# Sistemas Concurrente y Distribuidos: Tema 2
# 10 de Octubre del 2020
# A year to Forget

**Daniel Monjas Miguélez**

# Índice

# 1. Monitores como mecanismo de alto nivel

Inconvenientes de usar mecanismos como los semásforos:

- Basados en variables globles, esto impide un diseño modular y reduce la escalabilidad (incorporar más procesos al programa suele requerir la revisión del uso de las variables globales).
- El uso y función de las variables (protegidas de los semáforos) no se hace explícito en el programa, lo cual dificulta razonar sobre la correción de los programas.
- Las operaciones se encuentran dispersas y no protegidas (posibilidad de errores).

Por tanto, es necesario un mecanismo que permita el acceso estructurado a los datos del programa y la encapsulación de las estructuras de datos y que, además, proporcione herramientas para garantizar la exclusión mutua e implementar condiciones de sincronización.

**Concepto de monitor**: es un mecanismo de alto nivel que permite definir objetos abstractos compartidos:

- Una colección de variables encapsuladas (datos) que representan un recurso compartido por varios recursos.
- Un conjunto de procedimientos para manipular el recurso que afectan a las variables encapsuladas.

Ambos conjuntos de elementos permiten al programador invocar a los procedimientos de forma que en ellos,

- Se garantiza el acceso en exclusión mutua a al variables encapsuladas.
- Se implementan la sincronización requerida por el problema mediante esperas bloqueadas (los procesos del programa se suspenden sin consumir ciclos del procesador).

**Características generales de los monitores**

**Concepto de monitor:** módulo de las aplicaciones concurrentes que se usa como un objeto al que se accede concurrentemente por los procesos.

- **Modularidad** en el desarrollo de programas y aplicaciones.
- **Programa=** $\{monitores, procesos\}$

- **Estructuración** en el acceso a tipos de datos, variables compartidas, etc.
- **Capacidad de modelado** de interacciones cooperativas y competitivas entre procesos concurrentes lo más general posible.
- **Ocultación** a los procesos de las operaciones de sincronización sobre datos compartidos.
- **Reusabilidad** basada en parametrización de los módulos monitor.
- **Verificación** mediante reglas más simples que las de los semáforos.

La exclusión muta en el acceso a los procedimientos del monitor está garantizada por la propia definición del módulo monitor. La implementación del monitor garantiza que nunca dos procesos estarán ejecutando simultáneamente algún procedimiento del monitor (el mismo o distintos).

**Componentes de un monitor:**

- **Variables permanetes $->$ son el estado interno del monitor.** Sólo pueden ser accedidas dentro del monitor (en el cuerpo de los procedimientos y código de inicialización). Permanecen sin modificaciones entre dos llamadas consecutivas a procedimientos del monitor.
- **Procedimientos $->$ modifican el estado interno (garantizando la exclusión mutua durante dicho cambio).** Pueden tener variables y parámetros locales, que toman un nuevo valor en cada activación del procedimiento. Algunos (o todos) constituyen la interfaz externa del monitor y podrán ser llamados por los procesos que comparten el recurso.
- **Código de inicialización $->$ fija el estado interno inicial.** Este bloque es opcional, se ejecuta una única vez, antes de cualquier llamada a procedimientos del monitor.

La principal ventaja del monitor es que la implementación de las operaciones se puede cambiar sin modificar el código de los programas que las utilizan.

**Características de la programación con monitores:**

- Centralización de recursos críticos.
- Monitor como versión descentralizada del monitor monolítico de los sistemas operativos.
- Mantiene la seguridad de los datos compartidos incluso si sus procedimientos son ejecutados concurrentemente por múltiples procesos.

- Sólo un procedimiento ejecutado por un solo proceso dentro del monitor, aunque puede interrumpirse y reemprenderse posteriormente.
- Posibilidad de ejecución concurrente de monitores no relacionados.

**Instanciación de los monitores:** después de ejecutarse el código de inicialización, un monitor es un módulo pasivo del programa y el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos.

**El código de los procedimientos de los monitores ha de ser reentrante:**

- No dependerá de datos estáticos globales.
- No puede modificar su propio código.
- No puede llamar a funciones no-reentrantes.

