hacia su meta, se asume el siguiente axioma:

Definición: Hipótesis del Progreso Finito

- **Progreso Global:** Se garantiza que, en cualquier instante en que existan procesos listos para ejecutarse, al menos uno de ellos ejecutará su siguiente instrucción en un tiempo finito.
- **Progreso Individual (Equidad / Fairness):** Si un proceso está listo para ejecutarse de forma continuada, el planificador le asignará la CPU en algún momento en tiempo finito, evitando que quede postergado indefinidamente.

1.3 Modelos y Notaciones para Expresar Concurrencia

A lo largo de la historia de la computación se han formalizado diferentes estructuras sintácticas para modelar la concurrencia:

1.3.1 1. Grafos de Sincronización (DAG)

Un grafo de sincronización es un Grafo Dirigido Acíclico (DAG, *Directed Acyclic Graph*) donde los nodos representan bloques de instrucciones o procesos, y las aristas dirigidas (u, v) representan relaciones de precedencia temporal: la tarea v solo puede comenzar una vez que la tarea u haya concluido por completo.

1.3.2 2. Notaciones Estructuradas: Cobegin / Coend

Propuesta originalmente por Dijkstra (`parbegin` / `parend`), define bloques donde las sentencias interiores se ejecutan en paralelo y el bloque finaliza cuando todas las ramas internas han concluido:

```
S0;
cobegin
    S1;
    S2;
    S3;
coend;
S4; // S4 solo se ejecuta tras finalizar S1, S2 y S3
```

1.3.3 3. Notaciones No Estructuradas: Fork y Join

Introducidas por Dennis y Van Horn (1966) y adoptadas por UNIX:

- `fork(etiqueta)`: Inicia un nuevo flujo de ejecución concurrente en la etiqueta indicada, mientras el proceso actual continúa en la siguiente instrucción.
- `join(contador)`: Sincroniza múltiples flujos concurrentes, reduciendo un contador atómico y bloqueando hasta que todas las ramas hayan alcanzado este punto.

1.3.4 4. Hebras Modernas en C++11 y OpenMP

En los estándares modernos, la concurrencia se gestiona mediante bibliotecas del lenguaje:

```
#include <iostream>
#include <thread>
#include <vector>

void tarea(int id) {
    std::cout << "Hebra " << id << " ejecutandose.\n";
}

int main() {
    std::vector<std::thread> hebras;
    for (int i = 0; i < 4; ++i) {
        hebras.emplace_back(tarea, i); // Creacion concurrente (Fork)
    }
    for (auto& h : hebras) {
        h.join(); // Sincronizacion de finalizacion (Join)
    }
    return 0;
}
```

Listing 1.1: Concurrencia estándar con hebras en C++11

1.4 Exclusión Mutua y Sincronización

Los dos problemas fundamentales en programación concurrente son:

1.4.1 1. El Problema de la Exclusión Mutua

Surge cuando dos o más procesos acceden a un recurso compartido (variable, fichero, dispositivo de hardware) y al menos uno de ellos realiza operaciones de modificación.

Definición: Sección Crítica (SC)

Segmento de código de un proceso que accede a uno o varios recursos compartidos y que **no debe ser ejecutado por más de un proceso de forma simultánea**.

Estructura canónica de un proceso con sección crítica:

```
while (true) {
    < Protocolo de Entrada a la SC >
        Seccion_Critica();
    < Protocolo de Salida de la SC >
        Seccion_No_Critica();
}
```

1.4.2 2. Condición de Sincronización

A diferencia de la exclusión mutua (donde la restricción es de competencia por recursos), la **sincronización condicional** es una relación de **cooperación**: un proceso debe suspender su ejecución hasta que otro proceso colaborador produzca un evento o modifique el estado del sistema satisfaciendo una condición boolean B.

Ejemplo Práctico: Paradigma Productor-Consumidor Elemental

Dos procesos cooperantes comparten un búfer de tamaño 1:

- El **Productor** genera un dato y lo deposita en el búfer. No puede volver a producir si el búfer aún contiene un dato no leído (riesgo de sobrescritura).
- El **Consumidor** retira el dato del búfer. No puede consumir si el búfer está vacío (riesgo de lectura espuria).

Ambos procesos deben sincronizarse condicionalmente sobre los predicados *«búfer lleno»* y *«búfer vacío»*.

1.5 Propiedades de los Sistemas Concurrentes

Formalizadas por Leslie Lamport, las propiedades que determinan la corrección de un sistema concurrente se dividen en dos categorías complementarias:

1.5.1 1. Propiedades de Seguridad (Safety)

Definición: Propiedad de Seguridad

Establece que **«nada malo ocurrirá durante la ejecución»**. Si una propiedad de seguridad se viola, la violación ocurre en un instante temporal discreto y finito, siendo observable en un prefijo de la traza de ejecución.

Ejemplos clave de seguridad:

- **Exclusión Mutua:** Nunca hay dos procesos simultáneamente dentro de la sección crítica ($N_{SC} \leq 1$).

- **Ausencia de Interbloqueo (*Deadlock*):** El sistema no alcanza ningún estado donde todos los procesos queden bloqueados esperando recursos que otros tienen asignados, impidiendo cualquier avance.

- **Consistencia de Datos:** Las invariantes de integridad de los datos compartidos se mantienen intactas tras cada operación.

1.5.2 2. Propiedades de Vivacidad (Liveness)

Definición: Propiedad de Vivacidad

Establece que **«algo bueno terminará ocurriendo eventualmente»**. No puede refutarse observando una traza finita; solo se viola si transcurre un tiempo infinito sin que el evento deseado suceda.

Ejemplos clave de vivacidad:

- **Ausencia de Inanición (*Starvation-Freedom*):** Si un proceso solicita acceder a la sección crítica o a un recurso, se le concede el acceso en un tiempo finito.

- **Equidad (*Fairness*):** Ningún proceso listo es ignorado arbitrariamente por el planificador mientras otros procesos progresan de forma indefinida.

- **Terminación:** Un algoritmo o tarea eventualmente completa su ejecución y finaliza.

Importante / Ojo Examen: Seguridad vs. Vivacidad en el Diseño

Un sistema que no hace nada en absoluto (por ejemplo, todos los procesos bloqueados permanentemente) cumple trivialmente todas las propiedades de seguridad (¡nada malo ocurre!), pero fracasa estrepitosamente en las de vivacidad. El reto de la programación concurrente es garantizar ambas simultáneamente.

1.6 Verificación Formal de Programas Concurrentes

Para garantizar la corrección de un programa concurrente, las pruebas convencionales de software (*testing*) son insuficientes debido a la explosión combinatoria del número de trazas posibles. Se requieren métodos formales.

1.6.1 Enfoque Operacional vs. Enfoque Axiomático

- **Enfoque Operacional (Model Checking):** Construye el grafo explícito de todos los estados alcanzables del sistema. Aunque es automático, sufre del problema de la **explosión del espacio de estados**: para n procesos de k instrucciones, el número de estados puede crecer exponencialmente como $(k!)^n$.

- **Enfoque Axiomático (Lógica de Hoare):** Utiliza un sistema lógico-deductivo basado en reglas de inferencia y asertos sobre las variables del programa, demostrando matemáticamente que el código satisface sus especificaciones independientemente de la velocidad de ejecución.

1.6.2 Fundamentos de la Lógica de Hoare

La lógica de Hoare se fundamenta en los **tripletes de Hoare**:

$$\{P\}\ C\ \{Q\} \tag{1.1}$$

Donde:

- P es la **Precondición**: predicado lógico que describe el estado del sistema antes de ejecutar C.
- C es el **Comando** o bloque de código que se ejecuta.
- Q es la **Postcondición**: predicado lógico que debe ser verdadero inmediatamente después de que C concluya.

Definición: Corrección Parcial vs. Corrección Total

- **Corrección Parcial:** Si la precondición P es cierta antes de C, y **si la ejecución de C termina**, entonces la postcondición Q será cierta al finalizar.
- **Corrección Total:** Corrección parcial + Demostración de que el programa **garantiza la terminación** en tiempo finito.

1.6.3 Axiomas Fundamentales para Programas Secuenciales

1. **Axioma de la Asignación:**

$$\{P[x \leftarrow E]\}\ x := E\ \{P\} \tag{1.2}$$

Donde $P[x \leftarrow E]$ denota la fórmula obtenida al sustituir todas las apariciones libres de la variable x por la expresión E en la fórmula P.

2. **Regla de la Composición Secuencial:**

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}} \tag{1.3}$$

3. **Regla de la Sentencia Condicional:**

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}} \tag{1.4}$$

4. **Regla del Bucle While (Invariante de Bucle):**

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{I \wedge \neg B\}} \tag{1.5}$$

Donde I es el **invariante de bucle**: un aserto que permanece verdadero antes, durante y después de cada iteración del ciclo.

1.6.4 Extensión a Programas Concurrentes: No Interferencia de Owicki-Gries

Para extender la lógica de Hoare a sentencias concurrentes compuestas (`cobegin` $S_1 \parallel S_2$ `coend`):

$$\frac{\{P_1\}\ S_1\ \{Q_1\} \qquad \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \textbf{cobegin}\ S_1 \parallel S_2\ \textbf{coend}\ \{Q_1 \wedge Q_2\}} \tag{1.6}$$

Esta regla es válida **únicamente si se cumple el Teorema de No Interferencia de Owicki-Gries**:

Definición: Condición de No Interferencia de Owicki-Gries

Dadas las demostraciones secuenciales de dos procesos concurrentes, la prueba del proceso 1 no es interferida por el proceso 2 si, para cada aserto intermedio $\{A\}$ en la traza de S_1 y cada acción atómica elemental T perteneciente a S_2 con precondición $\{pre(T)\}$, se verifica:

$$\{A \wedge pre(T)\}\ T\ \{A\} \tag{1.7}$$

Es decir, la ejecución de cualquier acción de un proceso vecino preserva la validez de los asertos lógicos intermedios de los demás procesos.

1.6.5 Técnica del Invariante Global

Cuando los procesos comparten múltiples variables y la comprobación de no interferencia de Owicki-Gries se vuelve inmanejable por el número cruzado de asertos, se utiliza la **Técnica del Invariante Global**:

- Se define un predicado lógico global I_{glob} que involucra a las variables compartidas y debe mantenerse invariante en todo estado accesible.
- Cada proceso garantiza que sus transiciones atómicas preservan I_{glob}:

$$\{I_{glob} \wedge P_i\}\ \langle S_i \rangle\ \{I_{glob} \wedge Q_i\} \tag{1.8}$$

- De este modo, la consistencia de las variables compartidas queda demostrada para cualquier intercalación posible.

Tema 2

Exclusión Mutua y Mecanismos de Sincronización Básicos

Metadatos del Tema y Fuentes:

Fuentes utilizadas: Transparencias Oficiales del Profesor (Manuel I. Capel Tuñón), Temario Completo de Infomates (LosDelDGIIM) y Apuntes de Ismael Sallami Moreno.

2.1 El Problema de la Exclusión Mutua

Cuando dos o más procesos o hebras se ejecutan concurrentemente y comparten datos o recursos en memoria, surge la necesidad de coordinar sus accesos para evitar que interfieran mutuamente de manera destructiva.

Definición: Sección Crítica (SC) y Sección No Crítica (SNC)

- **Sección Crítica (SC):** Es la porción de código de un proceso en la cual se accede y manipula uno o más recursos o variables compartidas. Para evitar inconsistencias de datos, debe cumplirse la propiedad de que en cualquier instante de tiempo a lo sumo un solo proceso se encuentre ejecutando su sección crítica.
- **Sección No Crítica (SNC):** Cualquier porción del código en la que el proceso trabaja únicamente con variables locales privadas o recursos no compartidos. Las acciones en la SNC no interfieren con otros procesos.

2.1.1 Estructura Canónica de un Proceso Concurrente

Cualquier proceso que compita por una sección crítica responde al siguiente esquema cíclico:

```
void Proceso(int i) {
    while (true) {
        // 1. Protocolo de entrada: solicita permiso y espera si la SC esta
            ocupada
        protocolo_entrada(i);

        // 2. Seccion Critica (SC): acceso seguro a variables compartidas
        seccion_critica();

        // 3. Protocolo de salida: libera el recurso y avisa a otros procesos
        protocolo_salida(i);
```

```
11
12          // 4. Seccion No Critica (SNC): computo independiente
13          seccion_no_critica();
14      }
15  }
```

2.1.2 Las Cuatro Condiciones de Dijkstra

Edsger W. Dijkstra formuló en 1965 los cuatro requisitos formales indispensables que cualquier protocolo de entrada y salida debe satisfacer obligatoriamente para considerarse una solución válida y correcta al problema de la exclusión mutua:

Definición: Condiciones de Dijkstra

1. **Exclusión Mutua Estricta (Seguridad):** Nunca puede haber más de un proceso simultáneamente dentro de la sección crítica ($N_{SC} \leq 1$).
2. **No Suposiciones sobre el Hardware ni Velocidades:** No se puede asumir ninguna hipótesis sobre el número de procesadores, ni sobre la velocidad relativa de ejecución de los procesos.
3. **Progreso / Ausencia de Interbloqueo (Seguridad/Vivacidad):** Un proceso que se encuentre ejecutando en su sección no crítica (SNC) o que haya terminado no puede impedir que otros procesos interesados accedan a la sección crítica. Si ningún proceso está en la SC y hay procesos intentando entrar, la decisión de cuál entra debe tomarse en un tiempo finito.
4. **Espera Limitada / Ausencia de Inanición (Vivacidad):** Ningún proceso que desee entrar a la sección crítica debe ser postergado indefinidamente. El tiempo de espera para entrar debe ser finito.

2.2 Algoritmos Software con Espera Activa (Spinlocks)

En los inicios de la computación, antes de que los sistemas operativos y los procesadores ofrecieran instrucciones de sincronización dedicadas, se investigó cómo resolver la exclusión mutua exclusivamente mediante instrucciones de lectura y escritura ordinarias sobre memoria compartida.

2.2.1 El Método del Refinamiento Sucesivo: Los Cuatro Intentos

Para comprender la lógica de los algoritmos correctos, es clásico analizar los cuatro intentos fallidos para 2 procesos (P_0 y P_1):

Intento 1: Alternancia Estricta (Variable Turno)

```
int turno = 0; // Variable compartida

// Proceso P0:
while (turno != 0) { /* Espera activa */ }
seccion_critica();
turno = 1;

// Proceso P1:
while (turno != 1) { /* Espera activa */ }
seccion_critica();
```

```
11 turno = 0;
```

- **Cumple:** Exclusión mutua estricta.
- **Falla: Viola la condición de progreso**. Si P_0 ejecuta su SC, pone `turno = 1`, pero P_1 se queda permanentemente en su sección no crítica, P_0 jamás podrá volver a entrar a la SC porque `turno` sigue siendo 1. La velocidad de un proceso queda atada a la del otro.

Intento 2: Banderas de Estado

Se sustituye el turno por dos variables booleanas de presencia: `bool en_sc[2] = {false, false};`

```
// Proceso Pi:
while (en_sc[1 - i]) { /* Espera a que el otro salga */ }
en_sc[i] = true;
seccion_critica();
en_sc[i] = false;
```

- **Falla: Viola la exclusión mutua**. Si ambos procesos evalúan el bucle `while` a la vez cuando ambas banderas son `false`, ambos salen del bucle, ambos escriben `true` y ambos entran a la SC a la vez.

Intento 3: Banderas de Intención Previas

Para evitar el fallo del Intento 2, cada proceso declara su intención de entrar *antes* de comprobar al otro:

```
bool quiere_entrar[2] = {false, false};

// Proceso Pi:
quiere_entrar[i] = true;
while (quiere_entrar[1 - i]) { /* Espera */ }
seccion_critica();
quiere_entrar[i] = false;
```

- **Falla: Sufre de Interbloqueo (Deadlock)**. Si P_0 y P_1 ejecutan `quiere_entrar[i] = true` simultáneamente, ambos encuentran la bandera del vecino en `true` y ambos entran en bucle infinito de espera activa, bloqueándose para siempre.

Intento 4: Retirada de Cortesía

Para solucionar el interbloqueo del Intento 3, si un proceso ve que el otro también quiere entrar, baja su bandera momentáneamente para cederle el paso:

```
// Proceso Pi:
quiere_entrar[i] = true;
while (quiere_entrar[1 - i]) {
    quiere_entrar[i] = false;
    sleep(random);
    quiere_entrar[i] = true;
}
seccion_critica();
quiere_entrar[i] = false;
```

- **Falla: Sufre de Bloqueo Activo (Livelock)**. Si ambos procesos ejecutan a la par, pueden levantar y bajar sus banderas al unísono infinitamente sin que ninguno llegue a entrar jamás a la SC.

