¿Qué es la concurrencia?

La concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente. Permite a distintos objetos actuar al mismo tiempo

Ejemplos concurrencia:

- Navegador Web accediendo una página mientras atiende al usuario.
- Varios navegadores accediendo a la misma página.
- Acceso a disco mientras otras aplicaciones siguen funcionando
- Los sistemas biológicos

Procesamiento secuencial

El procesamiento secuencial implica realizar una tarea o proceso a la vez, siguiendo un orden temporal estricto. Hay un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.

Procesamiento concurrente

El procesamiento concurrente se refiere a la ejecución de múltiples tareas o procesos de manera aparentemente simultánea, pero no necesariamente en paralelo.

Puede llevarse a cabo en sistemas de un solo procesador mediante la alternancia rápida entre tareas o en sistemas con múltiples procesadores. No está restringido a una arquitectura particular de hardware ni a un número determinado de procesadores. Especificar la concurrencia implica especificar los procesos **concurrentes**, su **comunicación** y su **sincronización**.

La concurrencia sin paralelismo es la multiprogramación en un procesador en donde el tiempo de CPU es compartido entre varios procesos.

El procesamiento paralelo es la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.

Un programa concurrente especifica dos o más “programas secuenciales” que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos. Un proceso es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos si el hardware lo permite.

Un programa concurrente puede tener N procesos y disponer de M procesadores.

- Procesos: Cada proceso tiene su propio espacio de direcciones y recursos.
- Procesos livianos, threads o hilos:
    - Tiene su propio PC y su pila de ejecución, pero no controla el “contexto pesado” (ej. las tablas de página).
    - Todos los hilos de un proceso comparten el mismo espacio de direcciones y recursos.

    - Programador o lenguaje proporcionan mecanismos para evitar interferencias.
    - Concurrencia provista por lenguaje o Sistema Operativo

Los procesos cooperan y compiten (o son totalmente independientes, muy raro)

Objetivos de los sistemas concurrentes

- Ajustarse al problema mundo real a resolver
- Incrementar la performance.
- Algunas 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.

Administración recursos compartidos

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 y el overloading de un proceso (la carga asignada excede su capacidad de procesamiento).

Otro problema importante que se debe evitar es el deadlock.

Competencia

Típico en Sistemas Operativos y Redes, debido a recursos compartidos.

Pueden generar deadlock o inanición

Deadlock

Un deadlock (bloqueo) es una situación en la que dos o más procesos o hilos quedan atrapados en un estado en el que ninguno puede continuar su ejecución debido a que cada uno está esperando que el otro libere un recurso que necesita.

Condiciones para que ocurra:

1. Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
2. Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.
3. No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
4. Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

Con evitar que se cumpla alguna de las condiciones mencionadas, se evita el deadlock.

Inanición

La inanición es una situación en la que un proceso o hilo no puede avanzar o realizar su trabajo debido a la incapacidad de acceder a los recursos compartidos. Esto puede deberse a que otros procesos tienen prioridad sobre él y constantemente obtienen acceso a los recursos.

Ejemplo:

En un sistema operativo con múltiples procesos compitiendo por el acceso a una impresora compartida. Cada proceso necesita imprimir un documento en la impresora. Si se implementa una política de asignación de recursos que siempre otorga acceso a la impresora al mismo grupo de procesos o al proceso de mayor prioridad, otros procesos con menor prioridad pueden quedar en estado de inanición.

Si tenemos tres procesos: A, B y C, donde A y B tienen prioridad alta y C tiene prioridad baja, y la política de asignación de recursos da preferencia a los procesos de alta prioridad, entonces el proceso C puede quedarse esperando indefinidamente para imprimir su documento.

No determinismo

En el no determinismo no hay un orden preestablecido en la ejecución, la ejecución de esta “entrada” puede generar diferentes “salidas”.

El orden en que se ejecutan las operaciones o eventos no está preestablecido y puede variar de una ejecución a otra.

Comunicación entre procesos

Modo en que se organizan y trasmiten datos entre tareas concurrentes.

Los procesos se comunican:

- Por 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.
    - Todos los procesos acceden al mismo espacio de direcciones
- Por 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 trasmitir mensajes.

Puede haber:

Máquinas de memoria compartida

Interacción modificando datos en la memoria compartida.

- Esquemas UMA
    - Todas las unidades de procesamiento tardan el mismo tiempo para acceder a la memoria (acceden a través del bus)
- Esquemas NUMA
    - Cada unidad tiene cierto bloque de memoria. Si una unidad quiere acceder a la memoria de otro bloque, tarda más.

Memoria distribuida

Procesadores conectados por una red.

- En la memoria local (no hay problemas de consistencia).
- Interacción es sólo por pasaje de mensajes.
- Grado de acoplamiento de los procesadores:
    - Multicomputadores (máquinas fuertemente acopladas).
    - Memoria compartida distribuida.
    - Clusters.
    - Redes (multiprocesador débilmente acoplado).

Sincronización

La sincronización es la posesión de información acerca de otro proceso para coordinar actividades.

