**ÍNDICE**

1. [**Concurrencia**](#_gfd5dhem8ikw)…………………………………………………………………………………………..3

[**1.1** Concurrencia a nivel hardware](#_1ascrwjj09nv)

**1.2** [Posibles comportamientos de los procesos](#_zdmp9k3qvjv1)

**1.3** [Procesamiento secuencial, concurrente y paralelo](#_9uwuu7p4goog)

**1.4** [Programa concurrente](#_j6dv5q8abzz3)

**1.5** [Procesos e hilos](#_aa6b5vovusx7)

**1.6** [Secuencialidad y concurrencia](#_tljn662eagpl)

**1.7** [¿Cómo se mide el incremento de performance?](#_h604vxghjny1)

1. [**Conceptos básicos de concurrencia**](#_2nh69xaoqb07)………………………………………………………6

**2.1** [Comunicación entre procesos](#_aqs9fmhvq1q9)

**2.2** [Sincronización de procesos](#_c4n6z3ex69og)

**2.3** [Prioridad y granularidad](#_q65w8wurmy9h)

**2.4** [Manejo de recursos](#_5a9pomde4mld)

**2.5** [Deadlock](#_ouoypy7xsa3z)

**2.6** [Requerimientos para un lenguaje concurrente](#_ve04sup4l4j2)

**2.7** [Problemas asociados a la programación concurrente](#_r5q287m1227w)

**2.8** [Resumen de conceptos básicos](#_5wl32969uj4i)

1. [**Clases de instrucciones**](#_glubax8vmc56)………………………………………………………………………….10

**3.1** [Programación secuencial y concurrente](#_em76muuaajm)

1. [**Paradigmas de resolución de programas concurrentes**](#_21ndcn8fadzb)……………………...11
2. [**Clasificación de arquitecturas paralelas**](#_fbqfs5u206z7)……………………………………………….12

**5.1** [Por el mecanismo de control](#_z1hc54ms8gll)

**5.2** [Por la red de interconexión](#_p0slpk5vyjr3)

**5.3** [Por la granularidad](#_qkyquknjikdj)

**5.4** [Por la organización del espacio de direcciones](#_yim4ydtnvtrv)

1. [**Acciones atómicas y sincronización**](#_stqow85z2mk)…………………………………………………….14

**6.1** [Atomicidad de grano fino](#_qfde79j98y5o)

**6.2** [Propiedad de “a lo sumo una vez”](#_5erfjou0dijl)

**6.3** [Especificación de la sincronización](#_6eo6qa6yia4p)

1. [**Propiedades**](#_nwe1668izyns)…………………………………………………………………………………………..15
2. [**Fairness y políticas de scheduling**](#_jn20n4h2sox7)……………………………………………………….15
3. [**Sincronización por variables compartidas. Locks y barreras**](#_wfomi1f8n50v)…………...16

**9.1** [Problema de la sección crítica](#_dulghkqmk3u6)

**9.1.1** [Solución por hardware: deshabilitar interrupciones](#_angoeqgwm3y4)

**9.1.2** [Solución de grano grueso](#_3gdauadjeexm)

**9.1.3** [Solución de grano fino: “Spin Locks”](#_8f6iifo0rqju)

**9.1.4** [Implementación de sentencias await](#_re98m4f8uro7)

**9.1.5** [Solución fair: Algoritmo Tie-breaker](#_eipwef6p1i4m)

**9.1.6** [Solución fair: Algoritmo Ticket](#_wmshpxrazcrv)

**9.1.7** [Solución fair: Algoritmo Bakery](#_9x0nhyn719yr)

**9.2** [Sincronización Barrier](#_chtqpfqasxon)

**9.2.1** [Contador compartido](#_xrow0jhc5fc3)

**9.2.2** [Flags y coordinadores](#_f8ykgv4eh5h5)

**9.2.3** [Árboles](#_a9xm98sxyq0n)

**9.2.4** [Barreras simétricas](#_3yx5z4hu2kli)

**9.3** [Defectos de la sincronización por busy waiting](#_p6v9gthlruk)

1. [**Semáforos**](#_u14ezlluay58)……………………………………………………………………………………………..23

**10.1** [Problemas básicos y técnicas](#_mh4vnz749mtb)

**10.1.1** [Sección crítica: exclusión mutua](#_xg0xshhjsiz)

**10.1.2** [Barreras: señalización de eventos](#_6f4wnraba9sf)

**10.1.3** [Productores y consumidores: semáforos binarios divididos](#_sdehru9a18gd)

**10.1.4** [Buffers limitados: contadores de recursos](#_r5zfys90uqcg)

**10.1.5** [Problema de los filósofos: exclusión mutua selectiva](#_oedq9kwzq87e)

**10.1.6** [Lectores y escritores: exclusión mutua selectiva](#_c7qwc5wutlmg)

**10.1.7** [Técnica Passing the Baton](#_k8gexhrtrg42)

1. [**Alocación de recursos y scheduling**](#_9v5r7pksg7n2)…………………………………………………….28

**11.1** [Alocación Shortest-Job-Next (SJN)](#_b45sdoiirsqr)

1. [**Monitores**](#_x841ryqo9a35)……………………………………………………………………………………………...30

**12.1** [Notación](#_j4uyr95kmukn)

**12.2** [Sincronización](#_yilsf96r6utj)

**12.3** [Simulación de semáforos con monitores](#_tsrebdiwlfwl)

**12.4** [Técnicas de sincronización](#_a5zub4lx67qj)

**12.4.1** [Passing the condition](#_i4m8n3oz85qi)

**12.4.2** [Alocación SJN: wait con prioridad](#_iob51zsa1y1d)

**12.4.3**[Alocación SJN: variables condición privadas](#_a5q0c4a04am3)

**12.4.4** [Lectores y escritores: broadcast signal](#_etl2r268v2qi)

**12.4.5** [Diseño de un reloj biológico](#_lybphup3dro1)

**12.5** [Relación entre Passing the baton y Passing the condition](#_6yowc1m7p3yk)

1. [**Pasaje de mensajes**](#_pj1mgdmnsjge)……………………………………………………………………………...37

**13.1** [Conceptos generales](#_96epdh68dej)

**13.2** [Características](#_fdivedpowrg1)

**13.3** [Relación entre mecanismos de sincronización](#_c8dj6hdm4gpd)

**13.4** [Pasaje de mensajes asincrónicos (PMA)](#_gt19o9x1csw)

**13.4.1** [Características de los canales](#_ca2apiradht3)

**13.4.2** [Productores y consumidores (filtro). Red de ordenación](#_313wwdpdz2n)

**13.4.3** [Clientes y servidores. Monitores activos](#_gzoyvia8qgmx)

**13.4.4** [Pares (peers) interactuantes: Intercambio de valores](#_f6mju2y2jfpw)

**13.4.5** [Paradigmas para la interacción entre procesos](#_hoyeaeyylqqt)

**13.5** [Pasaje de mensajes sincrónicos (PMS)](#_3hhmd9m16jni)

**13.5.1** [Comentarios](#_3qnfkhegwy4l)

**13.5.2** [CSP - Lenguaje para PMS](#_vjvvnm48z1no)

**13.5.3** [MPI - Bibliotecas para manejo de PM](#_jz0k9hwveqz)

1. [**Conceptos de RPC y Rendezvous**](#_c1mfomifq8w)………………………………………………………...50

**14.1** [RPC](#_78o5oeeqxfsk)

**14.2** [RPC en Java. Remote Method Invocation (RMI)](#_xkfjflmhmk46)

**14.3** [Rendezvous](#_tgvn58oajc1l)

**14.3.1** [ADA](#_v3mi5il1v8b0)

**14.3.1.1** [Sincronización](#_n8pffin433xd)

**14.3.1.2** [Primitivas múltiples](#_g0a4rcdpzibi)

1. [**Programación paralela**](#_8ev4k7ocm29e)………………………………………………………………………..59

**15.1** [Computación científica](#_7qv7ri9w14el)

**15.2** [Diseño de algoritmos paralelos](#_ui7sy4bip1nk)

**15.2.1** [Métricas del paralelismo](#_r0ublccyxp7j)

**15.2.2**[Noción de granularidad](#_apqiwx1vmfdm)

**15.3**[Librería Phthreads](#_knk55tjh2z3m)

**15.3.1** [Semáforos con Phthreads](#_wgotlxllny70)

**15.3.2** [Monitores con Phthreads](#_dnq6c7cvgti3)

|  |
| --- |
| **1. Concurrencia** |

La concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente. Es un concepto de *software* no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores. Permite a distintos objetos actuar al mismo tiempo.

*¿Por qué es necesaria la Programación Concurrente?*

* No hay más ciclos de reloj, ahora hay **multicores**.
* Esto permite aplicaciones con **estructura más natural**. El mundo no es secuencial. Es más apropiado programar múltiples actividades independientes y concurrentes. Reacción a entradas asincrónicas (ej: sensores en un STR).
* Se obtiene una **mejora en la respuesta** → No bloquear la aplicación completa por E/S. Incremento en el rendimiento de la aplicación por mejor uso del hardware (ejecución paralela).
* Da lugar a **sistemas distribuidos** → Una aplicación en varias máquinas. Sistemas C/S o P2P.

*Los objetivos de los sistemas concurrentes son:*

* **Ajustar el modelo** de arquitectura de hardware y software al problema del mundo real a resolver.
* **Incrementar la performance**, mejorando los tiempos de respuesta de los sistemas de cómputo, a través de un enfoque de la arquitectura física y lógica de las soluciones.

*Ventajas:*

* La velocidad de ejecución que se puede alcanzar.
* Mejor utilización de la CPU de cada procesador.
* Explotación de la concurrencia inherente a la mayoría de los problemas reales.

|  |
| --- |
| 1.1 Concurrencia a nivel de hardware |

**Hay un límite físico en la velocidad de los procesadores**. Las máquinas monoprocesador ya no pueden mejorar, por lo que se optó por más procesadores por chip para tener mayor potencia de cómputo, pero para usarlos eficientemente se necesita *programación concurrente y paralela*.

* **Multiprocesadores de memoria compartida**:​ la interacción se da modificando datos almacenados en la MC. Esquemas UMA (uniform memory access, memoria compartida entre todos los procesadores, problemas de sincronización y consistencia) y NUMA (cada procesador tiene su memoria local, problemas de consistencia).
* **Multiprocesadores con memoria distribuida**: ​procesadores conectados por una red, c/u con su memoria local y la interacción es sólo por pasaje de mensajes.

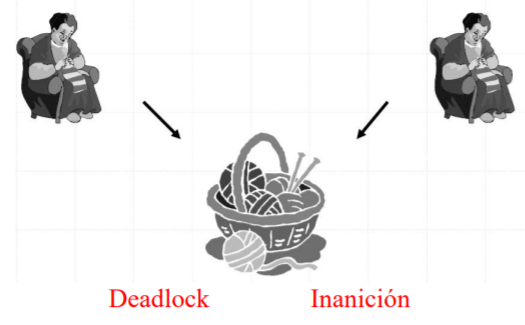
|  |
| --- |
| 1.2 Posibles comportamientos de los procesos |

Un **programa secuencial** (lo llamaremos **proceso**) tiene un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente. Tiene un único *thread* o flujo de control (monoprocesador).

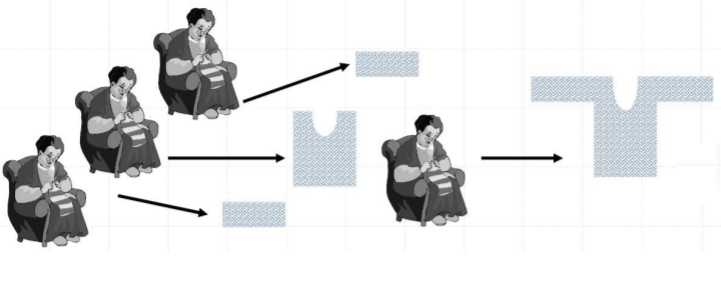
* **Procesos independientes**: relativamente raros y poco interesantes.



* **Competencia**: los procesos compiten, debido a los recursos compartidos.



* **Cooperación**: los procesos se combinan para resolver una tarea común. Sincronización.



|  |
| --- |
| 1.3 Procesamiento secuencial, concurrente y paralelo |

La solución secuencial nos fuerza a establecer un ***estricto orden temporal***. Al disponer de sólo una máquina, el ensamblado final del objeto se podrá realizar luego de N pasos de procesamiento (la fabricación de cada parte).

Si disponemos de ***N máquinas*** para fabricar el objeto, y *no hay dependencia* (por ejemplo de la materia prima), cada una puede trabajar **al mismo tiempo** en una parte → ***Solución paralela.***

* Consecuencias: menor tiempo para completar el trabajo. Menor esfuerzo individual. Paralelismo del hardware.
* Dificultades: Distribución de la carga de trabajo (diferente tamaño o tiempo de fabricación de cada parte). Necesidad de compartir recursos evitando conflictos. Necesidad de esperarse en puntos clave. Necesidad de comunicarse. Tratamiento de las fallas. Necesidad de decidir cuál de las máquinas será designada al ensamblado.

Otro enfoque: ***una sóla máquina*** dedica una parte del tiempo a cada componente del objeto → ***Concurrencia sin paralelismo de hardware*** → Menor speedup.

* Dificultades: Distribución de carga de trabajo. Necesidad de compartir recursos. Necesidad de esperarse en puntos clave. Necesidad de comunicarse. Necesidad de recuperar el “estado” de cada proceso al retomarlo.

Este último caso sería **multiprogramación** en un procesador. El tiempo de CPU es compartido entre varios procesos, por ejemplo por ***time slicing***. El sistema operativo controla y planifica procesos: si el *slice* expiró o el proceso se bloquea el sistema operativo hace ***context (process) switch***.

**Process switch**: suspender el proceso actual y restaurar otro

1. Salvar el estado actual en memoria. Agregar el proceso al final de la cola de ***ready*** o una cola de ***wait***.
2. Sacar un proceso de la cabeza de la cola ***ready***. Restaurar su estado y ponerlo a correr.

**Reanudar un proceso bloqueado**: mover un proceso de la cola de ***wait*** a la de ***ready***.

|  |
| --- |
| 1.4 Programa Concurrente |

Un **programa concurrente** especifica dos o más “programas secuenciales” que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos. Tiene múltiples *threads* o flujos de control

(como los procesos paralelos).

Un ***proceso*** o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite (por ejemplo los TASKs de ADA).

Un programa concurrente puede tener ***N procesos*** habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de ***M procesadores*** cada uno de los cuales puede ejecutar uno o más procesos.

**Características importantes**

* **No determinismo**: puede dar distintos resultados al ejecutarse sobre los mismos datos de entrada. Dificultad para la interpretación y debug.
* **Interacción**

**Concurrencia vs. Paralelismo**

La concurrencia no implica paralelismo.

|  |  |
| --- | --- |
| * **Concurrencia “Interleaved”**: Proceso lógicamente simultáneo. Ejecución intercalada en un único procesador. Pseudo-Paralelismo. |  |
| * **Concurrencia Simultánea**: Procesamiento físicamente simultáneo. Requiere un sistema multiprocesador o multicore. Paralelismo Full. |  |

|  |
| --- |
| 1.5 Procesos e hilos |

Cada ***proceso*** tiene su propio espacio de direcciones y recursos.

Un ***hilo* o *thread*** es un proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página). Todos los hilos de un proceso comparten el mismo espacio de direcciones y recursos (los del proceso). El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias. La concurrencia puede estar provista por el lenguaje (*ADA, Java*) o por el Sistema Operativo (*C/POSIX*).

|  |
| --- |
| 1.6 Secuencialidad y concurrencia |

Un programa secuencial es totalmente ordenado y **determinístico** → para los mismos datos de entrada, ejecuta siempre la misma secuencia de instrucciones y obtiene la misma salida.

Los programas concurrentes pueden ser **no determinísticos** → pueden dar resultados distintos al ejecutarse sobre los mismo datos de entrada.

El cómputo concurrente con mayor grado de concurrencia resulta en un *menor tiempo de ejecución*.

|  |
| --- |
| 1.7 ¿Cómo se mide el incremento de performance? |

El procesamiento paralelo lleva a los conceptos de *speedup* y *eficiencia*.

**Speedup**: Medida de la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores (comparando con el rendimiento al utilizar un solo procesador).

*Speed UP = Ts / Tp  ,* donde *Ts* → tiempo secuencial y *Tp →* tiempo paralelo

**Eficiencia**: es una medida relativa que permite la comparación de desempeño en diferentes entornos de computación paralela.

En la ejecución concurrente, el ***speedup*** es menor.

|  |
| --- |
| **2**. **Conceptos básicos de concurrencia** |

|  |
| --- |
| 2.1 Comunicación entre procesos |

La comunicación entre procesos concurrentes indica el modo en que se organiza y transmiten datos entre tareas concurrentes. Esta organización requiere especificar ***protocolos*** para controlar  
el progreso y la corrección. Los procesos se **COMUNICAN** a través de:

* Memoria compartida: Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a **bloquear y liberar** el acceso a la memoria. La solución más elemental es una variable de control tipo “semáforo” que habilite o no el acceso de un proceso a la memoria compartida.
* Pasaje de mensajes: Es necesario establecer un **canal** (lógico o físico) para transmitir información entre procesos. También el lenguaje debe proveer un protocolo adecuado. Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes para leer y cuando deben transmitir mensajes.

|  |
| --- |
| 2.2 Sincronización de procesos |

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. En la mayoría de los sistemas el tiempo absoluto no es importante. Con frecuencia los sistemas son actualizados con componentes más rápidas. La corrección del programa concurrente no debe depender del tiempo absoluto. El tiempo se ignora, sólo las secuencias son importantes. Puede haber distintos órdenes (***interleavings***) en que se ejecutan las instrucciones de los diferentes procesos; los programas deben ser correctos para todos ellos.