2.2.2 El Algoritmo de Dekker (1965)

Fue el primer algoritmo completamente correcto conocido en la historia de la computación para resolver el problema de la exclusión mutua para 2 procesos. Combina las banderas de intención del Intento 3 con una variable de desempate (`turno`) para romper la simetría cuando ambos desean entrar a la vez.

Algoritmo / Protocolo: Algoritmo de Dekker para 2 Procesos

```
bool quiere[2] = {false, false};
int turno = 0;

void Proceso(int i) {
    int j = 1 - i; // Identificador del otro proceso
    while (true) {
        quiere[i] = true;
        while (quiere[j]) {
            if (turno != i) {
                quiere[i] = false;
                while (turno != i) { /* Espera pasiva de su turno */ }
                quiere[i] = true;
            }
        }

        seccion_critica();

        turno = j;           // Cede el turno al otro proceso
        quiere[i] = false;   // Indica que ya ha salido

        seccion_no_critica();
    }
}
```

2.2.3 El Algoritmo de Peterson para 2 Procesos (1981)

Gary L. Peterson simplificó drásticamente la solución de Dekker en 1981 con un algoritmo sumamente elegante, considerado una obra maestra de la programación concurrente.

Algoritmo / Protocolo: Algoritmo de Peterson para 2 Procesos

```
bool quiere[2] = {false, false};
int turno = 0;

void Proceso(int i) {
    int j = 1 - i;
    while (true) {
        // PROTOCOLO DE ENTRADA:
        quiere[i] = true;  // 1. Declaro mi intencion de entrar
        turno = j;         // 2. Cortesamente ofrezco el turno al otro
        while (quiere[j] && turno == j) {
            // Espera activa mientras el otro quiera Y mantenga el
               turno
        }

        seccion_critica();

        // PROTOCOLO DE SALIDA:
        quiere[i] = false; // Indico que he salido de la SC
```

```
18
19          seccion_no_critica();
20      }
21  }
```

Demostración Formal de la Exclusión Mutua en Peterson

Demostremos que P_0 y P_1 nunca pueden estar juntos en la SC por reducción al absurdo:

Demostración. Supongamos que ambos procesos están en la SC simultáneamente. Para que P_0 haya salido de su bucle de espera, debe haber evaluado como falsa la condición:

$$\text{quiere}[1] == \text{false} \quad \vee \quad \text{turno} == 0$$

Como ambos están en la SC, obligatoriamente quiere[0] == true y quiere[1] == true. Por consiguiente, la única forma de que P_0 entrara es que viera turno == 0. Análogamente, para que P_1 haya entrado, debe haber visto turno == 1. Sin embargo, `turno` es una variable atómica escalar que solo puede tener un único valor (0 o 1). El valor de `turno` corresponde a la **última asignación realizada**. Si P_0 asignó turno = 1 en último lugar, entonces turno == 1, lo que impidió entrar a P_0 y permitió entrar solo a P_1. Es imposible que turno == 0 y turno == 1 simultáneamente. Contradicción. Por tanto, se garantiza la exclusión mutua. □

2.2.4 Exclusión Mutua para N Procesos

Para extender la exclusión mutua por software a N procesos existen dos algoritmos célebres:

- **Filtro de Peterson:** Generaliza el algoritmo para 2 procesos creando $N-1$ etapas o niveles de competición. En cada nivel, al menos un proceso queda esperando en el filtro, de forma que a la etapa $N-1$ solo llega un único ganador.
- **Algoritmo de la Panadería de Lamport (*Bakery Algorithm*):** Simula el sistema de turnos de una panadería:
  1. Cada proceso i que desea entrar toma un número de turno: $\text{turno}[i] = 1 + \text{máx}(\text{turno}[0], \ldots, \text{turno}[n-1])$.
  2. Se atiende a los procesos por orden lexicográfico estricto del par $(\text{turno}[i], i)$.
  3. **Propiedad excepcional:** Es correcto incluso si las lecturas y escrituras en memoria no son estrictamente atómicas (tolera lecturas solapadas con escrituras en variables compartidas).

2.3 Soporte Hardware para Exclusión Mutua

Aunque los algoritmos de software puros son teóricamente fascinantes, en la práctica presentan graves inconvenientes: consumen el 100 % de CPU en bucles de espera activa (*busy waiting*), son complejos para N procesos y en los procesadores modernos con ejecución fuera de orden (*out-of-order execution*) requieren barreras de memoria explícitas (*memory fences*).

Por ello, los procesadores modernos incorporan instrucciones atómicas a nivel de instrucción máquina:

2.3.1 1. Deshabilitación de Interrupciones

En sistemas monoprocesador clásicos, un proceso de nivel núcleo puede deshabilitar las interrupciones hardware antes de entrar a la SC y rehabilitarlas al salir. Sin embargo:

- Es inaceptable en espacio de usuario (un proceso con bucle infinito congelaría el sistema entero).
- Es completamente inútil en sistemas multiprocesador multinúcleo (deshabilitar interrupciones en el Core 1 no impide que el Core 2 acceda a la misma memoria RAM).

2.3.2 2. Instrucciones Atómicas Read-Modify-Write (RMW)

Estas instrucciones ejecutan una lectura y una modificación de una posición de memoria de forma indivisible a nivel de bus de memoria o caché:

Definición: Test-and-Set (TAS)

Lee el valor original de una variable booleana en memoria y escribe incondicionalmente `true`, devolviendo el valor anterior de forma atómica:

```
bool TestAndSet(bool &cerrojo) {
    bool anterior = cerrojo;
    cerrojo = true;
    return anterior;
}
```

Implementación de un cerrojo simple de exclusión mutua (*Spinlock*):

```
bool cerrojo = false; // Libre

void entrar() {
    while (TestAndSet(cerrojo)) { /* Espera activa */ }
}
void salir() {
    cerrojo = false; // Liberacion atomica
}
```

Definición: Compare-and-Swap (CAS)

Instrucción más potente y flexible. Compara el contenido de una dirección de memoria con un valor esperado; solo si coinciden, almacena el nuevo valor:

```
bool CompareAndSwap(int *dir, int valor_esperado, int nuevo_valor) {
    if (*dir == valor_esperado) {
        *dir = nuevo_valor;
        return true;
    }
    return false;
}
```

El CAS es la piedra angular de las estructuras de datos concurrentes libres de bloqueos (*lock-free programming*) y de la biblioteca `<atomic>` de C++11 (como `std::atomic<T>::compare_exchange_weak`).

2.4 Semáforos de Dijkstra

Para superar el grave problema del desperdicio de CPU que produce la espera activa, Edsger Dijkstra introdujo en 1965 el concepto de **Semáforo**.

Definición: Semáforo

Un semáforo S es una variable protegida especial con valor entero que dispone de una cola de espera de procesos bloqueados y sobre la que **únicamente se permiten tres operaciones**:

1. **Inicialización:** Establece el valor inicial no negativo de S.
2. **Operación P(S) / wait(S):** (Del holandés *proberen*, probar/decrementar). Decrementa el valor del semáforo. Si el valor resultante es negativo (o si valía 0), el proceso invocador suspende su ejecución y pasa a la cola de bloqueados del semáforo.
3. **Operación V(S) / signal(S) / post(S):** (Del holandés *verhogen*, incrementar). Incrementa el valor del semáforo. Si hay procesos esperando en la cola, despierta a uno de ellos y lo pasa a estado listo.

Tanto $P(S)$ como $V(S)$ son operaciones **estrictamente atómicas**.

2.4.1 Tipos de Semáforos

- **Semáforo Binario (Mutex):** Su valor solo puede ser 0 o 1. Se utiliza típicamente para exclusión mutua (inicializado a 1).
- **Semáforo General o Contador:** Su valor es un entero mayor o igual que 0. Se inicializa al número K de unidades disponibles de un recurso compartido.

2.4.2 El Invariante del Semáforo

Sea S_{init} el valor inicial del semáforo, N_P el número de operaciones P completadas con éxito y N_V el número de operaciones V completadas. Se verifica siempre la siguiente propiedad invariante fundamental:

$$S_{actual} = S_{init} + N_V - N_P \geq 0 \tag{2.1}$$

Por tanto, si un semáforo se inicializa a 1 (para exclusión mutua):

$$N_P - N_V \leq 1 \tag{2.2}$$

Como los procesos en la SC corresponden a los que han completado P y no han hecho V, el número de procesos en la SC cumple $N_{SC} = N_P - N_V \leq 1$, garantizando formalmente la exclusión mutua.

2.4.3 Semáforos en la Práctica: POSIX y C++20

En entornos UNIX/Linux se utiliza la API POSIX (`semaphore.h`):

```
#include <semaphore.h>

sem_t sem_mutex;

int main() {
    sem_init(&sem_mutex, 0, 1); // Inicializa a 1 (compartido entre hebras)

    // Protocolo de entrada:
    sem_wait(&sem_mutex); // Equivale a P()

    // Seccion Critica...

```

```
13      // Protocolo de salida:
14      sem_post(&sem_mutex); // Equivale a V()
15
16      sem_destroy(&sem_mutex);
17  }
```

Listing 2.1: Uso de semáforos POSIX en C++

2.4.4 Limitaciones de los Semáforos

Aunque los semáforos son muy potentes y resuelven el problema de la espera activa, su uso en proyectos grandes resulta muy propenso a errores:

- **Falta de Estructura:** Son variables globales dispersas por el código; no existe asociación sintáctica entre el semáforo y los datos que protege.
- **Fragilidad:** Un error de programación en un solo proceso (olvidar un `sem_post`, invertir el orden de dos `sem_wait` o duplicar una llamada) provoca inmediatamente interbloqueo o violación de la exclusión mutua en todo el sistema.
- **Doble Propósito Confuso:** Se usan tanto para exclusión mutua como para sincronización condicional, haciendo el código difícil de mantener y verificar.

2.5 Introducción a los Monitores como Mecanismo de Alto Nivel

Para superar las deficiencias de los semáforos, C.A.R. Hoare (1974) y Per Brinch Hansen diseñaron los **Monitores**: un mecanismo de sincronización estructurado de alto nivel integrado en el propio lenguaje de programación.

Definición: Monitor

Un monitor es un Tipo de Dato Abstracto (TDA) concurrente que encapsula:

1. Las variables compartidas que definen el estado del recurso protegido (variables permanentes privadas).
2. Un conjunto de procedimientos públicos accesibles por los procesos externos.
3. Un bloque de inicialización de las variables internas.

Regla de Oro del Monitor: El propio compilador o entorno de ejecución garantiza de forma automática e implícita la **exclusión mutua** en el acceso a sus procedimientos. En ningún momento puede haber más de un proceso ejecutándose activamente dentro del monitor.

2.5.1 Estructura Interna de un Monitor

Todo monitor dispone de dos niveles de colas:

- **Cola de Entrada al Monitor:** Donde esperan los procesos que intentan invocar un procedimiento del monitor mientras otro proceso ya está dentro de él.
- **Variables Condición (`cond`):** Colas internas donde se suspenden los procesos que ya están dentro del monitor pero que no pueden continuar porque no se cumple una condición lógica sobre las variables del monitor.

2.5.2 Variables Condición y sus Primitivas

Una variable de tipo `cond` (o `condition_variable`) no guarda un valor booleano ni numérico; representa exclusivamente una **cola de espera**. Sobre una variable condición c se definen dos operaciones básicas:

- `c.wait()`: El proceso que la invoca se bloquea en la cola de c y **libera automáticamente la exclusión mutua del monitor**, permitiendo que otro proceso entre a modificar el estado.
- `c.signal()`: Despierta a uno de los procesos suspendidos en la cola de c. Si no hay ningún proceso esperando en c, la llamada a `signal()` no tiene efecto (se pierde, a diferencia de un semáforo que incrementaría su valor).

2.6 Semánticas de Señalización en Monitores

El dilema fundamental en el diseño de un monitor ocurre en el instante exacto en que un proceso P dentro del monitor ejecuta `c.signal()` y despierta al proceso suspendido Q. En ese instante habría **dos procesos dentro del monitor** (P y Q), lo que violaría la exclusión mutua implícita.

Para resolver este conflicto, existen cuatro semánticas principales:

Semántica	**Acción sobre el señalador (P)**	**Acción sobre el despertado (Q)**	**¿Desplazante?**
SU (Señalar y Espera Urgente)	Pasa a una cola prioritaria de procesos urgentes.	Entra inmediatamente al monitor y reanuda su ejecución.	**Sí** (Hoare)
SC (Señalar y Continuar)	Continúa ejecutando dentro del monitor hasta salir o esperar.	Pasa a la cola general de entrada o lista de listos.	**No** (Mesa/Java)
SS (Señalar y Salir)	Debe salir del monitor en ese mismo instante.	Entra inmediatamente al monitor.	**Sí**
SE (Señalar y Esperar)	Pasa a la cola ordinaria de entrada.	Reanuda la ejecución dentro del monitor.	**Sí**

Tabla 2.1: Comparativa de semánticas de señalización en monitores.

Importante / Ojo Examen: La Regla de Oro: `if` vs. `while` al evaluar la condición

- En la semántica **SU (Hoare)**, el proceso despertado se ejecuta de inmediato sin que nadie modifique el estado entre la señal y su reanudación. Por tanto, basta con comprobar la condición con un condicional:

```
1    if (!condicion_cumplida) c.wait(); // Valido en semantica SU (
         Hoare)
```

- En la semántica **SC (Mesa / Hansen / C++ / Java)**, otros procesos pueden intercalarse y consumir el recurso antes de que el proceso despertado consiga retomar el monitor. Por ello, **es estrictamente obligatorio reevaluar la condición en un bucle `while`**:

```
1    while (!condicion_cumplida) c.wait(); // OBLIGATORIO en semantica
         SC
```

2.7 Implementación de Monitores usando Semáforos

Para entender cómo el sistema operativo o el runtime de un lenguaje materializa un monitor (con semántica SU de Hoare), se utiliza la construcción clásica de C.A.R. Hoare basada en semáforos:

Algoritmo / Protocolo: Esquema de Hoare para Implementar Monitores con Semáforos

Se definen las siguientes variables de sincronización internas:

```
sem_t mutex;        // Controla la entrada al monitor (inicializado a 1)
sem_t urgente;      // Cola para procesos que cedieron el monitor por
    signal (inic. a 0)
int num_urgentes = 0; // Contador de procesos en la cola urgente

// Por cada variable condicion c:
sem_t c_sem;        // Cola del semaforo para c.wait() (inic. a 0)
int c_count = 0;    // Contador de procesos esperando en c
```

1. Entrada y Salida de cada Procedimiento del Monitor:

```
void Procedimiento_Monitor() {
    // ENTRADA AL PROCEDIMIENTO:
    sem_wait(&mutex);

    // ... Codigo del procedimiento ...

    // SALIDA DEL PROCEDIMIENTO:
    // Si hay procesos en la cola urgente, tienen prioridad sobre los
        de entrada:
    if (num_urgentes > 0) {
        sem_post(&urgente);
    } else {
        sem_post(&mutex);
    }
}
```

2. Implementación de `c.wait()`:

```
void wait_c() {
    c_count++;
    // Si habia procesos urgentes, cede a uno urgente; si no, libera la
        entrada:
    if (num_urgentes > 0) {
        sem_post(&urgente);
    } else {
        sem_post(&mutex);
    }
    sem_wait(&c_sem); // Se bloquea en la condicion
    c_count--;
}
```

3. Implementación de `c.signal()` (Semántica SU):

```
void signal_c() {
    if (c_count > 0) {
        num_urgentes++;
        sem_post(&c_sem);     // Despierta al proceso en la condicion
        sem_wait(&urgente);   // El proceso senalador se bloquea en
            urgentes
        num_urgentes--;
    }
}
```

Este esquema demuestra de forma transparente cómo se garantiza tanto la exclusión mutua del monitor como la prioridad absoluta de la semántica urgente de Hoare.

Tema 3

Sincronización en Memoria Compartida (Parte II): Monitores y Patrones Clásicos

Metadatos del Tema y Fuentes:

Fuentes utilizadas: Apuntes de Ismael Sallami y Temario de Infomates (LosDelDGIIM). Pendiente de incorporar transparencias originales del profesor.

3.1 Monitores como Mecanismo de Alto Nivel

3.1.1 Limitaciones de los Semáforos y Motivación

En la primera parte de este tema se estudiaron los semáforos como primitiva clásica de sincronización en memoria compartida. Aunque los semáforos representan un avance fundamental frente a los algoritmos de espera activa (como Dekker, Dijkstra o Peterson), presentan serios inconvenientes en el desarrollo de software concurrente a media y gran escala:

1. **Falta de encapsulación y modularidad:** Los semáforos son variables globales que deben ser compartidas por todos los hilos del sistema. El estado de la sincronización y los datos compartidos residen separados, rompiendo los principios de la programación estructurada y orientada a objetos.