Los procesos se sincronizan:

- Por exclusión mutua.
    - Asegura que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
    - Si el programa tiene secciones críticas que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
- Por condición.
    - Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Pueden estar presentes más de un mecanismo de sincronización. Un ejemplo de esto es el problema de un "buffer limitado" compartido entre productores y consumidores. Se utilizan mecanismos de exclusión mutua para garantizar que un productor o consumidor acceda al buffer de manera exclusiva y se utilizan mecanismos de condición para permitir que los productores esperen si el buffer está lleno o que los consumidores esperen si el búfer está vacío.

Granularidad

La granularidad de una aplicación está dada por la relación entre el cómputo y la comunicación.

- Hay una relación y adaptación a la arquitectura.
- Grano fino y grano grueso
    - Poco código, comunicación → Grano fino
    - Mucho código, comunicación → Grano grueso.

Requerimientos de un lenguaje de programación concurrente:

Los lenguajes de programación concurrente deberán proveer:

- Indicar las tareas/procesos que se pueden ejecutar concurrentemente.
- Mecanismos de sincronización.
- Mecanismos de comunicación.

Problemas

1. Gestionar la exclusión mutua y sincronización agrega complejidad al diseño de programas concurrentes.
2. os procesos iniciados en programas concurrentes pueden no mantener su "liveness", lo que puede indicar deadlocks o una asignación ineficiente de recursos.
3. Dos ejecuciones del mismo programa no son necesariamente idénticas. Esto complica la interpretación y depuración del código.
4. Reducción de Performance por Overhead asociados a cambios de contexto, comunicación entre procesos, sincronización, etc.
5. Optimización para la ejecución paralela pueden requerir mucho tiempo y la paralelización de algoritmos secuenciales puede ser difícil.
6. Hay que adaptar el software concurrente al hardware paralelo subyacente.

Interferencia

La interferencia es una situación en la que un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Para evitar la interferencia en programación concurrente se utilizan las acciones atómicas y técnicas de sincronización.

Prioridad

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.

Atomicidad

Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas. Una acción atómica se ejecuta como una única unidad indivisible para otros procesos

En la ejecución de un programa concurrente hay un intercalado de las acciones atómicas ejecutadas por procesos individuales. El número de posibles historias (trace) de un programa concurrente es enorme y no todas son válidas, por eso se debe asegurar un orden temporal entre las acciones que ejecutan los procesos, es decir, como las tareas se intercalan. Para ello, deben fijarse restricciones.

Atomicidad de grano fino

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

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 será atómica.

Es decir-> si una variable que utiliza un proceso no es usada por ninguno otro, la evaluacion sera atomica.

Referencia crítica

En una expresión, es la referencia a una variable que es modificada por otro proceso. Toda referencia crítica es a una variable simple leída y escrita atómicamente.

Propiedad de “A lo sumo una vez”

como mucho 1 variable compartida

Una sentencia cumple ASV si hay a lo sumo una variable compartida, y esta tiene que ser referenciada a lo sumo una vez

Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución parece atómica, ya que 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*

Acciones atómicas condicionales e incondicionales

Acciones atómicas incondicionales: son acciones que se ejecutan sin considerar ninguna condición previa. Siempre se realizan de manera atómica, independientemente de las circunstancias. < x = 1 >

Acciones atómicas condicionales: son acciones atómicas que se ejecutan cuándo se cumple una cierta condición previa. < await S; x = 1 > (puede ser solo < await S >)

Propiedades de seguridad y vida

Seguridad (safety)

- Nada malo le ocurre a un proceso, no se producen estados inconsistentes o resultados incorrectos debido a la ejecución concurrente.
  - Una falla de seguridad indica que algo anda mal.
  - Ejemplos de propiedades de seguridad: exclusión mutua, ausencia de interferencia entre procesos, partial correctness.

Vida (liveness)

- Eventualmente ocurre algo bueno con una actividad. Asegura que los procesos no queden bloqueados (deadlocks) y que las actividades puedan avanzar y completarse.
  - Una falla de vida indica que las cosas dejan de ejecutar.
  - Ejemplos de vida: terminación, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC.

Fairness

Una política de scheduling se refiere a un conjunto de reglas y algoritmos utilizados para determinar cuál será el próximo proceso o hilo que se ejecutará en un sistema concurrente o en un sistema operativo. Cuando se habla de ejecutar el próximo proceso o hilo, se está hablando de ejecutar las acciones atómicas de dicho proceso.

Fairness se relaciona con la equidad en la ejecución de procesos o hilos en un sistema concurrente. Se esfuerza por garantizar que todos los procesos tengan la oportunidad de avanzar y realizar sus operaciones, evitando situaciones de bloqueo o inanición.

Fairness trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás, es decir, que se ejecuten acciones atómicas de todos los procesos.

Tipos de Fairness (varias definiciones porque soy tonta):

Fairness Incondicional: Esta política garantiza que toda acción atómica elegible eventualmente se ejecute, sin importar las condiciones o restricciones. Se enfoca en garantizar que todas las acciones tengan una oportunidad justa de ejecutarse.