**Cola del monitor para exclusión mútua:**

- Si un proceso está dentro del monitor y otro proceso intenta ejecutar un procedimiento del monitor, éste último proceso queda bloqueado y se inserta (espera) en la cola del monitor.
- Cuando un proceso abandona el monitor (finaliza la ejecución del procedimiento), se desbloquea un proceso de la cola, que ya puede entrar al monitor.
- Si la cola del monitor está vacía, el monitor está libra y el primer proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor.
- Para garantizar la propiedad de vivacidad del programa con monitores, la planificación de la cola del monitor debe seguir una política FIFO.

**Operaciones de sincronización en monitores:** para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición. Los monitores sólo disponen de sentencias de bloqueo y activación. Los valores de las variables permanentes del monitor determinan si la condición se cumple o no.

**El TDA cond de los monitores:** la sincronización y la exclusión mutua (en el acceso a recursos que protege el monitor) se programa dentro del monitor con variables del TDA cond (soporte de señales).

La exclusión mutua se levanta como consecuencia de ejecutar $c.wait() \rightarrow$ se evita el bloqueo del monitor. Se cede el acceso al monitor al proceso señalado (c.signal() con semántica desplazante) $\rightarrow$ se evita el robo de señal.

**Verificación de programas con monitores:** la verificación de la corrección de un programa concurrente con monitores supone:

- Probar la corrección de cada monitor.
- Probar la corrección de cada proceso de forma aislada
- Probar la corrección de la ejecución concurrente de los procesos implicados.

El programador no puede conocer a priori la traza concreta de llamadas a los procedimientos del monitor.

**Característica de un IM:**

- Es una función lógica que se puede evaluar como true o false en cada estado del monitor a lo largo de la ejecución.
- Su valor de verdad depende de la traza del monitor y de los valores de las variables permanentes de dicho monitor.
- Debe ser cierto en cualquier estado del programa concurrente, excepto cuando un proceso está ejecutando código del monitor, en E.M. (está en proceso de actualización de los valores de las variables permanentes).

**Axiomas de verificación de los monitores:**

- **Axioma (inicialización variables)**: $\{V\}$ inicialización variables permanentes $\{IM\}$. El invariante del monitor debe ser cierto en su estado inicial, justo después de la inicialización de las variables permanentes.
- **Axioma (procedimientos del monitor)**: $\{IN \wedge IM\}$ $procedimiento_i$ $\{OUT \wedge IM\}$
  - IN satisfecho por los p. in, in/out.
  - OUT satisfecho por los p. out, in/out.

**Axiomas de operaciones de sincronización con semántica desplazante** El invariante del monitor debe ser cierto:

- antes de cada wait.
- antes de cada signal.
- Además, justo antes de una operación signal sobre una variable condición c debe ser cierta la condición lógica C asociada a dicha variable.

- **Axioma (operacion c.wait()):** $\{IM \wedge L\}$ c.wait() $\{C \wedge L\}$. El proceso que ocasiona la ejecución de c.wait se bloquea y deja libre el monitor. Entra en la cola FIFO asociada a c. Las demostraciones de corrección no tienen en cuenta la obligación de que el proceso termine de ejecutar c.wait().

- **Axioma (operacion c.signal()):** $\{\neg vacio(c) \wedge L \wedge C\}$ c.signal() $\{IM \wedge L\}$. No tiene efecto si la cola c está vacía. Se supone semántica desplazante, el monitor mantiene el estado expresado por C hasta que un proceso bloqueado en c se reanuda. Los axiomas no tienen en cuenta el orden de desbloqueo de los procesos.