2. **Operaciones dispersas en el código:** Las operaciones de sincronización (`sem_wait` y `sem_post`) se encuentran dispersas por los cuerpos de las diferentes funciones de los hilos. No existe una localización centralizada de la política de sincronización.

3. **Vulnerabilidad ante errores de programación:** Olvidar una llamada a `signal`, intercambiar el orden de dos llamadas a `wait`, o invocar una operación en el lugar equivocado conduce de forma casi inmediata a situaciones de interbloqueo (*deadlock*), violación de la exclusión mutua o inanición (*starvation*). La depuración de estos fallos es extremadamente compleja debido a su naturaleza no determinista.

Importante / Ojo Examen: El Peligro del Código Disperso

Con semáforos, la corrección del programa depende de que **todos** los hilos colaboren de forma impecable. Si un solo hilo de un sistema de miles de líneas comete un error al manipular un semáforo compartido, el sistema entero colapsa.

Por estas razones, la comunidad de ciencias de la computación buscó un mecanismo estructurado que encapsulara los datos protegidos y sus operaciones de sincronización en un único módulo seguro.

3.1.2 Definición Formal de Monitor

El concepto de **Monitor** fue propuesto por C.A.R. Hoare en 1974 y consolidado experimentalmente por Per Brinch Hansen en el diseño del lenguaje Concurrent Pascal.

Definición: Monitor

Un **Monitor** es un Tipo de Dato Abstracto (TDA) concurrente que encapsula:

- Un conjunto de **variables privadas** que representan el estado interno del recurso compartido.
- Un conjunto de **procedimientos públicos** (métodos) mediante los cuales los procesos concurrentes pueden interactuar con los datos.
- Un **código de inicialización** que establece el estado de arranque de las variables antes de que cualquier proceso acceda al monitor.
- Una política de **exclusión mutua implícita** garantizada por el compilador y el entorno de ejecución.

3.1.3 Exclusión Mutua Implícita y Reentrancia

La característica esencial que distingue a un monitor de una clase u objeto secuencial convencional es la **exclusión mutua implícita**:

- En cualquier instante de tiempo, **a lo sumo un proceso o hilo puede estar ejecutando activamente dentro de cualquiera de los procedimientos del monitor**.
- Si un proceso invoca un procedimiento del monitor mientras otro proceso se encuentra ejecutando dentro de él, el proceso invocador queda automáticamente suspendido en una **cola de entrada al monitor** gestionada por el runtime.
- Cuando el proceso activo abandona el procedimiento (o cede el control voluntariamente), el sistema operativo o runtime selecciona automáticamente a uno de los procesos de la cola de entrada para que tome posesión del monitor.

Importante / Ojo Examen: Exclusión Mutua Automática

A diferencia de los semáforos o cerrojos explícitos (*mutex*), el programador **no tiene que invocar explícitamente** operaciones de bloqueo al entrar ni de desbloqueo al salir. La exclusión mutua es una propiedad estructural garantizada por la arquitectura del monitor.

Para que esta propiedad funcione correctamente en entornos multiprocesador, los procedimientos del monitor deben ser **reentrantes**: el código ejecutable debe residir en memoria compartida de solo lectura y las variables locales de cada procedimiento deben residir en la pila privada (*stack*) de cada hilo que lo invoca, mientras que las variables permanentes del monitor residen en memoria estática o en el montón (*heap*) común.

3.2 Mecanismos de Sincronización Condicional: Variables Condición

3.2.1 Por qué la Exclusión Mutua no es Suficiente

La exclusión mutua garantiza que dos procesos no interfieran destructivamente al leer o escribir variables compartidas. Sin embargo, en la mayoría de problemas de sincronización, un proceso necesita esperar a que el estado del sistema satisfaga una **condición lógica booleana** antes de poder proceder.

Ejemplo Práctico: La Necesidad de Espera Condicional

Pensemos en un búfer compartido. Un consumidor que entra en el monitor en exclusión mutua puede encontrarse con que el búfer está vacío. No puede extraer nada. Si simplemente ejecuta un bucle activo comprobando si el búfer se llena:

$$\text{while } (\text{tam} == 0)\{/* \text{ espera activa } */\}$$

cometerá un error catastrófico: como él retiene la exclusión mutua del monitor, **ningún productor podrá entrar al monitor para insertar un dato**. El sistema queda irremediablemente interbloqueado.

Por tanto, el monitor necesita una facilidad que permita a un proceso activo **suspenderse temporalmente y liberar de forma atómica la exclusión mutua del monitor**, permitiendo que otros procesos entren, modifiquen el estado y eventualmente le avisen cuando la condición se cumpla.

3.2.2 El Tipo de Dato Abstracto `cond`

Para resolver esta necesidad, Hoare introdujo las **variables de condición** (representadas habitualmente como variables de tipo `cond` o `condition_variable`).

Definición: Variable Condición

Una variable de condición es una cola abstracta de procesos bloqueados asociada a una condición lógica del monitor. Sobre una variable de condición c se definen exclusivamente dos operaciones fundamentales:

1. `c.wait()`: El proceso invocador suspende su ejecución, cede de forma atómica la exclusión mutua del monitor y se coloca a la cola de espera de la variable c.
2. `c.signal()` (o `c.notify_one()`): Si existen procesos esperando en la cola de c, despierta a uno de ellos. Si no hay ningún proceso esperando en c, la operación **no tiene ningún efecto y se descarta**.

Adicionalmente, se define la operación `c.signal_all()` (o `c.notify_all()`), que despierta a todos los procesos actualmente bloqueados en la cola de c.

3.2.3 Diferencias Críticas entre Variables Condición y Semáforos

Es uno de los errores conceptuales más frecuentes en exámenes confundir una variable condición con un semáforo binario o contador. La siguiente tabla clarifica las diferencias intrínsecas:

Característica	Semáforo	Variable Condición (`cond`)
Memoria / Contador	**Sí tiene memoria.** Posee un contador entero ($S \geq 0$). Un `signal` incrementa el contador aunque no haya nadie esperando.	**No tiene memoria.** Si se ejecuta `c.signal()` y la cola de c está vacía, la señal **se pierde para siempre**.
Efecto de Wait	Puede no bloquear si el contador $S > 0$.	**Siempre bloquea** incondicionalmente al proceso que lo invoca.
Contexto de ejecución	Se puede invocar desde cualquier parte del código global.	Solo tiene sentido y validez sintáctica **dentro de un procedimiento del monitor**.
Gestión de exclusión	El programador debe coordinar manualmente la exclusión mutua.	`c.wait()` **libera automáticamente** la exclusión mutua del monitor y la readquiere al despertar.

Tabla 3.1: Comparativa estricta entre Semáforos y Variables de Condición.

3.3 Semánticas de Señalización: Hoare vs. Mesa / Hansen

3.3.1 El Gran Dilema de la Señalización

Supongamos que un proceso Q está bloqueado en una variable condición c esperando que se cumpla la propiedad lógica B. En un instante dado, otro proceso P entra al monitor, ejecuta código que hace verdadera la propiedad B, e invoca la instrucción:

$$c.\text{signal}();$$

En este instante surge un dilema fundamental:

- El proceso Q acaba de ser despertado y está listo para continuar ejecutando dentro del monitor.
- Pero el proceso P también está dentro del monitor y aún tiene instrucciones pendientes tras el `signal`.
- **Conflicto de exclusión mutua:** Dos procesos no pueden estar simultáneamente activos dentro del monitor. Uno de los dos debe ceder el paso.

La decisión de cuál de los dos procesos ejecuta inmediatamente y qué sucede con el otro da lugar a las diferentes **semánticas de señalización**.

3.3.2 Semántica de Hoare (Señalar y Espera Urgente - SU)

En la semántica original propuesta por Hoare (1974), también conocida como **Signal-and-Urgent-Wait (SU)**:

1. El proceso señalador P **cede inmediatamente la posesión del monitor** al proceso recién despertado Q.
2. P queda suspendido en una cola especial de alta prioridad denominada **Cola de Espera Urgente** (*Urgent Queue*).
3. El proceso despertado Q toma el control del monitor de manera instantánea, sin que ningún otro proceso pueda intercalarse entre el `signal` de P y la reanudación de Q.
4. Cuando Q sale del monitor (o hace un nuevo wait), los procesos de la cola urgente tienen prioridad absoluta sobre los procesos que esperan en la cola de entrada del monitor.

Importante / Ojo Examen: La Consecuencia Lógica de la Semántica de Hoare

Bajo la semántica de Hoare, cuando Q despierta tras `c.wait()`, tiene la certeza matemática de que la condición B que provocó su señalización **sigue siendo exactamente verdadera**. Por tanto, en semántica de Hoare es formalmente correcto utilizar una sentencia condicional simple:

```
if (!condicion) {
    c.wait();
}
// Al llegar aqui, la condicion es verdadera garantizada
```

3.3.3 Semántica de Mesa / Hansen (Señalar y Continuar - SC)

Desarrollada en el Centro de Investigación de Xerox PARC para el lenguaje Mesa (Lampson y Redell, 1980) y adoptada por Per Brinch Hansen:

1. El proceso señalador P **continúa su ejecución dentro del monitor** hasta completar su procedimiento o realizar un `wait`.
2. El proceso despertado Q no toma el monitor inmediatamente: simplemente pasa de la cola de la variable de condición a la cola de listos (o a la cola de entrada ordinaria del monitor).
3. Cuando P finalmente abandona el monitor, Q debe competir con los demás procesos de la cola de entrada para volver a adquirir la exclusión mutua.

3.3.4 Por Qué Cambia el `if` por `while` (La Regla de Oro)

La semántica Señalar y Continuar (SC) es la que implementan los lenguajes industriales modernos:

- C++11 (`std::condition_variable`)
- Java (`synchronized`, `wait()`, `notify()`)
- POSIX Threads (`pthread_cond_wait`, `pthread_cond_signal`)
- C# (`Monitor.Wait`, `Monitor.Pulse`)

En esta semántica surge el fenómeno de **robo de condición** (*condition stealing*) y los **despertares espurios** (*spurious wakeups*):

Ejemplo Práctico: Robo de la Condición en Semántica SC

Consideremos un búfer de tamaño 1 con un elemento disponible:

1. Consumidor C_1 ve el búfer vacío y se bloquea en `puede_leer.wait()`.
2. Productor P inserta un elemento en el búfer y ejecuta `puede_leer.signal()`. Con semántica SC, P no se suspende; sigue ejecutando y sale del monitor. El consumidor C_1 es movido a la cola de entrada.
3. Justo antes de que el planificador de la CPU le conceda el turno a C_1, entra un segundo consumidor C_2 recién llegado desde la cola de entrada, comprueba que el búfer está lleno, extrae el dato y sale del monitor.
4. Ahora, el planificador le da el turno a C_1. Si C_1 hubiera utilizado un `if`:

```
    if (tam == 0) puede_leer.wait();
    dato = buffer[0]; // ERROR: Buffer vacio! Segment violation o
        dato invalido
```

¡C_1 intentaría extraer de un búfer que ya fue vaciado por C_2!

Definición: La Regla de Oro de la Sincronización en Monitores SC

En semántica Señalar y Continuar (SC), todo `wait()` debe estar obligatoriamente envuelto en un bucle `while`:

```
while (!condicion) {
    c.wait(lock);
}
// Al salir del bucle, se ha REEVALUADO y confirmado que la condicion
    es cierta
```

De este modo, cuando el proceso despierta, vuelve a evaluar la condición. Si otro proceso robó el recurso o si ocurrió un despertar espurio, vuelve a dormirse de forma segura.

3.3.5 Comparativa de Semánticas de Señalización

Semántica	¿Quién ejecuta tras signal?	Destino del señalador	Estructura de comprobación
SU (Hoare)	Proceso señalado (Q)	Pasa a Cola de Espera Urgente	`if (!cond) wait();`
SC (Mesa/Hansen)	Proceso señalador (P)	Continúa en el monitor	`while (!cond) wait();`
SS (Signal-Exit)	Proceso señalado (Q)	El señalador debe salir inmediatamente (`signal` es la última instrucción)	`if (!cond) wait();`
SE (Signal-Wait)	Proceso señalado (Q)	El señalador pasa a la cola de entrada general	`while (!cond) wait();`

Tabla 3.2: Comparativa rigurosa de las cuatro semánticas clásicas de señalización.

3.4 Patrones Clásicos de Concurrencia con Monitores

A continuación se analizan, deducen formalmente y resuelven en C++11 los cinco problemas canónicos de la programación concurrente utilizando la arquitectura de monitores con semántica SC (Mesa/Hansen).

3.4.1 Patrón 1: Productor-Consumidor con Búfer Acotado

Planteamiento e Invariante de Estado

Uno o varios procesos productores generan datos y los almacenan en un búfer circular de capacidad fija N. Simultáneamente, uno o varios procesos consumidores extraen los datos en el mismo orden (política FIFO).

- **Condición de parada del Productor:** No se puede escribir si el búfer está lleno (num_items == N).
- **Condición de parada del Consumidor:** No se puede leer si el búfer está vacío (num_items == 0).
- **Invariante de Seguridad:**

  $\mathcal{I}_{PC} \equiv (0 \leq \text{num\_items} \leq N) \wedge (\text{primera\_ocupada} \in [0, N-1]) \wedge (\text{num\_items} = (\text{primera\_libre} - \text{primera\_o}$

Implementación en C++11 con Monitor

```cpp
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <vector>

template <typename T, size_t CAPACIDAD>
class MonitorBufferFIFO {
private:
    T buffer[CAPACIDAD];
    size_t primera_ocupada;
    size_t primera_libre;
    size_t num_items;

    std::mutex cerrojo_monitor;
    std::condition_variable puede_escribir;
    std::condition_variable puede_leer;

public:
    MonitorBufferFIFO() : primera_ocupada(0), primera_libre(0), num_items(0)
        {}

    void escribir(const T& valor) {
        std::unique_lock<std::mutex> lock(cerrojo_monitor);

        // Semantica SC: Obligatorio 'while' para verificar la condicion
        while (num_items == CAPACIDAD) {
            puede_escribir.wait(lock);
        }

        // Seccion critica: insertar en posicion circular
        buffer[primera_libre] = valor;
        primera_libre = (primera_libre + 1) % CAPACIDAD;
        num_items++;

        // Notificar al consumidor que ahora seguro hay al menos un elemento
        puede_leer.notify_one();
    }

    T leer() {
        std::unique_lock<std::mutex> lock(cerrojo_monitor);

        while (num_items == 0) {
            puede_leer.wait(lock);
        }

        T valor = buffer[primera_ocupada];
        primera_ocupada = (primera_ocupada + 1) % CAPACIDAD;
        num_items--;

        // Notificar al productor que ahora seguro hay al menos un hueco
            libre
```

```
        puede_escribir.notify_one();
        return valor;
    }
};
```

Listing 3.1: Monitor Productor-Consumidor con Búfer Acotado

Análisis de Funcionamiento

La exclusión mutua está garantizada por el cerrojo RAII `std::unique_lock`. Cuando el productor se bloquea en `puede_escribir.wait(lock)`, libera atómicamente el cerrojo, permitiendo que un consumidor entre y ejecute `leer()`, el cual decrementa `num_items` y avisa al productor con `puede_escribir.notify_one()`.

3.4.2 Patrón 2: Lectores-Escritores

Planteamiento del Problema

Varios procesos acceden a una estructura de datos compartida (archivo, base de datos). Existen dos tipos de procesos:

- **Lectores:** Pueden leer simultáneamente el recurso sin interferirse entre sí.
- **Escritores:** Modifican el recurso. Requieren exclusión mutua absoluta: ningún lector ni ningún otro escritor puede acceder mientras un escritor está activo.

El invariante de seguridad fundamental para cualquier variante es:

$$\mathcal{I}_{LE} \equiv (n\_lectores \geq 0) \wedge (n\_escritores \in \{0,1\}) \wedge (n\_escritores == 1 \implies n\_lectores == 0)$$

Variante 1: Prioridad a los Lectores

Si un lector está leyendo y llega otro lector, este último entra directamente sin esperar, aunque haya escritores esperando. Los escritores solo entran si no hay ningún lector en la sala.