Fairness Débil: Además de ser incondicionalmente justa, esta política también garantiza que las acciones atómicas condicionales que se vuelven elegibles eventualmente se ejecuten siempre que su condición se cumpla. Esto evita que las acciones condicionales queden en espera indefinidamente si su condición se cumple.

Fairness Fuerte: Es la política más exigente en términos de equidad. Además de ser incondicionalmente justa, garantiza que las acciones atómicas condicionales se ejecuten con una frecuencia infinita si su condición se cumple. Esto evita que las acciones condicionales queden en espera incluso si su condición se cumple raramente.

Fairness Incondicional:

- Garantiza que toda acción atómica elegible eventualmente se ejecute, independientemente de las condiciones o restricciones.
- Se centra en proporcionar una oportunidad justa para todas las acciones sin importar sus condiciones.
- No tiene en cuenta si las condiciones de las acciones condicionales se cumplen o no, todas las acciones elegibles se ejecutarán en algún momento.

Fairness Débil:

- Además de ser incondicionalmente justa, también se preocupa por las acciones condicionales.
- Garantiza que las acciones atómicas condicionales que se vuelven elegibles eventualmente se ejecuten siempre que su condición se cumpla.
- Evita que las acciones condicionales queden en espera indefinidamente si su condición se cumple.
- Fairness Fuerte:
- La política más exigente en términos de equidad.
- Además de ser incondicionalmente justa, garantiza que las acciones atómicas condicionales se ejecuten con una frecuencia infinita si su condición se cumple.
- Evita que las acciones condicionales queden en espera, incluso si su condición se cumple raramente.

Fairness Incondicional. Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

Fairness Débil. Una política de scheduling es débilmente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda se convierte en true y de allí en adelante permanece true.

Fairness Fuerte. Una política de scheduling es fuertemente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda es true con infinita frecuencia.

Fairness incondicional: significa que todas las tareas que están listas para ejecutarse eventualmente se ejecutan, sin importar si son condicionales o no.

Fairness débil: significa que además de la fairness incondicional, las tareas condicionales que se vuelven elegibles se ejecutan, siempre y cuando su condición se mantenga verdadera hasta que la tarea sea asignada.

Fairness fuerte: significa que además de la fairness incondicional, las tareas condicionales que se vuelven elegibles se ejecutan, incluso si su condición cambia de valor muchas veces.

Las propiedades de vida se centran en garantizar que los procesos puedan avanzar y completar sus tareas, evitando situaciones de bloqueo y asegurando que las acciones críticas se ejecuten eventualmente. Dependen de la política de scheduling porque esta última determina el comportamiento de la ejecución de procesos concurrentes, y diferentes políticas de scheduling pueden afectar la equidad y la capacidad de los procesos para avanzar y completar sus tareas.

Locks

Para el problema de la sección crítica se implementan de acciones atómicas en software conocidas como locks

Busy waiting

Un proceso chequea repetidamente una condición hasta que sea verdadera:

- Se implementa con instrucciones de cualquier procesador.
- Ineficiente en multiprogramación, aceptable si cada proceso ejecuta en su procesador.

<SC>

Resolver <SC> puede llevarme a resolver <await b;s>

Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional ⟨ S;⟩ ⇒ SCEnter ; S; SCExit

Para una acción atómica condicional ⟨await (B) S;⟩ ⇒ SCEnter ; while (not B) {SCExit; SCEnter;} S; SCExit;

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

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

Para reducir contención de memoria ⇒ SCEnter ; while (not B) {SCExit; Delay; SCEnter;} S; SCExit;

¿Qué propiedades deben satisfacer los protocolos de entrada y salida a una sección critica?

Propiedades de Seguridad (Safety):

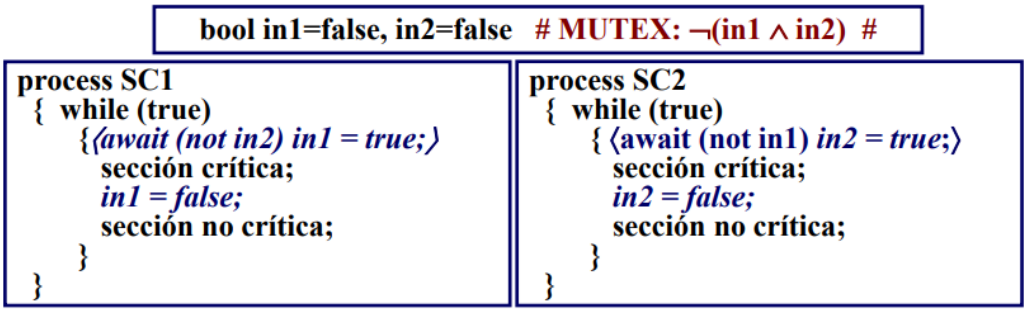
1. Exclusión Mutua: Como máximo un proceso está en su sección crítica en un momento dado. Evitar que dos o más procesos estén en la misma sección crítica al mismo tiempo.