* Una ***acción atómica*** es una acción que hace una transformación de estado indivisible (estados intermedios invisibles para otros procesos).
* Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
* Ejecución de un programa concurrente → ***intercalado*** (*interleaving*) de las acciones atómicas ejecutadas por procesos individuales.
* ***Historia*** de un programa concurrente (*trace*): ejecución de un programa concurrente con un interleaving particular.
* En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas.

**El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas.**

Sincronización por exclusión mutua: asegura que solo un proceso tenga acceso a un recurso compartido en un instante de tiempo. Si el proceso tiene **secciones críticas** que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrase en la misma sección crítica al mismo tiempo.

Sincronización por condición: permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Una **interferencia** se da cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso. Se puede dar por ejemplo en programas concurrentes con memoria compartida. Para evitarse, los procesos se deben sincronizar.

|  |
| --- |
| 2.3 Prioridad y granularidad |

* Un proceso que tiene mayor **prioridad** puede causar la suspensión (*preemption*) de otro proceso concurrente. Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.
* La **granularidad** de una aplicación está dada por la relación entre el cómputo y la comunicación. Grano fino y grano grueso.

|  |
| --- |
| 2.4 Manejo de recursos |

Uno de los temas principales de la programación concurrente es la ***administración de recursos compartidos***, esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.

Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (**fairness**).

Dos situaciones no deseadas en los programas concurrentes son la **inanición** de un proceso (no logra acceder a los recursos compartidos) y el **overloading** de un proceso (a un proceso se le asigna un recurso que implican más trabajo que el de otro). Otro problema importante que se debe evitar es el ***deadlock***.

|  |
| --- |
| 2.5 Deadlock |

Dos (o más) procesos pueden entrar en *deadlock*, si por error de programación ambos se quedan esperando que el otro libere un recurso compartido. La ausencia de *deadlock* es una propiedad necesaria en los procesos concurrentes.

**4 propiedades necesarias y suficientes para que exista deadlock son:**

* Recursos reusables seriamente: los procesos comparten recursos que pueden usar con exclusión mutua.
* Adquisición incremental: los proceso mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.
* No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que solo son liberados voluntariamente.
* Espera cíclica: existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor está esperando adquirir.

|  |
| --- |
| 2.6 Requerimientos para un lenguaje concurrente |

Independientemente del mecanismo de comunicación / sincronización entre procesos, los ***lenguajes de programación concurrente*** deberán proveer primitivas adecuadas para la especificación e implementación de las mismas.

**Requerimientos**

* Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
* Mecanismos de sincronización .
* Mecanismos de comunicación entre los procesos.

|  |
| --- |
| 2.7 Problemas asociados con la programación concurrente |

* Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas.
* Los procesos iniciados dentro de un programa concurrente pueden NO estar “vivos”. Esta pérdida de la propiedad de ***liveness*** puede indicar deadlocks o una mala distribución de recursos.
* Hay un ***no determinismo*** implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas → ***dificultad para la interpretación y debug.***
* Posible reducción de performance por ***overhead*** de *context switch*, comunicación, sincronización.
* Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos secuenciales.
* Necesidad de adaptar el software concurrente al hardware paralelo para mejora real en el rendimiento

|  |
| --- |
| 2.8 Resumen de conceptos básicos |

* La **programación paralela** se asocia con la ejecución concurrente en múltiples procesadores que pueden tener memoria compartida, y generalmente con un objetivo de incrementar performance.
* La **concurrencia** es un concepto de software. En Programación Concurrente la organización de procesos y procesadores constituyen la arquitectura del sistema concurrente.
* ***Especificar la concurrencia es esencialmente especificar los procesos concurrentes, su comunicación y sincronización.***
* La **programación distribuida** es un caso especial de la programación concurrente en la que se cuenta con varios procesadores pero no se posee una memoria compartida y la comunicación está dada por el pasaje de mensajes.

|  |
| --- |
| **3**. **Clases de instrucciones** |

|  |
| --- |
| 3.1 Programación secuencial y concurrente |

Un programa concurrente está formado por un conjunto de programas secuenciales.

* La programación secuencial estructurada puede expresarse con 3 clases de  
  instrucciones básicas: ***asignación***, ***alternativa*** (decisión) e ***iteración*** (repetición  
  con condición).
* Se requiere una clase de instrucción para representar la concurrencia.

#### Declaraciones de variables

* Variable simple: tipo variable = valor . Ej: *int x = 8; int z, y;*
* Arreglos:
* *int a[10];*
* *int c[3:10]*
* *int b[10] = ([10] 2)*

##### Asignación

* Asignación simple: x = e
* Sentencia de asignación compuesta:
* *x = x + 1; y = y – 1; z = x + y*
* *a[3] = 6; aa[2,5] = a[4]*
* Llamado a funciones: *x = f(y) + g(6) - 7*
* swap: v1 :=: v2
* **skip**: termina inmediatamente y no tiene efecto sobre ninguna variable de programa.

##### 

Comunicación guardada: Con frecuencia un proceso se quiere comunicar con más de un proceso (quizás por distintos ports) y no sabe el orden en el cual los otros procesos podrían querer comunicarse con él. Las sentencias de comunicación guardada soportan comunicación no determinística.

##### Alternativa

* **Sentencias de alternativa simple**

*if B → S*

**B** expresión booleana.

**S** instrucción simple o compuesta ({}). ***B “guarda” a S*** pues S no se ejecuta si B no es verdadera. **R** es la sentencia de recepción.

* **Sentencias de alternativa múltiple**

*if B1 → S1*

*\* B2 → S2*

*.....*

*\* Bn → Sn*

*fi*

Las guardas se evalúan en algún orden arbitrario. Elección no determinística. Si ninguna guarda es verdadera el if no tiene efecto.

* **Otra opción**

*if (cond) S;*

*if (cond) S1 else S*

##### Iteración

* **Sentencias de alternativa iterativa múltiple:**

*do B1 → S1*

*\* B2 → S2*

*.....*

*\* Bn → Sn*

*od*

Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas. La elección es *no determinística* si más de una guarda es verdadera.

* **For-all:** forma general de repetición e iteración

*fa cuantificadores → Secuencia de Instrucciones af*

*Cuantificador ≡ variable := exp\_inicial to exp\_final st B*

El cuerpo del fa se ejecuta 1 vez por cada combinación de valores de las variables de iteración.

* **Otra opción**: *while (cond) S;*

*for [i = 1 to n, j = 1 to n st (j mod 2 = 0)]*

##### 

##### Concurrencia

* **Sentencia co**:  
   ***co S1 // ..... // Sn oc*** → Ejecuta las Si tareas concurrentemente.  
   ***co [i=1 to n] { a[i]=0; b[i]=0 } oc***→ Crea *n* tareas concurrentes.  
   Cuantificadores.  
  La ejecución del *co* termina cuando todas las tareas terminaron.
* **Process**: otra forma de representar concurrencia  
   ***process A {sentencias}*** → proceso único independiente.  
   ***process B [i=1 to n] {sentencias}***→ n procesos independientes.  
   Cuantificadores.
* **Diferencia**: ***process*** ejecuta en ***background***, mientras el código que contiene un ***co*** espera a que el proceso creado por la sentencia co termine antes de ejecutar la siguiente sentencia.

|  |
| --- |
| **4**. **Paradigmas de resolución de programas concurrentes** |

Si bien el número de aplicaciones es muy grande, en general los “patrones” de resolución concurrentes son pocos:

* **Paralelismo iterativo**: un programa consta de un conjunto de procesos (posiblemente idénticos) c/u de los cuales tiene 1 o más loops → cada proceso es un programa iterativo. Los procesos cooperan para resolver un único problema (por ejemplo un sistema de ecuaciones), pueden trabajar independientemente, y comunicarse y sincronizar por *memoria compartida* o *pasaje de mensaje*. Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.
* **Paralelismo recursivo**: el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (*dividir y conquistar*). Ejemplos clásicos son el “*sorting by merging*”, el cálculo de raíces en funciones continuas, problema del viajante.
* **Productor-consumidor (*pipelines o workflows*)**: estos esquemas muestran procesos que se comunican. Es habitual que estos procesos se organicen en pipes a través de los cuales fluye la información. Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente.
* **Cliente-servidor:** es el esquema dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente o con multithreading a varios. Mecanismos de invocación variados (*rendezvous, RPC, monitores*). El soporte distribuido puede ser simple (*LAN*) o extendido a la *WEB.*
* **Pares que interactúan (*interacting peers*)**: los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea y completar el objetivo. Este esquema permite mayor grado de asincronismo que *cliente-servidor*. Posibles configuraciones: grilla, pipe circular, uno a uno, arbitraria.

|  |
| --- |
| **5**. **Clasificación de las arquitecturas paralelas** |

|  |
| --- |
| 5.1 Por el mecanismo de control |

Propuesta por *Flynn*. Se basa en la manera en que las ***instrucciones*** son ejecutadas sobre los ***datos***.

* **SISD** (*Single Instruction Single Data)*: Instrucciones ejecutadas en secuencia, una por ciclo de instrucción. La memoria afectada es usada sólo por esta instrucción. Usada por la mayoría de los uniprocesadores. La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memo recibe y almacena datos en las escrituras, y brinda datos en las lecturas. Ejecución *determinística*.
* **SIMD** (*Single Instruction Multiple Data*): Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos. Los procesadores en general son muy simples. El *host* hace *broadcast* de la instrucción. Ejecución *sincrónica y determinística*. Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones. Adecuados para aplicaciones con alto grado de regularidad, (x ej. procesamiento de imágenes). Ejemplos de máquina SIMD: Array Processors. CM-2, Maspar MP-1 y 2, Illiac IV
* **MISD** (*Multiple Instruction Single Data*): Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes. Operación *sincrónica*. No son máquinas de propósito general.
* **MIMD** (*Multiple Instruction Multiple Data*): Cada procesador tiene su propio flujo de instrucciones y de datos → cada uno ejecuta su propio programa. Pueden ser de memoria compartida o distribuida. Ejemplos de máquina MIMD: nCube 2, iPSC, CM-5, Paragon XP/S, máquinas DataFlow, red de transputers.

Subclasificación:

* + **MPMD** (*multiple program multiple data*): cada procesador ejecuta su propio programa (ejemplo con PVM).
  + **SPMD** (*single program multiple data*): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).

|  |
| --- |
| 5.2 Por la red de interconexión |

Tanto en memoria compartida como en pasaje de mensajes las máquinas pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

* Las **redes estáticas** constan de *links* punto a punto. Típicamente se usan para máquinas de pasaje de mensajes.
* Las **redes dinámicas** están construidas usando *switches* y enlaces de comunicación. Normalmente para máquinas de memoria compartida*.*

El diseño de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de *startup*, *paths* estáticos o dinámicos, operación sincrónica o asincrónica, topología, costo, etc.).

|  |
| --- |
| 5.3 Por la granularidad de los procesadores |

Relación entre el número de procesadores y el tamaño de memoria total. Puede verse también como la relación entre cómputo y comunicación.

* De **grano grueso** (*coarse-grained*): pocos procesadores muy poderosos.
* De **grano fino** (*fine-grained*): gran número de procesadores menos potentes.
* De **grano medio** (*medium-grained*).

Si las aplicaciones tienen concurrencia limitada pueden usar eficientemente pocos procesadores, por lo que convienen máquinas de grano grueso. Las máquinas de grano fino son más efectivas en costo para aplicaciones con alta concurrencia.

|  |
| --- |
| 5.4 Por la organización del espacio de direcciones |

* **Multiprocesadores de memoria compartida:** Interacción modificando datos en la memoria compartida. Problema de consistencia.
* **Multiprocesadores con memoria distribuida:** Memoria local (no hay problemas de consistencia). Interacción es sólo por pasaje de mensajes.

|  |
| --- |
| **6. Acciones atómicas y sincronización** |

|  |
| --- |
| 6.1 Atomicidad de grano fino |

Una acción atómica de **grano fino** (*fine grained*) se debe implementar por hardware.

La operación *A= A + 3* no es atómica, ya que es necesario cargar el valor de A en un registro, sumarle 3 y volver a cargar el valor del registro en A.

Supondremos máquinas con las siguientes características:

* Los valores de los tipos básicos se almacenan en elementos de memoria leídos y escritos como acciones atómicas.
* Los valores se cargan en registros, se opera sobre ellos, y luego se almacenan los resultados en memoria.
* Cada proceso tiene su propio conjunto de registros (*context switching*).
* Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso.

**Interferencia**: un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

* Si una expresión ***e*** en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino.
* Si una asignación ***x = e*** en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación también será atómica.

***Normalmente los programas concurrentes no son disjuntos → es necesario establecer algún requerimiento más débil.***

Una **referencia crítica** en una expresión es una referencia a una variable que es modificada por otro proceso. Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

|  |
| --- |
| 6.2 Propiedad de “A lo sumo una vez” |

Una sentencia de asignación **x = e** **satisface la propiedad de A lo sumo una vez (ASV) si**

1. ***e*** contiene a lo sumo una referencia crítica y ***x*** no es referenciada por otro proceso, o
2. ***e*** no contiene referencias críticas, en cuyo caso ***x*** puede ser leída por otro proceso

Una expresion e que no está en una sentencia de asignación satisface la propiedad de ***“A lo sumo una vez”*** si no contiene más de una referencia crítica.

**Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez.**

Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución **parece atómica**, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos

|  |  |
| --- | --- |
| *int x=0, y=0;*  *co x=x+1 // y=y+1 oc;* | No hay ref. críticas en ningún proceso. En todas las historias x = 1 e y = 1 |
| *int x = 0, y = 0;*  *co x=y+1 // y=y+1 oc;* | El 1er proceso tiene 1 ref. crítica. El 2do ninguna. Siempre y = 1 y x = 1 o 2 |
| *int x = 0, y = 0;*  *co x=y+1 // y=x+1 oc;* | Ninguna asignación satisface ASV.  Posibles resultados: x= 1 e y =2 / x = 2 e y = 1.  Nunca debería ocurrir x = 1 e y = 1 **→ ERROR** |

|  |
| --- |
| 6.3 Especificación de la sincronización |

Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente.

En general, es necesario ejecutar secuencias de sentencias como una única acción atómica → ***Acción atómica de Grano Grueso***

Mecanismo de sincronización para construir una acción atómica de **grano grueso** (*coarse grained*) como secuencia de acciones atómicas de grano fino (*fine grained*) que aparecen como indivisibles.

* ***〈e〉*** indica que la expresión ***e*** debe ser evaluada atómicamente.
* ***〈await (B) S;〉*** se utiliza para especificar sincronización. La expresión booleana ***B*** especifica una condición de demora. ***S*** es una secuencia de sentencias que se garantiza que termina. Se garantiza que ***B*** es *true* cuando comienza la ejecución de ***S***. **Ningún estado interno de *S* es visible para los otros procesos.**

*Await* → Sentencia con alto poder expresivo, pero el costo de implementación en su forma general (*exclusión mutua y sincronización por condición*) es alto.

Una acción atómica de **grano grueso** es una especificación en alto nivel del comportamiento requerido de un programa que puede ser implementada de distintas maneras, dependiendo del mecanismo de sincronización disponible.

En una arquitectura de grano grueso se tienen pocos procesadores con mucha capacidad de procesamiento por lo que son más adecuados para programas de grano grueso en los cuales se tienen pocos procesos que realizan mucho procesamiento y requieren de poca comunicación.

|  |
| --- |
| **7. Propiedades** |

Una *propiedad* de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo. Toda propiedad puede ser formulada en términos de dos clases:

* **Seguridad** (*safety*): nada malo le ocurre a un objeto: asegura estados consistentes. Una *falla de seguridad* indica que algo anda mal.
* Ejemplos: ausencia de interferencia (exclusión mutua) entre procesos, ***partial correctness***.
* **Vida** (*liveness*):eventualmente ocurre algo bueno con una actividad: progresa, no hay *deadlocks*. Una *falla de vida* indica que las cosas se dejan de ejecutar.
* Ejemplos: ***terminación***, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su sección crítica, etc → *dependen de las políticas de scheduling.*

**Ejemplo**

Puente sobre río con ancho sólo para una fila de tráfico ⇒ los autos pueden moverse concurrentemente si van en la misma dirección.

* ***Violación de seguridad*** si dos autos en distintas direcciones entran al puente al mismo tiempo.
* ***Vida***: ¿cada auto tendrá *eventualmente* oportunidad de cruzar el puente?.

|  |
| --- |
| **8. Fairness y políticas de scheduling** |

**Fairness:** trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás.

Una acción atómica en un proceso es **elegible** si es la próxima acción atómica en el proceso que será ejecutado. Si hay varios procesos → hay varias acciones atómicas elegibles.

Una **política de scheduling** determina cuál será la próxima en ejecutarse.