**Regal de la concurrencia para la verificación de programas con monitores:** $\{P_i\}S_i\{Q_i\}$, $1 \leq i \leq n$. Ninguna variable libre en $P_i$ o en $Q_i$ es modificada por $S_j$, $i \neq j$ todas las variables en $IM_k$ son locales al monitor $m_k$.

$$\{IM_1 \wedge \ldots \wedge IM_m \wedge P_1 \wedge \ldots \wedge P_n\}$$
$$cobegin S_1 \parallel S_2 \parallel \ldots \parallel S_n coend$$
$$\{IM_1 \wedge \ldots \wedge IM_m \wedge Q_1 \wedge \ldots \wedge Q_n\}$$

La programación concurrente con monitores excluye cualquier interferencia entre las demostraciones de los procedimientos y la de los procesos secuenciales del programa. Si los asertos de las demostraciones de los procesos no son críticos, entonces las demostraciones individuales $\{P_i\}S_i\{Q_i\}$ cooperan (no es necesario demostrar teoremas de no-interferencia).

**Patrones de uso de monitores:**

- Espera única (EU): un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (ocurre típicamente cuando un proceso debe leer una variable escrita por otro proceso, el primero se suele denominar Consumidor y el segundo Productor).

- Exclusión mútua(EM): acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.

- Problema del Productor/Consumidor(PC): similar a la espera única, pero de forma repetida en un bucle (un proceso Productor escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso Consumidor).

**Notación para la verificación del monitor EU**

- $\#N$ es el número de llamadas a notificar terminadas.

- $\#W$ es el número de llamadas a esperar terminadas.
- La estructura del proceso implica que $\#N \leq 1$ y $\#W \leq 1$.
- En el entrelazamiento E, L se cumple que $\#L \leq \#E$ siempre. En el entrelazamiento L, E (prohibido), no se cumple.
- Luego queremos demostrar que siempre $\#L \leq \#E$. Como veremos, esto es equivalente a demostrar que $\#W \leq \#N$.
- Para demostrar que $\#W \leq \#N$, usamos el invariante del monitor.

**Señales con prioridad:** la semántica de las señales no contempla el desbloqueo de los procesos según un orden prioritario. $c.wait(prioridad)$ bloquea a los procesos en la cola c, pero ordenándolos con respecto al valor del argumento con prioridad.

**Mecanismos de señalación alternativos de los monitores:**

- **SA (señales automáticas):** señal implícita.
- **SC (señales y continuar):** señal explícita, no desplazante.
- **SS (señal y salir):** señal explícita, desplazante, el proceso sale del monitor.
- **SE (señal y esperar):** señal explícita, desplazante, el proceso señalador espera en la cola de entrada al monitor.
- **SU (señales urgentes):** señal explícita, desplazante, el proceso señalador espera en la cola de procesos urgentes.

**Señalar y continuar (SC): características**

- El proceso señalado abandonará después la cola condición y espera en la cola del monitor para readquirir la E.M.
- Tanto el señalador como otros procesoso pueden hacer falsa la condición después de que el señalado abandone la cola condición.
- Por tanto, en el proceso señalado no se puede garantizar que la condición asociada a cond es cierta al terminar cond.wait() y, lógicamente, es necesario volver a comprobarla entonces.
- Esta semántica obliga a programar la operación wait en un bucle.
- Si hay código tras signal, no se ejecuta. El proceso señalado reanuda inmediatamente la ejecución de código del monitor.
- En este caso, la operación signa conllevas:

1. Liberar al proceso señalado
2. Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador
3. Está asegurado el estado qeu permite al proceso señalado continuar la ejecución del procedimiento dle monitor en el que se bloqueó (la condición de desbloqueo se cumple).
4. Esta semántica condiciona el estilo de programación ya que obliga a colocar siempre la operación sginal como última instrucción de los procedimientos de monitor que la usen.

- Señalador: abandona el monitor (ejecuta código tras la llamada al procedimiento del monitor). Si hay código en este proceso, tras ejecutar signal, no se ejecuta inmediatamente
- Señalado: reanuda inmediatamente la ejecución del código del procedimiento dle monitor, programado tras la operación wait.

**Señalar y esperar (SE): diagrama de estados del proceso**

Señalador: se bloquea en la cola del monitor hasta readquirir E.M. y ejecutar el código del monitor tras signal.

Señalado: reanuda inmediatamente la ejecución de código del monitor tras wait.

Señalar y esperar(SE): características

- El proceso señalado entra de forma inmediata en el monitor
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.