Propiedades de Vida (Liveness):

2. Ausencia de Deadlock (Livelock): Si dos o más procesos intentan entrar en sus secciones críticas, al menos uno de ellos tendrá éxito. Evitar que un proceso espere indefinidamente a que se libere un recurso que nunca se libera porque otros procesos también lo están esperando, básicamente evitar que los procesos queden bloqueados en un punto muerto y no puedan avanzar.

3. Ausencia de Demora Innecesaria: Si un proceso intenta entrar en su sección crítica y los otros procesos están en secciones no críticas o ya han terminado, el primer proceso no debe ser impedido de entrar en su sección crítica. Evitar espera innecesaria de un recurso ya disponible.

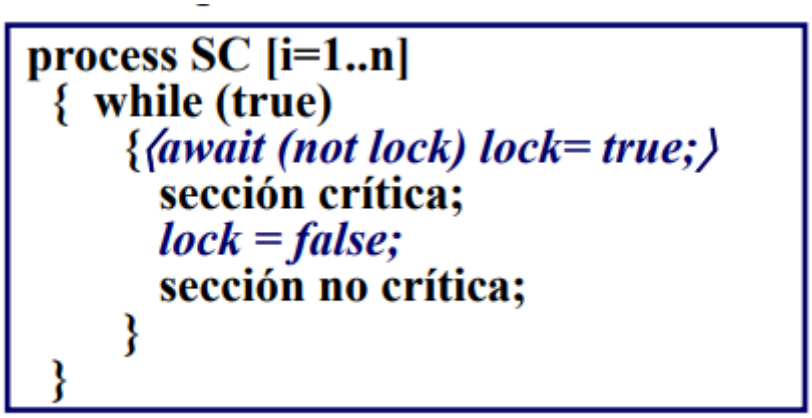
4. Eventual Entrada: Garantiza que un proceso que intenta entrar en su sección crítica tiene la posibilidad de hacerlo en algún momento (eventualmente lo hará). Se debe evitar la inanición.



Satisface las 4 propiedades

- Exclusión mutua: por construcción ambos se excluyen en el acceso a la SC.
- Ausencia de deadlock: nunca van a ser in1 y in2 true al mismo tiempo
- Ausencia de demora innecesaria: si SC1 está fuera de su SC o terminó, in1 es false; si SC2 está tratando de entrar a SC y no puede, in1 es true.
- Eventual Entrada: Se garantiza la eventual entrada con una política de scheduling fuertemente fair.

n procesos



Spin locks

Tienen como objetivo hacer “atómico” el await de grano grueso usando instrucciones atómicas disponibles en la mayoría de los procesadores.

En este tipo de soluciones los procesos se quedan iterando (spinning) mientras esperan que se limpie lock. Cumple las 4 propiedades si el scheduling es fuertemente fair. Política débilmente fair es aceptable

Es posible que alguno no entre nunca si el scheduling no es fuertemente fair ya que no hay orden.

Test & Set

```
bool TS (bool ok);
{  < bool inicial = ok;
       ok = true;
       return inicial; >
}
```

Test-and-test-and-set

TS (VIVA TAYLOR SWIFT) escribe siempre en lock aunque el valor no cambie, una mejora es Test-and-Test-and-Set

```
bool lock=false;
process SC[i=1 to n]
{  while (true)
     {  while (TS(lock)) skip ;
        sección crítica;
        lock = false;
        sección no crítica;
     }
}
```

```
while (lock) skip;
while (TS(lock))
      while (lock) skip;
```

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

Algoritmo Tie-breaker

El objetivo es el mismo que los Spin locks solo que en este caso se respeta el orden, para ello requiere scheduling sólo débilmente fair y no usa instrucciones especiales (más complejo).

Usa una variable por cada proceso para indicar que el proceso comenzó a entrar a la SC, y una variable adicional para romper empates que es compartida y de acceso protegido. Demora (quita prioridad) al último en comenzar su entry protocol

Solución Grano Grueso

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
   while (true) {      ultimo = 1; in1 = true;
                       ⟨await (not in2 or ultimo==2);⟩
                       sección crítica;
                       in1 = false;
                       sección no crítica;
   }
}

process SC2 {
   while (true) {      ultimo = 2; in2 = true;
                       ⟨await (not in1 or ultimo==1);⟩
                       sección crítica;
                       in2 = false;
                       sección no crítica;
   }
}
```

Solución Grano Fino

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
   while (true) {      in1 = true; ultimo = 1;
                       while (in2 and ultimo == 1) skip;
                       sección crítica;
                       in1 = false;
                       sección no crítica;
   }
}

process SC2 {
   while (true) {      in2 = true; ultimo = 2;
                       while (in1 and ultimo == 2) skip;
                       sección crítica;
                       in2 = false;
                       sección no crítica;
   }
}
```

Este algoritmo es complejo y costoso en tiempo, sobre todo cuando hay n procesos *(ni me molesto en poner la imagen no lo entiendo)*

Algoritmo Ticket

El objetivo es el mismo que los Spin locks solo que se respeta el orden. En este caso se reparten números y se espera a que sea el 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.