Importante / Ojo Examen: Inanición de Escritores

La prioridad a los lectores puede provocar **inanición total** (*starvation*) de los escritores si el flujo de lectores es continuo.

```
class MonitorLectoresEscritores_PrioridadLectores {
private:
    int n_lectores = 0;
    bool escribiendo = false;
    std::mutex cerrojo;
    std::condition_variable puede_leer;
    std::condition_variable puede_escribir;

public:
    void iniciar_lectura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        // Esperar mientras haya un escritor activo
        while (escribiendo) {
            puede_leer.wait(lock);
        }
        n_lectores++;
        // Con prioridad lectores, despertamos en cadena a otros posibles
            lectores
```

```
        puede_leer.notify_one();
    }

    void fin_lectura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        n_lectores--;
        if (n_lectores == 0) {
            // El ultimo lector en salir despierta a un escritor
            puede_escribir.notify_one();
        }
    }

    void iniciar_escritura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        // Esperar mientras haya lectores o alguien escribiendo
        while (n_lectores > 0 || escribiendo) {
            puede_escribir.wait(lock);
        }
        escribiendo = true;
    }

    void fin_escritura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        escribiendo = false;
        // Prioridad lectores: intentar despertar lectores primero
        // Si no hay lectores esperando, se despertara un escritor
        puede_leer.notify_one();
        puede_escribir.notify_one();
    }
};
```

Listing 3.2: Monitor Lectores-Escritores con Prioridad a Lectores

Variante 2: Prioridad a los Escritores

Para proteger a los escritores, si un escritor llega y desea acceder, se bloquea la entrada a los **nuevos lectores que vayan llegando**. Los lectores ya activos terminan su lectura, tras lo cual el escritor entra de inmediato.

```
class MonitorLectoresEscritores_PrioridadEscritores {
private:
    int n_lectores = 0;
    bool escribiendo = false;
    int escritores_esperando = 0;
    std::mutex cerrojo;
    std::condition_variable puede_leer;
    std::condition_variable puede_escribir;

public:
    void iniciar_lectura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        // Bloquear si hay un escritor activo O si hay escritores esperando
        while (escribiendo || escritores_esperando > 0) {
            puede_leer.wait(lock);
        }
        n_lectores++;
    }

    void fin_lectura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        n_lectores--;
        if (n_lectores == 0) {
            puede_escribir.notify_one();
```

```
        }
    }

    void iniciar_escritura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        escritores_esperando++;
        while (n_lectores > 0 || escribiendo) {
            puede_escribir.wait(lock);
        }
        escritores_esperando--;
        escribiendo = true;
    }

    void fin_escritura() {
        std::unique_lock<std::mutex> lock(cerrojo);
        escribiendo = false;
        if (escritores_esperando > 0) {
            puede_escribir.notify_one(); // Siguiente escritor prioritario
        } else {
            puede_leer.notify_all();     // No hay escritores: pasar a todos
                los lectores
        }
    }
};
```

Listing 3.3: Monitor Lectores-Escritores con Prioridad a Escritores

Variante 3: Solución Equitativa (Sin Inanición)

En la variante equitativa se utiliza un esquema de turnos o FIFO donde las peticiones se atienden en orden estricto de llegada, impidiendo que grupos continuos de lectores o escritores monopolicen el recurso.

3.4.3 Patrón 3: La Cena de los Filósofos

Planteamiento e Interbloqueo

Cinco filósofos se sientan alrededor de una mesa redonda alternando entre dos actividades: pensar y comer espaguetis. Entre cada par de filósofos adyacentes hay un único tenedor (en total 5 tenedores). Para comer, cada filósofo necesita tomar simultáneamente los dos tenedores adyacentes: el izquierdo (i) y el derecho ($(i+1) \pmod 5$).

Importante / Ojo Examen: Condición de Interbloqueo Canónico

Si los cinco filósofos tienen hambre al mismo tiempo y cada uno toma simultáneamente su tenedor izquierdo, todos quedan reteniendo un recurso y esperando indefinidamente por el tenedor derecho que retiene su vecino. Se produce una espera circular: **Interbloqueo total (Deadlock)**.

Estrategias de Solución

1. **Estrategia Asimétrica (Sin Camarero):** Romper la simetría circular haciendo que los filósofos pares tomen primero el tenedor izquierdo y luego el derecho, mientras que los filósofos impares tomen primero el derecho y luego el izquierdo.

2. **Estrategia con Camarero (Limitación de Aforo):** Un árbitro o camarero monitoriza la mesa y solo permite que se sienten simultáneamente a la mesa a lo sumo $N-1$ filósofos

(en este caso 4). Por el Principio del Palomar, si hay 4 filósofos y 5 tenedores, al menos uno de ellos podrá tomar ambos tenedores y comer, rompiendo la espera circular.

3. **Estrategia de Asignación Atómica en Monitor:** El filósofo solicita ambos tenedores a la vez dentro de un monitor. Si los dos no están disponibles, no toma ninguno y espera en una variable condición propia.

Implementación del Monitor con Asignación Atómica

```
class MonitorFilosofos {
private:
    static const int N = 5;
    enum Estado { PENSANDO, HAMBRIENTO, COMIENDO };
    Estado estado[N];
    std::condition_variable puede_comer[N];
    std::mutex cerrojo;

    int izq(int i) { return (i + N - 1) % N; }
    int der(int i) { return (i + 1) % N; }

    void comprobar(int i) {
        // Solo puede comer si esta hambriento y ningun vecino esta comiendo
        if (estado[i] == HAMBRIENTO &&
            estado[izq(i)] != COMIENDO &&
            estado[der(i)] != COMIENDO)
        {
            estado[i] = COMIENDO;
            puede_comer[i].notify_one();
        }
    }

public:
    MonitorFilosofos() {
        for (int i = 0; i < N; i++) estado[i] = PENSANDO;
    }

    void coger_tenedores(int i) {
        std::unique_lock<std::mutex> lock(cerrojo);
        estado[i] = HAMBRIENTO;
        comprobar(i);
        while (estado[i] != COMIENDO) {
            puede_comer[i].wait(lock);
        }
    }

    void dejar_tenedores(int i) {
        std::unique_lock<std::mutex> lock(cerrojo);
        estado[i] = PENSANDO;
        // Al terminar, comprueba si sus dos vecinos ahora pueden comer
        comprobar(izq(i));
        comprobar(der(i));
    }
};
```

Listing 3.4: Monitor de la Cena de los Filósofos

3.4.4 Patrón 4: El Estanco y los Fumadores

Planteamiento del Problema (Patil-Dijkstra)

En un estanco hay un estanquero y tres fumadores. Para liar un cigarrillo se necesitan tres ingredientes:

- Ingrediente 0: **Tabaco**
- Ingrediente 1: **Papel**
- Ingrediente 2: **Cerillas**

Cada fumador posee infinitos suministros de uno de los ingredientes: el fumador 0 posee tabaco, el 1 posee papel y el 2 posee cerillas. El estanquero coloca en el mostrador **dos ingredientes distintos** de forma aleatoria. El fumador que posee el ingrediente faltante debe recoger los dos suministros del mostrador, liar su cigarrillo, fumar y avisar al estanquero para que vuelva a colocar ingredientes.

Implementación del Monitor Estanco

```
class MonitorEstanco {
private:
    int mostrador; // -1: vacio, 0: falta tabaco (hay papel y cerillas),
                   // 1: falta papel, 2: falta cerillas
    std::mutex cerrojo;
    std::condition_variable puede_poner;
    std::condition_variable fumador_puede_fumar[3];

public:
    MonitorEstanco() : mostrador(-1) {}

    // Invocado por el fumador 'i' (0 <= i < 3)
    void obtenerIngrediente(int i) {
        std::unique_lock<std::mutex> lock(cerrojo);
        // El fumador 'i' solo puede actuar si en el mostrador falta su
            ingrediente
        while (mostrador != i) {
            fumador_puede_fumar[i].wait(lock);
        }
        // Retira los ingredientes del mostrador
        mostrador = -1;
        // Notifica al estanquero de que el mostrador esta libre
        puede_poner.notify_one();
    }

    // Invocado por el estanquero generando el ingrediente faltante 'i'
    void ponerIngrediente(int i) {
        std::unique_lock<std::mutex> lock(cerrojo);
        while (mostrador != -1) {
            puede_poner.wait(lock);
        }
        mostrador = i;
        // Despierta exclusivamente al fumador que necesita los ingredientes
            puestos
        fumador_puede_fumar[i].notify_one();
    }
};
```

Listing 3.5: Monitor para el Problema de los Fumadores del Estanco

3.4.5 Patrón 5: El Barbero Durmiente

Planteamiento del Problema

Una barbería consta de una sala con una silla de barbero y una sala de espera con N sillas.

- Si no hay clientes en la barbería, el barbero se sienta en su silla y se duerme.
- Cuando llega un cliente:
  - Si el barbero duerme, el cliente lo despierta y se sienta en la silla de pelar.
  - Si el barbero está pelando a otro y hay sillas libres en la sala de espera, el cliente se sienta a esperar su turno.
  - Si todas las N sillas están ocupadas, el cliente se marcha inmediatamente sin esperar (*balking*).
- Cuando el barbero termina un corte, despide al cliente, cobra, y si hay clientes en la sala de espera, llama al siguiente; si no hay nadie, vuelve a dormirse.

Implementación del Monitor de la Barbería

```
1  class MonitorBarberia {
2  private:
3      const int max_sillas_espera;
4      int clientes_esperando;
5      bool barbero_dormido;
6      bool silla_barbero_ocupada;
7      bool corte_terminado;
8
9      std::mutex cerrojo;
10     std::condition_variable cola_espera;
11     std::condition_variable despertar_barbero;
12     std::condition_variable silla_pelar;
13     std::condition_variable cliente_espera_fin_corte;
14
15 public:
16     MonitorBarberia(int n_sillas) :
17         max_sillas_espera(n_sillas), clientes_esperando(0),
18         barbero_dormido(true), silla_barbero_ocupada(false), corte_terminado(
               false) {}
19
20     bool cortarPelo(int id_cliente) {
21         std::unique_lock<std::mutex> lock(cerrojo);
22
23         if (clientes_esperando == max_sillas_espera && silla_barbero_ocupada)
                {
24             // No hay sitio: se marcha
25             return false;
26         }
27
28         clientes_esperando++;
29
30         // Si el barbero duerme, lo despertamos
31         if (barbero_dormido) {
32             barbero_dormido = false;
33             despertar_barbero.notify_one();
34         }
35
36         // Esperar a que la silla de pelar quede libre
37         while (silla_barbero_ocupada) {
```

```
38              cola_espera.wait(lock);
39          }
40
41          clientes_esperando--;
42          silla_barbero_ocupada = true;
43          corte_terminado = false;
44
45          // Esperar a que el barbero termine de cortar el pelo
46          while (!corte_terminado) {
47              cliente_espera_fin_corte.wait(lock);
48          }
49
50          silla_barbero_ocupada = false;
51          // Avisar al siguiente cliente en espera
52          cola_espera.notify_one();
53          return true;
54      }
55
56      void siguienteCliente() {
57          std::unique_lock<std::mutex> lock(cerrojo);
58
59          // Si no hay nadie esperando ni en la silla, dormir
60          while (clientes_esperando == 0 && !silla_barbero_ocupada) {
61              barbero_dormido = true;
62              despertar_barbero.wait(lock);
63          }
64      }
65
66      void finCliente() {
67          std::unique_lock<std::mutex> lock(cerrojo);
68          corte_terminado = true;
69          cliente_espera_fin_corte.notify_one();
70      }
71  };
```

Listing 3.6: Monitor del Barbero Durmiente

Importante / Ojo Examen: Sincronización Bilateral en el Barbero Durmiente

Nótese que en este patrón coexisten dos fases de sincronización distintas:

1. La sincronización de entrada (despertar al barbero y ocupar la silla de corte).
2. La sincronización de terminación (el cliente no puede marcharse hasta que el barbero complete el corte, y el barbero no puede llamar al siguiente hasta que el actual abandone la silla).

Tema 4

Sistemas Basados en el Paso de Mensajes y MPI

Metadatos del Tema y Fuentes:

Fuentes utilizadas: Apuntes de Ismael Sallami y Temario de Infomates (LosDelDGIIM). Pendiente de incorporar transparencias originales del profesor.

4.1 Fundamentos de Arquitecturas Paralelas y Distribuidas

4.1.1 El Cuello de Botella de Von Neumann

En la arquitectura clásica propuesta por John von Neumann en 1945, un computador digital consta de una Unidad Central de Procesamiento (CPU), una memoria principal que almacena tanto las instrucciones del programa como los datos, y un subsistema de entrada/salida.

Ambos flujos de información —las instrucciones a ejecutar y los operandos o datos leídos y escritos en memoria— deben transitar obligatoriamente a través del **mismo bus físico de interconexión**.

Definición: Cuello de Botella de Von Neumann

El **cuello de botella de Von Neumann** es la limitación física estructural del rendimiento de un computador debida a la contención en el bus compartido entre la CPU y la memoria. Puesto que el bus no puede transferir simultáneamente una instrucción y un dato, la velocidad de procesamiento queda restringida por la tasa de transferencia (*throughput*) y la latencia del bus, independientemente de la potencia de cálculo bruta de la CPU.

Con el paso de las décadas, la velocidad de conmutación de los transistores en las CPUs creció exponencialmente siguiendo la Ley de Moore, mientras que la latencia de acceso a la memoria DRAM mejoró a un ritmo significativamente más lento. Esta brecha creciente (conocida en la literatura como el *Memory Wall* o muro de la memoria) convirtió al acceso a memoria en el factor limitante absoluto del rendimiento secuencial.

La solución arquitectónica moderna a este estancamiento no radica en seguir acelerando un único procesador, sino en la **computación paralela y distribuida**: disponer de múltiples unidades de procesamiento que operen simultáneamente, descomponiendo el flujo único de cómputo en múltiples flujos concurrentes.

4.1.2 Clasificación de Flynn

En 1966, Michael J. Flynn propuso una taxonomía fundamental que clasifica las arquitecturas computacionales a partir de dos dimensiones independientes: el número de **flujos de instrucciones** (*Instruction Streams*) y el número de **flujos de datos** (*Data Streams*) que el hardware puede procesar simultáneamente.

	Un Solo Flujo de Datos (SD)	**Múltiples Flujos de Datos (MD)**
Un Solo Flujo de Instrucciones (SI)	**SISD** (*Single Instruction, Single Data*) Computador secuencial convencional (Von Neumann monoprocesador).	**SIMD** (*Single Instruction, Multiple Data*) Procesadores vectoriales, extensiones multimedia (AVX/SSE), GPUs masivas.
Múltiples Flujos de Instrucciones (MI)	**MISD** (*Multiple Instruction, Single Data*) Muy infrecuente. Utilizado en sistemas redundantes de alta tolerancia a fallos (aeroespacial).	**MIMD** (*Multiple Instruction, Multiple Data*) Multiprocesadores multinúcleo (*multicore*), supercomputadores y clusters distribuidos.

Figura 4.1: Matriz de la Clasificación de Flynn.

- **SISD:** Una única unidad de control emite un único flujo de instrucciones que operan secuencialmente sobre un único flujo de datos.
- **SIMD:** Una única unidad de control emite la misma instrucción en cada ciclo de reloj, pero esta se ejecuta en paralelo sobre múltiples elementos de datos independientes distribuidos en múltiples unidades aritmético-lógicas (ALUs).
- **MISD:** Múltiples unidades de control ejecutan instrucciones distintas sobre el mismo flujo de datos. Es un modelo marginal cuya aplicación principal es la verificación cruzada por redundancia en sistemas críticos (por ejemplo, tres algoritmos distintos calculando la trayectoria de un cohete con los mismos datos de sensores).
- **MIMD:** Múltiples procesadores autónomos ejecutan diferentes secuencias de instrucciones sobre flujos de datos distintos. Es la base de todos los sistemas distribuidos y paralelos actuales.

Importante / Ojo Examen: El Paradigma SPMD

Dentro de las arquitecturas MIMD, el modelo dominante de programación es **SPMD** (*Single Program, Multiple Data*): se escribe un único código fuente binario que se carga en todos los nodos de cómputo, pero cada proceso ejecuta bifurcaciones condicionales distintas en función de su identificador único (*rank*) y procesa una porción distinta del conjunto de datos.

4.1.3 Memoria Compartida vs. Memoria Distribuida (SMP vs. AMP)

En función de cómo está interconectada físicamente la memoria con los procesadores, los sistemas MIMD se dividen en dos categorías arquitectónicas:

Definición: SMP - Multiprocesamiento Simétrico

En una arquitectura **SMP** (*Symmetric Multiprocessing*), todos los procesadores comparten un único espacio de direcciones de memoria física global a través de un bus común o una red de conmutación cruzada (*crossbar*). Todos los procesadores son funcionalmente idénticos y tienen tiempos de acceso uniformes a cualquier celda de memoria (arquitecturas UMA - *Uniform Memory Access*).