1. El proceso señalador entre en la cola de procesos del monitor, por lo que está al mismo nivel que el resto de procesos que compiten por la exclusión mutua del monitor.
2. Puede considerarse una semántica injusta respecto al proceso señalador ya que dicho proceso ya había obtenido el acceso al monitor por lo que debería tener prioridad sobre el resto de procesos que compiten por el monitor.

**Señalar y espera urgente (SU): diagrama de estados del proceso**

Señalador: se bloquea en la cola de urgentes hasta readquirir la E.M. y ejecutar código del monitor tras signal. Para readquirir E.M. tiene más prioridad que los procesos en la cola del monitor.

Señalado: reanuda inmediatamente la ejecución de código del monitor tras la operación wait.

Señalar y espera urgente (SU): características

- El proceso señalador se bloquea justo después de ejecutar la operación signal:

    1. El proceso señalado entra de forma inmediata en el monitor.
    2. Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
    3. El proceso señalador entre en una nueva cola de procesos que esperan para acceder al monitor, que podemos llamar cola de procesos urgentes.
    4. Los procesos de la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los que esperan en la cola del monitor.

**Análisis comparativo de las diferencias semánticas de señales:**

**Facilidad de uso:** La semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos.

**Eficiencia:** Las semánticas SE y SU resultan ineficiente cuando no hay código tras signal, ya que en ese caso implican que el señalador emplea tiempo en bloquearse y después reactivarse, pero justo a continuación abandona el monitor sin hacer nada. La semántica SC también es un poco ineficiente al obligar a usar un bucle para cada instrucción signal.

**Reglas de verificación de las señales no-desplazantes SC:**

- **Axioma de la operación c.wait():** $\{IM \wedge L\}$ c.wait() $\{IM \wedge L\}$. La regla permite demostrar la corrección parcial de los programas independientemente de la aplicación de los procesos de la cola c. No demuestra, por tanto, ni la propiedad de vivacidad, ni detecta bloqueos.
- **Axioma de las operaciones c.signal(), c.signal_all():** $\{P\}$ c.signal() $\{P\}$.

**Intercambio de señales en programas que usan monitores**, condiciones que se han de cumplir para poder intercambiar SW y SC sin modificar adicionalmente el código del monitor:

1. Sólo se ha de exigir postcondición de c.wait() el invariante del monitor.
2. Después de una llamada a la operación c.signal() se ha de salir del monitor.

3. No se puede utilizar c.signal_all(): difusión de una señal a un grupo de procesos.

**Imprementación de los monitores con semáforos:**

- Cola de entrada al monitor: controlada por el semáforo mutex
- Cola de procesos urgentes: controlada por el semáforo next
- Número de procesos urgentes en cola: se contabiliza en la variable next_count
- Colas de procesos bloqueados en cada condición: controladas por el semáforo asociado a cada condición x_sem y el número de procesos en cada cola se contabiliza en una variable asociada a cada condición (x_sem_count).

**Introducción al problema de exclusión mútua:**

- 1962: Dekker propone el problema de la exclusión mútua para multiprocesadores → diseñar un protocolo que garantice el acceso mutuamente excluyente, sin que exista interbloqueo, a una sección crítica por parte de un determinado número de procesos que compiten por entrar a dicha sección...".
- 1965: Dijkstra propone una solución segura, libre de interbloqueo, pero que puede producir inanición.
- 1966: Knuth propone una solución sin inanición; garantiza retraso limitado de los procesos, pero no FIFO.
- 1974: Lamport permite a los procesos detenerse en la ejecución del protocolo de adquisición, solapar las operaciones de lectura con la escritura y retraso FIFO de los procesos que ya esperan entrar.
- 1981: Peterson propone una solución equitativa para 2 y n procesos; garantiza el retraso cuadrático de los procesos, es la solución más simple hasta la fecha para multiprocesadores.
- 1983: algoritmos totalmente distribuidos que resuelven el problema para multicomputadores; Ricart-Aggrawala, Suzuki-Kasami.

**Solución al problema con bucles de espera activa:** los procesos iteran en un bucle vacío hasta que la entrada en Sección Crítica (SC) sea segura. Aceptable si el sistema/aplicación no tuviera muchos procesos.