Grano grueso

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

{ TICKET: proximo > 0 ^ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]== proximo) ^
   (turno[i] >0) ⇒ (∀j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}

process SC [i: 1..n]
{ while (true)
   {  < turno[i] = numero;  numero = numero +1; >
      < await turno[i] == proximo; >
      sección crítica;
      < proximo = proximo + 1; >
      sección no crítica;
   }
}
```

- Hay a lo sumo un proceso en la SC
    - Número es leído e incrementado en una acción atómica y próximo es incrementado en una acción atómica
- La ausencia de deadlock y de demora innecesaria por valores de turno únicos
- Scheduling débilmente fair asegura eventual entrada

Grano fino

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

process SC [i: 1..n]
{ while (true)
   {  turno[i] = FA (numero, 1);
      while (turno[i] <> proximo) skip;
      sección crítica;
      proximo = proximo + 1;
      sección no crítica;
   }
}
```

Fetch-and-Add

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

En el Algoritmo Tickert si no existe FA se debe simular con una SC y la solución puede no ser fair.

Algoritmo Bakery

Si no existe FA se usa este, es más complejo, pero es fair y no requiere instrucciones especiales.

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.

Grano grueso

```
int turno[1:n] = ([n] 0);

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{  while (true)
       {    ⟨ turno[i] = max(turno[1:n] + 1; ⟩
            for [j = 1 to n st j <> i]  ⟨ await (turno[j] == 0 or turno[i] < turno[j]); ⟩
            sección crítica
            turno[i] = 0;
            sección no crítica
       }
}
```

Esta solución de grano grueso no es implementable directamente

Grano fino

```
int turno[1:n] = ([n] 0);

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{  while (true)
       {  turno[i] = 1;   //indica que comenzó el protocolo de entrada
          turno[i] = max(turno[1:n]) + 1;
          for [j = 1 to n st j != i]        //espera su turno
                while (turno[j] != 0)  and (  (turno[i],i) > (turno[j],j) ) →  skip;
          sección crítica
          turno[i] = 0;
          sección no crítica
       }
}
```

Sincronización barrier

Punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.

Puede haber

- Un contador compartido:
  - Cada proceso incrementa una variable cantidad al llegar
  - Cuando cantidad es n los procesos pueden pasar

```
int cantidad = 0;
process Worker[i=1 to n]
{ while (true)
        {    código para implementar la tarea i;
             ⟨ cantidad = cantidad + 1; ⟩
             ⟨ await (cantidad == n); ⟩
         }
}
```

  - Se puede implementar con: FA(cantidad,1); while (cantidad <> n) skip;
  - Si no existe FA puede distribuirse cantidad usando n variables (arreglo arribo[1...n]). Ineficiente
  - Puede usarse un coordinador (un proceso más): conjunto de valores adicionales y cada worker espera por un único valor

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{ while (true)
        {    código para implementar la tarea i;
             arribo[i] = 1;
             ⟨ await (continuar[i] == 1); ⟩
             continuar[i] = 0;
        }
}

process Coordinador
{ while (true)
        {    for [i = 1 to n]
                  { ⟨ await (arribo[i] == 1); ⟩
                    arribo[i] = 0;
                  }
             for [i = 1 to n] continuar[i] = 1;
         }
}
```

Con coordinador:

- Necesidad de un proceso adicional
    - Costo adicional tanto en términos de recursos de procesamiento como de tiempo de ejecución el cual es proporcional a n.
- Simples de implementar y funciona bien cuando se trabaja con un número reducido de procesos.
    - La sobrecarga de comunicación entre el coordinador y los trabajadores no tiene un impacto significativo en el tiempo de espera.
    - A medida que el número de procesos aumenta, el coordinador puede convertirse en un cuello de botella.
        - Solución: Combinar las acciones de Workers y Coordinador, haciendo que cada Worker sea también Coordinador
            - Se tienen Workers en forma de árbol en donde las señales de arribo van hacia arriba en el árbol, y las de continuar hacia abajo (combining tree barrier).
            - Rendimiento mejorado en sistemas con un gran número de procesos.
            - Requieren un diseño más elaborado para garantizar sincronización correcta

El butterfly barries chupo jajajaja, banco no entiendo nada eso

Butterfly barrier para 8 procesos

Semáforos

Los protocolos busy waiting son complejos y no hay clara separación entre variables de sincronización. Difícil diseñar para probar corrección. Técnica ineficiente

Los semáforos es una herramienta que se utiliza para solucionar lo anterior mencionado. Es una instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones (métodos) atómicas: P y V. Internamente el valor de un semáforo es un entero no negativo:

- V → Señala la ocurrencia de un evento (incrementa).
- P → Se usa para demorar un proceso hasta que ocurra un evento (decrementa).

- Semáforo general (o counting semaphore)
  P(s): < await (s > 0) s = s-1; >
  V(s): < s = s+1; >
- Semáforo binario
  P(b): < await (b > 0) b = b-1; >
  V(b): < await (b < 1) b = b+1; >

Barrera para dos procesos

generalmente inicializado en 0. Un proceso señala el evento con V(s); otros procesos esperan la ocurrencia del evento ejecutando P(s).

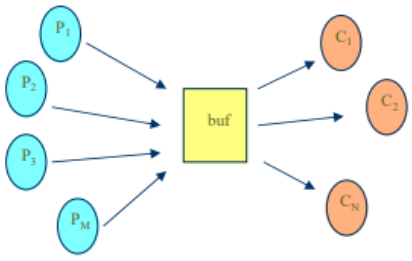
```
sem llega1=0, llega2=0;
process Worker1
{ .....
   V(llega1); P(llega2);
   .....
}