* **Fairness Incondicional**. Una política de scheduling es incondicionalmente fair (o imparcial) si toda acción atómica incondicional que es elegible eventualmente es ejecutada.
* **Fairness Débil**. Una política de scheduling es débilmente fair si:

1. Es incondicionalmente fair y
2. Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve ***true*** y permanece ***true*** hasta que es vista por el proceso que ejecuta la acción atómica condicional.

No es suficiente para asegurar que cualquier sentencia ***await*** elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de ***false*** a ***true*** y nuevamente a ***false***) mientras un proceso está demorado.

* **Fairness Fuerte**: Una política de scheduling es fuertemente fairsi:

1. Es incondicionalmente fair y
2. Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en ***true*** con infinita frecuencia.

No es simple tener una política que sea práctica y fuertemente fair. *Round-robin* es práctica pero no es fuertemente fair.

|  |
| --- |
| **9. Sincronización por variables compartidas. Locks y barreras** |

**Problema de la sección crítica:** implementación de acciones atómicas en software (***locks***)

**Barrera**: punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.

En la técnica de ***busy waiting*** un proceso chequea repetidamente una condición hasta que sea verdadera.

* La ventaja está en la implementación, que se realiza con instrucciones de cualquier procesador.
* Es ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es *interleaved*).
* Es aceptable si cada proceso ejecuta en su procesador.

El término ***busy waiting*** tiende a significar que está dispuesto a girar (*spin*) y esperar un cambio en un registro de hardware o una ubicación de memoria. El término no necesariamente significa bloqueo, pero implica esperar en un círculo cerrado, buscando repetidamente un cambio. Entonces, se implementa un ***spin-lock*** usando ***busy waiting***.

|  |
| --- |
| 9.1 Problema de la sección crítica |

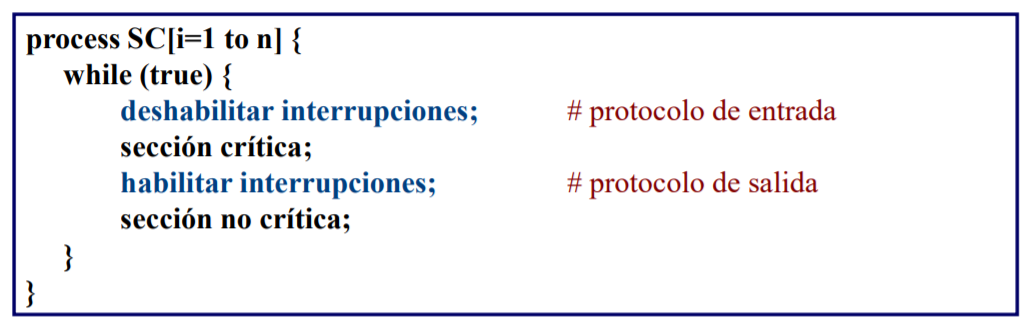
**Sección crítica**: n procesos repetidamente ejecutan una sección crítica de código, luego una sección no crítica. La sección crítica está precedida por un protocolo de entrada y seguido de un protocolo de salida. Cada sección crítica es una secuencia de sentencias que acceden a algún objeto compartido. Cada sección no crítica es otra secuencia de sentencias. Asumimos que un proceso que entra en su sección crítica eventualmente sale; así, un proceso solo puede finalizar fuera de su sección crítica.

Propiedades a cumplir:

* **Exclusión mutua**. A lo sumo un proceso está en su SC. (*propiedad de seguridad*)
* **Ausencia de Deadlock (Livelock):** Si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito. (*propiedad de seguridad*)
* **Ausencia de Demora Innecesaria:** Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC. (*propiedad de seguridad*)
* **Eventual Entrada:** Un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará). (*propiedad de vida*)

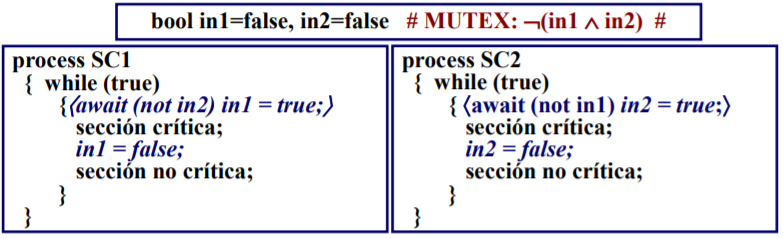
La solución trivial es hacer atómica la sección crítica → **〈SC〉**, pero cómo se implementan los **〈〉**?

|  |
| --- |
| 9.1.1 Solución por hardware: deshabilitar interrupciones |

****

* Solución correcta para una máquina monoprocesador.
* Durante la SC no se usa la multiprogramación → penalización de performance.
* La solución no es correcta en un multiprocesador.

|  |
| --- |
| 9.1.2 Solución de grano grueso |

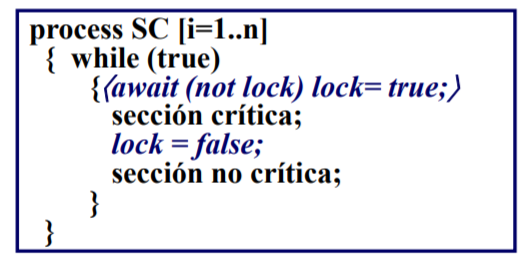


¿Satisface las 4 propiedades?

* **Exclusión mutua**: por construcción, P1 y P2 se excluyen en el acceso a la SC.
* **Ausencia de deadlock:** si hay deadlock, P1 y P2 están bloqueados en su protocolo de entrada ⇒ in1 e in2 serían *true* a la vez. Esto NO puede darse ya que ambas son falsas en ese punto (lo son inicialmente, y al salir de SC, cada proceso vuelve a serlo).
* **Ausencia de demora innecesaria**: si P1 está fuera de su SC o terminó, in1 es *false*; si P2 está tratando de entrar a SC y no puede, in1 es *true*; (¬ in1 ∧ in1 = *false*) ⇒ no hay demora innecesaria.
* **Eventual Entrada**:
* Si P1 está tratando de entrar a su SC y no puede, P2 está en SC (in2 es *true*). Un  
  proceso que está en SC eventualmente sale → in2 será *false* y la guarda de P1 *true*.
* Análogamente para P2.
* Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito,  
  las guardas son true con infinita frecuencia.

***Se garantiza la eventual entrada con una política de scheduling fuertemente fair.***

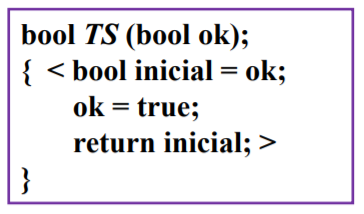
Si hay n procesos → cambio de variables: ***bool lock= false;***



|  |
| --- |
| 9.1.3 Solución de grano fino: “Spin Locks” |

* **Objetivo**: hacer “atómico” el await de grano grueso.
* **Idea**: usar instrucciones como *Test & Set* (**TS**), *Fetch & Add* (**FA**) o *Compare & Swap*, disponibles en la mayoría de los procesadores.

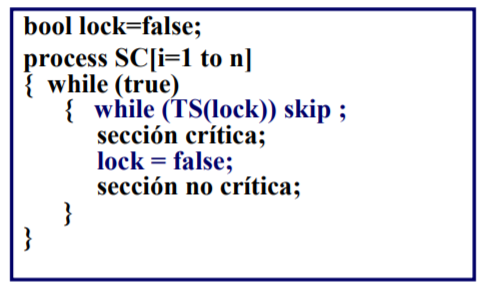
***Test & Set***



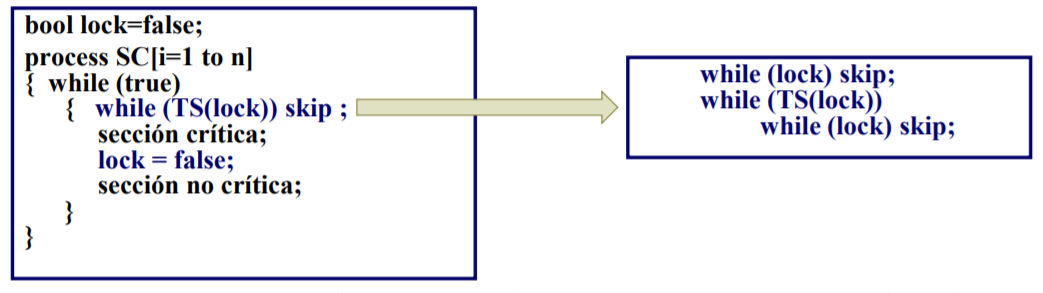
***Spin Locks:*** los procesos se quedan iterando (*spinning*) mientras esperan que se limpie ***lock****.*

Cumple las 4 propiedades si el *scheduling es* ***fuertemente fair***. Una política *débilmente fair* es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC).

* Baja performance en multiprocesadores si varios procesos compiten por el acceso.
* *lock* es una variable compartida y su acceso continuo es muy costoso (“*memory contention*”).
* Además, podría producirse un alto *overhead* por caché inválida.



**TS** escribe siempre en lock aunque el valor no cambie ⇒ Mejor ***Test-and Test-and-Set***

**

***Memory contention*** se reduce, pero no desaparece. En particular, cuando *lock* pasa a false posiblemente todos intenten hacer TS.

*Spin locks* no controla el orden de los procesos demorados, es posible que alguno no entre nunca si el scheduling no es *fuertemente fair* (***race conditions***)

|  |
| --- |
| 9.1.4 Implementación de sentencias *await* |

* Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional **〈S;〉:**

***SCEnter;*** *#protocolo de entrada a la SC*

***S;***

***SCExit;*** *#protocolo de salida de la SC*

* Para una acción atómica condicional ***〈await (B) S;〉:***

***SCEnter;***

***while (not B) {SCExit; SCEnter;}***

***S;***

***SCExit;***

Si **S** es ***skip***, y **B** cumple ASV*, 〈await (B);〉* puede implementarse por medio de ⇒ ***while (not B) skip***

**Correcto**, pero **ineficiente**: un proceso está *spinning* continuamente saliendo y entrando a SC hasta que otro altere una variable referenciada en B.

* Para reducir contención de memoria:

***SCEnter ;***

***while (not B) {SCExit; Delay; SCEnter;}***

***S;***

***SCExit;***

|  |
| --- |
| 9.1.5Solución Fair: Algoritmo Tie-breaker |

Protocolo de SC que requiere scheduling sólo *débilmente fair* y no usa instrucciones especiales, pero es más complejo.

Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada ⇒ esta última variable es compartida y de acceso protegido.

Demora (quita prioridad) al último en comenzar su *entry protocol*.

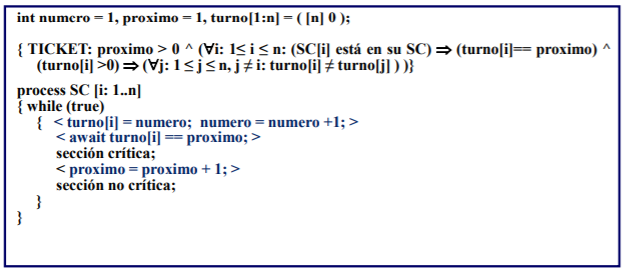
Generalización a n procesos:

* Si hay n procesos, el protocolo de entrada en cada uno es un loop que itera a través de n-1 etapas.
* En cada etapa se usan instancias de *tie-breaker* para dos procesos para determinar cuáles avanzan a la siguiente etapa.
* Si a lo sumo a un proceso a la vez se le permite ir por las n-1 etapas → a lo sumo uno a la vez puede estar en la SC.

Tie-Breaker para muchos procesos es *costoso en tiempo y complejo.*

|  |
| --- |
| 9.1.6Solución Fair: Algoritmo Ticket |

Se reparten números y se espera turno. Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los procesos con número más chico han sido atendidos.



Potencial problema: los valores de próximo y turno son ilimitados. En la práctica, podrían resetearse a un valor chico (por ejemplo, 1).

Cumplimiento de las propiedades:

* El predicado *TICKET* es un invariante global, pues **número** es leído e incrementado en una acción atómica y **próximo** es incrementado en una acción atómica ⇒ hay a lo sumo un proceso en la SC.
* La ausencia de *deadlock* y de demora innecesaria resultan de que los valores de **turno** son únicos.
* Con *scheduling débilmente fair* se asegura eventual entrada

El ***await*** puede implementarse con *busy waiting* (la expresión booleana referencia una sola variable compartida). El incremento de **próximo** puede ser un load/store normal (a lo sumo un proceso puede estar ejecutando su protocolo de salida).

Si no existe la instrucción *FA* la solución puede no ser *fair*.

***FA(var,incr): < temp = var; var = var + incr; return(temp) >***

|  |
| --- |
| 9.1.7Solución Fair: Algoritmo Bakery |

Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna uno mayor. Luego espera a que su número sea el menor de los que esperan. ***Los procesos se chequean entre ellos y no contra un global***.

Este algoritmo es más complejo, pero es *fair* y no requiere instrucciones especiales. No requiere un contador global **próximo** que se “entrega” a cada proceso al llegar a la SC. Esta solución de grano grueso no es implementable directamente.

|  |
| --- |
| 9.2 Sincronización Barrier |

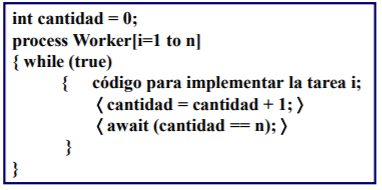
**Algoritmo Iterativo**: computan sucesivas mejores aproximaciones a una respuesta, y terminan al encontrarla o al converger. En cada iteración todos los procesos realizan el mismo trabajo sobre diferentes datos y requieren que haya finalizado el paso previa.

**Sincronización barrier**: el punto de demora al final de cada iteración es una barrera a la que deben llegar todos antes de permitirles pasar. Son un punto de encuentro (sincronización) en donde todos los procesos se demoran hasta que todos hayan llegado y luego de eso puedan continuar su ejecución.

|  |
| --- |
| 9.2.1 Contador compartido |

n procesos necesitan encontrarse en una barrera:

* Cada proceso incrementa una variable *Cantidad* al llegar.
* Cuando *Cantidad* es n los procesos pueden pasar.



Se puede implementar con: *FA(cantidad,1); while (cantidad <> n) skip*;

Problemas: *cantidad* necesita ser 0 en cada iteración, puede haber contención de memoria, coherencia de caché.

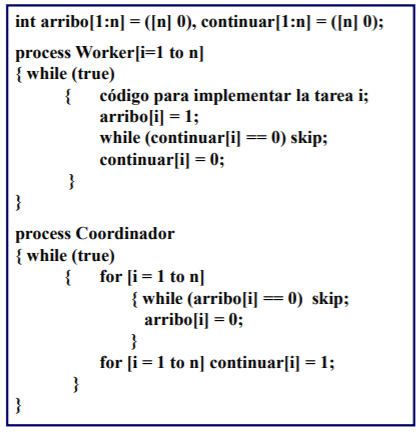
|  |
| --- |
| 9.2.2 Flags y coordinadores |

* Si no existe FA → Puede distribuirse Cantidad usando n variables

***(arreglo arribo[1..n])***

* El await pasaría a ser:  
  ***〈await (arribo[1] + … + arribo[n] == n);〉***
* Reintroduce contención de memoria y es ineficiente.

Puede usarse un conjunto de valores adicionales y un proceso más → Cada *Worker* espera por un único valor.



|  |
| --- |
| 9.2.3 Árboles |

Problema de la solución anterior:

* Requiere un proceso (y procesador) extra.
* El tiempo de ejecución del coordinador es proporcional a n.

Posible solución:

* Combinar las acciones de *Workers y Coordinador*, haciendo que cada *Worker* sea también *Coordinador*.
* Por ejemplo, *Workers* en forma de árbol: las señales de arribo van hacia arriba en el árbol, y las de continuar hacia abajo ⇒ ***combining tree barrier*** (más eficiente para n grande)

|  |
| --- |
| 9.2.4 Barreras simétricas |

En ***combining tree barrier*** los procesos juegan diferentes roles.  
Una ***Barrera Simétrica*** para n procesos se construye a partir de pares de barreras simples para dos procesos:

¿Cómo se combinan para construir una barrera n procesos? → ***Butterfly Barrier***

* ***log2n*** etapas: cada worker sincroniza con uno distinto en cada etapa.
* En la etapa s, un worker sincroniza con otro a distancia 2s-1.
* Cuando cada worker pasó ***log2n*** etapas, todos pueden seguir.

|  |
| --- |
| 9.3 Defectos de la sincronización por busy waiting |

**Protocolos “*busy waiting*”**

* complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados.
* Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.
* Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso ***spinning*** puede ser usado de manera más productiva por otro proceso.

**Necesidad de herramientas para diseñar protocolos de sincronización.**

|  |
| --- |
| **10. Semáforos** |

Instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones atómicas:

* ***P(s):*** Señala la ***ocurrencia de un evento*** (incrementa).