**Condiciones de Dijkstra** para obtener una solución parcialmente correcta al problema de exclusión mútua:

1. No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por el multiprocesador.
2. Ni tampoco acerca de la velocidad de ejecución de los procesos, excepto que no es cero (Progreso finito).
3. Cuando un procesos se encuentra ejecutando código fuera de la sección crítica no puede impedir a los otros procesos entrar en ésta.
4. La sección crítica siempre será alcanzada por alguno de los procesos que esperan entrar.

**Algoritmo de Dekker:** es un algoritmo concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos, fue inventado por Edsger Dijkstra. Existen cinco versiones del algoritmo Dekker, teniendo ciertos fallos los primeros cuatros:

- Versión 1: Alternancia estricta. Garantiza la exclusión mutua, pero su desventaja es que acopla los procesos fuertemente, esto significa que los procesos lentos atrasan a los procesos rápidos. A este esquema se le llama esquema de corrutinas y no cumple la tercera propiedad de Dijkstra.

```
Proceso P1
while true do
begin
<<resto instrucciones>>
while turno <> 1 do
nothing;
enddo;
<<seccion critica>>
turno:=2;
end
enddo;

Proceso P2
while true do
begin
<<resto instrucciones>>
while turno <> 2 do
nothing;
enddo;
<<seccion critica>>
turno:=1;
end
enddo;
```

- Versión 2: Colisión región crítica no garantiza la exclusión mutua. Este algoritmo no evita que dos procesos puedan acceder al mismo tiempo a la región crítica. No se cumple la propiedad de seguridad del algoritmo.

```
Proceso P1
begin
<<resto instrucciones>>
while c2=0 do
nothing;
enddo;
c1:=0;
<<seccion crítica>>
c1:=1;
end
enddo;

Proceso P2
while true do
begin
<<resto instrucciones>>
while c1=0 do
nothing;
enddo;
c2:=0;
<<seccion critica>>
c2:=1;
end
enddo;
```

- Versión 3: Problema interbloqueo. No existe alternancia, aunque ambos procesos caen a un mismo estado y nunca salen de ahí. El proceso que modifica la clave no sabe si el otro hace lo mismo concurrentemente con el.

```
Proceso P1
while true do
begin
<<resto instrucciones>>
c1:=0;
while c2=0 do
nothing;
enddo;
<<seccion critica>>
```

```
c1:=1;
end
enddo;

Proceso 2
while true do
begin
<<resto instrucciones>>
c2:=0;
while c1=0 do
nothing;
enddo;
<<seccion critica>>
c2:=1;
end
enddo;
```

- Versión 4: Postergación indefinida. Aunque los procesos no están en interbloqueo, un proceso o varios se quedan esperando a que suceda un evento que tal vez nunca suceda.

```
Proceso P1
while true do
begin
<<resto instrucciones>>
c1:=0;
while c2=0 do
begin
c1:=1
while c2=1 do
nothing;
enddo;
c1:=0;
end
enddo;
<<seccion critica>>
c1:=1;
end
enddo;

Proceso P2
while true do
begin
<<resto instrucciones>>
```

```
c2:=0;
while c1=0 do
begin
c2:=1;
while c1=1 do
nothing;
enddo;
c2:=0;
end
enddo;
<<seccion critica>>
c2:=1;
end
enddo;
```

- Versión 5: Algoritmo de Dekker:

```
Proceso P1
while true do
begin
<<resto instrucciones>>
c1:=0;
while c2=0 do
if turno=2 then
begin
c1:=1;
while turno=2 do
nothing;
enddo;
c1:=0;
end
endif;
enddo;
<<seccion critica>>
turno:=2;
c1:=1;
end
enddo;

Proceso P2
while true do
begin
<<resto instrucciones>>
c2:=0;
```

```
while c1=0 do
if turno=1 then
begin
c2:=1;
while turno=1 do
nothing;
enddo;
c2:=0;
end
endif
enddo;
<<seccion critica>>
turno:=1;
c2:=1;
end
enddo;
```