process Worker2
{ .....
   V(llega2); P(llega1);
   .....
}
```

Puede usarse la barrera para dos procesos para implementar una butterfly barrier para n, o sincronización con un coordinador central.

Semáforos binarios divididos

Tiene dos partes: un semáforo binario de acceso que controla la disponibilidad del recurso y otro semáforo binario de conteo que lleva un registro de cuántos procesos acceden al recurso. Pueden ser más de dos semáforos pero se tiene que cumplir que si sumo los semáforos me van a dar o 0 o 1.

Ejemplo: buffer unitario compartido con múltiples productores y consumidores. Dos operaciones: ***depositar*** y ***retirar*** que deben alternarse.




```
typeT buf;  sem vacio = 1, lleno = 0;

process Productor [i = 1 to M]
{ while(true)
    { …
      producir mensaje datos
      P(vacio); buf = datos; V(lleno); #depositar
    }
}

process Consumidor[j = 1 to N]
{ while(true)
    { P(lleno); resultado = buf;  V(vacio); #retirar
      consumir mensaje resultado
      …
    }
}
```

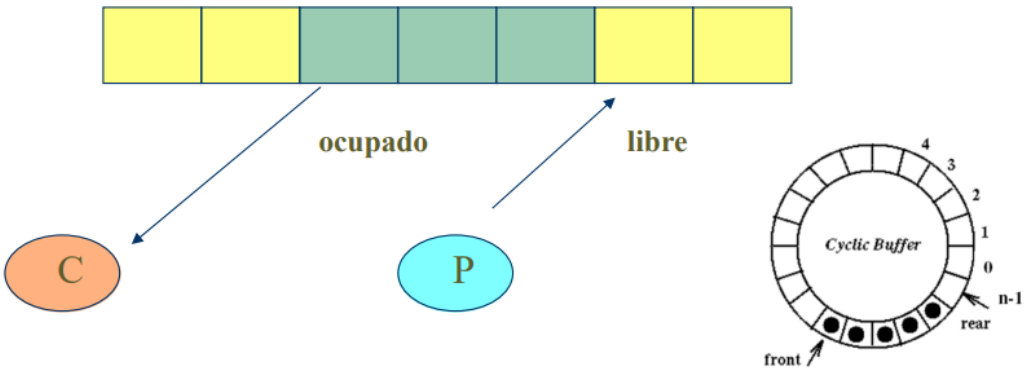
vacio y ***lleno*** (juntos) forman un ***"semáforo binario dividido"***.

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:

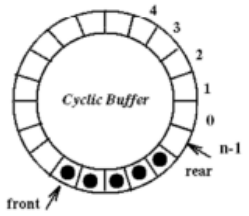
Existe UN productor y UN consumidor que depositan y retiran elementos del buffer.




```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;
process Productor
{ while(true)
    { …
      producir mensaje datos
      P(vacio); buf[libre] = datos; libre = (libre+1) mod n;  V(lleno); #depositar
    }
}
process Consumidor
{ while(true)
    { P(lleno); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(vacio); #retirar
      consumir mensaje resultado
      …
    }
}
```

Más de un productor y/o consumidor

```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;
sem mutexD = 1, mutexR = 1;
process Productor [i = 1..M]
{ while(true)
    {  producir mensaje datos
       P(vacio);
       P(mutexD); buf[libre] = datos; libre = (libre+1) mod n; V(mutexD);
       V(lleno);
    }
}
process Consumidor [i = 1..N]
{ while(true)
    {  P(lleno);
       P(mutexR); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(mutexR);
       V(vacio);
       consumir mensaje resultado
    }
}
```



Si hay deadlock lo más común es romper la espera circular haciendo que un proceso sea distinto al resto