Definición: AMP / Memoria Distribuida / Clusters

En una arquitectura de **Memoria Distribuida** (o multiprocesamiento asimétrico / multicomputadores), cada nodo de procesamiento posee su propia memoria local privada. No existe un espacio de direccionamiento global compartido: un procesador no puede acceder directamente a las variables en la memoria física de otro nodo. La cooperación, compartición de datos y sincronización se realizan obligatoriamente mediante el **intercambio explícito de mensajes a través de una red de comunicaciones**.

Criterio	Memoria Compartida (SMP)	Memoria Distribuida (Clusters / AMP)
Espacio de memoria	Único y global a todos los procesadores.	Local y privado a cada nodo de cómputo.
Mecanismo de comunicación	Lectura y escritura en variables compartidas protegidas por cerrojos o monitores.	Envío y recepción explícita de mensajes (`send` / `receive`) por red.
Escalabilidad	Muy limitada (máximo decenas de núcleos) por contención del bus y coherencia de caché.	Prácticamente ilimitada (cientos de miles de nodos en supercomputadores Top500).
Coste de hardware	Alto para sistemas grandes (placas base y conmutadores de memoria complejos).	Muy bajo por nodo (posibilidad de usar ordenadores comerciales *Commodity / Beowulf*).
Complejidad de software	Sincronización delicada (condiciones de carrera, interbloqueos de memoria).	Obliga a particionar los datos y gestionar la topología de red explícitamente.

Tabla 4.1: Comparativa entre arquitecturas SMP y Memoria Distribuida.

4.2 Mecanismos y Semántica del Paso de Mensajes

4.2.1 Primitivas Fundamentales: Send y Receive

En un sistema distribuido sin memoria compartida, toda interacción entre dos procesos P y Q se reduce a dos primitivas elementales:

- `send(destino, mensaje)`: El proceso emisor solicita el envío de un paquete de datos hacia el proceso destino.
- `receive(origen, variable)`: El proceso receptor solicita la llegada de datos procedentes del proceso origen y los almacena en su memoria local.

La semántica de estas dos operaciones varía drásticamente en función de dos factores ortogonales:

1. Si la comunicación utiliza un **búfer intermedio** en el sistema de transporte.
2. Si las operaciones son **bloqueantes o no bloqueantes**.

4.2.2 Mecanismo de Citas (Rendez-vous) / Comunicación Síncrona sin Búfer

Propuesto por Hoare en el modelo CSP (*Communicating Sequential Processes*, 1978) y adoptado por el lenguaje Ada:

Definición: Mecanismo de Citas (*Rendez-vous*)

El **mecanismo de citas** es una comunicación síncrona, bloqueante y **sin búfer intermedio**. Tanto el emisor como el receptor deben coincidir en el tiempo para que la transferencia de datos tenga lugar:

- Si el emisor ejecuta `send` antes de que el receptor ejecute `receive`, el emisor se suspende en la cita hasta que el receptor llegue.
- Si el receptor ejecuta `receive` antes de que el emisor ejecute `send`, el receptor se suspende en la cita hasta que el emisor llegue.
- En el instante en que ambos procesos se encuentran en la cita, los datos se copian directamente del espacio de memoria del emisor al del receptor sin pasar por búferes intermedios del sistema operativo, y ambos procesos reanudan su ejecución simultáneamente.

4.2.3 Paso de Mensajes Bloqueante con Búfer

En este modelo, el sistema operativo o el hardware de red proporciona una cola o búfer de mensajes intermedio:

- El emisor ejecuta `send`. Si hay espacio libre en el búfer del sistema, deposita el mensaje y **continúa su ejecución inmediatamente sin esperar** a que el receptor haya leído el dato. El emisor solo se bloquea si el búfer del canal está completamente lleno.
- El receptor ejecuta `receive`. Si hay al menos un mensaje en el búfer, lo extrae y continúa; si el búfer está vacío, se bloquea hasta que llegue un mensaje.

4.2.4 Paso de Mensajes No Bloqueante

En aplicaciones de alto rendimiento, bloquear un proceso mientras se transmite un mensaje a través de una red lenta degrada notablemente el rendimiento. Las primitivas no bloqueantes resuelven este problema desacoplando la iniciación de la transferencia de su finalización:

- `asynchronous_send(destino, mensaje)`: La llamada regresa al instante de forma inmediata. El hardware de red (mediante DMA o hilos en segundo plano) transfiere los datos mientras la CPU del proceso sigue ejecutando cálculos útiles en paralelo.
- **Condición de peligro:** El proceso emisor no debe modificar el búfer del mensaje en memoria hasta que la transferencia subyacente haya concluido con certeza; de lo contrario, se enviarían datos corruptos.
- Para sincronizar y confirmar la finalización de la transferencia se utilizan operaciones de comprobación (`wait` o `test`).

4.3 El Estándar MPI (Message Passing Interface)

4.3.1 Filosofía y Conceptos Clave de MPI

MPI (*Message Passing Interface*) es el estándar industrial por excelencia para la programación paralela en arquitecturas de memoria distribuida. No es un lenguaje de programación nuevo, sino una biblioteca portable y estandarizada con interfaces para C, C++ y Fortran.

En MPI, los procesos se ejecutan bajo el modelo SPMD. Cada proceso se identifica mediante:

- **Comunicador (`MPI_Comm`):** Representa un grupo de procesos y define el contexto cerrado dentro del cual pueden intercambiar mensajes. El comunicador por defecto que engloba a todos los procesos lanzados al arrancar el programa es `MPI_COMM_WORLD`.
- **Rango (`rank`):** Un entero identificador único asignado a cada proceso dentro de un comunicador, en el rango $[0, \text{size} - 1]$.
- **Tamaño (`size`):** Número total de procesos que componen el comunicador.
- **Etiqueta (`tag`):** Un entero que clasifica el tipo o propósito del mensaje, permitiendo al receptor discriminar entre diferentes tipos de datos que provienen del mismo emisor.
- **Tipo de dato (`MPI_Datatype`):** MPI define tipos primitivos para garantizar la portabilidad entre arquitecturas con distinto orden de bytes (*endianness*): `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_BYTE`, etc.

4.3.2 Estructura Mínima de un Programa MPI

Todo programa paralelo con MPI debe seguir un ciclo de vida estrictamente regulado:

```cpp
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[]) {
    // 1. Inicializar el entorno de ejecucion MPI
    MPI_Init(&argc, &argv);

    // 2. Obtener el numero total de procesos
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // 3. Obtener el identificador unico (rank) de este proceso
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    std::cout << "Hola desde el proceso " << rank << " de un total de " <<
        size << std::endl;

    // 4. Finalizar el entorno MPI antes de terminar el programa
    MPI_Finalize();
    return 0;
}
```

Listing 4.1: Estructura canónica de un programa MPI en C++

4.3.3 Comunicaciones Punto a Punto Bloqueantes

Las dos funciones fundamentales para enviar y recibir datos entre dos procesos específicos son:

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);

int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status* status);
```

Semántica de Bloqueo en `MPI_Send` y `MPI_Recv`

- `MPI_Send`: Es bloqueante en el sentido de que **no retorna hasta que el búfer de memoria del emisor puede ser reutilizado de forma segura**. Esto ocurre cuando el mensaje ha sido transmitido a la red o copiado a un búfer interno del sistema. No garantiza necesariamente que el receptor haya recibido el dato.
- `MPI_Recv`: No retorna hasta que el mensaje completo ha llegado desde la red y ha sido copiado en el búfer receptor local `buf`.

Comodines y la Estructura `MPI_Status`

El receptor puede no conocer de antemano qué proceso le va a enviar datos o qué etiqueta tendrá el mensaje. Para ello, MPI proporciona comodines:

- `MPI_ANY_SOURCE`: Acepta un mensaje proveniente de cualquier proceso del comunicador.
- `MPI_ANY_TAG`: Acepta un mensaje con cualquier identificador de etiqueta.

Cuando se usan comodines, la información real del mensaje recibido se inspecciona a través del parámetro `MPI_Status`:

```
MPI_Status status;
int dato;
MPI_Recv(&dato, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
    status);

int emisor_real = status.MPI_SOURCE; // Quien envio el mensaje
int etiqueta_real = status.MPI_TAG;  // Que etiqueta tenia
int rec_count;
MPI_Get_count(&status, MPI_INT, &rec_count); // Cuantos enteros se recibieron
```

4.3.4 Comunicaciones No Bloqueantes: `MPI_Isend` e `MPI_Irecv`

Para evitar interbloqueos mutuos causados por envíos cruzados y para solapar cómputo matemático con latencias de red, MPI proporciona operaciones inmediatas (prefijo `I`):

```
int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request* request);

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request* request);
```

Ambas funciones retornan inmediatamente, devolviendo un manipulador de petición (`MPI_Request`). Para verificar y esperar la finalización de la transferencia se emplean:

- `MPI_Wait(&request, &status)`: Bloquea el proceso hasta que la operación asociada a `request` se ha completado definitivamente.
- `MPI_Test(&request, &flag, &status)`: Operación no bloqueante. Consulta el estado de la transferencia y asigna `flag = true` si la operación ha concluido, o `flag = false` si aún está en curso.

4.3.5 Sondeo de Mensajes: `MPI_Probe` e `MPI_Iprobe`

En ocasiones, un proceso necesita saber si hay mensajes pendientes en la red y qué tamaño tienen antes de asignar memoria y llamar a `MPI_Recv`:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status);
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int* flag, MPI_Status*
    status);
```

Definición: Sondeo de Mensajes (*Probing*)

`MPI_Probe` bloquea el proceso hasta que llega un mensaje que coincida con `source` y `tag`, rellenando la estructura `status` **sin extraer el mensaje de la cola de red**. A continuación, el proceso puede averiguar el tamaño exacto del mensaje con `MPI_Get_count`, reservar un vector del tamaño exacto mediante `malloc` o `std::vector`, e invocar finalmente a `MPI_Recv` para consumirlo limpiamente.

4.4 Algoritmos Prácticos Clásicos en MPI

4.4.1 Algoritmo 1: La Criba de Eratóstenes en Pipeline / Anillo

Fundamento Teórico del Algoritmo

La criba de Eratóstenes es el método clásico para encontrar números primos hasta un límite M. En un entorno distribuido con paso de mensajes, la criba se estructura de forma extraordinariamente elegante como una **tubería de procesos (pipeline)**:

- Cada proceso del pipeline representa un **filtro primo**.
- El proceso 0 (Generador) genera la secuencia de números impares candidatos: $3, 5, 7, 9, 11, 13, \ldots$ y los inyecta en el extremo inicial de la tubería.
- Cada proceso filtro i en la tubería almacena un número primo P_i. Cuando recibe un número candidato x:
  - Si x es divisible por P_i ($x \pmod{P_i} == 0$), el número queda descartado (es compuesto) y no se propaga más.
  - Si x no es divisible por P_i, el proceso i reenvía x hacia el siguiente proceso filtro $i+1$.
- Cuando un proceso filtro aún no tiene asignado su primo, el primer número que recibe se convierte en su número primo asignado.
- **Protocolo de Terminación:** Al finalizar la generación, el proceso 0 envía un testigo de terminación (una marca centinela, habitualmente el valor -1). Cada proceso filtro, al recibir el -1, imprime su número primo, lo propaga al siguiente proceso de la cadena y finaliza ordenadamente.

Implementación Completa en C++ / MPI

```
#include <mpi.h>
#include <iostream>

const int TAG_DATO = 0;
const int CENTINELA_FIN = -1;
const int LIMITE_MAX = 50;

```

```
8  void proceso_generador(int rank, int size) {
9      std::cout << "[Generador " << rank << "] Primo encontrado: 2" << std::
           endl;
10     for (int num = 3; num <= LIMITE_MAX; num += 2) {
11         MPI_Send(&num, 1, MPI_INT, rank + 1, TAG_DATO, MPI_COMM_WORLD);
12     }
13     // Enviar marca de terminacion
14     int fin = CENTINELA_FIN;
15     MPI_Send(&fin, 1, MPI_INT, rank + 1, TAG_DATO, MPI_COMM_WORLD);
16 }
17
18 void proceso_filtro(int rank, int size) {
19     int mi_primo = 0;
20     int numero_recibido;
21     MPI_Status status;
22
23     // El primer numero recibido que no sea centinela es mi primo
24     MPI_Recv(&numero_recibido, 1, MPI_INT, rank - 1, TAG_DATO, MPI_COMM_WORLD
           , &status);
25
26     if (numero_recibido == CENTINELA_FIN) {
27         if (rank + 1 < size) {
28             MPI_Send(&numero_recibido, 1, MPI_INT, rank + 1, TAG_DATO,
                   MPI_COMM_WORLD);
29         }
30         return;
31     }
32
33     mi_primo = numero_recibido;
34     std::cout << "[Filtro " << rank << "] Primo asignado: " << mi_primo <<
           std::endl;
35
36     while (true) {
37         MPI_Recv(&numero_recibido, 1, MPI_INT, rank - 1, TAG_DATO,
               MPI_COMM_WORLD, &status);
38
39         if (numero_recibido == CENTINELA_FIN) {
40             // Propagar centinela al siguiente filtro si existe
41             if (rank + 1 < size) {
42                 MPI_Send(&numero_recibido, 1, MPI_INT, rank + 1, TAG_DATO,
                       MPI_COMM_WORLD);
43             }
44             break;
45         }
46
47         // Si no es divisible, propagar hacia el siguiente proceso de la
               tuberia
48         if (numero_recibido % mi_primo != 0) {
49             if (rank + 1 < size) {
50                 MPI_Send(&numero_recibido, 1, MPI_INT, rank + 1, TAG_DATO,
                       MPI_COMM_WORLD);
51             }
52         }
53     }
54 }
55
56 int main(int argc, char* argv[]) {
57     MPI_Init(&argc, &argv);
58     int rank, size;
59     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
60     MPI_Comm_size(MPI_COMM_WORLD, &size);
61
62     if (rank == 0) {
```

```
        proceso_generador(rank, size);
    } else {
        proceso_filtro(rank, size);
    }

    MPI_Finalize();
    return 0;
}
```

Listing 4.2: Criba de Eratóstenes distribuida en Pipeline con MPI

4.4.2 Algoritmo 2: El Problema del Museo con MPI

Planteamiento Arquitectónico

El problema del museo modela un sistema distribuido Cliente-Servidor donde se controla el aforo de una sala con capacidad máxima C:

- Un proceso centralizado denominado **Controlador de Sala** (*Servidor*).
- Múltiples procesos concurrentes denominados **Visitantes** (*Clientes*).
- Dos tipos de acciones diferenciadas por etiquetas:
  - `TAG_ENTRADA`: Un visitante solicita entrar.
  - `TAG_SALIDA`: Un visitante comunica que sale de la sala.
- **Regla de Sincronización:** El controlador acepta salidas incondicionalmente (decrementando el aforo). Sin embargo, solo acepta peticiones de entrada si el aforo actual es menor estrictamente que la capacidad máxima ($aforo < C$).