*〈 await s > 0 → s := s - 1 〉*

* ***V(s):*** Se usa para demorar un proceso ***hasta que ocurra un evento*** (decrementa)

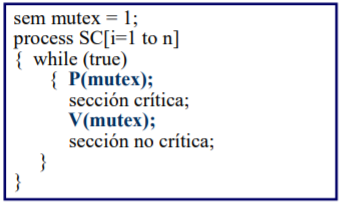
*〈 s := s + 1 〉*

Permiten proteger Secciones Críticas y pueden usarse para implementar *Sincronización por Condición*.

|  |
| --- |
| 10.1 Problemas básicos y técnicas |

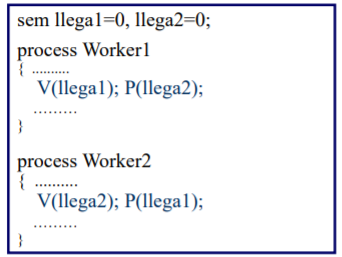
|  |
| --- |
| 10.1.1 Sección crítica: exclusión mutua |

Con un semáforo inicializado en uno podemos solucionar el problema de acceso a las secciones críticas.



|  |
| --- |
| 10.1.2 Barreras: señalización de eventos |

* **Idea:** un semáforo para cada *flag* de sincronización. Un proceso setea el flag ejecutando ***V***, y espera a que un *flag* sea seteado y luego lo limpia ejecutando ***P*.**
* **Barrera para dos procesos:** Necesitamos saber cada vez que un proceso llega o parte de la barrera → *relacionar los estados de los dos procesos*.
* **Semáforo de señalización:**  generalmente inicializado en 0.Un proceso señala el evento con V(s); otros procesos esperan la ocurrencia del evento ejecutando P(s).



|  |
| --- |
| 10.1.3 Productores y consumidores: semáforos binarios divididos |

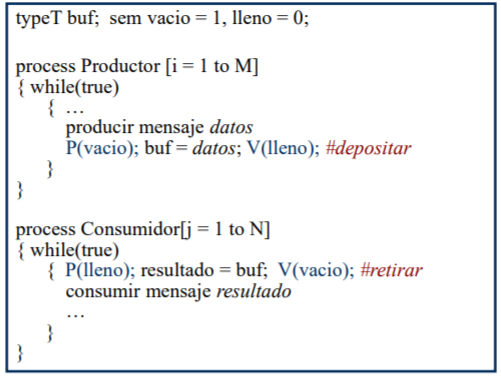
* **Semáforo Binario Dividido** *(Split Binary Semaphore)*. Los semáforos binarios b1, ...., bn forman un SBS en un programa si el siguiente es un invariante global:

***SPLIT: 0 ≤ b1, + ... + bn ≤ 1***

* Los **bi** pueden verse como un único semáforo binario **b** que fue dividido en **n** semáforos binarios.
* Importantes por la forma en que pueden usarse para implementar EM (en general la ejecución de los procesos inicia con un **P** sobre un semáforo y termina con un **V** sobre otro de ellos).
* Las sentencias entre el **P** y el **V** ejecutan con exclusión mutua.

Ejemplo: buffer unitario compartido con múltiples productores y consumidores. Dos operaciones: depositar y retirar que deben alternarse. Dos semáforos (vacío y lleno).

Inicializar vacío en 1 y lleno en 0.

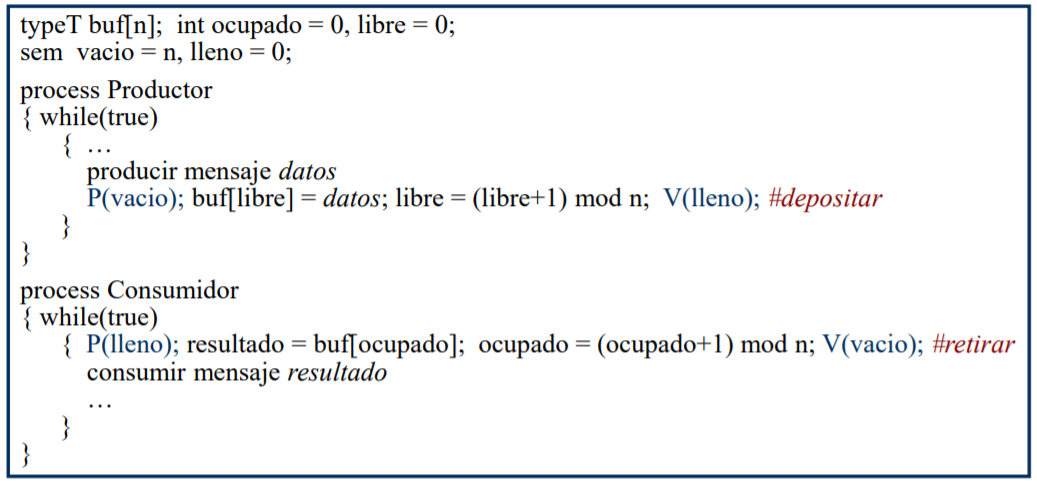


vacío y lleno (juntos) forman un “**semáforo binario dividido**”

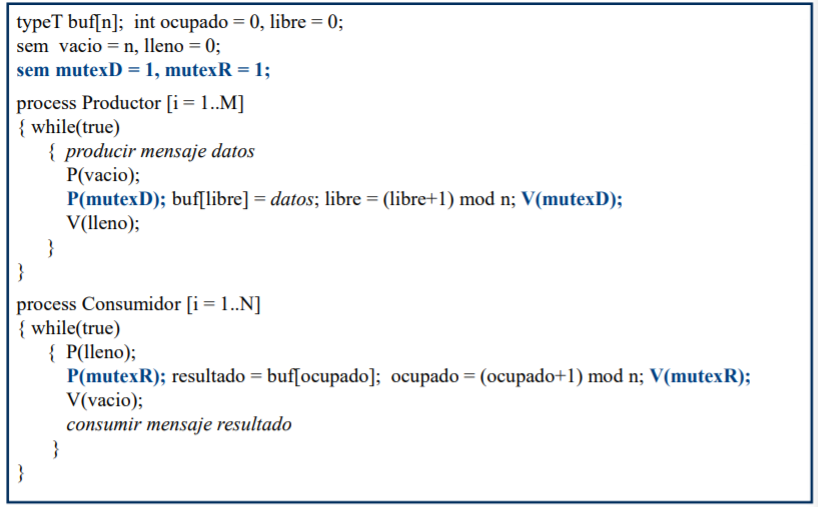
|  |
| --- |
| 10.1.4 Buffers limitados: contadores de recursos |

* **Contadores de Recursos**: cada semáforo cuenta el número de unidades libres de un recurso determinado. Esta forma de utilización es adecuada cuando los procesos compiten por recursos de *múltiples unidades*.

Ejemplo: un buffer es una cola de mensajes depositados y aún no buscados. Existe UN productor y UN consumidor que depositan y retiran elementos del buffer.

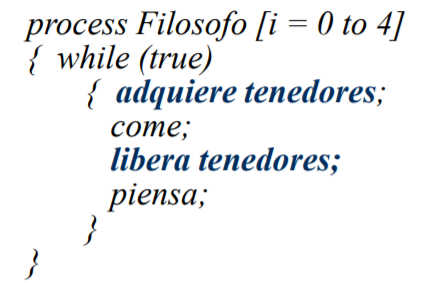


* ***vacio*** cuenta los lugares libres, y ***lleno*** los ocupados.
* ***depositar*** y ***retirar*** se pudieron asumir atómicas pues sólo hay un productor y un consumidor.
* Si hay más de un productor y/o más de un consumidor, las operaciones de depositar y retirar en sí mismas son SC y deben ejecutar con Exclusión Mutua. Si no se protege cada *slot*, podría retirarse dos veces el mismo dato o perderse datos por sobreescritura.

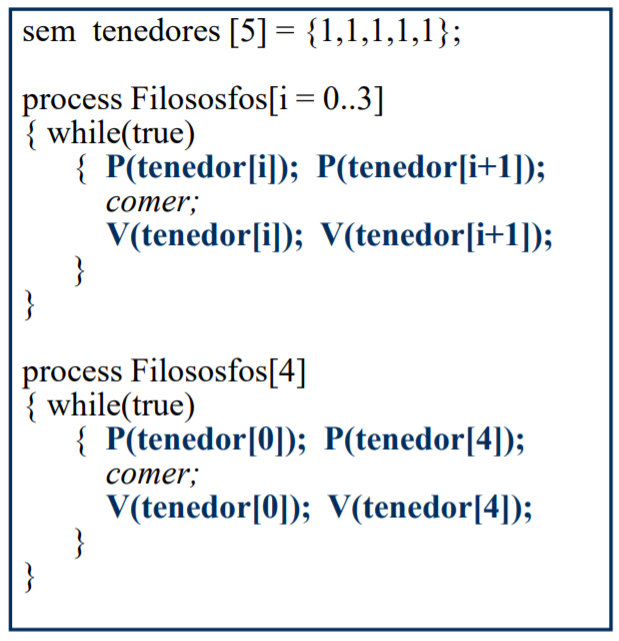


|  |
| --- |
| 10.1.5 Problema de los filósofos: exclusión mutua selectiva |

* Problema de varios procesos (**P**) y varios recursos (**R**) cada uno protegido por un lock.
* Un proceso debe adquirir los ***locks*** de todos los recursos que necesita.
* Puede caerse en deadlock cuando varios procesos compiten por conjuntos superpuestos de recursos.
* Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de recursos compartidas



* **Cada tenedor es una SC**: puede ser tomado por un único filósofo a la vez ⇒ pueden representarse los tenedores por un arreglo de semáforos.
* Levantar un tenedor ⇒ **P** Bajar un tenedor ⇒ **V**
* Cada filósofo necesita el tenedor izquierdo y el derecho.
* ¿Qué efecto puede darse si todos los filósofos hacen **exactamente** lo mismo?.



Si todos los filósofos levantaron un tenedor, luego ninguno puede levantar su segundo. Una condición necesaria para *deadlock* es que haya *waiting* circular, es decir, un proceso está esperando un recurso tomado por un segundo, el cual espera un recurso tomado por un tercero, y así siguiendo hasta un proceso que espera un recurso tomado por el primero. Así, para evitar *deadlock* es suficiente asegurar que no puede ocurrir *waiting* circular. Para este problema, una aproximación es tener uno de los procesos, digamos Filosofo[5], levantando sus tenedores en orden inverso.

Un problema es de **exclusión mutua selectiva** cuando cada proceso compite por el acceso a los recursos no con todos los demás procesos sino con un subconjunto de ellos. El problema de los filósofos es de exclusión mutua selectiva ya que para comer un filósofo no compite con todos los demás sino que solo compite con sus adyacentes. En el caso de que fueran 3 filósofos y no 5 como en la definición del problema general, no sería de exclusión mutua selectiva ya que un proceso compite con todos los demás procesos y no solo con un subconjunto de ellos, dado que el resto de los procesos son sus adyacentes.

|  |
| --- |
| 10.1.6 Lectores y escritores: exclusión mutua selectiva |

**Problema**: dos clases de procesos (lectores y escritores) comparten una Base de Datos. El acceso de los escritores debe ser exclusivo para evitar interferencia entre transacciones. Los lectores pueden ejecutar concurrentemente entre ellos si no hay escritores actualizando.

* Procesos **asimétricos** y, según el *scheduler*, con diferente prioridad.
* Es también un problema de ***exclusión mutua selectiva***: clases de procesos compiten por el acceso a la BD.

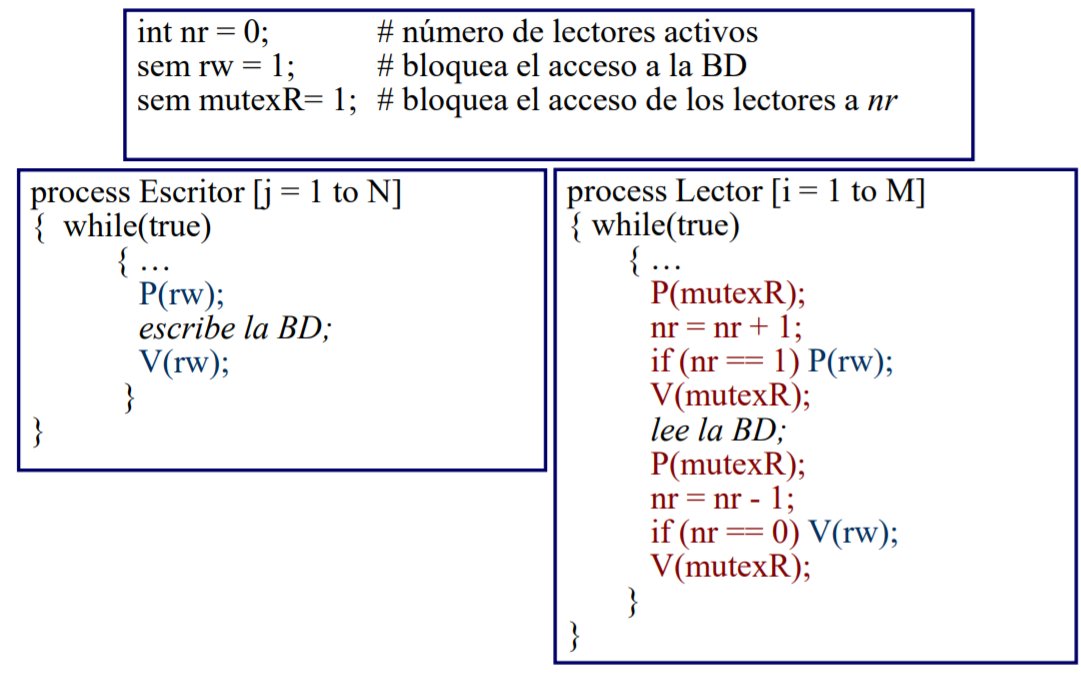
Diferentes soluciones:

1. **Como problema de exclusión mutua.**

Los escritores necesitan acceso mutuamente exclusivo. Los lectores (como grupo) necesitan acceso exclusivo con respecto a cualquier escritor.

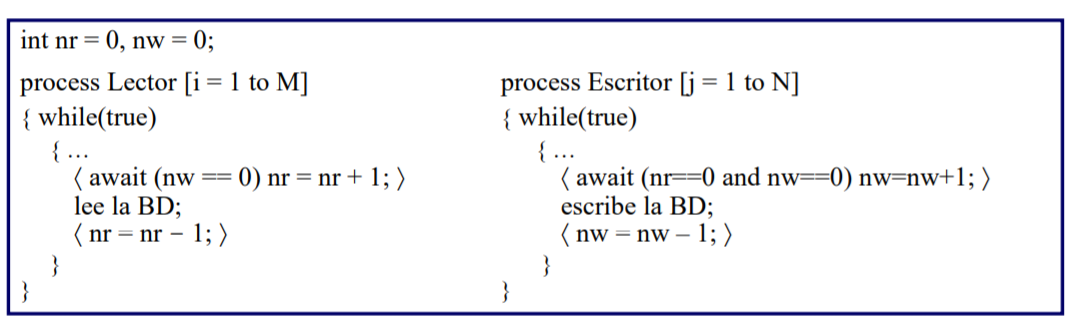
*No hay concurrencia entre lectores:*

* Los lectores (como grupo) necesitan bloquear a los escritores, pero sólo el primero necesita tomar el lock ejecutando *P(rw)*.
* Análogamente, sólo el último lector debe hacer *V(rw)*.



1. **Como problema de sincronización por condición.**

* Solución anterior ⇒ preferencia a los lectores ⇒ no es ***fair***.
* Otro enfoque ⇒ introduce la técnica ***passing the baton***: emplea *SBS* para brindar exclusión y despertar procesos demorados.
* Puede usarse para implementar ***await*** arbitrarios, controlando de forma precisa el orden en que los procesos son despertados
* En este caso, pueden contarse (por medio de *nr* y *nw*) los procesos de cada clase intentando acceder a la BD, y luego restringir el valor de los contadores. ¿Cuáles son los estados buenos y malos de ***nr*** y ***nw***?



|  |
| --- |
| 10.1.7 Técnica Passing the baton |

En algunos casos, ***await*** puede ser implementada directamente usando semáforos u otras operaciones primitivas. Pero no siempre. En el caso de las guardas de los *await* en la solución anterior, se superponen en que el protocolo de E/ para escritores necesita que tanto ***nw*** como ***nr*** sean 0, mientras para lectores sólo que ***nw*** sea 0. Ningún semáforo podría discriminar entre estas condiciones → ***Passing the baton***.

**Passing the baton**: técnica general para implementar sentencias ***await***. Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar. Cuando el proceso llega a un ***SIGNAL*** (sale de la SC), pasa el *baton* (control) a otro proceso. Si ningún proceso está esperando por el baton (es decir esperando entrar a la SC) el *baton* se libera para que lo tome el próximo proceso que trata de entrar.

* La sincronización se expresa con sentencias atómicas de la forma:

***F1 : 〈 Si 〉 o F2 : 〈 await (Bj ) Sj 〉***

Puede hacerse con semáforos binarios divididos (SBS).

* ***e*** semáforo binario inicialmente **1** (controla la entrada a sentencias atómicas). Utilizamos un semáforo ***bj*** y un contador ***dj*** cada uno con guarda diferente ***Bj*** ; todos inicialmente 0.
  + ***bj*** se usa para demorar procesos esperando que ***Bj*** sea true.
  + ***dj*** es un contador del número de procesos demorados sobre ***bj****.*
  + ***e*** y los ***bj*** se usan para formar un ***SBS***: a lo sumo uno a la vez es 1, y cada camino de ejecución empieza con un **P** y termina con un único **V**.

|  |
| --- |
| **11. Alocación de recursos y scheduling** |

El problema es decidir cuándo se le puede dar a un proceso determinado acceso a un recurso (cualquier objeto, elemento, componente, dato, SC, por la que un proceso puede ser demorado esperando adquirirlo).

Definición del problema: procesos que compiten por el uso de unidades de un recurso compartido (cada unidad está **libre** o en **uso**).