```
sem  tenedores [5] = {1,1,1,1,1};

process Filososfos[i = 0..3]
{ while(true)
    {  P(tenedor[i]);  P(tenedor[i+1]);
       comer;
       V(tenedor[i]);  V(tenedor[i+1]);
    }
}

process Filososfos[4]
{ while(true)
    {  P(tenedor[0]);  P(tenedor[4]);
       comer;
       V(tenedor[0]);  V(tenedor[4]);
    }
}
```

Passing the Baton

Es una técnica general utilizada para implementar sentencias await. En esta técnica, un proceso que necesita acceso a una sección crítica mantiene el "bastón" o "testimonio" que le otorga el permiso para ejecutar. Una vez que un proceso ha finalizado su tarea o ha salido de su sección crítica, pasa el bastón al siguiente proceso en la secuencia. Si no hay procesos esperando el bastón (es decir esperando entrar a la SC), este se libera para que lo tome el próximo proceso que trata de entrar.

La utilidad principal de esta técnica radica en garantizar que los procesos se ejecuten de manera ordenada y sin interferencias entre ellos.

Esta técnica emplea SBS para brindar exclusión y despertar procesos demorados.

Es importante diferencia el control de acceso con la exclusión mutua de un recurso.

Ejemplo:

En una montaña hay 30 escaladores que en una parte de la subida deben utilizar un único paso de a uno a la vez y de acuerdo al orden de llegada al mismo.

```
cola c;
sem mutex = 1, espera[30] = ([30] 0);
boolean libre = true;

Process Escalador[id: 0..29]
{ int aux;
  -- llega al paso
  P (mutex);
  if (libre)  { libre = false;  V (mutex); }
  else  { push (C, id);  V (mutex); P (espera[id]); };
  //Usa el paso con Exclusión Mutua
  P (mutex);
  if (empty (C)) libre = true
  else  { pop (C, aux);   V (espera[aux]);  };
  V (mutex);
}
```

Ejemplo

Hay control de acceso, hay lectores simultáneos.

```
int nr = 0, nw = 0, dr = 0, dw = 0;          sem e = 1, r = 0, w = 0:

process Lector [i = 1 to M]
{  while(true)
     {  P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        if (dr > 0) {dr = dr – 1; V(r); }
        else V(e);
        lee la BD;
        P(e);
        nr = nr – 1;
        if (nr == 0 and dw > 0)
              {dw = dw – 1; V(w); }
        else V(e);
      }
}
```

```
process Escritor [j = 1 to N]
{ while(true)
   {  P(e);
      if (nr > 0 or  nw > 0)
              {dw=dw+1; V(e); P(w);}
      nw = nw + 1;
      V(e);
      escribe la BD;
      P(e);
      nw = nw – 1;
      if (dr > 0) {dr = dr - 1; V(r); }
      elseif (dw > 0) {dw = dw - 1; V(w); }
      else V(e);
   }
}
```

Semáforos privados

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.

Ejemplo:

Hay C chicos y hay una bolsa con caramelos limitada a N caramelos administrada por UNA abuela. Cuando todos los chicos han llegado llaman a la abuela, y a partir de ese momento ella N veces selecciona un chico aleatoriamente y lo deja pasar a tomar un caramelo.

```
int contador = 0;      bool  seguir = true;
sem mutex = 1, espera_abuela = 0, barrera = 0, espera_chicos[C] = ([C] 0),  listo = 0;

Process Chico[id: 0..C-1]
{  int i;
   P(mutex);
   contador = contador + 1;
   if (contador == C)
      {  for i = 1..C  →  V(barrera);
         V(espera_abuela); }
   V(mutex);
   P(barrera);
   P(espera_chicos[id]);
   while (seguir)
      {  --tomar caramelo
         V(listo);
         --comer caramelo
         P(espera_chicos[id]);
      }
}
```

```
Process Abuela
{ int i, aux;
   P(espera_abuela);
   for i = 1..N
      { aux = (rand mod C);
        V(espera_chicos[aux]);
        P(listo);
      }
   seguir = false;
   for aux = 0..C-1  →  V(espera_chicos[aux]);
}
```

Ejemplo: Utilizar una cola que guarda el id, en la posición dependiendo del tiempo que va a tardar, asi se hace un SJN.

Alocación Shortest Job Next (SJN)

```
bool libre = true;  Pares = set of (int, int) = ∅;  sem e = 1, b[n] = ([n] 0);

Process Cliente [id: 1..n]
{ int sig;
  //Trabaja
  tiempo = //determina el tiempo de uso del recurso//
  P(e);
  if (! libre) { insertar (tiempo, id) en Pares;
                 V(e);
                 P(b[id]);
               }
  libre = false;
  V(e);
  //USA EL RECURSO
  P(e);
  libre = true;
  if (Pares ≠ ∅ ) { remover el primer par (tiempo, sig) de Pares;
                    V(b[sig]);
                  }
  else V(e);
}
```

¿Que modificaciones deberían realizarse para respetar el orden de llegada?

¿Que modificaciones deberían realizarse para generalizar la solución a recursos de múltiple unidad?

Monitores

Con los semáforos hay variables compartidas globales a los procesos, sentencias de control de acceso a la sección crítica dispersas en el código, al agregar procesos, se debe verificar acceso correcto a las variables compartidas, aunque exclusión mutua y sincronización por condición son conceptos distintos, se programan de forma similar. Una solución a todos estos problemas mencionados son los monitores.

Los monitores son módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos. Son un mecanismo de abstracción de datos:

- Encapsulan las representaciones de recursos.
- Brindan un conjunto de operaciones que son los únicos medios para manipula r esos recursos.

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

La exclusión mutua esta dada por los monitores. Es implícita. Un monitor va a estar siendo utilizado en un momento solo por UN proceso. Los demás se encolan y cuando el proceso termine de ser usado, se va a elegir uno encolado de forma no determinística.

En los programas concurrentes con monitores hay 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.

Los monitores tienen interfaz y cuerpo. Sólo los nombres de los procedures (la interfaz) son visibles desde afuera. Los procedures pueden acceder sólo a variables permanentes, sus variables locales, y parámetros que le sean pasados en la invocación.

La comunicación entre procesos será haciendo uso del monitor que va a contener los datos compartidos, que son variables o estructuras de datos accesibles por varios procesos. Dentro del monitor van a haber variables condicionales que permitirán a los procesos esperar o notificar eventos específicos.