Implementación en C++ / MPI

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

const int RANK_CONTROLADOR = 0;
const int CAPACIDAD_MUSEO = 3;
const int TAG_ENTRADA = 1;
const int TAG_PERMISO = 2;
const int TAG_SALIDA = 3;

void controlador_museo(int total_procesos) {
    int aforo = 0;
    int dummy;
    MPI_Status status;

    // Bucle de atencion a peticiones
    while (true) {
        // Si el aforo esta lleno, solo podemos recibir SALIDAS
        // Si hay espacio, podemos recibir tanto ENTRADAS como SALIDAS
        if (aforo == CAPACIDAD_MUSEO) {
            MPI_Recv(&dummy, 1, MPI_INT, MPI_ANY_SOURCE, TAG_SALIDA,
                MPI_COMM_WORLD, &status);
            aforo--;
            std::cout << "[Museo] Visitante " << status.MPI_SOURCE
                      << " ha salido. Aforo actual: " << aforo << std::endl;
        } else {
            // Recepcion con comodin de etiqueta
```

```
27              MPI_Recv(&dummy, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &status);
28              if (status.MPI_TAG == TAG_ENTRADA) {
29                  aforo++;
30                  std::cout << "[Museo] Visitante " << status.MPI_SOURCE
31                            << " entra al museo. Aforo actual: " << aforo <<
                                  std::endl;
32                  // Enviar confirmacion al visitante
33                  MPI_Send(&dummy, 1, MPI_INT, status.MPI_SOURCE, TAG_PERMISO,
                        MPI_COMM_WORLD);
34              } else if (status.MPI_TAG == TAG_SALIDA) {
35                  aforo--;
36                  std::cout << "[Museo] Visitante " << status.MPI_SOURCE
37                            << " ha salido. Aforo actual: " << aforo << std::
                                  endl;
38              }
39          }
40      }
41  }
42
43  void visitante(int rank) {
44      int dummy = 0;
45      MPI_Status status;
46
47      // 1. Solicitar entrada
48      MPI_Send(&dummy, 1, MPI_INT, RANK_CONTROLADOR, TAG_ENTRADA,
            MPI_COMM_WORLD);
49
50      // 2. Esperar confirmacion de acceso
51      MPI_Recv(&dummy, 1, MPI_INT, RANK_CONTROLADOR, TAG_PERMISO,
            MPI_COMM_WORLD, &status);
52
53      // 3. Estancia en el museo
54      std::cout << "Visitante " << rank << " esta disfrutando del museo..." <<
            std::endl;
55      sleep(1);
56
57      // 4. Notificar salida
58      MPI_Send(&dummy, 1, MPI_INT, RANK_CONTROLADOR, TAG_SALIDA, MPI_COMM_WORLD
            );
59  }
```

Listing 4.3: Controlador de Aforo del Museo con MPI

4.5 La Orden `select` y Recepción Selectiva

4.5.1 Motivación: Recepción No Determinista en Servidores

En un sistema basado en paso de mensajes, un proceso servidor a menudo necesita escuchar peticiones provenientes de múltiples clientes a través de diferentes canales o interfaces.

Si el servidor ejecuta un `receive` secuencial clásico sobre un canal A:

$$\text{receive}(A, \text{msg});$$

quedará bloqueado esperando a A. Si mientras tanto un cliente envía un mensaje urgente por el canal B, el servidor **no se enterará y no lo procesará** hasta que A termine. Este acoplamiento temporal estricto bloquea innecesariamente el sistema.

Para resolver este problema en sistemas concurrentes y distribuidos, C.A.R. Hoare (en CSP) y Jean Ichbiah (en el lenguaje Ada) introdujeron la **orden de espera selectiva** o sentencia `select`.

4.5.2 Sintaxis y Semántica Formal de las Guardas

La estructura general de una sentencia de espera selectiva está formada por un conjunto de ramas o alternativas:

```
select
    when condicion_1; receive(msg_1, proceso_1) do
        sentencias_1;
    or
    when condicion_2; receive(msg_2, proceso_2) do
        sentencias_2;
    ...
    or
    when condicion_n; receive(msg_n, proceso_n) do
        sentencias_n;
    else
        sentencias_alternativas;
end select;
```

Cada rama está encabezada por una **guarda**. Una guarda se compone de:

1. Una **expresión booleana** opcional (`when condicion`). Si se omite, equivale a `when true`.

2. Una **instrucción de entrada o recepción** (`receive`).

4.5.3 Estados de una Guarda y Criterio de Selección

En el instante en que el flujo de ejecución alcanza la sentencia `select`, el sistema evalúa todas las expresiones booleanas de las guardas. Se distinguen formalmente los siguientes estados:

Definición: Clasificación de Guardas

- **Guarda Abierta:** La expresión booleana se evalúa a `true`.
- **Guarda Cerrada:** La expresión booleana se evalúa a `false`. La alternativa completa se descarta en este ciclo.
- **Guarda Ejecutable:** Está abierta y el proceso emisor correspondiente ya ha iniciado la transmisión del mensaje (el mensaje ya está disponible en el canal).
- **Guarda Potencialmente Ejecutable:** Está abierta y tiene instrucción `receive`, pero el proceso emisor aún no ha enviado el mensaje correspondiente.

Algoritmo de Determinación de la Alternativa a Ejecutar

1. **Si existen una o más guardas ejecutables:** El entorno de ejecución selecciona una de ellas de forma **no determinista**. Se extrae el mensaje de esa alternativa y se ejecuta el bloque de sentencias asociado. El no determinismo es crucial para evitar que el servidor favorezca arbitrariamente a un canal sobre los demás.

2. **Si no hay ninguna guarda ejecutable, pero sí hay guardas potencialmente ejecutables:**

    - Si existe una rama `else`, el proceso ejecuta inmediatamente el bloque del `else` sin bloquearse.

- Si no existe rama `else`, el proceso se suspende en el `select` hasta que alguno de los procesos emisores de las guardas abiertas inicie un envío, convirtiendo esa guarda en ejecutable.

3. **Si todas las guardas están cerradas:** Si no existe rama `else`, se produce un error fatal en tiempo de ejecución (*Program_Error* o excepción de selección).

4.5.4 Guardas Indexadas y Sentencia `else`

Guardas Indexadas

Cuando un servidor atiende a una familia de N clientes idénticos, expandir las ramas a mano resulta inviable. Se utilizan **guardas indexadas**:

```
select
    for i in 1..N generate
        when canal_activo[i]; receive(msg, cliente[i]) do
            atender_peticion(i, msg);
end select;
```

Esta notación compacta equivale a declarar N alternativas `or` separadas, donde el índice i instancia dinámicamente cada rama.

Sentencia `else`

La presencia de una cláusula `else` transforma la espera selectiva en una operación de **sondeo no bloqueante**: si ninguna guarda está lista de inmediato, no se espera y se realiza trabajo alternativo en segundo plano.

4.5.5 Ejemplo Clásico: Productor-Consumidor con Búfer FIFO mediante `select`

A continuación se presenta el diseño canónico de un proceso intermedio (el Búfer FIFO) que media entre productores y consumidores utilizando la orden `select`:

```
process Intermedio_Buffer {
    const int TAM = 10;
    int buffer[TAM];
    int esc = 0;    // Indice de escritura
    int lec = 0;    // Indice de lectura
    int cont = 0;   // Numero de elementos en el bufer
    int valor;
    int peticion;

    while (true) {
        select
            // Guarda del Productor: solo abierta si el bufer no esta lleno
            when cont < TAM; receive(valor, Productor) do
                buffer[esc] = valor;
                esc = (esc + 1) % TAM;
                cont = cont + 1;
            or
            // Guarda del Consumidor: solo abierta si el bufer no esta vacio
            when cont > 0; receive(peticion, Consumidor) do
                // Enviar el dato extraido al consumidor
                send(buffer[lec], Consumidor);
                lec = (lec + 1) % TAM;
                cont = cont - 1;
        end select;
    }
}
```

Listing 4.4: Búfer FIFO con Orden Select en Pseudocódigo Distribuido

Importante / Ojo Examen: Elegancia de la Sincronización con Select

Nótese cómo la condición booleana de la guarda (`cont <TAM` y `cont >0`) modela de forma natural y transparente las condiciones de sincronización que en monitores requerían cerrojos y variables de condición. El propio lenguaje gestiona el bloqueo y la activación segura.

Tema 5

Sistemas de Tiempo Real (STR)

Metadatos del Tema y Fuentes:

Fuentes utilizadas: Apuntes de Ismael Sallami y Temario de Infomates (LosDelDGIIM). Pendiente de incorporar transparencias originales del profesor.

5.1 Introducción y Fundamentos de los Sistemas de Tiempo Real

Definición: Sistema de Tiempo Real (STR)

Un **Sistema de Tiempo Real** (STR) es aquel sistema informático en el que la corrección del sistema no solo depende del resultado lógico o funcional del cómputo, sino también del **instante temporal** en el que dicho resultado es producido y entregado.

En un sistema informático convencional (de propósito general o por lotes), la métrica fundamental de optimización es el rendimiento promedio (*throughput*) o el tiempo medio de respuesta: una respuesta computacionalmente correcta entregada con retraso sigue siendo válida y valiosa. Por el contrario, en un STR una respuesta computacionalmente perfecta que se entrega fuera de su ventana temporal límite (*deadline*) se considera un fallo del sistema, pudiendo ocasionar desde una degradación de la calidad de servicio hasta catástrofes físicas o humanas.

5.1.1 Distinción con otros tipos de sistemas

En la literatura y en la práctica industrial, los STR se confunden habitualmente con otras arquitecturas computacionales. Es crucial establecer las siguientes distinciones:

- **Sistemas Rápidos vs. Sistemas de Tiempo Real:** Un sistema de tiempo real no es simplemente un sistema de cómputo ultrarrápido. La rapidez por sí sola no garantiza predictibilidad. Un supercomputador puede procesar billones de operaciones de coma flotante por segundo pero sufrir oscilaciones impredecibles en el despacho de hilos debido a la memoria caché o al planificador. Lo esencial en un STR es el **determinismo temporal** y la garantía formal de cumplimiento de plazos en el peor caso previsible (*Worst-Case Execution Time*, WCET).
- **Sistemas en Línea (*On-line*):** Los sistemas en línea están permanentemente disponibles y conectados a transacciones o sensores (por ejemplo, una base de datos bancaria), pero no necesariamente garantizan plazos máximos inflexibles en cada transacción particular.

- **Sistemas Interactivos:** Diseñados para responder a peticiones humanas directas (como un procesador de textos o un navegador web). Aunque requieren agilidad para evitar frustración en el usuario, un retraso de unos milisegundos no compromete la integridad del sistema.

5.1.2 Definición de Sistema Operativo de Tiempo Real (RTOS)

Definición: RTOS — Real-Time Operating System

Un **Sistema Operativo de Tiempo Real** es aquel que tiene la capacidad demostrable de suministrar un nivel de servicio requerido a todos sus procesos de tiempo real en un tiempo limitado y **especificado a priori**, independientemente de la carga instantánea o de las perturbaciones del entorno.

5.1.3 Propiedades Fundamentales de los STR

Un STR debe reunir cuatro cualidades primordiales:

1. **Determinismo:** Dada una entrada en un estado determinado, el sistema produce siempre la misma secuencia de salidas en los mismos instantes temporales predecibles. El comportamiento es calculable analíticamente con antelación.

2. **Reactividad:** Capacidad de detectar eventos del entorno externo físico (mediante interrupciones de sensores o periféricos) y responder de forma coordinada a la velocidad que exige dicho entorno.

3. **Responsabilidad y Predictibilidad:** Capacidad de garantizar matemáticamente que todas las tareas concurrentes finalizarán dentro de sus plazos establecidos, incluso bajo condiciones de sobrecarga máxima prevista.

4. **Confiabilidad y Tolerancia a Fallos (*Dependability*):** Comportamiento robusto y seguro (*fail-safe*) ante anomalías de hardware o software, evitando que un fallo parcial desencadene una parada total o una catástrofe.

5.1.4 Clasificación según la Criticidad Temporal

Atendiendo a las consecuencias que produce el incumplimiento de un plazo límite (*deadline*), los STR se clasifican rigurosamente en tres categorías:

Categoría	Consecuencia del Fallo de Deadline	Ejemplos Representativos
Misión Crítica (*Hard Real-Time*)	El incumplimiento de un solo plazo puede provocar consecuencias catastróficas, daños materiales irreversibles o pérdida de vidas humanas.	Control de vuelo fly-by-wire, sistemas de frenado ABS, reactores nucleares, marcapasos.
Estrictos (*Firm Real-Time*)	El incumplimiento del plazo invalida el resultado por completo (utilidad nula), pero no desencadena un desastre catastrófico.	Transmisión de vídeo en tiempo real sin búfer, control de inyección en motores industriales.
Permisivos (*Soft Real-Time*)	El incumplimiento esporádico del plazo es tolerable; el resultado tardío aún posee utilidad decreciente y solo degrada la calidad del servicio.	Adquisición de datos meteorológicos, streaming multimedia con búfer, servidores de videojuegos.

5.2 Medición del Tiempo, Relojes y Temporizadores

5.2.1 Tiempo Absoluto frente a Tiempo Relativo

En el modelado de STR intervienen dos concepciones del tiempo:

- **Tiempo Absoluto:** Medido con respecto a una escala universal de referencia externa, invariante y continua, como el Tiempo Universal Coordinado (UTC), el Tiempo Atómico Internacional (TAI, basado en transiciones cuánticas del átomo de cesio-133) o las señales horarias de constelaciones GPS.
- **Tiempo Relativo o Intervalos:** Medición del tiempo transcurrido entre dos eventos dentro del propio sistema computacional (por ejemplo: «esperar 15 milisegundos a partir de ahora»). Es la métrica predominante en la programación de retardos y periodos.

5.2.2 Módulos de Reloj de Hardware

Un reloj en tiempo real físico consta de un oscilador de cuarzo a frecuencia fija que incrementa o decrementa un registro contador en cada ciclo. Sus atributos fundamentales son:

- **Resolución o Granularidad (Δt):** El intervalo temporal mínimo que el reloj puede discriminar entre dos eventos sucesivos.
- **Precisión y Deriva (*Drift*):** Desviación entre la frecuencia real de oscilación y la nominal teórica, provocada por temperatura, envejecimiento del cristal o variaciones de voltaje.
- **Rango antes de Desbordamiento (*Overflow*):** Intervalo máximo medible antes de que el registro de k bits vuelva a cero. Para un contador clásico de 32 bits sin signo ($2^{32} - 1 \approx 4{,}295 \times 10^9$ pulsos):

Resolución del Tick	Rango Máximo de Medición (32 bits)	Ámbito de Aplicación
100 ns	429,5 segundos ≈ 7,15 minutos	Medición de buses de comunicaciones ultrarrápidos
1 μs	4,294,9 s ≈ 71,58 minutos	Control de actuadores mecánicos de precisión
100 μs	429,496 s ≈ 119,3 horas ≈ 4,97 días	Planificación clásica de RTOS
1 ms	49,71 días	Sistemas embebidos con ticks milimétricos
1 s	136,18 años	Relojes de calendario y registro histórico

5.2.3 Temporizadores y Eliminación de la Deriva Acumulativa

Los temporizadores de software se gestionan mediante interrupciones del kernel. Se distinguen temporizadores **de un solo disparo** (*one-shot*) y **periódicos** (*periodic*).

Importante / Ojo Examen: El Peligro de la Deriva Acumulativa en Bucles Periódicos

Si una tarea periódica se implementa encadenando tiempos de espera relativos mediante llamadas tipo `sleep_for(T)`, el periodo real resultante sufrirá una **deriva acumulativa** inevitable debida al tiempo que consume la propia computación del cuerpo del

bucle y al tiempo de despacho del planificador:

$$T_{\text{real}} = T_{\text{deseado}} + C_{\text{cuerpo}} + t_{\text{cambio-contexto}}$$

En unos pocos minutos, la tarea se habrá desfasado completamente de su frecuencia de muestreo prescrita.

Para eliminar de raíz la deriva acumulativa, los estándares modernos de tiempo real (POSIX y C++11 en adelante con `<chrono>`) proporcionan primitivas de espera absoluta como `std::this_thread::sleep_until`:

```cpp
#include <chrono>
#include <thread>

void TareaPeriodica(const int periodo_ms) {
    using namespace std::chrono;
    // Seleccionar siempre un reloj monotono inmune a cambios horarios
    auto siguiente_activacion = steady_clock::now();

    while (true) {
        // 1. Ejecucion del trabajo periodico
        realizar_muestreo();

        // 2. Calculo exacto del proximo instante absoluto
        siguiente_activacion += milliseconds(periodo_ms);