**Verificación de propiedades de seguridad Exclusión mutua:**

1. $P_i$ entre en sección crítica sólo si $c[j] == 1$
2. $P_i$ comprueba la clave del otro, $c[j]$, sólo después de asignar su propia clave. Luego, cuando $P_i$ entre se cumple, $c[j] == 1 \wedge c[i] == 0$

**Verificación de propiedades de seguridad alcanzabilidad de la sección crítica**: Si $P_i$ y $P_j$ intentan entrar en sección crítica y $turno == i$:

1. Si $P_i$ encuentra la clave del otro $c[j] == 1$, entonces $P_i$ entra.
2. Si no, dependerá de quien tenga el turno:
   a) Si $turno == i$ espera que $P_j$ cambie su clave y, después, entra.
   b) Si $turno == j$ cambia su clave a 1 y se queda en espera activa.

El algoritmo de Dekker se puede generalizar para n procesos de la siguiente manera (Algoritmo de Dijkstra)

```
repeat(1) #c: array[0,...,n-1] of (pasivo, solicitando, en_SC)
repeat(2) #turno: 0,...,n-1;
E2:c[i]:=solicitando;
while turno <> i do
E3:if c[turno] = pasivo
then turno:= i
endif;
enddo;
```

```
E4:c[i]:=en_SC;

while(j<n) and (j=i or c[j] <> en_SC) do
j:=j+1;
enddo;
until j>= n(2)

<<seccion critica>>
E1:c[i]:=pasivo;
<<resto de instrucciones>>
until flase(1);
```

**Verificación de las propiedades de seguridad**, alcanzabilidad de la sección crítica:

1. turno es una variable compartida, será asignada por el último $P_i$ que cambie su clave, $c[j] == en_SC$.
2. Sean $\{P_1, \ldots, P_i, \ldots, P_m\}$ tales que $c[i] = en_SC$ y $turno == k$ con $1 \leq k \leq m$, entonces $P_k$ entrará en su sección en tiempo finito y el resto $P_i : 1 \leq i \leq m \wedge i \neq k$ se quedará ciclando.

**Algoritmo de Knuth para N procesos**:

```
repeat c:array[0,\ldots,n-1] of (pasivo,solicitando,en_SC)
repeat(2) turno: 0,\ldots,n-1;
E0: c[i]=solicitando;
j:=turno; ->variable local
E1:while j <> i do
if c[j] <> pasivo then j:= turno
else j:= (j-1) mod n
endif;
enddo;

E2: c[i]:=en_SC;
k:=0;
while (k<n) and (k=i or c[k]<>en_SC) do
k:=k+1;
enddo;

until k>= n;(2)

E3: turno:=i;
<<Sección Crítica>>
turno:=(i-1) mod n;
```

```
E4: c[i] := pasivo;
E5: <<resto de instrucciones>>
until false;
```

El algoritmo de Knuth da lugar a la imposibilidad de inanición de los procesoso si se supone que existe un número finito de ellos en el algoritmo.

**Algoritmo de Peterson**

```
Proceso P_1
repeat
solicitado[0]=true;
turno = 1;
while (solicitado[1] = true) and (turno = 1) do
nothing;
enddo;
Seccion Crítica_0;
solicitado[0] = false;
resto_0;
forever;

Proceso P_2
repeat
solicitado[1]=true;
turno = 0;
while (solicitado[0] = true) && (turno = 0) do
nothing;
enddo;
Seccion Critica_1;
Resto_1
forever
```

**Algoritmo de Peterson: Solución para N procesos:**

```
var c: array[0,\ldots,n-1] of -1,\ldots,n-2;
turno: array[0,\ldots,n-2] of 0,\ldots,n-1;

while true do
begin
<<resto de instrucciones>>
for j=0 to n-2 do
begin
c[i]:=j;
turno[j]:=i;
```

```
while ((exists k <> i: c[k] >= j) && turno[j]=i) do
nothing;
enddo;
c[i]:=n-1;
<<seción crítica>>
c[i]:=-1;
```

**Verificación de las propiedades del Algoritmo de peterson:** Se dice que $P_i$ precede a un proceso $P_j$ si y sólo si $c[i] > c[j]$.