***request (parámetros):*** *〈await (request puede ser satisfecho) tomar unidades;〉*

***release (parámetros):*** *〈retornar unidades;〉*

Puede usarse ***Passing the Baton***:

***request*** *(parámetros): P(e);*

*if (request no puede ser satisfecho) DELAY;*

*tomar las unidades;*

*SIGNAL;*

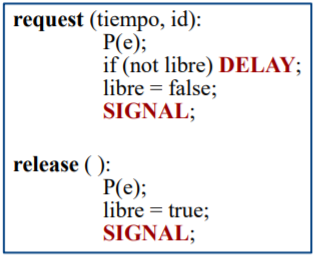
***release*** *(parámetros): P(e);*

*retornar unidades;*

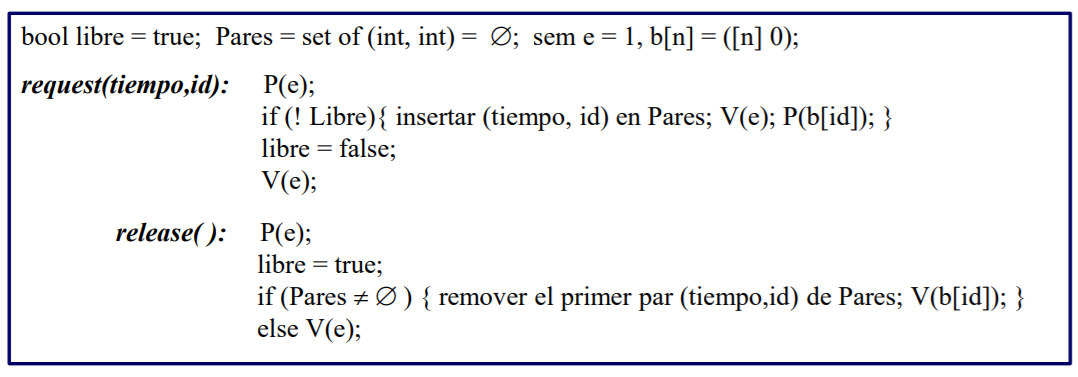
*SIGNAL;*

|  |
| --- |
| 11.1 Alocación Shortest-Job-Next (SJN) |

* Varios procesos que compiten por el uso de un recurso compartido de **una sola unidad**.
* ***request (tiempo,id)***. Si el recurso está libre, es alocado inmediatamente al proceso ***id***; sino, el proceso ***id*** se demora.
* ***release ( )***. Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) con el mínimo valor de tiempo. Si dos o más procesos tienen el mismo valor de tiempo, el recurso es alocado al que esperó más.
* **SJN** minimiza el tiempo promedio de ejecución, aunque no es ***fair****.* Puede mejorarse con la técnica de ***aging*** (dando preferencia a un proceso que esperó mucho tiempo).
* En **SJN**, un proceso que invoca a request debe demorarse hasta que el recurso esté libre y su pedido sea el próximo en ser atendido de acuerdo a la política. El parámetro tiempo entra en juego sólo si un pedido debe ser demorado.



* En ***DELAY*** un proceso inserta sus parámetros en un conjunto, cola o lista de espera (pares); libera la SC ejecutando *V(e)*; se demora en un semáforo hasta que *request* puede ser satisfecho.
* En ***SIGNAL*** un proceso: Cuando el recurso es liberado, si pares no está vacío, el recurso es asignado a un proceso de acuerdo a SJN
* Cada proceso tiene una condición de demora distinta, dependiendo de su posición en ***pares***. El proceso ***id*** se demora sobre el semáforo ***b[id]***.



**s** es un ***semáforo privado*** si exactamente un proceso ejecuta operaciones **P** sobre **s**. Resultan útiles para señalar procesos individuales. Los semáforos ***b[id]*** son de este tipo.

|  |
| --- |
| **12. Monitores** |

En semáforos existen variables compartidas globales a los procesos, sentencias de control de acceso a la **sección crítica** dispersas en el código. Además al agregar procesos, se debe verificar acceso correcto a las **variables compartidas**, y aunque **exclusión mutua** y **sincronización por condición** son conceptos distintos, se programan de forma similar.

Los **monitores** son módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.

Mecanismo de abstracción de datos:

* Encapsulan las representaciones de objetos (recursos).
* Brindan un conjunto de operaciones que son los únicos medios para manipular la representación.

Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él.

La***exclusión mutua*** está implícita asegurando que los *procedures* en el mismo monitor no ejecutan concurrentemente. Y la ***sincronización por condición*** se realiza de forma explícita con variables condición.

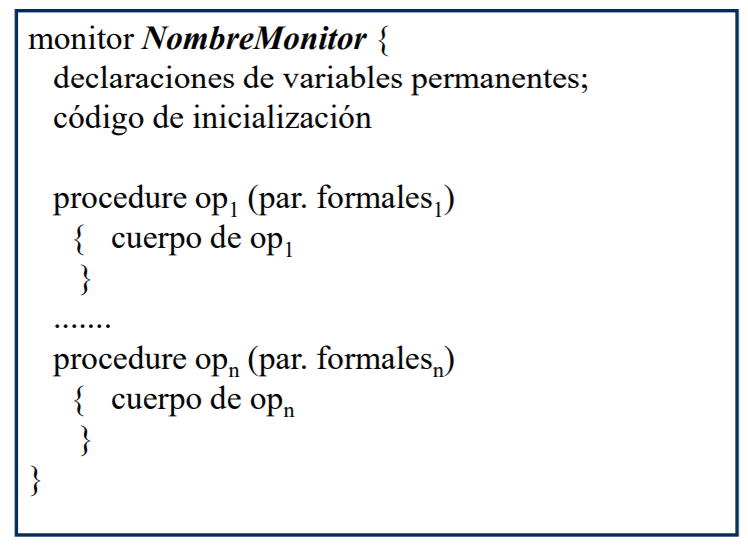
**Programa concurrente** → *procesos activos* y *monitores pasivos*. Dos procesos interactúan invocando *procedures* de un monitor.

Ventajas:

* Un proceso que invoca un ***procedure*** puede ignorar cómo está implementado.
* El programador del monitor puede ignorar cómo o dónde se usan los ***procedures***.

|  |
| --- |
| 12.1 Notación |

* Un monitor agrupa la representación y la implementación de un recurso compartido (clase). Lo que distingue a un monitor de un TAD es que puede ser compartido por procesos concurrentes.
* La **interfaz** del monitor específica operaciones (métodos) que brinda el recurso, y el **cuerpo** del monitor tiene variables que representan el estado del recurso y *procedures* que implementan las operaciones de la *interfaz*.
* Sólo los nombres de los ***procedures*** son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma: ***call NombreMonitor.operacion (argumentos).***
* Los ***procedures*** pueden acceder sólo variables permanentes, sus variables locales, y parámetros que le sean pasados en la invocación.
* El programador de un monitor no puede conocer a priori el orden de llamado.



|  |
| --- |
| 12.2 Sincronización |

La **sincronización por condición** es programada explícitamente con *variables condición* ***cond c***; el valor asociado a ***c*** es una cola de procesos demorados, ***no visible directamente al programador***.

Operaciones sobre las variables condición:

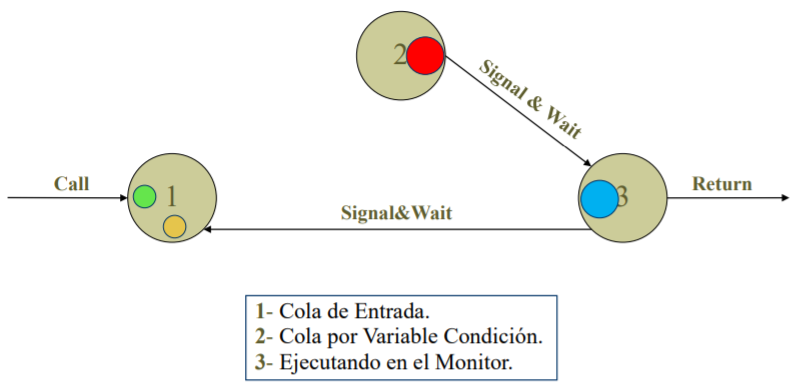
* ***wait(c)*** el proceso se demora al final de la cola de cv y deja el acceso exclusivo al monitor.
* ***signal(c)*** despierta al proceso que está al frente de la cola y lo saca de ella (cola vacía = skip). Ese proceso puede ejecutar cuando readquiera el acceso exclusivo al monitor (depende de la disciplina de señalización)
* ***signal\_all(c)*** despierta todos los procesos demorados en cv. El tiempo en que cada uno reinicie efectivamente la ejecución dependerá de las condiciones de exclusión mutua.
* ***empty(cv)*** → retorna true si la cola controlada por cv está vacía.
* ***wait(cv, rank)*** → el proceso se demora en la cola de cv en orden ascendente de acuerdo al parámetro rank y deja el acceso exclusivo al monitor.
* ***minrank(cv)*** → función que retorna el mínimo ranking de demora.

Disciplinas de sincronización:

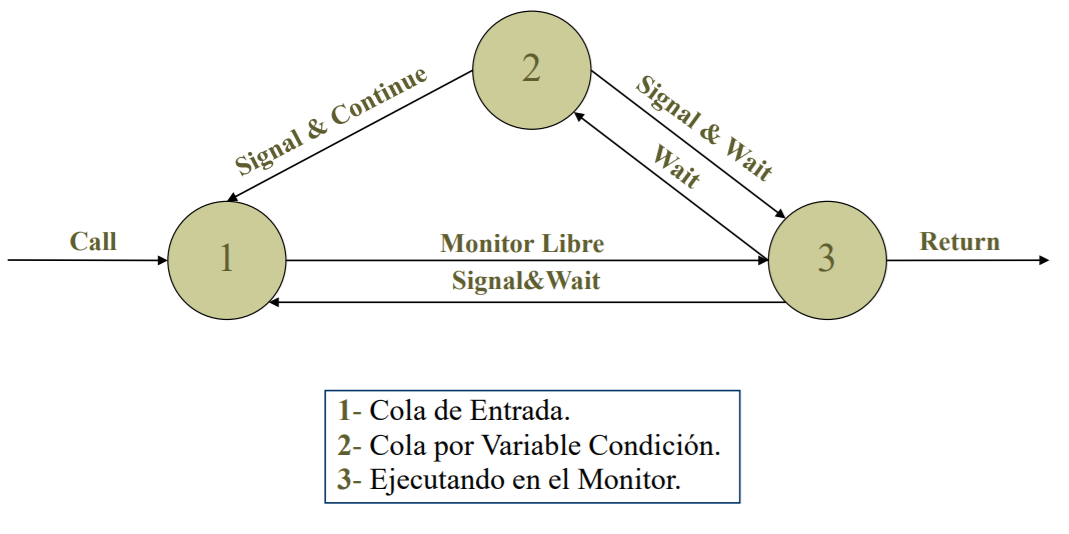
* **Signal and continued**: el proceso que hace el *signal* continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al *wait*). ***Este es el que usamos en la práctica.***
* *SIGNAL*:



* **Signal and wait**: el proceso que hace el *signal* pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al *wait.*
* *SIGNAL:*

**

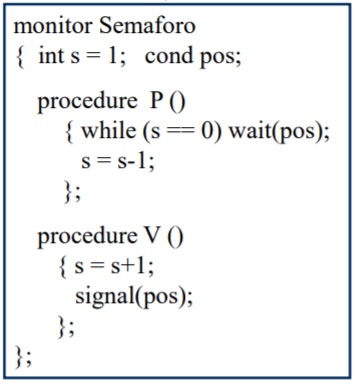
**Signal and continued VS signal and wait**



**Diferencia entre wait/signal con P/V**



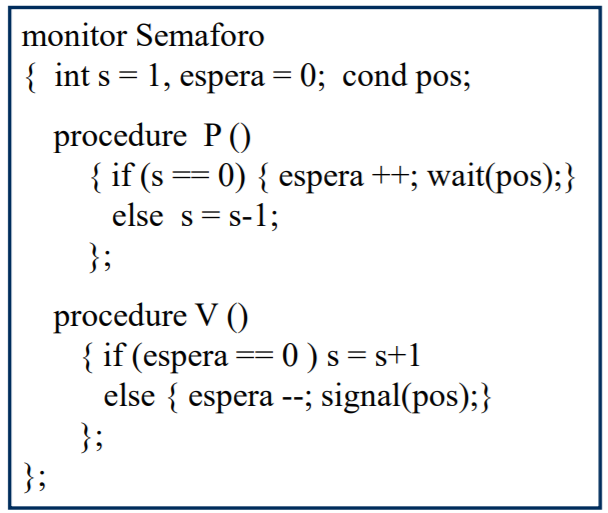
|  |
| --- |
| 12.3 Simulación de semáforos con monitores |



|  |
| --- |
| 12.4 Técnicas de sincronización |

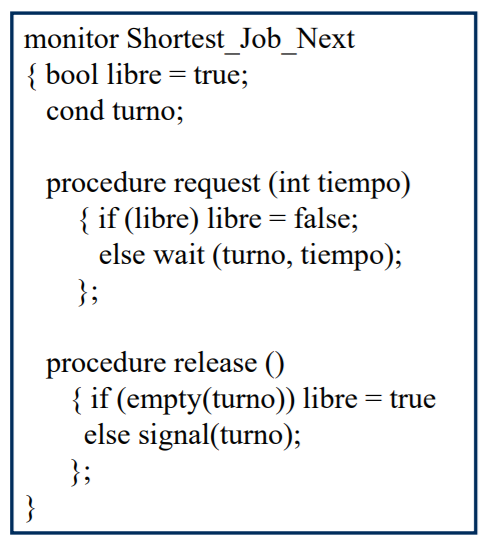
|  |
| --- |
| 12.4.1 Passing the condition |

Consiste en que un proceso señale una variable de condición si hay procesos demorados en ella, pero sin volver verdadera a la condición asociada; o bien vuelva verdadera la condición sin señalar la variable si no hay procesos demorados. Permite asegurar que al señalar una variable, el proceso despertado pueda continuar ejecutándose. Como el proceso que señala no vuelve verdadera a la condición, si otro proceso tomara control del monitor antes que el despertado, verá falsa la condición y pasará a la cola de espera, siendo el proceso despertado el único que podrá continuar.



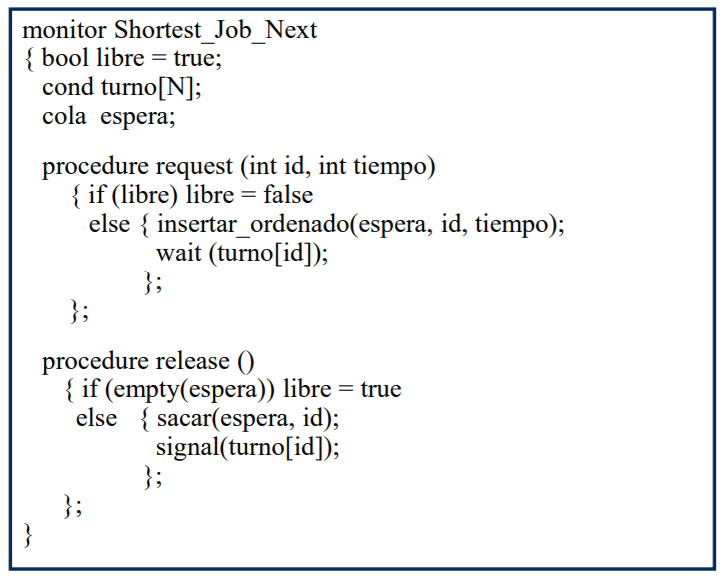
|  |
| --- |
| 12.4.2 Alocación SJN: Wait con Prioridad |

* Se usa ***wait*** con prioridad para ordenar los procesos demorados por la cantidad de tiempo que usarán el recurso.
* Se usa ***empty*** para determinar si hay procesos demorados.
* Cuando el recurso es liberado, si hay procesos demorados se despierta al que tiene mínimo ***rank***.
* ***Wait*** no se pone en un loop pues la decisión de cuándo puede continuar un proceso la hace el proceso que libera el recurso.