        // 3. Suspension hasta el instante absoluto precalculado
        // Esto compensa automaticamente el tiempo gastado en
            realizar_muestreo()
        std::this_thread::sleep_until(siguiente_activacion);
    }
}
```

Listing 5.1: Implementación rigurosa de tarea periódica sin deriva acumulativa

5.3 Modelo de Tareas y Atributos Temporales

Un programa de tiempo real se modela formalmente como un conjunto finito de n tareas concurrentes:

$$\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

5.3.1 Clasificación de Tareas por su Patrón de Activación

- **Tareas Periódicas:** Se activan en intervalos regulares e idénticos de tiempo T_i. Son la columna vertebral de los sistemas de muestreo digital y control por retroalimentación.
- **Tareas Aperiódicas:** Se activan como respuesta a eventos externos aleatorios o esporádicos del entorno. Sus instantes de activación son impredecibles.
- **Tareas Esporádicas:** Tareas aperiódicas que cuentan con una garantía física o de diseño de un **intervalo mínimo entre activaciones sucesivas** ($T_{i,\text{mín}}$). Gracias a este límite mínimo, pueden ser sometidas a tests matemáticos de planificabilidad en el peor caso tratándolas como tareas periódicas de periodo $T_{i,\text{mín}}$.

5.3.2 Parámetros y Atributos Temporales de una Tarea τ_i

Para cada tarea τ_i se definen las siguientes variables formales:

- T_i (**Periodo**): Tiempo transcurrido entre dos activaciones sucesivas de la tarea.
- C_i (**Tiempo de Cómputo en el Peor Caso** o WCET): Cota superior del tiempo de CPU que consume una instancia de la tarea sin considerar interferencias ni bloqueos.
- D_i (**Plazo de Respuesta Relativo** o *Deadline*): Tiempo máximo transcurrido desde que la tarea se activa hasta que completa su ejecución. En el modelo clásico simple, $D_i = T_i$; en modelos generales, $D_i \leq T_i$.
- $t_{a,k}$ (**Instante de Activación** de la instancia k): Momento en que la tarea pasa al estado de lista para ejecutar.
- $d_k = t_{a,k} + D_i$ (**Plazo Absoluto**): Momento temporal antes del cual debe finalizar la instancia k.
- R_i (**Tiempo de Respuesta en el Peor Caso** o WCRT): Tiempo máximo real transcurrido entre la activación y la finalización, teniendo en cuenta la interferencia de otras tareas del sistema.
- J_i (**Fluctuación o Jitter**): Variación máxima en el tiempo de comienzo efectivo o de finalización de una tarea entre diferentes ciclos.
- $H_i = D_i - C_i$ (**Holgura o Slack Time**): Margen temporal disponible antes de que la tarea incurra en incumplimiento de plazo si no sufre interferencias.
- ϕ_i (**Fase o Desplazamiento Inicial**): Instante en el que se produce la primera activación de la tarea τ_i tras el arranque del sistema.

Definición: Factor de Utilización del Procesador

El factor de utilización del procesador U_i correspondiente a una tarea periódica τ_i, y la utilización global U del sistema, se definen como la fracción del tiempo de CPU requerida para su ejecución:

$$U_i = \frac{C_i}{T_i}, \qquad U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{5.1}$$

Una condición obvia y necesaria para la viabilidad de cualquier sistema monoprocesador es que $U \leq 1$ (100 %).

5.4 Planificación Cíclica y Ejecutivos Cíclicos

5.4.1 Concepto y Funcionamiento

La **Planificación Cíclica** (o Ejecutivo Cíclico) es una técnica estática y determinista, gobernada por una tabla precalculada fuera de línea (*off-line*) que dicta con exactitud qué tareas deben ejecutarse y en qué orden dentro de intervalos regulares impulsados por las interrupciones del reloj hardware.

No existe un planificador dinámico en tiempo de ejecución: el ejecutivo cíclico se implementa típicamente como un bucle determinista sin hilos ni cambios de contexto complejos, lo que minimiza la sobrecarga computacional.

5.4.2 Parámetros de Diseño: Ciclo Mayor y Ciclo Menor

La línea temporal de ejecución se estructura en dos niveles:

- **Ciclo Mayor o Hiperperiodo (M):** Intervalo temporal tras el cual el patrón de ejecución de todas las tareas periódicas se repite de forma idéntica. Se calcula como el mínimo común múltiplo de los periodos:

$$M = \text{mcm}(T_1, T_2, \ldots, T_n) \tag{5.2}$$

- **Ciclo Menor o Marco Temporal / Frame (m):** Subdivisión regular del ciclo mayor. El reloj del sistema genera una interrupción periódica cada m unidades de tiempo, marcando el inicio de un nuevo marco temporal (*minor cycle*).

5.4.3 Las Cuatro Condiciones de Diseño del Ciclo Menor

Para que un ejecutivo cíclico sea válido y garantice que ninguna tarea sobrepase su plazo de respuesta, el tamaño del ciclo menor m debe satisfacer estrictamente cuatro restricciones:

Condiciones de Diseño del Ciclo Menor m

1. **Regla del Tiempo de Cómputo Máximo:**

$$m \geq \max_{1 \leq i \leq n} (C_i) \tag{5.3}$$

   Cada tarea debe caber íntegramente dentro de un ciclo menor sin ser expulsada (a menos que se divida artificialmente en subtareas).

2. **Regla de Periodicidad Armónica:**

$$M \pmod{m} = 0 \tag{5.4}$$

   El ciclo mayor debe ser un múltiplo entero exacto del ciclo menor, es decir, el número de ciclos menores por ciclo mayor es un entero $K = M/m$.

3. **Regla del Plazo de Respuesta Mínimo:**

$$m \leq \min_{1 \leq i \leq n} (D_i) \tag{5.5}$$

   Debe transcurrir como máximo un ciclo menor antes de que expire el deadline más exigente del sistema.

4. **Regla de Garantía de Cumplimiento de Deadline:** Para toda tarea τ_i, entre la activación de la tarea y su plazo de respuesta debe existir al menos un ciclo menor completo disponible para su ejecución:

$$2m - \gcd(m, T_i) \leq D_i, \qquad \forall i \in \{1, \ldots, n\} \tag{5.6}$$

   donde $\gcd(m, T_i)$ representa el máximo común divisor entre el marco m y el periodo T_i.

Ejemplo Práctico: Diseño Completo de un Ejecutivo Cíclico

Consideremos un conjunto de tres tareas periódicas independientes con $D_i = T_i$:

Tarea	Cómputo (C_i)	Periodo (T_i)	Plazo (D_i)
τ_1	1	4	4
τ_2	2	10	10
τ_3	1	20	20

Paso 1: Cálculo del Ciclo Mayor (M)

$$M = \text{mcm}(4, 10, 20) = 20$$

Paso 2: Evaluación de las condiciones sobre el Ciclo Menor (m)

- Condición 1: $m \geq \text{máx}(C_i) = \text{máx}(1, 2, 1) = 2$.
- Condición 2: $20 \pmod m = 0 \implies m \in \{2, 4, 5, 10, 20\}$.
- Condición 3: $m \leq \text{mín}(D_i) = \text{mín}(4, 10, 20) = 4$.
- Cruzando las condiciones 1, 2 y 3: los candidatos viables son $m = 2$ o $m = 4$.
- Condición 4 para $m = 4$:
  - Para τ_1: $2(4) - \gcd(4, 4) = 8 - 4 = 4 \leq 4$ **(Se cumple)**.
  - Para τ_2: $2(4) - \gcd(4, 10) = 8 - 2 = 6 \leq 10$ **(Se cumple)**.
  - Para τ_3: $2(4) - \gcd(4, 20) = 8 - 4 = 4 \leq 20$ **(Se cumple)**.

Por tanto, elegimos $m = 4$. El ciclo mayor consta de $K = M/m = 20/4 = 5$ ciclos menores.

Paso 3: Construcción de la Tabla de Despacho (Cuadro de Ejecución) En cada ciclo menor de duración 4 disponemos de tiempo suficiente para asignar las instancias:

Marco (k)	Intervalo	Tareas Asignadas	Tiempo Ocupado / Capacidad
Marco 1	$[0, 4)$	τ_1, τ_2	$1 + 2 = 3 \leq 4$
Marco 2	$[4, 8)$	τ_1, τ_3	$1 + 1 = 2 \leq 4$
Marco 3	$[8, 12)$	τ_1	$1 \leq 4$
Marco 4	$[12, 16)$	τ_1, τ_2	$1 + 2 = 3 \leq 4$
Marco 5	$[16, 20)$	τ_1	$1 \leq 4$

Todas las instancias se ejecutan sin interferencias y cumpliendo sus plazos de forma determinista.

5.4.4 Ventajas y Desventajas del Ejecutivo Cíclico

- **Ventajas:** Cero sobrecarga por cambio de contexto; determinismo absoluto y ausencia de condiciones de carrera sobre variables compartidas; no requiere primitivas complejas de sincronización en el sistema operativo.
- **Inconvenientes:** Extrema rigidez. Si una tarea excede su C_i o si se añade una nueva tarea con periodo primo, todo el plan se desmorona y exige recalcular M y m. Si una tarea tiene un C_i mayor que m, debe dividirse artificialmente en subtareas (*task slicing*), lo que complica la arquitectura del software.

5.5 Planificación con Prioridades Estáticas: Cadencia Monótona (RMS)

El algoritmo de **Cadencia Monótona** (*Rate Monotonic Scheduling*, RMS), introducido formalmente por C. L. Liu y J. W. Layland en su trabajo seminal de 1973, es el estándar por excelencia de la planificación estática expulsiva.

5.5.1 Regla de Asignación de Prioridades

Definición: Regla de Asignación RMS

A cada tarea periódica τ_i se le asigna de manera fija una prioridad estática P_i que es monótona respecto a su frecuencia de activación (inversamente proporcional a su periodo):

$$\forall i, j: \quad T_i < T_j \implies P_i > P_j \tag{5.7}$$

La tarea con menor periodo (mayor frecuencia de repetición) tiene siempre la mayor prioridad.

Importante / Ojo Examen: Urgencia frente a Criticidad

La prioridad en RMS modela la **urgencia temporal** de la tarea (con qué frecuencia debe despacharse), **no su criticidad**. Una tarea que monitoriza una válvula crítica cada 500 ms tendrá menor prioridad bajo RMS que una tarea de lectura de teclado que se activa cada 50 ms.

5.5.2 Hipótesis Fundamentales de Liu y Layland

El análisis formal clásico de RMS descansa sobre las siguientes hipótesis:

1. Todas las tareas son rigurosamente periódicas e independientes entre sí (no existen relaciones de precedencia ni exclusión mutua con bloqueos por recursos).

2. Para cada tarea, el plazo de respuesta relativo coincide exactamente con su periodo: $D_i = T_i$.

3. El tiempo de cómputo en el peor caso C_i es constante y conocido a priori.

4. El planificador es con expulsión (*preemptive*): una tarea de mayor prioridad interrumpe de inmediato a una de menor prioridad.

5. La sobrecarga del sistema operativo por cambios de contexto e interrupciones es despreciable.

Bajo estas hipótesis, Liu y Layland demostraron que RMS es un **algoritmo óptimo** en monoprocesador entre todos los esquemas de prioridad estática: si un conjunto de tareas con $D_i = T_i$ es planificable mediante alguna asignación fija de prioridades, también lo es asignando prioridades según RMS.

5.5.3 Test I de Liu y Layland: Condición Suficiente por Factor de Utilización

Definición: Teorema de Utilización Límites de RMS

Un conjunto de n tareas periódicas independientes que satisface las hipótesis de Liu y Layland es **garantizadamente planificable** bajo RMS si la utilización global del procesador U no supera la cota $U_0(n)$:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_0(n) = n\left(2^{1/n} - 1\right) \tag{5.8}$$

La función $U_0(n)$ decrece monótonamente conforme crece el número de tareas n, convergiendo asintóticamente a $\ln 2$:

$$\lim_{n\to\infty} U_0(n) = \lim_{n\to\infty} n\left(2^{1/n} - 1\right) = \ln 2 \approx 0{,}69315 \quad (69{,}3\,\%) \tag{5.9}$$

$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 10$	$n \to \infty$
100,0 %	82,84 %	77,98 %	75,68 %	74,35 %	71,77 %	69,31 %

5.5.4 Inexactitud del Test de Utilización de Liu y Layland

Es vital comprender que el test del factor de utilización es una **condición suficiente, pero NO necesaria**:

- Si $U \leq U_0(n)$, el conjunto de tareas es **garantizadamente planificable**.
- Si $U > 1$, el conjunto es **no planificable** en monoprocesador bajo ningún algoritmo.
- Si $U_0(n) < U \leq 1$, el test es **no concluyente**: el sistema puede ser planificable o no, requiriendo un análisis más profundo.

Por ejemplo, un conjunto de tareas cuyos periodos son múltiplos armónicos exactos (ej. $T = \{10, 20, 40\}$) puede alcanzar una utilización de hasta el 100 % ($U = 1$) bajo RMS sin perder ningún plazo.

5.5.5 Test II: Análisis Exacto del Tiempo de Respuesta en el Peor Caso (R_i)

Para obtener una condición **necesaria y suficiente** para cualquier conjunto de tareas bajo asignación estática de prioridades (incluso cuando $D_i \leq T_i$, conocido como *Deadline Monotonic Scheduling* o DMS), se utiliza el método de análisis del tiempo de respuesta introducido por Joseph y Pandya (1986) y refinado por Audsley.

El instante de máxima exigencia para una tarea τ_i se produce en el llamado **instante crítico**: cuando se activa simultáneamente junto a todas las tareas de mayor prioridad que ella.

El tiempo de respuesta en el peor caso R_i satisface la ecuación recurrente de punto fijo:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{5.10}$$

donde $hp(i)$ denota el conjunto de tareas con prioridad estrictamente mayor que τ_i, y $\lceil x \rceil$ es la función techo (número entero inmediatamente superior o igual a x). El término $\lceil R_i/T_j \rceil C_j$ representa la interferencia máxima que la tarea τ_j ejerce sobre τ_i.

Algoritmo / Protocolo: Algoritmo Iterativo de Punto Fijo para R_i

Para calcular R_i, se genera una sucesión monótona no decreciente $R_i^{(0)}, R_i^{(1)}, R_i^{(2)}, \ldots$:

$$R_i^{(0)} = C_i \tag{5.11}$$

$$R_i^{(k+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j \tag{5.12}$$

El algoritmo finaliza cuando:

1. $R_i^{(k+1)} = R_i^{(k)}$: El proceso converge al valor exacto del tiempo de respuesta R_i. Si $R_i \leq D_i$, la tarea τ_i es planificable.
2. $R_i^{(k+1)} > D_i$: La tarea incumple su plazo; el sistema es no planificable.

Ejemplo Práctico: Aplicación Práctica del Análisis de Tiempo de Respuesta

Sean tres tareas periódicas:

- $\tau_1 : C_1 = 2, T_1 = 5 \implies U_1 = 0{,}40$ (Mayor prioridad por RMS).
- $\tau_2 : C_2 = 2, T_2 = 8 \implies U_2 = 0{,}25$ (Prioridad media).
- $\tau_3 : C_3 = 2, T_3 = 10 \implies U_3 = 0{,}20$ (Menor prioridad).

La utilización global es $U = 0{,}40 + 0{,}25 + 0{,}20 = 0{,}85$ (85 %). Como $U_0(3) = 3(2^{1/3} - 1) \approx 0{,}7798$ (78 %), tenemos $U > U_0(3)$, por lo que el test de Liu y Layland **no es concluyente**. Aplicamos la recurrencia exacta:
1. Para τ_1: Como no tiene tareas de mayor prioridad ($hp(1) = \emptyset$), $R_1 = C_1 = 2 \leq 5$ **(Planificable)**.
2. Para τ_2 ($hp(2) = \{\tau_1\}$):

- $R_2^{(0)} = C_2 = 2$.
- $R_2^{(1)} = 2 + \lceil 2/5 \rceil \cdot 2 = 2 + 1 \cdot 2 = 4$.
- $R_2^{(2)} = 2 + \lceil 4/5 \rceil \cdot 2 = 2 + 1 \cdot 2 = 4$.
- Convergencia alcanzada: $R_2 = 4 \leq 8$ **(Planificable)**.

3. Para τ_3 ($hp(3) = \{\tau_1, \tau_2\}$):

- $R_3^{(0)} = C_3 = 2$.
- $R_3^{(1)} = 2 + \lceil 2/5 \rceil \cdot 2 + \lceil 2/8 \rceil \cdot 2 = 2 + 2 + 2 = 6$.
- $R_3^{(2)} = 2 + \lceil 6/5 \rceil \cdot 2 + \lceil 6/8 \rceil \cdot 2 = 2 + 2(2) + 1(2) = 2 + 4 + 2 = 8$.
- $R_3^{(3)} = 2 + \lceil 8/5 \rceil \cdot 2 + \lceil 8/8 \rceil \cdot 2 = 2 + 2(2) + 1(2) = 8$.
- Convergencia: $R_3 = 8 \leq 10$ **(Planificable)**.

Conclusión: A pesar de superar la cota de Liu y Layland (85 % > 78 %), el sistema es **completamente planificable** porque todas las tareas cumplen $R_i \leq D_i$.

5.6 Planificación con Prioridades Dinámicas: Earliest Deadline First (EDF)

5.6.1 Principio del Algoritmo EDF

A diferencia de los esquemas estáticos, en el algoritmo **Earliest Deadline First** (EDF) las prioridades se asignan de forma **dinámica** en tiempo de ejecución.

Definición: Regla de Asignación EDF

En cada instante de planificación, el procesador se asigna a la tarea activa cuyo **plazo de respuesta absoluto** ($d_k = t_{a,k} + D_i$) sea más inminente o próximo en el tiempo.

5.6.2 Teorema de Planificabilidad de EDF