1. Un proceso que precede a todos los demás puede avanzar al menos una etapa, ya que si no se cumple la condición del bucle o bien el proceso está en la etapa más avanzada o bien ha llegado otro proceso después a al etapa. El proceso $P_i$ puede ser adelantado en la etapa siguiente, podrían llegar más de un proceso a la etapa j, pero siempre se cumplirá que $P_i$ avanzará.

2. Un proceso que pasa de la etapa j a la j+1 ha de verificar alguna de estas condiciones: o bien prece a los demás o no estaba solo en la etapa j. Si precede a todos los demás se puede aplicar el paso anterior. Si no estaba solo en la etapa j implica que $turno[j] <> i$, luego al proceso se le unió otro.

    - Podría suceder que, justo cuando el proceso vaya a avanzar de etapa, porque se cumpla la primera condición, se le una otro proceso a su etapa, pero también se cumplirá.

3. Si existe al menos dos procesos en la etapa j, entonces existe al menos un proceso en cada una de las etapas anteriores.

4. El número máximo de procesos que puede haber en la etapa j es n-j, con $0 \leq j \leq n-2$.

**Alcanzabilidad de la SC:** Supongamos que todos los procesos se quedan bloqueados al llegar a una etapa y no avanzan más. El proceso precede a los demás $\Rightarrow$ contradice el primer Lema. Si no, el proceso llega a una etapa ocupada por al menos un proceso $\Rightarrow$ contradice el lema ".

**Propiedad de equidad:** el número máximo de turnos que un proceso cualquiera tendría que esperar con el algoritmo de Peterson es de $r(n) = n - 1 + r(n-1) = \frac{n \cdot (n+1)}{2}$ turnos.

# 2. Sincronización sin memoria compartida: algoritmos de exclusión mutua

Problea para la asignación de los algoritmos en los sistemas distribuidos.

- Los algoritmos no pueden utilizar sincronización global entre los procesos sólo utilizando variables en memoria compartida.
- Se utilizan operaciones atómicas de paso de mensajes para sincronizacrlos
- La red de comunicaciones ha de cumplir las siguientes condiciones
  - Red de comunicaciones completamente conectada.
  - Transimisión de mensajes sin errores.
  - Retraso variable en la entrega de mensajes con tiempo acotado.
  - Posibles desecuenciamientos en la etrega de mensajes en transmisión.

**Algoritmo de Ricart-Agrawala**

```
ns:0,\ldots,+INF; mns:0,\ldots,INF:=0;
solicitaSC, prioridad:boolean;
num_repesperadas:0,\ldots,n-1;
repretrasadas: array[1,\ldots,N] of boolean;

Hebra Main(i)::=
solicitaSC:=true;
ns:= mns+1;
num_repesperadas:=n-1;

for j:=1 to n do
if j<>i then
send(j,pet,ns,i);
endif;
enddo;
wait(sinc);
<<Sección Crítica>>

solicitaSC:=false;

for j:=1 to n do
if repretrasadas[j]:=fasle;
send(j,rep);
end;
endif;
enddo;
////////////////////////////////////////////////////////////////////////
```

```
Hebra Rep(i)::=
receive(rep);
num_repesperadas-=1;
if num_repesperadas=0
then
signal(sinc);
endif;

////////////////////////////////////////////////////////////////////////

Hebra Pet(i)::=
receive (pet,k,j)
mns:=max(ns,k);
prioridad:=solicitaSC and (k > ns or (k=ns and i < j));
if prioridad then
retrasadas[j]:=true;
else send(j,rep)
endif;
```

**Verificación de las propiedades del algoritmo de Ricart-Agrawala**: exclusión mutua entre procesos al acceder a la sección crítica.

- $P_i$ y $P_j$ consiguen entrar a la ez en SC (hipótesis de incorrección). $P_{i,j}$ ha transmitido su petición a $P_{j,i}$, recibiendo contestación favorable.

    1. $P_i$ ha enviado contestación favorable antes de generar su número de secuencia $\Rightarrow$ $P_j$ pospone la respuesta.
    2. $P_j$ ha enviado contestación favorable antes de generar su número de secuencia $\Rightarrow$ $P_i$ pospone la respuesta.
    3. Después de generar su número de secuencia es imposible que cada proceso envie contestación favorable al otro.