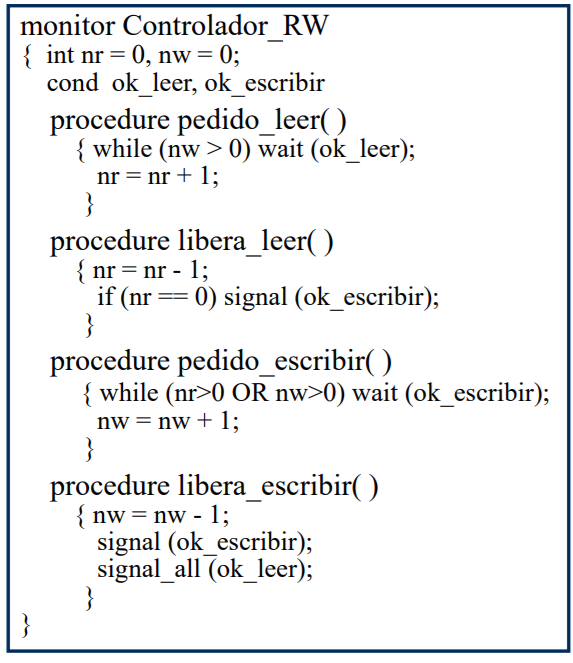
|  |
| --- |
| 12.4.3 Alocación SJN: Variables Condición Privadas |

Manejo del orden explícitamente por medio de una cola ordenada y variables condición privadas



|  |
| --- |
| 12.4.4 Lectores y escritores: Broadcast Signal |

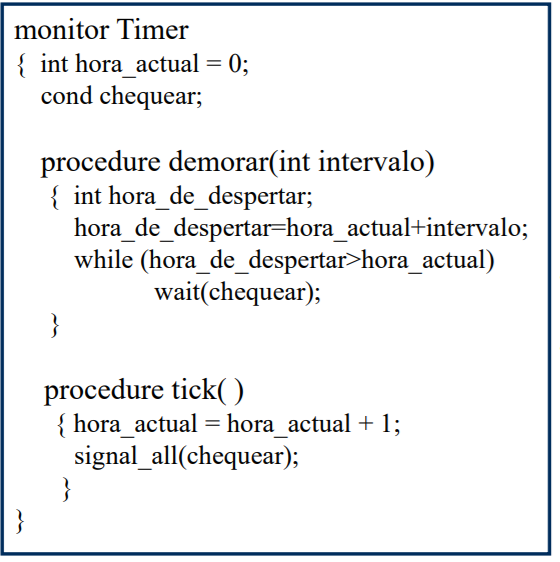
* El monitor arbitra el acceso a la BD.
* Los procesos dicen cuándo quieren acceder y cuándo terminaron ⇒ requieren un monitor con 4 *procedures*: *pedido\_leer, libera\_leer, pedido\_escribir, libera\_escribir*.

****

|  |
| --- |
| 12.4.5 Diseño de un reloj lógico |

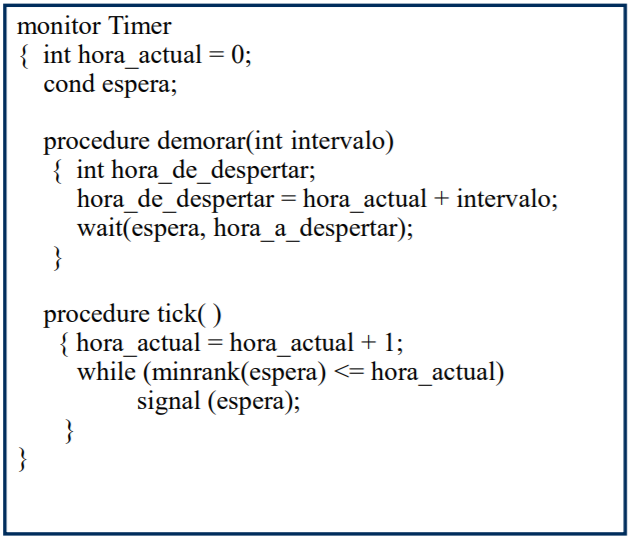
##### Usando covering conditions:

* ***Timer*** que permite a los procesos dormirse una cantidad de unidades de tiempo.
* Ejemplo de controlador de recurso (reloj lógico) con dos operaciones:
  + ***demorar(intervalo)***: demora al llamador durante intervalo ticks de reloj.
  + ***tick***: incrementa el valor del reloj lógico. Es llamada por un proceso que es despertado periódicamente por un timer de hardware y tiene alta prioridad de ejecución.



* **Es ineficiente** → mejor usar *wait con prioridad* o *variables condition privadas*

Usando *wait* con prioridad:



|  |
| --- |
| 12.5 Relación entre Passing the Baton y Passing the condition |

**Passing the baton** permite que un proceso pase a otro demorado el control (*mutex* e) y que éste sea el único que pueda ejecutar secciones exclusivas; **passing the condition** permite que un proceso avise a otro que están dadas las condiciones para que continúe y que éste sea el único que pueda continuar la ejecución de los que requieran dicha condición.

El proceso despertado por *passing the baton* se ejecutará antes que el resto de los que competían por el *mutex*; el proceso despertado por passing the condition se ejecutará antes que el resto de los que requieran la condición (tal vez tomen control del monitor, pero se bloquearán).

Aunque *passing the baton* puede utilizarse para cualquier tipo de sincronización, *passing the condition* sólo puede aplicarse cuando los procesos que ejecutan *wait* y *signal* realizan acciones complementarias entre sí (por ejemplo, uno decrementa un contador y el otro lo incrementa, o uno setea una variable a verdadero y el otro a falso).

*Passing the condition* es una técnica general para implementar sentencias *await* arbitrarias y para decidir cuál de los procesos es el próximo en seguir ejecutando (orden en el cual despertarlos) pasandole una condición.

|  |
| --- |
| **13. Pasaje de mensajes** |

|  |
| --- |
| 13.1 Conceptos generales |

Arquitecturas de memoria distribuida → *procesadores + memo local + red de comunicaciones +* ***mecanismo de comunicación / sincronización*** →***intercambio de mensajes***

**Programa distribuido** → programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).

**Primitivas de pasaje de mensajes**: interfaz con el sistema de comunicaciones → semáforos + datos + sincronización.

Los procesos SOLO **comparten canales** (físicos o lógicos). Variantes para los canales:

* Mailbox, input port, link.
* Uni o bidireccionales.
* Sincrónicos o asincrónicos.

|  |
| --- |
| 13.2 Características |

* Los canales son lo único que comparten los procesos
* Variables locales a un proceso (“cuidador”).
* La exclusión mutua no requiere mecanismo especial.
* Los procesos interactúan comunicándose.
* Accedidos por primitivas de envío y recepción.
* Mecanismos para el Procesamiento Distribuido:
* Pasaje de Mensajes Asincrónicos (PMA)
* Pasaje de Mensajes Sincrónico (PMS)
* Llamado a Procedimientos Remotos (RPC)
* Rendezvous
* La estructura de la comunicación entre los procesos depende del patrón de interacción:
* Productores y consumidores
* Clientes y servidores
* Peers
* **Cada mecanismo es más adecuado para determinados patrones**

|  |
| --- |
| 13.3 Relación entre mecanismos de sincronización |

* **Semáforos** → mejora respecto de ***busy waiting***.
* **Monitores** → combinan Exclusión Mutua implícita y señalización explícita.
* **PM** → extiende semáforos con datos.
* **RPC** y **rendezvous** → combina la interfaz procedural de monitores con PM implícito.

|  |
| --- |
| 13.4 Pasaje de mensajes asincrónicos (PMA) |

* PMA → **canales** = colas de mensajes enviados y aún no recibidos.
* **Operación send**: Un proceso agrega un mensaje al final de la cola (“ilimitada”) de un canal ejecutando un ***send***, que no bloquea al emisor.
* **Operación receive**: Un proceso recibe un mensaje desde un canal con ***receive***, que demora al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales.

*Receive* es una primitiva bloqueante, ya que produce un espera. Semántica: el proceso **NO** hace nada hasta recibir un mensaje en la cola correspondiente al canal. NO es necesario hacer polling

|  |
| --- |
| 13.4.1 Características de los canales |

* **Acceso a los contenidos de cada canal:** atómico y respeta orden FIFO.
* En principio los canales son ilimitados, aunque las implementaciones reales tendrán un tamaño de buffer asignado.
* Se supone que los mensajes NO se pierden ni modifican y que todo mensaje enviado en algún momento puede ser “leído”.
* **empty(ch)** determina si la cola de un canal está vacía. Es útil cuando el proceso puede hacer trabajo productivo mientras espera un mensaje, pero *debe usarse con cuidado*.
* La evaluación de ***empty*** podría ser ***true***, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.
* O podría ser ***false***, y no haber más mensajes cuando sigue ejecutando (si no es el único en recibir por ese canal).

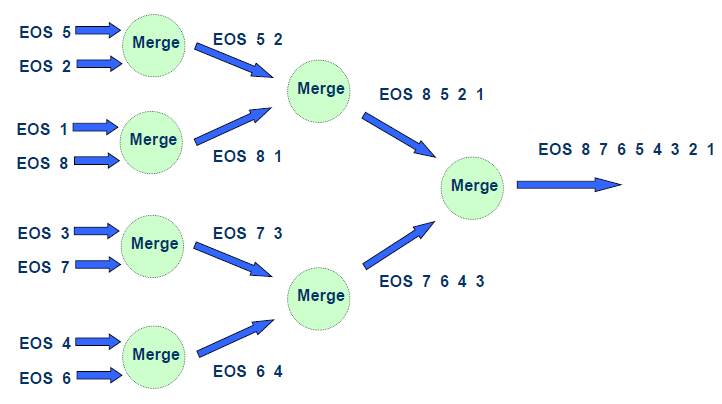
Los canales son declarados globales a los procesos, ya que pueden ser compartidos. Según la forma en que se usan podría ser:

* Cualquier proceso puede enviar o recibir por alguno de los canales declarados. En este caso suelen denominarse ***mailboxes***.
* En algunos casos un canal tiene un solo receptor y muchos emisores (***input port***).
* Si el canal tiene un único emisor y un único receptor se lo denomina ***link***: provee un “camino” entre el emisor y sus receptores.

|  |
| --- |
| 13.4.2 Productores y consumidores (filtro). Red de ordenación |

**Filtro:** Proceso que recibe mensajes de uno o más canales de entrada y envía mensajes a uno o más canales de salida. La salida de un filtro es función de su estado inicial y de los valores recibidos. Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de salida con los de entrada.

Problema: ordenar una lista de N números de modo ascendente. Podemos pensar en un filtro Sort con un canal de entrada (*N números desordenados*) y un canal de salida (*N números ordenados*). Una solución más eficiente que la “secuencial” es construir una red de pequeños procesos que ejecutan en paralelo e interactúan para “armar” la salida ordenada. ***(merge network)****.*



Para realizar esta red de ordenación, los canales de entrada y salida necesitan ser compartidos. En particular, el canal de salida usado por una instancia de Merge necesita ser el mismo que uno de los canales de entrada usada por la siguiente instancia de Merge en el grafo. Esto puede ser programado de dos maneras.

* La primera es usar ***static naming***: declarar todos los canales como un arreglo global, y que cada instancia de Merge reciba de dos elementos del arreglo y envíe a otro elemento. Esto requiere embeber el árbol en el arreglo para que los canales accedidos por Merge sean una función de i.
* La segunda forma es usar***dynamic naming***: declarar todos los canales globales, parametrizar los procesos y darle a cada proceso tres canales cuando es creado. Esto hace más fácil la programación de los procesos Merge pues cada uno es textualmente idéntico. Sin embargo, requiere tener un proceso principal que dinámicamente cree y pase canales a los procesos Merge.

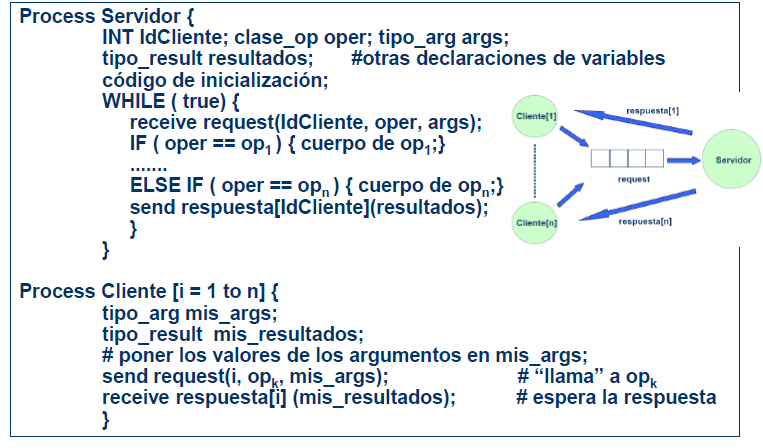
|  |
| --- |
| 13.4.3 Clientes y servidores. Monitores activos |

**Servidor**: proceso que maneja pedidos (“*requests*”) de otros procesos ***clientes***. Analizamos cómo convertir monitores en servidores y cómo implementar manejadores de recursos compartidos. Existe una dualidad entre monitores y PM: cada uno de ellos puede simular al otro.

**Monitor →** *manejador de recurso*. Encapsula variables permanentes que registran el estado, y provee un conjunto de *procedures*. Los simulamos, usando procesos servidores y PM, como procesos activos en lugar de como conjuntos pasivos de *procedures*.

**Monitor con múltiples operaciones**

Podemos generalizar esta solución de C/S con una única operación para considerar múltiples operaciones. El ***IF*** del Servidor será un *CASE* con las distintas clases de operaciones. El cuerpo de cada operación toma datos de un canal de entrada en *args* y los devuelve al cliente adecuado en resultados.

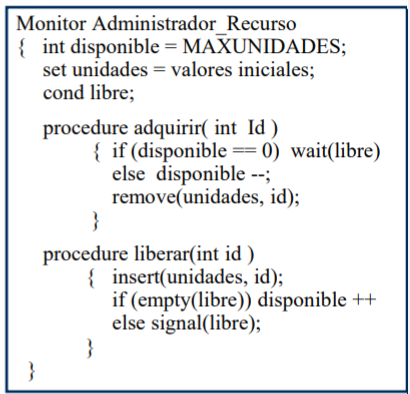
****

**Monitor con múltiples operaciones con “wait”**

Hasta ahora el monitor no requería variables condición ya que el Servidor no requería demorar la atención de un pedido de servicio. **Caso general**: monitor con múltiples operaciones y con sincronización por condición. Para los clientes, la situación es transparente ⇒ ***cambia el servidor***.

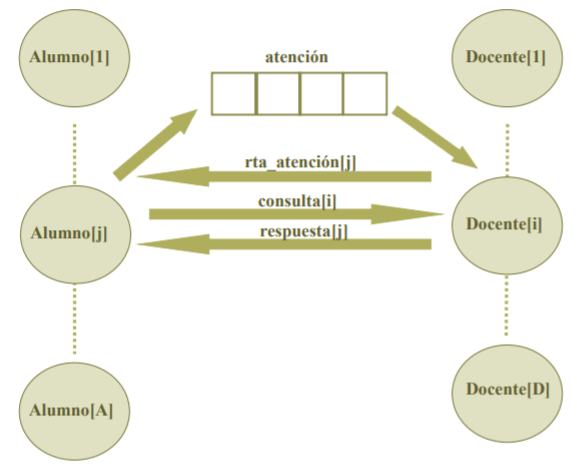
Consideramos un caso específico de manejo de múltiples unidades de un recurso (ejemplos: bloques de memoria, impresoras).

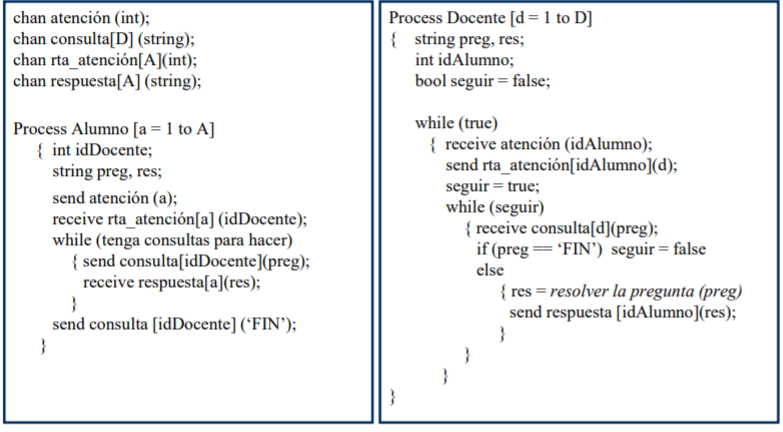
* Los clientes “adquieren” y devuelven unidades del recurso.
* Las unidades libres se insertan en un “conjunto” sobre el que se harán las operaciones de INSERTAR y REMOVER.
* El número de unidades disponibles es lo que “controla” nuestra variable de sincronización por condición



**Continuidad Conversacional**

Existen A alumnos que hacen consultas a D docentes. El alumno espera hasta que un docente lo atiende, y a partir de ahí le comienza a realizar las diferentes consultas hasta que no le queden dudas. Los alumnos son los procesos “*clientes*”, y los docentes los procesos “*Servidores*”. Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de un alumno. Todos los alumnos pueden pedir atención por un ***canal global*** y recibirán respuesta de un docente dado por un ***canal propio***.

****



La interacción entre un cliente y un server es un ejemplo de ***continuidad conversacional***.

***atención*** es un canal compartido por el que cualquier Docente puede recibir. Si cada canal puede tener un solo receptor, se necesita otro proceso intermedio

|  |
| --- |
| 13.4.4 Pares (peers) interactuantes: Intercambio de valores |

**Problema**: cada proceso tiene un dato local V y los N procesos deben saber cuál es el menor y cuál el mayor de los valores.

Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: ***centralizado, simétrico*** y en ***anillo circular***.

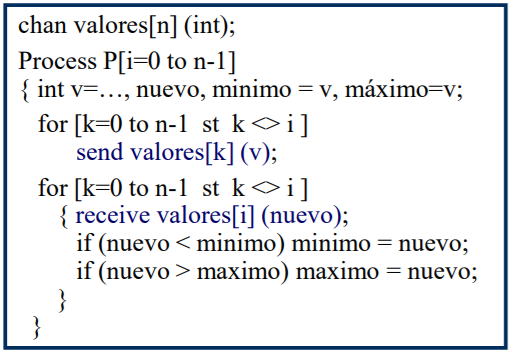
**Solución centralizada**

La **arquitectura centralizada** es apta para una solución en que todos envían su dato local ***V*** al procesador central, éste ordena los ***N*** datos y reenvía la información del mayor y menor a todos los procesos → ***2(N-1)*** mensajes. Si ***p[0]*** dispone de una primitiva ***broadcast*** se reduce a ***N*** mensajes.