```
monitor NombreMonitor {
  declaraciones de variables permanentes;
  código de inicialización

  procedure op1 (par. formales1)
    {  cuerpo de op1
    }
  .....
  procedure opn (par. formalesn)
    {  cuerpo de opn
    }
}
```

Sincronización por condición

Es con variables condición:

cond cv;

cv es una cola de procesos demorados, no visible directamente al programador.

- wait(cv) → el proceso se demora al final de la cola de cv y deja el acceso exclusivo al monitor.
- signal(cv) → despierta al proceso que está al frente de la cola (si hay alguno) y lo saca de ella. El proceso despertado recién podrá ejecutar cuando readquiera el acceso exclusivo al monitor.
- signal_all(cv) → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv.

Operaciones adicionales que NO SON USADAS EN LA PRÁCTICA sobre las variables condición:

- empty(cv) → retorna true si la cola controlada por cv está vacía.
    - Solución: usar una variable contadora de la cantidad de procesos dormido. Cuando un proceso se va a dormir incrementa una variable → ej: espera ++. El proceso que se encarga de despertalor la decrementa.
- 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.
    - Solución: utilizo una cola.
- minrank(cv) → función que retorna el mínimo ranking de demora.
    - Solución: utilizo una cola ordenada de min a max y siempre que hago signal(cv) despierto al mínimo.

Disciplinas

La diferencia principal entre ambas disciplinas recae en quien es el que va a ser encolado nuevamente para esperar el uso del monitor. En el caso de Signal and continued va a ser encolado (en la cola no determinística) el proceso despertado y en Signal and wait va a ser encola el proceso que despierta.

Signal and continued

El proceso que hace el signal continúa 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).

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.

Diferencia entre wait/signal con P/V

WAIT	P
El proceso siempre se duerme	El proceso sólo se duerme si el semáforo es 0.

SIGNAL	V
Si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior.	Incrementa el semáforo para que un proceso dormido o que hará un P continue. No sigue ningún orden al despertarlos.

Passing The Condition

Se refiere a la transferencia de la condición que se está esperando de un proceso a otro. La idea central es permitir que un proceso notifique a otro sobre un cambio en una condición específica, lo que generalmente se realiza mediante el uso de variables condicionales.

Si hay un proceso dormido esperando por determinada condición, el que lo estaba bloqueando lo despierta, básicamente le pasa la condición para que pueda seguir ejecutándose. Si no hay ningún proceso bloqueado, se hará verdadera la condición para que el próximo proceso que venga pueda usarla. La idea central es que un proceso notifique a otro sobre un cambio en una condición específica

La utilidad en la resolución de problemas con monitores es que se respete el orden de espera de los procesos.

La relación entre passing the condition y passing the baton es que ambas técnicas se basan en el concepto de transferir el uso o acceso de recursos críticos entre procesos, aunque passing the condition se centra más en la sincronización haciendo uso de condiciones y passing the baton en la transferencia de control de un proceso a otro.

Técnicas de Sincronización Lectores y escritores: Broadcast Signal

```
monitor Controlador_RW
{  int nr = 0, nw = 0;
   cond  ok_leer, ok_escribir
   procedure pedido_leer( )
      { while (nw > 0) wait (ok_leer);
        nr = nr + 1;
      }
   procedure libera_leer( )
      { nr = nr - 1;
        if (nr == 0) signal (ok_escribir);
      }
   procedure pedido_escribir( )
      { while (nr>0 OR nw>0) wait (ok_escribir);
        nw = nw + 1;
      }
   procedure libera_escribir( )
      { nw = nw - 1;
        signal (ok_escribir);
        signal_all (ok_leer);
      }
}
```

El monitor arbitra el acceso a la BD. Los procesos dicen cuándo quieren acceder y cuándo terminaron

Shortest Job Next

```
monitor Shortest_Job_Next
{ bool libre = true;
  cond turno[N];
  cola  espera;

  procedure request (int id, int tiempo)
     { if (libre) libre = false
       else { insertar_ordenado(espera, id, tiempo);
              wait (turno[id]);
            };
     };

  procedure release ()
     { if (empty(espera)) libre = true
       else  { sacar(espera, id);
               signal(turno[id]);
             };
     };
}
```

Se usa cola ordenada y variables condiciones privadas