**Alcanzabilidad de la sección crítica:** debido a la ordenación total de las peticiones, es imposible que se retrase indefinidamente la contestación favorable a algún proceso.

**Algoritmo de Suzuki-Kasami-Ricart-Agrawala:**

```
Variables locales P(i)
token_presente:boolean;
en_SC:boolean;
toker: array[1,\ldots,N] of 0,\ldots,+INF;
peticion: array[1,\ldots,N] of 0,\ldots,+INF;
```

```
##########################################################################

Hebra pet(i)::=
receive (pet,k,j);
peticion[j]:=max(peticion[j],k)

if token_presente and not en_SC then
for j:=i+1 to n, 1 to i-1 do
if peticion[j] > token[j] and token_presente then
begin
token_presente:=false;
send(j,acceso,token);
end;
endif;
enddo;

##########################################################################

Hebra mi)::=
if NOT token_presente then
begin
ns:= ns+1;
broadcast(pet,ns,i);
receive(acceso,token);
toker_presente:=true;
end;
endif;
en_SC:=true;
<<Sección crítica>>
token[i]:=ns;
en_SC:=false;
for j:=i+1 to n, 1 to i-1 do
if peticion[j] > token[j] and token_presente then
begin
token_presente:=false;
send(j,acceso,token)
end;
endif;
enddo;
```

**Verificación del algoritmo de Suzuki-Kasami-Ricart-Agrawala**

El que todo proceso acceda en exclusión mutua es equivalente a demostrar el siguiente aserto: globalmente, el número de variables token_presente=true

es identicamente igual a la unidad.

1. El aserto anterior se satisface inicialmente.
2. El aserto anterior se cumple cada vez que transmite el token. El protocolo necesita n mensajes.

**Alcanzabilidad de la sección crítica:**
Si ningún proceso posee el token, en un momento de la ejecución del algoritmo, este ha de estar necesariamente en transmisión.

**Propiedad de equidad**

- Se obliga a que $P_j$ transmita el token al primero que lo solicitó en el orden {j+1,j+2,...,n,n-1,...}

**Algoritmo de regeneración de token de Misra**

```
variable local m_i: int:=0;
while true do
Seleccionar
Cuando recibido (ping, num_ping) hacer
if m_i= num_ping then
begin # se ha perdido pong, hay que recuperarlo
num_ping:=(num_ping+1)mod n+1;
num_pong:= -num_ping;
end
else mi:= num_ping;
endif;
endhacer;

Cuando recibido (pong, num_pong) hacer
if m_i = num_pong then
begin # se ha perdido ping, hay que recuperarlo
num_pong := -((-num_ping + 1) mod n+1);
num_ping := -num_pong;
end
else m_i := num_pong;
endif;
endhacer;

Cuando se encuentrarn (ping, pong) hacer
begin # |num_ping| = |num_pong|, llevan el número colisiones
num_ping := (num_ping+1) mod n+1;
num_pong := -num_ping;
end
```

```
endhacer;
endSeleccionar;
enddo;
```

# 3. Glosario

- **Variables de condición:** posibilitan la espera de condiciones diferentes dentro de un monitor. Se define una variable por cada condición, dentro de los procedimientos.
- **c.queue():** función lógica que devuelve true si hay algún proceso esperando en la cola de cond, y false en caso contrario.
- **Invariante de un monitor (IM)**: es una propiedad que el monitor cumple siempre, pero específico de cada monitor diseñado por un programador. Unido a las propiedades de los procesos que invocan al monitor, el IM facilita la verificación de los programas concurrentes.
- **Cola de entrada al monitor:** controlada por el semáforo mutex.
- **Cola de procesos urgentes:** controlada por el semáforo next.
- **Número de procesos urgentes en la cola:** se contabiliza en la variable next_count.
- **Colas de procesos bloqueados en cada condición:** controladas por el semáforo asociado a cada condición x_sem y el número de procesos en cada cola se contabiliza en una variable asociada a cada condición (x_sem_count).