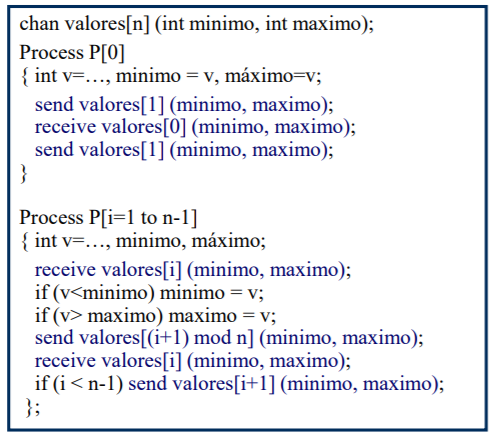
****

**Solución simétrica**

En la **arquitectura simétrica o “full connected”** hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo. Cada proceso trasmite su dato local ***V*** a los ***n-1*** restantes procesos. Luego recibe y procesa los ***n-1*** datos que le faltan, de modo que en paralelo toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los ***n*** datos. Ejemplo de solución SPMD: cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos ***n(n-1)*** mensajes → Si disponemos de una primitiva de ***broadcast***, serán nuevamente ***n*** mensajes.



**Solución en anillo circular**  
Un tercer modo de organizar la solución es tener un **anillo** donde ***P[i]*** recibe mensajes de ***P[i-1]*** y envía mensajes a ***P[i+1]***. ***P[n-1]*** tiene como sucesor a ***P[0]***. Esquema de 2 etapas. En la primera cada proceso recibe dos valores y los compara con su valor local, transmitiendo un máximo local y un mínimo local a su sucesor. En la segunda etapa todos deben recibir la circulación del máximo y el mínimo global. ***P[0]*** deberá ser algo diferente para “arrancar” el procesamiento. Se requerirán ***2 (n-1)*** mensajes. Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes.

**

*Conclusión*:

* ***Simétrica*** es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast). Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.
* ***Centralizada y anillo*** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance.
  + ***En centralizada***, los msgs al coordinador se envían casi al mismo tiempo → sólo el 1er *receive* del coordinador demora mucho.
  + ***En anillo***, todos los procesos son *productores y consumidores.* El último tiene que esperar a que todos los otros (uno por vez) reciban un msg, hacer poco cómputo, y enviar su resultado.
  + Los mensajes circulan 2 veces completas por el anillo → Solución inherentemente lineal y lenta para este problema, pero puede funcionar si cada proceso tiene mucho cómputo.

|  |
| --- |
| 13.4.5 Paradigmas para la interacción entre procesos |

* ***Servidores Replicados***: Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia. La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.

Ejemplo: problema de los filósofos.

1. **Modelo centralizado**: los procesos Filósofo se comunican con **UN** proceso *Mozo* que decide el acceso o no a los recursos.
2. **Modelo distribuido**: supone ***5 procesos Mozo*** cada uno manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos **NO se comunican entre ellos.**
3. **Modelo descentralizado**: cada Filósofo ve **un único Mozo.** Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a “su” Filósofo.

La **solución descentralizada** tiene un mozo por filósofo. El algoritmo usado por los mozos es token passing, donde tokens = tenedores. La solución puede adaptarse para coordinar acceso a archivos replicados, o dar solución eficiente al problema de EM distribuida.Cada tenedor es un token que tiene uno de los dos mozos o está en tránsito entre ellos.Cuando el filósofo quiere comer, le pide a su mozo que adquiera dos tenedores. Si el mozo no los tiene, interactúa con sus mozos vecinos para obtenerlos. Luego mantiene el control mientras el filósofo come. Debe evitarse el deadlock, que podría darse si un mozo necesita dos tenedores y no los puede lograr. La idea básica para evitar deadlock es que un mozo mantenga un tenedor si es necesario y aún no fue usado; en otro caso el mozo lo cede. Específicamente, cuando un filósofo comienza a comer, su mozo marca ambos tenedores como "sucios". Cuando otro mozo quiere un tenedor, si está sucio y no está siendo usado, el primer mozo lo limpia y lo cede. El primer mozo no puede tomar nuevamente el tenedor hasta que no haya sido usado pues solo son pasados tenedores sucios entre los mozos. Sin embargo, un tenedor sucio puede ser reusado hasta que lo necesite otro mozo.

* ***Algoritmos Heartbeat***: útil para soluciones iterativas que se quieren paralelizar. Usando un esquema “*divide & conquer*” se distribuye la carga (datos) entre los *workers*; cada uno es responsable de actualizar una parte. Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos. Cada “paso” debiera significar un progreso hacia la solución. Su uso más importante es paralelizar soluciones iterativas. **Ejemplos** de problemas

que se pueden resolver son computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).

* ***Algoritmos Pipeline***: Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida. Estos procesos (“*workers*”) pueden estar en procesadores que operan en paralelo, en un primer esquema a ***lazo abierto***. Un segundo esquema es el pipeline circular, donde *Wn* se conecta con *W1*. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe. En un tercer esquema posible (cerrado), existe un proceso coordinador que maneja la “realimentación” entre *Wn* y *W1*. **Ejemplos** son las redes de filtros o tratamiento de imágenes.
* ***Probes & Echos -sondas y ecos-*:** Arboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos. Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan. DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo. Este paradigma es el análogo concurrente de DFS. *Prueba-eco* se basa en el envío de un mensajes (“*probe*”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“*echo*”). Los *probes* se envían en paralelo a todos los sucesores. Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles)
* ***Algoritmos Broadcast***: En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva broadcast: ***broadcast ch(m);***. Los mensajes *broadcast* de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden. Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. **Ejemplo**: semáforos distribuidos, la base es un ***ordenamiento total de eventos de comunicación*** mediante el uso de *relojes lógicos*.
* ***Token Passing***: Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“*token*”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un **ejemplo** del primer tipo de algoritmos es el caso de tener que controlar ***exclusión mutua distribuida***.

Un token es una clase especial de mensaje que puede ser usado o para dar permiso para tomar una acción o para reunir información de estado global. El envío de información global y/o la toma de determinadas decisiones se basan en el uso de tokens enviados entre los procesos. La topología no necesariamente debe ser de anillo ni los tokens globales.

* ***Manager/workers***: El concepto de ***bag of tasks*** usando variables compartidas supone que un conjunto de *workers* comparten una “bolsa” con tareas independientes. Los *workers* sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa. La mayor virtud de este enfoque es la **escalabilidad** y la **facilidad** para equilibrar la carga de trabajo de los *workers*.

En **problemas de broadcast** sin conocimientos sobre la topología, como solución general sería conveniente Probe & Echo, ya que permite el envío de mensajes a todos los nodos sin importar la arquitectura de red -podría obviarse el hecho si no se requieren respuestas, dependiendo del problema. *Los algoritmos de broadcast* podrían utilizarse pero sólo si la primitiva broadcast es soportada para la red en cuestión, siendo preferible en esos casos. *Manager/workers y servidores replicados* no aplican, están orientados a problemas diferentes. *Token passing* serviría para topologías de anillo pero no sería la mejor opción para grafos generales. **Manager/Workers, Heartbeat y Pipeline son más comunes en programas paralelos**. Con pipeline los datos fluyen en una dirección y podría no ser posible hacer broadcast. Con heartbeat podrían compartirse datos con los vecinos y luego de varias iteraciones sería de conocimiento global, pero sin datos de la topología no se sabría cuántas iteraciones son necesarias, y obviamente la convergencia sería más lenta. Además el tipo de problemas en los que se aplican estos algoritmos raramente precisarán broadcasts.

**Algoritmo Heartbeat para el cálculo de la topología de una red**

Procesadores conectados por canales bidireccionales. C/ uno se comunica sólo con sus vecinos y conoce esos links Cómo puede cada procesador determinar la topología completa de la red?

Soluciones posibles:

* los procesos tienen acceso a MC
* distribuida: los vecinos interactúan para intercambiar info local

El Algoritmo Heartbeat se expande, enviando información; luego se contrae, incorporando nueva información.

|  |
| --- |
| 13.5 Pasaje de mensajes sincrónicos (PMS) |

La principal diferencia con PMA es que la primitiva de transmisión (llamemosla *sync\_send*) es bloqueante, el trasmisor queda esperando que el mensaje sea recibido. La cola de mensajes asociada con un *send* sobre un canal se reduce a 1 mensaje → MENOS memoria. Los canales son punto a punto (1 emisor – 1 receptor). ***Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (los emisores se bloquean).*** Si bien *send* y *sync\_sen*d son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de ***deadlock*** mayores en comunicación sincrónica.

|  |
| --- |
| 13.5.1 Comentarios |

Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras ***n*** operaciones, y luego se realizan mucho más lento durante otras ***n*** interacciones:

* Con PMS los pares *send/receive* se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10 a 1 significaría multiplicar por 10 los tiempos totales.
* Con PMA, al principio el productor es más rápido y sus mensajes se encolan. Luego el consumidor es más rápido y “descuenta” tiempo consumiendo la cola de mensajes.
* ***Mayor concurrencia en PMA*** → Para lograr el mismo efecto en PMS se debe interponer un proceso “***buffer***”.
* La concurrencia también se reduce en algunas interacciones Cliente/Servidor.
  + Cuando un cliente está liberando un recurso, no habría motivos para demorarlo hasta que el servidor reciba el mensaje, pero con PMS se tiene que demorar.
  + Otro ejemplo se da cuando un cliente quiere escribir en un display gráfico, un archivo u otro dispositivo manejado por un proceso servidor. Normalmente el cliente quiere seguir inmediatamente después de un pedido de write.
* Otra desventaja del PMS es la ***mayor probabilidad de deadlock***. El programador debe ser cuidadoso de que todas las sentencias *send* (*sync\_send*) y *receive* hagan *matching*.

|  |
| --- |
| 13.5.2 CSP - Lenguaje para PMS |

(*Communicating Sequential Processes, Hoare 1978*) Fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, MPD) se basan en CSP. Las ideas básicas introducidas por Hoare fueron PMS y ***comunicación guardada***: PM con waiting selectivo. Un canal es un *link* directo entre dos procesos en lugar de *mailbox global*. Son *half-duplex* y nominados. Las sentencias de **Entrada** (? o *query*) y **Salida** (! o *shriek o bang*) son el único medio por el cual los procesos se comunican. Para que se produzca la comunicación, deben ***matchear***, y luego se ***ejecutan simultáneamente***. Efecto: sentencia de asignación distribuida.

**Formas generales de las sentencias de comunicación**:

*Destino ! port(e1, ..., en);*

*Fuente ? port(x1, ..., xn);*

Destino y Fuente nombran un proceso simple, o un elemento de un arreglo de procesos. Fuente puede nombrar cualquier elemento de un arreglo (Fuente[\*]). ***port*** es un canal de comunicación simple en el proceso destino o un elemento de un arreglo de ports en el proceso destino. Los ports se usan para distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno). Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen ***matching*** → *A ! canaluno(dato); B ? canaluno(resultado);*

**Comunicación guardada**

Limitaciones de ? y ! ya que son bloqueantes. Hay problema si un proceso quiere comunicarse con otros (quizás por distintos *ports*) sin conocer el orden en que los otros quieren hacerlo con él.

Las sentencias de comunicación guardada soportan comunicación no determinística:

**B; C → S;**

**B** puede omitirse y se asume true. **B** y **C** forman la guarda. La guarda tiene éxito si **B** es true y ejecutar **C** no causa demora. La guarda falla si **B** es falsa. La guarda se bloquea si **B** es true pero **C** no puede ejecutarse inmediatamente.

Las sentencias de comunicación guardadas aparecen en **if** y **do**.

*if B1; comunicación1 → S1; *

*B2; comunicación2 → S2;*

*fi*

**Primero** se evalúan las guardas:

* Si todas las guardas fallan, el if termina sin efecto.
* Si al menos una guarda tiene éxito, una de ellas es elegida no determinísticamente.
* Si algunas guardas están bloqueadas, la ejecución se demora hasta que alguna guarda tenga éxito.

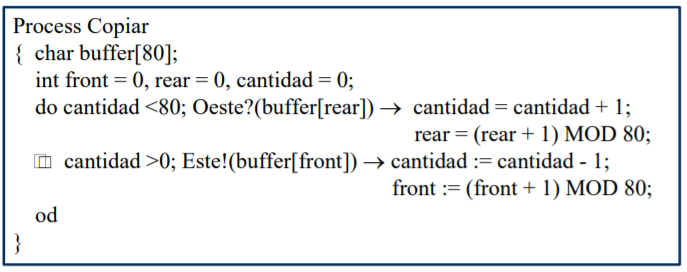
Dado que las variables no son compartidas, el valor de una expresión booleana en una guarda no puede cambiar hasta que el proceso ejecute sentencias de asignación. Así, una guarda bloqueada no puede tener éxito hasta que algún otro proceso alcanza una sentencia de comunicación matching.

**Segundo**, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.

**Tercero**, se ejecuta la sentencia *Si* .

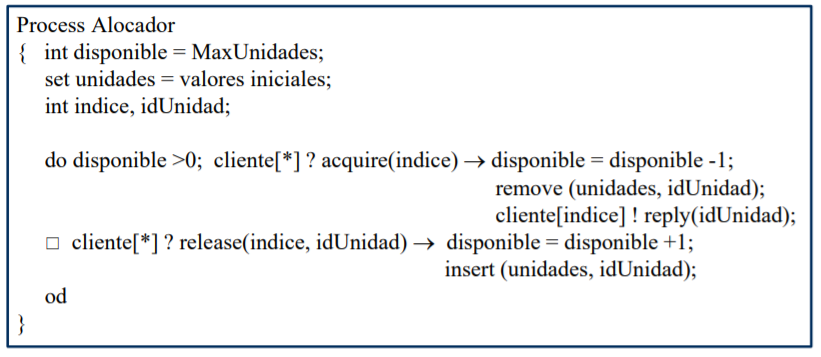
La ejecución del **do** es similar (se repite hasta que todas las guardas fallen).

**Ejemplo: Buffer limitado**



* Con **PMA**, procesos como *Oeste* y *Este* ejecutan a su propia velocidad pues hay *buffering* implícito.
* Con **PMS**, es necesario programar un proceso adicional para implementar *buffering* si es necesario.

**Ejemplo: Asignación de recursos**



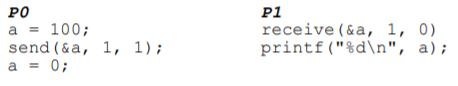
La solución es concisa. Usa múltiples ***ports*** y un brazo del ***do*** para atender cada uno. Se demora en un mensaje ***acquire*** hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

|  |
| --- |
| 13.5.3 MPI - Bibliotecas para manejo de PM |

**Conceptos generales**

*MPI* define una librería estándar para pasaje de mensajes que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes). El estándar *MPI* define la sintaxis y la semántica de más de 125 rutinas. Hay implementaciones de *MPI* de la mayoría de los proveedores de hardware. Modelo *SPMD*. Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: *MPI\_Init, MPI\_Finalize, MPI\_Comm\_size, MPI\_Comm\_rank, MPI\_Send y MPI\_Recv*.

**Comunicación punto a punto**

****

La semántica del *SEND* requiere que en P1 quede el valor 100 (no 0). Para asegurar la semántica del ***SEND*** → no devolver el control del *Send* hasta que el dato a transmitir esté seguro (Send bloqueante).

Diferentes protocolos para ***Send***.

* Send bloqueantes con buffering (*Bsend*).
* Send bloqueantes sin buffering (*Ssend*).
* Send no bloqueantes (*Isend*).

Diferentes protocolos para ***Recv***.

* *Recv* bloqueantes (*Recv*).
* *Recv* no bloqueantes (*Irecv*).

***MPI\_Send***: rutina básica para enviar datos a otro proceso.

*MPI\_Send (void \*buf, int cantidad, MPI\_Datatype tipoDato, int destino, int tag, MPI\_Comm comunicador)*

***MPI\_Recv***: rutina básica para recibir datos a otro proceso.

*MPI\_Recv (void \*buf, int cantidad, MPI\_Datatype tipoDato, int origen, int tag, MPI\_Comm comunicador, MPI\_Status \*estado)*

**Send y recv no bloqueantes (*lsend, lrecv*)**

* Comienzan la operación de comunicación e inmediatamente devuelven el control (no se asegura que la comunicación finalice correctamente).
* *MPI\_Test*: testea si la operación de comunicación finalizó.
* *MPI\_Wait*: bloquea al proceso hasta que finaliza la operación.

**Indagación por arribo de mensajes**

* Información de un mensaje antes de hacer el *Recv* (Origen, Cantidad de elementos, Tag).
* *MPI\_Probe*: bloquea el proceso hasta que llegue un mensaje que cumpla con el origen y el tag.
* *MPI\_Iprobe*: chequea por el arribo de un mensaje que cumpla con el origen y tag.

|  |
| --- |
| **14. Conceptos de RPC y Rendezvous** |

El Pasaje de Mensajes se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la ***comunicación unidireccional***. Para resolver Cliente/Servidor la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas). Además, cada cliente necesita un canal de *reply* distinto lo que lleva a un gran número de canales.