Bajo las hipótesis de Liu y Layland con $D_i = T_i$, EDF es óptimo en monoprocesador y presenta un resultado extraordinario:

Definición: Condición Necesaria y Suficiente para EDF

Un conjunto de n tareas periódicas independientes con $D_i = T_i$ es planificable bajo EDF si y solo si:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{5.13}$$

Esto significa que EDF permite aprovechar teóricamente el **100 % de la capacidad del procesador**, eliminando la restricción del 69,3 % de RMS.

5.6.3 Comparativa Detallada: RMS frente a EDF

Característica	Cadencia Monótona (RMS)	Earliest Deadline First (EDF)
Tipo de Prioridad	Estática (inversa al periodo T_i)	Dinámica (según deadline absoluto d_k)
Utilización Máxima	$n(2^{1/n} - 1) \to 69{,}3\,\%$ (100 % en armónicos)	100 % ($U \leq 1$) para $D_i = T_i$
Sobrecarga de Gestión	Mínima: ordenación estática fija	Mayor: requiere actualizar y reordenar colas de prioridad en cada evento
Sobrecarga Transitoria	**Predecible y Controlada:** solo sufren las tareas de menor prioridad	**Efecto Dominó Impredecible:** una tarea retrasada puede hacer fallar en cascada a tareas críticas
Implementación	Soportada nativamente en cualquier RTOS con prioridades fijas	Requiere soporte de colas dinámicas

5.7 Inversión de Prioridades y Compartición de Recursos

En aplicaciones reales de tiempo real, las tareas no son enteramente independientes: deben coordinarse y compartir recursos pasivos en exclusión mutua (variables compartidas, puertos serie, buses de comunicación) protegidos mediante cerrojos o semáforos.

5.7.1 El Problema de la Inversión de Prioridad No Acotada

Definición: Inversión de Prioridad

Se dice que se produce una **inversión de prioridad** cuando una tarea de alta prioridad se ve demorada en su ejecución esperando que una tarea de menor prioridad libere un recurso compartido, permitiendo que tareas de prioridad intermedia (que no usan dicho recurso) se ejecuten y prolonguen el bloqueo de forma arbitraria.

Consideremos el siguiente escenario clásico con tres tareas concurrentes:

- τ_A: Tarea de alta prioridad (*High*).
- τ_M: Tarea de prioridad media (*Medium*).
- τ_B: Tarea de baja prioridad (*Low*).

1. En t_0, τ_B entra en ejecución y adquiere el cerrojo de un recurso compartido R.
2. En t_1, se activa τ_A. Al ser de mayor prioridad, expulsa a τ_B y comienza a ejecutarse.
3. En t_2, τ_A solicita acceder al recurso compartido R. Al estar bloqueado por τ_B, τ_A se suspende. El procesador pasa de nuevo a τ_B para que termine su sección crítica.
4. En t_3, antes de que τ_B libere el recurso, se activa una tarea de prioridad intermedia τ_M que **no utiliza el recurso** R.
5. Como τ_M tiene mayor prioridad que τ_B, el planificador expulsa a τ_B y cede la CPU a τ_M.
6. **Efecto catastrófico:** τ_A (alta prioridad) queda bloqueada indirectamente esperando a τ_M (prioridad media), a pesar de que no comparten ningún recurso. Si existen múltiples tareas intermedias, τ_A puede sufrir un retraso no acotado que la conduzca al fallo de su deadline.

Ejemplo Práctico: Caso Real Histórico: La Misión Mars Pathfinder de la NASA (1997)

El incidente más famoso de inversión de prioridad en la historia aeroespacial ocurrió en julio de 1997 a millones de kilómetros de la Tierra. El vehículo *Mars Pathfinder* comenzó a sufrir reinicios espontáneos periódicos de su ordenador de a bordo, paralizando la misión.
Diagnóstico: El sistema ejecutaba el RTOS VxWorks con tres hilos clave:

- τ_A (Alta prioridad): Hilo del bus de información (gestor de comunicaciones del bus de datos).
- τ_M (Prioridad media): Tareas de cómputo científico y comunicaciones de radio que tomaban mucho tiempo de CPU.
- τ_B (Baja prioridad): Tarea meteorológica de adquisición de datos (*ASI/MET thread*) que ocasionalmente tomaba el mutex del bus.

Cuando τ_B tomaba el mutex y τ_A intentaba acceder, τ_A se bloqueaba. En ese instante, las tareas intermedias τ_M expulsaban a τ_B. Un temporizador de vigilancia (*watchdog timer*) detectaba que el hilo de alta prioridad τ_A no se ejecutaba durante demasiado tiempo, asumía que el sistema se había congelado y forzaba un reinicio total de la sonda.

Solución enviada desde la Tierra: Los ingenieros de la NASA reprodujeron el fallo en réplicas terrestres y enviaron un parche de código que activó el **Protocolo de Herencia de Prioridad** en el mutex de VxWorks, resolviendo el problema definitivamente.

5.7.2 Solución 1: Sección Crítica No Expulsable (SCNE)

El enfoque más elemental consiste en elevar la prioridad de cualquier tarea a la prioridad máxima del sistema durante la ejecución de su sección crítica, o deshabilitar las interrupciones del procesador.

- **Cálculo del Bloqueo:** El tiempo de bloqueo B_i que sufre una tarea τ_i es como máximo la duración de la sección crítica más larga de cualquier tarea de menor prioridad:

$$B_i = \max_{j>i,k} \mathrm{Dur}(s_{j,k}) \tag{5.14}$$

- **Inconveniente:** Penaliza severamente a tareas de alta prioridad que no utilizan ningún recurso compartido, las cuales sufren bloqueos artificiales prolongados.

5.7.3 Solución 2: Protocolo de Herencia de Prioridad (PIP)

Definición: Protocolo de Herencia de Prioridad — PIP

Bajo el protocolo PIP (*Priority Inheritance Protocol*), si una tarea de baja prioridad τ_B mantiene bloqueado un recurso que es solicitado por una tarea de mayor prioridad τ_A, la tarea τ_B **hereda dinámicamente** la prioridad de τ_A mientras dure la retención del recurso.

Al heredar la prioridad máxima de las tareas a las que mantiene bloqueadas:

$$\mathrm{prio}_{\mathrm{act}}(\tau_B) = \max\left(\mathrm{prio}_{\mathrm{est}}(\tau_B), \max_{\tau_k \in \mathrm{bloqueadas}(\tau_B)} \mathrm{prio}(\tau_k)\right)$$

las tareas intermedias τ_M ya no pueden expulsar a τ_B, permitiendo que esta finalice rápidamente su sección crítica, libere el recurso y devuelva el control a τ_A.

Tipos de Bloqueo en PIP

- **Bloqueo Directo:** Cuando una tarea solicita un recurso que está actualmente ocupado.
- **Bloqueo Indirecto (*Push-through Blocking*):** Cuando una tarea de mayor prioridad se ve demorada porque una de menor prioridad está ejecutando una sección crítica con prioridad heredada para desbloquear a otra tarea.

Importante / Ojo Examen: Limitaciones Críticas de PIP

Aunque PIP mitiga la inversión de prioridad no acotada, sufre dos problemas graves:

1. **No evita el Interbloqueo (*Deadlock*):** Si dos tareas intentan acceder a dos recursos compartidos en orden inverso, PIP no previene el abrazo mortal.
2. **Bloqueo Encadenado (*Chained Blocking*):** Si una tarea de alta prioridad necesita acceder a m recursos compartidos diferentes, en el peor caso puede

sufrir el bloqueo secuencial sucesivo de m secciones críticas de tareas de menor prioridad.

5.7.4 Solución 3: Protocolos de Techo de Prioridad (PCP y PPP)

Para erradicar por completo los interbloqueos y el bloqueo encadenado, se introdujeron los protocolos basados en techo de prioridad (Sha, Rajkumar y Lehoczky, 1990).

Definición: Techo de Prioridad de un Recurso

El **techo de prioridad** (*Priority Ceiling*) de un recurso compartido R_k se define estáticamente como la máxima prioridad de todas aquellas tareas que pueden llegar a solicitarlo en algún momento de su ejecución:

$$\text{Ceiling}(R_k) = \max\{\text{prio}(\tau) \mid \tau \text{ utiliza el recurso } R_k\} \tag{5.15}$$

Protocolo de Techo de Prioridad Inmediato (Immediate Priority Ceiling / PPP)

En los estándares POSIX (`PTHREAD_PRIO_PROTECT`) y en el lenguaje Ada (objetos protegidos), el protocolo predominante es el **Techo de Prioridad Inmediato** (también conocido como *Priority Protect Protocol*, PPP):

Mecanismo del Techo de Prioridad Inmediato (PPP)

1. Cada tarea posee una prioridad estática base por defecto.
2. Cada recurso tiene asignado estáticamente su techo de prioridad $\text{Ceiling}(R_k)$.
3. Tan pronto como una tarea τ adquiere el cerrojo del recurso R_k, su prioridad dinámica se eleva **inmediatamente** a $\text{Ceiling}(R_k)$, sin esperar a que otra tarea intente acceder al recurso.
4. Cuando la tarea libera el recurso, su prioridad vuelve instantáneamente a su nivel previo a la entrada a la sección crítica.

Propiedades Demostradas de los Protocolos de Techo

1. **Ausencia Absoluta de Interbloqueo (*Deadlock-free*):** Queda matemáticamente garantizado que ningún conjunto de tareas puede caer en interbloqueo mutuo.
2. **Eliminación del Bloqueo Encadenado:** Una tarea de alta prioridad τ_i solo puede sufrir, **como máximo, el bloqueo de una única sección crítica** de cualquier tarea de menor prioridad durante toda su activación.

5.7.5 Cálculo del Factor de Bloqueo B_i y Extensión del Análisis R_i

Bajo los protocolos de techo de prioridad, el factor de bloqueo en el peor caso B_i que puede experimentar una tarea τ_i se calcula como la duración de la sección crítica más larga ejecutada por cualquier tarea de menor prioridad sobre un recurso cuyo techo sea igual o superior a la prioridad de τ_i:

$$B_i = \max_{\substack{j>i \\ k|\text{Ceiling}(R_k)\geq P_i}} \text{Dur}(s_{j,k}) \tag{5.16}$$

Incorporando el factor de bloqueo B_i, la ecuación de análisis del tiempo de respuesta en el peor caso de Joseph y Pandya se generaliza como:

$$R_i^{(0)} = C_i + B_i, \qquad R_i^{(k+1)} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j \tag{5.17}$$

El sistema con recursos compartidos es garantizadamente planificable si y solo si $\forall i, R_i \leq D_i$.

5.8 Tratamiento de Tareas Aperiódicas y Esporádicas

En casi todo sistema de tiempo real coexisten tareas de control periódicas con eventos externos aperiódicos (peticiones de usuario, alarmas de red, comandos de depuración). El desafío consiste en responder a las peticiones aperiódicas con la mínima latencia posible sin comprometer jamás las garantías temporales de las tareas periódicas críticas.

5.8.1 1. Servicio en Segundo Plano (*Background Processing*)

Es el mecanismo más elemental: las tareas aperiódicas se ejecutan con la prioridad más baja del sistema.

- **Ventajas:** No interfiere en absoluto con la planificabilidad de las tareas periódicas.
- **Inconvenientes:** Tiempos de respuesta para las tareas aperiódicas inaceptablemente largos, especialmente cuando la utilización periódica U_p es elevada. Si el sistema está al 100 % de carga periódica, las aperiódicas sufren inanición.

5.8.2 2. Servidor por Sondeo (*Polling Server*)

Se introduce en el sistema una tarea periódica ficticia $\tau_s = (C_s, T_s)$ denominada **servidor por sondeo**:

- En cada activación periódica (cada T_s), el servidor comprueba si hay peticiones aperiódicas en la cola.
- Si hay peticiones pendientes, las atiende consumiendo hasta un máximo de C_s unidades de tiempo de cómputo.
- **Deficiencia fundamental:** Si en el instante de activación del servidor la cola aperiódica está vacía, el servidor suspende inmediatamente su ejecución y **pierde todo su saldo de cómputo** C_s hasta el siguiente periodo T_s. Si una petición llega inmediatamente después (en $t = \epsilon$), deberá esperar un periodo completo hasta la próxima activación.

5.8.3 3. Servidor Diferido (*Deferred Server*)

El **Servidor Diferido** (Strosnider, Lehoczky y Sha, 1995) resuelve brillantemente la deficiencia del servidor por sondeo mediante la preservación de capacidad.

Definición: Servidor Diferido — DS

Un Servidor Diferido es una tarea periódica de alta prioridad caracterizada por una capacidad máxima de cómputo C_s y un periodo de recarga T_s. Si al activarse no existen peticiones aperiódicas, el servidor **conserva su capacidad intacta** durante todo el periodo T_s, disponible para atender inmediatamente cualquier petición que aparezca de manera imprevista.

El saldo de cómputo del servidor se recarga periódicamente a su valor nominal C_s al comienzo de cada periodo T_s (a menos que no se haya consumido, en cuyo caso no se acumula por encima de C_s).

Análisis de Planificabilidad del Servidor Diferido

Dado que el Servidor Diferido puede retener su saldo de cómputo y consumirlo en el instante más desfavorable (interfiriendo dos veces con una tarea periódica en una ventana temporal), el factor de utilización máximo tolerable por RMS para un conjunto de n tareas periódicas junto al servidor diferido debe corregirse analíticamente.

Definiendo la utilización del servidor $U_s = C_s/T_s$ y la utilización periódica $U_p = \sum_{i=1}^{n} C_i/T_i$, la cota superior de utilización modificada U_{mls} para n tareas viene dada por:

$$U_{mls}(n) = U_s + n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n} - 1\right] \tag{5.18}$$

Cuando el número de tareas periódicas tiende a infinito ($n \to \infty$), el límite asintótico se convierte en:

$$\lim_{n\to\infty} U_{mls}(n) = U_s + \ln\left(\frac{U_s+2}{2U_s+1}\right) \tag{5.19}$$

Condición de Planificabilidad con Servidor Diferido

El conjunto de n tareas periódicas junto al Servidor Diferido es garantizadamente planificable bajo RMS si:

$$U_p \leq \ln\left(\frac{U_s+2}{2U_s+1}\right) \tag{5.20}$$

Ejemplo Práctico: Diseño de un Servidor Diferido

Supongamos un sistema con tareas periódicas cuya utilización acumulada es $U_p = 0{,}50$ (50 %). Deseamos incorporar un Servidor Diferido de alta prioridad y determinar la máxima utilización U_s que podemos asignarle garantizando el cumplimiento de plazos periódicos.

Aplicamos la condición de planificabilidad asintótica:

$$0{,}50 \leq \ln\left(\frac{U_s+2}{2U_s+1}\right) \implies e^{0{,}50} \leq \frac{U_s+2}{2U_s+1}$$

Sabiendo que $e^{0{,}50} \approx 1{,}6487$:

$$1{,}6487(2U_s+1) \leq U_s+2 \implies 3{,}2974U_s + 1{,}6487 \leq U_s + 2$$

$$2{,}2974U_s \leq 2 - 1{,}6487 = 0{,}3513 \implies U_s \leq \frac{0{,}3513}{2{,}2974} \approx 0{,}1529 \quad (15{,}3\,\%)$$

Podemos dimensionar con total seguridad un Servidor Diferido con una utilización de hasta el 15,2 % (por ejemplo, con $T_s = 20$ ms y capacidad de cómputo $C_s = 3$ ms), garantizando tiempos de respuesta ultrarrápidos para eventos aperiódicos sin que ninguna tarea periódica falle jamás su plazo.

Síntesis del Tema y Fórmulas Clave de Examen

Formulario Rápido de Sistemas de Tiempo Real

1. **Condición necesaria monoprocesador:** $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$.
2. **Ejecutivo Cíclico:** $M = \text{mcm}(T_i)$, con restricciones sobre m:

$$m \geq \text{máx}(C_i), \quad M \pmod m = 0, \quad m \leq \text{mín}(D_i), \quad 2m - \gcd(m, T_i) \leq D_i$$

3. **Cota de Liu y Layland (RMS, $D_i = T_i$):**

$$U \leq n(2^{1/n} - 1) \xrightarrow{n\to\infty} \ln 2 \approx 69{,}3\,\%$$

4. **Análisis de Tiempo de Respuesta en el Peor Caso (R_i):**

$$R_i^{(k+1)} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j, \qquad R_i \leq D_i$$

5. **EDF ($D_i = T_i$):** Condición necesaria y suficiente: $U \leq 1$.
6. **Techo de Prioridad Inmediato (PPP):**

$$\text{Ceiling}(R_k) = \max_{\tau \text{ usa } R_k} \text{prio}(\tau), \quad B_i = \max_{\substack{j>i \\ k|\text{Ceiling}(R_k)\geq P_i}} \text{Dur}(s_{j,k}) \quad \text{(máximo 1 bloqueo)}$$

7. **Servidor Diferido (DS):**

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right)$$