RPC (*Remote Procedure Call*) y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen un ***canal bidireccional*** que son ideales para programar aplicaciones Cliente/Servidor, donde la comunicación entre ellos debe ser sincrónica, el cliente solicita un servicio y el servidor le responde con lo solicitado ya que el cliente no debe realizar ninguna otra tarea hasta no obtener una respuesta del servidor.

RPC y Rendezvous combinan una interfaz “tipo monitor” con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).

**Diferencias entre RPC y Rendezvous**

Difieren en la manera en la cual se sirven las invocaciones de operaciones.

* Un enfoque es declarar un *procedure* para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (*RPC* porque el llamador y el cuerpo del *procedure* pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve (Ej: *JAVA*).
* El segundo enfoque es hacer *rendezvous* con un proceso existente. Un *rendezvous* es servido por una *sentencia de Entrada* (o ***accept***) que espera una invocación, la procesa y devuelve los resultados (Ej:*Ada*)

|  |
| --- |
| 14.1 RPC |

* Los programas se descomponen en **módulos** (con procesos y *procedures*), que pueden residir en espacios de direcciones distintos.
* Los procesos de un módulo pueden compartir variables y llamar a *procedures* de ese módulo.
* Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.
* Los módulos tienen especificación e implementación de *procedures.*

***module*** *Mname*

*headers de procedures exportados (visibles)*

***body***

*declaraciones de variables; código de inicialización*

*cuerpos de procedures exportados; procedures y procesos locales*

***end***

* Los procesos locales son llamados ***background*** para distinguirlos de las operaciones exportadas.
* Header de un *procedure* visible:

***op opname*** *(formales) [****returns*** *result]*

* El cuerpo de un *procedure* visible es contenido en una declaración *proc*:

***proc opname****(identif. formales)* ***returns*** *identificador resultado*

*declaración de variables locales; sentencias*

***end***

* Un proceso (o *procedure*) en un módulo llama a un *procedure* en otro ejecutando:

***call Mname.opname (argumentos)***

* Para un llamado local, el nombre del módulo se puede omitir.

La implementación de un llamado intermodulo es distinta que para un llamado local, pues dos módulos pueden estar en espacios de direcciones diferentes. En particular, un **nuevo proceso** sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server. El proceso llamador se demora mientras este proceso server ejecuta el cuerpo del *procedure* que implementa *opname*. Cuando el server vuelve de *opname*, envía los argumentos resultado y cualquier valor de retorno al proceso llamador, luego el *server* termina. Después de recibir resultados, el proceso llamador continúa. Si el proceso llamador y el *procedure* están en el mismo espacio de direcciones, con frecuencia es posible evitar crear un nuevo proceso para servir un llamado remoto (el proceso llamador puede temporariamente convertirse en server y ejecutar el cuerpo del *procedure*. Pero en general, un llamado será remoto, de modo que un proceso *server* debe ser creado o alocado de un pool preexistente de *servers* disponibles.



***Por sí mismo, RPC es solo un mecanismo de comunicación***. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado → la sincronización entre ambos es implícita).

Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende *Exclusión Mutua* y *Sincronización por Condición*.

Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan con ***exclusión mutua*** (un solo proceso por vez) o ***concurrentemente.***

* Si ejecutan con ***exclusión mutua*** las variables compartidas son protegidas automáticamente contra acceso concurrente, pero es necesario programar sincronización por condición.
* Si pueden ejecutar ***concurrentemente*** necesitamos mecanismos para programar exclusión mutua y sincronización por condición ***(cada módulo es un programa concurrente)* →** Podemos usar cualquier método ya descrito (semáforos, monitores, o incluso rendezvous).
* Es más general asumir que los procesos pueden ejecutar concurrentemente (más eficiente en un multiprocesador de MC).

|  |
| --- |
| 14.2 RPC en JAVA. Remote Method Invocation (RMI) |

Java soporta el uso de RPC en programas distribuidos mediante la invocación de métodos remotos (RMI). Una aplicación que usa RMI tiene 3 componentes:

* Una interfaz que declara los headers para métodos remotos.
* Una clase *server* que implementa la interfaz.
* Uno o más clientes que llaman a los métodos remotos.

El server y los clientes pueden residir en máquinas diferentes.

|  |
| --- |
| 14.3 Rendezvous |

Rendezvous combina **comunicación** y **sincronización**. Como con RPC, un proceso cliente ***invoca*** una operación por medio de un ***call***, pero esta operación es servida por un proceso existente en lugar de por uno nuevo. Un proceso servidor usa una ***sentencia de entrada*** para esperar por un ***call*** y actuar. Las operaciones se atienden una por vez más que concurrentemente.

La especificación de un módulo contiene declaraciones de los ***headers*** de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones. Si un módulo exporta ***opname***, el proceso server en el módulo realiza ***rendezvous*** ejecutando una ***sentencia de entrada***:

***in opname****(identif. formales)* ***→ S; ni***

Las partes entre ***in*** y ***ni*** se llaman ***operación guardada***. Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de ***opname***; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta ***S*** y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

|  |
| --- |
| 14.3.1 ADA |

Desde el punto de vista de la concurrencia, un programa *Ada* tiene “***tasks***” (tareas o procesos) que pueden ejecutar independientemente y que contienen primitivas de sincronización. Los puntos de invocación (entrada) a una task se denominan ***entrys*** y están especificados en la parte visible (header de la tarea). Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva ***accept***. Se puede declarar un ***type task***, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple).

Una especificación de *TASK* define una única tarea. Una instancia del correspondiente *task body* se crea en el bloque en el cual se declara el *TASK.*

Ada no provee la posibilidad de asociar sentencias de scheduling y de poder usar los parámetros formales de la operación tanto en las sentencias de sincronización como en las sentencias de scheduling.

|  |
| --- |
| 14.3.1.1 Sincronización |

* **Call: Entry Call**

El rendezvous es el principal mecanismo de sincronización en *Ada* y también es el mecanismo de comunicación primario.

Entry:

* Declaración de **entry simples y familia de entry** (parámetros IN, OUT y IN OUT).
* **Entry call**. La ejecución demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).

* **Entry call condicional:**

*select entry call;*

*sentencias adicionales;*

*else*

*sentencias;*

*end select;*

* **Entry call temporal:**

*select entry call;*

*sentencias adicionales;*

*or delay tiempo*

*sentencias;*

*end select;*

* **Sentencia de Entrada: Accept**

La tarea que declara un entry sirve llamados al entry con accept:

***accept*** *nombre (parámetros formales) do sentencias end nombre;*

Demora la tarea hasta que haya una invocación, copia los parámetros reales en los parámetros formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan.

La sentencia ***wait selectiva*** soporta comunicación guardada.

***select when B1*** *⇒* ***accept*** *E1; sentencias1*

***or*** *…*

***or when Bn*** *⇒* ***accept*** *En; sentenciasn*

***end select****;*

* Cada línea se llama alternativa. Las cláusulas *when* son opcionales.
* Puede contener una alternativa *else, or delay, or terminate*.
* Uso de atributos del entry: *count, calleable*.

|  |
| --- |
| 14.3.1.2 Primitivas múltiples |

RPC y rendezvous → un proceso inicia la comunicación con un ***call***, que bloquea al llamador hasta que la operación es servida y se retornan los resultados. Son ideales para interacciones C/S, pero difícil programar algoritmos filtros o *peers* que intercambian información (para éstos es mejor PMA)

**Notación de Primitivas Múltiples**

Combina ***RPC****,* ***Rendezvous*** *y* ***PMA*** en un paquete coherente.

* Brinda gran poder expresivo combinando ventajas de las 3 componentes, y poder adicional
* Programas → colecciones de módulos. Una operación visible se declara en la especificación del módulo. Puede ser invocada por procesos de otros módulos, y es servida por un proceso o *procedure* del módulo que la declara.
* También se usan operaciones ***locales***, que son declaradas, invocadas y servidas dentro del cuerpo de un único módulo.

Una operación puede ser invocada por ***call sincrónico*** o por ***send asincrónico***:

***call Mname.op(argumentos)***

***send Mname.op(argumentos)***

* El *call* termina cuando la operación fue servida y los resultados fueron retornados.
* El *send* termina tan pronto como los argumentos fueron evaluados.

Una operación es servida por un *procedure* (**proc**) o por *rendezvous* (sentencias **in**). La elección la toma el programador del módulo.

|  |
| --- |
| **15. Programación Paralela** |

* **Programa Concurrente**: múltiples procesos.
* **Programa distribuido**: programa concurrente en el cual los procesos se comunican y sincronizan por PM, RPC o Rendezvous.
* **Programa Paralelo**: Programa concurrente escrito para resolver un problema en menos tiempo que el secuencial. El objetivo principal es reducir el tiempo de ejecución, o resolver problemas más grandes o con más precisión en el mismo tiempo.

|  |
| --- |
| 15.1 Computación científica |

* Los dos modos tradicionales del descubrimiento científico son ***teoría*** y ***experimentación***.
* El 3er modo es la ***modelización computaciona***l, que usa computadoras para simular fenómenos y tratar cuestiones del tipo “***what if?***”
* Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:
* **Computación de grillas** (por ejemplo imágenes). Dividen una región espacial en un conjunto de puntos.
* **Computación de partículas** (modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares).
* **Computación de matrices** (sistemas de ecuaciones simultáneas).

|  |
| --- |
| 15.2 Diseño de algoritmos paralelos |

No se reduce a simples recetas, sino que es necesaria la creatividad. La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes. Pero puede darse un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de backtracking por malas elecciones ⇒ metodología de diseño que da un enfoque exploratorio en el cual aspectos independientes de la máquina tales como la concurrencia son considerados temprano, y los aspectos específicos de la máquina se demoran.

2 etapas:

* Descomposición.
* Mapeo

|  |
| --- |
| 15.2.1 Métricas del paralelismo |

En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance. Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución. A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (*sistema paralelo = algoritmo + arquitectura sobre la que se implementa*).

En la medición de performance es usual elegir un problema y testear el tiempo variando el n° de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la *ley de Amdahl*. Los modelos de tiempo fijo son una alternativa. Otro tema de interés es la **escalabilidad**, que da una medida de usar eficientemente un número creciente de procesadores.

La **ley de Amdahl** dice que para un dado problema existe un máximo speedup alcanzable independiente del número de procesadores. Esto significa que es el algoritmo el que decide la mejora de velocidad dependiendo de la cantidad de código no paralelizable y no del número de procesadores, llegando finalmente a un momento que no se puede paralelizar más el algoritmo.

* **Tamaño del problema (W):** Función del tamaño de la entrada. Está dado por el número de operaciones básicas necesarias para resolver el problema en el algoritmo secuencial más rápido.

El ***tiempo de ejecución paralelo***, para un sistema paralelo dado, es función del tamaño del problema y el número de procesadores (Tp(W,p)) .

* **Speedup (S):** *S* es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido (*Ts*) y el tiempo de ejecución paralelo del algoritmo elegido (*Tp*) S = Ts/ Tp. Rango de valores: en general entre 0 y Sóptimo. Mide cuanto más rápido es el algoritmo paralelo con respecto al algoritmo secuencial, es decir cuánto se gana por usar más procesadores.

Factores que limitan el speedup:

* Alto porcentaje de código secuencial (*Ley de Amdahl*).
* Alto porcentaje de entrada/salida respecto de la computación.
* Algoritmo no adecuado (necesidad de rediseñar).
* Excesiva contención de memoria (rediseñar código para localidad de datos).
* Tamaño del problema (puede ser chico, o fijo y no crecer con *p*).
* Desbalance de carga (produciendo esperas ociosas en algunos procesadores).
* **Eficiencia (E):** Cociente entre speedup y Speedup Óptimo. E = S / Sóptimo. Mide la fracción de tiempo en que los procesadores son útiles para el cómputo. El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.
* **Costo**: El costo de un sistema paralelo es el producto de Tp y p. Refleja la suma del tiempo que cada procesador utiliza en la resolución del problema. Puede expresarse la eficiencia como el cociente entre el tiempo de ejecución del algoritmo secuencial conocido más rápido y el costo de resolver el problema en *p* procesadores. También suele referirse como *trabajo*.
* **Grado de concurrencia o paralelismo**: C(W) es el número máximo de tareas que pueden ejecutarse simultáneamente en cualquier momento del algoritmo paralelo. Para un W dado, el algoritmo paralelo no puede usar más de C(W) procesadores. C(W) depende sólo del algoritmo, no de la arquitectura. Supone un número ilimitado de procesadores y otros recursos, lo que no siempre es posible de tener.

|  |
| --- |
| 15.2.2 Noción de granularidad |

Cuando el número de procesadores crece, normalmente la cantidad de procesamiento en c/u disminuye y las comunicaciones aumentan. Esta relación se conoce como ***granularidad***. Puede definirse la granularidad de una aplicación o una máquina paralela como la relación entre la cantidad mínima o promedio de operaciones aritmético-lógicas con respecto a la cantidad mínima o promedio de datos que se comunican. La relación cómputo/comunicación impacta en la complejidad de los procesadores: a medida que son más independientes y realizan más operaciones A-L entre comunicaciones, también deben ser más complejos. Si la granularidad del algoritmo es diferente a la de la arquitectura, normalmente se tendrá pérdida de rendimiento.

|  |
| --- |
| 15.3 Librería Phthreads |

|  |
| --- |
| 15.3.1 Semáforos con Pthreads |

* **Thread**: proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página).
* Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones “multithreading”.
* En principio estos mecanismos fueron heterogéneos y poco portables ⇒ a mediados de los 90 la organización POSIX auspició el desarrolló de una biblioteca en C para multithreading (***Pthreads***).
* Con esta biblioteca se pueden crear *threads*, asignarles atributos, darlos por terminados, identificarlos, etc.
* Declaración de variables para descriptores de thread: ***pthread\_t pid;***
* Creación de thread:

***pthread\_create(&tid, &attr, start\_func, arg);***

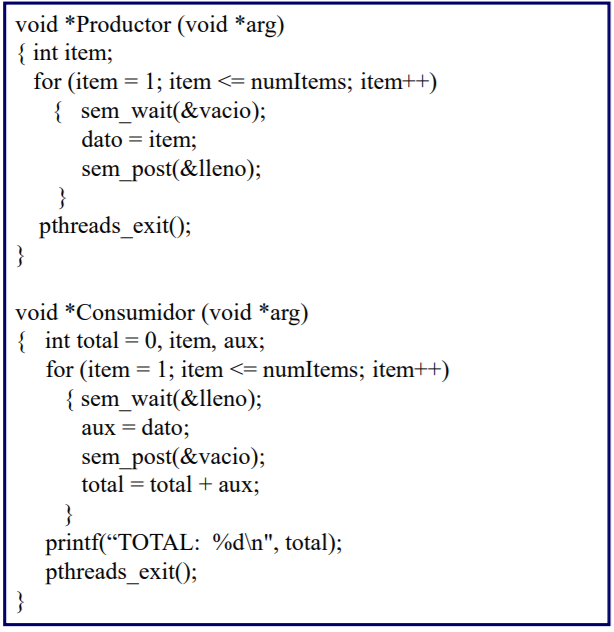
* *&tid* es la dirección de un descriptor que se llena si la creación tiene éxito.
* *&attr* es la dirección de un descriptor inicializado previamente.
* el thread comienza la ejecución llamando a *start\_func* con un argumento *arg*.
* Un thread termina su propia ejecución llamando a:***pthread\_exit(value);***
* Un thread padre puede esperar a que termine un hijo con:

***pthread\_join(tid, value\_ptr);***

* donde *tid* es un descriptor y *value\_ptr* es la dirección de una posición para el valor de retorno (que se llena cuando el hijo llama a exit).
* Los threads pueden sincronizar por semáforos (librería *semaphore.h*).
* Declaración y operaciones con semáforos en *Pthreads*:
* ***sem\_t semáforo***→ se declaran globales a los threads.
* ***sem\_init (&semaforo, alcance, inicial)*** → en esta operación se inicializa el semáforo **semáforo**. **Inicial** es el valor con que se inicializa el semáforo. **Alcance** indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos ( ≠ 0).
* ***sem\_wait(&semaforo)*** → equivale al P.
* ***sem\_post(&semaforo)*** → equivale al V.
* Existen funciones extras para: *wait* condicional, obtener el valor de un semáforo y destruir un semáforo.

**Productor/consumidor**

* Las funciones de *Productor y Consumidor* serán ejecutadas por *threads* independientes.
* Acceden a un buffer compartido (***datos***).
* El productor deposita una secuencia de enteros de 1 a ***numItems*** en el buffer.
* El consumidor busca estos valores y los suma.
* Los semáforos ***vacío y lleno*** garantizan el acceso alternativo de productor y consumidor sobre el buffer.



|  |
| --- |
| 15.3.2 Monitores con Pthreads |

**Variables mutex**

* Las secciones críticas se implementan utilizando ***mutex\_locks*** (bloqueo por exclusión mutua).
* Dos estados: ***locked*** (bloqueado) and ***unlocked*** (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un *mutex\_lock*.
* Para entrar en la SC un Thread debe bloquear el *mutex\_lock*. Y cuando sale de la SC debe desbloquear el *mutex\_lock*. Todos los *mutex\_lock* deben inicializarse como desbloqueados.im,p

**Variables condición**

* Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
* Una variable de condición siempre tiene un mutex asociada a ella.
* Con estas dos herramientas se simulan los monitores: con mutex se hace la exclusión mutua de los mismos, y con las variables condición la sincronización.